AMR-to-text generation as a Traveling Salesman Problem

Linfeng Song[1], **Yue Zhang**[3], **Xiaochang Peng**[1], **Zhiguo Wang**[2] and **Daniel Gildea**[1]
[1]Department of Computer Science, University of Rochester, Rochester, NY 14627
[2]IBM T.J. Watson Research Center, Yorktown Heights, NY 10598
[3]Singapore University of Technology and Design

Abstract

The task of AMR-to-text generation is to generate grammatical text that sustains the semantic meaning for a given AMR graph. We attack the task by first partitioning the AMR graph into smaller fragments, and then generating the translation for each fragment, before finally deciding the order by solving an asymmetric generalized traveling salesman problem (AGTSP). A Maximum Entropy classifier is trained to estimate the traveling costs, and a TSP solver is used to find the optimized solution. The final model reports a BLEU score of 22.44 on the SemEval-2016 Task8 dataset.

1 Introduction

Abstract Meaning Representation (AMR) (Banarescu et al., 2013) is a semantic formalism encoding the meaning of a sentence as a rooted, directed graph. Shown in Figure 1, the nodes of an AMR graph (e.g. "boy", "go-01" and "want-01") represent concepts, and the edges (e.g. "ARG0" and "ARG1") represent relations between concepts. AMR jointly encodes a set of different semantic phenomena, which makes it useful in applications like question answering and semantics-based machine translation. AMR has served as an intermediate representation for various text-to-text NLP applications, such as statistical machine translation (SMT) (Jones et al., 2012).

The task of AMR-to-text generation is to generate grammatical text containing the same semantic meaning as a given AMR graph. This task is important yet also challenging since each AMR graph usually has multiple corresponding sentences, and syntactic structure and function words are abstracted away when transforming a sentence into AMR (Banarescu et al., 2013). There has been work dealing with text-to-AMR parsing (Flanigan et al., 2014; Wang et al., 2015; Peng et al., 2015; Vanderwende et al., 2015; Pust et al., 2015; Artzi et al., 2015). On the other hand, relatively little work has been done on AMR-to-text generation. One recent exception is Flanigan et al. (2016), who first generate a spanning tree for the input AMR graph, and then apply a tree transducer to generate the sentence. Here, we directly generate the sentence from an input AMR by treating AMR-to-text generation as a variant of the traveling salesman problem (TSP).

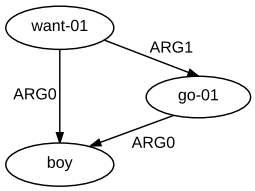

Figure 1: AMR graph for "The boy wants to go".

Given an AMR as input, our method first cuts the graph into several rooted and connected fragments (sub-graphs), and then finds the translation for each fragment, before finally generating the sentence for the whole AMR by ordering the translations. To cut the AMR and translate each fragment, we match the input AMR with rules, each consisting of a rooted, connected AMR fragment and a corresponding translation. These rules serve in a similar way to rules in SMT models. We learn the rules by a modified version of the sampling algorithm of Peng

et al. (2015), and use the rule matching algorithm of Cai and Knight (2013).

For decoding the fragments and synthesizing the output, we define a *cut* to be a subset of matched rules without overlap that covers the AMR, and an *ordered cut* to be a cut with the rules being ordered. To generate a sentence for the whole AMR, we search for an ordered cut, and concatenate translations of all rules in the cut. TSP is used to traverse different cuts and determine the best order. Intuitively, our method is similar to phrase-based SMT, which first cuts the input sentence into phrases, then obtains the translation for each source phrase, before finally generating the target sentence by ordering the translations. Although the computational cost of our method is low, the initial experiment is promising, yielding a BLEU score of 22.44 on a standard benchmark.

2 Method

We reformulate the problem of AMR-to-text generation as an asymmetric generalized traveling salesman problem (AGTSP), a variant of TSP.

2.1 TSP and its variants

Given a *non-directed* graph G_N with n cities, supposing that there is a traveling cost between each pair of cities, TSP tries to find a tour of the minimal total cost visiting each city exactly once. In contrast, the asymmetric traveling salesman problem (ATSP) tries to find a tour of the minimal total cost on a *directed* graph, where the traveling costs between two nodes are different in each direction. Given a directed graph G_D with n nodes, which are clustered into m groups, the asymmetric generalized traveling salesman problem (AGTSP) tries to find a tour of the minimal total cost visiting each *group* exactly once.

