Categorial Dependency Grammars with Iterated Sequences

Denis Béchet[1] and Annie Foret[2]

[1] LINA UMR CNRS 6241, Université de Nantes, France
Denis.Bechet@univ-nantes.fr
[2] IRISA, Université de Rennes 1, France
Annie.Foret@irisa.fr

Abstract. Some dependency treebanks use special sequences of dependencies where main arguments are mixed with separators. Classical Categorial Dependency Grammars (CDG) do not allow this construction because iterative dependency types only introduce the iterations of the same dependency. An extension of CDG is defined here that introduces a new construction for repeatable sequences of one or several dependency names. The learnability properties of the extended CDG when grammars are infered from a dependency treebank is also studied. It leads to the definition of new classes of grammars that are learnable in the limit from dependency structures.

Keywords: Categorial grammar · Dependency grammar · Iterated dependencies · Computational linguistics · Dependency Treebanks · Grammatical inference · Incremental learning

1 Introduction

Dependency grammars and dependency treebanks do not always use a unique linguistic model for lists of elements. Some of them define an enumeration as a linked list of elements. Other grammars define a list as a set of dependencies that link the same word, the head of the list, to the elements of the list.

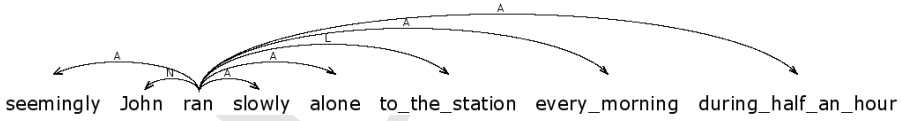


Fig. 1. A dependency structure with five dependencies A

Categorial Dependency Grammars [5] (CDG) allow the second model with iterated dependency types. This construction introduces a list of dependencies with the same name and the same governor. The dependency structures

(DS) in Figure 1 shows a dependency A that is iterated on the left and on the right five times. A CDG compatible with the example could assign the type $[N\backslash A\backslash S/A^*/L/A^*]$ to the word *ran*. The dependency name A appears three times, two times as the iterative dependency type A^*. With this type, other DS are also possible: Each A^* may introduce none, one or several arguments linked to *ran* by a dependency A.

However, iterated dependency types cannot be used when a list of elements needs to be mixed with a separator like the example of Figure 2 from corpus Sequoia [4] *"Les cyclistes et vététistes peuvent se réunir ce matin, à 9h, place Jacques-Bailleurs, à l'occasion d'une sortie d'entraînement." (fr. the cyclists and ATB bikers may meet themselves this morning, at 9, at Jacques-Bailleurs square, for a training ride)*[3].

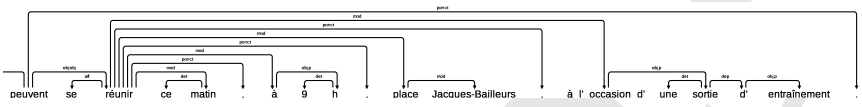


Fig. 2. A dependency structure with a list of modifiers separated by commas

In this example, several modifiers alternate with a punctuation sign. The verb *réunir* may have type $[aff\backslash obj : obj/mod/ponct/mod/ponct/mod/ponct/mod]$. A regular expression for the part that corresponds to the modifiers and commas would be $mod(punct\, mod)^*$ or $(mod\, punct)^* mod$. It is not an iterative choice between *mod* and *ponct* like the regular expression $(mod||ponct)^*$ but a repeatable sequence of *mod* and *ponct*. In order to formalize such structures, we propose to extend CDG types with a new construction that introduces finite sequences of dependencies. The system is an extension of classical CDG because iterated dependency types can be seen as sequence iterations where the sequence has a length of one dependency name.

We also study the learnability properties of CDG with sequence iteration when the grammar has to be infered from a dependency treebank. This concept of *identification in the limit* is due to E.M. Gold [7]. *Learning from strings* refers to hypothetical grammars generated from finite sets of strings. More generally, the hypothetical grammars may be generated from finite sets of structures defined by the target grammar. This kind of learning is called *learning from structures*. Both concepts were intensively studied (see excellent surveys in [2], [9] and [8]). This concept lead for CDG with sequence iterations to a new class of grammar that is learnable from positive examples of dependency structures (DS).

The plan of the paper is as follows. Section 2 introduces Categorial Dependency Grammars with sequence iteration and studies their parsing properties and expressive power. The section also presents the links with linear logic, non-commutative logic and Lambek Calculus. Section 3 studies the learnability properties of such grammars from positive examples of dependency structures and

[3] See `talc2.loria.fr/deep-sequoia/sequoia-7.0/html/annodis.er_00060.html`

defines new classes of such grammars that are learnable in this context. Section 4 presents experimental studies of sequence iterations in existent DS corpora. Section 5 concludes the paper.

2 CDG with Sequence Iterations

2.1 Classical Categorial Dependency Grammars

Categorial dependency grammars can be seen as an assignment to words of first order dependency types of the form: $t = [L_m \backslash \ldots \backslash L_1 \backslash g / R_1 / \ldots / R_n]^P$. Intuitively, $w \mapsto [\alpha \backslash d \backslash \beta]^P$ means that the word w has a left subordinate through dependency d (similar for the right part $[\alpha / d / \beta]^P$). Similarly $w \mapsto [\alpha \backslash d^* \backslash \beta]^P$ means that w may have $0, 1$ or several left subordinates through dependency d. The *head type* g in $w \mapsto [\alpha \backslash g / \beta]^P$ means that w is governed through dependency g. The assignment of Example 1 determines the projective DS in Figure 3.

Example 1.

$$
\begin{array}{ll}
in & \mapsto [c\_copul / prepos{-}l] \\
the & \mapsto [det] \\
beginning & \mapsto [det \backslash prepos{-}l] \\
was & \mapsto [c\_copul \backslash S / @fs / pred] \\
word & \mapsto [det \backslash pred] \\
. & \mapsto [@fs]
\end{array}
$$

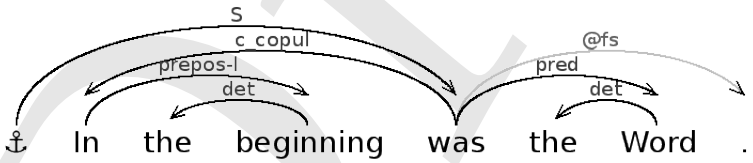


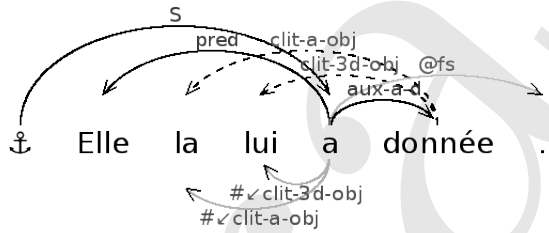
Fig. 3. Projective dependency structure.

The intuitive meaning of part P, called *potential*, is that it defines *discontinuous* dependencies of the word w. P is a string of *polarized valencies*, i.e. of symbols of four kinds: $\swarrow d$ (*left negative valency* d), $\searrow d$ (*right negative valency* d), $\nwarrow d$ (*left positive valency* d), $\nearrow d$ (*right positive valency* d). Intuitively, $v = \nwarrow d$ requires a subordinate through dependency d situated *somewhere* on the left, whereas the *dual* valency $\breve{v} = \swarrow d$ requires a governor through the same dependency d situated *somewhere* on the right. So together they describe the discontinuous dependency d. Similarly for the other pairs of dual valencies. For negative valencies $\swarrow d, \searrow d$ are provided a special kind of types $\#(\swarrow d)$,

$\#(\searrow d)$. Intuitively, they serve to check the adjacency of a distant word subordinate through discontinuous dependency d to a *host word*. The dependencies of these types are called *anchor*. For instance, the assignment of Example 2 determines the non-projective DS in Figure 4.

