Chapter 3

Deterministic planning

The simplest planning problems involves finding a sequence of actions that lead from a given initial state to a goal state. Only deterministic actions are considered. Determinism and the uniqueness of the initial state mean that the state of the transition system after any sequence of actions is exactly predictable. The problem instances in this chapter are deterministic succinct transition systems as defined in Section 2.3.1.

3.1 State-space search

The simplest possible planning algorithm generates all states (valuations of the state variables), constructs the transition graph, and then finds a path from the initial state I to a goal state $g \in G$ for example by a shortest-path algorithm. The plan is then simply the sequence of operators corresponding to the edges on the shortest path from the initial state to a goal state. However, this algorithm is not feasible when the number of state variables is higher than 20 or 30 because the number of valuations is very high: $2^{20} = 1048576 \sim 10^6$ for 20 Boolean state variables and $2^{30} = 1073741824 \sim 10^9$ for 30.

Instead, it will often be much more efficient to avoid generating most of the state space explicitly and to produce only the successor or predecessor states of the states currently under consideration. This form of plan search can be easiest viewed as the application of general-purpose search algorithms that can be employed in solving a wide range of search problems. The best known *heuristic search algorithms* are A$*$, IDA$*$ and their variants [Hart *et al.*, 1968; Pearl, 1984; Korf, 1985] which can be used in finding shortest plans or plans that are guaranteed to be close to the shortest ones.

There are two main possibilities to find a path from the initial state to a goal state: traverse the transition graph forwards starting from the initial state, or traverse it backwards starting from the goal states. The main difference between these possibilities is that there may be several goal states (and one state may have several predecessor states with respect to one operator) but only one initial state: in forward traversal we repeatedly compute the unique successor state of the current state, whereas with backward traversal we are forced to keep track of a possibly very high number of possible predecessor states of the goal states. Backward search is slightly more complicated to implement but it allows to simultaneously consider several paths leading to a goal state.

3.1.1 Progression and forward search

We have already defined *progression* for single states s as $app_o(s)$. The simplest algorithm for the deterministic planning problem does not require the explicit representation of the whole transition graph. The search starts in the initial state. New states are generated by progression. As soon as a state s such that $s \models G$ is found a plan is guaranteed to exist: it is the sequence of operators with which the state s is reached from the initial state.

A planner can use progression in connection with any of the standard search algorithms. Later in this chapter we will discuss how heuristic search algorithms together with heuristics yield an efficient planning method.

3.1.2 Regression and backward search

With backward search the starting point is a propositional formula G that describes the set of goal states. An operator is selected, the set of possible predecessor states is computed, and this set is again described by a propositional formula. A plan has been found when a formula that is true in the initial state is reached. The computation of a formula representing the predecessor states of the states represented by another formula is called *regression*. Regression is more powerful than progression because it allows handling potentially very big sets of states, but it is also more expensive.

Definition 3.1 *We define the condition $EPC_l(e)$ of literal l made true when an operator with the effect e is applied recursively as follows.*

$$\begin{aligned} EPC_l(\top) &= \bot \\ EPC_l(l) &= \top \\ EPC_l(l') &= \bot \text{ when } l \neq l' \text{ (for literals } l') \\ EPC_l(e_1 \wedge \cdots \wedge e_n) &= EPC_l(e_1) \vee \cdots \vee EPC_l(e_n) \\ EPC_l(c \rhd e) &= c \wedge EPC_l(e) \end{aligned}$$

The case $EPC_l(e_1 \wedge \cdots \wedge e_n) = EPC_l(e_1) \vee \cdots \vee EPC_l(e_n)$ is defined as a disjunction because it is sufficient that at least one of the effects makes l true.

Definition 3.2 *Let A be the set of state variables. We define the condition $EPC_l(o)$ of operator $o = \langle c, e \rangle$ being applicable so that literal l is made true as $c \wedge EPC_l(e) \wedge \bigwedge_{a \in A} \neg(EPC_a(e) \wedge EPC_{\neg a}(e))$.*

For effects e the truth-value of the formula $EPC_l(e)$ indicates in which states l is a literal to which the effect e assigns the value true. The connection to the earlier definition of $[e]_s^{det}$ is stated in the following lemma.

Lemma 3.3 *Let A be the set of state variables, s a state on A, l a literal on A, and o and operator with effect e. Then*

1. *$l \in [e]_s^{det}$ if and only if $s \models EPC_l(e)$, and*
2. *$app_o(s)$ is defined and $l \in [e]_s^{det}$ if and only if $s \models EPC_l(o)$.*

Proof: We first prove (1) by induction on the structure of the effect e.

Base case 1, $e = \top$: By definition of $[\top]_s^{det}$ we have $l \notin [\top]_s^{det} = \emptyset$, and by definition of $EPC_l(\top)$ we have $s \not\models EPC_l(\top) = \bot$, so the equivalence holds.

Base case 2, $e = l$: $l \in [l]_s^{det} = \{l\}$ by definition, and $s \models EPC_l(l) = \top$ by definition.

Base case 3, $e = l'$ for some literal $l' \neq l$: $l \notin [l']_s^{det} = \{l'\}$ by definition, and $s \not\models EPC_l(l') = \bot$ by definition.

Inductive case 1, $e = e_1 \wedge \cdots \wedge e_n$:

$$\begin{array}{lll} l \in [e]_s^{det} & \text{if and only if} & l \in [e']_s^{det} \text{ for some } e' \in \{e_1, \ldots, e_n\} \\ & \text{if and only if} & s \models EPC_l(e') \text{ for some } e' \in \{e_1, \ldots, e_n\} \\ & \text{if and only if} & s \models EPC_l(e_1) \vee \cdots \vee EPC_l(e_n) \\ & \text{if and only if} & s \models EPC_l(e_1 \wedge \cdots \wedge e_n). \end{array}$$

The second equivalence is by the induction hypothesis, the other equivalences are by the definitions of $EPC_l(e)$ and $[e]_s^{det}$ as well as elementary facts about propositional formulae.

Inductive case 2, $e = c \rhd e'$:

$$\begin{array}{lll} l \in [c \rhd e']_s^{det} & \text{if and only if} & l \in [e']_s^{det} \text{ and } s \models c \\ & \text{if and only if} & s \models EPC_l(e') \text{ and } s \models c \\ & \text{if and only if} & s \models EPC_l(c \rhd e'). \end{array}$$

The second equivalence is by the induction hypothesis. This completes the proof of (1).

(2) follows from the fact that the conjuncts c and $\bigwedge_{a \in A} \neg(EPC_a(e) \wedge EPC_{\neg a}(e))$ in $EPC_l(o)$ exactly state the applicability conditions of o. □

Note that any operator $\langle c, e \rangle$ can be expressed in normal form in terms of $EPC_a(e)$ as

$$\left\langle c, \bigwedge_{a \in A} (EPC_a(e) \rhd a) \wedge (EPC_{\neg a}(e) \rhd \neg a) \right\rangle.$$

The formula $EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$ expresses the condition for the truth $a \in A$ after the effect e is executed in terms of truth-values of state variables before: either a becomes true, or a is true before and does not become false.

Lemma 3.4 *Let $a \in A$ be a state variable, $o = \langle c, e \rangle \in O$ an operator, and s and $s' = app_o(s)$ states. Then $s \models EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$ if and only if $s' \models a$.*

Proof: Assume that $s \models EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$. We perform a case analysis and show that $s' \models a$ holds in both cases.

Case 1: Assume that $s \models EPC_a(e)$. By Lemma 3.3 $a \in [e]_s^{det}$, and hence $s' \models a$.

Case 2: Assume that $s \models a \wedge \neg EPC_{\neg a}(e)$. By Lemma 3.3 $\neg a \notin [e]_s^{det}$. Hence a is true in s'.

For the other half of the equivalence, assume that $s \not\models EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$. Hence $s \models \neg EPC_a(e) \wedge (\neg a \vee EPC_{\neg a}(e))$.

Case 1: Assume that $s \models a$. Now $s \models EPC_{\neg a}(e)$ because $s \models \neg a \vee EPC_{\neg a}(e)$, and hence by Lemma 3.3 $\neg a \in [e]_s^{det}$ and hence $s' \not\models a$.

Case 2: Assume that $s \not\models a$. Since $s \models \neg EPC_a(e)$, by Lemma 3.3 $a \notin [e]_s^{det}$ and hence $s' \not\models a$.

Therefore $s' \not\models a$ in all cases. □

The formulae $EPC_l(e)$ can be used in defining regression.

Definition 3.5 (Regression) *Let ϕ be a propositional formula and $o = \langle c, e\rangle$ an operator. The* regression *of ϕ with respect to o is $regr_o(\phi) = \phi_r \wedge c \wedge \chi$ where $\chi = \bigwedge_{a\in A} \neg(EPC_a(e) \wedge EPC_{\neg a}(e))$ and ϕ_r is obtained from ϕ by replacing every $a \in A$ by $EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$. Define $regr_e(\phi) = \phi_r \wedge \chi$ and use the notation $regr_{o_1;\ldots;o_n}(\phi) = regr_{o_1}(\cdots regr_{o_n}(\phi)\cdots)$.*

The conjuncts of χ say that none of the state variables may simultaneously become true and false. The operator is not applicable in states in which χ is false.

Remark 3.6 *Regression can be equivalently defined in terms of the conditions the state variables stay or become false, that is, we could use the formula $EPC_{\neg a}(e) \vee (\neg a \wedge \neg EPC_a(e))$ which tells when a is false. The negation of this formula, which can be written as $(EPC_a(e) \wedge \neg EPC_{\neg a}(e)) \vee (a \wedge \neg EPC_{\neg a}(e))$, is not equivalent to $EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$. However, if $EPC_a(e)$ and $EPC_{\neg a}(e)$ are not simultaneously true, we do get equivalence, that is,*

$$\begin{aligned}\neg(EPC_a(e) \wedge EPC_{\neg a}(e)) \models\ & ((EPC_a(e) \wedge \neg EPC_{\neg a}(e)) \vee (a \wedge \neg EPC_{\neg a}(e))) \\ & \leftrightarrow (EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e)))\end{aligned}$$

because $\neg(EPC_a(e) \wedge EPC_{\neg a}(e)) \models (EPC_a(e) \wedge \neg EPC_{\neg a}(e)) \leftrightarrow EPC_a(e)$.

An upper bound on the size of the formula obtained by regression with operators $o_1, \ldots, o_n$ starting from ϕ is the product of the sizes of $\phi, o_1, \ldots, o_n$, which is exponential in n. However, the formulae can often be simplified because there are many occurrences of $\top$ and $\bot$, for example by using the equivalences $\top \wedge \phi \equiv \phi$, $\bot \wedge \phi \equiv \bot$, $\top \vee \phi \equiv \top$, $\bot \vee \phi \equiv \phi$, $\neg\bot \equiv \top$, and $\neg\top \equiv \bot$. For unconditional operators $o_1, \ldots, o_n$ (with no occurrences of $\rhd$), an upper bound on the size of the formula (after eliminating $\top$ and $\bot$) is the sum of the sizes of $o_1, \ldots, o_n$ and ϕ.

The reason why regression is useful for planning is that it allows to compute the predecessor states by simple formula manipulation. The same does not seem to be possible for progression because there is no known simple definition of successor states of a *set* of states expressed in terms of a formula: simple syntactic progression is restricted to individual states only (see Section 4.2 for a general but expensive definition of progression for arbitrary formulae.)

The important property of regression is formalized in the following lemma.

Theorem 3.7 *Let ϕ be a formula over A, o an operator over A, and S the set of all states i.e. valuations of A. Then $\{s \in S | s \models regr_o(\phi)\} = \{s \in S | app_o(s) \models \phi\}$.*

Proof: We show that for any state s, $s \models regr_o(\phi)$ if and only if $app_o(s)$ is defined and $app_o(s) \models \phi$. By definition $regr_o(\phi) = \phi_r \wedge c \wedge \chi$ for $o = \langle c, e\rangle$ where ϕ_r is obtained from ϕ by replacing every state variable $a \in A$ by $EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$ and $\chi = \bigwedge_{a\in A} \neg(EPC_a(e) \wedge EPC_{\neg a}(e))$.

First we show that $s \models c \wedge \chi$ if and only if $app_o(s)$ is defined.

$s \models c \wedge \chi$	iff	$s \models c$ and $\{a, \neg a\} \not\subseteq [e]_s^{det}$ for all $a \in A$	by Lemma 3.3
	iff	$app_o(s)$ is defined	by Definition 2.13.

Then we show that $s \models \phi_r$ if and only if $app_o(s) \models \phi$. This is by structural induction over subformulae ϕ' of ϕ and formulae ϕ'_r obtained from ϕ' by replacing $a \in A$ by $EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$

Induction hypothesis: $s \models \phi'_r$ if and only if $app_o(s) \models \phi'$.

Base case 1, $\phi' = \top$: Now $\phi'_r = \top$ and both are true in the respective states.

Base case 2, $\phi' = \bot$: Now $\phi'_r = \bot$ and both are false in the respective states.

Base case 3, $\phi' = a$ for some $a \in A$: Now $\phi'_r = EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))$. By Lemma 3.4 $s \models \phi'_r$ if and only if $app_o(s) \models \phi'$.

Inductive case 1, $\phi' = \neg\theta$: By the induction hypothesis $s \models \theta_r$ iff $app_o(s) \models \theta$. Hence $s \models \phi'_r$ iff $app_o(s) \models \phi'$ by the truth-definition of $\neg$.

Inductive case 2, $\phi' = \theta \vee \theta'$: By the induction hypothesis $s \models \theta_r$ iff $app_o(s) \models \theta$, and $s \models \theta'_r$ iff $app_o(s) \models \theta'$. Hence $s \models \phi'_r$ iff $app_o(s) \models \phi'$ by the truth-definition of $\vee$.

Inductive case 3, $\phi' = \theta \wedge \theta'$: By the induction hypothesis $s \models \theta_r$ iff $app_o(s) \models \theta$, and $s \models \theta'_r$ iff $app_o(s) \models \theta'$. Hence $s \models \phi'_r$ iff $app_o(s) \models \phi'$ by the truth-definition of $\wedge$. □

Regression can be performed with any operator but not all applications of regression are useful. First, regressing for example the formula a with the effect $\neg a$ is not useful because the new unsatisfiable formula describes the empty set of states. Hence the sequence of operators of the previous regressions steps do not lead to a goal from any state. Second, regressing a with the operator $\langle b, c\rangle$ yields $regr_{\langle b,c\rangle}(a) = a \wedge b$. Finding a plan for reaching a state satisfying a is easier than finding a plan for reaching a state satisfying $a \wedge b$. Hence the regression step produced a subproblem that is more difficult than the original problem, and it would therefore be better not to take this regression step.

Lemma 3.8 *Let there be a plan $o_1, \ldots, o_n$ for $\langle A, I, O, G\rangle$. If $regr_{o_k;\ldots;o_n}(G) \models regr_{o_{k+1};\ldots;o_n}(G)$ for some $k \in \{1, \ldots, n-1\}$, then also $o_1, \ldots, o_{k-1}, o_{k+1}, \ldots, o_n$ is a plan for $\langle A, I, O, G\rangle$.*

Proof: By Theorem 3.7 $app_{o_{k+1};\ldots;o_n}(s) \models G$ for any s such that $s \models regr_{o_{k+1};\ldots;o_n}(G)$. Since $app_{o_1;\ldots;o_{k-1}}(I) \models regr_{o_k;\ldots;o_n}(G)$ and $regr_{o_k;\ldots;o_n}(G) \models regr_{o_{k+1};\ldots;o_n}(G)$ also $app_{o_1;\ldots;o_{k-1}}(I) \models regr_{o_{k+1};\ldots;o_n}(G)$. Hence $app_{o_1;\ldots;o_{k-1};o_{k+1};\ldots;o_n}(I) \models G$ and $o_1; \ldots; o_{k-1}; o_{k+1}; \ldots; o_n$ is a plan for $\langle A, I, O, G\rangle$. □

Therefore any regression step that makes the set of states smaller in the set-inclusion sense is unnecessary. However, testing whether this is the case may be computationally expensive. Although the following two problems are closely related to SAT, it could be possible that the formulae obtained by reduction to SAT would fall in some polynomial-time subclass. We show that this is not the case.

Lemma 3.9 *The problem of testing whether $regr_o(\phi) \not\models \phi$ is NP-hard.*

Proof: We give a reduction from SAT to the problem. Let ϕ be any formula. Let a be a state variable not occurring in ϕ. Now $regr_{\langle \neg\phi \rightarrow a, a\rangle}(a) \not\models a$ if and only if $(\neg\phi \rightarrow a) \not\models a$, because $regr_{\langle \neg\phi \rightarrow a, a\rangle}(a) = \neg\phi \rightarrow a$. $(\neg\phi \rightarrow a) \not\models a$ is equivalent to $\not\models (\neg\phi \rightarrow a) \rightarrow a$ that is equivalent to the satisfiability of $\neg((\neg\phi \rightarrow a) \rightarrow a)$. Further, $\neg((\neg\phi \rightarrow a) \rightarrow a)$ is logically equivalent to $\neg(\neg(\phi \vee a) \vee a)$ and further to $\neg(\neg\phi \vee a)$ and $\phi \wedge \neg a$.

Satisfiability of $\phi \wedge \neg a$ is equivalent to the satisfiability of ϕ as a does not occur in ϕ: if ϕ is satisfiable, there is a valuation v such that $v \models \phi$, we can set a false in v to obtain v', and as a does not occur in ϕ, we still have $v' \models \phi$, and further $v' \models \phi \wedge \neg a$. Clearly, if ϕ is unsatisfiable also $\phi \wedge \neg a$ is.

Hence $regr_{\langle \neg\phi \rightarrow a, a\rangle}(a) \not\models a$ if and only if ϕ is satisfiable. □

Also the problem of testing whether a regression step leads to an empty set of states is difficult.

