Fast Computing Grammar-driven Convolution Tree Kernel for Semantic Role Labeling

Wanxiang Che[1]*, **Min Zhang**[2], **Ai Ti Aw**[2], **Chew Lim Tan**[3], **Ting Liu**[1], **Sheng Li**[1]
[1]School of Computer Science and Technology
Harbin Institute of Technology, China 150001
{car,tliu}@ir.hit.edu.cn, lisheng@hit.edu.cn
[2]Institute for Infocomm Research
21 Heng Mui Keng Terrace, Singapore 119613
{mzhang,aaiti}@i2r.a-star.edu.sg
[3]School of Computing
National University of Singapore, Singapore 117543
tancl@comp.nus.edu.sg

Abstract

Grammar-driven convolution tree kernel (GTK) has shown promising results for semantic role labeling (SRL). However, the time complexity of computing the GTK is exponential in theory. In order to speed up the computing process, we design two fast grammar-driven convolution tree kernel (FGTK) algorithms, which can compute the GTK in polynomial time. Experimental results on the CoNLL-2005 SRL data show that our two FGTK algorithms are much faster than the GTK.

1 Introduction

Given a sentence, the task of semantic role labeling (SRL) is to analyze the propositions expressed by some target verbs or nouns and some constituents of the sentence. In previous work, data-driven techniques, including feature-based and kernel-based learning methods, have been extensively studied for SRL (Carreras and Màrquez, 2005).

Although feature-based methods are regarded as the state-of-the-art methods and achieve much success in SRL, kernel-based methods are more effective in capturing structured features than feature-based methods. In the meanwhile, the syntactic structure features hidden in a parse tree have been suggested as an important feature for SRL and need to be further explored in SRL (Gildea and Palmer, 2002; Punyakanok et al., 2005). Moschitti (2004) and Che et al. (2006) are two reported work to use convolution tree kernel (TK) methods (Collins and Duffy, 2001) for SRL and has shown promising results. However, as a general learning algorithm, the TK only carries out hard matching between two subtrees without considering any linguistic knowledge in kernel design. To solve the above issue, Zhang et al. (2007) proposed a grammar-driven convolution tree kernel (GTK) for SRL. The GTK can utilize more grammatical structure features via two grammar-driven approximate matching mechanisms over substructures and nodes. Experimental results show that the GTK significantly outperforms the TK (Zhang et al., 2007). Theoretically, the GTK method is applicable to any problem that uses syntax structure features and can be solved by the TK methods, such as parsing, relation extraction, and so on. In this paper, we use SRL as an application to test our proposed algorithms.

Although the GTK shows promising results for SRL, one big issue for the kernel is that it needs exponential time to compute the kernel function since it need to explicitly list all the possible variations of two sub-trees in kernel calculation (Zhang et al., 2007). Therefore, this method only works efficiently on such kinds of datasets where there are not too many optional nodes in production rule set. In order to solve this computation issue, we propose two fast algorithms to compute the GTK in polynomial time.

The remainder of the paper is organized as follows: Section 2 introduces the GTK. In Section 3, we present our two fast algorithms for computing the GTK. The experimental results are shown in Section 4. Finally, we conclude our work in Section 5.

*The work was mainly done when the author was a visiting student at I^{2}R

2 Grammar-driven Convolution Tree Kernel

The GTK features with two grammar-driven approximate matching mechanisms over substructures and nodes.

2.1 Grammar-driven Approximate Matching

Grammar-driven Approximate Substructure Matching: the TK requires exact matching between two phrase structures. For example, the two phrase structures "NP→DT JJ NN" (NP→*a red car*) and "NP→DT NN" (NP→*a car*) are not identical, thus they contribute nothing to the conventional kernel although they share core syntactic structure property and therefore should play the same semantic role given a predicate. Zhang et al. (2007) introduces the concept of optional node to capture this phenomenon. For example, in the production rule "NP→DT [JJ] NP", where [JJ] denotes an optional node. Based on the concept of optional node, the grammar-driven approximate substructure matching mechanism is formulated as follows:

$$M(r_1, r_2) = \sum_{i,j} (I_T(T^i_{r_1}, T^j_{r_2}) \times \lambda_1^{a_i+b_j}) \quad (1)$$