2.2 AMR-to-text Generation as AGTSP

Given an input AMR A, each node in the AGTSP graph can be represented as (c, r), where c is a concept in A and $r = (A_{sub}, T_{sub})$ is a rule that consists of an AMR fragment containing c and a translation of the fragment. We put all nodes containing the same concept into one group, thereby translating each concept in the AMR exactly once.

To show a brief example, consider the AMR in Figure 1 and the following rules,

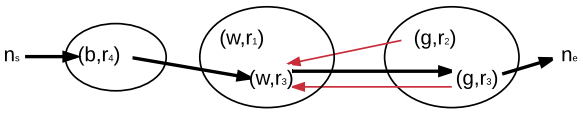

Figure 2: An example AGTSP graph

r_1	(w/want-01) ||| wants
r_2	(g/go-01) ||| to go
r_3	(w/want-01 :ARG1 g/go-01) ||| wants to go
r_4	(b/boy) ||| The boy

We build an AGTSP graph in Figure 2, where each circle represents a group and each tuple (such as (b, r_4)) represents a node in the AGTSP graph. We add two nodes n_s and n_e representing the start and end nodes respectively. Each belongs to a specific group that only contains that node, and a tour always starts with n_s and ends with n_e. Legal moves are shown in black arrows, while illegal moves are shown in red. One legal tour is $n_s \rightarrow (b, r_4) \rightarrow (w, r_3) \rightarrow (g, r_3) \rightarrow n_e$. The order in which nodes within a rule are visited is arbitrary; for a rule with N concepts, the number of visiting orders is $O(N!)$. To reduce the search space, we enforce the breadth first order by setting costs to zero or infinity. In our example, the traveling cost from (w, r_3) to (g, r_3) is 0, while the traveling cost from (g, r_3) to (w, r_3) is infinity. Traveling from (g, r_2) to (w, r_3) also has infinite cost, since there is overlap on the concept "w/want-01" between them.

The traveling cost is calculated by Algorithm 1. We first add n_s and n_e serving the same function as Figure 2. The traveling cost from n_s directly to n_e is infinite, since a tour has to go through other nodes before going to the end. On the other hand, the traveling cost from n_e to n_s is 0 (Lines 3-4), as a tour always goes back to the start after reaching the end. The traveling cost from n_s to $n_i = (c_i, r_i)$ is the model score only if c_i is the first node of the AMR fragment of r_i, otherwise the traveling cost is infinite (Lines 6-9). Similarly, the traveling cost from n_i to n_e is the model score only if c_i is the last node of the fragment of r_i. Otherwise, it is infinite (Lines 10-13). The traveling cost from $n_i = (c_i, r_i)$ to $n_j = (c_j, r_j)$ is 0 if r_i and r_j are the same rule and c_j is the next node of c_i in the AMR fragment of r_i (Lines 16-17).

A tour has to travel through an AMR fragment be-

Data: Nodes in AGTSP graph G
Result: Traveling Cost Matrix T

1 $n_s \leftarrow$ (“<s>”,“<s>”);
2 $n_e \leftarrow$ (“</s>”,“</s>”);
3 T[n_s][n_e] $\leftarrow \infty$;
4 T[n_e][n_s] $\leftarrow 0$;
5 **for** $n_i \leftarrow (c_i, r_i)$ ***in*** G **do**
6 **if** c_i = r_i.*frag.first* **then**
7 T[n_s][n_i] $\leftarrow$ ModelScore(n_s,n_i);
8 **else**
9 T[n_s][n_i] $\leftarrow \infty$;
10 **if** c_i = r_i.*frag.last* **then**
11 T[n_i][n_e] $\leftarrow$ ModelScore(n_i,n_e);
12 **else**
13 T[n_i][n_e] $\leftarrow \infty$;
14 **for** $n_i \leftarrow (c_i, r_i)$ ***in*** G **do**
15 **for** $n_j \leftarrow (c_j, r_j)$ ***in*** G **do**
16 **if** r_i = r_j *and* r_i.*frag.next*(c_i) = c_j **then**
17 T[n_i][n_j] $\leftarrow 0$
18 **else if** r_i.*frag* $\cap$ r_j.*frag* = $\emptyset$ *and* c_i = r_i.*frag.last and* c_j = r_j.*frag.first* **then**
19 T[n_i][n_j] $\leftarrow$ ModelScore(n_i,n_j)
20 **else**
21 T[n_i][n_j] $\leftarrow \infty$

Algorithm 1: Traveling cost algorithm

fore jumping to another fragment. We choose the breadth-first order of nodes within the same rule, which is guaranteed to exist, as each AMR fragment is rooted and connected. Costs along the breadth-first order within a rule r_i are set to 0, while other costs with a rule are infinite.

