2D Trie for Fast Parsing

Xian Qian Qi Zhang Xuanjing Huang Lide Wu
{qianxian,qi_zhang,xjhuang,ldwu}@fudan.edu.cn
School of Computer Science, Fudan University, Shanghai, 200433, P.R.China

Abstract

In practical applications, decoding speed is very important. Modern structured learning technique adopts template based method to extract millions of features. Complicated templates bring about abundant features which lead to higher accuracy but more feature extraction time. We propose Two Dimensional Trie (2D Trie), a novel efficient feature indexing structure which takes advantage of relationship between templates: feature strings generated by a template are prefixes of the features from its extended templates. We apply our technique to Maximum Spanning Tree dependency parsing. Experimental results on Chinese Tree Bank corpus show that our 2D Trie is about 5 times faster than traditional Trie structure, making parsing speed 4.3 times faster.[1]

1 Introduction

In practical applications, decoding speed is very important. Modern structured learning technique adopts template based method to generate millions of features. Such as shallow parsing (Sha and Pereira, 2003), named entity recognition (Kazama and Torisawa,), dependency parsing (McDonald et al., 2005), etc.

The problem arises when the number of templates increases, more features generated, making the extraction step time consuming. Especially for maximum spanning tree (MST) dependency parsing, since feature extraction requires quadratic time even using a first order model. According to Bohnet's report (Bohnet, 2009), a fast feature extraction beside of a fast parsing algorithm is important for the parsing and training speed. He takes 3 measures for a 40X speedup, despite the same inference algorithm. One important measure is to store the feature vectors in file to skip feature extraction, otherwise it will be the bottleneck.

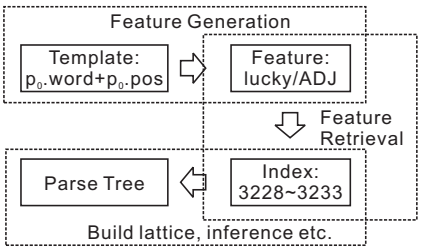

Figure 1: Flow chart of dependency parsing. p_0.*word*, p_0.*pos* denotes the word and POS tag of parent node respectively. Indexes correspond to the features conjoined with dependency types, e.g., *lucky/ADJ/OBJ*, *lucky/ADJ/NMOD*, etc.

Now we quickly review the feature extraction stage of structured learning. Typically, it consists of 2 steps. First, features represented by strings are generated using templates. Then a feature indexing structure searches feature indexes to get corresponding feature weights. Figure 1 shows the flow chart of MST parsing, where p_0.*word*, p_0.*pos* denote the word and POS tag of parent node respectively.

We conduct a simple experiment to investi-

[1]The Chinese dependency parser is available at http://ctbparser.sourceforge.net/ A language independent dependency parsing toolkit and source code is available at http://crfparser.sourceforge.net/

Step	Feature Generation	Index Retrieval	Other	Total
Time	300.27	61.66	59.48	421.41

Table 1: Time spent of each step (seconds) of MSTParser on CTB6 standard test data (2660 sentences). Details of the hardware and corpus are described in section 5

gate decoding time of MSTParser, a state-of-the-art java implementation of dependency parsing [2]. Chinese Tree Bank 6 (CTB6) corpus (Palmer and Xue, 2009) with standard train/development/test split is used for evaluation. Experimental results are shown in Table 1. The observation is that time spent of inference is trivial compared with feature extraction. Thus, speeding up feature extraction is critical especially when large template set is used for high accuracy.

General indexing structure such as Hash and Trie does not consider the relationships between templates, therefore they could not speed up feature generation, and are not completely efficient for searching feature indexes. For example, feature string s_1 generated by template "p_0.*word*" is prefix of feature s_2 from template "p_0.*word* + c_0.*word*" (word pair of parent and child), hence index of s_1 could be used for searching s_2. Further more, if s_1 is not in the feature set, then s_2 must be absent, its generation can be skipped. This is still correct if s_1 is removed after feature selection, as long as we use Trie structure.

