ELSEVIER

Artificial Intelligence 89 (1997) 285-315

Artificial Intelligence

Automatically selecting and using primary effects in planning: theory and experiments

Eugene Fink[a,1], Qiang Yang[b,*]

[a] *School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

[b] *School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6*

Received December 1995; revised January 1996

Abstract

The use of primary effects of operators is an effective approach to improving the efficiency of planning. The characterization of "good" primary effects, however, has remained at an informal level and there have been no algorithms for selecting primary effects of operators.

We formalize the use of primary effects in planning and present a criterion for selecting useful primary effects, which guarantees efficiency and completeness. We analyze the efficiency of planning with primary effects and the quality of the resulting plans.

We then describe a learning algorithm that automatically selects primary effects and demonstrate, both analytically and empirically, that the use of this algorithm significantly reduces planning time and does not compromise completeness.

Keywords: Planning; Machine learning; Primary effects; Partial-order planning; PAC learning

1. Introduction

Planning with primary effects is an effective approach to reducing search. The underlying idea of this approach is to select *primary* effects among the effects of each planning operator and to use an operator only when we need to achieve one of its primary effects. A *primary-effect restricted* planner never inserts an operator into a plan for achieving its side effects.

* Corresponding author. E-mail: qyang@cs.sfu.ca. Supported by Natural Sciences and Engineering Research Council of Canada (NSERC) under grant number OGP0184883.

[1] E-mail: eugene@cs.cmu.edu. Supported by Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Defence Advanced Research Projects Agency (DARPA) under grant number F33615-93-1-1330.

PII S0004-3702(96)00020-3

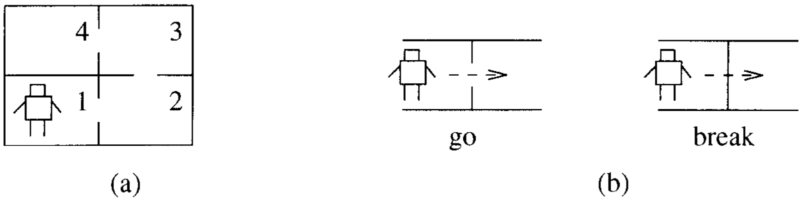

Fig. 1. The simple robot world in Example 1.2. (a) Map of the robot world. (b) Operators.

AI researchers have long recognized the advantages of using primary effects. For example, Fikes and Nilsson used primary effects to improve the quality of solutions generated by the STRIPS planner [5]. The authors of SIPE [20] distinguished between the main effects and side effects of operators and used this distinction to simplify the conflict resolution. The PRODIGY system [7, 19] allows the user to specify primary effects of operators in the form of control rules. The ABTWEAK planner [23] also enables the user to specify primary effects.

Despite the importance of primary effects, the characterization of "good" primary effects has remained at an informal level and the task of choosing primary effects has been left to the human user. If the user does not choose primary effects, then by default all effects are assumed to be primary. An important step in AI planning is to develop a system that determines good primary effects automatically.

We formalize the intuition underlying primary-effect restricted planners, develop a theory of planning with primary effects, and describe a learning algorithm that automatically selects appropriate primary effects of operators.

We begin by giving several examples of the use of primary effects (Section 1.1) and presenting an overview of our main results (Section 1.2).

1.1. Motivating examples

Example 1.1. Imagine a house with a fireplace in the living room. The fireplace may be used to warm and illuminate the room. If the occupant of the house has electric lamps in the living room, she may view the illumination as a *side* effect of using the fireplace. She does not use the fireplace when her *only* goal is illuminating the room, because electric lamps are easier to use and electricity is cheaper than wood. Warming the room, on the other hand, is a *primary* effect of using the fireplace.

Example 1.2. We describe a version of the robot world [5], which includes a robot and four rooms (see Fig. 1(a)). The robot can **go** between two rooms connected by a door and **break** through a wall to create a new doorway (see Fig. 1(b)). When the robot breaks a wall, it not only makes a new door but also moves to the room behind the broken wall.

Every change of the robot's location in this world can be accomplished by a series of **go** operators, without breaking through walls. We can view the location change as a side effect of the operator **break**: we use this operator only for making new doorways. When the only goal is to move the robot to some room, the planner uses only **go**.

This restriction reduces the number of alternatives to consider, which may lead to an exponential reduction in planning time (see Section 4).

Example 1.3. We consider a manufacturing domain with a number of machining operations, such as cutting, drilling, and polishing. We have to find plans for producing parts of different quality. The production of higher-quality parts requires more expensive machining operations.

A planner may use expensive operations for producing low-quality parts, which sometimes simplifies planning, but may lead to nonoptimal plans. For example, the planner may try to use a high-precision drilling operation even when normal drilling is sufficient. This use of high-precision drilling would result in a correct, but unnecessarily costly plan.

We may use primary effects to avoid an unnecessary application of high-quality operations. For example, we may view making a hole as a side effect of high-quality drilling, and the precise position of the hole as a primary effect. Then, the planner uses high-quality drilling only when high precision is important. In Section 7.4, we give a formal description of this manufacturing domain and present experiments on the use of primary effects to generate efficient machining plans.

1.2. Overview of the results

The use of primary effects reduces the branching factor of a planner's search space and may significantly improve efficiency; however, selecting appropriate primary effects is often a difficult task. An improper selection can cause three major problems in planning.

First, the use of primary effects may result in a loss of completeness: the planner may not be able to solve a solvable problem. For example, if the fireplace is the only source of light, but the illumination is not a primary effect of using the fireplace, then a primary-effect restricted planner will not solve the problem of illuminating the room.

Second, the use of primary effects may lead to generating costly plans, because primary effects bias the choice of planning operators. An improper bias can favor the use of expensive operators. For example, suppose that the operator **break** in the robot world of Example 1.2 is much more expensive than **go**. If the location change is a primary effect of **break**, then the planner may use this operator instead of **go** to change the robot's location, thus producing an unnecessarily costly plan.

Third, the use of primary effects may increase the depth of the planner's search. This increase sometimes results in an exponential increase in planning time, in spite of the reduction in branching factor of the search space.

The human user must select primary effects that ensure efficiency and completeness of planning, which is a hard problem. The user may not be sufficiently familiar with the domain to choose appropriate effects. The primary effects selected by an unexperienced user may compromise completeness or fail to improve efficiency. Even for domain experts, choosing good primary effects is sometimes difficult.

The purpose of our work is twofold. First, we formalize the reasons for a primary-effect restricted planner to be incomplete and nonoptimal, and describe methods for avoiding these dangers. We present a necessary and sufficient condition for completeness and optimality when planning with primary effects.

Second, we use this result to design an inductive learning algorithm that automatically selects primary effects. The learning algorithm receives as input a selection of primary effects proposed by the user or by a simple heuristic. The learner then tries to use this selection in solving example planning problems and revises it, as necessary, to ensure a high probability of completeness and of generating near-optimal solutions.

We demonstrate analytically that the primary effects selected by the algorithm (1) exponentially reduce planning time and (2) ensure a high probability of completeness and near optimality. We analyze the search reduction for backward-chaining planners that use best-first search to explore the space of possible plans. We estimate the planning time by the number of nodes in the planner's search space and the solution quality by the sum of the costs of operators in the solution plan.

We then experimentally confirm the analytical predictions using an advanced version of the TWEAK planner [3], called ABTWEAK [22]. We can readily generalize the techniques for learning primary effects to other backward-chaining planners, such as PRODIGY [19] and UCPOP [15].

Some researchers have used primary effects to improve the solution quality of depth-first search planners, by directing search to branches with low-cost operators. In particular, this approach was taken in the STRIPS planner and in a depth-first version of PRODIGY. We do not address this use of primary effects and concentrate on the problem of reducing planning time.

To summarize, our main goal is to improve the efficiency of backward-chaining planners by using primary effects. We develop a theory of planning with primary effects and design an algorithm for selecting primary effects, which ensures efficient planning and high solution quality.

1.3. Outline of the article

We first formalize the notion of planning with primary effects (Section 2) and describe conditions that ensure completeness and near optimality (Section 3). We then analyze the search space of planning with primary effects and derive conditions for search reduction (Section 4). In Section 5, we present an inductive learning algorithm that chooses primary effects automatically, by analyzing examples of planning problems. In Section 6, we determine the number of example problems required to ensure a high probability of completeness and near optimality when planning with primary effects. In Section 7, we give an experimental confirmation of the analysis, using a variety of planning domains. We discuss some extensions to the learning algorithm in Section 8. Finally, we conclude in Section 9, with a summary of the results.

2. Using primary effects in planning

We discuss the use of primary effects in planning and describe the Prim-TWEAK planner, a version of TWEAK [3] that uses primary effects. We first describe the

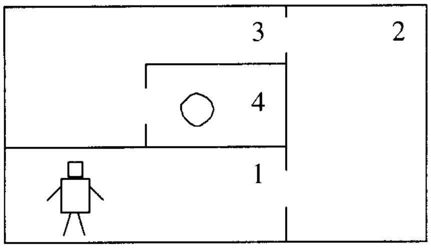

Fig. 2. The robot planning domain in Example 2.1.

representation of planning domains in the TWEAK system (Section 2.1). We then formalize the concept of planning with primary effects (Section 2.2) and present the Prim-TWEAK planning algorithm (Section 2.3).

2.1. Planning domains

A *planning domain* is defined by a library of operators. Each *operator* α in the domain is defined by a set of *effect literals*, denoted by *Eff*(α), and a set of *precondition literals*. If a literal l is an effect of α, we say that α *achieves* l. A literal is *achievable* if it can be achieved by at least one operator in the library.