If r_i is not equal to r_j, then the traveling cost is the model score if there is no overlap between r_i and r_j's AMR fragment and it moves from r_i's last node to r_j's first node (Lines 18-19), otherwise the traveling cost is infinite (Lines 20-21). All other cases are illegal and we assign infinite traveling cost. We do not allow traveling between overlapping nodes, whose AMR fragments share common concepts. Otherwise the traveling cost is evaluated by a maximum entropy model, which will be discussed in detail in Section 2.4.

2.3 Rule Acquisition

We extract rules from a corpus of (sentence, AMR) pairs using the method of Peng et al. (2015). Given an aligned (sentence, AMR) pair, a *phrase-fragment pair* is a pair $([i, j], f)$, where $[i, j]$ is a span of the sentence and f represents a connected and rooted AMR fragment. A *fragment decomposition forest* consists of all possible phrase-fragment pairs that satisfy the alignment agreement for phrase-based MT (Koehn et al., 2003). The rules that we use for generation are the result of applying an MCMC procedure to learn a set of likely phrase-fragment pairs from the forests containing all possible pairs. One difference from the work of Peng et al. (2015) is that, while they require the string side to be tight (does not include unaligned words on both sides), we expand the tight phrases to incorporate unaligned words on both sides. The intuition is that they do text-to-AMR parsing, which often involves discarding function words, while our task is AMR-to-text generation, and we need to be able to fill in these unaligned words. Since incorporating unaligned words will introduce noise, we rank the translation candidates for each AMR fragment by their counts in the training data, and select the top N candidates.[1]

We also generate *concept rules* which directly use a morphological string of the concept for translation. For example, for concept “w/want-01” in Figure 1, we generate concept rules such as “(w/want-01) ||| want”, “(w/want-01) ||| wants”, “(w/want-01) ||| wanted” and “(w/want-01) ||| wanting”. The algorithm (described in section 2.2) will choose the most suitable one from the rule set. It is similar to most MT systems in creating a translation candidate for each word, besides normal translation rules. It is easy to guarantee that the rule set can fully cover every input AMR graph.

Some concepts (such as “have-rel-role-91”) in an AMR graph do not contribute to the final translation, and we skip them when generating concept rules. Besides that, we use a verbalization list[2] for concept rule generation. For rule “VERBALIZE peacekeeping TO keep-01 :ARG1 peace”, we will create a concept rule “(k/keep-01 :ARG1 (p/peace)) ||| peacekeeping” if the left-hand-side fragment appears in the target graph.

[1] Our code for grammar induction can be downloaded from https://github.com/xiaochang13/AMR-generation

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

2.4 Traveling cost

Considering an AGTSP graph whose nodes are clustered into m groups, we define the traveling cost for a tour T in Equation 1:

$$\text{cost}(n_s, n_e) = -\sum_{i=0}^{m} \log p(\text{“yes”}|n_{T_i}, n_{T_{i+1}}) \quad (1)$$

where $n_{T_0} = n_s$, $n_{T_{m+1}} = n_e$ and each n_{T_i} ($i \in [1 \dots m]$) belongs to a group that is different from all others. Here $p(\text{“yes”}|n_j, n_i)$ represents a learned score for a move from n_j to n_i. The choices before n_{T_i} are independent from choosing $n_{T_{i+1}}$ given n_{T_i} because of the Markovian property of the TSP problem. Previous methods (Zaslavskiy et al., 2009) evaluate traveling costs $p(n_{T_{i+1}}|n_{T_i})$ by using a language model. Inevitably some rules may only cover one translation word, making only bigram language models naturally applicable. Zaslavskiy et al. (2009) introduces a method for incorporating a trigram language model. However, as a result, the number of nodes in the AGTSP graph grows exponentially.