We propose Two Dimensional Trie (2D Trie), a novel efficient feature indexing structure which takes advantage of relationship between templates. We apply our technique to Maximum Spanning Tree dependency parsing. Experimental results on CTB6 corpus show that our 2D Trie is about 5 times faster than traditional Trie structure, making parsing speed 4.3 times faster.

The paper is structured as follows: in section 2, we describe template tree which represents relationship between templates; in section 3, we describe our new 2D Trie structure; in section 4, we analyze the complexity of the proposed method and general string indexing structures for parsing; experimental results are shown in section 5; we conclude the work in section 6.

[2] http://sourceforge.net/projects/mstparser

Unit	Meaning
p_{-i}/p_i	the i^{th} node left/right to parent node
c_{-i}/c_i	the i^{th} node left/right to child node
r_{-i}/r_i	the i^{th} node left/right to root node
n.*word*	word of node n
n.*pos*	POS tag of node n
n.*length*	word length of node n
$\|^l$	conjoin current feature with linear distance between child node and parent node
$\|_d$	conjoin current feature with direction of dependency (left/right)

Table 2: Template units appearing in this paper

2 Template tree

2.1 Formulation of template

A template is a set of template units which are manually designed: $T = \{t_1, \ldots, t_m\}$. For convenience, we use another formulation: $T = t_1 + \ldots + t_m$. All template units appearing in this paper are described in Table 2, most of them are widely used. For example, "T = p_0.*word* + c_0.*word*$|^l$ " denotes the word pair of parent and child nodes, conjoined with their distance.

2.2 Template tree

In the rest of the paper, for simplicity, let s_i be a feature string generated by template T_i.

We define the relationship between templates: T_1 is the **ancestor** of T_2 if and only if $T_1 \subset T_2$, and T_2 is called the **descendant** of T_1. Recall that, feature string s_1 is prefix of feature s_2. Suppose $T_3 \subset T_1 \subset T_2$, obviously, the most efficient way to look up indexes of s_1, s_2, s_3 is to search s_3 first, then use its index id_3 to search s_1, and finally use id_1 to search s_2. Hence the relationship between T_2 and T_3 can be neglected.

Therefore we define **direct ancestor** of T_1: T_2 is a direct ancestor of T_1 if $T_2 \subset T_1$, and there is no template T' such that $T_2 \subset T' \subset T_1$. Correspondingly, T_1 is called the **direct descendant** of T_2.

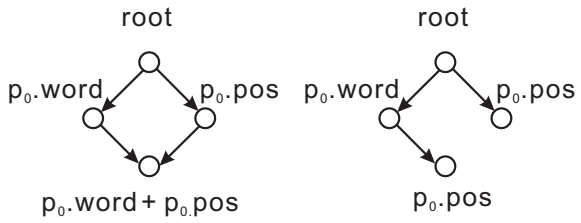

Figure 2: Left graph shows template graph for T_1 =p0.*word*, T_2 =p0.*pos* , T_3 =p0.*word* + p0.*pos*. Right graph shows the corresponding template tree, where each vertex saves the subset of template units that do not belong to its father

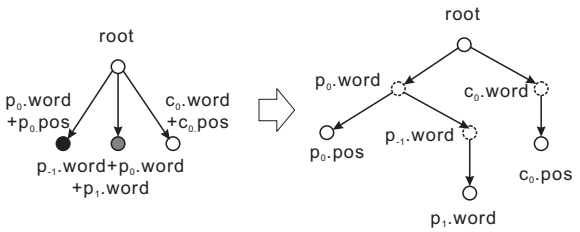

Figure 3: Templates that are partially overlapped: $T_{black} \cap T_{gray}$ =p0.*word*, virtual vertexes shown in dashed circle are created to extract the common unit

Template graph $G = (V, E)$ is a directed graph that represents the relationship between templates, where $V = \{T_1, \ldots, T_n\}$ is the template set, $E = \{e_1, \ldots, e_N\}$ is the edge set. Edge from T_i to T_j exists, if and only if T_i is the direct ancestor of T_j. For templates having no ancestor, we add an empty template as their common direct ancestor, which is also the root of the graph.

