Artificial Intelligence 272 (2019) 52–85

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Computing programs for generalized planning using a classical planner

Javier Segovia-Aguas [a,*], Sergio Jiménez [b], Anders Jonsson [a]

[a] Department of Information and Communication Technologies, Universitat Pompeu Fabra, Spain
[b] Department of Computer Systems and Computation, Universitat Politécnica de València, Spain

ARTICLE INFO

Article history:
Received 24 February 2017
Received in revised form 18 September 2018
Accepted 23 October 2018
Available online 30 January 2019

Keywords:
Generalized planning
Classical planning
Planning and learning
Program synthesis

ABSTRACT

Generalized planning is the task of generating a single solution (a *generalized plan*) that is valid for multiple planning instances. In this paper we introduce a novel formalism for representing generalized plans that borrows two mechanisms from structured programming: control flow and procedure calls. On one hand, control flow structures allow to compactly represent generalized plans. On the other hand, procedure calls allow to represent hierarchical and recursive solutions as well as to reuse existing generalized plans. The paper also presents a compilation from generalized planning into classical planning which allows us to compute generalized plans with off-the-shelf planners. The compilation can incorporate prior knowledge in the form of auxiliary procedures which expands the applicability of the approach to more challenging tasks. Experiments show that a classical planner using our compilation can compute generalized plans that solve a wide range of generalized planning tasks, including sorting lists of variable size or DFS traversing variable-size binary trees. Additionally the paper presents an extension of the compilation for computing generalized plans when generalization requires a high-level state representation that is not provided a priori. This extension brings a new landscape of benchmarks to classical planning since classification tasks can naturally be modeled as generalized planning tasks, and hence, as classical planning tasks. Finally the paper shows that the compilation can be extended to compute control knowledge for off-the-shelf planners and solve planning instances that are difficult to solve without such additional knowledge.

1. Introduction

Generalized planning is the task of generating a single solution valid for multiple planning instances. Unlike classical plans, generalized plans are built with algorithm-like structures that make their execution flexible and adaptable to different planning instances. As a consequence generalized plans can solve planning tasks beyond the scope of classical planning like planning tasks of unbound size, with partially observable states and/or with non-deterministic actions [1–5].

In this paper we introduce an innovative formalism for representing generalized plans that we call *planning programs* and that borrows two effective mechanisms from structured programming: control flow, in the form of conditional *goto* instructions, and procedure calls. While conditional *gotos* allow us to compactly encode complex solutions valid for a set

* Corresponding author.
E-mail address: javier.segovia@upf.edu (J. Segovia-Aguas).

https://doi.org/10.1016/j.artint.2018.10.006
0004-3702/

of planning instances, *procedure calls* allow us to represent hierarchical and recursive solutions as well as to reuse existing generalized plans.

Our approach for computing generalized plans of this kind is defining a compilation into classical planning, making it possible to generate generalized plans using an off-the-shelf classical planner. Briefly, our compilation encodes a bounded number of program lines and a program counter, and introduces actions for programming and executing the instructions on different program lines. A solution to the classical planning task resulting from the compilation is a sequence of actions that encodes the instructions in the program lines and executes them on all the instances of the generalized planning task. Interestingly the compilation can incorporate prior knowledge, in the form of auxiliary procedures, and automatically complete the definition of the remaining program. This allows us to reuse existing generalized plans and to address more challenging generalized planning tasks by incrementally creating hierarchies of generalized plans.

In some domains the computation of a compact generalized plan is only possible if a particular high-level representation of the states is available. For instance in the classic *blocksworld* domain, the *high-level state features* that capture when a block is at its goal position [6,7]. In most applications of generalized planning such features are hand-coded by domain experts. In this paper we present an extension of our compilation to integrate the computation of features with the computation of generalized plans, generating solutions that generalize without a prior high-level representation of the states. It is worth mentioning that this extension allows us to model supervised classification tasks as classical planning instances.

Even if the language for representing the generalized plans is expressive it may still be complex to achieve generalization if the instances in the generalized planning task do not share a clear common structure. In such cases representing and computing solutions with non-deterministic execution may be useful [8]. Solutions of this kind are more flexible because they allow open segments that are only determined when the solution is executed in a particular instance. In this paper we introduce two different extensions of our *planning program* formalism to achieve generalization through non-deterministic execution: *choice instructions* and *lifted action instructions*.

For clarity, we present each extension separately as a modification of basic planning programs that do not involve procedures. However, it is straightforward to combine multiple extensions, e.g. *high-level state features* and *choice instructions*. All source code and related documentation is available in a public repository (https://github.com/aig-upf/automated-programming-framework), which also explains how to configure different combinations of our extensions.

A first description of the compilation and its extensions previously appeared in several conference papers [9–11]. Compared to the conference papers, the present paper includes the following novel material:

- We set down the theoretical foundations of the *planning program* formalism and show that plan validation and plan existence for planning programs is PSPACE-complete. In addition we provide formal characterizations of the solutions that can be represented with our formalism and prove that the expressiveness of our different forms of planning programs is equivalent.
- We unify the different formulations of our compilation for computing planning programs and its extensions for computing planning programs with procedures and with high-level state features and provide formal proofs of their soundness and completeness.
- We introduce two novel formalisms for planning programs with non-deterministic execution: *choice instructions* and *lifted action instructions*.
- We show how to implement the validation mechanism for planning programs within our compilation. We use this mechanism to show that reusing generalized plans and using generalized plans as control knowledge allow us to solve planning instances that are difficult to solve using current classical planners, like blocksworld instances with 100 blocks.
- We report new experimental results using our compilation with different off-the-shelf planners. In addition we introduce new challenging domains for generalized planning that correspond to well-known programming tasks as well as a new domain from program synthesis and discuss the pros and cons of applying our approach to this kind of tasks.

The paper is structured as follows, Section 2 puts our work into context by reviewing previous work on generalized planning and program synthesis. Section 3 defines the planning models we will rely on along this work, classical planning with conditional effects and generalized planning. Section 4 presents our *planning program* formalism for representing generalized plans and defines their theoretical properties. Section 5 presents our compilation and its extension to compute planning programs and planning programs with callable procedures for solving generalized planning task. Section 6 shows how to extend the compilation to integrate the computation of features with the computation of generalized plans. Section 7 shows how to compute planning programs with non-deterministic execution. Finally Section 8 wraps-up our work and discusses open issues and future work.

2. Related work

The computation of general strategies for planning has been studied since the early days of AI. Different formalisms for representing planning solutions that generalize and different algorithms for computing them have been proposed.

Macro-actions (i.e. action sub-sequences) were among the first suggestions to compute general knowledge for planning and there are several approaches in the literature for computing them [12–14]. However, the sequential execution flow fixed

by macro-actions is usually too rigid and, even when macros are parameterized, they have to be combined with control-flow structures in order to generalize to more planning instances.

Generalized policies are a more flexible formalism than macros. A generalized policy is a set of rules mapping states and goals into actions; hence generalized policies are *reactive* and do not explicitly represent action sequences. Computing good generalized policies is however complex. Early algorithms for computing generalized policies [6,7] first compute sequential plans, and then attempt to generalize the policy rules from these plans, a difficult task because of the high number of symmetries and transpositions that commonly appear in sequential plans. Moreover, a *generalized policy* cannot be added straightforward to a domain theory. There are however, effective mechanisms for exploiting *generalized policies* as *Domain-specific Control Knowledge* (DCK) to guide the search process of heuristic planners [15,16].

Recent algorithms for generalized planning tightly integrate planning and generalization and interleave *programming* the solution with validating it on multiple test cases. Works following this approach impose a stronger constraint on the planning tasks to solve and share, not only the domain theory (actions and predicates schemes), but the state space [10] or at least the observation space [2]. Algorithms of this kind search in the space of possible solutions and, similar to what is done in SATPLAN approaches [17], the search is typically bound to a maximum size of the solution to keep it tractable. This approach includes works that compile the generalized planning task into a conformant planning task [2], a CSP [18] or a Prolog program [5]. The work presented in this paper is also included in this group. Compared to previous work, our contributions are 1) that generalized plans represented as programs are highly intuitive to humans; 2) that we can compute generalized plans using an off-the-shelf classical planner; and 3) that our *planning program* formalism can represent hierarchical and recursive solutions, to reuse existing generalized plans and to compute generalized plans with high-level state features.

An alternative approach is to look at a single planning instance, compute a solution that solves that instance, generalize it, and then merge it with the previously found solutions incrementally increasing the coverage of the generalized plan. This approach is related to previous works on *plan repair* [19] and *Case-Based Planning* (CBP) [20] since it demands identifying why a solution does not cover a given instance and adapting it to the uncovered instance. The Distill system [1] lies in this category and, as our work, uses programs to represent generalized plans. However, its representation is different and its performance was not tested over a wide range of diverse generalized planning tasks.

The Golog family of action languages has proven useful for programming autonomous behavior that is able to generalize [21]. Apart from conditionals, loops and recursive procedures, Golog programs can contain *non-deterministic parts*. A Golog program does not need to represent a fully specified solution, but a sketch of it, where the non-deterministic parts are gaps to be filled by the system. The Golog programmer can determine the right balance between predefined behavior and leaving certain parts to be solved by the system by means of search. The basic Golog interpreter uses the PROLOG back-tracking mechanism to resolve the search. This mechanism basically amounts to do a blind search so, when addressing planning tasks, it soon becomes unfeasible for all but the smallest instance sizes. IndiGolog [22] extends Golog to contain a number of built-in planning mechanisms. Furthermore the semantics compatibility between Golog and PDDL [23] can be exploited and a PDDL planner can be embedded [24] to address the sub-problems that are combinatorial in nature. We defined our own programming language that defines the space of possible generalized plans. In this language branching and loops are implemented with the same construct (*conditional gotos*) to keep the solution space as reduced as possible.

The series of works on generalized planning by Srivastava et al. proposes an expressive form of generalized plans with *choice actions* to decide the objects used by future plan steps [25,4]. Input instances in these works are expressed as an abstract first-order representation with transitive closure which allows to represent unbounded numbers of objects. The algorithm for computing generalized plans from this input implements a *bottom-up* strategy that identifies the class of open problems for an existing partial generalized plan and efficiently generate small instances of this class. The algorithm starts with an empty generalized plan, and incrementally increases its applicability by identifying a problem instance that it cannot solve, invoking a classical planner to solve that instance, generalizing the obtained solution and merging it back into the generalized plan. The process is repeated until producing a generalized plan that covers the entire desired class of instances (or when a predefined limit of the computation resources is reached). Although this approach benefits from off-the-shelf solvers to generate the solutions to the specific instances it requires developing specific techniques to generalize, adapt and merge plans. Our approach represents the generalized planning task to solve as a set of example tasks, similar to a *training set* in Machine Learning (ML). Each of these input tasks is a standard classical planning task coded in PDDL. We follow a *top-down* approach for generalized planning that, in a single planning episode, computes a generalized plan that covers all the input tasks and using exclusively an off-the-shelf classical planner.

Contingent planning [26] can be seen as an example of generalized planning where all the input instances to be solved (1), share the same goals and (2), are defined as the set of possible initial states of the contingent planning task. Contingent planners use state abstraction to represent the sets of possible states as *belief states* and include *sensing actions* that divide a given belief state in terms of the value of the sensed variable. A contingent plan can then be seen as a generalized plan. However, its tree-like representation can grow exponentially increasing the complexity of computing such plans. Previous work that automatically compute *finite state machines* (FSMs) propose changing the representation of contingent plans and allow loops that significantly reduce the size of contingent plans while increasing their coverage [2,9]. Generating high-level state features for generalized planning is also related to previous work on First Order MDPs [27,28]. These works adapt traditional dynamic programming algorithms to the symbolic setting and automatically generate first-order representations

of the value function with first-order regression. Here the contribution of our approach is following a compilation approach to generate useful state abstractions with off-the-shelf planners.

The task of synthesizing a program from a given specification is also related to our work on generalized planning. In the area of program synthesis there are two recent and successful approaches: *Programming by example* [29] and *Programming by sketching* [30]. In programming by example a program (or a set of programs) is generated to be consistent with a given set of given input–output examples using a domain specific search algorithm. Notable developed techniques for programming by example are part of the *Flash Fill* feature in Excel, Office 2013. In *Programming by sketching* programmers provide a partially specified program, a program that expresses the high-level structure of an implementation but that leaves holes in place of low level details to be filled by the synthesizer. This form of program synthesis relies on a programming language SKETCH for sketching partial programs and a SAT-based inductive synthesis procedure that efficiently synthesizes an implementation from a small number of test cases. This approach is able to deal with noisy examples and has been recently applied to supervised and unsupervised image classification achieving human performance results [31].

Bounded synthesis addresses the synthesis of finite-state transition systems that satisfy a given LTL formula [32]. The mainstream approach for bounded synthesis is compiling this task into a satisfiability modulo theories (SMT) problem. The compilation requires fixing a bound on the system parameters, such as the number of rejecting states. The main difference with our approach, is that we specify the task to solve by explicitly enumerating a set of test cases that the generalized plan should cover, similar to Machine Learning (ML) or to *unit tests* in Test-Driven Development.

Domain Control Knowledge is a general notion that refers to knowledge about the structure of planning solutions. *Planning with DCK* can also be seen as a form of generalized planning, that constrains the space of possible solutions, but requires a planner to produce a fully specified solution for a particular classical planing instance. This approach includes previous works for planning with control rules [33,34] and hierarchical planning [35,36].

Last but not least our approach for computing high-level state features is inspired by *version space learning* [37]. The hypothesis to learn consists of logic clauses with the form of *conjunctive queries* and the examples are logic facts that restrict the hypothesis forcing it to be consistent with the examples. Inductive Logic Programming (ILP) [38] also generates hypotheses from examples intersecting ML and Logic Programming. ILP has traditionally been considered a binary classification task but, in recent years, it covers the whole spectrum of ML such as regression, clustering and association analysis. The main contribution of our approach with respect to version space learning and ILP is the use of a classical planner to build and validate the learned hypotheses.

3. Background

To represent the set of planning instances to solve we use *classical planning with conditional effects*. Conditional effects make it possible to repeatedly refer to the same action even though their precise effects depend on the current state. This feature is useful to build generalized plans because, as shown in conformant planning [39], it can adapt the execution of a given sequence of actions to different initial states. We also introduce here our model for *generalized planning* in which the aim is to compute a plan that solves a *class* of planning instances, a set of instances that share some common structure, as opposed to a single planning problem.

3.1. Classical planning with conditional effects

We use F to denote a set of propositional variables or *fluents* describing a state. We will often assume that fluents are instantiated from predicates, as in PDDL [40]. Specifically, there exists a set of predicate symbols Ψ, and a *predicate* consists of a symbol $p \in \Psi$ and an associated argument list of arity $ar(p)$. Given a set of objects Ω, the set of fluents F is induced by assigning objects in Ω to the arguments of predicates, i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ where, given a set X, X^n is the n-th Cartesian power of X.

A *state* is an evaluation of fluents, i.e. a function $s : F \to \{0, 1\}$. A *partial state* is a partial function $s : F \to \{0, 1\}$. Given a (partial) state s and a subset of fluents $F' \subseteq F$, we use $s|_{F'}$ to denote the projection of s onto F', i.e. the partial state induced by restricting the domain of s to F'.

For each fluent $f \in F$ we introduce the term $\neg f$ representing the negation of f. Fluents and their negations are called *literals*. A state s corresponds to an equivalent literal set $\{f : s(f) = 1\} \cup \{\neg f : s(f) = 0\}$. The union $s \cup s'$ of two partial states is a new partial state which is well-defined as long as s and s' do not disagree on the value of some fluent. We can also use the notation $s' \subseteq s$ to denote that $s'(f) = s(f)$ whenever $s'(f)$ is defined. Explicitly including negative literals $\neg f$ simplifies subsequent definitions, but we often abuse notation by defining a state s only in terms of the fluents that are true in s, as is common in classical planning.

We consider the fragment of classical planning with conditional effects that includes negative conditions and goals. Under this formalism, a *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions with conditional effects. Each action $a \in A$ has a partial state $\mathsf{pre}(a)$ called the *precondition* and a set of conditional effects $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two partial states C (the condition) and E (the effect).

An action $a \in A$ is *applicable* in state s if and only if $\mathsf{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is

$$\mathsf{eff}(s, a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E,$$

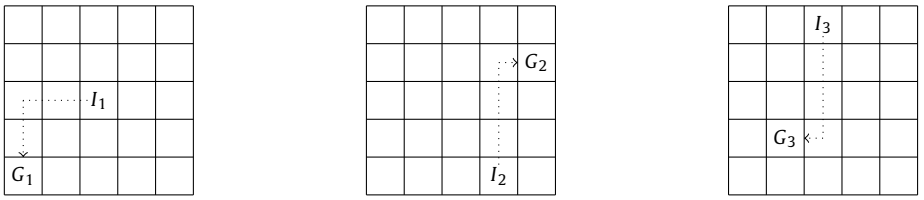


Fig. 1. Example of three individual classical planning tasks comprised in a generalized planning task $\mathcal{P} = \{P_1, P_2, P_3\}$.

i.e. effects whose conditions hold in s. We assume that $\mathsf{eff}(s, a)$ is a well-defined partial state for each state s and action a. The result of applying a in s is a new state $\theta(s, a)$ defined as

$$\theta(s,a)(f) = \begin{cases} \mathsf{eff}(s,a)(f), & \text{if } \mathsf{eff}(s,a)(f) \text{ is defined,} \\ s(f), & \text{otherwise.} \end{cases}$$

Given a planning frame $\Phi = \langle F, A \rangle$, a *classical planning problem* is a tuple $P = \langle F, A, I, G \rangle$, where I is an initial state and G is a goal condition, i.e. a partial state on F. A *plan* for P is an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ that induces a state sequence $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$ and, for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The plan π *solves* P if and only if $G \subseteq s_n$, i.e. if the goal condition is satisfied following the application of π in I.

3.2. Generalized planning

Our definition of the generalized planning task is based on that of Hu and De Giacomo [41], who define a generalized planning problem as a finite set of multiple individual planning problems $\mathcal{P} = \{P_1, \ldots, P_T\}$ that share the same observations and actions. Although actions are shared, in their formalism each action can have a different definition (precondition and conditional effects) in each individual planning problem.

In this work we restrict the above definition for generalized planning in two ways:

1. States are fully observable, so observations are equivalent to states, and sharing the observation set amounts to sharing the fluent set F.
2. Each action has the same definition (precondition and conditional effects) in each individual problem.

As a consequence, the individual problems $P_1 = \langle F, A, I_1, G_1 \rangle, \ldots, P_T = \langle F, A, I_T, G_T \rangle$ are classical planning problems that share the same planning frame $\Phi = \langle F, A \rangle$, and differ only in the particular initial state and goals.

Our definition of generalized planning is related to previous work on planning and learning that extract and reuse general knowledge from different tasks of the same domain [42,43]. In previous work, planning problems from the same domain share the same set of predicate symbols Ψ. In this paper, we impose a stronger constraint: the set of *objects* Ω assigned to the arguments of predicates in Ψ has to be shared, and hence also the induced fluent set F.

Fig. 1 shows an example of a generalized planning problem that comprises three individual classical planning tasks, $\mathcal{P} = \{P_1, P_2, P_3\}$. In this example F includes the fluents necessary for encoding the structure of the grid, the current and target position of the agent in the grid and the fluents that capture when the agent is at its target position. The shared set of actions, $A = \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$, comprises four actions each moving the agent one cell in one of the four cardinal directions. The three individual planning tasks are navigation tasks for moving the agent from cell $(3, 3)$ to cell $(1, 1)$ in the case of P_1, from cell $(4, 1)$ to cell $(5, 4)$ in the case of P_2 and from $(3, 5)$ to $(2, 2)$ in the case of P_3.

Note that our definition of generalized planning makes it possible to encode individual planning tasks with different grid sizes. To do so we first define enough fluents to represent the task with the largest number of rows and columns. We then define fluents of type $\mathsf{max}(x, a)$ and $\mathsf{max}(y, b)$ that encode variable grid boundaries: if $\mathsf{max}(x, a)$ is true, valid rows are in the range $[1, a]$, and likewise for columns. Hence smaller tasks will include the same fluents as larger tasks, but some of those fluents are unreachable (and thus not used) due to being outside of the established grid boundaries.

Given a generalized planning problem $\mathcal{P}$, by *generalized plan* we mean any construct that makes it possible to efficiently extract a solution plan π_t to each individual planning problem $P_t \in \mathcal{P}$, $1 \leq t \leq T$. In the literature, generalized plans have diverse forms that range from *DS-planners* [1] and *generalized polices* [7] to finite state machines (FSMs) [2]. Each of these representations has its own syntax and semantics but they all allow non-sequential execution flow to solve planning instances with different initial states and goals. A generalized plan for solving the generalized planning problem of Fig. 1 is given by: *moving the agent down and left until reaching the grid cell (1,1), moving the agent up until reaching the target row and finally, moving the agent right until reaching its target column*. In this work we restrict generalized plans to have the form of planning programs as defined in the next section.

According to our definition of the generalized planning task the particular case where $|\mathcal{P}| = 1$ corresponds to classical planning with conditional effects. The case for which all the individual problems in $\mathcal{P}$ share the same goals, $G_1 = G_2 = \cdots = G_{T-1} = G_T$, corresponds to conformant planning [39]. Nevertheless the form of the solutions is different. While solutions in

classical planning or in conformant planning are defined as sequences of actions, generalized plans relax this assumption and exploit a more expressive solution representation with non-sequential execution that can achieve more compact solutions.

