Supplemental Papers for DMTCS'03

C. S. Calude[1], M. J. Dinneen[1] and V. Vajnovszki[2] (Editors)

[1]University of Auckland, New Zealand
[2]Université de Bourgogne, France

CDMTCS-215
May 2003

Preface

These are the papers for the poster talks to be given at the Fourth International Conference *Discrete Mathematics and Theoretical Computer Science, 2003*, (DMTCS'03) to be held at the Dijon, France on July 7–12, 2003. The conference is jointly organized by the University of Bourgogne (France) and the CDMTCS (New Zealand).

On D-trivial Dependency Grammars[⋆]

Radu Gramatovici[1], Martin Plátek[2]

[1] Faculty of Mathematics, University of Bucharest
Academiei 14, RO-70109, Bucharest, Romania
radu@funinf.cs.unibuc.ro
[2] Faculty of Mathematics and Physics, Charles University,
Malostranské nám. 25, CZ-11800, Prague, Czech Republic
platek@ksi.ms.mff.cuni.cz

Abstract. An infinite sequence of incomparable classes of semilinear languages bounded between the class of regular and respectively context-sensitive languages is introduced. This sequence is obtained by the relaxation of word order in a sentence represented by a (possibly) non-projective DR-tree, while keeping the projectivity of the corresponding D-tree. Our results argue a significant difference between the (non-)projectivity constraints of DR-trees and respectively D-trees and focus on the intrinsic importance of DR-trees.

1 Introduction

The notion of free-order dependency grammar (or simply *dependency grammar*) was introduced in [3] as a formal system suitable for a dependency-based parsing of natural languages. In a certain way this notion enriches the types of dependency grammars described in [1].

The proposal of this system was based upon the experience acquired during the development of a grammar-checker for Czech and as a possible next step towards a complete syntactic analysis following the underlying ideas of the dependency-based framework of Functional Generative Description - FGD ([5]). Compared to FGD and other usual formal systems describing the syntax of natural languages, the framework introduced by dependency grammars takes a serious account for the freedom of word order in a sentence and assigns the same importance to linear precedence (LP) rules as to immediate dominance (ID) rules.

As the freedom of word order is not total, even in so-called *free-word-order languages*, one needs to constrain the formalism in order to not overgenerate the actual language.

In [4], a measure for the freedom of the word order was studied, based on the number of gaps issued in a sentence by the order of their words (*node-gaps-complexity*). Both *global* and *local constraints* on the maximal number of

[⋆] This work was supported by the grant of GAČR No. 201/02/1456 and by the project of the Ministry of Education of the Czech Republic No. LN00A063.

gaps at some node in the structure underlying the sentence were studied. In the view of node-gaps-complexity, word order relaxation means to stepwisely relax the constraints in order to obtain more complex language constructions. In this paper, we work only with global constraints.

Two types of syntactic structures are used in the relationship with dependency grammars, **DR**-*trees* (**D**elete **R**ewrite trees) and **D**-*trees* (**D**ependency trees). If D-trees concern the dependency structure of the sentence, DR-trees rather concern the generation/parsing of the sentence. The two types of structures are related by the fact that any DR-tree can be transformed in an uniform way into a D-tree. The measure for the number of gaps in a sentence computed in the nodes of the structure is introduced for both DR-trees and D-tree. A DR-tree (D-tree) with no gaps is called *projective*, while a DR-tree (D-tree) with at least one gap is called *non-projective*. In this paper, we work only with dependency grammars which possibly create non-projective DR-trees, but cannot create non-projective D-trees (*D-trivial grammars*).

The main result of this paper presents an infinite sequence of incomparable classes of semilinear languages generated by D-trivial grammars (Section 3). Moreover, all these classes of languages are strictly bounded between the class of regular languages and the class of context-sensitive languages, each of them containing context-free and non-context-free languages (Section 4).

We prove by the results we obtain in this paper that there is a significant difference between the projectivity of DR-trees and, respectively, D-trees. We also try to argue in this way the importance of the non-projective generation/parsing of the sentence, which is represented by the DR-tree compared to the non-projectivity of the sentence itself which is represented by the D-tree. In other word, even if is a sentence is represented by a projective D-tree, this D-tree can hide some non-projective concurrency phenomena raising in the generation or the parsing of the sentence.

2 DR-trees and D-trees

Let M be a set. Let Tr = *(Nod,Ed,Rt,Ann)* be a 4-tuple , where Nod is a set (*the set of nodes*), $Rt \in Nod$ is a special node (*the root*), $Ed : Nod \setminus \{Rt\} \to Nod$ is a function (*the set of edges*) and $Ann : Nod \to M$ is a function (*the annotation function*). We call **path** in Tr any sequence of nodes from Nod, $p = (n_1, n_2, \ldots, n_k)$, with $k \geq 1$, such that $Ed(n_i) = n_{i+1}$, for $i = 1, \ldots k-1$. We say that p is a *path of length* $k-1$ *from* n_1 *to* n_k. If $k > 1$, we denote p by $Path(n_1, n_k)$. We say that Tr is a M-**annotated tree** iff there is no path of positive length in Tr from a node $n \in Nod$ to itself. Let $n \in Nod$ be a node in Tr. We say that n is a **leaf** iff $n \neq Ed(n')$, for any $n' \in Nod$.

We say that Tr is a **finite** M-annotated tree iff its set of nodes, Nod, is finite. In the sequel of this paper, we will work only with finite annotated trees, without clearly mention it.

If Tr = *(Nod, Ed, Rt, Ann)* is an annotated tree and $n \in Nod$ is a node in Tr then there exists exactly one path from n to Rt. Moreover, if $n_1, n_2 \in Nod$

are two nodes in Tr then there exists at least one node $n_3 \in Nod$ such that there exist a path from n_1 to n_3 and a path from n_2 to n_3. This last remark allows us that for any two nodes if $n_1, n_2 \in Nod$ to denote by $sup(n_1, n_2)$ the first node in Tr which connects n_1 and n_2, i.e. $(n_1, \ldots, Rt)$ and $(n_2, \ldots, Rt)$ are the paths from n_1 respectively n_2 to Rt, then $sup(n_1, n_2)$ is the first node, which belongs to both of these paths. From the definition of an annotated tree, $sup(n_1, n_2)$ is uniquely defined by this property.

Let $Tr_1 = (Nod_1, Ed_1, Rt_1, Ann_1)$ and $Tr_2 = (Nod_2, Ed_2, Rt_2, Ann_2)$ be two M-annotated trees. We say that Tr_1 and Tr_2 are **equivalent** iff there is a bijection $f : Nod_1 \to Nod_2$ such that: i) $f(Ed_1(n)) = Ed_2(f(n))$, $\forall n \in Nod_1 \setminus \{Rt_1\}$; ii) $f(Rt_1) = Rt_2$; iii) $Ann_1(n) = Ann_2(f(n))$, $\forall n \in Nod_1$. We call f an **isomorphism** between Tr_1 and Tr_2.

In [3], (free-order) dependency grammars were introduced, as a rewriting device over two alphabets, of non-terminals and, respectively, terminals. In its general form, a dependency grammar can rewrite both non-terminals and terminals, by a finite set of rules (productions). Through out this paper, we will work with dependency grammars, which rewrite only non-terminals (in a similar way to context-free grammars), therefore, we will not use terminals on the lefthand-sides of the rules.

We call **dependency grammar** a structure $G = (N, T, S, P)$ such that N and T are non-empty, finite sets (the set of *nonterminals*, respectively, of *terminals*), $S \in N$ is the start symbol and P is a finite set, called the set of *productions* such that $P \subseteq (N \times VV \times \{L, R\}) \cup (N \times T)$, where $V = N \cup T$. Sometimes, we will write the productions in P as $A \to_L BC$, $A \to_R BC$, $A \to a$ instead of (A, BC, L), (A, BC, R), respectively (A, a).

Denote by Nat the set of natural numbers not equal to 0 and by $Nat_0 = Nat \cup \{0\}$.

Let $G = (T, N, S, P)$ be a dependency grammar and denote $V = N \cup T$. A **DR-tree created by** G is a V-annotated tree $Tr = (Nod, Ed, Rt, Ann)$ such that:

1. $Nod \subseteq Nat \times Nat$. If $Ed(i,j) = (k,l)$ then $j < l$. $Rt = (i, \max\{k \mid \exists j, (j,k) \in Nod\})$.
2. A node $(i,j) \in Nod$ is a leaf if and only if $j = 1$ and $Ann(i,j) \in T$.
3. If $(i,j) \in Nod$, with $j \neq 1$ and $Ann(i,j) = A$, then one of the following cases necessarily occurs:
   a. $j = 2$ and there is exactly one node $n \in Nod$ such that $Ed(n) = (i,j)$; in this case $n = (i,1)$ and if $Ann(n) = a$, the production $A \to a$ belongs to P.
   b. There are exactly two nodes $n_1, n_2 \in Nod$ such that $Ed(n_1) = Ed(n_2) = (i,j)$; in this case either:
      b1. $n_1 = (i,k)$ and $n_2 = (l,m)$ with $l > i$, $\max(k,m) = j - 1$ and if $Ann(n_1) = B$ and $Ann(n_2) = C$, the production $A \to_L BC$ belongs to P, or
      b2. $n_1 = (l,k)$ and $n_2 = (i,m)$ with $l < i$, $\max(k,m) = j - 1$ and if $Ann(n_1) = B$ and $Ann(n_2) = C$, the production $A \to_R BC$ belongs to P.

Let $n_o \in Nod, n_o = (i,j)$. We say that i is the **horizontal position** of n_o and j is the vertical position of n_o (see two examples of a DR-tree in Figure 1). Let $Ed(n_1) = n_2$, i.e., $e_1 = (n_1, n_2) \in Ed$. Let i be the horizontal position of n_1 and j the horizontal position of n_2. If $i = j$ we say that e_1 is a **V-edge**, if $i > j$ we say that e_1 is an **L-edge**, if $i < j$ we say that e_1 is a **R-edge**. We say that a DR-

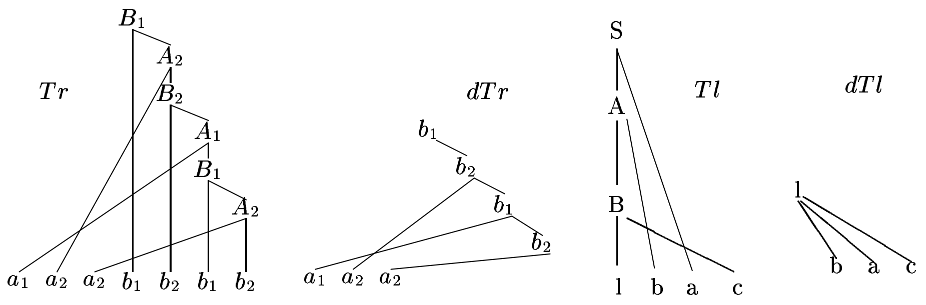

Fig. 1. Two examples of DR-trees and corresponding D-trees

tree *Tr = (Nod, Ed, Rt, Ann)* is **complete** iff for any leaf $(i,1) \in Nod$, if $i > 1$ then also $(i-1,1) \in Nod$. For any complete DR-tree *Tr = (Nod, Ed, Rt, Ann)* created by a dependency grammar $G = (N,T,S,P)$, we define the **sentence** associated with *Tr* by $s(Tr) = a_1 a_2 \ldots a_n$, where $n = \max\{i \mid (i,1) \in Nod\}$ and $Ann(i,1) = a_i$, for any $i \in [n]$. Obviously, $s(Tr) \subseteq T^+$.

Let $G = (N,T,S,P)$ be a dependency grammar. We denote by:

- $T(G)$ the set of complete DR-trees created by G and rooted by S;
- $DR\text{-}L(G) = \{s(Tr) \mid Tr \in T(G)\}$ the language generated by G, through the set of DR-trees.

Let $Tr_1 = (Nod_1, Ed_1, Rt_1, Ann_1)$ and $Tr_2 = (Nod_2, Ed_2, Rt_2, Ann_2)$ be two DR- trees created by the grammar $G = (N,T,S,P)$. We say that Tr_1 and Tr_2 are **DR-equivalent** iff there is an isomorphism $f : Nod_1 \to Nod_2$ between Tr_1 and Tr_2 as V-annotated trees and:

1. $f(i,j) = (s,j)$, for any node $(i,j) \in Nod_1$.
2. If $(i,j) \in Nod_1$, with $j \neq 1$ and $f(i,j) = (s,j)$, then:
   a. if there is exactly one node $n \in Nod_1$ such that $Ed(n) = (i,j)$ then $f(n) = (s, j-1)$.
   b. if there are exactly two nodes $n_1, n_2 \in Nod$ such that $Ed(n_1) = Ed(n_2) = (i,j)$ then either:
      b1. if $n_1 = (i,k)$ and $n_2 = (l,m)$ with $l > i$, then $f(n_1) = (s,k)$ and $f(n_2) = (t,m)$ with $t > s$ or
      b2. if $n_1 = (l,k)$ and $n_2 = (i,m)$ with $l < i$, then $f(n_1) = (t,k)$ and $f(n_2) = (s,m)$ with $t < s$.

We say that f is a **DR-isomorphism** between Tr_1 and Tr_2.

Let T be an alphabet and *Tr = (Nod, Ed, Rt, Ann)* be a T-annotated tree such that $Nod \subseteq Nat$. We call Tr a **D-tree over** T (see two examples of D-trees in Figure 1). Any DR-tree *Tr = (Nod, Ed, Rt, Ann)* can be uniformly transformed in a unique D-tree *dTr = (dNod, dEd, dRt, dAnn)* in the following way:

1. $dNod = \{i \mid (i,1) \in Nod\}$. $dRt = i$ iff $Rt = (i,j)$, for some $j \in Nat$.
2. Let $i \in dNod$ be a node in dTr and $(i,1) \in Nod$ the corresponding leaf in Tr. We consider the path $p = (n_1, n_2, \ldots, n_k)$ in Tr from $n_1 = (i,1)$ to $n_k = Rt$. We also consider the natural number $r = \max\{l \mid n_l = (i,j), j \in Nat\}$. Then one of the following cases necessarily occurs:
   If $r = k$ then $dRt = i$. If $r < k$ and $n_{r+1} = (s,t)$ then $dEd(i) = s$.
3. $dAnn(i) = Ann(i,1)$, for any $i \in dNod$.