Example 2.

$$
\begin{array}{ll}
elle & \mapsto [pred] \\
la & \mapsto [\#(\swarrow clit{-}a{-}obj)]^{\swarrow clit\text{–}a\text{–}obj} \\
lui & \mapsto [\#(\swarrow clit{-}3d{-}obj)]^{\swarrow clit\text{–}3d\text{–}obj} \\
a & \mapsto [\#(\swarrow clit{-}3d{-}obj)\backslash\#(\swarrow clit{-}a{-}obj)\backslash pred\backslash S/@fs/aux{-}a{-}d] \\
donnée & \mapsto [aux{-}a{-}d]^{\nwarrow clit\text{–}3d\text{–}obj \nwarrow clit\text{–}a\text{–}obj} \\
. & \mapsto [@fs]
\end{array}
$$



(*fr.* **she* $it_{\mathbf{g}=fem}$ *to him has given*)

Fig. 4. Non-projective dependency structure.

Definition 1 (CDG dependency structures). *Let* $W = a_1 \dots a_n$ *be a list of words and* $\{d_1, \dots, d_m\}$ *be a set of* dependency names*, with their dependency nature that can be either local, discontinuous or anchor. A graph* $D = (W, E)$ *with labeled arcs is a* dependency structure *(*DS*) of* W *if it has a* root*, i.e. a node* $a_i \in W$ *such that* (i) *for any node* $a \in W$*,* $a \neq a_i$*, there is a path from* a_i *to* a *and* (ii) *there is no arc* (a', d, a_i)*.*[4] *An arc* $(a, d, a') \in E$ *is called* dependency *d from a to* a'*. a is called a* governor *of* a' *and* a' *is called a* subordinate *of a through d. The linear order on* W *is the* precedence order *on* D*.*

Definition 2 (CDG types). *Let* $\mathbf{C}$ *be a set of* local dependency names *and* $\mathbf{V}$ *be a set of* valency names*.*

The expressions of the form $\swarrow v$*,* $\nwarrow v$*,* $\searrow v$*,* $\nearrow v$*, where* $v \in \mathbf{V}$*, are called* polarized valencies*.* $\nwarrow v$ *and* $\nearrow v$ *are* positive*,* $\swarrow v$ *and* $\searrow v$ *are* negative*;* $\nwarrow v$ *and* $\swarrow v$ *are* left*,* $\nearrow v$ *and* $\searrow v$ *are* right*. Two polarized valencies with the same valency name and orientation, but with the opposite signs are* dual*.*

[4] Evidently, every DS is connected and has a unique root.

An expression of one of the forms $\#(\swarrow v)$, $\#(\searrow v)$, $v \in \mathbf{V}$, *is called* anchor type *or just* anchor. *An expression of the form* d^* *where* $d \in \mathbf{C}$, *is called* iterated dependency type. *Local dependency names, iterated dependency types and anchor types are* primitive types.

An expression of the form $t = [L_m \backslash \ldots \backslash L_1 \backslash H / R_1 \ldots / R_n]$ *in which* $m, n \geq 0$, $L_1, \ldots, L_m, R_1, \ldots, R_n$ *are primitive types and* H *is either a local dependency name or an anchor type, is called a* basic dependency type. $L_1, \ldots, L_m$ *and* $R_1, \ldots, R_n$ *are respectively left and right* argument types *of* t. H *is called the* head type *of* t.

A (possibly empty) string P *of polarized valencies sorted using the standard lexicographical order* $<_{lex}$ *compatible with the polarity order* $\nwarrow < \searrow < \swarrow < \nearrow$, *is called a* potential. *A* dependency type *is an expression* B^P *in which* B *is a basic dependency type and* P *is a potential.* $\mathbf{CAT}(\mathbf{C}, \mathbf{V})$ *will denote the set of all dependency types over* $\mathbf{C}$ *and* $\mathbf{V}$.

CDG are defined using the following calculus of dependency types.[5] These rules are relativized with respect to the word positions in the sentence, which allows to interpret them as rules of construction of DS. Namely, when a type $B^{v_1 \cdots v_k}$ is assigned to the word in a position i, we encode it using the *state* $(B, i)^{(v_1,i)\cdots(v_k,i)}$. In these rules, types must be adjacent.

Definition 3 (Relativized calculus of dependency types).

L$^{\mathbf{l}}$. $\Gamma_1\,([C], i_1)^{P_1}([C\backslash\beta], i_2)^{P_2}\Gamma_2 \vdash \Gamma_1\,([\beta], i_2)^{P_1P_2}\Gamma_2$

I$^{\mathbf{l}}$. $\Gamma_1\,([C], i_1)^{P_1}([C^*\backslash\beta], i_2)^{P_2}\Gamma_2 \vdash \Gamma_1\,([C^*\backslash\beta], i_2)^{P_1P_2}\Gamma_2$

Ω$^{\mathbf{l}}$. $\Gamma_1\,([C^*\backslash\beta], i)^{P}\Gamma_2 \vdash \Gamma_1\,([\beta], i)^{P}\Gamma_2$

D$^{\mathbf{l}}$. $\Gamma_1\,\alpha^{P_1(\swarrow C, i_1)P(\nwarrow C, i_2)P_2}\Gamma_2 \vdash \Gamma_1\,\alpha^{P_1PP_2}\Gamma_2$,
if the potential $(\swarrow C, i_1)P(\nwarrow C, i_2)$ *satisfies the following pairing rule* **FA** *(*first available*) and where, moreover,* $i_1 < i_2$ *(non-internal constraint).*[6]

FA : *P has no occurrences of* $(\swarrow C, i)$ *or* $(\nwarrow C, i)$, *for any* i

L$^{\mathbf{l}}$ is the classical elimination rule. Eliminating the argument type $C \neq \#(\alpha)$ it constructs the (*projective*) dependency C and concatenates the potentials. $C = \#(\alpha)$ creates *anchor dependencies.* **I**$^{\mathbf{l}}$ derives $k > 0$ instances of C. **Ω**$^{\mathbf{l}}$ serves in particular for the case $k = 0$. **D**$^{\mathbf{l}}$ creates *discontinuous dependencies.* It pairs and eliminates dual valencies with name C satisfying the rule **FA** to create the discontinuous dependency C.

Now, in this relativized calculus, for every proof ρ represented as a sequence of rule applications, we may define the DS $DS_x(\rho)$ *constructed* in this proof. Namely, let us consider the calculus relativized with respect to a sentence x with the set of word occurrences W. Then $DS_x(\varepsilon) = (W, \emptyset)$ is the DS constructed in the empty proof $\rho = \varepsilon$. Now, let (ρ, R) be a nonempty proof with respect to x and $(W, E) = DS_x(\rho)$. Then $DS_x((\rho, R))$ is defined as follows:
If $R = \mathbf{L}^{\mathbf{l}}$ or $R = \mathbf{I}^{\mathbf{l}}$, then $DS_x((\rho, R)) = (W, E \cup \{(a_{i_2}, C, a_{i_1})\})$. When C is a local dependency name, the new dependency is *local.* In the case where C is an

[5] We show left-oriented rules. The right-oriented are symmetrical.

