Artificial Intelligence
Manuscript Draft

Manuscript Number: ARTINT-D-17-00064

Title: Computing programs for generalized planning using a classical planner

Article Type: Full Length Article

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.

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Artificial Intelligence 00 (2017) 1–33
Artificial Intelligence

Computing programs for generalized planning using a classical planner

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

[a]*Department of Information and Communication Technologies, Universitat Pompeu Fabra. Spain*

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.

Keywords: Generalized planning, Classical planning, Planning and learning, Program synthesis

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, 2, 3, 4, 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 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 generalized plans are only computable if certain high-level representation of the states is available. For instance, *high-level state features* indicating that a block in the classic *blocksworld* domain can be unstacked or that already is at its goal position [6, 7]. In most applications of generalized planning such features are hand-coded by a domain expert. 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 maybe complex to achieve generalization if the instances in the generalized planning task do not share a clear common structure. In such cases a useful approach may be to represent and compute solutions with non-deterministic execution [8]. Solutions of this kind are more flexible since 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 sequential instructions*.

A first description of the compilation and its extensions previously appeared in several conference papers [9, 10, 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 sequential 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 100 blocks blocksworld instances.
- 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 address tasks of this kind.

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, 13, 14]. However, the sequential execution flow fixed by macros is usually too rigid and, even when macros are parameterized, a solution involving macros may not be applicable to other planning instances.

A more flexible formalism than macros are *generalized policies*. 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. Algorithms for computing generalized policies first compute sequential plans, and then attempt to extract decision rules from these plans, a task made difficult because of the high number of symmetries and transpositions that commonly appear in sequential plans. Moreover, a *generalized policy* cannot be added to a domain theory even though there are effective mechanisms for integrating them into state-of-the-art 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 serie 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 [21, 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 alternative 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 the plans.

Contingent planning [22] can be seen as an example of generalized planning where the input instances to be solved share the same goals and 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].

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* [23] and *Programming by sketching* [24]. 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 [25].

Last but not least our approach for computing high-level state features is inspired by *version space learning* [26]. 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) [27] also generates hypotheses from examples intersecting Machine Learning (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. Generating high-level state features for generalized planning is also related to previous work on First Order MDPs [28, 29]. 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.

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 [30], 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 [31]. Specifically, there exists a set of predicates Ψ, and each predicate $p \in \Psi$ has an 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 in Ψ, 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 literal l is a valuation of a fluent $f \in F$, i.e. $l = f$ or $l = \neg f$. A set of literals L represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). Given L, let $\neg L = \{\neg l : l \in L\}$ be the complement of L. A *state* s is a set of literals such that $|s| = |F|$, i.e. a total assignment of values to fluents. The number of states is then bounded and given by $2^{|F|}$. Given a subset $F' \subseteq F$ of fluents, let $s|_{F'}$ be the *projection* of s onto F', defined as $s|_{F'} = (s \cap F') \cup (s \cap \neg F')$, i.e. $s|_{F'}$ contains all literals on F' that are present in s. Explicitly including negative literals $\neg f$ in states 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 STRIPS 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 set of literals $\mathsf{pre}(a)$ 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 sets of literals 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

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

i.e. effects whose conditions hold in s. The result of applying a in s is a new state $\theta(s, a) = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$.

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 set of literals 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 i such that $1 \leq i \leq n$, 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 [32], 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 interpretation in different states due to conditional effects.

A solution π to a generalized planning problem $\mathcal{P}$ is a generalized plan that solves each individual problem P_t, $1 \leq t \leq T$. 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.

In this work we restrict the above definition for generalized planning in two ways: 1) states are fully observable, so observations are equivalent to the state variables; and 2) each action has the same (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. This definition of the generalized planning task is related to previous works on planning and learning that extract and reuse general knowledge from different tasks of the same domain [33, 34]. We impose however a stronger constraint on the individual planning tasks since they do not only share the set of predicates but share also the set of fluents so all the individual planning tasks have the same state space. In addition in this work we assume that the solutions to the generalized planning task have the form of a planning program as defined next at Section 4.

Figure 1 shows an example of a generalized planning task 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. Note that our definition of generalized planning makes it possible to encode individual planning tasks with different grid sizes since F can include fluents of type $\mathsf{max}(x, a)$ and $\mathsf{max}(y, b)$ that encode variable grid boundaries. In the example 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. A generalized plan for solving this generalized planning task 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.*

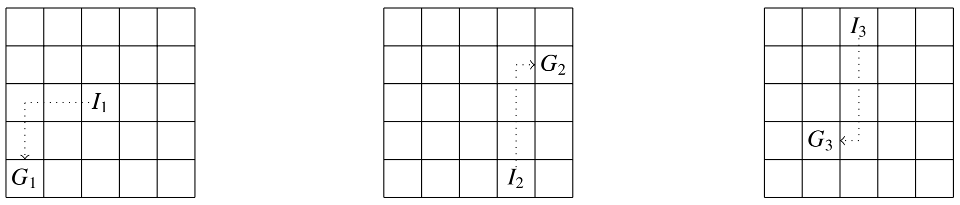


Figure 1: Example of three individual classical planning tasks comprised in a generalized planning task $\mathcal{P} = \{P_1, P_2, P_3\}$.

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 [30]. 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 STRIPS frame $\Phi = \langle F, A\rangle$, a basic planning program is a sequence of instructions $\Pi = \langle w_0, \ldots, w_n\rangle$. 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 a planning action $a \in A$, a 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 = `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 = `end`, 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 execution terminates and the goal condition holds in the resulting program state (s, i), i.e. $G \subseteq s \wedge w_i = \texttt{end}$. Executing Π on P can fail for three reasons:

1. Execution terminates in program state (s, i) but the goal condition does not hold, i.e. $G \nsubseteq s \wedge w_i = \texttt{end}$.
2. When executing an action $w_i \in A$ in program state (s, i), the precondition of w_i does not hold, i.e. $\mathsf{pre}(w_i) \nsubseteq s$.
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 family of planning problems defined by the STRIPS frame $\Phi = \langle F, A\rangle$.

