AMR Parsing with Cache Transition Systems

Xiaochang Peng and **Daniel Gildea**
Department of Computer Science
University of Rochester
Rochester, NY 14627
{xpeng, gildea}@cs.rochester.edu

Giorgio Satta
Department of Information Engineering
University of Padua
Via Gradenigo 6/A, 35131 Padova, Italy
satta@dei.unipd.it

Abstract

In this paper, we present a transition system that generalizes transition-based dependency parsing techniques to generate AMR graphs rather than tree structures. In addition to a buffer and a stack, we use a fixed-size cache, and allow the system to build arcs to any vertices present in the cache at the same time. The size of the cache provides a parameter that can trade off between the complexity of the graphs that can be built and the ease of predicting actions during parsing. Our results show that a cache transition system can cover almost all AMR graphs with a small cache size, and our end-to-end system achieves competitive results in comparison with other transition-based approaches for AMR parsing.

Introduction

In recent years, graph-based representations of semantic structures and the algorithms for producing them have gained renewed interest as deeper representations are investigated by statistical natural language processing systems. These algorithms usually take as input a sentence, and produce a graph representation of the semantics of the sentence itself as the output.

Abstract Meaning Representation (AMR) (Banarescu et al. 2013) is a semantic formalism where the meaning of a sentence is encoded as a rooted, directed graph. Figure 1 shows an example of an AMR in which the nodes represent the AMR concepts and the edges represent the relations between the concepts. AMR concepts consist of predicate senses, named entity annotations, and in some cases, simply lemmas of English words. AMR relations consist of core semantic roles drawn from the Propbank (Palmer, Gildea, and Kingsbury 2005) as well as very fine-grained semantic relations defined specifically for AMR. These properties render the AMR representation useful in applications like question answering and semantics-based machine translation.

Stack-based transition systems have been widely used for syntactic parsing as the performance of transition systems has improved, while speed becomes increasingly important for different applications. There are also a number of extensions of stack-based transition systems which deal with non-projective trees; see for instance (Attardi 2006; Nivre 2009; Choi and McCallum 2013; Gómez-Rodríguez and Nivre 2013; Pitler and McDonald 2015).

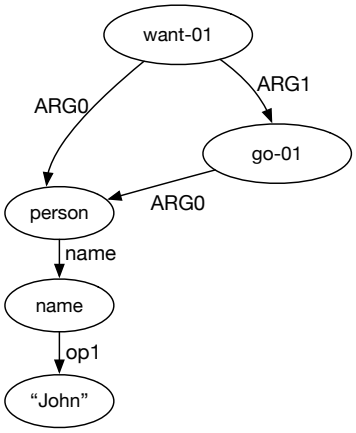

Figure 1: An example of AMR graph representing the meaning of: “John wants to go”

The task of AMR graph parsing is to map natural language strings to AMR semantic graphs. Different parsers have been developed to tackle this problem (Flanigan et al. 2014; Wang, Xue, and Pradhan 2015b; Artzi, Lee, and Zettlemoyer 2015; Peng, Song, and Gildea 2015; Peng et al. 2017; van Noord and Bos 2017; Konstas et al. 2017). Recently, stack-based transition systems have been used for parsing the set of AMR graphs (Wang, Xue, and Pradhan 2015b; 2015a; Damonte, Cohen, and Satta 2016; Wang and Xue 2017). In these systems, special transition actions are added separately to deal with local re-entrancies which are siblings in the generated AMR graph.

Gildea, Satta, and Peng (2018) present an extension of the existing stack-based transition framework to generate the set of semantic graphs. More specifically, they adapt the stack-based parsing system by adding a working set, which they refer to as a cache, to the traditional stack and buffer. They theoretically prove that the class of graphs that can be successfully constructed by this parsing system can be analyzed using the graph-theoretic notion of treewidth. The treewidth of a graph gives a measure of how tightly interconnected it

is, and the size of the cache in the parsing system is correlated to the treewidth statistics.

In this paper, we apply the cache transition system to AMR parsing and deal with some practical implementation issues when designing the parser. We maintain the tokens of each sentence and the newly generated concept in the buffer and the AMR concepts this new concept has access to in a fixed-sized cache. We also put the vertices that will be processed later in the stack structure. As the average treewidth of AMR is low, we assume that the set of AMR graphs can be built with relatively small cache size, which is confirmed in our experiments.

