Learning Context-Sensitive Grammars with Classical Planning

Diego Aineto and **Sergio Jiménez** and **Eva Onaindia**
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València.
Camino de Vera s/n. 46022 Valencia, Spain
{dieaigar,serjice,onaindia}@dsic.upv.es

Abstract

This paper presents a novel approach for learning *Context-Sensitive Grammars* (CSGs) from small sets of input strings. The approach guarantees that the computed grammar generalizes provided that the input strings are long and diverse enough to cover all the grammar rules. Our approach is to compile this learning task into a classical planning problem whose solutions are all sequences of actions with a prefix that build the CSG and and a postfix that validates the built grammar in the input strings. The compilation is flexible to implement the three canonical tasks of CSGs, *grammar generation*, *string production* and *string recognition* within the same classical planning model. The experimental validation of our compilation approach reports the empirical data collected when learning the CSGs for the languages $\{a^n b^n c^n : n \geq 1\}$ and $\{a^{2^n} : n \geq 1\}$ and shows that overcomes the performance of a previous compilation for learning *Context-Free Grammars*.

Introduction

A *formal grammar* is a set of symbols and production rules that describe how to form the possible strings of certain formal language (Hopcroft, Motwani, and Ullman 2001). Usually three canonical tasks are defined over formal grammars:

- *Learning*: Given a set of strings, compute a grammar that is compliant with the input strings.
- *Production*: Given a formal grammar, generate strings that belong to the language represented by the grammar.
- *Recognition* (also known as *parsing*): Given a formal grammar and a string, determine whether the string belongs to the language represented by the grammar.

Chomsky defined four types of formal grammars that differ in the form and generative capacity of their rules (Chomsky 2002). Each grammar type generates a different class of formal language that is recognizable with a different kind of automaton: **Type-0** corresponds to the *recursively enumerable* languages that can be recognized with a *Turing machine*. **Type-1** corresponds to the *recursively enumerable* languages that can be recognized with a *Linear-bounded Turing machine*. **Type-2** corresponds to *context-free* languages that are recognizable with a *non-deterministic pushdown automaton*. Finally, **type-3** corresponds to *regular* languages that can be recognized with a *Finite state machine*.

0. $S \rightarrow aBC$
1. $S \rightarrow aSBC$
2. $CB \rightarrow CZ$
3. $CZ \rightarrow WZ$
4. $WZ \rightarrow WC$
5. $WC \rightarrow BC$
6. $aB \rightarrow ab$
7. $bB \rightarrow bb$
8. $bC \rightarrow bc$
9. $cC \rightarrow cc$

(a)

1. S
1. aSBC
0. aaSBCBC
2. aaaBCBCBC
3. aaaBCZCBC
4. aaaBWZCBC
5. aaaBWCCBC
2. aaaBBCCBC
3. aaaBBCCZC
4. aaaBBCWZC
5. aaaBBCWCC
2. aaaBBCBCC
3. aaaBBCZCC
4. aaaBBWZCC
5. aaaBBWCCC
6. aaaBBBCCC
7. aaabBBCCC
7. aaabbBCCC
8. aaabbbCCC
9. aaabbbcCC
9. aaabbbccC
⊥. aaabbbccc

(b)

Figure 1: (a) Example of a context-sensitive grammar; (b) the corresponding *parse tree* for the string $aaabbbccc$.

Figure 1(a) shows an example of a CSG grammar (*Type-1*) that generates the $\{a^n b^n c^n : n \geq 1\}$ language. The grammar defines ten production rules, contains three *terminal symbols* (a, b and c) and five *non-terminal symbols* (S,B,C,W and Z). This CSG can generate, for instance, the string $aaabbbccc$ by applying the sequence of rules $\langle 1, 1, 0, 2, 3, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 9 \rangle$. The *parse tree* in Figure 1(b) exemplifies this rule application and proves that the string $aaabbbccc$ belongs to the $\{a^n b^n c^n : n \geq 1\}$ language defined by the CSG grammar.