where r_1 is a production rule, representing a two-layer sub-tree, and likewise for r_2. $T^i_{r_1}$ is the i^{th} variation of the sub-tree r_1 by removing one ore more optional nodes, and likewise for $T^j_{r_2}$. $I_T(\cdot,\cdot)$ is a binary function that is 1 iff the two sub-trees are identical and zero otherwise. λ_1 ($0 \leq \lambda_1 \leq 1$) is a small penalty to penalize optional nodes. a_i and b_j stand for the numbers of occurrence of removed optional nodes in subtrees $T^i_{r_1}$ and $T^j_{r_2}$, respectively.

$M(r_1, r_2)$ returns the similarity (i.e., the kernel value) between the two sub-trees r_1 and r_2 by summing up the similarities between all possible variations of the sub-trees.

Grammar-driven Approximate Node Matching: the TK needs an exact matching between two nodes. But, some similar POSs may represent similar roles, such as NN (*dog*) and NNS (*dogs*). Zhang et al. (2007) define some equivalent nodes that can match each other with a small penalty λ_2 ($0 \leq \lambda_2 \leq 1$). This case is called node feature *mutation*. The approximate node matching can be formulated as:

$$M(f_1, f_2) = \sum_{i,j} (I_f(f_1^i, f_2^j) \times \lambda_2^{a_i+b_j}) \quad (2)$$

where f_1 is a node feature, f_1^i is the i^{th} mutation of f_1 and a_i is 0 iff f_1^i and f_1 are identical and 1 otherwise, and likewise for f_2 and b_j. $I_f(\cdot,\cdot)$ is a function that is 1 iff the two features are identical and zero otherwise. Eq. (2) sums over all combinations of feature mutations as the node feature similarity.

2.2 The GTK

Given these two approximate matching mechanisms, the GTK is defined by beginning with the feature vector representation of a parse tree T as:

$$\Phi'(T) = (\#subtree_1(T), \ldots, \#subtree_n(T))$$

where $\#subtree_i(T)$ is the occurrence number of the i^{th} sub-tree type ($subtree_i$) in T. Now the GTK is defined as follows:

$$\begin{aligned} K_G(T_1, T_2) &= \langle \Phi'(T_1), \Phi'(T_2) \rangle \\ &= \textstyle\sum_i \#subtree_i(T_1) \cdot \#subtree_i(T_2) \\ &= \textstyle\sum_i ((\sum_{n_1 \in N_1} I'_{subtree_i}(n_1)) \\ &\quad \cdot (\sum_{n_2 \in N_2} I'_{subtree_i}(n_2))) \\ &= \textstyle\sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \Delta'(n_1, n_2) \end{aligned} \quad (3)$$

where N_1 and N_2 are the sets of nodes in trees T_1 and T_2, respectively. $I'_{subtree_i}(n)$ is a function that is $\lambda_1^a \cdot \lambda_2^b$ iff there is a $subtree_i$ rooted at node n and zero otherwise, where a and b are the numbers of removed optional nodes and mutated node features, respectively. $\Delta'(n_1, n_2)$ is the number of the common $subtrees$ rooted at n_1 and n_2, i.e.,

$$\Delta'(n_1, n_2) = \sum_i I'_{subtree_i}(n_1) \cdot I'_{subtree_i}(n_2) \quad (4)$$

$\Delta'(n_1, n_2)$ can be further computed by the following recursive rules:

R-A: if n_1 and n_2 are pre-terminals, then:

$$\Delta'(n_1, n_2) = \lambda \times M(f_1, f_2) \quad (5)$$

where f_1 and f_2 are features of nodes n_1 and n_2 respectively, and $M(f_1, f_2)$ is defined in Eq. (2), which can be computed in linear time $O(n)$, where n is the number of feature mutations.

