Generalized Linear Integer Numeric Planning

Xiaoyou Lin[1], Qingliang Chen[1], Liangda Fang[1,3,4*], Quanlong Guan[1,2*], Weiqi Luo[1], Kaile Su[5]

[1] Jinan University, Guangzhou 510632, China
[2] Guangdong Institute of Smart Education, Guangzhou 510632, China
[3] Pazhou Lab, Guangzhou 510330, China
[4] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China
[5] Griffith University, Brisbane 4111, Australia
linxiaoyou@stu2019.jnu.edu.cn; {tpchen, fangld, gql, lwq}@jnu.edu.cn; kailepku@gmail.com

Abstract

Classical planning aims to find a sequence of actions that guarantees goal achievement from an initial state. The representative framework of classical planning is based on propositional logic. Due to the weak expressiveness of propositional logic, many applications of interest cannot be formalized as classical planning problems. Some extensions such as numeric planning and generalized planning (GP) are therefore proposed. Qualitative Numeric Planning (QNP) is a decidable class of numeric and generalized extensions and serves as a numeric abstraction of GP. However, QNP is still far from being perfect and needs further improvement. In this paper, we introduce another generalized version of numeric planning, namely Generalized Linear Integer Numeric planning (GLINP), which is a more suitable abstract framework of GP than QNP. In addition, we develop a general framework to synthesize solutions to GLINP problems. Finally, we evaluate our approach on a number of benchmarks, and experimental results justify the feasibility and effectiveness of our proposed approach.

Introduction

Along the AI history, the planning community has focused on classical planning that identifies a sequence of actions that guarantees goal achievement from an initial state. The representative framework of classical planning is based on propositional logic. Due to the limited expressiveness of propositional logic, many applications of interest cannot be formalized as a classical planning problem. Therefore, some extensions to classical planning are proposed. One extension is numeric planning (Hoffmann 2003; Scala, Haslum, and Thiébaux 2016), which involves not only propositional variables but also numeric variables. Another extension is generalized planning (GP) (Levesque 2005; Srivastava, Immerman, and Zilberstein 2011), which solves planning problems for possibly infinitely many initial states rather than a single state. However, the above two extensions are in general undecidable (Helmert 2002; Levesque 2005).

Srivastava et al. (2011) proposed a decidable class of numeric and generalized extensions to classical planning, namely qualitative numeric planning (QNP). (1) any formula is a simple numeric formula, *i.e.*, a Boolean combination of simple numeric literals that are of the form $v > 0$ or $v = 0$; and (2) the value of variables is increased or decreased by an arbitrary positive amount. Under these two restrictions, the state space of a QNP problem can be compressed into a finite one with size $2^{|V|}$ where $|V|$ is the number of numeric variables, and thus QNP is decidable, more precisely, EXPTIME-Complete (Bonet and Geffner 2020).

In general, GP is formalized by a representative framework based on first-order logic (sometime involves the transitive closure) (Hu and Levesque 2010; Srivastava, Immerman, and Zilberstein 2011). The formalization therefore contains not only various numeric variables but also a number of predicates. Hu and Levesque (2010) observed that the solutions to many GP problems are to iteratively execute a unified plan for similar objects satisfying the same property. This was also observed in other literature (Winner and Veloso 2003; Srivastava, Immerman, and Zilberstein 2011; Illanes and McIlraith 2019). The concept language is able to capture the property of various objects. Although QNPs do not allow predicate symbols, many GP problem can be reformulated as a QNP problem by exploiting concept language (Bonet and Geffner 2018).

As an abstract framework for GP, QNPs however have several flaws. (1) QNPs do not support conditional effects and only allow simple numeric formulas. As a result, QNPs are less expressive so that they cannot formalize some domains. (2) The solution to QNPs is a policy that is in fact a loop structure of several conditional statements. Some useful and skillful loop structures are hidden in a policy and it is difficult to capture the intuition of the policy. (3) The definition of the termination property of policies is unintuitive and difficult to understand, requiring the notion of fairness, that is, infinite occurrences of an action must result in infinite occurrences of each one of its possible (non-deterministic) outcomes.

To address the above deficits, we propose a generalized version of numeric planning, namely generalized linear integer numeric planning (GLINP), with the following merits. (1) The initial and goal states and actions are represented in linear integer arithmetic. The effects of actions are allowed to be conditional. Thus, GLINP formalizes a wider range of planning domains than QNP. (2) To capture the notion of solutions to GLINP, we introduce an algorithmic-like struc-

*Both are corresponding authors

ture, namely planning programs, in which loop structures explicitly occur. In addition, planning programs turn out to be a more compact form of solutions than policies (Lang and Zanuttini 2013). (3) The semantics of planning programs is defined in terms of action sequences. Hence, we obtain a clear definition of the termination property of planning programs via the finiteness property of the action sequence.

In addition, we develop an inductive approach to synthesizing planning programs. It firstly generates a representative set of initial states together with their sequential plans. Based on regular expression inference, it then infers a skeleton of the planning programs that is a planning program without conditions of branch and loop structures. Finally, the conditions are completed according to the trace of each state-plan pair. We further evaluate our approach on several benchmarks originated from generalized planning and qualitative numeric planning. The experimental results justify the feasibility and effectiveness of the proposed approach.

Preliminaries

In this section, we first introduce the concepts of linear integer arithmetic with propositional logic ($\texttt{LIA}^\texttt{P}$), and regular expressions (regexes).

$\texttt{LIA}^\texttt{P}$ Let $\mathbb{B}$ be the set of Booleans constants $\{\top, \bot\}$ and $\mathbb{Z}$ is the set of integers. Throughout this paper, we fix a set $\mathcal{P}$ of propositional variables and a set $\mathcal{V}$ of numeric variables. The sets of *terms* ($\texttt{Term}$) and *formulas* ($\texttt{Form}$) of $\texttt{LIA}^\texttt{P}$ are defined recursively as:

$$e \in \texttt{Term} :: c \mid v \mid e + e \mid e - e$$

$$\phi \in \texttt{Form} :: \top \mid \bot \mid p \mid e = e \mid e < e \mid \neg\phi \mid \phi \wedge \phi$$

where $c \in \mathbb{Z}$ and $v \in \mathcal{V}$.

The formula $\phi_1 \vee \phi_2$ is the shorthand for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $e_1 \leq e_2$ for $e_1 = e_2 \vee e_1 < e_2$, and $e_1 \rightarrow e_2$ for $\neg e_1 \vee e_2$. The length $|\phi|$ of a formula ϕ is the number of occurrences of Boolean constants, integers, propositional variables, numeric variables, arithmetic operators and logical connectives in ϕ.

A state s of $\texttt{LIA}^\texttt{P}$ is a pair of mappings $\mathcal{P} \rightarrow \mathbb{B}$ and $\mathcal{V} \rightarrow \mathbb{Z}$. The notions $v(s)$ and $p(s)$ denote the value of numeric variables v and propositional variables p in s, respectively. Given a state s, we evaluate a term e into an integer $e(s)$ to which the expression simplifies when substituting every numeric variable v with their respective value $v(s)$. The Boolean value $\phi(s)$ of a formula ϕ can be determined in a similar way. A state s *satisfies* a formula ϕ, if $\phi(s) = \top$. A formula ϕ *entails* another one ψ, denoted by $\phi \models \psi$, if for every state s satisfying ϕ, $\psi(s) = \top$. A propositional variable p is *determined by* a formula ϕ, if $\phi \models p$ or $\phi \models \neg p$. A numeric variable v has a *maximum* c under ϕ, if $\phi \models v \leq c$ and for every integer c' s.t. $v \leq c'$, we have $c \leq c'$.

Regexes A *string* is a finite sequence of characters over an alphabet Δ. We use $|\pi|$ for the length of π and π_i for the i-th character of π. A *substring* of π is $\pi_i\pi_{i+1}\cdots\pi_j$, denoted by π_i^j, where $1 \leq i \leq j \leq |\pi|$. The substring π_1^i is a *prefix* of π while $\pi_i^{|\pi|}$ is a *suffix*. A *subsequence* of π is $\pi_{i_1}\pi_{i_2}\cdots\pi_{i_j}$ where $1 \leq i_1 < i_2 < \cdots < i_j \leq |\pi|$. We remark that a substring is a subsequence, but not vice versa. For example, aba is a subsequence of $abbba$, but it is not a substring.

The set of *regexes* ($\texttt{Reg}$) is recursively defined as:

$$r \in \texttt{Reg} :: \varepsilon \mid a \mid r \circ r \mid (r|r) \mid r^*$$

where ε denotes the empty string and $a \in \Delta$.

The regex $r_1 \circ r_2$ is called a *concatenation regex*, $r_1|r_2$ an *alternation regex*, and r^* an *iteration regex*. We say r is the *generator* of an iteration regex r^*.

