Generalized Linear Integer Numeric Planning

Xiaoyou Lin¹, Qingliang Chen¹, Liangda Fang¹,³,⁴*, Quanlong Guan¹,²*, Weiqi Luo¹, Kaile Su¹

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; kailepu@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 (sometimes 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-
tured, 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 (LIA¹), and regular expressions (regexes).

LIA¹ Let \( \mathbb{B} \) be the set of Booleans \( \{ \top, \bot \} \) and \( \mathbb{Z} \) is the set of integers. Throughout this paper, we fix a set \( P \) of propositional variables and a set \( V \) of numeric variables. The sets of terms (Term) and formulas (Form) of LIA¹ are defined recursively as:

\[
e \in \text{Term} :: c | v | e + e | e - e
\]

\[
\phi \in \text{Form} :: \top | \bot | p | e = e | e < e | \neg \phi | \phi \land \phi
\]

where \( c \in \mathbb{Z} \) and \( v \in V \).

The formula \( \phi_1 \lor \phi_2 \) is the shorthand for \( \neg(\neg \phi_1 \land \neg \phi_2) \), \( e_1 \leq e_2 \) for \( e_1 = e_2 \lor e_1 < e_2 \), and \( 1 \rightarrow e_2 \) for \( \neg e_1 \lor e_2 \). The length \( |\phi| \) of a formula \( \phi \) is the number of occurrences of Boolean constants, integers, propositional variables, numeric variables, arithmetic operators and logical connectives in \( \phi \).

A state \( s \) of LIA¹ is a pair of mappings \( P \rightarrow \mathbb{B} \) and \( V \rightarrow \mathbb{Z} \). 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 \( \phi \) can be determined in a similar way. A state \( s \) satisfies a formula \( \phi \), if \( \phi(s) = \top \).

A formula \( \phi \) entails another one \( \psi \), denoted by \( \phi \models \psi \), if for every state \( s \) satisfying \( \phi \), \( \psi(s) = \top \).

A propositional variable \( p \) is determined by a formula \( \phi \), if \( \phi \models p \) or \( \phi \models \neg p \).

A numeric variable \( v \) has a maximum \( c \) under \( \phi \), 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 \( \Delta \). We use \( |\pi| \) for the length of \( \pi \) and \( \pi_i \) for the \( i \)-th character of \( \pi \). A substring of \( \pi \) is \( \pi_i \pi_{i+1} \ldots \pi_j \), denoted by \( \pi_{i,j} \), where \( 1 \leq i \leq j \leq |\pi| \). The substring \( \pi^+ \) is a prefix of \( \pi \) while \( \pi^- \) is a suffix. A subsequence of \( \pi \) is \( \pi_{i_1} \pi_{i_2} \ldots \pi_{i_j} \) where \( 1 \leq i_1 < i_2 < \ldots < i_j \leq |\pi| \). We remark that a substring is a subsequence, but not vice versa. For example, \( aba \) is a subsequence of \( abba \), but it is not a substring.

The set of regexes (Reg) is recursively defined as:

\[
r \in \text{Reg} :: \varepsilon | a | r \circ r | (r|r) | r^*
\]

where \( \varepsilon \) 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 \( \text{SubReg}(r) \) of a regex \( r \) is recursively defined as:

\[
\text{SubReg}(\varepsilon) = \{ \varepsilon \} \text{ and } \text{SubReg}(a) = \{ a \} \text{ where } a \in \Delta;
\]

\[
\text{SubReg}(r_1 \circ r_2) = \{ r_1 \circ r_2 \} \cup \text{SubReg}(r_1) \cup \text{SubReg}(r_2);
\]

\[
\text{SubReg}(r_1|r_2) = \{ r_1|r_2 \} \cup \text{SubReg}(r_1) \cup \text{SubReg}(r_2);
\]

\[
\text{SubReg}(r^*_1) = \{ r^*_1 \} \cup \text{SubReg}(r_1).
\]

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

\[
L(\varepsilon) = \{ \varepsilon \} \text{ and } L(a) = \{ a \} \text{ where } a \in \Delta;
\]

\[
L(r_1 \circ r_2) = \{ s_1|s_2 \mid s_1 \in L(r_1) \text{ and } s_2 \in L(r_2) \};
\]

\[
L(r_1|r_2) = L(r_1) \cup L(r_2);
\]

\[
L(r^*_1) = \{ s_1 \ldots s_n \mid n \geq 1 \text{ and } s_1, \ldots, s_n \in L(r_1) \}.
\]

Generalized Linear Integer Numeric Planning

In this section, we first introduce concepts on Linear Integer Numeric Planning (LINP) formalized in LIA¹, and then provide the definition of GLINP, and finally give an algorithmic-like definition of solutions to GLINP problems.