R-B: else if both n_1 and n_2 are the same non-terminals, then generate all variations of sub-trees of *depth one* rooted at n_1 and n_2 (denoted by T_{n_1}

and T_{n_2} respectively) by removing different optional nodes, then:

$$\Delta'(n_1, n_2) = \lambda \times \sum_{i,j} I_T(T^i_{n_1}, T^j_{n_2}) \times \lambda_1^{a_i+b_j} \times \prod_{k=1}^{nc(n_1,i)} (1 + \Delta'(ch(n_1, i, k), ch(n_2, j, k))) \quad (6)$$

where $T^i_{n_1}$, $T^j_{n_2}$, $I_T(\cdot,\cdot)$, a_i and b_j have been explained in Eq. (1). $nc(n_1, i)$ returns the number of children of n_1 in its i^{th} subtree variation $T^i_{n_1}$. $ch(n_1, i, k)$ is the k^{th} child of node n_1 in its i^{th} variation subtree $T^i_{n_1}$, and likewise for $ch(n_2, j, k)$. λ ($0 < \lambda < 1$) is the decay factor.

R-C: else $\Delta'(n_1, n_2) = 0$

3 Fast Computation of the GTK

Clearly, directly computing Eq. (6) requires exponential time, since it needs to sum up all possible variations of the sub-trees with and without optional nodes. For example, supposing n_1 = "A→a [b] c [d]", n_2 = "A→a b c". To compute the Eq. (6), we have to list all possible variations of n_1 and n_2's subtrees, n_1: "A→a b c d", "A→a b c", "A→a c d", "A→a c"; n_2: "A→a b c". Unfortunately, Zhang et al. (2007) did not give any theoretical solution for the issue of exponential computing time. In this paper, we propose two algorithms to calculate it in polynomial time. Firstly, we recast the issue of computing Eq. (6) as a problem of finding common sub-trees with and without optional nodes between two subtrees. Following this idea, we rewrite Eq. (6) as:

$$\Delta'(n_1, n_2) = \lambda \times (1 + \sum_{p=lx}^{lm} \Delta_p(c_{n_1}, c_{n_2})) \quad (7)$$

where c_{n_1} and c_{n_2} are the child node sequences of n_1 and n_2, Δ_p evaluates the number of common sub-trees with exactly p children (at least including all non-optional nodes) rooted at n_1 and n_2, $lx = \max\{np(c_{n_1}), np(c_{n_2})\}$ and $np(\cdot)$ is the number of non-optional nodes, $lm = \min\{l(c_{n_1}), l(c_{n_2})\}$and $l(\cdot)$ returns the number of children.

Now let's study how to calculate $\Delta_p(c_{n_1}, c_{n_2})$ using dynamic programming algorithms. Here, we present two dynamic programming algorithms to compute it in polynomial time.

3.1 Fast Grammar-driven Convolution Tree Kernel I (FGTK-I)

Our FGTK-I algorithm is motivated by the string subsequence kernel (SSK) (Lodhi et al., 2002). Given two child node sequences $sx = c_{n_1}$ and $t = c_{n_2}$ (x is the last child), the SSK uses the following recursive formulas to evaluate the Δ_p:

$$\Delta'_0(s,t) = 1, \text{for all } s, t, \quad (8)$$

$$\Delta'_p(s,t) = 0, \text{if } \min(|s|,|t|) < p, \quad (9)$$

$$\Delta_p(s,t) = 0, \text{if } \min(|s|,|t|) < p, \quad (10)$$

$$\Delta'_p(sx,t) = \mu \times \Delta'_p(sx,t) + \sum_{j:t_j=x} (\Delta'_{p-1}(s, t[1:j-1] \times \mu^{|t|-j+2})), \quad p = 1, \ldots, n-1, \quad (11)$$

$$\Delta_p(sx,t) = \Delta_p(s,t) + \sum_{j:t_j=x} (\Delta'_{p-1}(s, t[1:j-1] \times \mu^2)). \quad (12)$$

where Δ'_p is an auxiliary function since it is only the interior gaps in the subsequences that are penalized; μ is a decay factor only used in the SSK for weighting each extra length unit. Lodhi et al. (2002) explained the correctness of the recursion defined above.

Compared with the SSK kernel, the GTK has three different features:

f1: In the GTK, only optional nodes can be skipped while the SSK kernel allows any node skipping;

f2: The GTK penalizes ***skipped optional*** nodes only (including both interior and exterior skipped nodes) while the SSK kernel weights the length of subsequences (all interior skipped nodes are counted in, but exterior nodes are ignored);

f3: The GTK needs to further calculate the number of common sub-trees rooted at each two matching node pair x and $t[j]$.