[6] This disallows internal primitive loops (the rule D^l cannot apply to a single word).

anchor, this is an *anchor* dependency.
If $R = \mathbf{\Omega^l}$, then $DS_x((\rho, R)) = DS_x(\rho)$.
If $R = \mathbf{D^l}$, then $DS_x((\rho, R)) = (W, E \cup \{(a_{i_2}, C, a_{i_1})\})$ and the new dependency is *discontinuous*.

Definition 4 (CDG). *A* categorial dependency grammar (CDG) *is a system* $G = (W, \mathbf{C}, \mathbf{V}, S, \lambda)$, *where* W *is a finite set of words,* $\mathbf{C}$ *is a finite set of local dependency names containing the selected name* S *(an axiom),* $\mathbf{V}$ *is a finite set of discontinuous dependency names and* λ, *called* lexicon, *is a finite substitution on* W *such that* $\lambda(a) \subset \mathbf{CAT}(\mathbf{C}, \mathbf{V})$ *for each word* $a \in W$. λ *is extended on sequences of words* W^* *in the usual way.*[7]

For $G = (W, \mathbf{C}, \mathbf{V}, S, \lambda)$, *a DS* D *and a sentence* x, *let* $G[D, x]$ *denote the relation:*

$$D = DS_x(\rho) \quad \begin{array}{l} \textit{where } \rho \textit{ is a proof of } (t_1, 1) \cdots (t_n, n) \vdash (S, j) \\ \textit{for some } n, j,\ 0 < j \leq n \textit{ and } t_1 \cdots t_n \in \lambda(x). \end{array}$$

Then the language *generated by* G *is the set* $L(G) =_{df} \{w \mid \exists D\ G[D, w]\}$ *and the* DS-language *generated by* G *is the set* $\Delta(G) =_{df} \{D \mid \exists w\ G[D, w]\}$. $\mathcal{D}(CDG)$ *and* $\mathcal{L}(CDG)$ *will denote the families of DS-languages and languages generated by these grammars.*

Example 3. The proof in Figure 5 shows that the DS in Figure 4 belongs to the DS-language generated by a grammar containing the type assignments shown above for the French sentence *Elle la lui a donnée* (the word positions are not shown on types).

CDG are very expressive. Evidently, they generate all CF-languages. They can also generate non-CF languages.

Example 4. The following CDG generates the language $\{a^n b^n c^n \mid n > 0\}$ [6]:[8]

$$\begin{array}{l} a \mapsto \#(\swarrow A)^{\swarrow A}, [\#(\swarrow A) \backslash \#(\swarrow A)]^{\swarrow A} \\ b \mapsto [B/C]^{\nwarrow A}, [\#(\swarrow A) \backslash S/C]^{\nwarrow A} \\ c \mapsto [C], [B \backslash C] \end{array}$$

2.2 CDG with Sequences and Sequence Iterations

The extended system introduced here defines sequences and sequence iterations. An extended type $[\alpha \backslash (C_1 \bullet \cdots \bullet C_n) \backslash \beta]^P$ is viewed as a type that contains a sequence of n primitive types. It is equivalent to $[\alpha \backslash C_n \backslash \cdots \backslash C_1 \backslash \beta]^P$ (the sequence appears in the reverse order). The *starred* version of a sequence $[\alpha \backslash (C_1 \bullet \cdots \bullet C_n)^* \backslash \beta]^P$ is handled as a sequence of n primitive types that can be repeated none, once or several times. This construction with $n > 1$ is not possible with classical CDG which allows only iteration of a primitive type (the case $n = 1$). This type is equivalent to an infinite list of types :

$[\alpha \backslash \beta]^P$,

[7] $\lambda(a_1 \cdots a_n) = \{t_1 \ldots t_n \mid t_1 \in \lambda(a_1), \ldots, t_n \in \lambda(a_n)\}$.
[8] One can see that a DS is not always a tree.

$$\dfrac{\dfrac{[pred]}{[\#(\swarrow clit-a-obj)]^{\swarrow clit-a-obj}}\quad \dfrac{[\#(\swarrow clit-3d-obj)]^{\swarrow clit-3d-obj}}{[\#(\swarrow clit-a-obj)\backslash pred\backslash S/aux-a-d]^{\swarrow clit-3d-obj}}\,(\mathbf{L}^l)}{[pred\backslash S/aux-a-d]^{\swarrow clit-a-obj\swarrow clit-3d-obj}}\,(\mathbf{L}^l)$$

$$\dfrac{\dfrac{[S/aux-a-d]^{\swarrow clit-a-obj\swarrow clit-3d-obj}}{[S]^{\swarrow clit-a-obj\swarrow clit-3d-obj\nwarrow clit-3d-obj\nwarrow clit-a-obj}}\,(\mathbf{L}^r)}{S}\,(\mathbf{D}^l\times 2)$$

$$[\#(\swarrow clit-3d-obj)]^{\swarrow clit-3d-obj}\quad [\#(\swarrow clit-3d-obj)\backslash \#(\swarrow clit-a-obj)\backslash pred\backslash S/aux-a-d]\,(\mathbf{L}^l)$$

$$[aux-a-d]^{\nwarrow clit-3d-obj\nwarrow clit-a-obj}$$

Fig. 5. Dependency structure correctness proof.

$[\alpha\backslash(C_1 \bullet \cdots \bullet C_n)\backslash\beta]^P \equiv [\alpha\backslash C_n\backslash\cdots\backslash C_1\backslash\beta]^P$,
$[\alpha\backslash(C_1 \bullet \cdots \bullet C_n \bullet C_1 \bullet \cdots \bullet C_n)\backslash\beta]^P \equiv [\alpha\backslash C_n\backslash\cdots\backslash C_1\backslash C_n\backslash\cdots\backslash C_1\backslash\beta]^P$,
etc.

Definition 5. *We call* `sequence iteration types` *the expressions B^P where P is a potential, $B = [L_m\backslash\cdots\backslash L_1\backslash H/\cdots/R_1\cdots/R_n]$, H is either a local dependency name or an anchor type and $L_m, \ldots L_1$, $R_1 \ldots$, R_n are either anchor types, local dependency names, sequences of local dependency names or sequence iterations of local dependency names (a sequence of one local dependency name is identified to a local dependency name).*

Rules for CDG with sequences and sequence iterations:

L^1. $\Gamma_1\,([C], i_1)^{P_1}([C\backslash\beta], i_2)^{P_2}\Gamma_2 \vdash \Gamma_1\,([\beta], i_2)^{P_1P_2}\Gamma_2$
C^1. $\Gamma_1\,([(\alpha)^*\backslash\beta], i)^P\Gamma_2 \vdash \Gamma_1\,([\alpha\backslash(\alpha)^*\backslash\beta], i)^P\Gamma_2$ $(\alpha)^*$ is a sequence iteration
W^1. $\Gamma_1\,([(\alpha)^*\backslash\beta], i)^P\Gamma_2 \vdash \Gamma_1\,([\beta], i)^P\Gamma_2$ $(\alpha)^*$ is a sequence iteration
S^1. $\Gamma_1\,([(\alpha \bullet C)\backslash\beta], i)^P\Gamma_2 \vdash \Gamma_1\,([C\backslash\alpha\backslash\beta], i)^P\Gamma_2$ $(\alpha \bullet C)$ is a sequence
D^1. $\Gamma_1\,\alpha^{P_1(\swarrow C, i_1)P(\nwarrow C, i_2)P_2}\Gamma_2 \vdash \Gamma_1\,\alpha^{P_1PP_2}\Gamma_2$,
if the potential $(\swarrow C, i_1)P(\nwarrow C, i_2)$ satisfies **FA** and if $i_1 < i_2$

2.3 Links with Noncommutative Logic and Lambek Calculus

From a logical point of view, a CDG type B^P consists of a projective part B and a potential P. B can be seen as a logical formula in a resource sensible logic like linear logic. Because the order of formulas is also important, B can be seen either as a formula in noncommutative logic [1] or a formula in Lambek calculus[10].