To tackle the problem, we treat it as a *local* binary (“yes” or “no”) classification problem whether we should move to n_j from n_i. We train a maximum entropy model, where $p(\text{“yes”}|n_i, n_j)$ is defined as:

$$p(\text{“yes”}|n_i, n_j) = \frac{1}{Z(n_i, n_j)} \exp\Big[\sum_{i=1}^{k} \lambda_i f_i(\text{“yes”}, n_i, n_j)\Big] \quad (2)$$

The model uses 3 real-valued features: a language model score, the word count of the concatenated translation from n_i to n_j, and the length of the shortest path from n_i's root to n_j's root in the input AMR. If either n_i or n_j is the start or end node, we set the path length to 0. Using this model, we can use whatever N-gram we have at each time. Although language models favor shorter translations, word count will balance the effect, which is similar to MT systems. The length of the shortest path is used as a feature because the concepts whose translations are adjacent usually have lower path length than others.

3 Experiments

3.1 Setup

We use the dataset of SemEval-2016 Task8 (Meaning Representation Parsing), which contains 16833 training instances, 1368 dev instances and 1371 test instances. Each instance consists of an AMR graph and a sentence representing the same meaning. Rules are extracted from the training data, and hyperparameters are tuned on the dev set. For tuning and testing, we filter out sentences that have more than 30 words, resulting in 1103 dev instances and 1055 test instances. We train a 4-gram language model (LM) with gigaword (LDC2011T07), and use BLEU (Papineni et al., 2002) as the evaluation metric. To solve the AGTSP, we use Or-tool[3].

System	Dev	Test
PBMT	13.13	16.94
OnlyConceptRule	13.15	14.93
OnlyInducedRule	17.68	18.09
OnlyBigramLM	17.19	17.75
All	21.12	22.44
JAMR-gen	**23.00**	**23.00**

Table 1: Main results.

Our graph-to-string rules are reminiscent of phrase-to-string rules in phrase-based MT (PBMT). We compare our system to a baseline (*PBMT*) that first linearizes the input AMR graph by breadth first traversal, and then adopts the PBMT system from Moses[4] to translate the linearized AMR into a sentence. To traverse the children of an AMR concept, we use the original order in the text file. The MT system is trained with the default setting on the same dataset and LM. We also compare with *JAMR-gen*[5] (Flanigan et al., 2016), which is trained on the same dataset but with a 5-gram LM from gigaword (LDC2011T07).

To evaluate the importance of each module in our system, we develop the following baselines: *OnlyConceptRule* uses only the concept rules, *OnlyInducedRule* uses only the rules induced from the fragment decomposition forest, *OnlyBigramLM* uses both types of rules, but the traveling cost is evaluated by a bigram LM trained with gigaword.

3.2 Results

The results are shown in Table 1. Our method (*All*) significantly outperforms the baseline (*PBMT*)

[3]https://developers.google.com/optimization/

[4]http://www.statmt.org/moses/

[5]https://github.com/jflanigan/jamr/tree/Generator

(w / want-01 :ARG0 (b / boy) :ARG1 (b2 / believe-01 :ARG0 (g / girl) :ARG1 b))
Ref: the boy wants the girl to believe him
All: a girl wanted to believe him
JAMR-gen: boys want the girl to believe

Table 2: Case study.

on both the dev and test sets. *PBMT* does not outperform *OnlyBigramLM* and *OnlyInducedRule*, demonstrating that our rule induction algorithm is effective. We consider rooted and connected fragments from the AMR graph, and the TSP solver finds better solutions than beam search, as consistent with Zaslavskiy et al. (2009). In addition, *OnlyInducedRule* is significantly better than *OnlyConceptRule*, showing the importance of induced rules on performance. This also confirms the reason that *All* outperforms *PBMT*. This result confirms our expectation that concept rules, which are used for fulfilling the coverage of an input AMR graph in case of OOV, are generally not of high quality. Moreover, *All* outperforms *OnlyBigramLM* showing that our maximum entropy model is stronger than a bigram language model. Finally, *JAMR-gen* outperforms *All*, while *JAMR-gen* uses a higher order language model than *All* (5-gram VS 4-gram).

For rule coverage, around 31% AMR graphs and 84% concepts in the development set are covered by our induced rules extracted from the training set.

3.3 Analysis and Discussions

We further analyze *All* and *JAMR-gen* with an example AMR and show the AMR graph, the reference, and results in Table 2. First of all, both *All* and *JAMR-gen* outputs a reasonable translation containing most of the meaning from the AMR. On the other hand, *All* fails to recognize "boy" as the subject. The reason is that the feature set does not include edge labels, such as "ARG0" and "ARG1". Finally, neither *All* and *JAMR-gen* can handle the situation when a re-entrance node (such as "b/boy" in example graph of Table 2) need to be translated twice. This limitation exists for both works.