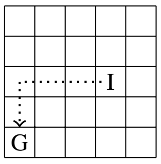
Figure 2(a) shows an example planning program Π for navigating to the (1, 1) cell in a grid. Variables x and y represent the position of the current grid cell. Instructions `dec(x)` and `dec(y)` decrement the value of x and y. The goto instructions `goto(0,!(x=1))` and `goto(2,!(y=1))` jump to line 0 when $x \neq 1$ and to line 2 when $y \neq 1$, respectively. The execution of Π on the planning problem illustrated in Figure 2(b) produces the following sequence of planning actions: ⟨`dec(x)`,`dec(x)`,`dec(x)`,`dec(y)`,`dec(y)`⟩. Unlike most other compact plan representations, Π solves *any* planning problem of this type 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 $|\Pi|$, 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)

Figure 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 the following two decision problems for planning programs:

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

BPE(X) (bounded plan existence for planning programs in class X)
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?

In the above definitions, X refers to any class of planning programs. Below we use PP to denote the class of basic planning programs as defined above.

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 STRIPS planning [35]. Given a DTM M, 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, $A = \{$`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 Φ 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 finite size of the tape. The goal state 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^x_M 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 $\text{NP}^{\text{PSPACE}} = \text{PSPACE}$.

Hardness: Given a DTM M and an input string x, consider the planning problem P^x_M from the proof of Theorem 1. There exists a planning program with 2 program lines, namely Π_M, that solves P^x_M 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 computing 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 Π 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 (*sequential*, *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 Π 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 classical planning action 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:

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

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 Π 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 are false for $2 \leq t \leq T$, and the initial state on fluents in F is I_1, i.e. $I'_n = I_1 \cup \{\mathsf{ins}_{i,\mathsf{nil}} : 0 \leq i \leq n\} \cup \{\mathsf{pc}_0\} \cup \{\mathsf{test}_1\}$, and 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, $1 \leq t \leq T$:

  $$\begin{aligned}\mathsf{pre}(\mathsf{end}_{t,i}) &= G_t \cup \{\mathsf{pc}_i, \mathsf{test}_t\}, t < T,\\ \mathsf{cond}(\mathsf{end}_{t,i}) &= \{\emptyset \rhd \{\neg\mathsf{pc}_i, \mathsf{pc}_0, \neg\mathsf{test}_t, \mathsf{test}_{t+1}\}\} \cup \ \{\{\neg f\} \rhd \{f\} : f \in I^{t+1}\} \cup \ \{\{f\} \rhd \{\neg f\} : f \notin I^{t+1}\}, t < T,\\ \mathsf{pre}(\mathsf{end}_{T,i}) &= G_T \cup \{\mathsf{pc}_i, \mathsf{test}_T\},\\ \mathsf{cond}(\mathsf{end}_{T,i}) &= \{\emptyset \rhd \{\mathsf{done}\}\}.\end{aligned}$$

  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, $0 < i \leq n$.

We proceed to formally prove several properties of the compilation for basic planning programs. In particular, we show 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 Π that solves $\mathcal{P}$.*

Proof. Because of the precondition $\mathsf{ins}_{i,\mathsf{nil}}$ of program actions $P(w_i)$, we can only program an instruction w on an empty line i. Because of the precondition $\mathsf{ins}_{i,w}$ of repeat actions $R(w_i)$, we have to program an instruction before we can execute it. Hence the actions of a plan π that solves P'_n have to repeatedly program and execute instructions on program lines, and once programmed, the instruction on a line i can no longer change. As a result, the fluents in F_{ins} implicitly induce a partial 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 a planning action 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, $1 \leq t \leq T$.

As detailed above, the execution of a basic planning program Π is a deterministic process that fails only under three conditions. If the plan π is generated via our compilation none of the three conditions is feasible since classical planners check the preconditions of actions and keep track of duplicate states, preventing programs with infinite loops. 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 Figure 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 sensible to the input size.

With regard to the size of the compilation, the number of $\mathsf{go}_i^{i',f}$ actions is the dominant element, growing quadratically with the maximum number of program lines n. We introduce here an optimization of the compilation that reduces the number of $\mathsf{R}(\mathsf{go}_i^{i',f})$ actions from $|F| \cdot n^2$ to $(|F| + n) \cdot n$. The idea is to split $\mathsf{goto}_i^{i',f}$ actions 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

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

Actions for *programming* a conditional goto instruction remain the same, but the execution of a goto instruction is done using actions $R(\mathsf{eval}_i^f)$ and $R(\mathsf{jmp}_i^{i'})$.

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 [36] (FD) in the LAMA-2011 setting [37] on a Intel Core i5 3.10Ghz x 4 processor with a 4 GB memory bound and a time limit of 3600s. For *validation* we use FD in the same setting and a BFS planner from the Lightweight Automated Planning ToolKiT (LAPKT) [38] for all experiments.

We evaluate our approach in the following domains: 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 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 domains 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.

In Table 1, for each domain we report 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 solve the compiled planning instance P'_n using a classical planner, generating a number of **facts** and **operators** in the preprocessing step. The resulting plan π induces a planning program Π that solves the generalized planning problem, and we report the **search**, **preprocess** and **total** time. The facts and operators provide intuition of the size of the output planning problem P'_n.

Domain	Lines	Inst	Facts	Oper	Search	Preprocess	Total
Find	4	3	671	1044	336.27	0.72	336.99
Reverse	4	2	666	1041	244.80	1.43	246.23
Select	4	4	1028	1688	248.02	36.48	284.50
Triangular	3	2	323	324	0.52	0.59	1.11

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

[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)

Figure 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 $\sum_{n=1}^{N} n$.

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 facts and operators 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.

Figure 3 shows the resulting planning programs in the four domains, which are the same as in previous work. In 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 Reverse, a initially points to the head and b to the tail. The program repeatedly swaps the contents of a and b and move a and b towards the middle of the vector. In 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 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 Figure 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 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 BFS (compiled tests), we also compare the 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. In all cases but one, providing the planning program as input (compiled tests) allows us to solve the given planning problem more quickly. The exception is that BFS incorrectly reports No-Solution-Found (NSF) in Find; we believe this is due to the fact that the planner is not properly handling conditional effects here. In several cases the planners were unable to compute a solution within the given time limit (TE = Time-Exceeded).