When designing the cache-transition-based AMR parser, we decompose the sequence of push or pop transitions by Gildea, Satta, and Peng (2018) into a sequence of four phases: 1) a push or pop action which either processes the next word in the buffer or moves a concept out of the cache, 2) a concept identification action which generates a concept (or an empty symbol) for the leftmost word, 3) a connect-arc action which builds a labeled arc between a new concept and a concept in the cache, and 4) a push-index action which takes a concept at a certain position of the cache and pushes it onto the stack. Each of the four phases is modeled with a separate feedforward neural classifier and learned separately with oracle actions.

Our preliminary results show that the cache transition system achieves competitive results with relatively simple feature settings in comparison with other transition-based AMR parsers, which shows promising future applications to AMR parsing using more refined features and advanced modeling techniques. Also, our cache transition system is general and can be applied for parsing to other graph structures by adjusting the size of the cache based on the complexity of the graphs.

Parsing to Graphs

Our parser is similar to standard shift-reduce dependency parsing in that its fundamental data structure consists of a stack. However, we allow both crossing arcs, as in non-projective dependency parsing (Attardi 2006; Nivre 2009; Choi and McCallum 2013; Gómez-Rodríguez and Nivre 2013; Pitler and McDonald 2015; Sun, Cao, and Wan 2017), and cyclic graphs, rather than restricting our output to trees. An early transition system producing cyclic graphs is that of Sagae and Tsujii (2008), who drop the constraint of a single head for each word. Wang, Xue, and Pradhan (2015b) provide a transition system that takes a syntactic tree, rather than a string, as input. We wish to take a string as input for greater generality, although our parser can and does use features from a syntactic parser in predicting actions. The most general approach in terms of the class of graphs that can be generated is that of Covington (2001), which was cast as a stack-based transition system by Nivre (2008). Covington (2001) considers building arcs to all previous words as each word is shifted onto the stack, allowing any graph to be built in a total running time of $O(n^2)$ in the sentence length. However, this broad coverage of graphs also makes the prediction of transitions more difficult, as a large number of actions are available to the system at each step. We aim to find a better trade-off between coverage and prediction by designing a system with the properties of semantic graphs in mind. Our use of a stack enforces a tree-like structure on the graphs at a high level, and is consistent with the fact that the AMR graphs tend to have low treewidth. We only build arcs to vertices in our cache, described formally below. Our use of a cache as a working set allows us the flexibility to produce non-projective and cyclic graphs. The size of the cache controls the degree of this flexibility, and provides a parameter that can be tuned to optimize the trade-off between coverage and ease of prediction.

Cache Transition Parser

In this section we precisely define a nondeterministic computational model for graph-based parsing, which is called a cache transition parser (Gildea, Satta, and Peng 2018). The model takes as input an ordered sequence of tokens, reads and processes each token strictly from left to right, and incrementally produces a graph as output.

We apply the cache transition parser to AMR graph parsing. The cache transition parser processes input tokens from a sentence and produces an output AMR graph. The AMR graph is defined on a set of vertices produced from the input tokens. Besides its stack and buffer, the parser also uses a cache. A cache is a fixed-size array of $m \geq 1$ elements and, along with the stack, represents the storage of the parser. At any time during the computation, a vertex that is in the storage of the parser is either in the cache or else in the stack, but not in both at the same time. The tokens of the sentence in the input buffer are first mapped to vertex symbols and then shifted into the cache *before* entering the stack. While in the cache, vertices can be directly accessed and edges between the new vertex and the vertices in the cache can be constructed.

Standard AMR parsing algorithms are usually decomposed into two phases. First the tokens in the string are mapped to vertices (concepts) in the graph. Then in a second phase directed, labeled arcs are made between concepts to build the target AMR graph. We design separate components in our transition system to model this procedure.

Cache Transition System

Formally, a **cache transition parser** consists of a stack, a cache, and an input buffer. The stack is a sequence σ of (integer, vertex) pairs, as explained below, with the topmost element always at the rightmost position. The buffer is a sequence of tokens β containing a suffix of the input sentence, with the first element to be read, possibly with a newly generated vertex of the graph at the leftmost position. Finally, the cache is a sequence of vertices $\eta = [v_1, \ldots, v_m]$. The element at the leftmost position is called the first element of the cache, and the element at the rightmost position is called the last element.