Lemma 3.10 *The problem of testing that $regr_o(\phi)$ is satisfiable is NP-hard.*

Proof: Proof is a reduction from SAT. Let ϕ be a formula. $\mathit{regr}_{\langle\phi,a\rangle}(a)$ is satisfiable if and only if ϕ is satisfiable because $\mathit{regr}_{\langle\phi,a\rangle}(a) \equiv \phi$.

The problem is NP-hard even if we restrict to operators that have a satisfiable precondition: ϕ is satisfiable if and only if $(\phi \vee \neg a) \wedge a$ is satisfiable if and only if $\mathit{regr}_{\langle\phi\vee\neg a,b\rangle}(a \wedge b)$ is satisfiable. Here a is a state variable that does not occur in ϕ. Clearly, $\phi \vee \neg a$ is true when a is false, and hence $\phi \vee \neg a$ is satisfiable. □

Of course, testing that $\mathit{regr}_o(\phi) \not\models \phi$ or that $\mathit{regr}_o(\phi)$ is satisfiable is not necessary for the correctness of backward search, but avoiding useless steps improves efficiency.

Early work on planning restricted to goals and operator preconditions that are conjunctions of state variables and to unconditional effects (STRIPS operators with only positive literals in preconditions.) In this special case both goals G and operator effects e can be viewed as sets of literals, and the definition of regression is particularly simple: regressing G with respect to $\langle c, e\rangle$ is $(G \backslash e) \cup c$. If there is $a \in A$ such that $a \in G$ and $\neg a \in e$, then the result of regression is $\bot$, that is, the empty set of states. We do not use this restricted type of regression in this lecture.

Some planners that use backward search and have operators with disjunctive preconditions and conditional effects eliminate all disjunctivity by branching. For example, the backward step from g with operator $\langle a \vee b, g\rangle$ yields $a \vee b$. This formula corresponds to two non-disjunctive goals, a and b. For each of these new goals a separate subtree is produced. Disjunctivity caused by conditional effects can similarly be handled by branching. However, this branching may lead to a very high branching factor and thus to poor performance.

In addition to being the basis of backward search, regression has many other applications in reasoning about actions. One of them is the composition of operators. The composition $o_1 \circ o_2$ of operators $o_1 = \langle c_1, e_1\rangle$ and $o_2 = \langle c_2, e_2\rangle$ is an operator that behaves like applying o_1 followed by o_2. For a to be true after o_2 we can regress a with respect to o_2, obtaining $\mathit{EPC}_a(e_2) \vee (a \wedge \neg \mathit{EPC}_{\neg a}(e_2))$. Condition for this formula to be true after o_1 is obtained by regressing with e_1, leading to

$$\begin{aligned}
&\mathit{regr}_{e_1}(\mathit{EPC}_a(e_2) \vee (a \wedge \neg \mathit{EPC}_{\neg a}(e_2)))\\
&= \mathit{regr}_{e_1}(\mathit{EPC}_a(e_2)) \vee (\mathit{regr}_{e_1}(a) \wedge \neg \mathit{regr}_{e_1}(\mathit{EPC}_{\neg a}(e_2)))\\
&= \mathit{regr}_{e_1}(\mathit{EPC}_a(e_2)) \vee ((\mathit{EPC}_a(e_1) \vee (a \wedge \neg \mathit{EPC}_{\neg a}(e_2))) \wedge \neg \mathit{regr}_{e_1}(\mathit{EPC}_{\neg a}(e_2))).
\end{aligned}$$

Since we want to define an effect $\phi \rhd a$ of $o_1 \circ o_2$ so that a becomes true whenever o_1 followed by o_2 would make it true, the formula ϕ does not have to represent the case in which a is true already before the application of $o_1 \circ o_2$. Hence we can simplify the above formula to

$$\mathit{regr}_{e_1}(\mathit{EPC}_a(e_2)) \vee (\mathit{EPC}_a(e_1) \wedge \neg \mathit{regr}_{e_1}(\mathit{EPC}_{\neg a}(e_2))).$$

An analogous formula is needed for making $\neg a$ false. This leads to the following definition.

Definition 3.11 (Composition of operators) *Let $o_1 = \langle c_1, e_1\rangle$ and $o_2 = \langle c_2, e_2\rangle$ be two operators on A. Then their* composition $o_1 \circ o_2$ *is defined as*

$$\left\langle c, \bigwedge_{a\in A} \begin{pmatrix} ((\mathit{regr}_{e_1}(\mathit{EPC}_a(e_2)) \vee (\mathit{EPC}_a(e_1) \wedge \neg \mathit{regr}_{e_1}(\mathit{EPC}_{\neg a}(e_2)))) \rhd a) \wedge \\ ((\mathit{regr}_{e_1}(\mathit{EPC}_{\neg a}(e_2)) \vee (\mathit{EPC}_{\neg a}(e_1) \wedge \neg \mathit{regr}_{e_1}(\mathit{EPC}_a(e_2)))) \rhd \neg a) \end{pmatrix} \right\rangle$$

where $c = c_1 \wedge \mathit{regr}_{e_1}(c_2) \wedge \bigwedge_{a\in A} \neg(\mathit{EPC}_a(e_1) \wedge \mathit{EPC}_{\neg a}(e_1))$.

Note that in $o_1 \circ o_2$ first o_1 is applied and then o_2, so the ordering is opposite to the usual notation for the composition of functions.

Theorem 3.12 *Let o_1 and o_2 be operators and s a state. Then $app_{o_1 \circ o_2}(s)$ is defined if and only if $app_{o_1;o_2}(s)$ is defined, and $app_{o_1 \circ o_2}(s) = app_{o_1;o_2}(s)$.*

Proof: Let $o_1 = \langle c_1, e_1 \rangle$ and $o_2 = \langle c_2, e_2 \rangle$. Assume $app_{o_1 \circ o_2}(s)$ is defined. Hence $s \models c_1 \wedge regr_{e_1}(c_2) \wedge \bigwedge_{a \in A} \neg (EPC_a(e_1) \wedge EPC_{\neg a}(e_1))$, that is, the precondition of $o_1 \circ o_2$ is true, and $s \not\models (regr_{e_1}(EPC_a(e_2)) \vee (EPC_a(e_1) \wedge \neg regr_{e_1}(EPC_{\neg a}(e_2)))) \wedge (((regr_{e_1}(EPC_{\neg a}(e_2)) \vee (EPC_{\neg a}(e_1) \wedge \neg regr_{e_1}(EPC_a(e_2))))))$ for all $a \in A$, that is, the effects do not contradict each other.

Now $app_{o_1}(s)$ in $app_{o_1;o_2}(s) = app_{o_2}(app_{o_1}(s))$ defined because $s \models c_1 \wedge \bigwedge_{a \in A} \neg (EPC_a(e_1) \wedge EPC_{\neg a}(e_1))$. Further $app_{o_1}(s) \models c_2$ by Theorem 3.7 because $s \models regr_{e_1}(c_2)$. From $s \not\models (regr_{e_1}(EPC_a(e_2)) \vee (EPC_a(e_1) \wedge \neg regr_{e_1}(EPC_{\neg a}(e_2)))) \wedge (((regr_{e_1}(EPC_{\neg a}(e_2)) \vee (EPC_{\neg a}(e_1) \wedge \neg regr_{e_1}(EPC_a(e_2))))))$ for all $a \in A$ logically follows $s \not\models regr_{e_1}(EPC_a(e_2)) \wedge regr_{e_1}(EPC_{\neg a}(e_2))$ for all $a \in A$. Hence by Theorem 3.7 $app_{o_1}(s) \not\models EPC_a(e_2) \wedge EPC_{\neg a}(e_2)$ for all $a \in A$, and by Lemma 3.3 $app_{o_2}(app_{o_1}(s))$ is defined.

For the other direction, since $app_{o_1}(s)$ is defined, $s \models c_1 \wedge \bigwedge_{a \in A} \neg (EPC_a(e_1) \wedge EPC_{\neg a}(e_1))$. Since $app_{o_2}(app_{o_1}(s))$ is defined, $s \models regr_{e_1}(c_2)$ by Theorem 3.7.

It remains to show that the effects of $o_1 \circ o_2$ do not contradict. Since $app_{o_2}(app_{o_1}(s))$ is defined $app_{o_1}(s) \not\models EPC_a(e_2) \wedge EPC_{\neg a}(e_2)$ and $s \not\models EPC_a(e_1) \wedge EPC_{\neg a}(e_1)$ for all $a \in A$. Hence by Theorem 3.7 $s \not\models regr_{e_1}(EPC_a(e_2)) \wedge regr_{e_1}(EPC_{\neg a}(e_2))$ for all $a \in A$. Assume that for some $a \in A$ $s \models regr_{e_1}(EPC_a(e_2)) \vee (EPC_a(e_1) \wedge \neg regr_{e_1}(EPC_{\neg a}(e_2)))$, that is, $a \in [o_1 \circ o_2]_s^{det}$. If $s \models regr_{e_1}(EPC_a(e_2))$ then $s \not\models regr_{e_1}(EPC_{\neg a}(e_2)) \vee \neg regr_{e_1}(EPC_a(e_2))$. Otherwise $s \models EPC_a(e_1) \wedge \neg regr_{e_1}(EPC_{\neg a}(e_2))$ and hence $s \not\models EPC_{\neg a}(e_1)$. Hence in both cases $s \not\models regr_{e_1}(EPC_{\neg a}(e_2)) \vee (EPC_{\neg a}(e_1) \wedge \neg regr_{e_1}(EPC_a(e_2)))$, that is, $\neg a \notin [o_1 \circ o_2]_s^{det}$. Therefore $app_{o_1 \circ o_2}(s)$ is defined.

We show that for any $a \in A$, $app_{o_1 \circ o_2}(s) \models a$ if and only if $app_{o_1}(app_{o_2}(s)) \models a$. Assume $app_{o_1 \circ o_2}(s) \models a$. Hence one of two cases hold.

1. Assume $s \models regr_{e_1}(EPC_a(e_2)) \vee (EPC_a(e_1) \wedge \neg regr_{e_1}(EPC_{\neg a}(e_2)))$.

   If $s \models regr_{e_1}(EPC_a(e_2))$ then by Theorem 3.7 and Lemma 3.3 $a \in [e_1]^{det}_{app_{o_1}(s)}$. Hence $app_{o_1;o_2}(s) \models a$.

   Assume $s \models EPC_a(e_1) \wedge \neg regr_{e_1}(EPC_{\neg a}(e_2))$. Hence by Lemma 3.3 $a \in [e_1]^{det}_s$ and $app_{o_1}(s) \models a$, and $app_{o_1}(s) \not\models EPC_{\neg a}(e_2)$ and $\neg a \notin [e_2]^{det}_{app_{o_1}(s)}$. Hence $app_{o_1;o_2}(s) \models a$.

2. Assume $s \models a$ and $s \not\models regr_{e_1}(EPC_{\neg a}(e_2)) \vee (EPC_{\neg a}(e_1) \wedge \neg regr_{e_1}(EPC_a(e_2)))$.

   Since $s \not\models regr_{e_1}(EPC_{\neg a}(e_2))$ by Theorem 3.7 $app_{o_1}(s) \not\models EPC_{\neg a}(e_2)$ and hence $\neg a \notin [e_2]^{det}_{app_{o_1}(s)}$.

   Since $s \not\models EPC_{\neg a}(e_1) \wedge \neg regr_{e_1}(EPC_a(e_2))$ by Lemma 3.3 $\neg a \notin [e_1]^{det}_s$ or $app_{e_1}(s) \models EPC_a(e_2)$ and hence by Theorem 3.7 $a \in [e_2]^{det}_{app_{o_1}(s)}$.

Hence either o_1 does not make a false, or if it makes, makes o_2 it true again so that $app_{o_1;o_2}(s) \models a$ in all cases.

Assume $app_{o_1;o_2}(s) \models a$. Hence one of the following three cases must hold.

1. If $a \in [e_2]^{det}_{app_{o_1}(s)}$ then by Lemma 3.3 $app_{o_1}(s) \models EPC_a(e_2)$. By Theorem 3.7 $s \models regr_{e_1}(EPC_a(e_2))$.

2. If $a \in [e_1]_s^{det}$ and $\neg a \notin [e_2]_{app_{o_1}(s)}^{det}$ then by Lemma 3.3 $app_{o_1}(s) \not\models EPC_{\neg a}(e_2)$. By Theorem 3.7 $s \models EPC_a(e_1) \wedge \neg regr_{e_1}(EPC_{\neg a}(e_2))$.
3. If $s \models a$ and $\neg a \notin [e_2]_{app_{o_1}(s)}^{det}$ and $\neg a \notin [e_1]_s^{det}$ then by Lemma 3.3 $app_{o_1}(s) \not\models EPC_{\neg a}(e_2)$. By Theorem 3.7 $s \not\models regr_{e_1}(EPC_{\neg a}(e_2))$.

   By Lemma 3.3 $s \not\models EPC_{\neg a}(e_1)$.

In the first two cases the antecedent of the first conditional in the definition of $o_1 \circ o_2$ is true, meaning that $app_{o_1 \circ o_2}(s) \models a$, and in the third case $s \models a$ and the antecedent of the second conditional effect is false, also meaning that $app_{o_1 \circ o_2}(s) \models a$. □

The above construction can be used to eliminate *sequential composition* from operator effects (Section 2.3.2).

3.2 Planning by heuristic search algorithms

Search for plans can be performed forwards or backwards respectively with progression or regression as described in Sections 3.1.1 and 3.1.2. There are several algorithms that can be used for the purpose, including depth-first search, breadth-first search, and iterative deepening, but without informed selection of operators these algorithms perform poorly.

The use of additional information for guiding search is essential for achieving efficient planning with general-purpose search algorithms. Algorithms that use heuristic estimates on the values of the nodes in the search space for guiding the search have been applied to planning very successfully. Some of the more sophisticated search algorithms that can be used are A$*$ [Hart *et al.*, 1968], WA$*$ [Pearl, 1984], IDA$*$ [Korf, 1985], and simulated annealing [Kirkpatrick *et al.*, 1983].

The effectiveness of these algorithms is dependent on good heuristics for guiding the search. The most important heuristics are estimates of distances between states. The distance is the minimum number of operators needed for reaching a state from another state. In Section 3.4 we will present techniques for estimating the distances between states and sets of states. In this section we will discuss how heuristic search algorithms are applied in planning.

When search proceeds forwards by progression starting from the initial state, we estimate the distance between the current state and the set of goal states. When search proceeds backwards by regression starting from the goal states, we estimate the distance between the initial state and the current set of goal states as computed by regression.

All the systematic heuristic search algorithms can easily be implemented to keep track of the search history which for planning equals the sequence of operators in the incomplete plan under consideration. Therefore the algorithms are started from the initial state I (forward search) or from the goal formula G (backward search) and then proceed forwards with progression or backwards with regression. Whenever the search successfully finishes, the plan can be recovered from the data structures maintained by the algorithm.

Local search algorithms do not keep track of the search history, and we have to define the elements of the search space as prefixes or suffixes of plans. For forward search we use sequences of operators (prefixes of plans)

$$o_1; o_2; \ldots; o_n.$$

The search starts from the empty sequence. The neighbors of an incomplete plan are obtained by adding an operator to the end of the plan or by deleting some of the last operators.

Definition 3.13 (Neighbors for local search with progression) *Let $\langle A, I, O, G\rangle$ be a succinct transition system. For forward search, the neighbors of an incomplete plan $o_1; o_2; \ldots; o_n$ are the following.*

1. $o_1; o_2; \ldots; o_n; o$ *for any* $o \in O$ *such that* $app_{o_1;\ldots;o_n;o}(I)$ *is defined*
2. $o_1; o_2; \ldots; o_i$ *for any* $i < n$

When $app_{o_1;o_2;\ldots;o_n}(I) \models G$ then $o_1; \ldots; o_n$ is a plan.

Also for backward search the incomplete plans are sequence of operators (suffixes of plans)

$$o_n; \ldots; o_1.$$

The search starts from the empty sequence. The neighbors of an incomplete plan are obtained by adding an operator to the beginning of the plan or by deleting some of the first operators.

Definition 3.14 (Neighbors for local search with regression) *Let $\langle A, I, O, G\rangle$ be a succinct transition system. For backward search, the children of an incomplete plan $o_n; \ldots; o_1$ are the following.*

1. $o; o_n; \ldots; o_1$ *for any* $o \in O$ *such that* $regr_{o;o_n;\ldots;o_1}(G)$ *is defined*
2. $o_i; \ldots; o_1$ *for any* $i < n$

When $I \models regr_{o_n;\ldots;o_1}(G)$ then $o_n; \ldots; o_1$ is a plan.

Backward search and forward search are not the only possibilities to define planning as a search problem. In partial-order planning [McAllester and Rosenblitt, 1991] the search space consists of incomplete plans which are partially ordered multisets of operators. The neighbors of an incomplete plan are those obtained by adding an operator or an ordering constraint. Incomplete plans can also be formalized as fixed length sequences of operators in which zero or more of the operators are missing. This leads to the constraint-based approaches to planning, including the planning as satisfiability approach that is presented in Section 3.6.

3.3 Reachability

The notion of reachability is important in defining whether a planning problem is solvable and in deriving techniques that speed up search for plans.

3.3.1 Distances

First we define the distances between states in a transition system in which all operators are deterministic. Heuristics in Section 3.4 are approximations of distances.