The set of *subregexes* $\texttt{SubReg}(r)$ of a regex r is recursively defined as:

- $\texttt{SubReg}(\varepsilon) = \{\varepsilon\}$ and $\texttt{SubReg}(a) = \{a\}$ where $a \in \Delta$;
- $\texttt{SubReg}(r_1 \circ r_2) = \{r_1 \circ r_2\} \cup \texttt{SubReg}(r_1) \cup \texttt{SubReg}(r_2)$;
- $\texttt{SubReg}(r_1|r_2) = \{r_1|r_2\} \cup \texttt{SubReg}(r_1) \cup \texttt{SubReg}(r_2)$;
- $\texttt{SubReg}(r_1^*) = \{r_1^*\} \cup \texttt{SubReg}(r_1)$.

The set of strings $\mathcal{L}(r)$ accepted by a regex r is recursively defined as:

- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$ and $\mathcal{L}(a) = \{a\}$ where $a \in \Delta$;
- $\mathcal{L}(r_1 \circ r_2) = \{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1) \text{ and } s_2 \in \mathcal{L}(r_2)\}$;
- $\mathcal{L}(r_1|r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$;
- $\mathcal{L}(r_1^*) = \{s_1 \cdots s_n \mid n \geq 1 \text{ and } s_1, \cdots, s_n \in \mathcal{L}(r_1)\}$.

Generalized Linear Integer Numeric Planning

In this section, we first introduce concepts on Linear Integer Numeric Planning (LINP) formalized in $\texttt{LIA}^\texttt{P}$, and then provide the definition of GLINP, and finally give an algorithmic-like definition of solutions to GLINP problems.

Definition 1. A LINP domain $\mathcal{D}$ is a tuple $\langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$ where

- $\mathcal{P}$: a finite set of propositional variables;
- $\mathcal{V}$: a finite set of numeric variables;
- $\mathcal{A}$: a finite set of actions defined by a pair $\langle \texttt{pre}, \texttt{eff} \rangle$ where $\texttt{pre} \in \texttt{Form}$ denotes the precondition and $\texttt{eff}$ is an finite set of propositional and numeric effects.

A *propositional effect* is a tuple $\langle \phi, p, \psi \rangle$ where $p \in \mathcal{P}$ and $\phi, \psi \in \texttt{Form}$. Intuitively, it means that if ϕ holds in a state s, then the Boolean value of p becomes $\psi(s)$ after performing the action; otherwise, it remains unchanged. A *numeric effect* is a tuple $\langle \phi, v, e \rangle$ where $\phi \in \texttt{Form}$, $v \in \mathcal{V}$ and $e \in \texttt{Term}$. The meaning of numeric effects is similar to that of propositional effect. We require that every action is not self-contradictory, that is, it is impossible that there is a propositional variable p (resp. numeric variable v) s.t. in a state s, there are two effects e and e' that both occur and the value of p (resp. v) following e is different from e'. An effect is *unconditional*, if $\phi = \top$.

An action a is *executable* in a state s, if $s \models \texttt{pre}(a)$. The *successor state* of applying an action a over s is written as $\tau(s, a)$, which results from s by mapping p to $\phi(s)$ (*i.e.*, $p(\tau(s, a)) = \phi(s)$) for every $\langle \phi, p, \psi \rangle \in \texttt{eff}(a)$, and by mapping v to $e(s)$ (*i.e.*, $v(\tau(s, a)) = e(s)$) for all $\langle \phi, v, e \rangle \in \texttt{eff}(a)$. We remark that $\tau(s, a)$ is well-defined even if a is not executable in s. The *resulting state* of performing a finite sequence $[a_1, \cdots, a_n]$ of actions on s is recursively defined by $\tau(s, [a_1, \cdots, a_n]) = \tau(\tau(s, [a_1, \cdots, a_{n-1}]), a_n)$

and $\tau(s, \varepsilon) = s$ where ε is an empty sequence. A sequence $[a_1, a_2, \cdots]$ of actions is *executable* in a state s, if $s \models \texttt{pre}(a_1)$ and $\tau(s, [a_1 \cdots a_i]) \models \texttt{pre}(a_{i+1})$ for $i \geq 1$.

A LINP problem is defined as a tuple $\langle \mathcal{D}, s_\mathcal{I}, \mathcal{G} \rangle$ where $\mathcal{D}$ is an LINP domain, $s_\mathcal{I}$ is an initial state, and $\mathcal{G} \in \texttt{Form}$ denotes a set of goal states. A solution to an LINP problem, namely *sequential plan*, is a finite sequence $[a_1, \cdots, a_n]$ of actions such that performing these actions one by one from $s_\mathcal{I}$ leads to a goal state. More formally, $[a_1, \cdots, a_n]$ is executable in s, and $\tau(s, [a_1, \cdots, a_n]) \models \mathcal{G}$. We say (s, π) a *state-plan pair* where π is a sequential plan for s. Given a set Υ of state-plan pairs, we use $S(\Upsilon)$ for the set of states and $\Pi(\Upsilon)$ for the set of plans.

Generalized LINP (GLINP) problems are an extension to LINP problems that involve a possibly infinite number of initial states which are represented by a $\texttt{LIA}^\texttt{P}$-formula $\mathcal{I}$.

Definition 2. A generalized LINP (GLINP) problem Σ is a tuple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where

- $\mathcal{D}$: an LINP domain $\langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$;
- $\mathcal{I} \in \texttt{Form}$: a formula denoting a set of initial states;
- $\mathcal{G} \in \texttt{Form}$: a formula denoting a set of goal states.

Each LINP problem $\langle \mathcal{D}, s_\mathcal{I}, \mathcal{G} \rangle$ is an instance of an GLINP problem $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ where $s_\mathcal{I}$ is an initial state, that is, $s_\mathcal{I} \models \mathcal{I}$. It is easily verified that the existence of solutions to GLINP is undecidable from the undecidability result for LINP problems (Helmert 2002).

We then illustrate the GLINP problem with the Delivery problem (Srivastava, Immerman, and Zilberstein 2011).

Example 1. A truck can load and unload a package, and move to a dock or a company. Initially, all packages are at the dock. The truck does not carry any package and its location is uncertain. The goal of the truck is to first transport all packages to the company, and then go to the dock.

The propositional variable at_d denotes the truck is at the dock. The numeric variable num_d, num_c and num_t represents the number of packages at the dock, at the company and on the truck, respectively. The numeric variable cap denotes the capacity of the truck. The actions $move_d$, $load_d$ and $unload_d$ mean the truck moves to the dock, loads and unloads a package at the dock, respectively. The meaning of actions $move_c$, $load_c$ and $unload_c$ are similar. Due to space limit, we only give the preconditions and effects of the actions $move_d$, $load_d$ and $unload_d$.

- $\mathcal{P} : \{at_d\}$;
- $\mathcal{V} : \{num_d, num_c, num_t, cap\}$;
- $\mathcal{A} : \{move_d, move_c, load_d, unload_d, load_c, unload_c\}$;
- $\texttt{pre}(move_d) : \neg at_d$;
- $\texttt{eff}(move_d) : \{\langle \top, at_d, \top \rangle\}$;
- $\texttt{pre}(load_d) : at_d \wedge num_d > 0 \wedge num_t < cap$;
- $\texttt{eff}(load_d) : \{\langle \top, num_d, num_d - 1 \rangle, \langle \top, num_t, num_t + 1 \rangle\}$;
- $\texttt{pre}(unload_d) : at_d \wedge num_t > 0$;
- $\texttt{eff}(unload_d) : \{\langle \top, num_d, num_d + 1 \rangle, \langle \top, num_t, num_t - 1 \rangle\}$;
- $\mathcal{I} : num_d > 0 \wedge num_c = 0 \wedge num_t = 0 \wedge cap > 0$;
- $\mathcal{G} : at_d \wedge num_d = 0 \wedge num_t = 0$. □

The solutions to GLINP problems are *planning programs*.

Definition 3. The set of planning programs ($\texttt{Prog}$) for an LINP domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$ is recursively defined by

$$\delta \in \texttt{Prog} :: \varepsilon \mid a \mid \delta; \delta \mid \textbf{if } \phi \textbf{ then } \delta \textbf{ else } \delta \textbf{ fi} \mid \textbf{while } \phi \textbf{ do } \delta \textbf{ od}$$

where $a \in \mathcal{A}$ and $\phi \in \texttt{Form}$.

The construct $\delta_1; \delta_2$ is the sequential structure; **if** ϕ **then** δ_1 **else** δ_2 **fi** is the branch structure and **while** ϕ **do** δ **od** is the loop structure. We say ϕ is the *condition* of the branch structure **if** ϕ **then** δ_1 **else** δ_2 **fi**. Likewise, ϕ is the condition of the loop structure **while** ϕ **do** δ **od**.

Definition 4. Let δ be a planning program. The length $|\delta|$ of δ is recursively defined as:

- $|\varepsilon| = 0$ and $|a| = 1$ where $a \in \mathcal{A}$;
- $|\delta_1; \delta_2| = |\delta_1| + |\delta_2| + 1$;
- $|\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ fi}| = |\phi| + |\delta_1| + |\delta_2| + 1$;
- $|\textbf{while } \phi \textbf{ do } \delta_1 \textbf{ od}| = |\phi| + |\delta_1| + 1$.

where $|\phi|$ is the length of ϕ.