Operationally, the functioning of the parser can be described in terms of configurations and transitions. Each transition is a binary relation defined on the set of configurations. A **configuration** of our parser has the form:

$$C = (\sigma, \eta, \beta, G_p)$$

where σ, η and β are as described above, and G_p is the partial graph that has been built so far. The initial configuration of the parser is $([], [\$, \ldots, \$], [w_1, \ldots, w_n], \emptyset)$, meaning that the stack and the partial graph are initially empty, and the cache is filled with m occurrences of the special symbol \$. The buffer is initialized with all the tokens in the sentence. The final configuration is $([], [\$, \ldots, \$], [], G)$, where the stack and the cache are as in the initial configuration and the buffer is empty. The constructed graph is the goal AMR graph.

The **transitions** of the parser are specified as follows.

- *Pop* pops a pair (i, v) from the stack, where the integer i records the position in the cache that it originally came from. We place v in position i in the cache shifting the remainder of the cache one position to the right, and discarding the last element in the cache.
- *ConceptGen*(c) generates an unaligned concept $c \in P_\epsilon$ and appends it to the left of the buffer. The symbol P_ϵ is the unaligned concept set learned from the training data. A concept is *unaligned* if it is not mapped to any token. The generated concept c is then ready for further processing.[1]
- *ConceptID*(c_i) reads the next token w_i of the buffer, and replaces it with a concept $c_i \in (Q(w_i) \cup \{\epsilon\})$ generated from the token. The symbol Q is a mapping from tokens in the string to concepts or collapsed categories representing subgraphs in the graph. A token is *unaligned* if it is mapped to ϵ and the generated ϵ is shifted out of the buffer immediately, otherwise the generated concept c_i (we call it the candidate concept) is ready for further processing in the next two steps.
- *Arc(i, d, l)* builds an arc with direction d and label l between the candidate concept and the i-th vertex in the cache.[2]
- *PushIndex*(i) shifts the candidate concept out of the buffer and moves it into the last position of the cache. We also take the vertex v_i appearing at position i in the cache and push it onto the stack σ, along with the integer i recording the position in the cache from which it came.

Given the sentence "John wants to go", our cache transition parser can construct the AMR graph shown in Figure 1 using the run shown in Figure 2 with cache size of 2. Each time we process the leftmost word in the buffer, we either rewrite the leftmost word with a candidate concept (person name category *Per* for "John") or an empty symbol and shift it out of the buffer ($\emptyset$ for *to*). If a candidate concept is generated, we proceed to make new arcs between this concept and the concepts in the cache. For example, for the candidate concept *go-01*, the arc actions *Arc(1, L, ARG0)* and *Arc(2, R, ARG1)* make an arc from *go-01* to the first cache concept *Per* and another arc from the second cache concept *want-01* to *go-01*, thus creating the re-entrancy as desired.

[1] In this paper, we don't deal with this type of transitions and leave it as future work. During training, we ignore all training example involving this action.

[2] For ease of this operation, we consider arc choices made to all elements in the cache and then select a vertex to be pushed onto the stack, which is different from Gildea, Satta, and Peng (2018) where a vertex is selected to be pushed onto the stack first and the arc choices are made to the remaining elements in the cache.

Oracle Extraction Algorithm

A cache transition parser is a nondeterministic automaton: given a fixed input sequence π which initializes the buffer and an individual graph G, there may be several runs of the parser on π, each constructing G through a different series of transitions having input order π.

In this section we develop an **oracle** (Nivre 2008) algorithm that can be used to drive a cache transition parser with cache size m, in such a way that the parser becomes deterministic. This means that at most one computation is possible for each pair of G and π. More precisely, our algorithm takes as input a configuration C of the parser obtained when running on π, and a graph G to be constructed. Then the algorithm computes the unique transition that should be applied to C in order to construct G according to the input order π. If the graph G can not be constructed through a sequence of such transitions, then the algorithm fails at some configuration obtained when running on π.

Let E_G be the set of edges of the gold graph G. We also have the alignment table Q from tokens in the input to vertices in the graph. The vertices in G that are not aligned to any token in π are called **unaligned vertices**. We maintain the set of vertices that is not yet shifted into the cache as S, which is initialized with all vertices in G. We also order all the vertices in G according to their aligned position in π and the unaligned vertices are listed according to their order in the depth-first traversal of the graph, which we call a sequence ϕ. The oracle algorithm can look into E_G and Q in order to decide which transition to use at C, or else to decide that it should fail. This decision is based on the mutually exclusive rules listed below.