We say that dTr is the D-tree **corresponding** to Tr. The D-tree dTr represented in Figure 1 is corresponding to the DR-tree Tr from the same figure. Similarly dTl corresponds to Tl. If dTr is a D-tree corresponding to a DR-tree created by a dependency grammar G, we say that dTr is created by G as well. A D-tree is **complete** if it corresponds to a complete DR-tree. Let $dEd(i) = j$, i.e., $e_1 = [i,j] \in dEd$. If $i > j$ we say that e_1 is an **L-edge**, if $i < j$ we say that e_1 is a **R-edge**.

Let $Tr = (Nod, Ed, Rt, Ann)$ be an annotated tree, $n \in Nod$ be a node. We define the **covering subtree** of n in Tr by the following annotated tree, $Tr_n = (Nod_n, Ed_n, Rt_n, Ann_n)$ such that :

i) $Nod_n = \{n' \mid$ there is a path from n' to n in $Tr\}$;
ii) $Ed_n(n') = Ed(n')$, $\forall n' \in Nod_n \setminus \{n\}$;
iii) $Rt_n = n$;
iv) $Ann_n(n') = Ann(n')$, $\forall n' \in Nod_n$.

Let $Tr = (Nod, Ed, Rt, Ann)$ be a complete DR-tree, $n \in Nod$ and let $Tr_n = (Nod_n, Ed_n, Rt_n, Ann_n)$ be the covering subtree of n in Tr. We define the **coverage** of n in Tr by the set $Cov(n, Tr) = \{i \in Nat \mid$ there is a node $(i,1) \in Nod_n\}$. Let $n \in Nod$ be a node in Tr such that $Cov(n, Tr) = \{i_1, i_2, \ldots, i_m\}$, with $i_1 < i_2 < \ldots < i_m$ and $i_{j+1} - i_j > 1$ for some $j \in Nat$, $j < m$. We say that the pair (i_j, i_{j+1}) is a **gap** in Tr at the node n. Let $Tr = (Nod, Ed, Rt, Ann)$ be a complete DR-tree, $n \in Nod$ be a node and $Cov(n, Tr)$ be its coverage. The symbol *DR-Ng(n,Tr)* represents the number of gaps in Tr at the node n. The symbol *DR-Ng(Tr)* is the maximum number of gaps in Tr at any node $n \in Nod$:

$$\textit{DR-Ng(Tr)} = \max\{\textit{DR-Ng(n,Tr)} \mid n \in Nod\}.$$

We say that *DR-Ng(Tr)* is the **DR-node-gaps complexity** of Tr. We say that Tr is **projective** iff *DR-Ng(Tr)=0*.

Let $G = (N, T, S, P)$ be a dependency grammar. We denote by:

– $T(G,i) \subseteq T(G)$ the set of complete and rooted by S DR-trees Tr created by G with at most i gaps, *DR-Ng(Tr)* $\leq i$;

- *DR-L(G,i)* $= \{s(Tr) \mid Tr \in T(G,i)\}$ the language generated by G, through DR-trees with at most i gaps.

We mention the following obvious claims.

Claim. Let G be a dependency grammar. Then the following inclusions hold for any $i \in Nat_0$.

$T(G,i) \subseteq T(G,i+1) \subseteq T(G)$, *DR-L(G,i)* $\subseteq$ *DR-L(G,i+1)* $\subseteq$ *DR-L(G)*.

Claim. For any complete DR-tree Tr_1 created by a dependency grammar G there exists a DR-equivalent projective complete DR-tree Tr_2 created by the same grammar G.

Let us suppose that Tr_3 is a DR-tree DR-equivalent to Tr_1. Then Tr_3 is also created by G.

The same node-gaps-complexity measure can be defined for D-trees. In this paper, we are interested only in projective D-trees, therefore we will define in the following only this notion.

Let $dTr = (dNod, dEd, dRt, dAnn)$ be a complete D-tree, $n \in dNod$ be a node and $dTr_n = (dNod_n, dEd_n, dRt_n, dAnn_n)$ be the covering subtree of n in dTr. We define the **coverage** of n in dTr by the set $Cov(n, dTr) = dNod_n$. Let $n \in dNod$ be a node in dTr such that $Cov(n, dTr) = \{i_1, i_2, \ldots, i_m\}$, with $i_1 < i_2 < \ldots < i_m$ and $i_{j+1} - i_j > 1$ for some $j \in Nat$, $j < m$. We say that the pair (i_j, i_{j+1}) is a **gap** in dTr at the node n. We say that dTr is **projective** iff dTr has no gaps at any node $n \in dNod$. The following result between the projectivity of DR-trees and corresponding D-trees can be easily proved.

Claim. If Tr is a projective complete DR-tree, then dTr, the corresponding D-tree is also projective.

The reverse statement is not true as we may see in the below example.

Example 1. Let us consider $G = (N,T,S,P)$ a dependency grammar, with $N = \{A,B,C,S\}$, $T = \{a,b,c,l\}$, $P = \{S \to Aa, S \to l, A \to_L Bb, B \to_L Cc, B \to_L lc, C \to_L Aa\}$. We can easily define a non-projectively parsed DR-tree for which the corresponding D-tree is projectively parsed (see in Figure 1 the DR-tree Tl and D-tree dTl).

3 D-trivial grammars with restrictions

In this section, we will define and study several particular forms of DR-trees and dependency grammars. In the sequel of this paper, we will consider only dependency grammars in a normal form described by the following properties: the start symbol does not occur in the righthand side of any production; in any production of the form $A \to a$, the lefthand side nonterminal is the start symbol; any production is used in at least one complete DR-tree created by the grammar and rooted by the start symbol.

It is not hard to see that the D-grammars with the previous restrictions keeps the power of nonrestricted D-grammars due to the generation of the languages and of the sets of D-trees.

Definition 1. *Let G be a dependency grammar. We say that G is a* **D-trivial** *grammar if G creates only projective D-trees.*

Definition 2. *Let $dTr = (dNod, dEd, dRt, dAnn)$ be a D-tree and $dNod = \{1, ..., n\}$. We say that dTr is a* **D-trivial D-tree** *iff it holds:*

a) Let Pl be a path of dTr which contains an L-edge. Then all edges in Pl are L-edges. Obviously, if Pr is a path which contains an R-edge then all edges in Pr are R-edges.

b) There exists at most one node $n_R \in dNod$ ($n_L \in dNod$) such that into n_R (n_L) lead at least two R-edges (L-edges). We call the n_R the **left cross-node** *of dTr and the n_L the* **right cross-node** *of dTr.*

We can see that if $n_R = n_L$ then also $n_R = dRt$. In this case we call dTr an **LR-bush**.

If $n_R \neq n_L$ we call the covering subtree of n_R (n_L) the **R-bush (L-bush)** *of dTr. We can see that $Cov(n_R, dTr) = \{1, ..., n_R\}$ ($Cov(n_L, dTr) = \{n_L, ..., n\}$) and the nodes $\{1, ..., n_R - 1\}$ ($\{n_L + 1, ..., n\}$) are leaves of dTr.*

We will say that two D-trees dTr_1, dTr_2 are **D-equivalent** *iff there are two DR-equivalent DR-trees Tr_1, Tr_2 such that dTr_1 corresponds to Tr_1 and dTr_2 corresponds to Tr_2.*

Let us focus on Fig. 2, which contains some types of D-trivial D-trees. The picture a) depicts an LR-tree. The picture b) depicts a so called L-path. The D-tree c) does not contain the R-bush, but does contain the L-bush. The D-tree d) contains only L-edges. The D-tree dTl in Fig. 1 is an L-bush.

We can make gradually the following observations on the form of D-trees created by a D-trivial grammar.

Claim. i) Let dTr be a D-trivial D-tree. We can see that any D-tree D-equivalent to dTr is D-projective and D-trivial.

ii) To any D-tree which is not a D-trivial D-tree there is a D-equivalent D-tree which is not D-projective.

From the previous observations and from Claim 2, it follows the next statement.

Claim. Let $G = (N, T, S, P)$ be a dependency grammar. Then G is a D-trivial grammar iff any D-tree created by G is a D-trivial D-tree.

Based on the above claims and observations, the following result characterizes the form of productions in a D-trivial grammar.

Proposition 1. *Let $G = (N, T, S, P)$ be a dependency grammar. Then G is a D-trivial grammar iff there is a partition of N such that $N = \bigcup_{1 \leq i \leq 12} N_i$, $N_i \cap N_j = \emptyset$, for any $1 \leq i < j \leq 12$, $N_1 = \{S\}$, and the set of productions is described by:*

$$P \subseteq \{N_1 \times [T \cup ((TT \cup TN_2 \cup N_9(N_4 \cup T)) \times \{R\}) \cup \\ \cup ((TT \cup N_3T \cup (N_5 \cup T)N_{11}) \times \{L\})]\} \cup$$

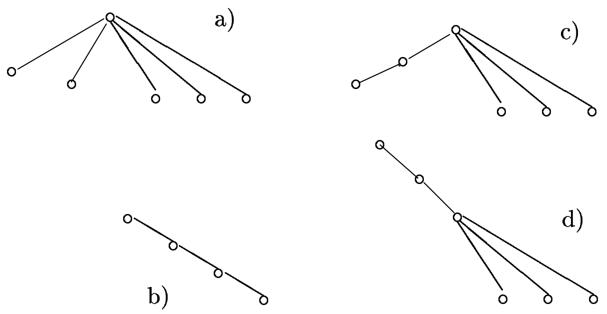

Fig. 2. Four examples of typical D-trivial D-trees.

$$\{N_2 \times [((TT \cup TN_2) \times \{R\}) \cup ((TT \cup N_6T \cup (N_7 \cup T)N_{11}) \times \{L\})]\} \cup$$
$$\{N_3 \times [((TT \cup TN_6 \cup N_9(N_8 \cup T)) \times \{R\}) \cup ((TT \cup N_3T) \times \{L\})]\} \cup$$
$$\{N_4 \times (TT \cup N_8T \cup TN_{11}) \times \{L\}\} \cup \{N_5 \times (TT \cup TN_7 \cup N_9T) \times \{R\})\} \cup$$
$$\{N_6 \times [((TT \cup TN_6) \times \{R\}) \cup ((TT \cup N_6T) \times \{L\})]\} \cup$$
$$\{N_7 \times (TT \cup TN_7) \times \{R\}\} \cup \{N_8 \times (TT \cup N_8T) \times \{L\}\} \cup$$
$$\{N_9 \times (TT \cup N_9T \cup TN_{10}) \times \{R\}\} \cup \{N_{10} \times (TT \cup TN_{10}) \times \{R\}\} \cup$$
$$\{N_{11} \times (TT \cup TN_{11} \cup N_{12}T) \times \{L\}\} \cup \{N_{12} \times (TT \cup N_{12}T) \times \{L\}\}$$

Definition 3. *Let $G = (N, T, S, P)$ be a D-trivial grammar and let us denote $V = N \cup T$. We say that G is a* **LD-trivial** *grammar if G does not contain productions of the form $A \to_R BC$, i.e. $P \subseteq (N \times (V \setminus \{S\})(V \setminus \{S\}) \times \{L\}) \cup (\{S\} \times T)$.*

Remark 1. Let us consider $G = (N, T, S, P)$ a LD-trivial grammar, $dTr = (dNod, dEd, dRt, dAnn)$ a complete D-tree created by G and $dNod = \{1, ..., n\}$. Then dTr is D-trivial and contains the L-edges only. It means that dTr is either an L-bush, or consists from an L-bush, and from the path leading from the (right) cross-node to the root, or it does not contain the L-bush (in this case we call it an **L-path**).

The following result characterizing LD-trivial grammars can be easily derived from the previous observation, or from Proposition 1.

Claim. Let $G = (N, T, S, P)$ be a dependency grammar. G is a LD-trivial grammar iff there exists a partition of the set of nonterminals such that $N = N_1 \cup N_2 \cup N_3$, $N_1 = \{S\}$, $N_i \cap N_j = \emptyset$, for any $1 \leq i < j \leq 3$ and holds:
$P \subseteq \{N_1 \times [T \cup ((TT \cup TN_2) \times \{L\})]\} \cup \{N_2 \times (TT \cup TN_2 \cup N_3T) \times \{L\}\} \cup \{N_3 \times (TT \cup N_3T) \times \{L\}$.

Let us observe that the dependency grammar defined in Example 1 is a LD-trivial grammar.

Using Claim 3 (or directly, as in [2]) one can prove the following result characterizing the form of DR-trees created by a LD-trivial grammar.

Claim. Let $Tr = (Nod, Ed, Rt, Ann)$ be a complete DR-tree created by a LD-trivial grammar $G = (N, T, S, P)$. The following properties hold:
a) Let $n_1 = (i, j) \in Nod$ be a node in Tr. Let Tr_{n_1} be the covering subtree of n_1 in Tr. Then $(k, l) \in Tr_{n_1}$ implies $i \leq k$.
b) Denote $k = \max\{i \mid \exists j > 1, (i, j) \in Nod\}$. If $(k, l) \in Nod$ then $(k, j) \in Nod$, for any $1 \leq j \leq l$.
c) Denote $l = \max\{j \mid (k, j) \in Nod\}$. Then for any node $(i, j) \in Nod$, with $i < k$, $j > 1$ implies $j = k - i + l$.
d) $Rt = (1, k + l - 1)$.
e) Denote $m = \max\{i \mid \exists j \in Nat, (i, j) \in Nod\}$. Then $m = k + l - 1$.
f) If Tr contains m leaves, then the longest path in Tr has exactly m nodes.

We say that a DR-tree created by a LD-trivial grammar is a **LD-trivial DR-tree**. Using notations from Claim 3, we say that a LD-trivial DR-tree Tr is a **DRL-bush** if $k = 1$ and a **DRL-path** if $l = 2$. Let us note that these notations are derived from the shape of corresponding D-trees.