Definition 5. Let δ be a planning program. The depth $\#(\delta)$ of δ is recursively defined as:

- $\#(\varepsilon) = 0$ and $\#(a) = 0$ where $a \in \mathcal{A}$;
- $\#(\delta_1; \delta_2) = \max(\#(\delta_1), \#(\delta_2))$;
- $\#(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ fi}) = \max(\#(\delta_1), \#(\delta_2))$;
- $\#(\textbf{while } \phi \textbf{ do } \delta_1 \textbf{ od}) = 1 + \#(\delta_1)$.

A loop structure is *simple* if its depth equals to 1; otherwise, it is *nested*.

The action sequence of executing a program δ in a state s is defined as follows:

Definition 6. Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$ be an LINP domain, δ a program for $\mathcal{D}$ and s a state. The action sequence $\Theta(s, \delta)$ of executing δ in s is recursively defined as:

- $\Theta(s, \varepsilon) = \varepsilon$.
- $\Theta(s, a) = a$ where $a \in \mathcal{A}$.
- $\Theta(s, \delta_1; \delta_2) =$
$$\begin{cases} \Theta(s, \delta_1) \circ \Theta(\tau(s, \delta_1), \delta_2), & \text{if } \Theta(s, \delta_1) \text{ is finite;} \\ \Theta(s, \delta_1), & \text{otherwise.} \end{cases}$$
- $\Theta(s, \textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ fi}) = \begin{cases} \Theta(s, \delta_1), & \text{if } \phi(s) = \top; \\ \Theta(s, \delta_2), & \text{otherwise.} \end{cases}$
- $\Theta(s, \textbf{while } \phi \textbf{ do } \delta_1 \textbf{ od}) =$
$$\begin{cases} \Theta(s, \delta_1) \circ \Theta(\tau(s, \delta_1), \textbf{while } \phi \textbf{ do } \delta_1 \textbf{ od}), & \\ \qquad \text{if } \phi(s) = \top \text{ and } \Theta(s, \delta_1) \text{ is finite;} & \\ \Theta(s, \delta_1), \quad \text{if } \phi(s) = \top \text{ and } \Theta(s, \delta_1) \text{ is infinite;} & \\ \varepsilon, \quad \text{otherwise.} & \end{cases}$$

where $\tau(s, \delta)$ is $\tau(s, \Theta(s, \delta))$ when $\Theta(s, \delta)$ is a finite sequence and $\Theta_1 \circ \Theta_2$ is the concatenation of two sequences Θ_1 and Θ_2.

Given a GLINP problem, we are interested in synthesizing a program δ satisfying the following three properties.

Definition 7. Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$ be an LINP domain, δ a program for $\mathcal{D}$ and s a state. The program δ is

- *terminating* in s, iff $\Theta(s, \delta)$ is finite;
- *executable* in s, iff $\Theta(s, \delta)$ is executable in s;

- ϕ-*reaching* in s, iff δ is terminating and executable in s only if $\tau(s, \delta) \models \phi$.

A planning program is a solution to a GLINP problem, if it satisfies the above three properties for every initial state.

Definition 8. Let $\Sigma = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a GLINP problem. A program δ is a solution to Σ, if δ is terminating, executable and $\mathcal{G}$-reaching in every initial state s.

The Main Framework

The notion of regexes is highly related to planning programs. Suppose that the alphabet Δ is the set $\mathcal{A}$ of actions. A sequence π of actions is a string over Δ. Each construct of a regex corresponds to a structure of planning programs. For example, the sequential structure $\delta_1; \delta_2$ corresponds to the concatenation regex $\delta_1 \circ \delta_2$. In addition, if the condition of the branch structure **if** ϕ **then** δ_1 **else** δ_2 **fi** is omitted, then it corresponds to $\delta_1 | \delta_2$. Similarly, the loop structure **while** ϕ **do** δ **od** corresponds to the iteration regex δ^*. Hence, regexes can be considered as skeletons of the planning programs. In the following, we do not differentiate terminologies of planning and of regexes, and use them interchangeably, *e.g.*, sequences of actions and strings.

Inspired by the intimate connection between regexes and planning programs, we develop an approach to synthesizing planning programs from a set of state-plan pairs shown in Figure 1, consisting of three procedures: (1) generate a set Υ of state-plan pairs, including generating initial states s by imposing some restrictions on propositional and numeric variables and computing the corresponding solutions π by the planner; (2) infer a skeleton of the planning program r expressed as a regex according to the plans of Υ; and (3) obtain a complete planning program δ by filling the missing conditions in r according to Υ.

The main framework consists of four essential components: `GenStatePlanPairs`, `InfSkeleton`, `Complete` and `Plan`. The first three ones will be sequentially explained in the following sections. The final procedure Plan can be implemented by directly invoking existing numeric planners, *e.g.*, Metric-FF (Hoffmann 2003).

Generation of State-Plan Pairs

In this section, we introduce a method to generate a set of initial states so as to facilitate identifying branch and loop structures in the planning program.

Firstly, it can be observed that the occurrence of branch structures is due to the uncertainty of Boolean values of some propositional variables. For example, the initial location of the truck is uncertain. If the truck is initially at the company, then it has to go to the dock first. Conversely, it directly starts transferring the packages.

Secondly, inference of a regex r with iteration operators from one *representative string* was investigated in the area of grammatical inference (Brāzma 1993; Kinber 2010). A string π is *representative* for the regex r, if the generator of any iteration subregex of r consecutively occurs in π at least twice. We observe from most planning domains that the sequential plan for the state s is representative, if s is such that the values of some numeric variables are large enough. For example, in the Delivery problem, suppose that the capacity of the truck is more than 1. Currently, the truck carrying no packages is at the dock and a number of packages still are at the dock. In this case, the truck should iteratively load the packages until reaching the capacity.

From the above two observations, we require the set S of initial states to be conformed with the two principles: (1) for every propositional variable $p \in \mathcal{P}$ that is not determined by $\mathcal{I}$, S must contain two states s_1 and s_2 in which Boolean value of p are distinct, (*i.e.*, $p(s_1) \neq p(s_2)$); and (2) for every state $s \in S$ and every numeric variable $v \in \mathcal{V}$, if $v(s)$ has the maximum c under the initial formula $\mathcal{I}$, then $v(s) = c$, otherwise, $v(s) \geq b$ where b is a bound. We assign the bound b an integer 3 and generate 3 initial states in this procedure. Some initial states, which do not conform with the above principles, will not be generated in this procedure.

Finally, the numeric planner computes the sequential plan for each initial state.

Example 2. We continue with the Delivery problem. A state s is represented by a vector $(at_d(s), num_d(s), num_c(s), num_t(s), cap(s))$. The initial formula $\mathcal{I}$ is $num_d > 0 \wedge num_c = 0 \wedge num_t = 0 \wedge cap > 0$. Both num_c and num_t have the maximum 0 under $\mathcal{I}$. However, neither num_d nor cap has a maximum. The variable at_d is not determined by $\mathcal{I}$. In light of the principles of initial states, the procedure `GenStatePlanPairs` first generates three initial states: $s_1 : (\top, 8, 0, 0, 3)$, $s_2 : (\bot, 10, 0, 0, 4)$ and $s_3 : (\bot, 9, 0, 0, 3)$. The numeric planner then computes the plan π_i for each state s_i as follows. For brevity, we abbreviate $load_d$ as L , $unload_c$ as U, $move_c$ as C and $move_d$ as D.

- $\pi_1 = [L, L, L, C, U, U, U, D, L, L, L, C, U, U, U, D, L, L, C, U, U, D]$;
- $\pi_2 = [D, L, L, L, L, C, U, U, U, U, D, L, L, L, L, C, U, U, U, U, D, L, L, C, U, U, D]$;
- $\pi_3 = [D, L, L, L, C, U, U, U, D, L, L, L, C, U, U, U, D, L, L, L, C, U, U, U, D]$. □

Inference of Skeletons of Planning Programs

In this section, we introduce the procedure `InfSkeleton` that aims to guess a suitable skeleton of the planning programs expressed by a regex with iteration and alternation subregexes. The main insight behind the procedure illustrated in Algorithm 1, is to infer a regex based on a set of strings. It consists of three steps: (1) identification of iteration subregexes, (2) alignment of iteration subregexes, and (3) identification of alternation subregexes.

