Language Generation via DAG Transduction

Yajie Ye, Weiwei Sun and **Xiaojun Wan**
Institute of Computer Science and Technology, Peking University
The MOE Key Laboratory of Computational Linguistics, Peking University
{yeyajie,ws,wanxiaojun}@pku.edu.cn

Abstract

A DAG automaton is a formal device for manipulating graphs. By augmenting a DAG automaton with transduction rules, a DAG transducer has potential applications in fundamental NLP tasks. In this paper, we propose a novel DAG transducer to perform graph-to-program transformation. The target structure of our transducer is a program licensed by a declarative programming language rather than linguistic structures. By executing such a program, we can easily get a surface string. Our transducer is designed especially for natural language generation (NLG) from type-logical semantic graphs. Taking Elementary Dependency Structures, a format of English Resource Semantics, as input, our NLG system achieves a BLEU-4 score of 68.07. This remarkable result demonstrates the feasibility of applying a DAG transducer to resolve NLG, as well as the effectiveness of our design.

1 Introduction

The recent years have seen an increased interest as well as rapid progress in semantic parsing and surface realization based on graph-structured semantic representations, e.g. Abstract Meaning Representation (AMR; Banarescu et al., 2013), Elementary Dependency Structure (EDS; Oepen and Lønning, 2006) and Depedendency-based Minimal Recursion Semantics (DMRS; Copestake, 2009). Still underexploited is a formal framework for manipulating graphs that parallels automata, tranducers or formal grammars for strings and trees. Two such formalisms have recently been proposed and applied for NLP. One is graph grammar, e.g. Hyperedge Replacement Grammar (HRG; Ehrig et al., 1999). The other is DAG automata, originally studied by Kamimura and Slutzki (1982) and extended by Chiang et al. (2018). In this paper, we study DAG transducers in depth, with the goal of building accurate, efficient yet robust natural language generation (NLG) systems.

The meaning representation studied in this work is what we call type-logical semantic graphs, i.e. semantic graphs grounded under type-logical semantics (Carpenter, 1997), one dominant theoretical framework for modeling natural language semantics. In this framework, adjuncts, such as adjective and adverbal phrases, are analyzed as (higher-order) functors, the function of which is to consume complex arguments (Kratzer and Heim, 1998). In the same spirit, generalized quantifiers, prepositions and function words in many languages other than English are also analyzed as higher-order functions. Accordingly, all the linguistic elements are treated as roots in type-logical semantic graphs, such as EDS and DMRS. This makes the typological structure quite flat rather than hierachical, which is an essential distinction between natural language semantics and syntax.

To the best of our knowledge, the only existing DAG transducer for NLG is the one proposed by Quernheim and Knight (2012). Quernheim and Knight introduced a DAG-to-tree transducer that can be applied to AMR-to-text generation. This transducer is designed to handle hierarchical structures with limited reentrencies, and it is unsuitable for meaning graphs transformed from type-logical semantics. Furthermore, Quernheim and Knight did not describe how to acquire graph recognition and transduction rules from linguistic data, and reported no result of practical generation. It is still unknown to what extent a DAG transducer suits realistic NLG.

The design for string and tree transducers

Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Long Papers), pages 1928–1937
Melbourne, Australia, July 15 - 20, 2018. ©2018 Association for Computational Linguistics

(Comon et al., 1997) focuses on not only the logic of the computation for a new data structure, but also the corresponding control flow. This is very similar the imperative programming paradigm: implementing algorithms with exact details in explicit steps. This design makes it very difficult to transform a type-logical semantic graph into a string, due to the fact their internal structures are highly diverse. We borrow ideas from declarative programming, another programming paradigm, which describes what a program must accomplish, rather than how to accomplish it. We propose a novel DAG transducer to perform graph-to-program transformation (§3). The input of our transducer is a semantic graph, while the output is a program licensed by a declarative programming language rather than linguistic structures. By executing such a program, we can easily get a surface string. This idea can be extended to other types of linguistic structures, e.g. syntactic trees or semantic representations of another language.

We conduct experiments on richly detailed semantic annotations licensed by English Resource Grammar (ERG; Flickinger, 2000). We introduce a principled method to derive transduction rules from DeepBank (Flickinger et al., 2012). Furthermore, we introduce a fine-to-coarse strategy to ensure that at least one sentence is generated for any input graph. Taking EDS graphs, a variable-free ERS format, as input, our NLG system achieves a BLEU-4 score of 68.07. On average, it produces more than 5 sentences in a second on an x86_64 GNU/Linux platform with two Intel Xeon E5-2620 CPUs. Since the data for experiments is newswire data, i.e. WSJ sentences from PTB (Marcus et al., 1993), the input graphs are quite large on average. The remarkable accuracy, efficiency and robustness demonstrate the feasibility of applying a DAG transducer to resolve NLG, as well as the effectiveness of our transducer design.