To reflect the three considerations, we modify the SSK kernel as follows to calculate the GTK:

$$\Delta_0(s,t) = opt(s) \times opt(t) \times \lambda_1^{|s|+|t|}, \text{for all } s, t, \quad (13)$$

$$\Delta_p(s,t) = 0, \text{if } \min(|s|,|t|) < p, \quad (14)$$

$$\Delta_p(sx,t) = \lambda_1 \times \Delta_p(sx,t) \times opt(x) + \sum_{j:t_j=x} (\Delta_{p-1}(s, t[1:j-1]) \times \lambda^{|t|-j} \times opt(t[j+1:|t|]) \times \Delta'(x, t[j])). \quad (15)$$

where $opt(w)$ is a binary function, which is 0 if non-optional nodes are found in the node sequence w and 1 otherwise (*f1*); λ_1 is the penalty to penalize skipped optional nodes and the power of λ_1 is the number of skipped optional nodes (*f2*); $\Delta'(x, t[j])$ is defined in Eq. (7) (*f3*). Now let us compare

the FGTK-I and SSK kernel algorithms. Based on Eqs. (8), (9), (10) and (11), we introduce the $opt(\cdot)$ function and the penalty λ_1 into Eqs. (12), (13) and (14), respectively. $opt(\cdot)$ is to ensure that in the GTK only optional nodes are allowed to be skipped. And only those skipped optional nodes are penalized with λ_1. Please note that Eqs. (10) and (11) are merged into Eq. (14) because of the different meaning of μ and λ_1. From Eq. (8), we can see that the current path in the recursive call will stop and its value becomes zero once non-optional node is skipped (when $opt(w) = 0$).

Let us use a sample of n_1 = "A→a [b] c [d]", n_2 = "A→a b c" to exemplify how the FGTK-I algorithm works. In Eq. (14)'s vocabulary, we have s = "a [b] c", t = "a b c", x = "[d]", $opt(x) = opt([d]) = 1$, $p = 3$. Then according to Eq (14), $\Delta_p(c_{n_1}, c_{n_2})$ can be calculated recursively as Eq. (15) (Please refer to the next page).

Finally, we have $\Delta_p(c_{n_1}, c_{n_2}) = \lambda_1 \times \Delta'(\text{a, a}) \times \Delta'(\text{b, b}) \times \Delta'(\text{c, c})$

By means of the above algorithm, we can compute the $\Delta'(n_1, n_2)$ in $O(p|c_{n_1}| \cdot |c_{n_2}|^2)$ (Lodhi et al., 2002). This means that the worst case complexity of the FGTK-I is $O(p\rho^3|N_1| \cdot |N_2|^2)$, where ρ is the maximum branching factor of the two trees.

3.2 Fast Grammar-driven Convolution Tree Kernel II (FGTK-II)

Our FGTK-II algorithm is motivated by the partial trees (PTs) kernel (Moschitti, 2006). The PT kernel algorithm uses the following recursive formulas to evaluate $\Delta_p(c_{n_1}, c_{n_2})$:

$$\Delta_p(c_{n_1}, c_{n_2}) = \sum_{i=1}^{|c_{n_1}|} \sum_{j=1}^{|c_{n_2}|} \Delta'_p(c_{n_1}[1:i], c_{n_2}[1:j]) \quad (16)$$

where $c_{n_1}[1:i]$ and $c_{n_2}[1:j]$ are the child subsequences of c_{n_1} and c_{n_2} from 1 to i and from 1 to j, respectively. Given two child node sequences $s_1 a = c_{n_1}[1:i]$ and $s_2 b = c_{n_2}[1:j]$ (a and b are the last children), the PT kernel computes $\Delta'_p(\cdot,\cdot)$ as follows:

$$\Delta'_p(s_1 a, s_2 b) = \begin{cases} \mu^2 \Delta'(a,b) D_p(|s_1|, |s_2|) & \text{if } a = b \\ 0 & \text{else} \end{cases} \quad (17)$$

where $\Delta'(a,b)$ is defined in Eq. (7) and D_p is recursively defined as follows:

$$\begin{aligned} D_p(k,l) = {} & \Delta'_{p-1}(s_1[1:k], s_2[1:l]) \\ & + \mu D_p(k, l-1) + \mu D_p(k-1, l) \\ & - \mu^2 D_p(k-1, l-1) \end{aligned} \quad (18)$$

$$D_1(k,l) = 1, \text{for all } k, l \quad (19)$$

where μ used in Eqs. (17) and (18) is a factor to penalize the length of the child sequences.