Identification of Iteration Subregexes

The first step is to fold each representative plan $\pi \in \Pi$ into a regex without alternation t (Lines 1 - 8). It starts from the original alphabet $\Delta = \mathcal{A}$ (Line 3), and infers a regex t' with iteration subregexes over Δ accepting π (Line 4). If the current regex t' is different from the previous one t, then it means that new iteration subregexes are found in this iteration. These subregexes are considered as new single characters and are used to enlarge the alphabet Δ. The regex t' can

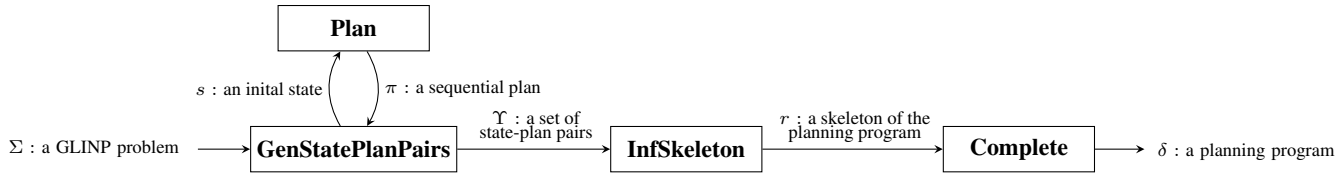

Figure 1: The Main Framework

Algorithm 1: `InfSkeleton`(Σ, Υ)

Input: Σ: the planning problem $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ where $\mathcal{D} = \langle \mathcal{P}, \mathcal{V}, \mathcal{A} \rangle$
Υ: the finite set of state-plan pairs
Output: r: a skeleton of the planning programs

1 $R \leftarrow \emptyset$ and $\Delta^* \leftarrow \emptyset$
2 **foreach** $\pi \in \Pi(\Upsilon)$ **do**
3 $t \leftarrow \pi$ and $\Delta \leftarrow \mathcal{A}$
4 $(t', \Delta') \leftarrow$ `FoldString`(t, Δ)
5 **while** $t \neq t'$ **do**
6 $\Delta \leftarrow \Delta \cup \Delta'$ and $t \leftarrow t'$
7 $(t', \Delta') \leftarrow$ `FoldString`(t, Δ)
8 $R \leftarrow R \cup \{t\}$ and $\Delta^* \leftarrow \Delta^* \cup \Delta$
9 **foreach** $t \in R$ **do**
10 **foreach** *misalgined subregex of* t *with the form* $(u_1u_2)^*u_1$ **do**
11 Replace every occurrence of $(u_1u_2)^*u_1$ in t by $u_1(u_2u_1)^*$
12 $(\xi_1, \cdots, \xi_l) \leftarrow$the sequence of non-extensible
13 common strings of R over Δ^*
14 Compute each i-th individual components $\eta_{i,j}$ of t_j s.t. $t_j = \eta_{1,j} \circ \xi_1 \circ \cdots \circ \eta_{l,j} \circ \xi_l \circ \eta_{l+1,j}$ for $1 \leq i \leq l+1$ and $1 \leq j \leq k$
15 $\eta_i \leftarrow \eta_{i,1} | \cdots | \eta_{i,k}$ for $1 \leq i \leq l+1$
16 $r \leftarrow \eta_1 \circ \xi_1 \circ \eta_2 \circ \xi_2 \circ \cdots \circ \eta_l \circ \xi_l \circ \eta_{l+1}$
17 Simplify r

be considered as a string over the new alphabet Δ. The above computation will continue until no new iteration subregexes are identified. Generating a regex t' with iteration subregexes u^* for a regex t is done by the process `FoldString`. It recognizes the iteration subregex u^* when the substring u over Δ consecutively occurs in t at least twice, and then replaces all of the longest substrings $u \cdots u$ in t by u^*. After the above computation, the regex t is grouped into the set R and the alphabet Δ^* is enlarged by Δ (Line 8).

Example 3. We continue with Example 2. We first infer the regex t_1 for the string π_1. The construction is shown in Table 1. At the first iteration, the alphabet Δ is the set of action symbols of the Delivery problem. The process `FoldString` discovers two iteration subregexes L^* and U^* since L occurs in the first 3 characters of π_1 and U occurs from the 5-th to 7-th characters. It generates t_1 as $L^*CU^*DL^*CU^*D$. At the second iteration, the alphabet Δ is extended by the above two iteration regexes L^* and U^*. As t_1 repeats L^*CU^*D twice, $(L^*CU^*D)^*$ is identified and t_1 becomes $(L^*CU^*D)^*$. At the third iteration, no iteration subregexes are found. Hence, the process of identifying

Iter.	t	Δ
1	$L^*CU^*DL^*CU^*D$	$\mathcal{A} \cup \{L^*, U^*\}$
2	$(L^*CU^*D)^*$	$\mathcal{A} \cup \{L^*, U^*, (L^*CU^*D)^*\}$
3	$(L^*CU^*D)^*$	$\mathcal{A} \cup \{L^*, U^*, (L^*CU^*D)^*\}$

Table 1: The process of identifying iteration subregexes of π_1 where t denotes the resulting regex at each iteration and Δ denotes the extended alphabet of each iteration.

iteration subregexes terminates and t_1 remains unchanged.

Similarly, we obtain $t_2 = t_3 = (DL^*CU^*)^*D$. Since t_2 and t_3 are identical, we get that $R = \{(L^*CU^*D)^*, (DL^*CU^*)^*D\}$. □

Alignment of Iteration Subregexes

In the above example, t_2 has an equivalent regex $t_2' = D(L^*CU^*D)^*$. Compared to t_2, the regex t_2' is more similar to t_1 since t_1 and t_2' have a common subregex $(L^*CU^*D)^*$. In order to synthesize a good skeleton of the program from the set R of regexes, it is necessary to adjust some regexes of R such that every regex of R has more common subregexes with the others. In the following, we give the definition of misalignment of subregexes.

Definition 9. Let R be a set of regexes and $t \in R$. A subregex of t is *misaligned* w.r.t. R, if the following hold:

1. it is of the form $(u_1u_2)^*u_1$ where u_1 and u_2 are subregexes of t;
2. $(u_2u_1)^*$ is a subregex of some $t' \in R$ where $t \neq t'$;
3. every regex $t' \in R$ s.t. $t \neq t'$ has no subregex $(u_1u_2)^*$.

In case of misalignment of a subregex of t, t can be transformed into another one t' via replacing each occurrence of $(u_1u_2)^*u_1$ in t by $u_1(u_2u_1)^*$. Condition (1) says that the above transformation preserves equivalence, *i.e.*, $\mathcal{L}(t) = \mathcal{L}(t')$. Conditions (2) requires that $(u_2u_1)^*$ occurs in other regexes and Condition (3) means that $(u_1u_2)^*$ does not. The regex t' therefore has more common subregexes with the other regexs of R than t.

We do not consider replacing the occurrence of subregex of the form $u_1(u_2u_1)^*$ by $(u_1u_2)^*u_1$ in every regex of R. The reason is as follows. Suppose that a string π contains the substring $\pi' = u_1 \circ u_2 \circ \cdots \circ u_1 \circ u_2 \circ u_1$ where u_1 and u_2 are strings. Two regexes $(u_1u_2)^*u_1$ and $u_1(u_2u_1)^*$ accept the above substring. However, the process `FoldString` identifies the subregex $(u_1u_2)^*$ rather than $(u_2u_1)^*$ since it scans π from left to right.

The alignment procedure, sketched in Lines 9 - 11 of Algorithm 1, simply replaces every occurrence of misaligned

subregexes of the form $(u_1u_2)^*u_1$ in t_i by $u_1(u_2u_1)^*$ for every regex t of R.

Example 4. We continue with Example 3. The set R contains two regexes: $t_1 = (L^*CU^*D)^*$ and $t_2 = (DL^*CU^*)^*D$. Clearly, only the second one itself is a misaligned subregex $(DL^*CU^*)^*D$. We then transform t_2 into an equivalent regex $t_2' = D(L^*CU^*D)^*$. □

Identification of Alternation Subregexes

Finally, the procedure `InfSkeleton` merges all regexes of R into a final regex r with the alternation connective in light of the notions of common substrings and common subsequences (Lines 12 - 16).

A string ξ is a *common substring* of R, if ξ is a substring of $t_j \in R$ for $1 \leq j \leq k$ where k is the number of regex in R. Similarly, a string ξ is a *common subsequence* of R, if ξ is a subsequence of $t_j \in R$ for $1 \leq j \leq k$. The sequence of non-extensible common strings of R is $(\xi_1, \cdots, \xi_l)$ s.t. $\xi_1 \circ \cdots \circ \xi_l$ is the longest common subsequence of R, and ξ_i is a non-extensible common substring of R (more precisely, the concatenation $(\xi_{i-1})_{|\xi_{i-1}|} \circ \xi_i$ of the last character of ξ_{i-1} and ξ_i is not a common substring of R for $i > 1$, and the concatenation $\xi_i \circ (\xi_{i+1})_1$ of ξ_i and the first character of ξ_{i+1} is a not common substring of R for $i < l$). It can be easily observed that each $t_j \in R$ is the form of $\eta_{1,j} \circ \xi_1 \circ \cdots \eta_{l,j} \circ \xi_l \circ \eta_{l+1,j}$ where $\eta_{i,j}$ may be an empty string for $1 \leq i \leq l+1$ and $1 \leq j \leq k$. Each $\eta_{i,j}$ is called the i-th *individual component* of t_j. Meanwhile, each ξ_i is called the *common component* of the R.

We remind that each regex t of R is a string over Δ^*. The final regex r is obtained as follows. We firstly generate the sequence of non-extensible common strings of R, namely $(\xi_1, \cdots, \xi_l)$, and each i-th individual component of t_j (Lines 12 and 13). We then obtain the regex η_i via combining all of the i-th independent component of t_j's with alternation connectives (*i.e.*, $\eta_{i,1}|\cdots|\eta_{i,k}$) for each $1 \leq i \leq l+1$ (Line 14). We concatenate the combination of individual components η_i and the common component ξ_i alternatively in an increasing order (Line 15). In the end, some redundant subregexes in r are removed (Line 16) (*e.g.*, $u_1|u_2|\cdots|u_{k-1}|u_2|u_k$ is simplified as $u_1|u_2|\cdots|u_{k-1}|u_k$).