Definition 3.15 *Let I be an initial state and O a set of operators. Define the* forward distance sets D_i^{fwd} *for I, O that consist of those states that are reachable from I by at most i operator applications as follows.*

$$\begin{aligned} D_0^{fwd} &= \{I\} \\ D_i^{fwd} &= D_{i-1}^{fwd} \cup \{s | o \in O, s \in img_o(D_{i-1}^{fwd})\} \text{ for all } i \geq 1 \end{aligned}$$

Definition 3.16 *Let I be a state, O a set of operators, and $D_0^{fwd}, D_1^{fwd}, \ldots$ the forward distance sets for I, O. Then* the forward distance *of a state s from I is*

$$\delta_I^{fwd}(s) = \begin{cases} 0 \text{ if } s = I \\ i \text{ if } s \in D_i^{fwd} \backslash D_{i-1}^{fwd}. \end{cases}$$

If $s \notin D_i^{fwd}$ for all $i \geq 0$ then $\delta_I^{fwd}(s) = \infty$. States that have a finite forward distance are reachable *(from I with O).*

Distances can also be defined for formulae.

Definition 3.17 *Let ϕ be a formula. Then the* forward distance $\delta_I^{fwd}(\phi)$ *of ϕ is i if there is state s such that $s \models \phi$ and $\delta_I^{fwd}(s) = i$ and there is no state s' such that $s' \models \phi$ and $\delta_I^{fwd}(s) < i$. If $I \models \phi$ then $\delta_I^{fwd}(\phi) = 0$.*

A formula ϕ has a finite distance $< \infty$ if and only if $\langle A, I, O, \phi \rangle$ has a plan.

Reachability and distances are useful for implementing efficient planning systems. We mention two applications.

First, if we know that no state satisfying a formula ϕ is reachable from the initial states, then we know that no operator $\langle \phi, e \rangle$ can be a part of a plan, and we can ignore any such operator.

Second, distances help in finding a plan. Consider a deterministic planning problem with goal state G. We can now produce a shortest plan by finding an operator o so that $\delta_I^{fwd}(regr_o(G)) < \delta_I^{fwd}(G)$, using $regr_o(G)$ as the new goal state and repeating the process until the initial state I is reached.

Of course, since computing distances is in the worst case just as difficult as planning (PSPACE-complete) it is in general not useful to use subprocedures based on exact distances in a planning algorithm. Instead, different kinds of *approximations* of distances and reachability have to be used. The most important approximations allow the computation of useful reachability and distance information in polynomial time in the size of the succinct transition system. In Section 3.4 we will consider some of them.

3.3.2 Invariants

An *invariant* is a formula that is true in the initial state and in every state that is reached by applying an operator in a state in which it holds. Invariants are closely connected to reachability and distances: a formula ϕ is an invariant if and only if the distance of $\neg\phi$ from the initial state is ∞. Invariants can be used for example to speed up algorithms based on regression.

Definition 3.18 *Let I be a set of initial states and O a set of operators. An formula ϕ is* an invariant *of I, O if $s \models \phi$ for all states s that are reachable from I by a sequence of 0 or more operators in O.*

An invariant ϕ is *the strongest invariant* if $\phi \models \psi$ for any invariant ψ. The strongest invariant exactly characterizes the set of all states that are reachable from the initial state: for every state s, $s \models \phi$ if and only if s is reachable from the initial state. We say "the strongest invariant" even though there are actually several strongest invariants: if ϕ satisfies the properties of the strongest invariant, any other formula that is logically equivalent to ϕ, for example $\phi \vee \phi$, also does. Hence the uniqueness of the strongest invariant has to be understood up to logical equivalence.

Example 3.19 Consider a set of blocks that can be on the table or stacked on top of other blocks. Every block can be on at most one block and on every block there can be one block at most. The actions for moving the blocks can be described by the following schematic operators.

$$\langle \text{ontable}(x) \wedge \text{clear}(x) \wedge \text{clear}(y), \text{on}(x,y) \wedge \neg\text{clear}(y) \wedge \neg\text{ontable}(x) \rangle$$
$$\langle \text{clear}(x) \wedge \text{on}(x,y), \text{ontable}(x) \wedge \text{clear}(y) \wedge \neg\text{on}(x,y) \rangle$$
$$\langle \text{clear}(x) \wedge \text{on}(x,y) \wedge \text{clear}(z), \text{on}(x,z) \wedge \text{clear}(y) \wedge \neg\text{clear}(z) \wedge \neg\text{on}(x,y) \rangle$$

We consider the operators obtained by instantiating the schemata with the objects A, B and C. Let all the blocks be initially on the table. Hence the initial state satisfies the formula

$$\text{clear}(A) \wedge \text{clear}(B) \wedge \text{clear}(C) \wedge \text{ontable}(A) \wedge \text{ontable}(B) \wedge \text{ontable}(C) \wedge$$
$$\neg\text{on}(A,B) \wedge \neg\text{on}(A,C) \wedge \neg\text{on}(B,A) \wedge \neg\text{on}(B,C) \wedge \neg\text{on}(C,A) \wedge \neg\text{on}(C,B)$$

that determines the truth-values of all state variables uniquely. The strongest invariant of this problem is the conjunction of the following formulae.

$$\text{clear}(A) \leftrightarrow (\neg\text{on}(B,A) \wedge \neg\text{on}(C,A)) \qquad \text{clear}(B) \leftrightarrow (\neg\text{on}(A,B) \wedge \neg\text{on}(C,B))$$
$$\text{clear}(C) \leftrightarrow (\neg\text{on}(A,C) \wedge \neg\text{on}(B,C)) \qquad \text{ontable}(A) \leftrightarrow (\neg\text{on}(A,B) \wedge \neg\text{on}(A,C))$$
$$\text{ontable}(B) \leftrightarrow (\neg\text{on}(B,A) \wedge \neg\text{on}(B,C)) \qquad \text{ontable}(C) \leftrightarrow (\neg\text{on}(C,A) \wedge \neg\text{on}(C,B))$$
$$\neg\text{on}(A,B) \vee \neg\text{on}(A,C) \qquad \neg\text{on}(B,A) \vee \neg\text{on}(B,C)$$
$$\neg\text{on}(C,A) \vee \neg\text{on}(C,B)$$
$$\neg\text{on}(B,A) \vee \neg\text{on}(C,A) \qquad \neg\text{on}(A,B) \vee \neg\text{on}(C,B)$$
$$\neg\text{on}(A,C) \vee \neg\text{on}(B,C)$$
$$\neg(\text{on}(A,B) \wedge \text{on}(B,C) \wedge \text{on}(C,A)) \qquad \neg(\text{on}(A,C) \wedge \text{on}(C,B) \wedge \text{on}(B,A))$$

We can schematically give the invariants for any set X of blocks as follows.

$\text{clear}(x) \leftrightarrow \forall y \in X \backslash \{x\}.\neg\text{on}(y,x)$
$\text{ontable}(x) \leftrightarrow \forall y \in X \backslash \{x\}.\neg\text{on}(x,y)$
$\neg\text{on}(x,y) \vee \neg\text{on}(x,z)$ when $y \neq z$
$\neg\text{on}(y,x) \vee \neg\text{on}(z,x)$ when $y \neq z$
$\neg(\text{on}(x_1,x_2) \wedge \text{on}(x_2,x_3) \wedge \cdots \wedge \text{on}(x_{n-1},x_n) \wedge \text{on}(x_n,x_1))$ for all $n \geq 1, \{x_1,\ldots,x_n\} \subseteq X$

The last formula says that the *on* relation is acyclic. ■

3.4 Approximations of distances

The approximations of distances are based on the following idea. Instead of considering the number of operators required to reach individual states, we approximately compute the number of operators to reach a state in which a certain state variable has a certain value. So instead of using distances of states, we use distances of literals.

The estimates are not accurate for two reasons. First, and more importantly, distance estimation is done one state variable at a time and dependencies between state variables are ignored. Second, to achieve polynomial-time computation, satisfiability tests for a formula and a set of literals to test the applicability of an operator and to compute the distance estimate of a formula, have to be performed by an inaccurate polynomial-time algorithm that approximates NP-hard satisfiability testing. As we are interested in computing distance estimates efficiently the inaccuracy is a necessary and acceptable compromise.

3.4.1 Admissible max heuristic

We give a recursive procedure that computes a lower bound on the number of operator applications that are needed for reaching from a state I a state in which state variables $a \in A$ have certain values. This is by computing a sequence of sets D_i^{max} of literals. The set D_i^{max} consists of literals that are true in all states that have distance $\leq i$ from the state I.

Recall Definition 3.2 of $EPC_l(o)$ for literals l and operators $o = \langle c, e\rangle$:

$$EPC_l(o) = c \wedge EPC_l(e) \wedge \bigwedge_{a \in A} \neg(EPC_a(e) \wedge EPC_{\neg a}(e)).$$

Definition 3.20 *Let $L = A \cup \{\neg a | a \in A\}$ be the set of literals on A and I a state. Define the sets D_i^{max} for $i \geq 0$ as follows.*

$$\begin{array}{rcl} D_0^{max} & = & \{l \in L | I \models l\} \\ D_i^{max} & = & D_{i-1}^{max} \backslash \{l \in L | o \in O, D_{i-1}^{max} \cup \{EPC_{\bar{l}}(o)\} \text{ is satisfiable}\}, \text{ for } i \geq 1 \end{array}$$

Since we consider only finite sets A of state variables and $|D_0^{max}| = |A|$ and $D_{i+1}^{max} \subseteq D_i^{max}$ for all $i \geq 0$, necessarily $D_i^{max} = D_j^{max}$ for some $i \leq |A|$ and all $j > i$.

The above computation starts from the set D_0^{max} of all literals that are true in the initial state I. This set of literals characterizes those states that have distance 0 from the initial state. The initial state is the only such state.

Then we repeatedly compute sets of literals characterizing sets of states that are reachable with 1, 2 and more operators. Each set D_i^{max} is computed from the preceding set D_{i-1}^{max} as follows. For each operator o it is tested whether it is applicable in one of the distance $i-1$ states and whether it could make a literal l false. This is by testing whether $EPC_{\bar{l}}(o)$ is true in one of the distance $i-1$ states. If this is the case, the literal l could be false, and it will not be included in D_i^{max}.

The sets of states in which the literals D_i^{max} are true are an upper bound (set-inclusion) on the set of states that have forward distance i.

Theorem 3.21 *Let $D_i^{fwd}, i \geq 0$ be the forward distance sets and D_i^{max} the max-distance sets for I and O. Then for all $i \geq 0$, $D_i^{fwd} \subseteq \{s \in S | s \models D_i^{max}\}$ where S is the set of all states.*

Proof: By induction on i.

Base case $i = 0$: D_0^{fwd} consists of the unique initial state I and D_0^{max} consists of exactly those literals that are true in I, identifying it uniquely. Hence $D_i^{fwd} = \{s \in S | s \models D_i^{max}\}$.

Inductive case $i \geq 1$: Let s be any state in D_i^{fwd}. We show that $s \models D_i^{max}$. Let l be any literal in D_i^{max}.

Assume $s \in D_{i-1}^{fwd}$. As $D_i^{max} \subseteq D_{i-1}^{max}$ also $l \in D_{i-1}^{max}$. By the induction hypothesis $s \models l$.

Otherwise $s \in D_i^{fwd} \backslash D_{i-1}^{fwd}$. Hence there is $o \in O$ and $s_0 \in D_{i-1}^{fwd}$ with $s = app_o(s_0)$. By $D_i^{max} \subseteq D_{i-1}^{max}$ and the induction hypothesis $s_0 \models l$. As $l \in D_i^{max}$, by definition of D_i^{max} the set $D_{i-1}^{max} \cup \{EPC_{\bar{l}}(o)\}$ is not satisfiable. By $s_0 \in D_{i-1}^{fwd}$ and the induction hypothesis $s_0 \models D_{i-1}^{max}$. Hence $s_0 \not\models EPC_{\bar{l}}(o)$. By Lemma 3.3 applying o in s_0 does not make l false. Hence $s \models l$. □

The sets D_i^{max} can be used for estimating the distances of formulae. The distance of a formula is the minimum of the distances of states that satisfy the formula.

Definition 3.22 *Let ϕ be a formula. Define*

$$\delta_I^{max}(\phi) = \begin{cases} 0 \text{ iff } D_0^{max} \cup \{\phi\} \text{ is satisfiable} \\ d \text{ iff } D_d^{max} \cup \{\phi\} \text{ is satisfiable and } D_{d-1}^{max} \cup \{\phi\} \text{ is not satisfiable, for } d \geq 1. \end{cases}$$

Lemma 3.23 *Let I be a state, O a set of operators, and $D_0^{max}, D_1^{max}, \ldots$ the sets given in Definition 3.20 for I and O. Then $app_{o_1;\ldots;o_n}(I) \models D_n^{max}$ for any operators $\{o_1, \ldots, o_n\} \subseteq O$.*

Proof: By induction on n.

Base case $n = 0$: The length of the operator sequence is zero, and hence $app_\epsilon(I) = I$. The set D_0^{max} consists exactly of those literals that are true in s, and hence $I \models D_0^{max}$.

Inductive case $n \geq 1$: By the induction hypothesis $app_{o_1;\ldots;o_{n-1}}(I) \models D_{n-1}^{max}$.

Let l be any literal in D_n^{max}. We show it is true in $app_{o_1;\ldots;o_n}(I)$. Since $l \in D_n^{max}$ and $D_n^{max} \subseteq D_{n-1}^{max}$, also $l \in D_{n-1}^{max}$, and hence by the induction hypothesis $app_{o_1;\ldots;o_{n-1}}(I) \models l$. Since $l \in D_n^{max}$ it must be that $D_{n-1}^{max} \cup \{EPC_{\bar{l}}(o_n)\}$ is not satisfiable (definition of D_n^{max}) and further that $app_{o_1;\ldots;o_{n-1}}(I) \not\models EPC_{\bar{l}}(o_n)$. Hence applying o_n in $app_{o_1;\ldots;o_{n-1}}(I)$ does not make l false, and consequently $app_{o_1;\ldots;o_n}(I) \models l$.

□

The next theorem shows that the distance estimates given for formulae yield a lower bound on the number of actions needed to reach a state satisfying the formula.

Theorem 3.24 *Let I be a state, O a set of operators, ϕ a formula, and $D_0^{max}, D_1^{max}, \ldots$ the sets given in Definition 3.20 for I and O. If $app_{o_1;\ldots;o_n}(I) \models \phi$, then $D_n^{max} \cup \{\phi\}$ is satisfiable.*

Proof: By Lemma 3.23 $app_{o_1;\ldots;o_n}(I) \models D_n^{max}$. By assumption $app_{o_1;\ldots;o_n}(I) \models \phi$. Hence $D_n^{max} \cup \{\phi\}$ is satisfiable. □

Corollary 3.25 *Let I be a state and ϕ a formula. Then for any sequence $o_1, \ldots, o_n$ of operators such that $app_{o_1;\ldots;o_n}(I) \models \phi$, $n \geq \delta_I^{max}(\phi)$.*

The estimate $\delta_s^{max}(\phi)$ never overestimates the distance from s to ϕ and it is therefore an admissible heuristic. It may severely underestimate the distance, as discussed in the end of this section.

Distance estimation in polynomial time

The algorithm for computing the sets D_i^{max} runs in polynomial time except that the satisfiability tests for $D \cup \{\phi\}$ are instances of the NP-complete SAT problem. For polynomial time computation we perform these tests by a polynomial-time approximation that has the property that if $D \cup \{\phi\}$ is satisfiable then asat(D, ϕ) returns true, but not necessarily vice versa. A counterpart of Theorem 3.21 can be established when the satisfiability tests $D \cup \{\phi\}$ are replaced by tests asat(D, ϕ).

The function asat(D, ϕ) tests whether there is a state in which ϕ and the literals D are true, or equivalently, whether $D \cup \{\phi\}$ is satisfiable. This algorithm does not accurately test satisfiability, and may claim that $D \cup \{\phi\}$ is satisfiable even when it is not. This, however, never leads to

overestimating the distances, only underestimating. The algorithm runs in polynomial time and is defined as follows.

$$\begin{aligned}
\text{asat}(D, \bot) &= \text{false} \\
\text{asat}(D, \top) &= \text{true} \\
\text{asat}(D, a) &= \text{true iff } \neg a \notin D \text{ (for state variables } a \in A) \\
\text{asat}(D, \neg a) &= \text{true iff } a \notin D \text{ (for state variables } a \in A) \\
\text{asat}(D, \neg\neg\phi) &= \text{asat}(D, \phi) \\
\text{asat}(D, \phi_1 \vee \phi_2) &= \text{asat}(D, \phi_1) \text{ or asat}(D, \phi_2) \\
\text{asat}(D, \phi_1 \wedge \phi_2) &= \text{asat}(D, \phi_1) \text{ and asat}(D, \phi_2) \\
\text{asat}(D, \neg(\phi_1 \vee \phi_2)) &= \text{asat}(D, \neg\phi_1) \text{ and asat}(D, \neg\phi_2) \\
\text{asat}(D, \neg(\phi_1 \wedge \phi_2)) &= \text{asat}(D, \neg\phi_1) \text{ or asat}(D, \neg\phi_2)
\end{aligned}$$

In this and other recursive definitions about formulae the cases for $\neg(\phi_1 \wedge \phi_2)$ and $\neg(\phi_1 \vee \phi_2)$ are obtained respectively from the cases for $\phi_1 \vee \phi_2$ and $\phi_1 \wedge \phi_2$ by the De Morgan laws.

The reason why the satisfiability test is not accurate is that for formulae $\phi \wedge \psi$ (respectively $\neg(\phi \vee \psi)$) we make recursively two satisfiability tests that do not require that the subformulae ϕ and ψ (respectively $\neg\phi$ and $\neg\psi$) are *simultaneously* satisfiable.

We give a lemma that states the connection between $\text{asat}(D, \phi)$ and the satisfiability of $D \cup \{\phi\}$.

Lemma 3.26 *Let ϕ be a formula and D a consistent set of literals (it contains at most one of a and $\neg a$ for every $a \in A$.) If $D \cup \{\phi\}$ is satisfiable, then asat(D, ϕ) returns true.*

Proof: The proof is by induction on the structure of ϕ.