Definition 1. A LINP domain \( D \) is a tuple \( (P, V, A) \) where

\[
P: \text{a finite set of propositional variables};
\]

\[
V: \text{a finite set of numeric variables};
\]

\[
A: \text{a finite set of actions defined by a pair } (\text{pre}, \text{eff})\text{ where } \text{pre} \in \text{Form} \text{ denotes the precondition and } \text{eff} \text{ is an finite set of propositional and numeric effects}.
\]

A propositional effect is a tuple \( (\phi, p, \psi) \) where \( p \in P \) and \( \phi, \psi \in \text{Form} \). Intuitively, it means that if \( \phi \) 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 \( (\phi, v, e) \) where \( \phi \in \text{Form}, v \in V \) and \( e \in \text{Term} \). 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 \( s \models \text{pre}(a) \). The successor state of applying an action \( a \) over \( s \) is written as \( s(a), \) which results from \( s \) by mapping \( p \) to \( p(s) \) (i.e., \( p(\tau(s,a)) = \phi(s) \)) for every \( (\phi, p, \psi) \in \text{eff}(a) \), and by mapping \( v \) to \( e(s) \) (i.e., \( v(\tau(s,a)) = \psi(s) \)) for all \( (\phi, v, e) \in \text{eff}(a) \). We remark that \( (s,a) \) is well-defined even if \( a \) is not executable in \( s \). The resulting state of performing a finite sequence \( [a_1, \ldots, a_n] \) of actions on \( s \) is recursively defined by \( \tau(s, [a_1, \ldots, a_n]) = \tau(\tau(s, [a_1, \ldots, a_{n-1}]), a_n) \).
The solutions to GLINP problems are planning programs.

**Definition 3.** The set of planning programs (Prog) for an LINP domain \( D = (P, V, A) \) is recursively defined by
\[
\delta \in \text{Prog} :: \varepsilon | a | \delta ; \delta | \text{if } \phi \text{ then } \delta \text{ else } \delta \text{ } \text{fi} | \text{while } \phi \text{ do } \delta \text{ od}
\]
where \( a \in A \) and \( \phi \in \text{Form} \).

The construct \( \delta_1 ; \delta_2 \) is the sequential structure; if \( \phi \) then \( \delta_1 \) else \( \delta_2 \) fi is the branch structure and while \( \phi \) do \( \delta \) od is the loop structure. We say \( \phi \) is the condition of the branch structure if \( \phi \) then \( \delta_1 \) else \( \delta_2 \) fi. Likewise, \( \phi \) is the condition of the loop structure while \( \phi \) do \( \delta \) od.

**Definition 4.** Let \( \delta \) be a planning program. The length \( |\delta| \) of \( \delta \) is recursively defined as:
\[
|\varepsilon| = 0 \text{ and } |a| = 1 \text{ where } a \in A; \\
|\delta_1 ; \delta_2| = |\delta_1| + |\delta_2| + 1; \\
|\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ } \text{fi}| = |\phi| + |\delta_1| + |\delta_2| + 1; \\
|\text{while } \phi \text{ do } \delta \text{ od}| = |\phi| + |\delta_1| + 1.
\]

where \( |\phi| \) is the length of \( \phi \).

**Definition 5.** Let \( \delta \) be a planning program. The depth \( \#(\delta) \) of \( \delta \) is recursively defined as:
\[
\#(\varepsilon) = 0 \text{ and } \#(a) = 0 \text{ where } a \in A; \\
\#(\delta_1 ; \delta_2) = \max(\#(\delta_1), \#(\delta_2)); \\
\#(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ } \text{fi}) = \max(\#(\delta_1), \#(\delta_2)); \\
\#(\text{while } \phi \text{ do } \delta \text{ od}) = 1 + \#(\delta_1).
\]

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

The action sequence of executing a program \( \delta \) in a state \( s \) is defined as follows:

**Definition 6.** Let \( D = (P, V, A) \) be an LINP domain, \( \delta \) a program for \( D \) and \( s \) a state. The action sequence \( \Theta(s, \delta) \) of executing \( \delta \) in \( s \) is recursively defined as:
\[
\Theta(s, \varepsilon) = \varepsilon. \\
\Theta(s, a) = a \text{ where } a \in A. \\
\Theta(s, \delta_1 ; \delta_2) = \\
\left\{ \begin{array}{ll}
\Theta(s, \delta_1) \circ \Theta(\tau(s, \delta_1), \delta_2), & \text{if } \Theta(s, \delta_1) \text{ is finite;} \\
\Theta(s, \delta_1), & \text{otherwise.}
\end{array} \right.
\]
\[
\Theta(s, \text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ } \text{fi}) = \Theta(s, \delta_1), \text{ if } \phi(s) = T; \\
\Theta(s, \delta_2), \text{ otherwise.}
\]
\[
\Theta(s, \text{while } \phi \text{ do } \delta \text{ od}) = \\
\left\{ \begin{array}{ll}
\Theta(s, \delta_1) \circ \Theta(\tau(s, \delta_1), \text{while } \phi \text{ do } \delta \text{ od}), & \text{if } \phi(s) = T \text{ and } \Theta(s, \delta_1) \text{ is finite;} \\
\Theta(s, \delta_1), & \text{if } \phi(s) = T \text{ and } \Theta(s, \delta_1) \text{ is infinite;} \\
\varepsilon, & \text{otherwise.}
\end{array} \right.
\]

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 \( \Theta_1 \) and \( \Theta_2 \).

Given a GLINP problem, we are interested in synthesizing a program \( \delta \) satisfying the following three properties.

**Definition 7.** Let \( D = (P, V, A) \) be an LINP domain, \( \delta \) a program for \( D \) and \( s \) a state. The program \( \delta \) is
\[
\text{terminating in } s, \text{ if } \Theta(s, \delta) \text{ is finite;} \\
\text{executable in } s, \text{ if } \Theta(s, \delta) \text{ is executable in } s;
\]
• $\phi$-reaching in $s$, iff $\delta$ 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 $\Sigma = (\mathcal{D}, \mathcal{I}, \mathcal{G})$ be a GLINP problem. A program $\delta$ is a solution to $\Sigma$, if $\delta$ 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 $\Delta$ is the set $\mathcal{A}$ of actions. A sequence $\pi$ of actions is a string over $\Delta$. 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 $\phi$ then $\delta_1$ else $\delta_2$ fi is omitted, then it corresponds to $\delta_1 \mid \delta_2$. Similarly, the loop structure while $\phi$ do $\delta$ od corresponds to the iteration regex $\delta^*$. 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 $\Upsilon$ of state-plan pairs, including generating initial states $s$ by imposing some restrictions on propositional and numeric variables and computing the corresponding solutions $\pi$ by the planner; (2) infer a skeleton of the planning program $r$ expressed as a regex according to the plans of $\Upsilon$; and (3) obtain a complete planning program $\delta$ by filling the missing conditions in $r$ according to $\Upsilon$.

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 (Brãzma 1993; Kinber 2010). A string $\pi$ is representative for the regex $r$, if the generator of any iteration subregex of $r$ consecutively occurs in $\pi$ 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 $(\text{load}_d(s), \text{unload}_d(s), \text{num}_c(s), \text{num}_m(s), \text{cap}(s))$. The initial formula $\mathcal{I}$ is $\text{num}_d > 0 \land \text{num}_c = 0 \land \text{num}_m = 0 \land \text{cap} > 0$. Both $\text{num}_c$ and $\text{num}_m$ have the maximum 0 under $\mathcal{I}$. However, neither $\text{num}_d$ nor $\text{cap}$ has a maximum. The variable $\text{load}_d$ is not determined by $\mathcal{I}$. In light of the principles of initial states, the procedure GenStatePlanPairs first generates three initial states: $s_1 : (T, 8, 0, 0, 3)$, $s_2 : (L, 10, 0, 0, 4)$ and $s_3 : (L, 9, 0, 0, 3)$. The numeric planner then computes the plan $\pi_i$ for each state $s_i$ as follows. For brevity, we abbreviate $\text{load}_d$ as $L$, $\text{unload}_d$ as $U$, $\text{move}_c$ as $C$ and $\text{move}_d$ as $D$.