Domain	**Compiled Tests**				**Classical Tests**			
	Facts	**Oper**	**FD-Total**	**BFS-Total**	**Facts**	**Oper**	**FD-Total**	**BFS-Total**
Find	164	9	1.04	NSF	427	5	TE	3.15
Reverse	404	7	TE	3.84	394	4	TE	TE
Select	168	9	7.39	0.95	264	4	TE	2.27
Triangular Number	251	6	TE	65.56	242	5	TE	90.50

Table 2: Generalized plan validation. In Compiled Tests, we compute the facts, operators and total time (in seconds) to obtain a plan for FD and BFS. In Classical Tests, we compute the facts, operators and time taken by FD and BFS to solve the instance without using the planning program.

Even though validation only involves deterministic execution of a planning program on a given instance, the preprocessing step of FD has to ground the instance, which often leads to timeouts when instances are large. As an alternative, the BFS setting of the LAPKT planner is able to solve the test problems more easily when there is only one applicable action at every time-step. In some cases, the compiled version has less facts than the classical version. Since the initial state encodes the planning program, many facts are not reachable and therefore pruned in the preprocessing step. In addition, only a few operators from the full compiled domain are going to be applicable in reachable states.

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 STRIPS 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 and $L \subseteq F$ is a subset of *local fluents*. The instruction set $\mathcal{I}$ is extended as

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

where $\mathcal{I}_{call}$ is the set of *procedure calls*, 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 now 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).

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. $(s_g, \emptyset) \wedge G \subseteq s_g$. To ensure that the execution model remains bounded we impose an upper bound ℓ on the size of the call stack. As a consequence, there is now a fourth failure condition for the execution of planning programs with procedures:

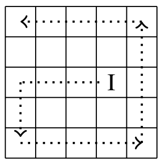
4. Execution does not terminate because, when executing a call instruction `call(j')` in program state (s_g, Σ), the size of Σ equals ℓ, i.e. $|\Sigma| = \ell$.

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 family of planning problems defined by the STRIPS frame $\Phi = \langle F, A\rangle$.

Figure 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 Figure 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. Figure 4(b) shows an example execution of the program on a planning problem whose initial state places the agent at position (4, 3).

```
Π0:  0. call(1)     Π2:  0. inc(x)              Π4:  0. dec(x)
     1. call(2)          1. goto(0,!(x=n))           1. goto(0,!(x=1))
     2. call(3)          2. end                      2. end
     3. call(4)
     4. end         Π3:  0. inc(y)
                         1. goto(0,!(y=n))
                         2. end
```

(a)



(b)

Figure 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 defined by the program in Figure 2) and an execution example of the program in a 5×5 grid starting from cell (4, 3).

5.2. Planning Programs with Variables

Apart from control flow, the use of *variables* is another powerful mechanism to compactly represent solutions that generalize. Referring to variables instead of concrete objects increases the expressiveness of actions and allows to represent more compact and general solutions.

As a case in point, consider the action `dec(x)` from the example in Figure 2, which implicitly considers x as a variable and decrements the current value of x using conditional effects. Exactly the same planning task could be defined using a set of actions of type `dec(x,v)` for each v, $1 < v \leq n$, that decrement the value of variable x from v to $v-1$ without using conditional effects. Under this alternative representation, the classical plan for solving the example problem in Figure 2 becomes ⟨`dec(x,4)`,`dec(x,3)`,`dec(x,2)`,`dec(y,3)`,`dec(y,2)`⟩. Since actions are no longer repeated in this plan, it is much less amenable to generalization.

Just as in the example above, defining actions on variables is mostly a matter of problem representation. However, in this section we introduce an explicit notation for representing variables in planning programs. As already mentioned in Section 3, we assume that the fluents of a STRIPS 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 *invariants*. All other predicates in Ψ are instantiated on value objects in Ω_x only.

In Section 6 we describe a transformation that takes as input a planning problem with object-centric actions and outputs a planning problem with variable-centric actions.

5.3. Planning Programs With Procedural Arguments

In this section we take advantage of the variable representation from the previous subsection 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.

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 $\Pi^{j'}$.