Base case 1, $\phi = \bot$: The set $D \cup \{\bot\}$ is not satisfiable, and hence the implication trivially holds.

Base case 2, $\phi = \top$: $\text{asat}(D, \top)$ always returns true, and hence the implication trivially holds.

Base case 3, $\phi = a$ for some $a \in A$: If $D \cup \{a\}$ is satisfiable, then $\neg a \notin D$, and hence $\text{asat}(D, a)$ returns true.

Base case 4, $\phi = \neg a$ for some $a \in A$: If $D \cup \{\neg a\}$ is satisfiable, then $a \notin D$, and hence $\text{asat}(D, \neg a)$ returns true.

Inductive case 1, $\phi = \neg\neg\phi'$ for some ϕ': The formulae are logically equivalent, and by the induction hypothesis we directly establish the claim.

Inductive case 2, $\phi = \phi_1 \vee \phi_2$: If $D \cup \{\phi_1 \vee \phi_2\}$ is satisfiable, then either $D \cup \{\phi_1\}$ or $D \cup \{\phi_2\}$ is satisfiable and by the induction hypothesis at least one of $\text{asat}(D, \phi_1)$ and $\text{asat}(D, \phi_2)$ returns true. Hence $\text{asat}(D, \phi_1 \vee \phi_2)$ returns true.

Inductive case 3, $\phi = \phi_1 \wedge \phi_2$: If $D \cup \{\phi_1 \wedge \phi_2\}$ is satisfiable, then both $D \cup \{\phi_1\}$ and $D \cup \{\phi_2\}$ are satisfiable and by the induction hypothesis both $\text{asat}(D, \phi_1)$ and $\text{asat}(D, \phi_2)$ return true. Hence $\text{asat}(D, \phi_1 \wedge \phi_2)$ returns true.

Inductive cases 4 and 5, $\phi = \neg(\phi_1 \vee \phi_2)$ and $\phi = \neg(\phi_1 \wedge \phi_2)$: Like cases 2 and 3 by logical equivalence. □

The other direction of the implication does not hold because for example $\text{asat}(\emptyset, a \wedge \neg a)$ returns true even though the formula is not satisfiable. The procedure is a polynomial-time approximation of the logical consequence test from a set of literals: $\text{asat}(D, \phi)$ always returns true if $D \cup \{\phi\}$ is satisfiable, but it may return true also when the set is not satisfiable.

Informativeness of the max heuristic

The max heuristic often underestimates distances. Consider an initial state in which all n state variables are false and a goal state in which all state variables are true and a set of n operators each of which is always applicable and makes one of the state variables true. The max heuristic assigns the distance 1 to the goal state although the distance is n.

The problem is that assigning every state variable the desired value requires a different operator, and taking the maximum number of operators for each state variable ignores this fact. In this case the actual distance is obtained as the *sum* of the distances suggested by each of the n state variables. In other cases the max heuristic works well when the desired state variable values can be reached with the same operators.

Next we will consider heuristics that are not admissible like the max heuristic but in many cases provide a much better estimate of the distances.

3.4.2 Inadmissible additive heuristic

The max heuristic is very optimistic about the distances, and in many cases very seriously underestimates them. If two goal literals have to be made true, the maximum of the goal costs (distances) is assumed to be the combined cost. This however is only accurate when the easier goal is achieved for free while achieving the more difficult goal. Often the goals are independent and then a more accurate estimate would be the sum of the individual costs. This suggests another heuristic, first considered by Bonet and Geffner [2001] as a more practical variant of the max heuristic in the previous section. Our formalization differs from the one given by Bonet and Geffner.

Definition 3.27 *Let I be a state and $L = A \cup \{\neg a | a \in A\}$ the set of literals. Define the sets D_i^+ for $i \geq 0$ as follows.*

$$\begin{aligned} D_0^+ &= \{l \in L | I \models l\} \\ D_i^+ &= D_{i-1}^+ \backslash \{l \in L | o \in O, \mathit{cost}(\mathit{EPC}_{\bar{l}}(o), i) < i\} \text{ for all } i \geq 1 \end{aligned}$$

We define $\mathit{cost}(\phi, i)$ by the following recursive definition.

$$\begin{aligned} \mathit{cost}(\bot, i) &= \infty \\ \mathit{cost}(\top, i) &= 0 \\ \mathit{cost}(a, i) &= 0 \text{ if } \neg a \notin D_0^+, \text{ for } a \in A \\ \mathit{cost}(\neg a, i) &= 0 \text{ if } a \notin D_0^+, \text{ for } a \in A \\ \mathit{cost}(a, i) &= j \text{ if } \neg a \in D_{j-1}^+ \backslash D_j^+ \text{ for some } j < i \\ \mathit{cost}(\neg a, i) &= j \text{ if } a \in D_{j-1}^+ \backslash D_j^+ \text{ for some } j < i \\ \mathit{cost}(a, i) &= \infty \text{ if } \neg a \in D_j^+ \text{ for all } j < i \\ \mathit{cost}(\neg a, i) &= \infty \text{ if } a \in D_j^+ \text{ for all } j < i \\ \mathit{cost}(\phi_1 \vee \phi_2, i) &= \min(\mathit{cost}(\phi_1, i), \mathit{cost}(\phi_2, i)) \\ \mathit{cost}(\phi_1 \wedge \phi_2, i) &= \mathit{cost}(\phi_1, i) + \mathit{cost}(\phi_2, i) \\ \mathit{cost}(\neg\neg\phi, i) &= \mathit{cost}(\phi, i) \\ \mathit{cost}(\neg(\phi_1 \wedge \phi_2), i) &= \min(\mathit{cost}(\neg\phi_1, i), \mathit{cost}(\neg\phi_2, i)) \\ \mathit{cost}(\neg(\phi_1 \vee \phi_2), i) &= \mathit{cost}(\neg\phi_1, i) + \mathit{cost}(\neg\phi_2, i) \end{aligned}$$

Note that a variant of the definition of the max heuristic could be obtained by replacing the sum $+$ in the definition of costs of conjunctions by $\max$. The definition of $\text{cost}(\phi, i)$ approximates

satisfiability tests similarly to the definition of asat(D, ϕ) by ignoring the dependencies between state variables.

Similarly to max distances we can define distances of formulae.

Definition 3.28 *Let ϕ be a formula. Define*

$$\delta_I^+(\phi) = \mathit{cost}(\phi, n)$$

where n is the smallest i such that $D_i^+ = D_{i-1}^+$.

The following theorem shows that the distance estimates given by the sum heuristic for literals are at least as high as those given by the max heuristic.

Theorem 3.29 *Let $D_i^{max}, i \geq 0$ be the sets defined in terms of the approximate satisfiability tests asat(D, ϕ). Then $D_i^{max} \subseteq D_i^+$ for all $i \geq 0$.*

Proof: The proof is by induction on i.

Base case $i = 0$: By definition $D_0^+ = D_0^{max}$.

Inductive case $i \geq 1$: We have to show that $D_{i-1}^{max} \backslash \{l \in L | o \in O, \text{asat}(D_{i-1}^{max}, \mathit{EPC}_{\bar{l}}(o))\} \subseteq D_{i-1}^+ \backslash \{l \in L | o \in O, \text{cost}(\mathit{EPC}_{\bar{l}}(o), i) < i\}$. By the induction hypothesis $D_{i-1}^{max} \subseteq D_{i-1}^+$. It is sufficient to show that cost$(\mathit{EPC}_{\bar{l}}(o), i) < i$ implies asat$(D_{i-1}^{max}, \mathit{EPC}_{\bar{l}}(o))$.

We show this by induction on the structure of $\phi = \mathit{EPC}_{\bar{l}}(o)$.

Induction hypothesis: cost$(\phi, i) < i$ implies asat(D_{i-1}^{max}, ϕ)=true.

Base case 1, $\phi = \bot$: cost$(\bot, i) = \infty$ and asat$(D_i^{max}, \bot)$=false.

Base case 2, $\phi = \top$: cost$(\top, i) = 0$ and asat$(D_i^{max}, \top)$=true.

Base case 3, $\phi = a$: If cost$(a, i) < i$ then $\neg a \notin D_j^+$ for some $j < i$ or $\neg a \notin D_0^+$. Hence $\neg a \notin D_{i-1}^+$. By the outer induction hypothesis $\neg a \notin D_{i-1}^{max}$ and consequently $\neg a \notin D_i^{max}$. Hence asat(D_i^{max}, a)=true.

Base case 4, $\phi = \neg a$: Analogous to the case $\phi = a$.

Inductive case 5, $\phi = \phi_1 \vee \phi_2$: Assume cost$(\phi_1 \vee \phi_2, i) < i$. Since cost$(\phi_1 \vee \phi_2, i) = \min(\text{cost}(\phi_1, i), \text{cost}(\phi_2, i))$, either cost$(\phi_1, i) < i$ or cost$(\phi_2, i) < i$. By the induction hypothesis cost$(\phi_1, i) < i$ implies asat(D_{i-1}^{max}, ϕ_1), and cost$(\phi_2, i) < i$ implies asat(D_{i-1}^{max}, ϕ_2). Hence either asat(D_{i-1}^{max}, ϕ_1) or asat(D_{i-1}^{max}, ϕ_2). Therefore by definition asat$(D_{i-1}^{max}, \phi_1 \vee \phi_2)$.

Inductive case 6, $\phi = \phi_1 \wedge \phi_2$: Assume cost$(\phi_1 \wedge \phi_2, i) < i$. Since $i \geq 1$ and cost$(\phi_1 \vee \phi_2, i) =$ cost(ϕ_1, i) + cost(ϕ_2, i), both cost$(\phi_1, i) < i$ and cost$(\phi_2, i) < i$. By the induction hypothesis cost$(\phi_1, i) < i$ implies asat(D_{i-1}^{max}, ϕ_1), and cost$(\phi_2, i) < i$ implies asat(D_{i-1}^{max}, ϕ_2). Hence both asat(D_{i-1}^{max}, ϕ_1) an asat(D_{i-1}^{max}, ϕ_2). Therefore by definition asat$(D_{i-1}^{max}, \phi_1 \wedge \phi_2)$.

Inductive case 7, $\phi = \neg\neg\phi_1$: By the induction hypothesis cost$(\phi_1, i) < i$ implies asat(D_{i-1}^{max}, ϕ_1). By definition cost$(\neg\neg\phi_1, i) =$ cost(ϕ_1, i) and asat$(D, \neg\neg\phi) =$ asat(D, ϕ). By the induction hypothesis cost$(\neg\neg\phi_1, i) < i$ implies asat$(D_{i-1}^{max}, \neg\neg\phi_1)$.

Inductive case 8, $\phi = \neg(\phi_1 \vee \phi_2)$: Analogous to the case $\phi = \phi_1 \wedge \phi_2$.

Inductive case 9, $\phi = \neg(\phi_1 \wedge \phi_2)$: Analogous to the case $\phi = \phi_1 \vee \phi_2$. □

That the sum heuristic gives higher estimates than the max heuristic could in many cases be viewed as an advantage because the estimates would be more accurate. However, in some cases this leads to overestimating the actual distance, and therefore the sum distances are not an admissible heuristic.

Example 3.30 Consider an initial state such that $I \models \neg a \wedge \neg b \wedge \neg c$ and the operator $\langle \top, a \wedge b \wedge c \rangle$. A state satisfying $a \wedge b \wedge c$ is reached by this operator in one step but $\delta_I^+(a \wedge b \wedge c) = 3$. ■

3.4.3 Relaxed plan heuristic

The max heuristic and the additive heuristic represent two extremes. The first assumes that sets of operators required for reaching the individual goal literals maximally overlap in the sense that the operators needed for the most difficult goal literal include the operators needed for all the remaining ones. The second assumes that the required operators are completely disjoint.

Usually, of course, the reality is somewhere in between and which notion is better depends on the properties of the operators. This suggests yet another heuristic: we attempt to find a set of operators that approximates, in a sense that will become clear later, the smallest set of operators that are needed to reach a state from another state. This idea has been considered by Hoffman and Nebel [2001]. If the approximation is exact, the cardinality of this set equals the actual distance between the states. The approximation may both overestimate and underestimate the actual distance, and hence does not yield an admissible heuristic.

The idea of the heuristic is the following. We first choose a set of goal literals the truth of which is sufficient for the truth of G. These literals must be reachable in the sense of the sets D_i^{max} which we defined earlier. Then we identify those goal literals that were the last to become reachable and a set of operators making them true. A new goal formula represents the conditions under which these operator can make the literals true, and a new set of goal literals is produced by a simplified form of regression from the new goal formula. The computation is repeated until we have a set of goal literals that are true in the initial state.

The function goals(D, ϕ) recursively finds a set M of literals such that $M \models \phi$ and each literal in M is consistent with D. Note that M itself is not necessarily consistent, for example for $D = \emptyset$ and $\phi = a \wedge \neg a$ we get $M = \{a, \neg a\}$. If a set M is found goals$(D, \phi) = \{M\}$ and otherwise goals$(D, \phi) = \emptyset$.

Definition 3.31 *Let D be a set of literals.*

$$
\begin{aligned}
\textit{goals}(D, \bot) &= \emptyset \\
\textit{goals}(D, \top) &= \{\emptyset\} \\
\textit{goals}(D, a) &= \{\{a\}\} \textit{ if } \neg a \notin D \\
\textit{goals}(D, a) &= \emptyset \textit{ if } \neg a \in D \\
\textit{goals}(D, \neg a) &= \{\{\neg a\}\} \textit{ if } a \notin D \\
\textit{goals}(D, \neg a) &= \emptyset \textit{ if } a \in D \\
\textit{goals}(D, \neg\neg\phi) &= \textit{goals}(D, \phi) \\
\textit{goals}(D, \phi_1 \vee \phi_2) &= \begin{cases} \textit{goals}(D, \phi_1) \textit{ if goals}(D, \phi_1) \neq \emptyset \\ \textit{goals}(D, \phi_2) \textit{ otherwise} \end{cases} \\
\textit{goals}(D, \phi_1 \wedge \phi_2) &= \begin{cases} \{L_1 \cup L_2\} & \textit{if goals}(D, \phi_1) = \{L_1\} \textit{ and goals}(D, \phi_2) = \{L_2\} \\ \emptyset & \textit{otherwise} \end{cases} \\
\textit{goals}(D, \neg(\phi_1 \wedge \phi_2)) &= \begin{cases} \textit{goals}(D, \neg\phi_1) \textit{ if goals}(D, \neg\phi_1) \neq \emptyset \\ \textit{goals}(D, \neg\phi_2) \textit{ otherwise} \end{cases} \\
\textit{goals}(D, \neg(\phi_1 \vee \phi_2)) &= \begin{cases} \{L_1 \cup L_2\} & \textit{if goals}(D, \neg\phi_1) = \{L_1\} \textit{ and goals}(D, \neg\phi_2) = \{L_2\} \\ \emptyset & \textit{otherwise} \end{cases}
\end{aligned}
$$

Above in the case for $\phi_1 \vee \phi_2$ if both ϕ_1 and ϕ_2 yield a set of goal literals the set for ϕ_1 is always chosen. A practically better implementation is to choose the smaller of the two sets.

Lemma 3.32 *Let D be a set of literals and ϕ a formula.*

1. *goals$(D, \phi) \neq \emptyset$ if and only if asat$(D, \phi) =$ true.*
2. *If goals$(D, \phi) = \{M\}$ then $\{\bar{l} | l \in M\} \cap D = \emptyset$ and asat$(D, \bigwedge_{l \in M} l) =$ true.*

Proof:

1. This is by an easy induction proof on the structure of ϕ based on the definitions of asat(D, ϕ) and goals(D, ϕ).
2. This is because $\bar{l} \notin D$ for all $l \in M$. This can be shown by a simple induction proof.

□

Lemma 3.33 *Let D and $D' \subseteq D$ be sets of literals. If goals$(D, \phi) = \emptyset$ and goals$(D', \phi) = \{M\}$ for some M, then there is $l \in M$ such that $\bar{l} \in D \backslash D'$.*

Proof: Proof is by induction in the structure of formulae ϕ.

Induction hypothesis: If goals$(D, \phi) = \emptyset$ and goals$(D', \phi) = \{M\}$ for some M, then there is $l \in M$ such that $\bar{l} \in D \backslash D'$.

Base cases 1 & 2, $\phi = \top$ and 2 $\phi = \bot$: Trivial as the condition cannot hold.

Base case 3, $\phi = a$: If goals$(D, a) = \emptyset$ and goals$(D', a) = M = \{\{a\}\}$, then respectively $\neg a \in D$ and $\neg a \notin D'$. Hence there is $a \in M$ such that $\overline{a} \in D \backslash D'$.

Inductive case 1, $\phi = \neg\neg\phi'$: By the induction hypothesis as goals$(D, \neg\neg\phi') =$ goals(D, ϕ').

Inductive case 2, $\phi = \phi_1 \vee \phi_2$: Assume goals$(D, \phi_1 \vee \phi_2) = \emptyset$ and goals$(D', \phi_1 \vee \phi_2) = \{M\}$ for some M. Hence goals$(D, \phi_1) = \emptyset$ and goals$(D, \phi_2) = \emptyset$, and goals$(D', \phi_1) = \{M\}$ or goals$(D', \phi_2) = \{M\}$. Hence by the induction hypothesis with ϕ_1 or ϕ_2 there is $l \in M$ such that $\bar{l} \in D \backslash D'$.

Inductive case 3, $\phi = \phi_1 \wedge \phi_2$: Assume goals$(D, \phi_1 \wedge \phi_2) = \emptyset$ and goals$(D', \phi_1 \wedge \phi_2) = \{M\}$ for some M. Hence goals$(D, \phi_1) = \emptyset$ or goals$(D, \phi_2) = \emptyset$, and goals$(D', \phi_1) = \{L_1\}$ and goals$(D', \phi_2) = \{L_2\}$ for some L_1 and L_2 such that $M = L_1 \cup L_2$. Hence by the induction hypothesis with ϕ_1 or ϕ_2 there is either $l \in L_1$ or $l \in L_2$ such that $\bar{l} \in D \backslash D'$.