We define a *planning problem* by an *initial state* and a *goal state*, where a *state* is a set of literals. A *total-order plan* is a finite sequence of operators. We view the initial state as the first operator of a plan, denoted by α_{init}. This operator has no preconditions and its effects are the initial-state literals. Similarly, the last operator of a plan, α_{goal}, represents the goal of the planning problem. The preconditions of α_{goal} are the goal literals and its effect set is empty. We say that a precondition l of some operator α in a total-order plan is *satisfied*, if there is an operator α_1 before α that achieves l and no operator between α_1 and α achieves $\neg l$. A total-order plan is *correct* if all preconditions of all operators are satisfied. In this case, we say that the plan *solves* the planning problem $(\alpha_{\text{init}}, \alpha_{\text{goal}})$.

A *partial-order plan* is a partially ordered set of operators. The partial order represents the order of execution of the operators in the plan. As in total-order plans, α_{init} precedes all other operators and α_{goal} is preceded by all other operators. A *linearization* of a partial-order plan is a total order of the operators consistent with the plan's partial order. We say that a statement about a partial-order plan is *possibly* true if it is true for *some* linearization of the plan. For example, an operator α_1 is possibly before α_2 if α_1 occurs before α_2 in one of the linearizations. We say that a statement is *necessarily* true if it is true for *all* linearizations of the plan. In particular, a partial-order plan is *necessarily correct* (or simply *correct*) if all its linearizations are correct.

The number of operators in a plan, *not* including the initial and goal states, is called the *size* of the plan.

Example 2.1. We describe an extended version of the robot world presented in Example 1.2. The new robot world contains a robot, a ball, and four rooms (see Fig. 2). The ball is initially in room 4. To describe the state of the domain, we have to specify

Table 1
The operators in the robot planning domain

Operator	Preconditions	Effects	Cost
go(x,y)	robot-in(x), door(x,y)	robot-in(y), $\neg$robot-in(x)	2
throw(x,y)	robot-in(x), ball-in(x), door(x,y)	ball-in(y), $\neg$ball-in(x)	2
carry-ball(x,y)	robot-in(x), ball-in(x), door(x,y)	robot-in(y), ball-in(y), $\neg$robot-in(x), $\neg$ball-in(x)	3
break(x,y)	robot-in(x)	robot-in(y), $\neg$robot-in(x), door(x,y)	4

the locations of the robot and the ball, and the pairs of rooms connected by doorways, which can be done with three predicates, robot-in(x), ball-in(x), and door(x,y). We obtain literals describing a specific state by substituting particular room numbers for x and y. For example, the literal robot-in(1) means that the robot is in room 1, ball-in(4) means that the ball is in room 4, and door(1,2) means that room 1 and room 2 are connected by a doorway.

The robot can go between two rooms connected by a door, throw the ball through a door into an adjacent room, carry the ball through a door, or break through a wall. We give a formal description of these operators in Table 1.

Consider a planning problem with the initial state as shown in Fig. 2 and the goal ball-in(3). The robot can achieve this goal by breaking through the wall of room 4 (**break**(1,4)) and then throwing the ball into room 3 (**throw**(4,3)).

We measure the quality of a plan by the total cost of operators in the plan. We associate some positive cost with each operator and define the cost of a plan as the sum of the costs of its operators.

Definition 2.2. The *cost* of a plan is the sum of the costs of its operators (not including the initial and goal states). An *optimal* solution to a planning problem is a plan with the lowest cost that solves the problem.

For example, suppose that we move the robot from room 1 to room 4 using the plan (**go**(1,2), **go**(2,3), **go**(3,4)). The cost of this plan is $2+2+2=6$. The solution is not optimal, since the same goal can be achieved by the operator **break**(1,4), with a cost of 4.

2.2. *Primary-effect justified plans*

If an operator α has several effects, we may choose certain important effects among them and use α only for achieving these important effects. The chosen important effects are called *primary* and denoted by *Prim-Eff*(α). The other effects are called *side* effects. For example, we may view the predicate door(x,y) as a primary effect of the operator **break**(x,y), and robot-in(y) as its side effect.

To formalize the use of primary effects in planning, we use the notion of *primary-effect justified plans* [22]. We begin by defining useful, or *justified*, effects of operators in a plan.

Definition 2.3. Let l be a primary effect of an operator α_1 in some plan. We say that l is a *justified primary effect* if there is an operator α with a precondition l such that

(1) α_1 is necessarily before α,
(2) there is no operator, necessarily after α_1 and before α, that achieves l or $\neg l$.

Informally, this condition means that the precondition l of α is achieved by α_1 in some linearization of the plan.

A plan is *primary-effect justified* if every operator has a justified primary effect. Intuitively, primary-effect justification means that no operator is used for the sake of its side effects. We presented a general discussion of justified plans in our previous research on improving the plan quality [9].

For example, consider the robot domain of Example 2.1 and suppose that the predicate `robot-in`(x, y) is a primary effect of **go**(x, y) and a side effect of **break**(x, y). The plan (**go**$(1,2)$, **go**$(2,3)$) is primary-effect justified for achieving the goal `robot-in(3)`. On the other hand, the plan **break**$(1,3)$ is not primary-effect justified, because changing the robot's position is *not* a primary effect of **break**.

2.3. Primary-effect restricted planners

Given a planning problem with an initial state α_{init} and a goal α_{goal}, a partial-order backward-chaining planner starts with the two-operator plan $(\alpha_{\text{init}}, \alpha_{\text{goal}})$ and modifies it until a solution is found. A plan may be modified by inserting a new operator or imposing an ordering constraint. When inserting a new operator to achieve some subgoal literal l, an *unrestricted* planner may use any operator that achieves l. A *primary-effect restricted* planner always uses an operator that achieves l as a primary effect.

In Fig. 3, we present a primary-effect restricted version of the TWEAK planning algorithm, called Prim-TWEAK. We have used this algorithm in the experimental studies of planning with primary effects. For simplicity, we do not show in Fig. 3 how Prim-TWEAK treats the variables in the preconditions and effects of operators. The implementation of Prim-TWEAK uses Chapman's codesignation technique to generate plans with variables.

The search for unsatisfied preconditions at Steps 1 and 2 of Prim-TWEAK is based on Chapman's modal truth criterion [3]. This search is performed by a very fast algorithm. Prim-TWEAK may achieve an unsatisfied precondition l of α in two different ways: by ordering the operators in the plan (see Step 3(A)) or by inserting a new operator α_1 that achieves l (see Step 3(B)). The algorithm then makes sure, at Step 6, that there is no operator with an effect $\neg l$ between α_1 and α.

Step 3(B) is the only place where the Prim-TWEAK planner is restricted to the use of primary effects. If *all* effects of all operators are selected as primary, then the planner may use any operator that achieves the unsatisfied precondition l, in which case Prim-TWEAK is identical to the original TWEAK.

```
Prim-tweak($\Pi$)
1. If all operator preconditions in the plan $\Pi$ are necessarily satisfied, then return $\Pi$.
2. Choose some operator $\alpha$ with a possibly unsatisfied precondition $l$.
3. Let $\alpha_1$ be either
   (A) an operator of the plan $\Pi$ possibly before $\alpha$ with an effect $l$, or
   (B) an operator in the library with a primary effect $l$.
   Branching point: each choice of $\alpha_1$ corresponds to a different branch in the search
   space.
4. If $\alpha_1$ is an operator from the library, add it to $\Pi$ (without ordering constraints).
5. Order $\alpha_1$ before $\alpha$.
6. For every $\alpha_2$ with an effect $\neg l$ that is possibly after $\alpha_1$ and before $\alpha$,
   (A) order $\alpha_2$ before $\alpha_1$, or
   (B) order $\alpha_2$ after $\alpha$, or
   (C) choose an operator with an effect $l$ in the plan $\Pi$ and order it between $\alpha_2$
       and $\alpha$.
   Branching point: different orderings correspond to different branches of the search
   space.
7. Recursively call Prim-TWEAK on the resulting plan.
```

Fig. 3. Primary-effect restricted version of the TWEAK planner.

Branching points in the description of the algorithm indicate places where the planner may consider different modifications of the current plan, thus creating several different branches of the search space. To ensure completeness, the planner must consider *all* alternatives: it must try all possible operators at Step 3 and all possible orderings at Step 6.

To solve a planning problem, we call Prim-TWEAK with the initial plan $(\alpha_{\text{init}}, \alpha_{\text{goal}})$. The planner recursively adds operators and ordering constraints to this plan until it generates a solution. If the planning problem does not have a solution, then Prim-TWEAK either terminates without any output or runs forever.

Example 2.4. We again consider the robot domain described in Example 2.1 (see Table 1), with the following selection of primary effects:

go(x,y)	{`robot-in`(y)}
throw(x,y)	{`ball-in`(y)}
carry-ball(x,y)	{`ball-in`(y)}
break(x,y)	{`door`(x,y)}

Suppose that the initial state is as shown in Fig. 2 and the robot has to move to room 3. The robot may achieve this goal by breaking through the wall between rooms 1 and 3. Prim-TWEAK will not consider this plan, however, because changing the robot's location is not a primary effect of breaking through a wall. Instead, Prim-TWEAK generates the solution (**go**(1, 2), **go**(2, 3)).

3. Completeness and cost increase

We now discuss the possible loss of completeness and optimality due to the use of primary effects and ways of avoiding this danger (Sections 3.1 and 3.2). We derive a necessary and sufficient condition for completeness and optimality (Section 3.3). We use this condition in Section 5 to design an algorithm for learning primary effects.

3.1. Completeness

A planner is *complete* if it can find a solution plan for every solvable problem. The unrestricted TWEAK planner is complete [3].