To prove the main results of this paper we will consider two types of languages of a particular kind. Let $n \in Nat$ be a natural number, $V = \{b_1, \ldots, b_n\}$ be an alphabet and $l \notin V$ be a distinct symbol. Denote by $L_{lb_1 \ldots b_n}$ and $L^{total}_{lb_1 \ldots b_n}$ two languages over $V \cup \{l\}$ such that $L_{lb_1 \ldots b_n} = \{l(b_1)^m \ldots (b_n)^m \mid m \in Nat_0\}$ and $L^{total}_{lb_1 \ldots b_n} = \{lw \in V^+ \mid |w|_{b_1} = \ldots = |w|_{b_n}\}$, where $|w|_b$ denotes the number of occurrences of the symbol b in the string w. Note that the grammar G in Example 1 generates the language L^{total}_{labc}.

We will show the following result which will allow us to work with LD-trivial grammars instead of D-trivial grammars.

Proposition 2. *Let $i, n \in Nat$, $V = \{b_1, \ldots, b_n\}$ be an alphabet, $l \notin V$ be a distinct symbol, L be a language over $V \cup \{l\}$ such that $L_{lb_1 \ldots b_n} \subseteq L \subseteq L^{total}_{lb_1 \ldots b_n}$ and G be a D-trivial grammar such that DR-$L(G, i) = L$. Then, there exists a LD-trivial grammar G'' such that DR-$L(G'', i) = L$.*

Proof. Let us consider $G = (N, T, S, P)$ a D-trivial grammar such that $L_{lb_1 \ldots b_n} \subseteq$ DR-$L(G, i) \subseteq L^{total}_{lb_1 \ldots b_n}$. From the fact that all sentences generated by G should begin with l, it results that any (D-trivial) D-tree generated by G does not contain the R-bush. In any such tree there is at most one path leading from a leave trough the R-edges only (let us call it R-path). Let us consider such a D-trivial D-tree dTr with an R-path. We can see that there is exactly one LD-trivial D-tree dTL such that it has the same set of nodes as dTr, and the same L-bush as dTr and the same annotation function as dTr. We say that dTL is L-transformed from dTr.

Considering the previous claims, remarks and Proposition 1 it is not hard to see, that it is possible to construct an LD-grammar G'' such that DR-$L(G'', i) =$ DR-$L(G, i)$ for any natural i, where the D-trees generated by G'' are the L-transformations from D-trees generated by G. □

We introduce the main classes of languages which we address in our paper. We denote by:

- *DR-$\mathcal{L}$* (*tDR-$\mathcal{L}$*, *tlDR-$\mathcal{L}$*) the class of languages generated by all dependency grammars (respectively, D-trivial and LD-trivial grammars) through the set of DR-trees;
- *DR-$\mathcal{L}(i)$* (*tDR-$\mathcal{L}(i)$*, *tlDR-$\mathcal{L}(i)$*) the class of languages generated by all dependency grammars (respectively, D-trivial and LD-trivial grammars) through the set of DR-trees with at most i gaps.

Proposition 3. *Let $i \in Nat_0$ be a natural number, $V = \{b_1, \ldots, b_{2i+1}\}$ be an alphabet and $l \notin V$ be a distinct symbol. Then there exists a language L over $V \cup \{l\}$ such that $L_{lb_1 \ldots b_{2i+1}} \subseteq L \subseteq L^{total}_{lb_1 \ldots b_{2i+1}}$ and $L \in tlDR - \mathcal{L}(i)$.*

Proof. Consider $G = (N, V \cup \{l\}, B_{2i+1}, P)$ a LD-trivial grammar such that $N = \{B_j \mid j \in [2i+1]\}$, $P = \{B_{j+1} \to B_j b_j \mid j \in [2i]\} \cup \{B_1 \to B_{2i+1} b_{2i+1}, B_{2i+1} \to l\}$. We take L = DR-L$(G, i)$, hence $L \in tlDR - \mathcal{L}(i)$.

It is easy to observe that $L \subseteq L^{total}_{lb_1 \ldots b_{2i+1}}$.

Let $m \in Nat_0$ be a natural number and $Tr_m = (Nod_m, Ed_m, Rt_m, Ann_m)$ be a D-trivial left Dr-tree such that

- $Nod_m = \{(1, t+1) \mid t \in [(2i+1)m+1]\} \cup \{(s, 1) \mid s \in [(2i+1)m+1]\}$.
-

$$Ed_m(s,t) = \begin{cases} (1, t+1), & \text{if } s = 1; \\ (x + (2i+1)(y-1) + 2, 1) & \text{if } s = (x-1)m + y + 1, t = 1, \\ & x = 2(z-1)+1, z \in [i+1], \\ & y \in [m]; \\ (x + (2i+1)(m-y) + 2, 1) & \text{if } s = (x-1)m + y + 1, t = 1, \\ & x = 2z, z \in [i+1], y \in [m], \end{cases}$$

for any $(s, t) \in Nod_m$.

- $Rt_m = (1, (2i+1)m + 2)$;
-

$$Ann_m(s,t) = \begin{cases} l, & \text{if } s = 1, t = 1; \\ B_{2i+1} & \text{if } s = 1, t = 2; \\ B_x & \text{if } s = 1, t = x + (2i+1)(y-1) + 2, x \in [2i+1], \\ & y \in [m]; \\ b_x & \text{if } s = (x-1)m + y + 1, t = 1, x \in [2i+1], y \in [m], \end{cases}$$

for any $(s, t) \in Nod_m$.

We can count the number of gaps of any node (s, t) in Tr_m in the following way:

$$DR\text{-}Ng((s,t), Tr_m) = \begin{cases} z, & \text{if } s = 1, t = 2z+2 \text{ or } t = 2z+3, z \in [i]; \\ i, & \text{if } s = 1, 2i+4 \leq t \leq (2i+1)(m-1)+3; \\ i - z, & \text{ifs} = 1, t = (2i+1)(m-1) + 2z + 2 \text{ or} \\ & t = (2i+1)(m-1) + 2z + 3, z \in [i-1]; \\ 0 & \text{otherwise.} \end{cases}$$

We have that $DR\text{-}Ng(Tr_m) = \max\{DR\text{-}Ng(no, Tr_m) \mid no \in Nod_m\} = i$, which proves that Tr_m is a LD-trivial DR-tree with at most i gaps, created by G. Thus $l(b_1)^m \dots (b_n)^m = s(Tr_m) \in L$, for any $m \in Nat \cup \{0\}$, hence $L_{lb_1 \dots b_{2i+1}} \subseteq L$. □

Lemma 1. (pumping lemma) *Let $L \in tlDR\text{-}\mathcal{L}(i)$ be a language. Then, there exist two natural constants $p, r \in Nat$ such that for any sentence $w \in L$ with $|w| > p$, there exists a decomposition of w in $w = \alpha_1 a_1 \alpha_2 \dots \alpha_r a_r \alpha_{r+1}$ such that the following conditions hold:*

i) $|a_h| > 0$*, for any* $h \in [r]$*;*
ii) $|a_1 \dots a_r| < p$*;*
iii) $\alpha_1 (a_1)^j \alpha_2 \dots \alpha_r (a_r)^j \alpha_{r+1} \in L$*, for any* $j \in Nat_0$*.*
iv) If $r > i+1$ *there are at most* i *distinct indices* h *such that* $1 < h < r+1$ *and* $|\alpha_h| > p$*.*

Proof. Let $G = (N, T, S, P)$ be a LD-trivial grammar such that $L = DR\text{-}L(G,i)$. Let n be the number of nonterminals in G. Denote $p = n+1$. Let us consider a sentence $w \in L$ with $|w| > p$ and a LD-trivial DR-tree Tr created by G such that $s(Tr) = w$. Since Tr has exactly $|w|$ leaves, from Claim 3 f), we have that there exists at least a path $(n_1, n_2, \dots, n_{|w|})$ in Tr (the longest path in Tr). Since $|w| > n+1$, nodes on this path are annotated by more than n nonterminals, thus in the sequence $Ann(n_2), \dots, Ann(n_{|w|})$ there exist at least two equal nonterminals. Let us consider the first two equal nonterminals in this sequence, i.e. let us consider two indices $1 < s < t \le |w|$ such that $Ann(n_s) = Ann(n_t)$ and all nonterminals in the sequence $Ann(n_2), \dots, Ann(n_{t-1})$ are distinct.

In the following, we will use for Tr notations from Claim 3. There are three possible cases:
1. $1 < s < t \le l$. We consider the sequence of nodes $n_t = (k,t), n_{t-1} = (k, t-1), \dots, n_{s+1} = (k, s+1)$. Each of these nodes has a left dominated daughter, which is a leaf. Consider this sequence of leaves, $(no_1, 1), \dots, (no_{t-s}, 1)$ such that $Ed(no_i, 1) = n_{s+i}$, for any $1 \le i \le t-s$. We denote by a_h, with h from 1 to a certain r, a sequence of terminals $Ann(no_x, 1)Ann(no_{x+1}, 1) \dots Ann(no_{x+y}, 1)$, where $no_x, no_{x+1}, \dots, no_y$ is a maximal sequence of consecutive natural numbers. Let $w = \alpha_1 a_1 \alpha_2 \dots \alpha_r a_r \alpha_{r+1}$. We have $|a_1 \dots a_r| = r$. Since the path from n_{s+1} to n_t has at least one node and at most n nodes (as all nonterminals on this path are distinct), we obtain $r < p$, which, together with $|a_h| > 0$, for any $h \in [r]$ fulfills conditions i) and ii) of the pumping lemma. We may replace the covering subtree Tr_{n_t} with the covering subtree Tr_{n_s} (preserving the completeness under the transformation) and the resulted DR-tree Tr_0 is still a complete DR-tree created by G. Obviously the transformation will not increase the number of gaps. It results $s(Tr_0) = \alpha_1 \alpha_2 \dots \alpha_{r+1} \in L$, which proves condition iii) of the lemma for $j = 0$. For $j = 1$, $\alpha_1 a_1 \alpha_2 \dots \alpha_r a_r \alpha_{r+1} = w$, which obviously belongs to L. Moreover, we may replace the covering subtree Tr_{n_s} with the covering subtree Tr_{n_t} in such a way that new introduced leaves corresponding to a string a_h, $h \in [r]$ will stick immediately after the leaves from the initial covering subtree Tr_{n_s} corresponding to the same sequence a_h. The resulted DR-tree Tr_2 is still a complete DR-tree created by G. Again the transformation

will not increase the number of gaps. We can repeat this transformation for an unlimited number of times, obtaining in this way an infinite sequence Tr_j, with $j \geq 2$, of complete DR-trees created by G with at most i gaps. It results that $s(Tr_j) = \alpha_1(a_1)^j\alpha_2 \dots \alpha_r(a_r)^j\alpha_{r+1} \in L$, for any $j \geq 2$ which completes condition iii) of the lemma.

Finally, if $r > i + 1$, suppose towards a contradiction that there exist $i + 1$ distinct indices h_j such that $1 < h_j < r + 1$ and $|\alpha_{h_j}| > p$, for all $j \in [i + 1]$. We observe that none of the gaps induced by α_{h_j}, with $j \in [i+1]$ in the coverage of n_t in Tr can be fulfilled by the coverage of n_s in Tr, since the latest one has at most p elements. It follows that $DR\text{-}Ng(n_t, Tr) \geq i + 1$ which is a contradiction with the assumption that $Tr \in T(G, i)$. This means that also the condition iv) of the lemma is true.

2. $1 < s < l < t$. If we replace the covering subtree Tr_{n_s} with the covering subtree Tr_{n_t} in a similar way as above, we obtain a DR-tree created by G, but which is not a LD-trivial DR-tree. This contradicts the fact that G is a LD-trivial grammar, which creates only LD-trivial DR-trees. Thus, this case is not possible under the pumping lemma's assumptions.

3. $l \leq s < t \leq |w|$. We proceed in a similar way as for the first case, by taking $r = 1$. Condition iv) of the lemma does not apply in this case. □

Proposition 4. *Let $i, k \in Nat_0$ be two natural numbers such that $i < k$, $V = \{b_1, \dots, b_{2k+1}\}$ be an alphabet, $l \notin V$ be a distinct symbol and L be a language over $V \cup \{l\}$ such that $L_{lb_1 \dots b_{2k+1}} \subseteq L \subseteq L^{total}_{lb_1 \dots b_{2k+1}}$. Then $L \notin tlDR\text{-}\mathcal{L}(i)$.*

Proof. Suppose towards a contradiction that $L \in tlDR - \mathcal{L}(i)$. It follows that the pumping lemma holds for L. Consider p and r the two constants from the pumping lemma, a natural number n such that $n > 2p$ and the sentence $w = l(b_1)^n \dots (b_{2k+1})^n$. We have $|w| = (2k+1)n + 1 > p$, hence, from the pumping lemma, it should exists a decomposition of w in $w = \alpha_1 a_1 \alpha_2 \dots \alpha_r a_r \alpha_{r+1}$ such that conditions i)-iv) of the lemma hold. Take $j = 0$ in condition iii). It results that $w_0 = \alpha_1 a_1 \alpha_2 \dots \alpha_r a_r \alpha_{r+1} \in L$. Since $L \subseteq L^{total}_{lb_1 \dots b_{2k+1}}$, it follows that $|w_0|_{b_1} = \dots = |w_0|_{b_{2k+1}}$ and further that also $|a_1 \dots a_r|_{b_1} = \dots = |a_1 \dots a_r|_{b_{2k+1}}$. Since $|a_1 \dots a_r| < p$ and $n > 2p$, it results that a_h for some $h \in [r]$ cannot include more than two distinct symbols from w, hence $r \geq k + 1 > i + 1$ and there are at least $k > i$ distinct indices h such that $1 < h < r + 1$ and $|\alpha_h| > p$, which yields a contradiction with condition iv) from the pumping lemma. It results that $L \notin tlDR\text{-}\mathcal{L}(i)$. □

Corollary 1. *Let $i, k \in Nat_0$ be two natural numbers such that $i < k$, $V = \{b_1, \dots, b_{2k+1}\}$ be an alphabet, $l \notin V$ be a distinct symbol and L be a language over $V \cup \{l\}$ such that $L_{lb_1 \dots b_{2k+1}} \subseteq L \subseteq L^{total}_{lb_1 \dots b_{2k+1}}$. Then $L \notin tDR\text{-}\mathcal{L}(i)$.*