Inductive cases $\phi = \neg(\phi_1 \wedge \phi_2)$ and $\phi = \neg(\phi_1 \vee \phi_2)$ are analogous to cases 2 and 3. □

Definition 3.34 *Define $\delta_I^{rlx}(\phi) =$ relaxedplan(A, I, O, ϕ).*

Like the sum heuristic, the relaxed plan heuristic gives higher distance estimates than the max heuristic.

Theorem 3.35 *Let ϕ be a formula and $\delta_I^{max}(\phi)$ the max-distance defined in terms of asat(D, ϕ). Then $\delta_I^{rlx}(\phi) \geq \delta_I^{max}(\phi)$.*

Proof: We have to show that for any formula G the procedure call *relaxedplan*(A,I,O,G) returns a number $\geq \delta_I^{max}(G)$.

First, the procedure returns ∞ if and only if asat$(D_i^{max}, G) =$ false for all $i \geq 0$. In this case by definition $\delta_I^{max}(G) = \infty$.

1: **procedure** relaxedplan(A,I,O,G);
2: $L := A \cup \{\neg a | a \in A\}$; *(* Set of all literals *)*
3: compute sets D_i^{max} as in Definition 3.20;
4: **if** asat$(D_i^{max}, G) =$ false for all $i \geq 0$ **then return** ∞; *(* Goal not reachable *)*
5: $t := \delta_I^{max}(G)$;
6: $L_{t+1}^G := \emptyset$;
7: $N_{t+1} := \emptyset$;
8: $G_t := G$;
9: **for** $i := t$ **downto** 1 **do**
10: **begin**
11: $L_i^G := (L_{i+1}^G \backslash N_{i+1}) \cup \{l \in M | M \in \text{goals}(D_i^{max}, G_i)\}$; *(* The goal literals *)*
12: $N_i := \{l \in L_i^G | \bar{l} \in D_{i-1}^{max}\}$; *(* Goal literals that become true between $i-1$ and i *)*
13: $T_i :=$ a minimal subset of O so that $N_i \subseteq \{l \in L | o \in T_i, \text{asat}(D_{i-1}^{max}, EPC_l(o))\}$;
14: $G_{i-1} := \bigwedge_{l \in N_i} \bigvee \{EPC_l(o) | o \in T_i\}$; *(* New goal formula *)*
15: **end**
16: **return** $|T_1| + |T_2| + \cdots + |T_t|$;

Figure 3.1: Algorithm for finding a relaxed plan

Otherwise $t = \delta_I^{max}(G)$. Now $t = 0$ if and only if asat$(D_0^{max}, G) =$ true. In this case the procedure returns 0 without iterating the loop starting on line 9.

We show that if $t \geq 1$ then for every $i \in \{1, \ldots, t\}$ the set T_i is non-empty, entailing $|T_1| + \cdots + |T_t| \geq t = \delta_I^{max}(G)$. This is by an induction proof from t to 1.

We use the following auxiliary result. If asat$(D_{i-1}^{max}, G_i) =$ false and asat$(D_i^{max}, G_i) =$ true and $\bar{l} \notin D_i^{max}$ for all $l \in L_i^G$ then T_i is well-defined and $T_i \neq \emptyset$. The proof is as follows.

> By Lemma 3.32 goals$(D_{i-1}^{max}, G_i) = \emptyset$ and goals$(D_i^{max}, G_i) = \{M\}$ for some M. By Lemma 3.33 there is $l \in M$ such that $\bar{l} \in D_{i-1}^{max}$ and hence $N_i \neq \emptyset$. By definition $\bar{l} \in D_{i-1}^{max}$ for all $l \in N_i$. By $N_i \subseteq L_i^G$ and the assumption about L_i^G $\bar{l} \notin D_i^{max}$ for all $l \in N_i$. Hence $\bar{l} \in D_{i-1}^{max} \backslash D_i^{max}$ for all $l \in N_i$. Hence by definition of D_i^{max} for every $l \in N_i$ there is $o \in O$ such that asat$(D_{i-1}^{max}, EPC_l(o))$. Hence there is $T_i \subseteq O$ so that $N_i \subseteq \{l \in L | o \in T_i, \text{asat}(D_{i-1}^{max}, EPC_l(o))\}$ and the value of T_i is defined. As $N_i \neq \emptyset$ also $T_i \neq \emptyset$.

In the induction proof we establish the assumptions of the auxiliary result and then invoke the auxiliary result itself.

Induction hypothesis: For all $j \in \{i, \ldots, t\}$

1. $\bar{l} \notin D_j^{max}$ for all $l \in L_j^G$,
2. asat$(D_j^{max}, G_j) =$ true and asat$(D_{j-1}^{max}, G_j) =$ false, and
3. $T_j \neq \emptyset$.

Base case $i = t$:

1. $\bar{l} \notin D_t^{max}$ for all $l \in L_t^G$ by (2) of Lemma 3.32 because $L_t^G = \{l \in \text{goals}(D_t^{max}, G_t)\}$.
2. As $t = \delta_I^{max}(G_t)$ by definition asat$(D_{t-1}^{max}, G_t) =$ false and asat$(D_t^{max}, G_t) =$ true.

3. By the auxiliary result from the preceding case.

Inductive case $i < t$:

1. We have $\bar{l} \notin D_i^{max}$ for all $l \in L_i^G$ because $L_i^G = (L_{i+1}^G \setminus N_{i+1}) \cup \{l \in \text{goals}(D_i^{max}, G_i)\}$ and by the induction hypothesis $\bar{l} \notin D_{i+1}^{max}$ for all $l \in L_{i+1}^G$ and by (2) of Lemma 3.32 $\bar{l} \notin D_i^{max}$ for all $l \in M$ for $M \in \text{goals}(D_i^{max}, G_i)$.

2. By definition $G_i = \bigwedge_{l \in N_{i+1}} \bigvee \{EPC_l(o) | o \in T_{i+1}\}$. By definition of T_{i+1} for every $l \in N_{i+1}$ there is $o \in T_{i+1}$ such that $\text{asat}(D_i^{max}, EPC_l(o)) = \text{true}$. By definition of $\text{asat}(D_i^{max}, \phi_1 \vee \phi_2)$ and $\text{asat}(D_i^{max}, \phi_1 \wedge \phi_2)$ for ϕ_1 and ϕ_2 also $\text{asat}(D_i^{max}, G_i) = \text{true}$.

   Then we show that $\text{asat}(D_{i-1}^{max}, G_i) = \text{false}$. By definition of D_i^{max}, $\text{asat}(D_{i-1}^{max}, EPC_{\bar{l}}(o)) = \text{false}$ for all $l \in D_i^{max}$ and $o \in O$. Hence $\text{asat}(D_{i-1}^{max}, EPC_l(o)) = \text{false}$ for all $l \in N_{i+1}$ and $o \in O$ because $\bar{l} \in D_i^{max}$. Hence $\text{asat}(D_{i-1}^{max}, EPC_l(o)) = \text{false}$ for all $l \in N_{i+1}$ and $o \in T_{i+1}$ because $T_{i+1} \subseteq O$. By definition $G_i = \bigwedge_{l \in N_{i+1}} \bigvee \{EPC_l(o) | o \in T_{i+1}\}$. Hence by definition of $\text{asat}(D, \phi)$ also $\text{asat}(D_{i-1}^{max}, G_i) = \text{false}$.

3. By the auxiliary result from the preceding case.

□

3.5 Algorithm for computing invariants

Planning with backward search and regression suffers from the following problem. Often only a fraction of all valuations of state variables represent states that are reachable from the initial state and represent possible world states. The goal formula and many of the formulae produced by regression often represent many unreachable states. If the formulae represent only unreachable states a planning algorithm may waste a lot of effort determining that a certain sequence of actions is not the suffix of any plan[1]. Also planning with propositional logic (Section 3.6) suffers from the same problem.

Planning can be made more efficient by restricting search to states that are reachable from the initial state. However, determining whether a given state is reachable from the initial state is PSPACE-complete. Consequently, exact information on the reachability of states could not be used for speeding up the basic forward and backward search algorithms: solving the subproblem would be just as complex as solving the problem itself.

In this section we will present a polynomial time algorithm for computing a class of invariants that approximately characterize the set of reachable states. These invariants help in improving the efficiency of planning algorithms based on backward search and on satisfiability testing in the propositional logic (Section 3.6).

Our algorithm computes invariants that are clauses with at most n literals, for some fixed n. For representing the strongest invariant arbitrarily high n may be needed. Although the runtime is polynomial for any fixed n, the runtimes grow quickly as n increases. However, for many applications short invariants of length $n = 2$ are sufficient, and longer invariants are less important.

[1] A symmetric problem arises with forward search because with progression one may reach states from which goal states are unreachable.

```
 1: procedure preserved(φ,C,o);
 2: φ = l_1 ∨ ··· ∨ l_n for some l_1, ..., l_n and o = ⟨c, e⟩ for some c and e;
 3: for each l ∈ {l_1, ..., l_n} do
 4:   if C ∪ {EPC_l̄(o)} is unsatisfiable then goto OK;        (* l cannot become false. *)
 5:   for each l' ∈ {l_1, ..., l_n}\{l} do   (* Otherwise another literal in φ must be true. *)
 6:     if C ∪ {EPC_l̄(o)} ⊨ EPC_l'(o) then goto OK;          (* l' becomes true. *)
 7:     if C ∪ {EPC_l̄(o)} ⊨ l' ∧ ¬EPC_l̄'(o) then goto OK;    (* l' was and stays true. *)
 8:   end do
 9:   return false;                  (* Truth of the clause could not be guaranteed. *)
10:   OK:
11: end do
12: return true;
```

Figure 3.2: Algorithm that tests whether o may falsify $l_1 \vee \cdots \vee l_n$ in a state satisfying C

The algorithm first computes the set of all 1-literal clauses that are true in the initial state. This set exactly characterizes the set of distance 0 states consisting of the initial state only. Then the algorithm considers the application of every operator. If an operator is applicable it may make some of the clauses false. These clauses are removed and replaced by weaker clauses which are also tested against every operator. When no further clauses are falsified, we have a set of clauses that are guaranteed to be true in all distance 1 states. This computation is repeated for distances 2, 3, and so on, until the clause set does not change. The resulting clauses are invariants because they are true after any number of operator applications.

The flavor of the algorithm is similar to the distance estimation in Section 3.4: starting from a description of what is possible in the initial state, inductively determine what is possible after i operator applications. In contrast to the distance estimation method in Section 3.4 the state sets are characterized by sets of clauses instead of sets of literals.

Let C_i be a set of clauses that characterizes those states that are reachable by i operator applications. Similarly to distance computation, we consider for each operator and for each clause in C_i whether applying the operator may make the clause false. If it can, the clause could be false after i operator applications and therefore will not be in the set C_{i+1}.

Figure 3.2 gives an algorithm that tests whether applying an operator $o \in O$ in some state s may make a formula $l_1 \vee \cdots \vee l_n$ false assuming that $s \models C \cup \{l_1 \vee \cdots \vee l_n\}$.

The algorithm performs a case analysis for every literal in the clause, testing in each case whether the clause remains true: if a literal becomes false, either another literal becomes true simultaneously or another literal was true before and does not become false.

Lemma 3.36 *Let C be a set of clauses, $\phi = l_1 \vee \cdots \vee l_n$ a clause, and o an operator. If preserved(ϕ,C,o) returns* true*, then $app_o(s) \models \phi$ for any state s such that $s \models C \cup \{\phi\}$ and o is applicable in s. (It may under these conditions also return* false*).*

Proof: Assume s is a state such that $s \models C \wedge \phi$, $app_o(s)$ is defined and $app_o(s) \not\models \phi$. We show that the procedure returns *false*.

Since $s \models \phi$ and $app_o(s) \not\models \phi$ at least one literal in ϕ is made false by o. Let $\{l_1^\perp, \ldots, l_m^\perp\} \subseteq \{l_1, \ldots, l_n\}$ be the set of all such literals. Hence $s \models l_1^\perp \wedge \cdots \wedge l_m^\perp$ and $\{\overline{l_1^\perp}, \ldots, \overline{l_m^\perp}\} \subseteq [e]_s^{det}$. The literals in $\{l_1, \ldots, l_n\}\backslash\{l_1^\perp, \ldots, l_m^\perp\}$ are false in s and o does not make them true.

```
 1: procedure invariants(A, I, O, n);
 2: C := {a ∈ A | I ⊨ a} ∪ {¬a | a ∈ A, I ⊭ a};        (* Clauses true in the initial state *)
 3: repeat
 4:   C' := C;
 5:   for each o ∈ O and l_1 ∨ ··· ∨ l_m ∈ C such that not preserved(l_1 ∨ ··· ∨ l_m,C',o) do
 6:     C := C\{l_1 ∨ ··· ∨ l_m};
 7:     if m < n then                                   (* Clause length within pre-defined limit. *)
 8:       begin                                         (* Add weaker clauses. *)
 9:         C := C ∪ {l_1 ∨ ··· ∨ l_m ∨ a | a ∈ A, {a, ¬a} ∩ {l_1, ..., l_m} = ∅};
10:         C := C ∪ {l_1 ∨ ··· ∨ l_m ∨ ¬a | a ∈ A, {a, ¬a} ∩ {l_1, ..., l_m} = ∅};
11:       end
12:   end do
13: until C = C';
14: return C;
```

Figure 3.3: Algorithm for computing a set of invariant clauses

Choose any $l \in \{l_1^\perp, \ldots, l_m^\perp\}$. We show that when the outermost *for each* loop starting on line 3 considers l the procedure will return *false*.

Since $\overline{l} \in [e]_s^{det}$ and o is applicable in s by Lemma 3.3 $s \models EPC_{\overline{l}}(o)$. Since by assumption $s \models C$, the condition of the *if* statement on line 4 is not satisfied and the execution proceeds by iteration of the inner *for each* loop.

Let l' be any of the literals in ϕ except l. Since $app_o(s) \not\models \phi$, $l' \notin [e]_s^{det}$. Hence by Lemma 3.3 $s \not\models EPC_{l'}(o)$, and as $s \models C \cup \{EPC_{\overline{l}}(o)\}$ the condition of the *if* statement on line 6 is not satisfied and the execution continues from line 7. Analyze two cases.

1. If $l' \in \{l_1^\perp, \ldots, l_m^\perp\}$ then by assumption $\overline{l'} \in [e]_s^{det}$ and by Lemma 3.3 $s \models EPC_{\overline{l'}}(o)$. Hence $C \cup \{EPC_{\overline{l}}(o)\} \not\models \neg EPC_{\overline{l'}}(o)$ and the condition of the *if* statement on line 7 is not satisfied.

2. If $l' \notin \{l_1^\perp, \ldots, l_m^\perp\}$ then $s \not\models l'$. Hence $C \cup \{EPC_{\overline{l}}(o)\} \not\models l'$ and the condition of the *if* statement on line 7 is not satisfied.

Hence on none of the iterations of the inner *for each* loop is a *goto OK* executed, and as the loop exits, the procedure returns *false*. □

Figure 3.3 gives the algorithm for computing invariants consisting of at most n literals. The loop on line 5 is repeated until there are no $o \in O$ and clauses ϕ in C such that preserved(ϕ,C',o) returns false. This exit condition for the loop is critical for the correctness proof.

Theorem 3.37 *Let A be a set of state variables, I a state, O a set of operators, and $n \geq 1$ an integer. Then the procedure call invariants(A, I, O, n) returns a set C of clauses with at most n literals so that for any sequence $o_1; \ldots; o_m$ of operators from O $app_{o_1;\ldots;o_m}(I) \models C$.*

Proof: Let C_0 be the value first assigned to the variable C in the procedure *invariants*, and $C_1, C_2, \ldots$ be the values of the variable in the end of each iteration of the outermost *repeat* loop.

Induction hypothesis: for every $\{o_1, \ldots, o_i\} \subseteq O$ and $\phi \in C_i$, $app_{o_1;\ldots;o_i}(I) \models \phi$.

Base case $i = 0$: $app_\epsilon(I)$ for the empty sequence is by definition I itself, and by construction C_0 consists of only formulae that are true in the initial state.

Inductive case $i \geq 1$: Take any $\{o_1, \ldots, o_i\} \subseteq O$ and $\phi \in C_i$. First notice that preserved(ϕ,C_i,o) returns *true* because otherwise ϕ could not be in C_i. Analyze two cases.

1. If $\phi \in C_{i-1}$, then by the induction hypothesis $app_{o_1;\ldots;o_{i-1}}(I) \models \phi$. Since $\phi \in C_i$ preserved(ϕ,C_{i-1},o) returns *true*. Hence by Lemma 3.36 $app_{o_1;\ldots;o_i}(I) \models \phi$.

2. If $\phi \notin C_{i-1}$, it must be because preserved(ϕ',C_{i-1},o') returns *false* for some $o' \in O$ and $\phi' \in C_{i-1}$ such that ϕ is obtained from ϕ' by conjoining some literals to it. Hence $\phi' \models \phi$.

   Since $\phi' \in C_{i-1}$ by the induction hypothesis $app_{o_1;\ldots;o_{i-1}}(I) \models \phi'$. Since $\phi' \models \phi$ also $app_{o_1;\ldots;o_{i-1}}(I) \models \phi$. Since the function call preserved(ϕ,C_i,o) returns *true* by Lemma 3.36 $app_{o_1;\ldots;o_i}(I) \models \phi$.