Figure 5 shows a planning program with two parameterized procedures, $\Pi^1(aux)$ and $\Pi^2(aux)$, for visiting the corners of an $n \times n$ grid, the same navigation problem introduced in Figure 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 Figure 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 Figure 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
```

Figure 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.

5.4. 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. In what follows we use PP-P to denote the class of planning programs that includes procedures. We first show that PP and PP-P are *equivalent* in the sense that each can be reduced to the other.

Lemma 5. *Planning programs with procedures are as expressive as basic planning programs, i.e. PP ⊆ PP-P.*

Proof. Let $\langle F, A \rangle$ be a planning frame and let Π be an associated basic planning program. Π is trivially identical to a planning program with procedures $\Pi' = \langle \{\Pi\}, \emptyset \rangle$, i.e. Π' has a single procedure Π which acts as the main program and no local fluents (furthermore, Π contains no call instructions). □

Lemma 6. *Basic planning programs are as expressive as planning programs with procedures, i.e. PP-P ⊆ PP.*

Proof. Given a planning problem $P = \langle F, A, I, G \rangle$ and a planning program with procedures $\Pi = \langle \{\Pi^0, \ldots, \Pi^m\}, L \rangle$, we construct a planning problem $P' = \langle F', A', I', G' \rangle$ and associated basic planning program Π' such that Π solves P if and only if Π' solves P'. Recall that program lines of Π are expressed as pairs (j, i) of a procedure Π^j, $0 \leq j \leq m$, and a line i, $0 \leq i \leq |\Pi^j|$.

The reduction idea is to use the fluents in F' to encode a program state (s_g, Σ) for Π. To do so we introduce fluents that encode the current stack level l, $0 \leq l \leq \ell$, the program line (j, i) on stack level l, and the local state $s_l \subseteq L$ on stack level l. The fluent set $F \setminus L$ that encodes the global state remains unchanged. We can now define a single action `simulate` with one conditional effect for each program line (j, i), i.e. $A' = \{\texttt{simulate}\}$. The conditional effect updates the program state according to the execution model for the instruction w_i^j on line (j, i).

The initial state I' encodes a program state defined by stack level $l = 1$, program line $(0, 0)$, local state $I \cap L$ and global state $I \cap (F \setminus L)$. The goal state G' encodes that the call stack should be empty and the goal state of P satisfied, i.e. $G' = G \cup \{l = 0\}$. The basic planning program Π' is given by

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

Since the action `simulate` repeatedly simulates the execution of the instruction on the current program line of Π, the basic planning program Π' solves P' if and only if Π solves P. □

Theorem 7. VAL(PP-P) *and* BPE(PP-P) *are* PSPACE-*complete.*

Proof. Follows directly from Lemmas 5 and 6. □

In the next subsection we describe a compilation that takes as input a generalized planning problem $\mathcal{P}$, a number of procedures m, a number of program lines n and a bound ℓ and outputs a single classical planning problem $P_{m,n}^{\ell}$ whose solution corresponds to a planning program with procedures Π that solves $\mathcal{P}$. We remark that the planning problem P' that we construct in the proof of Lemma 6 is quite similar to $P_{m,n}^{\ell}$, in that the fluents of $P_{m,n}^{\ell}$ also encode a call stack for Π. The difference is that $P_{m,n}^{\ell}$ also includes actions for *programming* the instructions of the planning program Π in the solution. Since each program line needs a separate programming action, $P_{m,n}^{\ell}$ also contains one simulate action per program line, each effectively implementing one conditional effect of the action `simulate` from the proof.

We also remark that although there is no complexity theoretic benefit of extending planning programs with procedures, in practice decomposing a planning program into procedures results in more compact planning programs, and usually increases the efficiency of a solver trying to generate a planning program. Procedures also make it possible to decompose a problem into sub-problems and compute a separate planning program for each sub-problem.

5.5. 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 [39]. 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 $F_{top}^{\ell} = \{\mathsf{top}^l\}_{0 \leq l \leq \ell}$ 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 $P^{\ell}_{n,m} = \langle F^{\ell}_{n,m}, A^{\ell}_{n,m}, I^{\ell}_{n,m}, G' \rangle$. 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:

- $F^{\ell}_{n,m} = (F \setminus L) \cup L^{\ell} \cup F^{\ell}_{top} \cup F^{\ell}_{pc} \cup F^{m}_{ins} \cup \{\mathsf{done}\}$ 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 parametized 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

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

    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, $1 \leq l < \ell$:

    $$\mathsf{pre}(\mathsf{call}^{j',l}_{i,j}) = \{\mathsf{pc}^l_i, \mathsf{ins}_{i,j,call(j')}, \mathsf{top}^l, \mathsf{proc}^l_j\}, \quad (1)$$

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

    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 $l+1$, $0 \leq l < \ell$:

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

    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:

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

  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 fluents $F \subset F^{\ell}_{n,m}$ define a new set of local fluents $\{\mathsf{assign}(v, x) : v \in \Omega_v, x \in \Omega_x\}$ 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.

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.

With regard to the compilation size the classical planning problem output by the extended compilation is larger because fluents and actions are parameterized with extra parameters: the index of the procedure and the 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 the nodes of a binary tree of arbitrary size. 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. For example, in the experimental setup we used to compute the planning program of Figure 4, the four auxiliary procedures $p1$, $p2$, $p3$ and $p4$ were given while the lines of the main procedure were empty. Reducing the number of empty program lines decreases the number of possible planning programs and hence the number of applicable actions. Evidently, 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). In the navigation example from Figure 2, to compute the auxiliary procedure $p1$, i.e., a program for navigating to position $(0, 0)$, we defined a series of planning problems with different initial states whose goal condition is to be at position $(0, 0)$. Similarly, we defined suitable planning problems to compute the remaining auxiliary procedures $p2$, $p3$ and $p4$ shown in Figure 4.

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 HGN planning where a network of subgoals is specified to boost the search of a hierarchical planner [40]. An interesting open research direction is to automatically discover these decompositions of planning problems and previous work on the automatic generation of planning hierarchies [41, 42] is a good starting point to address this research question.

5.6. *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 previous work [10]. In Blocks, 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. Figure 5). In Sorting, the aim is to sort the elements of a vector. In Trees, the aim is to visit all nodes in a 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 *Flash Fill*, a feature of Microsoft Excel that automatically programs macros from a set of 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 given as MM:SS.HH (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 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.

Just as before, we can represent some domains using the same planning frame $\Phi = \langle F, A\rangle$. For example, instances of Triangular and Fibonacci can be represented using a Variables domain that include operators that increment or decrement variables, add the value of a variable to another, and assign the value of a variable to another. Likewise, instances of Hall-A and Visit-all can be represented using a generic Grid domain that includes operators for visiting cells and move in any cardinal direction. All Excel problems are also represented using the same domain.

In each domain except for Trees, plan generation proceeds in several steps. The overall problem is decomposed into two or more subproblems, and for each subproblem we provide a generalized planning problem whose instances correspond to that subproblem. When generating $P^{\ell}_{n,m}$ for a subproblem, we include the solutions to previous subproblems as procedures, i.e. only the program lines of the main program are empty in the initial state, while the program lines of all procedures are already programmed. In Trees, the problem is not decomposed, but the resulting main program makes recursive calls to itself.

Table 3 shows the results for plan generation. Compared to Table 1, we have added bounds on the number of procedures. 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,

Domain	Proc	Lines	Inst	Stack	Facts	Oper	Search	Preprocess	Total
Blocks	2	(4,3)	(6,5)	(2,2)	(306,250)	(932,559)	(0.90,0.04)	(2.63,0.53)	4.1
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

Table 3: Plan generation for planning programs with procedures. Number of procedures; for each procedure: program lines, instances, stack size, facts, operators, search time, preprocessing time; total time (in seconds) elapsed while computing the overall solution.

number of facts, number of operators, search time, and preprocessing time. We also report the total time to solve all subproblems related to a domain.

As an illustration, we show the resulting planning programs with procedures for four domains: Gripper, Sorting, Trees and Excel. The solution to **Gripper** appears in Figure 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 all actions are implemented using conditional effects in order for the planning program to generalize.