4 Related Work

Our work is related to prior work on AMR (Banarescu et al., 2013). There has been a list of work on AMR parsing (Flanigan et al., 2014; Wang et al., 2015; Peng et al., 2015; Vanderwende et al., 2015; Pust et al., 2015; Artzi et al., 2015), which predicts the AMR structures for a given sentence. On the reverse direction, Flanigan et al. (2016) and our work here study sentence generation from a given AMR graph. Different from Flanigan et al. (2016) who map a input AMR graph into a tree before linearization, we apply synchronous rules consisting of AMR graph fragments and text to directly transfer a AMR graph into a sentence. In addition to AMR parsing and generation, there has also been work using AMR as a semantic representation in machine translation (Jones et al., 2012).

Our work also belongs to the task of text generation (Reiter and Dale, 1997). There has been work on generating natural language text from a bag of words (Wan et al., 2009; Zhang and Clark, 2015), surface syntactic trees (Zhang, 2013; Song et al., 2014), deep semantic graphs (Bohnet et al., 2010) and logical forms (White, 2004; White and Rajkumar, 2009). We are among the first to investigate generation from AMR, which is a different type of semantic representation.

5 Conclusion

In conclusion, we showed that a TSP solver with a few real-valued features can be useful for AMR-to-text generation. Our method is based on a set of graph to string rules, yet significantly better than a PBMT-based baseline. This shows that our rule induction algorithm is effective and that the TSP solver finds better solutions than beam search.

Acknowledgments

We are grateful for the help of Jeffrey Flanigan, Lin Zhao, and Yifan He. This work was funded by NSF IIS-1446996, and a Google Faculty Research Award. Yue Zhang is funded by NSFC61572245 and T2MOE201301 from Singapore Ministry of Education.

References

Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. 2015. Broad-coverage CCG semantic parsing with AMR. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-15)*, pages 1699–1710.

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.

Bernd Bohnet, Leo Wanner, Simon Mill, and Alicia Burga. 2010. Broad coverage multilingual deep sentence generation with a stochastic multi-level realizer. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING-10)*, pages 98–106.

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 (ACL-13)*, pages 748–752.

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 (ACL-14)*, pages 1426–1436.

Jeffrey Flanigan, Chris Dyer, Noah A. Smith, and Jaime Carbonell. 2016. Generation from abstract meaning representation using tree transducers. In *Proceedings of the 2016 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-16)*, pages 731–739.

Bevan Jones, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, and Kevin Knight. 2012. Semantics-based machine translation with hyperedge replacement grammars. In *Proceedings of the International Conference on Computational Linguistics (COLING-12)*, pages 1359–1376.

Philipp Koehn, Franz Josef Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *Proceedings of the 2003 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-03)*, pages 48–54.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Conference of the Association for Computational Linguistics (ACL-02)*, pages 311–318.

Xiaochang Peng, Linfeng Song, and Daniel Gildea. 2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning (CoNLL-15)*, pages 731–739.

Michael Pust, Ulf Hermjakob, Kevin Knight, Daniel Marcu, and Jonathan May. 2015. Parsing English into abstract meaning representation using syntax-based machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-15)*, pages 1143–1154.

Ehud Reiter and Robert Dale. 1997. Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–87.

Linfeng Song, Yue Zhang, Kai Song, and Qun Liu. 2014. Joint morphological generation and syntactic linearization. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-14)*, pages 1522–1528.

Lucy Vanderwende, Arul Menezes, and Chris Quirk. 2015. An AMR parser for English, French, German, Spanish and Japanese and a new AMR-annotated corpus. In *Proceedings of the 2015 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-15)*, pages 26–30.

Stephen Wan, Mark Dras, Robert Dale, and Cécile Paris. 2009. Improving grammaticality in statistical sentence generation: Introducing a dependency spanning tree algorithm with an argument satisfaction model. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL-09)*, pages 852–860.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015. 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)*, pages 366–375.

Michael White and Rajakrishnan Rajkumar. 2009. Perceptron reranking for CCG realization. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-09)*, pages 410–419.

Michael White. 2004. Reining in CCG chart realization. In *International Conference on Natural Language Generation (INLG-04)*, pages 182–191.

Mikhail Zaslavskiy, Marc Dymetman, and Nicola Cancedda. 2009. Phrase-based statistical machine translation as a traveling salesman problem. In *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics (ACL-09)*, pages 333–341.

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

Yue Zhang. 2013. Partial-tree linearization: Generalized word ordering for text synthesis. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-13)*, pages 2232–2238.