The left part of Figure 2 shows a template graph for templates T_1 =p0.*word*, T_2 =p0.*pos* , T_3 =p0.*word* + p0.*pos*. In this example, T_3 has 2 direct ancestors, but in fact s_3 has only one prefix which depends on the order of template units in generation step. If $s_3 = s_1 + s_2$, then its prefix is s_1, otherwise its prefix is s_2. In this paper, we simply use the breadth-first tree of the graph for disambiguation, which is called **template tree**. The only direct ancestor T_1 of T_2 in the tree is called **father** of T_2, and T_2 is a **child** of T_1. The right part of Figure 2 shows the corresponding template tree, where each vertex saves the subset of template units that do not belong to its father.

2.3 Virtual vertex

Consider the template tree in the left part of Figure 3, black vertex and gray vertex are partially overlapped, their intersection is p0.*word*, if string s from template T =p0.*word* is absent in feature set, then both nodes can be neglected. For efficiently pruning candidate templates, each vertex in template tree is restricted to have exactly one template unit (except root). Another important reason for such restriction will be given in the next section.

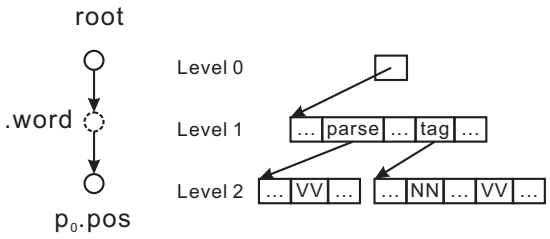

Figure 4: A node in 2D Trie is string(word or pos tag), whereas a node represents a character in an ordinary Trie

To this end, virtual vertexes are created for multi-unit vertexes. For efficient pruning, the new virtual vertex should extract the most common template unit. A natural goal is to minimize the virtual vertex number. Here we use a simple greedy strategy, for the vertexes sharing a common father, the most frequent common unit is extracted as new vertex. Virtual vertexes are iteratively created in this way until all vertexes have one unit. The final template tree is shown in the right part of Figure 3, newly created virtual vertexes are shown in dashed circle.

3 2D Trie

3.1 Single template case

Trie stores strings over a fixed alphabet, in our case, feature strings are stored over several alphabets, such as word list, POS tag list, etc. which are extracted from training corpus.

To illustrate 2D Trie clearly, we first consider a simple case, where only one template used. The

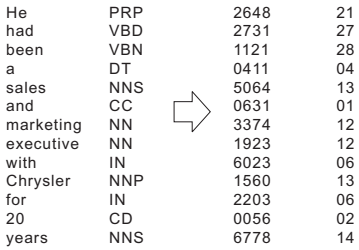

Figure 5: Look up indexes of words and POS tags beforehand.

template tree degenerates to a sequence, we could use a Trie like structure for feature indexing, the only difference from traditional Trie is that nodes at different levels could have different alphabets. One example is shown in Figure 4. There are 3 feature strings from template "p_0.*word* + p_0.*pos*": {*parse/VV, tag/NN, tag/VV*}. Alphabets at level 1 and level 2 are the word set, POS tag set respectively, which are determined by corresponding template vertexes.

As mentioned before, each vertex in template tree has exactly one template unit, therefore, at each level, we look up an index of a word or POS tag in sentence, not their combinations. Hence the number of alphabets is limited, and all the indexes could be searched beforehand for reuse. For example, suppose the template is "r_0.*word* + r_1.*word*", index of the $i+1^{th}$ word is required for feature extraction for the i^{th} and $i+1^{th}$ words. As shown in Figure 5, the token table is converted to an index table, these index values stand for unique identifiers of original strings.