### 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 $\rho$ (Lines 1 - 8). It starts from the original alphabet $\Delta = \mathcal{A}$ (Line 3), and infers a regex $\rho'$ with iteration subregexes over $\Delta$ accepting $\pi$ (Line 4). If the current regex $\rho'$ is different from the previous one $\rho$, 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. The regex $\rho'$ can
be considered as a string over the new alphabet $\Delta$. 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 $\Delta$ 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 $\Delta^*$ is enlarged by $\Delta$ (Line 8).

**Example 3.** We continue with Example 2. We first infer the regex $t_1$ for the string $\pi_1$. The process is shown in Table 1. At the first iteration, the alphabet $\Delta$ 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 $\pi_1$ and $U$ occurs from the 5-th to 7-th characters. It generates $t_2$ as $(L^*CU^*DL^*CU^*D)^*$. At the second iteration, the alphabet $\Delta$ 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

**Algorithm 1: InfSkeleton($\Sigma, \Upsilon$)**

**Input:** $\Sigma$: the planning problem $(D, I, G)$ where $D = (\mathcal{P}, \mathcal{V}, \mathcal{A})$; $\Upsilon$: the finite set of state-plan pairs

**Output:** $r$: a skeleton of the planning program

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. $(t', \Delta') \leftarrow \text{FoldString}(t, \Delta)$
   5. while $t \neq t'$ do
      6. $\Delta \leftarrow \Delta \cup \Delta'$ and $t \leftarrow t'$
      7. $(t', \Delta') \leftarrow \text{FoldString}(t, \Delta)$
   8. $R \leftarrow R \cup \{t\}$ and $\Delta^* \leftarrow \Delta^* \cup \Delta$

9. foreach $t \in R$ do
   10. foreach misaligned subregex of $t$ with the form $(u_1u_2)^*$ do
      11. Replace every occurrence of $(u_1u_2)^*u_1$ in $t$ by $u_1(u_2u_1)^*$
      12. $(\xi_1, \ldots, \xi_l) \leftarrow$ the sequence of non-extensible common strings of $R$ over $\Delta^*$
      13. Compute each $i$-th individual components $\eta_{i,j}$ of $t_1$ s.t.
          $t_1 = \eta_{1,1} \circ \xi_1 \circ \cdots \circ \eta_{i,j} \circ \xi_i \circ \eta_{i+1,j}$ for $1 \leq i \leq l + 1$
          and $1 \leq j \leq k$
      14. $\eta_{i} \leftarrow \eta_{i,1} \cdots \eta_{i,k}$ for $1 \leq i \leq l + 1$
      15. $r \leftarrow \eta_1 \circ \xi_1 \circ \eta_2 \circ \xi_2 \cdots \circ \eta_l \circ \xi_l \circ \eta_{l+1}$
   16. Simplify $r$

Table 1: The process of identifying iteration subregexes of $\pi_1$ where $t$ denotes the resulting regex at each iteration and $\Delta$ denotes the extended alphabet of each iteration.

<table>
<thead>
<tr>
<th>Iter.</th>
<th>$t$</th>
<th>$\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$L^*CU^*DL^*CU^*D$</td>
<td>$\Delta \cup {L^<em>, U^</em>}$</td>
</tr>
<tr>
<td>2</td>
<td>$(L^*CU^<em>D)^</em>$</td>
<td>$\Delta \cup {L^<em>, U^</em>, (L^*CU^<em>D)^</em>}$</td>
</tr>
<tr>
<td>3</td>
<td>$(L^*CU^<em>D)^</em>$</td>
<td>$\Delta \cup {L^<em>, U^</em>, (L^*CU^<em>D)^</em>}$</td>
</tr>
</tbody>
</table>

**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 $(u_1u_2)^*u_1$. 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 regexes of $R$ than $t$.