4. Planning programs

This section introduces the basic version of the *planning program* formalism. In its simplest form, a planning program is a sequence of planning actions enhanced with *goto instructions*, i.e. conditional constructs for jumping to arbitrary locations of the program, allowing for non-sequential plan execution with branching and loops. We first define basic planning programs and prove that they are PSPACE-complete to compute. We then describe a compilation from generalized planning to classical planning which allows us to automatically compute planning programs, showing that the compilation is sound and complete. Finally we present several experimental results.

4.1. Basic planning programs

Given a classical planning frame $\Phi = \langle F, A \rangle$, a *basic planning program* $\Pi = \langle w_0, \ldots, w_n \rangle$ is a sequence of instructions. Each instruction w_i, $0 \leq i \leq n$, is associated with a *program line* i and is drawn from a set of instructions $\mathcal{I}$ defined as

$$\mathcal{I} = A \cup \mathcal{I}_{go} \cup \{\texttt{end}\}, \quad \mathcal{I}_{go} = \{\texttt{goto(i',!f)} : 0 \leq i' \leq n, f \in F\}.$$

In other words, each instruction is either an action instruction $a \in A$, a conditional goto instruction `goto(i',!f)` or a termination instruction `end`. A termination instruction acts as an explicit marker that program execution should end, similar to a `return` statement in programming. We explicitly require that the last instruction w_n should equal `end`, and since this instruction is fixed, we say that Π has $|\Pi| = n$ program lines, even though Π in fact contains $n + 1$ instructions.

The execution model for a planning program Π consists of a *program state* (s, i), i.e. a pair of a planning state $s \subseteq F$ and a program counter whose value is the current program line i, $0 \leq i \leq n$. Given a program state (s, i), the execution of instruction w_i on line i is defined as follows:

- If $w_i \in A$, the new program state is $(s', i + 1)$, where $s' = \theta(s, w_i)$ is the result of applying action w_i in planning state s, and the program counter is simply incremented.
- If $w_i = \texttt{goto(i',!f)}$, the new program state is $(s, i + 1)$ if $f \in s$, and (s, i') otherwise. We adopt the convention of jumping to line i' whenever f is *false* in s. Note that the planning state s remains unchanged.
- If $w_i = \texttt{end}$, the execution terminates.

To execute a planning program Π on a planning problem $P = \langle F, A, I, G \rangle$, we set the initial program state to $(I, 0)$, i.e. the initial state of P and program line 0. We say that Π *solves* P if and only if the execution terminates and the goal condition holds in the resulting program state (s, i), i.e. $G \subseteq s \wedge w_i = \texttt{end}$. A planning program Π can fail to solve P for three reasons:

1. Execution terminates in program state (s, i) but the goal condition does not hold, i.e. $G \not\subseteq s \wedge w_i = \texttt{end}$.
2. When executing an action $w_i \in A$ in program state (s, i), the precondition of w_i does not hold, i.e. $\mathsf{pre}(w_i) \not\subseteq s$.
3. Execution enters an infinite loop that never reaches an `end` instruction.

This execution model is *deterministic* and hence a basic planning program can be viewed as a form of compact reactive plan for the *class* of planning problems defined by the corresponding generalized planning task.

Fig. 2(a) shows an example planning program Π for navigating to the $(1, 1)$ cell in a grid. Variables x and y represent the position of the current grid cell. Instructions `dec(x)` and `dec(y)` decrement the value of x and y. The goto instructions `goto(0,!(x=1))` and `goto(2,!(y=1))` jump to line 0 when $x \neq 1$ and to line 2 when $y \neq 1$, respectively. For completeness, the formal PDDL definition of the grid navigation domain and the example planning problem in Fig. 2(b) appear in Appendix A. Note that the conditional effects of actions `dec(x)` and `dec(y)` are limited by the values in the domain of variables x and y, which are explicitly modeled as objects `v1`,...,`v5`. For example, attempting to decrement the value of x when $x = 1$ has no effect, i.e. the value of x remains equal to 1.

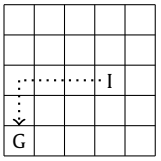
The execution of Π on the classical planning problem illustrated in Fig. 2(b) produces the following sequence of planning actions: $\langle$`dec(x)`, `dec(x)`, `dec(x)`, `dec(y)`, `dec(y)`$\rangle$. Note that Π solves *any* planning problem of this domain whose goal is to be at cell $(1, 1)$, no matter the grid size.

4.2. Theoretical properties of basic planning programs

In this section we prove several theoretical properties regarding basic planning programs. First, we show that *plan validation* and *plan existence* is PSPACE-complete. Since a planning program Π is defined in terms of the number of program lines $n = |\Pi|$, we focus on *bounded* plan existence; if the number of program lines is unbounded, a basic planning program can represent any sequential plan without the need for goto instructions.

```
0. dec(x)
1. goto(0,!(x=1))
2. dec(y)
3. goto(2,!(y=1))
4. end
```

(a)



(b)

Fig. 2. (a) Example planning program Π for navigating to cell $(1, 1)$; (b) An execution of Π starting at cell (4,3).

We formally define two decision problems for the class of basic planning programs, which we call PP.

VAL(PP) (plan validation for basic planning programs)
INSTANCE: A planning problem $P = \langle F, A, I, G\rangle$ and a planning program $\Pi \in X$.
QUESTION: Does Π solve P?

BPE(PP) (bounded plan existence for basic planning programs)
INSTANCE: A planning problem $P = \langle F, A, I, G\rangle$ and an integer n.
QUESTION: Does there exist a planning program $\Pi \in X$ with at most n program lines that solves P?

We proceed to prove that the complexity of both decision problems is PSPACE-complete.

Theorem 1. VAL(PP) *is* PSPACE-*complete.*

Proof. Membership: Simply use the execution model to check whether a given planning program Π solves a planning problem P. To store the program state (s, i) we need $|F| + \log n$ space. Processing an instruction and testing for failure conditions 1 and 2 can be easily done in polynomial time and space. To check whether execution enters into an infinite loop, we can maintain a count of the number of instructions processed. If this count exceeds $2^{|F|}n$ without reaching the `end` instruction, this means that at least one program state has been repeated, in which case we stop and report failure. To store the count we also need $|F| + \log n$ space, which is polynomial in P and Π.

Hardness: We adapt Bylander's reduction from polynomial-space deterministic Turing machine (DTM) acceptance to plan existence for classical planning [44]. Given a DTM M with a tape of fixed size K, the idea is to define a planning frame $\Phi_M = \langle F, A\rangle$ such that the set F contains fluents derived from the following predicates:

- $\mathsf{at}(j, q)$: Is M currently at tape position j and state q?
- $\mathsf{in}(j, \sigma)$: Is σ the symbol in tape position j of M?
- accept: Does M accept on a given input?

We define a single action `simulate` with empty precondition and one conditional effect per transition of M. In other words, $A = \{\texttt{simulate}\}$. Each conditional effect of `simulate` is on the following form:

$$\{\mathsf{at}(j, q), \mathsf{in}(j, \sigma)\} \rhd \{\neg\mathsf{at}(j, q), \neg\mathsf{in}(j, \sigma), \mathsf{at}(j', q'), \mathsf{in}(j, \sigma')\}.$$

When M is at tape position j and state q and σ is the symbol in j, the transition replaces σ with σ' and moves to tape position $j' \in \{j - 1, j + 1\}$ and state q'. If, instead, M accepts the current configuration, the conditional effect becomes

$$\{\mathsf{at}(j, q), \mathsf{in}(j, \sigma)\} \rhd \{\mathsf{accept}\}.$$

Given the planning frame Φ_M and an input string x, we can construct a planning problem $P^x_M = \langle F, A, I, G\rangle$ such that the initial state I initializes the tape position to 1 and the state to q_0, and encodes the input string x on the tape:

$$I = \{\mathsf{at}(1, q_0), \mathsf{in}(0, \#), \mathsf{in}(1, x_1), \ldots, \mathsf{in}(k, x_k), \mathsf{in}(k + 1, \#), \ldots, \mathsf{in}(K, \#)\},$$

where # is the blank symbol, k is the length of the input string x and K is the fixed size of the tape. The goal condition is always defined as $G = \{\mathsf{accept}\}$.

We now construct the following planning program Π_M with two program lines:

```
0. simulate
1. goto(0,!accept)
2. end
```

Because of the conditional effects of `simulate`, lines 0 and 1 constitute a loop that repeatedly simulates a transition of M starting from the initial state. This loop only terminates if M eventually accepts, in which case the goal state G is trivially satisfied when execution terminates on line 2. Hence, for any input string x, the planning program Π_M solves P^x_M if and

only if M accepts on input x. Since the size of P_M^x and Π_M is polynomial in the size of M, we have produced a reduction from DTM acceptance to VAL(PP). Since the former is a PSPACE-complete decision problem, this proves that VAL(PP) is PSPACE-hard. □

Theorem 2. BPE(PP) *is* PSPACE*-complete for* $n \geq 2$.

Proof. Membership: Non-deterministically guess a planning program Π with n program lines. Due to Theorem 1, validating whether Π solves P is in PSPACE. Hence the overall procedure is in NPSPACE = PSPACE.

Hardness: Given a DTM M and an input string x, consider the planning problem P_M^x from the proof of Theorem 1, whose size is polynomial in M. There exists a planning program with 2 program lines, namely Π_M, that solves P_M^x if and only if M accepts on input x. Hence we have reduced DTM acceptance to BPE(PP) for $n = 2$, implying that the latter is PSPACE-hard. □

4.3. Computing basic planning programs

In this section we describe an approach to automatically compute basic planning programs. The idea is to define a compilation that takes as input a generalized planning problem $\mathcal{P}$ and a number of program lines n and outputs a classical planning problem P_n. We later prove that the compilation is sound and complete, such that any solution π to P_n can be transformed into a planning program $\Pi = \langle w_0, \ldots, w_n \rangle$ that solves $\mathcal{P}$. The intuition behind the compilation is to extend a given planning frame $\langle F, A \rangle$ with new fluents for encoding the instructions on the program lines of the planning program, as well as the program state (s, i). With respect to the actions, the compilation replaces the actions in A with:

- *Programming actions* that program an instruction (*action*, *conditional goto* or *termination*) on a given program line. Only empty lines can be programmed and initially all the program lines are empty.
- *Execution actions* that implement the execution model described in Section 4, thereby updating the program state. To execute an instruction on a program line, the instruction has to be programmed first. However, it is not necessary to program all instructions before executing: rather, programming and execution are interleaved, and an instruction is only programmed on demand, i.e. upon reaching an empty program line for the first time.

For simplicity, we first define the compilation for a single planning problem, and later extend it to generalized planning. Given a planning problem $P = \langle F, A, I, G \rangle$ and a number of program lines n, the output of the compilation is a classical planning problem $P_n = \langle F_n, A_n, I_n, G_n \rangle$. The idea is to define P_n such that any plan π that solves P_n induces a planning program $\Pi = \langle w_0, \ldots, w_n \rangle$ that solves P.

To specify P_n we have to introduce prior notation. Let $F_{pc} = \{\mathsf{pc}_i : 0 \leq i \leq n\}$ be the fluents encoding the program counter and let $F_{ins} = \{\mathsf{ins}_{i,w} : 0 \leq i \leq n, w \in A \cup \mathcal{I}_{go} \cup \{\mathsf{nil}, \mathsf{end}\}\}$ be the fluents encoding that instruction w was programmed on line i. Here, nil denotes an empty instruction, indicating that a line has not yet been programmed. Finally, let done be a fluent modeling that we are done programming and executing the planning program.

Let $w \in \mathcal{I}$ be an instruction in the *instruction set* $\mathcal{I} = A \cup \mathcal{I}_{go} \cup \{\mathsf{end}\}$. For each program line i, $0 \leq i \leq n$, we define w_i as a planning action in the set A_n that executes instruction w on line i. In doing so, we disallow instructions other than end on the last line n, and we disallow end on the first line 0.

- For each $a \in A$, let a_i, $0 \leq i < n$, be a classical planning action with precondition $\mathsf{pre}(a_i) = \mathsf{pre}(a) \cup \{\mathsf{pc}_i\}$ and conditional effects $\mathsf{cond}(a_i) = \mathsf{cond}(a) \cup \{\emptyset \rhd \{\neg \mathsf{pc}_i, \mathsf{pc}_{i+1}\}\}$.
- For each goto instruction $goto(i', !f) \in \mathcal{I}_{go}$, let $\mathsf{go}_i^{i',f}$, $0 \leq i < n$, be a classical action defined as

$$\mathsf{pre}(\mathsf{go}_i^{i',f}) = \{\mathsf{pc}_i\},$$
$$\mathsf{cond}(\mathsf{go}_i^{i',f}) = \{\emptyset \rhd \{\neg \mathsf{pc}_i\}, \{\neg f\} \rhd \{\mathsf{pc}_{i'}\}, \{f\} \rhd \{\mathsf{pc}_{i+1}\}\}.$$

- Let end_i, $0 < i \leq n$, be a classical action defined as $\mathsf{pre}(\mathsf{end}_i) = G \cup \{\mathsf{pc}_i\}$ and $\mathsf{cond}(\mathsf{end}_i) = \{\emptyset \rhd \{\mathsf{done}\}\}$, corresponding to the termination instruction.

Since w may be executed multiple times, we define two versions: $\mathsf{P}(w_i)$, that is only applicable on an empty line i and programs w on that line, and $\mathsf{R}(w_i)$, that is only applicable when instruction w already appears on line i and repeats the execution of w:

$$\mathsf{pre}(\mathsf{P}(w_i)) = \mathsf{pre}(w_i) \cup \{\mathsf{ins}_{i,\mathsf{nil}}\},$$
$$\mathsf{cond}(\mathsf{P}(w_i)) = \{\emptyset \rhd \{\neg \mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,w}\}\},$$
$$\mathsf{pre}(\mathsf{R}(w_i)) = \mathsf{pre}(w_i) \cup \{\mathsf{ins}_{i,w}\},$$
$$\mathsf{cond}(\mathsf{R}(w_i)) = \mathsf{cond}(w_i).$$

In other words, $P(w_i)$ programs w_i on an empty line i, and $R(w_i)$ repeats the execution of w_i when it is already programmed on line i.

At this point we are ready to define $P_n = \langle F_n, A_n, I_n, G_n \rangle$:

- $F_n = F \cup F_{pc} \cup F_{ins} \cup \{\mathsf{done}\}$,
- $A_n = \{\mathsf{P}(a_i), \mathsf{R}(a_i) : a \in A, 0 \leq i < n\} \cup \{\mathsf{P}(\mathsf{go}_i^{i',f}), \mathsf{R}(\mathsf{go}_i^{i',f}) : goto(i', !f) \in \mathcal{I}_{go}, 0 \leq i < n\}$
  $\cup \{\mathsf{P}(\mathsf{end}_i), \mathsf{R}(\mathsf{end}_i) : 0 < i \leq n\}$,
- $I_n = I \cup \{\mathsf{ins}_{i,\mathsf{nil}} : 0 \leq i \leq n\} \cup \{\mathsf{pc}_0\}$,
- $G_n = \{\mathsf{done}\}$.

We next extend the compilation to address generalized planning problems $\mathcal{P} = \{P_1, \ldots, P_T\}$ defined over multiple planning instances. In this case the solution plan π is a sequence of actions that programs $\Pi = \langle w_0, \ldots, w_n \rangle$ and simulates the execution of the induced program Π on each of the T classical planning instances, each with a different initial state and goal condition. Specifically, the compilation starts executing the induced planning program Π on P_1 and, after validating that Π reaches G_1 starting from I_1, resets the program state to $(I_2, 0)$ (the initial state of P_2 and program line 0). This execution of Π is repeated for each planning problem P_t, $1 \leq t \leq T$, thus validating that Π solves $\mathcal{P}$.

Given a generalized planning task $\mathcal{P} = \{P_1, \ldots, P_T\}$, the output of the compilation is a classical planning task $P'_n = \langle F'_n, A'_n, I'_n, G'_n \rangle$. Since P'_n is similar to the planning task P_n presented above, we only describe the differences:

- The set of fluents $F'_n = F_n \cup F_{test}$ includes a fluent set $F_{test} = \{\mathsf{test}_t : 1 \leq t \leq T\}$ that models the active individual planning problem. Initially test_1 is true and test_t is false for $2 \leq t \leq T$, and the initial state on fluents in F is I_1, i.e. $I'_n = I_1 \cup \{\mathsf{ins}_{i,\mathsf{nil}} : 0 \leq i \leq n\} \cup \{\mathsf{pc}_0\} \cup \{\mathsf{test}_1\}$. The goal is $G'_n = \{\mathsf{done}\}$.
- The set of actions A'_n contains all actions in A_n, but redefines the actions corresponding to the termination instructions. Actions $\mathsf{end}_{t,i}$, $0 < i \leq n$ are now defined differently for each individual planning problem t, $1 \leq t \leq T$:

$$\mathsf{pre}(\mathsf{end}_{t,i}) = G_t \cup \{\mathsf{pc}_i, \mathsf{test}_t\}, t < T,$$

$$\mathsf{cond}(\mathsf{end}_{t,i}) = \{\emptyset \rhd \{\neg\mathsf{pc}_i, \mathsf{pc}_0, \neg\mathsf{test}_t, \mathsf{test}_{t+1}\}\} \cup \{\{\neg f\} \rhd \{f\} : f \in I^{t+1}\} \cup \{\{f\} \rhd \{\neg f\} : f \notin I^{t+1}\}, t < T,$$

$$\mathsf{pre}(\mathsf{end}_{T,i}) = G_T \cup \{\mathsf{pc}_i, \mathsf{test}_T\},$$

$$\mathsf{cond}(\mathsf{end}_{T,i}) = \{\emptyset \rhd \{\mathsf{done}\}\}.$$

  For $t < T$, action $\mathsf{end}_{t,i}$ is applicable when G_t and test_t hold, and the effect is resetting the program counter to pc_0, incrementing the current active test and setting fluents in F to their value in the initial state I^{t+1} of the next planning problem. Action $\mathsf{end}_{T,i}$ is defined as the previous action end_i, and is needed to achieve the goal fluent done. As before, we add actions $\mathsf{P}(\mathsf{end}_{t,i})$ and $\mathsf{R}(\mathsf{end}_{t,i})$ to the set A'_n for each t, $1 \leq t \leq T$, and i, $0 < i \leq n$.

Before proving several properties of the compilation, we highlight how programs are represented. Recall that fluents in the set F_{ins} model the instruction programmed on each line. In the initial state I'_n, each program line i is empty, represented by the fluent $\mathsf{ins}_{i,\mathsf{nil}}$. The only type of action in A'_n that has an effect on fluents in F_{ins} is the programming action $P(w_i)$. Because of the precondition $\mathsf{ins}_{i,\mathsf{nil}}$ of $P(w_i)$, we can only program an instruction w on an empty line i. Once programmed, there is no action in A'_n that deletes instruction w from line i.

The execution of a planning program on a generalized planning problem is simulated using actions of type $R(w_i)$. Because of the precondition $\mathsf{ins}_{i,w}$ of $R(w_i)$, we are forced to program an instruction w on line i before we can execute it. As a consequence, if we are given a plan π that solves the compiled planning problem P'_n, the subsequence of programming actions of type $P(w_i)$ uniquely determines the instructions programmed on lines, which in turn define a planning program Π. We exploit this fact when we later use our compilation to generate planning programs.

We remark that our compilation is flexible with respect to the program represented in the initial state. In particular, instead of using fluent $\mathsf{ins}_{i,\mathsf{nil}}$ to indicate that line i is initially empty, we can use a fluent $\mathsf{ins}_{i,w}$ to indicate that instruction w is already programmed on line i from the outset. In this case, instruction w can be immediately executed the first time we reach line i, and there is no need to program an instruction. This mechanism allows us to define partial programs in the initial state, and we also exploit this property of the compilation in our experiments with plan validation, in which the program Π is already given.

We proceed to formally prove that the compilation is sound and complete and provide a bound on its size.

Theorem 3 *(Soundness). Any plan π that solves P'_n induces a planning program $\Pi = \langle w_0, \ldots, w_n \rangle$ that solves $\mathcal{P}$.*

Proof. As already argued, the subsequence of programming actions of type $P(w_i)$ that appear in the plan π induces a planning program Π. The only way to achieve the goal fluent done is to execute the termination instruction $\mathsf{end}_{T,i}$. Hence

the last instruction programmed has to be a termination instruction, ensuring that the induced planning program Π is well-defined. (Note that Π might have less than n program lines since we could program the termination instruction on a line i satisfying $0 < i < n$.)

The remainder of the proof follows from observing that the repeat actions $R(w_i)$ precisely implement the execution model for basic planning programs. Executing an action instruction a in program state (s, i) has the effect of updating the planning state as $s' = \theta(s, a)$ and incrementing i. Executing a goto action $\mathsf{go}_i^{i',f}$ in (s, i) has the effect of jumping to line i' if f holds in s and else increment i. Finally, executing a termination action $\mathsf{end}_{t,i}$ is only possible if the goal condition G_t holds for the current planning problem P_t, $1 \leq t \leq T$. The effect of $\mathsf{end}_{t,i}$ is to reset the program state to $(I_{t+1}, 0)$, i.e. the initial state of the next planning problem P_{t+1} and program line 0.

