Using Positional Suffix Trees to Perform Agile Tree Kernel Calculation

Gustavo Henrique Paetzold
University of Sheffield / Sheffield, United Kingdom
ghpaetzold1@sheffield.ac.uk

Abstract

Tree kernels have been used as an efficient solution for many tasks, but are difficult to calculate. To address this problem, in this paper we introduce the Positional Suffix Trees: a novel data structure devised to store tree structures, as well as the *MFTK* and *EFTK* algorithms, which use them to estimate Subtree and Subspace Tree Kernels. Results show that the Positional Suffix Tree can store large amounts of trees in a scalable fashion, and that our algorithms are up to 22 times faster than the state-of-the-art approach.

1 Introduction

A tree kernel is a type of convolution kernel that represents as features the substructures that compose a tree. They can be interpreted as a function $K(T_1, T_2)$, of which the value is a similarity measure between tree structures T_1 and T_2. Recently, tree kernels have become popular, and shown to be an efficient solution in tasks such as Question Classification (Moschitti, 2006), Relation Extraction (Zelenko et al., 2003), Named Entity Recognition (Culotta and Sorensen, 2004), Syntactic Parsing (Collins and Duffy, 2002), Semantic Role Labeling (Moschitti, 2006), Semantic Parsing (Moschitti, 2004), Glycan Classification (Yamanishi et al., 2007) and Plagiarism Detection (Son et al., 2006).

However efficient, tree kernels can be very difficult to calculate in a reasonable amount of time. Calculating $K(T_1, T_2)$ usually requires many verifications between node labels and can easily achieve quadratic complexity. Although algorithms of much lower complexity have been proposed (Moschitti, 2006), their performance can still be unsatisfactory in solving problems which involve large datasets.

In this paper, we present the findings of an ongoing work which focuses on proposing faster algorithms for the calculation of tree kernels. Our strategy uses a time-space tradeoff: we reduce processing time by querying syntactic patterns stored in a Positional Suffix Tree (PST), a novel data structure devised for the storage of trees and graphs. We introduce the the *MFTK* and *EFTK* algorithms, which use PST's to calculate Subtree (ST) and Subspace Tree Kernels (SST). Our experiments show that, while the *MFTK* algorithm is over 3.5 times faster than the state-of-the-art algorithm, the *EFTK* algorithm is over 22 times faster. We also demonstrate that PST's grow in log-linear fashion, scaling well to large tree datasets.

2 Positional Suffix Trees

In order for us to create faster algorithms for the calculation of tree kernels, we have conceived the Positional Suffix Tree: an adaptation of the well known Suffix Tree (Weiner, 1973). Its goal is to store tree structures, such as syntactic parses, and allow efficient search for patterns in them. In other words, it is a tree that stores other trees. To avoid confusion, throughout the rest of the paper we will refer to PST nodes as "trie nodes".

Each trie node of the PST represents a tree node of a certain label in a given position. In constituency parses, for an example, a trie node could represent an "*NP*" (Noun Phrase) node as the leftmost child of an "*S*" (Sentence) node. A PST trie node is composed of the following components:

- **Label:** The identifier of the tree node being represented, such as "*DT*".
- **Tree ID Set:** A set containing the identifiers of each and every tree added to the PST which contains the tree node in question.
- **Children:** A vector of size n, where n is the *the largest number of children parented by*

the tree node in question. In each position i of the children vector is stored a hash structure H that maps the set of tree node labels L to the trie nodes that represent them. The L set contains all the labels of child nodes observed in position i parented by the tree node in question.

As an example, consider the trie node that represents the "**NP**" node highlighted in the three parse trees of Figure 1.

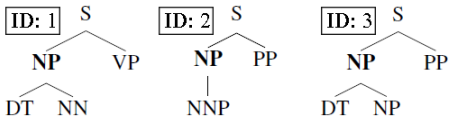

Figure 1: Parse trees with "NP" node

Figure 2 illustrates such trie node.

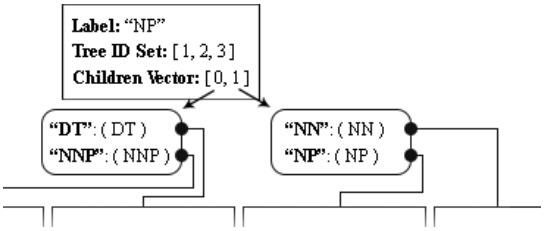

Figure 2: Valid PST trie node

Differently from standard Suffix Trees, the PST node labels can be of any hashable type, any node can have one or more children, and the ordering of a trie node's children is not determined by the alphabetical order of their labels, but rather the position in which the children nodes appear in each tree individually.

