A* search for shift-reduce constituent parsing

Thang Le Quang
SoICT - HUST
lelightwin@gmail.com

Yusuke Miyao
National Informatic Institute
Yusuke@nii.ac.jp

Hiroshi Noji
National Informatic Institute
Noji@nii.ac.jp

Abstract

The feature-based shift-reduce constituent parsers now has become one of the state-of-the-art parsing systems due to their efficiency. But there still remains a problem: most of the popular shift-reduce constituency parsers relied on inexact search which cannot guarantee the exactness. In this paper, we will propose a novel idea which based on A* algorithm to perform an exact search for shift-reduce constituent parsing. An exact search will lead to an optimal model for global training of shift-reduce parser and gives a better performance than inexact search. The final experiments has shown that our parser has outperformed inexact search with the same feature set in terms of F-score and can get a comparable accuracies to the previous state-of-the-art parsers.

1 Introduction

Shift-reduce constituent parsing becomes more and more popular thanks to their efficiency. In this method, the parsing system considers a syntactic parse tree as a set of shift-reduce actions. The parsing system assigns an score for each state, and then select the parse tree which has the highest sum score of its actions as a predicted one. The advantage of this approach is that the parsing system can use a large feature set to produce a efficient parsing model with a high parsing speed. Wang et al. (2006) is one of the first research on shift-reduce parsing which used SVM to decide the local action at each point of shift-reduce process. (Zhang, 2009) has presented the global training strategy for shift-reduce constituent parser to obtain a higher ParseVal F-score. (Zhu, 2012) is an extension of the parsing system described in (Zhang, 2009) giving more features to achieve a state-of-the-art performance in terms of F-score and parsing speed.

Unfortunately, because of the large set of possible parse trees caused by the rich extracted features, the process of decoding the final candidate is very difficult to achieve. In our knowledge, most of the current research on shift-reduce parsing had to sacrifice the global exactness of decoding process by using some inexact search. Wang et al. (2006) use a greedy search which gives a local optimal training instead of global one. (Zhang, 2009) has used beam search to exploit more the space of shift-reduce process leading to inexact training. (Zhu, 2012) has a state-of-the-art performance but it has to inject many supervised and unsupervised features as a compensation for exactness.

Concerning about the exact search study for shift-reduce parsing, there are actually several solving methods which has been published. In constituent parsing, (Sagae, 2006) has used the best first search strategy with the help of probability but it is still very far behind the state-of-the-art performance due to its limitation. With the dependency parsing problem, there are some possible research for exact search. (Huang, 2010) has proposed a method applying the dynamic programming into shift-reduce dependency parsing to reduce the time complexity to O(n^7) and use beam search to exploit the candidate space[1]. This approach is still based on inexact search but the most important character is that it has made the candidate space become observable. (Zhao, 2013) has improved (Huang, 2010) with the help of best-first search and edge-factored model (Eisner, 2000) to guarantee the exactness for dependency parsing with the time complexity equaling O(n^6). However, in our knowledge, the exact search problem in feature based constituent shift reduce parsing system (Zhang, 2009) still remains unsolved.

In this paper, we propose a strategy for achiev-

[1] *n is a length of input sentence*

ing the exactness of global discriminative model for shift-reduce constituent parsing. The key idea is to apply A* search into (Huang, 2010) dynamic shift-reduce parsing to guarantee the exactness with acceptable time. Our system has been train by structured perceptron (Collins, 2004) to achieve an global optimal training. The decoder with exact search will lead to better training and parsing in terms of F-score. Our experiments on WSJ dataset has shown that our parser can overcome the previous constituent shift-reduce parsers and give a comparable accuracy to the state-of-the-art parsers.

The rest of this paper is organized as follows. Firstly, section 2 describes our baseline shit-reduce parsing system with inexact search which is based on the baseline parser of (Zhu, 2012). Secondly, section 3 discusses about the feature templates to apply into our system. After that, section 4 will give a clear detail of how we can apply A* search into our baseline parsing system to guarantee the exactness. And the next, all our experiment will be presented in section 5. Finally, we also want to discuss some of our future research directions and conclusions in section 6.