In Lambek calculus, a sequence of primitive types is the product of primitive types. In the same perspective, a sequence iteration of primitive types has no equivalent in Lambek calculus.

In noncommutative logic, a type $B = [L_m\backslash\cdots\backslash L_1\backslash H/\cdots/R_1\cdots/R_n]$ can be seen as the linear type $L_m \multimap \cdots \multimap L_1 \multimap H \circ\!\!- R_1 \cdots \circ\!\!- R_n$ where $\multimap$ and $\circ\!\!-$ are the left and right linear implications. The sequence of primitive types $(C_1 \bullet \cdots \bullet C_n)$ is the multiplicative noncommutative product $(C_1 \odot \cdots \odot C_n)$. The following implications are valid in noncommutative logic. They justify the rules for CDG sequences:

$$(C_1 \odot \cdots \odot C_n) \multimap \beta \vdash C_n \multimap \cdots \multimap C_1 \multimap \beta$$

$$C_n \multimap \cdots \multimap C_1 \multimap \beta \vdash (C_1 \odot \cdots \odot C_n) \multimap \beta$$

The sequence iteration of primitive types $(C_1 \bullet \cdots \bullet C_n)^*$ corresponds to $?(C_1 \odot \cdots \odot C_n)$: An iteration is seen as the dual of the exponential of the multiplicative product of the primitive types. The following provable sequents justify the rules for CDG sequence iterations:

$$?(C_1 \odot \cdots \odot C_n) \multimap \beta \vdash (C_1 \odot \cdots \odot C_n \odot ?(C_1 \odot \cdots \odot C_n)) \multimap \beta$$

$$?(C_1 \odot \cdots \odot C_n) \multimap \beta \vdash \beta$$

Thus, it is possible to interpret the projective part of CDG types as a formula of noncommutative logic. The search for a valid analysis of a sentence becomes the proof search in noncommutative logic of a sequent where the formulae are one of the possible lists of types of the words through the lexicon of a grammar. This interpretation gives automatically a compositional semantic interpretation *à la Montague*.

2.4 Parsing and Expressive Power

Sequences can be seen as syntactic sugar for types. Thus, they don't change the parsing properties of languages and the expressive power of grammars. From a formal point of view, sequence iterations do not introduce new languages of string with respect to classical CDG. In fact, it is possible to emulate a sequence iteration by a simple iteration where each dependent corresponds to an element of the sequence (for instance the leftmost element of the sequence) and governs the other elements of the sequence. In contrast, sequence iterations introduce a new construction that is very common on DS corpora. For instance, the treebank Sequoia[4] models a list of elements as the alternative of an element and a punctuation mark. The introduction presents an example where the modifiers of the verb *réunir* alternate with commas: *"Les cyclistes et vététistes peuvent se réunir ce matin, à 9h, place Jacques-Bailleurs, à l'occasion d'une sortie d'entraînement." (fr. the cyclists and ATB bikers may meet themselves this morning, at 9, at Jacques-Bailleurs square, for a training ride).*

The parsing of CDG with sequence iterations is not very different from the parsing of classical CDG (i.e. with iterated dependency type). A sequence iteration at the leftmost position of a type $[(d_1 \bullet \cdots \bullet d_n)^* \backslash L_1 \cdots \backslash H / R_1 / \cdots]^{P_2}$ is rewritten into $[d_{n-1} \backslash \cdots \backslash d_1 \backslash (d_1 \bullet \cdots \bullet d_n)^* \backslash L_1 \cdots \backslash H / R_1 / \cdots]^{P_1 P_2}$ when the type $[d_n]^{P_1}$ is on its left (potentials $P_1 P_2$ may generate non-projective dependencies).

3 Learnability Results

The section studies the learnability properties of CDG with sequence iterations from positive examples of dependency structures (because sequences can be seen as syntactic sugar, the grammar are supposed to contain no sequence). It ends with the definition of a new family of classes of such grammars that are learnable in this context.

3.1 Inference Algorithm

A vicinity corresponds for a word to the part of a type that is used in a DS.

Definition 6 (Vicinity). *Given a DS D, the incoming and outgoing dependencies of a word w can be either local, anchor or discontinuous. For a discontinuous dependency d on a word w, we define its polarity p ($\nwarrow, \searrow, \swarrow, \nearrow$), according to its direction (left, right) and as negative if it is incoming to w, positive otherwise.*

Let D be a DS in which an occurrence of a word w has : the incoming projective dependency or anchor H (or the axiom S), the left projective dependencies or anchors $L_k, \ldots, L_1$ (in this order), the right projective dependencies or anchors $R_1, \ldots, R_m$ (in this order), and the discontinuous dependencies $d_1, \ldots, d_n \in \mathbf{V}$ with their respective polarities $p_1, \ldots, p_n$.

Then the vicinity *of w in D is the type*

$$V(w, D) = [L_1 \backslash \cdots \backslash L_k \backslash H / R_m / \cdots / R_1]^P,$$

in which P is a permutation of $p_1 d_1, \ldots, p_n d_n$ in the standard lexicographical order $<_{lex}$ compatible with the polarity order $\nwarrow < \searrow < \swarrow < \nearrow$.

For instance, *donnée* in Figure 4 has the vicinity $[aux{-}a{-}d]^{\nwarrow clit{-}a{-}obj \nwarrow clit{-}3d{-}obj}$. This vicinity is nearly the same as the type of *donnée* in the lexicon because this type doesn't have a sequence iteration (or an iterated dependency type). The difference comes from the order of the polarized valencies $\nwarrow clit-a-obj$ and $\nwarrow clit{-}3d{-}obj$ that appear in a different order. The vicinity of the verb *réunir* in Figure 2 is $[aff\backslash obj{:}obj/mod/ponct/mod/ponct/mod/ponct/mod]$. A type that is compatible with this vicinity could be $[aff\backslash obj{:}obj/(ponct \bullet mod)^*/mod]$. In this case, the type in the lexicon and the vicinity are different.

Definition 7 (Algorithm).

Figure 6 presents an inference algorithm $\mathbf{TGE}^{(K)}_{J-seq}$ which, for every next DS in a training sequence, transforms the observed local, anchor and discontinuous dependencies of every word into a type with repeated local dependency sequences by introducing a sequence iteration for each group of at least K consecutive identical sequences of local dependencies. J indicates the maximum internal length of the sequences that are transformed into sequence iterations.

Definition 8 (Generalization). *The notation $TGen^{(K)}_{J-seq}(t_w)$, that applies the inner loop algorithm in Figure 6 to a type t_w, is extended to sets of types, lexicons and grammars, in a usual way, such that each assignment $w \mapsto t$ becomes $w \mapsto TGen^{(K)}_{J-seq}(t)$*

Ambiguities. Note that this process may be ambiguous. For instance, for $K = J = 2$, the generalization of $[a\backslash b\backslash a\backslash b\backslash a\backslash b\backslash a\backslash H]$ could be $[(b \bullet a)^*\backslash a\backslash H]$ or $[a\backslash(a \bullet b)^*\backslash H]$. With the same conditions on K and J, the generalization of $[b\backslash a\backslash a\backslash a\backslash a\backslash H]$ could be $[b\backslash a^*\backslash H]$ or $[b\backslash(a \bullet a)^*\backslash a\backslash H]$. There are several ways to overcome this, such as : **[ALL mode]** adds all such types in the internal loop ; or **[LML mode]** adds only the type corresponding to a leftmost longest sequence iteration with the shortest pattern. We could also consider different limiting neighbourhood conditions around the repeating pattern.