3.2 General case

Generally, for vertex in template tree with K children, children of corresponding Trie node are arranged in a matrix of K rows and L columns, L is the size of corresponding alphabet. If the vertex is not virtual, i.e., it generates features, one more row is added at the bottom to store feature indexes. Figure 6 shows the 2D Trie for a general template tree.

3.3 Feature extraction

When extracting features for a pair of nodes in a sentence, template tree and 2D Trie are visited in

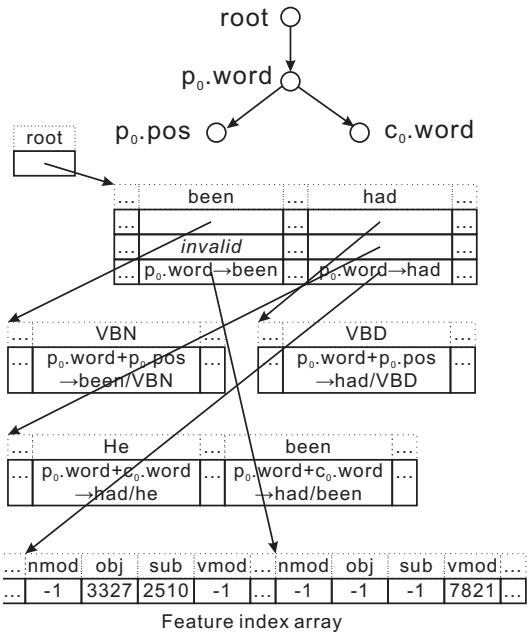

Figure 6: 2D trie for a general template tree. Dashed boxes are keys of columns, which are not stored in the structure

breath first traversal order. Each time, an alphabet and a token index j from index table are selected according to current vertex. For example, POS tag set and the index of the POS tag of parent node are selected as alphabet and token index respectively for vertex "p_0.*pos*". Then children in the j^{th} column of the Trie node are visited, valid children and corresponding template vertexes are saved for further retrieval or generate feature indexes if the child is at the bottom and current Trie node is not virtual. Two queues are maintained to save the valid children and Trie nodes. Details of feature extraction algorithm are described in Algorithm 1.

3.4 Implementation

We use Double Array Trie structure (Aoe, 1989) for implementation. Since children of 2D Trie node are arranged in a matrix, not an array, so each element of the base array has a list of bases, not one base in standard structure. For children that store features, corresponding bases are feature indexes. One example is shown in Figure 7. The root node has 3 bases that point to three rows of

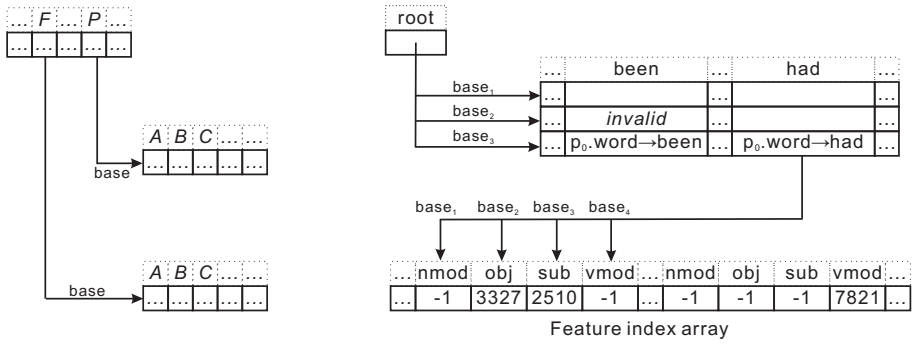

Figure 7: Difference between ordinary base array (left) and base array for 2D Trie (right)

Algorithm 1 Feature extraction using 2D Trie

Input: 2D Trie that stores features, template tree, template graph, a table storing token indexes, parent and child positions

Output: Feature index set S of dependency from parent to child.