We do not consider replacing the occurrence of subregex of the form $(u_1u_2)^*u_1$ by $(u_1u_2)^*u_1$ in every regex of $R$. The reason is as follows. Suppose that a string $\pi$ contains the substring $\pi' = u_1 \circ u_2 \circ \cdots \circ u_l \circ u_1 \circ u_2 \circ u_1$ 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 $\pi$ 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 \(\xi\) is a common substring of \(R\), if \(\xi\) 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 \(\xi\) is a common subsequence of \(R\), if \(\xi\) is a substring 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 \(\xi_l\) is a non-extensible common substring of \(R\) (more precisely, the concatenation \((\xi_{l-1})\xi_{l+1} \circ \xi_l\) of the last character of \(\xi_{l-1}\) and \(\xi_l\) is not a common substring of \(R\) for \(i > 1\), and the concatenation \(\xi_l \circ (\xi_{l+1})\) of \(\xi_l\) and the first character of \(\xi_{l+1}\) 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 \(\eta_{i,j} \circ \xi_l \circ \xi_{i,j} \circ \eta_{i,j+1}\) where \(\eta_{i,j}\) may be an empty string for \(1 \leq i \leq l + 1\) and \(1 \leq j \leq k\). Each \(\eta_{i,j}\) called the \(i\)-th individual component of \(t_j\). Meanwhile, each \(\xi_l\) is called the common component of \(R\).

We remind that each regex \(t\) of \(R\) is a string over \(\Delta^*\).

The final regex \(r\) is obtained as follows. We firstly generate all the \(i\)-th individual component of \(t_j\)'s with alternation connectives (i.e., \(\eta_{i,j} \circ \cdots \circ \eta_{i,k}\)) for each \(1 \leq i \leq l + 1\) (Line 14). We concatenate the combination of individual components \(\eta_i\) and the common component \(\xi_i\) alternatively in increasing order (Line 15). In the end, some redundant subregexes in \(r\) are removed (Line 16) (e.g., \(u_1|u_2| \cdots |u_{k-1}|u_2|u_1\) simplified as \(u_1|u_2| \cdots |u_{k-1}|u_1\)).

**Example 5.** We continue with the Example 4. The set \(R\) contains two regexes: \(t_1 = (L^*CU^*D)^*\) and \(t_2 = (DL^*CU^*D)^*\). 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 \(\eta_1\) and \(\xi_1\), 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 \(\Sigma = (D, I, \mathcal{G})\) be a GLNIP problem and \(\Upsilon\) a set of state-plan pairs. Then, `InfSkeleton`(\(\Sigma, \Upsilon\)) 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 \(u_{j} \in \pi_{j}\) by \(u^*\), and obtain the regex \(t_j\). So each \(t_j\), which is added into the set \(R\), accepts \(\pi_j\). Then, the alignment step preserves equivalence. In addition, each \(t_j\) is divided into a sequence of subregexes (i.e., \(\eta_{1,j} \circ \xi_1 \circ \cdots \circ \eta_{i,j} \circ \xi_i \circ \eta_{i,j+1}\) where \(\eta_{i,j}\) is an individual component of \(t_j\) and \(\xi_i\) is a common component of \(R\)). The final regex \(r\) is \(\eta_1 \circ \xi_1 \circ \cdots \circ \eta_k \circ \xi_k\). Obviously, \(r\) accepts every \(\pi_i \in \Pi(\Upsilon)\).
Completion of Planning Programs

In this section, we illustrate the procedure Complete which takes a skeleton $r$ and a set of state-plan pairs $\mathcal{Y}$ as input, and outputs a complete planning program $\delta$.

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 $\mathcal{Y}$ for each condition $\phi$ in a recursive way. A state $s$ is a positive (resp. negative) state of $\phi$, 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 $\phi$ that is consistent with the two sets $S^+$ and $S^-$. If $\phi(s)$ is also synthesized based on $s$ according to the three state-plan pairs: $(s_1, \pi_1)$, $(s_2, \pi_2)$ and $(s_3, \pi_3)$ 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 $\pi_1$ for $s_1$. Clearly, $\varepsilon \in \mathcal{L}(\varepsilon)$ and $\pi_1 \in \mathcal{L}(\mathcal{L}^{\ast}(\mathcal{C}^u \mathcal{D})^\ast)$. So $(s_1, \varepsilon) \in \mathcal{Y}$ and $(s_1, \pi_1) \in \mathcal{Y}$. Similarly, we get that $\mathcal{Y}_1 = \{(s_1, \varepsilon), (s_2, D), (s_3, D)\}$ and $\mathcal{Y}_2 = \{(s_1, \pi_1), (s_2, \pi_2), (s_3, \pi_3)\}$ where $s_1 = (\top, 10, 0, 0, 4)$ and $s_2 = (\bot, 9, 0, 0, 3)$.