Definition 9 (LML mode). *We consider three parameters of the repeated sequence: the start position, the pattern length, the total length. In the [LML mode], the three parameters have the priorities in that order: We consider first the leftmost position as the start position, then the smallest pattern length, then the maximal number of repetitions.*

Algorithm $\mathbf{TGE}^{(K)}_{J-seq}$ (type-generalize-expand):
Input: σ, a training sequence of length N.
Output: CDG $\mathbf{TGE}^{(K)}_{J-seq}(\sigma)$.

let $G_H = (W_H, \mathbf{C}_H, \mathbf{V}_H, S, \lambda_H)$ where $W_H := \emptyset$; $\mathbf{C}_H := \{S\}$; $\mathbf{V}_H := \emptyset$; $\lambda_H := \emptyset$;
(loop) **for** $i = 1$ **to** N // *loop on* σ
  let D **such that** $\sigma[i+1] = \sigma[i] \cdot D$; // *the i-th DS of* σ
  let $(X, E) = D$;
  (loop) **for every** $w \in X$ // *the order of the loop is not important*
    $W_H := W_H \cup \{w\}$;
    let $t_w = V(w, D)$ // *the vicinity of* w *in* D
    $\mathbf{C}_H := \mathbf{C}_H \cup \{\ d \mid d$ is a local dependency name of $t_w\ \}$
    $\mathbf{V}_H := \mathbf{V}_H \cup \{\ d \mid \#(\swarrow d)$ or $\#(\searrow d)$ is an anchor type of $t_w\ \}$
      $\cup \{\ d \mid \nwarrow d, \swarrow d, \nearrow d$ or $\searrow d$ is a polarized valency of $t_w\ \}$
    // *– computing the generalization of* t_w: $TGen^{(K)}_{J-seq}(t_w)$
      $t'_w := t_w$
      (loop) **while** $t'_w = [\alpha\backslash\delta\backslash\cdots\backslash\delta\backslash\beta]^P$
        with at least K consecutive occurrences of $\delta = d_j\backslash\cdots\backslash d_1$ $(j \leq J)$,
        $d_1, \ldots, d_j \in \mathbf{C}_H, COND_{ll}(\alpha, \delta)$ (or α not present) and $COND_{lr}(\beta, \delta)$
          $t'_w := [\alpha\backslash(d_1 \bullet \cdots \bullet d_j)^*\backslash\beta]^P$
      (loop) **while** $t'_w = [\alpha/\delta/\cdots/\delta/\beta]^P$
        with at least K consecutive occurrences of $\delta = d_j/\cdots/d_1$ $(j \leq J)$,
        $d_1, \ldots, d_j \in \mathbf{C}_H, COND_{rl}(\alpha, \delta)$ and $COND_{rr}(\beta, \delta)$ (or β not present)
          $t'_w := [\alpha/(d_1 \bullet \cdots \bullet d_j)^*/\beta]^P$
    // *– the final* t'_w *defines* $TGen^{(K)}_{J-seq}(t_w)$
    $\lambda_H(w) := \lambda_H(w) \cup \{t'_w\}$; // *expansion*
  end end

where $COND_{ll}(\alpha, \delta) = \alpha$ does not end in δ
  $COND_{lr}(\beta, \delta) = \beta$ does not start with $\delta\backslash$
  $COND_{rl}(\alpha, \delta) = \alpha$ does not end in $/\delta$
  $COND_{rr}(\beta, \delta) = \beta$ does not start with δ

Fig. 6. *Inference algorithm* $\mathbf{TGE}^{(K)}_{J-seq}$; the inner loop defines $TGen^{(K)}_{J-seq}(t_w)$ on types.

This mode is detailed by the following examples.
- $TGen^{(2)}_{2-seq}([a\backslash b\backslash a\backslash b\backslash a\backslash b\backslash a\backslash H]) = [(b \bullet a)^*\backslash a\backslash H]$ and not $[b\backslash(a \bullet b)^*\backslash H]$ because the leftmost repeated sequences for $K = J = 2$ start with the leftmost a of $[a\backslash b\backslash a\backslash b\backslash a\backslash b\backslash a\backslash H]$
- $TGen^{(2)}_{2-seq}([H/a/a/a/a/a]) = [H/a^*]$ and not $[H/(a \bullet a)^*]$ because the sequences for a^* and $(a \bullet a)^*$ both start with the leftmost a in $[H/a/a/a/a/a]$ but the pattern length of a^* is one (the smallest) and the pattern length of $(a \bullet a)^*$ is two.
- $TGen^{(2)}_{2-seq}([H/a/b/a/b/a/b/a]) = [H/(b \bullet a)^*/a]$ and not $[H/(b \bullet a)^*/a/b/a]$ because for $K = J = 2$ even if there are two repeated sequences starting at the leftmost a with a pattern length of two $(b \bullet a)$ that are $a/b/a/b$ and $a/b/a/b/a/b$, the maximal number of repetitions is three and corresponds to $a/b/a/b/a/b$.

3.2 Algorithm Properties

Some terminology The following definitions are introduced for ease of writing.

Definition 10 (argument-form). *By an* argument-form *we mean a part of a type with the form $L_m\backslash ...\backslash L_1\backslash$ or the form $/R_1 .../R_n$ where each L_i, R_i is a possible argument in a CDG type (in short an* argument-form *is a writing fragment on one side in a CDG type).*

Definition 11 (Component). *By a* star-component *in a type or an argument-form t, we mean any $x^*\backslash$ or $/x^*$ occurring in the writing of t. By a* primitive component *in a type or an argument-form t, we mean any $x^*\backslash$, $/x^*$, $d\backslash$, or $/d$ where d is a local dependency name or an anchor type, occurring in the writing of t. These notions are extended to the form without $\backslash$ or $/$.*

Definition 12 (Parallel Decomposition). *If t' is the result of the algorithm $TGen^{(K)}_{J-seq}$ on $t = [L_1\backslash \cdots \backslash H/ \cdots /R_1]^P$ in the LML mode, we can decompose in parallel: $t = [\alpha_1 \cdots H \cdots \alpha_N]^P$ and $t' = [\beta_1 \cdots H \cdots \beta_N]^{P'}$ where $P' = sort(P)$, each α_i is an argument-form, β_i is a primitive component and:*

$\beta_1 = TGen^{(K)}_{J-seq}(\alpha_1) \ldots \beta_j = TGen^{(K)}_{J-seq}(\alpha_j) \ldots$ *and* $\beta_N = TGen^{(K)}_{J-seq}(\alpha_N)$

The pair $(\alpha_1 \ldots \alpha_N, \beta_1 \ldots \beta_N)$ defines the parallel decomposition *of (t, t') in the LML mode ; we call (α_i, β_i) a block and we say that each index i selects* block *(α_i, β_i) in the decomposition.*

Construction and key lemmas

Definition 13 (Expansion). *For any type t, we define its* full expansion *$FE(t)$ as the set of types obtained from t by erasing or by replacing its star-components x^* (d^* or $(d_1 \bullet d_2)^*$ when $J = 2$) by any successive repetitions of x.*

Note. This set is infinite when there is at least one star-component, but is used as an intermediate for proofs. It corresponds to the possible vicinities that can be associated to a word in a DS.