This finishes the induction proof. The iteration of the procedure stops when $C_i = C_{i-1}$, meaning that the claim of the theorem holds for arbitrarily long sequences $o_1; \ldots; o_m$ of operators. □

The algorithm does not find the strongest invariant for two reasons. First, only clauses until some fixed length are considered. Expressing the strongest invariant may require clauses that are longer. Second, the test performed by *preserved* tries to prove for one of the literals in the clause that it is true after an operator application. Consider the clause $a \vee b \vee c$ and the operator $\langle b \vee c, \neg a \rangle$. We cannot show for any literal that it is true after applying the operator but we know that either b or c is true. The test performed by *preserved* could be strengthened to handle cases like these, for example by using the techniques discussed in Section 4.2, but this would make the computation more expensive and eventually lead to intractability.

To make the algorithm run in polynomial time the satisfiability and logical consequence tests should be performed by algorithms that approximate these tests in polynomial time. The procedure asat(D, ϕ) is not suitable because it assumes that D is a set of literals, whereas for *preserved* the set C usually contain clauses with 2 or more literals. There are generalizations of the ideas behind asat(D, ϕ) to this more general case but we do not discuss the topic further.

3.5.1 Applications of invariants in planning by regression and satisfiability

Invariants can be used to speed up backward search with regression. Consider the blocks world with the goal *AonB*∧*BonC*. Regression with the operator that moves B onto C from the table yields *AonB* ∧ *Bclear* ∧ *Cclear* ∧ *BonT*. This formula does not correspond to an intended blocks world state because *AonB* is incompatible with *Bclear*, and indeed, ¬*AonB* ∨ ¬*Bclear* is an invariant for the blocks world. Any regression step that leads to a formula that is incompatible with the invariants can be ignored because that formula does not represent any state that is reachable from the initial state, and hence no plan extending the current incomplete plan can reach the goals.

Another application of invariants and the intermediate sets C_i produced by our invariant algorithm is improving the heuristics in Section 3.4. Using D_i^{max} for testing whether an operator precondition, for example $a \wedge b$, has distance i from the initial state, the distances of a and b are used separately. But even when it is possible to reach both a and b with i operator applications, it might still not be possible to reach them both simultaneously with i operator applications. For example, for $i = 1$ and an initial state in which both a and b are false, there might be no single operator that makes them both true, but two operators, each of which makes only one of them true. If $\neg a \vee \neg b \in C_i$, we know that after i operator applications one of a or b must still be false, and then we know that the operator in question is not applicable at time point i. Therefore the invariants and the sets C_i produced during the invariant computation can improve distance estimates.

3.6 Planning as satisfiability in the propositional logic

A very powerful approach to deterministic planning was introduced in 1992 by Kautz and Selman [1992; 1996]. In this approach the problem of reachability of a goal state from a given initial state is translated into propositional formulae $\phi_0, \phi_1, \phi_2, \ldots$ so that every valuation that satisfies formula ϕ_i corresponds to a plan of length i. Planning proceeds by first testing the satisfiability of ϕ_0. If ϕ_0 is unsatisfiable, continue with ϕ_1, ϕ_2, and so on, until a satisfiable formula ϕ_n is found. From a valuation that satisfies ϕ_n a plan of length n can be constructed.

3.6.1 Actions as propositional formulae

First we need a representation of actions in the propositional logic. We can view arbitrary propositional formulae as actions, or we can translate operators into formulae in the propositional logic. We discuss both of these possibilities.

Given a set of state variables $A = \{a_1, \ldots, a_n\}$, one could describe an action directly as a propositional formula ϕ over propositional variables $A \cup A'$ where $A' = \{a'_1, \ldots, a'_n\}$. Here the variables A represent the values of state variables in the state s in which an action is taken, and variables A' the values of state variables in a successor state s'.

A pair of valuations s and s' can be understood as a valuation of $A \cup A'$ (the state s assigns a value to variables A and s' to variables A'), and a transition from s to s' is possible if and only if $s, s' \models \phi$.

Example 3.38 The action that reverses the values of state variables a_1 and a_2 is described by $\phi = (a_1 \leftrightarrow \neg a'_1) \wedge (a_2 \leftrightarrow \neg a'_2)$. The following 4×4 incidence matrix represents this action.

	$a'_1a'_2$	$a'_1a'_2$	$a'_1a'_2$	$a'_1a'_2$
a_1a_2	00	01	10	11
00	0	0	0	1
01	0	0	1	0
10	0	1	0	0
11	1	0	0	0

The matrix can be equivalently represented as the following truth-table.

a_1	a_2	a'_1	a'_2	ϕ
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

■

Example 3.39 Let the set of state variables be $A = \{a_1, a_2, a_3\}$. The formula $(a_1 \leftrightarrow a'_2) \wedge (a_2 \leftrightarrow a'_3) \wedge (a_3 \leftrightarrow a'_1)$ represents the action that rotates the values of the state variables a_1, a_2 and a_3 one position right. The formula can be represented as the following adjacency matrix. The rows correspond to valuations of A and the columns to valuations of $A' = \{a'_1, a'_2, a'_3\}$.

	000	001	010	011	100	101	110	111
000	1	0	0	0	0	0	0	0
001	0	0	0	0	1	0	0	0
010	0	1	0	0	0	0	0	0
011	0	0	0	0	0	1	0	0
100	0	0	1	0	0	0	0	0
101	0	0	0	0	0	0	1	0
110	0	0	0	1	0	0	0	0
111	0	0	0	0	0	0	0	1

A more conventional way of depicting the valuations of this formula would be as a truth-table with one row for every valuation of $A \cup A'$, a total of 64 rows. ■

The action in Example 3.39 is deterministic. Not all actions represented by propositional formulae are deterministic. A sufficient (but not necessary) condition for determinism is that the formula is of the form $(\phi_1 \leftrightarrow a'_1) \wedge \cdots \wedge (\phi_n \leftrightarrow a'_n) \wedge \psi$ where $A = \{a_1, \ldots, a_n\}$ is the set of all state variables, ϕ_i are formulae over A (without occurrences of $A' = \{a'_1, \ldots, a'_n\}$). There are no restrictions on ψ. Formulae of this form uniquely determine the value of every state variable in the successor state in terms of the values in the predecessor state. Therefore they represent deterministic actions.

3.6.2 Translation of operators into propositional logic

We first give the simplest possible translation of deterministic planning into the propositional logic. In this translation every operator is separately translated into a formula, and the choice between the operators is represented as disjunction.

Definition 3.40 *The formula $\tau_A(o)$ which represents the operator $o = \langle c, e\rangle$ is defined by*

$$\tau_A(e) = \bigwedge_{a\in A}((\mathit{EPC}_a(e) \vee (a \wedge \neg \mathit{EPC}_{\neg a}(e))) \leftrightarrow a') \wedge \bigwedge_{a\in A} \neg(\mathit{EPC}_a(e) \wedge \mathit{EPC}_{\neg a}(e))$$
$$\tau_A(o) = c \wedge \tau_A(e).$$

The formula $\tau_A(e)$ expresses the value of a in the successor state in terms of the values of the state variables in the predecessor state and requires that executing e may not make any state variable simultaneously true and false. This is like in the definition of regression in Section 3.1.2. The formula $\tau_A(o)$ additionally requires that the operator's precondition is true.

Example 3.41 Consider operator $\langle a \vee b, (b \rhd a) \wedge (c \rhd \neg a) \wedge (a \rhd b)\rangle$. The corresponding propositional formula is

$$\begin{aligned}
&(a \vee b) \wedge((b \vee (a \wedge \neg c)) \leftrightarrow a')\\
&\qquad\wedge((a \vee (b \wedge \neg\bot)) \leftrightarrow b')\\
&\qquad\wedge((\bot \vee (c \wedge \neg\bot)) \leftrightarrow c')\\
&\qquad\wedge\neg(b \wedge c) \wedge \neg(a \wedge \bot) \wedge \neg(\bot \wedge \bot)\\
\equiv\ &(a \vee b) \wedge((b \vee (a \wedge \neg c)) \leftrightarrow a')\\
&\qquad\wedge((a \vee b) \leftrightarrow b')\\
&\qquad\wedge(c \leftrightarrow c')\\
&\qquad\wedge\neg(b \wedge c).
\end{aligned}$$

■

Lemma 3.42 *Let s and s' be states and o an operator. Let $v : A \cup A' \to \{0, 1\}$ be a valuation such that*

1. *for all $a \in A$, $v(a) = s(a)$, and*
2. *for all $a \in A$, $v(a') = s'(a)$.*

Then $v \models \tau_A(o)$ if and only if $s' = \mathit{app}_o(s)$.

Proof: Assume $v \models \tau_A(o)$. Hence $s \models c$ and $s \models \bigwedge_{a\in A} \neg(\mathit{EPC}_a(e) \wedge \mathit{EPC}_{\neg a}(e))$, and therefore $\mathit{app}_o(s)$ is defined. Consider any state variable $a \in A$. By Lemma 3.4 and the assumption $v \models (\mathit{EPC}_a(e) \vee (a \wedge \neg \mathit{EPC}_{\neg a}(e))) \leftrightarrow a'$, the value of every state variable in s' matches the definition of $\mathit{app}_o(s)$. Hence $s' = \mathit{app}_o(s)$.

Assume $s' = \mathit{app}_o(s)$. Since s' is defined, $v \models \tau_A(o)$ and $v \models \bigwedge_{a\in A} \neg(\mathit{EPC}_a(e) \wedge \mathit{EPC}_{\neg a}(e))$. By Lemma 3.4 $v \models \mathit{EPC}_a(e) \vee (a \wedge \neg \mathit{EPC}_{\neg a}(e))$ if and only if $s' \models a$. □

Definition 3.43 *Define $\mathcal{R}_1(A, A') = \tau_A(o_1) \vee \cdots \vee \tau_A(o_n)$.*

The valuations that satisfy this formula do not uniquely determine which operator was applied because for a given state more than one operator may produce the same successor state. However, in such cases it does not matter which operator is applied, and when constructing a plan from the valuation any of the operators may be chosen arbitrarily.

It has been noticed that extending $\mathcal{R}_1(A, A')$ by 2-literal invariants (see Section 3.5) reduces runtimes of algorithms that test satisfiability. Note that invariants do not affect the set of models of a formula representing planning: any satisfying valuation of the original formula also satisfies the invariants because the values of variables describing the values of state variables at any time point corresponds to a state that is reachable from the initial state, and hence this valuation also satisfies any invariant.

3.6.3 Finding plans by satisfiability algorithms

We show how plans can be found by first translating succinct transition systems $\langle A, I, O, G\rangle$ into propositional formulae, and then finding satisfying valuations by a satisfiability algorithm.

In Section 3.6.1 we showed how operators can be described by propositional formulae over sets A and A' of propositional variables, the set A describing the values of the state variables in the state in which the operator is applied, and the set A' describing the values of the state variables in the successor state of that state.

For a fixed plan length n, we use sets $A^0, \ldots, A^n$ of variables to represent the values of state variables at different time points, with variables A^i representing the values at time i. In other words, a valuation of these propositional variables represents a sequence $s_0, \ldots, s_n$ of states. If $a \in A$ is a state variable, then we use the propositional variable a^i for representing the value of a at time point i.

Then we construct a formula so that the state s_0 is determined by I, the state s_n is determined by G, and the changes of state variables between any two consecutive states corresponds to the application of an operator.

Definition 3.44 *Let $\langle A, I, O, G\rangle$ be a deterministic transition system. Define $\iota^0 = \bigwedge\{a^0 | a \in A, I(a) = 1\} \cup \{\neg a^0 | a \in A, I(a) = 0\}$ for the initial state and G^n as the formula G with every variable $a \in A$ replaced by a^n. Define*

$$\Phi_n^{seq} = \iota^0 \wedge \mathcal{R}_1(A^0, A^1) \wedge \mathcal{R}_1(A^1, A^2) \wedge \cdots \wedge \mathcal{R}_1(A^{n-1}, A^n) \wedge G^n$$

where $A^i = \{a^i | a \in A\}$ for all $i \in \{0, \ldots, n\}$.

A plan can be found by using the formulae Φ_i^{seq} as follows. We start with plan length $i = 0$, test the satisfiability of Φ_i^{seq}, and depending on the result, either construct a plan (if Φ_i^{seq} is satisfiable), or increase i by one and repeat the previous steps, until a plan is found.

If there are no plans, it has to be somehow decided when to stop increasing i. An upper bound on plan length is $2^{|A|} - 1$ where A is the set of state variables but this upper bound does not provide a practical termination condition for this procedure. Some work on more practical termination conditions are cited in Section 3.8.

The construction of a plan from a valuation v that satisfies Φ_i^{seq} is straightforward. The plan has exactly i operators, and this plan is known to be the shortest one because the formula Φ_{i-1}^{seq} had already been determined to be unsatisfiable. First construct the execution $s_0, \ldots, s_i$ of the plan from v as follows. For all $j \in \{0, \ldots, i\}$ and $a \in A$, $s_j(a) = v(a_j)$. The plan has the

form $o_1, \ldots, o_i$. Operator o_j for $j \in \{1, \ldots, i\}$ is identified by testing for all $o \in O$ whether $app_o(s_{j-1}) = s_j$. There may be several operators satisfying this condition, and any of them can be chosen.

Example 3.45 Let $A = \{a, b\}$. Let the state I satisfy $I \models a \wedge b$. Let $G = (a \wedge \neg b) \vee (\neg a \wedge b)$ and $o_1 = \langle \top, (a \rhd \neg a) \wedge (\neg a \rhd a) \rangle$ and $o_2 = \langle \top, (b \rhd \neg b) \wedge (\neg b \rhd b) \rangle$. The following formula is satisfiable if and only if $\langle A, I, \{o_1, o_2\}, G \rangle$ has a plan of length 3.

$$
\begin{array}{l}
(a^0 \wedge b^0) \\
\wedge(((a^0 \leftrightarrow a^1) \wedge (b^0 \leftrightarrow \neg b^1)) \vee ((a^0 \leftrightarrow \neg a^1) \wedge (b^0 \leftrightarrow b^1))) \\
\wedge(((a^1 \leftrightarrow a^2) \wedge (b^1 \leftrightarrow \neg b^2)) \vee ((a^1 \leftrightarrow \neg a^2) \wedge (b^1 \leftrightarrow b^2))) \\
\wedge(((a^2 \leftrightarrow a^3) \wedge (b^2 \leftrightarrow \neg b^3)) \vee ((a^2 \leftrightarrow \neg a^3) \wedge (b^2 \leftrightarrow b^3))) \\
\wedge((a^3 \wedge \neg b^3) \vee (\neg a^3 \wedge b^3))
\end{array}
$$

One of the valuations that satisfy the formula is the following.

	time i			
	0	1	2	3
a^i	1	0	0	0
b^i	1	1	0	1

This valuation corresponds to the plan that applies operator o_1 at time point 0, o_2 at time point 1, and o_2 at time point 2. There are also other satisfying valuations. The shortest plans have length 1 and respectively consist of the operators o_1 and o_2. ■

Example 3.46 Consider the following problem. There are two operators, one for rotating the values of bits abc one step right, and the other for inverting the values of all the bits. Consider reaching from the initial state 100 the goal state 001 with two actions. This is represented as the following formula.

$$
\begin{array}{l}
(a^0 \wedge \neg b^0 \wedge \neg c^0) \\
\wedge(((a^0 \leftrightarrow b^1) \wedge (b^0 \leftrightarrow c^1) \wedge (c^0 \leftrightarrow a^1)) \vee ((\neg a^0 \leftrightarrow a_1) \wedge (\neg b^0 \leftrightarrow b^1) \wedge (\neg c^0 \leftrightarrow c^1))) \\
\wedge(((a^1 \leftrightarrow b^2) \wedge (b^1 \leftrightarrow c^2) \wedge (c^1 \leftrightarrow a^2)) \vee ((\neg a^1 \leftrightarrow a^2) \wedge (\neg b^1 \leftrightarrow b^2) \wedge (\neg c^1 \leftrightarrow c^2))) \\
\wedge(\neg a^2 \wedge \neg b^2 \wedge c^2)
\end{array}
$$

Since the literals describing the initial and the goal state must be true, we can replace occurrences of these state variables in the subformulae for operators by $\top$ and $\bot$.

$$
\begin{array}{l}
(a^0 \wedge \neg b^0 \wedge \neg c^0) \\
\wedge(((\top \leftrightarrow b^1) \wedge (\bot \leftrightarrow c^1) \wedge (\bot \leftrightarrow a^1)) \vee ((\neg \top \leftrightarrow a_1) \wedge (\neg \bot \leftrightarrow b^1) \wedge (\neg \bot \leftrightarrow c^1))) \\
\wedge(((a^1 \leftrightarrow \bot) \wedge (b^1 \leftrightarrow \top) \wedge (c^1 \leftrightarrow \bot)) \vee ((\neg a^1 \leftrightarrow \bot) \wedge (\neg b^1 \leftrightarrow \bot) \wedge (\neg c^1 \leftrightarrow \top))) \\
\wedge(\neg a^2 \wedge \neg b^2 \wedge c^2)
\end{array}
$$

After simplifying we have the following.

$$
\begin{array}{l}
(a^0 \wedge \neg b^0 \wedge \neg c^0) \\
\wedge((b^1 \wedge \neg c^1 \wedge \neg a^1) \vee (\neg a_1 \wedge b^1 \wedge c^1) \\
\wedge((\neg a^1 \wedge b^1 \wedge \neg c^1) \vee (a^1 \wedge b^1 \wedge \neg c^1)) \\
\wedge(\neg a^2 \wedge \neg b^2 \wedge c^2)
\end{array}
$$

The only way of satisfying this formula is to make the first disjuncts of both disjunctions true, that is, b^1 must be true and a^1 and c^1 must be false. The resulting valuation corresponds to taking the rotation action twice.