The use of primary effects, however, may compromise completeness of TWEAK. For example, consider the robot domain with the primary effects given in Example 2.4. Suppose that the initial state is as shown in Fig. 2 and the robot must move out of room 1. The formal description of this goal is {¬`robot-in`(1)}. The robot may achieve this goal by going into room 2 or breaking into room 3. A primary-effect restricted planner, however, will fail to solve this problem, because ¬`robot-in` is not a primary effect of *any* operator. To preserve completeness, we have to select additional primary effects:

go(x, y)	{`robot-in`(y), ¬`robot-in`(x)}
throw(x, y)	{`ball-in`(y), ¬`ball-in`(x)}
carry-ball(x, y)	{`ball-in`(y)}
break(x, y)	{`door`(x, y)}

Planning with primary effects is complete if (1) every solvable problem has a primary-effect justified solution and (2) the primary-effect restricted planner can solve every problem that has a primary-effect justified solution. The Prim-TWEAK planner satisfies the second condition.

Theorem 3.1. *If* Prim-TWEAK *searches the space of plans in the best-first order of plan costs, then it will solve every problem that has a primary-effect justified solution.*

The proof of this theorem is similar to the completeness proof for the unrestricted TWEAK planner, presented elsewhere [3,21].

Thus, to guarantee completeness of Prim-TWEAK, we have to ensure that every solvable problem has a primary-effect justified solution. We will characterize selections of primary effects with this property in Section 3.3 and describe an algorithm for generating such selections in Section 5.2.

3.2. Solution quality and the cost increase

The unrestricted TWEAK planner is able to find an optimal solution to every problem, but the use of primary effects may result in generating nonoptimal plans. For example, consider the last selection of primary effects in the robot domain and suppose that initially the robot is in room 4. The optimal plan for moving from room 4 to room 1 is

break$(4,1)$, the cost of which is 4. A primary-effect restricted planner, however, will generate the plan $(\mathbf{go}(4,3), \mathbf{go}(3,2), \mathbf{go}(2,1))$, with a cost of 6.

The ratio of the cost of a cheapest primary-effect justified plan to the cost of an optimal plan is called the *cost increase* for a planning problem. For the problem of moving the robot from room 4 to room 1, the cost increase is $6/4 = 1.5$.

3.3. Condition for completeness

We now derive a necessary and sufficient condition for completeness and limited cost increase when planning with primary effects. We use this condition in designing an algorithm that selects primary effects.

Consider a one-operator plan $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$, where the initial state α_{init} satisfies the preconditions of the operator α and the goal $\alpha_{\text{side-eff-goal}}$ is to achieve all the side effects of α while preserving all the literals of the initial state that are not changed by α. The goal does *not* include the primary effects of α and does *not* require preserving the literals of the initial state that are changed by the primary effects.

The side effects of α may be described in terms of the set difference as $(\mathit{Eff}(\alpha) - \mathit{Prim\text{-}Eff}(\alpha))$ and the literals of the initial state α_{init} not changed by α are $(\alpha_{\text{init}} - \mathit{Eff}(\alpha))$. Thus, the goal $\alpha_{\text{side-eff-goal}}$ is defined as follows:

$$\alpha_{\text{side-eff-goal}} = (\mathit{Eff}(\alpha) - \mathit{Prim\text{-}Eff}(\alpha)) \cup (\alpha_{\text{init}} - \mathit{Eff}(\alpha)).$$

A *replacing plan* for $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ is a primary-effect justified plan that achieves the goal $\alpha_{\text{side-eff-goal}}$ from the same initial state α_{init}. In other words, a replacing plan must (1) achieve all side effects of α and (2) leave all other literals of α_{init} unchanged.

For example, suppose that initially the robot is in room 4 and consider the one-operator plan **break**$(4,1)$. The side effects of this operator are `robot-in(1)` and `¬robot-in(4)`. The plan $(\mathbf{go}(4,3), \mathbf{go}(3,2), \mathbf{go}(2,1))$ is a replacing plan, since it is a primary-effect justified plan that achieves both side effects of the operator **break**$(4,1)$ and does not change any other literals.

The *replacing cost increase* C_α of the plan $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ is the ratio of the cost of an optimal replacing plan Π to the cost of α; that is, $C_\alpha = \mathit{cost}(\Pi)/\mathit{cost}(\alpha)$. For example, the cost of the operator **break**$(4,1)$ is 4 and the cost of the replacing plan $(\mathbf{go}(4,3), \mathbf{go}(3,2), \mathbf{go}(2,1))$ is 6; thus, the replacing cost increase is $6/4 = 1.5$.

Suppose that we can generate a primary-effect justified replacing plan for every operator and every initial state, and the replacing cost increases have a finite upper bound, $C_{\max}$. Then, every problem has a primary-effect justified solution with a bounded cost increase.

Theorem 3.2. *Completeness: Primary-effect restricted planning is complete if and only if, for every operator α and every initial state α_{init} that satisfies the preconditions of α, the one-operator plan $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ has a replacing plan.*

Cost increase: If the replacing cost increases of such one-operator plans have a finite maximum, $C_{\max}$, then the cost increases for all solution plans of all sizes are at most $\max(1, C_{\max})$.

Proof. Intuitively, given an unrestricted plan that solves some problem, we may replace all its operators by corresponding replacing plans. The resulting plan is a primary-effect justified solution to the problem and its cost is at most $C_{\max}$ times larger than the cost of the unrestricted solution.

We formalize this intuition to prove the second part of the theorem; the proof of the first part is similar. We consider an arbitrary problem, with an optimal total-order solution $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, and show how to construct a primary-effect justified solution.

If α_n is not primary-effect justified, we substitute an optimal replacing plan for α_n. If α_{n-1} is not primary-effect justified in the resulting plan, we substitute an optimal replacing plan for α_{n-1}. We repeat this operation for all other operators, considering them in the reverse order, from α_{n-2} to α_1.

When we replace an operator α_i, all operators *after* it remain primary-effect justified. We therefore obtain a primary-effect justified solution. For every replaced operator α_i, the cost of the replacing plan is at most $C_{\max} \cdot cost(\alpha_i)$, which implies that the total cost of the primary-effect justified solution is at most $\max(1, C_{\max}) \cdot cost(\alpha_1, \alpha_2, \ldots, \alpha_n)$. □

According to Theorem 3.2, we have to consider only one-operator plans when selecting primary effects of operators. To ensure completeness, we have to demonstrate that we can find a replacing plan for every operator and every initial state satisfying the preconditions of this operator.

Finding replacing plans for all one-operator plans may require an intractable search. We may, however, guarantee a *high probability* of completeness by finding replacing plans for a small random selection of one-operator plans. We use this probabilistic approach to design a learning algorithm that selects primary effects of operators (Section 5.2).

The condition of Theorem 3.2 is necessary for completeness when planning goals may include any collection of literals. If we encounter only a subclass of possible goals, then we may be able to select fewer primary effects without compromising completeness. We discuss some methods for selecting primary effects for a subclass of goals in Section 8.3.

3.4. Avoiding redundant primary effects

We are interested in finding a *minimal* selection of primary effects that ensures completeness and a small cost increase. A primary effect is *redundant* if we can demote it to a side effect without compromising completeness or increasing solution costs.

For example, suppose that the operator **carry-ball**(x, y) in the robot domain has two primary effects, `robot-in(`y`)` and `ball-in(`y`)`. Then, `robot-in` is a redundant primary effect. Demoting `robot-in` to a side effect of **carry-ball** does not compromise completeness and does not increase the costs of primary-effect justified solutions, because we may use the cheaper operator **go** for changing the robot's location.

Redundant primary effects increase the branching factor of search without reducing search depth or improving solution quality. Avoiding redundancy is one of the main goals in designing an algorithm for learning primary effects.

3.5. Summary of terminology

The following list summarizes the terminology introduced in Sections 2 and 3:

- *Unrestricted planner.* A planner that does not distinguish between primary and side effects of operators.
- *Primary-effect restricted planner.* A planner that inserts an operator into a plan only for achieving a primary effect of the operator.
- *Primary-effect justified plan.* A plan in which every operator has a *justified* primary effect, which is a primary effect necessary for achieving a precondition of some other operator.
- *Cost of a plan.* The sum of the costs of all operators in the plan.
- *Cost increase.* The ratio of the cost of a cheapest primary-effect justified solution to the cost of an optimal solution.
- *Replacing cost increase.* The ratio of the cost of a cheapest primary-effect justified plan that achieves the side effects of an operator to the cost of the operator.
- *Redundant primary effect.* A primary effect that can be demoted to a side effect without affecting completeness or increasing solution costs.

4. Analysis of the search reduction

We analyzed the search space of backward-chaining planners and identified the factors that determine the efficiency of planning with primary effects [11]. The analysis is an approximation based on several simplifying assumptions about properties of planning domains.

We present here an analytical comparison of planning efficiency with and without primary effects. The purpose of the comparison is to demonstrate that the use of primary effects may significantly reduce planning time and that the reduction is exponential in the size of the solution plan. In Section 7 we give experimental confirmation of this analytical prediction.

When searching for a solution to a planning problem, a planner expands a search space, whose nodes are incomplete plans. The planner creates a node by inserting a new operator into a plan or by imposing a constraint on the order of executing old operators. We assume that the planner uses best-first search and that all operators have the same cost.

Suppose that we use a planning algorithm to solve some problem. We denote the average branching factor of the unrestricted planner by B_u and its search depth by D_u. Then, the total number of nodes expanded by the best-first search is approximately

$$1 + B_u + B_u^2 + \cdots + B_u^{D_u} = \frac{B_u^{D_u+1} - 1}{B_u - 1}. \tag{1}$$

Similarly, we denote the average branching factor of the primary-effect restricted planner by B_p and its search depth for the given planning problem by D_p. The number of nodes expanded by the primary-effect restricted planner when solving the problem is approximately

$$\frac{B_p^{D_p+1}-1}{B_p-1}.$$

Let R denote the ratio of the planning times with and without primary effects. We assume that planning time is proportional to the number of nodes in the search space and estimate R by the ratio of the search-space sizes:

$$R=\frac{(B_p^{D_p+1}-1)/(B_p-1)}{(B_u^{D_u+1}-1)/(B_u-1)}=\mathrm{O}\left(\frac{B_p^{D_p}}{B_u^{D_u}}\right). \tag{2}$$

We next describe a relationship between the search depth of an unrestricted planner, D_p, and the search depth of a primary-effect restricted planner, D_u. We note that the search depth of most planners is proportional to the size of the solution plan. In particular, we demonstrated this proportion, both analytically and experimentally, for the TWEAK planner [14].