2 Our baseline Shift-Reduce parser

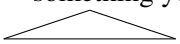
In order to achieve our goal, we built a baseline parser which is adapted from (Zhu, 2012). Our baseline parsing system follows the shift-reduce process of (Sagae, 2005) and use structured perceptron to obtain the global discriminative training.

2.1 Shift-Reduce Constituent Parsing

Shift-reduce parsing can be considered as a state transition system, which means that it includes a set of states and actions. From a current state, each action will create a new corresponding state. A state consists of two data structures: a stack *S* of constructed phrases and a queue *Q* of remain tagged words in the input sentence. Whereas a set of actions in the shift-reduce parsing process includes:

- SHIFT: pop the front word from *Q* and push it onto *S*
- B-REDUCE-L/R(X): pop the two top nodes from *S* and use the binary rule set to combine them into new constituent node with label X, and push X onto *S*. L means that head of X is its left child, and vice versa with R.
- U-REDUCE(X): pop the top constituent off *S*, use the unary rule set to combine them into new constituent node with label X, and push X onto *S*.
- FINISH: pop the root node (Sentence) off *S* and end the shift-reduce parsing.

something you cannot do



a simple tree

The parsing process begins with a start state: empty stack and queue of input tagged sentence and go through a sequence of shift-reduce actions until it perform the FINISH action and reach the final state: an empty stack and an empty queue. Each shift-reduce action has been assigned a score causing that the score of each final state will be sum of the score of all its sequence states. The parsing system will select the parse tree corresponding to the final state with highest accumulative score.

2.2 Average perceptron training

As we mentioned above, we used the averaged perceptron algorithm adapted from (Collins, 2004) to train our model as in (Zhu, 2012). For a given input sentence x, the initial state has an empty stack and a queue that contains all the input words from the sentence. An beam search strategy will be used to find the candidate final state. Specifically, an agenda is used to keep the k-best states in terms of accumulative score at each step. At initialization, the agenda contains only the initial state. At each step, every states in the agenda are popped out and create a set of new states by applying the shift-reduce actions, and the top k from the newly constructed states are put back onto the agenda. The process repeats until the agenda is empty, and the final state with highest score $F(x)$ is selected as the output. We can call it as a beam search decoder.

$$F(x) = arg \max_{s \in Beam(x)} Score(s) \qquad (1)$$

Here $Beam(x)$ is the set of states existed in the beam width of the decoder. The score of a state s is the total score of the shift-reduce actions that have been applied to build the state:

$$Score(s) = \sum_{i=0}^{N} \Phi(a_i).\vec{w} \qquad (2)$$

Table 1: Averaged perceptron algorithm

Inputs	Training examples (x_i, y_i)
Initialization	$\vec{w}$ = 0
Output	averaged weights $\vec{w}$
Algorithm	For $t = 1...T, i = 1...n$ — Calculate $F(x_i) = argmax_{y \in Beam(x_i)} Score(y)$ — If $(F(x_i) \neq y_i)$ then $\vec{w} = \vec{w} + \Phi(y_i) - \Phi(z_i)$

Here $\Phi(a_i)$ represents the feature vector for the i^{th} action a_i in state s. It is computed by applying the feature templates into the context of a_i. N is the total number of actions in s. The parser will go through all the example pairs of sentences and golden final states (x_i, y_i) from the training corpus and use the beam search decoder to produce the candidate $F(x_i)$ and update the weights $\vec{w}$ if $F(x_i)$ does not match with y_i. The pseudo-code for average perceptron training is shown as in Table 1

2.3 Our modified shift-reduce actions

One of the bottle-neck in shift-reduce constituent is the unary rules problem. Because of these unary rules, the parse trees for the same sentence can have a different numbers of nodes which leads to different numbers of shift-reduce actions. This is an important problem to solve because the inconsistency in the number of shift-reduce actions can make the may lead to inconsistency performance. In (Zhu, 2012), they use a padding method which adds the IDLE action in order to extend the sooner completed states. The IDLE action does not change the current state itself, just to keep the number of actions consistency. However, this method creates the redundancy which may increase the number of actions to $4n$ (n is the length of input sentence) and can only guarantee the local consistency in the beam width. With our system, we the following shift-reduce actions to solve the unary rules problem:

- SHIFT: perform the regular shift action.
- SHIFT&U-REDUCE(X): From the current state A, perform the regular shift action creating new state B. And then perform regular unary reduce action to create new state C with new constituent label X. This action returns state C.
- B-REDUCE(Y): Perform the regular binary reduce action to form the state with new constituent label Y.
- B-REDUCE(Y)&U-REDUCE(X): From the current state A, perform the regular binary reduce action to create a state B with constituent label Y. After that, perform the unary reduce action from B to create a new state C with new constituent label X. This action returns state C, too.
- FINISH: perform the regular finish action.

It is clear to see that we had combine the unary reduce actions with shift and binary reduce actions. It means that there is no unary actions anymore and we have only two main types of actions: *shift* or *binary reduce*. This method has two benefits: first, it can guarantee the global consistency, means that the number of shift-reduce actions for each parse tree will always be $2n$. Second, it does not create any redundant states which make the model become sparser. With the number of actions equaling $2n$, the constituent parsing nearly reduces to dependency parsing.

2.4 Dynamic programming

In order to achieve the exactness in parsing, dynamic programming is clearly an important problem to exploit the full space of candidates with the time complexity which is within polynomial limitation. (Huang, 2010) has proposed the way of using dynamic algorithm into shift-reduce dependency parsing which is based on the graph-structured stack of (Tomita, 1991) and the classical CYK algorithm. The key of their approach is to merge the shift states (states after shift action) and b-reduce states (states after binary reduce action) together if they are similar. However, due to the unary actions in constituent parsing, the dynamic programming is difficult to implement because the unary states could not be merged with shift states or b-reduce states. Fortunately, with our modified

shift-reduce actions, the unary states are not existed which causes the dynamic programming to be easy to implement.

3 The influence of feature templates on time complexity

In shift-reduce parsing, the search space will be very large depending on the feature templates we used. Therefore, in this section, we would like to make some evaluation about the time complexity of feature templates and then design a feature template used for exact search in shift-reduce constituent parsing.

3.1 Evaluation from the baseline feature template

In (Zhang, 2009), they has designed an efficient baseline feature template which is widely used for shift-reduce constituent parsing. The feature template can be viewed as in table 2 as long as its components and sub-components has been shown in table 3 and 4, respectively.

To evaluate the time complexity of shift-reduce parsing system applying this feature template, we will reply on the observation that the baseline feature has focused on top four nodes in stack S (table 3). With each node, we could consider the parsing process as a CYK process. As we know, dynamic algorithm such as CYK can parse an input sentence within $O(n^3.|G|)$ (cubic time) with unlexicalized case and $O(n^5.|G|)$ with lexicalized case. The table 4 has shown that the baseline feature template uses head word as one of its components, so this is the lexicalized case. Therefore, with the combination of four nodes in the lexicalized case, the parsing system which applies the baseline feature must take a very large time complexity about $O((n^5.|G|)^4) = O(n^{20}.|G|^4)$ in the worst case. In general form, if the feature template focuses on top k nodes in stack S, then we will have the time complexity $O((n^5 * |G|)^k)$ with lexicalized parsing and $O((n^3 * |G|)^k)$ with unlexicalized parsing.

Following the result published in (Zhao, 2013), the exact search will be possible to perform if we can reduce the time complexity to $O(n^6)$. Comparing this complexity to the general form of time complexity in shift-reduce parsing: $O((n^5 * |G|)^k)$ (lexicalized) and $O((n^3 * |G|)^k)$ (unlexicalized), we will have $k = 1$ (lexicalized) or $k = 2$ (unlexicalized). It means that we have two directions to create our own feature template:

Table 3: The components which are used in baseline feature template

component	explanation
s_0	1^{th} top node in stack S
s_1	2^{nd} top node in stack S
s_2	3^{rd} top node in stack S
s_3	4^{th} top node in stack S
q_0	1^{th} word in queue Q
q_1	2^{th} word in queue Q
q_2	3^{th} word in queue Q
q_3	4^{th} word in queue Q
s_0l	left child of s_0
s_0r	right child of s_0
s_0u	unary child of s_0
s_1l	left child of s_1
s_1r	right child of s_1
s_1u	unary child of s_1

Table 4: The sub-components which are used in baseline feature template

$X.t$	PoS tag of X's head word
$X.w$	X's head word
$X.c$	X's constituent tag

- $k = 1(lexicalized)$: designing a feature template which focuses only on top node in stack S which can use the head word features.
- $k = 2(unlexicalized)$: designing a feature template which focuses on top two nodes in stack S without the head word features.

3.2 Our simplified feature template

As we concluded in the previous section, we will have two strategies of creating a feature template for exact search. With the first approach, the feature template will be so small and lack of information so we created our own feature template based on the second one.

With the second one, the only problem is the lack of head lexical information. Fortunately, there are many studies on unlexicalized constituent parsing which give high parsing accuracy. (Dan2, 2003) proposed their first research on unlexicalized parsing. This is a theoretically splitting constituent method which can reach 85.77% F-score. (Petrov, 2007) has proposed out an extension of (Dan2, 2003) method which splits the

Table 2: A table of baseline feature template

Description	Templates
1-grams	$s_0.t + s_0.c$; $s_0.w + s_0.c$; $s_1.t + s_1.c$; $s_1.w + s_1.c$ $s_2.t + s_2.c$; $s_2.w + s_2.c$; $s_3.t + s_3.c$; $s_3.w + s_3.c$ $q_0.w + q_0.t$; $q_1.w + q_1.c$; $q_2.w + q_2.t$; $q_3.w + q_3.c$ $s_0l.w + s_0l.c$; $s_0u.w + s_0u.c$; $s_0r.w + s_0r.c$ $s_1l.w + s_1l.c$; $s_1u.w + s_1u.c$; $s_1r.w + s_1r.c$
2-grams	$s_0.w + s_1.w$; $s_0.w + s_1.c$; $s_0.c + s_1.w$; $s_0.c + s_1.c$ $s_0.w + q_0.w$; $s_0.w + q_0.t$; $s_0.c + q_0.w$; $s_0.c + q_0.t$ $q_0.w + q_1.w$; $q_0.w + q_1.t$; $q_0.t + q_1.w$; $q_0.t + q_1.t$ $s_1.w + q_0.w$; $s_1.w + q_0.t$; $s_1.c + q_0.w$; $s_1.c + q_0.t$
3-grams	$s_0.c + s_1.c + s_2.c$; $s_0.w + s_1.c + s_2.c$; $s_0.c + s_1.w + s_2.c$; $s_0.c + s_1.c + s_2.w$ $s_0.c + s_1.c + q_0.t$; $s_0.w + s_1.c + q_0.t$; $s_0.c + s_1.w + q_0.t$; $s_0.c + s_1.c + q_0.w$

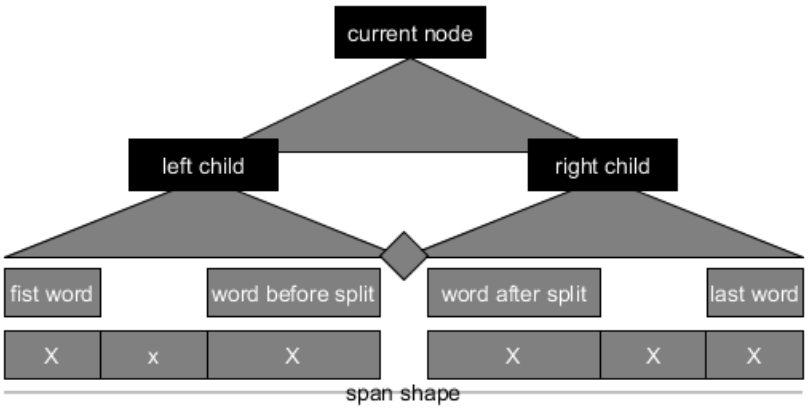


Figure 1: The span feature from adapted from (David, 2014).

grammar automatically to exploit the latent variables and attain the F-score equaling 90.7%.

Time complexity of baseline feature template

$O(n^{20}.\|G\|^4)$			
$O(n^5.\|G\|)$	$O(n^5.\|G\|)$	$O(n^5.\|G\|)$	$O(n^5.\|G\|)$
s_0	s_1	s_2	s_3

Time complexity of non-head feature template

$O(n^6.\|G\|^2)$	
$O(n^3.\|G\|)$	$O(n^3.\|G\|)$
s_0	s_1

Figure 2: Comparison of time complexity between baseline and non-head feature template.

As in our system, we have designed a feature template which was inspired by the span features in (David, 2014). This is a very simple feature set which relied only on the local features within the span of a node and achieve an amazing performace. In the experiment reported in (David, 2014), it has reached an F-score = 89.9% and outperformed Berkeley parser in terms of parsing many different languages. The span features from this article include the following components for a constituent node (table 5):

- The first and last word of the span of node.
- the length of the span of node
- The word before and after the split point between left and right child of node.
- The shape of word sequence within the span of node (illustrated in figure 3):
  - X: if the current word's first letter is capital letter.
  - x: if the current word is normal word.

Table 5: The sub-components which are used in our simplified feature template

component	explanation
$X.ft$	The first PoS in X's span
$X.fw$	The first word in X's span
$X.lt$	the last PoS in X's span
$X.lw$	the last word in X's span
$X.aSw$	The word after-split in X's span
$X.aSt$	The PoS after-split in X's span
$X.bSw$	The word before-split in X's span
$X.bSt$	The PoS before-split in X's span
$X.len$	The length of X's span
$X.shape$	The shape of X's span

  - N: if the current word is a number.
  - special character such as [' " , .] will remain unchanged.
  - O: the other cases.

Our simplified feature template has been shown in table 6. It has focused only on two top nodes in stack S(s_0 and s_1) and did not use the head lexical information. Therefore, based on our previous analysis, the time complexity of this feature template will be $O(n^6 * |G|^2)$. You can see the illustration of comparison between the baseline and simplified feature template in 2.

4 A* decoder for Shift-Reduce Parsing

A* search is a very well-known algorithm which belongs to the best-first algorithm. The best-first algorithm uses an agenda to store all the processed points in the searching process. In each step, the most promising point will be used to extend until we reach the final point.

However, unlike the regular best first algorithm which uses only the path cost $g(x)$ of each point x in agenda to calculate the best point, A* algorithm also uses an heuristic $h(x)$ which is an admissible heuristic of the path from x to the final point. It means that each point x in A* algorithm will have the score calculated by:

$$score(x) = g(x) + h(x) \tag{3}$$

A* algorithm can guarantee to find the goal with smallest path cost in a shorter time than regular best-first search. To make sure that A* algorithm is efficient, $h(x)$ should satisfy two characters:

- admissible: the heuristic must be greater or equal than the real highest cost we have to take in order to get to the final point: $h(x) \geq h_{real}(x)$
- consistency: $h(x) \leq d(x, y)+h(y)$, for every (x, y) where y is next to x, $d(x, y)$ is the cost from x to y. This is an additional condition.

Therefore, it is plain to see that $h(x)$ is the key of A* algorithm's efficiency.

4.1 Best first decoder

In order to utilize A* heuristic, we firstly built our best first decoder which is similar the one in (David, 2014) but for constituent parsing. Let us denote that $BFS(x)$ is the candidate final state returned by the best first decoder. The best first decoder can be described as follows:

- Set up an agenda adding the initial state.
- In each step, popping out the best candidate in agenda and extend it by using the shift-reduce actions. After that, put the newly constructed states back into agenda.
- Looping until the popped candidate from agenda is the final state.

In best first decoder, the model score of each state s is the cost path, the model score of each shift-reduce actions a_i is the transition cost between each state. Let us recall that the model score of state s is the sum of model score of all its shift-reduce actions a_i as in (4). The model score of a_i is calculated as in equation (5). If we use A* decoder, $score(s)$ will be calculated by $g(s) + h(s)$, where $h(s)$ is the heuristic which will be discussed in the next section. The training process of shift-reduce parsing with best first decoder is illustrated in table 7.

$$score(s) = g(s) = \sum_{i=0}^{N} score(a_i) \tag{4}$$

$$score(a_i) = \Phi(a_i).\vec{w} \tag{5}$$

The only remain bottle-neck is that the negative score problem happened in the shift-reduce parsing process. In best-first search algorithm, the transition of each shift-reduce actions must be positive. However, due to the learning strategy of averaged perceptron, there still exist a negative value

Table 6: A table of our simplified feature template

Description	Templates
Queue	$q_0.w$+$q_0.t$; $q_1.w$+$q_1.c$; $q_2.w$+$q_2.t$; $q_3.w$+$q_3.c$
s_0's features	$s_0.c$+$s_0.ft$; $s_0.c$+$s_0.fw$; $s_0.c$+$s_0.lt$; $s_0.c$+$s_0.lw$ $s_0.c$+$s_0.aSt$; $s_0.c$+$s_0.aSw$; $s_0.c$+$s_0.bSt$; $s_0.c$+$s_0.bSw$ $s_0.c$+$s_0.ft$+$s_0.lt$; $s_0.c$+$s_0.ft$+$s_0.lw$; $s_0.c$+$s_0.fw$+$s_0.lt$; $s_0.c$+$s_0.fw$+$s_0.lw$ $s_0.c$+$s_0.ft$+$s_0.len$; $s_0.c$+$s_0.fw$+$s_0.len$; $s_0.c$+$s_0.lt$+$s_0.len$; $s_0.c$+$s_0.lw$+$s_0.len$ $s_0.c$+$s_0.ft$+$s_0.lt$+$s_0.len$; $s_0.c$+$s_0.ft$+$s_0.lw$+$s_0.len$ $s_0.c$+$s_0.fw$+$s_0.lt$+$s_0.len$; $s_0.c$+$s_0.fw$+$s_0.lw$+$s_0.len$ $s_0.c$+$s_0.len$; $s_0.c$+$s_0.shape$
s_1's features	$s_1.c$+$s_1.ft$; $s_1.c$+$s_1.fw$; $s_1.c$+$s_1.lt$; $s_1.c$+$s_1.lw$ $s_1.c$+$s_1.aSt$; $s_1.c$+$s_1.aSw$; $s_1.c$+$s_1.bSt$; $s_1.c$+$s_1.bSw$ $s_1.c$+$s_1.ft$+$s_1.lt$; $s_1.c$+$s_1.ft$+$s_1.lw$; $s_1.c$+$s_1.fw$+$s_1.lt$; $s_1.c$+$s_1.fw$+$s_1.lw$ $s_1.c$+$s_1.ft$+$s_1.len$; $s_1.c$+$s_1.fw$+$s_1.len$; $s_1.c$+$s_1.lt$+$s_1.len$; $s_1.c$+$s_1.lw$+$s_1.len$ $s_1.c$+$s_1.ft$+$s_1.lt$+$s_1.len$; $s_1.c$+$s_1.ft$+$s_1.lw$+$s_1.len$ $s_1.c$+$s_1.fw$+$s_1.lt$+$s_1.len$; $s_1.c$+$s_1.fw$+$s_1.lw$+$s_1.len$ $s_1.c$+$s_1.len$; $s_1.c$+$s_1.shape$

Table 7: Averaged perceptron algorithm with best first search decoder.

Inputs	Training examples (x_i, y_i)
Initialization	$\vec{w}$ = 0
Output	averaged weights $\vec{w}$
Algorithm	For $t = 1...T, i = 1...n$ — Calculate $F(x_i) = BFS(x_i)$ — If $(F(x_i) \neq y_i)$ then $\vec{w} = \vec{w} + \Phi(y_i) - \Phi(z_i)$

in the weights vector $\vec{w}$, and this may lead to negative score problem. Concerning about this problem, we solved this by adding a certain offset value to the transition score of each actions to keep it positive. Because the number of shift-reduce actions is always $2n$ (section 2.3) in our system, the rank of final states is still the same as before. So it will be able to apply this method.

4.2 Grammar projection

Grammar projection is an interesting idea published in (Dan1, 2003) and this may be the first research on A* parsing in our knowledge. In this method, they have projected the original grammar rules to a simpler grammar rules. Each projected production rule always have the PCFG probability that is the maximum value of all the original production rules projecting to it. With this conversion, the outside PCFG score of the parse tree constructed by this projected grammar is always greater or equal to the real outside PCFG score and it can be an heuristic for A* search.

In our system, we design our own grammar projection based on the filter projection from (Dan1, 2003) with some modification which is suitable with shift-reduce parsing:

- project all the complete constituent tags such as [NP, VP, ADJP...] into one single label called [XTAG].
- keeping all the incomplete constituent tags such as [NP*, AP*, ADJP*] and the PoS tag as before.

Corresponding to this projection, the features will also be projected in our system. The projected features will have the weight equaling the maximum value of all features which project it. For example, there are shift features whose types are $s_0.fw$+$s_0.c$ with values: ([google+VP]; [google+NP]; [google+AP]) with the weights (14.65; 5.67; 8.2), respectively. They will be projected into one value: [google+XTAG] with the weight equaling to 14.65.

In our A* decoder, with each state s, we will perform the best first search with these projected

Table 8: The experiment on the development set.

System	F-score on BF	parsing speed on BF	F-score on SF	parsing speed on SF
beam search decoder with $beam = 16$	89.07%	25 sentences/s	88.72%	30 sentences/s
beam search decoder with $beam = 32$	89.87%	10 sentences/s	89.33%	13 sentences/s
beam search decoder with $beam = 64$	90.3%	3.4 sentences/s	90.15%	4.5 sentences/s
beam search decoder with $beam = 128$				
$A*$ decoder	N/A	N/A	90.7%	1.2 sentences/s

grammar rules and features to find the heuristic final state f'. After projecting, the time complexity of decoding proces will be reduced significantly and can be performed by best first search. The transition score from s to f' will be the heuristic of s in A* search. Because the projected features is the maximum summarization of the original features in terms of weight values, the heuristic transition score will always be greater or equal than the real transition score. It means that the grammar projection heuristic presented here is an *admissible* heuristic.

4.3 The less constituent heuristic

The *less constituent* is an heuristic which have been proposed by us based on this observation: if we reduce the number of nodes focused in the feature template, then we can significantly reduce the time complexity. Therefore, if we focused on only s_0 instead of both s_0 and s_1, the time complexity for calculation will be reduced to $O(n^3.|G|)$ which can be decoded by best first search (similar to normal CYK algorithm). As in grammar projection, the heuristic transition score is calculated by best first search after considering less constituent will be the heuristic for A* search. More specifically, to calculate this heuristic, the s_0's features in table 6 will still be the same but the s_1's features will be ignored and carry the maximum weight value of their types.

For example, the current state s have three attached features: $[s_0.c$+$s_0.ft]$, $[s_0.c$+$s_0.fw]$ and $[s_1.c$+$s_1.fw]$. Then the two features whose type is $[s_0.c$+$s_0.ft]$, $[s_0.c$+$s_0.fw]$ will be extracted from s and calculated normally, but the $[s_1.c$+$s_1.fw]$ feature of s will be assigned the maximum weight value of all features whose type is $[s_1.c$+$s_1.fw]$. Because the s_1's weights after being ignored are always greater or equal (maximum) than the original s_1's weights, it is clear to see that the heuristic transition score calculated by this way is always greater or equal than the real transition score and this heuristic is also admissible.

4.4 Our cascaded heuristic

Both of those two above heuristics have their own problem: calculating the *grammar projection* heuristic will have time complexity = $O(n^6 * |g|^2)$ where g is the projected grammar, and calculate the *less constituent* heuristic will have time complexity = $O(n^3 * |G|)$. Both of calculation may be a little expensive so we combine them together to reduce the total time complexity.

The process of calculating our heuristic could be described as follows: we use the *less constituent* method to calculate A* heuristic for the original best first decoder. And in the process of calculating less constituent heuristic, we use the *grammar projection* to guide the best first search go faster. As you can see, this heuristic is a cascaded type in which an A* heuristic is calculated by another A* heuristic. This method will lead to significant speed-up: the time complexity of calculating *less constituent* will be as fast as the *filter*

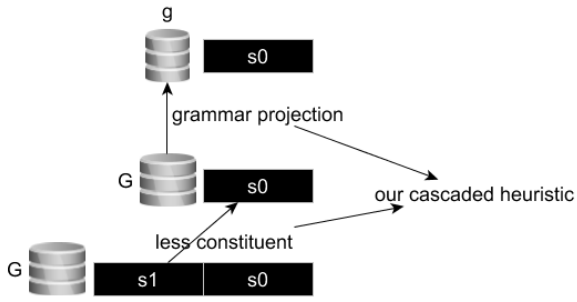


Figure 3: The span feature from adapted from (David, 2014).

heuristic in (Dan1, 2003), and the *less constituent* is also an effective heuristic which will lead the original best first decoder go faster to the best final state.

5 Experiment

5.1 Preparation

We evaluate our A* shift-reduce parser on the Wall Street Journal (WSJ) corpus of the Penn Treebank project with the standard split: sections 2-21 were used for training, section 22 as a development set, and section 23 was used for testing. All our experiments reported in this paper was performed on [our computer configuration].

We used the head finder of (Collins, 1999) to define the head constituent tag for each phrase in the corpus, and adapted the binarization of (Zhang, 2009) to binarize our grammar rules. We also used Stanford PoS tagger (with the accuracy = 97%) to produce the tagged sentence as an input for our shift-reduce constituent parsing. To evaluate the ParseVal F-score, we utilize EVALB program.

5.2 The experiment results on development set

On the development set, we make an experiment of comparing our proposal A* decoder with beam search decoder. We also test these decoders in both baseline and simplified feature template. The experiment results can be viewed in table 8.

5.3 The experiment results on test set

On the test set, we make an experiment for comparing our A* parser with the state-of-the-art parsing systems. The results of this experiment are shown in table 9.

Table 9: Results of the final experiment on test set to compare our A* shift reduce parser with other parsing system.

System	F-score
Johnson and Charniak (2005)	91.4%
Mc Closky (2006)	92.1%
Collins (1999), Bikel (2004)	87.7%
Berkeley Parser	90.1%
Stanford Parser	90.4%
SSN parser - Henderson(2004)	89.4%
Zhang Yue - baseline (2012)	89.8%
This paper	91.1%

6 Conclusion and future works

References

David Hall, Greg Durrett and Dan Klein. 2014. Less Grammar, More Features. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages: 228-337, Baltimore, Maryland.

Kai Zhao, Jame Cross and Liang Huang. 2013. Optimal Incremental Parsing via Best-First Dynamic Programming. In Proceedings of EMNLP 2013.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang and Jingbo Zhu. 2013. Fast and Accurate Shift-Reduce Constituent Parsing In proceedings of ACL 2013. Sophia, Bulgaria. August.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In Proceedings of ACL 2010.

Yue Zhang and Stephen Clark. 2009. Transition-based parsing of the Chinese Treebank using a global discriminative model. In Proceedings of IWPT, Paris, France, October.

Xavier Carreras, Michael Collins, and Terry Koo. 2008. Tag, dynamic programming, and the perceptron for efficient, feature-rich parsing. In Proceedings of CoNLL, pages 9-16, Manchester, England.

Liang Huang. 2008. Forest reranking: discriminative parsing with non-local features. In Proceedings of ACL, pages 586-594, Ohio, USA.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In Proceedings of HLT/NAACL, pages 404-411, Rochester, New York, April.

David McClosky, Eugene Charniak, and Mark Johnson. 2006. Effective self-training for parsing. In Proceedings of the HLT/NAACL, Main Conference, pages 152-159, New York City, USA, June.

Kenji Sagae and Alon Lavie. 2006. A best-first probabilistic shift-reduce parser In Proceedings of the COLING/ACL, Main Conference, poster sessions, pages 691-698

Eugune Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In Proceeding of ACL, pages 173-180.

James Henderson. 2004. Discriminative training of a neural network statistical parser. In Proceedings of ACL.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In Proceedings of ACL, Stroudsburg, PA, USA.

Ben Taskar, Dan Klein, Michael Collins, Daphne Koller, and Chris Manning. 2004. Max-margin parsing. In Proceedings of EMNLP.

Dan Klein and Christopher D Manning. 2003. Accurate Unlexicalized Parsing In Proceedings of ACL 2003.

Dan Klein and Christopher D Manning. 2003. A* parsing: Fast exact Viterbi parse selection. In Proceedings of HLT-NAACL.

Michael Collins. 2000. Discriminative reranking for natural language processing. In Proceedings of ICML, pages 175182, Stanford, CA, USA.

Michael Collins. 1999. Head-driven statistical models for natural language parsing. Ph.D. thesis, University of Pennsylvania.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In Proceedings of ACL, Madrid, Spain.

Masaru Tomita. 1991. Generalized LR Parsing. Kluwer Academic Publishers.