TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik
Institut für Theoretische Informatik
Lehrstuhl für Grundlagen der Programmierung

Masterarbeit

Transition-based Generation from Abstract Meaning Representations

eingereicht von: Timo Schick
timo.schick@tu-dresden.de

eingereicht am: 04. Juli 2017

Verantw. Hochschullehrer: Prof. Dr.-Ing. habil. Heiko Vogler

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Dresden, den 04.07.2017

_____

Timo Schick

Aufgabenstellung für die Masterarbeit

„Generierung von Abstract Meaning Representations“

Technische Universität Dresden
Fakultät Informatik

Student:	Timo Schick
Geburtsdatum:	4. Oktober 1993
Matrikelnummer:	3905977
Studiengang:	Master Informatik
Immatrikulationsjahr:	2015
Studienleistung:	Master-Arbeit
Beginn am:	6. März 2017
Einzureichen am:	14. August 2017
Verantw. Hochschullehrer:	Prof. Dr.-Ing. habil. Heiko Vogler

Semantische Repräsentationen natürlichsprachiger Sätze sind für viele Teilbereiche der Sprachverarbeitung interessant; beispielsweise können sie zur Verbesserung der Mensch-Computer-Interaktion, zur Informationsextraktion oder zur maschinellen Übersetzung genutzt werden. Um einen einheitlichen Rahmen für solche semantischen Repräsentationen zu schaffen, wurde von Banarescu u. a. [Ban+13] eine Darstellung als gerichteter Graph eingeführt: sogenannte *Abstract Meaning Representations* (AMRs).

Bedeutende Aufgabenstellungen im Umgang mit AMRs sind insbesondere das Erzeugen solcher AMRs aus natürlichsprachigen Sätzen (*Parsing*), sowie der umgekehrte Prozess, die Generierung natürlichsprachiger Sätze aus AMRs. Diese Generierung kann beispielsweise in der Mensch-Computer-Interaktion eingesetzt werden, um vorliegende Informationen in einen Satz zu transformieren. Außerdem kann die Kombination eines Parsers und eines Generators zur Übersetzung natürlichsprachiger Sätze verwendet werden [Jon+12].

Der AMR-Parser CAMR [Wan+15] nutzt ein *Transitionssystem*, um die Dependency Structure eines Satzes in einen AMR-Graphen zu konvertieren. Aufgrund der guten Resultate, die CAMR erzielt, ist es naheliegend, diese Idee versuchsweise auch auf die Generierung zu übertragen, also einen AMR-Graphen transitionsbasiert in eine – nicht zwangsläufig dem zugehörigen Dependency Tree gleiche – Baumstruktur zu überführen und das Yield dieser Baumstruktur als generierten Satz zu betrachten.

Aufgabe Es soll untersucht werden, ob durch eine "Umkehrung" der Transitionen im AMR-Parser *CAMR* ein Transition System zur Natural Language Generation aus AMR konstruiert werden kann.
Zuvor wird eine syntaktische Rekonstruktion durchgeführt; die dadurch erhaltenen syntaktischen Informationen werden genutzt, um Transitionen zu bewerten.

Syntaktische Rekonstruktion Es sollen folgende syntaktische Informationen, sofern möglich, rekonstruiert werden:

- POS-Tag (wird das Konzept als Substantiv, Adjektiv, Verb, Gerund, …realisiert?)
- für Verben: Genus Verbi (passiv / aktiv), Zeitform
- für Substantive: Numerus (singular / plural), Determination (the / a / -)

Die Rekonstruktion syntaktischer Informationen erfolgt top-down, um bereits rekonstruierte Informationen über Elternknoten mit einbeziehen zu können. Bei Uneindeutigkeit werden mehrere Ergebnisse in Schritt 2 berücksichtigt.

Transition System Für das Transition System werden mindestens folgende Klassen von Aktionen benötigt:

- DELETE-NODE `organization :name NATO` → `NATO`
- MERGE `good :degree more` → `better`
- SWAP `possible-01 :domain (see-01 :ARG0 he)` → `see-01 :domain-of possible-01 :ARG0 he`
- INSERT-PARENT `live :location Singapore` → `live :ins (in :location Singapore)`
- INSERT-CHILD `car` → `car :ins the`
- REALIZE-NODE `possible-01` → `can`
- REORDER-CHILDREN
- DELETE-REENTRANCE

Um gute Resultate zu erzielen, soll ein n-gram Language Model integriert werden. Daher erfolgt die Verarbeitung mittels Transition System bottom-up, sodass das Language Model immer auf die bereits erzeugten Teilsätze angewandt werden kann.

Post-Processing Weil das Resultat des Transition Systems ein Baum ist, sind immer noch Informationen über semantische Beziehungen vorhanden. Diese sowie das yield des Baums können genutzt werden, um die Realisierung einzelner Knoten (evtl. iterativ) zu verfeinern.

Form. Die Arbeit muss den üblichen Standards wie folgt genügen. Die Arbeit muss in sich abgeschlossen sein und alle nötigen Definitionen und Referenzen enthalten. Die Urheberschaft von Inhalten – auch die eigene – muss klar erkennbar sein. Fremde Inhalte, z.B. Algorithmen, Konstruktionen, Definitionen, Ideen, etc., müssen durch genaue Verweise auf die entsprechende Literatur kenntlich gemacht werden. Lange wörtliche Zitate sollen vermieden werden. Gegebenenfalls muss erläutert werden, inwieweit und zu welchem Zweck fremde Inhalte modifiziert wurden. Die Struktur der Arbeit muss klar erkenntlich sein, und der Leser soll gut durch die Arbeit geführt werden. Die Darstellung aller Begriffe und Verfahren soll mathematisch formal fundiert sein. Für jeden wichtigen Begriff sollen Erläuterungen und Beispiele angegeben werden, ebenso für die Abläufe der beschriebenen Verfahren. Wo es angemessen ist, sollen Illustrationen die Darstellung vervollständigen. Bei Diagrammen, die Phänomene von Experimenten beschreiben, muss deutlich erläutert werden, welche Werte auf den einzelnen Achsen aufgetragen sind, und beschrieben werden, welche Abhängigkeit unter den Werten der verschiedenen Achsen dargestellt ist.

Für die Implementierung soll eine ausführliche Dokumentation erfolgen, die sich angemessen auf den Quelltext und die schriftliche Ausarbeitung verteilt. Dabei muss die Funktionsfähigkeit des Programms glaubhaft gemacht und durch geeignete Beispielläufe dokumentiert werden.