```
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

Figure 6: Generalized plan in the for of *planning program with procedures* for the Gripper domain.

The solution to **Sorting** appears in Figure 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 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`.

```
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`

Figure 7: Planning program with procedures for Sorting.

In **Trees**, there are two variables `current` and `child` that point to nodes of the tree. Figure 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.

```
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
```

Figure 8: Planning program with recursion for Trees.

The planning programs for **Excel** are shown in Figure 9. Each string has two indices lo and hi. Each of the five problems 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.

```
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

Figure 9: Planning programs for Excel, where Π^1 is a common procedure.

Compared to previous work [10], the results in Fibonacci are better since we needed 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 domain Grid.

Just as for basic planning programs, we ran a second set of experiments in which we validated the generated planning programs. In Blocks, we tested the planning program on a tower of 100 blocks where the green block was third from 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.

In Table 4 we present results from the same experiments as in the last section with the FD and BFS planners. The reported statistics include the number of facts and operators, and the total time FD and BFS needs to compute a solution. The statistics are obtained from two different versions of the instance: 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. As before, Time-Exceeded (TE) indicates that no solution was found within the given time limit, and No-Solution-Found (NSF) indicates that a planner (incorrectly) concluded that an instance was unsolvable.

Domain	**Compiled Tests**				**Classical Tests**			
	Facts	**Oper**	**FD-Total**	**BFS-Total**	**Facts**	**Oper**	**FD-Total**	**BFS-Total**
Blocks	834	13	3.11	3.07	804	3	1.47	1.42
Excel-1	656	9	0.48	0.49	792	56	0.53	0.69
Excel-2	176	9	0.96	0.22	206	56	0.11	0.14
Excel-3	1208	10	1.29	1.38	1480	106	3.27	5.99
Excel-4	826	10	1.19	1.14	1000	106	1.50	1.76
Excel-5	152	4	0.68	0.69	856	106	1.18	1.16
Fibonacci	71	11	TE	0.34	56	8	TE	0.24
Gripper	342	14	0.86	NSF	308	5	0.55	0.39
Hall-A	20442	45	7.39	318.69	20400	5	2.29	TE
Sorting	2820	17	TE	1545.50	-	-	TE	NSF
Trees	1294	178	0.90	0.75	118	10	0.12	NSF
Visit-all	2080	18	8.09	25.27	2046	5	TE	TE
Visual-M	84	21	0.30	0.50	48	5	0.27	NSF

Table 4: Plan validation for planning programs with procedures. In Compiled Tests, we compute the facts, operators and total time (in seconds) to obtain a plan for FD and BFS. In Classical Tests, we compute the facts, operators and time taken by FD and BFS to solve the instance without using the planning program.

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. However, the resulting solution does not generalize and is much less compact than the planning program, often exceeding 100 operators. In Sorting and Visit-all, no planner is able to solve the planning instance without the given planning program.

6. Non-Deterministic Planning Programs

Defining a solution that is valid for multiple planning instances while compact, may be complex and sometimes even unfeasible. Sometimes the instances in the generalized planning task do not share a clear common structure and need to be treated separately. An alternative approach for such scenarios is to represent and compute solutions with non-deterministic execution [8], i.e. solutions with open segments that are only determined when the solution is executed on a particular instance. This section introduces *choice instructions* and *lifted sequential instructions*, two different extensions of our *planning program* formalism to achieve generalization through non-deterministic execution. The section describes both how to represent and compute solutions of this kind.

6.1. Planning Programs with Choice Instructions

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

The extension is based on the formalism for *planning programs with variables* defined in Section 5: for any planning frame $\Phi = \langle F, A\rangle$, fluents in F 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 the following set of choice instructions:

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

A choice instruction $\texttt{choose}(\omega, p)$ assigns a value to each of the variables in ω. These values result from a unification of the predicate p with the current state, which means that a fluent instantiated by assigning those values to p holds in the state where the choice instruction is executed. In general, 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 model for a choice instruction w_i is defined for basic planning programs as follows (we assume that the current program state is (s, i) and that w_i is the instruction on the current line i):

- If w_i = `choose`(ω, 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 a unification of p with the current state. The second step assigns to each variable $v \in \omega$ the corresponding argument value from the chosen unification. The assignment is done making fluent $\mathsf{assign}(v, x)$ true and fluents $\mathsf{assign}(v, x')$ false for each value object $x' \neq x$ that is not in the chosen unification. Other than this assignment, s' is identical to s.

A basic planning program with choice instructions introduces a fourth failure condition:

4. Execution does not terminate because, when executing a choice instruction `choose`(ω, p), we cannot unify predicate p with the current state.

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 instance of BLOCKSWORLD for which the goal is to put all the blocks on the table. This generalized planning task is more complex than unstacking a single tower of blocks, a task commonly solved by FSCs [2], because there can now be an arbitrary number of towers, each with different height. A compact generalized plan that solves this task can be defined using one choice instruction:

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

The choice instruction `choose(v,clear)` assigns to the variable v a block object that is currently clear and captures the key knowledge of *the block to move next*. This block is then unstacked and put down on the table. The derived predicate `all-clear` 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).

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 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:

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

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 Sequential Instructions

Here we go one step further and increase the part of a planning program that is not specified. A *planning program with lifted sequential instructions* is a planning program in which sequential instructions have unknown arguments until the instruction is executed in a particular planning instance.

To define the execution of *lifted sequential instructions* we assume that the actions of a planning frame $\Phi = \langle F, A\rangle$ are instantiated from a set of action schemes $\mathcal{A}$ like in PDDL. For example, in BLOCKSWORLD, the action scheme `unstack(?b1,?b2)` defines the preconditions and effects for unstacking any block `?b1` from any other block `?b2`. For simplicity, we describe the execution model of a lifted sequential instruction for the program state (s, i) of a basic planning program, although we could also include such instructions in planning programs with procedures.

- If $w_i \in \mathcal{A}$, the new program state is $(s', i+1)$, where $s' = \theta(s, w_i)$ is the result of applying an instantiation $a \in A$ of w_i that is chosen non-deterministically among the instantiations of w_i that are applicable in the current state, i.e. $pre(a) \subseteq s$. The program counter is also incremented as in the basic formalism for planning programs.

Domain	Proc	Lines	Inst	Stack	Facts	Oper	Search	Preprocess	Total
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
Blocks	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)	(0.37,0.01,4.80)	(2.24,0.29,4.86)	12.57
Blocksworld	2	(3,4)	(3,2)	2	(214,214)	(651,651)	0.04	0.82	0.86

Table 5: Plan generation for non-deterministic planning programs. Number of procedures; for each procedure: number of lines and instances, stack size, number of facts and operators, search time and preprocessing time; total time (in seconds) elapsed while computing the solution.

For example, a planning program with lifted sequential instructions that puts all the blocks on the table can be defined for the original definition of the BLOCKSWORLD domain. In this program, each execution of `unstack(?b1,?b2)` has to non-deterministically assign concrete block objects to the parameters `?b1` and `?b2` among all the possible applicable instantiations.

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

According to our definition, a planning program with lifted sequential instructions is no longer a generalized plan, but a piece of Domain-specific Control Knowledge (DCK) [43, 34]. The reason is that, similar to *HTNs* [44], a planning program with lifted sequential instructions requires a planner to compute a fully specified solution, constraining the form of the solution by pruning the actions that do not agree with the program. Furthermore we can also specify hierarchies of programs with the *call stack* mechanism explained in Section 5.

Unlike HTNs, we can exploit this form of DCK in a straightforward way with an off-the-shelf classical planner. A program with lifted actions can be computed and executed by introducing a small modification in the compilation in Section 4 for the computation of basic planning programs. The modification is a redefinition of the classical planning actions for programming and executing sequential instructions. Given an instantiation $a \in A$ of a lifted action in $w_i \in \mathcal{A}$, the actions for programming and executing a lifted sequential instruction w_i are defined as follows:

$$\begin{aligned}
\mathsf{pre}(\mathsf{P}(w_i, a)) &= \mathsf{pre}(a) \cup \{\mathsf{ins}_{i,\mathsf{nil}}\},\\
\mathsf{cond}(\mathsf{P}(w_i, a)) &= \{\emptyset \rhd \{\neg\mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,w_i}\}\},\\
\mathsf{pre}(\mathsf{R}(w_i, a)) &= \mathsf{pre}(a) \cup \{\mathsf{ins}_{i,w_i}\},\\
\mathsf{cond}(\mathsf{R}(w_i, a)) &= \mathsf{cond}(a).
\end{aligned}$$

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, sequential instructions have parameters, so the resulting planning programs include lifted instructions. In previous experiments, because we were using parameter-free domains, the execution of planning programs was deterministic. In the new domains with parameterized instructions, the execution of a planning program requires a planner that selects the values of the action parameters each time a sequential instruction is executed.