Create template vertex queue Q_1 and Trie node queue Q_2. Push roots of template tree and Trie into Q_1, Q_2 respectively. $S = \emptyset$
while Q_1 is not empty, **do**
  Pop a template vertex T from Q_1 and a Trie node N from Q_2. Get token index j from index table according to T.
  for $i = 1$ to child number of T
    if child of N at row i column j is valid, push it into Q_2 and push the i^{th} child of T into Q_1.
    else
      remove decedents of i^{th} child of T from template tree
    end if
  end for
  if T is not virtual and the last child of N in column j is valid
      Enumerate dependency types, add valid feature indexes to S
  end if
end while
Return S.

the child matrix of vertex "p_0.*word*" respectively. Number of bases in each element need not to be stored, since it can be obtained from template vertex in extraction procedure.

Building algorithm is similarly to Double Array Trie, when inserting a Trie node, each row of the child matrix is independently insert into base and check arrays using brute force strategy. The insertion repeats recursively until all features stored.

4 Complexity analysis

Let

$|T|$ = number of templates
$|t|$ = number of template units
$|V|$ = number of vertexes in template tree
$|F|$ = number of features
$|f|$ = average length of feature strings

The procedure of 2D Trie for feature extraction consists of 2 steps: tokens in string table are mapped to their indexes, then Algorithm 1 is carried out for all node pairs of sentence. In the first step, we use double array Trie for efficient mapping. In fact, time spent is trivial compared with step 2 even by binary search. The main time spent of Algorithm 1 is the traversal of the whole template tree, in the worst case, no vertexes removed, so the time complexity for each word pair is $|V|$.

For other indexing structures, feature generation is a primary step of retrieval. For each word pair, $|t|$ template units are processed, including concatenations of tokens and split symbols (split tokens in feature strings), boundary check (e.g, p_{-1}.*word* is out of boundary for beginning node

Structure	Generation	Retrieval
2D Trie	$\|V\|$	
Hash / Trie	$\|t\|$	$\|f\|\|T\|$
Binary Search	$\|t\|$	$\|T\| \log \|F\|$

Table 3: Time complexity of different indexing structures.

of sentence). Notice that, time spent of each process varies on the length of tokens.

For feature string s with length $|s|$, if perfect hashing technique is adopted for index retrieval, it takes $|s|$ calculations to get hash value and a string comparison to check the string at the calculated position. So the time complexity is proportional to $|s|$, which is the same as Trie. Hence the total time for a word pair is $|f||T|$. If binary search is used instead, $\log |F|$ string comparisons are required, complexity for a word pair is $|T| \log |F|$.

Time complexity of these structures is summarized in Table 3.

5 Experiments

5.1 Experimental settings

We use Chinese Tree Bank 6.0 corpus for evaluation. The constituency structures are converted to dependency trees by Penn2Malt [3] toolkit and the standard training/development/test split is used. 257 sentences that failed in the conversion were removed, yielding 23316 sentences for training, 2060 sentences for development and 2660 sentences for testing respectively.

Since all the dependency trees are projective, a first order projective MST parser is naturally adopted. Previous studies show that a second order model achieve higher performance, in this case, complexities of decoding and feature extraction are l times of the first order model if second order features are based on word triples. Therefore the effect of the proposed method are insensitive to order of model, so we only focus on the first order model. Online Passive Aggressive algorithm (Crammer et al., 2006) is used for fast training. The quality of the parser is measured by the labeled attachment score (LAS).

[3] http://w3.msi.vxu.se/ nivre/research/Penn2Malt.html

Group	IDs	#Temp.	#Vert.	#Feat.	LAS
1	1-2	72	91	$3.23M$	79.55%
2	1-3	128	155	$10.4M$	81.38%
3	1-4	240	275	$25.0M$	81.97%
4	1-5	332	367	$34.8M$	82.44%

Table 5: Parsing accuracy and number of templates, vertexes in template tree, features in decoding stage (zero weighted features are excluded) of each group.