Einer späteren Veröffentlichung der Implementierung unter einer Open-Source-Lizenz stimmt der Student zu. Der Student verpflichtet sich, ihm im Rahmen dieser Arbeit zugänglich gemachte Daten und Software (einschließlich Quellcode) lediglich zur Erledigung der Aufgaben zu verwenden und ansonsten vertraulich zu behandeln.

Dresden, 3. Februar 2017

Unterschrift von Heiko Vogler

Unterschrift von Timo Schick

Literatur

[Ban+13] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer und Nathan Schneider. „Abstract Meaning Representation for Sembanking“. In: *Proc. 7th Linguistic Annotation Workshop, ACL Workshop*. 2013.

[Jon+12] Bevan Jones, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann und Kevin Knight. „Semantics-Based Machine Translation with Hyperedge Replacement Grammars“. In: *Proc. 24th Intl. Conf. on Computational Linguistics (COLING 2012)*. 2012.

[Wan+15] Chuan Wang, Nianwen Xue, Sameer Pradhan und Sameer Pradhan. „A Transition-based Algorithm for AMR Parsing.“ In: *HLT-NAACL*. 2015, S. 366–375.

Abstract

This work addresses the task of generating English sentences from Abstract Meaning Representation (AMR) graphs. To cope with this task, we transform each input AMR graph into a structure similar to a dependency tree and annotate it with syntactic information by applying various predefined actions to it. Subsequently, a sentence is obtained from this tree structure by visiting its nodes in a specific order. We train maximum entropy models to estimate the probability of each individual action and devise an algorithm that efficiently approximates the best sequence of actions to be applied. Our generator achieves a Bleu score of 27.4 on the LDC2014T12 test set.

Contents

1 Introduction

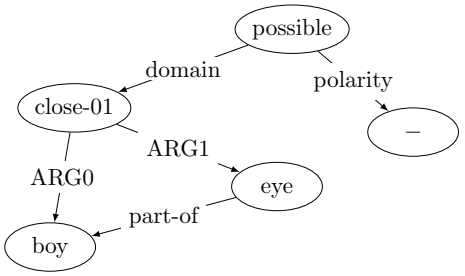
Semantic representations of natural language are of great interest for various aspects of natural language processing (NLP). For example, semantic representations may be useful for challenging tasks such as information extraction (Palmer et al., 2005), question answering (Shen and Lapata, 2007), natural language generation (Langkilde and Knight, 1998) and machine translation (Jones et al., 2012).

To provide a coherent framework for semantic representations, Banarescu et al. (2013) introduced *Abstract Meaning Representation* (AMR), a semantic representation language that encodes the meanings of natural language sentences as directed acyclic graphs with labels assigned to both vertices and edges. Within this formalism, vertices represent so-called *concepts* and edges encode *relations* between them. As AMR abstracts away various kinds of information, each graph typically corresponds to not just one, but a number of different sentences. An exemplary AMR graph can be seen in Figure 1a; several sentences corresponding to this graph are listed in Figure 1b. For AMR to be useful in solving the above-mentioned tasks, one must of course be able to convert sentences into AMR graphs and vice versa. Therefore, two important domain-specific problems are *(text-to-AMR) parsing*, the task of finding the graph corresponding to a given natural language sentence, and *(AMR-to-text) generation*, the inverse task of finding a good natural language realization for a given AMR graph. To give a simple example of how solutions to these tasks may be beneficial for NLP, a parser and a generator can easily be combined into a machine translation system (Jones et al., 2012).

While many approaches have been proposed for the text-to-AMR parsing task (see Flanigan et al., 2014; Peng et al., 2015; Pust et al., 2015; Wang et al., 2015; Puzikov et al., 2016; Zhou et al., 2016; Buys and Blunsom, 2017; van Noord and Bos, 2017; Konstas et al., 2017), the number of currently published AMR-to-text generators is comparably low (see Flanigan et al., 2016; Pourdamghani et al., 2016; Song et al., 2016, 2017; Konstas et al., 2017).

In this work, we tackle the problem of natural language generation from AMR by successively transforming input AMR graphs into structures that resemble dependency trees. To this end, we define a set of actions (*transitions*) such as the deletion, merging and swapping of edges and vertices. After applying these transitions to the input, we turn the obtained tree structure into a sentence by visiting its vertices in a specific order. We embed the different kinds of required actions into a *transition system*, a formal framework that, in the context of NLP, is often used for dependency parsing (see Nivre, 2008). To predict the correct sequence of transitions to be applied for each input, we train maximum entropy models (Berger et al., 1996) from a corpus of AMR graphs and corresponding realizations. As is done in all previous works on this topic, we restrict ourselves to generating English sentences; we do so simply because no reasonably large corpus for any other natural language is available to date. However, we are confident that our results can be transferred to many other languages with some effort.

Our transition-based approach is to a large extent inspired by the likewise transition-based parser CAMR (Wang et al., 2015). In fact, this parser may be seen as the direct inverse of our system: While we turn AMR graphs into ordered trees which, in turn, are



(a)

- It is not possible for the boy to close his eyes.
- The boy is unable to close his own eyes.
- The boys couldn't close their eyes.
- There was no possibility for the boy to close his eyes.

(b)

Figure 1: Visualization of an AMR graph and corresponding sentences

converted into sentences, the parser by Wang et al. (2015) generates dependency trees from sentences and subsequently transforms these trees into AMR graphs. Accordingly, several transitions used by CAMR have a direct counterpart in our generator.

In a way, the task performed by our system is simpler than its inverse. This is because we are not required to transform input AMR graphs into actual dependency trees; any tree is sufficient as long as the sentence obtained from it is a good realization of the input. For this very reason, there is also no need for us to assign dependency labels as they have no representation in the generated sentence. In other respects, however, the transformation from AMR graphs to suitable trees is much more challenging than going the opposite way. For example, we have to somehow cope with the fact that AMR graphs, in contrast to dependency trees, are unordered. Furthermore, AMR abstracts away tense, number and voice as well as function words such as articles, pronouns and prepositions; all this information must somehow be retrieved. Finally, the inclusion of a language model into our generation pipeline – which is indispensable to obtain competitive results – makes it very difficult to efficiently determine the best sequence of transitions for a given input.

We address these challenges in various ways. For instance, we devise a set of special transitions to establish an order on the vertices of our input. We try to compensate for lacking syntactic information by training several maximum entropy models to estimate this very information; this idea is formalized by introducing the concept of *syntactic annotations*. To actually implement our system, we develop a novel generation algorithm that incorporates a language model but is still sufficiently efficient.