Apart from existing domains, we added the Blocksworld domain. Our approach is to first generate a procedure that puts all the blocks on the table, and then generate a second procedure that non-deterministically chooses a block on the table to stack on another block. The goal state in Blocksworld usually tests if blocks are well placed (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 5 shows the results of the experiments with non-deterministic planning programs. The table includes the same information as Table 3: the number of procedures; for each procedure: number of program lines, number of instances, stack size, number of facts and operators, search time and preprocessing time; the total time to solve the overall problem. Note that the planner fails to solve even 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 sequential instructions are lifted, the planner has to assign objects to action parameters and perform search to reach the goal. Figure 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.

```
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

Figure 10: Non-deterministic planning program for Blocksworld.

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, lifted instructions make it possible to compute planning programs for some domains that are not solvable by deterministic planning programs, such as Blocksworld.

For validation, we used the same instances as in Section 5. For Blocksworld, we used an instance with 100 blocks distributed into 8 towers of 10 blocks and 1 tower of 20 blocks, with a goal state that has 10 towers of 5 blocks and 1 with 50 blocks. Table 6 reports results from the same experiments described in Table 4, but with non-deterministic DCK instead of execution of determinstic planning programs with procedures. The table includes the domain that we want to validate, and statistics such as the number of facts and operators, and total time that FD and BFS needed to compute a plan. As before, 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).

Domain	**Compiled Tests**				**Classical Tests**			
	Facts	**Oper**	**FD-Total**	**BFS-Total**	**Facts**	**Oper**	**FD-Total**	**BFS-Total**
Hall-A	-	-	-	-	20200	10396	1.64	TE
Fibonacci	74	481	0.29	0.008	56	8	TE	0.02
Visit-all	2018	1065	9.79	26.43	1984	1081	3.91	158.02
Blocks	735	20209	7.63	54.01	705	399	0.94	0.83
Sorting	258	26073	TE	33.58	241	26060	186.32	TE
Triangular Number	151	1835	0.31	0.012	242	7740	1.35	0.02
Visual-M	62	48	0.33	0.37	26	41	0.02	NSF
Blocksworld	10540	20009	112.45	NSF	10504	20000	269.59	TE

Table 6: Plan validation of non-deterministic planning programs. In Compiled Tests, we compute the facts, operators and total time (in seconds) to obtain a plan for FD and BFS. In Classical Tests, we compute the facts, operators and time taken by FD and BFS to solve the instance without using the planning program.

Table 6 shows the results of plan validation for the different domains. Although search is constrained because of the planning program, BFS usually performs worse than FD. Only in Fibonacci is it better to use BFS, because of the preprocessing required for the classical instance. In contrast, the Blocksworld domain provides one example of how complex it is to solve the classical instance versus the compiled instance with a non-deterministic planning program.

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 [45]. Likewise in generalized 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 the fluents $x = n$, $y = n$ or $aux = n$

$$\begin{aligned}\mathsf{equal}(x,n) &\leftarrow \exists v_3.\mathsf{assign}(x,v_3) \wedge \mathsf{assign}(n,v_3),\\ \mathsf{lt}(x,n) &\leftarrow \exists v_3,v_4.\mathsf{assign}(x,v_3) \wedge \mathsf{assign}(n,v_4) \wedge \mathsf{lessthan}(v_3,v_4).\end{aligned}$$

Figure 11: Two derived predicates in the form of conjunctive queries that correspond to the high-level state features $a = b$ and $a < b$, respectively.

from the programs in Figures 4 and 5 that are not needed for solving an individual grid navigation task with explicit goal state, but are essential to represent and compute a solution 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 generate programs with conjunctive queries using 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 [46] to description logic formulae [7], LTL formulae [47], PDDL derived predicates [48] and, more recently, observation formulae [2].

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 [49] and for more efficiently solving optimal planning problems [50].

For computation purposes in this work we assume that the set of possible high-level state features is restricted to arbitrary formulae over predicates to *conjunctive queries* from database theory [51]. 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

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

where $v_1, \dots, v_k$ are *free variables*, $v_{k+1}, \dots, v_m$ are *bound variables*, and $\phi_1, \dots, \phi_q$ 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 $v_{k+1}, \dots, v_m$ 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.

Figure 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 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 is considered to 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 (Figure 4). Consider also the derived predicate $\varphi = \mathsf{equal}(x, n)$ from Figure 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 $P_{n,q}^b = \langle F_{n,q}^b, A_{n,q}^b, I_{n,q}^b, G_n' \rangle$. A solution plan π to $P_{n,q}^b$ is a sequence of actions that encodes a planning program Π with conjunctive queries and validates that the execution of Π solves every instance in $\mathcal{P}$.

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

- For each pair of program lines i, i' such that $i' \neq i + 1$, $F_{n,q}^b$ contains a fluent $\mathsf{ins}_{i,\mathsf{goto}(i')}$ indicating that the instruction on line i is a goto instruction with a conjunctive query, $goto(i', !\varphi)$.
- 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_i^k 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 $\mathsf{atom}\text{-}p_i^k(\omega)$ 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 $(o_1, \ldots, o_b) \in \Omega_x^b$, 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

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

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

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

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

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

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

$$\begin{aligned}\mathsf{pre}(\mathsf{eval}_i) &= \{\mathsf{pc}_i, \mathsf{slot}^q\},\\ \mathsf{cond}(\mathsf{eval}_i) &= \{\emptyset \rhd \{\mathsf{eval}\}\} \cup \{\{\mathsf{poss}^q(o_1, \ldots, o_b)\} \rhd \{\mathsf{acc}\} : (o_1, \ldots, o_b) \in \Omega^b_x\}.\end{aligned}$$

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

$$\begin{aligned}\mathsf{pre}(\mathsf{jmp}_{i,i'}) &= \{\mathsf{pc}_i, \mathsf{ins}_{i,\mathsf{goto}(i')}, \mathsf{slot}^q, \mathsf{eval}\},\\ \mathsf{cond}(\mathsf{jmp}_{i,i'}) &= \{\emptyset \rhd \{\neg\mathsf{pc}_i, \neg\mathsf{eval}, \neg\mathsf{acc}, \neg\mathsf{slot}^q, \mathsf{slot}^0\}\}\\ &\quad \cup \ \{\{\neg\mathsf{acc}\} \rhd \{\mathsf{pc}_{i'}\}, \{\mathsf{acc}\} \rhd \{\mathsf{pc}_{i+1}\}\}\\ &\quad \cup \ \{\emptyset \rhd \{\neg\mathsf{poss}^k(o_1, \ldots, o_b) : 1 \le k \le q, (o_1, \ldots, o_b) \in \Omega^b_x\}\}.\end{aligned}$$

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 t^{th} example:

- F comprises the set of fluents required for representing the learning examples and their labels.
- A contains the actions necessary to associate a given example with a class in C. For instance, in a binary classification task $|C| = 2$ and $A = \{\mathsf{setPositive}, \mathsf{setNegative}\}$.
- I_t contains the fluents that describe the t^{th} example while G_t is the fluent that defines the label of the t^{th} example.

According to this formulation the 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.

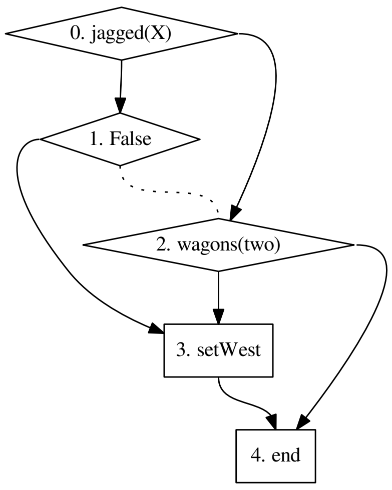


Figure 12: Planning program that encodes a noise-free classifier for the Michalski's train problem.