Example 5. We continue with the Example 4. The set R contains two regexes: $t_1 = (L^*CU^*D)^*$ and $t_2' = D(L^*CU^*D)^*$. The two regexes have only one common component $\xi_1 = (L^*CU^*D)^*$. The regex t_1 contains an independent component $\eta_{1,1} = \varepsilon$. Similarly, the only independent component $\eta_{1,2}$ of t_2 is D. So $\eta_1 = \eta_{1,1}|\eta_{1,2} = \varepsilon|D$. By concatenating η_1 and ξ_1, we get the final regex $r = \eta_1 \circ \xi_1 = (\varepsilon|D)(L^*CU^*D)^*$. □

The following theorem states that the procedure `InfSkeleton` infers the regex r that accepts every string $\pi \in \Pi(\Upsilon)$.

Theorem 1. *Let $\Sigma = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a GLINP problem and Υ a set of state-plan pairs. Then,* `InfSkeleton`(Σ, Υ) *outputs a regex r accepting each $\pi \in \Pi(\Upsilon)$.*

Proof sketch: For each $\pi_j \in \Pi(\Upsilon)$, the FoldString process iteratively replaces the consecutive occurrence of substring u in π_j by u^*, and obtain the regex t_j. So each t_j, which is added into the set R, accepts π_j. Then, the alignment step preserves equivalence. In addition, each t_j is divided into a sequence of subregexes (*i.e.*, $\eta_{1,j} \circ \xi_1 \circ \cdots \circ \eta_{l,j} \circ \xi_l \circ \eta_{l+1,j}$ where $\eta_{i,j}$ is an individual component of t_j and ξ_i is a common component of the R). The final regex r is $\eta_1 \circ \xi_1 \circ \eta_2 \circ \xi_2 \circ \cdots \circ \eta_l \circ \xi_l \circ \eta_{l+1}$ where $\eta_i = \eta_{i,1}|\cdots|\eta_{i,k}$. Obviously, r accepts every $\pi_i \in \Pi(\Upsilon)$. □

Algorithm 2: `Complete`(r, Υ)

Input: r: a skeleton of the planning program
Υ: a set of state-plan pairs
Output: δ: a complete planning program

1 **switch** r **do**
2 **case** $r_1|r_2$ **do**
3 $S^+ \leftarrow \emptyset$ and $S^- \leftarrow \emptyset$
4 $\Upsilon_1 \leftarrow \emptyset$ and $\Upsilon_2 \leftarrow \emptyset$
5 **foreach** $(s, \pi) \in \Upsilon$ **do**
6 **if** $\pi \in \mathcal{L}(r_1)$ **then**
7 $S^+ \leftarrow S^+ \cup \{s\}$
8 $\Upsilon_1 \leftarrow \Upsilon_1 \cup \{(s, \pi)\}$
9 **else** /* $\pi \in \mathcal{L}(r_2)$ */
10 $S^- \leftarrow S^- \cup \{s\}$
11 $\Upsilon_2 \leftarrow \Upsilon_2 \cup \{(s, \pi)\}$
12 $\delta_1 \leftarrow$ `Complete`(r_1, Υ_1)
13 $\delta_2 \leftarrow$ `Complete`(r_2, Υ_2)
14 $\phi \leftarrow$ `SynFormula`(S^+, S^-)
15 $\delta \leftarrow$ **if** ϕ **then** δ_1 **else** δ_2 **fi**
16 **case** r_1^* **do**
17 $S^+ \leftarrow \emptyset$ and $S^- \leftarrow \emptyset$
18 $\Upsilon_1 \leftarrow \emptyset$
19 **foreach** $(s, \pi) \in \Upsilon$ **do**
20 Split π into a sequence $(\pi_1, \pi_2, \cdots, \pi_k)$ s.t. each $\pi_i \in \mathcal{L}(r_1)$;
21 **for** $i \leftarrow 1$ **to** k **do**
22 $S^+ \leftarrow S^+ \cup \{s\}$
23 $\Upsilon_1 \leftarrow \Upsilon_1 \cup \{(s, \pi_i)\}$
24 $s \leftarrow \tau(s, \pi_i)$
25 $S^- \leftarrow S^- \cup \{s\}$
26 $\delta_1 \leftarrow$ `Complete`(r_1, Υ_1)
27 $\phi \leftarrow$ `SynFormula`(S^+, S^-)
28 $\delta \leftarrow$ **while** ϕ **do** δ_1 **od**
29 **case** $r_1 \circ r_2$ **do**
30 $\Upsilon_1 \leftarrow \emptyset$ and $\Upsilon_2 \leftarrow \emptyset$
31 **foreach** $(s, \pi) \in \Upsilon$ **do**
32 Split π into the prefix π_1^i and suffix $\pi_{i+1}^{|\pi|}$ s.t. $\pi_1^i \in \mathcal{L}(r_1)$ and $\pi_{i+1}^{|\pi|} \in \mathcal{L}(r_2)$
33 $\Upsilon_1 \leftarrow \Upsilon_1 \cup \{(s, \pi_1^i)\}$
34 $\Upsilon_2 \leftarrow \Upsilon_2 \cup \{(\tau(s, \pi_1^i), \pi_{i+1}^{|\pi|})\}$
35 $\delta_1 \leftarrow$ `Complete`(r_1, Υ_1)
36 $\delta_2 \leftarrow$ `Complete`(r_2, Υ_2)
37 $\delta \leftarrow \delta_1; \delta_2$
38 **otherwise do** /* $r = \varepsilon$ or $r = a$ */
39 $\delta \leftarrow r$

Completion of Planning Programs

In this section, we illustrate the procedure `Complete` which takes a skeleton r and a set of state-plan pairs Υ as input, and outputs a complete planning program δ.

The procedure `Complete`, illustrated in Algorithm 2, first collects a set S^+ of positive states and a set S^- of negative states from the set of state-plan pairs Υ for each condition ϕ in a recursive way. A state s is a *positive* (resp. *negative*) state of ϕ, if $\phi(s) = \top$ (resp. $\phi(s) = \bot$). It then infers these conditions via the algorithm `SynFormula`, proposed by Udupa et al. (2013), which aims to generate candidate formulas in an increasing size until it finds an expected formula ϕ that is consistent with the two sets S^+ and S^-.

In the case where $r = r_1|r_2$ (Lines 2 - 15). This regex corresponds to the branch structure **if** ϕ **then** δ_1 **else** δ_2 **fi**. The sets Υ_1 and Υ_2 are two sets of state-plan pairs for completing r_1 and r_2, respectively. For each $(s, \pi) \in \Upsilon$, if r_1 accepts π, then the execution of π enters the branch expressed by r_1, and hence the state s is a positive state of ϕ. The set S^+ is enlarged by s, and the state-plan pair (s, π) is added into Υ_2 (Lines 6 - 8). Otherwise, the execution of π goes to another branch described by r_2. The state s is a negative state of ϕ. The opposite case is similar (Lines 9 - 11). Then, we obtain the subprograms δ_1 and δ_2 by invoking `Complete`(r_1, Υ_1) and `Complete`(r_2, Υ_2), respectively (Lines 12 & 13). Finally, the algorithm `SynFormula` constructs the condition ϕ based on S^+ and S^- (Line 14).

In the case where $r = r_1^*$ (Lines 16 - 28). This regex corresponds to the loop structure **while** ϕ **do** δ_1 **od**. For each $(s, \pi) \in \Upsilon$, if the sequential plan π is not an empty plan, then there is a sequence $(\pi_1, \pi_2, \cdots, \pi_k)$ s.t. their concatenation is π and r_1 accepts each π_i (Line 20). In other words, the program expressed by r_1 is executed k times, where the action sequence of the i-th execution is π_i. It is easily observed that the execution of the loop structure enters the body δ_1 in the following states: $s, \tau(s, \pi_1), \cdots, \tau(s, \pi_1 \cdots \pi_{k-1})$. So the above states are positive states of the condition ϕ and are added into S^+ (Line 22). Meanwhile, the following state-plan pairs $(s, \pi_1), (\tau(s, \pi_1), \pi_2), \cdots, (\tau(s, \pi_1 \cdots \pi_{k-1}), \pi_k)$ are put into Υ_1 (Line 23). When the loop is completed, the execution is out of the loop and terminates in the state $\tau(s, \pi)$. We consider this state as a negative state of ϕ which is added into S^- (Line 25). Then, the subprogram δ_1 is obtained according to r_1 and Υ_1 (Line 26). Finally, the condition ϕ is also synthesized based on S^+ and S^- (Line 27).

In the case where $r = r_1 \circ r_2$ (Lines 29 - 37). This regex corresponds to the sequential structure $\delta_1; \delta_2$. For each $(s, \pi) \in \Upsilon$, there are a prefix π_1^i of π and a suffix $\pi_{i+1}^{|\pi|}$ s.t. r_1 accepts the former and r_2 accepts the latter (Line 32). We collect the state-plan pair (s, π_1^i) and $(\tau(s, \pi_1^i), \pi_{i+1}^{|\pi|})$ for Υ_1 and Υ_2, respectively (Lines 33 and 34). Finally, the subprograms δ_1 and δ_2 are constructed recursively (Lines 35 & 36).