We give here an informal justification for the linear relationship between search depth and solution size. The TWEAK planner has to achieve every precondition of every operator in the plan, either by inserting a new operator or by adding an ordering constraint. The planner may have to achieve a precondition more than once, if newly inserted operators negate some preconditions. We assume that the average number of times the planner re-achieves each precondition is the same for all problems. If the domain satisfies this assumption, then the search depth is proportional to the total number of the preconditions of operators in the solution plan which, in turn, is proportional to the size of the plan.

We denote the cost increase of the given problem by C, which means that the solution generated by the primary-effect restricted planner is C times longer than the solution of the unrestricted planner. Since the search depth is proportional to the size of the solution plan, we conclude that the search depth of planning with primary effects is C times larger than that without primary effects:

$$D_p=C\cdot D_u. \tag{3}$$

We substitute this estimate into Eq. (2) and obtain the following expression for the planning-time ratio:

$$R=\mathrm{O}\left(\frac{B_p^{C\cdot D_u}}{B_u^{D_u}}\right)=\mathrm{O}\left(\left(\frac{B_p^C}{B_u}\right)^{D_u}\right). \tag{4}$$

Let us denote the base of the power in Eq. (4) by r:

$$r=\frac{B_p^C}{B_u}. \tag{5}$$

Then, we may rewrite Eq. (4) as $R=\mathrm{O}(r^{D_u})$. If $r<1$, then the saving in planning time grows exponentially with the search depth, which implies that the saving is exponential in the solution size. The smaller the value of r, the greater the saving.

```
Initial-Choice
1. For every operator $\alpha$ in the planning domain,
    (A) ask the user to specify primary effects of $\alpha$,
    (B) make all user-selected effects primary.
2. For every achievable literal $l$ that is not chosen by the user as a primary effect,
    (A) find the cheapest operator $\alpha_{\text{cheap}}$ that achieves $l$,
    (B) make $l$ a primary effect of $\alpha_{\text{cheap}}$.
```

Fig. 4. Generating an initial selection of primary effects.

Observe that the use of primary effects improves the efficiency of planning only if $r < 1$, which means that $B_p^C/B_u < 1$. Solving this inequality with respect to the cost increase C, we conclude that primary effects improve performance when

$$C < \frac{\log B_u}{\log B_p}. \tag{6}$$

We can draw some other conclusions from the expression for r (Eq. (5)). First, if we reduce the number of primary effects, the branching factor of primary-effect restricted planning, B_p, becomes smaller, whereas the cost increase, C, becomes larger. The value of r decreases with B_p; however, r increases with C. To minimize r, we have to strike the right balance between B_p and C [11].

Second, we conclude from Eq. (5) that we should always avoid redundant primary effects. Recall that a primary effect is redundant if we can make it a side effect without increasing solution costs. Demoting a redundant primary effect to a side effect decreases B_u without increasing C and, hence, improves the efficiency.

5. Automatically selecting primary effects

We describe an algorithm that automatically selects primary effects of operators. The selected primary effects improve the efficiency of the planner, preserve completeness with high probability, and guarantee that the planner almost always finds near-optimal solutions.

The algorithm consists of two parts. The Initial-Choice procedure (see Fig. 4) generates an initial selection of primary effects, using a simple heuristic (Section 5.1). Then, the Prim-Learner procedure (see Fig. 5) revises the initial selection to ensure a high probability of completeness and near optimality (Sections 5.2–5.5).

5.1. Initial choice of primary effects

We present the algorithm for generating an initial selection of primary effects in Fig. 4. The algorithm first asks the user to specify primary effects of operators (see Step 1). If the user selects too few (or no) primary effects, the initial-choice algorithm and the learning algorithm will add missing primary effects automatically.

```
**Prim-Learner**($C_u, \varepsilon_u, \delta_u$)
($C_u$ is the maximal allowed cost increase; $\varepsilon_u$ and $\delta_u$ determine the completeness probability.)
  1. Compute $m$ from given $\varepsilon_u$ and $\delta_u$ (see Section 6).
  2. For every operator $\alpha$, repeat $m$ times:
     (A) Generate a random state $\alpha_{\text{init}}$ that satisfies the preconditions of $\alpha$ (see Section 5.4).
     (B) Generate the goal, $\alpha_{\text{side-eff-goal}} = (\alpha_{\text{init}} - \mathit{Eff}(\alpha)) \cup (\mathit{Eff}(\alpha) - \mathit{Prim\text{-}Eff}(\alpha))$.
     (C) Call the Prim-TWEAK planner to find a primary-effect justified plan that
         achieves the goal $\alpha_{\text{side-eff-goal}}$ from the initial state $\alpha_{\text{init}}$, with a cost at most
         $C \cdot \mathit{cost}(\alpha)$.
     (D) If no such plan is found, promote an arbitrary side effect of $\alpha$ to a primary effect.
```

Fig. 5. Learning additional primary effects.

The initial-choice algorithm makes sure that every achievable literal is a primary effect of some operator (see Step 2). If some literal were not selected as a primary effect of any operator, a primary-effect restricted planner would not be able to achieve it, which would compromise completeness. For every literal l that is *not* a primary effect of any operator in the user's selection, the algorithm finds a cheapest operator α_{cheap} that achieves l and makes l a primary effect of α_{cheap}.

Example 5.1. Suppose that we apply the initial-choice algorithm to the robot domain (see Fig. 2) and the user has selected `ball-in` as a primary effect of **carry-ball**. The algorithm finds cheapest operators achieving the remaining literals, `robot-in`, ¬`robot-in`, ¬`ball-in`, and `door`. The cheapest operator that achieves the literals `robot-in` and ¬`robot-in` is **go**, the cheapest operator for ¬`ball-in` is **throw**, and the cheapest operator for `door` is **break**. Thus, the algorithm selects the following primary effects:

go(x, y)	$\{\texttt{robot-in}(y), \neg\texttt{robot-in}(x)\}$
throw(x, y)	$\{\neg\texttt{ball-in}(x)\}$
carry-ball(x, y)	$\{\texttt{ball-in}(y)\}$
break(x, y)	$\{\texttt{door}(x, y)\}$

5.2. *Learning additional primary effects*

We now describe a learning algorithm that selects additional primary effects to ensure a high probability of completeness and to limit cost increase. We present the algorithm in Fig. 5.

The input of the algorithm includes three user-specified parameters, C_u, ε_u, and δ_u. The first value, C_u, is the maximal cost increase allowed by the user. The other two values are the probability requirements for the success of the inductive learning. They are the standard parameters of the probably approximately correct (PAC) learning [18]. We now briefly describe the meaning of these two values. In Section 6, we present the detailed explanation of their use.

The ε_u value determines the required probability of completeness and limited cost increase. The learner must ensure that Prim-TWEAK solves a randomly selected solvable problem, within the cost increase C_u, with probability at least $(1-\varepsilon_u)$. In other words, at most ε_u of all solvable problems may become unsolvable due to the use of primary effects.

The δ_u value determines the probability of success of the inductive learning. The probability that at most ε_u of all solvable problems may become unsolvable must be at least $(1-\delta_u)$. To summarize, the learner ensures with probability at least $(1-\delta_u)$ that the Prim-TWEAK planner solves $(1-\varepsilon_u)$ of all solvable problems within the cost increase C_u.

The learning algorithm is based on the completeness condition presented in Theorem 3.2. The algorithm verifies the completeness of planning with primary effects by generating one-operator plans $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ and finding corresponding replacing plans. In each of these one-operator plans, the goal $\alpha_{\text{side-eff-goal}}$ is to achieve all side effects of α and to preserve all literals of the initial state α_{init} that are not changed by α (see Section 3.3). When the learner cannot find a replacing plan, it promotes one of the side effects of α to a primary effect.

The number of one-operator plans considered by the learner depends on the success-probability parameters ε_u and δ_u. The smaller the values of ε_u and δ_u, the more plans the learner must consider to guarantee the required probability of completeness. We denote the number of plans considered for every operator α by m. In Section 6, we show how to compute the value of m from given ε_u and δ_u.

For every operator α in the library, the learner randomly generates m initial states that satisfy the preconditions of α. We describe the random generation of the initial states in Section 5.4. After generating an initial state, the learner considers the corresponding plan $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ and calls Prim-TWEAK to search for a primary-effect justified replacing plan, with a cost at most $C_u \cdot cost(\alpha)$. If Prim-TWEAK does not find such a replacing plan, the learner promotes one of the side effects of α to a primary effect and uses this new selection of primary effects in subsequent learning. After the learner has considered m one-operator plans for every operator in the library, it terminates and outputs the resulting selection of primary effects.

5.3. Example of learning primary effects

We describe the application of the learning algorithm to the robot domain (see Fig. 2), with the initial selection described in Example 5.1:

go(x, y)	$\{\texttt{robot-in}(y), \neg\texttt{robot-in}(x)\}$
throw(x, y)	$\{\neg\texttt{ball-in}(x)\}$
carry-ball(x, y)	$\{\texttt{ball-in}(y)\}$
break(x, y)	$\{\texttt{door}(x, y)\}$

We assume that the maximal allowed cost increase is $C_u = 2$ and the learner first considers the operator **throw**. The side effect of this operator in the initial selection is the new position of the ball.