We proceed as follows: After giving a succinct overview of previous work on AMR-to-text generation and related tasks in Section 2, we discuss basic notation and other preliminaries such as the AMR formalism, transition systems and maximum entropy models in Section 3. We introduce our generator in Section 4, which constitutes the core of this work. This section includes a detailed definition of all required transitions as well as a thorough derivation of our generation algorithm and an explanation of the required training procedure. In Section 5, we discuss our Java-based implementation of the generator. Results obtained with this implementation are reported in Section 6 where we also compare our generator with other approaches. We conclude with a concise summary of our work and an outlook on future research topics in Section 7.

2 Related Work

In this section, we give a short overview of previous work on AMR-related tasks, but we restrict ourselves to only such work that is closely related to the generation of natural language sentences from AMR. For a general introduction to AMR, we refer to Section 3.3 of this work and to Banarescu et al. (2013).

Alignments Both generation and parsing methods are often trained using an *AMR corpus*, a large set of AMR graphs and corresponding reference sentences. For such training procedures, it is useful to somehow link vertices of each AMR graph G to corresponding words of its reference sentence s. These links are commonly referred to as an *alignment*; several methods have been proposed for automatically generating such alignments.

The methods described by Jones et al. (2012) and Pourdamghani et al. (2014) both bijectively convert an AMR graph G into a string s_G through a simple breadth first search and depth first search, respectively.[1] Then, a string-to-string alignment between s_G and s is obtained using one of the models described in Brown et al. (1993); these models originate from the field of machine translation and are commonly referred to as *IBM Models*. The obtained alignment can then easily be converted into the desired format by retransforming s_G into G.

A fundamentally different approach is proposed by Flanigan et al. (2014), where a set of alignment rules is defined by hand; these rules are then greedily applied in a specified order.[2] An example of such a rule is the *Minus Polarity Tokens* rule, which aligns the words "no", "not" and "non" to vertices with the label "−"; this label is used in AMR to indicate negative polarity. The set of all rules used by this rule-based aligner can be found in Flanigan et al. (2014).

Parsing Many approaches for parsing English sentences into AMR graphs have been proposed. However, as the subject of this work is generation, we consider here only the transition-based parser CAMR introduced by Wang et al. (2015).[3] We consider this specific parser because several of its transitions are either equal or inverse to the transitions used by our generator. The idea behind CAMR is to make use of the fact that AMR graphs and dependency trees share some structural similarities. Therefore, given a sentence s, CAMR relies on some dependency parser to first generate the dependency tree D_s corresponding to s. Subsequently, several transitions are applied to D_s in order to successively turn it into the desired AMR graph G. These transitions include, for example, deleting and renaming both vertices and edges, swapping vertices or merging them into a single one as well as adding new edges. After each application of a transition, the transition to be applied next is determined using a linear classifier which, in turn, is trained with the aid of the alignment method described in Flanigan et al. (2014).

[1] The aligner by Pourdamghani et al. (2014) is available at `isi.edu/~damghani/papers/Aligner.zip`; the aligner by Jones et al. (2012) is not publicly available.

[2] The aligner by Flanigan et al. (2016) is available at `github.com/jflanigan/jamr`.

[3] The CAMR parser by Wang et al. (2015) is available at `github.com/c-amr/camr`.

Generation The first system for generating English strings from AMR graphs was published by Flanigan et al. (2016).[4] The core idea of this system is to convert AMR graphs into trees and to train a special kind of tree-to-string transducer (see Huang et al., 2006) on these trees. To obtain rules for the transducer, the greedy rule-based aligner of Flanigan et al. (2014) is used and several rule extraction mechanisms are tried out. An obvious problem with this approach is that the conversion of an AMR graph into a tree in general requires us to remove edges from it; the information encoded by these edges is therefore lost.

Song et al. (2016) treat AMR generation as a variant of the traveling salesman problem (TSP).[5] Input AMR graphs are first partitioned into several disjoint subgraphs and for each subgraph, a corresponding English phrase is determined using a set of rules extracted from a training set. Afterwards, an order among all subgraphs is specified. To this end, a *traveling cost* for visiting one subgraph after another is learned and the cost of each order is set to the sum of all traveling costs of adjacent subgraphs. For the final output, the order with the lowest score is determined using a TSP solver and the extracted phrases are concatenated in this very order.

The core idea of Pourdamghani et al. (2016) is to convert AMR graphs into strings, a process referred to as *linearization*, and then train a string-to-string translation model on the so-obtained pairs of linearized AMR graphs and corresponding sentences. For the linearization process, a simple depth first search is performed. However, since there is no order among vertices of an AMR graph, siblings can be visited in any order. As it may be helpful for the string-to-string translation model if the linearized AMR graph resembles English word order, a linear classifier is trained to decide for each pair of sibling vertices (v_1, v_2) whether v_1 should be visited before v_2 or vice versa. The actual string-to-string translation is then performed using a phrase-based model implemented in *Moses* (Koehn et al., 2007).

Another approach that requires AMR graphs to be linearized is proposed by Konstas et al. (2017). Their generator uses a sequence-to-sequence model built upon a *long short-term memory* (LSTM) neural network architecture. As this architecture requires a large set of training data to achieve good results, Konstas et al. (2017) use a text-to-AMR parser to automatically annotate millions of unlabeled sentences before training their system; the so-obtained AMR graphs are then used as additional training data.

Yet another approach is to tackle the problem of AMR generation using *synchronous node replacement grammars* (Song et al., 2017). A synchronous node replacement grammar is a rewriting formalism primarily defined by a set of rules that simultaneously produce graph fragments and phrases. Through repeated application of such rules, AMR graphs and corresponding sentences can be obtained; a sequence of rule applications is called a *derivation*. Given an AMR graph G, the approach of Song et al. (2017) is to assign scores to all possible derivations which produce G and to take the sentence produced by the highest-scoring such derivation as the output of the generator.

[4]The generator by Flanigan et al. (2016) is available at `github.com/jflanigan/jamr/tree/Generator`.
[5]The generator by Song et al. (2016) is available at `github.com/xiaochang13/AMR-generation`.

3 Preliminaries

3.1 Basic Notation

Set theory Let A and B be sets. We write $a \in A$ if an object a is an element of A. The cardinality of A is denoted by $|A|$. If A is a subset of B, we write $A \subseteq B$ and $A \subset B$ if $A \neq B$. The Cartesian product of A and B, their union, intersection and difference are written $A \times B$, $A \cup B$, $A \cap B$ and $A \setminus B$, respectively. For $n \in \mathbb{N}$, the n-fold Cartesian product of A with itself is written A^n. The power set of A is denoted by $\mathcal{P}(A)$. We denote the empty set as $\emptyset$, the set $\{0, 1, 2, \ldots\}$ of natural numbers as $\mathbb{N}$ and $\mathbb{N} \setminus \{0\}$ as $\mathbb{N}^+$. In an analogous manner, we write the set of integers as $\mathbb{Z}$, the set of real numbers as $\mathbb{R}$, the set of nonnegative reals as $\mathbb{R}_0^+$ and the set of positive reals as $\mathbb{R}^+$. For $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, 2, \ldots, n\}$ and $[n]_0$ denotes $[n] \cup \{0\}$.