As detailed above, the execution of a basic planning program Π is a deterministic process that fails to solve a generalized planning problem $\mathcal{P}$ only under three conditions. If the plan π is generated via our compilation, none of the three conditions hold: the precondition of each action instruction has to hold during execution, the goal condition is checked once execution terminates, and infinite loops would prevent the plan π from solving the compiled planning problem P'_n. Execution starts on the first individual classical planning problem $P_1 \in \mathcal{P}$ and finishes when the last problem P_T has been solved. The only way to achieve this condition is by programming the instructions of Π and iteratively validating that the program solves all the problems in $\mathcal{P}$, iteratively switching from one problem $P_t \in \mathcal{P}$ to the next. Switching is only possible when G_t, the goal condition of problem P_t, holds. Hence for π to solve P'_n, the simulated execution of the induced planning program Π has to solve each problem P_t, $1 \leq t \leq T$, i.e. Π solves $\mathcal{P}$. □

Theorem 4 *(Completeness). If there exists a planning program Π that solves $\mathcal{P}$ such that $|\Pi| \leq n$, there exists a corresponding plan π that solves P'_n.*

Proof. We construct a plan π as follows. Whenever we are on an empty line i, we program the instruction specified by Π. Otherwise we repeat execution of the instruction already programmed on line i. The plan π constructed this way has the effect of programming Π and simulating the execution of Π on each planning problem in $\mathcal{P}$. Since Π solves $\mathcal{P}$, Π solves each individual problem P_t, $1 \leq t \leq T$, and hence the goal condition G_t is satisfied after simulating the execution of Π on P_t. This implies that the plan π solves P'_n. □

Note that the compilation is not complete in the sense that the bound n on the number of program lines may be too small to accommodate a planning program Π that solves $\mathcal{P}$. In many domains a bounded program can only solve a generalized planning task if a high-level state representation is available that accurately discriminates among states. For instance, the program in Fig. 2 cannot be computed if $n < 4$. Larger values of n do not formally affect to the completeness of our approach but they do affect its practical performance since classical planners are sensitive to the input size.

Theorem 5 *(Size). Given a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ on a planning frame $\Phi = \langle F, A \rangle$ and a bound n on the number of program lines, the size of the compiled problem $P'_n = \langle F'_n, A'_n, I'_n, G'_n \rangle$ is given by $|F'_n| = O(n|A| + n^2|F| + T)$ and $|A'_n| = O(n|A| + n^2|F| + nT)$.*

Proof. By inspection of the fluent set F'_n and the action set A'_n. The set F'_n is composed of F, F_{pc}, F_{ins}, F_{test} and the single fluent done. The size of F_{pc} is $n+1$, the size of F_{test} is T, while F_{ins} contains $n+1$ copies of each action in A, goto instruction in $\mathcal{I}_{go}$, empty marker nil and end instruction end. Hence the size of F_{ins} is $(n+1)(|A| + (n+1)|F| + 2) = O(n|A| + n^2|F|)$, which dominates the sizes of F and F_{pc}. The action set A'_n defines one action per instruction in $\mathcal{I}$, including T copies of the end instruction, for a total of $|A| + (n+1)|F| + T$. There are n copies of each such instruction, one per program line, and two versions that program and repeat an instruction on a given line, for a total of $2n(|A| + (n+1)|F| + T) = O(n|A| + n^2|F| + nT)$. □

Note that the number of goto instructions is the dominant term, growing as $O(n^2|F|)$. We now introduce an optimization that reduces the number of goto instructions from $O(n^2|F|)$ to $O(n|F| + n^2)$. The idea is to split actions of type $\mathsf{goto}_i^{i',f}$ into two actions: eval_i^f, that evaluates condition f on line i, and $\mathsf{jmp}_i^{i'}$, that performs the conditional jump according to the evaluation outcome. This is inspired by assembly languages that separate comparison instructions that modify flags registers, e.g., `CMP` and `TEST` in the *x86 assembly* language, from jump instructions that update the program counter according to these flag registers, e.g., `JZ` and `JNZ` in *x86 assembly*.

To implement the split we introduce two new fluents acc and eval, initially false. Fluent acc records the outcome of the evaluation, while eval indicates that the evaluation has been performed. Actions eval_i^f and $\mathsf{jmp}_i^{i'}$ are defined as

$$\mathsf{pre}(\mathsf{eval}_i^f) = \{\mathsf{pc}_i, \neg\mathsf{eval}\},$$
$$\mathsf{cond}(\mathsf{eval}_i^f) = \{\{f\} \rhd \{\mathsf{acc}\}\} \cup \{\emptyset \rhd \{\mathsf{eval}\}\},$$

Table 1
Plan generation for Planning Programs. Program lines and number of used instances; fluents and actions; search, preprocess and total time (in seconds) elapsed while computing the solution.

	Lines	Instances	Fluents	Actions	Search(s)	Preprocess(s)	Total(s)
Find	4	3	671	1044	274.20	0.66	274.86
Reverse	4	2	666	1041	86.96	0.92	87.86
Select	4	4	1028	1688	178.94	25.26	204.20
Triangular	3	2	323	324	0.38	0.47	0.85

$$\mathsf{pre}(\mathsf{jmp}_i^{i'}) = \{\mathsf{pc}_i, \mathsf{eval}\},$$
$$\mathsf{cond}(\mathsf{jmp}_i^{i'}) = \{\emptyset \rhd \{\neg\mathsf{pc}_i, \neg\mathsf{eval}\}, \{\neg\mathsf{acc}\} \rhd \{\mathsf{pc}_{i'}\}, \{\mathsf{acc}\} \rhd \{\mathsf{pc}_{i+1}, \neg\mathsf{acc}\}\}.$$

Likewise, we can replace each fluent $\mathsf{ins}_{i,w} \in F_{ins}$ which indicates that a goto instruction $w = goto(i', !f)$ has been programmed on line i by two fluents $\mathsf{ins}_{i,f}$ and $\mathsf{ins}_{i,i'}$, where the former indicates that f is the condition of the goto instruction on line i, and the latter indicates that we should jump to line i' if f is false. As a result of the optimization, the size of the planning problem P'_n becomes $|F'_n| = O(n(|A| + |F| + n) + T)$ and $|A'_n| = O(n(|A| + |F| + n + T))$.

4.4. Experiments

We perform two sets of experiments for planning programs. In the first set of experiments we take as input a generalized planning problem $\mathcal{P}$, and use the compilation P'_n to automatically generate a planning program Π, with at most n lines, that solves $\mathcal{P}$. In the second set of experiments we take as input a planning problem P and a planning program Π, and determine whether Π solves P. Thus the two sets of experiments roughly correspond to the two decision problems BPE(PP) and VAL(PP), although plan generation goes beyond plan existence in that we actually produce the planning program Π that solves the instance (or set of instances). In all experiments we use the *Automated Programming Framework* (APF)[1] to compute the compilation P'_n.

For *plan generation* we use the classical planner Fast Downward [45] (FD) in the Lama-2011 setting [46] on a Intel Core i7 2.60 GHz x 4 processor with a 4 GB memory bound and a time limit of 3600 s. For *validation* we use FD in the same setting and a Breadth First Search (BrFS) planner from the *Lightweight Automated Planning ToolKiT* (LAPKT) [47] for all the experiments.

We evaluate our approach in the following generalized planning tasks: *Find*, *Reverse*, *Select* and *Triangular*. In **Find** we must count the number of occurrences of a specific element in a vector. In **Reverse** we have to reverse the content of a vector. In **Select**, given a vector of integers we have to search for the minimum element and corresponding index. In **Triangular** the aim is to compute the triangular number $\sum_{n=1}^{N} n$ for a given integer N.

In previous work [10], the above domains were defined and fine-tuned independently. However, when a set of domains share many features, they can be defined on a common planning frame $\Phi = \langle F, A \rangle$. In this work, we defined a PDDL domain called *Pointers* that represents a vector with pointers pointing to certain elements, and actions that *increment* or *decrement* pointers and that *swap* the values of two pointers. This common planning frame allows us to represent all instances of the three generalized planning tasks Find, Reverse and Select. Defining a common planning frame in this way is similar to programming, in which the set of statements is fixed for different problems.

Table 1 reports the number of **lines** required to generate a planning program, and the number of **instances** of the generalized planning problem $\mathcal{P}$ provided as input, where each instance may test a corner case. Then, we use a classical planner to solve the compiled planning instance P'_n. Heuristic search planners usually involve a preprocessing step where fluents and actions are generated. We use FD in the LAMA-2011 setting as a classical planner, and report the number of fluents and actions as an estimate of the complexity of the output planning problem P'_n. The resulting plan π induces a planning program Π that solves the generalized planning task, and we report the **search**, **preprocess** and **total** time.

In previous work [10] the reported planning times for *Find*, *Reverse* and *Select* were shorter. The reason is that defining the three domains on a common planning frame results in less optimized planning instances, since there may be additional fluents and actions that are not relevant for the particular problem. These extra operators that allow us to model more planning problems using the same domain at the same time cause the solution time to increase.

Fig. 3 shows the obtained planning programs, which are the same as in our previous work. In Π^{find}, pointer a initially points to the head of the vector, while counter c equals 0. Lines 2–3 use a to iterate over all elements, while lines 0–1 increment c whenever the content of a equals the element we are looking for. In $\Pi^{reverse}$, a initially points to the head and b to the tail of the vector. The program repeatedly swaps the contents of a and b and move a and b towards the middle of the vector. In Π^{select}, a and b initially point to the head, and again, lines 2–3 use a to iterate over all elements. Whenever the content of a is less than that of b, line 1 assigns a to b, effectively storing the minimum element in b. In $\Pi^{triangular}$, y initially stores the integer N, and the program stores the result $\sum_{n=1}^{N} n$ in x.

In a second set of experiments we validate the planning programs in Fig. 3 on a set of larger instances. Because the planning programs are given as input, we directly assign them to the initial state of P'_n. In *Find*, *Reverse* and *Select*, we

[1] The public APF repository is at the following URL: https://github.com/aig-upf/automated-programming-framework.

```
0. goto(2,!(found(a)))
1. inc(c)
2. inc-pointer(a)
3. goto(0,!(eq(a,tail)))
4. end
```
(a)

```
0. swap(*a,*b)
1. inc-pointer(a)
2. dec-pointer(b)
3. goto(0,!(lt(b,a)))
4. end
```
(b)

```
0. goto(2,!(lt(*a,*b)))
1. assign(b,a)
2. inc-pointer(a)
3. goto(0,!(eq(a,tail)))
4. end
```
(c)

```
0. add(x,y)
1. dec(y)
2. goto(0,!(eq(y,0))
3. end
```
(d)

Fig. 3. Illustration of the generated programs. (a) Π^{find} for counting the number of occurrences of an element in a vector; (b) $\Pi^{reverse}$ for reversing a vector; (c) Π^{select} for selecting the minimum element of a vector; (d) $\Pi^{triangular}$ for computing $\sum_{n=1}^{N} n$.

Table 2
Generalized plan validation. In Compiled Tests, we compute the fluents, actions, expanded nodes and total time (in seconds) to obtain a plan for FD and BrFS. In Classical Tests, we compute the fluents, actions, expanded nodes and time taken by FD and BrFS to solve the instance without using the planning program.

	Compiled tests				**Classical tests**			
	F	*A*	**Exp/T (FD)**	**Exp/T (Brfs)**	*F*	*A*	**Exp/T (FD)**	**Exp/T (Brfs)**
Find	118	9	161/1.06	220/1.05	427	5	-/TE	-/TE
Reverse	404	7	-/TE	97/2.91	394	4	-/TE	-/TE
Select	119	9	154/7.65	213/1.05	264	4	-/TE	-/TE
Triangular	251	6	-/TE	76/70.66	242	5	-/TE	4719/92.70

tested the planning programs on vectors of size 30, significantly larger than those used as input for plan generation. In *Triangular*, the aim was to compute the sum of the first 15 natural numbers. Apart from validating each planning program using the two planners FD and BrFS (compiled tests), we also compare the total number of expanded states and time taken for each planner to compute the solution from scratch without using the planning program (classical tests).

The results of the second set of experiments are shown in Table 2. For Compiled Tests, BrFS performs always faster than FD and capable of solving every compiled instance. In several cases, and mostly for Classical Tests, the planners were unable to compute a solution within the given time limit (TE = Time-Exceeded) showing how helpful DCK is.

Even though validation only involves the deterministic execution of a program on a given instance, the preprocessing step of FD often struggles to generate the corresponding fluents and actions. In particular, this happens when the number of objects of the input planning instance is very large. For instance, the *Find* domain is validated on an instance with 31 objects corresponding to values and indices of a vector, and the total time of FD is dominated by preprocessing, while search is extremely fast. This is the reason that FD fails to validate many input instances. The BrFS setting of the LAPKT planner performs a simpler preprocessing step, so the total time is always smaller than that of FD.

4.5. Generalization ability of the compilation

In this section we use the same benchmarks as for basic planning programs: *Find*, *Reverse*, *Select* and *Triangular*. We analyze the generalization ability of the compilation in two experiments. In the first experiment, we assign the same instances for program generation as in previous experiments and iteratively increment the bounds of the compilation. In the second experiment, we provide the correct bounds for which a generalized plan can be found but the input instances are randomly generated. Both experiments give an insight into how well the compilation generalizes to different parameters and inputs. We used the FD system in the LAMA-2011 setting for these experiments, and a time bound of 600 seconds for generating a planning program and 600 seconds more to validate every instance. Finally we analyze FD on different heuristics to check how informative they are for generating planning programs.

We describe the methodology that we use in order to generate and validate programs. Our framework requires a configuration file that specifies the compilation type (basic or extension), the domain, the set of instances for generation, the set of instances for validation and the required bounds (number of lines, size of the stack, maximum execution time, etc.). Often, for a planning program to generalize it is necessary to define appropriate derived predicates and provide just the right amount of informative input instances. For instance, a grid domain where the problem is to reach the rightmost cell can be captured in diverse settings. One option is to include a derived predicated that is true in the goal cell (similar to looping on observations), another option is to provide two instances of different row lengths to avoid converging on a constant row length, etc.

In Table 3 we show the results of the first experiment. Every row has a bounded number of lines and stack size that are used to check when FD finds a solution. In this case the only human input is the set of instances, while bounds are incremented automatically. For each combination of lines and stack size we report the total time that FD takes to solve the problem. NSF means no solution was found, and TE means that preprocessing did not finish within the alloted time bound. In case a program is generated, we validate it over a random set of instances as a metric of generalization. There are four subcolumns F (Find), R (Reverse), S (Select) and T (Triangular) for each column of Instances, Total Time and Validation, indicating the different domains. Results in bold indicate the best solution obtained for each domain. We remark that for the best solutions found, failure to solve some instances was due to timeout in the preprocessing step, not due to an incorrect solution.

Table 3
Plan generation for different bounds in Find (**F**), Reverse (**R**), Select (**S**) and Triangular (**T**) domains using FD in the LAMA-2011 setting. Program lines, stack size and number of instances used; total time (in seconds including preprocessing and search) elapsed while computing the solution for each domain. In case the planning program is correctly generated, the last column shows the validation over multiple instances. We report here No-Solution-Found (NSF) when the planner explores the state space without finding a solution or if there is no solution within the time bound. Also we use Time-Exceeded (TE) when the preprocessing did not finish within the time bound.

Lines	Stack	Instances				Total time(s)				Validation			
		F	R	S	T	F	R	S	T	F	R	S	T
2	1	3	2	4	2	NSF	NSF	NSF	NSF	0/40	0/40	0/40	0/6
2	5	3	2	4	2	NSF	NSF	NSF	NSF	0/40	0/40	0/40	0/6
2	10	3	2	4	2	NSF	NSF	NSF	NSF	0/40	0/40	0/40	0/6
2	20	3	2	4	2	NSF	NSF	NSF	NSF	0/40	0/40	0/40	0/6
3	1	3	2	4	2	NSF	NSF	NSF	**1.27**	0/40	0/40	0/40	**4/6**
3	5	3	2	4	2	NSF	NSF	NSF	4.39	0/40	0/40	0/40	**4/6**
3	10	3	2	4	2	NSF	NSF	NSF	8.86	0/40	0/40	0/40	**4/6**
3	20	3	2	4	2	NSF	NSF	TE	17.76	0/40	0/40	0/40	**4/6**
4	1	3	2	4	2	**245.96**	90.58	**199.57**	1.49	**27/40**	**20/40**	**35/40**	**4/6**
4	5	3	2	4	2	NSF	376.62	NSF	5.40	0/40	**20/40**	0/40	**4/6**
4	10	3	2	4	2	NSF	NSF	NSF	10.63	0/40	0/40	0/40	**4/6**
4	20	3	2	4	2	NSF	NSF	TE	20.94	0/40	0/40	0/40	**4/6**
5	1	3	2	4	2	570.36	**66.13**	NSF	1.97	2/40	**20/40**	0/40	**4/6**
5	5	3	2	4	2	NSF	290.10	NSF	7.15	0/40	**20/40**	0/40	**4/6**
5	10	3	2	4	2	NSF	30.20	NSF	13.38	0/40	5/40	0/40	**4/6**
5	20	3	2	4	2	NSF	71.55	TE	27.43	0/40	5/40	0/40	**4/6**
6	1	3	2	4	2	NSF	188.90	NSF	3	0/40	**20/40**	0/40	**4/6**
6	5	3	2	4	2	NSF	12.46	NSF	12.40	0/40	5/40	0/40	**4/6**
6	10	3	2	4	2	NSF	24.90	TE	26.53	0/40	5/40	0/40	**4/6**
6	20	3	2	4	2	NSF	49.22	TE	42.31	0/40	5/40	0/40	**4/6**
7	1	3	2	4	2	NSF	4.66	NSF	5.56	0/40	5/40	0/40	**4/6**
7	5	3	2	4	2	NSF	17.50	NSF	21.24	0/40	5/40	0/40	**4/6**
7	10	3	2	4	2	NSF	34.01	TE	41.06	0/40	5/40	0/40	**4/6**
7	20	3	2	4	2	NSF	73.74	TE	84.23	0/40	5/40	0/40	**4/6**
8	1	3	2	4	2	100.9	5.98	NSF	7.36	3/40	5/40	0/40	**4/6**
8	5	3	2	4	2	78.20	20.10	NSF	28.39	3/40	5/40	0/40	**4/6**
8	10	3	2	4	2	NSF	47.31	TE	55.61	0/40	5/40	0/40	**4/6**
8	20	3	2	4	2	NSF	97.97	TE	116.26	0/40	5/40	0/40	**4/6**
9	1	3	2	4	2	NSF	6.91	NSF	10.38	0/40	5/40	0/40	**4/6**
9	5	3	2	4	2	NSF	27.88	NSF	40.65	0/40	5/40	0/40	**4/6**
9	10	3	2	4	2	NSF	48.65	TE	81.49	0/40	5/40	0/40	**4/6**
9	20	3	2	4	2	NSF	100.40	TE	NSF	0/40	5/40	0/40	0/6
10	1	3	2	4	2	121.37	7.76	NSF	15.6	3/40	5/40	0/40	2/6
10	5	3	2	4	2	NSF	28.78	NSF	64.04	0/40	5/40	0/40	2/6
10	10	3	2	4	2	NSF	56.30	TE	320.81	0/40	5/40	0/40	2/6
10	20	3	2	4	2	NSF	116.31	TE	NSF	0/40	5/40	0/40	0/6

From Table 3 we can conclude that when the bounds are too small, it is impossible for the planner to find a solution. On the other hand, when the bounds are too large, the planner often does not find a solution, either because preprocessing takes too long, or because search cannot handle the larger state space. In addition, when the bounds are too large, the planner often finds a program with more lines that does not generalize well to other instances. It is clear that the compilation is relatively sensitive to the bounds, but the selection of bounds can still be automated by iterating over different combinations of the bounds.

In the second experiment, instances are randomly generated while the number of lines and stack size are fixed. In each row we specify the number of instances that are chosen from the full set of randomly generated instances. Then we run four independent experiments, each with a different set of instances. Table 4 reports the best program found among the four experiments for each row. We evaluate programs according to how many instances they solve in the validation phase, and as a tiebreaker we use the preprocessing and search time.

Since previous experiments established the best possible validation performance of generated programs, we can compare the obtained solutions to the best results in Table 3. In the Find domain, the best validation outcome is produced with an input of 4 random instances, while the results in Table 3 were obtained using only three instances, and the resulting program does not generalize as well, solving 15 instances compared to 27. In Reverse, the best validation is with just one instance, but generalization is again worse, solving 17 instances compared to 20. In Select and Triangular, the best solution achieves the same validation measure as in Table 3 (35 and 4, respectively), but Select needs 7 input instances, more than the 4 instances used in Table 3. Again, we can conclude that the compilation is relatively sensitive to the nature and number of input instances.

Finally, we ran experiments with different heuristics to find out whether some heuristics are better at searching solutions to our planning problems compilation. Since the structure of the output planning problems is similar for all domains, we

Table 4
Plan generation with random instances in Find (**F**), Reverse (**R**), Select (**S**) and Triangular (**T**) using FD in the LAMA-2011 setting. For each setting we ran four random experiments with different input instances, reporting the program lines, stack size, and total time (in seconds). For each setting we choose the best result among the four randomly generated inputs in terms of the validation showed in the last column. We report No-Solution-Found (NSF) when the planner explores the state space or exceeds the time bound, and Time-Exceeded (TE) when the preprocessing step exceeds the time bound.