Proof. Suppose towards a contradiction that $L \in tDR - \mathcal{L}(i)$. It follows that there is a D-trivial grammar G such that $L = DR - L(G, i)$. From Proposition 3, it results that there is a LD-trivial grammar G' such that $DR - L(G', i) = DR - L(G, i) = L$. We obtain that $L \in tlDR - \mathcal{L}(i)$, which is a contradiction with Proposition 4. □

Corollary 2. *$tlDR\text{-}\mathcal{L}(k) \setminus tDR\text{-}\mathcal{L}(i) \neq \emptyset$, for any $i, k \in Nat_0$, with $i < k$.*

Proof. Let $V = \{b_1, \ldots, b_{2k+1}\}$ be an alphabet and $l \notin V$ be a distinct symbol. From Proposition 3, we have that there exists a language L over $V \cup \{l\}$ such that $L_{lb_1 \ldots b_{2k+1}} \subseteq L \subseteq L^{total}_{lb_1 \ldots b_{2k+1}}$ and $L \in tlDR\text{-}\mathcal{L}(k)$. From $i < k$ and from Corollary 1, we have that $L \notin tDR\text{-}\mathcal{L}(i)$. Thus $L \in tlDR\text{-}\mathcal{L}(k) \setminus tDR\text{-}\mathcal{L}(i)$. □

Proposition 5. *$tlDR\text{-}\mathcal{L}(i) \setminus tDR\text{-}\mathcal{L}(k) \neq \emptyset$, for any $i, k \in Nat$, with $i < k$.*

Proof. Let $V = \{b_1, \ldots, b_{2i+3}\}$ be an alphabet and $l \notin V$ be a distinct symbol. One can prove in a similar way as we did for Proposition 2, that there exists a language L over $V \cup \{l\}$ such that $\{l(b_1)^m \ldots (b_{2i})^m (b_{2i+1} b_{2i+2} b_{2i+3})^m \mid m \in Nat_0\} \subseteq L \subseteq L^{total}_{lb_1 \ldots b_{2i+3}}$ and $L_1 \in tlDR\text{-}\mathcal{L}(i)$. A similar kind of LD-trivial grammar should be defined and only the LD-trivial DR-tree corresponding to a sentence $l(b_1)^m \ldots (b_{2i})^m (b_{2i+1} b_{2i+2} b_{2i+3})^m$, $m \in Nat_0$, has a different form, but a maximal number of i gaps.

Suppose to a contradiction that $L \in tDR\text{-}\mathcal{L}(k)$. It means that there exists a D-trivial grammar G such that $L = DR\text{-}L(G,k)$. From Proposition 2, it results that there is a LD-trivial grammar G' such that $DR\text{-}L(G', k) = DR\text{-}L(G, k) = L$. Let $m \in Nat$ be a natural number. Since $w_m = l(b_1)^m \ldots (b_{2i})^m (b_{2i+1} b_{2i+2} b_{2i+3})^m \in L$, there exists a LD-trivial DR-tree Tr_m such that $s(Tr_m) = w_m$. We can suppose - without any loss of generality - that Tr_m is a L-bush[1].

We disregard the manner in which leaves are distributed in Tr_m and keeping the spine of Tr_m we build, bottom-up, a L-bush Tr'_m in the following way: We consider all nodes on the spine of Tr_m, bottom-up, from $(1, 3)$ to Rt (the leaf $(1, 1)$ will remain in the same position).

For any node of the spine, we arrange its right daughter (which is a leaf) annotated with a symbol b_x in a sequence bounded to the left by the leaf $((x - 1)m + 2, 1)$ and to the right by the leaf $(xm + 1, 1)$.

The leaves are arranged consecutively on a first-met-first-arranged base, left to right for leaves annotated with a symbol b_{2z-1}, $z \in [i + 2]$ and right to left for leaves annotated with a symbol b_{2z}, $z \in [i + 1]$.

After the L-bush Tr'_m is completed, we observe that:

i) $s(Tr'_m) = l(b_1)^m \ldots (b_{2i+3})^m$;

ii) $Tr'_m \in T(G)$; iii) $Tr'_m \in T(G, i + 1)$ (at any moment we have at most $i + 2$ islands of natural numbers in the coverage of any node in Tr'_m, thus Tr'_m cannot have more than $i + 1$ gaps).

Since $i < k$, it results that $Tr'_m \in T(G, k)$, hence $l(b_1)^m \ldots (b_{2i+3})^m \in DR\text{-}L(G,k)$. We obtain that $\{l(b_1)^m \ldots (b_{2i+3})^m \mid m \in Nat_0\} \subseteq L \subseteq L^{total}_{lb_1 \ldots b_{2i+3}}$. Using this last statement, the fact that $i < j$ and Proposition 4, we have that $L \notin tlDR\text{-}\mathcal{L}(i)$, which is a contradiction with the initial assumption.

It follows that $L \in tlDR\text{-}\mathcal{L}(i) \setminus tDR\text{-}\mathcal{L}(k)$. □

[1] Tr_m may include only a bounded "path" prefix, since otherwise we would be able to "pump" in w_m only a part of the symbols $b_1, \ldots, b_{2i+3}$, keeping the same number of gaps for the DR-tree, hence either $L \neq DR\text{-}L(G', k)$ or $L \not\subseteq L^{total}_{lb_1 \ldots b_{2i+3}}$. In this case, the rest of proof would be done for a natural number m greater than the longest "path" prefix in such a LD-trivial Dr-tree Tr_m. Still, the final conclusion would hold.

Theorem 1. *The classes of languages, generated by D-trivial grammars with a bounded number of gaps, form an infinite sequence:*

$$tDR\text{-}\mathcal{L}(1), tDR\text{-}\mathcal{L}(2), \ldots, tDR\text{-}\mathcal{L}(n), \ldots$$

such that any two different classes in this sequence cannot be compared.

Proof. It results from the fact that $tlDR\text{-}\mathcal{L}(i) \subseteq tDR\text{-}\mathcal{L}(i)$, for any $i \in Nat_0$, from Corollary 2 and Proposition 5. □

4 Other properties

We denote with REG, LIN, CF and CS the classes of regular, linear, context-free, and respectively, context-sensitive languages.

Let $V = \{a_1, \ldots, a_n\}$ be an alphabet. We define the Parikh mapping $\Phi_V : V^* \to Nat_0^n$ by $\Phi_V(x) = (|x|_{a_1}, \ldots, |x|_{a_2})$, for any $x \in V^*$. We extend the mapping in a natural way to languages. Two languages $L_1, L_2 \in V^*$ such that $\Phi_V(L_1) = \Phi_V(L_2)$ are said to be *letter equivalent.* Consider the operations of componentwise addition and multiplication by a constant over the set of natural vectors of a given dimension. A subset M of Nat_0^n is said to be *linear* if there exist the vectors $v_0, v_1, \ldots, v_m$, $m \geq 0$, such that $M = \{v_0 + \sum_{i=1}^{m} p_i v_i \mid p_i \in Nat_0, 1 \geq i \geq m\}$. A finite union of linear sets is called a *semilinear* set. A language $L \subseteq V$ is called *semilinear* if $\Phi_V(L)$ is a semilinear set.

Proposition 6. *Let $L \in$ DR-L(i), for some $i \in Nat$. Then L is semilinear.*

Proof. Let G be a dependency grammar such that $L =$ *DR-L(G,i)*. Denote $L_0 = DR\text{-}L(G, 0)$. From Claim 2, we know that any DR-tree $Tr \in T(G, i)$ is DR-equivalent with a projective DR-tree $Tr_0 \in T(G, 0)$. Then $\Phi_V(s(Tr)) = \Phi_V(s(Tr_0))$. Since $L_0 \subseteq L$ (Claim 2), it results that $\Phi_V(L) = \Phi_V(L_0)$, i.e. L is letter equivalent to L_0. But L_0 is a context-free language (see e.g. [4]) and from the Parikh Theorem it is semilinear. It results that also L is semilinear.

□

Proposition 7. *i) $tlDR\text{-}\mathcal{L}(0) = REG$;*
ii) $LIN \subseteq tDR\text{-}\mathcal{L}(0)$;
iii) $REG \subset tlDR\text{-}\mathcal{L}(i)$, for any $i \in Nat$ (all inclusions are strict);
iv) $tDR\text{-}\mathcal{L}(i) \subset CS$, for any $i \in Nat_0$ (all inclusions are strict).

Proof. i) If $G = (N, T, S, P)$ is a regular grammar, then we can define a LD-trivial grammar $G' = (N, T, S, P')$ with $P' = \{(A, aB, L) \mid (A, aB) \in P\} \cup \{(A, a) \mid (A, a) \in P\}$. It results $DR\text{-}L(G', 0) = L(G)$. Let $G = (N, T, S, P)$ be a LD-trivial grammar and $N = N_1 \cup N_2 \cup N_3$ be the partition of the set of nonterminals defined in Claim 3. We can define a non-deterministic finite automaton $M = (Q, T, S, F, \delta)$ such that $Q = N \cup \{B^A \mid A \in N_2, B \in N_3\} \cup \{qF\} \cup \{p_b \mid (A, ab, L) \in P, A \in N_1 \cup N_2\} \cup \{r^B_{A,b} \mid (A, ab, L) \in P, B \in N_2, A \in$

$N_3\}$, $F = \{q_f\}$ and the transition function δ is defined by:

$$\delta(q,a) = \begin{cases} \{qF \mid (q,a) \in P\} \cup \{q' \mid (q,aq',L) \in P\} \cup & \\ \quad \cup \{p_b \mid (q,ab,L) \in P\} & \text{if } q = S, \\ \{q' \mid (q,aq',L) \in P\} \cup \{p_b \mid (q,ab,L) \in P\} \cup & \\ \quad \cup \{r^q_{A,b} \mid (A,ab,L) \in P, A \in N_3\} & \text{if } q \in N_2, \\ \{qF\} & \text{if } q = p_a, \\ \{B^A\} & \text{if } q = r^A_{B,a}, \\ \{C^A \mid (C,Ba,L) \in P, C \in N_3\} \cup \{qF \mid (A,Ba,L) \in P\} & \text{if } q = B^A, \\ \emptyset & \text{otherwise.} \end{cases}$$

We have *L(M)=DR-L(G,0)*.

ii) Let $G = (N,T,S,P)$ be a linear grammar. We may suppose that G contains rules of the form: $A \rightarrow aB$, $A \rightarrow Ba$, $A =\rightarrow ab$, $S \rightarrow a$, where $A, B \in N$, with $B \neq S$, and $a, b \in T$. We define a D-trivial grammar $G' = (N,T,S,P')$ with $P' = \{(A,aB,R) \mid (A,aB) \in P\} \cup \{(A,Ba,L) \mid (A,aB) \in P\} \cup \{(A,ab,L) \mid (A,ab) \in P\} \cup \{(A,a) \mid (A,a) \in P\}$. It results $DR\text{-}L(G',0) = L(G)$.

iii) Like for i), if G is a regular grammar, we can define a LD-trivial grammar $G' = (N,T,S,P')$ with $P' = \{(A,aB,L) \mid (A,aB) \in P\} \cup \{(A,a) \mid (A,a) \in P\}$ and we have $DR\text{-}L(G',0) = L(G)$. But G creates only projective DR-trees ($T(G',i) = T(G',0)$, for any $i \in Nat_0$), hence $DR\text{-}L(G',i) = DR\text{-}L(G',0) = L(G)$, for any $i \in Nat_0$. If $i \in Nat$, from the item i) of this proposition and from Corollary 2, we obtain the strictness of the inclusion.

iv) Let $L \in tDR\text{-}\mathcal{L}(i)$, $i \in Nat_0$ be a language, $G = (N,T,S,P)$ be a D-trivial grammar such that $L =$ *DR-L(G,i)*. We can define a linear bounded automaton $M = (Q,T,\Sigma,\$,\#,q0,F,\delta)$ with a work space of $n+2$ cells where n is the length of the input word. \$ and # are the left (respectively, right) side markers (none of these two tape symbols is in T). $q0$ is the initial state of the automaton M, while the set of final states is defined by $F = \{qF\}$. The set of states Q, the tape's alphabet and the transition function δ are defined according to the method described in the following. We define an instant configuration of M as a triple (q,α,i), where $q \in Q$, $\alpha \in \Sigma^+$, $|\alpha| = n+2$, for some $n \in Nat$ and $1 \leq i \leq n+2$. We will use instant configurations to describe the behavior of M.

The initial configuration of M is $(q_0, \$w\#, 2)$, with $w \in T^+$, $|w| = n$. In the first step, the automaton "guesses" the horizontal position h of the start symbol in the DR-tree corresponding to w. First the application of a production of the form $S \rightarrow a$, with $a \in T$ is investigated. In this case, the symbol on the position $h = 1$ should equal a. If this is the case, the automaton rewrites a by S and checks for the rest of the tape. If there are no more other terminal symbols on the tape, M enters the final state qF and accepts the initial input word.

If the application of a (unique) production $S \rightarrow a$ fails, the automaton starts to rewrite (every time two) symbols from the tape, according to the productions from G. At this stage, two types of transitions are possible.

1. A production of the form $A \rightarrow_L xa$ (respectively $A \rightarrow_R ax$), with $x \in N \cup T$ and $a \in T$ is under investigation.

In this case, the symbol on the position h should be equal to x, if x is a terminal, or to x^k if x is a non-terminal (the variable k, with $0 \leq k \leq i$, counts

the current number of gaps, see below) and somewhere on the tape, to the right (respectively, to the left) of the position h, an unmarked terminal equal to a should exist. Then, the application of the production consists in marking the corresponding terminal symbol and rewriting the symbol on the position h with A^l, where l is modified according to the transformation occurred in the number of gaps. The value of l can increase/decrease with 1, or remain the same with k (if x is a terminal, then k is treated as 0), according to the following procedure (we consider only productions of the form $A \rightarrow_L xa$).
If the new marked symbol is not near (to the left or to the right of) an already marked symbol (or is not on the position $h+1$), then the indicator for the number of gaps will increase with 1 (a new gap is created; obviously, the indicator cannot be ever greater than i), as in:
$(q_1, \$a_1 ... a_{h-1} x^k a_{h+1} ... bac ... a_n \#, h) \overset{*}{\vdash} (q_1, \$a_1 ... a_{h-1} A^{h+1} a_{j+1} ... b\overline{a}c ... a_n \#, h)$.