Consider the same problem but now with the goal state 101.

$$
\begin{array}{l}
(a^0 \wedge \neg b^0 \wedge \neg c^0) \\
\wedge(((a^0 \leftrightarrow b^1) \wedge (b^0 \leftrightarrow c^1) \wedge (c^0 \leftrightarrow a^1)) \vee ((\neg a^0 \leftrightarrow a_1) \wedge (\neg b^0 \leftrightarrow b^1) \wedge (\neg c^0 \leftrightarrow c^1))) \\
\wedge(((a^1 \leftrightarrow b^2) \wedge (b^1 \leftrightarrow c^2) \wedge (c^1 \leftrightarrow a^2)) \vee ((\neg a^1 \leftrightarrow a^2) \wedge (\neg b^1 \leftrightarrow b^2) \wedge (\neg c^1 \leftrightarrow c^2))) \\
\wedge(a^2 \wedge \neg b^2 \wedge c^2)
\end{array}
$$

We simplify again and get the following formula.

$$
\begin{array}{l}
(a^0 \wedge \neg b^0 \wedge \neg c^0) \\
\wedge((b^1 \wedge \neg c^1 \wedge \neg a^1) \vee (\neg a_1 \wedge b^1 \wedge c^1)) \\
\wedge((\neg a^1 \wedge b^1 \wedge c^1) \vee (\neg a^1 \wedge b^1 \wedge \neg c^1)) \\
\wedge(a^2 \wedge \neg b^2 \wedge c^2)
\end{array}
$$

Now there are two possible plans, to rotate first and then invert the values, or first invert and then rotate. These respectively correspond to making the first disjunct of the first disjunction and the second disjunct of the second disjunction true, or the second and the first disjunct. ■

3.6.4 Parallel application of operators

For states s and sets T of operators we define $app_T(s)$ as the result of simultaneously applying all operators $o \in T$: the preconditions of all operators in T must be true in s and the state $app_T(s)$ is obtained from s by making the literals in $\bigcup_{\langle p,e\rangle \in T} [e]_s^{det}$ true. Analogously to sequential plans we can define $app_{T_1;T_2;\ldots;T_n}(s)$ as $app_{T_n}(\cdots app_{T_2}(app_{T_1}(s))\cdots)$.

Next we show how the translation of deterministic operators into the propositional logic in Section 3.6.2 can be extended to the simultaneous application of operators as in $app_T(s)$.

Consider the formula $\tau_A(o)$ representing one operator $o = \langle c, e\rangle$.

$$
c \wedge \bigwedge_{a \in A} ((EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))) \leftrightarrow a') \wedge \bigwedge_{a \in A} \neg(EPC_a(e) \wedge EPC_{\neg a}(e)).
$$

This formula can be rewritten to the following logically equivalent formula that separately says which state variables are changed by the operator and which state variables retain their values.

$$
\begin{array}{l}
c\wedge \\
\bigwedge_{a\in A}(EPC_a(e) \rightarrow a')\wedge \\
\bigwedge_{a\in A}(EPC_{\neg a}(e) \rightarrow \neg a')\wedge \\
\bigwedge_{a\in A}((a \wedge \neg a') \rightarrow EPC_{\neg a}(e))\wedge \\
\bigwedge_{a\in A}((\neg a \wedge a') \rightarrow EPC_a(e))
\end{array}
$$

We use this formulation of $\tau_A(o)$ as basis of obtaining encodings of planning that allow *several operators in parallel*. Every operator applied at a given time point causes its effects to be true and requires its precondition to be true. This is expressed by the first three conjuncts. The last two conjuncts say that, assuming the operator that is applied is the only one, certain state variables retain their value. These formulae have to be modified to accommodate the possibility of executing several operators in parallel.

We introduce propositional variables o for denoting the execution of operators $o \in O$.

Definition 3.47 *Let A be the set of state variables and O a set of operators. Let the formula $\tau_A(O)$ denote the conjunction of formulae*

$$(o \rightarrow c) \wedge$$
$$\bigwedge_{a \in A}(o \wedge \mathit{EPC}_a(e) \rightarrow a') \wedge$$
$$\bigwedge_{a \in A}(o \wedge \mathit{EPC}_{\neg a}(e) \rightarrow \neg a')$$

for all $\langle c, e\rangle \in O$ and

$$\bigwedge_{a \in A}((a \wedge \neg a') \rightarrow ((o_1 \wedge \mathit{EPC}_{\neg a}(e_1)) \vee \cdots \vee (o_n \wedge \mathit{EPC}_{\neg a}(e_n)) \wedge$$
$$\bigwedge_{a \in A}((\neg a \wedge a') \rightarrow ((o_1 \wedge \mathit{EPC}_a(e_1)) \vee \cdots \vee (o_n \wedge \mathit{EPC}_a(e_n)))$$

where $O = \{o_1, \ldots, o_n\}$ and $e_1, \ldots, e_n$ are the respective effects.

The difference to the definition of $\tau_A(o)$ in Section 3.6.2 is that above the formulae do not assume that there is only one operator explaining the changes that take place.

The formula $\tau_A(O)$ matches the definition of $\mathit{app}_T(s)$.

Lemma 3.48 *Let s and s' be states and O and $T \subseteq O$ sets of operators. Let $v : A \cup A' \cup O \rightarrow \{0, 1\}$ be a valuation such that*

1. *for all $o \in O$, $v(o) = 1$ iff $o \in T$,*
2. *for all $a \in A$, $v(a) = s(a)$, and*
3. *for all $a \in A$, $v(a') = s'(a)$.*

Then $v \models \tau_A(O)$ if and only if $s' = \mathit{app}_T(s)$.

Proof: For the proof from right to left we assume that $s' = \mathit{app}_T(s)$ and show that $v \models \tau_A(O)$.

For the formulae $o \rightarrow c$ consider any $o = \langle c, e\rangle \in O$. If $o \notin T$ then $v \not\models o$ and $v \models o \rightarrow c$. So assume $o \in T$. By assumption s is a state such that $\mathit{app}_T(s)$ is defined. Hence $s \models c$. Hence $v \models o \rightarrow c$.

For the formulae $o \wedge \mathit{EPC}_a(e) \rightarrow a'$ consider any $o = \langle c, e\rangle \in O$. If $o \notin T$ then $v \not\models o$ and $v \models o \wedge \mathit{EPC}_l(e) \rightarrow l$ for all literals l. So assume $o \in T$. Now $v \models o \wedge \mathit{EPC}_l(e) \rightarrow l$ because if $s \models \mathit{EPC}_l(e)$ then $l \in [e]_s^{det}$ by Lemma 3.3 and $s' \models l$. Proof for $o \wedge \mathit{EPC}_{\neg a}(e) \rightarrow \neg a'$ is analogous.

For the formulae $((a \wedge \neg a') \rightarrow ((o_1 \wedge \mathit{EPC}_{\neg a}(e_1)) \vee \cdots \vee (o_n \wedge \mathit{EPC}_{\neg a}(e_n))$ consider any $a \in A$. According to the definition of $s' = \mathit{app}_T(s)$, a can be true in s and false in s' only if $\neg a \in [o]_s^{det}$ for some $o \in T$. By Lemma 3.3 $\neg a \in [o]_s^{det}$ if and only if $s \models \mathit{EPC}_{\neg a}(o)$. So if the antecedent of $(a \wedge \neg a') \rightarrow ((o_1 \wedge \mathit{EPC}_{\neg a}(o_1)) \vee \cdots \vee (o_m \wedge \mathit{EPC}_{\neg a}(o_m)))$ is true, then one of the disjuncts of the consequent is true, where $O = \{o_1, \ldots, o_m\}$. The proof for the change from false to true is analogous.

For the proof from left to right we assume $v \models \tau_A(O)$ and show that $s' = \mathit{app}_T(s)$.

The precondition c of every $o \in T$ is true in s because $v \models o$ and $v \models o \rightarrow c$, and $s' \models [e]_s^{det}$ for every $o = \langle c, e\rangle \in T$ because $v \models o$ and $v \models o \wedge \mathit{EPC}_l(e) \rightarrow l$ for every literal l. This also means that $[T]_s^{det}$ is consistent and $\mathit{app}_T(s)$ is defined.

For state variables a not occurring in $[T]_s^{det}$ we have to show that $s(a) = s'(a)$. Since a does not occur in $[T]_s^{det}$, for every $o \in \{o_1, \ldots, o_m\} = O = \{\langle c_1, e_1\rangle, \ldots, \langle c_m, e_m\rangle\}$ either $o \notin T$ or both

$a \notin [e]_s^{det}$ and $\neg a \notin [e]_s^{det}$. Hence either $v \not\models o$ or (by Lemma 3.3) $v \models \neg(EPC_a(e)) \wedge \neg EPC_{\neg a}(e)$. This together with the assumptions that $v \models (a \wedge \neg a') \rightarrow ((o_1 \wedge EPC_{\neg a}(e_1)) \vee \cdots \vee (o_m \wedge EPC_{\neg a}(e_m)))$ and $v \models (\neg a \wedge a') \rightarrow ((o_1 \wedge EPC_a(o_1)) \vee \cdots \vee (o_m \wedge EPC_a(e_m)))$ implies $v \models (a \rightarrow a') \wedge (\neg a \rightarrow \neg a')$. Therefore every $a \in A$ not occurring in $[T]_s^{det}$ remains unchanged. Hence $s' = app_T(s)$. □

Example 3.49 Let $o_1 = \langle \neg LAMP1, LAMP1 \rangle$ and $o_2 = \langle \neg LAMP2, LAMP2 \rangle$. The application of none, one or both of these operators is described by the following formula.

$$
\begin{array}{l}
(\neg LAMP1 \wedge LAMP1') \rightarrow ((o_1 \wedge \top) \vee (o_2 \wedge \bot)) \\
(LAMP1 \wedge \neg LAMP1') \rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \bot)) \\
(\neg LAMP2 \wedge LAMP2') \rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \top)) \\
(LAMP2 \wedge \neg LAMP2') \rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \bot)) \\
o_1 \rightarrow LAMP1' \\
o_1 \rightarrow \neg LAMP1 \\
o_2 \rightarrow LAMP2' \\
o_2 \rightarrow \neg LAMP2
\end{array}
$$

■

3.6.5 Partially-ordered plans

In this section we consider a more general notion of plans in which several operators can be applied simultaneously. This kind of plans are formalized as sequences of sets of operators. In such a plan the operators are partially ordered because there is no ordering on the operators taking place at the same time point. This notion of plans is useful for two reasons.

First, consider a number of operators that affect and depend on disjoint state variables so that they can be applied in any order. If there are n such operators, there are $n!$ plans that are equivalent in the sense that each leads to the same state. When a satisfiability algorithm shows that there is no plan of length n consisting of these operators, it has to show that none of the $n!$ plans reaches the goals. This may be combinatorially very difficult if n is high.

Second, when several operators can be applied simultaneously, it is not necessary to represent all intermediate states of the corresponding sequential plans: partially-ordered plans require less time points than the corresponding sequential plans. This reduces the number of propositional variables that are needed for representing the planning problem, which may make testing the satisfiability of these formulae much more efficient.

In Section 3.6.4 we have shown how to represent the parallel application of operators in the propositional logic. However, this definition is too loose because it allows plans that cannot be executed.

Example 3.50 The operators $\langle a, \neg b \rangle$ and $\langle b, \neg a \rangle$ may be executed simultaneously resulting in a state satisfying $\neg a \wedge \neg b$, although this state is not reachable by the two operators sequentially. ■

A realistic way of interpreting parallelism in partially ordered plans is that any total ordering of the simultaneous operators is executable and results in the same state in all cases. This is the definition used in planning research so far.

Definition 3.51 (Step plans) *For a set of operators O and an initial state I,* a step plan for O and I *is a sequence $T = \langle T_0, \ldots, T_{l-1} \rangle$ of sets of operators for some $l \geq 0$ such that there is a sequence of states $s_0, \ldots, s_l$ (the execution of T) such that*

1. *$s_0 = I$,*
2. *for all $i \in \{0, \ldots, l-1\}$ and every total ordering $o_1, \ldots, o_n$ of T_i, $app_{o_1;\ldots;o_n}(s_i)$ is defined and equals s_{i+1}.*

Theorem 3.52 *Testing whether a sequence of sets of operators is a step plan is co-NP-hard.*

Proof: The proof is by reduction from the co-NP-complete validity problem TAUT. Let ϕ be any propositional formula. Let $A = \{a_1, \ldots, a_n\}$ be the set of propositional variables occurring in ϕ. Our set of state variables is A. Let $o_z = \langle \phi, \top \rangle$ and $O = \{\langle \top, a_1 \rangle, \ldots, \langle \top, a_n \rangle, o_z\}$. Let s and s' be states such that $s \not\models a$ and $s' \models a$ for all $a \in A$. We show that ϕ is a tautology if and only if $T = \langle O \rangle$ is a step plan for O and s.

Assume ϕ is a tautology. Now for any total ordering $o_0, \ldots, o_n$ of O the state $app_{o_0;\ldots;o_n}(s)$ is defined and equals s' because all preconditions are true in all states and the set of effects of all operators is A (the set is consistent and making the effects true in s yields s'.) Hence T is a step plan.

Assume T is a step plan. Let v be any valuation. We show that $v \models \phi$. Let $O_v = \{\langle \top, a \rangle | a \in A, v \models a\}$. The operators O can be ordered to $o_0, \ldots, o_n$ so that the operators $O_v = \{o_0, \ldots, o_k\}$ precede o_z and $O \backslash (O_v \cup \{o_z\})$ follow o_z. Since T is a step plan, $app_{o_0;\ldots;o_n}(s)$ is defined. Since also $app_{o_0;\ldots;o_k;o_z}(s)$ is defined, the precondition ϕ of o_z is true in $v = app_{o_0;\ldots;o_k}(s)$. Hence $v \models \phi$. Since this holds for any valuation v, ϕ is a tautology. □

To avoid intractability it is better to restrict to a class of step plans that are easy to recognize. One such class is based on the notion of *interference*.

Definition 3.53 (Affect) *Let A be a set of state variables and $o = \langle c, e \rangle$ and $o' = \langle c', e' \rangle$ operators over A. Then o* affects *o' if there is $a \in A$ such that*

1. *a is an atomic effect in e and a occurs in a formula in e' or it occurs negatively in c', or*
2. *$\neg a$ is an atomic effect in e and a occurs in a formula in e' or it occurs positively in c'.*

Definition 3.54 (Interference) *Operators o and o'* interfere *if o affects o' or o' affects o.*

Testing for interference of two operators is easy polynomial time computation. Non-interference not only guarantees that a set of operators is executable in any order, but it also guarantees that the result equals to applying all the operators simultaneously.

Lemma 3.55 *Let s be a state and T a set of operators so that $app_T(s)$ is defined and no two operators interfere. Then $app_T(s) = app_{o_1;\ldots;o_n}(s)$ for any total ordering $o_1, \ldots, o_n$ of T.*

Proof: Let $o_1, \ldots, o_n$ be any total ordering of T. We prove by induction on the length of a prefix of $o_1, \ldots, o_n$ the following statement for all $i \in \{0, \ldots, n-1\}$ by induction on i: $s \models a$ if and only if $app_{o_1;\ldots;o_i}(s) \models a$ for all state variables a occurring in an antecedent of a conditional effect or a precondition of operators $o_{i+1}, \ldots, o_n$.

Base case $i = 0$: Trivial.

Inductive case $i \geq 1$: By the induction hypothesis the antecedents of conditional effects of o_i have the same value in s and in $app_{o_1;\ldots;o_{i-1}}(s)$, from which follows $[o_i]^{det}_s = [o_i]^{det}_{app_{o_1;\ldots;o_{i-1}}(s)}$. Since o_i does not interfere with operators $o_{i+1}, \ldots, o_n$, no state variable occurring in $[o_i]^{det}_s$ occurs in an antecedent of a conditional effect or in the precondition of $o_{i+1}, \ldots, o_n$, that is, these state variables do not change. Since $[o_i]^{det}_s = [o_i]^{det}_{app_{o_1;\ldots;o_{i-1}}(s)}$ this also holds when o_i is applied in $app_{o_1;\ldots;o_{i-1}}(s)$. This completes the induction proof.

Since $app_T(s)$ is defined, the precondition of every $o \in T$ is true in s and $[o]^{det}_s$ is consistent. By the fact we established above, the precondition of every $o \in T$ is true also in $app_{o_1;\ldots;o_k}(s)$ and $[o]^{det}_{app_{o_1;\ldots;o_k}(s)}$ is consistent for any $\{o_1, \ldots, o_k\} \subseteq T \backslash \{o\}$. Hence any total ordering of the operators is executable. By the fact we established above, $[o]^{det}_s = [o]^{det}_{app_{o_1;\ldots;o_k}(s)}$ for every $\{o_1, \ldots, o_k\} \subseteq T \backslash \{o\}$. Hence every operator causes the same changes no matter what the total ordering is. Since $app_T(s)$ is defined, no operator in T undoes the effects of another operator. Hence the same state $s' = app_T(s)$ is reached in every case. □

For finding plans by using the translation of parallel actions from Section 3.6.4 it remains to encode the condition that no two parallel actions are allowed to interfere.

Definition 3.56 *Define*

$$\mathcal{R}_2(A, A', O) = \tau_A(O) \wedge \bigwedge \{\neg(o \wedge o') | \{o, o'\} \subseteq O, o \neq o', o \text{ and } o' \text{interfere}\}$$

Definition 3.57 *Let $\langle A, I, O, G\rangle$ be a deterministic succinct transition system. Define*

$$\Phi^{par}_n = \iota^0 \wedge \mathcal{R}_2(A^0, A^1, O^0) \wedge \mathcal{R}_2(A^1, A^2, O^1) \wedge \cdots \wedge \mathcal{R}_2(A^{n-1}, A^n, O^{n-1}) \wedge G^n$$

where $A^i = \{a^i | a \in A\}$ for all $i \in \{0, \ldots, n\}$ and $O^i = \{o^i | o \in O\}$ for all $i \in \{1, \ldots, n\}$ and $\iota^0 = \bigwedge\{a^0 | a \in A, I(a) = 1\} \cup \{\neg a^0 | a \in A, I(a) = 0\}$ and G^n is G with every $a \in A$ replaced by a^n.