Binary relations Let A, B and C be sets. A *binary relation between A and B* is a set $R \subseteq A \times B$. If $A = B$, we call R a *binary relation on A*. We sometimes denote $(a, b) \in R$ as $a\,R\,b$. The *inverse* of a relation $R \subseteq A \times B$, denoted by R^{-1}, is the relation $\{(b, a) \mid (a, b) \in R\} \subseteq B \times A$. The *domain of R* is the set $\mathrm{dom}(R) = \{a \in A \mid \exists b \in B : (a, b) \in R\}$. For relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, their *composition* is defined as

$$R_1 R_2 = \{(a, c) \in A \times C \mid \exists b \in B : (a, b) \in R_1 \wedge (b, c) \in R_2\}.$$

In the following, let R be a binary relation on A and let $A' \subseteq A$. R is called *irreflexive* if for all $a \in A$, $(a, a) \notin R$ and *transitive* if for all $a, b, c \in A$, $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$. The *transitive closure* of R, denoted by R^+, is the smallest relation on A such that $R \subseteq R^+$ and R^+ is transitive. We call a relation that is both irreflexive and transitive a *strict order*. R is a *total order on A'* if R is a strict order and for all $a, b \in A'$, $(a, b) \in R$ or $(b, a) \in R$. If A' is a finite set with n elements and R is a total order on A', the *A'-sequence induced by R* is the uniquely determined sequence $(a_1, \ldots, a_n)$ where for all $i \in [n-1]$, $(a_i, a_{i+1}) \in R \cap A' \times A'$.

Functions Let A, B and C be sets. We call a binary relation f between A and B a *partial function from A to B* and write $f : A \nrightarrow B$ if for all $a \in A$, there is at most one $b \in B$ such that $(a, b) \in f$; we also denote b by $f(a)$. If $\mathrm{dom}(f) = A$, we call f a *(total) function* and write $f : A \to B$. We call $f : A \to B$ a *bijective* function or *bijection* if for all $b \in B$, there is exactly one $a \in A$ such that $f(a) = b$. For $f : A \nrightarrow B$, $a \in A$ and $b \in B$, the function $f[a \mapsto b] : \mathrm{dom}(f) \cup \{a\} \to B$ is defined by

$$f[a \mapsto b](x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases}$$

for all $x \in \mathrm{dom}(f) \cup \{a\}$. Let $f : A \nrightarrow B$, $a_1, \ldots, a_n \in A$, $b_1, \ldots, b_n \in B$, $n \in \mathbb{N}$. We write $f[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ as a shorthand for $(\ldots(f[a_1 \mapsto b_1])\ldots)[a_n \mapsto b_n]$. For $f : A \nrightarrow (B \nrightarrow C)$, $a_1, \ldots, a_n \in A$, $b_1, \ldots, b_n \in B$, $c_1, \ldots, c_n \in C$, we write

$$f[a_1(b_1) \mapsto c_1, \ldots, a_n(b_n) \mapsto c_n]$$

as a shorthand for $f[a_1 \mapsto f(a_1)[b_1 \mapsto c_1], \ldots, a_n \mapsto f(a_n)[b_n \mapsto c_n]]$.

For $g\colon A \to \mathbb{R}$ and $\mathrm{op} \in \{\min, \max\}$, $\arg\mathrm{op}_{x \in A}\, g(x)$ usually denotes the set

$$S_{\mathrm{op}} = \{x \in A \mid \nexists x' \in A\colon g(x') \lozenge g(x)\} \text{ where } \lozenge = \begin{cases} > & \text{if op} = \max \\ < & \text{if op} = \min\,. \end{cases}$$

However, we are often just interested in one arbitrary $x \in S_{\mathrm{op}}$. We therefore identify $\arg\mathrm{op}_{x \in A}\, g(x)$ with some element of the set S_{op} for the rest of this work.

Formal languages An *alphabet* Σ is a nonempty set of distinguishable symbols.[6] A *string* over Σ is a finite sequence of symbols from Σ; Σ^* denotes the set of all such strings. The concatenation of two strings $a, b \in \Sigma^*$ is written $a \cdot b$ or ab. We abbreviate the n-fold concatenation of the same symbol $a \in \Sigma$ by a^n. Let $w = (w_1, \ldots, w_n)$ be a string over some alphabet Σ with $w_i \in \Sigma$ for all $i \in [n]$. We denote w_i also by $w(i)$. We sometimes write $w_1 \ldots w_n$ as an abbreviation for $(w_1, \ldots, w_n)$. If we are only interested in the first $m \leq n$ symbols of w, we also denote w as $w_1{:}w_2{:}\ldots{:}w_m{:}w'$ with $w' = (w_{m+1}, \ldots, w_m)$. The length of w is written $|w|$, ε denotes the empty string. For $\Sigma' \subseteq \Sigma$, we define $w \setminus \Sigma'$ to be the sequence $w'_1 \cdot \ldots \cdot w'_n$ with

$$w'_i = \begin{cases} w_i & \text{if } w_i \notin \Sigma' \\ \varepsilon & \text{otherwise} \end{cases}$$

for all $i \in [n]$, i.e. $w \setminus \Sigma'$ is obtained from w by removing from it all $w_i \in \Sigma'$.

An alphabet frequently used throughout this work is the set of all English words, hereafter denoted by Σ_{E}. We define Σ_{E} to contain not only all English words and word forms, but also punctuation marks, numbers and special characters. Notwithstanding the above definitions, we always separate symbols from Σ_{E} by spaces. That is, we write "the house" rather than "(the, house)" or "the $\cdot$ house".

Probability theory Let Ω be a countable set. A *probability measure on* Ω is a function $P\colon \mathcal{P}(\Omega) \to [0, 1]$ such that $P(\Omega) = 1$ and

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)$$

for every countable sequence $A_1, A_2, \ldots$ of pairwise disjoint sets $A_i \subseteq \Omega$ (i.e. $A_i \cap A_j = \emptyset$ for all $i, j \in \mathbb{N}$ with $i \neq j$). For $\omega \in \Omega$ and $A, B \subseteq \Omega$, we abbreviate $P(\{\omega\})$ by $P(\omega)$ and $P(A \cap B)$ by $P(A, B)$.