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* [52] 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 finding the rules that cause 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.

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

- Fluents F induced from Ψ = {(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.

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

- Each initial state I_t defines the t^{th} train and G_t defines its associated class (traveling east or traveling west).

Figure 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 ← jagged(X) and setWest ← ¬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 10 (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 $\mathsf{poss}^k(o_1,\ldots,o_b)$ 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 $\mathsf{poss}^{k-1}(o_1,\ldots,o_b)$ 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-$p^k_i(\omega)$, $\mathsf{poss}^k(o_1,\ldots,o_b)$ becomes true if and only if $\mathsf{poss}^{k-1}(o_1,\ldots,o_b)$ 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 11 (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 Figure 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 $P^b_{n,q}$.

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.

Domain	Lines	Vars	Slots	Facts	Oper	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

Table 7: Plan generation for planning programs with high-level state features. Number of program lines, variables and slots; number of facts and operators; search, preprocess and total time (in seconds) elapsed while computing the solution.

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 7 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 Figure 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 and presented a compilation to compute generalized plans with off-the-shelf planners. 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. 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.

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, e.g. ignoring the delete effects of programming instructions makes it possible to simultaneously program multiple instructions on the same line. In addition, a program suitable for a particular individual planning problem in a generalized planning task might not be suitable for solving the next problem, 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 goals [53]. 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 a given 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 [54] and active learning [55] 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.

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, Artificial Intelligence 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. Mller, J. Schaeffer, Macro-ff: Improving ai planning with automatically learned macro-operators, Journal of Artificial Intelligence Research 24 (2005) 581–621.

[13] A. Coles, A. Smith, Marvin: A heuristic search planner with online macro-action learning., J. Artif. Intell. Res.(JAIR) 28 (2007) 119–156.

[14] A. Jonsson, The role of macros in tractable planning, Journal of Artificial Intelligence Research (2009) 471–511.

[15] S. Yoon, A. Fern, R. Givan, Learning control knowledge for forward search planning, The Journal of Machine Learning Research 9 (2008) 683–718.

[16] T. De la Rosa, S. Jiménez, R. Fuentetaja, D. Borrajo, Scaling up heuristic planning with relational decision trees, Journal of Artificial Intelligence Research (2011) 767–813.

[17] J. Rintanen, Planning as satisfiability: Heuristics, Artificial Intelligence Journal 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 Computing Surveys (CSUR) 47 (2) (2015) 35.

[21] S. Srivastava, N. Immerman, S. Zilberstein, A new representation and associated algorithms for generalized planning, Artificial Intelligence 175 (2) (2011) 615 – 647.

[22] A. Albore, H. Palacios, H. Geffner, A translation-based approach to contingent planning, in: International Joint Conference on Artificial Intelligence, 2009.

[23] S. Gulwani, Automating string processing in spreadsheets using input-output examples, in: ACM SIGPLAN Notices, Vol. 46, ACM, 2011, pp. 317–330.

[24] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat, Combinatorial sketching for finite programs, ACM SIGOPS Operating Systems Review 40 (2006) 404–415.

[25] B. M. Lake, R. Salakhutdinov, J. B. Tenenbaum, Human-level concept learning through probabilistic program induction, Science 350 (6266) (2015) 1332–1338.

[26] T. M. Mitchell, Generalization as search, Artificial intelligence 18 (2) (1982) 203–226.

[27] S. Muggleton, Inductive logic programming: issues, results and the challenge of learning language in logic, Artificial Intelligence 114 (1) (1999) 283–296.

[28] C. Boutilier, R. Reiter, B. Price, Symbolic dynamic programming for first-order mdps, in: IJCAI, Vol. 1, 2001, pp. 690–700.

[29] 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.

[30] H. Palacios, H. Geffner, Compiling uncertainty away in conformant planning problems with bounded width, Journal of Artificial Intelligence Research 35 (2009) 623–675.

[31] M. Fox, D. Long, Pddl2. 1: An extension to pddl for expressing temporal planning domains., J. Artif. Intell. Res.(JAIR) 20 (2003) 61–124.

[32] 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.

[33] A. Fern, R. Khardon, P. Tadepalli, The first learning track of the international planning competition, Machine Learning 84 (1-2) (2011) 81–107.

[34] S. Jiménez, T. De La Rosa, S. Fernández, F. Fernández, D. Borrajo, A review of machine learning for automated planning, The Knowledge Engineering Review 27 (04) (2012) 433–467.

[35] T. Bylander, The Computational Complexity of Propositional STRIPS Planning, Artificial Intelligence 69 (1994) 165–204.

[36] M. Helmert, The Fast Downward Planning System, Journal of Artificial Intelligence Research 26 (2006) 191–246.

[37] S. Richter, M. Westphal, The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks, Journal of Artificial Intelligence Research 39 (2010) 127–177.

[38] M. Ramirez, N. Lipovetzky, C. Muise, Lightweight Automated Planning ToolKiT, `http://lapkt.org/`, accessed: 2016-12-19 (2015).

[39] C. Domshlak, Fault tolerant planning: Complexity and compilation, in: Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS, 2013.

[40] 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-Volume 2, International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 981–988.

[41] C. Hogg, H. Munoz-Avila, U. Kuter, Htn-maker: Learning htns with minimal additional knowledge engineering required., in: AAAI, 2008, pp. 950–956.

[42] D. Lotinac, A. Jonsson, Constructing hierarchical task models using invariance analysis, in: European Conference on Artificial Intelligence, 2016.

[43] T. Zimmerman, S. Kambhampati, Learning-assisted automated planning: looking back, taking stock, going forward, AI Magazine 24 (2) (2003) 73.

[44] 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.

[45] C. M. Bishop, Pattern recognition, Machine Learning 128.

[46] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, J. Blythe, Integrating planning and learning: The prodigy architecture, Journal of Experimental & Theoretical Artificial Intelligence 7 (1) (1995) 81–120.

[47] S. Cresswell, A. M. Coddington, Compilation of ltl goal formulas into pddl., in: ECAI, 2004, pp. 985–986.

[48] J. Hoffmann, S. Edelkamp, The deterministic part of ipc-4: An overview, Journal of Artificial Intelligence Research (2005) 519–579.

[49] S. Thiébaux, J. Hoffmann, B. Nebel, In defense of pddl axioms, Artificial Intelligence 168 (1) (2005) 38–69.

[50] F. Ivankovic, P. Haslum, Optimal planning with axioms, in: International Joint Conference on Artificial Intelligence, AAAI Press, 2015, pp. 1580–1586.

[51] 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.

[52] R. S. Michalski, J. G. Carbonell, T. M. Mitchell, Machine learning: An artificial intelligence approach, Springer Science & Business Media, 2013.

[53] 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.

[54] A. Fern, S. W. Yoon, R. Givan, Learning domain-specific control knowledge from random walks., in: ICAPS, 2004, pp. 191–199.

[55] R. Fuentetaja, D. Borrajo, Improving control-knowledge acquisition for planning by active learning, in: Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings, 2006, pp. 138–149.