Suppose that the learner generates the initial state α_{init} in which the robot and the ball are in room 1; this state satisfies the preconditions of **throw**(1, 2). The learner then generates the goal $\alpha_{\text{side-eff-goal}}$; this goal includes moving the ball to room 2 (which is the side effect of **throw**) and leaving the robot in room 1 (which is the part of the initial state that must remain unchanged).

The learner calls Prim-TWEAK to generate a primary-effect justified plan, with a cost at most $C_u \cdot cost(\textbf{throw})$, which is $2 \cdot 2 = 4$. Prim-TWEAK does not find such a plan, because the cheapest primary-effect justified plan that achieves the goal is (**carry-ball**(1, 2), **go**(2, 1)), the cost of which is 5.

Since Prim-TWEAK has not found a replacing plan, the learner chooses the side effect of **throw**, `ball-in`, as a new primary effect. If the operator had several side effects, the learner could choose any of them; however, the operator **throw** has only one side effect. The selection of primary effects becomes as follows:

go(x, y)	{`robot-in`(y), ¬`robot-in`(x)}
throw(x, y)	{`ball-in`(y), ¬`ball-in`(x)}
carry-ball(x, y)	{`ball-in`(y)}
break(x, y)	{`door`(x, y)}

We assume that the learner next considers the operator **break**. Suppose that the robot is initially in room 4 and the goal $\alpha_{\text{side-eff-goal}}$ is to move the robot to room 1, which can be achieved by **break**(4, 1).

The learner calls Prim-TWEAK to generate a primary-effect justified plan for achieving this goal, with a cost at most $C_u \cdot cost(\textbf{break})$, which is $2 \cdot 4 = 8$. Prim-TWEAK finds such a plan, (**go**(4, 3), **go**(3, 2), **go**(2, 1)), the cost of which is 6.

Since the planner has found a primary-effect justified replacing plan within the specified cost bound, the learner does *not* choose a new primary effect of **break**.

5.4. Generating random initial states

For every operator α in the library, the learning algorithm has to generate m random initial states that satisfy the preconditions of α, based on some probability distribution over the set of all states that satisfy α's preconditions.

The probability of generating a state during the learning process should be the same as the probability of encountering this state in planning. We approximate the probability of encountering a state that satisfies the preconditions of α by the frequency with which this state appears immediately before α in total-order solution plans.

If we have a large library of previously generated solutions, we may use this library to determine the frequencies with which different states appear immediately before α and use these frequencies in generating random states for the learning algorithm.

In the absence of a library of solutions, we generate solutions for random planning problems and use these solutions to determine the frequencies of states. The generation of random planning problems is based on the assumption that all possible planning problems occur equally often.

5.5. Order of processing operators

We have not specified the order in which the learning algorithm processes the operators. Different orders may result in different selections of primary effects. Experiments in several domains demonstrated that processing operators in the increasing order of their costs usually, although not always, helps to avoid redundant primary effects. We used this processing order in the implementation of the learner.

We now give an informal justification for this order. If we may use some operator α_1 in a replacing plan for α_2, then the learner should process α_1 before α_2, in order to use the newly selected primary effects of α_1 in constructing a replacing plan for α_2. Since the algorithm usually uses cheap operators in replacing plans for more expensive operators, it should process cheap operators first.

If two operators, α_1 and α_2, have the same cost and α_1 has fewer side effects than α_2, then the learning algorithm processes α_1 before α_2. This heuristic is based on the observation that the larger the number of primary effects, the higher the probability to choose a redundant primary effect among them. If the algorithm uses the newly selected primary effects of α_1 in constructing a replacing plan for α_2, it reduces the number of candidates for a new primary effect among the side effects of α_2.

6. Sample complexity of the learning algorithm

The learning algorithm considers m randomly selected initial states for every operator α (see Fig. 5). The value of m depends on the success-probability parameters ε_u and δ_u, specified by the user. We determine the required value of m, using the theory of *probably approximately correct* (*PAC*) *learning* [18]. Researchers have investigated various ways of applying this theory in designing learning and search algorithms [4]. The dependency between m and the values of ε_u and δ_u is called the *sample complexity* of the learning algorithm.

We first define an *approximately correct* selection of primary effects of an operator, which ensures that the operator almost always has a primary-effect justified replacing plan. We derive a dependency between m and the probability of learning an approximately correct primary-effect selection for a given operator (Section 6.1).

We then relate the probability of completeness and near optimality of planning to the approximate-correctness probabilities for individual operators. We use this relationship in computing the required number m of initial states from the user-specified parameters ε_u and δ_u (Section 6.2).

6.1. Number of states to learn an operator's primary effects

We consider learning primary effects of some operator α. We denote the number of side effects of α in the initial selection by s and the side effects themselves by

$l_1, l_2, \ldots, l_s$. The learning algorithm generates m initial states that satisfy the preconditions of α. For each initial state, the learner calls Prim-TWEAK to search for a primary-effect justified plan that achieves the side effects of α.

The randomly selected initial states are *learning examples*. Let us denote the set of *all* states that satisfy the preconditions of α by $States_\alpha$. The algorithm selects learning examples from this set of states, $States_\alpha$, using some probability distribution over $States_\alpha$. We assume that the probability of selecting a state during the learning process is the same as the probability of encountering this state in planning with the learned primary effects. This assumption, called the *stationary assumption* of PAC learning, is essential for deriving the dependency between m and the values of ε_u and δ_u.

When Prim-TWEAK cannot find a replacing plan, the learner promotes one of the side effects of α to a primary effect. We assume that the learning algorithm first promotes l_1, then l_2, then l_3, and so on. If the algorithm promotes j effects of α during the learning process, then $l_1, \ldots, l_j$ are primary effects in the learned selection and $l_{j+1}, \ldots, l_s$ are side effects.

The number of promoted primary effects, j, depends on the initial states used in learning. Different values of j correspond to different selections of primary effects of α. The number of promoted effects is between 0 and s, which means that the algorithm generates one of $(s+1)$ possible selections of primary effects. These selections are the *hypotheses* of PAC learning. The set of all selections that can be generated by the learner is called the *hypothesis space*; it contains $(s+1)$ different hypotheses.

We denote the maximal number of effects of an operator in the planning domain by E. Since s is the number of side effects of α in the initial selection, we have $s \leqslant E$. Therefore, for every operator α in the domain, the hypothesis space contains at most $(E+1)$ hypotheses.

We say that the learned selection of primary effects of α is *consistent* with an initial state α_{init} that satisfies the preconditions of α, if Prim-TWEAK can find a primary-effect justified replacing plan, with a cost at most $C_u \cdot cost(\alpha)$, for the corresponding one-operator plan $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$. Observe that selecting additional primary effects does not violate consistency. Therefore, the learned selection will be consistent with all m initial states used by the learning algorithm; however, the selection may not be consistent with other states that satisfy the preconditions of α.

The *error* of PAC learning for a specific operator α is the probability that the learned selection of primary effects is not consistent with a randomly selected initial state α_{init}. The selection is *approximately correct* if the error is no larger than a certain small positive value ε_α.

Since the m states used in learning are selected at random from $States_\alpha$, we intuitively expect that the learned selection is consistent with most other states from $States_\alpha$. Therefore, if m is sufficiently large, the learned selection is likely to be approximately correct.

The theory of PAC learning formalizes this intuition and gives an upper bound for the probability that the learned selection of primary effects is *not* approximately correct. For a hypothesis space with at most $(E+1)$ hypotheses, the probability of learning a hypothesis that is not approximately correct is no larger than the following expression [2]:

$$(E+1) \cdot (1-\varepsilon_\alpha)^m.$$

The learning algorithm is *probably approximately correct* if this probability is no larger than a certain small positive value δ_α:

$$(E+1) \cdot (1-\varepsilon_\alpha)^m \leqslant \delta_\alpha.$$

It is a classical PAC learning inequality, used for determining the required number of learning examples, m. The inequality holds if m satisfies the following condition [2, 16]:

$$m \geqslant \frac{1}{\varepsilon_\alpha} \cdot \ln \frac{E+1}{\delta_\alpha}. \tag{7}$$

We assume that the selected values of ε_α are the same for all operators in the planning domain and that the values of δ_α are also the same for all operators. Then, if m satisfies inequality (7), learning is probably approximately correct for all operators in the domain.

6.2. Number of initial states for the user-specified parameters

We now relate the required number of learning examples, m, to the user-specified parameters ε_u and δ_u of the learning algorithm (see Section 5.2).

The value ε_u is the maximal allowed probability of failure to solve a randomly selected planning problem, within the cost increase C_u, when planning with the learned primary effects. We estimate this failure probability ε_u in terms of ε_α.

Suppose that the learned selection of primary effects is approximately correct and that we use this selection to solve problems with optimal-solution sizes up to a certain number N. We consider the use of the learned primary effects in solving some planning problem. If we can replace every operator α in an optimal solution by a primary-effect justified replacing plan, with a cost no larger than $C_u \cdot cost(\alpha)$, then the overall problem has a primary-effect justified solution within the cost increase C_u (see the proof of Theorem 3.2).

The probability that we can replace all operators of the optimal plan is at least $(1-\varepsilon_\alpha)^N$. Therefore, the probability that we cannot replace at least one operator is at most

$$1-(1-\varepsilon_\alpha)^N \leqslant N \cdot \varepsilon_\alpha.$$

We must ensure that this probability is no larger than the user-specified value ε_u:

$$N \cdot \varepsilon_\alpha \leqslant \varepsilon_u. \tag{8}$$

We next estimate the probability that the learned primary effects are not approximately correct. Suppose that the planning domain contains L different operators. The selection is not approximately correct if the selected primary effects of at least one operator are not approximately correct, the probability of which is at most

$$1-(1-\delta_\alpha)^L \leqslant L \cdot \delta_\alpha.$$

This probability must be no larger than the user-specified bound δ_u:

$$L \cdot \delta_\alpha \leqslant \delta_u. \tag{9}$$

We now rewrite inequality (8) as $1/\varepsilon_\alpha \geqslant N/\varepsilon_u$ and inequality (9) as $1/\delta_\alpha \geqslant L/\delta_u$, and substitute these lower bounds for $1/\varepsilon_\alpha$ and $1/\delta_\alpha$ into inequality (7):

$$m \geqslant \frac{N}{\varepsilon_u} \cdot \ln \frac{L \cdot (E+1)}{\delta_u}.$$

We use the minimal value of m that satisfies this inequality in the learning algorithm:

$$m = \left\lceil \frac{N}{\varepsilon_u} \cdot \ln \frac{L \cdot (E+1)}{\delta_u} \right\rceil, \tag{10}$$

where ε_u and δ_u are the user-specified parameters, L is the number of operators in the problem domain, E is the maximal number of effects of an operator, and N is the maximal possible size of an optimal solution for the planning problems that we need to solve.

