Transition-Based Deep Input Linearization

by

Ratish Puduppully, Manish Shrivastava, Yue zhang

in

the 14th European Conference on Artificial Life (ECAL 2017)

LyonTech Campus in Lyon, France.

Report No: IIIT/TR/2017/-1

Centre for Language Technologies Research Centre
International Institute of Information Technology
Hyderabad - 500 032, INDIA
September 2017

Transition-Based Deep Input Linearization

Ratish Puduppully †[*], Yue Zhang ‡, Manish Shrivastava †
†Kohli Center on Intelligent Systems (KCIS),
International Institute of Information Technology, Hyderabad (IIIT Hyderabad)
‡Singapore University of Technology and Design
ratish.surendran@research.iiit.ac.in yue_zhang@sutd.edu.sg
m.shrivastava@iiit.ac.in

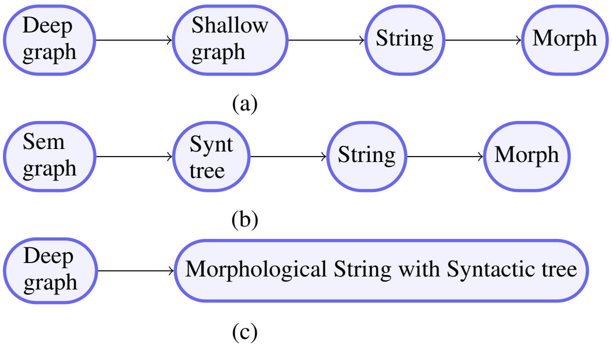
Abstract

Traditional methods for deep NLG adopt pipeline approaches comprising stages such as constructing syntactic input, predicting function words, linearizing the syntactic input and generating the surface forms. Though easier to visualize, pipeline approaches suffer from error propagation. In addition, information available across modules cannot be leveraged by all modules. We construct a transition-based model to jointly perform linearization, function word prediction and morphological generation, which considerably improves upon the accuracy compared to a pipelined baseline system. On a standard deep input linearization shared task, our system achieves the best results reported so far.

1 Introduction

Natural language generation (NLG) (Reiter and Dale, 1997; White, 2004) aims to synthesize natural language text given input syntactic, semantic or logical representations. It has been shown useful in various tasks in NLP, including machine translation (Chang and Toutanova, 2007; Zhang et al., 2014), abstractive summarization (Barzilay and McKeown, 2005) and grammatical error correction (Lee and Seneff, 2006).

A line of traditional methods treat the problem as a pipeline of several independent steps (Bohnet et al., 2010; Wan et al., 2009; Bangalore et al., 2000; H. Oh and I. Rudnicky, 2000; Langkilde and Knight, 1998). For example, shown in Figure 1b, a pipeline based on the meaning text theory (MTT) (Melčuk, 1988) splits NLG into three independent steps 1. syntactic generation: generating an unordered and lemma-formed syntactic tree from a semantic graph, introducing function words; 2. syntactic linearization: linearizing the unordered syntactic tree; 3. morphological generation: generating the inflection for each lemma in the string.



(a)

(b)

(c)

Figure 1: Linearization pipelines (a) NLG pipeline with deep input graph, (b) pipeline based on the meaning text theory, (c) this paper.

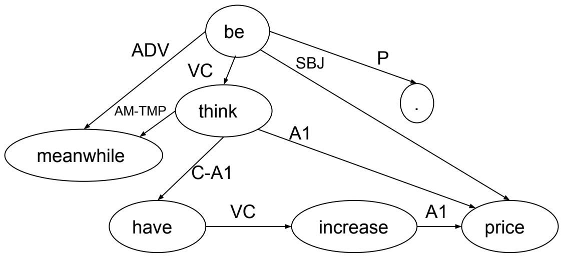


Figure 2: Sample deep graph for the sentence: *meanwhile, prices are thought to have increased.* Note that words are replaced by their lemmas. The function word *to* and comma are absent in graph.

In this paper we focus on deep graph as input. Exemplified in Figure 2, the deep input type is intended to be an abstract representation of the meaning of a sentence. Unlike semantic input, where the nodes are semantic representations of input, deep input is more surface centric, with lem-

[*] Part of the work was done when the author was a visiting student at Singapore University of Technology and Design.

mas for each word being connected by semantic labels (Banarescu et al., 2013; Melčuk, 2015). In contrast to shallow syntactic trees, function words in surface forms are not included in deep graphs (Belz et al., 2011). Deep inputs can more commonly occur as input of NLG systems where entities and content words are available, and one has to generate a grammatical sentence using them with only provision for inflections of words and introduction of function words. Such usecases include summarization, dialog generation etc.

A pipeline of deep input linearization is shown in Figure 1a. Generation involves predicting the correct word order, deciding inflections and also filling in function words at the appropriate positions. The worst-case complexity is $n!$ for permuting *n* words, 2^n for function word prediction (assuming that a function word can be inserted after each content word), and 2^n for inflection generation (assuming two morphological forms for each lemma). On the dataset from the First Surface Realisation Shared Task, Bohnet et al. (2011) achieved the best reported results on linearizing deep input representation, following the pipeline of Figure 1b (with input as deep graph instead of semantic graph). They construct a syntactic tree from deep input graph followed by function word prediction, linearization and morphological generation. A rich set of features are used at each stage of the pipeline and for each adjacent pair of stages, an SVM decoder is defined.

Pipelined systems suffer from the problem of error propagation. In addition, because the steps are independent of each other, information available in a later stage is not made use of in the earlier stages. We introduce a transition-based (Nivre, 2008) method for *joint* deep input surface realisation integrating linearization, function word prediction and morphological generation. The model is shown in Fig 1c, as compared with the pipelined baseline in Fig 1a. For a directly comparable baseline, we construct a pipeline system of function words prediction, linearization and morphological generation similar to the pipeline of Bohnet et al. (2011), but with the following difference. Our baseline pipeline system makes function word prediction for a deep input graph, whereas Bohnet et al. (2011) have a preprocessing step to construct a syntactic tree from the deep input graph, which is given as input to the function word prediction module. Our pipeline is directly comparable to the joint system with regard to the use of information.

Standard evaluations show that: 1. Our joint model for deep input surface realisation achieves significantly better scores over its pipeline counterpart. 2. We achieve the best results reported on the task. Our system scores 1 BLEU point better over Bohnet et al. (2011) without using any external resources. We make the source code available at `https://github.com/SUTDNLP/ZGen/releases/tag/v0.3`.