Definition 14 (Expansion of Rank K'). *For any t, type or argument-form, we define its* full expansion of rank K', *$FE^{K'}(t)$, as the set of types obtained from t by erasing or by replacing all its star-components x^* by any successive repetitions of x not more than K' times.*

Lemma 1. *Let $K > 1$, $J = 1$ or 2 and $K' \geq K + 1$. For any type t:*

$$TGen^{(K)}_{J-seq}(FE^{K'}(t)) = TGen^{(K)}_{J-seq}(FE^{K+1}(t)) \qquad (1)$$

Proof. We show (1). Obviously $TGen^{(K)}_{J-seq}(FE^{K+1}(t)) \subseteq TGen^{(K)}_{J-seq}(FE^{K'}(t))$. We show the converse for $J = 2$ ($J = 1$ is a subcase of $J = 2$). Suppose $t_1 \in FE^{K'}(t_0)$, let $t_2 = TGen^{(K)}_{J-seq}(t_1)$ and let α_j, β_j, for $1 \leq j \leq N$ denote the parallel decomposition of (t_1, t_2) in the LML mode. We discuss by induction on the construction of t_0, considering the parallel decomposition.

We consider the leftmost star-component x^* in t_0 repeated more than $K + 1$ times in t_1. We show that we can replace it by t'_1 with only $K + 1$ repetitions of this pattern instead (unchanged elsewhere).

- If $|x| = 1$, then x^* of t_0 corresponds to $d\backslash d\backslash \cdots d\backslash$ or $/d/d\backslash \cdots d$ in t_1.
(1.1) If this argument-form of t_1 (and x^* of t_0) corresponds to a unique block i in the parallel decomposition of (t_1, t_2), then α_i contains more than $K+1$ x and $\beta_i = x^*$; in that case, we define t'_1 by replacing in α_i all the repetition of x with only $K+1$ repetitions of x. In this case, x^* of t_0 corresponds to $K+1$ x in t'_1 and the algorithm yields the same type.
(1.2) If the argument-form corresponds to several adjacent blocks in the parallel decomposition of (t_1, t_2), the leftmost x is the end of a block i with $\beta_i = (x \bullet d_1)^*$ and the others are in the block $i+1$ with $\beta_{i+1} = x^*$. α_{i+1} contains at least K x. We define t'_1 by replacing in α_{i+1} all the repetition of x by only K repetitions of x. In this case, x^* of t_0 corresponds to $K+1$ x in t'_1 which yields the same type (algorithm output).
- If $|x| = 2$, then x is the succession of d_1 and d_2 ($x = d_2 \bullet d_1$ and it corresponds to $d_1\backslash d_2\backslash d_1 \cdots \backslash d_1\backslash d_2\backslash$ or $/d_1/d_2/d_1 \cdots /d_1/d_2$):
(2.1) If x^* of t_0 corresponds to a unique block i, in that case, as in (1.1), we define t'_1 by replacing in α_i the repetition of d_1 and d_2 with $K+1$ repetitions of d_1 and d_2. In this case, x^* of t_0 corresponds to $K+1$ x in t'_1 which yields the same type (algorithm output).
(2.2) if $d_1 \neq d_2$ and x^* corresponds to several adjacent blocks in the parallel decomposition of (t_1, t_2) starting at block i, this means that in the LML mode the leftmost d_1 corresponds to the end of block i, the rightmost d_2 correspond to the beginning of block $i+2$ and the other local dependency names $d_2, d_1, d_2 \ldots, d_2, d_1$ correspond to block $i+1$ with $\beta_{i+1} = (d_2 \bullet d_1)^*$. We define t'_1 by replacing in α_{i+1} the repetition of d_2 and d_1 with K repetitions of d_2 and d_1. In this case, x^* of t_0 corresponds to $K+1$ x in t'_1 which yields the same type (algorithm output).
(2.3) if $d_1 = d_2$, we have the same cases as in (1.1) and (1.2) but with more than $2K+2$ local dependency names.

We can repeat this process until no expansion is made more than $K+1$ times, hence the converse inclusion.

For example, if $t_0 = a\backslash a\backslash (b \bullet a)^*\backslash b\backslash b\backslash H$, with $J = 2, K = 2, K' = 4$:
the decomposition for $t_1 = a\backslash a\backslash a\backslash b\backslash a\backslash b\backslash a\backslash b\backslash a\backslash b\backslash b\backslash b\backslash H$ (with $K' = K+2$ repetitions) can be compared to that of $t'_1 = a\backslash a\backslash a\backslash b\backslash a\backslash b\backslash a\backslash b\backslash b\backslash b\backslash H$ with $K+1$ repetitions (we recall that the display order is reverted for internal sequence as arguments):

$$TGen_{2-seq}^{(2)} \begin{vmatrix} \alpha_1 = a\backslash a\backslash a\backslash \\ \beta_1 = a^* \end{vmatrix} \begin{vmatrix} \alpha_2 = b\backslash a\backslash b\backslash a\backslash b\backslash a\backslash \\ \beta_2 = (a \bullet b)^* \end{vmatrix} \begin{vmatrix} \alpha_3 = b\backslash b\backslash b\backslash \\ \beta_3 = b^* \end{vmatrix} \begin{Vmatrix} t_1 \\ t_2 \end{Vmatrix}$$

$$TGen_{2-seq}^{(2)} \begin{vmatrix} \alpha_1 = a\backslash a\backslash a\backslash \\ \beta_1 = a^* \end{vmatrix} \begin{vmatrix} \alpha_2 = b\backslash a\backslash b\backslash a\backslash \\ \beta_2 = (a \bullet b)^* \end{vmatrix} \begin{vmatrix} \alpha_3 = b\backslash b\backslash b\backslash \\ \beta_3 = b^* \end{vmatrix} \begin{Vmatrix} t'_1 \\ t_2 \end{Vmatrix}$$

Note that $a\backslash a\backslash a\backslash b\backslash a\backslash b\backslash b\backslash b\backslash$, with K repetitions only, yields a different decomposition.

Corollary 1. *Let $K > 1$ and $J = 1$ or 2. For any type t the result of the algorithm $TGen^{(K)}_{J-seq}$ on the full extension of t is a finite set and is the same set as the result of this algorithm on $FE^{K+1}(t)$.*

The definitions of FE^K and FE^{K+1} are extended to sets, lexicons and grammars in the usual way.

Lemma 2. *Let $K > 1$ and $J = 1$ or 2. Let G be a CDG with sequence iterations. We have:*
(1) all vicinities of words in DS of $\Delta(G)$ belong to some $FE(t)$, where t is assigned by G.
(2) if σ is a finite sequence in $\Delta(G)$, then $\Delta(TGE^{(K)}_{J-seq}(\sigma)) \subseteq \Delta(G')$ where G' is $TGE^{(K)}_{J-seq}$ on $FE^{K+1}(G)$

Proof. If G generates $D \in \sigma$ where a word w occurs with a vicinity t_w, for which G uses the assignment $w \mapsto t$ in the derivation, then t_w must be in $FE(t)$. Finally, we use Corollary 1 relating $FE(t)$ to $FE^{K+1}(t)$.