Let $A, B \subseteq \Omega$. For $P(B) \neq 0$, the *conditional probability of A given B* is defined as

$$P(A \mid B) = P(A, B) \cdot P(B)^{-1}\,.$$

For some $C \subseteq \Omega$ with $P(C) \neq 0$, we say that A and B are *conditionally independent given C* if $P(A, B \mid C) = P(A \mid C) \cdot P(B \mid C)$. Let $n \in \mathbb{N}$, $A_i \subseteq \Omega$ for $i \in [n]$ and

[6] While alphabets are commonly defined as *finite* sets, we explicitly allow them to be of infinite size.

$(B_i \mid i \in I)$ be a countable partition of Ω. We will make frequent use of the following two identities:

$$P(A_1, \ldots, A_n) = P(A_1, \ldots, A_{n-1}) \cdot P(A_n \mid A_1, \ldots, A_{n-1}) \qquad \text{(General product rule)}$$
$$P(A) = \sum\nolimits_{i \in I} P(A, B_i) \qquad \text{(Law of total probability)}$$

Let X be a countable set. A *random variable* is a function $\mathbb{X}\colon \Omega \to X$. For $x \in X$, we use $\mathbb{X} = x$ as an abbreviation for the set $\{\omega \in \Omega \mid \mathbb{X}(\omega) = x\}$. Thus,

$$P(\mathbb{X} = x) = \sum_{\omega \in \Omega\colon\, \mathbb{X}(\omega) = x} P(\omega)\,.$$

Throughout this work, we drop random variables $\mathbb{X}$ from our notation whenever they are clear from the context, i.e. we simply write $P(x)$ instead of $P(\mathbb{X} = x)$.

Let X and Y be countable sets. A *probability distribution of* X is a function $p\colon X \to [0,1]$ such that $\sum_{x \in X} p(x) = 1$. A *conditional probability distribution of* X *given* Y is a function $q\colon Y \to (X \to [0,1])$ such that for all $y \in Y$, $\sum_{x \in X} q(z)(x) = 1$. We denote $q(z)(x)$ also by $q(x \mid z)$.

3.2 Labeled Ordered Graphs

Definition 3.1 (Labeled ordered graph) Let L_E and L_V be two sets (*edge labels* and *vertex labels*). A *(labeled ordered)* (L_E, L_V)*-graph* is a tuple $G = (V, E, L, \prec)$ where $V \neq \emptyset$ is a finite set of *vertices* (or *nodes*), $E \subseteq V \times L_E \times V$ is a finite set of labeled *edges*, $L : V \to L_V$ is a *vertex labeling* and $\prec \subseteq V \times V$ is a strict order. △

If we are not interested in the particular sets of edge and vertex labels, we refer to a (L_E, L_V)-graph simply as *graph*. In the following, let $G = (V, E, L, \prec)$ be a graph. For each $v \in V$, $L(v)$ is called the *label of* v and for each $e = (v_1, l, v_2) \in E$, l is called the *label of* e. We define a *walk in* G to be a sequence of vertices $w = (v_0, \ldots, v_n)$, $n \in \mathbb{N}^+$ such that for all $i \in [n]$, there is some $l_i \in L_E$ with $(v_{i-1}, l_i, v_i) \in E$. A *cycle* is a walk $(v_0, \ldots, v_n)$ where $v_0 = v_n$ and $v_i \neq v_j$ for all other $i, j \in [n]_0$ with $i \neq j$. We call G *cyclic* if it contains at least one cycle and *acyclic* otherwise. For each node $v \in V$, we denote by

$$\mathrm{in}_G(v) = \{e \in E \mid \exists v' \in V, l \in L_E : e = (v', l, v)\}$$
$$\mathrm{out}_G(v) = \{e \in E \mid \exists v' \in V, l \in L_E : e = (v, l, v')\}$$

the set of its *incoming edges* and *outgoing edges*, respectively. Correspondingly,

$$\mathrm{pa}_G(v) = \{v' \in V \mid \exists l \in L_E : (v', l, v) \in E\}$$
$$\mathrm{ch}_G(v) = \{v' \in V \mid \exists l \in L_E : (v, l, v') \in E\}$$

denote the set of v's *parents* and *children*. If G is acyclic, the sets of *successors* and *predecessors* of v are defined recursively as

$$\mathrm{succ}_G(v) = \mathrm{ch}_G(v) \cup \bigcup_{v' \in \mathrm{ch}_G(v)} \mathrm{succ}_G(v') \qquad \mathrm{pred}_G(v) = \mathrm{pa}_G(v) \cup \bigcup_{v' \in \mathrm{pa}_G(v)} \mathrm{pred}_G(v')\,.$$

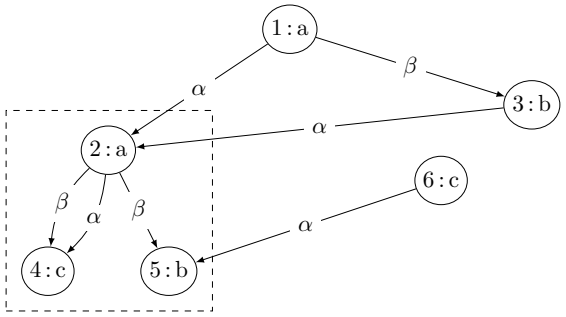


Figure 2: Graphical representation of the graph $G_0 = (V_0, E_0, L_0, \prec_0)$ as described in Example 3.3. Each node $v \in V_0$ is inscribed with $v : L_0(v)$. $G_0|_2$ is framed by dashed lines.

From the above notations, we sometimes drop the subscript if the corresponding graph is clear from the context; for example, we often simply write $\mathrm{pa}(v)$ and $\mathrm{ch}(v)$ instead of $\mathrm{pa}_G(v)$ and $\mathrm{ch}_G(v)$. We call $v \in V$ a *root of* G if $\mathrm{pa}_G(v) = \emptyset$. If V contains exactly one root, G is called a *rooted* graph; we denote this vertex by $\mathrm{root}(G)$. G is called a *tree* if it is rooted, acyclic and $|\mathrm{in}_G(v)| = 1$ for all $v \in V \setminus \{\mathrm{root}(G)\}$. We say that G is *totally ordered* if for all $v \in V$, $\prec$ is a total order on $\mathrm{ch}_G(v) \cup \{v\}$.