If the new marked symbol is near an already marked symbol, but not to the left and to the right in the same time (or on the position $h+1$, but not near other marked symbol), then the indicator for the number of gaps will remain unchanged, as in (only the left case is considered):
$(q_1, \$a_1 ... a_{h-1} x^k a_{h+1} ... \overline{b}ac ... a_n \#, h) \overset{*}{\vdash} (q_1, \$a_1 ... a_{h-1} A^k a_{h+1} ... \overline{ba}c ... a_n \#, h)$.

If the new marked symbol is near an already marked symbol, both to the left and to the right (or on the position $h+1$, and near other marked symbol), then the indicator for the number of gaps will decrease with 1 (a gap was filled-in), as in:
$(q_1, \$a_1 ... a_{h-1} x^k a_{h+1} ... \overline{b}a\overline{c} ... a_n \#, h) \overset{*}{\vdash} (q_1, \$a_1 ... a_{h-1} A^{k-1} a_{h+1} ... \overline{bac} ... a_n \#, h)$.

2. A production of the form $A \rightarrow_L xB$ (respectively $A \rightarrow_R Bx$), with $x \in N \cup T$ and $B \in N$ is under investigation.

In this case, the symbol on the position h should be equal to x, if x is a terminal, or to x^k if x is a non-terminal. The automaton writes A^k on the position h (or A^0 if x is a terminal) and remembers the non-terminal B under the name of the current state. Consider for discussion the case $A \rightarrow_R Bx$ only (the procedure for a production of the form $A \rightarrow_L xB$ is similar, but is carried on to the right). The automaton can apply a (possibly empty) sequence of productions of the form $A \rightarrow Bb$, with $B \in N$, $b \in T$, using transitions of the form:
$(q^A, \$a_1 \ldots a_{j-1} a_j a_{j+1} \ldots a_n \#, j) \overset{*}{\vdash} (q^B, \$a_1 \ldots a_{j-1} b a_{j+1} \ldots a_n \#, j-1)$,
until an arbitrary state g^B is reached. Than, M proceeds to the same kind of rewriting as in the general case 1., starting with the current position lh instead of h and considering only productions of the form $A \rightarrow_L xa$. This phase is ran until all the terminals from 1 to $lh-1$ are marked. If in this state, the symbol on the position lh equals B^0, then M goes back to continue the analysis at the position h.

If the automaton M succeeds to rewrite all the input symbols on the tape and the symbol on the position j is equal to S^0, then M enters the final state qF and accepts the initial input word. The detailed definitions of Q and δ are left to the reader. Since all languages in *tDR*-$\mathcal{L}(i) \subseteq$ *DR*-$\mathcal{L}(i)$ are semilinear

(Proposition 6), we can easily find a non-semilinear language in $CS \setminus \text{tDR-}\mathcal{L}(i)$ (like $\{a^{2^n} \mid n \in Nat_0\}$). $\square$

5 Conclusions

The main aim of this contribution was to discuss the fact that there is a significant difference between the (non-)projectivity of DR-trees and respectively D-trees. We stressed on the fact that D-trees can hide some word-order freedom phenomena raising in the generation or the parsing of the sentence. As outcome, an infinite sequence of incomparable classes of semilinear languages bounded between the class of regular and respectively context-sensitive languages was obtained. The massive incomparability achieved trough the global restrictions only is the main novelty of this contribution. This result strengthens the results from [4], where some infinite hierarchies of classes of languages were obtained. Those hierarchies were obtained by using stronger combinations of local and global restrictions applied to free-order dependency grammars.

In the close future, we will study the same types of global word-order restrictions as here, but applied on dependency grammars without any further restrictive condition, like D-triviality. We believe that we will achieve new sequences of incomparable classes of languages. Moreover, we believe that to this aim we can use the sequence of "witness" languages, which we already used in this paper.

We will study also free-order dependency grammars with several kinds of topological restrictions in order to understand complex word-order and concurrency phenomena occurring in the syntax of natural languages. We believe that the study of free-order dependency grammars can also contribute to the understanding of concurrency phenomena, in general, as well.

Acknowledgements. The authors acknowledge Tomáš Holan for fruitful discussions.

References

1. A.Ja. Dikovskij, L.S. Modina: "Dependencies on the other side of the curtain", In: Traitement Automatique des Langues (TAL), 41(1):79-111, 2000
2. R. Gramatovici, M. Plátek: "D-trivial Dependency Grammars with Global Word-Order Restrictions", Technical Report TR-2002-16, Institute of Computational Linguistics, Faculty of Mathematics and Physics, Charles University, Prague, 2002.
3. T. Holan, V. Kuboň, K. Oliva, M. Plátek: "Two Useful Measures of Word Order Complexity", In: *Proceedings of the Coling '98 Workshop "Processing of Dependency-Based Grammars"*, A. Polguere and S. Kahane (eds.), University of Montreal, Montreal, 1998
4. M. Plátek, T. Holan, V. Kuboň: On Relax-ability of Word-Order by D-grammars., In: *Combinatorics, Computability and Logic*, C. S. Calude and M.J. Dinneen (eds.), Springer Verlag, Berlin, 2001, pp. 159-174
5. P. Sgall, E. Hajičová, J. Panevová: *The Meaning of the Sentence in Its Semantic and Pragmatic Aspects*, Dordrecht: Reidel and Prague: Academia, 1986

Representations of Ω in Number Theory: Finitude Versus Parity

Toby Ord and Tien D. Kieu

[1] Department of Philosophy, The University of Melbourne, 3010, Australia
t.ord@pgrad.unimelb.edu.au
[2] Centre for Atom Optics and Ultrafast Spectroscopy, Swinburne University of Technology, Hawthorn 3122, Australia kieu@swin.edu.au

Abstract. We present a new method for expressing Chaitin's random real, Ω, through Diophantine equations. Where Chaitin's method causes a particular quantity to express the bits of Ω by fluctuating between finite and infinite values, in our method this quantity is always finite and the bits of Ω are expressed in its fluctuations between odd and even values, allowing for some interesting developments. We then use exponential Diophantine equations to simplify this result and finally show how both methods can also be used to create polynomials which express the bits of Ω in the number of positive values they assume.

1 Recursive Enumerability, Algorithmic Randomness and Ω

One of the most startling recent developments in the theory of computation is the discovery of the number Ω, through the subfield of algorithmic information theory. Ω is a real number between 0 and 1 which was introduced by G. J. Chaitin [2] as an example of a number with two conflicting properties: it is both recursively enumerable and algorithmically random. Very roughly, this means that Ω has a simple definition and can be computed in the limit from below, yet we can determine only finitely many of its digits with certainty—for the rest we can do no better than random.

Understanding the full importance of these properties requires some familiarity with the recursive functions—commonly presented through models of computation such as Turing machines or the lambda calculus. For the purposes of algorithmic information theory, however, it is convenient to abstract some of the details from these models and consider a programming language in which the (partial) recursive functions are represented by finite binary strings.[1] These strings are just programs for a universal Turing machine (or universal lambda expression) and they take input in the form of a binary string then output another binary string or diverge (fail to halt). For convenience, we will often consider these inputs and outputs to encode tuples of positive integers.

[1] For more details see Chaitin [3].

On top of this simplified picture of computation, we impose one restriction which is necessary for the development of algorithmic information theory (and hence Ω). The set of strings that encode the recursive functions must be prefix-free. This means that no program can be an extension of another, and thus each program is said to be self-delimiting. As algorithmic information theory is intricately linked with communication as well as computation, this is quite a natural constraint—if you wish to use a permanent binary communication channel, then you need to know when the end of a message has been reached and this cannot be done if some messages are extensions of others.

There are many prefix-free sets that one could choose and many recursive mappings between these and the recursive functions. These different choices of 'programming language' lead to different values of Ω, but this does not matter much as almost all of its significant properties will remain the same regardless. However, to allow talk of Ω as a specific real number we will use the same language as Chaitin [3].

Now that we have explained what we mean by a programming language, we can give a quick overview of computability in terms of programs. A program computes a set of n-tuples if, when provided with input $\langle x_1, \dots, x_n \rangle$, it returns 1 if this is a member of the set and 0 otherwise. A program computes an infinite sequence if, when provided with input n, it returns the value of the n-th element in the sequence. A program computes a real, r, if it computes a sequence of rationals $\{r_n\}$ which converges to r and $|r - r_n| < \frac{1}{2^n}$. These sets, sequences and reals that are computed by programs are said to be recursive.

There are also many sets, sequences and reals that cannot be computed, but can be approximated in an important way. A program semi-computes a set of n-tuples if, when provided with input $\langle x_1, \dots, x_n \rangle$, it returns 1 if this is a member of the set and diverges otherwise. A program semi-computes an infinite sequence of bits if, when provided with input n, it returns 1 if the n-th bit in the sequence is 1 and diverges otherwise. A program semi-computes a real, r, if, when provided with input n, it computes a rational number, r_n, where $\{r_n\}$ converges to r from below. These sets, infinite bitstrings and reals that are semi-computed by programs are said to be recursively enumerable or r.e.

There is an important point that needs to be made concerning reals and their representations. Each real number between 0 and 1 has a binary expansion: a binary point followed by an infinite sequence of bits that represents the real.[2] Throughout this paper, we shall be making considerable use of the binary expansions of real numbers so it is important to point out an oddity in the definitions above: a real is recursive if and only if its binary expansion is recursive, but a real may be r.e. even if its binary expansion is not r.e. We shall thus take care to distinguish the weaker property of being an *r.e. real* from the stronger one of

[2] For numbers that can be expressed with a representation ending in an infinite string of 0's, there is another representation ending in an infinite sequence of 1's, but we shall remove this ambiguity by only using representations with an infinite number of 0's. This will not affect the important reals in this paper, Ω and τ, as they are irrational and thus have unique representations regardless.

being a *real whose binary expansion is r.e.*

An example of a real that is r.e. but not recursive is τ: the real number between 0 and 1, whose k-th digit is 1 if the k-th program (in the usual lexical ordering of finite bitstrings) halts when given the empty string as input and 0 if the k-th program diverges. Equivalently:

$$\tau = \sum_{p_n \text{ halts}} 2^{-n} \tag{1}$$

τ is an r.e. real because there is a computable sequence of rationals $\{\tau_i\}$, where

$$\tau_i = \sum_{\substack{n \leq i \\ p_n \text{ halts in } \leq i \text{ steps}}} 2^{-n} \tag{2}$$

such that $\{\tau_i\}$ converges to τ from below.

Furthermore, it is clear that the binary representation of τ is also r.e. because there is a program that simulates the k-th program, halting if and only if it does. This program is a slightly modified universal program that first determines the bits of the k-th program and then simulates it.

τ is not recursive, however, because if a program could compute it to arbitrary accuracy, it would determine whether each program halts or not when given the empty string as input. This is known as the *blank tape problem* and is easily shown to be equivalent to the more general *halting problem*—'does a given program halt on a given input?'. The halting problem is fundamental to the theory of computation and is the most famous problem that cannot be recursively solved. τ merely encodes the information necessary to solve the halting problem into the binary expansion of a real number and thus provides a very simple example of a non-computable real to which we can contrast the more exotic properties possessed by Ω.

Ω encodes the halting problem in a more subtle way: it is the *halting probability.* We could, theoretically, generate a random program one bit at a time, by flipping a fair coin and writing down a 1 when it comes up heads and a 0 for tails—stopping if we reach a valid program. The chance of generating any given n bit program is therefore $\frac{1}{2^n}$. Ω is the chance that this method of random program construction generates a program that halts. Letting $|p|$ represent the size of p in bits, we can also express Ω as

$$\Omega = \sum_{p \text{ halts}} 2^{-|p|} \tag{3}$$

As was the case for τ, there is a computable sequence of rationals $\{\Omega_i\}$, where

$$\Omega_i = \sum_{\substack{|p| \leq i \\ p \text{ halts in } \leq i \text{ steps}}} 2^{-|p|} \tag{4}$$

which converges to Ω from below, showing it to be an r.e. real. However, we shall see shortly that the binary representation of Ω is *not* r.e.

A real is said to be *algorithmically random* [3] if and only if the 'algorithmic complexity' of each n-bit initial segment of its binary expansion becomes and remains arbitrarily greater than n.[3] In other words a real, r, is algorithmically random if and only if any program that has access to outside advice in the form of binary messages requires more than n bits of advice to compute the first n bits of r's binary expansion (for all values of n above some threshold).[4] Thus a random real is one for which only finitely many prefixes of its binary expansion can be compressed.

It is easy to see that a random real cannot have an r.e. binary expansion. Let x be an arbitrary real whose binary expansion is r.e. By definition, there must be a program, p_x, that takes a positive integer, k, and halts if and only if the k-th bit of x is 1. To determine n bits of x, we just need to know how many of these n values of k make p_x halt. We could then simply run p_x on all the values of k and stop when this many have halted, knowing that no more will halt and thus determining the n bits of x. Since all positive integers less than n can be encoded in $\log n$ bits (rounding up), we only need to send a message of about $(\log n + \log\log n)$ bits. In this manner, any prefix of x can be significantly compressed, so x cannot be random.

Because of this, we can see that τ too is not random. However, Chaitin [3] has proven that Ω *is* random and so cannot be compressed in this manner.[5] For sufficiently high values of n, n bits of Ω provide n bits of algorithmically incompressible information.

In addition to recursive incompressibility, random reals are also characterised by *recursive unpredictability* [3]. Consider a 'predictive' program that takes a finite initial segment of an infinite bitstring and returns a value indicating either 'the next bit is 1', 'the next bit is 0' or 'no prediction'. If any such program is run on all finite prefixes of the binary expansion of a random real and makes an infinite amount of predictions, the limiting relative frequency of correct predictions approaches $\frac{1}{2}$. In other words when any program is used to predict infinitely many bits of a random real, such as Ω, it does no better than random—even with information about all the prior bits.

[3] This is only one of four common definitions of algorithmic randomness, however, all have been shown to be equivalent.