2 Related Work

Related work can be broadly summarized into three areas: abstract word ordering, applications of meaning-text theory and joint modelling of NLP tasks. In abstract word ordering (Wan et al., 2009; Zhang, 2013; Zhang and Clark, 2015), De Gispert et al. (2014) compose phrases over individual words and permute the phrases to achieve linearization. Schmaltz et al. (2016) show that strong surface-level language models are more effective than models trained with syntactic information for the task of linearization. Transition-based techniques have also been explored (Liu et al., 2015; Liu and Zhang, 2015; Puduppully et al., 2016). To our knowledge, we are the first to use transition-based techniques for *deep* input linearization.

There has been work done in the area of sentence linearization using meaning-text theory (Melčuk, 1988). Belz et al. (2011) organized a shared task on both shallow and deep linearization according to meaning-text theory, which provides a standard benchmark for system comparison. Song et al. (2014) achieved the best results for the task of *shallow*-syntactic linearization. Using SVM models with rich features, Bohnet et al. (2011) achieved state-of-art results on the task of *deep* realization. While they built a pipeline system, we show that joint models can be used to overcome limitations of the pipeline approach giving the best results.

Joint models for NLP have shown effectiveness in recent years. Though having to tackle increased search space, they overcome issues with error propagation in pipelined models. Joint models have been explored for grammar-based approaches to surface realisation using HPSG and CCG (Carroll and Oepen, 2005; Velldal and Oepen, 2006; Espinosa et al., 2008; White and Rajkumar, 2009; White, 2006; Carroll et al., 1999).

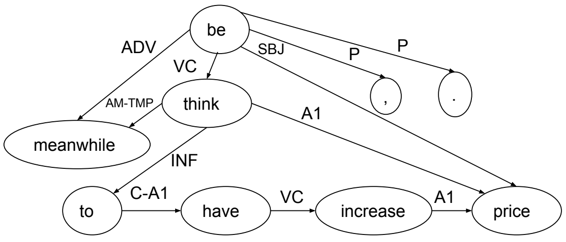


Figure 3: Equivalent shallow graph for Figure 2.

Joint models have been proposed for word segmentation and POS-tagging (Zhang and Clark, 2010), POS-tagging and syntactic chunking (Sutton et al., 2007), segmentation and normalization (Qian et al., 2015), syntactic linearization and morphologization (Song et al., 2014), parsing and NER (Finkel and Manning, 2009), entity and relation extraction (Li and Ji, 2014) and so on. We propose a first joint model for deep realization, integrating linearization, function word prediction and morphological generation.

3 Baseline

We build a baseline following the pipeline in Figure 1a. Three stages are involved: 1. prediction of function words, inserting the predicted function words in the deep graph, resulting in a *shallow* graph; 2. linearizing the shallow graph; 3. generating the inflection for each lemma in the string.

3.1 Function Word Prediction

In the First Surface Realisation Shared Task dataset (Belz et al., 2011), there are three classes of function words to predict: *to* infinitive, *that* complementizer and *comma*. We implement classifiers to predict these classes of function words locally at respective positions in the deep graph resulting in a shallow graph (Figure 3). At each location the input is a node and output is a class indicating if *to* or *that* need to inserted under the node or the count of *comma* to be introduced under the node.

Table 1 shows the feature templates for classification of *to* infinitives and *that* complementizers and Table 2 shows the feature templates for predicting the count of *comma* child nodes for each non-leaf node in the graph. These feature templates are a subset of features used in the joint model (Section 4) with the exceptions being word order features, which are not available here for the pipeline system, since earlier stages cannot leverage features in subsequent outcomes. We use averaged perceptron classifier (Collins, 2002) to predict function words, which is consistent with the joint model.

Features for predicting function words including *to* infinitive, *that* complementizer
WORD(n); POS(n); WORD(c)

Table 1: Feature templates for the prediction of function words- *to* infinitive and *that* complementizer. Indices on the surface string: n – word index; c – child of n; Functions: WORD – word at index; POS – part-of-speech at index.

Features for predicting count of *comma*

WORD(n); POS(n)
BAG(WORD-MOD(n))
BAG(LABEL-MOD(n))

Table 2: Feature templates for the comma prediction system. Indices on the surface string: n – word index; Functions: WORD – word at index; POS – part-of-speech at index; WORD-MOD – modifiers of index; LABEL-MOD – dependency labels of modifiers; BAG – set.

3.2 Linearization

The next step is linearizing the graph, which we solve using a novel transition-based algorithm.

3.2.1 Transition-Based Tree Linearization

Liu et al. (2015) introduce a transition-based model for tree linearization. The approach extends from transition-based parsers (Nivre and Scholz, 2004; Chen and Manning, 2014), where *state* consists of *stack* to hold partially built outputs and a *queue* to hold input sequence of words. In case of linearization, the input is a set of words. Liu et al. therefore use a set to hold the input instead of a queue. State is represented by a tuple *(σ, ρ, A)*, where σ is stack to store partial derivations, ρ is set of input words and A is the set of dependency relations that have been built. There are three transition actions:

- SHIFT-*Word*-POS – shifts *Word* from ρ, assigns POS to it and pushes it to top of stack as S_0;
- LEFTARC-LABEL – constructs dependency arc $S_1 \xleftarrow{LABEL} S_0$ and pops out second element from top of stack S_1;
- RIGHTARC-LABEL – constructs dependency arc $S_1 \xrightarrow{LABEL} S_0$ and pops out top of stack S_0.

Input lemmas: $\{$think$_1$, price$_2$, .$_3$, increase$_4$, be$_5$, have$_6$, meanwhile$_7$, ,$_8$, to$_9\}$

	Transition	σ	ρ	A
0		[]	{1...7}	$\emptyset$
1	SH-meanwhile	[7]	{1...6,8,9}	
2	SH-,	[7 8]	{1...6,9}	
3	SH-price	[7 8 2]	{1,3,4,5,6,9}	
4	SH-be	[7 8 2 5]	{1,3,4,6,9}	
5	SH-think	[7 8 2 5 1]	{3,4,6,9}	
6	SH-to	[7 8 2 5 1 9]	{3,4,6}	
7	SH-have	[7 8 2 5 1 9 6]	{3,4}	
8	SH-increase	[7 8 2 5 1 9 6 4]	{3}	
9	RA	[7 8 2 5 1 9 6]	{3}	$A \cup \{6 \rightarrow 4\}$
10	RA	[7 8 2 5 1 9]	{3}	$A \cup \{9 \rightarrow 6\}$
11	RA	[7 8 2 5 1]	{3}	$A \cup \{1 \rightarrow 9\}$
12	RA	[7 8 2 5]	{3}	$A \cup \{5 \rightarrow 1\}$
13	SH-.	[7 8 2 5 3]	{}	
14	RA	[7 8 2 5]	{}	$A \cup \{5 \rightarrow 3\}$
15	LA	[7 8 5]	{}	$A \cup \{2 \leftarrow 5\}$
16	LA	[7 5]	{}	$A \cup \{8 \leftarrow 5\}$
17	LA	[5]	{}	$A \cup \{7 \leftarrow 5\}$

Table 3: Transition action sequence for linearizing the graph in Figure 3. SH - SHIFT, RA - RIGHTARC, LA - LEFTARC. POS is not shown in SHIFT actions.

The sequence of actions to linearize the set $\{$*he, goes, home*$\}$ is SHIFT-*he*, SHIFT-*goes*, SHIFT-*home*, RIGHTARC-OBJ, LEFTARC-SBJ.

The full set of feature templates are shown in Table 2 of Liu et al. (2015), partly shown in Table 4. The features include word(w), POS(p) and dependency label (l) of elements on stack and their descendants S_0, S_1, $S_{0,l}$, $S_{0,r}$ etc. For example, word on top of stack is S_0w and word on first left child of S_0 is $S_{0,l}w$. These are called configuration features. They are combined with all possible actions to score the action. Puduppully et al. (2016) extend Liu et al. (2015) by redefining features to address feature sparsity and introduce lookahead features, thereby achieving highest accuracies on task of abstract word ordering.

3.2.2 Shallow Graph Linearization

Our transition based graph linearization system extends from Puduppully et al. (2016). In our case, the input is a shallow graph instead of a syntactic tree, and hence the search space is larger. On the other hand, the same set of actions can still be applied, with additional constraints on valid actions given each configuration (Section 3.2.3). Table 3 shows the sequence of transition actions to linearize shallow graph in Figure 3.

3.2.3 Obtaining Possible Transition Actions Given a Configuration

The purpose of a GETPOSSIBLEACTIONS function is to find out the set of transition actions that can lead to a valid output given a certain state.

Algorithm 1: GETPOSSIBLEACTIONS for shallow graph linearization

```
Input: A state $s = ([\sigma|j\ i], \rho, A)$ and input graph $C$
Output: A set of possible transition actions $T$
$T \leftarrow \emptyset$
if $s.\sigma == \emptyset$ then
    for $k \in s.\rho$ do
        $T \leftarrow T \cup (\textsc{Shift}, POS, k)$
else
    if $\exists k, k \in (\textsc{DirectChildren}(i) \cap s.\rho)$ then
        $\textsc{ShiftSubtree}(i, \rho)$
    else
        if A.LEFTCHILD(i) is NIL then
            $\textsc{ShiftSubtree}(i, \rho)$
        if $\{j \rightarrow i\} \in C \wedge$ A.LEFTCHILD(j) is NIL then
            $T \leftarrow T \cup (\textsc{RightArc})$
            if $i \in \textsc{Descendant}(j)$ then
                $\textsc{ProcessDescendant}(i, j)$
            if $i \in \textsc{Sibling}(j)$ then
                $\textsc{ProcessSibling}(i, j)$
        else if $\{j \leftarrow i\} \in C$ then
            $T \leftarrow T \cup (\textsc{LeftArc})$
            if $i \in \textsc{Sibling}(j)$ then
                $\textsc{ProcessSibling}(i, j)$
        else
            if $size(s.\sigma)$ == 1 then
                $\textsc{ShiftParentAndSiblings}(i)$
            else
                if $i \in \textsc{Descendant}(j)$ then
                    $\textsc{ProcessDescendant}(i, j)$
                if $i \in \textsc{Sibling}(j)$ then
                    $\textsc{ProcessSibling}(i, j)$
return $T$
```

Algorithm 2: DIRECTCHILDREN

```
Input: A state s=([σ|j i], ρ, A), input_node and graph C.
Output: DC direct child nodes of input node
DC ← ∅
for k ∈ (C.CHILDREN(input_node)) do
    Parents ← C.PARENTS(k)
    if Parents.size == 1 then
        DC ← DC ∪ k
    else
        for m ∈ Parents do
            if A.LEFTCHILD(m) is not NIL ∨ m == input_node then
                continue
            if m ∩ s.ρ then
                goto OutsideLoop
            if m ∈ σ∧ σ.ISANCESTOR(m,C) then
                goto OutsideLoop
        DC ← DC ∪ k
    OutsideLoop:
return DC
```

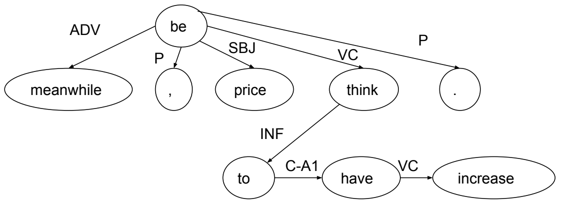


Figure 4: Equivalent syntactic tree for Figure 2.

Algorithm 3: SHIFTSUBTREE

```
  Input: A state $s = ([\sigma|j\ i], \rho, A)$, graph $C$, head $k$
  Output: a set of possible Transition actions $T$
1 $T \leftarrow \emptyset$
2 $T \leftarrow T \cup$ (SHIFT, POS, k)
3 queue q
4 q.push(k)
5 while q is not empty do
6     front = q.pop()
7     for $m \in$ (C.CHILDREN(front) $\cap$ s.$\rho$) do
8         q.push(m)
9         $T \leftarrow T \cup$ (SHIFT, POS, m)
```

This is because not all sequences of actions correspond to a well-formed output. Essentially, given a state $s = ([\sigma|j\ i], \rho, A)$ and an input graph *C*, the Decoder extracts syntactic tree from the graph (cf. Figure 4 extracted from Figure 3), outputting RIGHTARC, LEFTARC only if the corresponding arc exists in *C*. The corresponding pseudocode is shown in Algorithm 1.

In particular, if node *i* has *direct child nodes* in *C*, the descendants of *i* are shifted (line 6-7) (see Algorithm 3). Here *direct child nodes* (see Algorithm 2) include those child nodes of *i* for which *i* is the only parent or if there is more than one parent then every other parent is shifted on to the stack without possibility to reduce the child node. If no *direct child node* is in the buffer, then all graph descendants of *i* are shifted. Now, there are three configurations possible between *i* and *j*: 1. *i* and *j* are directly connected in *C*. This results in RIGHTARC or LEFTARC action; 2. *i* is descendant of *j*. In this case the parents of *i* (such that they are descendants of *j*) and siblings of *i* through such parents are shifted. 3. *i* is sibling of *j*. In this case, parents of *i* and their descendants are shifted such that *A* remains consistent. Because the input is a graph, more than one of the above configuration can occur simultaneously. More detailed discussion related to GETPOSSIBLEACTIONS is given in Appendix A.

Unigrams
S_0w; S_0p; $S_{0,l}w$; $S_{0,l}p$; $S_{0,l}l$; $S_{0,r}w$; $S_{0,r}p$; $S_{0,r}l$;
Bigram
$S_0wS_{0,l}w$; $S_0wS_{0,l}p$; $S_0wS_{0,l}l$; $S_0pS_{0,l}w$;
Linearization
w_0; p_0; $w_{-1}w_0$; $p_{-1}p_0$; $w_{-2}w_{-1}w_0$; $p_{-2}p_{-1}p_0$

Table 4: Baseline linearization feature templates. A subset is shown here. For the full feature set, refer to Table 2 of Liu et al. (2015).

3.2.4 Feature Templates

There are three sets of features. The first is the set of baseline linearization feature templates from Table 2 in Liu et al. (2015), partly shown in Table 4. The second is a set of *lookahead features* similar to that of Puduppully et al. (2016), shown in Table 5.[1] Parent lookahead feature in Puduppully et al. (2016) is defined for the only parent. For graph linearization, however, the parent lookahead feature need to be defined for set of parents. The third set of features in Table 6 are newly introduced for Graph Linearization. Arc_{left} is a binary feature indicating if there is left arc between S_0 and S_1, whereas Arc_{right} indicates if there is a right arc. $L_{is\_descendant}$ is a binary feature indicating if L is descendant of S_0, and $L_{is\_parent\_or\_sibling}$ indicates if it is a parent or sibling of S_0. $S_{0descendants\_shifted}$ is binary feature indicating if all the descendants of S_0 are shifted.

Not having POS in the input dataset, we compute the feature templates for POS making use of the most frequent POS of the lemma in the gold training data. For the features with dependency labels, we use the input graph labels.

3.2.5 Search and Learning

We follow Puduppully et al. (2016) and Liu et al. (2015), applying the learning and search framework of Zhang and Clark (2011). Pseudocode is shown in Algorithm 4. It performs beam search holding *k* best states in an agenda at each incremental step. At the start of decoding, agenda holds the initial state. At a step, for each state in the

[1] Here L_{cls} represents set of arc labels of child nodes (of word to shift *L*) shifted on the stack, L_{clns} represents set of arc labels of child nodes not shifted on the stack, L_{cps} the POS set of shifted child nodes, L_{cpns} the POS set of unshifted child nodes, L_{sls} the set of arc labels of shifted siblings, L_{slns} the set of arc labels of unshifted siblings, L_{sps} the POS set of shifted siblings, L_{cpns} the POS set of unshifted siblings, L_{pls} the set of arc labels of shifted parents, L_{plns} the set of arc labels of unshifted parents, L_{pps} the POS set of shifted parents, L_{ppns} the POS set of unshifted parents.

set of label and POS of child nodes of L
$L_{cls}; L_{clns}; L_{cps}; L_{cpns};$ $S_0wL_{cls}; S_0pL_{cls}; S_1wL_{cls}; S_1pL_{cls};$
set of label and POS of first-level siblings of L
$L_{sls}; L_{slns}; L_{sps}; L_{spns};$ $S_0wL_{sls}; S_0pL_{sls}; S_1wL_{sls}; S_1pL_{sls};$
set of label and POS of parents of L
$L_{pls}; L_{plns}; L_{pps}; L_{ppns};$ $S_0wL_{pls}; S_0pL_{pls}; S_1wL_{pls}; S_1pL_{pls};$

Table 5: Lookahead linearization feature templates for the word L to shift. A subset is shown here. For the full feature set, refer to Table 2 of Puduppully et al. (2016). An identical set of feature templates are defined for S_0.

arc features between S_0 and S_1
$Arc_{left}; Arc_{right};$
lookahead features for L
$L_{is\_descendant}; L_{is\_parent\_or\_sibling};$
are all descendants of S_0 shifted
$S_{0descendants\_shifted};$
feature combination
$S_{0descendants\_shifted}Arc_{left};$ $S_{0descendants\_shifted}Arc_{right};$ $S_{0descendants\_shifted}\ L_{is\_descendant};$ $S_{0descendants\_shifted}\ L_{is\_parent\_or\_sibling};$

Table 6: Graph linearization feature templates

Algorithm 4: transition-based linearization

```
Input: $C$, a set of input syntactic constraints
Output: The highest-scored final state
candidates ← $([\,], set(1..n), \emptyset)$
agenda ← $\emptyset$
for $i \leftarrow 1..2n$ do
    for $s$ in candidates do
        for action in GETPOSSIBLEACTIONS(s, C) do
            agenda ← APPLY($s$, action)
    candidates ← TOP-K(agenda)
    agenda ← $\emptyset$
best ← BEST(candidates)
return best
```

Rules for *be*
attr['partic'] == 'pres' → being
attr['partic'] == 'past' → been
attr['tense'] == 'past'
sbj.attr['num'] == 'sg' → was
sbj.attr['num'] == 'pl' → were
other → [was,were]
attr['tense'] == 'pres'
sbj.attr['num'] == 'sg' → is
sbj.attr['num'] == 'pl' → are
other → [am,is,are]
Rules for other verbs
attr['partic'] == 'pres' → wik.get(lemma, VBG)
attr['partic'] == 'past' → wik.get(lemma, VBN)
attr['tense'] == 'past' → wik.get(lemma, VBD)
attr['tense'] == 'pres'
sbj.attr['num'] == 'sg' → wik.get(lemma, VBZ)
other → wik.getall(lemma)
Rules for other types
lemma==a → [a,an]
lemma==not → [not,n't]
attr['num'] == 'sg' → wik.get(lemma,NNP/NN)
attr['num'] == 'pl' → wik.get(lemma,NNPS/NNS)

Table 7: Lemma rules. All rules are in the format: conditions → candidate inflections. Nested conditions are listed in multi-lines with indentation. *wik* denotes english wiktionary.

agenda, each of transition actions in GETPOSSIBLEACTIONS is applied. The top-*k* states are updated in the agenda for the next step. The process repeats for 2*n* steps as each word needs to be shifted once on to the stack and reduced once. After 2*n* steps, the highest scoring state in agenda is taken as the output. The complexity of algorithm is *n*^{2}, as it takes 2*n* steps to complete and during each step, the number of transition actions is proportional to ρ. Given a configuration C, the score of a possible action a is calculated as:

$$Score(a) = \vec{\theta} \cdot \Phi(\vec{C}, a),$$

where $\vec{\theta}$ is the model parameter vector and $\Phi(\vec{C}, a)$ denotes a feature vector consisting of *configuration* and *action* components. Given a set of labeled training examples, the averaged perceptron with early update (Collins and Roark, 2004) is used.

3.3 Morphological Generation

The last step is to inflate the lemmas in the sentence. There are three POS categories, including nouns, verbs and articles, for which we need to generate morphological forms. We use Wiktionary[2] as a basis and write a small set of rules similar to that used in Song et al. (2014), listed in Table 7, to generate a candidate set of inflections. An averaged perceptron classifier (Collins, 2002) is trained for each lemma. For distinguishing between singular and plural candidate verb forms, the feature templates in Table 8 are used.

4 Joint Method

We design a joint method for function word prediction (Section 3.1), linearization (Section 3.2) and morphological generation (Section 3.3) by further extending the transition-based system of Section 3.2, integrating actions for function word prediction and morphological generation.

[2] https://en.wiktionary.org/

Features for predicting singular/ plural verb forms
WORD(*n-1*)WORD(*n-2*)WORD(*n-3*); COUNT_SUBJ(*n*);
COUNT(*n-1*)COUNT(*n-2*)COUNT(*n-3*); SUBJ(*n*);
WORD(*n-1*)WORD(*n-2*); COUNT(*n-1*)COUNT(*n-2*);
WORD(*n-1*); COUNT(*n-1*); WORD(*n+1*); COUNT(*n+1*);

Table 8: Feature templates for predicting singular/ plural verb forms. Indices on the surface string: *n* – word index; Functions: WORD – word at index *n*; COUNT – word at *n* is singular or plural form; SUBJ – word at subject of *n*; COUNT_SUBJ – word at subject of *n* is singular or plural form.

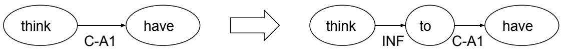


Figure 5: Example for SPLITARC-*to*.

4.1 Transition Actions

In addition to SHIFT, LEFTARC and RIGHTARC in Section 3.2.1, we use the following new transition actions for inserting function words:

- INSERT, inserts comma at the present position;
- SPLITARC-*Word*, splits an arc in the input graph *C*, inserting a function word between the words connected by the arc. Here *Word* specifies the function word being inserted (Figure 5).

We generate a candidate set of inflections for each lemma following the approach in Section 3.3. For each candidate inflection, we generate a corresponding SHIFT transition action. The rules in Table 7 are used to prune impossible inflections.[3]

Input lemmas: $\{\text{think}_1, \text{price}_2, ._3, \text{increase}_4, \text{be}_5, \text{have}_6, \text{meanwhile}_7\}$

	Transition	σ	ρ	A
0		[]	{1...7}	$\emptyset$
1	SH-meanwhile	[7]	{1...6}	
2	IN	[7]	{1...6}	
3	SH-prices	[7 2]	{1,3,4,5,6}	
4	SH-are	[7 2 5]	{1,3,4,6}	
5	SH-thought	[7 2 5 1]	{3,4,6}	
6	SP-to	[7 2 5 1]	{3,4,6}	
7	SH-have	[7 2 5 1 6]	{3,4}	
8	SH-increased	[7 2 5 1 6 4]	{3}	
9	RA	[7 2 5 1 6]	{3}	$A \cup \{6 \rightarrow 4\}$
10	RA	[7 2 5 1]	{3}	$A \cup \{1 \rightarrow 6\}$
11	RA	[7 2 5]	{3}	$A \cup \{5 \rightarrow 1\}$
12	SH-.	[7 2 5 3]	{}	
13	RA	[7 2 5]	{}	$A \cup \{5 \rightarrow 3\}$
14	LA	[7 5]	{}	$A \cup \{2 \leftarrow 5\}$
15	LA	[5]	{}	$A \cup \{7 \leftarrow 5\}$

Table 9: Transition action sequence for linearizing the sentence in Figure 2. SH - SHIFT, SP - SPLITARC, RA - RIGHTARC, LA - LEFTARC, IN - INSERT. POS is not shown in SHIFT actions.

Table 9 shows the transition actions to linearize the graph in Figure 2. These newly introduced transition actions result in variability in the number of transition actions. With function word prediction, the number of transition actions for a bag of *n* words is not necessarily *2n-1*. For example, considering an INSERT, SPLITARC-*to* or SPLITARC-*that* action post each SHIFT action, the maximum number of possible actions is *5n-1*. This variance in the number of actions can impact the linear separability of state items. Following Zhu et al. (2013), we use IDLE actions as a form of padding method, which results in completed state items being further expanded up to *5n-1* steps. The joint model uses the same perceptron training algorithm and similar features compared to the baseline model.

4.2 Obtaining Possible Transition Actions Given a Configuration

Given a state $s = ([\sigma|j\ i], \rho, A)$ and an input graph *C*, the possible transition actions include as a subset the transition actions in Algorithm 1 for shallow graph linearization. In addition, for each lemma being shifted, we enumerate its inflections and create SHIFT transition actions for each inflection. Further, we predict SPLITARC, INSERT and IDLE actions to handle function words. If node *i* has a child node in *C*, which is not shifted, we predict SPLITARC and INSERT. If *i* is sibling to *j*, we predict INSERT. If both the stack and buffer are empty, we predict IDLE. Pseudocode for GETPOSSIBLEACTIONS for the joint method is shown in Algorithm 5.

5 Experiments

5.1 Dataset

We work on the deep dataset from the Surface Realisation Shared Task (Belz et al., 2011)[4]. Sentences are represented as sets of unordered nodes with labeled semantic edges between them. Semantic representation is obtained by merging Nombank (Meyers et al., 2004), Propbank (Palmer et al., 2005) and syntactic dependencies. Edge labeling follows PropBank annotation scheme such as {*A0, A1, ... An*}. The nodes are annotated with lemma and where appropriate number, tense and participle features. Function words including

[3] For example in Figure 2, *price* is the subject of *be* and if *be* is in present tense and *price* is in plural form, the inflections {*am, is, was, were*} are impossible and *are* is the correct inflection for *be*. We therefore generate transition action as SHIFT-*are*.

[4] http://www.nltg.brighton.ac.uk/research/sr-task/

Algorithm 5: GETPOSSIBLEACTIONS for deep graph linearization, where C is a input graph

```
Input: A state $s = ([\sigma|j\ i], \rho, A)$ and graph $C$
Output: A set of possible transition actions $T$
$T \leftarrow \emptyset$
if $s.\sigma == \emptyset$ then
    for $k \in s.\rho$ do
        $T \leftarrow T \cup (\text{SHIFT}, POS, k)$
else
    if $\exists k, k \in (\text{DIRECTCHILDREN}(i) \cap s.\rho)$ then
        $\text{SHIFTSUBTREE}(i, \rho)$
    else
        if $A$.LEFTCHILD$(i)$ is NIL then
            $\text{SHIFTSUBTREE}(i, \rho)$
        if $\{j \rightarrow i\} \in C \wedge A$.LEFTCHILD$(j)$ is NIL then
            $T \leftarrow T \cup (\text{RIGHTARC})$
            if $i \in \text{DESCENDANT}(j)$ then
                $\text{PROCESSDESCENDANT}(i, j)$
            if $i \in \text{SIBLING}(j)$ then
                $\text{PROCESSSIBLING}(i, j)$
        else if $\{j \leftarrow i\} \in C$ then
            $T \leftarrow T \cup (\text{LEFTARC})$
            if $i \in \text{SIBLING}(j)$ then
                $\text{PROCESSSIBLING}(i, j)$
        else
            if $size(s.\sigma)$ == 1 then
                $\text{SHIFTPARENTANDSIBLINGS}(i)$
            else
                if $i \in \text{DESCENDANT}(j)$ then
                    $\text{PROCESSDESCENDANT}(i, j)$
                if $i \in \text{SIBLING}(j)$ then
                    $\text{PROCESSSIBLING}(i, j)$
if $C$.Children$(i)$ $\wedge s.\rho \neq \emptyset$ then
    $T \leftarrow T \cup (\text{SPLITARC} - to)$
    $T \leftarrow T \cup (\text{SPLITARC} - that)$
if $C$.Children$(i)$ $\wedge s.\rho \neq \emptyset \vee i \in \text{SIBLING}(j)$ then
    $T \leftarrow T \cup (\text{INSERT})$
if $s.\sigma == \emptyset \wedge s.\rho == \emptyset$ then
    $T \leftarrow T \cup (\text{IDLE})$
return $T$
```

that complementizer, *to* infinitive and commas are omitted from the input. There are two punctuation features for information about brackets and quotes. Table 10 shows a sample training instance.

Out of 39k total training instances, 2.8k are non-projective, which we discard. We exclude instances which result in non-projective dependencies mainly because our transition actions predict only projective dependencies. It has been derived from the arc-standard system (Nivre, 2008). There are 1.8k training instances with a mismatch between edges in the input deep graph and *gold output tree*. The gold output tree is the corresponding shallow tree from the shared task. We approach the task of linearization as extracting a linearized tree from the input semantic graph. So we exclude those instances which do not have edges corresponding to gold tree i.e mismatch between edges of gold tree and input graph. After excluding these instances, we have 34.3k training instances. We also exclude 800 training instances where the function words *to* and *that* have more than one child, and around 100 training instances where function words' parent and child nodes are not connected by an arc in the deep graph. The above cases are deemed annotation mistakes. We thus train on a final subset of 33.4k training instances. The development set comprises 1034 instances and the test set comprises 2398 instances. Evaluation is done using the BLEU metric (Papineni et al., 2002).

Input (unordered lemma-formed graph):

Sem	**ID**	**PID**	**Lemma**	**Attr**	**Lexeme**
SROOT	1	0	be	tense=pres	are
ADV	2	1	meanwhile		meanwhile
P	3	1	.		.
SBJ	4	1	start.02	num=pl	starts
A1	5	4	housing	num=sg	housing
AM-TMP	6	4	september	num=sg	september
VC	9	1	think.01	partic=past	thought
A1	4	9			
C-A1	10	9	have		have
VC	11	10	inch.01	partic=past	inched
A1	4	11			
A5	12	11	upward		upward

Table 10: Deep type training instance from Surface Realisation Shared Task 2011. *Sem* – semantic label, *ID* – unique ID of node within graph, *PID* – the ID of the parent, *Attr* – Attributes such as partic (participle), tense or number, *Lexeme* – lexeme which is resolved using wiktionary and rules in Table 7.

6 Development Results

6.1 Influence of Beam Size

We study the effect of beam size on the accuracies of joint model in Figure 6, by varying the beam size and comparing the accuracies on development dataset over training iterations. Beam sizes of 64 and 128 perform the best. However, beam size 128 does not improve the performance significantly, yet is twice as slow compared to a beam size 64. So we retain a 64 beam for further experiments.

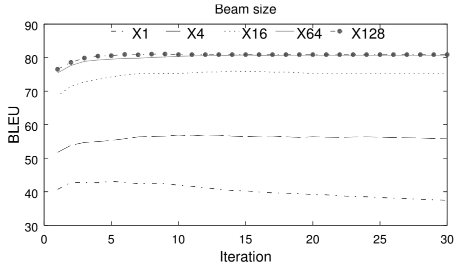


Figure 6: Influence of beam sizes.

	Pipeline	Joint
to infinitive	92.7	94.1
that complementizer	70.6	76.5
count of *comma*	60.2	63.3

Table 11: Average F-measure for function word prediction for development set.

6.2 Pipeline vs Joint Model

We compare the results of the joint model with the pipeline baseline system. Table 11 shows the development results of function word prediction, and Table 12 shows the overall development results. Our joint model of Transition-Based Deep Input Linearization (TBDIL) achieves an improvement of 5 BLEU points over the pipeline using the same feature source and training algorithm. Thanks to the sharing of word order information, the joint model improves function word prediction compared to the pipeline, which forbids such feature integration because function word prediction is the first step, taken before order becomes available.

7 Final Results

Table 13 shows the final results. The best performing system for the Shared Task was STUMABA-D by Bohnet et al. (2011), which leverages a large-scale n-gram language model. The joint model TBDIL significantly outperforms the pipeline system and achieves an improvement of 1 BLEU point over STUMABA-D, obtaining 80.49 BLEU without making use of external resources.

8 Analysis

Table 14 shows sample outputs from the Pipeline system and the corresponding output from TBDIL. In the first instance, the function word *to* is incorrectly predicted in the arc between nodes *does* and *yield* in the pipeline system. In case of TBDIL, the n-gram feature helps avoid incorrect insertion of *to* which demonstrates the advantage of integrating information across stages. In the second instance, because of incorrect linearization, there is error propagation to morphological generation in the pipeline system. In particular, *economists* is linearized to the object part of the sentence and the subject is singular. This, in turn, results in the incorrect prediction of morphological form of verb *read* as its singular variant. In TBDIL, in contrast, the joint modelling of linearization and morphology helps ordering the sentence correctly.

System	BLEU Score
Pipeline	75.86
TBDIL	**80.77**

Table 12: Development results.

System	BLEU Score
STUMABA-D	79.43
Pipeline	70.99
TBDIL	**80.49**

Table 13: Test results.

	output
ref.	if it does n't yield on these matters and eventually begin talking directly to the anc
Pipeline	if it does not to yield on these matters and eventually begin talking directly to the anc
TBDIL	if it does n't yield on these matters and eventually begin talking directly to the anc
ref.	economists who read september 's low level of factory job growth as a sign of a slowdown
Pipeline	september 's low level of factory job growth who as a sign of a slowdown reads economists
TBDIL	economists who read september 's low level of factory job growth as a sign of a slowdown

Table 14: Example outputs.

9 Conclusion

We showed the usefulness of a joint model for the task of Deep Linearization, by taking (Puduppully et al., 2016) as the baseline and extending it to perform joint graph linearization, function word prediction and morphological generation. To our knowledge, this is the first work to use Transition-Based method for joint NLG from semantic structure. Our system gave the highest scores reported for the NLG 2011 shared task on Deep Input Linearization (Belz et al., 2011).

Acknowledgments

We thank Litton Kurisinkel for helpful discussions and the anonymous reviewers for their detailed and constructive comments. Yue Zhang is supported by the Singapore Ministry of Education (MOE) AcRF Tier 2 grant T2MOE201301.

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider, 2013. *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, chapter Abstract Meaning Representation for Sembanking, pages 178–186. Association for Computational Linguistics.

Srinivas Bangalore, Owen Rambow, and Steve Whittaker, 2000. *INLG'2000 Proceedings of the First International Conference on Natural Language Generation*, chapter Evaluation Metrics for Generation.

Regina Barzilay and Kathleen R McKeown. 2005. Sentence fusion for multidocument news summarization. *Computational Linguistics*, 31(3):297–328.

Anja Belz, Michael White, Dominic Espinosa, Eric Kow, Deirdre Hogan, and Amanda Stent. 2011. The first surface realisation shared task: Overview and evaluation results. In *Proceedings of the 13th European workshop on natural language generation*, pages 217–226. Association for Computational Linguistics.

Bernd Bohnet, Leo Wanner, Simon Mille, and Alicia Burga. 2010. Broad coverage multilingual deep sentence generation with a stochastic multi-level realizer. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 98–106. Association for Computational Linguistics.

Bernd Bohnet, Simon Mille, Benoît Favre, and Leo Wanner. 2011. Stumaba: from deep representation to surface. In *Proceedings of the 13th European workshop on natural language generation*, pages 232–235. Association for Computational Linguistics.

John Carroll and Stephan Oepen. 2005. High efficiency realization for a wide-coverage unification grammar. In *Second International Joint Conference on Natural Language Processing: Full Papers*.

John Carroll, Ann Copestake, and Dan Flickinger. 1999. An efficient chart generator for (semi-) lexicalist grammars.

Pi-Chuan Chang and Kristina Toutanova. 2007. A discriminative syntactic word order model for machine translation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 9–16. Association for Computational Linguistics.

Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. *Proceedings of the 2014 Conference on EMNLP*, 1:740–750.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 111. Association for Computational Linguistics.

Michael Collins, 2002. *Proceedings of the 2002 Conference on EMNLP (EMNLP 2002)*, chapter Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms.

A De Gispert, M Tomalin, and W Byrne. 2014. Word ordering with phrase-based grammars. *14th Conference of the European Chapter of the Association for Computational Linguistics 2014, EACL 2014*, pages 259–268.

Dominic Espinosa, Michael White, and Dennis Mehay. 2008. Hypertagging: Supertagging for surface realization with ccg. In *Proceedings of ACL-08: HLT*, pages 183–191. Association for Computational Linguistics.

Rose Jenny Finkel and D. Christopher Manning. 2009. Joint parsing and named entity recognition. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 326–334. Association for Computational Linguistics.

Alice H. Oh and Alexander I. Rudnicky, 2000. *ANLP-NAACL 2000 Workshop: Conversational Systems*, chapter Stochastic Language Generation for Spoken Dialogue Systems.

Irene Langkilde and Kevin Knight. 1998. Generation that exploits corpus-based statistical knowledge. In *COLING 1998 Volume 1: The 17th International Conference on Computational Linguistics*.

John Lee and Stephanie Seneff. 2006. Automatic grammar correction for second-language learners. In *INTERSPEECH*, pages 1978–1981.

Qi Li and Heng Ji. 2014. Incremental joint extraction of entity mentions and relations. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 402–412. Association for Computational Linguistics.

Jiangming Liu and Yue Zhang. 2015. An empirical comparison between n-gram and syntactic language models for word ordering. In *Proceedings of the 2015 Conference on EMNLP*, pages 369–378, Lisbon, Portugal, September. Association for Computational Linguistics.

Yijia Liu, Yue Zhang, Wanxiang Che, and Bing Qin. 2015. Transition-based syntactic linearization. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*, pages 113–122.

Igor Aleksandrovič Melčuk. 1988. *Dependency Syntax: theory and practice*. SUNY press.

Igor Aleksandrovič Melčuk. 2015. *Semantics: From meaning to text*, volume 3. John Benjamins Publishing Company.

Adam Meyers, Ruth Reeves, Catherine Macleod, Rachel Szekely, Veronika Zielinska, Brian Young, and Ralph Grishman. 2004. Annotating noun argument structure for nombank. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC'04)*. European Language Resources Association (ELRA).

Joakim Nivre and Mario Scholz. 2004. Deterministic dependency parsing of english text. In *Proceedings of the 20th international conference on Computational Linguistics*, page 64. Association for Computational Linguistics.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics, Volume 34, Number 4, December 2008*.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics, Volume 31, Number 1, March 2005*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*.

Ratish Puduppully, Yue Zhang, and Manish Shrivastava. 2016. Transition-based syntactic linearization with lookahead features. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 488–493. Association for Computational Linguistics.

Tao Qian, Yue Zhang, Meishan Zhang, Yafeng Ren, and Donghong Ji. 2015. A transition-based model for joint segmentation, pos-tagging and normalization. In *Proceedings of the 2015 Conference on EMNLP*, pages 1837–1846, Lisbon, Portugal, September. Association for Computational Linguistics.

Ehud Reiter and Robert Dale. 1997. Building applied natural language generation systems. *Natural Language Engineering*, 3(01):57–87.

Allen Schmaltz, Alexander M Rush, and Stuart M Shieber. 2016. Word ordering without syntax. *arXiv preprint arXiv:1604.08633*.

Linfeng Song, Yue Zhang, Kai Song, and Qun Liu. 2014. Joint morphological generation and syntactic linearization. *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 1522–1528.

Charles Sutton, Andrew McCallum, and Khashayar Rohanimanesh. 2007. Dynamic conditional random fields: Factorized probabilistic models for labeling and segmenting sequence data. *Journal of Machine Learning Research*, 8(Mar):693–723.

Erik Velldal and Stephan Oepen, 2006. *Proceedings of the 2006 Conference on EMNLP*, chapter Statistical Ranking in Tactical Generation, pages 517–525. Association for Computational Linguistics.

Stephen Wan, Mark Dras, Robert Dale, and Cécile Paris. 2009. Improving grammaticality in statistical sentence generation: Introducing a dependency spanning tree algorithm with an argument satisfaction model. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 852–860. Association for Computational Linguistics.

Michael White and Rajakrishnan Rajkumar. 2009. Perceptron reranking for ccg realization. In *Proceedings of the 2009 Conference on EMNLP*, pages 410–419. Association for Computational Linguistics.

Michael White. 2004. Reining in ccg chart realization. In *Natural Language Generation*, pages 182–191. Springer Berlin Heidelberg.

Michael White. 2006. Efficient realization of coordinate structures in combinatory categorial grammar. *Research on Language and Computation*, 4(1):39–75.

Yue Zhang and Stephen Clark. 2010. A fast decoder for joint word segmentation and pos-tagging using a single discriminative model. In *Proceedings of the 2010 Conference on EMNLP*, pages 843–852. Association for Computational Linguistics.

Yue Zhang and Stephen Clark. 2011. Syntactic processing using the generalized perceptron and beam search. *Computational linguistics*, 37(1):105–151.

Yue Zhang and Stephen Clark. 2015. Discriminative syntax-based word ordering for text generation. *Computational Linguistics*, 41(3):503–538.

Yue Zhang, Kai Song, Linfeng Song, Jingbo Zhu, and Qun Liu. 2014. Syntactic smt using a discriminative text generation model. In *Proceedings of the 2014 Conference on EMNLP*, pages 177–182, Doha, Qatar, October. Association for Computational Linguistics.

Yue Zhang. 2013. Partial-tree linearization: generalized word ordering for text synthesis. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 2232–2238. AAAI Press.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *ACL (1)*, pages 434–443.

A Obtaining possible transition actions given a configuration for Shallow Graph

During shallow linearization, a state is represented by $s = ([\sigma|j\ i], \rho, A)$ and C is the input graph. Given C, the Decoder outputs actions which extract syntactic tree from the graph. Thus the Decoder outputs RIGHTARC or LEFTARC only if corresponding arc exists in C. The detailed pseudocode is given in Algorithm 1. If *i* has *direct child nodes* in *C*, the descendants of *i* are shifted (line 6-7) (see Algorithm 3). Here, *direct child nodes* (see Algorithm 2) include those child nodes of *i* for which *i* is the only parent or if there is more than one parent then every other parent is shifted on to the stack without possibility to reduce the child node. If no *direct child node* is in buffer, then descendants of *i* are shifted (line 9-10). Now, there are three configurations possible between *i* and *j*: 1. *i* and *j* are connected by arc in *C*. This results in RIGHTARC or LEFTARC action; 2. *i* is descendant of *j*. In this case the parents of *i* (such that they are descendants of *j*) and siblings of *i* through such parents are shifted. 3. *i* is sibling of *j*. In this case, the parents of *i* and their descendants are shifted such that *A* remains consistent. Additionally, because the input is a graph structure, more than one of the above configuration can occur simultaneously. We analyse the three configurations in detail below.

Since the *direct child nodes* of *i* are shifted, $\{j \leftarrow i\}$ results in a LEFTARC action (line 18). Also because the input is a graph, *i* can be a sibling node of *j*. In this case, the valid parents and siblings of *i* are shifted. We iterate through the other elements in stack to identify the valid parents and siblings. These conditions are encapsulated in PROCESSSIBLING (line 20). Conditions for RIGHTARC are similar to that of LEFTARC with the following differences. We ensure that there is no left arc relationship for *j* in *A* (line 11). If there is a left arc relationship for *j* in *A*, it means that in an arc-standard setting, the RIGHTARC actions for *j* have already been made. If *i* is a descendant of *j*, valid parents and siblings of *i* are shifted. We iterate through the parents of *i* and those parents which are in turn descendants of *j* and not shifted on to the stack are valid parents. We shift the parent and the subtree through each such parent. These conditions are denoted by PROCESSDESCENDANT (line 14).

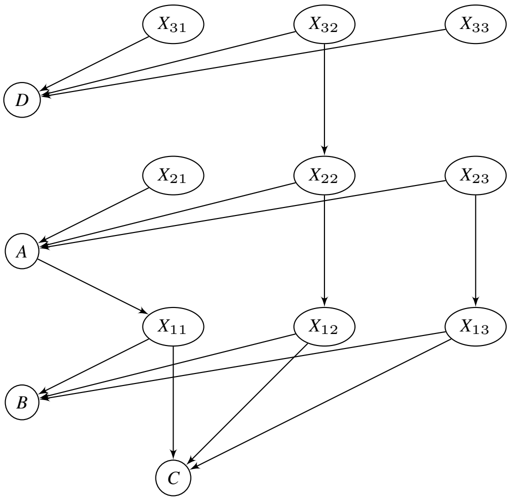


Figure 7: Sample graph to illustrate PROCESSSIBLING

If there is no arc between *j* and *i* and there is only one element on the stack, then the parents and siblings of *i* are shifted (line 22-23). If there is more than one element on the stack, and if *i* is descendant of *j*, then we use PROCESSDESCENDANT (line 25-26). If *i* is sibling to *j* we use PROCESSSIBLING (line 27-28).

Consider an example to see the working of PROCESSSIBLING in detail. In PROCESSSIBLING, we need to ensure that *i* is in stack because of sibling relation with *j* and we need to shift the valid parent nodes of i and their descendants. We call these valid nodes *inflection points*. Consider the following stack entries [*D, A, B, C*] with *C* as stack top. Assume that the input graph is as in Figure 7. *C* is sibling of *B* through *B*'s parents X_{11}, X_{12}, X_{13}. Out of these, only X_{11} and X_{12} are valid parents. X_{13} is sibling to *A* through *A*'s parent X_{23}. But X_{23} is in turn neither descendant of *D* nor sibling of *D*. Thus X_{13} is not a valid inflection point for C. Now, X_{12} is sibling of *A* through *A*'s parent X_{22}. X_{22} is in turn sibling of *D* through X_{32}. Thus there is a path to the stack bottom through a path of siblings/ descendant. In case of X_{11}, X_{11} is descendant of stack element *A* and is thus valid. X_{11} and X_{12} are called valid inflection points. If inflection point is a common parent to both S_0 and S_1 then inflection point and its descendants are shifted. Instead, if inflection point is ancestor to S_0, then parents of S_0 (say P_0) which are descendants of inflection point are shifted. Additionally, descendants of P_0 are shifted.