Theorem 1 (Convergence). *Let $K > 1$ and $J = 1$ or 2. Let G be any CDG. The algorithm $TGE^{(K)}_{J-seq}$ stabilizes on every training sequence in $\Delta(G)$ to a grammar with assignments in $TGE^{(K)}_{J-seq}$ on $(FE^{K+1}(G))$.*

Proof. We have (1) $TGE^{(K)}_{J-seq}(\sigma[i]) \subseteq TGE^{(K)}_{J-seq}(\sigma[i{+}1]) \subseteq ...$ As observed in Lemma 2, the vicinities for the words of the DS in σ belong to $FE(G)$. If we had an infinite chain of types $t'_i = TGE^{(K)}_{J-seq}(t_i)$, with assignments $w_i \mapsto t'_i$ in $TGE^{(K)}_{J-seq}(\sigma[i])$, but not in $TGE^{(K)}_{J-seq}(\sigma[i-1])$ (we could consider one such chain concerning a same word w as the lexicon of G is finite) ; now all t_i also belong to some $FE^{K_i}(G)$, then if $K' > K+1$, there exists t''_i in $FE^{K+1}(G)$, such that $t'_i = TGE^{(K)}_{J-seq}(t''_i)$, we can thus view the set of t'_i as the result of $TGE^{(K)}_{J-seq}$ on a subset of $FE^{K+1}(G)$; obviously $FE^{K+1}(G)$ is finite, we would then have a contradiction.
Therefore for any G and any $K > 1$:
$\exists N, \forall N' \geq N\ TGE^{(K)}_{J-seq}(\sigma[N']) = TGE^{(K)}_{J-seq}(\sigma[N])$
Furthermore, if $w \mapsto t' \in TGE^{(K)}_{J-seq}(\sigma[N])$ there exists $w \mapsto t'' \in FE^{K+1}(G)$, such that $t' = TGE^{(K)}_{J-seq}(t'')$: in that sense the assignments in $TGE^{(K)}_{J-seq}(\sigma[N])$ are in $TGE^{(K)}_{J-seq}$ on $(FE^{K+1}(G))$.

Proposition 1. *Let $K > 1$ and $J = 1$ or 2.*
If G is a CDG and σ is a sequence in $\Delta(G)$ then
(1) $TGE^{(K)}_{J-seq}(\sigma[i]) \subseteq TGE^{(K)}_{J-seq}(\sigma[i+1])$ *monotonicity/incrementality*
(2) $\sigma[i] \subseteq \Delta(TGE^{(K)}_{J-seq}(\sigma[i]))$ *expansivity*
(3) $\Delta(TGE^{(K)}_{J-seq}(\sigma[i])) \subseteq \Delta(G')$ where G' is $TGE^{(K)}_{J-seq}$ on $FE^{K+1}(G)$

Proof. (1) holds by definition of the algorithm (that expands the lexicon) ; (2) can be shown by adapting the derivation ; (3) follows from a preceeding lemma.

3.3 A Family of Learnable Classes

Definition 15. *Two grammars are said* strongly equivalent *if they generate the same dependency structure language. The* strong equivalence criterion*:*

(i) G is strongly equivalent to $TGE^{(K)}_{J-seq}$ on $FE^{K+1}(G)$

defines the subclass written $\mathcal{CCDG}^{K}_{J-seq}$ of grammars satisfying (i).

Theorem 2. *Let $K > 1$ and $J = 1$ or 2. The algorithm $TGE^{(K)}_{J-seq}$ learns the class of CDG satisfying the strong equivalence criterion (i), from labelled dependency structures.*

Proof. From Proposition 1(1) : $TGE^{(K)}_{J-seq}(\sigma[i]) \subseteq TGE^{(K)}_{J-seq}(\sigma[i{+}1]) \subseteq ...$
The stabilization property holds (Theorem 1):

$\exists N, \forall N' \geq N\ TGE^{(K)}_{J-seq}(\sigma[N']) = TGE^{(K)}_{J-seq}(\sigma[N])$

Then by Proposition 1(2): $\Delta(G) \subseteq \Delta(TGE^{(K)}_{J-seq}(\sigma[N]))$,

and using (i) and Proposition 1(3): $\Delta(G) \subseteq \Delta(TGE^{(K)}_{J-seq}(\sigma[N])) \subseteq \Delta(G)$.
Therefore for any grammar, such that (i) we get the convergence to a grammar generating the same structure language.

Observe that this class does not impose a bound on the number of types associated to a word (in contrast to k-valued grammars). The learnability for $J = 1$ was studied in [3], with a special case of our algorithm.

4 Extended CDG and Dependency Treebanks

From dependency treebanks to vicinities. Our workflow applies to data in the Conll format[9]. The CDG potentials in this section are considered as empty[10].

For each governor unit in each corpus we have computed (using `MySQL` and `Camelis`[11]): (1) its *vicinity* in the root simplified form $[l_1 \backslash ... \backslash l_n \backslash root / r_m / .... / r_1]$ (where l_1 to l_n on the left and r_1 to r_m on the right are the successive dependency names from that governor), then (2) its generalization as *star-vicinity*, replacing consecutive repetitions of d_k on a same side with d_k^*; and (3) its generalization as *vicinity_2seq* following the LML mode of the algorithm in Figure 6 for J=K=2.

Our development allows to mine repetitions and to call several kinds of viewers : we use the item/word description interactive viewer `camelis` and the sentence parse conll viewer [11] or grew[12].

Figure 7 on its left, shows the *root simplified vicinities* computed on corpus Sequoia; the resulting file has been loaded as an interactive information context, in `Camelis` ; this tool manages three synchronised windows: the current query is on the top, selecting the objects on the right, their properties can be browsed in the multi-facets index on the left.

[9] `http://universaldependencies.org/format.html`
[10] this complies with Sequoia data, but may be a simplification for some other corpora
[11] `www.irisa.fr/LIS/softwares`
[12] `http://talc2.loria.fr/grew/`