Example 6. In Example 5, the procedure `InfSkeleton` infers the regex r as $(\varepsilon|D)(L^*CU^*D)^*$. To be clear, we first present the corresponding skeleton of the program via translating concatenation (resp. alternation/iteration) subregexes to sequential (resp. branch/loop) structure with ϕ_i.

if ϕ_1 **then** ε **else** $move_d$ **fi**;
while ϕ_2 **do**
  while ϕ_3 **do** $load_d$ **od**;
  $move_c$;
  while ϕ_4 **do** $unload_c$ **od**;
  $move_d$
od

The skeleton is a sequential structure $\delta_1; \delta_2$ where the skeleton of δ_1 is "**if** ϕ_1 **then** ε **else** $move_d$ **fi**" and δ_2 is a nested loop structure with the condition ϕ_2. The program contains four conditions ϕ_1, ϕ_2, ϕ_3 and ϕ_4. To distinguish the set of positive states for various conditions, we use S_i^+ for the the set of positive states for ϕ_i and the definition of S_i^- is similar. We collect S_i^+ and S_i^- for each condition ϕ_i according to the three state-plan pairs: (s_1, π_1), (s_2, π_2) and (s_3, π_3) where $s_1 : (\top, 8, 0, 0, 3)$, $s_2 : (\bot, 10, 0, 0, 4)$ and $s_3 : (\bot, 9, 0, 0, 3)$.

Consider the plan π_1 for s_1. Clearly, $\varepsilon \in \mathcal{L}(\varepsilon|D)$ and $\pi_1 \in \mathcal{L}((L^*CU^*D)^*)$. So $(s_1, \varepsilon) \in \Upsilon_1$ and $(s_1, \pi_1) \in \Upsilon_2$. Similarly, we get that $\Upsilon_1 = \{(s_1, \varepsilon), (s_2, D), (s_3, D)\}$ and $\Upsilon_2 = \{(s_1, \pi_1), (s_2', (\pi_2)_2^{|\pi_2|}), (s_3', (\pi_3)_3^{|\pi_3|})\}$ where s_2' is $(\top, 10, 0, 0, 4)$ and s_3' is $(\top, 9, 0, 0, 3)$.

We start to synthesize the condition ϕ_1 from Υ_1. It is easily verified that $S_1^+ = \{s_1\}$ and $S_1^- = \{s_2, s_3\}$. The algorithm `SynFormula` deduces that ϕ_1 is at_d.

We now synthesize the condition ϕ_2 from Υ_2. By tracking the trace of performing π_1 (resp. $(\pi_2)_2^{|\pi_2|}/(\pi_3)_3^{|\pi_3|}$) on s_1 (resp. s_2'/s_3'), we get that $S_2^+ = \{(\top, 8, 0, 0, 3), (\top, 10, 0, 0, 4), (\top, 9, 0, 0, 0, 3), (\top, 5, 3, 0, 3), (\top, 6, 4, 0, 4), (\top, 6, 3, 0, 3), (\top, 2, 6, 0, 3), (\top, 2, 8, 0, 4), (\top, 3, 6, 0, 3)\}$ and $S_2^- = \{(\top, 0, 8, 0, 3), (\top, 0, 10, 0, 4), (\top, 0, 9, 0, 3)\}$. The algorithm `SynFormula` synthesizes ϕ_2 as $num_d \geq 1$. During the process of synthesizing ϕ_2, the conditions ϕ_3 and ϕ_4 of the two inner loops of δ_2 are also constructed as $num_d \geq 1 \wedge cap > num_t$ and $num_t \geq 1$ respectively.

The entire planning program is as follows:

if at_d **then** ε **else** $move_d$ **fi**;
while $num_d \geq 1$ **do**
  while $num_d \geq 1 \wedge cap > num_t$ **do** $load_d$ **od**;
  $move_c$;
  while $num_t \geq 1$ **do** $unload_c$ **od**;
  $move_d$
od □

The following theorem states that the procedure `Complete` synthesizes a correct program for every state $s \in S(\Upsilon)$.

Theorem 2. *Let $\Sigma = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$. Let r be a skeleton of a planning program of $\mathcal{D}$ and Υ a set of state-plan pairs where r accepts each plan π with $\pi \in \Pi(\Upsilon)$. Then,* `Complete`(r, Υ) *outputs a program δ that is terminating, executable and $\mathcal{G}$-reaching in every state $s \in S(\Upsilon)$.*

Proof. We firstly prove that $\pi = \Theta(s, \delta)$ for $(s, \pi) \in \Upsilon$. We prove by induction on δ.

- $\delta = \varepsilon$ or $\delta = a$: Suppose that $\delta = \varepsilon$. Since r accepts π, $\pi = \varepsilon$, and hence $\Theta(s, \delta) = \varepsilon = \pi$. Similarly, $\Theta(s, a) = [a] = \pi$ when $\delta = a$.

- $\delta = \delta_1; \delta_2$: It follows that $r = r_1 \circ r_2$ where r_i corresponds to the program δ_i for $i = 1, 2$. Since r accepts π, there is an index i s.t. r_1 accepts π_1^i and r_2 accepts $\pi_{i+1}^{|\pi|}$. According to the procedure `Complete` (Algorithm 2), $(s, \pi_1^i) \in \Upsilon_1$ and $(\tau(s, \pi_1^i), \pi_{i+1}^{|\pi|}) \in \Upsilon_2$. By the inductive assumption, the procedure `Complete` synthesizes two programs δ_1 and δ_2 s.t. $\pi_1^i = \Theta(s, \delta_1)$ and $\pi_{i+1}^{|\pi|} = \Theta(\tau(s, \pi_1^i), \delta_2)$. Hence, $\pi = \pi_1^i \circ \pi_{i+1}^{|\pi|} = \Theta(s, \delta_1) \circ \Theta(\tau(s, \pi_1^i), \delta_2) = \Theta(s, \delta_1; \delta_2)$.
- $\delta =$ **if** ϕ **then** δ_1 **else** δ_2 **fi**: It follows that $r = r_1 | r_2$ where r_i corresponds to the program δ_i for $i = 1, 2$. Suppose that $\phi(s) = \top$. Thus, $\Theta(s, \delta) = \Theta(s, \delta_1)$. Since the procedure `SynFormula` constructs the condition ϕ consistent with S^+ and S^-. So $s \in S^+$ and $(s, \pi) \in \Upsilon_1$. By the inductive assumption, the procedure `Complete` synthesizes a program δ_1 s.t. $\pi = \Theta(s, \delta_1)$. Hence, $\pi = \Theta(s, \delta)$. The case where $\phi(s) = \bot$ can be similarly proved.
- $\delta =$ **while** ϕ **do** δ_1 **od**: It follows that $r = r_1^*$ where r_1 corresponds to the program δ_1. The case where $\pi = \varepsilon$ is easier. Suppose that $\pi \neq \varepsilon$. Since r accepts π, there is a sequence of substrings $(\pi_1, \pi_2, \cdots, \pi_k)$ s.t. $\pi_1 \circ \pi_2 \circ \cdots \circ \pi_k = \pi$ and $\pi_i \in \mathcal{L}(r_1)$ for $1 \leq i \leq k$. According to the procedure `Complete`, $(s, \pi_1) \in \Upsilon_1$ and $(\tau(s, \pi_1 \circ \cdots \circ \pi_i), \pi_{i+1}) \in \Upsilon_1$ for $1 \leq i < k$. By the inductive assumption, the procedure `Complete` synthesizes a program δ_1 s.t. $\pi_1 = \Theta(s, \delta_1)$ and $\pi_{i+1} = \Theta(\tau(s, \pi_1 \circ \cdots \circ \pi_i), \delta_1)$ for $1 \leq i < k$. In addition, $s \in S^+$, $\tau(s, \pi_1 \circ \pi_i) \in S^+$, and $\tau(s, \pi_1 \circ \pi_k) \in S^-$. So $s^+ \models \phi$ for $s^+ \in S^+$ and $s^- \models \neg\phi$ for $s^- \in S^-$. By Definition 6, we get that $\Theta(s,$ **while** ϕ **do** δ_1 **od**$) = \Theta(s, \delta_1) \circ \cdots \circ \Theta(\tau(s, \pi_1 \circ \cdots \circ \pi_{k-1}), \delta_1)$. Hence, $\pi = \Theta(s,$ **while** ϕ **do** δ_1 **od**$)$.

Since π is also a solution to the LINP problem $\langle \mathcal{D}, s, \mathcal{G} \rangle$ for $(s, \pi) \in \Pi$, π is finite and executable in s, and $\tau(s, \pi) \models \mathcal{G}$. This, together with the fact that $\pi = \Theta(s, \delta)$, imply δ is a terminating, executable and $\mathcal{G}$-reaching in s. □

Theorems 1 and 2 together guarantees that the synthesized planning program is suitable for the given initial states. Furthermore, the planning program is also correct for all of the initial states in practice, which will be verified in the experimental evaluation. This is because that the procedure `GenStatePlanPairs` generates the initial states according to the two principles so that the procedure `InfSkeleton` is able to extract the branch and loop structures of the program from their corresponding sequential plans. In addition, we have developed a method for verifying correctness of planning programs and identified a class of planning programs of which correctness verification becomes decidable. All domains with only unconditional effects considered in this paper are in this class, and their programs are verified in an automated way. For other domains, we verify their correctness manually. However, due to the space limit, we do not present the work about the automated verification of planning program in this paper, which will be clarified in future work.