3 Tree Kernel Modeling

Our algorithms calculate two types of kernels: the Subtree and the Subspace Tree Kernels. Given two tree structures, the Subtree Kernel (Vishwanathan and Smola, 2004) represents the overlap of "subtrees" between them, which consist of every non-leaf node along with all its descendants. The Subspace Kernel (Collins and Duffy, 2002), on the other hand, represents the overlap of "subspace trees", which are all subtrees and their generalized versions, where some or all of their leaves can be non-terminals. While the *EFTK* algorithm explores those definitions directly, the *MFTK* algorithm explores the ST/SST model of Moschitti (2006), in which the kernel value between two trees is represented as:

$$K(T_1,T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \triangle(n_1,n_2) \qquad (1)$$

Where N_{T_i} is the set of nodes of tree T_i, and $\triangle(n_1,n_2)$ is a function that represents the similarity between nodes n_1 and n_2, subject to the following rules:

- $\triangle(n_1,n_2) = 0$ if $n_1 \neq n_2$
- $\triangle(n_1,n_2) = \alpha$ if $n_1 = n_2$ and both n_1 and n_2 are leaf nodes.
- Otherwise:

$$\triangle(n_1,n_2) = \alpha \prod_{j=1}^{nc(n_1)} \left(\sigma + \triangle\left(c_{n_1}^j, c_{n_2}^j\right)\right) \qquad (2)$$

Where α is a decay factor and $\sigma \in \{0,1\}$. When $\sigma = 0$, $K(T_1,T_2)$ calculates the ST Kernel, and when $\sigma = 1$, the SST Kernel.

4 Algorithms

4.1 PST Storing Algorithm

The recursive algorithm below adds a node N_t of a given tree identified by ID in position i of the children vector of trie node N_{pst}.

Algorithm $AddNode(N_t, ID, i, N_{pst})$:
1. $C = N_{pst}$.Children
2. $L = N_t$.Label
3. **if** L in $C[i]$:
4. **then** $N = C[i][L]$
5. **else** N = new PSTNode()
6. N.Label = L
7. $C[i][L] = N$
8. Add ID in N.TreeIDSet
9. **for** j in $[0, \|N_t.\text{Children}\|]$:
10. $C_t = N_t$.Children$[j]$
11. $AddNode(C_t, ID, j, N)$

4.2 The *MFTK* Algorithm

The *MFTK* (Much Faster Tree Kernel) algorithm calculates both ST and SST Kernels depending on the σ parameter. Unlike state-of-the-art algorithms, it does not calculate $K(T_i,T_j)$ individually, but rather $K(T_i,\{T_1,\ldots,T_n\})$ directly. It receives as input a target tree T_i and a PST containing all subtrees, each with an individual ID, of every source tree T_j. It also requires a hash M, which

maps the subtrees' IDs to its respective tree T_j. The algorithm below creates a valid PST and M map.

Algorithm *CreatePST*$(T_i, T_1, \ldots, T_n)$:
1. PST = new PST()
2. M = new Hash()
3. **for** i in $[1, n]$:
4. **for** c in T_i.Nodes:
5. ID = new ID()
6. *AddNode*$(c, ID, 0, PST$.Root$)$
7. $M[ID] = i$
8. return PST, M

Given a PST, a map M, α and a $\sigma \in \{0, 1\}$, the following algorithm calculates $K(T_i, \{T_1, T_2, \ldots, T_{n-1}, T_n\})$.

Algorithm *Score*$(PST, M, T_i, \{T_1, \ldots, T_n\}, \alpha, \sigma)$:
1. K = new Hash()
2. **for** j in $[1, n]$:
3. $K[j] = 0$
4. **for** c in T_i.Nodes:
5. $S_s = MFTK(c, PST$.Root, 0, α, σ, nil$)$
6. **for** M_{atch} in S_s.Keys:
7. $S = S_s[M_{atch}]$
8. $K[M[M_{atch}]]$ += S
9. return K

The function *MFTK*, described in the algorithm below, uses the PST to estimate kernel values. It receives a tree node N_t, a trie node N_{pst}, a position i, parameters α and σ and an auxiliary static hash Ss. It returns a hash $S_s[N_t]$ which maps a subtree ID M_{atch} to its ST/SST score.

Algorithm *MFTK*$(N_t, N_{pst}, i, \alpha, \sigma, S_s)$:
1. **if** S_s is nil:
2. **then** Ss = new Hash()
3. $S_s[N_t]$ = new Hash()
4. $C_{pst} = N_{pst}$.Children$[i][N_t$.Label$]$
5. **for** ID in C_{pst}.TreeIDSet:
6. $S_s[N_t] = \alpha$
7. **for** j in $[0, \|N_t$.Children$\|]$:
8. $N_{tc} = N_t$.Children$[j]$
9. *MFTK*$(N_{tc}, C_{pst}, j, \alpha, \sigma, S_s)$
10. $M_{iss} = \{S_s[N_t]$.Keys$\}$-$\{S_s[N_{tc}]$.Keys$\}$
11. **for** ID in $Ss[N_{tc}]$.Keys:
12. $S_s[N_t][ID]$ *= σ + $S_s[N_{tc}][ID]$
13. **for** ID in M_{iss}:
14. $S_s[N_t][ID]$ *= σ
15. return $R_{esult} = S_s[N_t]$

4.3 The *EFTK* Algorithm

The *EFTK* (Even Faster Tree Kernel) algorithm uses a very unique strategy: instead of using the model of Moschitti (2006), it calculates the number of common subtrees between two tree structures directly. The *EFTK* can only calculate the ST Kernel. It employs the same *CreatePST* and *Score* routines described in Section 4.2, but instead of the *MFTK* function, it applies the one below.

Algorithm *EFTK*(N_t, N_{pst}, i, S_s):
1. $C_{pst} = N_{pst}$.Children$[i][N_t$.Label$]$
2. **if** S_s is nil:
3. **then** $S_s = C_{pst}$.TreeIDSet
4. **else** $S_s = S_s \cap C_{pst}$.TreeIDSet
5. **for** j in $[0, \|N_t$.Children$\|]$:
6. $C_t = N_t$.Children$[j]$
7. **if** C_t.Label in C_{pst}.Children$[j]$.Keys:
8. **then** *EFTK*(C_t, C_{pst}, j, S_s)
9. **else** $S_s = \{\}$
10. R_{esult} = new Hash()
11. **for** ID in Ss:
12. $R_{esult}[ID] = 1$
13. return R_{esult}

5 Experiments

5.1 Performance Comparison

In this experiment, we conduct a performance comparison between the *MFTK* and *EFTK* algorithms, the baseline QTK (Quadratic Tree Kernel) and the state-of-the-art FTK (Fast Tree Kernel), both of which were proposed by Moschitti (2006).

We have chosen to estimate the processing time taken by the algorithms to calculate the kernel values between the constituency parses produced by the Stanford Parser (Klein and Manning, 2003) of 500 test and 5452 training questions. The datasets were devised for the task of Question Classification (Li and Roth, 2002). The algorithms were implemented in Python, and ran in a computer with a quad-core Intel® Core i7-4500U 1.8GHz and 8Gb of RAM running at 1600MHz. Since the time taken by both FTK and *MFTK* to calculate ST and SST kernels do not vary, we choose to present the performance results for ST kernel calculation only. Figure 3 illustrate the results obtained for increasing portions of the training set, and Figure 4 the average time taken to calculate $K(T_i, T_i)$ for T_i of different node sizes.

The *MFTK* algorithm is on average 3.54 times faster than FTK for different corpus sizes, while

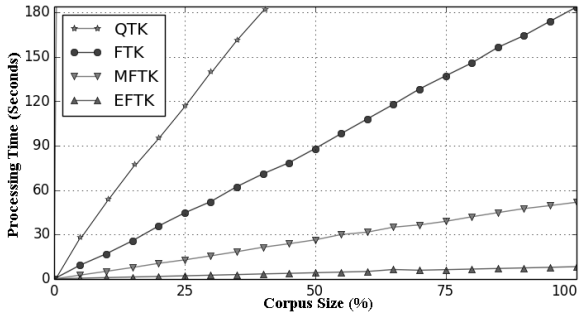

Figure 3: Processing time over different portions of the dataset

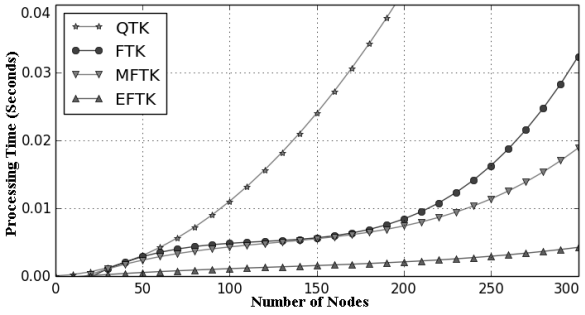

Figure 4: Processing time over trees of different node sizes

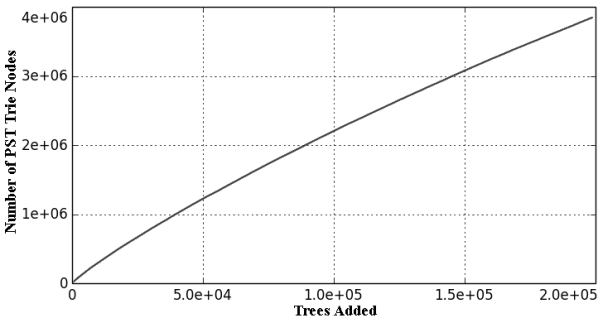

Figure 5: Number of trie nodes over the numbers of trees stored

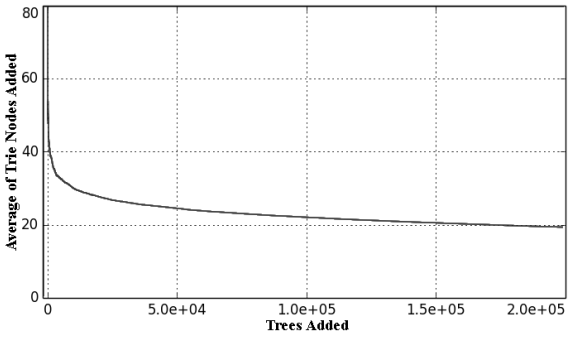

Figure 6: Average number of new trie nodes over the number of trees stored

the *EFTK* algorithm is on average 22.15 times faster. It can also be noticed that, while the processing time of *EFTK* grows almost linearly as the number of nodes rise, the FTK shows a square-like behavior, which is also outperformed by the *MFTK* algorithm.

5.2 Storage Scalability

In this experiment, we evaluate how well Positional Suffix Trees scale with respect to large datasets. To that purpose, we chose to store a dataset of 200k constituency parses of sentences taken from Wikipedia and Simple Wikipedia (Paetzold and Specia, 2013) in a PST, and collect statistics about its size. The PST was implemented in Python and the script ran in the same computer used in the experiment of Section 5.1. Figure 5 shows the number of trie nodes in the PST as the number of stored trees rise, and Figure 6 illustrates the average number of new PST trie nodes added after each tree is stored.

It is noticeable that the curve in Figure 5 shows a convergence pattern, which is in conformity with what is observed in Figure 6, where it is shown that the number of average new nodes tends to converge to an ever lower amount. Such phenomena provide evidence that the PST can indeed store large amounts of tree structures in a scalable fashion, since the more trees are added, the less it needs to grow to represent them.

6 Conclusions and Future Work

In this paper we have introduced the Positional Suffix Tree, a data structure designed to store trees and graphs, and also two algorithms which use them to estimate Subtree and Subspace Tree Kernel values: the *MFTK* and the *EFTK*.

Our experiments revealed that, while the *MFTK* algorithm calculates both ST and SST Kernels in nearly half an order of magnitude faster than state-of-the-art algorithms, the *EFTK* algorithm calculates ST Kernels over an order of magnitude faster. We have also found that the PST provides a scalable storage solution for syntactic parse trees.

In the future we intend to devise algorithms for other kernels, such as the Partial Tree Kernel (Moschitti, 2006), and also explore ways to calculate approximate, faster to estimate, versions of such kernels.

Acknowledgments

I would like to thank the University of Sheffield for supporting this project.

References

Michael Collins and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 263–270, Stroudsburg, PA, USA. Association for Computational Linguistics.

Aron Culotta and Jeffrey Sorensen. 2004. Dependency tree kernels for relation extraction. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 423–429, Barcelona, Spain, July.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *In Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430.

Xin Li and Dan Roth. 2002. Learning question classifiers. In *Proceedings of the 19th International Conference on Computational Linguistics - Volume 1*, COLING '02, pages 1–7, Stroudsburg, PA, USA. Association for Computational Linguistics.

Alessandro Moschitti. 2004. A study on convolution kernels for shallow semantic parsing. In *Proceedings of the 42Nd Annual Meeting on Association for Computational Linguistics*, ACL '04, Stroudsburg, PA, USA. Association for Computational Linguistics.

Alessandro Moschitti. 2006. Efficient convolution kernels for dependency and constituent syntactic trees. In *ECML*, pages 318–329, Berlin, Germany, September. Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Proceedings.

Gustavo H. Paetzold and Lucia Specia, 2013. *Proceedings of the 9th Brazilian Symposium in Information and Human Language Technology*, chapter Text Simplification as Tree Transduction.

Jeong-Woo Son, Seong-Bae Park, and Se-Young Park. 2006. Program plagiarism detection using parse tree kernels. In *Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence*, PRICAI'06, pages 1000–1004, Berlin, Heidelberg. Springer-Verlag.

S V N Vishwanathan and Alex Smola. 2004. Fast kernels for string and tree matching.

Peter Weiner. 1973. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, pages 1–11, Washington, DC, USA. IEEE Computer Society.

Yoshihiro Yamanishi, Francis Bach, and Jean-Philippe Vert. 2007. Glycan classification with tree kernels. *Bioinformatics*, 23(10):1211–1216.

Dmitry Zelenko, Chinatsu Aone, and Anthony Richardella. 2003. Kernel methods for relation extraction. *Journal of Machine Learning Research*, 3:2003.