Fundamenta Informaticae XX (2010) 1–24
IOS Press

Efficient Graph Kernels for Textual Entailment Recognition

Fabio Massimo Zanzotto
University of Rome "Tor Vergata"
Via del Politecnico 1
00133 Roma, Italy
zanzotto@info.uniroma2.it

Lorenzo Dell'Arciprete
University of Rome "Tor Vergata"
Via del Politecnico 1
00133 Roma, Italy
lorenzo.dellarciprete@gmail.com

Alessandro Moschitti
Department of Information Engineering and Computer Science
Via Sommarive
38123 Povo, (TN) Italy
moschitti@disi.unitn.it

Abstract. One of the most important research area in Natural Language Processing concerns the modeling of semantics expressed in text. Since foundational work in Natural Language Understanding has shown that a deep semantic approach is still not feasible, current research is focused on shallow methods combining linguistic models and machine learning techniques. The latter aim at learning semantic models, like those that can detect the entailment between the meaning of two text fragments, by means of training examples described by specific features. These are rather difficult to design since there is no linguistic model that can effectively encode the lexico-syntactic level of a sentence and its corresponding semantic models. Thus, the adopted solution consists in exhaustively describing training examples by means of all possible combinations of sentence words and syntactic information. The latter, typically expressed as parse trees of text fragments, is often encoded in the learning process using graph algorithms.

In this paper, we propose a class of graphs, the tripartite directed acyclic graphs (tDAGs), which can be efficiently used to design algorithms for graph kernels for semantic natural language tasks involving sentence pairs. These model the matching between two pairs of syntactic trees in terms of all possible graph fragments. Interestingly, since tDAGs encode the association between identical or similar words (i.e. variables), it can be used to represent and learn first-order rules, i.e. rules describable by first-order logic. We prove that our matching function is a valid kernel and we empirically show that, although its evaluation is still exponential in the worst case, it is extremely efficient and more accurate than the previously proposed kernels.

1. Introduction

The automatic design of classifiers using machine learning and linguistically annotated data is a widespread trend in Natural Language Processing (NLP) community. Part-of-speech tagging, named entity recognition, information extraction, and syntactic parsing are NLP tasks that can be modeled as classification problems, where manually tagged sets of examples are used to train the corresponding classifiers. The training algorithms have their foundation in machine learning research but, to induce better classifiers for complex NLP problems, like for example, question-answering, textual entailment recognition (RTE) [12, 13], and semantic role labeling [17], syntactic and/or semantic representations of text fragments have to be modeled as well. Kernel-based machines (see e.g. [11]) can be used for this purpose as they allow to directly describe the similarity, i.e. the kernel function, between two text fragments (or their representations) instead of explicitly describing them in terms of feature vectors. As syntactic and semantic information is often expressed with graphs, kernel design requires efficient algorithms to compare graphs, e.g. by counting the common subgraphs.

In this perspective, previous work has been devoted to kernels for trees [8, 29] since these are extremely important in many linguistic theories such as syntax [6, 26, 4, 7]. The designed tree kernels measure the similarity between two trees by means of efficient algorithms for counting the common subtrees.

However, syntactic and semantic information can also be encoded in projective and non-projective graphs [42, 18, 33], directed-acyclic graphs [37], or generic graphs for which the available tree kernels are inapplicable. This is a critical issue since algorithms for computing the similarity between two general graphs in term of common subgraphs are exponential [40] thus only approximated methods have been proposed. For example, the graph kernel in [16] just counts the number of shared paths whereas the one proposed in [41] can be just applied to a particular class of graphs, i.e. the hierarchical directed acyclic graphs.

In this paper, we define a class of graphs, the tripartite directed acyclic graphs (tDAGs) and we show that the similarity between tDAGs in terms of subgraphs can be used as kernel function in Support Vector Machines (SVMs) [10] to derive semantic implications between pairs of sentences. We show that such model can capture first-order rules (FOR) (rules that can be expressed by first-order logic) for entailment recognition (at least at the syntactic level). Most importantly, we provide an algorithm for efficiently computing our tDAGs kernel, which is extremely more efficient than the one proposed in [31].

The paper is organized as follows. In Section 2, we introduce the needed background. In Section 3, we describe tDAGs and its use for modeling FOR. In Section 4, we introduce the similarity function for FOR spaces. We then introduce our efficient algorithm for computing the similarity among tDAGs. In Section 6, we empirically analyze the computational efficiency of our algorithm and we compare it against the one we proposed in [31]. Finally, in Section 7, we draw conclusions and future work.

2. Background and related work

2.1. Supervised Machine Learning, Kernels, and Natural Language Processing

A recent trend in NLP research is to design systems by combining linguistic theory and machine learning (ML). The latter is typically used for automatically designing classifiers. A classifier is a function:

$$C : I \to 2^T$$

that assigns a subset of the categories in T to elements of the set I. In supervised ML, the function C is learnt using a set of training instances Tr. Each training instance is a pair $(i, t) \in Tr$, where $i \in I$ and $t \in 2^T$, i.e. a class label subset.

ML algorithms extract regularities from training instances observing their description in feature spaces $\mathcal{F} = \mathcal{F}_1 \times ... \times \mathcal{F}_n$. Each dimension j of the space $\mathcal{F}$ is a feature and $\mathcal{F}_j$ is the set of the possible values of j. For example, if we want to learn a classifier that decides if an animal is a cat or a dog (i.e., the set $T = \{cat, dog\}$), we can use features such as the number of teeth, the length of the teeth, the shape of the head, and so on. Each of the features has values in the range defined with the set $\mathcal{F}_j$. We can then define a function F that maps instances $i \in I$ onto points in the feature space, i.e.

$$F(i) = (f_1, ..., f_n) \tag{1}$$

Once $\mathcal{F}$ and Tr have been defined, ML algorithms can be applied for learning C, e.g., decision trees in [38].

One interesting class of algorithms are the-so-called kernel-based machines (see e.g. [11]) like for example the Support Vector Machines (SVMs) [10] or the simpler perceptron [1]. Their most important property is the possibility to use huge feature spaces, implicitly defined by means of kernel functions, e.g. [30, 8]. The classification decisions of kernel machines are made according to an explicit similarity among two instances i_1 and i_2. Given the two feature vectors $F(i_1)$ and $F(i_2)$ for the two instances, their similarity is computed according to a function $sim(F(i_1), F(i_2))$, which is in general the dot product of the two vectors:

$$sim(F(i_1), F(i_2)) = F(i_1)F(i_2) \tag{2}$$

The function $sim(F(i_1), F(i_2))$ encodes data in the learning algorithm (i.e. it is the only one that actually uses instances). We can then associate such function with the kernel $K(i_1, i_2)$, which evaluates it without explicitly using the feature vectors $F(i_1)$ and $F(i_2)$. Therefore a function K, implicitly defines a feature vector space, enabling the use of huge spaces without explicitly representing data on it.

In NLP, this trick is widely used to represent structures in the huge space of substructures, e.g. to represent the syntactic structure of sentences. An interesting example is the tree kernel defined in [8]. In this case a feature j is a syntactic tree fragment, e.g. (S (NP) (VP)) [1]. Thus in the feature vector of an instance (a tree) i, the feature j assumes a value different from 0, if the subtree (S (NP) (VP)) belongs to i. The subtree space is very large but the scalar product just counts the common subtrees between the two syntactic trees, i.e.:

$$K(i_1, i_2) = F(i_1)F(i_2) = |S(i_1) \cap S(i_2)| \tag{3}$$

[1]A sentence S composed by a noun phrase NP and a verbal phrase VP.

where $S(\cdot)$ extracts the subtrees from i_1 or i_2. Efficient kernels for trees have been defined in [8, 29]. Yet, some important NLP tasks such as Recognition of Textual Entailment [12, 13] and some linguistic theories such as HPSG [37] require more general graphs and, then, more general algorithms for computing similarity among graphs.

2.2. Machine Learning for Textual Entailment Recognition

Recognition of Textual Entailment (RTE) [12, 13] is an important basic task in natural language processing and understanding. The task is defined as follows: given a text T and a hypothesis H, we need to determine whether a text T implies a hypothesis H. For example, we need to determine whether or not "*Farmers feed cows animal extracts*" entails "*Cows eat animal extracts*" (T_1, H_1). It should be noted that a model suitable to approach the complex natural language understanding task must also be capable of recognizing textual entailment [5]. Overall, in more specific NLP challenges, where we want to build models for specific tasks, systems and models solving RTE can play a very important role.

RTE has been proposed as a generic task tackled by systems for open domain question-answering [43], multi-document summarization [14], information extraction [32], and machine translation. In question-answering, a subtask of the problem of finding answers to questions can be rephrased as an RTE task. A system could answer the question "*Who played in the 2006 Soccer World Cup?*" using a retrieved text snippet "*The Italian Soccer team won the World Championship in 2006*". Yet, knowing that "*The Italian soccer team*" is a candidate answer, the system has to solve the problem of deciding whether or not the sentence "*The Italian football team won the World Championship in 2006*" entails the sentence "*The Italian football team played in the 2006 Soccer World Cup?*". The system proposed in [20], the answer validation exercise [35], and the correlated systems (e.g., [47]) use this reformulation of the question-answering problem. In multi-document summarization (extremely useful for intelligence activities), again, part of the problem, i.e., the detection of redundant sentences, can be framed as a RTE task [21]. The detection of redundant or implied sentences is a very important tasks as it is the way of correctly reducing the size of the documents.

RTE models are then extremely important as these enable the possibility of building final NLP applications. Yet, as any NLP model, textual entailment recognizers need a big amount of knowledge. This knowledge ranges from simple equivalence, similarity, or relatedness between words to more complex relations between generalized text fragments. For example, to deal with the above example, an RTE system should have:

- a similarity relationship between the words *soccer* and *football*, even if this similarity is valid only under specific conditions;
- the entailment relation between the words *win* and *play*
- the entailment rule $\boxed{\text{X}}won\boxed{\text{Y}}in\boxed{\text{Z}} \rightarrow \boxed{\text{X}}played\boxed{\text{Y}}in\boxed{\text{Z}}$

This knowledge is generally extracted in a supervised setting using annotated training examples (e.g., [48]) or in unsupervised setting using large corpora (e.g., [25, 34, 49]). The kind of knowledge that can be extracted from the two methods is extremely different as unsupervised methods can induce positive entailment rules whereas supervised learning methods can learn both positive and negative entailment rules. A rule such as *tall* does not entail *short*, even if the two words are related, can be learned only using supervised machine learning approaches.

To use supervised machine learning approaches, we have to frame the RTE task as a classification problem [48]. This is in fact possible as an RTE system can be seen as a classifier that, given a (T, H) pair, outputs one of these two classes: *entails* if T entails H or *not-entails* if T does not entails H. Yet, this classifier as well as its learning algorithm have to deal with an extremely complex feature space in order to be effective. If we represent T and H as graphs, the classifier and the learning algorithm has to deal with two interconnected graphs since, to model the relation between T and H, we need to connect words in T and words in H.

In [39, 19, 22], the problem of dealing with interconnected graphs is solved outside the learning algorithm and the classifier. The two connected graphs representing the two texts T and H are used to compute similarity features, i.e., features representing the similarity between T and H. The underlying idea is that lexical, syntactic, and semantic similarities between sentences in a pair are relevant features to classify sentence pairs in classes such as *entail* and *not-entail*. In this case, features are not subgraphs. Yet, these models can easily fail as two similar sentences may in one case be an entailment pair and in the other not. For example, the sentence "*All companies pay dividends*" (A) entails that "*All insurance companies pay dividends*" (B) but does not entail "*All companies pay cash dividends*" (C). In number of different words, the difference between (A) and (B) is the same existing between (A) and (C).

If we want to better exploit training examples to learn textual entailment classifiers, we need to use first-order rules (FOR) (that describe entailment in the training instances). Suppose that the instance "*Pediatricians suggest women to feed newborns breast milk*" entails "*Pediatricians suggest that newborns eat breast milk*", (T_2, H_2), is contained in the training data. For classifying (T_1, H_1), the first-order rule $\rho = \ feed\boxed{\text{Y}}\boxed{\text{Z}} \ \rightarrow \ \boxed{\text{Y}}eat\boxed{\text{Z}}$ must be learned from (T_2, H_2). The feature space describing first-order rules, that we introduced in [46], allows for highly accurate textual entailment recognition with respect to traditional feature spaces. Unfortunately, this model, as well as the one proposed in [31], shows two major limitations: they can represent rules with less than seven variables and the similarity function is not a valid kernel.

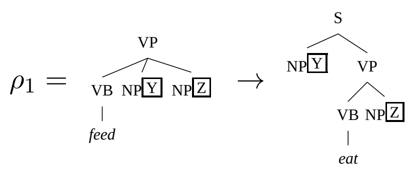
In [27], first-order rules have been explored. Yet, the associated spaces are extremely small. Only some features representing first-order rules were explored. Pairs of graphs are used here to determine if a feature is active or not, i.e., if the rule fires or not. A larger feature space of rewrite rules was implicitly explored in [45] but they considered only ground rewrite rules. Also in machine translation, some methods such as [15] learn graph based rewrite rules for generative purposes. Yet, the method presented in [15] can model first-order rewrite rules only with a very small amount of variables, i.e., two or three.

3. Representing first-order rules and sentence pairs as tripartite directed acyclic graphs

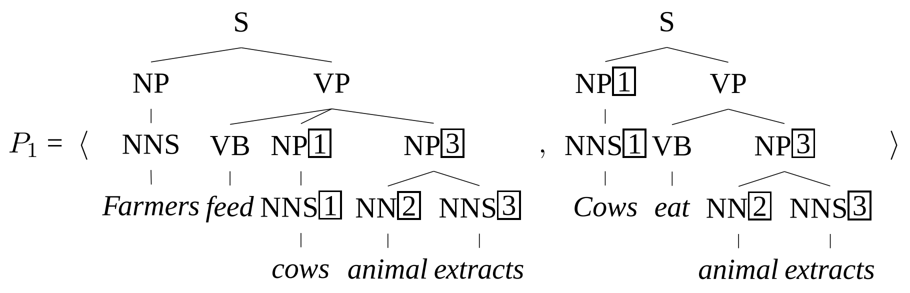
To define and build feature spaces for first-order rules we cannot rely on existing kernel functions over tree fragment feature spaces [8, 29]. These feature spaces are not enough expressive for describing rules with variables. In this section, we motivate with an example why we cannot use tree fragments and we will then introduce the tripartite directed acyclic graphs ($tDAGs$) as a subclass of graphs useful to model first order rules. We intuitively show that, if sentence pairs are described by $tDAG$s, determining whether or not a pair triggers first-order rewrite rule is a graph matching problem.

To explore the problem of defining first-order feature spaces, we can consider the rule $\rho = \ feed\boxed{\text{Y}}\boxed{\text{Z}} \ \rightarrow$

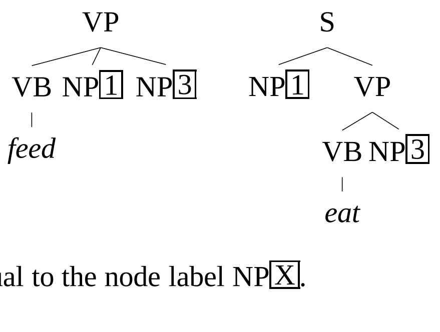
$\boxed{Y}eat\boxed{Z}$ and the above sentence pair (T_1, H_1). The rule ρ encodes the entailment relation of the verb *to feed* and the verb *to eat*. If represented over a syntactic interpretation, the rule has the following aspect:



A similar tree-based representation can be derived for the pair (T_1, H_1) where the syntactic interpretations of both sentences in the pair are represented and the connection between the text T and the hypothesis H are somehow explicit in the structure. This representation of the pair (T_1, H_1) has the following aspect:


P_1 = ⟨
S
NP VP
NNS VB NP$\boxed{1}$ NP$\boxed{3}$
Farmers feed NNS$\boxed{1}$ NN$\boxed{2}$ NNS$\boxed{3}$
cows animal extracts
,
S
NP$\boxed{1}$ VP
NNS$\boxed{1}$ VB NP$\boxed{3}$
Cows eat NN$\boxed{2}$ NNS$\boxed{3}$
animal extracts
⟩

Augmenting node labels with numbers is one of the way of co-indexing parts of the trees. In this case, co-indexes indicate that a part of the tree is significantly related with another part of the other tree, e.g., the co-index $\boxed{1}$ on the NNS nodes describes the relation between the two nodes describing the plural common noun (NNS) *cows* in the two trees and the same co-index on the NP nodes indicates the relation between the noun phases (NP) having *cows* as semantic head [37]. These co-indexes are frequently used as additional parts of node labels in computational linguistics to indicate relations among different parts in a syntactic tree (e.g., [26]). Yet, the names used for the co-indexes have a precise meaning only within the trees where these are used. Then, having a similar representation for the rule ρ and the pair P_1, we need to determine whether or not the pair P_1 triggers the rule ρ. Considering both variables in the rule ρ and co-indexes in the pair P_1 as extensions of the node tags, we would like to see it as a tree matching problem. In this case, we could easily apply existing kernels for tree fragment feature spaces [8, 29]. However this simple example shows that this is not the case, as the two trees representing the rule ρ cannot be matched with the subgraph:



as the node label NP$\boxed{1}$ is not equal to the node label NP$\boxed{X}$.

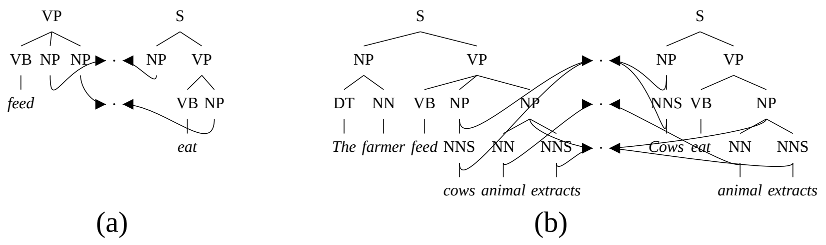


Figure 1. A simple rule and a simple pair as a graph

To solve the above problem, similarly to the case of feature structures [3], we can represent the rule ρ and the pair P_1 as graphs. We start the discussion describing the graph for the rule ρ. Since we are interested to the relation between the right hand side and the left hand side of the rule, we can substitute each variable with an unlabeled node. We then connect tree nodes having variables with the corresponding unlabeled node. The result is the graph in Figure 1(a). The variables $\boxed{\text{Y}}$ and $\boxed{\text{Z}}$ are represented by the unlabeled nodes between the trees.

In the same way we can represent the sentence pair (T_1, H_1) using graph with explicit links between related words and nodes (see Figure 1(b)). We can link words using anchoring methods as in [39]. These links can then be propagated in the syntactic tree using semantic heads of the constituents [37]. The rule ρ_1 matches over the pair (T_1, H_1) if the graph ρ_1 is among the subgraphs of the graph in Figure 1(b).

Both rules and sentence pairs are graphs of the same type. These graphs are basically two trees connected through an intermediate set of nodes representing variables in the rules and relations between nodes in the sentence pairs. We will hereafter call these graphs *tripartite directed acyclic graphs* (tDAGs). The formal definition follows.

Definition 3.1. tDAG: A *tripartite directed acyclic graph* is a graph $G = (N, E)$ where

- the set of nodes N is partitioned in three sets N_t, N_g, and A
- the set of edges is partitioned in four sets E_t, E_g, E_{A_t}, and E_{A_g}

such that $t = (N_t, E_t)$ and $g = (N_g, E_g)$ are two trees and $E_{A_t} = \{(x, y)|x \in N_t \text{ and } y \in A\}$ and $E_{A_g} = \{(x, y)|x \in N_g \text{ and } y \in A\}$ are the edges connecting the two trees.

A $tDAG$ is a partially labeled graph. The labeling function L only applies to the subsets of nodes related to the two trees, i.e., $L : N_t \cup N_g \rightarrow \mathcal{L}$. Nodes in the set A are not labeled.

The explicit representation of the tDAG in Figure 1(b) shows that to *fire* the appropriate rule for sentence pair is a graph matching problem. To simplify our explanation we will then describe a tDAG with an alternative and more convenient representation: a tDAG $G = (N, E)$ can be seen as pair $G = (\tau, \gamma)$ of *extended trees* τ and γ where $\tau = (N_t \cup A, E_t \cup E_{A_t})$ and $\gamma = (N_g \cup A, E_g \cup E_{A_g})$. These are extended trees as each tree contains the relations with the other tree.

As for the feature structures, we will graphically represent $(x, y) \in E_{A_t}$ and $(z, y) \in E_{A_g}$ as boxes $\boxed{\text{y}}$ respectively on the node x and on the node z. These nodes will then appear as $L(x)\boxed{\text{y}}$ and $L(z)\boxed{\text{y}}$, e.g., NP$\boxed{1}$. The name y is not a label but a placeholder representing an unlabelled node. This representation

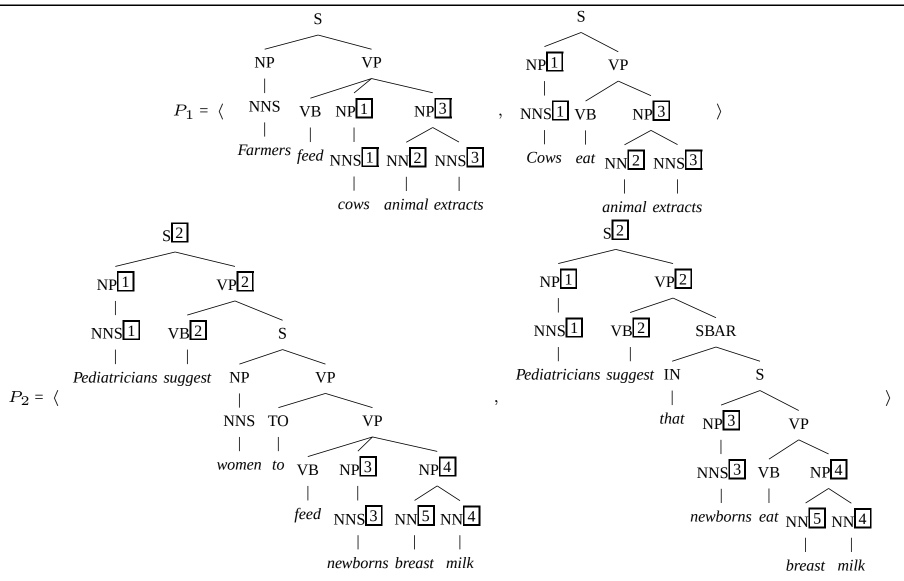


Figure 2. Two tripartite DAGs

is used for rules and for sentence pairs. The sentence pair in Figure 1(b) is then represented as reported in Figure 2 (see P_1).

4. An efficient algorithm for computing the first-order rule space kernel

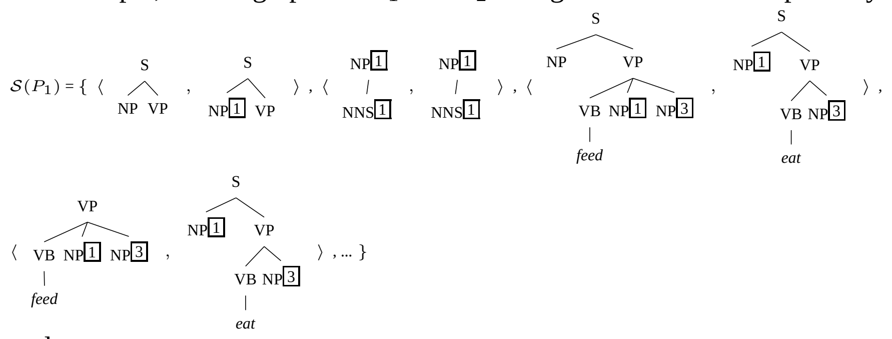
In this section, we present our efficient algorithm implementing feature spaces for deriving first-order rules (FOR). In Section 4.1, we firstly define the similarity function, i.e., the kernel $K(G_1, G_2)$, that implements the feature spaces for learning first-order rules. This kernel is based on the isomorphism between graphs and our efficient approach for detecting the isomorphism between tDAGs (Section 4.2). Then, we present the basic idea and the formalization of our efficient algorithm for computing $K(G_1, G_2)$ based on the properties of the tDAGs isomorphism (Section 4.3). We demostrate that our algorithm and, then, our kernel function computes the FOR feature space. We finally describe the ancillary algorithms and properties for making the computation possible (Section 4.4).

4.1. Kernel functions over first-order rule feature spaces

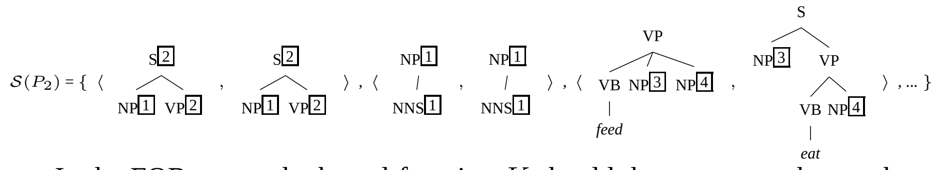
In this section we introduce FOR and we then define the prototypical kernel function that implicitly defines it. FOR is in general the space of all the possible first-order rules defined with tDAGs. Within this space it is possible to define the function $\mathcal{S}(G)$ that computes all the subgraphs (features) of the tDAG G. Therefore, we need to take into account the subgraphs of G that represent first-order rules.

Definition 4.1. $S(G)$: Given a tDAG $G = (\tau, \gamma)$, $\mathcal{S}(G)$ is the set of subgraphs of G of the form (t, g), where t and g are extended subtrees of τ and γ, respectively.

For example, the subgraphs of P_1 and P_2 in Figure 2 are hereafter partially represented:



and



In the FOR space, the kernel function K should then compute the number of subgraphs in common among two tDAGs G_1 and G_2. The trivial way to describe K is using the intersection operator, i.e., the kernel $K(G_1, G_2)$ is the following:

$$K(G_1, G_2) = |\mathcal{S}(G_1) \cap \mathcal{S}(G_2)|, \tag{4}$$

where, a graph g in the intersection $\mathcal{S}(G_1) \cap \mathcal{S}(G_2)$ belongs to both $\mathcal{S}(G_1)$ and $\mathcal{S}(G_2)$.

We point out that determining whether two graphs, g_1 and g_2, are the *same* graph $g_1 = g_2$ is not trivial. For example, it is not sufficient to naively compare graphs to determine that ρ_1 belongs both to $\mathcal{S}(G_1)$ and $\mathcal{S}(G_2)$. If we compare the string representation of the fourth tDAG in $\mathcal{S}(P_1)$ and the third in $\mathcal{S}(P_2)$, we cannot derive that the two graphs are the *same* graph.

We need to use a correct comparison for $g_1 = g_2$, i.e., the *isomorphism* between two graphs. Let us to define $Iso(g_1, g_2)$ as the predicate indicating the isomorphism between the two graphs. When $Iso(g_1, g_2)$ is true, both g_1 and g_2 can represent the two graphs. Unfortunately computing $Iso(g_1, g_2)$ has an exponential complexity [24].

To solve the complexity problem we need to differently define the intersection operator between sets of graphs. We will use the same symbol but we will use the prefix notation.

Definition 4.2. Given two tDAGs G_1 and G_2, we define the intersection between the two sets of subgraphs $\mathcal{S}(G_1)$ and $\mathcal{S}(G_2)$ as:

$$\cap(\mathcal{S}(G_1), \mathcal{S}(G_2)) = \{g_1 | g_1 \in \mathcal{S}(G_1), \exists g_2 \in \mathcal{S}(G_2), Iso(g_1, g_2)\}$$

4.2. Isomorphism between tDAGs

An isomorphism between graphs is the critical point for defining an effective graph kernel so we here review its definition and we adapt it to tDAGs. An isomorphism between two tDAGs can be divided in two sub-problems:

- finding an partial isomorphism between two pairs of *extended trees*

- checking whether the partial isomorphism found between the two pairs of *extended trees* are compatible.

Consider the general definition for graph isomorphism.

Definition 4.3. Two graphs, $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ are *isomorphic* (or match) if $|N_1| = |N_2|$, $|E_1| = |E_2|$, and a bijective function $f : N_1 \rightarrow N_2$ exists such that these properties hold:

- for each node $n \in N_1$, $L(f(n)) = L(n)$
- for each edge $(n_1, n_2) \in E_1$ an edge $(f(n_1), f(n_2))$ is in E_2

The bijective function f is a member of the combinatorial set $\mathcal{F}$ of all the possible bijective functions between the two sets N_1 and N_2.

The trivial algorithm for detecting if two graphs are isomorphic is exponential [24]. It explores all the set $\mathcal{F}$. It is still undetermined if the general graph isomorphism problem is NP-complete. Yet, we can use the fact that tDAGs are two extended trees for building an efficient algorithm since for them there is an efficient algorithm (as the one used in [8]).

Given two tDAGs $G_1 = (\tau_1, \gamma_1)$ and $G_2 = (\tau_2, \gamma_2)$ the isomorphism can be reduced to the problem of detecting two properties:

1. *Partial isomorphism*. Two tDAGs G_1 and G_2 are *partially isomorphic*, if τ_1 and τ_2 are isomorphic and if γ_1 and γ_2 are isomorphic. The partial isomorphism produces two bijective functions f_τ and f_γ.
2. *Constraint compatibility*. Two bijective functions f_τ and f_γ are compatible on the sets of nodes A_1 and A_2, if for each $n \in A_1$, it happens that $f_\tau(n) = f_\gamma(n)$.

We can rephrase the second property, i.e., the constraint compatibility, as follows. We define two constraints $c(\tau_1, \tau_2)$ and $c(\gamma_1, \gamma_2)$ representing the functions f_τ and f_γ restricted to the sets A_1 and A_2. The two constraints are defined as follows: $c(\tau_1, \tau_2) = \{(n, f_\tau(n)) | n \in A_1\}$ and $c(\gamma_1, \gamma_2) = \{(n, f_\gamma(n)) | n \in A_1\}$. Then two partially isomorphic tDAGs are isomorphic if the constraints match, i.e., $c(\tau_1, \tau_2) = c(\gamma_1, \gamma_2)$.

For example, the fourth pair of $\mathcal{S}(P_1)$ and the third pair of $\mathcal{S}(P_2)$ are isomorphic as: (1) these are partially isomorphic, i.e., the right hand sides τ and the left hand sides γ are isomorphic; (2) both pairs of extended trees generate the constraint $c_1 = \{(\boxed{1}, \boxed{3}), (\boxed{3}, \boxed{4})\}$. In the same way, the second pair of $\mathcal{S}(P_1)$ and the second pair of $\mathcal{S}(P_2)$ generate $c_2 = \{(\boxed{1}, \boxed{1})\}$.

Given the above considerations, we need to define what a constraint is and we need to demonstrate that two tDAGs satisfying the two properties are isomorphic.

Definition 4.4. Given two tDAGs, $G_1 = (N_{t_1} \cup N_{g_1} \cup A_1, E_1)$ and $G_2 = (N_{t_2} \cup N_{g_2} \cup A_2, E_2)$, a constraint c is a bijective function between the sets A_1 and A_2.

We can then enunciate the theorem.

Theorem 4.1. Two tDAGs $G_1 = (N_1, E_1) = (\tau_1, \gamma_1)$ and $G_2 = (N_2, E_2) = (\tau_2, \gamma_2)$ are isomorphic if they are partially isomorphic and constraint compatibility holds for the two partial isomorphism functions f_τ and f_γ.

Proof:
First we show that $|N_1| = |N_2|$. Since *partial isomorphism* holds, we have that $\forall n \in \tau_1.L(n) = L(f_\tau(n))$. However, since nodes in N_{t_1} and N_{t_2} are labeled whereas nodes in A_1 and A_2 are unlabeled it follows that $\forall n \in N_{t_1}.f_\tau(n) \in N_{t_2}$ and $\forall n \in A_1.f_\tau(n) \in A_2$. Thus we have that $|N_{t_1}| = |N_{t_2}|$ and $|A_1| = |A_2|$. Similarly, we can show that $|N_{g_1}| = |N_{g_2}|$, and since N_t, N_g and A are disjoint sets, we can conclude that $|N_{t_1} \cup N_{g_1} \cup A_1| = |N_{t_2} \cup N_{g_2} \cup A_2|$, i.e. $|N_1| = |N_2|$.

Now we show that $|E_1| = |E_2|$. By *partial isomorphism* we know that $|E_{t_1} \cup E_{A_{t_1}}| = |E_{t_2} \cup E_{A_{t_2}}|$ and $|E_{g_1} \cup E_{A_{g_1}}| = |E_{g_2} \cup E_{A_{g_2}}|$, so $|E_{t_1} \cup E_{A_{t_1}}| + |E_{g_1} \cup E_{A_{g_1}}| = |E_{t_2} \cup E_{A_{t_2}}| + |E_{g_2} \cup E_{A_{g_2}}|$. Since these are all disjoint sets, it trivially follows that $|E_{t_1} \cup E_{A_{t_1}} \cup E_{g_1} \cup E_{A_{g_1}}| = |E_{t_2} \cup E_{A_{t_2}} \cup E_{g_2} \cup E_{A_{g_2}}|$, i.e. $|E_1| = |E_2|$.

Finally, we have to show the existence of a bijective function $f : N_1 \rightarrow N_2$ such as the one described in the definition of graph isomorphism. Consider the following restricted functions for f_τ and f_γ: $f_\tau|_{N_{t_1}} : N_{t_1} \rightarrow N_{t_2}$, $f_\gamma|_{N_{g_1}} : N_{g_1} \rightarrow N_{g_2}$, $f_\tau|_{A_1} : A_1 \rightarrow A_2$, $f_\gamma|_{A_1} : A_1 \rightarrow A_2$. By *constraint compatibility*, we have that $f_\tau|_{A_1} = f_\gamma|_{A_1}$. Now we can define function f as follows:

$$f(n) = \begin{cases} f_\tau(n) \text{ if } n \in N_{t_1} \\ f_\gamma(n) \text{ if } n \in N_{g_1} \\ f_\tau(n) = f_\gamma(n) \text{ if } n \in A_1 \end{cases}$$

Since the properties described in the definition of graph isomorphism hold for both f_τ and f_γ, they hold for f as well. □

4.3. General idea for an efficient kernel function

As discussed above, two tDAGs are isomorphic if the two properties, the *partial isomorphism* and the *constraint compatibility*, hold. To compute the kernel function $K(G_1, G_2)$ defined in Section 4.1, we can exploit these properties in the reverse order. Given a constraint c, we can select all the graphs that meet the constraint c (*constraint compatibility*). Having determined the set of all the tDAGs meeting the constraint, we can detect the *partial isomorphism*. We split each pair of tDAGs into the four extended trees and we determine if these extended trees are compatible.

We introduce this method to compute the kernel $K(G_1, G_2)$ in FOR in two steps. Firstly, we give an intuitive explanation and, secondly, we formally define the kernel.

4.3.1. Intuitive explanation

To give an intuition of the kernel computation, without loss of generality and for sake of simplicity, we use two non-linguistic tDAGs, P_a and P_b (see Figure 3), and the subgraph function $\widetilde{\mathcal{S}}(\theta)$ where θ is one of the extended trees of the pairs, i.e., γ or τ. This latter is an approximated version of $\mathcal{S}(\theta)$ that generates tDAGs with subtrees rooted in the root of the initial trees of θ.

To exploit the *constraint compatibility* property, we define C as *the set of all the relevant alternative constraints*, i.e., the constraints c that could be generated when detecting the *partial isomorphism*. For P_a and P_b, this set is $C = \{c_1, c_2\} = \{\{(\boxed{1},\boxed{1}),(\boxed{2},\boxed{2})\},\{(\boxed{1},\boxed{1}),(\boxed{2},\boxed{3})\}\}$.

We can informally define $\cap(\widetilde{\mathcal{S}}(P_a), \widetilde{\mathcal{S}}(P_b))|_c$ as the common subgraphs that meet the constraint c. For example in Fig. 4, the first tDAG of the set $\cap(\widetilde{\mathcal{S}}(P_a), \widetilde{\mathcal{S}}(P_b))|_{c_1}$ belongs to the set as its constraint

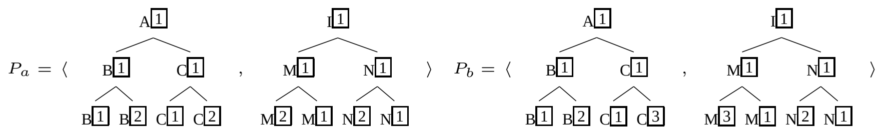


Figure 3. Simple non-linguistic tDAGs

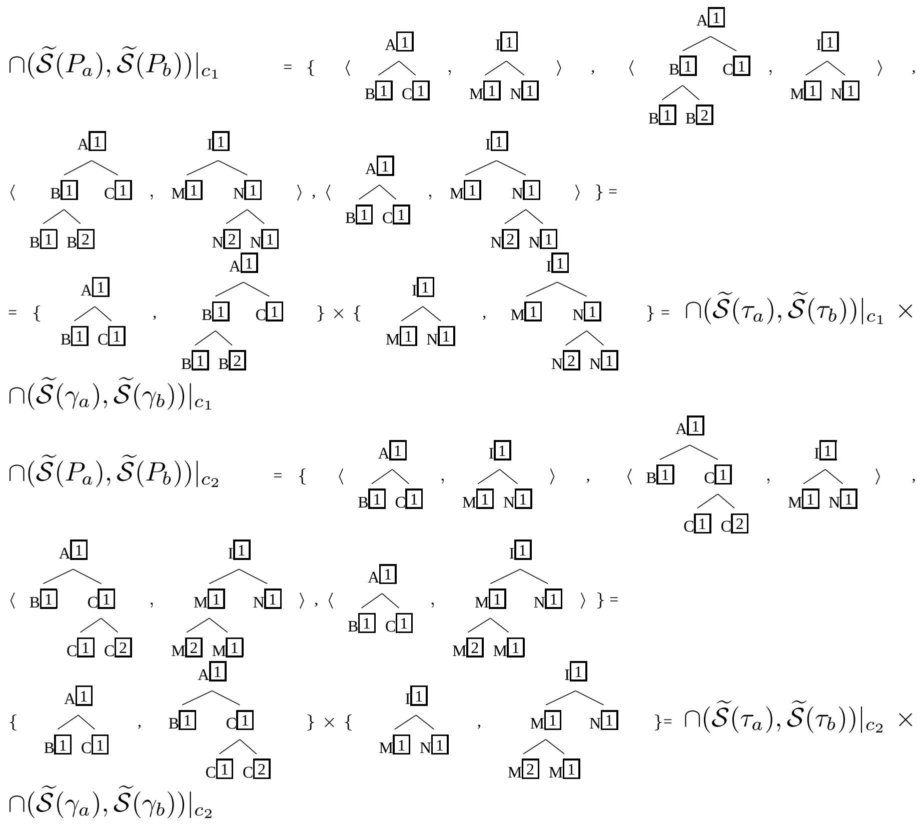


Figure 4. Intuitive idea for the kernel computation

$c' = \{(\boxed{1}, \boxed{1})\}$ is a subset of c_1. Then, we can determine the kernel $K(P_a, P_b)$ as:

$$\begin{aligned} K(P_a,P_b) = & \ |\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))| = \\ = & \left|\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_1} \bigcup \cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_2}\right| \end{aligned} \tag{5}$$

Looking at Figure 4, we compute the value of the kernel for the two pairs as $K(P_a, P_b) = 7$. For better computing the cardinality of the union of the sets, it is possible to use the inclusion-exclusion principle. The value of the kernel for the example can be derived as:

$$\begin{aligned} K(P_a,P_b) = & \left|\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_1} \bigcup \cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_2}\right| = \\ = & \left|\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_1}\right| + \left|\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_2}\right| + \\ & - \left|\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_1} \bigcap \cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_2}\right| \end{aligned} \tag{6}$$

A nice property that can be easily demonstrated is that:

$$\begin{aligned} \cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_1} \bigcap \cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_2} = \\ = \cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_1 \cap c_2} \end{aligned} \tag{7}$$

Expressing the kernel computation in this way is important since elements in $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c$ already satisfy the property of *constraint compatibility*. We can exploit now the *partially isomorphic* property. We will then find the elements in $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c$ using the partial isomorphism. Then, we can write the following equivalence:

$$\begin{aligned} & \cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c = \\ & = \cap(\widetilde{\mathcal{S}}(\tau_a),\widetilde{\mathcal{S}}(\tau_b))|_c \times \cap(\widetilde{\mathcal{S}}(\gamma_a),\widetilde{\mathcal{S}}(\gamma_b))|_c \end{aligned} \tag{8}$$

Figure 4 reports this equivalence for the two sets derived using the constraints c_1 and c_2. Note that this equivalence is not valid if a constraint is not applied, i.e., $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b)) \neq \cap(\widetilde{\mathcal{S}}(\tau_a),\widetilde{\mathcal{S}}(\tau_b)) \times \cap(\widetilde{\mathcal{S}}(\gamma_a),\widetilde{\mathcal{S}}(\gamma_b))$. The pair P_a itself does not belong to $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))$ but it does belong to $\cap(\widetilde{\mathcal{S}}(\tau_a),\widetilde{\mathcal{S}}(\tau_b)) \times \cap(\widetilde{\mathcal{S}}(\gamma_a),\widetilde{\mathcal{S}}(\gamma_b))$.

Equivalence (8) allows to compute the cardinality of $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c$ using the cardinalities of $\cap(\widetilde{\mathcal{S}}(\tau_a),\widetilde{\mathcal{S}}(\tau_b))|_c$ and $\cap(\widetilde{\mathcal{S}}(\gamma_a),\widetilde{\mathcal{S}}(\gamma_b))|_c$. The latter sets contain only extended trees where the equivalences between unlabelled nodes are given by c. We can then compute the cardinalities of these two sets using methods developed for trees (e.g., the kernel function $K_S(\theta_1, \theta_2)$ proposed in [8] and refined in $K_S(\theta_1, \theta_2, c)$ for extended trees in [31, 48]). The cardinality of $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c$ is then computed as:

$$\begin{aligned} & \left|\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c\right| = \\ & = \left|\cap(\widetilde{\mathcal{S}}(\tau_a),\widetilde{\mathcal{S}}(\tau_b))|_c\right| \left|\cap(\widetilde{\mathcal{S}}(\gamma_a),\widetilde{\mathcal{S}}(\gamma_b))|_c\right| = K_S(\tau_a,\tau_b,c)K_S(\gamma_a,\gamma_b,c) \end{aligned} \tag{9}$$

4.3.2. Formalization

The intuitive explanation along with the associated examples suggests the following steps for computing the desired kernel function:

- Given a set of alternative constraints C, we can divide the original intersection in a union of intersections over the projection of the original set on the constraints (Equation 5). This is the application of the *constraint compatibility*.
- The cardinality of the union of intersection can be computed using the inclusion-exclusion principle (Equation 6). Given the property in Equation 7, we can transfer the intersections from the sets to the constraints.
- Applying the *partial isomorphism detection*, we can transfer the computation of the intersection from tDAGs to the extended trees (Equation 8) and, then, apply efficient algorithms for computing the cardinality of these intersections between extended trees [8, 31, 48]

In the rest of the paper, we will use again the general formulation of the $\mathcal{S}(G)$ function instead of the previous $\widetilde{\mathcal{S}}$ version.

To provide the theorem proving the validity of the algorithm, we need to provide some definitions. Firstly, we define the projection operator of an intersection of tDAGs or extended trees given a constraint c.

Definition 4.5. Given two tDAGs G_1 and G_2, the set $\cap(\mathcal{S}(G_1), \mathcal{S}(G_2))|_c$ is the intersection of the related sets $\mathcal{S}(G_1)$ and $\mathcal{S}(G_2)$ projected on the constraint c. A tDAG $g' = (\tau', \gamma') \in \mathcal{S}(G_1)$ is in $\cap(\mathcal{S}(G_1), \mathcal{S}(G_2))|_c$ if $\exists g'' = (\tau'', \gamma'') \in \mathcal{S}(G_2)$ such that g' is partially isomorphic to g'', and $c' = c(\tau', \tau'') = c(\gamma', \gamma'')$ is *covered* by and *compatible* with the constraint c, i.e., $c' \subseteq c$.

We can then generalize the property (7) as follows.

Lemma 4.1. Given two tDAGs G_1 and G_2, the following property holds:

$$\bigcap_{c \in C} \cap(\mathcal{S}(G_1), \mathcal{S}(G_2))|_c = \cap(\mathcal{S}(G_1), \mathcal{S}(G_2))|_{\bigcap_{c \in C} c}$$

We omit this proof that can be easily demonstrated.

Secondly, we can generalize the equivalence (8) in the following form.

Lemma 4.2. Let $G_1 = (\tau_1, \gamma_1)$ and $G_2 = (\tau_2, \gamma_2)$ be two tDAGs. Then:

$$\cap(\mathcal{S}(G_1), \mathcal{S}(G_2))|_c = \cap(\mathcal{S}(\tau_1), \mathcal{S}(\tau_2))|_c \times \cap(\mathcal{S}(\gamma_1), \mathcal{S}(\gamma_2))|_c$$

Proof:
First we show that if $g = (\tau, \gamma) \in \cap(\mathcal{S}(G_1), \mathcal{S}(G_2))|_c$ then $\tau \in \cap(\mathcal{S}(\tau_1), \mathcal{S}(\tau_2))|_c$ and $\gamma \in \cap(\mathcal{S}(\gamma_1), \mathcal{S}(\gamma_2))|_c$. To show that a tree τ belongs to $\cap(\mathcal{S}(\tau_1), \mathcal{S}(\tau_2))|_c$, we have to show that $\exists \tau' \in \mathcal{S}(\tau_1), \tau'' \in \mathcal{S}(\tau_2)$ such that τ, τ' and τ'' are isomorphic and $f_\tau|_{A_1} \subseteq c$, i.e. $c(\tau', \tau'') \subseteq c$. Since $g = (\tau, \gamma) \in \cap(\mathcal{S}(G_1), \mathcal{S}(G_2))|_c$, we have that $\exists g' = (\tau', \gamma') \in \mathcal{S}(G_1), g'' = (\tau'', \gamma'') \in \mathcal{S}(G_2)$ such that τ, τ' and τ'' are isomoprhic, γ, γ' and γ'' are isomoprhic, $c(\tau', \tau'') \subseteq c$ and $c(\gamma', \gamma'') \subseteq c$. It follows by definition that $\tau \in \cap(\mathcal{S}(\tau_1), \mathcal{S}(\tau_2))|_c$ and $\gamma \in \cap(\mathcal{S}(\gamma_1), \mathcal{S}(\gamma_2))|_c$.

It is then trivial to show that if $\tau \in \cap(\mathcal{S}(\tau_1), \mathcal{S}(\tau_2))|_c$ and $\gamma \in \cap(\mathcal{S}(\gamma_1), \mathcal{S}(\gamma_2))|_c$ then $g = (\tau, \gamma) \in \cap(\mathcal{S}(G_1), \mathcal{S}(G_2))|_c$. □

Given the nature of the constraint set C, we can efficiently compute the previous equation as two different J_1 and J_2 in $2^{\{1,...,|C|\}}$ often generate the same c, i.e.

$$c = \bigcap_{i \in J_1} c_i = \bigcap_{i \in J_2} c_i \tag{10}$$

Then, we can define the set C^* of all intersections of constraints in C.

Definition 4.6. Given the set of alternative constraints $C = \{c_1, ..., c_n\}$, the set C^* is the *set of all the possible intersections* of elements of the set C:

$$C^* = \{c(J) | J \in 2^{\{1,...,|C|\}}\} \tag{11}$$

where $c(J) = \bigcap_{i \in J} c_i$.

The previous lemmas and definitions are used to formulate the main theorem that can be used to build the algorithm for counting the subgraphs in common between two tDAGs and, then, computing the related kernel function.

Theorem 4.2. Given two tDAGs G_1 and G_2, the kernel $K(G_1, G_2)$ that counts the common subgraphs of the set $\mathcal{S}(G_1) \cap \mathcal{S}(G_2)$ follows this equation:

$$K(G_1, G_2) = \sum_{c \in C^*} K_S(\tau_1, \tau_2, c) K_S(\gamma_1, \gamma_2, c) N(c) \tag{12}$$

where

$$N(c) = \sum_{\substack{J \in 2^{\{1,...,|C|\}} \\ c = c(J)}} (-1)^{|J|-1} \tag{13}$$

and

$$K_S(\theta_1, \theta_2, c) = |\cap(\mathcal{S}(\theta_1), \mathcal{S}(\theta_2))|_c| \tag{14}$$

Proof:

Given Lemma 4.2, $K(G_1, G_2)$ can be written as:

$$K(G_1,G_2) = \left|\bigcup_{c \in C} \cap(\mathcal{S}(\tau_1),\mathcal{S}(\tau_2))|_c \times \cap(\mathcal{S}(\gamma_1),\mathcal{S}(\gamma_2))|_c\right| \tag{15}$$

The cardinality of the set can be computed using the inclusion-exclusion property, i.e.,

$$|A_1 \cup \cdots \cup A_n| = \sum_{J \in 2^{\{1,...,n\}}} (-1)^{|J|-1} |A_J| \tag{16}$$

where $2^{\{1,...,n\}}$ is the set of all the subsets of $\{1, \ldots, n\}$ and $A_J = \bigcap_{i \in J} A_i$. Given (15), (16), and (14), we can rewrite $K(G_1, G_2)$ as:

$$K(G_1, G_2) = \sum_{J \in 2^{\{1,...,|C|\}}} (-1)^{|J|-1} K_S(\tau_1,\tau_2,c(J)) K_S(\gamma_1,\gamma_2,c(J)) \tag{17}$$

Finally, defining $N(c)$ as in (13), equation (12) can be derived from equation (17). □

4.4. Enabling the efficient kernel function

The above idea for computing the kernel function is promising but we need to make it viable by describing the way we can determine efficiently the three main parts of equation (12): 1) the set of alternative constraints C (Sec. 4.4.2); 2) the set C^* of all the possible intersections of constraints in C (Sec. 4.4.3); and, finally, 3) the numbers $N(c)$ (Sec. 4.4.4). Before describing the above steps, we need to point out some properties of constraints and introduce a new operator.

4.4.1. Unification of constraints

We manipulated constraints as sets, but since they represent restrictions on bijective functions, they must be treated carefully. In particular, the union of two constraints may generate a semantically meaningless result. For example, the union of $c_1 = \{(\boxed{1},\boxed{1}),(\boxed{2},\boxed{2})\}$ and $c_2 = \{(\boxed{1},\boxed{2}),(\boxed{2},\boxed{1})\}$ would produce the set $c = c_1 \cup c_2 = \{(\boxed{1},\boxed{1}),(\boxed{2},\boxed{2}),(\boxed{1},\boxed{2}),(\boxed{2},\boxed{1})\}$ but c is clearly a contradictory and not valid constraint. Thus we introduce a more useful partial operator.

Definition 4.7. Unification ($\sqcup$): Given two constraints $c_1 = (p'_1, p''_1), \ldots, (p'_n, p''_n)$ and $c_2 = (q'_1, q''_1), \ldots, (q'_m, q''_m)$, their *unification* is $c_1 \sqcup c_2 = c_1 \cup c_2$ if $\nexists (p', p'') \in c_1, (q', q'') \in c_2 | p' = q'$ and $p'' \neq q''$ or vice versa; otherwise it is undefined and we write $c_1 \sqcup c_2 = \bot$.

4.4.2. Determining the set of alternative constraints

The first step of equation (12) is to determine the set of alternative constraints C. We can use the possibility of dividing tDAGs in two trees. We build C starting from sets C_τ and C_γ, that are respectively the constraints obtained from pairs of isomorphic extended trees $t_1 \in \mathcal{S}(\tau_1)$ and $t_2 \in \mathcal{S}(\tau_2)$, and the constraints obtained from pairs of isomorphic extended trees $t_1 \in \mathcal{S}(\gamma_1)$ and $t_2 \in \mathcal{S}(\gamma_2)$. The idea for an efficient algorithm is that we can compute the set C without explicitly looking at all the involved subgraphs. We instead use and combine the constraints derived from the comparison between the production rules of the extended trees. We can compute then C_τ with the productions of τ_1 and τ_2 and C_γ with the productions of γ_1 and γ_2. For example (see Fig. 2), focusing on the τ, the rule $NP\boxed{3} \rightarrow NN\boxed{2}NNS\boxed{3}$ of G_1 and $NP\boxed{4} \rightarrow NN\boxed{5}NNS\boxed{4}$ of G_2 generates the constraint $c = \{(\boxed{3},\boxed{4}),(\boxed{2},\boxed{5})\}$.

To express the above idea in a formal way, for each pair of nodes $n_1 \in \tau_1, n_2 \in \tau_2$ (the same holds when considering γ_1 and γ_2), we need to determine a set of constraints $LC = \{c_i | \exists t_1, t_2$ subtrees rooted in n_1 and n_2 respectively such that t_1 and t_2 are isomorphic according to $c_i\}$. This can be done by applying the procedure described in figure 5 to all pairs of nodes.

Although the procedure shows a recursive structure, adopting a dynamic programming technique, i.e. storing the results of the procedure in a persistent table, allows the number of executions to be limited to the number of node pairs, $|N_{\tau_1}| \times |N_{\tau_2}|$.

Once we have obtained the sets of local alternative constraints LC_{ij} for each node pair, we can simply merge the sets to produce the final set:

$$C_\tau = \bigcup_{\substack{1 \leq i \leq |N_{\tau_1}| \\ 1 \leq j \leq |N_{\tau_2}|}} LC_{ij}$$

Algorithm *Procedure* `getLC`(n', n'')

```
$LC \leftarrow \emptyset$
$c \leftarrow$ constraint according to which the productions in $n'$ and $n''$ are equivalent
IF no such constraint exists RETURN $\emptyset$
ELSE
    add $c$ to $LC$
    FORALL pairs of children $ch'_i, ch''_i$ of $n', n''$
        $LC_i \leftarrow$ getLC($ch'_i, ch''_i$)
        FORALL $c' \in LC_i$
            IF $c \sqcup c' \neq \perp$ add $c \sqcup c'$ to $LC$
    FORALL $c_i, c_j \in AC$ such that $i \neq j$
        IF $c_i \sqcup c_j \neq \perp$ add $c_i \sqcup c_j$ to $LC$
    RETURN $LC$
```

Figure 5. Algorithm for computing LC for a pair of nodes

The same procedure is applied to produce C_γ.

The alternative constraint set C is then obtained as $c' \sqcup c'' | c' \in C_\tau, c'' \in C_\gamma$, so that each constraint in C contains at least one of the constraints in C_τ and one of the constraints in C_γ. In the last step, we reduce the size of the final set. For this purpose, we remove from C all constraints c such that $\exists c' \supseteq c \in C$, since their presence is made redundant by the use of inclusion-exclusion property.

Lemma 4.3. The alternative constraint set C obtained by the above procedures satisfies the following two properties:

1. for each isomorphic sub-tDAG according to a constraint c, $\exists c' \in C$ such that $c \subseteq c'$;
2. $\nexists c', c'' \in C$ such that $c' \subset c''$ and $c' \neq \emptyset$.

Proof:
Property 2 is trivially assured by the last described step. As for property 1, let $G(t, g)$ be the isomorphic tDAG according to constraint c; then $\exists t_1 \in \mathcal{S}(\tau_1), t_2 \in \mathcal{S}(\tau_2), g_1 \in \mathcal{S}(\gamma_1), g_2 \in \mathcal{S}(\gamma_2)$ such that t, t_1, t_2 are isomorphic and g, g_1, g_2 are isomorphic and $c_t = f_\tau|_{A_1} \subseteq c$ and $c_t = g_\gamma|_{A_1} \subseteq c$ and $c_t \sqcup c_g = c$. By definition of LC, we have that $c_t \in LC_{ij}$ for some $n_i \in \tau_1, n_j \in \tau_2$ and $c_g \in LC_{kl}$ for some $n_k \in \gamma_1, n_l \in \gamma_2$. Thus $c_t \in C_\tau$ and $c_g \in C_\gamma$, and then $\exists c' \in C | c' \supseteq c_t \sqcup c_g = c$.

□

4.4.3. Determining the set C^*

The set C^* is defined as the set of all possible intersections of alternative constraints in C. Figure 6 presents the algorithm determining C^*. Due to the property (7) discussed in Sec. 4.3, we can empirically

```
Algorithm Build the set $C^*$ from the set $C$
    $C^+ \leftarrow C$ ; $C_1 \leftarrow C$ ; $C_2 \leftarrow \emptyset$
    WHILE $|C_1| > 1$
        FORALL $c' \in C_1$
            FORALL $c'' \in C_1$ such that $c' \neq c''$
                $c \leftarrow c' \cap c''$
                IF $c \notin C^+$ add $c$ to $C_2$
        $C^+ \leftarrow C^+ \cup C_2$ ; $C_1 \leftarrow C_2$; $C_2 \leftarrow \emptyset$
    $C^* \leftarrow C \cup C^+ \cup \{\emptyset\}$
```

Figure 6. Algorithm for computing C^*

demonstrate that the average complexity of the algorithm is not bigger than $O(|C|^2)$ but, again, the worst case complexity is exponential.

4.4.4. Determining the values of $N(c)$

The factor $N(c)$ (equation 13) represents the number of times the constraint c is considered in the sum of equation 12, keeping into account the sign of the corresponding addend. To determine its value, we exploit the following property.

Lemma 4.4. For the factor $N(C)$ the following recursive equation holds:

$$N(c) = 1 - \sum_{\substack{c' \in C^* \\ c' \supset c}} N_{c'} \tag{18}$$

Proof:
Let us call $N_n(c)$ the cardinality of the set $\{J \in 2^{\{1,\ldots,|C|\}}.c(J) = c, |J| = n\}$. We can rewrite equation (13) as:

$$N(c) = \sum_{n=1}^{|C|} (-1)^{n-1} N_n(c) \tag{19}$$

We note that the following properties hold:

$$\begin{aligned} N_n(c) &= |\{J \in 2^{\{1,\ldots,|C|\}}.c(J) = c, |J| = n\}| = \\ &= |\{J \in 2^{\{1,\ldots,|C|\}}.c(J) \supseteq c, |J| = n\}| - \\ &- |\{J \in 2^{\{1,\ldots,|C|\}}.c(J) \supset c, |J| = n\}| \end{aligned}$$

Now let x_c be the number of alternative constraints which include the constraint c, i.e. $x_c = |\{c' \in C.c' \supseteq c\}|$. Then, by combinatorial properties and by the definition of $N_n(c)$, the previous equation becomes:

$$N_n(c) = \binom{x_c}{n} - \sum_{\substack{c' \in C^* \\ c' \supset c}} N_n(c') \tag{20}$$

From equations 19 and 20, it follows that $N(c)$ can be written as:

$$\begin{aligned} N(c) &= \sum_{n=1}^{|C|} (-1)^{n-1} (\binom{x_c}{n} - \sum_{\substack{c' \in C^* \\ c' \supset c}} N_n(c')) = \\ &= \sum_{n=1}^{|C|} (-1)^{n-1} \binom{x_c}{n} - \sum_{n=1}^{|C|} (-1)^{n-1} \sum_{\substack{c' \in C^* \\ c' \supset c}} N_n(c') = \\ &= \sum_{n=0}^{x_c} (-1)^{n-1} \binom{x_c}{n} + \binom{x_c}{0} - \\ &- \sum_{\substack{c' \in C^* \\ c' \supset c}} \sum_{n=1}^{|C|} (-1)^{n-1} N_n(c') \end{aligned}$$

We now observe that, exploiting the binomial theorem, we can write:

$$\sum_{K=0}^{N} (-1)^K \binom{N}{K} = \sum_{K=0}^{N} (1)^{N-K} (-1)^K \binom{N}{K} = (1 - 1)^N = 0$$

thus

$$\sum_{n=0}^{x_c} (-1)^{n-1} \binom{x_c}{n} = -\sum_{n=0}^{x_c} (-1)^n \binom{x_c}{n} = 0$$

Finally, since $\binom{x_c}{0} = 1$, and according to the definition of $N(c)$ in Equation 19, we can derive the property in Eq. (18), i.e.:

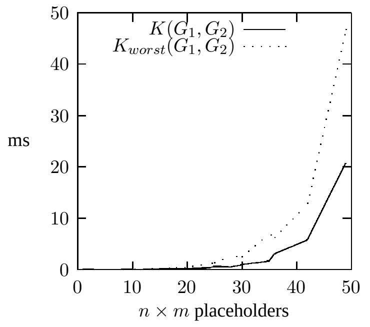
$$N(c) = 1 - \sum_{\substack{c' \in C^* \\ c' \supset c}} N_{c'}$$

□

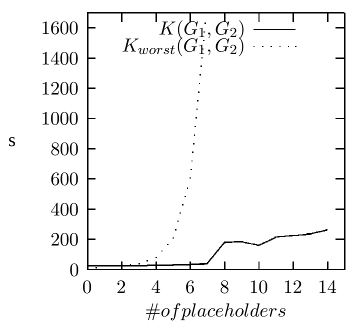
This recursive formulation of the equation allows us to easily determine the value of $N(c)$ for every c belonging to C^*.

5. Worst-case complexity and average computing time analysis

We can now both analyze the worst-case complexity and the average computing time of the algorithm we proposed with the Theorem 4.2. The computation of Equation 12 strongly depends on the cardinality of C and the related cardinality of C^*. The worst-case complexity is $O(|C^*|n^2|C|)$ where n is the



(a) Mean execution time in milliseconds (ms) of the two algorithms wrt. $n \times m$ where n and m are the number of placeholders of the two tDAGs



(b) Total execution time in seconds (s) of the training phase on RTE2 wrt. different numbers of allowed placeholders

Figure 7. Comparison of the execution times

cardinality of the node sets of the extended trees. Then, the worst-case computational complexity is still exponential with respect to the size of the set of anchors of the two tDAGs, A_1 and A_2. In the worst-case C is equal to $\mathcal{F}_{(A_1,A_2)}$, i.e., the set of the possible correspondences between the nodes A_1 and A_2. This is a combinatorial set. Then, the worst-case complexity is $O(e^{|A|}n^2)$.

Yet, there are some hints that suggests that the average case complexity [44] and the average computing time can be promising. The set C is generally very small with respect to the worst case. It happens that $|C| << |\mathcal{F}_{(A_1,A_2)}|$ where $|\mathcal{F}_{(A_1,A_2)}|$ is the worst case. For example, in the case of P_1 and P_2, the cardinality of $C = \{\{(\boxed{1},\boxed{1})\},\{(\boxed{1},\boxed{3}),(\boxed{3},\boxed{4}),(\boxed{2},\boxed{5})\}\}$ is extremely smaller than the one of $\mathcal{F}_{(A_1,A_2)} = \{\{(\boxed{1},\boxed{1}), (\boxed{2},\boxed{2}), (\boxed{3},\boxed{3})\}, \{(\boxed{1},\boxed{2}), (\boxed{2},\boxed{1}), (\boxed{3},\boxed{3})\}, \{(\boxed{1},\boxed{2}), (\boxed{2},\boxed{3}), (\boxed{3},\boxed{1})\}, ..., \{(\boxed{1},\boxed{3}), (\boxed{2},\boxed{4}), (\boxed{3},\boxed{5})\}\}$. Moreover, the set C^* is extremely smaller than $2^{\{1,\dots,|C|\}}$ due to the above property (7).

We estimated the behavior of the algorithms on a large distribution of cases. We compared the computational times of our algorithm with the worst-case, i.e., $C = \mathcal{F}_{(A_1,A_2)}$. We refer to the our algorithm as K and to the worst case as K_{worst} We implemented both algorithms $K(G_1,G_2)$ and $K_{worst}(G_1,G_2)$ in SVMs and we experimented with both implementations on the same machine.

For the first set of experiments, the source of examples is the one of the recognizing textual entailment challenge, i.e., RTE2 [2]. The dataset of the challenge has 1,600 sentence pairs. To derive tDAGs for sentence pairs, we used the following resources:

- The Charniak parser [4] and the `morpha` lemmatiser [28] to carry out the syntactic and morphological analysis. These have been used to build the initial syntactic trees.
- The `wn::similarity` package [36] to compute the Jiang&Conrath (J&C) distance [23] as in [9] for finding relations between similar words in order to find co-indexes between the H and the T trees.

The computational cost of both $K(G_1,G_2)$ and $K_{worst}(G_1,G_2)$ depends on the number of place-

Kernel	**Accuracy**	**Used training examples**	**Support Vectors**
K_{max}	59.32	4223	4206
K	60.04	4567	4544

Table 1. Comparative performances of K_{max} and K

holders $n = |A_1|$ of G_1 and on $m = |A_2|$ the number of placeholders of G_2. Then, in the first experiment we want to determine the relation between the computational time and the factor $n \times m$. The results are reported in Figure 7(a) where the computation times are plotted with respect to $n \times m$. Each point in the curve represents the average execution time for the pairs of instances having $n \times m$ placeholders. As expected, the computation of the function K is more efficient than the computation K_{worst}. The difference between the two execution times increases with $n \times m$.

We then performed a second experiment that determines the relation of the total execution time with the maximum number of placeholders in the examples. This is useful to estimate the behavior of the algorithm with respect to its application in learning models. Using the RTE2 data, we artificially build different versions with increasing number of placeholders, i.e. with one placeholder, two placeholders, three placeholders and so on at most in each pair. In other words, the number of pairs is the same whereas the maximal number of placeholders changes. The results are reported in Figure 7(b) where the execution time of the training phase (in seconds (s)) is plotted for each different set. We see that the computation of K_{worst} looks exponential with respect to the number of placeholders and it becomes intractable after 7 placeholders. The plot associated with the computation of K is instead flatter. This can be explained as the computation of K is related to the real alternative constraints that appear in the dataset. Therefore, the computation time of K is extremely shorter than the one of K_{worst}.

6. Discussion

To better show the benefit of our approach in terms of efficiency and effectiveness, we compare it with the algorithm we presented in [31]. We will hereafter call the latter algorithm K_{max}; this induces an approximation of FOR and it is not difficult to demonstrate that $K_{max}(G_1, G_2) \leq K(G_1, G_2)$. The K_{max} approximation is based on maximization over the set of possible correspondences of the placeholders, i.e.:

$$K_{max}(G_1, G_2) = \max_{c \in \mathcal{F}_{(A_1, A_2)}} K_S(\tau_1, \tau_2, c) K_S(\gamma_1, \gamma_2, c) \tag{21}$$

where $\mathcal{F}_{(A_1,A_2)}$ are all the possible correspondences between the nodes A_1 and A_2 of the two tDAGs as the one presented in Section 4.3. This formulation has always the computational complexity worst case of our method (K_{max} behaves exactly as K_{worst}).

We showed that K_{max} is very accurate for RTE [2] but, since K computes a slightly different similarity function, we need to show that its accuracy is comparable with K_{max}. Thus, we performed an experiment by using all the data derived from RTE1, RTE2, and RTE3 for training (i.e., 4567 training examples) and the RTE-4 data for testing (i.e., 1000 testing examples). The results are reported in Table 1. The table shows that the accuracy of K is higher than the accuracy of K_{max}. Our explanation for this result is that (a) K_{max} is an approximation of K and (b) K can use sentence pairs with more than 7 placeholders, i.e. the complete training set as the third column of the table shows.

7. Conclusions

In this paper, we have presented a feature space model for the representation of first order rules for automatically learning textual entailment from data. The model uses *tripartite directed acyclic graphs* ($tDAG$s) to describe syntactic and semantic relations contained in pairs of sentences. More specifically, the rules can be learned from examples using kernel-machines (e.g. SVMs) by measuring the similarity between $tDAG$s in the FOR space, which in our case is the space of all possible subgraphs.

Since previous work only proposed algorithms for graph matching providing either approximated solutions or exact solutions with an exponential computational complexity, we designed a novel and efficient algorithm to efficiently compute the number of shared subgraphs. We proved that our approach leads to a valid kernel and we have empirically shown that it outperforms previous approaches for recognizing textual entailment in terms of accuracy and efficiency. In the future, we would like to investigate if our algorithm can be generalized for the computation of the similarity of two general directed acyclic graphs.

References

[1] Aizerman, A., Braverman, E. M., Rozoner, L. I.: Theoretical foundations of the potential function method in pattern recognition learning, *Automation and Remote Control*, **25**, 1964, 821–837.

[2] Bar-Haim, R., Dagan, I., Dolan, B., Ferro, L., Giampiccolo, D., Magnini, Bernardo Szpektor, I.: The Second PASCAL Recognising Textual Entailment Challenge, in: *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, Venice, Italy, 2006.

[3] Carpenter, B.: *The Logic of Typed Feature Structures*, Cambridge University Press, Cambridge, England, 1992.

[4] Charniak, E.: A Maximum-Entropy-Inspired Parser, *Proc. of the 1st NAACL*, Seattle, Washington, 2000.

[5] Chierchia, G., McConnell-Ginet, S.: *Meaning and Grammar: An introduction to Semantics*, MIT press, Cambridge, MA, 2001.

[6] Chomsky, N.: *Aspect of Syntax Theory*, MIT Press, Cambridge, Massachussetts, 1957.

[7] Collins, M.: Head-Driven Statistical Models for Natural Language Parsing, *Comput. Linguist.*, **29**(4), 2003, 589–637, ISSN 0891-2017.

[8] Collins, M., Duffy, N.: New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron, in: *Proceedings of ACL02*, 2002.

[9] Corley, C., Mihalcea, R.: Measuring the Semantic Similarity of Texts, in: *Proc. of the ACL Workshop on Empirical Modeling of Semantic Equivalence and Entailment*, Association for Computational Linguistics, Ann Arbor, Michigan, June 2005, 13–18.

[10] Cortes, C., Vapnik, V.: Support Vector Networks, *Machine Learning*, **20**, 1995, 1–25.

[11] Cristianini, N., Shawe-Taylor, J.: *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*, Cambridge University Press, March 2000, ISBN 0521780195.

[12] Dagan, I., Glickman, O.: Probabilistic Textual Entailment: Generic Applied Modeling of Language Variability, *Proceedings of the Workshop on Learning Methods for Text Understanding and Mining*, Grenoble, France, 2004.

[13] Dagan, I., Glickman, O., Magnini, B.: The PASCAL Recognising Textual Entailment Challenge, *LNAI 3944: MLCW 2005* (Q.-C. et al., Ed.), Springer-Verlag, Milan, Italy, 2006.

[14] Dang, H. T.: Overview of DUC 2005, *Proceedings of the 2005 Document Understanding Workshop*, 2005.

[15] Eisner, J.: Learning Non-Isomorphic Tree Mappings for Machine Translation, *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL), Companion Volume*, Sapporo, July 2003.

[16] Gärtner, T.: A Survey of Kernels for Structured Data, *SIGKDD Explorations*, 2003.

[17] Gildea, D., Jurafsky, D.: Automatic Labeling of Semantic Roles, *Computational Linguistics*, **28**(3), 2002, 245–288.

[18] Grinberg, D., Lafferty, J., Sleator, D.: A robust parsing algorithm for link grammar, *4th International workshop on parsing tecnologies*, Prague, 1996.

[19] Haghighi, A. D., Ng, A. Y., Manning, C. D.: Robust textual inference via graph matching, *HLT '05: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Morristown, NJ, USA, 2005.

[20] Harabagiu, S., Hickl, A.: Methods for Using Textual Entailment in Open-Domain Question Answering, *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Sydney, Australia, July 2006.

[21] Harabagiu, S., Hickl, A., Lacatusu, F.: Satisfying information needs with multi-document summaries, *Information Processing & Management*, **43**(6), 2007, 1619 – 1642, ISSN 0306-4573, Text Summarization.

[22] Hickl, A., Williams, J., Bensley, J., Roberts, K., Rink, B., Shi, Y.: Recognizing Textual Entailment with LCCs GROUNDHOG System, *Proceedings of the Second PASCAL Recognizing Textual Entailment Challenge* (B. Magnini, I. Dagan, Eds.), Springer-Verlag, Venice, Italy, 2006.

[23] Jiang, J. J., Conrath, D. W.: Semantic similarity based on corpus statistics and lexical taxonomy, in: *Proc. of the 10th ROCLING*, Tapei, Taiwan, 1997, 132–139.

[24] Köbler, J., Schöning, U., Torán, J.: *The graph isomorphism problem: its structural complexity*, Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993, ISBN 0-8176-3680-3.

[25] Lin, D., Pantel, P.: DIRT: discovery of inference rules from text, *Knowledge Discovery and Data Mining*, 2001.

[26] Marcus, M. P., Santorini, B., Marcinkiewicz, M. A.: Building a Large Annotated Corpus of English: The Penn Treebank, *Computational Linguistics*, **19**, 1993, 313–330.

[27] de Marneffe, M.-C., MacCartney, B., Grenager, T., Cer, D., Rafferty, A., D. Manning, C.: Learning to distinguish valid textual entailments, *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, Venice, Italy, 2006.

[28] Minnen, G., Carroll, J., Pearce, D.: Applied morphological processing of English, *Natural Language Engineering*, **7**(3), 2001, 207–223.

[29] Moschitti, A.: A study on Convolution Kernels for Shallow Semantic Parsing, *proceedings of the ACL*, Barcelona, Spain, 2004.

[30] Moschitti, A., Pighin, D., Basili, R.: Tree Kernels for Semantic Role Labeling, *Computational Linguistics*, **34**(2), 2008, 193–224.

[31] Moschitti, A., Zanzotto, F. M.: Fast and Effective Kernels for Relational Learning from Texts, in: *Proceedings of the International Conference of Machine Learning (ICML)*, Corvallis, Oregon, 2007.

[32] MUC-7: Proceedings of the Seventh Message Understanding Conference (MUC-7), *Columbia, MD*, Morgan Kaufmann, 1997.

[33] Nivre, J., Hall, J., Kübler, S., McDonald, R., Nilsson, J., Riedel, S., Yuret, D.: The CoNLL 2007 Shared Task on Dependency Parsing, in: *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, Association for Computational Linguistics, Prague, Czech Republic, June 2007, 915–932.

[34] Pantel, p., Pennacchiotti, M.: Espresso: A Bootstrapping Algorithm for AutomaticallyHarvesting Semantic Relations, *Proceedings of the 21st Coling and 44th ACL*, Sydney, Australia, July 2006.

[35] Peas, A., lvaro Rodrigo, Verdejo, F.: Overview of the Answer Validation Exercise 2007., *CLEF* (C. Peters, V. Jijkoun, T. Mandl, H. Mller, D. W. Oard, A. Peas, V. Petras, D. Santos, Eds.), 5152, Springer, 2007, ISBN 978-3-540-85759-4.

[36] Pedersen, T., Patwardhan, S., Michelizzi, J.: WordNet::Similarity - Measuring the Relatedness of Concepts, in: *Proc. of 5th NAACL*, Boston, MA, 2004.

[37] Pollard, C., Sag, I.: *Head-driven Phrase Structured Grammar*, Chicago CSLI, Stanford, 1994.

[38] Quinlan, J.: *C4:5:programs for Machine Learning*, Morgan Kaufmann, San Mateo, 1993.

[39] Raina, R., Haghighi, A., Cox, C., Finkel, J., Michels, J., Toutanova, K., MacCartney, B., de Marneffe, M.-C., Christopher, M., Ng, A. Y.: Robust Textual Inference Using Diverse Knowledge Sources, *Proceedings of the 1st Pascal Challenge Workshop*, Southampton, UK, 2005.

[40] Ramon, J., Gärtner, T.: Expressivity versus Efficiency of Graph Kernels, *First International Workshop on Mining Graphs, Trees and Sequences*, 2003.

[41] Suzuki, J., Hirao, T., Sasaki, Y., Maeda, E.: Hierarchical Directed Acyclic Graph Kernel: Methods for Structured Natural Language Data, *In Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, 2003.

[42] Tesniere, L.: *Elements de syntaxe structural*, Klincksiek, Paris, France, 1959.

[43] Voorhees, E. M.: The TREC question answering track, *Nat. Lang. Eng.*, **7**(4), 2001, 361–378, ISSN 1351-3249.

[44] Wang, J.: Average-case computational complexity theory, 1997, 295–328.

[45] Wang, R., Neumann, G.: Recognizing Textual Entailment Using a Subsequence Kernel Method, *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-07), July 22-26, Vancouver, Canada*, 2007.

[46] Zanzotto, F. M., Moschitti, A.: Automatic Learning of Textual Entailments with Cross-Pair Similarities, in: *Proceedings of the 21st Coling and 44th ACL*, Sydney, Australia, July 2006, 401–408.

[47] Zanzotto, F. M., Moschitti, A.: *Experimenting a "General Purpose" Textual Entailment Learner in AVE*, vol. 4730, Springer, DEU, 2007, ISBN 978-3-540-74998-1, 510–517.

[48] Zanzotto, F. M., Pennacchiotti, M., Moschitti, A.: A Machine Learning Approach to Textual Entailment Recognition, *NATURAL LANGUAGE ENGINEERING*, **15-04**, 2009, 551–582, ISSN 1351-3249.

[49] Zanzotto, F. M., Pennacchiotti, M., Pazienza, M. T.: Discovering asymmetric entailment relations between verbs using selectional preferences, *Proceedings of the 21st Coling and 44th ACL*, Sydney, Australia, July 2006.