Previous work showed that a classical planning compilation can implement the three canonical tasks (namely *grammar generation*, *string production* and *string recognition*) for *Type-3* and *Type-2* grammars (Segovia-Aguas, Jiménez, and Jonsson 2017). In this work we show that a novel classical planning compilation implements these three canonical tasks for grammars of the three last three types (*Type-3*, *Type-2* and *Type-1*). In addition, this novel compilation has fewer input parameters since it does not require to specify

any bound on the size of the grammar rules. The experimental validation of our compilation approach reports the empirical data collected when learning the CSGs for the languages $\{a^n b^n c^n : n \geq 1\}$ and $\{a^{2^n} : n \geq 1\}$ and shows that overcomes the performance of a previous compilation for learning *Context-Free Grammars*.

Background

This section defines the formalization of CSGs and the classical planning model that we follow in this work.

Classical planning

Our approach for learning CSGs is compiling this inductive learning task into a classical planning task.

We use F to denote the set of *fluents* (propositional variables) describing a state. A *literal* l is a valuation of a fluent $f \in F$; i.e. either $l = f$ or $l = \neg f$. A set of literals L represents a partial assignment of values to fluents (without loss of generality, we will assume that L does not contain conflicting values). We use $\mathcal{L}(F)$ to denote the set of all literal sets on F; i.e. all partial assignments of values to fluents.

A *state* s is a full assignment of values to fluents; $|s| = |F|$, so the size of the state space is $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions but often we will abuse of notation by defining a state s only in terms of the fluents that are true in s, as it is common in STRIPS planning.

A *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions. An action $a \in A$ is defined with *preconditions*, $\mathsf{pre}(a) \subseteq \mathcal{L}(F)$, *positive effects*, $\mathsf{eff}^+(a) \subseteq \mathcal{L}(F)$, and *negative effects* $\mathsf{eff}^-(a) \subseteq \mathcal{L}(F)$. We say that an action $a \in A$ is *applicable* in a state s iff $\mathsf{pre}(a) \subseteq s$. The result of applying a in s is the *successor state* denoted by $\theta(s, a) = \{s \setminus \mathsf{eff}^-(a)) \cup \mathsf{eff}^+(a)\}$.

The result of applying action a in state s is the *successor* state $\theta(s, a) = \{s \setminus \mathsf{eff}_c^-(s, a)) \cup \mathsf{eff}_c^+(s, a)\}$ where $\mathsf{eff}_c^-(s, a) \subseteq triggered(s, a)$ and $\mathsf{eff}_c^+(s, a) \subseteq triggered(s, a)$ are, respectively, the triggered *negative* and *positive* effects.

A *classical planning problem* is a tuple $P = \langle F, A, I, G \rangle$, where I is an initial state and $G \subseteq \mathcal{L}(F)$ is a goal condition. A *plan* for P is an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ that induces the *state trajectory* $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$ and a_i $(1 \leq i \leq n)$ is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The *plan length* is denoted with $|\pi| = n$. A plan π *solves* P iff $G \subseteq s_n$; i.e. if the goal condition is satisfied in the last state resulting from the application of the plan π in the initial state I.

Context-Sensitive Grammars

We define a CSGs as a tuple $\mathcal{G} = \langle V, \Sigma, R \rangle$, where:

- V is the finite set of non-terminal symbols, also called variables. With $v_0 \in V$ is the start non-terminal symbol that represents the whole grammar.
- Σ is the finite set of terminal symbols, which are disjoint from the set of non-terminal symbols, i.e. $V \cap \Sigma \neq \emptyset$. The set of terminal symbols is the alphabet of the language defined by $\mathcal{G}$
- Rules in R are defined as $\alpha v \beta \rightarrow \alpha \gamma \beta$ where $v \in V$, $\alpha, \beta \in (V \cup \Sigma)^*$ and $\gamma \in (V \cup \Sigma)^+$. That is, the left-hand and right-hand sides of any production rules $r \in R$ may be surrounded by a context of terminal and nonterminal symbols.

For any two strings (e_1, e_2) we say that e_1 *directly yields* e_2, denoted by $e_1 \Rightarrow e_2$, iff e_1 can be written as $e_3 \alpha A \beta e_4$ and $e2$ can be written as $e_3 \alpha \gamma \beta e_4$, for some production rule $(\alpha A \beta \rightarrow \alpha \gamma \beta) \in R$, and some context strings $e_3, e_4 \in (V \cup \Sigma)^*$. Furthermore we say e_1 *yields* e_2, denoted by $e_1 \Rightarrow^* e_2$, iff $\exists k \geq 0$ and $\exists u_1, \ldots, u_k$ such that $e_1 = u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k = e_2$ for some $k \geq 1$ and some strings $u_1, \ldots, u_{k-1} \in (V \cup \Sigma)^*$. In other words, the relation $\Rightarrow^*$ is the reflexive transitive closure of the $\Rightarrow$ relation.

For instance, Figure 1(b) shows how the string S yields the string *aaabbbccc*. The language of a CSG, $L(\mathcal{G}) = \{e \in \Sigma^* : v_0 \Rightarrow^* e\}$, is the set of strings that contain only terminal symbols and that can be yielded from the string that contains only the initial non-terminal symbol v_0 (which is denoted by S in the example of Figure 1(b)).

Given a CSG $\mathcal{G}$ and a string $e \in L(\mathcal{G})$ that belongs to its language, we define a *parse tree* $t_{\mathcal{G},e}$ as an ordered, rooted tree that determines a concrete syntactic structure of e according to the rules in $\mathcal{G}$:

- Each node in a parse tree $t_{\mathcal{G},e}$ is either:
  - An *internal node* that corresponds to the application of a rule $r \in R$.
  - A *leaf node* that corresponds to a terminal symbol $\sigma \in \Sigma$ and has no outgoing branches.
- Edges in a parse tree $t_{\mathcal{G},e}$ connect non-terminal symbols to terminal or non-terminal symbols following the rules R in $\mathcal{G}$.

Linear-bounded Turing Machines

We define a *Turing machine* as a tuple $M = \langle Q, q_o, Q_\perp, \mathcal{T}, \square, \Sigma, \delta \rangle$, where:

- Q, is a finite and non-empty set of machine states such that $q_0 \in Q$ it the initial state of the machine and $Q_\perp \subseteq Q$ is the subset of terminal states.
- $\mathcal{T}$ is the *tape alphabet*, that is a finite an non-empty set of symbols that includes the *blank symbol* $\square \in \mathcal{T}$ (the only symbol allowed to occur on the tape infinitely often at any step during the computation).
- Σ is the *input alphabet*, the set of symbols allowed to initially appear in the tape.
- $\delta : (Q \setminus Q_\perp) \times \mathcal{T} \rightarrow Q \times \mathcal{T} \times \{left, right\}$ is the *transition function*. If δ is not defined for the current state of the machine and the current tape symbol, then the machine halts.

A table is the most common convention to represent the transitions defined by δ, where the table rows are indexed with the current tape symbol, while the table columns are indexed by the current machine state. For each possible tape

	q_0	q_1	q_2	q_3	q_4	$\underline{q_5}$
a	x,r,q_1	a,r,q_1	-	a,l,q_3	-	-
b	-	y,r,q_2	b,r,q_2	b,l,q_3	-	-
c	-	-	z,l,q_3	-	-	-
x	-	-	-	x,r,q_0	-	-
y	y,r,q_4	y,r,q_1	-	y,l,q_3	y,r,q_4	-
z	-	-	z,r,q_2	z,l,q_3	z,r,q_4	-
$\square$	-	-	-	-	$\square$,r,q_5	-

Figure 2: Example of a seven-symbol six-state *Turing Machine* for recognizing the $\{a^n b^n c^n : n \geq 1\}$ language ($\underline{q_5}$ is the acceptor state).

symbol and state of the machine there is a table entry that defines: (1) the tape symbol to print at the current position of the header (2) whether the header is shifted *left* or *right* after the print operation and (3), the new state of the machine after the print operation. For instance, Figure 2 shows the table that represents the δ function of a *Turing Machine* for recognizing the $\{a^n b^n c^n : n \geq 1\}$ language. In this example the tape alphabet is $\Sigma = \{a, b, c, x, y, z, \square\}$ while the possible machine states are $Q = \{q_0, q_1, q_2, q_3, q_4, \underline{q_5}\}$ where $\underline{q_5}$ is the acceptor state.

A *Turing machine* is an abstract model of computation that operates on an infinite memory tape. An approach to implement in practice a Turing machine, is to bound the size of its tape. This can be done by:

1. Extending the input alphabet with two special symbols, serving as *left* and *right endmarkers*.
2. Limiting computation to the portion of the tape containing the input plus the two tape cells holding the endmarkers. This means that transitions may not print other symbols over the endmarkers and that may neither move to the left of the left endmarker nor to the right of the right endmarker.

Learning Context-Sensitive Grammars

Our approach to learning CSGs is to leverage on classical planning to learn an STRIPS action model s.t. each action schema in the model encodes a transition of a *Linear-bounded Turing Machine* that recognices the strings given as input.

First we show how an entry in the table that defines the transitions of a *Linear-bounded Turing Machine* can be modeled as STRIPS action schema. Next we show how classical planning can be effectively used to learn action models of this kind.

Modeling a Linear-bounded Turing Machine with Classical Planning

Given δ, the *transition function* of a *Linear-bounded Turing Machines* M, it can be encoded as a classical planning frame $\Phi = \langle F, A \rangle$ as follows.

We assume that fluents F are instantiated from a set of *predicates* Ψ, as in PDDL (Fox and Long 2003). Each predicate $p \in \Psi$ has an argument list of arity $ar(p)$. Given a set of *objects* Ω that represent the cells in the tape of the given Turing Machine M, the set of fluents F is induced by assigning objects in Ω to the arguments of the predicates in Ψ; i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$, where Ω^k is the k-th Cartesian power of Ω. In more detail there are four predicates in Ψ:

- `(head ?x)` that encodes the current position of the header in the tape.
- `(next ?x1 ?x2)` that encodes that the cell `?x2` follows cell `?x1` in the tape.
- `(symbol-`σ `?x)` that encodes that the tape cell `?x` contains the symbol $\sigma \in \Sigma$.
- `(state-`q`)` that encodes that $q \in Q$ is the current machine state.

Likewise we assume that actions $a \in A$ are instantiated from STRIPS operator schema. For each transition in δ, a STRIPS action model is defined such that:

- The **header** of the schema is `rule-id(?xl ?x ?xr)` where id uniquely identifies the transition in δ and the parameters $?xl$, $?x$ and $?xr$ are tape cells.
- The **precoditions** of the schema includes `(head ?x)` and `(next ?xl ?x)` `(next ?x ?xr)` to force that $?x$ is the current tape cell pointed by the header and that $?xl$ and $?xr$ respectively are its left and right neighbours. Additionally the schema includes preconditions `(symbol-`σ `?x)` and `(state-`q`)` to capture the current symbol pointed by the header and the currrent machine state.
- The **delete effects** remove the current symbol pointed by the header and the currrent machine state. Finally, the **positive effects** set the new symbol pointed by the header and the new machine state

To illustrate this, Figure 3 shows the rule $a, q_0 \rightarrow x, r, q_1$ of the Turing Machine defined in Figure 2. The full encoding of the Turing Machine defined in Figure 2 produces a total of sixteen STRIPS action schema with the same structure as the one of Figure 3.

```
(:action rule1 ;; a,q_0 -> x,r,q_1
  :parameters (?xl ?x ?xr)
  :precondition (and (head ?x)
                     (next ?xl ?x) (next ?x ?xr)
                     (symbol-a ?x) (state-q_0))
  :effect (and (not (head ?x))
               (not (symbol-a ?x))
               (not (state-q_0))
               (symbol-x ?x) (head ?xr) (state-q_1)))
```

Figure 3: STRIPS action schema that models the first rule $a, q_0 \rightarrow x, r, q_1$ of the Turing Machine defined in Figure 2.

With this regard, the execution of a *Linear-bounded Turing Machine* can then be defined as a *plan trace* $\mathcal{T} = \langle s_0, a_1, s_1, \ldots, a_n, s_n \rangle$ such that s_0 encodes the initial state of the tape as well as the initial machine state and for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$.

Learning STRIPS Models with Classical Planning

Here we show that the existing classical planning compilation for learning STRIPS action schemes from plan traces (Aineto, Jiménez, and Onaindia 2018) can be adapted to address the learning of CSGs. Note that in this case both the preconditions and negative effects of the STRIPS action schemes are already known so the learning task reduces to learning positive effects for these same action schemes.

We formalize the learning of STRIPS action schemes from plan traces as a tuple $\Lambda = \langle \mathcal{M}, \mathcal{T}, \Psi \rangle$ where:

- $\mathcal{M}$, the set of *partially specified* operator schemas.
- $\mathcal{T} = \langle s_0, a_1, s_1, \ldots, a_n, s_n \rangle$ is a *plan trace* obtained watching the execution of the classical plan $\pi = \langle a_1, \ldots, a_n \rangle$ such that, for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$.
- Ψ is the set of predicates that define the abstract state space of a given planning frame.

A solution to a $\Lambda = \langle \mathcal{M}, \mathcal{T}, \Psi \rangle$ learning task is a set of operator schema $\mathcal{M}'$ that is compliant with the input model $\mathcal{M}$, the plan trace $\mathcal{T}$ and the predicates Ψ.

Given a learning task $\Lambda = \langle \mathcal{M}, \mathcal{T}, \Psi \rangle$ the compilation defined by Aineto, Jiménez, and Onaindia outputs a classical planning task $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$:

- F_Λ contains:
  - The set of fluents F built instantiating the predicates Ψ with the objects Ω that appear in the plan trace. Formally, $\Omega = \bigcup_{s \in \mathcal{O}} obj(s)$, where obj is a function that returns the objects that appear in a given state.
  - Fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_v(\xi)$, that represent the programmed action model. If a fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ holds, it means that f is a precondition/negative/positive effect in the schema $\xi \in \mathcal{M}'$.
  - The fluents $mode_{prog}$ and $mode_{val}$ to indicate whether the operator schemas are programmed or validated, and the fluents $\{test_i\}_{1 \leq i \leq n}$, indicating the state where the learned action model is validated.
- I_Λ encodes the first observation, $s_0 \subseteq F$, the fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$ that encode the *partially specified* operator schemas and sets $mode_{prog}$ to true.
- $G_\Lambda = \bigcup_{1 \leq i \leq n} \{test_i\}$, requires that the programmed action model is validated in all de states of the input trace.
- A_Λ comprises three kinds of actions:
  1. Actions for *programming* operator schema $\xi \in \mathcal{M}$. For our purpose we only need to define the actions that add a *positive* effect $f \in F_v(\xi)$ to the action schema $\xi \in \mathcal{M}$.

$$\begin{aligned} \mathsf{pre}(\mathsf{programEff}_{\mathsf{f},\xi}) &= \{\neg del_f(\xi), \neg add_f(\xi), mode_{prog}\}, \\ \mathsf{cond}(\mathsf{programEff}_{\mathsf{f},\xi}) &= \{pre_f(\xi)\} \rhd \{del_f(\xi)\}, \\ &\quad \{\neg pre_f(\xi)\} \rhd \{add_f(\xi)\}. \end{aligned}$$

  2. Actions for *applying* a programmed operator schema $\xi \in \mathcal{M}$ bound with objects $\omega \subseteq \Omega^{ar(\xi)}$. Since operators headers are given as input, the variables $pars(\xi)$ are bound to the objects in ω that appear at the same position.

$$\begin{aligned} \mathsf{pre}(\mathsf{apply}_{\xi,\omega}) &= \{pre_f(\xi) \implies p(\omega)\}_{\forall p \in \Psi, f = p(pars(\xi))} \\ &\quad \cup \{\neg mode_{val}\}, \\ \mathsf{cond}(\mathsf{apply}_{\xi,\omega}) &= \{del_f(\xi)\} \rhd \{\neg p(\omega)\}_{\forall p \in \Psi, f = p(pars(\xi))}, \\ &\quad \{add_f(\xi)\} \rhd \{p(\omega)\}_{\forall p \in \Psi, f = p(pars(\xi))}, \\ &\quad \{mode_{prog}\} \rhd \{\neg mode_{prog}\}, \\ &\quad \{\emptyset\} \rhd \{mode_{val}\}. \end{aligned}$$

  3. Actions for *validating* the partially observed state $s_i \in \mathcal{T}$, $1 \leq i < n$.

$$\begin{aligned} \mathsf{pre}(\mathsf{validate}_\mathsf{i}) &= s_i \cup \{test_j\}_{j \in 1 \leq j < i} \cup \{\neg test_j\}_{j \in i \leq j \leq |\mathcal{O}|} \\ &\quad \cup \{mode_{val}\}, \\ \mathsf{cond}(\mathsf{validate}_\mathsf{i}) &= \{\emptyset\} \rhd \{test_i, \neg mode_{val}\}. \end{aligned}$$

Learning CSGs with Classical Planning

Now we show that the learning of CSGs is a particular case of the $\Lambda = \langle \mathcal{M}, \mathcal{T}, \Psi \rangle$ such that:

- $\mathcal{M}$ is the set of *partially specified* operator schemas that encodes the rules of a *Linear-bounded Turing Machine* M.
- $\mathcal{T}$ represents the execution of the *Linear-bounded Turing Machine* M on a given string.
- $\Psi = \{$`(head ?x)`, `(next ?x1 ?x2)`, `(symbol-`σ `?x)`, `(state-q)`$\}$.

When learning a CSGs the *plan trace* $\mathcal{T}$ is partially observable:

1. The states s_i with $1 \leq i \leq n$ do not contain the fluents $(state - q)$.
2. The exact executed actions a_i are unknown while it is known that shoud be an action built instantiaing the schema corresponding to the transition rule $\sigma, q \rightarrow \sigma', r, l, q'$ provided that σ is the symbol at the cell of the tape that is currently pointed by the header at state s_i.

To handle the first source of partial observability the compilation does not require any modification because s_0 is fully observable since initially the machine state is always q_0 (likewise the machine state is always q_n at the last state that corresponds to a string accepted by the learned automaton). The second source of partial observability can be addressed by adding extra preconditions to the apply actions so only are appliable actions built instantiaing the schema corresponding to the transition rule $\sigma, q \rightarrow \sigma', r, l, q'$ provided that σ is the symbol at the cell of the tape that is currently pointed by the header at state s_i.

Lemma 1. *Soundness. Any classical plan π that solves P_Λ induces an action model $\mathcal{M}'$ that solves $\Lambda = \langle \mathcal{M}, \mathcal{T}, \Psi \rangle$.*

Proof sketch. Once operator schemas $\mathcal{M}'$ are programmed, they can only be applied and validated, because of the $mode_{prog}$ fluent. In addition, P_Λ is only solvable if fluents $\{test_i\}$, $1 \leq i \leq n$ hold at the last reached state. These goals can only be achieved executing an applicable sequence of programmed operator schemas that reaches every state $s_i \in \mathcal{T}$, starting from the corresponding initial

state and following the sequence of actions defined by the plans in Π. This means that the programmed action model $\mathcal{M}'$ complies with the provided input knowledge and hence, solves Λ. □

Lemma 2. *Completeness. Any* STRIPS *action model $\mathcal{M}'$ that solves a $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{T} \rangle$ learning task, is computable solving the corresponding classical planning task P_Λ.*

Proof sketch. By definition, $F_v(\xi) \subseteq F_\Lambda$ fully captures the full set of elements that can appear in a STRIPS action schema $\xi \in \mathcal{M}$ given its header and the set of predicates Ψ. The compilation does not discard any possible STRIPS action schema definable within F_v that satisfies the state trajectory constraint given by $\mathcal{T}$. □

The size of the classical planning task P_Λ output by the compilation depends on:

- The arity of the actions headers in $\mathcal{M}$ and the predicates Ψ that are given as input to the Λ learning task. The larger these numbers, the larger the size of the $F_v(\xi)$ sets. This is the term that dominates the compilation size because it defines the $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ fluents set and the corresponding set of *programming* actions.
- The number of given plan trace. The larger $|\mathcal{T}|$, the more $test_i$ fluents and $\mathsf{validate}_\mathsf{i}$ actions in P_Λ.

Parsing and Production of CSGs with classical planning

The task of parsing can be implemented with the planning model introduced in the previous section by defining a classical planning instace whose fluents and actions encode the transition of the corresponding Turing machine, the initial state encodes the tape with the string to recognize and the goal conditions require that the turing machine ends in on its acceptor states.

The task of production requires to include actions that only write once a single symbol from the *tape alphabet* at the tape location indicated by the header.

Evaluation

Related work

The learning of CFGs can also be understood in terms of activity recognition, such that the library of activities is formalized as a CFG, the library is initially unknown, and the input strings encode observations of the activities to recognize. *Activity recognition* is traditionally considered independent of the research done on automated planning, using handcrafted libraries of activities and specific algorithms (Ravi et al. 2005). An exception is the work by Ramírez and Geffner [2009; 2010] where goal recognition is formulated and solved with planning. As far as we know our work is the first that tightly integrates the tasks of (1) grammar learning, (2) recognition and (3) production using a common planning model and an off-the-shelf classical planner.

Hierarchical Task Networks (HTNs) is a powerful formalism for representing libraries of plans (Nau et al. 2003). HTNs are also defined at several levels such that the tasks at one level are decomposed into other tasks at lower levels with HTN decomposition methods sharing similarities with production rules in CFGs. There is previous work in generating HTNs (Hogg, Munoz-Avila, and Kuter 2008; Lotinac and Jonsson 2016) and an interesting research direction is to extend our approach for computing HTNs from flat sequences of actions. This aim is related to Inductive Logic Programming (ILP) (Muggleton 1999) that learns logic programs from examples. Unlike logic programs (or HTNs) the CFGs that we generate are propositional and do not include variables. Techniques for learning high level state features that include variables are promising for learning lifted grammars (Lotinac et al. 2016).

Conclusions

There is exhaustive previous work on learning CFGs given a corpus of correctly parsed input strings (Sakakibara 1992; Langley and Stromsten 2000) or using positive and negative examples (De la Higuera 2010; Muggleton et al. 2014). This work addresses generating CFGs using only a small set of positive examples (in some cases even one single string that belongs to the language). Furthermore we follow a compilation approach that benefits straightforwardly from research advances in classical planning and that is also suitable for *production* and *recognition* tasks with arbitrary CFGs.

Our compilation bounds the number of rules m, the length of these rules n, the size of the stack ℓ and the length of the input strings z. If these bounds are too small, the classical planner used to solve the output planning task will not be able to find a solution. Larger values for these bounds do not formally affect to our approach, but in practice, the performance of classical planners is sensitive to the size of the input. Interestingly our approach can also follow an incremental strategy where we generate the CFG for a given sublanguage and then encode this *sub-grammar* as an auxiliary procedure for generating more challenging CFGs (Segovia-Aguas, Jiménez, and Jonsson 2016).

The size of the compilation output also depends on the number of examples. Empirical results show that our approach is able to generate non-trivial CFGs from very small data sets. Another interesting extension would be to add negative input strings, which would require a mechanism for validating that a given CFG does *not* generate a given string, or to accept incomplete input strings that would require combining the generation and production mechanisms.

References

Aineto, D.; Jiménez, S.; and Onaindia, E. 2018. Learning strips action models with classical planning. In *ICAPS*.

Chomsky, N. 2002. *Syntactic structures*. Walter de Gruyter.

De la Higuera, C. 2010. *Grammatical inference: learning automata and grammars*. Cambridge University Press.

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

Hogg, C.; Munoz-Avila, H.; and Kuter, U. 2008. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *AAAI*, 950–956.

Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32(1):60–65.

Langley, P., and Stromsten, S. 2000. Learning context-free grammars with a simplicity bias. In *European Conference on Machine Learning*, 220–228. Springer.

Lotinac, D., and Jonsson, A. 2016. Constructing Hierarchical Task Models Using Invariance Analysis. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI'16)*.

Lotinac, D.; Segovia, J.; Jiménez, S.; and Jonsson, A. 2016. Automatic generation of high-level state features for generalized planning. In *IJCAI*.

Muggleton, S. H.; Lin, D.; Pahlavi, N.; and Tamaddoni-Nezhad, A. 2014. Meta-interpretive learning: application to grammatical inference. *Machine learning* 94(1):25–49.

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

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. Shop2: An htn planning system. *J. Artif. Intell. Res.(JAIR)* 20:379–404.

Ramírez, M., and Geffner, H. 2009. Plan recognition as planning. In *IJCAI*, 1778–1783.

Ramırez, M., and Geffner, H. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the Conference of the Association for the Advancement of Artificial Intelligence (AAAI 2010)*, 1121–1126.

Ravi, N.; Dandekar, N.; Mysore, P.; and Littman, M. L. 2005. Activity recognition from accelerometer data. In *AAAI*, volume 5, 1541–1546.

Sakakibara, Y. 1992. Efficient learning of context-free grammars from positive structural examples. *Information and Computation* 97(1):23–60.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016. Hierarchical finite state controllers for generalized planning. In *IJCAI*.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2017. Generating context-free grammars using classical planning. In *IJCAI*.