Domain	RegexSkeleton			QNP2FOND	
	$\#(\delta)$	$\|\delta\|$	t	$\|p\|$	t
Arith	1	13	0.097	-	-
Chop	1	5	0.028	5	0.039
ClearBlock	1	7	0.027	15	0.088
Corner-A	1	11	0.048	29	0.226
Corner-R	1	13	0.064	-	-
Delivery	2	31	5.161	229	65.31
D-Return	1	41	0.215	-	-
D-Return-R	2	52	325.2	-	-
Gripper	2	31	5.571	201	112.9
Hall-A	1	37	0.135	-	-
Hall-R	1	45	0.166	-	-
NestVar	1	13	0.030	48	0.501
NestVar8	1	541	6.315	TO	TO
MNestVar	2	11	0.034	-	-
MNestVar8	8	287	311.8	-	-
PlaceBlock	1	23	0.045	71	3.234
Rewards	2	11	0.046	28	0.232
Snow	1	21	0.037	115	3.822
Spanner	1	21	0.094	-	-
TestOn	1	13	0.062	41	0.433
VisitAll	2	37	0.129	-	-
VisitAll-R	2	51	0.232	-	-

Table 2: Experimental results. "$\#\delta$", "$|\delta|$" and "$|p|$" are the depth and the length of the planning program, and the length of the policy, respectively; and "t" is the running time of synthesizing the planning programs and the policy. "TO" denotes timeout; "-" indicates that the problem cannot be formalized in QNP.

Experimental Evaluation

We have implemented a prototype of the above approach, namely RegexSkeleton, for synthesizing the planning programs in Python using Z3 (de Moura and Bjørner 2008) and Metric-FF (Hoffmann 2003). The experiments are run on a machine with Intel Core i7-7700 3.60GHz CPU and 8GB RAM. The timeout period is set to 30 minutes.

We run RegexSkeleton on 22 domains that originate from the work on GP and QNP (Levesque 2005; Bonet, Palacios, and Geffner 2009; Bonet, Francès, and Geffner 2019; Srivastava, Immerman, and Zilberstein 2011; Srivastava et al. 2011; Bonet and Geffner 2020). We compare the performance of RegexSkeleton with the state-of-the-art QNP planner: QNP2FOND (Bonet and Geffner 2020) and two numeric planners: Metric-FF (Hoffmann 2003) and ENHSP (Scala, Haslum, and Thiébaux 2016). Although there have been some implementations on synthesis of GP so far, they do not support the problems with multiple numeric variables. For instance, the methods proposed in (Srivastava, Immerman, and Zilberstein 2011; Bonet, Palacios, and Geffner 2009; Bonet and Geffner 2015; Segovia-Aguas, Jiménez, and Jonsson 2018, 2019, 2021) can not solve the problems with numeric variables and those proposed in (Levesque 2005; Hu and Giacomo 2011) can only solve the problems with only one numeric variable while almost all GLINP domains have multiple variables. Thus, we do not compare with their methods.

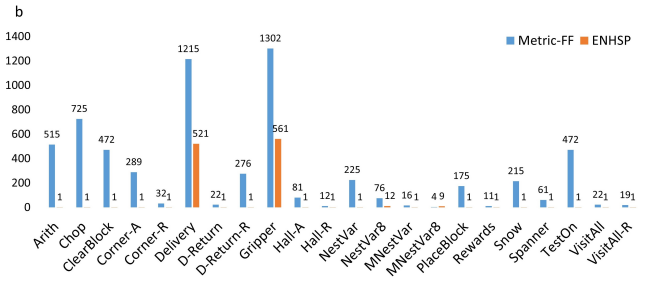

Figure 2: The threshold of bound b for each domain when RegexSkeleton outperforms Metric-FF and ENHSP in efficiency, respectively.

We first compare RegexSkeleton to the QNP planner QNP2FOND. The results for each domain are reported in Table 2, from which we can make several observations. (1) RegexSkeleton is able to solve all of 22 domains but QNP2FOND only solves 9 domains. NestVar8 causes a timeout for QNP2FOND and the other domains cannot be formalized in QNP. Firstly, the initial formula of Spanner and the goal formulas of Arith, VisitAll and Hall-A are not simple numeric formulas. Furthermore, the effects of MNestVar and MNestVar8 require assigning numeric variables to a $\mathtt{LIA}^{\mathtt{P}}$-formula. Finally, it is necessary to formalize the effects of Hall-R, Corner-R, VisitAll-R and D-Return-R by conditional effects. (2) For every domain solved by QNP2FOND, RegexSkeleton obtains a more compact solution, which verifies the succinctness results between planning programs and policies. For example, the sizes of policies for Delivery and Gripper is 229 and 201, respectively. But those of planning programs are both 31. (3) RegexSkeleton is more efficient than QNP2FOND for all problems that can be formalized in QNP. In particular, QNP2FOND solves Delivery and Gripper in approximately 65s and 112s, respectively, but fails to solve NestVar8, while RegexSkeleton succeeds in synthesizing the planning program to the above three problems within 7s. This is because that RegexSkeleton firstly extracts the branch and loop structures of the planning program and generates a skeleton of the planning program which accelerates the whole synthesis process. In contrast, the solution to QNPs is of the policy form which is in fact a loop structure of several conditional statements. It is difficult for policies to express the loop structure whose body is a sequential plan and even a loop structure. Hence, QNP2FOND does not analyze the common pattern of the solution.

We now turn to compare RegexSkeleton to the two state-of-the-art numeric planners: Metric-FF and ENHSP on LINP problems with a different bound b. Given a GLINP problem $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, we generate a LINP problem $\langle \mathcal{D}, s_{\mathcal{I}}, \mathcal{G} \rangle$ where $s_{\mathcal{I}}$ is an initial state s.t. if the numeric variable v has no maximum under $\mathcal{I}$, then $v(s_{\mathcal{I}}) = b$ for every $v \in \mathcal{V}$. Metric-FF and ENHSP solve every problem with different bound individually. With the growth of the bound b, Metric-FF and ENHSP spend an increasing amount of time in solving the LINP problem. In contrast, RegexSkeleton learns a planning program from sequential plans of some LINP problems with a small bound. Hence, no matter how large the bound b is, the time in generating a planning program by RegexSkeleton is steady. Furthermore, RegexSkeleton outperforms Metric-FF and ENHSP in efficiency when the bound b exceeds a certain threshold, which is shown in Figure 2. For example, Metric-FF solves the Arith problem slower than RegexSkeleton when the bound b is greater than 515. Finally, we remark that Metric-FF and ENHSP cannot solve the LINP problems even with small bound. Metric-FF fails on MNestVar8 problem with bound 4 and ENHSP fails on Corner-R, D-Return-R, Hall-R and VisitAll-R problems with bound 1. Hence, our approach is a promising alternative to numeric planning.

Related Work

Levesque (2005) firstly proposed GP problem that aims to generate a loopy plan to solve problems for infinitely many states. He identified a class of GP problems, called one-dimensional (1d) planning problems in (Hu and Levesque 2010), that contains only one numeric variable. Levesque's method is similar to our approach, which firstly searches a plan without loop structure that works for the case where the value of the variable is less than a small threshold, and then try to roll the plan into a planning program via the pattern match mechanism implemented in Prolog language. Hu and Levesque (2010) proved that the plan existence of 1d planning problems is decidable, more precisely, in EXPSPACE (Hu and Giacomo 2011). Our approach provides an effective way of constructing planning programs to GLINP with multiple variables while Levesque's method only suits GP problems with only one variable.

Srivastava, Immerman, and Zilberstein (2011) proposed a method to generate a finite-state automata (FSA) plan based on state abstraction using 3-valued logic (Sagiv, Reps, and Wilhelm 2002). It starts from an extended sequential plan $\pi : [(s_0, a_0), \cdots, (s_n, a_n), (s_{n+1})]$ that is a sequence of state-action pairs (s_i, a_i) with a final state s_{n+1}. Based on the concrete plan, it generates an abstract sequential plan $\pi' : [(S_0, a_0), \cdots, (S_n, a_n), (S_{n+1})]$ by generalizing each concrete state s_i into an abstract one S_i. If two pairs (S_j, a_j) and (S_k, a_k) of π' where $j < k$ are identical, then such repeated pairs mean some properties that are true in S_j hold again in the successor abstract state S_k. Hence, a cycle representing the repetition of the sequence of actions $[a_j, \cdots, a_{k-1}]$ should be created in the FSA plan. The solution generated by the Srivastava, Immerman, and Zilberstein's approach does not guarantee goal achievement. In order to measure the condition when the plan is guaranteed to terminate and lead to the goal, namely *applicability condition*, they devised an algorithm for computing such condition of the FSA plan when the plan involves only simple loop structures (Srivastava, Immerman, and Zilberstein 2012). However, the FSA plan can not contain a nested loop and is not applicable to every initial state. In contrast, the planning programs we synthesize can involve nested loop structures and the experimental results demonstrate that the planning programs are correct for every initial state.