Compared with the PT kernel, the GTK has two different features which are the same as *f1* and *f2* when defining the FGTK-I.

To reflect the two considerations, based on the PT kernel algorithm, we define another fast algorithm of computing the GTK as follows:

$$\begin{aligned} \Delta_p(c_{n_1}, c_{n_2}) = {} & \sum_{i=1}^{|c_{n_1}|} \sum_{j=1}^{|c_{n_2}|} \Delta'_p(c_{n_1}[1:i], c_{n_2}[1:j]) \\ & \times opt(c_{n_1}[i+1 : |c_{n_1}|]) \\ & \times opt(c_{n_2}[j+1 : |c_{n_2}|]) \\ & \times \lambda_1^{|c_{n_1}|-i+|c_{n_2}|-j} \end{aligned} \quad (20)$$

$$\Delta'_p(s_1 a, s_2 b) = \begin{cases} \Delta'(a,b) D_p(|s_1|, |s_2|) & \text{if } a = b \\ 0 & \text{else} \end{cases} \quad (21)$$

$$\begin{aligned} D_p(k,l) = {} & \Delta'_{p-1}(s_1[1:k], s_2[1:l]) \\ & + \lambda_1 D_p(k, l-1) \times opt(s_2[l]) \\ & + \lambda_1 D_p(k-1, l) \times opt(s_1[k]) \\ & - \lambda_1^2 D_p(k-1, l-1) \times opt(s_1[k]) \times opt(s_2[l]) \end{aligned} \quad (22)$$

$$\begin{aligned} D_1(k,l) = {} & \lambda_1^{k+l} \times opt(s_1[1:k]) \times opt(s_2[1:l]), \\ & \text{for all } k, l \end{aligned} \quad (23)$$

$$\Delta'_p(s_1, s_2) = 0, \text{if } \min(|s_1|, |s_2|) < p \quad (24)$$

where $opt(w)$ and λ_1 are the same as them in the FGTK-I.

Now let us compare the FGTK-II and the PT algorithms. Based on Eqs. (16), (18) and (19), we introduce the $opt(\cdot)$ function and the penalty λ_1 into Eqs. (20), (22) and (23), respectively. This is to ensure that in the GTK only optional nodes are allowed to be skipped and only those skipped optional nodes are penalized. In addition, compared with Eq. (17), the penalty μ^2 is removed in Eq. (21) in view that our kernel only penalizes skipped nodes. Moreover, Eq. (24) is only for fast computing. Finally, the same as the FGTK-I, in the FGTK-II the current path in a recursive call will stop and its value becomes zero once non-optional node is skipped (when $opt(w) = 0$). Here, we still can use an example to derivate the process of the algorithm step by step as that for FGTK-I algorithm. Due to space limitation, here, we do not illustrate it in detail.

By means of the above algorithms, we can compute the $\Delta'(n_1, n_2)$ in $O(p|c_{n_1}| \cdot |c_{n_2}|)$ (Moschitti,

$$
\begin{aligned}
\Delta_p(c_{n_1}, c_{n_2}) &= \Delta_p(\text{“a [b] c [d]”}, \text{“a b c”}) \\
&= \lambda_1 \times \Delta_p(\text{“a [b] c”}, \text{“a b c”}) + 0 && //\text{Since } x \not\subseteq t, \text{ the second term is 0} \\
&= \lambda_1 \times (0 + \Delta_{p-1}(\text{“a [b]”}, \text{“a b”}) \times \lambda_1^{3-3} \times \Delta'(c, c)) && //\text{Since } opt(\text{“c”}) = 0, \text{ the first term is 0} \\
&= \lambda_1 \times \Delta'(c, c) \times (0 + \Delta_{p-2}(\text{“a”}, \text{“a b”}) \times \lambda_1^{2-2} \times \Delta'(\text{b, b})) && //\text{Since } p - 1 > |\text{“a”}|, \Delta_{p-2}(\text{“a”}, \text{“a b”}) = 0 \\
&= \lambda_1 \times \Delta'(c, c) \times (0 + \Delta'(\text{a, a}) \times \Delta'(\text{b, b})) && //\Delta_{p-2}(\text{“a”}, \text{“a”}) = \Delta'(\text{a, a})
\end{aligned} \tag{15}
$$