We compare the proposed structure with Trie and binary search. For easy comparison, all feature indexing structures and the parser are implemented with C++. All experiments are carried out on a 64bit linux platform (CPU: Intel(R) Xeon(R) E5405, 2.00GHz, Memory: 16G Bytes). For each template set, we run the parser five times on test data and the averaged parsing time is reported.

5.2 Parsing speed comparison

To investigate the scalability of our method, rich templates are designed to generate large feature sets, as shown in Table 4. All templates are organized into 4 groups. Each row of Table 5 shows the details of a group, including parsing accuracy and number of templates, vertexes in template tree, and features in decoding stage (zero weighted features are excluded).

There is a rough trend that parsing accuracy increases as more templates used. Though such trend is not completely correct, the clear conclusion is that, abundant templates are necessary for accurate parsing.

Results of parsing time comparison are shown in Table 6. We can see that though time complexity of dynamic programming is cubic, parsing time of all systems is consistently dominated by feature extraction. When efficient indexing structure adopted, i.e, Trie or Hash, time index retrieval is greatly reduced, about 4-5 times faster than binary search. However, general structures search features independently, their results could not guide feature generation. Hence, feature generation is still time consuming. The reason is that processing each template unit includes a series of steps, much slower than one integer comparison in Trie search.

On the other hand, 2D Trie greatly reduces the

ID	Templates			
1	p_i.*word*	p_i.*pos*	p_i.*word*+p_i.*pos*	
	c_i.*word*	c_i.*pos*	c_i.*word*+c_i.*pos*	$(\|i\| \leq 2)$
	p_i.*length*	p_i.*length*+p_i.*pos*		
	c_i.*length*	c_i.*length*+c_i.*pos*		$(\|i\| \leq 1)$
	p_0.*length*+c_0.*length*$\|_d^l$	p_0.*length*+c_0.*length*+c_0.*pos*$\|_d^l$	p_0.*length*+p_0.*pos*+c_0.*length*$\|_d^l$	
	p_0.*length*+p_0.*pos*+c_0.*pos*$\|_d^l$	p_0.*pos*+c_0.*length*+c_0.*pos*$\|_d^l$	p_0.*length*+p_0.*pos*+c_0.*length*+c_0.*pos*$\|_d^l$	
	p_i.*length*+p_j.*length*+c_k.*length*+c_m.*length*$\|_d^l$			$(\|i\| + \|j\| + \|k\| + \|m\| \leq 2)$
	r_0.*word*	r_{-1}.*word*+r_0.*word*	r_0.*word*+r_1.*word*	
	r_0.*pos*	r_{-1}.*pos*+r_0.*pos*	r_0.*pos*+r_1.*pos*	
2	p_i.*pos*+c_j.*pos*$\|_d$	p_i.*word*+c_j.*word*$\|_d$	p_i.*pos*+c_j.*word*+c_j.*pos*$\|_d$	
	p_i.*word*+p_i.*pos*+c_j.*pos*$\|_d$	p_i.*word*+p_i.*pos*+c_j.*word*$\|_d$	p_i.*word*+c_j.*word*+c_j.*pos*$\|_d$	
	p_i.*word*+p_i.*pos*+c_j.*word*+c_j.*pos*$\|_d$			$(\|i\| + \|j\| = 0)$
	Conjoin templates in the row above with $\|^l$			
3	Similar with 2 $\|i\| + \|j\| = 1$			
4	Similar with 2 $\|i\| + \|j\| = 2$			
5	p_i.*word* + p_j.*word* + c_k.*word*$\|_d$	p_i.*word* + c_j.*word* + c_k.*word*$\|_d$		
	p_i.*pos* + p_j.*pos* + c_k.*pos*$\|_d$	p_i.*pos* + c_j.*pos* + c_k.*pos*$\|_d$		$(\|i\| + \|j\| + \|k\| \leq 2)$
	Conjoin templates in the row above with $\|^l$			
	p_i.*word* + p_j.*word* + p_k.*word* + c_m.*word*$\|_d$		p_i.*word* + p_j.*word* + c_k.*word* + c_m.*word*$\|_d$	
	p_i.*word* + c_j.*word* + c_k.*word* + c_m.*word*$\|_d$			
	p_i.*pos* + p_j.*pos* + p_k.*pos* + c_m.*pos*$\|_d$		p_i.*pos* + p_j.*pos* + c_k.*pos* + c_m.*pos*$\|_d$	
	p_i.*pos* + c_j.*pos* + c_k.*pos* + c_m.*pos*$\|_d$			$(\|i\| + \|j\| + \|k\| + \|m\| \leq 2)$
	Conjoin templates in the row above with $\|^l$			