Bonet, Palacios, and Geffner (2009) developed an automatic way for deriving a finite state controller (FSC) for

a contingent planning problem. The experimental results show that FSCs derived by this method are general in the sense that they not only solve the original contingent planning problem but also many variations including changes in the size of the problem. However, it is not clear which variation of the original problem the resulting FSC is capable of solving. This was clarified by Bonet and Geffner (2015) via providing a characterization of the common structure that allows for policy generalization. Observations in an FSC essentially corresponds to conditions of branch structures and loop structures in a planning problem. Our approach automatically synthesizes conditions but observations in Bonet, Palacios, and Geffner's method are user-provided.

Segovia-Aguas, Jiménez, and Jonsson (2018) encoded a generalized planning problem with a finite set S of initial states as a classical planning problem. The form of their solutions are hierarchical finite state controllers, allowing nested loops. Later, they extended their method to synthesize planning programs (Segovia-Aguas, Jiménez, and Jonsson 2019). The planning program they investigate on, which uses goto statement rather than while statement to denote the loop structure, is slightly different from ours. The experimental results show that the planning program synthesized by their approach is not applicable to other initial states that are not mentioned in S. This is because that their approach does not identify the common structure of all solutions to all states of S. Segovia-Aguas, Jiménez, and Jonsson (2021) further uses a heuristic search paradigm to generate a planning program applicable to other initial states not in S in practices. However, the above work cannot handle numeric variables while our approach does.

Conformant planning, proposed by Goldman and Boddy (1996), is the problem of finding a plan that guarantees goal achievement in a finite-state nondeterministic domain. To solve this class of problems, Nguyen et al. (2012) developed a generate-and-complete approach similar to our approach. It first finds a plan for a subproblem by utilizing a classical planner and then tries to repair it to account for other initial states. Our approach considers the planning problem with infinite states while Nguyen et al.'s method only suits conformant planning problems that contains only finite states.

Golog, proposed by Levesque et al. (1997), is a high-level logic programming language based on the situation calculus. It is very close to the planning program of our work. As far as we know, there is little work on synthesis of Golog programs, and most of the research focus on verification (Claßen et al. 2014; Li and Liu 2015).

A majority of state-of-the-art approaches to inferring a regex (Lee, So, and Oh 2016; Jain, Kinber, and Stephan 2017) requires negative strings, that is, the string not accepted by the target regex. However, these approaches are not suitable for our framework since we do not generate an action sequence that is not an solution for any initial state. Fernau (2009) developed an approach to regex inference from only positive strings. Yet, this method only generate regexes of depth at most 1. Another work, proposed by Kinber (2010), can learn a regex of depth at most 2 from one representative string. However, the learned regex contains no alternation subregexes. Compared with the above methods, our method does not rely on negative examples and is able to infer the regex of depth more than 2 that contains alternation subregexes.

Conclusion and Future Work

In this paper, we have proposed a generalized version of numeric planning (GLINP), which is a more suitable abstract framework of GP than QNP. Then, we have developed an approach to synthesizing planning programs inductively from a set of state-plan pairs. Finally, we have implemented our approach and experimental results have justified the feasibility and effectiveness of the approach.

Our approach have some limitations, and hence leading to several avenues for future work. Firstly, in this paper, our approach is able to solve a class of GLINP problems where the goal is to decrease the values of some numeric variables. Thus, we would like to extend our approach to handle more complex GLINP problems. Secondly, the existence of solutions to GLINP is in general undecidable. We will identify the decidable fragment in future work.

Acknowledgements

This paper was supported by National Natural Science Foundation of China (Nos. 62077028, 61906216, 61877029 and 61603152), the Guangdong Basic and Applied Basic Research Foundation (Nos. 2021A1515011873, 2021B0101420003, 2021B1515120048, 2020A1515010642, 2020B0909030005, 2020B1212030003, 2020ZDZX3013, 2019A050510024 and 2019B1515120010), the Specially Creative University Project for Guangdong (Nos. 2019KTSCX010 and 2018KTSCX016), the Science and Technology Planning Project of Guangzhou (Nos. 202102080307 and 201902010041), the Teaching Reform Research Projects of Jinan University (No. JG2021112), the Project of Guangxi Key Laboratory of Trusted Software (No. kx202007), and the High Performance Public Computing Service Platform of Jinan University.

References

Bonet, B.; Francès, G.; and Geffner, H. 2019. Learning Features and Abstract Actions for Computing Generalized Plans. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-2019)*, 2703–2710.

Bonet, B.; and Geffner, H. 2015. Policies that Generalize: Solving Many Planning Problems with the Same Policy. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-2015)*, 2798–2804.

Bonet, B.; and Geffner, H. 2018. Features, Projections, and Representation Change for Generalized Planning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-2018)*, 4667–4673.

Bonet, B.; and Geffner, H. 2020. Qualitative Numerical Planning: Reductions and Complexity. *Journal of Artificial Intelligence Research*, 923–961.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In *Proceedings of the Nineteenth International*

Conference on Automated Planning and Scheduling (ICAPS-2009), 34–41.

Brāzma, A. 1993. Efficient Identification of Regular Expressions from Representative Examples. In *Proceedings of the Sixth Annual ACM Conference on Computational Learning Theory (COLT-1993)*, 236–242.

Claßen, J.; Liebenberg, M.; Lakemeyer, G.; and Zarrieß, B. 2014. Exploring the Boundaries of Decidable Verification of Non-Terminating Golog Programs. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI-2014)*, 1012–1019.

de Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In *Proceedings of Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-2008)*, 337–340. Springer.

Fernau, H. 2009. Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4): 521–541.

Goldman, R. P.; and Boddy, M. S. 1996. Expressive Planning and Explicit Knowledge. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-1996)*, 110–117.

Helmert, M. 2002. Decidability and Undecidability Results for Planning with Numerical State Variables. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS-2002)*, 44–53.

Hoffmann, J. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research*, 20: 291–341.

Hu, Y.; and Giacomo, G. D. 2011. Generalized Planning: Synthesizing Plans that Work for Multiple Environments. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI-2011)*, 918–923.

Hu, Y.; and Levesque, H. J. 2010. A Correctness Result for Reasoning about One-Dimensional Planning Problems. In *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR-2010)*, 362–371.

Illanes, L.; and McIlraith, S. A. 2019. Generalized Planning via Abstraction: Arbitrary Numbers of Objects. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-2019)*, 7610–7618.

Jain, S.; Kinber, E.; and Stephan, F. 2017. Automatic learning from positive data and negative counterexamples. *Information and Computation*, 255: 45–67.

Kinber, E. 2010. Learning Regular Expressions from Representative Examples and Membership Queries. In *Proceedings of the Tenth International Colloquium Conference on Grammatical Inference (ICGI-2010)*, 94–108.

Lang, J.; and Zanuttini, B. 2013. Knowledge-Based Programs as Plans: Succinctness and the Complexity of Plan Existence. In *Proceedings of the Fourteenth Conference on Theoretical Aspects of Rationality and Knowledge (TARK-2013)*, 138–147.

Lee, M.; So, S.; and Oh, H. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the Fifteenth International Conference on Generative Programming: Concepts and Experience (GPCE-2016)*, 70–80.

Levesque, H. J. 2005. Planning with Loops. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-2005)*, 509–515.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1): 59–83.

Li, N.; and Liu, Y. 2015. Automatic Verification of Partial Correctness of Golog Programs. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-2015)*, 3113–3119.

Nguyen, K. H.; Tran, V. D.; Son, T. C.; and Pontelli, E. 2012. On Computing Conformant Plans Using Classical Planners: A Generate-And-Complete Approach. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS-2012)*, 190–198.

Sagiv, M.; Reps, T.; and Wilhelm, R. 2002. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3): 217–298.

Scala, E.; Haslum, P.; and Thiébaux, S. 2016. Heuristics for Numeric Planning via Subgoaling. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-2016)*, 3228–3234.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2018. Computing Hierarchical Finite State Controllers with classical planner. *Journal of Artificial Intelligence Research*, 62: 755–797.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2019. Computing programs for generalized planning using a classical planner. *Artificial Intelligence*, 272: 52–85.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2021. Generalized Planning as Heuristic Search. In *Proceedings of the Thirty-first International Conference on Automated Planning and Scheduling (ICAPS-2021)*, 569–577.

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

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2012. Applicability conditions for plans with loops: Computability results and algorithms. *Artificial Intelligence*, 191: 1–19.

Srivastava, S.; Zilberstein, S.; Immerman, N.; and Geffner, H. 2011. Qualitative Numeric Planning. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI-2011)*, 1010–1016.

Udupa, A.; Raghavan, A.; Deshmukh, J. V.; Mador-Haim, S.; Martin, M. M. K.; and Alur, R. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the Thirty-Fourth annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-2013)*, 287–296.

Winner, E.; and Veloso, M. 2003. DISTILL: Learning Domain-Specific Planners by Example. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, 800–807.