We use Eq. (10) to compute the value of m at Step 1 of the learning algorithm (see Fig. 5). This dependency between the number of learning examples, m, and the success-probability parameters ε_u and δ_u is called the *sample complexity* of the learning algorithm.

7. Search reduction: experiments

We present a series of experiments on planning with primary effects selected by the learning algorithm. The experiments confirm the analytical prediction that the use of primary effects exponentially reduces planning time.

We implemented unrestricted TWEAK, Prim-TWEAK, and the algorithm that selects primary effects in Allegro Common Lisp, on a Sun 1000 machine. The planners in the experiments used best-first search.

We describe experiments with artificial planning domains (Sections 7.1 and 7.2), with an extended version of the robot domain (Section 7.3), and with a manufacturing domain (Section 7.4).

7.1. Artificial planning domains

We consider a family of artificial domains, similar to the domains used by Barrett and Weld for evaluating the efficiency of partial-order planning [1]. We can independently vary different features of these domains, which enables us to perform controlled experiments.

We define a planning problem by n initial-state literals, $init_0, init_1, \ldots, init_{n-1}$, and n goal literals, $goal_0, goal_1, \ldots, goal_{n-1}$. The domain contains n operators, $Op_0, Op_1, \ldots,$ Op_{n-1}. Every operator Op_i has the single precondition $init_i$ and $(k+1)$ effects, which include negating the initial-state literal $init_{i-1}$ and achieving the initial and goal literals $goal_i, goal_{i+1}, \ldots, goal_{i+k-1}$. The initial and goal literals are enumerated modulo n; that is, a more rigorous notation for the goal literals achieved by Op_i is $goal_{i \bmod n}, goal_{(i+1) \bmod n}, \ldots, goal_{(i+k-1) \bmod n}$. We denote the cost of the operator Op_i by $cost_i$.

For example, suppose that $n = 6$. If $k = 1$, then every operator Op_i achieves only one goal literal, $goal_i$, and the solution plan is $(Op_0, Op_1, Op_2, Op_3, Op_4, Op_5)$. If $k = 3$, then Op_0 achieves $goal_0$, $goal_1$, and $goal_2$, and Op_3 achieves $goal_3$, $goal_4$, and $goal_5$; therefore, the optimal solution is (Op_0, Op_3).

We vary the following features of the artificial domains in the controlled experiments:

- *Goal size.* The goal size is the number of goal literals, n. The size of an optimal solution changes in proportion to the number of the goal literals.
- *Effect overlap.* The effect overlap, k, is the average number of operators achieving the same literal.
- *Cost variation.* The cost variation is the statistical *coefficient of variation* of the costs of operators; that is, the ratio of the standard deviation of the costs to their mean. Intuitively, the cost variation is a measure of the relative difference between the costs of different operators.

The artificial domains model two important properties of real-world problems. First, if the goal size increases, the size of the optimal solution also increases. Second, if the effect overlap increases, then every operator can achieve more goal literals and the size of the solution decreases.

7.2. *Controlled experiments*

We now present the results of controlled experiments in the artificial domains. We varied the goal size, n, from 1 to 20 and used random permutations of the literals $goal_1, goal_2, \ldots, goal_n$ as planning goals. We considered the cost-increase values $C_u = 2$ and $C_u = 5$, and the effect-overlap values $k = 3$ and $k = 5$. We did not consider $k = 1$, because in this case all effects must be selected as primary and, hence, primary-effect restricted planning is equivalent to unrestricted planning. We also varied the cost variation, from 0 to 2.4. Finally, we considered two different values of ε_u and δ_u, which are $\varepsilon_u = \delta_u = 0.2$ and $\varepsilon_u = \delta_u = 0.4$.

We restricted the experiments to problems that the TWEAK planner solved within one minute. The optimal-solution sizes of such problems varied from four to seven operators, depending on the values of n and k.

In Figs. 6 and 7, we show the planning time of unrestricted TWEAK (UT) and Prim-TWEAK (PT) in the artificial domains. Prim-TWEAK used the primary effects selected by the learning algorithm. The horizontal axes of the graphs show the optimal-solution sizes of the problems and the vertical axes show the planning time. Note that the planning-time scale is *logarithmic*. Every point on each graph is the average planning time for ten problems. The vertical lines through points on the graphs show the *95%-confidence intervals*.

The Prim-TWEAK algorithm found solutions to *all* planning problems, within the user-specified cost increase. The use of primary effects considerably reduced planning time

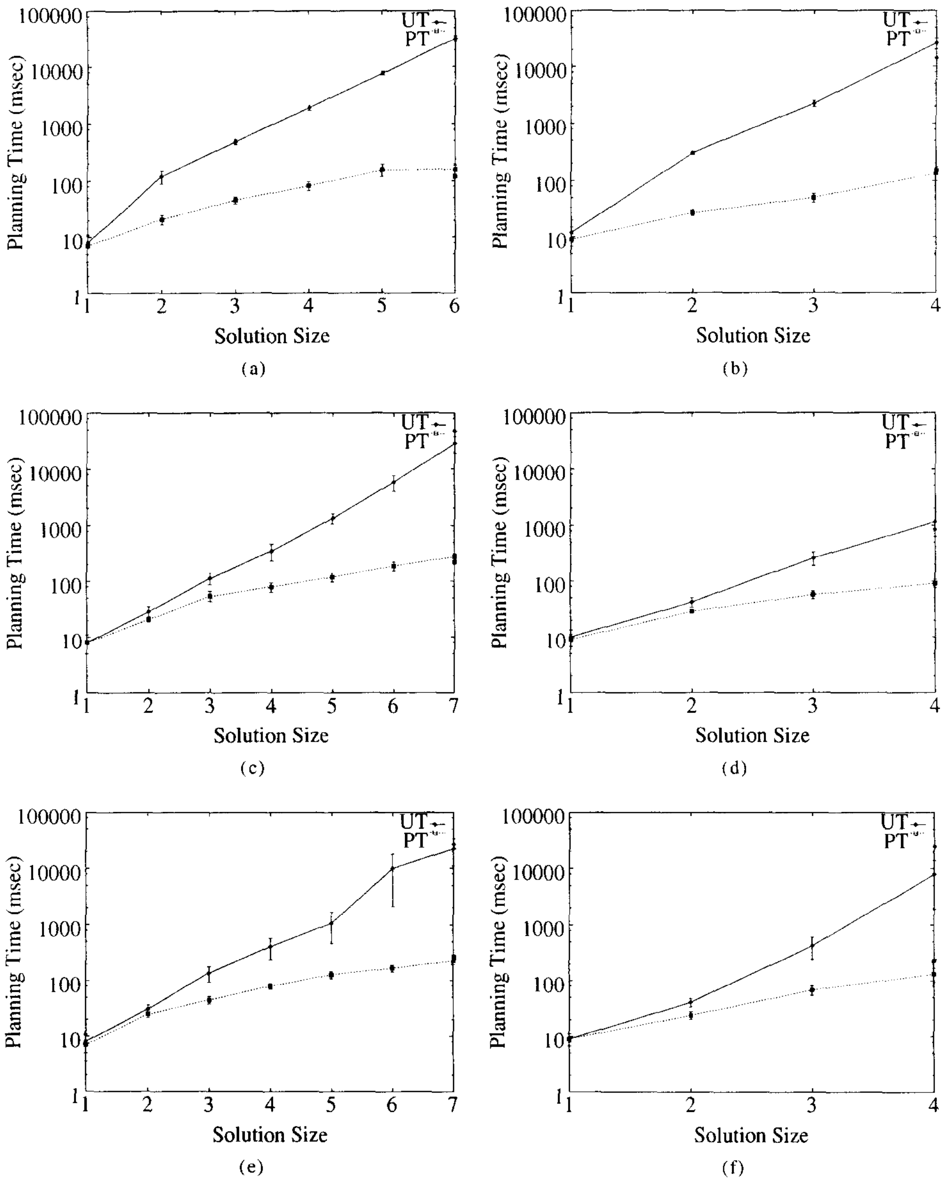

Fig. 6. Unrestricted TWEAK (UT) and Prim-TWEAK (PT) in the artificial domains: experiments with different effect overlap k (3 and 5) and cost variation (0, 0.4, and 2.4). (a) Cost increase C_u is 5, effect overlap k is 3, cost variation is 0, and $\varepsilon_u = \delta_u = 0.2$. (b) Cost increase C_u is 5, effect overlap k is 5, cost variation is 0, and $\varepsilon_u = \delta_u = 0.2$. (c) Cost increase C_u is 5, effect overlap k is 3, cost variation is 0.4, and $\varepsilon_u = \delta_u = 0.2$. (d) Cost increase C_u is 5, effect overlap k is 5, cost variation is 0.4, and $\varepsilon_u = \delta_u = 0.2$. (e) Cost increase C_u is 5, effect overlap k is 3, cost variation is 2.4, and $\varepsilon_u = \delta_u = 0.2$. (f) Cost increase C_u is 5, effect overlap k is 5, cost variation is 2.4, and $\varepsilon_u = \delta_u = 0.2$.