Table 4: Templates used in Chinese dependency parsing.

Group	Structure	Total	Generation	Retrieval	Other	Memory	sent/sec
1	Trie	87.39	63.67	10.33	13.39	$402M$	30.44
	Binary Search	127.84	62.68	51.52	13.64	$340M$	20.81
	2D Trie	39.74	26.29		13.45	$700M$	66.94
2	Trie	264.21	205.19	39.74	19.28	$1.3G$	10.07
	Binary Search	430.23	212.50	198.72	19.01	$1.2G$	6.18
	2D Trie	72.81	53.95		18.86	$2.5G$	36.53
3	Trie	620.29	486.40	105.96	27.93	$3.2G$	4.29
	Binary Search	982.41	484.62	469.44	28.35	$2.9G$	2.71
	2D Trie	146.83	119.56		27.27	$5.9G$	18.12
4	Trie	854.04	677.32	139.70	37.02	$4.9G$	3.11
	Binary Search	1328.49	680.36	609.70	38.43	$4.1G$	2.00
	2D Trie	198.31	160.38		37.93	$8.6G$	13.41

Table 6: Parsing time of 2660 sentences (seconds) on a 64bit linux platform (CPU: Intel(R) Xeon(R) E5405, 2.00GHz, Memory: 16G Bytes). Title "Generation" and "Retrieval" are short for feature generation and feature index retrieval steps respectively.

Group	Trie	2D Trie
1	999,905,192	425,994,232
2	3,140,497,344	605,173,980
3	7,808,018,972	970,346,188
4	11,516,171,430	1,180,082,473

Table 7: Numbers of checking check values of two Tries

number of feature generations by pruning the template graph. In fact, no string concatenation occurs when using 2D Trie, since all tokens are converted to indexes beforehand. The improvement is significant, 2D Trie is about 5 times faster than Trie on the largest feature set, yielding 13.4 sentences per second parsing speed, about 4.3 times faster.

The reason why 2D Trie is slower than Trie retrieval is the similar with feature generation, many other computations are required beside search, such as push and pop of two queues, boundary check, etc. Table 7 shows the numbers of checking check values of two structures, which are theoretically analyzed in Table 3. We could see that 2D Trie requires much less checks than Trie when large feature set used, hence is more efficient without other computations.

5.3 Comparison against state-of-the-art

Recent works on dependency parsing speedup mainly focus on inference, such as expected linear time non-projective dependency parsing (Nivre, 2009), integer linear programming (ILP) for higher order non-projective parsing (Martins et al., 2009). They achieve 0.632 seconds per sentence over several languages. On the other hand, Goldberg and Elhadad proposed splitSVM (Goldberg and Elhadad, 2008) for fast low-degree polynomial kernel classifiers, and applied it to transition based parsing (Nivre, 2003). They achieve 53 sentences per second parsing speed on English corpus, which is faster than our results, since transition based parsing is linear time, while for graph based method, complexity of feature extraction is quadratic. Xavier Lluís et al. (Lluís et al., 2009) achieve 8.07 seconds per sentence speed on CoNLL09 (Hajič et al., 2009) Chinese Tree Bank test data with a second order graphic model. Bernd Bohnet (Bohnet, 2009) also uses