We start to synthesize the condition $\phi_1$ from $\mathcal{Y}_1$. It is easily verified that $S^+_1 = \{s_1\}$ and $S^-_1 = \{s_2, s_3\}$. The algorithm SynFormula deduces that $\phi_1$ is $\text{at}_{\delta_1}$. We now synthesize the condition $\phi_2$ from $\mathcal{Y}_2$. By tracking the trace of performing $\pi_1$ (resp. $\pi_2$) on $s_1$ (resp. $s_2$) on $s_1$), we get that $S^+_2 = \{(\top, 8, 0, 0, 3), (\top, 10, 0, 0, 4), (\top, 9, 0, 0, 3), (\top, 5, 3, 0, 3), (\top, 6, 4, 0, 4), (\top, 6, 3, 0, 3), (\top, 2, 0, 0, 3), (\top, 2, 8, 0, 4), (\top, 3, 6, 0, 3)\}$ and $S^-_2 = \{(\top, 0, 8, 0, 3), (\top, 0, 10, 0, 4), (\top, 9, 0, 3)\}$. The algorithm SynFormula synthesizes $\phi_2$ as $\text{num}_d \geq 1$. During the process of synthesizing $\phi_2$, the conditions $\phi_3$ and $\phi_4$ of the two inner loops of $\delta_2$ are also constructed as $\text{num}_d \geq 1 \land \text{cap} > \text{num}_d$ and $\text{num}_d \geq 1$ respectively.

The entire planning program is as follows:

if $\text{at}_{\delta_1}$ then $\varepsilon \in \text{else move}_{\delta_1}$; while $\phi_2$ do

while $\phi_1$ do load$_{\delta}$ od;

while $\phi_2$ do unload$_{\delta}$ od;

od

The following theorem states that the procedure Complete synthesizes a correct program for every state $s \in S(\mathcal{T})$.

**Theorem 2.** Let $\Sigma = (\mathcal{D}, \mathcal{I}, \mathcal{G})$. Let $r$ be a skeleton of a planning program of $\mathcal{D}$ and $\mathcal{Y}$ a set of state-plan pairs where $r$ accepts each plan $\pi$ with $\pi \in \mathcal{P}(\mathcal{T})$. Then, $\text{Complete}(r, \mathcal{Y})$ outputs a program $\delta$ that is terminating, executable and $\mathcal{G}$-reaching in every state $s \in S(\mathcal{T})$.

**Proof.** We firstly prove that $\pi = \Theta(s, \delta)$ for $(s, \pi) \in \mathcal{T}$. We prove by induction on $\delta$.

- $\delta = \varepsilon$ or $\delta = \alpha$: Suppose that $\delta = \varepsilon$. Since $r$ accepts $\pi$, $\pi = \varepsilon$, and hence $\Theta(s, \delta) = \varepsilon = \pi$. Similarly, $\Theta(s, \alpha) = |a| = \pi$ when $\delta = \alpha$.
• $\delta = \delta_1; \delta_2$: It follows that $r = r_1 \circ r_2$ where $r_i$ corresponds to the program $\delta_i$ for $i = 1, 2$. Since $r$ accepts $\pi$, there is an index $i$ s.t. $r_1$ accepts $\pi^i_1$ and $r_2$ accepts $\pi^i_{i+1}$. According to the procedure Complete (Algorithm 2), $(s, \pi^i_1) \in T_1$ and $(\tau(s, \pi^i_1), \pi^i_{i+1}) \in T_2$. By the inductive assumption, the procedure Complete synthesizes two programs $\delta_1$ and $\delta_2$ s.t. $\pi^i_1 = \Theta(s, \delta_1)$ and $\pi^i_{i+1} = \Theta(\tau(s, \pi^i_1), \delta_2)$. Hence, $\pi = \pi^i_1 \circ \pi^i_{i+1} = \Theta(s, \delta_1) \circ \Theta(\tau(s, \pi^i_1), \delta_2) = \Theta(s, \delta_1; \delta_2)$.