2 Previous Work and Challenges

2.1 Preliminaries

A node-labeled **simple** graph over alphabet Σ is a triple $G = (V, E, \ell)$, where V is a finite set of nodes, $E \subseteq V \times V$ is an finite set of edges and $\ell : V \to \Sigma$ is a labeling function. For a node $v \in V$, sets of its incoming and outgoing edges are denoted by $in(v)$ and $out(v)$ respectively. For an edge $e \in E$, its source node and target node are denoted by $src(e)$ and $tar(e)$ respectively. Generally speaking, a DAG is a **directed acyclic simple graph**. Different from trees, a DAG allows nodes to have multiple incoming edges. In this paper, we only consider DAGs that are **unordered**, **node-labeled**, **multi-rooted**[1] and **connected**.

Conceptual graphs, including AMR and EDS, are both node-labeled and edge-labeled. It seems that without edge labels, a DAG is inadequate, but this problem can be solved easily by using the strategies introduced in (Chiang et al., 2018). Take a labeled edge `proper_q` $\xrightarrow{\texttt{BV}}$ `named` for example[2]. We can represent the same information by replacing it with two unlabeled edges and a new labeled node: `proper_q` $\to$ `BV` $\to$ `named`.

2.2 Previous Work

DAG automata are the core engines of graph transducers (Bohnet and Wanner, 2010; Quernheim and Knight, 2012). In this work, we adopt Chiang et al. (2018)'s design and define a weighted DAG automaton as a tuple $M = \langle \Sigma, Q, \delta, \mathbb{K} \rangle$:

- Σ is an alphabet of node labels.
- Q is a finite set of states.
- $(\mathbb{K}, \oplus, \otimes, 0, 1)$ is a semiring of weights.
- $\delta : \Theta \to \mathbb{K} \backslash \{0\}$ is a weight function that assigns nonzero weights to a finite transition set Θ. Every transition $t \in \Theta$ is of the form

$$\{q_1, \cdots, q_m\} \xrightarrow{\sigma} \{r_1, \cdots, r_n\}$$

  where q_i and r_j are states in Q. A transition t gets m states on the incoming edges of a node and puts n states on the outgoing edges. A transition that does not belong to Θ recieves a weight of zero.

A *run* of M on a DAG $D = \langle V, E, \ell \rangle$ is an edge labeling function $\rho : E \to Q$. The weight of a run ρ (denoted as $\delta'(\rho)$) is the product of all weights of local transitions:

$$\delta'(\rho) = \bigotimes_{v \in V} \delta\left(\rho(in(v)) \xrightarrow{\ell(v)} \rho(out(v))\right)$$

Here, for a function f, we use $f(\{a_1, \cdots, a_n\})$ to represent $\{f(a_1), \cdots, f(a_n)\}$. If $\mathbb{K}$ is a boolean semiring, the automata fall backs to an unweighted

[1] A node without incoming edges is called *root* and a node without outgoing edges is called *leaf*.

[2] `proper_q` and `named` are node labels, while `BV` is the edge label.

DAG automata or DAG acceptor. A *accepting run* or *recognition* is a run, the weight of which is 1, meaning *true*.

2.3 Challenges

The DAG automata defined above can only be used for recognition. In order to generate sentences from semantic graphs, we need DAG transducers. A DAG transducer is a DAG automata-augmented computation model for transducing well-formed DAGs to other data structures. Quernheim and Knight (2012) focused on feature structures and introduced a *DAG-to-Tree* transducer to perform graph-to-tree transformation. The input of their transducer is limited to single-rooted DAGs. When the labels of the leaves of an output tree in order are interpreted as words, this transducer can be applied to generate natural language sentences.

When applying Quernheim and Knight's DAG-to-Tree transducer on type-logic semantic graphs, e.g. ERS, there are some significant problems. First, it lacks the ability to *reverse* the direction of edges during transduction because it is difficult to keep acyclicy anymore if edge reversing is allowed. Second, it cannot handle multiple roots. But we have discussed and reached the conclusion that multi-rootedness is a necessary requirement for representing type-logical semantic graphs. It is difficult to decide which node should be the tree root during a 'top-down' transduction and it is also difficult to merge multiple unconnected nodes into one during a 'bottom-up' transduction. At the risk of oversimplifying, we argue that the function of the existing DAG-to-Tree transducer is to transform a hierachical structure into another hierarchical structure. Since the type-local semantic graphs are so flat, it is extremely difficult to adopt Quernheim and Knight's design to handle such graphs. Third, there are unconnected nodes with direct dependencies, meaning that their correpsonding surface expressions appear to be very close. The conceptual nodes `_even_x_deg` and `_steep_a_1` in Figure 4 are an example. It is extremely difficult for the DAG-to-Tree transducer to handle this situation.

3 A New DAG Transducer

3.1 Basic Idea

In this paper, we introduce a design of transducers that can perform structure transformation towards many data structures, including but not limited to trees. The basic idea is to give up the *rewritting* method to directly generate a new data structure piece by piece, while recognizing an input DAG. Instead, our transducer obtains target structures based on side effects of DAG recognition. The output of our transducer is no longer the target data structure itself, e.g. a tree or another DAG, and is now a program, i.e. a bunch of statements licensed by a particular **declarative** programming language. The target structures are constructed by executing such programs.

Since our main concern of this paper is natural language generation, we take strings, namely sequences of words, as our target structures. In this section, we introduce an extremely simple programming language for string concatenation and then details about how to leverage the power of declarative programming to perform DAG-to-string transformation.

3.2 A Declarative Programming Language

The syntax in the BNF format of our declarative programming language, denoted as $\mathcal{L}_c$, for string calculation is:

$$\begin{aligned}\langle program\rangle &::= \langle statement\rangle* \\ \langle statement\rangle &::= \langle variable\rangle = \langle expr\rangle \\ \langle expr\rangle &::= \langle variable\rangle \mid \langle string\rangle \\ &\quad \mid \langle expr\rangle + \langle expr\rangle\end{aligned}$$

Here a *string* is a sequence of characters selected from an alphabet (denoted as Σ_{out}) and can be empty (denoted as ϵ). The semantics of '=' is *value assignment*, while the semantics of '+' is *string concatenation*. The value of variables are strings. For every statement, the left hand side is a variable and the right hand side is a sequence of string literals and variables that are combined through '+'. Equation (1) presents an exmaple program licensed by this language.

$$\begin{aligned} S &= x_{21} + \texttt{want} + x_{11} \\ x_{11} &= \texttt{to} + \texttt{go} \\ x_{21} &= x_{41} + \texttt{John} \\ x_{41} &= \epsilon \end{aligned} \tag{1}$$

After solving these statements, we can query the values of all variables. In particular, we are interested in S, which is related to the desired natural language expression `John want to go`[3].

[3] The expression is a sequence of lemmas rather than inflected words. Refer to §4 for more details.

Using the relation between the variables, we can easily convert the statements in (1) to a rooted tree. The result is shown in Figure 1. This tree is significantly different from the target structures discussed by Quernheim and Knight (2012) or other normal tree transducers (Comon et al., 1997). This tree represents calculation to solve the program. Constructing such *internal* trees is an essential function of the *compiler* of our programming language.

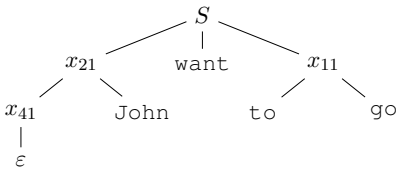

Figure 1: Variable relation tree.

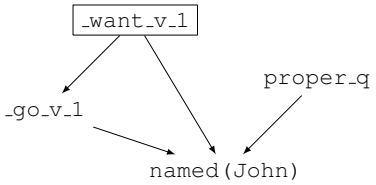

Figure 2: An input graph. The intended reading is *John wants to go.*

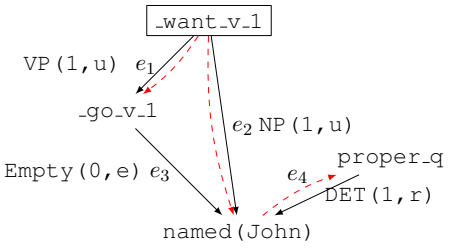

Figure 3: A run of the graph in Figure 2.

3.3 Informal Illustration

We introduce our DAG transducer using a simple example. Figure 2 shows the original input graph $D = (V, E, \ell)$. Without any loss of generality, we remove edge labels. Table 1 lists the rule set—R—for this example. Every row represents an applicable transduction rule that consists of two parts. The left column is the recognition part displayed in the form $I \xrightarrow{\sigma} O$, where I, O and σ decode the state set of incoming edges, the state set of outgoing edges and the node label respectively. The right column is the generation part which consists of (multiple) templates of statements licensed by the programming language defined in the previous section. In practice, two different rules may have a same recognition part but different generation parts.

Every state q is of the form $l(n, d)$ where l is the finite state label, n is the count of possible variables related to q, and d denotes the direction. The value of d can only be `r` (reversed), `u` (unchanged) or `e`(empty). Variable $v_{l(j,d)}$ represents the jth ($1 \leq j \leq n$) variable related to state q. For example, $v_{\texttt{X(2,r)}}$ means the second variable of state `X(3,r)`. There are two special variables: S, which corresponds to the whole sentence and L, which corresponds to the output string associated to current node label. It is reasonable to assume that there exists a function $\psi : \Sigma \rightarrow \Sigma^*_{\text{out}}$ that maps a particular node label, i.e. concept, to a surface string. Therefore L is determined by ψ.

Now we are ready to apply transduction rules to translate D into a string. The transduction consists of two steps:

Recognition The goal of this step is to find an edge labeling function $\rho : E \rightarrow Q$ which satisfies that for every node v, $\rho(in(v)) \xrightarrow{\ell(v)} \rho(out(v))$ matches the recognition part of a rule in R. The recognition result is shown in Figure 3. The red dashed edges in Figure 3 make up an *intermediate graph* $T(\rho)$, which is a subgraph of D if edge direction is not taken into account. Sometimes, $T(\rho)$ paralles the syntactic structure of an output sentence. For a labeling function ρ, we can construct *intermediate graph* $T(\rho)$ by checking the direction parameter of every edge state. For an edge $e = (u, v) \in E$, if the direction of $\rho(e)$ is `r`, then (v, u) is in $T(\rho)$. If the direction is `u`, then (u, v) is in $T(\rho)$. If the direction is `e`, neither (u, v) nor (v, u) is included. The recognition process is slightly different from the one in Chiang et al. (2018). Since incoming edges with an `Empty(0,e)` state carry no semantic information, they will be ignored during recognition. For example, in Figure 3, we will only use e_2 and e_4 to match transducation rules for node `named(John)`.

Instantiation We use $rule(v)$ to denotes the rule used on node v. Assume s is the generation part of $rule(v)$. For every edge e_i adjacent to v, assume $\rho(e_i) = l(n, d)$. We replace L with $\psi(\ell(v))$ and replace every occurrence of $v_{l(j,d)}$ in s with a new variable x_{ij} ($1 \leq j \leq n$). Then we

$Q = \{\texttt{DET(1,r)}, \texttt{Empty(0,e)}, \texttt{VP(1,u)}, \texttt{NP(1,u)}\}$		
Rule	For Recognition	For Generation
1	$\{\} \xrightarrow{\texttt{proper\_q}} \{\texttt{DET(1,r)}\}$	$v_{\texttt{DET(1,r)}} = \epsilon$
2	$\{\} \xrightarrow{\texttt{\_want\_v\_1}} \{\texttt{VP(1,u)}, \texttt{NP(1,u)}\}$	$S = v_{\texttt{NP(1,u)}} + L + v_{\texttt{VP(1,u)}}$
3	$\{\texttt{VP(1,u)}\} \xrightarrow{\texttt{\_go\_v\_1}} \{\texttt{Empty(0,e)}\}$	$v_{\texttt{VP(1,u)}} = \texttt{to} + L$
4	$\{\texttt{NP(1,u)}, \texttt{DET(1,r)}\} \xrightarrow{\texttt{named}} \{\}$	$v_{\texttt{NP(1,u)}} = v_{\texttt{DET(1,r)}} + L$

Table 1: Sets of states (Q) and rules (R) that can be used to process the graph in Figure 2.

get a newly generated expression for v. For example, node _want_v_1 is recognized using Rule 2, so we replace $v_{\texttt{NP(1,u)}}$ with x_{21}, $v_{\texttt{VP(1,u)}}$ with x_{11} and L with want. After instantiation, we get all the statements in Equation (1).

Our transducer is suitable for type-logical semantic graphs. Because declarative programming brings in more freedom for graph transduction. We can arrange the variables in almost any order without regard to the edge directions in original graphs. Meanwhile, the multi-rooted problem can be solved easily because the generation is based on side effects. We do not need to decide which node is the tree root.

3.4 Definition

The formal definition of our DAG transducer described above is a tuple $M = (\Sigma, Q, R, w, V, S)$ where:

- Σ is an alphabet of node labels.
- Q is a finite set of edge states. Every state $q \in Q$ is of the form $l(n, d)$ where l is the state label, n is the variable count and d is the direction of state which can be r, u or e.
- R is a finite set of rules. Every rule is of the form $I \xrightarrow{\sigma} \langle O, E \rangle$. E can be any kind of statement in a declarative programming language. It is called the generation part. I, σ and O have the same meanings as they do in the previous section and they are called the recognition part.
- w is a score function. Given a particular run and an anchor node, w assigns a score to measure the preference for a particular rule at this anchor node.
- V is the set of parameterized variables that can be used in every expression.
- $S \in V$ is a distinguished, global variable. It is like the 'goal' of a program.

4 DAG Transduction-based NLG

Different languages exhibit different morpho-syntactic and syntactico-semantic properties. For example, Russian and Arabic are morphologically-rich languages and heavily utilize grammatical markers to indicate grammatical as well as semantic functions. On the contrary, Chinese, as an analytic language, encodes grammatical and semantic information in a highly configurational rather than either inflectional or derivational way. Such differences affects NLG significantly. Considering generating Chinese sentences, it seems sufficient to employ our DAG transducer to obtain a sequence of lemmas, since no morpholical production is needed. But for morphologically-rich languages, we do need to model complex morphological changes.

To unify a general framework for DAG transduction-based NLG, we propose a two-step strategy to achive meaning-to-text transformation.

- In the first phase, we are concerned with syntactico-semantic properties and utilize our DAG transducer to translate a semantic graph into sequential lemmas. Information such as tense, apsects, gender, etc. is attached to anchor lemmas. Actually, our transducer generates "want.PRES" rather than "wants". Here, "PRES" indicates a particular tense.
- In the second phase, we are concerned with morpho-syntactic properties and utilize a neural sequence-to-sequence model to obtain final surface strings from the outputs of the DAG transducer.

5 Inducing Transduction Rules

We present an empirical study on the feasibility of DAG transduction-based NLG. We focus on

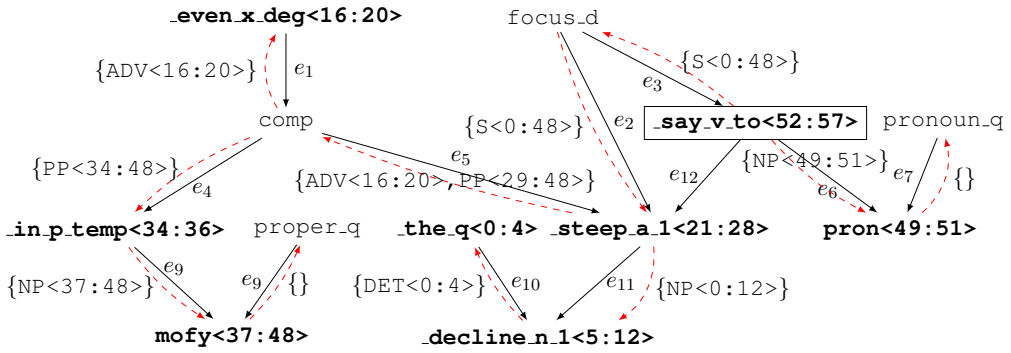

Figure 4: An example graph. The intended reading is *"the decline is even steeper than in September", he said.* Original edge labels are removed for clarity. Every edge is associated with a span list, and spans are written in the form `label<begin:end>`. The red dashed edges belong to the intermediate graph T.

variable-free MRS representations, namely EDS (Oepen and Lønning, 2006). The data set used in this work is DeepBank 1.1 (Flickinger et al., 2012).

5.1 EDS-specific Constraints

In order to generate reasonable strings, three constraints must be kept during transduction. First, for a rule $I \xrightarrow{\sigma} \langle O, E \rangle$, a state with direction `u` in I or a state with direction `r` in O is called *head state* and its variables are called *head variables*. For example, the head state of rule 3 in Table 1 is `VP(1,u)` and the head state of rule 2 is `DET(1,r)`. There is at most one head state in a rule and only head variables or S can be the left sides of statements. If there is no head state, we assign the global S as its head. Otherwise, the number of statements is equal to the number of head variables and each statement has a distinguished left side variable. An empty state does not have any variables. Second, every rule has no-copying, no-deleting statements. In other words, all variables must be used exactly once in a statement. Third, during recognition, a labeling function ρ is valid only if $T(\rho)$ is a rooted tree.

After transduction, we get result ρ^*. The first and second constraints ensure that for all nodes, there is at most one incoming red dashed edge in $T(\rho^*)$ and 'data' carried by variables of the only incoming red dashed edge or S is separated into variables of outgoing red dashed edges. The last constraint ensures that we can solve all statements by a bottom-up process on tree $T(\rho^*)$.

5.2 Fine-to-Coarse Transduction

Almost all NLG systems that heavily utilize a symbolic system to encode deep syntactico-semantic information lack some robustness, meaning that some input graphs may not be successfully processed. There are two reasons: (1) some explicit linguistic constraints are not included; (2) exact decoding is too time-consuming while inexact decoding cannot cover the whole search space. To solve the robustness problem, we introduce a fine-to-coarse strategy to ensure that at least one sentence is generated for any input graph. There are three types of rules in our system, namely *induced rules*, *extended rules* and *dynamic rules*. The most fine-grained rules are applied to bring us precision, while the most coarse-grained rules are for robustness.

In order to extract reasonable rules, we will use both EDS graphs and the corresponding derivation trees provided by ERG. The details will be described step-by-step in the following sections.

5.3 Induced Rules

Figure 4 shows an example for obtaining induced rules. The *induced rules* are directly obtained by following three steps:

Finding intermediate tree T EDS graphs are highly regular semantic graphs. It is not difficult to generate T based on a highly customized 'breadth-first' search. The generation starts from the 'top' node (`_say_v_to` in Figure 4) given by the EDS graph and traverse the whole graph. No more than thirty heuristic rules are used to decide the visiting order of nodes.

Assigning states EDS graphs also provide span information for nodes. We select a group of *lexical* nodes which have corresponding substrings in the original sentence. In Figure 4, these nodes are in bold font and directly followed by a span. Then we merge spans from the bottom of T to the top to assign each red edge a span list. For each node n in T, we collect spans of every outgoing dashed edge of n into a list s. Some additional spans may be inserted into s. These spans do not occur in the EDS graph but they do occur in the sentence, i.e. `than<29:33>`. Then we merge continuous spans in s and assign the remaining spans in s to the incoming red dashed edge of n. We apply a similar method to the derivation tree. As a result, every inner node of the derivation tree is associated with a span. Then we align the edges in T to nodes of the inner derivation tree by comparing their spans. Finally edge labels in Figure 4 are generated.

We use the concatenation of the edge labels in a span list as the state label. The edge labels are joined in order with '_'. `Empty(0,e)` is the state of the edges that do not belong to T (ignoring direction), such as e_{12}. The variable count of a state is equal to the size of the span list and the direction of a state is decided by whether the edge in T related to the state and its corresponding edge in D have different directions. For example, the state of e_5 should be `ADV_PP(2,r)`.

Generating statements After the above two steps, we are ready to generate statements according to how spans are merged. For all nodes, spans of the incoming edges represent the left hand side and the outgoing edges represent the right hand side. For example, the rule for node `comp` will be:

$$\{\texttt{ADV(1,r)}\} \xrightarrow{\texttt{comp}} \{\texttt{PP(1,u)}, \texttt{ADV\_PP(2,r)}\}$$
$$v_{\texttt{ADV\_PP(1,r)}} = v_{\texttt{ADV(1,r)}}$$
$$v_{\texttt{ADV\_PP(2,r)}} = \texttt{than} + v_{\texttt{PP(1,u)}}$$

5.4 Extended Rules

Extended rules are used when no induced rules can cover a given node. In theory, there can be unlimited modifier nodes pointing to a given node, such as `PP` and `ADJ`. We use some manually written rules to slightly change an induced rule (prototype) by addition or deletion to generate a group of extended rules. The motivation here is to deal with the data sparseness problem.

For a group of selected non-head states in I, such as `PP` and `ADJ`. We can produce new rules by removing or duplicating more of them. For example:

$$\{\texttt{NP(1,u)}, \texttt{ADJ(1,r)}\} \xrightarrow{\texttt{\_X\_n\_1}} \{\}$$
$$v_{\texttt{NP(1,u)}} = v_{\texttt{ADJ(1,r)}} + L$$

As a result, we get the two rules below:

$$\{\texttt{NP(1,u)}\} \xrightarrow{\texttt{\_X\_n\_1}} \{\} \quad v_{\texttt{NP(1,u)}} = L$$

$$\{\texttt{NP(1,u)}, \texttt{ADJ(1,r)}_1, \texttt{ADJ(1,r)}_2\} \xrightarrow{\texttt{\_X\_n\_1}} \{\}$$
$$v_{\texttt{NP(1,u)}} = v_{\texttt{ADJ(1,r)}_1} + v_{\texttt{ADJ(1,r)}_2} + L$$

5.5 Dynamic Rules

During decoding, when neither induced nor extended rule is applicable, we *create* a dynamic rule on-the-fly. Our rule creator builds a new rule following the Markov assumption:

$$P(O|C) = P(q_1|C) \prod_{i=2}^{n} P(q_i|C) P(q_i|q_{i-1}, C)$$

$C = \langle I, D \rangle$ represents the context.
$O = \{q_1, \cdots, q_n\}$ denotes the outgoing states and I, D have the same meaning as before. Though they are unordered multisets, we can give them an explicit alphabet order by their edge labels. There is also a group of hard constraints to make sure that the predicted rules are well-formed as the definition in §5 requires. This Markovization strategy is widely utilized by lexicalized and unlexicalized PCFG parsers (Collins, 1997; Klein and Manning, 2003). For a dynamic rule, all variables in this rule will appear in the statement. We use a simple perceptron-based scorer to assign every variable a score and arrange them in an decreasing order.

6 Evaluation and Analysis

6.1 Set-up

We use DeepBank 1.1 (Flickinger et al., 2012), i.e. gold-standard ERS annotations, as our main experimental data set to train a DAG transducer as well as a sequence-to-sequence morpholyzer, and wikiwoods (Flickinger et al., 2010), i.e. automatically-generated ERS annotations by ERG, as additional data set to enhance the sequence-to-sequence morpholyzer. The training,

development and test data sets are from DeepBank and split according to DeepBank's recommendation. There are 34,505, 1,758 and 1,444 sentences (all disconnected graphs as well as their associated sentences are removed) in the training, development and test data sets. We use a small portion of wikiwoods data, c.a. 300K sentences, for experiments.

37,537 induced rules are directly extracted from the training data set, and 447,602 extended rules are obtained. For DAG recognition, at one particular position, there may be more than one rule applicable. In this case, we need a disambiguation model as well as a decoder to search for a globally optimal solution. In this work, we train a structured perceptron model (Collins, 2002) for disambiguation and employ a beam decoder. The perceptron model used by our dynamic rule generator are trained with the induced rules. To get a sequence-to-sequence model, we use the open source tool—OpenNMT[4].

6.2 The Decoder

We implement a fine-to-coarse beam search decoder. Given a DAG D, our goal is to find the highest scored labeling function ρ:

$$\rho = \arg\max_{\rho} \prod_{i=1}^{n} \sum_{j} w_j \cdot f_j(rule(v_i), D)$$

$$\text{s.t.} \quad rule(v_i) = \rho(in(v_i)) \xrightarrow{\ell(v_i)} \langle \rho(out(v_i)), E_i \rangle$$

where n is the node count and $f_j(\cdot,\cdot)$ and w_j represent a feature and the corresponding weight, respectively. The features are chosen from the context of the given node v_i. We perform 'top-down' search to translate an input DAG into a morphology-function-enhanced lemma sequence. Each hypothesis consists of the current DAG graph, the partial labeling function, the current hypothesis score and other graph information used to perform rule selection. The decoder will keep the corresponding partial intermediate graph T acyclic when decoding. The algorithm used by our decoder is displayed in Algorithm 1. Function $\texttt{FindRules}(h, n, R)$ will use hard constraints to select rules from the rule set R according to the contextual information. It will also perform an acyclic check on T. Function $\texttt{Insert}(h, r, n, B)$ will create and score a new hypothesis made from the given context and then insert it into beam B.

[4] https://github.com/OpenNMT/OpenNMT/

After we get the edge labeling function ρ, we use a simple linear equation solver to convert all statements to a sequence of lemmas.

Algorithm 1: Algorithm for our decoder.

Input: D is the EDS graph. R_I and R_E are induced-rules and extended-rules respectively.
Result: The edge labeling function ρ.

$Q \leftarrow$ all the roots in D
$B1 \leftarrow$ empty beam
$E \leftarrow \varnothing$
Insert initial hypothesis into $B1$
while Q *is not empty*:
  $B2 \leftarrow$ empty beam
  $n \leftarrow$ dequeue a node from Q
  for $h \in B1$:
    $rules \leftarrow \texttt{FindRules}(h, n, R_I)$
    if $rules$ *is not empty*:
      for $r \in rules$:
        $\texttt{Insert}(h, r, n, B2)$
    else:
      $rules \leftarrow \texttt{FindRules}(h, n, R_E)$
      for $r \in rules$:
        $\texttt{Insert}(h, r, n, B2)$
  if $B2$ *is still empty*:
    for $h \in B1$:
      $r \leftarrow \texttt{RuleGenerator}(h, n)$
      $\texttt{Insert}(h, r, n, B2)$
  $B1 \leftarrow B2$
  for $e \in out(n)$:
    $E \leftarrow E \cup \{e\}$
    if $in(tar(e)) \subseteq E$:
      $Q \leftarrow Q \cup \{tar(e)\}$
Extract ρ from best hypothesis in $B1$

6.3 Accuracy

In order to evaluate the effectiveness of our transducer for NLG, we try a group of tests showed in Table 2. All sequence-to-sequence models (either from lemma sequences to lemma sequences or lemma sequences to sentences) are trained on DeepBank and wikiwoods data set and tuned on the development data. The second column shows the BLEU-4 scores between generated lemma sequences and golden sequences of lemmas. The third column shows the BLEU-4 scores between generated sentences and golden sentences. The fourth column shows the fraction of graphs in the test data set that can reach output sentences.

Transducer	Lemmas	Sentences	Coverage
I	89.44	74.94	67%
I+E	88.41	74.03	77%
I+E+D	82.04	68.07	100%
DFS-NN	50.45		100%
AMR-NN		33.8	100%
AMR-NRG		25.62	100%

Table 2: Accuracy (BLEU-4 score) and coverage of different systems. *I* denotes transduction only using induced rules; *I+E* denotes transduction using both induced and extended rules; *I+E+D* denotes transduction using all kinds of rules. *DFS-NN* is a rough implementation of Konstas et al. (2017) but with the EDS data, while *AMR-NN* includes the results originally reported by Konstas et al., which are evaluated on the AMR data. *AMR-NRG* includes the results obtained by a synchronous graph grammar (Song et al., 2017).

The graphs that cannot received any natural language sentences are removed while conducting the BLEU evaluation.

As we can conclude from Table 2, using only induced rules achieves the highest accuracy but the coverage is not satisfactory. Extended rules lead to a slight accuracy drop but with a great improvement of coverage (c.a. 10%). Using dynamic rules, we observe a significant accuracy drop. Nevertheless, we are able to handle all EDS graphs. The full-coverage robustness may benefit many NLP applications. The lemma sequences generated by our transducer are really close to the golden one. This means that our model actually works and most reordering patterns are handled well by induced rules.

Compared to the AMR generation task, our transducer on EDS graphs achieves much higher accuracies. To make clear how much improvement is from the data and how much is from our DAG transducer, we implement a purely neural baseline. The baseline converts a DAG into a concept sequence by a pre-order DFS traversal on the intermediate tree of this DAG. Then we use a sequence-to-sequence model to transform this concept sequence to the lemma sequence for comparison. This is a kind of implementation of Konstas et al.'s model but evaluated on the EDS data. We can see that on this task, our transducer is much better than a pure sequence-to-sequence model on DeepBank data.

Transducer	Average (s)	Maximal (s)
I	0.090	0.40
I+E	0.093	0.46
I+E+D	0.18	3.2

Table 3: Efficiency of our NL generator.

6.4 Efficiency

Table 3 shows the efficiency of the beam search decoder with a beam size of 128. The platform for this experiment is x86_64 GNU/Linux with two Intel Xeon E5-2620 CPUs. The second and third columns represent the average and the maximal time (in seconds) to translate an EDS graph. Using dynamic rules slow down the decoder to a great degree. Since the data for experiments is newswire data, i.e. WSJ sentences from PTB (Marcus et al., 1993), the input graphs are quite large on average. On average, it produces more than 5 sentences per second on CPU. We consider this a promising speed.

7 Conclusion

We extend the work on DAG automata in Chiang et al. (2018) and propose a general method to build flexible DAG transducer. The key idea is to leverage a declarative programming language to minimize the computation burden of a graph transducer. We think may NLP tasks that involve graph manipulation may benefit from this design. To exemplify our design, we develop a practical system for the semantic-graph-to-string task. Our system is accurate (BLEU 68.07), efficient (more than 5 sentences per second on a CPU) and robust (full-coverage). The empirical evaluation confirms the usefulness a DAG transducer to resolve NLG, as well as the effectiveness of our design.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (61772036, 61331011) and the Key Laboratory of Science, Technology and Standard in Press Industry (Key Laboratory of Intelligent Press Media Technology). We thank the anonymous reviewers for their helpful comments. Weiwei Sun is the corresponding author.

References

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

Bernd Bohnet and Leo Wanner. 2010. Open soucre graph transducer interpreter and grammar development environment. In *LREC*.

B. Carpenter. 1997. *Type-Logical Semantics*. Bradford books. MIT Press.

David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. 2018. Weighted DAG automata for semantic graphs. *Computational Linguistics*. To appear.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Madrid, Spain. Association for Computational Linguistics.

Michael Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 1–8. Association for Computational Linguistics.

Hubert Comon, Max Dauchet, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. 1997. Tree automata techniques and applications. Technical report.

Ann Copestake. 2009. ***Invited Talk:*** slacker semantics: Why superficiality, dependency and avoidance of commitment can be the right way to go. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 1–9, Athens, Greece. Association for Computational Linguistics.

H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

Dan Flickinger. 2000. On building a more efficient grammar by exploiting types. *Nat. Lang. Eng.*, 6(1):15–28.

Dan Flickinger, Stephan Oepen, and Gisle Ytrestl. 2010. Wikiwoods: Syntacto-semantic annotation for English wikipedia. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta. European Language Resources Association (ELRA).

Daniel Flickinger, Yi Zhang, and Valia Kordoni. 2012. Deepbank: A dynamically annotated treebank of the wall street journal. In *Proceedings of the Eleventh International Workshop on Treebanks and Linguistic Theories*, pages 85–96.

Tsutomu Kamimura and Giora Slutzki. 1982. Transductions of dags and trees. *Mathematical Systems Theory*, 15(3):225–249.

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

Ioannis Konstas, Srinivasan Iyer, Mark Yatskar, Yejin Choi, and Luke Zettlemoyer. 2017. Neural amr: Sequence-to-sequence models for parsing and generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 146–157, Vancouver, Canada. Association for Computational Linguistics.

Angelika Kratzer and Irene Heim. 1998. *Semantics in generative grammar*. Blackwell Oxford.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: the penn treebank. *Computational Linguistics*, 19(2):313–330.

Stephan Oepen and Jan Tore Lønning. 2006. Discriminant-based mrs banking. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC-2006)*, Genoa, Italy. European Language Resources Association (ELRA). ACL Anthology Identifier: L06-1214.

Daniel Quernheim and Kevin Knight. 2012. Towards probabilistic acceptors and transducers for feature structures. In *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation*, SSST-6 '12, pages 76–85, Stroudsburg, PA, USA. Association for Computational Linguistics.

Linfeng Song, Xiaochang Peng, Yue Zhang, Zhiguo Wang, and Daniel Gildea. 2017. Amr-to-text generation with synchronous node replacement grammar. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 7–13, Vancouver, Canada. Association for Computational Linguistics.