1. If there is no edge (v_m, v) in E_G such that vertex v is in S, the oracle chooses transition *Pop*.
2. Otherwise, if the next token in the buffer is unaligned, the oracle chooses transition *ConceptID(ϵ)* and simply shifts it out of the buffer.
3. If the next token w_i in the buffer is aligned to concept c_i, the oracle chooses transition *ConceptID(c_i)*. We replace the leftmost token with the candidate concept c_i for further processing, which includes the next two steps.
4. For each vertex in the cache, we make a binary decision about whether there is an arc between the candidate concept c_i and this vertex. If there is an arc, and if d and l are the direction and the label of the arc and the cache vertex is at position j, the oracle chooses transition *Arc(j, d, l)*.
5. In this step, the oracle first chooses an index i in the cache and removes the vertex at this position and then places its index, vertex pair onto the stack. The oracle chooses transition *PushIndex(i)*.
6. If the stack and buffer are both empty, and the cache is in the initial state, the oracle finishes with success, otherwise we proceed to the first step.

stack	cache	buffer	edges	resulting from action
[ ]	[$, $]	[j, w, t, g]	$\emptyset$	—
[ ]	[$, $]	[Per, w, t, g]	$\emptyset$	*ConceptID*(Per)
[ ]	[$, $]	[Per, w, t, g]	$\emptyset$	—
[1, $]	[$, Per]	[w, t, g]	$\emptyset$	*PushIndex*(1)
[1, $]	[$, Per]	[want-01, t, g]	$\emptyset$	*ConceptID*(*want-01*)
[1, $]	[$, Per]	[want-01, t, g]	E_1	*Arc*(*2, L, ARG0*)
[1, $, 1, $]	[Per, want-01]	[t, g]	E_1	*PushIndex*(1)
[1, $, 1, $]	[Per, want-01]	[g]	E_1	*ConceptID*($\emptyset$)
[1, $, 1, $]	[Per, want-01]	[go-01]	E_1	*ConceptID*(*go-01*)
[1, $, 1, $]	[Per, want-01]	[go-01]	E_2	*Arc*(*1, L, ARG0*); *Arc*(*2, R, ARG1*)
[1, $, 1, $, 1, Per]	[want-01, go-01]	[ ]	E_2	*PushIndex*(1)
[1, $, 1, $]	[Per, want-01]	[ ]	E_2	*Pop*
[1, $]	[$, Per]	[ ]	E_2	*Pop*
[ ]	[$, $]	[ ]	E_2	*Pop*

Figure 2: Example run of the cache transition system constructing the graph for the sentence "John wants to go" with cache size of 2. j="John", w="wants", t="to", g="go". E_1 = $\{(Per, want\text{-}01, L\text{-}ARG0)\}$, E_2 = $\{(Per, want\text{-}01, L\text{-}ARG0), (Per, go\text{-}01, L\text{-}ARG0), (want\text{-}01, go\text{-}01, R\text{-}ARG1)\}$.

To decide which position in the cache to take out, we need to develop some additional notation. For $j \in [|\beta|]$, we write β_j to denote the j-th vertex in β (with the candidate concept moved out). We choose a vertex v_{i^*} in η such that:

$$i^* = \underset{i \in [m]}{\operatorname{argmax}} \ \min \ \{j \mid (v_i, \beta_j) \in E_G\}. \qquad (1)$$

In words, v_{i^*} is the vertex from the cache whose closest neighbour in the buffer β is furthest forward in β. In case of ties in the $\min$ and argmax operators, we choose the left-most position. If a vertex in η has no edges pointing to vertices in β, that vertex should be selected. The main idea here is that we want to process vertices by giving higher priority to those vertices with closer forward neighbours. We therefore move out of the cache vertex v_{i^*} and push it onto the stack, for later processing.

AMR Parsing

Training

The basic AMR parsing pipeline is shown in Figure 3. We first run the oracle algorithm on the training data and extract the training examples for each type of transition. We use feedforward neural networks for the training:

$$h = g(W_1 e(f(C)) + b)$$
$$p(a|C;\theta) = softmax(W_2 h)$$

where C is the current configuration and a is the target transition. The function $f(C)$ extracts features from the current configuration C. The function $e()$ is an embedding layer which maps token features to their continuous vector representation and $g()$ is a non-linear activation function. Symbol h is the hidden layer representation, W_1 and W_2 are linear weights between the layers, and b is the bias. As the transitions in our cache-transition system are very diverse, we extract different features for each type of transition and predict each type with a separate classifier. We use separate feedforward neural classifiers for the following transitions:

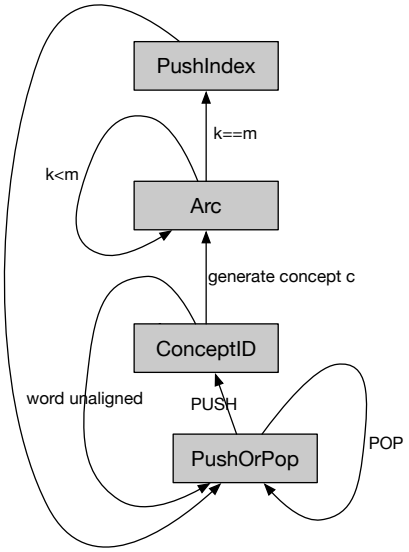

Figure 3: Cache transition AMR parsing pipeline

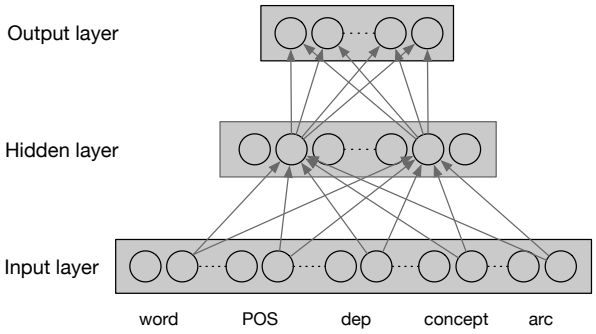

Figure 4: Neural network architecture

Transition type	word, lemma, POS	dep	concept	arc
PushOrPop	η_{-j} ($j = 1$ *to* 3), β_1	#(η_{-1}, β), $l(\eta_{-1}, \beta)$	η_{-j} ($j = 1$ *to* 3)	$\emptyset$
ConceptID	$\beta_1 + i$ ($i = -2$ *to* 2)	$\emptyset$	$\emptyset$	$\emptyset$
ArcBinary	β_1, η_{cur}, *wdist*()	*I*(), *l*(), *ddist*()	β_1, η_{cur}	*children*(β_1), *children*(η_{cur})
ArcLabel	β_1, η_{cur}, *wdist*()	*l*(), *ddist*(), *deps*(β_1), *deps*(η_{cur})	β_1, η_{cur}	*children*(β_1), *children*(η_{cur})
PushIndex	η	$\emptyset$	η	$\emptyset$

Table 1: Features used for each classifier

- We use a binary classifier to decide whether to *Pop* or not. If the next action is not *Pop*, we proceed to process the buffer.
- We also predict the concept labels for the next word in the buffer using a classifier.[3] As the output vocabulary is too large, we only predict the possible candidates from the alignment of the training data. If a word is unseen, we build its candidate using its POS tag and lemma l: if it's a verb, we use *l-01*. Otherwise we use l as the candidate.
- When connecting arcs from the candidate concept to concepts in the cache, we first use a binary classifier to predict whether there is an arc between the two concepts (*ArcBinary*). If there is an arc, another classifier is used to predict the direction and label of the arc (*ArcLabel*). This procedure continues until all the decisions between the candidate concept and the concepts in the cache are decided.
- Finally we predict the index of the concept in the cache that needs to be pushed onto the stack and place the candidate concept in the last position of the cache.

Feature Extraction

We extract features separately for each classifier based on the current configuration. Figure 4 shows the neural architecture for each feedforward classifier. The detailed features are shown in Table 1. Here η_{-j} is the word position for the j-th rightmost cache element and η_{cur} is the word position for the current cache element. The symbol β_1 is the word position of the leftmost element in the buffer and β, η mean we look at all the positions in the buffer and cache separately. For concept features, these indices indicate the concept generated from the word at each position.[4] The function #(η_{-1}, β) computes the number of dependency arcs between the rightmost cache position and all the positions in the buffer, while the function l is the dependency arc label for each connection.

For ArcBinary and ArcLabel, function $wdist$ computes the word distance of two words while $ddist$ computes their distance in the dependency tree. The function I is an indicator of whether there is a dependency arc between the candidate word and the current cache position, while l is the dependency arc label. The function $deps$ looks at the dependencies of the word while $children$ looks at the concept generated from a certain word and returns all the arcs generated so far for the concept.

Categorization of Data

As the AMR data is very sparse, we first collapse some subgraphs and some spans into corresponding categories based on the alignment. We define some special categories such as named entities (*NE*), dates (*DATE*), verbalization[5] (*VB*), numbers (*NUMBER*) and phrases (*PHRASE*). The phrases are extracted based on the multiple to one alignment in the training data. We consider a span of tokens to be a phrase if these tokens align to the same concept more frequently than to separate concepts. One example phrase is *more_than* which aligns to a single concept *more-than*. We first collapse spans and subgraphs into these categories based on the alignment from an automatic aligner (Pourdamghani et al. 2014). This categorization procedure enables the parser to capture mappings from continuous spans on the sentence side to connected subgraphs on the AMR side.

During decoding, our output has categories, and we need to map each category to the corresponding AMR concepts or subgraphs. When we categorize the tokens or spans on the sentence side, we save the mapping from each category to its original token or span as a table Q. Given table Q, we can track which source side token or span it comes from and use either the most frequent concept or subgraph learned from the training data or use heuristic rules to generate the target-side AMR subgraph based on the source side tokens (for example, *NE* and *DATE*).

Experiments

We evaluate our system on the released dataset (LDC2015E86) for SemEval 2016 task 8 on meaning representation parsing (May 2016). The dataset contains 16,833 training, 1,368 development and 1,371 test sentences which mainly cover domains like newswire, discussion forum, etc. All parsing results are measured by Smatch (version 2.0.2) (Cai and Knight 2013).

Experiment Settings

We categorize the training data using the automatic alignment and dump a template for date entities and frequent phrases from the multiple to one alignment. We also generate an alignment table from tokens or phrases to their candidate target-side subgraphs. For the dev and test data, we

[3] In practice, using feedforward, average perceptron or most frequent concept for this step perform similarly.

[4] When we generate a candidate concept or shift the candidate concept into the cache, we also keep the position of the word it is generated from.

[5] http://amr.isi.edu/download/lists/verbalization-list-v1.06.txt

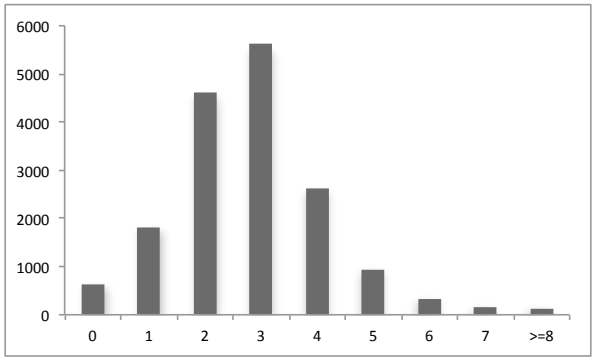

Figure 5: Statistics of AMR graphs that can be processed using different cache size

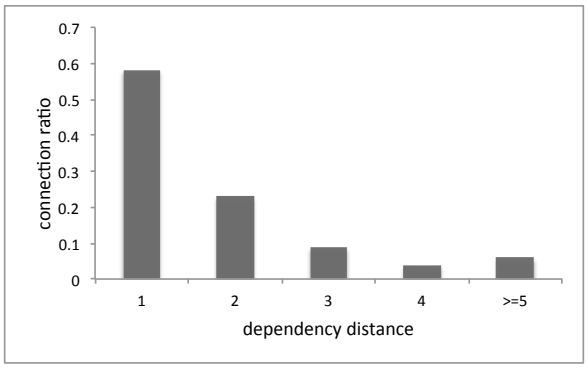

Figure 6: Connection ratio relative to distance in the dependency tree

first extract the named entities using the Illinois Named Entity Tagger (Ratinov and Roth 2009) and extract date entities by matching spans with the date template. We further categorize the dataset with the other categories we have defined. After categorization, we use Stanford CoreNLP (Manning et al. 2014) to get the POS tags and dependencies of the categorized dataset. We run the oracle algorithm separately for training and dev data (with alignment) to get the statistics of the AMR dataset and also the training and evaluation examples for each classifier. As for the feedforward classifiers, we use one hidden layer with 200 tanh hidden units and a learning rate of 0.005. The embeddings for words are pre-trained from an April 2010 snapshot of the Wikipedia corpus (Shaoul and Westbury 2010) using the skip-gram model of *word2vec* (Mikolov et al. 2013a; 2013b). The embeddings for other token features are randomly initialized. The number of dimensions for words and other token features are 50. The concatenation of these token embeddings and the numerical feature embeddings is fed as input to the network.

Results

First we compute the coverage statistics of the training data. Our transition system can already cover 90% of the AMR graphs with cache size 4. Using cache size 7, the coverage can go up to 99%. This shows that the cache transition system is more capable for generating the set of AMR graphs. In our AMR parsing experiments, we use a cache size of 4.

Classifier	accuracy
PushOrPop	0.87
ConceptID	0.88
ArcBinary	0.83
ArcLabel	0.81
PushIndex	0.87

Table 2: Performance breakdown of each classifier

We proceed to more detailed evaluation of our parser on the AMR dataset. We first evaluate the prediction accuracy for each classifier. We can see from Table 2 that all the classifiers can achieve an accuracy between 0.80 and 0.90. For the *ConceptID* classifier, part of the errors are propagated from the alignment errors from the automatic aligner. Another type of alignment error comes from the limits we put on the current parser. Currently we don't design specific actions to handle alignment from discontinuous tokens to a concept or from one token to disconnected components of the graph, while this kind of alignment does appear in the data (although not frequently). When one token maps to multiple disconnected concepts on the graph, we randomly pick one concept as the aligned concept and the others are considered unaligned. When discontinuous tokens align to the same concept, we only consider the alignment from the first continuous span.

The *ArcBinary* and *ArcLabel* actions are directly related to the labeled arcs in the AMR graph. Figure 6 shows the distribution of connected arcs relative to different dependency distance. We can see that most of arcs are connected between words that are within a distance of 3 in the dependency tree (around 90%). In practice, we get the best performance when only allowing arcs to be connected within a dependency distance of 3. Table 2 shows the accuracy of the *ArcBinary* using this constraint. For *ArcLabel*, we have also used a constraint on the arc labels that are allowed between different categories of concepts. While Damonte, Cohen, and Satta (2016) have designed specific rules for different types of concepts and used Propbank to constrain the ARGs for different predicates, here we use the frequency of concepts and their category information to constrain the set of arc labels between the two concepts. Here we have used a frequency threshold of 100 for concepts in the training data and the categories we have introduced. For other low-frequency concepts, we map predicates to the type *PRED-01* and others to *OTHER*. The basic intuition is that if a concept appears frequently enough, it is very likely that the possible arc labels already appear in a certain context of that concept. In practice, the labeling accuracy using this constraint is 0.81.

Finally, we compare our cache transition parser with the two other transition-based AMR parsers: CAMR and Damonte, Cohen, and Satta (2016). Table 3 shows the com-

Sentence: the pirates have consistently expressed willingness to negotiate the financial figures .

Reference AMR:

```
(e/express-01 :ARG0 (p/pirate)
              :ARG1 (w/will-02 :ARG0 p
                               :ARG1 (n/negotiate-01 :ARG0 p
                                                     :ARG2 (f/figure :mod (f/finance))))
              :manner (c/consistent))
```

Output AMR:

```
(e/express-01 :ARG0 (p/pirate)
              :ARG1 (w/will-02 :ARG0 p
                               :ARG1 (n/negotiate-01 :ARG0 (c/consistent)
                                                     :ARG2 (f/figure :mod (f/finance)))
                               :manner c)
              :purpose n
              :manner c)
```

Figure 7: An example AMR output for predicting re-entrancies.

System	P	R	F
Our system	0.69	0.59	0.64
Damonte et al.			0.64
CAMR (github)	0.64	0.62	0.63
CAMR (full)	0.70	0.63	0.66

Table 3: Comparison to other AMR parsers.

parison with other transition-based parsers. CAMR (github) shows the performance with the actions and features described in Wang, Xue, and Pradhan (2015b), and CAMR (full) is with a refined action set and additional features such as coreference information, semantic role labeling and word cluster features. We can see that with the relatively simple feature settings and using feedforward neural networks, our parser can already achieve competitive results in comparison with the other transition-based parsers.

Predicting re-entrancy is an important but also challenging part of AMR parsing. In Figure 7, we can see an output AMR graph where re-entrancy appears. In this example, our parser can accurately predict the re-entrancy from *will-02* to *pirate*. However, it also makes the wrong re-entrancy from *negotiate-01* to *consistent* instead of *pirate*. This error results from the decision of the *ArcBinary* classifier, which usually prefers making arcs between words that are close and not if they are distant. The parser also makes an additional re-entrancy from *purpose* to *negotiate-01*. This re-entrancy results from their adjacency in the dependency tree. More refined features would be needed to solve these re-entrancy issues concerning distance.