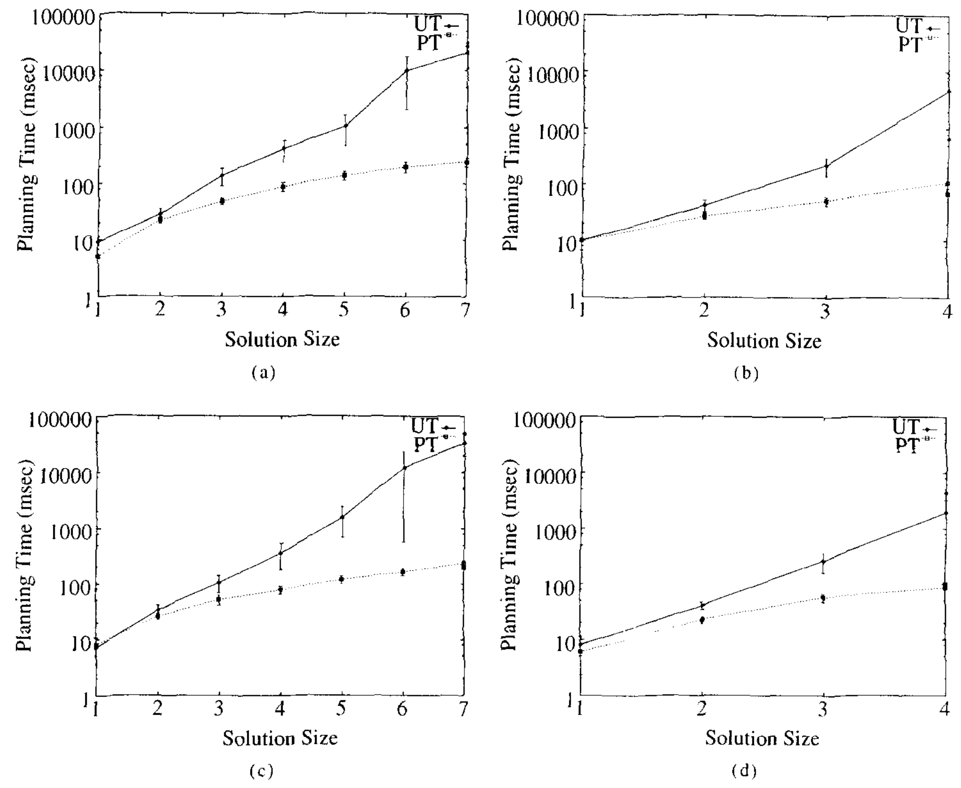

Fig. 7. Unrestricted TWEAK and Prim-TWEAK in the artificial domains (continued): experiments with different effect overlap k (3 and 5) and cost increase C_u (2 and 5). (a) Cost increase C_u is 2, effect overlap k is 3, cost variation is 2.4, and $\varepsilon_u = \delta_u = 0.2$. (b) Cost increase C_u is 2, effect overlap k is 5, cost variation is 2.4, and $\varepsilon_u = \delta_u = 0.2$. (c) Cost increase C_u is 5, effect overlap k is 3, cost variation is 2.4, and $\varepsilon_u = \delta_u = 0.4$. (d) Cost increase C_u is 5, effect overlap k is 5, cost variation is 2.4, and $\varepsilon_u = \delta_u = 0.4$.

in all experiments. The time reduction varied depending on the goal size, cost increase, effect overlap, and cost variation; however, the reduction was significant in *all* cases. The time saving grew exponentially with the optimal-solution size, which confirmed the analytical results (see Section 4).

We do not show the time for learning primary effects on the graphs. Instead, we summarize the learning time in Table 2. Observe that the learning time is much smaller than the planning time of unrestricted TWEAK. Also note that we need to learn primary effects only once for a planning domain. If we solve many problems in the domain, the amortized learning time is usually negligible.

7.3. Experiments in a robot domain

We now describe experiments in an extended robot world (see Fig. 8), where the robot can move between rooms, open and close doors, carry boxes, and climb tables (with or without boxes). In Table 3, we give the operators in this domain and their

Table 2
The time for learning primary effects in the artificial domains, for $\varepsilon_u = \delta_u = 0.2$

Cost variation	Effect overlap k	Learning time (msec)
0.0	3	30
0.0	5	40
0.4	3	30
0.4	5	60
2.4	3	40
2.4	5	50

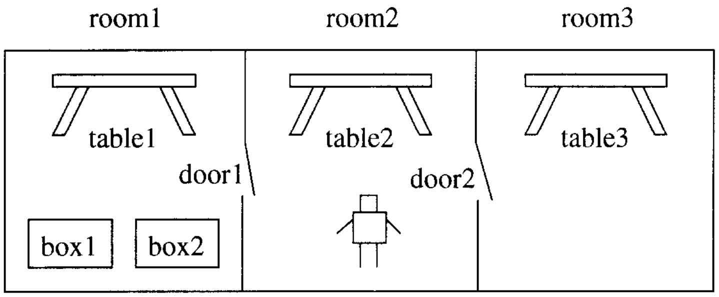

Fig. 8. The extended robot domain used in the experiments.

```
(defoperator                        || (defoperator
 :name '(climb-up ?table)           ||  :name '(carry-up ?box ?table)
 :prec '((robot-at ?table)          ||  :prec '((robot-at ?table)
         (robot-on-floor))          ||          (robot-on-floor)
 :eff '((robot-on ?table)           ||          (box-at ?table)
        (not robot-on-floor))       ||          (box-on-floor ?box))
 :cost 1)                           ||  :eff '((robot-on ?table)
                                    ||         (box-on-table ?box ?table)
                                    ||         (not box-on-floor ?box)
                                    ||         (not robot-on-floor))
                                    ||  :cost 4)
```

Fig. 9. The encoding of the table-climbing operators.

effects. The words preceded by "?" in the operator description denote variables; for example, ?box is a variable that denotes an arbitrary box, and ?from-loc and ?to-loc are variables that denote locations within a room. We show the full encoding of the two table-climbing operators in Fig. 9.

We used the learning algorithm to select primary effects, with $C_u = 5$ and $\varepsilon_u = \delta_u = 0.1$. The algorithm selected primary effects in 44 msec. In Table 3, we show the chosen primary effects.

Table 3
The effects of operators in the extended robot domain

Effects	*Cost*	*Selected primary effects*
(go-within-room ?from-loc ?to-loc ?room)		
(robot-at ?to-loc) (not robot-at ?from-loc)	1	(robot-at ?to-loc) (not robot-at ?from-loc)
(go-thru-door ?from-room ?to-room ?door)		
(robot-in-room ?to-room) (not robot-in-room ?from-room)	2	(robot-in-room ?to-room) (not robot-in-room ?from-room)
(climb-up ?table)		
(robot-on-table ?table) (not robot-on-floor)	1	(robot-on-table ?table) (not robot-on-floor)
(climb-down ?table ?room)		
(robot-on-floor) (not robot-on-table ?table)	1	(robot-on-floor) (not robot-on-table ?table)
(open ?door)		
(status ?door open) (not status ?door closed)	1	(status ?door open) (not status ?door closed)
(close ?door)		
(status ?door closed) (not status ?door open)	1	(status ?door closed) (not status ?door open)
(carry-within-room ?box ?from-loc ?to-loc ?room)		
(robot-at ?to-loc) (box-at ?box ?to-loc) (not robot-at ?from-loc) (not box-at ?box ?from-loc)	2	(box-at ?box ?to-loc) (not box-at ?box ?from-loc)
(carry-thru-door ?box ?from-room ?to-room ?door)		
(robot-in-room ?to-room) (box-in-room ?to-room) (not robot-in-room ?from-room) (not box-in-room ?from-room)	4	(box-in-room ?to-room) (not box-in-room ?from-room)
(carry-up ?box ?table)		
(robot-on-table ?table) (box-on-table ?box ?table) (not robot-on-floor) (not box-on-floor ?box)	4	(box-on-table ?box ?table) (not box-on-floor ?box)
(carry-down ?box ?table)		
(robot-on-floor) (box-on-floor ?box) (not robot-on-table ?table) (not box-on-table ?box ?table)	4	(box-on-floor ?box) (not box-on-table ?box ?table)

Table 4
Unrestricted TWEAK (UT) and Prim-TWEAK (PT) in the robot domain

Planning goal	Optimal-solution size	Planning time (msec)		Average branching factor	
		UT	PT	UT	PT
(robot-on-table table2)	1	30	20	1.50	1.00
(status door1 open)	2	90	90	2.00	1.43
(robot-in-room room1)	3	150	130	1.73	1.33
(box-on-table box1 table1)	5	550	360	2.28	1.27
(robot-on-table table3)	5	570	370	2.03	1.27
(robot-on-table table1) and (status door1 closed)	6	2250	1160	2.11	1.37
(box-on-table box2 table1) and (status door2 open)	7	4650	2520	2.27	1.41
(box-at box2 table2)	7	6200	2760	2.06	1.28
(box-on-table box2 table2)	8	15090	4410	2.09	1.36

In Table 4, we summarize the performance of unrestricted TWEAK and Prim-TWEAK on nine different problems, with randomly selected goals. The initial state of all problems is as shown in Fig. 8, with both doors closed. The Prim-TWEAK algorithm found optimal solutions to all nine problems. The use of primary effects considerably reduced planning time.

7.4. *Experiments in a manufacturing domain*

We next describe the use of primary effects in a manufacturing domain, similar to the domain used by Smith and Peot in their analysis of abstraction planning [17]. This manufacturing domain is a simplified version of the PRODIGY process-planning domain [12]. In Table 5, we give the operators of the manufacturing domain and their effects.