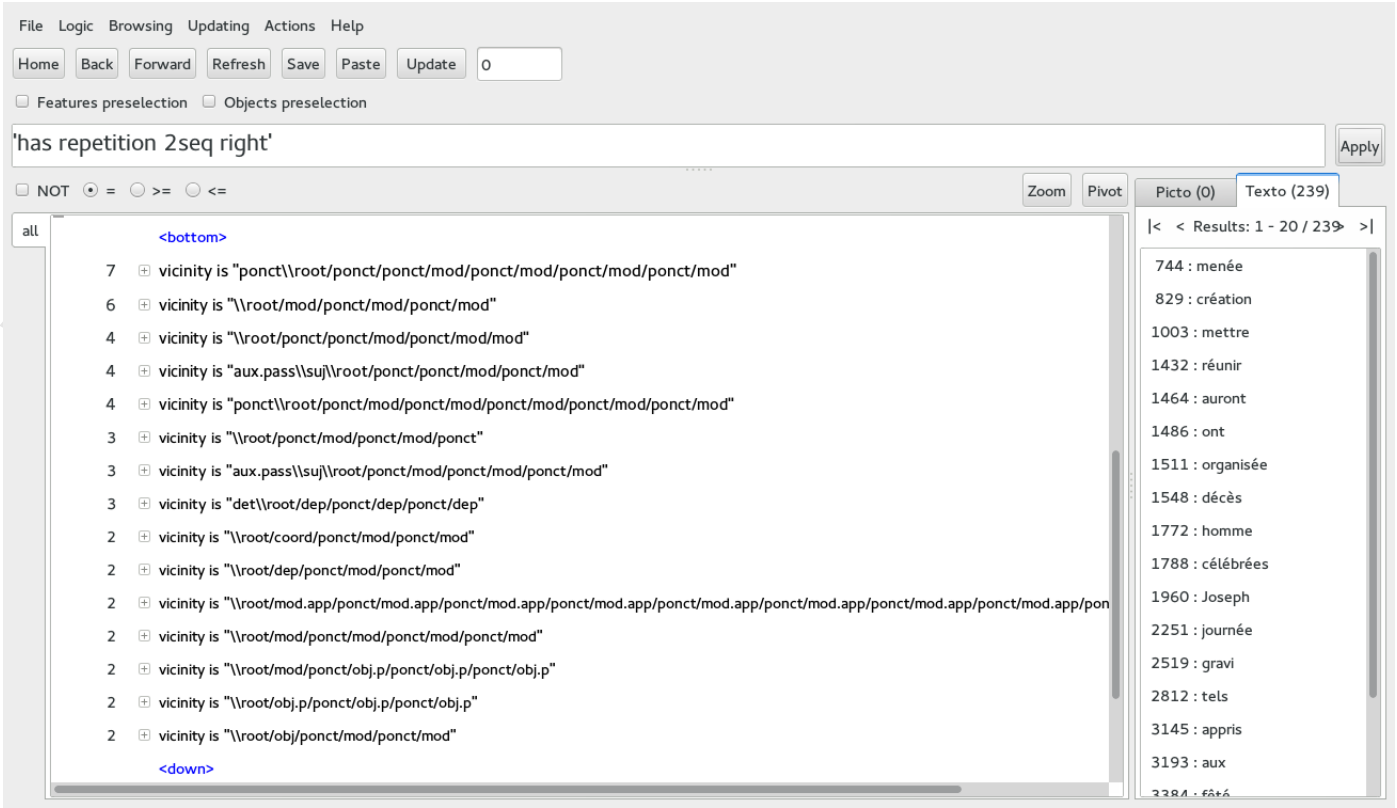


Fig. 7. Simplified vicinities computed on corpus Sequoia

Results on the French corpus Sequoia. We consider a version of corpus Sequoia [4] that defines dependency structures. the study uses only the surface syntax dependency tree. Sequoia is not validated by a dependency grammar in the sense of Mel'čuk and does not have to follow the repeatable principle.

The process yields 530 distinct star-vicinities having repetition(s) (a star), among 2660 distinct vicinities (on 67038 units, among which 37883 governors). For example the form "notables"[13] with postag "NC" has:

vicinity `det\root/mod/mod/dep/dep` and **star-vicinity** `det\root/mod*/dep* .`

We observe that:[14] *consecutive* repeatable dependencies $d_1.d_1$ on the left are : aff, **dep**, det, **mod**, **ponct** ; *consecutive* repeatable dependencies on the right are: **coord**, **dep** (+ dep.coord), **mod** (+ mod.app), obj:obj+obj.p, p_obj.o, **ponct**

The most frequent vicinity_star is `"det\root/mod*"` (204 units), the most frequent vicinity_2seq is `"\root/(mod . ponct)*"` (25 units), 166 units correspond to a repetition `"(mod . ponct)*"`. Several *repeated sequences of length 2* occur, either on the left or on the right, these patterns always include a ponct dependency: (suj . ponct) (ponct . suj) (ponct . obj.p) (ponct . obj) (ponct . mod.voc) (ponct . mod.rel) (ponct . mod.app) (ponct . mod) (ponct . dep.coord) (ponct . dep) (ponct . coord) (p_obj.o . ponct) (obj.p . ponct) (obj . ponct) (obj.cpl . ponct) (mod.voc . ponct) (mod . ponct) (mod.app . ponct) (dep . ponct) (dep.coord . ponct) (de_obj . ponct) (coord . ponct) (ats . ponct)

Repeated sequences of length 3, with three distinct dependencies seem to be rare. We found one sentence[15] illustrating this case: " Ils ont vidé les supermarchés de nourriture, les pharmacies de médicaments, les usines de matériel médical, ils ont cambriolé les maisons et torturé des voisins et des amis.", with vicinity:

`"aux.tps\\suj\\root/ponct/mod/ponct/de_obj/obj/ponct/de_obj/obj/ponct/de_obj/obj"`

Other corpus. Our development can handle other treebanks in the conll format. Table 4 summarizes some observations on two corpus, with the number of units corresponding to repetition patterns.

Treebank	sentences	units	governors	J=1	J=2	J=3, left
sequoia	3099	67038	37883	1667	378	0
fr-ud-train	3312	74979	33568	1942	220	0

Fig. 8. Dependency repetitions, for K=2 and sequence length J

In the fr-ud-train corpus, the most frequent vicinity_star is `"det\root/adpmod*"` (194 units) , the most frequent vicinity_2seq is `"\root/(p . conj)*"` ; 45 units correspond to a repetition `(adpmod . p)*`. The 18 repeating patterns are:
(p . parataxis) (p . nsubj) (p . mwe) (p . dep) (p . conj) (p . compmod) (parataxis . p) (p . appos)

[13] `talc2.loria.fr/deep-sequoia/sequoia-7.0/html/frwiki_50.1000_00315.html`
[14] bold denotes the frequent ones
[15] `talc2.loria.fr/deep-sequoia/sequoia-7.0/html/frwiki_50.1000_00091.html`

(p . advmod) (p . adpmod) (nmod . p) (conj . p) (compmod . p) (cc . conj) (aux . neg) (amod . p) (advmod . p) (adpmod . p)

5 Conclusion

In this paper, we have extended classical Categorial Dependency Grammars with a new construction to handle repeatable sequences of several dependencies. The work was motivated by the observation of such patterns. We have proposed a learning algorithm. A version of this algorithm has been implemented and applied to some treebanks (in Conll). Some design and computational variants are possible depending on the *repetition principle reading*. On the formal side, further analysis could consider richer patterns. On the experimental side, other treebanks could be explored as well. It would also be interesting to reconsider these notions in other formalisms or application domains.

References

1. Abrusci, V., Ruet, P.: Non-commutative logic i: the multiplicative fragment. Annals of Pure and Applied Logic 101(1), 29 – 64 (1999)
2. Angluin, D.: Inductive inference of formal languages from positive data. Information and Control 45, 117–135 (1980)
3. Béchet, D., Dikovsky, A.J., Foret, A.: Two models of learning iterated dependencies. In: FG. Lecture Notes in Computer Science, vol. 7395, pp. 17–32. Springer (2010)
4. Candito, M., Perrier, G., Guillaume, B., Ribeyre, C., Fort, K., Seddah, D., de la Clergerie, E.: Deep syntax annotation of the sequoia french treebank. In: Proceedings of LREC. pp. 2298–2305. European Language Resources Association (ELRA) (May 2014)
5. Dekhtyar, M., Dikovsky, A., Karlov, B.: Categorial dependency grammars. Theoretical Computer Science 579, 33–63 (2015)
6. Dikovsky, A.: Dependencies as categories. In: Kruijff, G.J.M., Duchier, D. (eds.) COLING 2004 Recent Advances in Dependency Grammar. pp. 82–89. COLING, Geneva, Switzerland (August 28 2004), `http://aclweb.org/anthology/W04-1512`
7. Gold, E.M.: Language identification in the limit. Information and control 10, 447–474 (1967)
8. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, New York, NY, USA (2010)
9. Kanazawa, M.: Learnable classes of categorial grammars. Studies in Logic, Language and Information, FoLLI & CSLI (1998)
10. Lambek, J.: On the calculus of syntactic types. In: Jakobson, R. (ed.) Structure of languages and its mathematical aspects, pp. 166–178. American Mathematical Society, Providence RI (1961)
11. Rosa, R.: Terminal-based CoNLL-file viewer (2014), `http://hdl.handle.net/11234/1-1456`, LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague