A Transition-based Algorithm for AMR Parsing

Chuan Wang
Brandeis University
cwang24@brandeis.edu

Nianwen Xue
Brandeis University
xuen@brandeis.edu

Sameer Pradhan
Harvard Medical School
Sameer.Pradhan@childrens.harvard.edu

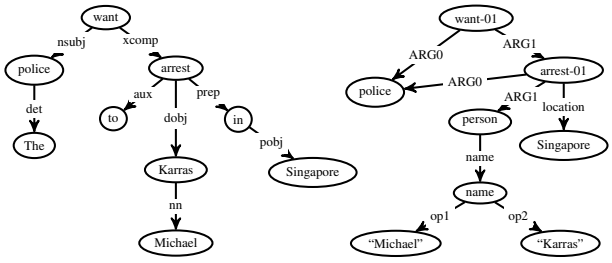
Abstract

We present a two-stage framework to parse a sentence into its Abstract Meaning Representation (AMR). We first use a dependency parser to generate a dependency tree for the sentence. In the second stage, we design a novel transition-based algorithm that transforms the dependency tree to an AMR graph. There are several advantages with this approach. First, the dependency parser can be trained on a training set much larger than the training set for the tree-to-graph algorithm, resulting in a more accurate AMR parser overall. Our parser yields an improvement of 5% absolute in F-measure over the best previous result. Second, the actions that we design are linguistically intuitive and capture the regularities in the mapping between the dependency structure and the AMR of a sentence. Third, our parser runs in nearly linear time in practice in spite of a worst-case complexity of $O(n^2)$.

1 Introduction

Abstract Meaning Representation (AMR) is a rooted, directed, edge-labeled and leaf-labeled graph that is used to represent the meaning of a sentence. The AMR formalism has been used to annotate the AMR Annotation Corpus (Banarescu et al., 2013), a corpus of over 10 thousand sentences that is still undergoing expansion. The building blocks for an AMR representation are concepts and relations between them. Understanding these concepts and their relations is crucial to understanding the meaning of a sentence and could potentially benefit a number of natural language applications such as Information Extraction, Question Answering and Machine Translation.

The property that makes AMR a graph instead of a tree is that AMR allows reentrancy, meaning that the same concept can participate in multiple relations. Parsing a sentence into an AMR would seem to require graph-based algorithms, but moving to graph-based algorithms from the typical tree-based algorithms that we are familiar with is a big step in terms of computational complexity. Indeed, quite a bit of effort has gone into developing grammars and efficient graph-based algorithms that can be used to parse AMRs (Chiang et al., 2013).



(a) Dependency tree (b) AMR graph

Figure 1: Dependency tree and AMR graph for the sentence, "The police want to arrest Micheal Karras in Singapore."

Linguistically, however, there are many similarities between an AMR and the dependency structure of a sentence. Both describe relations as holding between a head and its dependent, or between a parent and its child. AMR concepts and relations abstract away from actual word tokens, but there are regularities in their mappings. Content words generally be-

Human Language Technologies: The 2015 Annual Conference of the North American Chapter of the ACL, pages 366–375,
Denver, Colorado, May 31 – June 5, 2015.

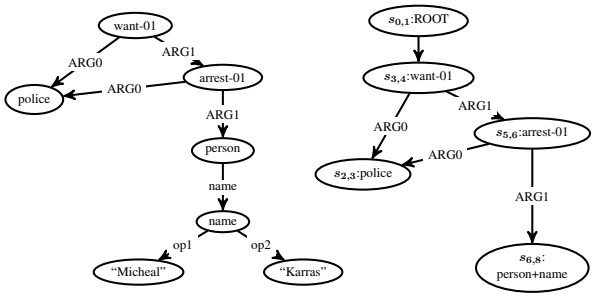
come concepts while function words either become relations or get omitted if they do not contribute to the meaning of a sentence. This is illustrated in Figure 1, where 'the' and 'to' in the dependency tree are omitted from the AMR and the preposition 'in' becomes a relation of type *location*. In AMR, reentrancy is also used to represent co-reference, but this only happens in some limited contexts. In Figure 1, 'police' is both an argument of 'arrest' and 'want' as the result of a control structure. This suggests that it is possible to transform a dependency tree into an AMR with a limited number of actions and learn a model to determine which action to take given pairs of aligned dependency trees and AMRs as training data.

This is the approach we adopt in the present work, and we present a transition-based framework in which we parse a sentence into an AMR by taking the dependency tree of that sentence as input and transforming it to an AMR representation via a series of actions. This means that a sentence is parsed into an AMR in two steps. In the first step the sentence is parsed into a dependency tree with a dependency parser, and in the second step the dependency tree is transformed into an AMR graph. One advantage of this approach is that the dependency parser does not have to be trained on the same data set as the dependency to AMR transducer. This allows us to use more accurate dependency parsers trained on data sets much larger than the AMR Annotation Corpus and have a more advantageous starting point. Our experiments show that this approach is very effective and yields an improvement of 5% absolute over the previously reported best result (Flanigan et al., 2014) in F-score, as measure by the Smatch metric (Cai and Knight, 2013).

The rest of the paper is as follows. In §2, we describe how we align the word tokens in a sentence with its AMR to create a span graph based on which we extract contextual information as features and perform actions. In §3, we present our transition-based parsing algorithm and describe the actions used to transform the dependency tree of a sentence into an AMR. In §4, we present the learning algorithm and the features we extract to train the transition model. In §5, we present experimental results. §6 describes related work, and we conclude in §7.

2 Graph Representation

Unlike the dependency structure of a sentence where each word token is a node in the dependency tree and there is an inherent alignment between the word tokens in the sentence and the nodes in the dependency tree, AMR is an abstract representation where the word order of the corresponding sentence is not maintained. In addition, some words become abstract concepts or relations while other words are simply deleted because they do not contribute to meaning. The alignment between the word tokens and the concepts is non-trivial, but in order to learn the transition from a dependency tree to an AMR graph, we have to first establish the alignment between the word tokens in the sentence and the concepts in the AMR. We use the aligner that comes with JAMR (Flanigan et al., 2014) to produce this alignment. The JAMR aligner attempts to greedily align every concept or graph fragment in the AMR graph with a contiguous word token sequence in the sentence.



(a) AMR graph (b) Span graph

Figure 2: AMR graph and its span graph for the sentence, "The police want to arrest Micheal Karras."

We use a data structure called **span graph** to represent an AMR graph that is aligned with the word tokens in a sentence. For each sentence $w = w_0, w_1, \ldots, w_n$, where token w_0 is a special root symbol, a **span graph** is a directed, labeled graph $G = (V, A)$, where $V = \{s_{i,j} | i, j \in (0, n) \text{ and } j > i\}$ is a set of nodes, and $A \subseteq V \times V$ is a set of arcs. Each node $s_{i,j}$ of G corresponds to a continuous span $(w_i, \ldots, w_{j-1})$ in sentence w and is indexed by the starting position i. Each node is assigned a **concept** label from a set L_V of concept labels and each arc is assigned a **relation** label from a set L_A

of relation labels, respectively.

For example, given an AMR graph G_{AMR} in Figure 2a, its span graph G can be represented as Figure 2b. In span graph G, node $s_{3,4}$'s sentence span is *(want)* and its concept label is *want-01*, which represents a single node *want-01* in AMR. To simplify the alignment, when creating a span graph out of an AMR, we also collapse some AMR subgraphs in such a way that they can be deterministically restored to their original state for evaluation. For example, the four nodes in the AMR subgraph that correspond to span *(Micheal, Karras)* is collapsed into a single node $s_{6,8}$ in the span graph and assigned the concept label *person+name*, as shown in Figure 3. So the concept label set that our model predicts consists of both those from the concepts in the original AMR graph and those as a result of collapsing the AMR subgraphs.

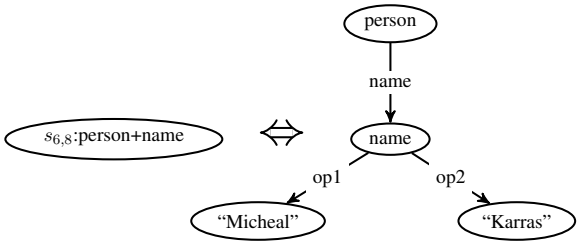


Figure 3: Collapsed nodes

Representing AMR graph this way allows us to formulate the AMR parsing problem as a joint learning problem where we can design a set of actions to simultaneously predict the concepts (nodes) and relations (arcs) in the AMR graph as well as the labels on them.

3 Transition-based AMR Parsing

3.1 Transition System

Similar to transition-based dependency parsing (Nivre, 2008), we define a *transition system* for AMR parsing as a quadruple $S = (S, T, s_0, S_t)$, where

- S is a set of parsing *states* (configurations).
- T is a set of parsing *actions* (transitions), each of which is a function $t : S \rightarrow S$.
- s_0 is an *initialization function*, mapping each input sentence w and its dependency tree D to an *initial state*.
- $S_t \subseteq S$ is a set of *terminal states*.

Each state (configuration) of our transition-based parser is a triple (σ, β, G). σ is a buffer that stores indices of the nodes which have not been processed and we write $\sigma = \sigma_0|\sigma'$ to indicate that σ_0 is the topmost element of σ. β is also a buffer $[\beta_0, \beta_1, \ldots, \beta_j]$ and each element β_i of β indicates the edge (σ_0, β_i) which has not been processed in the partial graph. We also write $\beta = \beta_0|\beta'$ to indicate the topmost element of β is β_0. We use span graph G to store the partial parses for the input sentence w. Note that unlike traditional transition-based syntactic parsers which store partial parses in the stack structure and build a tree or graph incrementally, here we use σ and β buffers only to guide the parsing process (which node or edge to be processed next) and the actual tree-to-graph transformations are applied to G.

When the parsing procedure starts, σ is initialized with a post-order traversal of the input dependency tree D with topmost element σ_0, β is initialized with node σ_0's children or set to null if σ_0 is a leaf node. G is initialized with all the nodes and edges of D. Initially, all the nodes of G have a span length of one and all the labels for nodes and edges are set to null. As the parsing procedure goes on, the parser will process all the nodes and their outgoing edges in dependency tree D in a bottom-up left-right manner, and at each state certain action will be applied to the current node or edge. The parsing process will terminate when both σ and β are empty.

The most important part of the transition-based parser is the set of actions (transitions). As stated in (Sartorio et al., 2013), the design space of possible actions is actually infinite since the set of parsing states is infinite. However, if the problem is amenable to transition-based parsing, we can design a finite set of actions by categorizing all the possible situations we run into in the parsing process. In §5.2 we show this is the case here and our action set can account for almost all the transformations from dependency trees to AMR graphs.

We define 8 types of actions for the actions set T, which is summarized in Table 1. The action set could be divided into two categories based on conditions of buffer β. When β is not empty, parsing decisions are made based on the edge (σ_0, β_0); oth-

Action	Current state ⇒ Result state	Assign labels	Precondition
NEXT EDGE-l_r	$(\sigma_0\|\sigma', \beta_0\|\beta', G) \Rightarrow (\sigma_0\|\sigma', \beta', G')$	$\delta[(\sigma_0, \beta_0) \rightarrow l_r]$	β is not empty
SWAP-l_r	$(\sigma_0\|\sigma', \beta_0\|\beta', G) \Rightarrow (\sigma_0\|\beta_0\|\sigma', \beta', G')$	$\delta[(\beta_0, \sigma_0) \rightarrow l_r]$	
REATTACH$_k$-l_r	$(\sigma_0\|\sigma', \beta_0\|\beta', G) \Rightarrow (\sigma_0\|\sigma', \beta', G')$	$\delta[(k, \beta_0) \rightarrow l_r]$	
REPLACE HEAD	$(\sigma_0\|\sigma', \beta_0\|\beta', G) \Rightarrow (\beta_0\|\sigma', \beta = CH(\beta_0, G'), G')$	NONE	
REENTRANCE$_k$-l_r	$(\sigma_0\|\sigma', \beta_0\|\beta', G) \Rightarrow (\sigma_0\|\sigma', \beta_0\|\beta', G')$	$\delta[(k, \beta_0) \rightarrow l_r]$	
MERGE	$(\sigma_0\|\sigma', \beta_0\|\beta', G) \Rightarrow (\tilde{\sigma}\|\sigma', \beta', G')$	NONE	
NEXT NODE-l_c	$(\sigma_0\|\sigma_1\|\sigma', [], G) \Rightarrow (\sigma_1\|\sigma', \beta = CH(\sigma_1, G'), G')$	$\gamma[\sigma_0 \rightarrow l_c]$	β is empty
DELETE NODE	$(\sigma_0\|\sigma_1\|\sigma', [], G) \Rightarrow (\sigma_1\|\sigma', \beta = CH(\sigma_1, G'), G')$	NONE	

Table 1: Transitions designed in our parser. $CH(x, y)$ means getting all node x's children in graph y.

erwise, only the current node σ_0 is examined. Also, to simultaneously make decisions on the assignment of concept/relation label, we augment some of the actions with an extra parameter l_r or l_c. We define $\gamma : V \rightarrow L_V$ as the concept labeling function for nodes and $\delta : A \rightarrow L_A$ as the relation labeling function for arcs. So $\delta[(\sigma_0, \beta_0) \rightarrow l_r]$ means assigning relation label l_r to arc (σ_0, β_0). All the actions update buffer σ, β and apply some transformation $G \Rightarrow G'$ to the partial graph. The 8 actions are described below.

- NEXT-EDGE-l_r (ned). This action assigns a relation label l_r to the current edge (σ_0, β_0) and makes no further modification to the partial graph. Then it pops out the top element of buffer β so that the parser moves one step forward to examine the next edge if it exists.

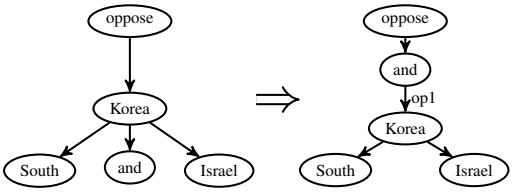


Figure 4: SWAP action

- SWAP-l_r (sw). This action reverses the dependency relation between node σ_0 and β_0 and then makes node β_0 as new head of the subgraph. Also it assigns relation label l_r to the arc (β_0, σ_0). Then it pops out β_0 and inserts it into σ right after σ_0 for future revisiting. This action is to resolve the difference in the choice of head between the dependency tree and the AMR graph. Figure 4 gives an example of applying SWAP-op1 action for arc (*Korea*, *and*) in the dependency tree of sentence "South Korea and Israel oppose ...".
- REATTACH$_k$-l_r (reat). This action removes the current arc (σ_0, β_0) and reattaches node β_0 to some node k in the partial graph. It also assigns a relation label l_r to the newly created arc (k, β_0) and advances one step by popping out β_0. Theoretically, the choice of node k could be any node in the partial graph under the constraint that arc (k, β_0) doesn't produce a self-looping cycle. The intuition behind this action is that after swapping a head and its dependent, some of the dependents of the old head should be reattached to the new head. Figure 5 shows an example where node *Israel* needs to be reattached to node *and* after a head-dependent swap.

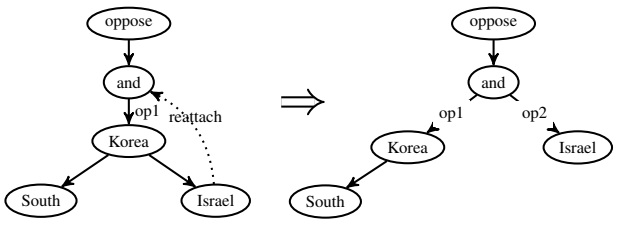


Figure 5: REATTACH action

- REPLACE-HEAD (rph). This action removes node σ_0, replaces it with node β_0. Node β_0 also inherits all the incoming and outgoing arcs of σ_0. Then it pops out β_0 and inserts it into the top position of buffer σ. β is re-initialized with all the children of β_0 in the transformed graph G'. This action targets nodes in the dependency tree that do not correspond to concepts in AMR

graph and become a relation instead. An example is provided in Figure 6, where node *in*, a preposition, is replaced with node *Singapore*, and in a subsequent NEXT-EDGE action that examines arc (*live*, *Singapore*), the arc is labeled *location*.

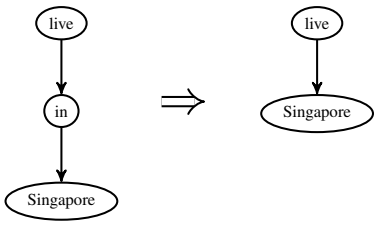


Figure 6: REPLACE-HEAD action

- REENTRANCE$_k$-l_r (reen). This is the action that transforms a tree into a graph. It keeps the current arc unchanged, and links node β_0 to every possible node k in the partial graph that can also be its parent. Similar to the REATTACH action, the newly created arc (k, β_0) should not produce a self-looping cycle and parameter k is bounded by the sentence length. In practice, we seek to constrain this action as we will explain in §3.2. Intuitively, this action can be used to model co-reference and an example is given in Figure 7.

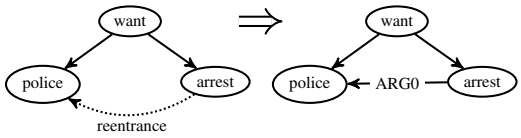


Figure 7: REENTRANCE action

- MERGE (mrg). This action merges nodes σ_0 and β_0 into one node $\tilde{\sigma}$ which covers multiple words in the sentence. The new node inherits all the incoming and outgoing arcs of both nodes σ_0 and β_0. The MERGE action is intended to produce nodes that cover a continuous span in the sentence that corresponds to a single name entity in AMR graph. see Figure 8 for an example.

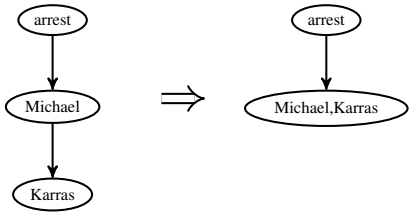


Figure 8: MERGE action

When β is empty, which means all the outgoing arcs of node σ_0 have been processed or σ_0 has no outgoing arcs, the following two actions can be applied:

- NEXT-NODE-l_c (nnd). This action first assigns a concept label l_c to node σ_0. Then it advances the parsing procedure by popping out the top element σ_0 of buffer σ and re-initializes buffer β with all the children of node σ_1 which is the current top element of σ. Since this action will be applied to every node which is kept in the final parsed graph, concept labeling could be done simultaneously through this action.
- DELETE-NODE (dnd). This action simply deletes the node σ_0 and removes all the arcs associated with it. This action models the fact that most function words are stripped off in the AMR of a sentence. Note that this action only targets function words that are leaves in the dependency tree, and we constrain this action by only deleting nodes which do not have outgoing arcs.

When parsing a sentence of length n (excluding the special root symbol w_0), its corresponding dependency tree will have n nodes and $n-1$ arcs. For projective transition-based dependency parsing, the parser needs to take exactly $2n-1$ steps or actions. So the complexity is $O(n)$. However, for our tree-to-graph parser defined above, the actions needed are no longer linearly bounded by the sentence length. Suppose there are no REATTACH, REENTRANCE and SWAP actions during the parsing process, the algorithm will traverse every node and edge in the dependency tree, which results in $2n$ actions. However, REATTACH and REENTRANCE actions would add extra edges that need to be re-processed and the SWAP action adds both nodes and edges that need to be re-visited. Since the

space of all possible extra edges is $(n-2)^2$ and revisiting them only adds more actions linearly, the total asymptotic runtime complexity of our algorithm is $O(n^2)$.

In practice, however, the number of applications of the REATTACH action is much less than the worst case scenario due to the similarities between the dependency tree and the AMR graph of a sentence. Also, nodes with reentrancies in AMR only account for a small fraction of all the nodes, thus making the REENTRANCE action occur at constant times. These allow the tree-to-graph parser to parse a sentence in nearly linear time in practice.

3.2 Greedy Parsing Algorithm

Algorithm 1 Parsing algorithm

Input: sentence $w = w_0 \dots w_n$ and its dependency tree D_w
Output: parsed graph G_p

1: $s \leftarrow s_0(D_w, w)$
2: **while** $s \notin S_t$ **do**
3: $\quad \mathcal{T} \leftarrow$ all possible actions according to s
4: $\quad bestT \leftarrow \arg\max_{t \in \mathcal{T}} score(t, c)$
5: $\quad s \leftarrow$ apply $bestT$ to s
6: **end while**
7: **return** G_p

Our parsing algorithm is similar to the parser in (Sartorio et al., 2013). At each parsing state $s \in S$, the algorithm greedily chooses the parsing action $t \in T$ that maximizes the score function $score()$. The score function is a linear model defined over parsing action t and parsing state s.

$$score(t, s) = \vec{\omega} \cdot \phi(t, s) \quad (1)$$

where $\vec{\omega}$ is the weight vector and ϕ is a function that extracts the feature vector representation for one possible state-action pair $\langle t, s \rangle$.

First, the algorithm initializes the state s with the sentence w and its dependency tree D_w. At each iteration, it gets all the possible actions for current state s (line 3). Then, it chooses the action with the highest score given by function $score()$ and applies it to s (line 4-5). When the current state reaches a terminal state, the parser stops and returns the parsed graph.

As pointed out in (Bohnet and Nivre, 2012), constraints can be added to limit the number of possible actions to be evaluated at line 3. There could be formal constraints on states such as the constraint that the SWAP action should not be applied twice to the same pair of nodes. We could also apply soft constraints to filter out unlikely concept labels, relation labels and candidate nodes k for REATTACH and REENTRANCE. In our parser, we enforce the constraint that NEXT-NODE-l_c can only choose from concept labels that co-occur with the current node's lemma in the training data. We also empirically set the constraint that REATTACH$_k$ could only choose k among σ_0's grandparents and great grandparents. Additionally, REENTRANCE$_k$ could only choose k among its siblings. These constraints greatly reduce the search space, thus speeding up the parser.

4 Learning

4.1 Learning Algorithm

As stated in section 3.2, the parameter of our model is weight vector $\vec{\omega}$ in the score function. To train the weight vector, we employ the averaged perceptron learning algorithm (Collins, 2002).

Algorithm 2 Learning algorithm

Input: sentence $w = w_0 \dots w_n, D_w, G_w$
Output: $\vec{\omega}$

$s \leftarrow s_0(D_w, w)$
while $s \notin S_t$ **do**
$\quad \mathcal{T} \leftarrow$ all possible actions according to s
$\quad bestT \leftarrow \arg\max_{t \in \mathcal{T}} score(t, s)$
$\quad goldT \leftarrow oracle(s, G_w)$
$\quad$ **if** $bestT \neq goldT$ **then**
$\quad\quad \vec{\omega} \leftarrow \vec{\omega} - \phi(bestT, s) + \phi(goldT, s)$
$\quad$ **end if**
$\quad s \leftarrow$ apply $goldT$ to s
end while

For each sentence w and its corresponding AMR annotation G_{AMR} in the training corpus, we could get the dependency tree D_w of w with a dependency parser. Then we represent G_{AMR} as span graph G_w, which serves as our learning target. The learning algorithm takes the training instances (w, D_w, G_w), parses D_w according to Algorithm 1, and get the best action using current weight vector $\vec{\omega}$. The

gold action for current state s is given by consulting span graph G_w, which we formulate as a function $oracle()$ (line 5). If the gold action is equal to the best action we get from the parser, then the best action is applied to current state; otherwise, we update the weight vector (line 6-7) and continue the parsing procedure by applying the gold action.

4.2 Feature Extraction

Single node features
$\bar{\sigma}_0.w, \bar{\sigma}_0.lem, \bar{\sigma}_0.ne, \bar{\sigma}_0.t, \bar{\sigma}_0.dl, \bar{\sigma}_0.len$
$\bar{\beta}_0.w, \bar{\beta}_0.lem, \bar{\beta}_0.ne, \bar{\beta}_0.t, \bar{\beta}_0.dl, \bar{\beta}_0.len$
$\bar{k}.w, \bar{k}.lem, \bar{k}.ne, \bar{k}.t, \bar{k}.dl, \bar{k}.len$
$\bar{\sigma}_{0p}.w, \bar{\sigma}_{0p}.lem, \bar{\sigma}_{0p}.ne, \bar{\sigma}_{0p}.t, \bar{\sigma}_{0p}.dl$
Node pair features
$\bar{\sigma}_0.lem + \bar{\beta}_0.t, \bar{\sigma}_0.lem + \bar{\beta}_0.dl$
$\bar{\sigma}_0.t + \bar{\beta}_0.lem, \bar{\sigma}_0.dl + \bar{\beta}_0.lem$
$\bar{\sigma}_0.ne + \bar{\beta}_0.ne, \bar{k}.ne + \bar{\beta}_0.ne$
$\bar{k}.t + \bar{\beta}_0.lem, \bar{k}.dl + \bar{\beta}_0.lem$
Path features
$\bar{\sigma}_0.lem + \bar{\beta}_0.lem + path_{\sigma_0,\beta_0}$
$\bar{k}.lem + \bar{\beta}_0.lem + path_{k,\beta_0}$
Distance features
$dist_{\sigma_0,\beta_0}$
$dist_{k,\beta_0}$
$dist_{\sigma_0,\beta_0} + path_{\sigma_0,\beta_0}$
$dist_{\sigma_0,\beta_0} + path_{k,\beta_0}$
Action specific features
$\bar{\beta}_0.lem + \bar{\beta}_0.nswp$
$\bar{\beta}_0.reph$

Table 2: Features used in our parser. $\bar{\sigma}_0, \bar{\beta}_0, \bar{k}, \bar{\sigma}_{0p}$ represents elements in feature context of nodes $\sigma_0, \beta_0, k, \sigma_{0p}$, separately. Each atomic feature is represented as follows: w - word; lem - lemma; ne - name entity; t - POS-tag; dl - dependency label; len - length of the node's span.

For transition-based dependency parsers, the feature context for a parsing state is represented by the neighboring elements of a word token in the stack containing the partial parse or the buffer containing unprocessed word tokens. In contrast, in our tree-to graph parser, as already stated, buffers σ and β only specify which arc or node is to be examined next. The feature context associated with current arc or node is mainly extracted from the partial graph G. As a result, the feature context is different for the different types of actions, a property that makes our parser very different from a standard transition-based dependency parser. For example, when evaluating action SWAP we may be interested in features about individual nodes σ_0 and β_0 as well as features involving the arc (σ_0, β_0). In contrast, when evaluating action REATTACH$_k$, we want to extract not only features involving σ_0 and β_0, but also information about the reattached node k. To address this problem, we define the feature context as $\langle \bar{\sigma}_0, \bar{\beta}_0, \bar{k}, \bar{\sigma}_{0p} \rangle$, where each element $\bar{x}$ consists of its atomic features of node x and σ_{0p} denotes the immediate parent of node σ_0. For elements in feature context that are not applicable to the candidate action, we just set the element to NONE and only extract features which are valid for the candidate action. The list of features we use is shown in Table 2.

Single node features are atomic features concerning all the possible nodes involved in each candidate state-action pair. We also include path features and distance features as described in (Flanigan et al., 2014). A path feature $path_{x,y}$ is represented as the dependency labels and parts of speech on the path between nodes x and y in the partial graph. Here we combine it with the lemma of the starting and ending nodes. Distance feature $dist_{x,y}$ is the number of tokens between two node x, y's spans in the sentence. Action-specific features record the history of actions applied to a given node. For example, $\bar{\beta}_0.nswp$ records how many times node β_0 has been swapped up. We combine this feature with the lemma of node β_0 to prevent the parser from swapping a node too many times. $\bar{\beta}_0.reph$ records the word feature of nodes that have been replaced with node β_0. This feature is helpful in predicting relation labels. As we have discussed above, in an AMR graph, some function words are deleted as nodes but they are crucial in determining the relation label between its child and parent.

5 Experiments

5.1 Experiment Setting

Our experiments are conducted on the newswire section of AMR Annotation Corpus (LDC2013E117) (Banarescu et al., 2013).

We follow Flanigan et al. (2014) in setting up the train/development/test splits[1] for easy comparison: 4.0k sentences with document years 1995-2006 as the training set; 2.1k sentences with document year 2007 as the development set; 2.1k sentences with document year 2008 as the test set, and only using AMRs that are tagged `::preferred`. Each sentence w is preprocessed with the Stanford CoreNLP toolkit (Manning et al., 2014) to get part-of-speech tags, name entity information, and basic dependencies. We have verified that there is no overlap between the training data for the Stanford CoreNLP toolkit[2] and the AMR Annotation Corpus. We evaluate our parser with the Smatch tool (Cai and Knight, 2013), which seeks to maximize the semantic overlap between two AMR annotations.

5.2 Action Set Validation

One question about the transition system we presented above is whether the action set defined here can cover all the situations involving a dependency-to-AMR transformation. Although a formal theoretical proof is beyond the scope of this paper, we can empirically verify that the action set works well in practice. To validate the actions, we first run the $oracle()$ function for each sentence w and its dependency tree D_w to get the "pseudo-gold" G'_w. Then we compare G'_w with the gold-standard AMR graph represented as span graph G_w to see how similar they are. On the training data we got an overall 99% F-score for all $\langle G'_w, G_w \rangle$ pairs, which indicates that our action set is capable of transforming each sentence w and its dependency tree D_w into its gold-standard AMR graph through a sequence of actions.

5.3 Results

Table 3 gives the precision, recall and F-score of our parser given by Smatch on the test set. Our parser achieves an F-score of 63% (Row 3) and the result is 5% better than the first published result reported in (Flanigan et al., 2014) with the same training and test set (Row 2). We also conducted experiments on the test set by replacing the parsed graph with gold relation labels or/and gold concept labels. We can see in Table 3 that when provided with gold concept and relation labels as input, the parsing accuracy improves around 8% F-score (Row 6). Rows 4 and 5 present results when the parser is provided with just the gold relation labels (Row 4) or gold concept labels (Row 5), and the results are expectedly lower than if both gold concept and relation labels are provided as input.

	Precision	Recall	F-score
JAMR	.52	.66	.58
Our parser	**.64**	.62	**.63**
Our parser +l_{gr}	.68	.65	.67
Our parser +l_{gc}	.69	.67	.68
Our parser +l_{grc}	.72	.70	.71

Table 3: Results on the test set. Here, l_{gc} - gold concept label; l_{gr} - gold relation label; l_{grc} - gold concept label and gold relation label.

5.4 Error Analysis

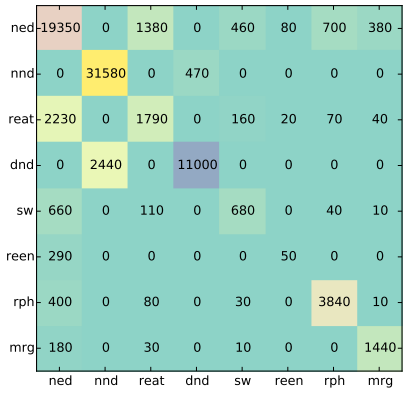


Figure 9: Confusion Matrix for actions $\langle t_g, t \rangle$. Vertical direction goes over the correct action type, and horizontal direction goes over the parsed action type.

Wrong alignments between the word tokens in the sentence and the concepts in the AMR graph account for a significant proportion of our AMR parsing errors, but here we focus on errors in the transition from the dependency tree to the AMR graph. Since in our parsing model, the parsing process has been decomposed into a sequence of actions applied to the input dependency tree, we can use the $oracle()$ function during parsing to give us the cor-

[1] A script to create the train/dev/test partitions is available at the following URL: `http://goo.gl/vA32iI`

[2] Specifically we used CoreNLP toolkit v3.3.1 and parser model wsjPCFG.ser.gz trained on the WSJ treebank sections 02-21.

rect action t_g to take for a given state s. A comparison between t_g and the best action t actually taken by our parser will give us a sense about how accurately each type of action is applied. When we compare the actions, we focus on the structural aspect of AMR parsing and only take into account the eight action types, ignoring the concept and edge labels attached to them. For example, NEXT-EDGE-$ARG0$ and NEXT-EDGE-$ARG1$ would be considered to be the same action and counted as a match when we compute the errors even though the labels attached to them are different.

Figure 9 shows the confusion matrix that presents a comparison between the parser-predicted actions and the correct actions given by $oracle()$ function. It shows that the NEXT-EDGE (ned), NEXT-NODE (nnd), and DELETENODE (dnd) actions account for a large proportion of the actions. These actions are also more accurately applied. As expected, the parser makes more mistakes involving the REATTACH (reat), REENTRANCE (reen) and SWAP (sw) actions. The REATTACH action is often used to correct PP-attachment errors made by the dependency parser or readjust the structure resulting from the SWAP action, and it is hard to learn given the relatively small AMR training set. The SWAP action is often tied to coordination structures in which the head in the dependency structure and the AMR graph diverges. In the Stanford dependency representation which is the input to our parser, the head of a coordination structure is one of the conjuncts. For AMR, the head is an abstract concept signaled by one of the coordinating conjunctions. This also turns out to be one of the more difficult actions to learn. We expect, however, as the AMR Annotation Corpus grows bigger, the parsing model trained on a larger training set will learn these actions better.

6 Related Work

Our work is directly comparable to JAMR (Flanigan et al., 2014), the first published AMR parser. JAMR performs AMR parsing in two stages: concept identification and relation identification. They treat concept identification as a sequence labeling task and utilize a semi-Markov model to map spans of words in a sentence to concept graph fragments. For relation identification, they adopt the graph-based techniques for non-projective dependency parsing. Instead of finding maximum-scoring trees over words, they propose an algorithm to find the maximum spanning connected subgraph (MSCG) over concept fragments obtained from the first stage. In contrast, we adopt a transition-based approach that finds its root in transition-based dependency parsing (Yamada and Matsumoto, 2003; Nivre, 2003; Sagae and Tsujii, 2008), where a series of actions are performed to transform a sentence to a dependency tree. As should be clear from our description, however, the actions in our parser are very different in nature from the actions used in transition-based dependency parsing.

There is also another line of research that attempts to design graph grammars such as hyperedge replacement grammar (HRG) (Chiang et al., 2013) and efficient graph-based algorithms for AMR parsing. Existing work along this line is still theoretical in nature and no empirical results have been reported yet.

7 Conclusion and Future Work

We presented a novel transition-based parsing algorithm that takes the dependency tree of a sentence as input and transforms it into an Abstract Meaning Representation graph through a sequence of actions. We show that our approach is linguistically intuitive and our experimental results also show that our parser outperformed the previous best reported results by a significant margin. In future work we plan to continue to perfect our parser via improved learning and decoding techniques.

Acknowledgments

We want to thank the anonymous reviewers for their suggestions. We also want to thank Jeffrey Flanigan, Xiaochang Peng, Adam Lopez and Giorgio Satta for discussion about ideas related to this work during the Fred Jelinek Memorial Workshop in Prague in 2014. This work was partially supported by the National Science Foundation via Grant No.0910532 entitled Richer Representations for Machine Translation. All views expressed in this paper are those of the authors and do not necessarily represent the view of the National Science Foundation.

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. Association for Computational Linguistics.

Bernd Bohnet and Joakim Nivre. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1455–1465. Association for Computational Linguistics.

Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 748–752. Association for Computational Linguistics.

David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. 2013. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 924–932, Sofia, Bulgaria, August. Association for Computational Linguistics.

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. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 1–8. Association for Computational Linguistics.

Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1426–1436, Baltimore, Maryland, June. Association for Computational Linguistics.

Dan Klein and Christopher D Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*. Citeseer.

Joakim Nivre. 2007. Incremental non-projective dependency parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 396–403. Association for Computational Linguistics.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*, pages 351–359. Association for Computational Linguistics.

Kenji Sagae and Jun'ichi Tsujii. 2008. Shift-reduce dependency dag parsing. In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pages 753–760. Association for Computational Linguistics.

Francesco Sartorio, Giorgio Satta, and Joakim Nivre. 2013. A transition-based dependency parser using a dynamic parsing strategy. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 135–144. Association for Computational Linguistics.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3.