Efficient kernels for sentence pair classification

Fabio Massimo Zanzotto
DISP
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

Abstract

In this paper, we propose a novel class of graphs, the tripartite directed acyclic graphs (tDAGs), to model first-order rule feature spaces for sentence pair classification. We introduce a novel algorithm for computing the similarity in first-order rewrite rule feature spaces. Our algorithm is extremely efficient and, as it computes the similarity of instances that can be represented in explicit feature spaces, it is a valid kernel function.

1 Introduction

Natural language processing models are generally positive combinations between linguistic models and automatically learnt classifiers. As trees are extremely important in many linguistic theories, a large amount of works exploiting machine learning algorithms for NLP tasks has been developed for this class of data structures (Collins and Duffy, 2002; Moschitti, 2004). These works propose efficient algorithms for determining the similarity among two trees in tree fragment feature spaces.

Yet, some NLP tasks such as textual entailment recognition (Dagan and Glickman, 2004; Dagan et al., 2006) and some linguistic theories such as HPSG (Pollard and Sag, 1994) require more general graphs and, then, more general algorithms for computing similarity among graphs. Unfortunately, algorithms for computing similarity among two general graphs in term of common subgraphs are still exponential (Ramon and Gärtner, 2003). In these cases, approximated algorithms have been proposed. For example, the one proposed in (Gärtner, 2003) counts the number of subpaths in common. The same happens for the one proposed in (Suzuki et al., 2003) that is applicable to a particular class of graphs, i.e. the hierarchical directed acyclic graphs. These algorithms do not compute the number of subgraphs in common between two graphs. Then, these algorithms approximate the feature spaces we need in these NLP tasks. For computing similarities in these feature spaces, we have to investigate if we can define a particular class of graphs for the class of tasks we want to solve. Once we focused the class of graph, we can explore efficient similarity algorithms.

A very important class of graphs can be defined for tasks involving sentence pairs. In these cases, an important class of feature spaces is the one that represents first-order rewrite rules. For example, in textual entailment recognition (Dagan et al., 2006), we need to determine whether a text T implies a hypothesis H, e.g., whether or not "*Farmers feed cows animal extracts*" entails "*Cows eat animal extracts*" (T_1, H_1). If we want to learn textual entailment classifiers, we need to exploit first-order rules hidden in training instances. To positively exploit the training instance "*Pediatricians suggest women to feed newborns breast milk*" entails "*Pediatricians suggest that newborns eat breast milk*" (T_2, H_2) for classifying the above example, learning algorithms should learn that the two instances hide the first-order rule $\rho = feed\boxed{Y}\boxed{Z} \rightarrow \boxed{Y}eat\boxed{Z}$. The first-order rule feature space, introduced by (Zanzotto and Moschitti, 2006), gives high performances in term of accuracy for textual entailment recognition with respect to other features spaces.

In this paper, we propose a novel class of graphs, the tripartite directed acyclic graphs (tDAGs), that model first-order rule feature spaces and, using this class of graphs, we introduce a novel algorithm for computing the similarity in first-order rewrite rule feature spaces. The possibility of explicitly representing the first-order feature space as subgraphs of tDAGs makes the derived similarity function a valid kernel. With respect to the algorithm proposed in (Moschitti and Zanzotto, 2007), our algorithm is more efficient

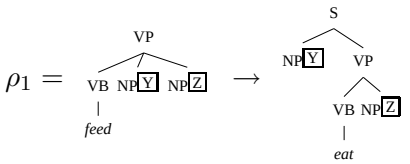
and it is a valid kernel function.

The paper is organized as follows. In Sec. 2, we firstly describe tripartite directed acyclic graphs (tDAGs) to model first-order feature (FOR) spaces. In Sec. 3, we then present the related work. In Sec. 4, we introduce the similarity function for these FOR spaces. This can be used as kernel function in kernel-based machines (e.g., support vector machines (Cortes and Vapnik, 1995)). We then introduce our efficient algorithm for computing the similarity among tDAGs. In Sec. 5, we analyze the computational efficiency of our algorithm showing that it is extremely more efficient than the algorithm proposed in (Moschitti and Zanzotto, 2007). Finally, in Sec. 6, we draw conclusions and plan the future work.

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

As first step, we want to define the *tripartite directed acyclic graphs* ($tDAGs$). This is an extremely important class of graphs for the first-order rule feature spaces we want to model. We want here to intuitively show that, if we model first-order rules and sentence pairs as $tDAG$s, determining whether or not a sentence pair can be unified with a first-order rewrite rule is a graph matching problem. This intuitive idea helps in determining our efficient algorithm for exploiting first-order rules in learning examples.

To illustrate the above idea we will use an example based on the above rule ρ= $feed$ Y Z $\rightarrow$ Y eat 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:



As in the case of feature structures (Carpenter, 1992), we can observe this rule as a graph. As we are not interested in the variable names but we need to know the relation between the right hand side and the left hand side of the rule, we can substitute each variable with an unlabelled node. We then connect tree nodes having variables with the corresponding unlabelled node. The result is a graph as the one in Fig. 1. The variables Y and Z are represented by the unlabelled nodes between the trees.

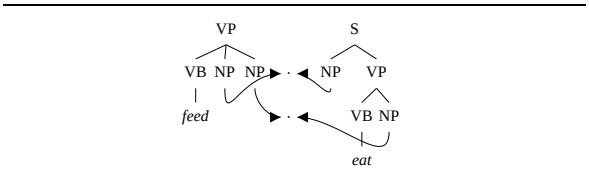


Figure 1: A simple rewrite rule seen as a graph

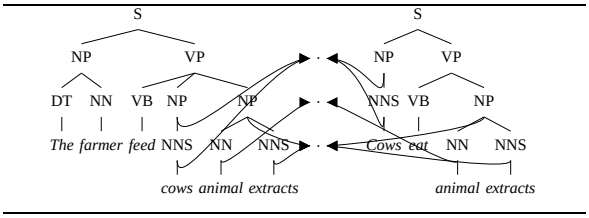


Figure 2: A sample pair seen as a graph

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 Fig. 2). We can link words using anchoring methods as in (Raina et al., 2005). These links can then be propagated in the syntactic tree using semantic heads of the constituents (Pollard and Sag, 1994). 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 Fig. 2.

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 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 Fig. 2 has been useful to show that the unification of a rule and a sentence pair is a graph matching problem. Yet, it is complex to follow. 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 a $(x, y) \in E_{A_t}$ and a $(z, y) \in E_{A_g}$ as boxes $\boxed{y}$ respectively on the node x and on the node z. These nodes will then appear as $L(x)\boxed{y}$ and $L(z)\boxed{y}$, e.g., NP$\boxed{1}$. The name y is not a label but a placeholder representing an unlabelled node. This representation is used for rules and for sentence pairs. The sentence pair in Fig. 2 is then represented as reported in Fig. 3.

3 Related work

Automatically learning classifiers for sentence pairs is extremely important for applications like textual entailment recognition, question answering, and machine translation.

In textual entailment recognition, it is not hard to see graphs similar to tripartite directed acyclic graphs as ways of extracting features from examples to feed automatic classifiers. Yet, these graphs are generally not tripartite in the sense described in the previous section and they are not used to extract features representing first-order rewrite rules. In (Raina et al., 2005; Haghighi et al., 2005; Hickl et al., 2006), two connected graphs representing the two sentences s_1 and s_2 are used to compute distance features, i.e., features representing the distance between s_1 and s_2. 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 (de Marneffe et al., 2006), first-order rewrite rule feature spaces have been explored. Yet, these spaces are extremely small. Only some features representing first-order rules have been explored. Pairs of graphs are used here to determine if a feature is active or not, i.e., the rule fires or not. A larger feature space of rewrite rules has been implicitly explored in (Wang and Neumann, 2007) but this work considers only ground rewrite rules.

In (Zanzotto and Moschitti, 2006), tripartite directed acyclic graphs are implicitly introduced and exploited to build first-order rule feature spaces. Yet, both in (Zanzotto and Moschitti, 2006) and in (Moschitti and Zanzotto, 2007), the model proposed has two major limitations: it can represent rules with less than 7 variables and the proposed kernel is not a completely valid kernel as it uses the max function.

In machine translation, some methods such as (Eisner, 2003) learn graph based rewrite rules for generative purposes. Yet, the method presented in (Eisner, 2003) can model first-order rewrite rules only with a very small amount of variables, i.e., two or three variables.

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

In this section, we present our idea for an efficient algorithm for exploiting first-order rule feature spaces. In Sec. 4.1, we firstly define the similarity function, i.e., the kernel $K(G_1, G_2)$, that we need to determine for correctly using first-order rules feature spaces. This kernel is strongly based on the isomorphism between graphs. A relevant idea of this paper is the observation that we can define an efficient way to detect the isomorphism between the tDAGs (Sec. 4.2). This algorithm exploits the efficient algorithms of tree isomorphism as the one implicitly used in (Collins and Duffy, 2002). After describing the isomorphism between tDAGs, We can present the idea of our efficient algorithm for computing $K(G_1, G_2)$ (Sec. 4.3). We introduce the algorithms to make it a viable solution (Sec. 4.4). Finally, in Sec. 4.5, we report the kernel computation we compare against presented by (Zanzotto and Moschitti, 2006; Moschitti and Zanzotto, 2007).

4.1 Kernel functions over first-order rule feature spaces

The first-order rule feature space we want to model is huge. If we use kernel-based machine learning models such as SVM (Cortes and Vapnik, 1995), we can implicitly define the space by defining its similarity functions, i.e., its kernel functions. We firstly introduce the first-order rule feature space and we then define the prototypical kernel function over this space.

The first-order rule feature space (FOR) is in general the space of all the possible first-order

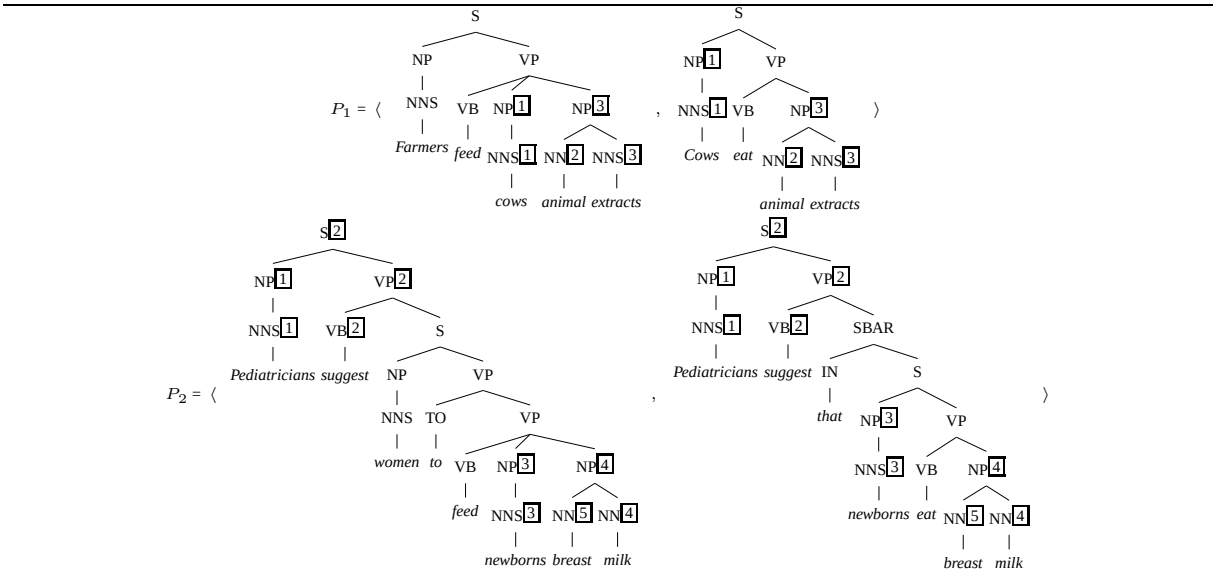
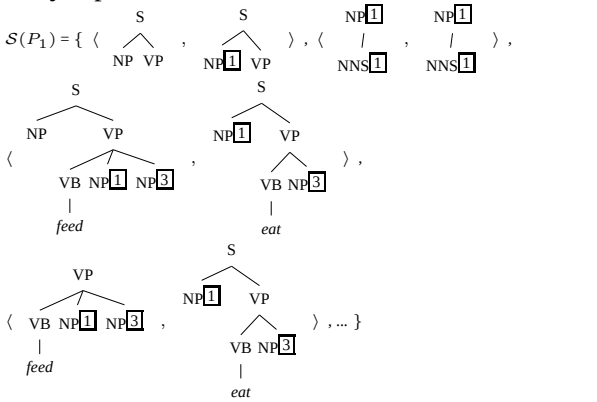
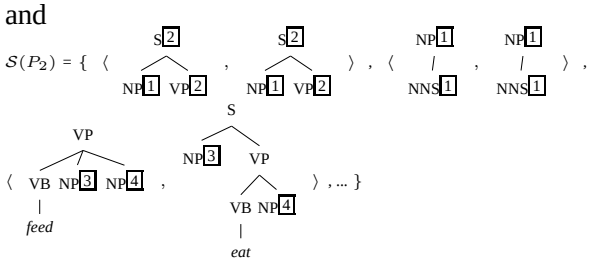


Figure 3: Two tripartite DAGs

rules defined as tDAGs. Within this space it is possible to define the function $\mathcal{S}(G)$ that determines all the possible active features of the tDAG G in FOR. The function $\mathcal{S}(G)$ determines all the possible and meaningful subgraphs of G. We want that these subgraphs represent first-order rules that can be matched with the pair G. Then, meaningful subgraphs of $G = (\tau, \gamma)$ are graphs as (t, g) where t and g are subtrees of τ and γ. For example, the subgraphs of P_1 and P_2 in Fig. 3 are hereafter partially represented:



and



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

$$K(G_1, G_2) = |\mathcal{S}(G_1) \cap \mathcal{S}(G_2)| \qquad (1)$$

This is very simple to write and it is in principle correct. A graph g in the intersection $\mathcal{S}(G_1) \cap \mathcal{S}(G_2)$ is a graph that belongs to both $\mathcal{S}(G_1)$ and $\mathcal{S}(G_2)$. Yet, this hides a very important fact: 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 superficially compare graphs to determine that ρ_1 belongs both to $\mathcal{S}_1$ and $\mathcal{S}_2$. We need to use the correct property for $g_1 = g_2$, i.e., the *isomorphism* between two graphs. We can call the operator $Iso(g_1, g_2)$. When two graphs verify the property $Iso(g_1, g_2)$, both g_1 and g_2 can be taken as the graph g representing the two graphs. Detecting $Iso(g_1, g_2)$ has an exponential complexity (Köbler et al., 1993).

This complexity of the intersection operator between sets of graphs deserves a different way to represent the operation. We will use the same symbol but we will use the prefix notation. The operator is hereafter re-defined:

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

4.2 Isomorphism between tDAGs

As isomorphism between graphs is an essential activity for learning from structured data, we here review its definition and we adapt it to tDAGs.

We then observe that isomorphism between two tDAGs can be divided in two sub-problems:

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

In general, two tDAGs, $G_1 = (N_1, E_1)$ 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 (Köbler et al., 1993). 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 a better algorithm. There is an efficient algorithm for computing isomorphism between trees (as the one implicitly used in (Collins and Duffy, 2002)).

Given two tDAGs $G_1 = (\tau_1, \gamma_1)$ and $G_2 = (\tau_2, \gamma_2)$ the isomorphism problem can be divided in 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 on the sets A_1 and A_2. The two constraints are defined as $c(\tau_1, \tau_2) = \{(n, f_\tau(n)) | n \in A_1\}$ and $c(\gamma_1, \gamma_2) = \{(n, f_\gamma(n)) | n \in A_1\}$. Two partially isomorphic tDAGs are isomorphic if the constraints match, i.e., $c(\tau_1, \tau_2) = c(\gamma_1, \gamma_2)$.

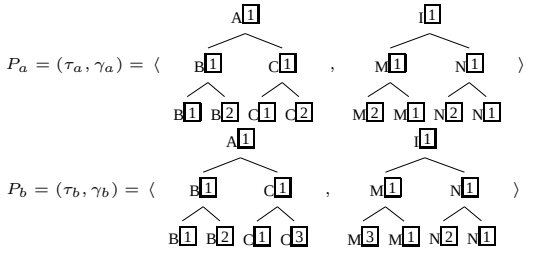


Figure 5: Simple non-linguistic tDAGs

For example, the third pair of $\mathcal{S}(P_1)$ and the second 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 fourth pair of $\mathcal{S}(P_1)$ and the third pair of $\mathcal{S}(P_2)$ generate $c_2 = \{(\boxed{1}, \boxed{1})\}$

4.3 General idea for an efficient kernel function

As above discussed, 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 Sec. 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 the two set of all the tDAGs meeting the constraint, we can detect the *partial isomorphism*. We split each pair of tDAGs in the four extended trees and we determine if these extended trees are compatible.

We introduce this innovative method to compute the kernel $K(G_1, G_2)$ in the FOR space 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 Fig. 5), and the subgraph function $\widetilde{\mathcal{S}}(\theta)$. 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 are likely to be generated when detecting the *partial isomorphism*. For P_a and P_b, this set is $C = \{c_1, c_2\} =$

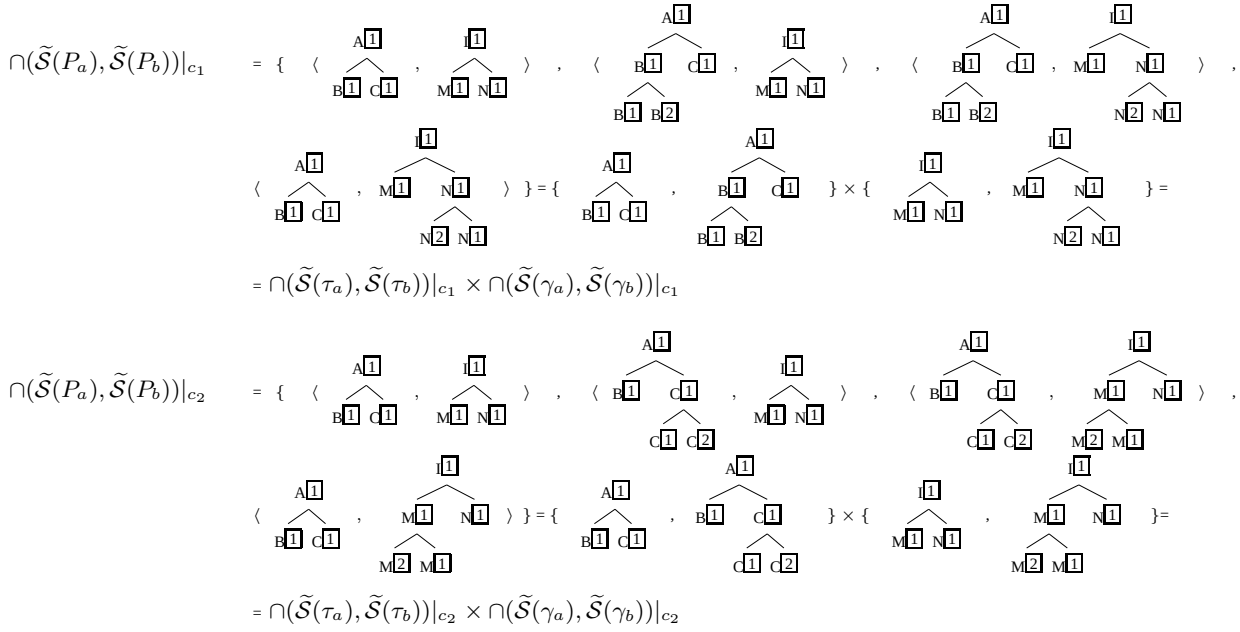


Figure 4: Intuitive idea for the kernel computation

$\{\{([\boxed{1},\boxed{1}),(\boxed{2},\boxed{2})\},\{([\boxed{1},\boxed{1}),(\boxed{2},\boxed{3})\}\}$. We can then determine the kernel $K(P_a, P_b)$ as:

$$K(P_a,P_b) = |\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))| = \left|\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_1} \bigcup \cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_{c_2}\right|$$

where $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c$ are the common subgraphs that meet the constraint c. A tDAG $g' = (\tau', \gamma')$ in $\widetilde{\mathcal{S}}(P_a)$ is in $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c$ if $g'' = (\tau'', \gamma'')$ in $\widetilde{\mathcal{S}}(P_b)$ exists, 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., $c' \subseteq c$. 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 $c' = \{([\boxed{1},\boxed{1})\}$ is a subset of c_1.

Observing the kernel computation in this way is important. Elements in $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c$ already satisfy the property of *constraint compatibility*. We only need to determine if the *partially isomorphic* properties hold for elements in $\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c$. Then, we can write the following equivalence:

$$\cap(\widetilde{\mathcal{S}}(P_a),\widetilde{\mathcal{S}}(P_b))|_c = \cap(\widetilde{\mathcal{S}}(\tau_a),\widetilde{\mathcal{S}}(\tau_b))|_c \times \cap(\widetilde{\mathcal{S}}(\gamma_a),\widetilde{\mathcal{S}}(\gamma_b))|_c \quad (2)$$

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 $\cap(\widetilde{\mathcal{S}}(\tau_a),\widetilde{\mathcal{S}}(\tau_b)) \times \cap(\widetilde{\mathcal{S}}(\gamma_a),\widetilde{\mathcal{S}}(\gamma_b))$.

The equivalence (2) 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 $\cap(\widetilde{\mathcal{S}}(\gamma_a),\widetilde{\mathcal{S}}(\gamma_b))|_c$. These 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 $K_S(\theta_1, \theta_2)$ introduced in (Collins and Duffy, 2002)).

4.3.2 Formal definition

Given the idea of the previous section, it is easy to demonstrate that the kernel $K(G_1, G_2)$ can be written as follows:

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

where C is set of alternative constraints and $\cap(\mathcal{S}(\theta_1),\mathcal{S}(\theta_2))|_c$ are all the common extended trees compatible with the constraint c.

We can compute the above kernel using the inclusion-exclusion property, i.e.,

$$|A_1 \cup \cdots \cup A_n| = \sum_{J\in 2^{\{1,\ldots,n\}}} (-1)^{|J|-1}|A_J| \quad (3)$$

where $2^{\{1,\ldots,n\}}$ is the set of all the subsets of $\{1, \ldots, n\}$ and $A_J = \bigcap_{i\in J} A_i$.

To describe the application of the inclusion-exclusion model in our case, let firstly define:

$$K_S(\theta_1, \theta_2, c) = |\cap(\mathcal{S}(\theta_1),\mathcal{S}(\theta_2))|_c| \quad (4)$$

where θ_1 can be both τ_1 and γ_1 and θ_2 can be both τ_2 and γ_2. Trivially, we can demonstrate that:

$$K(G_1, G_2) = \\ = \sum_{J \in 2^{\{1,\ldots,|C|\}}} (-1)^{|J|-1} K_S(\tau_1, \tau_2, c(J)) K_S(\gamma_1, \gamma_2, c(J)) \quad (5)$$

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

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

$$c = \bigcap_{i \in J_1} c_i = \bigcap_{i \in J_2} c_i \quad (6)$$

Then, we can define C^* as the set of all intersections of constraints in C, i.e. $C^* = \{c(J) | J \in 2^{\{1,\ldots,|C|\}}\}$. We can rewrite the equation as:

$$K(G_1, G_2) = \\ = \sum_{c \in C^*} K_S(\tau_1, \tau_2, c) K_S(\gamma_1, \gamma_2, c) N(c) \quad (7)$$

where

$$N(c) = \sum_{\substack{J \in 2^{\{1,\ldots,|C|\}} \\ c = c(J)}} (-1)^{|J|-1} \quad (8)$$

The complexity of the above kernel strongly depends on the cardinality of C and the related cardinality of C^*. The worst-case computational complexity is still exponential with respect to the size of A_1 and A_2. Yet, the average case complexity (Wang, 1997) is promising.

The set C is generally very small with respect to the worst case. If $\mathcal{F}_{(A_1,A_2)}$ are all the possible correspondences between the nodes A_1 and A_2, 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 $\mathcal{F}_{(A_1,A_2)} = \{\{(\boxed{1},\boxed{1}),(\boxed{2},\boxed{2}),(\boxed{3},\boxed{3})\}, \{(\boxed{1},\boxed{2}),(\boxed{2},\boxed{1}),(\boxed{3},\boxed{3})\}, \{(\boxed{1},\boxed{2}),(\boxed{2},\boxed{3}),(\boxed{3},\boxed{1})\}, ..., \{(\boxed{1},\boxed{3}),(\boxed{2},\boxed{4}),(\boxed{3},\boxed{5})\}\}$. In Sec. 4.5 we argue that the algorithm presented in (Moschitti and Zanzotto, 2007) has the worst-case complexity.

Moreover, the set C^* is extremely smaller than $2^{\{1,\ldots,|C|\}}$ due to the above property (6).

We will analyze the average-case complexity with respect to the worst-case complexity in Sec. 5.

4.4 Enabling the efficient kernel function

The above idea for computing the kernel function is extremely interesting. Yet, we need to make it viable by describing the way we can determine efficiently the three main parts of the equation (7): 1) the set of alternative constraints C (Sec. 4.4.1); 2) the set C^* of all the possible intersections of constraints in C (Sec. 4.4.2); and, finally, 3) the numbers $N(c)$ (Sec. 4.4.3).

4.4.1 Determining the set of alternative constraints

The first step of equation (7) is to determine the alternative constraints C. We can here strongly use the possibility of dividing tDAGs in two trees. We build C as $C_\tau \cup C_\gamma$ where: 1) C_τ are the constraints obtained from pairs of isomorphic extended trees $t_1 \in \mathcal{S}(\tau_1)$ and $t_2 \in \mathcal{S}(\tau_2)$; 2) C_γ are 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 C without explicitly looking at all the subgraphs involved. We instead use and combine the constraints derived comparing the productions 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. 3), 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 $c = \{(\boxed{3},\boxed{4}), (\boxed{2},\boxed{5})\}$.

Using the above intuition it is possible to define an algorithm that builds an alternative constraint set C with the following two properties:

1. for each common subtree according to a set of constraints c, $\exists c' \in C$ such that $c \subseteq c'$;
2. $\nexists c', c'' \in C$ such that $c' \subset c''$ and $c' \neq \emptyset$.

4.4.2 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 (6) discussed in Sec. 4.3, we can empirically demonstrate that the average complexity of the algorithm is not bigger than $O(|C|^2)$. Yet, again, the worst case complexity is exponential.

4.4.3 Determining the values of $N(c)$

The multiplier $N(c)$ (Eq. 8) represents the number of times the constraint c is considered in the sum of equation 5, keeping into account the sign of

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^*

the corresponding addend. It is possible to demonstrate that:

$$N(c) = 1 - \sum_{\substack{c' \in C^* \\ c' \supset c}} N_{c'} \quad (9)$$

This recursive formulation of the equation allows us to easily determine the value of $N(c)$ for every c belonging to C^*. It is possible to prove this property using set properties and the binomial theorem. The proof is omitted for lack of space.

4.5 Reviewing the strictly related work

To understand if ours is an efficient algorithm, we compare it with the algorithm presented by (Moschitti and Zanzotto, 2007). We will hereafter call this algorithm K_{max}. The K_{max} algorithm and kernel is an approximation of what is a kernel needed for a FOR space as 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. Following our formulation, this kernel appears as:

$$K_{max}(G_1, G_2) = \\ = \max_{c \in \mathcal{F}_{(A_1, A_2)}} K_S(\tau_1, \tau_2, c) K_S(\gamma_1, \gamma_2, c) \quad (10)$$

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 Sec. 4.3. This formulation of the kernel has the worst case complexity of our formulation, i.e., Eq. 7.

For computing the basic kernel for the extended trees, i.e. $K_S(\theta_1, \theta_2, c)$ we use the model algorithm presented by (Zanzotto and Moschitti, 2006) and refined by (Moschitti and Zanzotto, 2007) based on the algorithm for tree fragment feature spaces (Collins and Duffy, 2002). As we are using the same basic kernel, we can empirically compare the two methods.

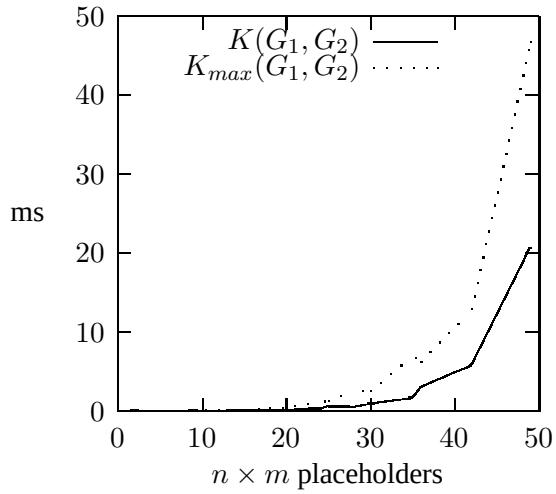


Figure 7: Mean execution time in milliseconds (ms) of the two algorithms wrt. $n \times m$ where n and m are the number of placeholders of the two tDAGs

5 Experimental evaluation

In this section we want to empirically estimate the benefits on the computational cost of our novel algorithm with respect to the algorithm proposed by (Moschitti and Zanzotto, 2007). Our algorithm is in principle exponential with respect to the set of alternative constraints C. Yet, due to what presented in Sec. 4.4 and as the set C^* is usually very small, the average complexity is extremely low. Following the theory on the average-cost computational complexity (Wang, 1997), we estimated the behavior of the algorithms on a large distribution of cases. We then compared the computing times of the two algorithms. Finally, as K and K_{max} compute slightly different kernels, we compare the accuracy of the two methods. We implemented both algorithms $K(G_1, G_2)$ and $K_{max}(G_1, G_2)$ in support vector machine classifier (Joachims, 1999) and we experimented with both implementations on the same machine. We hereafter analyze the results in term of execution time (Sec. 5.1) and in term of accuracy (Sec. 5.2).

5.1 Average computing time analysis

For this first set of experiments, the source of examples is the one of the recognizing textual entailment challenge, i.e., RTE2 (Bar-Haim et al.,

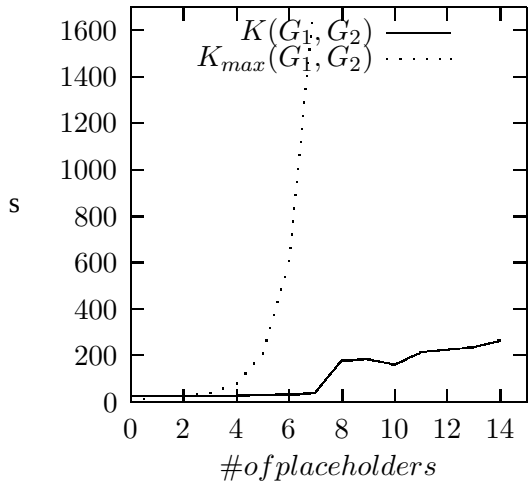


Figure 8: Total execution time in seconds (s) of the training phase on RTE2 wrt. different numbers of allowed placeholders

2006). The dataset of the challenge has 1,600 sentence pairs.

The computational cost of both $K(G_1, G_2)$ and $K_{max}(G_1, G_2)$ depends on the number of placeholders $n = |A_1|$ 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 $n \times m$. Results are reported in Fig. 7 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_{max}. The difference between the two execution times increases with $n \times m$.

We then performed a second experiment that wants to determine the relation of the total execution 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. We then have RTE2 with 1 placeholder at most in each pair, RTE2 with 2 placeholders, etc. The number of pairs in each set is the same. What changes is the maximal number of placeholders. Results are reported in Fig. 8 where the execution time of the training phase in seconds (s) is plotted for each different set. We see that the computation of K_{max} is exponential with respect to the number of placeholders and it becomes intractable after 7 placeholders. The computation of K is instead more flat. This can be explained as the computation of K is related to the real alternative constraints that appears in the dataset. The computation of the kernel K then outperforms the computation of the kernel K_{max}.

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

5.2 Accuracy analysis

As K_{max} that has been demonstrated very effective in term of accuracy for RTE and K compute a slightly different similarity function, we want to show that the performance of our more computationally efficient K is comparable, and even better, to the performances of K_{max}. We then performed an experiment taking as training all the data derived from RTE1, RTE2, and RTE3, (i.e., 4567 training examples) and taking as testing RTE-4 (i.e., 1000 testing examples). The results are reported in Tab. 1. As the table shows, the accuracy of K is higher than the accuracy of K_{max}. There are two main reasons. The first is that K_{max} is an approximation of K. The second is that we can now consider sentence pairs with more than 7 placeholders. Then, we can use the complete training set as the third column of the table shows.

6 Conclusions and future work

We presented an interpretation of first order rule feature spaces as *tripartite directed acyclic graphs* ($tDAG$s). This view on the problem gave us the possibility of defining a novel and efficient algorithm for computing the kernel function for first order rule feature spaces. Moreover, the resulting algorithm is a valid kernel as it can be written as dot product in the explicit space of the tDAG fragments. We demonstrated that our algorithm outperforms in term of average complexity the previous algorithm and it yields to better accuracies for the final task. We are investigating if this is a valid algorithm for two general directed acyclic graphs.

References

Roy Bar-Haim, Ido Dagan, Bill Dolan, Lisa Ferro, Danilo Giampiccolo, and Idan Magnini, Bernardo Szpektor. 2006. The second pascal recognising textual entailment challenge. In *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*. Venice, Italy.

Bob Carpenter. 1992. *The Logic of Typed Feature Structures*. Cambridge University Press, Cambridge, England.

Michael Collins and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of ACL02*.

C. Cortes and V. Vapnik. 1995. Support vector networks. *Machine Learning*, 20:1–25.

Ido Dagan and Oren Glickman. 2004. Probabilistic textual entailment: Generic applied modeling of language variability. In *Proceedings of the Workshop on Learning Methods for Text Understanding and Mining*, Grenoble, France.

Ido Dagan, Oren Glickman, and Bernardo Magnini. 2006. The pascal recognising textual entailment challenge. In Quionero-Candela et al., editor, *LNAI 3944: MLCW 2005*, pages 177–190, Milan, Italy. Springer-Verlag.

Marie-Catherine de Marneffe, Bill MacCartney, Trond Grenager, Daniel Cer, Anna Rafferty, and Christopher D. Manning. 2006. Learning to distinguish valid textual entailments. In *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, Venice, Italy.

Jason Eisner. 2003. Learning non-isomorphic tree mappings for machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL), Companion Volume*, pages 205–208, Sapporo, July.

Thomas Gärtner. 2003. A survey of kernels for structured data. *SIGKDD Explorations*.

Aria D. Haghighi, Andrew Y. Ng, and Christopher D. Manning. 2005. Robust textual inference via graph matching. In *HLT '05: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 387–394, Morristown, NJ, USA. Association for Computational Linguistics.

Andrew Hickl, John Williams, Jeremy Bensley, Kirk Roberts, Bryan Rink, and Ying Shi. 2006. Recognizing textual entailment with LCCs GROUNDHOG system. In Bernardo Magnini and Ido Dagan, editors, *Proceedings of the Second PASCAL Recognizing Textual Entailment Challenge*, Venice, Italy. Springer-Verlag.

Thorsten Joachims. 1999. Making large-scale svm learning practical. In B. Schlkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods-Support Vector Learning*. MIT Press.

Johannes Köbler, Uwe Schöning, and Jacobo Torán. 1993. *The graph isomorphism problem: its structural complexity*. Birkhauser Verlag, Basel, Switzerland, Switzerland.

Alessandro Moschitti and Fabio Massimo Zanzotto. 2007. Fast and effective kernels for relational learning from texts. In *Proceedings of the International Conference of Machine Learning (ICML)*. Corvallis, Oregon.

Alessandro Moschitti. 2004. A study on convolution kernels for shallow semantic parsing. In *proceedings of the ACL*, Barcelona, Spain.

C. Pollard and I.A. Sag. 1994. *Head-driven Phrase Structured Grammar*. Chicago CSLI, Stanford.

Rajat Raina, Aria Haghighi, Christopher Cox, Jenny Finkel, Jeff Michels, Kristina Toutanova, Bill MacCartney, Marie-Catherine de Marneffe, Manning Christopher, and Andrew Y. Ng. 2005. Robust textual inference using diverse knowledge sources. In *Proceedings of the 1st Pascal Challenge Workshop*, Southampton, UK.

Jan Ramon and Thomas Gärtner. 2003. Expressivity versus efficiency of graph kernels. In *First International Workshop on Mining Graphs, Trees and Sequences.*

Jun Suzuki, Tsutomu Hirao, Yutaka Sasaki, and Eisaku Maeda. 2003. Hierarchical directed acyclic graph kernel: Methods for structured natural language data. In *In Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 32–39.

Rui Wang and Günter Neumann. 2007. Recognizing textual entailment using a subsequence kernel method. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-07), July 22-26, Vancouver, Canada*.

Jie Wang. 1997. Average-case computational complexity theory. pages 295–328.

Fabio Massimo Zanzotto and Alessandro Moschitti. 2006. Automatic learning of textual entailments with cross-pair similarities. In *Proceedings of the 21st Coling and 44th ACL*, pages 401–408. Sydney, Australia, July.