• $\delta = \text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2$: It follows that $r = r_1 \circ r_2$ where $r_1$ corresponds to the program $\delta_1$ for $i = 1, 2$. Suppose that $\phi(s) = T$. Thus, $\Theta(s, \delta) = \Theta(s, \delta_1)$. Since the procedure SynFormula constructs the condition $\phi$ consistent with $S^+$ and $S^-$. So $s \in S^+$ and $(s, \pi) \in T_1$. By the inductive assumption, the procedure Complete synthesizes a program $\delta_1$ s.t. $\pi = \Theta(s, \delta_1)$. Hence, the case where $\phi(s) = \bot$ can be similarly proved.

• $\delta = \text{while } \phi \text{ do } \delta_1 \text{ od}$: It follows that $r = r_1^i$ where $r_1$ corresponds to the program $\delta_1$. The case where $\pi = \varepsilon$ is easier. Suppose that $\pi \neq \varepsilon$. Since $r$ accepts $\pi$, there is a sequence of substrings $(\pi_1, \pi_2, \ldots, \pi_k)$ s.t. $\pi_1 \circ \pi_2 \circ \cdots \circ \pi_k = \pi$ and $\pi_i \in L(r_1)$ for $1 \leq i \leq k$. According to the procedure Complete, $(s, \pi_1) \in T_1$ and $(\tau(s, \pi_1 \circ \cdots \circ \pi_k), \pi_{i+1}) \in T_1$ for $1 \leq i < k$. By the inductive assumption, the procedure Complete synthesizes a program $\delta_1$ s.t. $\pi_1 = \Theta(s, \delta_1)$ and $\pi_1 \circ \cdots \circ \pi_k$ is a terminating, executable and $G$-reaching in $s$.

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

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.

<table>
<thead>
<tr>
<th>Domain</th>
<th>RegexSkeleton</th>
<th>QNP2FOND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$#(\delta)$</td>
<td>$</td>
</tr>
<tr>
<td>Arith</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>Chop</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>ClearBlock</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Corner-A</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>Corner-R</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>Delivery</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>D-Return</td>
<td>1</td>
<td>41</td>
</tr>
<tr>
<td>D-Return-R</td>
<td>2</td>
<td>52</td>
</tr>
<tr>
<td>Gripper</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>Hall-A</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>Hall-R</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>NestVar</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>NestVar8</td>
<td>1</td>
<td>541</td>
</tr>
<tr>
<td>MNestVar</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>MNestVar8</td>
<td>8</td>
<td>287</td>
</tr>
<tr>
<td>PlaceBlock</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>Rewards</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Snow</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>Spanner</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>TestOn</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>VisitAll</td>
<td>2</td>
<td>37</td>
</tr>
<tr>
<td>VisitAll-R</td>
<td>2</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 2: Experimental results. “$\#(\delta)$”, “$|\delta|$”, “$|p|$”, “$t$” 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 Gefenner 2009; Bonet, Francès, and Gefenner 2019; Srivastava, Immerman, and Zilberstein 2011; Srivastava et al. 2011; Bonet and Gefenner 2020). We compare the performance of RegexSkeleton with the state-of-the-art QNP planner: QNP2FOND (Bonet and Gefenner 2020) and two numeric planners: Metric-FF (Hoffmann 2003) and ENHSP (Scala, Hashlam, 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 Gefenner 2009; Bonet and Gefenner 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.
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 LIA²-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 LINP problem $\langle D, I, G \rangle$, we generate a LINP problem $\langle D, s_T, G \rangle$ where $s_T$ is an initial state s.t. if the numeric variable $v$ has no maximum under $I$, then $v(s_T) = b$ for every $v \in V$. 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 $\pi : \{(s_0, a_0), \ldots, (s_n, a_n), (s_{n+1})\}$ 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 $\pi' : \{(S_0, a_0), \ldots, (S_k, a_k), (S_{k+1})\}$ 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 $\pi'$ 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 $[a_j, \ldots, a_{k-1}]$ 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
Bonet, Palacios, and Geffner’s method are user-provided. automatically synthesizes conditions but observations in and loop structures in a planning problem. Our approach essentially corresponds to conditions of branch structures allows for policy generalization. Observations in an FSC 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, Kimber, 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, 2020B090930005, 2020B1212030003, 2020ZDZX3013, 2019A050510024 and 2019B1515120010), the Special Creative University Project for Guangdong (Nos. 2019KTSCX010 and 2018KTSCX016), the Science and Technology Planning Project of Guanzhou (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**


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