If Φ^{par}_n is satisfiable and v is a valuation such that $v \models \Phi^{par}_n$, then define $T_i = \{o \in O | v \models o^i\}$ for every $i \in \{1, \ldots, n\}$. Then $\langle T_1, \ldots, T_n\rangle$ is a plan for the transition system, that is, $app_{T_1;\ldots;T_n}(I) \models G$.

It may be tempting to think that non-interference implies that the actions occurring in parallel in a plan could always be executed simultaneously in the real world. This however is not the case. For genuine temporal parallelism the formalization of problems as operators has to fulfill much stronger criteria than when sequential execution is assumed.

Example 3.58 Consider the operators

$$\begin{aligned}\text{transport-A-with-truck-1} &= \langle \text{AinFreiburg}, \text{AinStuttgart} \wedge \neg\text{AinFreiburg}\rangle \\ \text{transport-B-with-truck-1} &= \langle \text{BinFreiburg}, \text{BinKarlsruhe} \wedge \neg\text{BinFreiburg}\rangle\end{aligned}$$

which formalize the transportation of two objects with one vehicle. The operators do not interfere, and our notion of plans allows the simultaneous execution of these operators. However, these actions cannot really be simultaneous because the corresponding real world actions involve the same vehicle going to different destinations. ■

3.7 Computational complexity

In this section we discuss the computational complexity of the main decision problems related to deterministic planning.

The plan existence problem of deterministic planning is PSPACE-complete. The result was proved by Bylander [1994]. He proved the hardness part by giving a simulation of deterministic polynomial-space Turing machines, and the membership part by giving an algorithm that solves the problem in polynomial space. We later generalize his Turing machine simulation to alternating Turing machines to obtain an EXP-hardness proof for nondeterministic planning with full observability in Theorem 4.53.

Theorem 3.59 *The problem of testing the existence of a plan is PSPACE-hard.*

Proof: Let $\langle \Sigma, Q, \delta, q_0, g \rangle$ be any deterministic Turing machine with a polynomial space bound $p(x)$. Let σ be an input string of length n.

We construct a deterministic succinct transition system for simulating the Turing machine. The succinct transition system has a size that is polynomial in the size of the description of the Turing machine and the input string.

The set A of state variables in the succinct transition system consists of

1. $q \in Q$ for denoting the internal states of the TM,
2. s_i for every symbol $s \in \Sigma \cup \{|, \square\}$ and tape cell $i \in \{0, \ldots, p(n)\}$, and
3. h_i for the positions of the R/W head $i \in \{0, \ldots, p(n)+1\}$.

The initial state of the succinct transition system represents the initial configuration of the TM. The initial state I is as follows.

1. $I(q_0) = 1$
2. $I(q) = 0$ for all $q \in Q \backslash \{q_0\}$.
3. $I(s_i) = 1$ if and only if ith input symbol is $s \in \Sigma$, for all $i \in \{1, \ldots, n\}$.
4. $I(s_i) = 0$ for all $s \in \Sigma$ and $i \in \{0, n+1, n+2, \ldots, p(n)\}$.
5. $I(\square_i) = 1$ for all $i \in \{n+1, \ldots, p(n)\}$.
6. $I(\square_i) = 0$ for all $i \in \{0, \ldots, n\}$.
7. $I(|_0) = 1$
8. $I(|_i) = 0$ for all $n \in \{1, \ldots, p(n)\}$
9. $I(h_1) = 1$
10. $I(h_i) = 0$ for all $i \in \{0, 2, 3, 4, \ldots, p(n)+1\}$

The goal is the following formula.

$$G = \bigvee \{q \in Q | g(q) = \text{accept}\}$$

To define the operators, we first define effects corresponding to all possible transitions.

For all $\langle s, q\rangle \in (\Sigma \cup \{|, \square\}) \times Q$, $i \in \{0, \ldots, p(n)\}$ and $\langle s', q', m\rangle \in (\Sigma \cup \{|\}) \times Q \times \{L, N, R\}$ define the effect $\tau_{s,q,i}(s', q', m)$ as $\alpha \wedge \kappa \wedge \theta$ where the effects α, κ and θ are defined as follows.

The effect α describes what happens to the tape symbol under the R/W head. If $s = s'$ then $\alpha = \top$ as nothing on the tape changes. Otherwise, $\alpha = \neg s_i \wedge s'_i$ to denote that the new symbol in the ith tape cell is s' and not s.

The effect κ describes the change to the internal state of the TM. Again, either the state changes or does not, so $\kappa = \neg q \wedge q'$ if $q \neq q'$ and $\top$ otherwise. We define $\kappa = \neg q$ when $i = p(n)$ and $m = R$ so that when the space bound gets violated, no accepting state can be reached.

The effect θ describes the movement of the R/W head. Either there is movement to the left, no movement, or movement to the right. Hence

$$\theta = \begin{cases} \neg h_i \wedge h_{i-1} & \text{if } m = L \\ \top & \text{if } m = N \\ \neg h_i \wedge h_{i+1} & \text{if } m = R \end{cases}$$

By definition of TMs, movement at the left end of the tape is always to the right. Similarly, we have state variable for R/W head position $p(n) + 1$ and moving to that position is possible, but no transitions from that position are possible, as the space bound has been violated.

Now, these effects that represent possible transitions are used in the operators that simulate the Turing machine. Let $\langle s, q\rangle \in (\Sigma \cup \{|, \square\}) \times Q$, $i \in \{0, \ldots, p(n)\}$ and $\delta(s, q) = \{\langle s', q', m\rangle\}$. If $g(q) = \exists$, then define the operator

$$o_{s,q,i} = \langle h_i \wedge s_i \wedge q, \tau_{s,q,i}(s', q', m)\rangle.$$

We claim that the succinct transition system has a plan if and only if the Turing machine accepts without violating the space bound.

If the Turing machine violates the space bound, the state variable $h_{p(n)+1}$ becomes true and an accepting state cannot be reached because no further operator will be applicable.

So, because all deterministic Turing machines with a polynomial space bound can be in polynomial time translated into a planning problem, all decision problems in PSPACE are polynomial time many-one reducible to deterministic planning, and the plan existence problem is PSPACE-hard. □

Theorem 3.60 *The problem of testing the existence of a plan is in PSPACE.*

Proof: A recursive algorithm for testing m-step reachability between two states with $\log m$ memory consumption is given in Figure 3.4. The parameters of the algorithm are the set O of operators, the starting state s, the terminal state s', and m characterizing the maximum number 2^m of operators needed for reaching s' from s.

We show that when the algorithm is called with the number $n = |A|$ of state variables as the last argument, it consumes a polynomial amount of memory in n. The recursion depth is n. At the recursive calls memory is needed for storing the intermediate states s''. The memory needed for this is polynomial in n. Hence at any point of time the space consumption is $\mathcal{O}(m^2)$.

A succinct transition system $\langle A, I, O, G\rangle$ with $n = |A|$ state variables has a plan if and only if reach(O,I,s',n) returns *true* for some s' such that $s' \models G$. Iteration over all states s' can be performed in polynomial space and testing $s' \models G$ can be performed in polynomial time in the

```
  1:  procedure reach(O,s,s',m)
  2:  if m = 0 then                                   (* Plans of length 0 and 1 *)
  3:    if s = s' or there is o ∈ O such that s' = app_o(s) then return true
  4:    else return false
  5:  else
  6:    begin                                         (* Longer plans *)
  7:      for all states s'' do                       (* Iteration over intermediate states *)
  8:        if reach(O,s,s'',m − 1) and reach(O,s'',s',m − 1) then return true
;  9:      end
 10:      return false;
 11:    end
```

Figure 3.4: Algorithm for testing plan existence in polynomial space

size of G. Hence the whole memory consumption is polynomial. □

Part of the high complexity of planning is due to the fact that plans can be exponentially long. If a polynomial upper bound for plan length exists, testing the existence of plans is still intractable but much easier.

Theorem 3.61 *The problem of whether a plan having a length bounded by a given polynomial exists is NP-hard.*

Proof: We reduce the satisfiability problem of the classical propositional logic to the plan existence problem. The length of the plans, whenever they exist, is bounded by the number of propositional variables and hence is polynomial.

Let ϕ be a formula over the propositional variables in A. Let $N = \langle A, \{(a,0)|a \in A\}, O, \phi\rangle$ where $O = \{\langle \top, a\rangle | a \in A\}$ We show that N has a plan if and only if the formula ϕ is satisfiable.

Assume $\phi \in SAT$, that is, there is a valuation $v : A \to \{0,1\}$ such that $v \models \phi$. Now take the operators $\{\langle \top, a\rangle | v \models a, a \in A\}$ in any order: these operators form a plan that reach the state v that satisfies ϕ.

Assume N has a plan $o_1, \ldots, o_m$. The valuation $v = \{(a,1)|(\top, a) \in \{o_1, \ldots, o_m\}\} \cup \{(a,0)|a \in A, (\top, a) \notin \{o_1, \ldots, o_m\}\}$ of A is the terminal state of the plan and satisfies ϕ. □

Theorem 3.62 *The problem of whether a plan having a length bounded by a given polynomial exists is in NP.*

Proof: Let $p(m)$ be a polynomial. We give a nondeterministic algorithm that runs in polynomial time and determines whether a plan of length $p(m)$ exists.

Let $N = \langle A, I, O, G\rangle$ be a deterministic succinct transition system.

1. Nondeterministically guess a sequence of $l \leq p(m)$ operators $o_1, \ldots, o_l$ from the set O. Since l is bounded by the polynomial $p(m)$, the time consumption $\mathcal{O}(p(m))$ is polynomial in the size of N.

2. Compute $s = app_{o_l}(app_{o_{l-1}}(\cdots app_{o_2}(app_{o_1}(I))\cdots))$. This takes polynomial time in the size of the operators and the number of state variables.

3. Test $s \models G$. This takes polynomial time in the size of the operators and the number of state variables.

This nondeterministic algorithm correctly determines whether a plan of length at most $p(m)$ exists and it runs in nondeterministic polynomial time. Hence the problem is in NP. □

These theorems show the NP-completeness of the plan existence problem for polynomial-length plans.

3.8 Literature

Progression and regression were used early in planning research [Rosenschein, 1981]. Our definition of regression in Section 3.1.2 is related to the weakest precondition predicates for program synthesis [de Bakker and de Roever, 1972; Dijkstra, 1976]. Instead of using the general definition of regression we presented, earlier work on planning with regression and a definition of operators that includes disjunctive preconditions and conditional effects has avoided all disjunctivity by producing only goal formulae that are conjunctions of literals [Anderson *et al.*, 1998]. Essentially, these formulae are the disjuncts of $\mathit{regr}_o(\phi)$ in DNF, although the formulae $\mathit{regr}_o(\phi)$ are not generated. The search algorithm then produces a search tree with one branch for every disjunct of the DNF formula. In comparison to the general definition, this approach often leads to a much higher branching factor and an exponentially bigger search tree.

The use of algorithms for the satisfiability problem of the classical propositional logic in planning was pioneered by Kautz and Selman, originally as a way of testing satisfiability algorithms, and later shown to be more efficient than other planning algorithms at the time [Kautz and Selman, 1992; 1996]. In addition to Kautz and Selman [1996], parallel plans were used by Blum and Furst in their Graphplan planner [Blum and Furst, 1997]. Parallelism in this context serves the same purpose as partial-order reduction [Godefroid, 1991; Valmari, 1991], reducing the number of orderings of independent actions to consider. There are also other notions of parallel plans that may lead to much more efficient planning [Rintanen *et al.*, 2005]. Ernst et al. [1997] have considered translations of planning into the propositional that utilize the regular structure of sets of operators obtained from schematic operators. Planning by satisfiability has been extended to model-checking for testing whether a finite or infinite execution satisfying a given Linear Temporal Logic (LTL) formula exists [Biere *et al.*, 1999]. This approach to model-checking is called *bounded model-checking*.

It is trickier to use a satisfiability algorithm for showing that no plans of any length exist than for finding a plan of a given length. To show that no plans exist all plan lengths up to $2^n - 1$ have to be considered when there are n state variables. In typical planning applications n is often some hundreds or thousands, and generating and testing the satisfiability of all the required formulae is practically impossible. That no plans of a given length $n < 2^{|A|}$ do not exist does not directly imply anything about the existence of longer plans. Some other approaches for solving this problem based on satisfiability algorithms have been recently proposed [McMillan, 2003; Mneimneh and Sakallah, 2003].

The use of general-purpose heuristic search algorithms has recently got a lot of attention. The class of heuristics currently in the focus of interest was first proposed by McDermott [1999] and Bonet and Geffner [2001]. The distance estimates $\delta_I^{max}(\phi)$ and $\delta_I^{+}(\phi)$ in Section 3.4 are based on the ones proposed by Bonet and Geffner [2001]. Many other distance estimates similar to Bonet

and Geffner's exist [Haslum and Geffner, 2000; Hoffmann and Nebel, 2001; Nguyen *et al.*, 2002]. The $\delta_I^{rlx}(\phi)$ estimate generalizes ideas proposed by Hoffmann and Nebel [2001].

Other techniques for speeding up planning with heuristic state-space search include symmetry reduction [Starke, 1991; Emerson and Sistla, 1996] and partial-order reduction [Godefroid, 1991; Valmari, 1991; Alur *et al.*, 1997], both originally introduced outside planning in the context of reachability analysis and model-checking in computer-aided verification. Both of these techniques address the main problem in heuristic state-space search, high branching factor (number of applicable operators) and high number of states.

The algorithm for invariant computation was originally presented for simple operators without conditional effects [Rintanen, 1998]. The computation parallels the construction of planning graphs in the Graphplan algorithm [Blum and Furst, 1997], and it would seem to us that the notion of planning graph emerged when Blum and Furst noticed that the intermediate stages of invariant computation are useful for backward search algorithms: if a depth-bound of n is imposed on the search tree, then formulae obtained by m regression steps (suffixes of possible plans of length m) that do not satisfy clauses C_{n-m} cannot lead to a plan, and the search tree can be pruned. A different approach to find invariants has been proposed by Gerevini and Schubert [1998].

Some researchers extensively use Graphplan's planning graphs [Blum and Furst, 1997] for various purposes but we do not and have not discussed them in more detail for certain reasons. First, the graph character of planning graphs becomes inconvenient when preconditions of operators are arbitrary formulae and effects are conditional. As a result, the basic construction steps of planning graphs become unintuitive. Second, even when the operators have the simple form, the practically and theoretically important properties of planning graphs are not graph-theoretic. We can equivalently represent the contents of planning graphs as sequences of sets of literals and 2-literal clauses, as we have done in Section 3.5. In general it seems that the graph representation does not provide advantages over more conventional logic-based and set-based representations and is primarily useful for visualization purposes.

The algorithms presented in this section cannot in general be ordered in terms of efficiency. The general-purpose search algorithms with distance heuristics are often very effective in solving big problem instances with a sufficiently simple structure. This often entails better runtimes than in the SAT/CSP approach because of the high overheads with handling big formulae or constraint nets in the latter. Similarly, there are problems that are quickly solved by the SAT/CSP approach but on which heuristic state-space search fails.

There are few empirical studies on the behavior of different algorithms on planning problems in general or average. Bylander [1996] gives empirical results suggesting the existence of hard-easy pattern and a phase transition behavior similar to those found in other NP-hard problems like propositional satisfiability [Selman *et al.*, 1996]. Bylander also demonstrates that outside the phase transition region plans can be found by a simple hill-climbing algorithm or the inexistence of plans can be determined by using a simple syntactic test. Rintanen [2004b] complemented Bylander's work by analyzing the behavior of different types of planning algorithms on difficult problems inside the phase transition region, suggesting that current planners based on heuristic state space search are outperformed by satisfiability algorithms on difficult problems.

The PSPACE-completeness of the plan existence problem for deterministic planning is due to Bylander [1994]. The same result for another succinct representation of graphs had been established earlier by Lozano and Balcazar [1990].

Any computational problem that is NP-hard – not to mention PSPACE-hard – is considered too difficult to be solved in general. As planning even in the deterministic case is PSPACE-hard there

has been interest in finding restricted special cases in which efficient (polynomial-time) planning is always guaranteed. Syntactic restrictions have been investigated by several researchers [Bylander, 1994; Bäckström and Nebel, 1995] but the restrictions are so strict that very few interesting problems can be represented.

Schematic operators increase the conciseness of the representations of some problem instances exponentially and lift the worst-case complexity accordingly. For example, deterministic planning with schematic operators is EXPSPACE-complete [Erol *et al.*, 1995]. If function symbols are allowed, encoding arbitrary Turing machines becomes possible and the plan existence problem is undecidable [Erol *et al.*, 1995].

3.9 Exercises

3.1 Show that regression for goals G that are sets (conjunctions) of state variables and operators with preconditions p that are sets (conjunctions) of state variables and effects that consist of an add list a (a set of state variables that become true) and a delete list d (a set of state variables that become false) can equivalently be defined as $(G \backslash a) \cup p$ when $d \cap G = \emptyset$.

3.2 Show that the problem in Lemma 3.9 is in NP and therefore NP-complete.

3.3 Satisfiability testing in the propositional logic is tractable in some special cases, like for sets of clauses with at most 2 literals in each, and for Horn clauses, that is sets of clauses with at most one positive literal in each clause.

Can you identify special cases in which existence of an n-step plan can be determined in polynomial time (in n and the size of the problem instance), because the corresponding formula transformed to CNF is a set of 2-literal clauses or a set of Horn clauses?