We ran the learning algorithm with $C_u = 5$ and $\varepsilon_u = \delta_u = 0.1$. The algorithm selected the primary effects shown in Table 5. We then used the planner to solve one hundred randomly generated problems, with and without the use of primary effects.

In Fig. 10, we show the performance of unrestricted TWEAK and Prim-TWEAK on problems with different goal sizes; we also show the 95%-confidence intervals. The Prim-TWEAK planner solved all the problems and its running time was much smaller than the running time of unrestricted TWEAK. The time saving grew exponentially with the goal size.

8. Extensions to the learning algorithm

We briefly describe three heuristics that often improve the quality of the primary effects selected by the learning algorithm. Note that we did *not* use these heuristics in the experiments of Section 7.

Table 5
The effects of the operators in the manufacturing domain

Effects		*Cost*	*Selected primary effects*
(cut-roughly ?part)			
(cut ?part)	(not finely-cut ?part)	1	(cut ?part)
(not drilled ?part)	(not finely-drilled ?part)		
(not polished ?part)	(not finely-polished ?part)		
(drill-roughly ?part)			
(drilled ?part)	(not finely-drilled ?part)	1	(drilled ?part)
(not polished ?part)	(not finely-polished ?part)		
(polish-roughly ?part)			
(polished ?part)	(not finely-polished ?part)	1	(polished ?part)
(cut-finely ?part)			
(cut ?part)	(finely-cut ?part)	2	(finely-cut ?part)
(not drilled ?part)	(not finely-drilled ?part)		
(not polished ?part)	(not finely-polished ?part)		
(drill-finely ?part)			
(drilled ?part)	(finely-drilled ?part)	2	(finely-drilled ?part)
(not polished ?part)	(not finely-polished ?part)		
(polish-finely ?part)			
(polished ?part)	(finely-polished ?part)	2	(finely-polished ?part)

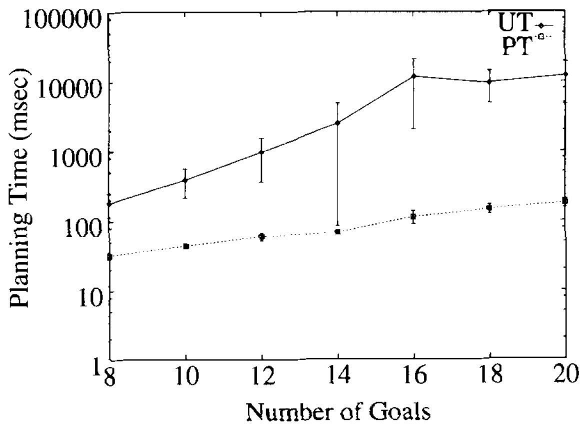

Fig. 10. Unrestricted TWEAK (UT) and Prim-TWEAK (PT) in the manufacturing domain.

8.1. Heuristic for choosing primary effects

When the learning algorithm described in Section 5.2 cannot find a replacing plan that achieves all side effects of some operator α, the algorithm promotes an arbitrary side effect of α to a primary effect (see Step 2(D) in Fig. 5). We now describe a heuristic for choosing a new primary effect among the side effects of α. The idea underlying this heuristic is to determine which side effects of α cannot be achieved by a replacing plan.

The algorithm generates a primary-effect justified plan, with a cost at most $C_u \cdot cost(\alpha)$, that achieves as many side effects of α as possible, but not necessarily all of them. If the resulting plan does not achieve some side effects of α, then the learner chooses one of these unachieved side effects as a new primary effect. Experiments show that the use of this heuristic reduces redundancy in selecting primary effects, especially in large-scale domains.

8.2. Primary effects and abstraction planning

The use of primary effects is closely related to abstraction planning. In particular, the ALPINE abstraction-generating algorithm may use the knowledge of primary effects of operators in constructing an abstraction hierarchy [13].

We may use the relationship between primary effects and abstraction in selecting primary effects [10]. We implemented an algorithm that selects primary effects in such a way as to maximize the number of levels in the abstraction hierarchy generated by ALPINE [8].

Experiments show that this selection heuristic helps to avoid redundant primary effects and choose primary effects that correspond to the human intuition. If we use the ALPINE algorithm for generating an abstraction hierarchy, then the selected primary effects also improve the quality of the hierarchy.

8.3. Primary effects for a subclass of goals

We assumed in designing the learning algorithm that planning goals may include any collection of literals. If we encounter only a subclass of possible goals, then we may select fewer primary effects without compromising completeness.

When goals are limited to a certain subclass, we may be able to remove some operators from the domain and solve all goals of the subclass with the remaining operators [6]. We can disregard the predicates that are not in the preconditions of any of the remaining operators.

We thus obtain a new planning domain, with a reduced set of operators and predicates. We then use the learning algorithm to select primary effects in this domain, which reduces the number of primary effects.

9. Conclusions

We have formalized the use of primary effects in planning and described a learning algorithm that selects primary effects automatically. We give a brief summary of the two main results.

Theory

Planning with properly selected primary effects is much more efficient than planning without primary effects. The saving in search time grows exponentially with the complexity of the planning problem. On the other hand, an improper selection of primary effects may increase planning time and result in the loss of completeness and optimality.

We have presented a necessary and sufficient condition for (1) completeness of planning with primary effects and (2) limited increase in the costs of solution plans. We have also identified the factors that determine the search reduction. The most important factor is the maximal cost increase, which determines the quality of solutions found by a primary-effect restricted planner and the search reduction due to the use of primary effects.

Implementation

We have implemented an inductive learning algorithm that selects primary effects of operators. The algorithm guarantees a high probability of completeness and limited increase in the solution costs. The time for learning primary effects is much smaller than the planning time. The experiments on planning with the learned primary effects have demonstrated a significant search reduction and confirmed that the search saving grows exponentially with problem complexity.

Acknowledgements

We would like to thank Manuela Veloso, Steve Minton, Yury Smirnov, Henry Rowley, and the anonymous reviewers for valuable comments and suggestions. We are grateful to the Computer Science Department of the University of Waterloo for providing facilities to perform the experiments. We used the graphical software implemented by Steven Woods in the experiments.

References

[1] A. Barrett and D.S. Weld, Partial order planning: evaluating possible efficiency gains, *Artif. Intell.* **67** (1994) 71-112.

[2] A. Blumer, A. Ehrenfeucht, D. Haussler and M.K. Warmuth, Occam's razor, *Inform. Process. Lett.* **24** (1987) 377-380.

[3] D. Chapman, Planning for conjunctive goals, *Artif. Intell.* **32** (1987) 333-377.

[4] W.W. Cohen, R. Greiner and D. Schuurmans, Probabilistic hill-climbing, in: S.J. Hanson, T. Petsche, M. Kearns and R.L. Rivest, eds., *Computational Learning Theory and Natural Learning Systems* (MIT Press, Boston, MA, 1994) 171-181.

[5] R.E. Fikes and N.J. Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving, *Artif. Intell.* **2** (1971) 189-208.

[6] E. Fink, Systematic approach to the design of representation-changing algorithms, in: *Proceedings Symposium on Abstraction, Reformulation, and Approximation* (1995) 54-61.

[7] E. Fink and M.M. Veloso, Formalizing the PRODIGY planning algorithm, in: M. Ghallab and A. Milani, eds., *New Directions in AI Planning* (IOS Press, Amsterdam, 1996) 261-271.

[8] E. Fink and Q. Yang, Automatically abstracting effects of operators, in: *Proceedings First International Conference on AI Planning Systems* (1992) 243-251.

[9] E. Fink and Q. Yang, Formalizing plan justifications, in: *Proceedings Ninth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, Vancouver, BC (1992) 9–14.

[10] E. Fink and Q. Yang, Search reduction in planning with primary effects, in: *Proceedings Workshop on Theory Reformulation and Abstraction* (1994) 39–55.

[11] E. Fink and Q. Yang, Planning with primary effects: experiments and analysis, in: *Proceedings IJCAI-95*, Montreal, Que. (1995) 1606–1611.

[12] Y. Gil, A specification of process planning for PRODIGY, Tech. Rept. CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1991).

[13] C.A. Knoblock, Automatically generating abstractions for planning, *Artif. Intell.* **68** (1994) 243–302.

[14] C.A. Knoblock and Q. Yang, Evaluating the trade-offs in partial-order planning algorithms, in: *Proceedings Tenth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, Banff, Alta. (1994) 279–286.

[15] J.S. Penberthy and D.S. Weld, UCPOP: a sound, complete, partial-order planner for ADL, in: *Proceedings Third International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA (1992) 103–114.

[16] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach* (Prentice-Hall, Englewood Cliffs, NJ, 1995).

[17] D.E. Smith and M.A. Peot, A critical look at Knoblock's hierarchy mechanism, in: *Proceedings First International Conference on AI Planning Systems* (1992) 307–308.

[18] L.G. Valiant, A theory of the learnable, *Commun. ACM* **27** (1984) 1134–1142.

[19] M.M. Veloso, J.G. Carbonell, M.A. Pérez, D. Borrajo, E. Fink and J. Blythe, Integrating planning and learning: the PRODIGY architecture, *J. Exper. Theoret. Artif. Intell.* **7** (1995) 81–120.

[20] D.E. Wilkins, *Practical Planning: Extending the Classical AI Planning Paradigm* (Morgan Kaufmann, San Mateo, CA, 1988).

[21] Q. Yang and C. Murray, An evaluation of the temporal coherence heuristic in partial-order planning, *Comput. Intell.* **10** (1994) 245–267.

[22] Q. Yang and J. Tenenberg, ABTWEAK: abstracting a non-linear, least-commitment planner, in: *Proceedings AAAI-90*, Boston, MA (1990) 204–209.

[23] Q. Yang, J. Tenenberg and S. Woods, On the implementation and evaluation of ABTWEAK, *Comput. Intell.* **12** (1996) 295–318.