Throughout this work, we often represent a graph $G = (V, E, L, \prec)$ graphically. In such a visualization, each vertex $v \in V$ is represented by an ellipse inscribed either with $L(v)$ or $v : L(v)$. Each edge $(v_1, l, v_2) \in E$ is represented by an arrow line connecting the graphical representations of v_1 and v_2; this line is inscribed with l. We do not depict the order $\prec$ in this visualization, but whenever $\prec$ is of relevance, we explicitly specify it.

Definition 3.2 (v-Subgraph) Let $G = (V, E, L, \prec)$ be an acyclic graph. For $v \in V$, the *v-subgraph of G*, denoted by $G|_v$, is the graph $(V', E', L', \prec')$ where

$$V' = \mathrm{succ}(v) \cup \{v\} \qquad E' = \{(v_1, l, v_2) \in E \mid v_1, v_2 \in V'\}$$
$$L' = \{(v, l) \in L \mid v \in V'\} \qquad \prec' = \{(v_1, v_2) \in \prec \mid v_1, v_2 \in V'\}. \qquad \triangle$$

Example 3.3 Let $L_E = \{\alpha, \beta\}$ be a set of edge labels and $L_V = \{a, b, c\}$ be a set of vertex labels. The (L_E, L_V)-graph $G_0 = (V_0, E_0, L_0, \prec_0)$ where

$$V_0 = \{1, 2, 3, 4, 5, 6\}$$
$$E_0 = \{(1, \alpha, 2), (1, \beta, 3), (3, \alpha, 2), (2, \alpha, 4), (2, \beta, 4), (2, \beta, 5), (6, \alpha, 5)\}$$
$$L_0 = \{(1, a), (2, a), (3, b), (4, c), (5, b), (6, c)\}$$
$$\prec_0 = \{(v_1, v_2) \in V_0 \times V_0 \mid v_1 <_{\mathbb{N}} v_2\}$$

is acyclic and totally ordered, but not rooted. The 2-subgraph of G_0 is the rooted graph $G_0|_2 = (\{2, 4, 5\}, \{(2, \alpha, 4), (2, \beta, 4), (2, \beta, 5)\}, \{(2, a), (4, c), (5, b)\}, \{(2, 4), (2, 5), (4, 5)\})$. A graphical representation of both G_0 and $G_0|_2$ can be found in Figure 2. $\triangle$

Definition 3.4 (Yield) Let $G = (V, E, L, \prec)$ be an acyclic and totally ordered graph. Furthermore, let Σ be an alphabet, V' be a set with $V \subseteq V'$ and $\rho : V' \to \Sigma^*$. The function $\text{yield}_{(G,\rho)} : V \to \Sigma^*$ is defined for each $v \in V$ as

$$\text{yield}_{(G,\rho)}(v) := \text{yield}_{(G,\rho)}(c_1) \cdot \ldots \cdot \text{yield}_{(G,\rho)}(c_k) \cdot \rho(v) \cdot \text{yield}_{(G,\rho)}(c_{k+1}) \cdot \ldots \cdot \text{yield}_{(G,\rho)}(c_{|\text{ch}(v)|})$$

where $(c_1, \ldots, c_k, v, c_{k+1}, \ldots, c_{|\text{ch}(v)|})$, $k \in [|\text{ch}(v)|]_0$ is the $(\text{ch}(v) \cup \{v\})$-sequence induced by $\prec$. If G is rooted, we write $\text{yield}_\rho(G)$ as a shorthand for $\text{yield}_{(G,\rho)}(\text{root}(G))$. △

Let $G = (V, E, L, \prec)$ and ρ be defined as above. We observe that for all $u, v, w \in V$, if u is a successor of v and the term $\rho(w)$ occurs in $\text{yield}_\rho(G)$ between the terms $\rho(u)$ and $\rho(v)$, then w must also be a successor of v; in analogy to a similar property studied in the context of dependency trees (see Nivre, 2008), we refer to this property of yield as *projectivity*.

Example 3.5 Let $\Sigma_0 = \{x, y, z\}$ and let $\rho_0 = \{(1, x), (2, y), (3, x), (4, z), (5, x), (6, y)\}$. We consider the graph $G_0 = (V_0, E_0, L_0, \prec_0)$ defined in Example 3.3. All of the following statements are true:

$$\begin{aligned}
&\text{yield}_{(G_0,\rho_0)}(2) = \rho_0(2) \cdot \rho_0(4) \cdot \rho_0(5) = yzx \\
&\text{yield}_{(G_0,\rho_0)}(3) = \text{yield}_{(G_0,\rho_0)}(2) \cdot \rho_0(3) = yzx \cdot x \\
&\text{yield}_{(G_0,\rho_0)}(1) = \rho_0(1) \cdot \text{yield}_{(G_0,\rho_0)}(2) \cdot \text{yield}_{(G_0,\rho_0)}(3) = x \cdot yzx \cdot yzxx \\
&\text{yield}_{(G_0,L_0)}(6) = L_0(5) \cdot L_0(6) = bc\,.
\end{aligned}$$ △

Definition 3.6 (Bottom-up traversal) Let $G = (V, E, L, \prec)$ be an acyclic graph. We call a sequence of vertices $s \in V^*$ a *bottom-up traversal of* G if there is some total order $\lessdot$ on V such that for all $v \in V$ and $v' \in \text{ch}_G(v)$ it holds that $v' \lessdot v$ and s is the V-sequence induced by $\lessdot$. △

Example 3.7 We consider once more the graph $G_0 = (V_0, E_0, L_0, \prec_0)$ defined in Example 3.3. The sequences

$$s_1 = (4, 5, 6, 2, 3, 1) \quad s_2 = (4, 5, 2, 3, 1, 6) \quad s_3 = (5, 4, 2, 6, 3, 1)$$

are bottom-up traversals of G_0. In contrast, $(4, 5, 6, 3, 2, 1)$ is not a bottom-up traversal of G_0 because the corresponding order $\lessdot = \{(4, 5), (5, 6), (6, 3), (3, 2), (2, 1)\}^+$ does not contain the tuple $(2, 3)$ although $2 \in \text{ch}_{G_0}(3)$. △

3.3 Abstract Meaning Representation

Abstract Meaning Representation (AMR) is a semantic representation language that encodes the meaning of a sentence as a rooted, acyclic graph (Banarescu et al., 2013). To this end, AMR makes use of *PropBank framesets* (Kingsbury and Palmer, 2002; Palmer et al., 2005). A PropBank frameset mainly consists of

1. a *frameset id* ("want-01", "see-01", "develop-02", ...) which in turn consists of a verb and a number; the latter is used to differentiate between several meanings of the same verb and also referred to as the *sense tag* of the frameset id;

want-01	**sleep-01**	**develop-02**
ARG0: wanter	ARG0: sleeper	ARG0: creator
ARG1: thing wanted	ARG1: cognate object	ARG1: thing created
ARG2: beneficiary		ARG2: source
ARG3: in-exchange-for		ARG3: benefactive
ARG4: from		

Table 1: PropBank framesets corresponding to the concepts *want-01*, *sleep-01* and *develop-02*, extracted from `propbank.github.io`. For each frameset, the specific meanings of the corresponding semantic roles are briefly described.