[4] The reason that slightly more than n bits of advice are needed is because in algorithmic information theory the advice comes in self-delimiting messages (which are actually programs that generate the advice—like self-extracting archives) and in order to be self-delimiting, these messages need slightly more bits than they would otherwise. In general, an n bit string requires about $(n + \log n)$ bits. Chaitin [3] provides further details.

[5] Indeed, it has since been shown through the work of R. Solovay, C. S. Calude, P. Hertling, B. Khoussainov, Y. Wang and T. A. Slaman that the only r.e. random reals are Ω's for different programming languages. See Calude [1] for more details.

The power of this unpredictability can be seen when compare the predictability of τ. In this case, the predictive program can easily predict an infinite amount of bits with no errors. This is because infinitely many bits of τ are 'easy' to compute. For example, consider the halting behaviour of Turing machines: there are infinitely many Turing machines which have no loops in their transition graphs and thus cannot possibly diverge. When the predictive program is asked to predict the n-th bit of τ, it can just check to see if the n-th program corresponds to such a machine, returning 'the next bit is 1' if it does and 'no prediction' otherwise.[6]

With its inherent incompressibility and unpredictability, Ω really does go beyond the type of uncomputability present in a more typical non-recursive real such as τ. However, its contrasting property of being an r.e. real makes Ω seem to be just beyond our reach. In the next section, we will introduce Diophantine equations and show how these can be used to bring uncomputability into the more classical field of number theory. Then, in Section 3, we will show two ways of using Diophantine equations to bring Ω and randomness to number theory—Chaitin's original method and our new technique.

2 Diophantine Equations and Hilbert's Tenth Problem

A Diophantine equation is a polynomial equation in which all of the coefficients and variables take only positive integer values. Many natural phenomena with discrete quantities are modelled well by Diophantine equations and they occur frequently in number theory. It is often convenient to express a Diophantine equation with all terms on the left hand side:

$$D(x_1, \ldots, x_m) = 0 \tag{5}$$

Here D is a polynomial of $x_1, \ldots, x_m$ in which the coefficients can take both positive and negative integer values.

The number of solutions for a Diophantine equation varies widely. For example, $3x_1 + 6 = 0$ has one solution, while $x_1x_2 - 2 = 0$ has two and $x_1x_2 - x_2 = 0$ has infinitely many. Some however, such as $2 - 3x_1 = 0$, have no solutions at all. There are many different methods for deciding whether Diophantine equations of certain forms have solutions and determining what these solutions are, but there has been a great desire for a single method that takes an arbitrary Diophantine equation and determines whether or not it has solutions. In 1900, David Hilbert [5] gave the problem of finding such a method as the tenth in his famous list of important problems to be addressed by mathematicians in the 20th Century. Since then, the task of finding this method has become known simply as Hilbert's Tenth Problem.

[6] From the definition of binary programs in algorithmic information theory, there must be a recursive mapping between programs and Turing machines (or any such model).

Another area of research concerns families of Diophantine equations. A family of Diophantine equations is a relation of the form:

$$D(a_1, \ldots, a_n, x_1, \ldots, x_m) = 0 \tag{6}$$

in which we distinguish between two types of variable. The variables $x_1, \ldots, x_m$ are called unknowns, while $a_1, \ldots, a_n$ and called parameters. By assigning values to each of the parameters (and treating them as constants), we pick out an individual Diophantine equation from the family. For example, the family $a_1 - 3x_1 = 0$ consists of the equations: $1 - 3x_1 = 0$, $2 - 3x_1 = 0$, $3 - 3x_1 = 0$ and so on.

Each family of Diophantine equations is naturally associated with a certain set of n-tuples of positive integers, $\mathcal{D}$, in the following manner:

$$\langle a_1, \ldots, a_n \rangle \in \mathcal{D} \iff \exists x_1 \ldots x_m D(a_1, \ldots, a_n, x_1, \ldots, x_m) = 0 \tag{7}$$

In other words, a tuple is in the set if the equation it corresponds to has a solution. Such sets are said to be Diophantine or to have a Diophantine representation. For example, the set of all multiples of 3 is Diophantine because it is represented by the family $a_1 - 3x_1 = 0$.

Over the 1950's and 1960's, M. Davis, H. Putnam and J. Robinson established several important results regarding which sets are Diophantine. Their key result concerned a characterisation, not of Diophantine sets, but their close relation: *exponential* Diophantine sets.

A family of exponential Diophantine equations is a relation of the form:

$$D(a_1, \ldots, a_n, x_1, \ldots, x_m, 2^{x_1}, \ldots, 2^{x_m}) = 0 \tag{8}$$

where D is once again a polynomial, but now some of its variables are exponential functions of others. Davis, Putnam and Robinson [4] used this additional flexibility to show that all r.e. sets are exponential Diophantine. It had long been known that all exponential (and standard) Diophantine sets are r.e. because it is trivial to write a program that searches for a solution to a given equation and halts if and only if it finds one. Therefore, the new result meant that the exponential Diophantine sets were precisely the r.e. sets.

In 1970, Yu. Matiyasevich [6] completed the final step, proving that all exponential Diophantine sets are also Diophantine and thus that the Diophantine sets are exactly the r.e. sets—a result now known as the DPRM Theorem.

The DPRM Theorem provides an intimate link between Diophantine equations and computability, reducing the task of determining whether a set has a Diophantine representation to a matter of programming. For instance, there is a program that takes a single input k and halts if and only if the k-th bit of τ is 1. Thus, the set of positive integers that includes k if and only if the k-th program halts is an r.e. set and via the DPRM Theorem, there is a family of Diophantine equations with a parameter k, that has solutions if and only if the k-th program halts.

This family of equations provides an example of uncomputability in number theory and shows that Hilbert's Tenth Problem must be recursively undecidable because a program that finds whether arbitrary Diophantine equations have solutions could be used to determine the bits of τ and thus to solve the halting problem. Indeed, it was long known that the recursive undecidability of Hilbert's Tenth Problem would follow immediately from the DPRM Theorem and this was the main motivation for its proof—the Diophantine representations for all other r.e. sets being largely a bonus.

3 Expressing Omega Through Diophantine Equations

While the DPRM Theorem demonstrates the existence of τ and uncomputability in number theory, it also denies the possibility of finding a similar family of Diophantine equations expressing Ω and randomness. This is due to the fact discussed in Section 1 that, while Ω is an r.e. real, its sequence of bits is *not* r.e. However, the DPRM Theorem only prohibits a direct Diophantine representation of Ω and says nothing about the more subtle properties of Diophantine equations in which these bits could perhaps be encoded.

Chaitin [3] takes such an approach. While there is no program of one variable, k, that halts if and only if the k-th bit of Ω is 1, Chaitin provides a program, P, that takes two variables, k and N, and computes Ω somewhat less directly. For a given value of k, P can be thought of as making an infinite series of 'guesses' as to the value of the k-th bit of Ω—when P is run on k and N, it gives the N-th guess as to the k-th bit of Ω. What is impressive is that P gets infinitely many of these guesses right and only finitely many wrong.

How does P do this? It simply computes the sequence $\{\Omega_i\}$ discussed in Section 1 until it gets to Ω_N and then returns the k-th bit of Ω_N. Just as $\{\Omega_i\}$ forms a sequence of approximations to Ω, so the k-th bit of each $\{\Omega_i\}$ forms a sequence of approximations to the k-th bit of Ω.

Consider this k-th bit of each $\{\Omega_i\}$ as i is increased. This bit could change between 0 and 1 many times, but since $\{\Omega_i\}$ approaches Ω, it must eventually remain fixed, at which point it must have the same value as the k-th bit of Ω. Therefore, if the k-th bit of Ω is 1, the k-th bit of $\{\Omega_i\}$ must be 0 for only finitely many values of i, and so P must return 0 for finitely many values of N and 1 for infinitely many. On the other hand, if the k-th bit of Ω is 0, then the k-th bit of $\{\Omega_i\}$ must be 1 for only a finite number of values of i and P must return 1 for finitely many values of N and 0 for infinitely many. Either way, as N increases, the output of P applied to k and N limits to the k-th bit of Ω.

It may seem as though this program is computing the bits of Ω but this is not quite the case. P just computes the N-th 'guess' of the k-th bit. From the infinite sequence of such guesses, the k-th bit could be determined but P does not and cannot put the guesses together like that—it just returns one of them.

Since recursive functions are just a special type of r.e. function, we can apply the DPRM Theorem and see that there must be a family of Diophantine equations

$$\chi_1(k, N, x_1, \ldots, x_m) = 0 \tag{9}$$

that has solutions for given values of k and N if and only if P returns 1 when provided with these as input. For a given value of k, there are solutions for infinitely many values of N if and only if the k-th bit of Ω is 1.

Thus, by using a more subtle property of the family of Diophantine equations, Chaitin was able to show that algorithmic randomness occurs in number theory: as k is varied, there is simply no recursive pattern to whether this family of equations has solutions for finitely or infinitely many values of N.

By modifying Chaitin's method slightly, we can find a new way of expressing the bits of Ω through a family of Diophantine equations [7]. Consider a new program, Q, that also takes inputs k and N, and begins to compute the sequence $\{\Omega_i\}$. For each value of Ω_i, Q checks to see if it is greater than $\frac{N}{2^k}$, halting if this is so, and continuing through the sequence otherwise. Since $\{\Omega_i\}$ approaches Ω from below, we can see that $\Omega_i > \frac{N}{2^k}$ implies that $\Omega > \frac{N}{2^k}$ and conversely, if $\Omega > \frac{N}{2^k}$ there must be some value of i such that $\Omega_i > \frac{N}{2^k}$. Therefore, Q will halt on k and N if and only if $\Omega > \frac{N}{2^k}$. Alternatively, we could say that Q recursively enumerates the pairs $\langle k, N\rangle$ such that $\Omega > \frac{N}{2^k}$.

Just as we could determine the k-th bit of Ω from the number of values of N that make P return 1, so we can determine it from the number of values of N for which Q halts. In what follows, we shall refer to these quantities as as p_k and q_k respectively.

Unlike p_k, q_k is always finite. Indeed, an upper bound is easily found. Since $\Omega < 1$, only values of k and N such that $\frac{N}{2^k} < 1$ can possibly be less than Ω and thus make Q halt. Since both k and N take only values from the positive integers we also know that $\frac{N}{2^k} > 0$ and thus for a given k, there are less than 2^k values of N for which Q halts and $q_k \in \{0, 1, \ldots, 2^k - 1\}$.

From the value of q_k, it is quite easy to derive the first k bits of Ω. Firstly, note that q_k is equal to the largest value of N such that $\frac{N}{2^k} < \Omega$—unless there is no such N, in which case it equals 0. Either way, its value can be used to provide a very tight bound on the value of Ω: $\frac{q_k}{2^k} < \Omega \leq \frac{q_k+1}{2^k}$. Since Ω is irrational, we can strengthen this to $\frac{q_k}{2^k} < \Omega < \frac{q_k+1}{2^k}$, which means that the first k bits of $\frac{q_k}{2^k}$ are exactly the first k bits of Ω.

This gives some nice results connecting q_k and Ω. The first k bits of $\frac{q_k}{2^k}$ are just the bits of q_k when written with enough leading zeros to make k digits in total. Thus q_k, when written in this manner, provides the first k bits of Ω. Additionally, we can see that q_k is odd if and only if the k-th bit of Ω is 1.

Now that we know the power and flexibility of q_k, it is a simple matter to follow Chaitin in bringing these results to number theory. The function computed by Q is r.e. so, by the DPRM Theorem, there must be a family of Diophantine equations

$$\chi_2(k, N, x_1, \ldots, x_m) = 0 \tag{10}$$

that has a solution for specified values of k and N if and only if Q halts when given these values as inputs. Therefore, for a particular value of k, this equation only has solutions for values of N between 0 and $2^k - 1$ with the number of solutions, q_k, being odd if and only if the k-th bit of Ω is 1.

This new family of Diophantine equations improves upon the original one in a couple of ways. Whereas the first method expressed the bits of Ω in the fluctuations between a finite and infinite amount of values of N that give solutions, the second keeps this value finite and bounded, with the bits of Ω expressed through the more mundane property of parity. It is the fact that this quantity is always finite that leads to many of the new features of this family of Diophantine equations. p_k is infinite when the k-th bit of Ω is 1 and, since there is only one way in which it can be infinite, it can provide no more than this one bit of information. On the other hand, q_k can be odd (or even) in 2^{k-1} ways, which is enough to give $k-1$ additional bits of information, allowing the first k bits of Ω to be determined.

The fact that q_k is always finite also provides a direct reduction of the problem of determining the bits of Ω to Hilbert's Tenth Problem. To find the first k bits of Ω, one need only determine for how many values of N the new family of Diophantine equations has solutions. Since we know that there can be no solutions for values of N greater than or equal to 2^k, we could determine the first k bits of Ω from the solutions to 2^k instances of Hilbert's Tenth Problem. In fact, we can lower this number by taking advantage of the fact that if there is a solution for a given value of N then there are solutions for all lower values. All we need is to find the highest value of N for which there is a solution and we can do this with a bisection search, requiring the solution of only k instances of Hilbert's Tenth Problem.[7]

Finally, the fact that q_k is always finite allows the generalisation of these results from binary to any other base, b. If we replace all above references to 2^k with b^k we get a new program, Q_b, with its associated family of Diophantine equations. For this family, the value of q_k now gives us the first k digits of the base b expansion of Ω: it is simply the base b representation of q_k with enough leading zeroes to give k digits. The value of the k-th digit of Ω is simply q_k mod b.

Chaitin [3] did not stop with his Diophantine representation of Ω, but instead moved to exponential Diophantine equations where his result could be presented more clearly. He made this move to take advantage of the theorem that all r.e. sets have *singlefold* exponential Diophantine representations, where a representation is singlefold if each equation in the family has at most one solution.

We can denote the singlefold family of exponential Diophantine equations for the program P by

$$\chi_1^e(k, N, x_1, \ldots, x_{m'}) = 0 \tag{11}$$

For a given k, this equation will have exactly one solution for each of infinitely many values of N if the k-th bit of Ω is 1 and exactly one solution for each of finitely many values of N if the k-th bit of Ω is 0. We can make use of this to express the bits of Ω through a more intuitive property.