2006). This means that the worst complexity of the FGTK-II is $O(p\rho^2|N_1| \cdot |N_2|)$. It is faster than the FGTK-I's $O(p\rho^3|N_1| \cdot |N_2|^2)$ in theory. Please note that the average ρ in natural language parse trees is very small and the overall complexity of the FGTKs can be further reduced by avoiding the computation of node pairs with different labels (Moschitti, 2006).

4 Experiments

4.1 Experimental Setting

Data: We use the CoNLL-2005 SRL shared task data (Carreras and Màrquez, 2005) as our experimental corpus.

Classifier: SVM (Vapnik, 1998) is selected as our classifier. In the FGTKs implementation, we modified the binary Tree Kernels in SVM-Light Tool (SVM-Light-TK) (Moschitti, 2006) to a grammar-driven one that encodes the GTK and the two fast dynamic algorithms inside the well-known SVM-Light tool (Joachims, 2002). The parameters are the same as Zhang et al. (2007).

Kernel Setup: We use Che et al. (2006)'s hybrid convolution tree kernel (the best-reported method for kernel-based SRL) as our baseline kernel. It is defined as $K_{hybrid} = \theta K_{path} + (1 - \theta)K_{cs}$ $(0 \leq \theta \leq 1)$[1]. Here, we use the GTK to compute the K_{path} and the K_{cs}.

In the training data (WSJ sections 02-21), we get 4,734 production rules which appear at least 5 times. Finally, we use 1,404 rules with optional nodes for the approximate structure matching. For the node approximate matching, we use the same equivalent node sets as Zhang et al. (2007).

4.2 Experimental Results

We use 30,000 instances (a subset of the entire training set) as our training set to compare the different kernel computing algorithms [2]. All experiments are conducted on a PC with CPU 2.8GH and memory 1G. Fig. 1 reports the experimental results, where training curves (time vs. # of instances) of five kernels are illustrated, namely the TK, the FGTK-I, the FGTK-II, the GTK and a polynomial kernel (only for reference). It clearly demonstrates that our FGTKs are faster than the GTK algorithm as expected. However, the improvement seems not so significant. This is not surprising as there are only 30.4% rules (1,404 out of 4,734)[3] that have optional nodes and most of them have only one optional node[4]. Therefore, in this case, it is not time consuming to list all the possible sub-tree variations and sum them up. Let us study this issue from computational complexity viewpoint. Suppose all rules have exactly one optional node. This means each rule can only generate two variations. Therefore computing Eq. (6) is only 4 times (2*2) slower than the GTK in this case. In other words, we can say that given the constraint that there is only one optional node in one rule, the time complexity of the GTK is also $O(|N_1| \cdot |N_2|)$ [5], where N_1 and N_2 are the numbers of tree nodes, the same as the TK.

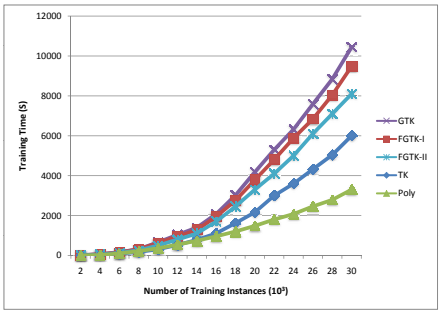

Figure 1: Training time comparison among different kernels with rule set having less optional nodes.

Moreover, Fig 1 shows that the FGTK-II is faster than the FGTK-I. This is reasonable since as dis-

[1] K_{path} and K_{cs} are two TKs to describe predicate-argument link features and argument syntactic structure features, respectively. For details, please refer to (Che et al., 2006).