2. a list of associated *semantic roles* (ARG0 – ARG5). These roles have no intrinsic meaning but are defined on a verb-by-verb basis; for many verbs, only some semantic roles are defined. The meanings of all semantic roles specified for the frameset ids "want-01", "see-01" and "develop-02" can be seen in Table 1.

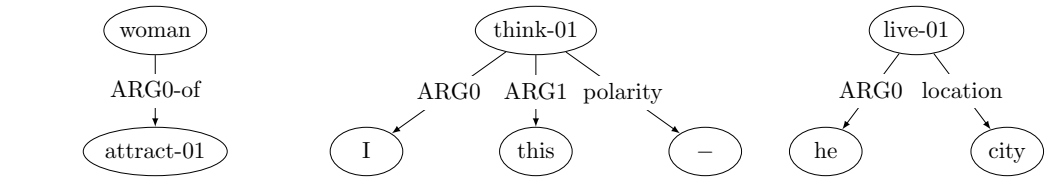
The key components of an AMR graph are *concepts*, represented by the set of possible vertex labels, *instances* of these concepts, represented by actual vertices, and *relations* between these instances, represented by edges. For example, an edge $e = (v_0, \text{ARG0}, v_1)$ connecting two nodes v_0 and v_1 with labels "sleep-01" and "boy", respectively, would indicate that an instance of the concept "boy", i.e. an actual boy, is the zeroth argument of an instance of the frameset "sleep-01", or in other words, he is the person who is sleeping. A simple graph consisting only of the nodes v_0 and v_1 and the edge e can thus be seen as a semantic representation of the phrase "a boy sleeps".

The set of all AMR concepts, hereafter denoted by L_{C}, consists of English words, numbers, names, PropBank framesets and so-called *special keywords*. The latter include logical conjunctions ("and", "or", ...), grammatical mood indicators ("interrogative", "imperative", ...), polarity ("−", "+"), quantities ("monetary-quantity", "distance-quantity", ...) and special entity types ("rate-entity", "date-entity", ...). For further details on the meaning of these keywords and a complete list thereof, we refer to AMR Specification 1.2.2.[7]

Following Banarescu et al. (2013), we can roughly divide the set of possible relation labels, hereafter denoted by L_{R}, into five categories:

1. PropBank semantic roles (ARG0 – ARG5), also referred to as *core roles*;
2. General semantic relations (location, cause, purpose, manner, topic, time, duration, direction, instrument, accompanier, age, frequency, name, ...);
3. Relations for quantities (quant, unit, scale, ...);
4. Relations for date-entities (day, month, year, weekday, century, era, quarter, season, timezone, ...);
5. Relations for enumerations and listings (OPi, $i \in \mathbb{N}$).

[7] AMR Specification 1.2.2 can be found at `amr.isi.edu/language.html`.



an attractive woman
the attractive women
there is an attractive woman
the woman who attracts

(a)

this is not what I think
this is not my thought
this was not a thought of mine
these were not my thoughts

(b)

he lives in a city
he is living in the city
his life in the city
he lived in the city

(c)

Figure 3: Graphical representation of three exemplary AMR graphs; each vertex is inscribed with its label. Below each AMR graph, some of its realizations are shown.

For each relation r from this list, the corresponding *inverse relation*, denoted by r-of, is also included in L_{R}; it is sometimes necessary to exchange a relation by its inverse in order to make the corresponding AMR graph rooted. We define for all $r \in L_{\mathrm{R}}$:

$$r^{-1} = \begin{cases} r' & \text{if } r = r'\text{-of for some } r' \in L_{\mathrm{R}} \\ r\text{-of} & \text{otherwise.} \end{cases}$$

To give an example, $\mathrm{ARG0}^{-1}$ equals ARG0-of and purpose-of^{-1} equals purpose. For a complete list of all possible relation labels, we again refer to AMR Specification 1.2.2.

Definition 3.8 (AMR graph) An AMR graph is a rooted, acyclic $(L_{\mathrm{R}}, L_{\mathrm{C}})$-graph $G = (V, E, L, \prec)$ with $\prec = \emptyset$.[8] The set of all AMR graphs is denoted by $\mathcal{G}_{\mathrm{AMR}}$. △

Given an AMR graph G, we call every sentence whose meaning is represented by G a *realization of* G. An important goal of AMR is to assign the same graph to semantically equal sentences, even if they differ syntactically. To this end, words are mapped to PropBank framesets whenever possible; this applies not only to verbs, but also to other parts of speech (POS) such as nouns and adjectives. Examples of this are shown in the three AMR graphs depicted in Figure 3 where the words "attractive", "thought" and "life" are represented by the framesets "attract-01", "think-01" and "live-01", respectively.

Parts of speech are by no means the only information that is not represented in AMR graphs. As can be seen in Figure 3c, prepositions such as "in", "to" and "for" have no direct representation in AMR but are instead encoded through relation labels such as "location", "direction" and "purpose". Other limitations of AMR include that in general, neither definiteness nor grammatical number (see Figure 3a) nor tense (Figure 3b and 3c) of a sentence can directly be represented by its AMR graph. However, it is possible to explicitly include some of this information through special relations and concepts. To give an example, the grammatical number of a noun may be indicated by using the

[8] Note that this definition differs slightly from the format introduced by Banarescu et al. (2013) where only leaf nodes have labels assigned.

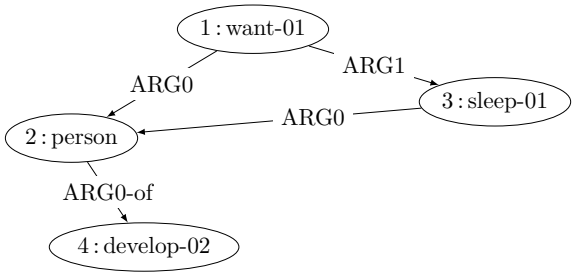


Figure 4: Graphical representation of the AMR graph G_1 introduced in Example 3.9

relation "quant" in combination with either a numerical value or an English word like "many", "few" or "some".

