One-shot Learning of 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 a single input string. Our approach is to compile this learning task into a classical planning problem whose solutions are sequences of actions whose prefixes build the CSG and whose post-fixes validate that the built grammar is compliant with the input string. The compilation is flexible to implement the three canonical tasks for CSGs, *grammar generation*, *string production* and *string recognition* within the same classical planning model. Our experimental evaluation shows that the learned CSGs achieve generalization provided that the input strings are long and diverse enough to cover all the grammar rules.

Introduction

A *formal grammar* is a set of symbols and production rules that describe how to form the possible strings of certain formal language. 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* automaton. **Type-1** corresponds to the *recursively enumerable* languages that can be recognized with a *Turing machine* automaton. **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 automaton*.

Figure 1(a) shows an example of a CSG grammar that generates the $\{a^n b^n c^n : n \geq 1\}$ language (*Type-1*). The grammar defines ten production rules, contains three terminal symbols (a, b and c) and five non-terminal symbol (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 *generation chain* 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.

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)

```
S
aSBC
aaSBCBC
aaaBCBCBC
aaaBCZCBC
aaaBWZCBC
aaaBWCCBC
aaaBBCCBC
aaaBBCCZC
aaaBBCWZC
aaaBBCWCC
aaaBBCBCC
aaaBBCZCC
aaaBBWZCC
aaaBBWCCC
aaaBBBCCC
aaabBBCCC
aaabbBCCC
aaabbbCCC
aaabbbcCC
aaabbbccC
aaabbbccc
```

(b)

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

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 number of non-terminal symbols or in the size of the grammar rules. The reported empirical results show that the learned CSGs achive generalization provided that the input strings *complete*, meaning that they are long and diverse enough to cover all the grammar rules.

Background

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

Classical planning

Our approach for the one-shot learning of 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

A *context-sensitive grammar* (CSGs) is a formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of terminal and nonterminal symbols. CSGs are more general than *context-free grammars*, in the sense that there are languages that can be described by CSG but not by context-free grammars. On the other hand CSGs are less general than unrestricted 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 $\mathcal{G}$
- $R : V \rightarrow (V \cup \Sigma)^*$ is the finite set of production rules in the grammar. By definition rules $r \in R$ always contain a single non-terminal symbol on the left-hand side.

```
(:action update-rule1 ;;; 0,A/1,R,B
  :parameters (?xl ?x ?xr)
  :precondition (and (head ?x)
                     (next ?xl ?x) (next ?x ?xr)
                     (symbol0 ?x) (stateA))
  :effect (and (not (head ?x))
               (not (symbol0 ?x))
               (not (stateA))
               (symbol1 ?x) (head ?xr) (stateB)))
```

Figure 2: .

Learning Context-Sensitive Grammars

Parsing and Production of CSGs with classical planning

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

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

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

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

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*.