If we treat N in this equation as an unknown instead of a parameter, we get a new (very similar) family of exponential Diophantine equations with only one

[7] For details see [7].

parameter

$$\chi_1^e(k, x_0, x_1, \ldots, x_{m'}) = 0 \tag{12}$$

Since the previous family was singlefold and N has become another unknown, there will be exactly one solution to this single parameter family for each value of N that gave a solution to the double parameter family. Thus, (12) has infinitely many solutions if and only if the k-th bit of Ω is 1.

This same approach can be used with our method [7]. There is a two-parameter singlefold family of exponential Diophantine equations for Q and this can be converted to a single parameter family of exponential Diophantine equations

$$\chi_2^e(k, x_0, x_1, \ldots, x_{m'}) = 0 \tag{13}$$

with between 0 and $2^k - 1$ solutions, the quantity being odd if and only if the k-th bit of Ω is 1.

Finally, we have also shown [7] that both Chaitin's finitude-based method and our parity-based method can be used to generate polynomials for Ω. For a given family of Diophantine equations with two parameters,

$$D(k, N, x_1, \ldots, x_m) = 0 \tag{14}$$

we can construct a polynomial, W, where

$$W(k, x_0, x_1, \ldots, x_m) \equiv x_0 \left(1 - (D(k, x_0, x_1, \ldots, x_m))^2\right) . \tag{15}$$

Note that the parameter, N, is again treated as an unknown and thus denoted x_0.

If we restrict the values of the variables to positive integers then, for a given k, this polynomial takes on exactly the set of all values of N for which (14) has solutions. We can thus use this method on $\chi_1 = 0$ and $\chi_2 = 0$, generating polynomials that express p_k and q_k in the number of distinct positive integer values they take on for different values of k. We therefore have a polynomial whose number of distinct positive integer values fluctuates from odd to even and back in an algorithmically random manner as a parameter k is increased.

Acknowledgements

We would like to thank Gregory Chaitin for helpful discussion and Cristian Calude for kindly offering to present this paper for us at DMTCS'03. TDK wishes to acknowledge the continuing support of Peter Hannaford.

References

1. Cristian S. Calude. A characterization of c.e. random reals. *Theoretical Computer Science*, 271:3–14, 2002.

2. Gregory J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22(3):329–340, July 1975.
3. Gregory. J. Chaitin. *Algorithmic Information Theory.* Cambridge University Press, Cambridge, 1987.
4. Martin Davis, Hilary Putnam, and Julia Robinson. The decision problem for exponential Diophantine equations. *Annals of Mathematics, Second Series*, 74(3):425–436, 1961.
5. David Hilbert. Mathematical problems. lecture delivered before the International Congress of Mathematicians at Paris in 1900. *Bulletin of the American Mathematical Society*, 8:437–479, 1902.
6. Yuri V. Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, Cambridge, Massachusetts, 1993.
7. Toby Ord and Tien D. Kieu. On the existence of a new family of Diophantine equations for Ω. To appear, 2003.

T-information: A New Measure for Similarity Comparison

Jia Yang and Ulrich Speidel*

Department of Computer Science, The University of Auckland
jyan055@cs.auckland.ac.nz,ulrich@cs.auckland.ac.nz

Abstract. Similarity is an important topic in many fields such as Internet search engines, information classification and genetic research. This paper focuses on similarity measures based on algorithms that detect pattern sharing. The pattern sharing concept is introduced using the Lempel-Ziv algorithm as an example. We then discuss T-information, a measure based on a self-learning automaton that also falls in the category of pattern and structure-detecting algorithms. At the end of this paper, we discuss several similarity measures using T-information, including two newly presented measures, m_2 and m_3.

1 Introduction

Similarity measures are important in many fields including Internet search engines, information classification and genetic research. However, similarity is a fuzzy concept which has no clear definitions.

An often-used method to evaluate the similarity between two objects works as follows:

- Map each object to a point in a vector space. This point should represent the main distinguishing features abstracted from the object.
- Compute the distance between these points in the vector space according to some metric. Use the distance as a similarity measure.

However, there are some problems when using this type of measure:

- We may know little or nothing about which features distinguish objects.
- Even if distinguishing features are found, it may be difficult to map these features to a point in a vector space unless a suitable mapping can be found.
- Different features have different weight in representing objects. This needs to be taken into account when choosing a suitable metric. Features also have different properties that make the choice of a metric difficult (e.g. features of 3D-objects described in angles vs. features described as lengths or areas).

Another method [17, 18, 20–23] defines the distance between two objects in a different way. It uses the length of the shortest program which is needed to

* nee Guenther

transform the two objects into each other as the distance between these two objects.

There are some problems using this type of measure, too. When considering DNA, for example, this distance does not properly measure evolutionary sequence distance.

A huge class of objects are strings and all objects can be mapped into strings, one way or another. When we talk about the similarity between two strings, we often mean the amount of pattern and structure they share. What we thus need is an algorithm that detects and measures this shared structure and pattern. This paper discusses the application of such algorithms and points out the problems associated with their use.

A well-known algorithm that detects repeated patterns in strings is the Lempel-Ziv algorithm [1], used both as a string complexity measure and – in various variations – as a popular compression algorithm. It describes a self-learning automaton parsing a string. In the next section, we shall use it to demonstrate the principle of similarity detection with such algorithms. We shall then discuss an alternative approach, a T-information based similarity measure.

2 Using the Lempel-Ziv algorithm for similarity comparisons between strings

The principle described in this section is easily verified on most desktop computers. All that is needed is a number of more or less similar files of an appreciable size (i.e., more than a few dozen bytes), and a Lempel-Ziv-based compression program such as WinZip (under Windows) or `gzip` (under Linux/Unix).

Assume we have two files A and B. By concatenating A and B, we get another file AB. Now use the compression program to compress A, B, and AB, and call the compressed files CA, CB, and CAB. Let $s(X)$ be the size of a file X. In general, we expect:

$$\max(s(CA), s(CB)) \leq s(CAB) \leq s(CA) + s(CB). \tag{1}$$

Reformulate this to:

$$\max(s(CA), s(CB)) + K_1 = s(CAB) \tag{2}$$
$$s(CA) + s(CB) - K2 = s(CAB), \tag{3}$$

where K_1 and K_2 usually are two positive numbers. The more similar A and B are, the more patterns they share and the more the size of the compressed concatenated file decreases. We expect K_1 to decrease and K_2 to increase as the similarity between A and B increases.

This shows how we could use the Lempel-Ziv algorithm for similarity comparisons between strings. However, there are some problems: The result is affected by both the absolute and the relative lengths of the files. If the file lengths are fixed, then this is not a problem. However, in all other cases, an appropriate normalization is required in order to use the algorithm in practical similarity comparisons. For a practical application, one would probably use the Lempel-Ziv production complexity of the strings rather than the compressed file size, but the constraints mentioned above remain.

In the category of pattern and structure-detecting approaches, T-information [10, 7, 9, 13] is another measure which can be used for similarity comparisons between strings either standalone or in combination with Lempel-Ziv. Like the Lempel-Ziv complexity, T-information is also based on a self-learning automaton, but the parsing (called T-decomposition) of the string is entirely different from Lempel-Ziv. The next sections give a brief introduction to the T-information measure and to the underlying concepts of T-complexity and T-codes.

3 T-codes

T-codes [19, 4, 11] were first proposed by Titchener and have since been investigated by a number of other authors [24–27, 2, 5, 28].

What are T-codes? Given a finite alphabet S, a code set $C \subset S^\star$ is a *T-code set* (or *T-code*) if and only if:

- $C = S$, or
- it can be derived from an existing T-code set using a process known as *T-augmentation*, which is defined below.

The T-augmentation $T(C, p, k)$ of a code set $C \subset S^\star$ is defined as follows:

$$\begin{aligned} T(C,p,k) = \Big\{ & x \mid x = p^{k'} y \text{ where} \\ & 0 \le k' \le k \ \wedge \ y \in C \backslash \{p\} \Big\} \\ & \cup \{p^{k+1}\} \end{aligned} \tag{4}$$

where $p \in C$ and $k \in I\!N^+$. A code set C' can be derived from C using T-augmentation iff:

$$\exists p \in C, k \in I\!N^+ \text{ s.t. } C' = T(C, p, k). \tag{5}$$

In this case, we call p the *T-prefix* and k the *T-expansion parameter* of the T-augmentation. In the case of several sequential T-augmentations, these are given subscripts, e.g., p_1, k_1, etc.

The number of T-augmentations used in the derivation of a T-Code set from its alphabet S is called the set's *T-augmentation level.*

In Figure 1, we derive a T-code set from an alphabet S. For simplicity, we choose the binary alphabet $S = \{0, 1\}$ and trees to represent the respective T-code sets, such that each leaf node in the depicted trees represents a codeword in a T-code set. T-augmentation may be regarded as a copy-and-append process in trees.

Firstly, we choose the codeword 1 in S as our first T-prefix p_1. Secondly, we make $k_1 = 2$ copies of S and then append these copies to the original tree (and each other) via the respective leaf nodes corresponding to the T-prefix codeword 1. Thus we get the new tree shown. It represents the T-augmented T-code set, which we denote as follows: $S_{(p_1)}^{(k_1)} = S_{(1)}^{(2)}$. More generally, every T-code set can be written in the form $S_{(p_1,p_2,\ldots,p_n)}^{(k_1,k_2,\ldots,k_n)}$, where $p_1, p_2, \ldots p_n$ represent the prefixes, $k_1, k_2, \ldots k_n$ represent the T-expansion parameters and n represents the T-augmentation level of the T-code set.

In the second T-augmentation in our example, we choose $p_2 = 10$ and $k_2 = 1$ to obtain the second-level set $S_{(1,10)}^{(2,1)}$.

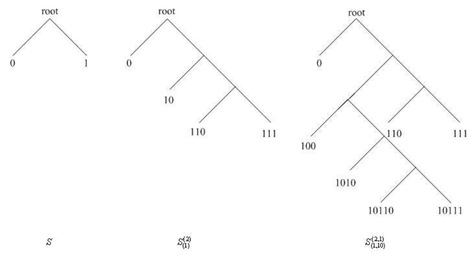

Fig. 1. The construction of $S_{(1)}^{(2)}$ and $S_{(1,10)}^{(2,1)}$ using T-augmentation from $S (S = \{0, 1\})$

T-codes are also strongly self-synchronizing codes [19, 4, 11]. In essence, this means that a T-code decoder will generally parse the suffix of a sufficiently long string identically, independent of the actual decoder state at the beginning of the decoding. A T-code decoder which repeatedly encounters the same (and sufficiently long) pattern in a string should thus parse at least part of this pattern identically each time.

4 T-decomposition

Over a certain alphabet, every finite string can be mapped via a recursive parsing algorithm to a unique T-code set [2, 8] in which this string is one of the longest codewords. This algorithm is called *T-decomposition.* The basic process of this algorithm is shown as follows.

(Let $x \in S^+$ and $a \in S$. We will use T-decomposition to find the unique T-code set in which xa is one of the longest codewords.)

1. Set $n = 0$.
2. Decode xa over $S_{(p_1,p_2,\ldots p_n)}^{(k_1,k_2,\ldots k_n)}$. If the value of n equals to 0, decode xa over S.
3. If the result of 2 is a single codeword, goto 7.
4. Find the second-to-last codeword (we call it d) in the result of 2, and set p_{n+1} to d.
5. Count the number (r) of the adjacent copies of d that immediately precede the second-to-last codeword. Let $k_{n+1} = r + 1$.

6. Add 1 to n and then goto 2.
7. End. $S^{(k_1,k_2,\ldots k_n)}_{(p_1,p_2,\ldots p_n)}$ is the corresponding T-code set for xa (In this T-code set, xa is one of the longest codeword).

Note that in xa the last (rightmost) symbol a does not contribute to construct the corresponding T-code set. If we were to change a to any other symbol $a_1 \in S$, we would still get the same T-code set. This is because in a T-code set, there are more than one longest codewords.

If the cardinality of S is $\#S$, then there are $\#S$ longest codewords in the T-code set $S^{(k_1,k_2,\ldots k_n)}_{(p_1,p_2,\ldots p_n)}$. These longest codewords are all in this form

$$p_n{}^{k_n} p_{n-1}{}^{k_{n-1}} p_{n-2}{}^{k_{n-2}} \ldots p_2{}^{k_2} p_1{}^{k_1} a$$

where $p_n{}^{k_n}$ is the string composed of k_n copies of p_n and $a \in S$. All these longest codewords in $S^{(k_1,k_2,\ldots k_n)}_{(p_1,p_2,\ldots p_n)}$ differ from each other only in the last symbol a.

Example: Find the binary T-code set for which the string 0110001010100 is one of the longest codewords by using T-decomposition:

- First, decode the string over $S = \{0, 1\}$. The result is (commas are used to indicate the boundaries between codewords):
  Result$_1$ $= 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0$
  We find that the second-to-last codeword in Result$_1$ is 0, so $p_1 = 0$. Since there are no other copies of 0 that immediately precede the second-to-last codeword, $k_1 = 1$.
- Decode the string again, this time over $S^{(1)}_{(0)} = \{1, 00, 01\}$. The result is:
  Result$_2$ $= 01, 1, 00, 01, 01, 01, 00$
  The second-to-last codeword in Result$_2$ is 01, so $p_2 = 01$. Since there are 2 other copies of 01 that immediately precede the second-to-last codeword, $k_2 = 3$.
- Now we decode the string over $S^{(1,3)}_{(0,01)}$, and the result is:
  Result$_3$ $= 011, 00, 01010100$
  The second-to-last codeword in Result$_3$ is 00, and there are no other copies of 00 that immediately precede the second-to-last codeword. So $p_3 = 00$ and $k_3 = 1$.
- Decoding the string over $S^{(1,3,1)}_{(0,01,00)}$, we get the result:
  Result$_4$ $= 011, 0001010100$
  Result$_4$ yields $p_4 = 011$ and $k_4 = 1$.
- Since Result$_4$ is not a single codeword, we have to continue the process. After we decode the string over $S^{(1,3,1,1)}_{(0,01,00,011)}$, we find that the result consists of a single codeword only:
  Result$_5$ $= 0110001010100$
  According to the T-decomposition algorithm, we can stop the process now. We have obtained the T-code set $S^{(1,3,1,1)}_{(0,01,00,011)}$, in which the string 0110001010100 is one of the longest codewords.