Conclusion

In this paper, we have designed a cache transition system for AMR parsing. Our experiments show that the set of AMR graphs can be constructed with relatively small cache size and competitive results can be achieved using our transition system. It would be interesting to extend our current feature settings to model long-distance dependencies and re-entrancies. The cache transition system also provides a good trade-off between the set of graph structures that can be constructed and the speed for building the graphs. By adjusting the size of the cache, the cache transition system can be easily applied to other parsing to graph tasks.

Acknowledgments

This work was supported in part by a Google Faculty Award.

References

Artzi, Y.; Lee, K.; and Zettlemoyer, L. 2015. Broad-coverage CCG semantic parsing with AMR. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 1699–1710. Lisbon, Portugal: Association for Computational Linguistics.

Attardi, G. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, 166–170.

Banarescu, L.; Bonial, C.; Cai, S.; Georgescu, M.; Griffitt, K.; Hermjakob, U.; Knight, K.; Koehn, P.; Palmer, M.; and Schneider, N. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, 178–186.

Cai, S., and Knight, K. 2013. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13)*, 748–752.

Choi, J. D., and McCallum, A. 2013. Transition-based dependency parsing with selectional branching. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13)*, 1052–1062.

Covington, M. A. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, 95–102.

Damonte, M.; Cohen, S. B.; and Satta, G. 2016. An incremental parser for abstract meaning representation. *CoRR* abs/1608.06111.

Flanigan, J.; Thomson, S.; Carbonell, J.; Dyer, C.; and Smith, N. A. 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)*, 1426–1436. Baltimore, Maryland: Association for Computational Linguistics.

Gildea, D.; Satta, G.; and Peng, X. 2018. Cache transition systems for graph parsing. *Computational Linguistics*. to appear.

Gómez-Rodríguez, C., and Nivre, J. 2013. Divisible transition systems and multiplanar dependency parsing. *Computational Linguistics* 39:799–846.

Konstas, I.; Iyer, S.; Yatskar, M.; Choi, Y.; and Zettlemoyer, L. 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)*, 146–157. Vancouver, Canada: Association for Computational Linguistics.

Manning, C. D.; Surdeanu, M.; Bauer, J.; Finkel, J. R.; Bethard, S.; and McClosky, D. 2014. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, 55–60.

May, J. 2016. Semeval-2016 task 8: Meaning representation parsing. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, 1063–1073.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013a. Efficient estimation of word representations in vector space. *CoRR* abs/1301.3781.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, 3111–3119.

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

Nivre, J. 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*, 351–359.

Palmer, M.; Gildea, D.; and Kingsbury, P. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics* 31(1):71–106.

Peng, X.; Wang, C.; Gildea, D.; and Xue, N. 2017. Addressing the data sparsity issue in neural amr parsing. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, 366–375. Valencia, Spain: Association for Computational Linguistics.

Peng, X.; Song, L.; and Gildea, D. 2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning*, 32–41. Beijing, China: Association for Computational Linguistics.

Pitler, E., and McDonald, R. 2015. A linear-time transition system for crossing interval trees. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 662–671.

Pourdamghani, N.; Gao, Y.; Hermjakob, U.; and Knight, K. 2014. Aligning English strings with abstract meaning representation graphs. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 425–429. Doha, Qatar: Association for Computational Linguistics.

Ratinov, L., and Roth, D. 2009. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL-2009)*, 147–155. Boulder, Colorado: Association for Computational Linguistics.

Sagae, K., and Tsujii, J. 2008. Shift-reduce dependency DAG parsing. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING-08)*, 753–760.

Shaoul, C., and Westbury, C. 2010. The Westbury lab wikipedia corpus.

Sun, W.; Cao, J.; and Wan, X. 2017. Semantic dependency parsing via book embedding. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 828–838. Vancouver, Canada: Association for Computational Linguistics.

van Noord, R., and Bos, J. 2017. Neural semantic parsing by character-based translation: Experiments with abstract meaning representations. *arXiv preprint arXiv:1705.09980*.

Wang, C., and Xue, N. 2017. Getting the most out of AMR parsing. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 1268–1279. Copenhagen, Denmark: Association for Computational Linguistics.

Wang, C.; Xue, N.; and Pradhan, S. 2015a. Boosting transition-based AMR parsing with refined actions and auxiliary analyzers. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, 857–862. Beijing, China: Association for Computational Linguistics.

Wang, C.; Xue, N.; and Pradhan, S. 2015b. A transition-based algorithm for AMR parsing. In *Proceedings of the 2015 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-15)*, 366–375.