[2] There are about 450,000 identification instances are extracted from training data.

[3] The percentage is even smaller if we consider all production (it becomes 14.4% (1,404 out of 9,700)).

[4] There are 1.6 optional nodes in each rule averagely.

[5] Indeed it is $O(4 \cdot |N_1| \cdot |N_2|)$. The parameter 4 is omitted when discussing time complexity.

cussed in Subsection 3.2, the FGTK-I's time complexity is $O(p\rho^3|N_1| \cdot |N_2|^2)$ while the FGTK-II's is $O(p\rho^2|N_1| \cdot |N_2|)$.

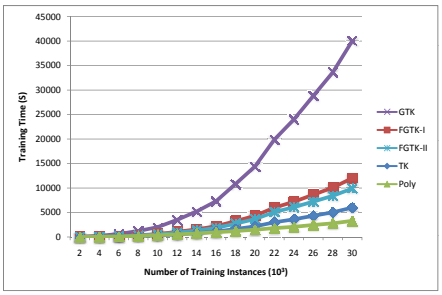

Figure 2: Training time comparison among different kernels with rule set having more optional nodes.

To further verify the efficiency of our proposed algorithm, we conduct another experiment. Here we use the same setting as that in Fig 1 except that we randomly add more optional nodes in more production rules. Table 1 reports the statistics on the two rule set. Similar to Fig 1, Fig 2 compares the training time of different algorithms. We can see that Fig 2 convincingly justify that our algorithms are much faster than the GTK when the experimental data has more optional nodes and rules.

Table 1: The rule set comparison between two experiments.

	# rules	# rule with at least optional nodes	# optional nodes	# average optional nodes per rule
Exp1	4,734	1,404	2,242	1.6
Exp2	4,734	4,520	10,451	2.3

5 Conclusion

The GTK is a generalization of the TK, which can capture more linguistic grammar knowledge into the later and thereby achieve better performance. However, a biggest issue for the GTK is its computing speed, which needs exponential time in theory. Therefore, in this paper we design two fast grammar-driven convolution tree kennel (FGTK-I and II) algorithms which can compute the GTK in polynomial time. The experimental results show that the FGTKs are much faster than the GTK when data set has more optional nodes. We conclude that our fast algorithms enable the GTK kernel to easily scale to larger dataset. Besides the GTK, the idea of our fast algorithms can be easily used into other similar problems.

To further our study, we will use the FGTK algorithms for other natural language processing problems, such as word sense disambiguation, syntactic parsing, and so on.

References

Xavier Carreras and Lluís Màrquez. 2005. Introduction to the CoNLL-2005 shared task: Semantic role labeling. In *Proceedings of CoNLL-2005*, pages 152–164.

Wanxiang Che, Min Zhang, Ting Liu, and Sheng Li. 2006. A hybrid convolution tree kernel for semantic role labeling. In *Proceedings of the COLING/ACL 2006*, Sydney, Australia, July.

Michael Collins and Nigel Duffy. 2001. Convolution kernels for natural language. In *Proceedings of NIPS-2001*.

Daniel Gildea and Martha Palmer. 2002. The necessity of parsing for predicate argument recognition. In *Proceedings of ACL-2002*, pages 239–246.

Thorsten Joachims. 2002. *Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms*. Kluwer Academic Publishers.

Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. 2002. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444.

Alessandro Moschitti. 2004. A study on convolution kernels for shallow statistic parsing. In *Proceedings of ACL-2004*, pages 335–342.

Alessandro Moschitti. 2006. Syntactic kernels for natural language learning: the semantic role labeling case. In *Proceedings of the HHLT-NAACL-2006*, June.

Vasin Punyakanok, Dan Roth, and Wen tau Yih. 2005. The necessity of syntactic parsing for semantic role labeling. In *Proceedings of IJCAI-2005*.

Vladimir N. Vapnik. 1998. *Statistical Learning Theory*. Wiley.

Min Zhang, Wanxiang Che, Aiti Aw, Chew Lim Tan, Guodong Zhou, Ting Liu, and Sheng Li. 2007. A grammar-driven convolution tree kernel for semantic role classification. In *Proceedings of ACL-2007*, pages 200–207.