As mentioned above, the suffixes of sufficiently long repeated patterns are generally parsed identically by a T-code decoder. If we apply this to T-decomposition, we may reformulate this statement as: The suffixes of repeated patterns are generally parsed identically by a T-code decoder at a sufficiently low level. Note that the identical parsing at alphabet level is trivially guaranteed! Note also that identical parsing implies a lower number of decoding passes during T-decomposition: as the copy of a pattern is used as a T-prefix, the remaining copies to the left are absorbed into new codewords at the next level. This means that these copies are no longer available as T-prefixes for future decoding passes. As a result, fewer decoding passes are required.

5 T-complexity and T-information

T-complexity [10, 7, 9, 13, 15] is based on the T-decomposition algorithm. Its definition is as follows:

Let $x \in S^+$. Further, let $S^{(k_1,k_2,\dots,k_n)}_{(p_1,p_2,\dots,p_n)}$ be the T-code set in which xa with $a \in S$ is one of the longest codewords. The *T-complexity* $T_C(x)$ of x is defined as follows:

$$C_T(x) = \sum_{i=1}^{n} \log_2(k_i + 1) \tag{6}$$

Physically speaking, the T-complexity of x is the logarithm of the number of internal nodes in the tree of $S^{(k_1,k_2,\dots,k_n)}_{(p_1,p_2,\dots,p_n)}$.

Titchener [10, 7, 9, 13] used the T-complexity measure to develop the concept of *T-information*. The T-information $IT(x)$ of a string x is defined to be the inverse logarithmic integral of the T-complexity $C_T(x)$ divided by a scaling constant:

$$I_T(x) = li^{-1}(\frac{C_T(x)}{\ln 2}). \tag{7}$$

$I_T(x)$ is the total T-information for x and the natural logarithm gives the T-information the unit of *nats*.

Note that the T-complexity (and hence the T-information) depends strongly on the number of decoding passes in the T-decomposition. Having repeated patterns in a string will lower its T-complexity and, by inference, its T-information.

For example, consider the strings $x =$ “apples and grapefruit” and $y =$ “apples and pineapples”. Notice that x has the repeated pattern “apples”.

The T-complexity and T-information values of x and y are as follows:

$C_T(x) = 17.00$ $I_T(x) = 44.34$

$C_T(y) = 11.58$ $I_T(y) = 25.23$

As we can see, the T-complexity and T-information values of y are much lower than those of x. This is a result primarily of the fact that the pattern “apples” repeats in y, and its leftmost copy parses in a single T-decomposition pass.

6 Using T-information as a similarity measure

As we have seen, both T-complexity and T-information of a string depend strongly on the amount of repeating patterns within the string. The advantage of

T-information over T-complexity is that its growth as a function of string length is linearized, i.e., for most strings of length $2L$, the T-information is roughly twice that of their first (or last) L characters. This makes it a more desirable measure to use in similarity comparison, as it makes it easier to compensate for string length related effects.

The principle here is the same as in the case of Lempel-Ziv. Assume we have two strings x and y whose similarity we want to investigate. For the T-complexity, we could simply use equations equivalent to those for Lempel-Ziv:

$$\max(C_T(x), C_T(y)) + K_1 = C_T(xy) \tag{8}$$

$$C_T(x) + C_T(y) - K_2 = C_T(xy), \tag{9}$$

where K_1 and K_2 are again two numbers that behave the same under (dis)similarity as in the case of Lempel-Ziv.

However, only the first of the above equations applies without restriction to T-information as well:

$$\max(I_T(x), I_T(y)) \leq I_T(xy). \tag{10}$$

That is, the T-information in a concatenated string is generally higher than that in its constituting parts. Substitute I_T for C_T in Equation 8 and 9:

$$\max(I_T(x), I_T(y)) + K_1 = I_T(xy) \tag{11}$$

$$I_T(x) + I_T(y) - K_2 = I_T(xy). \tag{12}$$

Usually K_1 is a positive number. The inspection of practical cases yields the observation that K_2 commonly assumes negative values if x and y are dissimilar, i.e., the amount of T-information in xy may be higher than the sum of the T-information in its parts. However, if there is a high degree of similarity between x and y, K_2 assumes positive values. In summary, the larger the value of K_2 is, the more similar the strings are.

As an example, consider the three strings

$$x = \text{"new zealand is a beautiful country"}$$
$$y = \text{"a garden of flowers"}$$
$$y' = \text{"countries of beauty"}$$

x and y' share the patterns "beaut" and "countr", thus we can say that x seems more similar to y' than to y.

The T-information values of these strings and the corresponding concatenated strings are as follows:

$I_T(x) = 64.27$ $\quad I_T(y) = 29.92$
$I_T(y') = 36.94$ $\quad I_T(xy) = 99.42$
$I_T(xy') = 85.84$

Substituting these values into the corresponding variables in Equation 11 and 12, we get:

$$\max(I_T(x), I_T(y)) + K'_1 = I_T(xy) \tag{13}$$

$$I_T(x) + I_T(y) - K'_2 = I_T(xy) \tag{14}$$

and

$$\max(I_T(x), I_T(y')) + K_1'' = I_T(xy') \tag{15}$$

$$I_T(x) + I_T(y') - K_2'' = I_T(xy'), \tag{16}$$

such that:

$K_1' = 35.15$ $K_2' = -5.32$

$K_1'' = 21.57$ $K_2'' = 25.37$

As we expect, K_1' is much larger than K_1'' and K_2' is much smaller than K_2''.

This shows how we could use the T-information for similarity comparisons between strings. However, this measure has the same problem as Lempel-Ziv algorithm when dealing with similarity comparisons. C_T and I_T generally increase as the length of the string increases, although I_T tends to increase more linearly even for short strings (with a constant entropy rate). In practical application, we must thus take the length of the strings into account.

The following measure given by one of the authors [16] could be a feasible measure to similarity comparison between two strings x and y:

$$m_1(x, y) = \frac{I_T(x) + I_T(y) - I_T(xy)}{|xy|}, \tag{17}$$

where $|xy|$ is the length of xy.

We can use m_1 to measure the similarity between x, y and y' from the previous example:

$$m_1(x, y) = -0.0986$$
$$m_1(x, y') = 0.2900$$

$m_1(x, y) < m_1(x, y')$, which indicates that x is more similar to y' than to y.

Note that $I_T(xy) \neq I_T(yx)$. So m_1 is not a symmetrical measure, which means the result will be affected by the order of x and y in the concatenated string. To resolve this problem, we propose a more symmetric measure:

$$m_2(x, y) = \frac{I_T(x) + I_T(y) - \frac{1}{2}I_T(xy) - \frac{1}{2}I_T(yx)}{|xy|}. \tag{18}$$

Using m_2 to measure the similarity between x, y and y' from the previous example:

$$m_2(x, y) = -0.0986$$
$$m_2(x, y') = 0.3730$$

Again, $m_2(x, y) < m_2(x, y')$, which indicates that x is more similar to y' than to y.

Another possibility of accounting for the length is simply to have a look at the amount of information per symbol added by x when it is concatenated with y:

$$m_3(x, y) = \frac{I_T(xy) - I_T(y)}{|x|}. \tag{19}$$

This equation is highly asymmetrical. In the case of similarity, we would expect little information to be added. This implicitly assumes that the information content of y is larger than that of x, which is the case for most Internet search engines. In this case, x is the query string that we are looking for, and y is the larger record in the data repository in which we hope to find content similar to x. In fact, our experiments indicate that m_3 seems to work well when measuring the similarity between a long string y and a much shorter string x. If y merely contains patterns that are found in x, but x contains extra information, then this approach is problematic.

Note that while m_1 and m_2 grow with similarity, m_3 shrinks.

Experiments with real data (see `http://kiwitrails.tcs.auckland.ac.nz/~jyan055/cgi-bin/similaritysearch.html`) show that all three measures work reasonably well – at least sufficiently well to permit a pre-classification of typical text files of several kB in length compared to fuzzy search strings of several dozen bytes in length. The web site also features an interface through which users can enter values for x and y and compute the three measures described above for their own data.

7 Conclusions

Similarity measures are a naturally fuzzy topic. T-information, based on a self-learning automaton, introduces a parsing algorithm to reveal the sharing of patterns and structure between strings. The research on similarity comparisons using T-information is still at the beginning and a lot of work still has to be done in order to derive a really practical measure. However, we believe that T-information-based measures can at least complement existing measures.

8 Acknowledgements

We would like to thank Mark Titchener for suggesting a re-examination of measures along the lines of m_3.

References

1. A. Lempel and J. Ziv: *On the Complexity of Finite Sequences.* IEEE Trans. Inform. Theory", 22(1), January 1976, pp. 75-81.
2. R. Nicolescu: *Uniqueness Theorems for T-Codes.* Technical Report. Tamaki Report Series no.9, The University of Auckland, 1995.
3. M. R. Titchener: *Generalized T-Codes: an Extended Construction Algorithm for Self-Synchronizing Variable-Length Codes*, IEE Proceedings – Computers and Digital Techniques, 143(3), June 1996, pp. 122-128.
4. U. Guenther: *Data Compression and Serial Communication with Generalized T-Codes*, Journal of Universal Computer Science, V. 2, N 11, 1996, pp. 769-795. `http://www.iicm.edu/jucs_2_11`

5. U. Guenther, P. Hertling, R. Nicolescu, and M. R. Titchener: *Representing Variable-Length Codes in Fixed-Length T-Depletion Format in Encoders and Decoders*, CDMTCS Research Report no.44, Centre of Discrete Mathematics and Theoretical Computer Science, The University of Auckland, August 1997.
6. U. Guenther, P. Hertling, R. Nicolescu, and M. R. Titchener: *Representing Variable-Length Codes in Fixed-Length T-Depletion Format in Encoders and Decoders*, Journal of Universal Computer Science, 3(11), November 1997, pp. 1207–1225. `http://www.iicm.edu/jucs_3_11`.
7. M. R. Titchener: *A Deterministic Theory of Complexity, Information and Entropy*, *IEEE Information Theory Workshop*, February 1998, San Diego.
8. R. Nicolescu and M. R. Titchener, *Uniqueness Theorems for T-Codes*, Romanian Journal of Information Science and Technology, 1(3), March 1998, pp. 243–258.
9. M. R. Titchener, *A novel deterministic approach to evaluating the entropy of language texts*, *Third International Conference on Information Theoretic Approaches to Logic, Language and Computation*, June 16-19, 1998, Hsi-tou, Taiwan.
10. M. R. Titchener, *Deterministic computation of string complexity, information and entropy*, *International Symposium on Information Theory*, August 16-21, 1998, MIT, Boston.
11. U. Guenther: *Robust Source Coding with Generalized T-Codes.* PhD Thesis, The University of Auckland, 1998. `http://www.tcs.auckland.ac.nz/~ulrich/phd.ps.gz`
12. M. R. Titchener, P. M. Fenwick, and M. C. Chen: *Towards a Calibrated Corpus for Compression Testing*, Data Compression Conference, DCC-99, Snowbird, Utah, March 1999.
13. M. R. Titchener: *A measure of Information*, IEEE Data Compression Conference, Snowbird, Utah, March 2000.
14. W. Ebeling, R. Steuer, and M. R. Titchener: *Partition-Based Entropies of Deterministic and Stochastic Maps*, accepted for publication in *Stochastics and Dynamics.*
15. U. Guenther: *T-complexity and T-information Theory-an Executive Summary*, CDMTCS Research Report No. 149, Centre of Discrete Mathematics and Theoretical Computer Science, Auckland, February 2001.
16. U.Guenther: sl Similarity Searches Using a Recursive String Parsing Algorithm, in Supplemental Papers for the 2nd International Conference on Unconventional Models of Computation, UMC2K, Brussels, Dec. 13-16, 2000, p 54
17. M. Li and P. M. B. Vitanyi:*Reversibility and adiabatic computation: trading time and space for energy*, Proc. Royal Society of London, Series A, 452(1996), 769-789.
18. M. Li and P. M. B. Vitanyi:*An Introduction to Kolmogorov Complexity and its Applications*, Springer Verlag, New York, 2nd Edition, 1997.
19. M. R. Titchener:*Generalized T-Codes: an Extended Construction Algorithm for Self-Synchronizing Variable-Length Codes*, IEE Proceedings — Computers and Digital Techniques, 143(3), June 1996, p 122-128.
20. C. H. Bennett, P. Gacs, M. Li, P. M. B. Vatanyi, and W. Zurek:*Information Distance*, IEEE Transactions of Information Theory,44(4), 1998, p 1407-1423.
21. M. Li, J. H. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang:*An Information-based Sequence Distance and Its Application to Whole Mitochondrial Genome Phylogeny*, Bioinformatics, 17(2), 2001, p 149-154.
22. D. Hammer, A. E. Romashchenko, A. Kh. Shen'. N. K. Verashchagin:*Inequalities for shannon entropies and Kolmogorov Complexities*, Proc. 12th IEEE Conf. Computational Complexity, 1997, p 13-23.
23. A. A. Muchnik and N. K. Vereshchagin:*Logical Operations and Kolmogorov Complexity*, Proc. 16th IEEE Conf. Computational Complexity, 2001.

24. G. R. Higgie:*Analysis of the Families of Variable-Length Self-Synchronizing Codes Called T-Codes*, PhD thesis, The University of Auckland, 1993.
25. G. R. Higgie:*Database of Best T-Codes*, Computers and Digital Techniques, 143, July 1996, p 213-218.
26. B. Honary, F. Zolghadr and M. Darnell:*Statistical Real-Time Channel Evaluation (SRTCE) Technique Using Variable-Length T-Codes*, IEE Processdings, 136(4), August 1989, 259-266.
27. N. G. Harlick:*A Comparison between JPEG Compressed Images Using Huffman and T-Codes and Transmitted over a UHF Radio Link*, M. Phil (Engineering) project report, The University of Auckland, Auckland, New Zealand, Decenber, 1993.
28. M. J. Roberts:*Techniques for Determining the Best T-Codes*, Master Thesis, The University of Auckland, October 1993.