System	sec/sent
(Martins et al., 2009)	0.63
(Goldberg and Elhadad, 2008)	0.019
(Lluís et al., 2009)	8.07
(Bohnet, 2009)	15.3
(Galley and Manning, 2009)	15.6
ours group1	0.015
ours group2	0.027
ours group3	0.055
ours group4	0.075

Table 8: Comparison against state of the art, direct comparison of parsing time is difficult due to the differences in data, models, hardware and implementations.

second order model, and achieves 610 minutes on CoNLL09 English data (2399 sentences, 15.3 seconds per sentence). Although direct comparison of parsing time is difficult due to the differences in data, models, hardware and implementations, these results demonstrate that our structure can actually result in a very fast implementation of a parser. Moreover, our work is orthogonal to others, and could be used for other learning tasks.

6 Conclusion

We proposed 2D Trie, a novel feature indexing structure for fast template based feature extraction. The key insight is that feature strings generated by a template are prefixes of the features from its extended templates, hence indexes of searched features can be reused for further extraction. We applied 2D Trie to dependency parsing task, experimental results on CTB corpus demonstrate the advantages of our technique, about 5 times faster than traditional Trie structure, yielding parsing speed 4.3 times faster, while using only 1.7 times as much memory.

Acknowledgement The author wishes to thank the anonymous reviewers for their helpful comments. This work was partially funded by 973 Program (2010CB327906), The National High Technology Research and Development Program of China (2009AA01A346), Shanghai Leading Academic Discipline Project (B114), Doctoral Fund of Ministry of Education of China (200802460066), and Shanghai Science and Technology Development Funds (08511500302).

References

Aoe, Jun'ichi. 1989. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on software andengineering*, 15(9):1066–1077.

Bohnet, Bernd. 2009. Efficient parsing of syntactic and semantic dependency structures. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL 2009): Shared Task*, pages 67–72, Boulder, Colorado, June. Association for Computational Linguistics.

Crammer, Koby, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. In *JMLR 2006*.

Galley, Michel and Christopher D. Manning. 2009. Quadratic-time dependency parsing for machine translation. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 773–781, Suntec, Singapore, August. Association for Computational Linguistics.

Goldberg, Yoav and Michael Elhadad. 2008. splitsvm: Fast, space-efficient, non-heuristic, polynomial kernel computation for nlp applications. In *Proceedings of ACL-08: HLT, Short Papers*, pages 237–240, Columbus, Ohio, June. Association for Computational Linguistics.

Hajič, Jan, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. 2009. The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL 2009): Shared Task*, pages 1–18, Boulder, Colorado, June. Association for Computational Linguistics.

Kazama, Jun'ichi and Kentaro Torisawa. A new perceptron algorithm for sequence labeling with non-local features. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 315–324.

Lluís, Xavier, Stefan Bott, and Lluís Màrquez. 2009. A second-order joint eisner model for syntactic and semantic dependency parsing. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL 2009): Shared Task*, pages 79–84, Boulder, Colorado, June. Association for Computational Linguistics.

Martins, Andre, Noah Smith, and Eric Xing. 2009. Concise integer linear programming formulations for dependency parsing. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 342–350, Suntec, Singapore, August. Association for Computational Linguistics.

McDonald, Ryan, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 91–97. Association for Computational Linguistics.

Nivre, Joakim. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 11th International Conference on Parsing Techniques*, pages 149–160.

Nivre, Joakim. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, Suntec, Singapore, August. Association for Computational Linguistics.

Palmer, Martha and Nianwen Xue. 2009. Adding semantic roles to the Chinese Treebank. *Natural Language Engineering*, 15(1):143–172.

Sha, Fei and Fernando Pereira. 2003. Shallow parsing with conditional random fields. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 134–141, May.