Instances	Stack	Lines				Total time(s)				Validation			
		F	R	S	T	F	R	S	T	F	R	S	T
1	1	4	4	4	3	0.99	**40.47**	3.96	0.19	3/40	**17/40**	2/40	1/6
2	1	4	4	4	3	0.61	48.4	NSF	**0.96**	4/40	15/40	0/40	**4/6**
3	1	4	4	4	3	416.27	48.03	TE	TE	6/40	15/40	0/40	0/6
4	1	4	4	4	3	**20.94**	TE	NSF	TE	**15/40**	0/40	0/40	0/6
5	1	4	4	4	3	NSF	49.23	TE	TE	0/40	15/40	0/40	0/6
6	1	4	4	4	3	NSF	TE	TE	TE	0/40	0/40	0/40	0/6
7	1	4	4	4	3	NSF	TE	**423.04**	TE	0/40	0/40	**35/40**	0/6
8	1	4	4	4	3	NSF	99.25	TE	TE	0/40	15/40	0/40	0/6
9	1	4	4	4	3	NSF	TE	TE	TE	0/40	0/40	0/40	0/6
10	1	4	4	4	3	NSF	TE	TE	TE	0/40	0/40	0/40	0/6

Table 5
Heuristics evaluations for planning programs generation in Find, Reverse, Select and Triangular domains. The columns indicate the number of expanded, evaluated and generated nodes during the search phase. We report the plan size, preprocessing time and search time (in seconds). There are cases where heuristics do not help to find the planning program reporting Time-Exceeded (TE) or Memory-Exceeded (ME) that corresponds to slow exploration and fast generation of nodes. The results with best computational total time for each domain are marked in bold.

Domain	Heuristic	Expanded	Evaluated	Generated	Plan length	Preprocessing(s)	Search(s)
Find	Additive	–	–	–	–	0.66	ME
	Blind	–	–	–	–	0.71	ME
	Causal graph	–	–	–	–	0.72	ME
	Context-enhanced	–	–	–	–	0.67	ME
	FF	**139703**	**526586**	**605448**	**51**	**0.71**	**30.96**
	Goal count	–	–	–	–	0.75	ME
	LAMA-2011	298514	2558305	5970288	51	0.66	274.20
	Landmark count	–	–	–	–	0.70	ME
Reverse	Additive	–	–	–	–	1.03	ME
	Blind	–	–	–	–	1.03	ME
	Causal graph	1846988	1846989	3108541	22	0.92	20.88
	Context-enhanced	–	–	–	–	1.01	TE
	FF	–	–	–	–	0.92	TE
	Goal count	–	–	–	–	0.98	ME
	LAMA-2011	226342	557364	7095513	22	0.92	86.96
	Landmark count	**2785250**	**2785251**	**4507445**	**22**	**1.05**	**19.52**
Select	Additive	–	–	–	–	31.36	TE
	Blind	–	–	–	–	28.94	ME
	Causal graph	3931314	3931315	7291302	73	31.58	235.86
	Context-enhanced	–	–	–	–	28.47	TE
	FF	185522	3282971	3435140	73	25.1	604.48
	Goal count	–	–	–	–	29.12	ME
	LAMA-2011	152195	434383	6279369	73	25.26	178.94
	Landmark count	**10643057**	**10643058**	**21145893**	**73**	**28.20**	**99.44**
Triangular	**Additive**	**46**	**62**	**314**	**26**	**0.30**	**0.02**
	Blind	266183	266184	400632	26	0.28	1.60
	Causal graph	540	541	987	26	0.32	0.04
	Context-enhanced	**46**	**62**	**314**	**26**	**0.30**	**0.02**
	FF	46	62	314	26	0.40	0.12
	Goal count	266183	266184	400632	26	0.28	1.18
	LAMA-2011	2872	5744	8406	26	0.47	0.38
	Landmark count	75906	75907	139145	26	0.66	0.36

believe that this is an indicative of how heuristics handle these problems. The outcome of this experiment appears in Table 5 and we use the same setting as in Table 1, but only one heuristic is allowed for each generation phase.

The heuristic evaluation shown in Table 5 is about the performance on generation of basic planning programs for different domains like Find, Reverse, Select and Triangular. The results show that delete-free relaxation is only helpful for easy problems like Triangular but becomes unfeasible when problems have deeper plan lengths yielding in most of the cases either to Time-Exceeded or Memory-Exceeded. We have to remark that *Landmark count* has shown a great computational time performance even in complex domains like Select, but the main limitation using this heuristic is memory because of its fast generation rate. Thus, LAMA-2011 setting is not the one with best performance but has shown great adaptability to every domain balancing exploration and generation rates, being the only one capable of generating every single planning program.

5. Planning programs with procedures

In this section we extend basic planning programs with *procedures*. Intuitively, a procedure is itself a planning program. For procedures to interact, we introduce *call instructions* that allow procedures to call other procedures. The section also introduces two other extensions to basic planning programs: *variables* and *procedural arguments*. We prove that the theoretical properties of planning programs remain the same when procedures are allowed, and extend the compilation for basic planning programs to include procedures.

5.1. Extending planning programs with procedures

Given a classical planning frame $\Phi = \langle F, A \rangle$, a *planning program with procedures* is a pair $\Pi = \langle \Theta, L \rangle$ where:

- $\Theta = \{\Pi^0, \ldots, \Pi^m\}$ is a set of planning programs defined in $\Phi = \langle F, A \rangle$.
- $L \subseteq F$ is a subset of *local fluents*.

The instruction set $\mathcal{I}$ is extended to include *procedure calls*

$$\mathcal{I} = A \cup \mathcal{I}_{go} \cup \{\texttt{end}\} \cup \mathcal{I}_{call}, \quad \mathcal{I}_{call} = \{\texttt{call(j')} : 0 \leq j' \leq m\},$$

where $\mathcal{I}_{call}$ is the set of *call instructions*, allowing any procedure to call any other procedure on an arbitrary program line. We adopt the convention of designating procedure Π^0 as the *main program*, and $\{\Pi^1, \ldots, \Pi^m\}$ as the set of *auxiliary procedures*. We define $|\Pi| = |\Pi^0| + \cdots + |\Pi^m|$, i.e. $|\Pi|$ is the total number of program lines of all procedures.

To define the execution model for planning programs with procedures we first introduce the notion of a *call stack* that keeps track of where control should return when the execution of a procedure terminates. In addition each procedure has a *local state* defined on the set of local fluents L. Given L, each state s can be partitioned as $s = s_g \cup s_l$, where $s_g = s|_{F \setminus L}$ is the *global state* and $s_l = s|_L$ is the *local state*. Each element of the call stack is a tuple (j, i, s_l), where j is an index that refers to a procedure Π^j, $0 \leq j \leq m$, i is a program line, $0 \leq i \leq |\Pi^j|$, and s_l is a local state. In what follows we use $\Sigma \oplus (j, i, s_l)$ to denote a call stack recursively defined by a call stack Σ and a top element (j, i, s_l). To ensure that the execution model remains bounded we impose an upper bound ℓ on the size of the call stack.

The execution model for a planning program with procedures consists of a *program state* (s_g, Σ), where s_g is a global state and Σ is a call stack. Given a program state $(s_g, \Sigma \oplus (j, i, s_l))$, the execution of instruction w_i^j on line i of procedure Π^j is defined as follows:

- If $w_i^j \in A$, the new program state is $(s'|_{F \setminus L}, \Sigma \oplus (j, i+1, s'|_L))$, where $s' = \theta(s_g \cup s_l, w_i^j)$ is the state resulting from applying action w_i^j in state $s = s_g \cup s_l$ and $s'|_{F \setminus L}$ and $s'|_L$ are the corresponding global and local states. Just as in the execution model for basic programs, the program line i is incremented.
- If $w_i^j = \texttt{goto(i', !f)}$, the new program state is $(s_g, \Sigma \oplus (j, i+1, s_l))$ if $f \in s_g \cup s_l$ and $(s_g, \Sigma \oplus (j, i', s_l))$ otherwise. The only effect is changing the program line, and a jump only occurs if f is false, like in the execution model for basic programs.
- If $w_i^j = \texttt{call(j')}$, the new program state is $(s_g, \Sigma \oplus (j, i+1, s_l) \oplus (j', 0, \emptyset))$. In other words, calling a procedure $\Pi^{j'}$ has the effect of (1) incrementing the program line i at the top of the stack; and (2) pushing a new element onto the call stack to start the execution of the new procedure $\Pi^{j'}$ on line 0 with an empty local state.
- If $w_i^j = \texttt{end}$, the new program state is (s_g, Σ), i.e. a termination instruction has the effect of terminating a procedure by popping element (j, i, s_l) from the top of the call stack. The execution of a planning program with procedures does not necessarily terminate when executing an end instruction. Instead, execution terminates when the call stack becomes empty, i.e. in program state $(s_g, \emptyset)$.

To execute a planning program with procedures Π on a planning problem $P = \langle F, A, I, G \rangle$, we set the initial program state to $(I|_{F \setminus L}, (0, 0, I|_L))$, i.e. the initial state of P is partitioned into global and local states and execution is initially on program line 0 of the main program Π^0. We assume that the goal condition G is completely defined on the set of global fluents $F \setminus L$ and we say that Π *solves* P if and only if execution terminates and the goal condition holds in the resulting program state, i.e. $(s_g, \emptyset) \wedge G \subseteq s_g$. As a consequence of the bound ℓ on the size of the call stack, there is now a fourth reason why a planning program with procedures may fail to solve a generalized planning problem:

4. Execution does not terminate because, when executing a call instruction `call(j')` in program state (s_g, Σ), the size of Σ equals ℓ, i.e. $|\Sigma| = \ell$.

Executing such a procedure call would result in a call stack whose size exceeds the upper bound ℓ, i.e. a *stack overflow*. The extended execution model is still *deterministic*, so a planning program with procedures can again be viewed as a form of compact reactive plan for the *class* of planning problems defined by the corresponding generalized planning task.

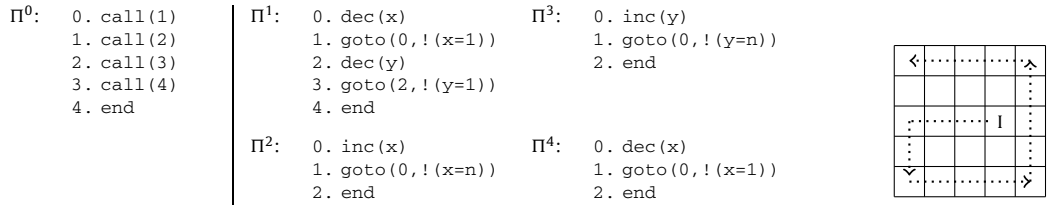


Fig. 4. Planning program with procedures $\langle\{\Pi^0, \ldots, \Pi^4\}, \emptyset\rangle$ for visiting the four corners of an $n \times n$ grid (Π^1 is also defined in Fig. 2) and an execution example of the program in a 5×5 grid starting from cell $(4, 3)$.

Fig. 4(a) shows an example planning program with procedures $\langle\{\Pi^0, \ldots, \Pi^4\}, \emptyset\rangle$ for visiting the four corners of an $n \times n$ grid starting from any initial position in the grid. Variables x and y that represent the agent position in the grid are global, and there are no local fluents, i.e. $s|_{F \setminus L} = s$ and $s|_L = \emptyset$ for every state s. Procedure Π^1 refers to the basic planning program defined in Fig. 2, Π^2 is a procedure for reaching the last column of an $n \times n$ grid, Π^3 is a procedure for reaching the last row of an $n \times n$ grid and Π^4 is a procedure for reaching the first column.

To allow for arbitrary grid sizes, we define derived fluents $x = n$ and $y = n$ whose values are true if the agent is currently in the last column or row, respectively. The PDDL definition in Appendix A illustrates this idea: fluent `max-value(x,v5)` indicates that the maximum value in the domain of variable x is 5, and the derived fluent `is-max(x)` is true precisely when the current assignment to x equals the maximum value. The conditional effect of action `inc(x)` does not trigger if `is-max(x)` is true, i.e. if the value of x is already maximal. Fig. 4(b) shows an example execution of the program on a planning problem whose initial state places the agent at position $(4, 3)$.

5.2. Planning programs with procedural arguments

In this section we take advantage of variable representation to extend procedure calls with *arguments*. Procedural arguments make it possible to reduce the size of programs and to represent compact plans for tasks that demand recursive solutions.

Defining actions on variables is mostly a matter of problem representation. As already mentioned in Section 3, we assume that the fluents of a classical planning frame $\Phi = \langle F, A\rangle$ are instantiated from a set of predicates Ψ and a set of objects Ω. We now introduce the additional assumption that there exists a predicate $\mathsf{assign}(v, x) \in \Psi$ and that Ω is partitioned into two sets Ω_v (the *variable objects*) and Ω_x (the *value objects*). Intuitively, a fluent $\mathsf{assign}(v, x)$, $v \in \Omega_v$ and $x \in \Omega_x$, is true if and only if x is the value currently assigned to the variable v. A given variable represents exactly one value at a time, so for a given v, fluents $\mathsf{assign}(v, x)$, $x \in \Omega_x$, are *mutex invariants* (only one is true at any moment). All other predicates in Ψ are instantiated on value objects in Ω_x only.

Let $K = \{\mathsf{assign}(v, x) : v \in \Omega_v, x \in \Omega_x\}$ be the subset of fluents induced by the predicate assign. Given a planning program with procedures $\Pi = \langle\Theta, L\rangle$, we assume that $K \subseteq L$, i.e. that all fluents in K are local. Furthermore, to each procedure $\Pi^j \in \Theta$, $0 \leq j \leq m$, we associate an arity $ar(j)$ and a parameter list $\Lambda(j) \in \Omega_v^{ar(j)}$ consisting of $ar(j)$ variable objects. We also redefine the set of procedure calls as

$$\mathcal{I}_{call} = \{\texttt{call(j',}\omega\texttt{)} : 0 \leq j' \leq m, \omega \in \Omega_v^{ar(j')}\},$$

where ω is the list of $ar(j')$ variable objects passed as arguments when calling procedure $\Pi^{j'}$.

The execution model for planning programs with procedural arguments (including the conditions for *termination*, *success* and *failure*) is inherited from the execution model previously defined for planning programs with procedures. The only term that has to be redefined is the execution of a procedure call instruction with arguments:

- If $w_i^j = \texttt{call(j',}\omega\texttt{)}$ and the current program state is $(s_g, \Sigma \oplus (j, i, s_l))$, then the new program state is $(s_g, \Sigma \oplus (j, i+1, s_l) \oplus (j', 0, s_l'))$ where the local state s_l' is obtained as follows. For each value object $x \in \Omega_x$ and each z, $1 \leq z \leq ar(j')$, we set $\mathsf{assign}(\Lambda(j')_z, x)$ to true if and only if $\mathsf{assign}(\omega_z, x)$ is true in s_l. This has the effect of *copying* the value of variable ω_z onto its corresponding variable $\Lambda(j')_z$ in the parameter list of procedure $\Pi^{j'}$.

Fig. 5 shows a planning program with two parameterized procedures, $\Pi^1(aux)$ and $\Pi^2(aux)$, for visiting the four corners of an $n \times n$ grid, the same navigation problem introduced in Fig. 4. In this particular example the set of *variable objects* is $\Omega_v = \{x, y, aux\}$ while the set of *value objects* depends on the size of the grid, e.g. for a 5×5 grid $\Omega_x = \{1, 2, 3, 4, 5\}$. Both auxiliary procedures have one argument, i.e. their arity is 1. Moreover, both have the same parameter list $\Lambda(\Pi^1) = \Lambda(\Pi^2) = [aux]$. Since each stack level has a separate fluent set, variables in the parameter lists can be reused for different procedures and procedure calls, like in Fig. 5 where the variable named *aux* is used for the two different auxiliary procedures $\Pi^1(aux)$ and $\Pi^2(aux)$. Compared to the program of Fig. 4, this new program with procedural arguments is significantly more compact.

```
Π⁰:  0. call(1,x)        Π¹(aux): 0. dec(aux)
     1. call(1,y)                 1. goto(0,!(aux=1))
     2. call(2,x)                 2. end
     3. call(2,y)
     4. call(1,x)        Π²(aux): 0. inc(aux)
     5. end                       1. goto(0,!(aux=n))
                                  2. end
```

Fig. 5. Planning program with two parameterized procedures for visiting the four corners of an $n \times n$ grid starting from any initial cell. Procedure $\Pi^1(aux)$ decrements variable *aux* until reaching value 1 while $\Pi^2(aux)$ increments *aux* until reaching value n.

5.3. Theoretical properties of planning programs with procedures

In this section we prove that plan validation and plan existence are still PSPACE-complete for planning programs with procedures. Hence including procedures in planning programs does not increase the worst-case complexity of the related decision problems, as long as we bound the size of the call stack.

We extend the two decision problems from Section 4 to the class PP-P of planning programs with procedures.

VAL(PP-P) (plan validation for planning programs with procedures)
INSTANCE: A planning problem $P = \langle F, A, I, G \rangle$, a planning program $\Pi \in X$ and an integer ℓ.
QUESTION: Does Π solve P using a call stack of size no more than ℓ?

BPE(PP-P) (bounded plan existence for planning programs with procedures)
INSTANCE: A planning problem $P = \langle F, A, I, G \rangle$ and integers ℓ, m and n.
QUESTION: Does there exist a planning program $\Pi \in X$ with at most m procedures and n program lines that solves P using a call stack of size no more than ℓ?

Theorem 6. VAL(PP-P) *is* PSPACE*-complete.*

Proof. Membership: Similar to the proof for basic planning programs, we can use the execution model to check whether a planning program with procedures Π solves a planning problem P using a call stack of size no more than ℓ. To store a program state (s_g, Σ) we need $|F| - |L| + \ell(\log m + \log n + |L|)$ space: $|F| - |L|$ space to store the global state $s_g \in F \setminus L$, and $\log m + \log n + |L|$ space to store each element (j, i, s_l) of the call stack, with a maximum of ℓ such elements. Processing an instruction and testing preconditions and the goal condition can be done in polynomial time and space. Testing whether we exceed the size of the call stack is also trivial given the current program state and instruction. To check whether execution enters into an infinite loop, we can maintain a count of the number of instructions processed, and report failure if this count exceeds the total number $2^{|F|-|L|} + (mn2^{|L|})^{\ell}$ of possible program states. Maintaining this count also requires $|F| - |L| + \ell(\log m + \log n + |L|)$ space, which is polynomial in P, Π and ℓ.

Hardness: We show that there is a polynomial-time reduction from VAL(PP) to VAL(PP-P). Given a planning problem $P = \langle F, A, I, G \rangle$ and a basic planning program Π, construct a planning program with procedures $\Pi' = \langle \{\Pi\}, F \rangle$, i.e. Π' has a single procedure that acts as the main program and is equal to Π, and the set of local fluents is $L = F$. Since Π is a basic planning program it contains no recursive calls to itself, so a call stack of size $\ell = 1$ is sufficient to execute Π'. Since Π is the main program of Π', Π' solves P if and only if Π solves P. Since VAL(PP) is PSPACE-complete due to Theorem 1, this implies that VAL(PP-P) is PSPACE-hard. □

Theorem 7. BPE(PP-P) *is* PSPACE*-complete for* $\ell \geq 1$, $m \geq 1$ *and* $n \geq 2$.

Proof. Membership: Non-deterministically guess a planning program with procedures Π with m procedures and n program lines. Due to Theorem 6, validating whether Π solves P using a call stack of size at most ℓ is in PSPACE. Hence the overall procedure is in NPSPACE = PSPACE.

Hardness: By reduction from BPE(PP). Namely, there exists a basic planning program Π with n program lines that solves P if and only if there exists a planning program with procedures $\Pi' = \langle \{\Pi\}, F \rangle$ that solves P using a call stack of size $\ell = 1$, where Π' has $m = 1$ procedures. Since BPE(PP) is PSPACE-complete for $n \geq 2$, this implies that BPE(PP) is PSPACE-hard for $\ell \geq 1$, $m \geq 1$ and $n \geq 2$. □

We remark that although there is no complexity theoretic benefit of extending planning programs with procedures, in practice decomposing a planning program into procedures often results in more compact solutions easier to be computed by planners. Procedures also make it possible to decompose a problem into sub-problems, compute a separate planning program for each sub-problem and reuse them at different problems.

5.4. Computing programs with procedures

In this section we extend the compilation from Section 4 with procedures, implementing the procedure call mechanism with a finite-size *stack* that can be modeled in PDDL and that is inspired by the compilation of *fault tolerant planning* into classical planning [48]. Our finite-size stack is a pair $\langle L, \ell \rangle$ where $L \subseteq F$ is the subset of *local fluents*, i.e., fluents that can be allocated in the stack, and ℓ is the maximum number of levels in the stack. Implicitly this stack model defines:

- A set of fluents $L^\ell = \{f^l : f \in L, 1 \leq l \leq \ell\}$ that contains replicas of the fluents in L parameterized with the stack level l. These fluents represent the ℓ partial states that can be stored in the stack.
- A set of fluents $F^\ell_{top} = \{\mathsf{top}^l\}_{0 \leq l \leq \ell}$ representing which is the top level of the stack at the current time.
- Actions push_Q and pop that are the canonical stack operations, with push_Q pushing a subset of stackable fluents $Q \subseteq L$ to the top level of the stack and pop popping any fluent in L from the top level of the stack.

To compute programs with callable procedures we extend our compilation with new local fluents representing 1) the current procedure; 2) the current program line of the procedure; and 3) the local state of the procedure. We also add new actions that implement programming and execution of *procedure call* instructions as well as *termination* instructions for the procedures. Intuitively the execution of a *procedure call* instruction pushes onto the stack the current procedure, the program line and the local state. Likewise the execution of a *termination* instruction pops all this information from the stack.

As a first step we detail the compilation for computing programs with procedures without arguments, and then we explain the extension of the compilation to deal with procedural arguments. Formally the new compilation takes as input a generalized planning problem $\mathcal{P} = \{\langle F, A, I_1, G_1\rangle, \ldots, \langle F, A, I_T, G_T\rangle\}$ and three bounds n, m and ℓ and outputs a classical planning problem $P^\ell_{n,m} = \langle F^\ell_{n,m}, A^\ell_{n,m}, I^\ell_{n,m}, G^\ell_{n,m}\rangle$. Here, n bounds the number of lines for each procedure, m bounds the number of procedures and ℓ bounds the stack size.

Given the planning problem P_t, $1 \leq t \leq T$, let $I_{t,g} = I_t \cap (F \setminus L)$ be the initial global state of P_t, and let $I^1_{t,l} = \{f^1 : f \in I_t \cap L\}$ be the initial local state of P_t encoded on level $l = 1$ of the stack. The planning problem $P^\ell_{n,m}$ is defined as follows:

- $F^\ell_{n,m} = (F \setminus L) \cup L^\ell \cup F^\ell_{top} \cup F^\ell_{pc} \cup F^m_{ins} \cup F_{test} \cup \{\mathsf{done}\}$ where
  - F^ℓ_{pc} contains the local fluents for indicating the current line and procedure executed. Formally, $F^\ell_{pc} = \{\mathsf{pc}^l_i : 0 \leq i \leq n, 1 \leq l \leq \ell\} \cup \{\mathsf{proc}^l_j : 0 \leq j \leq m, 1 \leq l \leq \ell\}$.
  - F^m_{ins} encodes the instructions of the main and auxiliary procedures. In other words, the same fluents F_{ins} defined for the previous compilation but parametrized with the procedure id, plus new fluents that encode *call instructions*: $F^m_{ins} = \{\mathsf{ins}_{i,j,w} : 0 \leq i \leq n, 0 \leq j \leq m, w \in A \cup \mathcal{I}_{go} \cup \mathcal{I}_{call} \cup \{\mathsf{nil}, \mathsf{end}\}\}$.
- The initial state sets all the program lines (`main` and auxiliary procedures) as empty and sets the procedure on stack level 1 to 0 (the `main` procedure) with the program counter pointing to the first line of that procedure. The initial state on fluents in F is I_1, hence $I^\ell_{n,m} = I_{1,g} \cup I^1_{1,l} \cup \{\mathsf{ins}_{i,j,\mathsf{nil}} : 0 \leq i \leq n, 0 \leq j \leq m\} \cup \{\mathsf{top}^1, \mathsf{pc}^1_0, \mathsf{proc}^1_0\}$. As before, the goal condition is defined as $G^\ell_{n,m} = \{\mathsf{done}\}$.
- The actions are defined as follows:
  - For each instruction $w \in A \cup \mathcal{I}_{go}$, an action $w^l_{i,j}$ parameterized not only on the program line i but also on the procedure j and stack level l. Let w^l_i be the corresponding action defined in Section 4.3 with superscript l added to all program counters and local fluents. Then $w^l_{i,j}$ is defined as

$$\mathsf{pre}(w^l_{i,j}) = \mathsf{pre}(w^l_i) \cup \{\mathsf{top}^l, \mathsf{proc}^l_j\},$$
$$\mathsf{cond}(w^l_{i,j}) = \mathsf{cond}(w^l_i).$$

    Note that these actions do not alter the call stack.
  - For each call instruction `call(j')` $\in \mathcal{I}_{call}$, an action $\mathsf{call}^{j',l}_{i,j}$ also parameterized on i, j and l, $1 \leq l < \ell$:

$$\mathsf{pre}(\mathsf{call}^{j',l}_{i,j}) = \{\mathsf{pc}^l_i, \mathsf{ins}_{i,j,call(j')}, \mathsf{top}^l, \mathsf{proc}^l_j\}, \tag{1}$$
$$\mathsf{cond}(\mathsf{call}^{j',l}_{i,j}) = \{\emptyset \rhd \{\neg\mathsf{pc}^l_i, \mathsf{pc}^l_{i+1}, \neg\mathsf{top}^l, \mathsf{top}^{l+1}, \mathsf{pc}^{l+1}_0, \mathsf{proc}^{l+1}_{j'}\}\}. \tag{2}$$

    Note that the effect is to push a new program line $(j', 0)$ onto the stack. Also note that $j = j'$ implies a recursive call.
  - An action $\mathsf{end}^{l+1}_{i,j}$ that simulates the termination on line i of procedure j on stack level $l+1$, $0 \leq l < \ell$:

$$\mathsf{pre}(\mathsf{end}^{l+1}_{i,j}) = \{\mathsf{pc}^{l+1}_i, \mathsf{ins}_{i,j,end}, \mathsf{top}^{l+1}, \mathsf{proc}^{l+1}_j\},$$
$$\mathsf{cond}(\mathsf{end}^{l+1}_{i,j}) = \{\emptyset \rhd \{\neg\mathsf{pc}^{l+1}_i, \neg\mathsf{top}^{l+1}, \neg\mathsf{proc}^{l+1}_j, \mathsf{top}^l\}, \emptyset \rhd \{\neg f^{l+1} : f \in L\}\}.$$

    Note that the effect is to pop the program line (j, i) from the stack, deleting all local fluents.

- As before, the action set $A^{\ell}_{n,m}$ is composed of the program action $P(w^l_{i,j})$ and repeat action $R(w^l_{i,j})$ for each action $w^l_{i,j}$ defined above.
- For each planning problem P_t, $1 \leq t \leq T$, we also need a termination action term_t that simulates the successful termination of the planning program on P_t when the stack is empty:

$$\begin{aligned}
\mathsf{pre}(\mathsf{term}_t) &= G_t \cup \{\mathsf{top}^0\}, t < T,\\
\mathsf{cond}(\mathsf{term}_t) &= \{\emptyset \rhd I^1_{t+1,l} \cup \{\neg\mathsf{top}^0, \mathsf{top}^1, \mathsf{pc}^1_0, \mathsf{proc}^1_0\}\} \cup \{\{\neg f\} \rhd \{f\} : f \in I_{t+1,g}\}\\
&\quad \cup \{\{f\} \rhd \{\neg f\} : f \notin I_{t+1,g}\}, t < T,\\
\mathsf{pre}(\mathsf{term}_T) &= G_T \cup \{\mathsf{top}^0\},\\
\mathsf{cond}(\mathsf{term}_T) &= \{\emptyset \rhd \{\mathsf{done}\}\}.
\end{aligned}$$

  Note that the effect of term_t, $t < T$, is to reset the program state to the initial state of problem P_{t+1}.

Now we define the extension to our compilation for programming and executing *parameterized calls to procedures*. Apart from the program counter and the current procedure, a *procedure call* with arguments also pushes onto the stack the arguments of the call. Formally the classical planning task $P^{\ell}_{n,m}$ that results from the compilation is extended as follows:

- We assume that there exists a new set of local fluents $\{\mathsf{assign}(v, x) : v \in \Omega_v, x \in \Omega_x\} \subseteq L$ that encode assignments of type $v = x$.
- The actions for programming a call instruction are redefined to indicate not only the called procedure but also the specific values passed to that procedure. To define the actions that execute a call to procedure j' passing a list of parameters we use the actions $\mathsf{call}^{j',l}_{i,j}$ defined in Equations (1) and (2). For each variable combination $\Gamma(j') \in \Omega_v^{ar(j')}$, we introduce a new action $\mathsf{call}^{j',l}_{i,j}\Gamma(j')$ formulated as:

$$\begin{aligned}
\mathsf{pre}(\mathsf{call}^{j',l}_{i,j}\Gamma(j')) &= \mathsf{pre}(\mathsf{call}^{j',l}_{i,j}),\\
\mathsf{cond}(\mathsf{call}^{j',l}_{i,j}\Gamma(j')) &= \mathsf{cond}(\mathsf{call}^{j',l}_{i,j})\\
&\quad \cup \{\{\mathsf{assign}^l(v_q, x)\} \rhd \{\mathsf{assign}^{l+1}(u_q, x)\} : v_q \in \Gamma(j'), x \in \Omega_x, u_q \in \Lambda(j'), 1 \leq q \leq |\Lambda(j')|\}.
\end{aligned}$$

  In other words, $\mathsf{call}^{j',l}_{i,j}\Gamma(j')$ has the effect of *copying* the value of each variable $v_q \in \Gamma(j')$ on level l of the stack to the corresponding local variable $u_q \in \Lambda(j')$ on level $l+1$ of the stack. Recall that $\Lambda(j') \in \Omega_v^{ar(j')}$ is the parameter list of procedure j consisting of $ar(j')$ variable objects.

We formally prove several properties of the extended compilation for planning programs with procedures. In particular, we show that the compilation is sound and complete and prove a bound on the compilation size.

Theorem 8 *(Soundness). Any plan π that solves $P^{\ell}_{n,m}$ induces a planning program with procedures Π that solves $\mathcal{P}$.*

Proof. The proof is very similar to the proof of Theorem 3. Whenever the current program line of a procedure is empty, π has to program an instruction on that line, else repeat execution of the instruction already programmed on that line. Hence the fluent set F^m_{ins} implicitly induces a planning program with procedures Π.

Although the execution model is more complicated than for basic planning programs, the repeat actions of $P^{\ell}_{n,m}$ precisely implement the execution model for planning programs with procedures. Hence the plan π has the effect of simulating the execution of Π on each planning problem in $\mathcal{P}$. To solve $P^{\ell}_{n,m}$, the goal condition G_t has to hold for each problem $P_t \in \mathcal{P}$, proving that Π solves $\mathcal{P}$. □

Theorem 9 *(Completeness). If there exists a planning program with procedures Π that solves $\mathcal{P}$ such that 1) Π contains at most m auxiliary procedures; 2) each procedure of Π contains at most n program lines; and 3) executing Π on the planning problems in $\mathcal{P}$ does not require a call stack whose size exceeds ℓ, then there exists a plan π that solves $P^{\ell}_{n,m}$.*

Proof. Construct a plan π by always programming the instruction indicated by Π, and repeatedly simulate the execution of Π on the planning problems in $\mathcal{P}$, terminating when the stack becomes empty. Since Π solves $\mathcal{P}$ and fits within the given bounds, the plan π constructed this way is guaranteed to solve $P^{\ell}_{n,m}$. □

The extended compilation adds a new source of incompleteness. The bound ℓ on the stack size limits the depth of nested procedure calls which can also make $P^{\ell}_{n,m}$ unsolvable. For example, a program that implements the recursive version of DFS,

needs at least $\ell \geq 3$ stack levels for solving the problem of visiting all the nodes of a tree with depth 3 without causing a stack overflow.

Theorem 10 *(Size). Given a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ on a planning frame $\Phi = \langle F, A \rangle$, a subset $L \subseteq F$ of local fluents and bounds ℓ, m and n, the size of the compiled problem $P^{\ell}_{n,m} = \langle F^{\ell}_{n,m}, A^{\ell}_{n,m}, I^{\ell}_{n,m}, G^{\ell}_{n,m} \rangle$ is given by $|F^{\ell}_{n,m}| = O(\ell(|L| + m + n) + mn(|A| + |F| + m + n) + T)$ and $|A^{\ell}_{n,m}| = O(\ell mn(|A| + |F| + m + n + T))$.*

Proof. By inspection of the fluent set $F^{\ell}_{n,m}$ and the action set $A^{\ell}_{n,m}$. The set $F^{\ell}_{n,m}$ is defined as $(F \setminus L) \cup L^{\ell} \cup F^{\ell}_{top} \cup F^{\ell}_{pc} \cup F^{m}_{ins} \cup F_{test} \cup \{\mathsf{done}\}$. The collective size of $L^{\ell} \cup F^{\ell}_{top} \cup F^{\ell}_{pc}$ equals $(\ell + 1)(|L| + 1 + n + 1 + m + 1) = O(\ell(|L| + m + n))$. The size of F^{m}_{ins} equals $O(mn(|A| + |F| + m + n))$, where the term $m^2 n$ is due to call instructions and we have used the optimization for goto instructions from Section 4. The action set $A^{\ell}_{n,m}$ defines the same instructions as before, plus call instructions, for a total of $|A| + |F| + n + 1 + T + m + 1$. There are ℓmn copies of each such instruction, one per stack level, procedure and program line, and two versions that program and repeat an instruction, for a total of $2\ell mn(|A| + |F| + n + 1 + T + m + 1) = O(\ell mn(|A| + |F| + m + n + T))$. □

Compared to the compilation for basic planning programs, the number of actions increases approximately by a factor of ℓm, which is to be expected since actions are parameterized on the procedure and stack level.

The benefit of extending the compilation with procedure calls comes from:

1. Representing solutions compactly using procedural arguments and recursion. An example is our program generated for DFS traversing binary trees whose depth does not exceed the size of the call stack. A classical plan for this task consists of an action sequence whose length is linear in the number of tree nodes, and hence exponential in the depth of the tree. In contrast, a recursive definition of Depth-First Search (DFS) only requires a 6-line program, as reported in the experiments below.
2. Reusing existing programs as auxiliary procedures. If auxiliary procedures are provided, the only instructions that a plan has to program are those of the main program Π^0. Reducing the number of empty program lines decreases the number of possible planning programs and hence the number of applicable actions. In the experiments below we computed the planning program of Fig. 4 given the four auxiliary procedures Π^1, Π^2, Π^3 and Π^4.

The final benefit of including a given auxiliary procedure is contingent on how much the reused procedure contributes to solving the overall problem. A key issue for effectively reusing existing programs as auxiliary procedures is how to generate auxiliary procedures that are helpful to solve a given generalized planning task. One option is for a domain expert to hand-craft auxiliary procedures [8], and this might be the best choice if such knowledge is readily available. However, since each procedure is indeed a program of its own, we can use our compilation to compute each of these program from examples (albeit without auxiliary procedures). For instance, to compute the auxiliary procedure Π^1 for navigating to the $(0, 0)$ position (see Fig. 2), we defined a generalized planning problem $\mathcal{P}^1$ by generating individual planning problems with different initial states whose goal condition is to be at position $(0, 0)$. We then used the compilation from Section 4 to generate a basic planning program Π^1 that solves $\mathcal{P}^1$. Similarly, we generated programs Π^2, Π^3, Π^4 for reaching the other three corners of a grid. We then designated these programs as procedures $\Pi^1, \ldots, \Pi^4$ of a planning program with procedures.

To incrementally compute and reuse auxiliary procedures we need to assume the existence of a specific decomposition of the overall problem into a set of subtasks, and appropriately extend the definition of a generalized planning problem. A similar assumption is done in Hierarchical Goal Network (HGN) planning where a network of subgoals is specified to boost the search of a hierarchical planner [49]. An interesting open research direction is to automatically discover these decompositions of planning problems and previous work on the automatic generation of planning hierarchies [50,51] is a good starting point to address this research question.

5.5. Experiments

We perform two sets of experiments for *planning programs with procedures*, corresponding to plan generation and plan validation. Most domains used in the experiments are the same as in our previous work [10]. In *GreenBlock*, the aim is to find and collect a green block in a tower of blocks. In *Fibonacci*, the aim is to compute the *n*-th number in the Fibonacci sequence. In *Gripper*, the aim is to move all balls from one room to the next. In *Hall-A*, the aim is to visit the four corners of a grid (cf. Fig. 5). In *Sorting*, the aim is to sort the elements of a vector. In *Trees*, the aim is to visit all the nodes of a given binary tree. In *Visit-all*, the aim is to visit all cells of a grid. Finally, in *Visual-M*, the aim is to find a marked block in a configuration of multiple towers of blocks.

We also introduce a novel domain called *Excel*, inspired by the *Flash Fill* feature of Microsoft Excel to automatically program macros from examples. There are five problems in this domain: 1) add a parenthesis at the end of a word; 2) extract the number of seconds from a timer in the MM:SS.HH format (minutes, seconds and hundredths of a second); 3) given name and surname strings, form a single string formatted as surname, space and name; 4) given name and surname strings, form

Table 6
Plan generation for planning programs with procedures. Number of procedures; for each procedure: program lines, instances, stack size, fluents, actions, search time, preprocessing time; total time (in seconds) elapsed while computing the overall solution.

	Proc	Lines	Inst	Stack	Fluents	Actions	Search(s)	Preprocess(s)	Total(s)
GreenBlock	2	(4,3)	(6,5)	(2,2)	(306,250)	(932,559)	(0.90,0.04)	(2.63,0.53)	4.10
Excel-1	2	(3,2)	(2,1)	(2,2)	(4167,4265)	(11517,11838)	(3.16,0.53)	(2.92,3.14)	9.75
Excel-2	2	(3,2)	(2,1)	(2,2)	(4167,3282)	(11517,6122)	(3.23,0.24)	(2.92,5.53)	11.92
Excel-3	2	(3,3)	(2,1)	(2,2)	(4167,7301)	(11517,20538)	(3.09,5.29)	(2.92,8.64)	19.94
Excel-4	2	(3,3)	(2,1)	(2,2)	(4167,7301)	(11517,20538)	(3.13,328.41)	(2.83,8.54)	342.91
Excel-5	2	(3,3)	(2,1)	(2,2)	(4167,7301)	(11517,20538)	(3.09,0.64)	(2.94,8.63)	15.30
Fibonacci	2	(3,3)	(2,4)	(2,2)	(321,341)	(579,607)	(1.01,1.48)	(1.03,1.42)	4.94
Gripper	3	(3,3,3)	(2,2,2)	(2,2,2)	(305,307,403)	(651,665,859)	(0.05,0.04,1.06)	(0.34,0.38,0.62)	2.49
Hall-A	5	(5,5, 5,5,4)	(2,2, 2,2,2)	(2,2, 2,2,2)	(1029,1041, 1053,1065,888)	(3925,3951, 3977,4003,2624)	(86.77,69.18,100.97, 350.23,455.73)	(1.06,1.16, 1.32,1.43,1.27)	1069.12
Sorting	2	(4,4)	(4,3)	(2,2)	(556,549)	(1988,1779)	(11.56,11.55)	(1.94,3.86)	28.91
Trees	1	6	1	4	638	4164	154.76	1.09	155.85
Visit-all	3	(3,2,4)	(2,2,2)	(2,2,2)	(801,582,816)	(1911,879,2569)	(0.22,0.04,24.70)	(0.54,0.38,0.79)	26.67
Visual-M	3	(4,4,4)	(4,2,5)	(2,2,2)	(274,238,279)	(724,649,650)	(0.38,0.03,5.90)	(2.22,0.42,3.30)	12.25

```
0. call(1)
1. call(2)
2. goto(0,!(no-balls-in-rooma))
3. end
```

(a) Π^0: pick up and drop balls until none left

```
0. pick-left
1. pick-right
2. move
3. end
```

(b) Π^1: pick up balls

```
0. drop-left
1. drop-right
2. move
3. end
```

(c) Π^2: drop balls

Fig. 6. Program with procedures for the *Gripper*.

a string with the name, space and the first letter of the surname; and 5) given name and surname strings, form a string with first letter of the name, a space and the first letter of the surname.

Again we can represent different generalized planning tasks using the same planning frame $\Phi = \langle F, A \rangle$. For example, instances of *Triangular* and *Fibonacci* are represented using the *Variables* PDDL domain that include operators to increment or decrement variables, add the value of a variable to another, and assign the value of a variable to another. Likewise, the instances of the *Hall-A* and *Visit-all* tasks are represented using a generic *Grid* domain that includes operators for *visiting* cells and *moving* one cell, in any cardinal direction. All *Excel* instances are also represented using the same PDDL domain.

In each domain, except for *Trees* whose solution comprises just one procedure, plan generation proceeds in several steps. We manually decompose the overall problem into two or more subproblems. For each subproblem we provide a separate generalized planning problem whose instances correspond to that subproblem. For example, in *Sorting* the subproblem is to select the minimum element from a vector, and for each instance there is a pointer that starts at the first position of the vector, and the goal condition is to bring the pointer to the minimum element. In *Trees*, the problem is not decomposed, but the resulting main program makes recursive calls to itself, so the call stack is still needed to solve the problem.

Let $\mathcal{P}^0, \ldots, \mathcal{P}^k$ be the sequence of generalized planning problems, where $\mathcal{P}^0$ corresponds to the overall problem we want to solve. For each problem $\mathcal{P}^j$ in decreasing order of j $(k, k-1, \ldots)$, we generate a planning program Π^j that solves $\mathcal{P}^j$. While doing so, the programs $\Pi^{j+1}, \ldots, \Pi^k$ are included as auxiliary procedures. In other words, only the lines of the main program are empty, and the remaining programs are encoded as part of the initial state. Hence generating the program Π^j amounts to programming the instructions of the main program, which can include calls to the auxiliary procedures, and then executing the program to ensure it solves $\mathcal{P}^j$. Also note that the number of procedures increases for each subproblem, such that Π^k has no auxiliary procedures, while Π^0 uses all the other programs $\Pi^2, \ldots, \Pi^k$ as auxiliary procedures.

Table 6 reports the plan generation results. Compared to Table 1, we added bounds on the number of procedures and stack size. Since subproblems are solved separately, each procedure corresponds to a separate call to the classical planner. For each procedure, we therefore report the number of program lines, number of instances, stack size, number of fluents, number of actions, search time, and preprocessing time. We also report the total time to solve all subproblems related to a domain. Times are reported in seconds.

As an illustration, we show four of the obtained programs. The solution to **Gripper** appears in Fig. 6. In Π^1 the agent picks up balls with both grippers and moves to the second room. In Π^2 the agent drops both balls and moves back to the first room. Π^0 makes repeated calls to Π^1 and Π^2 until there are no balls left in the first room. Note that we changed the representation of the Gripper domain in order to generalize: instead of representing individual balls, we represent the *number* of balls in each room, and the conditional effect of actions such as `pick-left` is to decrement the number of balls in the current room of the robot.

The solution to **Sorting** appears in Fig. 7 and corresponds to the *selection sort* algorithm. There are four pointers: `itermax`, pointing to the tail of the vector; `outer`, indicating the start position in every loop; `inner`, iterating from `outer` to `itermax` in each loop; and `mark`, pointing to the minimum element found so far in each loop. Procedure Π^1 repeatedly increments `inner`, and assigns `inner` to `mark` if its content is the smallest found so far. Procedure Π^0 repeatedly calls Π^1 to select the minimum element (stored in `mark`), and then swaps the contents of `outer` and `mark`. We

```
0. call(1)
1. swap( *mark, *outer )
2. inc-pointer( outer )
3. goto( 0, !( eq( inner, itermax ) ) )
4. end
```

(a) Π^0: repeatedly select minimum value and swap contents

```
0. inc-pointer( inner )
1. goto( 3, !( lt( *inner, *mark ) ) )
2. assign( mark, inner )
3. goto( 0, !( eq( inner, itermax ) ) )
4. end
```

(b) Π^1: select minimum value from current position `outer`

Fig. 7. Program with procedures for *Sorting*.

```
0. visit( current )
1. copy-left( child, current )
2. goto( 6, !( isinternal( current ) ) )
3. copy-right( current, current )
4. call( 0, child )
5. call( 0, current )
6. end
```

Fig. 8. Recursive program for *Trees*.

```
0. append-str(res,str-var)
1. inc-loindex(str-var)
2. goto(0,!(empty(str-var)))
3. end
```

(a) Π^1: copy substring of `str-var` to `res`

```
0. call(1,str-var)
1. append-char(res,')')
2. end
```

(b) $\Pi^{(0,E-1)}$: append ')' to a string

```
0. get-substr(str-var,':','.')
1. call(1,str-var)
2. end
```

(c) $\Pi^{(0,E-2)}$: get the seconds from a timer

```
0. call(1,surname-var)
1. append-char(res,' ')
2. call(1,name-var)
3. end
```

(d) $\Pi^{(0,E-3)}$: copy surname, space and name

```
0. call(1,name-var)
1. append-char(res,' ')
2. append-str(res,surname-var)
3. end
```

(e) $\Pi^{(0,E-4)}$: copy name, space and initial

```
0. append-str(res,name-var)
1. append-char(res,' ')
2. append-str(res,surname-var)
3. end
```

(f) $\Pi^{(0,E-5)}$: copy space-separated initials

Fig. 9. Planning programs for *Excel*, where Π^1 is a common procedure.

remark that the action `inc-pointer(outer)` on line 2 has the secondary effect of setting the pointer `inner` equal to `outer`; this is the reason why line 3 refers to `inner` instead of `outer`.

In **Trees**, there are two variables `current` and `child` that point to nodes of the tree. Fig. 8 shows the resulting planning program. The program first visits the current node and copies the left child of the current node to `child`. In case the current node is a leaf (i.e. not internal) execution finishes, else the right child of the current node is copied to `current`. Then the program makes two recursive calls to itself for each of the two variables `child` and `current`. In this way it visits all the tree nodes in a depth-first search fashion.

The programs for **Excel** are shown in Fig. 9. Each string has two indices `lo` and `hi`. Each of the five Excel tasks share the same procedure Π^1 which is parameterized on a string `str-var` and copies all characters of `str-var` between `lo` and `hi` to another string `res`. Program $\Pi^{(0,E-1)}$ calls Π^1 and then appends a right parenthesis. Program $\Pi^{(0,E-2)}$ selects the substring between ':' and '.' by setting the two indices appropriately before copying. Program $\Pi^{(0,E-3)}$ first copies the surname, then appends a space character, and then copies the name. Program $\Pi^{(0,E-4)}$ copies the name and then appends a space character and the first letter of the surname. Finally, program $\Pi^{(0,E-5)}$ appends the first character of the name and surname with a space character in between.

With respect to the results reported in our previous work [10], the performance in *Fibonacci* is better. Now we need one less instance to generalize, the 5th number in the Fibonacci sequence. On the contrary, the performance in *Hall-A* and *Visit-all* is worse since we implemented them on a common generic PDDL domain *Grid*.

Like in Section 4, we ran a second set of experiments in which we validated the generated programs. In *GreenBlock*, we tested the planning program on a tower of 100 blocks where the green block was the third starting from the bottom. In *Excel* we used the name "MAXIMILIAN" and surname "FEATHERSTONEHAUGH" for problems [3, 5], and the same surname in problem 1. In Problem 2 the given timer was 01:59.23. In *Fibonacci*, we tested the program on the 6th Fibonacci number. In *Gripper*, the test consisted in moving 30 balls to the next room. In *Hall-A*, the agent had to visit the four corners of a 100×100 grid. In Sorting, the test was to sort a vector of 50 random elements. In *Trees*, we tested the program on a binary tree with 20 nodes and maximum depth 8. In *Visit-all*, all cells of a 30×30 grid must be visited. Finally, in *Visual-M*, the agent had to process 10 towers with maximum height 10, with the marked block in the last tower.

Table 7 presents the plan validation results obtained with the FD and BrFS planners. The reported data include the number of fluents and actions, total number of expanded states and the total time FD and BrFS needs to compute a solution. Results are obtained for two configurations: i) the instance that encodes the given planning program in the initial state (compiled tests); and ii) the classical domain and instance without using the planning program (classical tests). This way, we can compare how hard it is to solve a given classical planning instance compared to validating a planning program on the instance (i.e. using the program as control knowledge). As before, Time-Exceeded (TE) indicates that no solution was

Table 7
Plan validation for planning programs with procedures. In Compiled Tests, we compute the fluents, actions, expanded nodes and total time (in seconds) to obtain a plan for FD and BrFS. In Classical Tests, we compute the fluents, actions, expanded nodes and time taken by FD and BrFS to solve the instance without using the planning program.

	Compiled tests				**Classical tests**			
	F	*A*	**Exp/T (FD)**	**Exp/T (Brfs)**	*F*	*A*	**Exp/T (FD)**	**Exp/T (Brfs)**
GreenBlock	421	14	401/3.40	501/3.05	404	3	198/1.63	591/1.38
Excel-1	352	10	76/0.57	94/0.57	433	56	70/0.60	-/ME
Excel-2	106	10	12/1.05	14/0.27	117	56	4/0.13	379/0.17
Excel-3	640	11	114/1.42	141/1.45	798	106	256/3.53	-/ME
Excel-4	450	11	45/1.24	55/1.31	558	106	44/1.69	/ME
Excel-5	347	4	4/0.74	4/0.81	486	106	3/1.34	164/0.38
Fibonacci	71	11	-/TE	55/0.35	56	8	-/TE	17337/0.48
Gripper	206	15	181/0.92	196/0.66	158	5	172/0.77	-/ME
Hall-A	10226	46	1210/8.65	1609/7.39	10206	5	7797/2.75	-/TE
Sorting	2820	17	-/TE	9544/36.85	2803	4	-/TE	-/TE
Trees	661	197	109/0.79	126/0.68	61	10	35/0.07	993304/8.54
Visit-all	1045	19	6511/8.13	8341/2.02	1029	5	-/TE	-/TE
Visual-M	45	22	845/0.37	1146/0.35	27	5	45/0.02	161/0.08

found within the given time limit, Memory-Exceeded (ME) for no solved problems within the given memory limit, and the hyphen symbol (-) for no reported numbers related to TE or ME.

We see that in most domains, the planners are able to quickly compute a solution even in the absence of a planning program, sometimes even faster than when a planning program is provided. The reason of being faster without the compilation is because some classical instances require less steps than compiled instances, and no matter how many objects the instance has if it can always be solved with few actions, thus the complexity on those domains is on generating programs but not on validating them. A similar situation is found when the actions to solve an instance, like in GreenBlock domain, are repetitive becoming an straight forward search in the classical instance so DCK only adds complexity. Also, FD spend all the alloted time in preprocessing for some domains like *Fibonacci* and *Sorting* where predicates with 2 or 3 arity like `(sum v1 v2 v3)` or `(lessthan v1 v2)` require to ground all value combinations, while a simple blind search is enough to get a plan. Remarkably in *Sorting* and *Visit-all* domains, no planner is able to solve the planning instance without the given planning program showing that DCK becomes important for complex domains where heuristic search is not helpful at all.

6. Non-deterministic planning programs

The planning program formalism, presented in Sections 4 and 5, unambiguously capture the *action to apply next*, for solving every instance in a given generalized planning task. Defining a planning program that is valid for multiple planning instances, while compact, may be complex and sometimes, even unfeasible (e.g. instances may not share a clear common structure and need to be treated separately).

An alternative approach is to compute solutions with non-deterministic execution, i.e. solutions with open segments that are only determined when executing the solution on a particular instance [8]. Programs with non-deterministic execution still constrain the space of possible solutions but requires a planner to produce a fully specified solution for a particular instance. This section introduces *choice instructions* and *lifted action instructions*, two different extensions of our *planning program* formalism to achieve generalization through non-deterministic execution. The section describes how to represent and compute planning solutions of this kind.

6.1. Planning programs with choice instructions

The first extension refers to *choice actions* that, inspired by `let` instructions from functional programming, assign a value to a variable in a generalized plan. Our approach closely follows that of Srivastava et al. who first introduced choice actions for generalized planning [4].

The extension is based on the formalism for *planning programs with variables* defined in Section 5: fluents F in a planning frame $\Phi = \langle F, A \rangle$ are instantiated from a set of predicates Ψ and a set of objects Ω. In addition, Ω is partitioned into *variable objects* Ω_v and *value objects* Ω_x and there exists a predicate $\mathsf{assign}(v, x) \in \Psi$ such that $v \in \Omega_v$ and $x \in \Omega_x$. With this defined, we can now extend the instruction set $\mathcal{I}$ with *choice instructions*:

$$\mathcal{I}_{choice} = \{\texttt{choose}(\omega, p) : \omega \in \Omega_v^{ar(p)}, p \in \Psi\}.$$

Given the current program state (s, i), the execution model for a choice instruction, $\texttt{choose}(\omega, p)$, is defined (for basic planning programs) as follows:

- If $w_i = \texttt{choose}(\omega, p)$, the new program state is $(s', i + 1)$, where the new state s' is constructed in two steps. The first step non-deterministically chooses $x \in \Omega_x^{ar(p)}$ such that $p(x) \in s$. The second step assigns x to ω making fluents

$\mathsf{assign}(\omega_j, x_j)$, $1 \leq j \leq |ar(p)|$ true while making $\mathsf{assign}(\omega_j, y)$ false for each value object $y \neq x_j$. Other than this assignment, s' is identical to s.

A choice instruction `choose`(ω, p) non-deterministically assigns a value object to each of the variable objects in ω. This assignment results from a *unification* of the predicate p with the current state s: given an assignment $x \in \Omega_x^{ar(p)}$ of value objects to the arguments of p such that the induced fluent $p(x)$ is true in s, x is a valid assignment to ω, which we can represent using fluents $\mathsf{assign}(\omega_j, x_j)$, $1 \leq j \leq |ar(p)|$. In the literature, the arguments of a choice action are determined by evaluating a given first order formula [4], but we restrict this formula to a single predicate (the formula could however be extended to a *conjunctive query* following the ideas in Section 7).

The execution of a *basic planning program with choice instructions* introduces a fourth failure condition:

4. Execution does not terminate because, when executing a `choose`(ω, p) instruction, predicate p cannot be unified with the current state s, i.e. there does not exist $x \in \Omega_x^{ar(p)}$ such that $p(x) \in s$.

We illustrate the idea of a *planning program with choice instructions* using the well-known Blocksworld domain. The following planning program is able to solve any Blocksworld instance for which the goal is to put every block on the table. This task is more complex than unstacking a single tower of blocks, commonly solved by FSCs [2], because here there can be an arbitrary number of towers and each with different height. The only choice instruction `0.choose(v,clear)` assigns a block object, that is currently clear, to the variable v capturing the key knowledge of *the block to move next*. This block is then unstacked and put down on the table. This process is repeated until all blocks are on the table (`all-clear` is a derived predicate that tests whether all blocks are clear, which is only possible if they are all on the table). This program assumes that the actions `unstack(v)` and `putdown(v)` are defined in terms of a variable object v rather than a block object (the latter is usually how the Blocksworld domain is modeled).

```
0. choose(v,clear)
1. unstack(v)
2. putdown(v)
3. goto(0,!(all-clear))
4. end
```

To compute *planning programs with choice actions* we extend the compilation explained in Section 4 for the computation of basic planning programs. For each choice instruction `choose`$(\omega, p) \in \mathcal{I}_{choice}$, let $\mathsf{choose}_i^{\omega,p,\chi}$ be a classical planning action where $\chi \in \Omega_x^{ar(p)}$ is a list of value objects that can be used to unify predicate p with the current state:

$$\begin{aligned}
\mathsf{pre}(\mathsf{choose}_i^{\omega,p,\chi}) =& \{\mathsf{pc}_i, p(\chi)\},\\
\mathsf{cond}(\mathsf{choose}_i^{\omega,p,\chi}) =& \{\emptyset \rhd \{\neg\mathsf{pc}_i, \mathsf{pc}_{i+1}\}\}\\
&\cup \{\emptyset \rhd \{\mathsf{assign}(v_q, \chi_q)\} : v_q \in \omega, 1 \leq q \leq |ar(p)|\}\\
&\cup \{\emptyset \rhd \{\neg\mathsf{assign}(v_q, x)\} : v_q \in \omega, x \in \Omega_x, x \neq \chi_q, 1 \leq q \leq |ar(p)|\}.
\end{aligned}$$

The compilation is extended with two versions of the previous classical planning action, $\mathsf{P}(\mathsf{choose}_i^{p,\omega,\chi})$, that is only applicable on an empty line i and programs the corresponding choice instruction on that line, and $\mathsf{R}(\mathsf{choose}_i^{p,\omega,\chi})$, that is only applicable when the choice instruction already appears on line i and repeats its execution.

6.2. Planning programs with lifted action instructions

Here we go one step further and increase the portion of a planning program that can be unspecified. A *planning program with lifted action instructions* is a planning program in which action instructions have unknown arguments until the instruction is executed on a particular planning instance.

A PDDL action scheme (also called *operators* or *lifted actions*) is parameterized on a set of arguments. Similar to how fluents are induced from predicates, a set of actions is induced from a PDDL lifted action by assigning objects to its arguments. For example, the lifted action `inc(?var)` from the grid navigation domain (Appendix A) has a single argument `?var`, so actions `inc(x)` and `inc(y)` are induced assigning objects `x` and `y` to the argument `?var`. In the following we assume that the actions of a planning frame $\Phi = \langle F, A\rangle$ are induced from a set of lifted actions $\mathcal{A}$ and a set of objects Ω. Under this assumption, the instruction of a planning program can be a lifted action $\alpha \in \mathcal{A}$. Since the arguments of a lifted action are unspecified, a lifted action instruction effectively models a non-deterministic choice.

Given the current program state (s, i), the execution model for lifted action instructions is defined as:

- If $w_i \in \mathcal{A}$, the new program state is $(s', i+1)$, where $s' = \theta(s, w_i(x))$ is the result of applying an action $w_i(x) \in A$ induced from w_i, where every argument $x \in \Omega^{ar(w_i)}$ of w_i is non-deterministically chosen such that $\mathsf{pre}(w_i(x)) \subseteq s$, i.e. such that $w_i(x)$ is applicable in s.

Table 8
Generation of non-deterministic planning programs. Number of procedures; for each procedure: number of lines and instances, stack size, number of fluents and actions, search time and preprocessing time; total time (in seconds) elapsed while computing the solution.

	Proc	Lines	Inst	Stack	Fluents	Actions	Search(s)	Preprocess(s)	Total(s)
Hall-A	5	(5,5, 5,5,4)	(2,2, 2,2,2)	2	(1029,-, -,-,-)	(4645,-, -,-,-)	(TE,TE, TE,TE,TE)	(1.22,TE, TE,TE,TE)	TE
Fibonacci	2	(3,3)	(2,4)	2	(321,341)	(2043,2209)	(0.11,0.24)	(0.46,0.63)	1.44
Visit-all	3	(3,2,4)	2	2	(585,438,616)	(1875,1038,2375)	(0.10,0.06,42.63)	(0.46,0.42,0.82)	44.49
GreenBlock	2	(4,3)	(6,5)	2	(306,250)	(1124,713)	(203.16,0.09)	(1.51,0.99)	205.75
Sorting	2	(4,4)	(4,3)	2	(540,549)	(2484,8981)	(11.99,13.91)	(3.57,3.78)	33.25
Triangular	1	3	2	1	291	588	0.02	0.29	0.31
Visual-M	3	(4,2,4)	(4,2,5)	2	(274,135,279)	(964,284,782)	(0.37,0.01,249.1)	(2.24,0.29,3.04)	12.57
Blocksworld	2	(3,4)	(3,2)	2	(214,214)	(651,651)	0.04	0.82	0.86

Additionally, the execution of a *lifted action instruction* fails if there is no possible applicable instantiations for that instruction.

Now we show a *planning program with lifted action instructions* for solving the Blocksworld task of putting all the blocks on the table. Each execution of `unstack(?b1,?b2)` non-deterministically assigns concrete block objects to parameters `?b1` and `?b2` among the possible applicable instantiations. Likewise each `putdown(?b3)` execution assigns a concrete block object to parameter `?b3`.

```
0. unstack(?b1,?b2)
1. putdown(?b3)
2. goto(0,!(all-clear))
3. end
```

We remark that the execution semantics of lifted action instructions is *angelic* [52]: not every assignment of blocks to the arguments `?b1`, `?b2` and `?b3` leads to a valid plan. Rather, it is necessary to use a planner to determine valid assignments of objects to arguments.

To compute *planning programs with lifted action* a small modification in the compilation of Section 4 for basic planning programs is required. The modification is a redefinition of the classical planning actions for programming and executing action instructions. A lifted action $w_i \in \mathcal{A}$ induces one action $w_i(x)$ for each assignment $x \in \Omega^{ar(w_i)}$ to the arguments of w_i. The actions for programming and executing a lifted action instruction are defined as follows:

$$\mathsf{pre}(\mathsf{P}(w_i(x)) = \mathsf{pre}(w_i(x)) \cup \{\mathsf{ins}_{i,\mathsf{nil}}\},$$
$$\mathsf{cond}(\mathsf{P}(w_i(x)) = \{\emptyset \rhd \{\neg\mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,w_i}\}\},$$
$$\mathsf{pre}(\mathsf{R}(w_i(x)) = \mathsf{pre}(w_i(x)) \cup \{\mathsf{ins}_{i,w_i}\},$$
$$\mathsf{cond}(\mathsf{R}(w_i(x)) = \mathsf{cond}(w_i(x)).$$

6.3. Experiments

In these experiments we use several domains from previous sections, but with a slight modification. In the new version of the domains, action instructions have parameters, so the resulting planning programs include lifted instructions. In previous experiments, because we were using parameter-free actions, the execution of planning programs was deterministic. In the new domains with parameterized instructions, the execution of a planning program requires the planner to select the values of the action parameters each time an action instruction is executed. Similar to Hierarchical Task Networks (HTNs) [53], non-deterministic planning programs require the planner to compute a fully specified solution, constraining the form of the solution by pruning the actions that do not agree with the given program. Therefore planning program with *choice instructions* or *lifted action instructions*, can be viewed as Domain-specific Control Knowledge [54,43]. However, unlike HTNs, planning program with *choice instructions* or *lifted action instructions* can be exploited straightforward with an off-the-shelf classical planner.

Apart from existing domains, we added the Blocksworld domain. Our approach is to manually decompose the problem into two subproblems, where the first subproblem is to put all blocks on the table, and the second subproblem is to stack the blocks to form towers. The goal condition for the first subproblem is that all blocks should be on the table, and the goal condition for the second subproblem is usually expressed using fluents on (A,B), on (B,C), etc. However, in our simpler version, the goal is instead to form several towers of a given height, regardless of the specific placement of blocks.

Table 8 shows the results of the experiments for computing non-deterministic planning programs. The table reports the number of procedures; for each procedure: number of program lines, number of instances, stack size, number of fluents and actions, search time and preprocessing time; the total time to solve the overall problem. Note that the planner fails to solve one subproblem for Hall-A when actions are given as lifted instructions. The programs obtained for each domain are similar to those described in Section 5. However, since action instructions are lifted, the planner has to assign objects to

```
0. call(1)
1. pick-up( ?b1 )
2. stack( ?b2, ?b3 )
3. goto( 1, !( eq( current-towers, target-towers ) ) )
4. end
```

(a) Π^0: stack blocks on others until reaching the number of target towers

```
0. unstack( ?b1, ?b2 )
1. put-down( ?b3 )
2. goto( 0, !( all-clear ) )
3. end
```

(b) Π^1: put all blocks on table

Fig. 10. Non-deterministic planning program for Blocksworld.

Table 9
Validation of non-deterministic planning programs. In Compiled Tests, we compute the fluents, actions, expanded nodes and total time (in seconds) to obtain a plan for FD and BrFS. In Classical Tests we report the obtained results without using the planning program.

	Compiled tests				**Classical tests**			
	F	*A*	**Exp/T (FD)**	**Exp/T (Brfs)**	*F*	*A*	**Exp/T (FD)**	**Exp/T (Brfs)**
Hall-A	–	–	–	–	10206	10396	822/1.79	-/ME
Fibonacci	64	482	49/0.32	55/0.32	56	654	28/0.07	15929/0.20
Visit-all	1045	1066	6511/9.55	8341/1.02	1029	1081	5176/3.29	-/ME
GreenBlock	421	20210	401/7.07	600/5.71	404	399	198/1.05	296/0.86
Sorting	258	26073	-/TE	801666/17.57	241	26060	6319/195.91	-/ME
Triangular	147	1835	61/2.13	76/1.94	242	7861	35/1.41	4353/0.34
Visual-M	45	49	872/0.35	1182/0.38	27	41	17/0.01	-/NSF
Blocksworld	10422	20210	3358/87.53	-/ME	10404	20200	65129/217.89	-/ME

action parameters and perform search to reach the goal. Fig. 10 shows the resulting planning program for the **Blocksworld** domain. Procedure Π^1 repeatedly unstacks a block from another and puts it on the table until all blocks are clear. Procedure Π^0 first calls Π^1, then repeatedly picks up a block and stacks it on top of another block, until the number of current towers equals the number of target towers.

Regarding performance, FD has to ground the lifted instructions by assigning objects to their parameters, which causes an increase in the number of operators. As a result, both the preprocessing time and the search time is generally larger than for deterministic planning programs. On the flip side, there are domains such as Blocksworld that can not be solved by deterministic planning programs, while lifted instructions make it possible.

For validation we use the same instances as in Section 5 and address the same experiment reported in Table 7, but with non-deterministic programs acting as DCK instead of deterministic planning program with procedures. For the *Blocksworld*, we used an instance with 100 blocks to provide an example of how hard is to solve large classical planning instances versus the corresponding compiled instance with a non-deterministic planning program acting as DCK. Table 9 summarizes the obtained results and reports the number of fluents and actions, and total time that FD and BrFS needed to compute a plan. Again, we tested both the compiled problem that encodes the planning program in the initial state (compiled tests), and the classical domain and instance without DCK (classical tests). Only in *Fibonacci* BrFS performs better than FD because of the required preprocessing burden. And the classical instance of Visual-M is No-Solution-Found (NSF) because the BrFS planner does not support axioms and is a requirement for the classical domain.

7. Planning programs with high-level state features

In machine learning it is generally agreed upon that *feature extraction* is key to generate models that generalize well [55]. Likewise in automated planning it is well-known that, for many tasks, generalized plans are only computable if an informative state representation is available [6]. A good example are fluents $x = n$, $y = n$ or $aux = n$ (taken from the programs in Figs. 4 and 5) that are unnecessary for solving individual grid navigation tasks, but essential to compute a solution plan that generalizes to different grid locations and grid sizes.

In most applications of generalized planning such informative state features are hand-coded by a domain expert. This section shows how to compute generalized plans that contain high-level state features in the form of *conjunctive queries* without having a prior high-level state representation. The section first provides our definition of high-level state features as conjunctive queries. Then it extends our notion of programs such that *conditional goto instructions* depend on the evaluation of a conjunctive query. The section ends extending our compilation to compute programs with conjunctive queries with an off-the-shelf classical planner.

7.1. High-level state features

The notion of *high-level state feature* is very general and has been used in different areas of AI and for different purposes. If we restrict ourselves to planning, a high-level state feature can broadly be viewed as a state abstraction to compactly represent planning tasks or solutions to planning tasks. For instance, $x = n$ abstracts the set of states where variables x and n have the same value, no matter what this particular value is. In the literature we can find diverse formalisms for representing high-level state features in planning that range from first order clauses [34] to description logic formulae [7], LTL formulae [56], PDDL derived predicates [57] and, more recently, observation formulae [2].

$$\mathsf{equal}(x, n) \leftarrow \exists v_3.\mathsf{assign}(x, v_3) \wedge \mathsf{assign}(n, v_3),$$

$$\mathsf{lt}(x, n) \leftarrow \exists v_3, v_4.\mathsf{assign}(x, v_3) \wedge \mathsf{assign}(n, v_4) \wedge \mathsf{lessthan}(v_3, v_4).$$

Fig. 11. Two derived predicates in the form of conjunctive queries that correspond to the high-level state features $x = n$ and $x < n$, respectively.

Our planning model (defined in Section 3) considered that states are represented by instantiating a set of predicates Ψ with a set of objects Ω. In this case, a high-level state feature corresponds to an arbitrary formula over predicates in Ψ. High-level state features are also known as *derived predicates* if they produce a new predicate whose truth value is determined by the corresponding formula. Derived predicates have proven useful for concisely representing planning problems with complex conditions and effects [58] and for more efficiently solving optimal planning problems [59].

For computation purposes in this work we assume that the set of possible high-level state features is restricted to *conjunctive queries* from database theory [60]. Conjunctive queries are a simple fragment of first-order logic in which formulae are constructed from atoms using conjunction and existential quantification (disallowing all other logical symbols). A conjunctive query φ can be written as

$$\varphi = (v_1, \ldots, v_k).\exists v_{k+1}, \ldots, v_m.\phi_1 \wedge \cdots \wedge \phi_q,$$

where $v_1, \ldots, v_k$ are *free variables*, $v_{k+1}, \ldots, v_m$ are *bound variables*, and $\phi_1, \ldots, \phi_q$ are *atoms*. In addition, we make the following further assumptions:

1. We adopt the model of planning programs with variables from Section 5.2, i.e. there exists a predicate $\mathsf{assign} \in \Psi$, a set of *variable objects* Ω_v and a set of *value objects* Ω_x.
2. We define a new set Ω_b of *bound variable objects* that represent the bound variables $v_{k+1}, \ldots, v_m$ of a conjunctive query φ, and let $\Omega = \Omega_v \cup \Omega_x \cup \Omega_b$ be the global set of objects.
3. Each atom of a conjunctive query is on the form $p(\omega)$, where $p \in \Psi$ is a predicate and $\omega \in \Omega^{ar(p)}$ is an argument list of appropriate arity. If $p = \mathsf{assign}$, $\omega = (\omega_1, \omega_2)$ consists of a variable object $\omega_1 \in \Omega_v$ and a bound variable object $\omega_2 \in \Omega_b$, else $\omega \in \Omega_b^{ar(p)}$ consists exclusively of bound variable objects.

Fig. 11 shows two derived predicates in the form of *conjunctive queries* that correspond to the high-level state features $x = n$ and $x < n$, respectively. In the example, x and n are variable objects in Ω_v, while v_3 and v_4 are bound variables in Ω_b. The predicate $\mathsf{lessthan}$ models the relation $<$ on value objects in Ω_x. Note that x and n are implicit assignments to two free variables v_1 and v_2; although the two derived predicates are valid for *any* pair of variable objects $x, n \in \Omega_v$, our model assumes that free variables are always instantiated on variable objects this way.

7.2. Computing programs with conjunctive queries

We incorporate conjunctive queries into planning programs by replacing the fluent f of a conditional goto instruction $goto(i', !f)$ with a conjunctive query φ. The execution of a program with conjunctive queries proceeds exactly as explained in Section 4, except for conditional goto instructions. Given a program state (s, i), the execution of a conditional goto instruction on a conjunctive query is defined as follows:

- If $w_i = goto(i', !\varphi)$, the new program state is $(s, i+1)$ if φ unifies with the current state s, and (s, i') otherwise.

We next explain how to extend the compilation from Section 4 to simultaneously compute the program and the high-level state features necessary to solve a given generalized planning task. To do so, we need a unification strategy that can be encoded in PDDL and integrated with our classical planning compilation.

Let $\varphi = (v_1, \ldots, v_k).\exists v_{k+1}, \ldots, v_m.\phi_1 \wedge \cdots \wedge \phi_q$ be a conjunctive query, and let $u \in \Omega_v^k$ be the corresponding assignment of variable objects to free variables. Our strategy for unifying φ with the current state s is to maintain a subset $\Phi \subseteq \Omega_x^{m-k}$ of possible joint assignments of value objects to the bound variables $v_{k+1}, \ldots, v_m$. The strategy unifies the atoms of φ with s one atom at a time, starting with ϕ_1, and updates the set Φ as we go along. After processing all atoms, φ will unify with s if and only if Φ is non-empty, i.e. if there remains at least one possible joint assignment to the bound variables.

To illustrate this unification strategy, consider again the navigation problem of visiting the four corners of an $n \times n$ grid (Fig. 4) and the derived predicate $\varphi = \mathsf{equal}(x, n)$ from Fig. 11. Assume that the agent is in the initial position $(4, 3)$ in a 5×5 grid, represented by the fluents $\{\mathsf{assign}(x, 4), \mathsf{assign}(y, 3), \mathsf{assign}(n, 5)\}$. Unification proceeds one atom at a time, and initially $\Phi = \Omega_x^1 = \{1, 2, 3, 4, 5, \ldots\}$, i.e. all assignments to the only bound variable v_3 are possible. The first atom $\mathsf{assign}(x, v_3)$ of φ unifies only with $v_3 = 4$, so joint assignments in Φ that do not assign the value object 4 to v_3 are no longer possible, resulting in an updated assignment set $\Phi = \{4\}$. The second atom $\mathsf{assign}(n, v_3)$ unifies only with $v_3 = 5$, but since this value object is no longer in Φ, the assignment set Φ becomes empty, and φ is considered *non-unifiable* with s.

With the unification strategy defined, we are ready to extend the compilation of Section 5 to compute planning programs with conjunctive queries. The extended compilation takes as input a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ and three constants n, q and b: n, the number of program lines; q, the number of atoms; and b, the number of bound variable

objects in Ω_b. The output of the compilation is a single planning problem $P^b_{n,q} = \langle F^b_{n,q}, A^b_{n,q}, I^b_{n,q}, G'_n \rangle$. A solution plan π to $P^b_{n,q}$ is a sequence of actions that encodes a planning program Π with conjunctive queries and validates that the execution of Π solves every classical planning instance in $\mathcal{P}$.

Since programming and executing action and termination instructions is identical to the compilation for basic planning programs (Section 4), we only describe here the part of $P^b_{n,q}$ that corresponds to programming and evaluating conjunctive queries:

- For each pair of program lines i, i' such that $i' \neq i$ and $i' \neq i+1$, then $F^b_{n,q}$ contains a fluent $\mathsf{ins}_{i,\mathsf{goto}(i')}$ indicating that the instruction on line i is a $goto(i', !\varphi)$ instruction with a conjunctive query.
- We define a set of bound variable objects $\Omega_b = \{v_1, \ldots, v_b\}$ and a set of *slots* $\Sigma = \{\sigma_1, \ldots, \sigma_q\}$. Each slot is a placeholder for an atom of a conjunctive query, and we also define a dummy slot σ_0.
  - For each slot $\sigma_k \in \Sigma \cup \{\sigma_0\}$, we add a fluent slot^k indicating that σ_k is the current slot (our unification strategy processes one atom at a time).
  - For each line i and slot $\sigma_k \in \Sigma$, a fluent eslot^k_i indicating that slot σ_k on line i is empty.
  - For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \Psi$ and tuple of bound variable objects $\omega \in \Omega_b^{ar(p)}$, a fluent $\mathsf{atom}\text{-}p^k_i(\omega)$ indicating that $p(\omega)$ is the atom in slot σ_k of line i. (If $p = \mathsf{assign}$, the first element ω_1 of ω is instead a variable object in Ω_v.)
  - For each slot $\sigma_k \in \Sigma$ and object tuple $(o_1, \ldots, o_b) \in \Omega^b_x$, a fluent $\mathsf{poss}^k(o_1, \ldots, o_b)$ indicating that at σ_k, $(o_1, \ldots, o_b)$ is a possible joint assignment of objects to the bound variables $v_1, \ldots, v_b$.
- A fluent eval indicating that we are done evaluating a conjunctive query and a fluent acc representing the outcome of the evaluation (true or false).

In the initial state, all fluents above appear as false except slot^0, indicating that we are ready to program and unify the atoms of any conjunctive query. The initial state on other fluents is identical to the original compilation, as is the goal condition. We next describe the set of actions in $A^b_{n,q}$ that have to be added to the original compilation to implement the mechanism for programming and evaluating conjunctive queries.

A conjunctive query φ is activated by programming a goto instruction $goto(i', !\varphi)$ on a given line i. As a result of programming the goto instruction, all slots on line i are marked as empty. For each pair of program lines i, i', the action $\mathsf{pgoto}_{i,i'}$ for programming $goto(i', !\varphi)$ on line i is defined as

$$\mathsf{pre}(\mathsf{pgoto}_{i,i'}) = \{\mathsf{pc}_i, \mathsf{ins}_{i,\mathsf{nil}}\},$$
$$\mathsf{cond}(\mathsf{pgoto}_{i,i'}) = \{\emptyset \rhd \{\neg\mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,\mathsf{goto}(i')}, \mathsf{eslot}^1_i, \ldots, \mathsf{eslot}^q_i\}\}.$$

The precondition contains two fluents from the original compilation: pc_i, modeling that the program counter equals i, and $\mathsf{ins}_{i,\mathsf{nil}}$, modeling that the instruction on line i is empty.

Once activated, we have to program the individual atoms in the slots of the conjunctive query φ. After programming an atom, the slot is no longer empty. For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \Psi$ and tuple of bound variable objects $\omega \in \Omega_b^{ar(p)}$, the action $\mathsf{patom}\text{-}p^k_i(\omega)$ is defined as

$$\mathsf{pre}(\mathsf{patom}\text{-}p^k_i(\omega)) = \{\mathsf{pc}_i, \mathsf{slot}^{k-1}, \mathsf{eslot}^k_i\},$$
$$\mathsf{cond}(\mathsf{patom}\text{-}p^k_i(\omega)) = \{\emptyset \rhd \{\neg\mathsf{eslot}^k_i, \mathsf{atom}\text{-}p^k_i(\omega)\}\}.$$

Again, if $p = \mathsf{assign}$, the first element ω_1 of ω is instead a variable object in Ω_v (effectively assigning ω_1 to a free variable of the conjunctive query).

The key ingredient of the compilation are step actions that iterate over the atoms in each slot while propagating the remaining possible values of the bound variables. For each line i, slot $\sigma_k \in \Sigma$, predicate $p \in \Psi$ and tuple of bound variable objects $\omega \in \Omega_b^{ar(p)}$, step action $\mathsf{step}\text{-}p^k_i(\omega)$ is defined as

$$\mathsf{pre}(\mathsf{step}\text{-}p^k_i(\omega)) = \{\mathsf{pc}_i, \mathsf{slot}^{k-1}, \mathsf{atom}\text{-}p^k_i(\omega)\},$$
$$\mathsf{cond}(\mathsf{step}\text{-}p^k_i(\omega)) = \{\emptyset \rhd \{\neg\mathsf{slot}^{k-1}, \mathsf{slot}^k\}\}$$
$$\cup\ \{\{\mathsf{poss}^{k-1}(o_1, \ldots, o_b), p(o(\omega))\} \rhd \{\mathsf{poss}^k(o_1, \ldots, o_b)\} : (o_1, \ldots, o_b) \in \Omega^b_x\}.$$

To apply a step action, an atom has to be programmed first. The unconditional effect is moving from slot σ_{k-1} to slot σ_k. In addition, the step action updates the possible assignments to the bound variables $v_1, \ldots, v_b$.

For an assignment $(o_1, \ldots, o_b)$ to be possible at slot σ^k, it has to be possible at σ^{k-1}, and the atom $p(\omega)$ programmed at slot k has to induce a fluent $p(o(\omega))$ that is currently true. Here, $o(\omega) \in \Omega_x^{ar(p)}$ denotes the tuple of $ar(p)$ value objects that is induced by $(o_1, \ldots, o_b)$ and ω. For example, if $\omega = (v_3, v_1, v_2)$, then $o(\omega) = o(v_3, v_1, v_2) = (o_3, o_1, o_2)$. Note that there is one conditional effect for each possible assignment $(o_1, \ldots, o_b)$ to $v_1, \ldots, v_b$. If $k = 1$, the condition $\mathsf{poss}^{k-1}(o_1, \ldots, o_b)$ is

removed since all assignments are possible prior to evaluating the first atom. If $p = \mathsf{assign}$, the first element in $o(\omega)$ simply equals $\omega_1 \in \Omega_v$, the variable object programmed as the first argument of p.

Once we have iterated over all atoms, we have to check whether there remains at least one possible assignment, thereby evaluating the entire conjunctive query. For each line i, let eval_i be an action defined as

$$\mathsf{pre}(\mathsf{eval}_i) = \{\mathsf{pc}_i, \mathsf{slot}^q\},$$
$$\mathsf{cond}(\mathsf{eval}_i) = \{\emptyset \rhd \{\mathsf{eval}\}\} \cup \{\{\mathsf{poss}^q(o_1, \ldots, o_b)\} \rhd \{\mathsf{acc}\} : (o_1, \ldots, o_b) \in \Omega_x^b\}.$$

Action eval_i is only applicable once we are at the last slot σ_q. The conditional effects add the fluent acc if and only if there remains a possible assignment to $v_1, \ldots, v_b$ at σ_q.

Finally, we can now use the result of the evaluation to determine the program line that we jump to. For each pair of lines i, i', let $\mathsf{jmp}_{i,i'}$ be an action defined as

$$\mathsf{pre}(\mathsf{jmp}_{i,i'}) = \{\mathsf{pc}_i, \mathsf{ins}_{i,\mathsf{goto}(i')}, \mathsf{slot}^q, \mathsf{eval}\},$$
$$\mathsf{cond}(\mathsf{jmp}_{i,i'}) = \{\emptyset \rhd \{\neg\mathsf{pc}_i, \neg\mathsf{eval}, \neg\mathsf{acc}, \neg\mathsf{slot}^q, \mathsf{slot}^0\}\}$$
$$\cup\ \{\{\neg\mathsf{acc}\} \rhd \{\mathsf{pc}_{i'}\}, \{\mathsf{acc}\} \rhd \{\mathsf{pc}_{i+1}\}\}$$
$$\cup\ \{\emptyset \rhd \{\neg\mathsf{poss}^k(o_1, \ldots, o_b) : 1 \leq k \leq q, (o_1, \ldots, o_b) \in \Omega_x^b\}\}.$$

The effect is to jump to line i' if acc is false, else continue execution on line $i + 1$. We also delete fluents eval and acc, as well as all instances of $\mathsf{poss}^k(o_1, \ldots, o_b)$ in order to reset the evaluation mechanism prior to the next evaluation of a conjunctive feature. The current slot is also reset to σ_0.

7.3. Classification with planning programs

Our extension of planning programs with conjunctive queries allows us to model supervised classification tasks as if they were generalized planning problems and therefore address them using our compilation and a classical planner. Although this approach is not competitive with current Machine Learning techniques for supervised classification it brings a brand new landscape of benchmarks to classical planning.

Formally, learning a noise-free classifier from a set of labeled examples $\{e_1, \ldots, e_T\}$, where each example e_t, $1 \leq t \leq T$, is labeled with one class in $C = \{c_1, \ldots, c_Z\}$, can be viewed as a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ such that each individual planning problem $P_t = \langle F, A, I_t, G_t \rangle$, $1 \leq t \leq T$, models the classification of the tth example:

- F is the set of fluents needed to model the features and labels of the learning examples.
- A contains the actions to label a given example with a class. For instance, a binary classification task can be modeled as $C = \{0, 1\}$ and $A = \{\mathsf{setNegative}, \mathsf{setPositive}\}$, where setNegative labels an example with class 0, and setPositive labels an example with class 1.
- $I_t \subseteq F$ contains the fluents that describe the tth example while G_t is the fluent that defines the class label of the tth example.

According to this formulation, a solution Π to a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ (that models a classification task) is a noise-free classifier that covers the T learning examples.

This model is particularly natural for classification tasks in which both the examples and the classifier are described using logic. The classic *Michalski's train* [61] is a good example of such tasks. It is a binary classification problem that defines 10 different trains (5 traveling east and 5 traveling *west*) and the classification target is explaining what causes a train to travel *east* or *west*. Trains are defined using the following relations: which wagon is in a given train, which wagon is in front, the wagon's shape, its number of wheels, whether it has a roof or not (closed or open), whether it is long or short, the shape of the objects the wagon is loaded with and the class of the train (*east* or *west*).

In more detail, the generalized planning problem encoding the *Michalski's train* task would be:

- Fluents F induced from $\Psi = \{$(wagons ?Number), (hasCar ?Car), (infront ?Car ?Car), (shape ?Car ?Shape), (wheels ?Car ?Number), (closed ?Car), (open ?Car), (long ?Car), (short ?Car), (double ?Car), (jagged ?Car), (load ?Car ?Shape ?Number), (eastbound), (westbound)$\}$.
- Actions $A = \{$setWest$\}$. Although this is a binary classification task we assume that any example has initial class eastbound, causing the resulting planning programs to be more compact.

$$\mathsf{pre}(\mathsf{setWest}) = \{\emptyset\},$$
$$\mathsf{cond}(\mathsf{setWest}) = \{\emptyset \rhd \{\neg\mathsf{eastbound}, \mathsf{westbound}\}\}.$$

- Each initial state I_t defines the tth train and G_t defines its associated class (traveling east or traveling west).

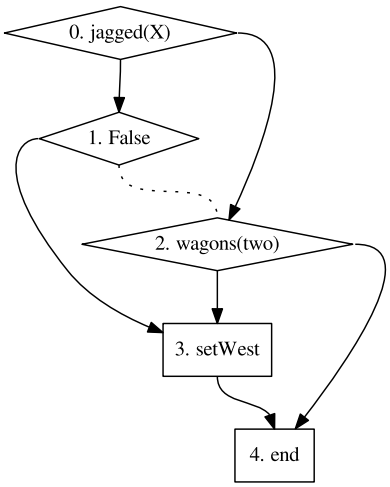


Fig. 12. Planning program that encodes a noise-free classifier for the Michalski's train problem.

Fig. 12 shows a planning program computed using our compilation and that encodes a noise-free classifier for the Michalski's train problem. As explained, the program assumes that any example initially has class eastbound. Line 1 of the program is an unconditional jump to line 3. The program encodes the following two classification rules: setWest $\leftarrow$ jagged(X) and setWest $\leftarrow \neg$jagged(X), wagons(two) that capture that a given train is traveling west if it has a jagged wagon or if it does not have a jagged wagon but it is a train that has exactly two wagons.

7.4. Properties of the compilation

With regard to the correctness of the approach we provide here a formal proof of its soundness and completeness. We focus here on the correctness of programming and unifying high-level state features since the proofs for the remaining functionality of the compilation is analogous to the proofs of the corresponding theorems in Section 4.

Theorem 11 *(Soundness). Any plan π that solves $P^b_{n,q}$ induces a program Π with conjunctive queries that solves $\mathcal{P}$.*

Proof. Since planning programs with conjunctive queries are identical to basic planning programs except for goto instructions, the arguments from the proof of Theorem 3 still apply. Programming atoms in slots is only possible when slots are empty, and once programmed, atoms cannot change, so the fluents of type atom-$p^k_i(\omega)$ effectively encode a conjunctive query with q atoms on a line i. All that remains is to show that after iterating over the atoms of a conjunctive query φ, the fluent acc is true if and only if φ unifies with the current state s.

We show by induction on k that after iterating over the atoms of φ, the fluent poss$^k(o_1, \ldots, o_b)$ is true if and only if the assignment $(o_1, \ldots, o_b)$ to the bound variables $v_1, \ldots, v_b$ causes the set of atoms $\{\phi_1, \ldots, \phi_k\}$ of φ to unify with s. The base case is given by $k = 0$, in which case the set of atoms $\{\phi_1, \ldots, \phi_0\}$ is empty and thus trivially unifies with s. For $k > 0$, by hypothesis of induction the fluent poss$^{k-1}(o_1, \ldots, o_b)$ is true if and only if $(o_1, \ldots, o_b)$ causes $\{\phi_1, \ldots, \phi_{k-1}\}$ to unify with s. Because of the definition of action step-$p^k_i(\omega)$, poss$^k(o_1, \ldots, o_b)$ becomes true if and only if poss$^{k-1}(o_1, \ldots, o_b)$ is true and the fluent $p(o(\omega))$ induced by p, ω and $(o_1, \ldots, o_b)$ is true in s. This corresponds precisely to $\{\phi_1, \ldots, \phi_k\}$ unifying with s. □

Theorem 12 *(Completeness). If there exists a planning program Π with conjunctive queries that solves $\mathcal{P}$ such that $|\Pi| \leq n$, there exists a corresponding plan π that solves $P^b_{n,q}$.*

Proof. Again, the only difference with respect to the proof of Theorem 4 is the treatment of goto instructions. For each possible atom $p(\omega)$, there is a corresponding action patom-$p^k_i(\omega)$ that programs $p(\omega)$ in slot σ^k of line i. Hence we can emulate any conjunctive query φ by programming the appropriate atoms in the slots of a line. The resulting plan π also has to check whether φ unifies with the current state s, but this is a deterministic process. The only issue is that we have to ensure that the bounds q and b are generous enough to accommodate the conjunctive queries of Π. □

Just as the bound n on the number of program lines, setting the bounds q and b too small may make $P^b_{n,q}$ unsolvable. For example, our compilation can only generate the derived predicates in Fig. 11 if $q \geq 3$ and $b \geq 2$. Setting the bounds too high does not formally affect the completeness of the approach, but causes an increase in the size of the resulting classical planning problem and thus affects the performance of the classical planner used to solve $P^b_{n,q}$.

Table 10
Plan generation for planning programs with high-level state features. Number of program lines, variables and slots; number of fluents and actions; search, preprocess and total time (in seconds) elapsed while computing the solution.

Domain	Lines	Vars	Slots	Fluents	Actions	Search	Preprocess	Total
List	3	1	2	333	208	0.03	0.42	0.45
Triangular	3	1	2	450	243	1.18	1.56	2.74
Trains	4	1	1	706	440	21.86	1.40	23.26
And	4	1	2	231	152	0.12	0.19	0.31
Or	4	1	2	231	152	0.11	0.18	0.29
Xor	4	2	2	327	200	0.06	0.26	0.32

With respect to the size of the compilation it is not only influenced by the values of its parameters n, b and q but also by the number of predicates and their arity since the space of possible high-level state features for a given generalized planning tasks depends on these numbers.

7.5. Experiments

We evaluate our method in two kinds of benchmarks. We first consider benchmarks from generalized planning where the target is generating a plan that generalizes without providing any prior high-level representation of the states. This set of benchmarks include iterating over a list and computing the N-th triangular number $\sum_{n=1}^{N} n$. On the other hand, we consider binary classification tasks which include Michalski's train, where we must decide if a train goes east or west based on a given group of features, as well as generating the classifiers corresponding to the logic functions $y = and(X_1, X_2)$, $y = or(X_1, X_2)$ and $y = xor(X_1, X_2)$. Table 10 summarizes the obtained results using Fast Downward in the Lama-2011 setting. We report the number of program lines used to solve the generalized planning problem, the number of bounded variables and slots required to generate the features, and the search, preprocess and total time taken to generate the program.

We briefly describe the features and programs generated for the different domains. In *List*, we generate the Boolean feature $f = \mathsf{equal}(X, i) \wedge \mathsf{equal}(X, n)$, that is equivalent to $(i == n)$. In Triangular, we generate the feature $f = \mathsf{equal}(X, y) \wedge \mathsf{equal}(X, \mathsf{sum}(X, X))$, which is actually equivalent to $(y == 0)$. For Michalski's train, we generate the program and features shown in Fig. 12. In the $and(X_1, X_2)$ and $or(X_1, X_2)$ domains, two features are generated: one that represents whether a given variable is false or true, respectively, and a second feature $f = \mathsf{equal}(\mathsf{X}, \mathsf{y}) \wedge \mathsf{equal}(X, False)$ that captures if the instance is already classified (the value of y is already set). The 4-line program generated for $and(X_1, X_2)$ is:

```
0. goto(3,!equal(X,False))
1. setFalse
2. goto(4,!(equal(X,y) ∧ equal(X,False)))
3. setTrue
4. end
```

In contrast to the *and* and *or* functions, the $xor(X_1, X_2)$ function requires 2 variables for the first feature to detect that one is true and the other is false or vice versa, while the second feature is just to know if it has been classified.

8. Conclusion

We introduced a novel formalism for representing generalized plans, that can branch and loop, and presented a compilation to compute them with off-the-shelf planners. Unlike previous work for planning with loops [62], the input to our approach is a set of example tasks to be covered, like a *training set* in ML. These input tasks are represented as standard classical planning tasks in PDDL, and can be solved using an off-the-shelf classical planner.

So far, the utility of our compilation is limited to tasks solvable with small planning programs. Part of the reason is that the number of possible programs is exponential in the number of program lines. In addition, some domains require complex state queries to compute a compact generalized plan. For instance, a compact while general solution for *sokoban* requires defining complex connectivity properties, such as *reachable*, that involve recursion [59]. For these reasons our current approach cannot compute effective DCK for arbitrary classical planning domains. We have shown that we can address more challenging tasks if a subtask decomposition is available. In this case we can separately compute auxiliary procedures for each of the subtasks and incrementally reuse them. An interesting open research direction is then to automatically discover these subtask decompositions.

Fast Downward, the planner used in the evaluation, is the most widely used in the planning community and is a good touchstone to evaluate the capacity of classical planners to address generalized planning tasks. On the other hand, Fast Downward is a heuristic-search based planner whose heuristics are based in the delete relaxation of the planning problem and typically has difficulties with problems that include dead-ends, such as those in our compilations. The domains resulting from our compilation encode key information in the delete effects of actions: when an instruction is programmed on a line i, the fluent $\mathsf{ins}_{i,\mathsf{nil}}$ is deleted, preventing us from programming another instruction on the same line. In a delete-free relaxation, $\mathsf{ins}_{i,\mathsf{nil}}$ remains true, making it possible to program any number of instructions on the same line. As a consequence, a solution to the delete-free relaxation can effectively use different programs to solve the individual instances of a

generalized planning problem, which results in a poor approximation of how difficult the original problem is. In addition, a program suitable for a particular individual instance might not be suitable for solving the next instance, causing dead-ends and deep backtracking. In the future it would be interesting to explore novel planning techniques that deal better with these issues.

In experiments, the resulting generalized plan solves all instances of a given domain. In general, however, there are no such guarantees and the planner only validates that the resulting generalized plan satisfies the planning problems that comprise the generalized planning task. A different approach is defining the goals of generalized tasks using expressive logic formulas instead of a set of tests cases, for example using LTL [63] or quantified goals [64]. In that case the planner should verify that the generated planning program satisfies the formula.

Related to this issue is the selection of relevant examples that can produce a solution that generalize. A key issue is to determine which instances generalize most efficiently. Currently this selection of informative instances is done by hand, and an interesting research direction is to develop techniques for automatic instance selection. In this case implicit representations of the planning instances [4] seem more useful since they can act as generative models. Since planning problems are highly structured there is no guarantee that randomly sampled problems are relevant for solving a given generalized planning task. An interesting research direction would be to study how to generate examples of this kind. A good starting point could be previous work on generating random walks [65] and active learning [66] in planning domains.

Finally, we are only able to generate high-level state features in the form of conjunctive queries, and hence we cannot produce from scratch programs that contain features with unbounded transitive or recursive closures. This kind of features are known to be useful for some planning domains, e.g. the *above* feature (the transitive closure of *on*) for the Blocksworld domain. In the near future we would like to extend our approach to generating more expressive features.

Acknowledgements

Anders Jonsson is partially supported by the grants TIN2015-67959 and PCIN-2017-082 of the Spanish Ministry of Science. Sergio Jiménez is partially supported by the grants, RYC-2015-18009 and TIN2017-88476-C2-1-R of the Spanish Ministry of Science.

Appendix A. PDDL definition of the grid navigation domain and example planning problem

```
(define (domain gridnav)                        ;; Grid navigation domain
  (:requirements :conditional-effects :typing)
  (:types variable value)
  (:predicates
    (assignment ?var - variable ?val - value) ;; Variable \xch{assignment}{assigment}
    (next ?val1 ?val2 - value)                ;; Static relations between values
    (max-value ?var - variable ?val - value)  ;; Maximum value in the domain of ?var
    (is-max ?var - variable)                  ;; Whether ?var has its maximum value
  )
  (:action inc
    :parameters (?var - variable)
    :precondition (and)
    :effect (and
      (forall (?val1 ?val2 - value)
        (when (and (not (is-max ?var)) (assignment ?var ?val1) (next ?val1 ?val2))
              (and (not (assignment ?var ?val1)) (assignment ?var ?val2)))))
  )
  (:action dec
    :parameters (?var - variable)
    :precondition (and)
    :effect (and
      (forall (?val1 ?val2 - value)
        (when (and (assignment ?var ?val2) (next ?val1 ?val2))
              (and (not (assignment ?var ?val2)) (assignment ?var ?val1)))))
  )
  (:derived (is-max ?var - variable)
    (exists (?val - value) (and (assignment ?var ?val) (max-value ?var ?val))))
)



(define (problem gridnav-4-3)                   ;; Example grid navigation problem
  (:domain gridnav)
  (:objects
```

```
    x y - variable
    v1 v2 v3 v4 v5 v6 v7 - value
  )
  (:init
    (assignment x v4)
    (assignment y v3)
    (max-value x v5)
    (max-value y v5)
    (next v1 v2)
    (next v2 v3)
    (next v3 v4)
    (next v4 v5)
    (next v5 v6)
    (next v6 v7)
  )
  (:goal (and
    (assignment x v1)
    (assignment y v1))
  )
)
```

References

[1] E. Winner, M. Veloso, Distill: learning domain-specific planners by example, in: International Conference on Machine Learning, 2003, pp. 800–807.
[2] B. Bonet, H. Palacios, H. Geffner, Automatic derivation of finite-state machines for behavior control, in: AAAI Conference on Artificial Intelligence, 2010.
[3] Y. Hu, H.J. Levesque, A correctness result for reasoning about one-dimensional planning problems, in: International Joint Conference on Artificial Intelligence, 2011, pp. 2638–2643.
[4] S. Srivastava, N. Immerman, S. Zilberstein, T. Zhang, Directed search for generalized plans using classical planners, in: International Conference on Automated Planning and Scheduling, 2011, pp. 226–233.
[5] Y. Hu, G. De Giacomo, A generic technique for synthesizing bounded finite-state controllers, in: International Conference on Automated Planning and Scheduling, 2013.
[6] R. Khardon, Learning action strategies for planning domains, Artif. Intell. 113 (1) (1999) 125–148.
[7] M. Martín, H. Geffner, Learning generalized policies from planning examples using concept languages, Appl. Intell. 20 (2004) 9–19.
[8] J.A. Baier, C. Fritz, S.A. McIlraith, Exploiting procedural domain control knowledge in state-of-the-art planners, in: ICAPS, 2007, pp. 26–33.
[9] S. Jiménez, A. Jonsson, Computing plans with control flow and procedures using a classical planner, in: Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS-15, 2015, pp. 62–69.
[10] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Generalized planning with procedural domain control knowledge, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2016.
[11] D. Lotinac, J. Segovia, S. Jiménez, A. Jonsson, Automatic generation of high-level state features for generalized planning, in: International Joint Conference on Artificial Intelligence, 2016.
[12] A. Botea, M. Enzenberger, M. Müller, J. Schaeffer, Macro-ff: improving AI planning with automatically learned macro-operators, J. Artif. Intell. Res. 24 (2005) 581–621.
[13] A. Coles, A. Smith Marvin, A heuristic search planner with online macro-action learning, J. Artif. Intell. Res. 28 (2007) 119–156.
[14] A. Jonsson, The role of macros in tractable planning, J. Artif. Intell. Res. (2009) 471–511.
[15] S. Yoon, A. Fern, R. Givan, Learning control knowledge for forward search planning, J. Mach. Learn. Res. 9 (2008) 683–718.
[16] T. De la Rosa, S. Jiménez, R. Fuentetaja, D. Borrajo, Scaling up heuristic planning with relational decision trees, J. Artif. Intell. Res. (2011) 767–813.
[17] J. Rintanen, Planning as satisfiability: heuristics, Artif. Intell. J. 193 (2012) 45–86.
[18] C. Pralet, G. Verfaillie, M. Lemaître, G. Infantes, Constraint-based controller synthesis in non-deterministic and partially observable domains, in: European Conference on Artificial Intelligence, 2010, pp. 681–686.
[19] M. Fox, A. Gerevini, D. Long, I. Serina, Plan stability: replanning versus plan repair, in: ICAPS, vol. 6, 2006, pp. 212–221.
[20] D. Borrajo, A. Roubíčková, I. Serina, Progress in case-based planning, ACM Comput. Surv. 47 (2) (2015) 35.
[21] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, R.B. Scherl, Golog: a logic programming language for dynamic domains, J. Funct. Logic Program. 31 (1–3) (1997) 59–83.
[22] S. Sardina, G. De Giacomo, Y. Lespérance, H.J. Levesque, On the semantics of deliberation in indigolog—from theory to implementation, Ann. Math. Artif. Intell. 41 (2–4) (2004) 259–299.
[23] G. Röger, M. Helmert, B. Nebel, On the relative expressiveness of adl and golog: the last piece in the puzzle, in: KR, 2008.
[24] J. Claßen, V. Engelmann, G. Lakemeyer, G. Röger, Integrating golog and planning: an empirical evaluation, in: Non-Monotonic Reasoning Workshop, 2008.
[25] S. Srivastava, N. Immerman, S. Zilberstein, A new representation and associated algorithms for generalized planning, Artif. Intell. 175 (2) (2011) 615–647.
[26] A. Albore, H. Palacios, H. Geffner, A translation-based approach to contingent planning, in: International Joint Conference on Artificial Intelligence, 2009.
[27] C. Boutilier, R. Reiter, B. Price, Symbolic dynamic programming for first-order mdps, in: IJCAI, vol. 1, 2001, pp. 690–700.
[28] C. Gretton, S. Thiébaux, Exploiting first-order regression in inductive policy selection, in: Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence, AUAI Press, 2004, pp. 217–225.
[29] S. Gulwani, Automating String Processing in Spreadsheets Using Input–Output Examples, ACM SIGPLAN Not., vol. 46, ACM, 2011, pp. 317–330.
[30] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat, Combinatorial sketching for finite programs, Oper. Syst. Rev. 40 (2006) 404–415.
[31] B.M. Lake, R. Salakhutdinov, J.B. Tenenbaum, Human-level concept learning through probabilistic program induction, Science 350 (6266) (2015) 1332–1338.
[32] B. Finkbeiner, S. Schewe, Bounded synthesis, Int. J. Softw. Tools Technol. Transf. 15 (5–6) (2013) 519–539.
[33] F. Bacchus, F. Kabanza, Using temporal logics to express search control knowledge for planning, Artif. Intell. 116 (1) (2000) 123–191.

[34] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, J. Blythe, Integrating planning and learning: the prodigy architecture, J. Exp. Theor. Artif. Intell. 7 (1) (1995) 81–120.
[35] V. Shivashankar, U. Kuter, D. Nau, R. Alford, A hierarchical goal-based formalism and algorithm for single-agent planning, in: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems, vol. 2, International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 981–988.
[36] D.S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, F. Yaman, Shop2: an HTN planning system, J. Artif. Intell. Res. 20 (2003) 379–404.
[37] T.M. Mitchell, Generalization as search, Artif. Intell. 18 (2) (1982) 203–226.
[38] S. Muggleton, Inductive logic programming: issues, results and the challenge of learning language in logic, Artif. Intell. 114 (1) (1999) 283–296.
[39] H. Palacios, H. Geffner, Compiling uncertainty away in conformant planning problems with bounded width, J. Artif. Intell. Res. 35 (2009) 623–675.
[40] M. Fox, D. Long, Pddl2. 1: an extension to PDDL for expressing temporal planning domains, J. Artif. Intell. Res. 20 (2003) 61–124.
[41] Y. Hu, G. De Giacomo, Generalized planning: synthesizing plans that work for multiple environments, in: International Joint Conference on Artificial Intelligence, 2011, pp. 918–923.
[42] A. Fern, R. Khardon, P. Tadepalli, The first learning track of the international planning competition, Mach. Learn. 84 (1–2) (2011) 81–107.
[43] S. Jiménez, T. De La Rosa, S. Fernández, F. Fernández, D. Borrajo, A review of machine learning for automated planning, Knowl. Eng. Rev. 27 (04) (2012) 433–467.
[44] T. Bylander, The computational complexity of propositional STRIPS planning, Artif. Intell. 69 (1994) 165–204.
[45] M. Helmert, The fast downward planning system, J. Artif. Intell. Res. 26 (2006) 191–246.
[46] S. Richter, M. Westphal, The LAMA planner: guiding cost-based anytime planning with landmarks, J. Artif. Intell. Res. 39 (2010) 127–177.
[47] M. Ramirez, N. Lipovetzky, C. Muise, Lightweight automated planning ToolKiT, http://lapkt.org/, 2015 (Accessed 19 December 2016).
[48] C. Domshlak, Fault tolerant planning: complexity and compilation, in: Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS, 2013.
[49] V. Shivashankar, U. Kuter, D. Nau, R. Alford, A hierarchical goal-based formalism and algorithm for single-agent planning, in: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems, vol. 2, International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 981–988.
[50] C. Hogg, H. Munoz-Avila, U. Kuter, HTN-maker: learning HTNs with minimal additional knowledge engineering required, in: AAAI, 2008, pp. 950–956.
[51] D. Lotinac, A. Jonsson, Constructing hierarchical task models using invariance analysis, in: European Conference on Artificial Intelligence, 2016.
[52] B. Marthi, S. Russell, J. Wolfe, Angelic semantics for high-level actions, in: Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS, 2007, pp. 232–239.
[53] R. Alford, U. Kuter, D.S. Nau, Translating HTNs to PDDL: a small amount of domain knowledge can go a long way, in: International Joint Conference on Artificial Intelligence, 2009, pp. 1629–1634.
[54] T. Zimmerman, S. Kambhampati, Learning-assisted automated planning: looking back, taking stock, going forward, AI Mag. 24 (2) (2003) 73.
[55] C.M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.
[56] S. Cresswell, A.M. Coddington, Compilation of ltl goal formulas into PDDL, in: ECAI, 2004, pp. 985–986.
[57] J. Hoffmann, S. Edelkamp, The deterministic part of ipc-4: an overview, J. Artif. Intell. Res. (2005) 519–579.
[58] S. Thiébaux, J. Hoffmann, B. Nebel, In defense of PDDL axioms, Artif. Intell. 168 (1) (2005) 38–69.
[59] F. Ivankovic, P. Haslum, Optimal planning with axioms, in: International Joint Conference on Artificial Intelligence, AAAI Press, 2015, pp. 1580–1586.
[60] A. Chandra, P. Merlin, Optimal implementation of conjunctive queries in relational data bases, in: Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, 1977.
[61] R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Machine Learning: An Artificial Intelligence Approach, Springer Science & Business Media, 2013.
[62] H.J. Levesque, Planning with loops, in: IJCAI, 2005, pp. 509–515.
[63] F. Patrizi, N. Lipoveztky, G. De Giacomo, H. Geffner, Computing infinite plans for ltl goals using a classical planner, in: Twenty-Second International Joint Conference on Artificial Intelligence, 2011.
[64] G. Frances, H. Geffner, ∃-strips: existential quantification in planning and constraint satisfaction, in: IJCAI, 2016, pp. 3082–3088.
[65] A. Fern, S.W. Yoon, R. Givan, Learning domain-specific control knowledge from random walks, in: ICAPS, 2004, pp. 191–199.
[66] R. Fuentetaja, D. Borrajo, Improving control-knowledge acquisition for planning by active learning, in: Machine Learning: ECML 2006: Proceedings of the 17th European Conference on Machine Learning, Berlin, Germany, September 18–22, 2006, 2006, pp. 138–149.