Example 3.9 The meaning of the sentence "The developer wants to sleep" can be represented by the AMR graph $G_1 = (\{1, 2, 3, 4\}, E_1, L_1, \emptyset)$ with

$$E_1 = \{(1, \text{ARG0}, 2), (1, \text{ARG1}, 3), (3, \text{ARG0}, 2), (2, \text{ARG0-of}, 4)\}$$
$$L_1 = \{(1, \text{want-01}), (2, \text{person}), (3, \text{sleep-01}), (4, \text{develop-02})\}.$$

A graphical representation of G_1 can be seen in Figure 4. The required PropBank framesets along with their roles are shown in Table 1. Note that the noun "developer" is represented by a combination of the English word "person" and the PropBank frameset "develop-02". Unlike the examples shown in Figure 3, G_1 is not a tree as the node labeled "person" is the zeroth argument to instances of both "want-01" and "sleep-01". △

3.3.1 Generation and Parsing

Common tasks with regard to AMR involve *parsing*, the problem of finding the AMR graph corresponding to a sentence, and the inverse problem of *generation*, i.e. finding a good natural-language realization of a given AMR graph.

Definition 3.10 (Generator) A function $g\colon \mathcal{G}_{\text{AMR}} \to \Sigma_{\text{E}}^*$ is called a *generator*. Given a generator g and an AMR graph $G \in \mathcal{G}_{\text{AMR}}$, we call $g(G)$ the *sentence generated from G by g* or the *realization of G according to g*. △

Definition 3.11 (Parser) A function $p\colon \Sigma_{\text{E}}^* \to \mathcal{G}_{\text{AMR}}$ is called a *parser*. Given a parser p and a sentence $w \in \Sigma_{\text{E}}^*$, we call $p(w)$ the *parse of w according to p*. △

While according to the above definition, any function that maps English sentences to AMR graphs is called a parser, one would ideally like to find a parser that assigns to each English sentence w the AMR graph $\hat{G}$ that best represents its meaning. As determining this unique AMR graph given an English sentence is an exceedingly difficult task, one is also interested in finding parsers that assign to each sentence w an AMR graph G that is at least roughly equal to $\hat{G}$. In order to be able to evaluate the quality of a parser, Cai and Knight (2013) define the *semantic match* (Smatch) metric which, given one or more

pairs of graphs $(\hat{G}_i, G_i)$, $i \in [n]$ for some $n \in \mathbb{N}$, measures how similar all related graphs $\hat{G}_i$ and G_i are and aggregates these similarity values to a cumulative score ranging from 0 to 1. Given a sequence $C = (G_1, w_1), \ldots, (G_n, w_n)$ of AMR graphs and corresponding sentences, Smatch can be used to automatically compare AMR parsers by calculating

$$\text{score}(p) = \text{Smatch}((G_1, p(w_1)), \ldots, (G_n, p(w_n)))$$

for each parser p and comparing the scores of all parsers. Details on how exactly the Smatch score can be calculated are beyond the scope of this work; we refer to Cai and Knight (2013) for an in-depth explanation.

Of course, the very same need for an evaluation metric arises when dealing with generation from AMR graphs: We require some way to measure the quality of generators in order to make comparisons between them. However, it is considerably more complex to evaluate a generator than a parser because given an AMR graph G, there is not necessarily just a single sentence $\hat{w}$ that corresponds to G; as the examples in Figure 3 show, there may be several equally good realizations of G.

The most common approach to the problem of evaluating generators is to make use of the *bilingual evaluation understudy* (Bleu) score (Papineni et al., 2002) that originates from the field of machine translation. Given a *candidate sentence* w and a *reference sentence* $\hat{w}$, the basic idea of Bleu is to count the number of matching n-grams (i.e. contiguous phrases consisting of n words) between w and $\hat{w}$.[9] This number is then divided by the total number of n-grams in the candidate sentence w. Typically, this computation is done not just for one but for several values of n and the results are averaged subsequently; a common choice is $n = 1, \ldots, 4$. Some modifications such as clipping the count of candidate n-gram matches must be made in order to make the resulting score more meaningful; we will, however, not discuss these modifications here and refer to Papineni et al. (2002) for further details.

Just as Smatch, Bleu can be extended to compute a cumulative score ranging from 0 to 1 and measuring the pairwise similarity of each sentence pair $(\hat{w}_i, w_i)$, $i \in [n]$ contained within a sequence of $n \in \mathbb{N}$ sentence pairs. This allows us to compare a set of generators given a sequence $C = (G_1, w_1), \ldots, (G_n, w_n)$ of AMR graphs $G_i \in \mathcal{G}_{\text{AMR}}$ and corresponding realizations $w_i \in \Sigma_{\text{E}}^*$ by calculating

$$\text{score}(g) = \text{Bleu}((w_1, g(G_1)), \ldots, (w_n, G(w_n)))$$

for each generator g. A common modification to the above definition of Bleu is to scale the result by some factor $s \in \mathbb{N}^+$, resulting in the total score ranging from 0 to s; the usual choice in the context of AMR generation is $s = 100$. Also, w_i and $g(G_i)$ are often not directly used to compute the Bleu score but are converted to lower case beforehand. We refer to the so-obtained score as the *case insensitive Bleu score*.

Especially in the scenario of AMR generation where given a graph G, there are often many – and equally good – realizations that may differ significantly with regards to

[9] The Bleu score is actually designed to support several reference sentences $\hat{w}_1, \ldots, \hat{w}_k$. While this might sound useful to our application scenario, all currently published AMR corpora unfortunately feature only a single realization per graph (see Section 3.3.2).

the choice of words and syntactic structure, even scores well below the maximum do not necessarily imply that a generator performs poorly. Consider, for example, the lowercased sentence pair

$$\hat{w} = \text{the boys couldn't close their eyes}$$
$$w = \text{it is not possible for the boy to close his eyes}$$

where $\hat{w}$ serves as a reference sentence and w is the output of a generator. Although both sentences are equally good realizations of the AMR graph shown in Figure 1a, they have only three common unigrams ("the", "close", "eyes") and not a single common n-gram for $n \in \{2, 3, 4\}$, resulting in a very low score. As this example demonstrates, the Bleu score of a single generator would scarcely be meaningful. Nevertheless, it is an established baseline for relative judgments in comparison with other generators.

3.3.2 Corpora

As we have seen in the previous section, the evaluation of parsers and generators using Smatch or Bleu requires a sequence of AMR graphs along with reference realizations; we refer to such a sequence as an *AMR corpus*.

Definition 3.12 (AMR corpus) A sequence $C = ((G_1, w_1), \ldots, (G_n, w_n))$, $n \in \mathbb{N}$ where $G_i \in \mathcal{G}_{\text{AMR}}$ and $w_i \in \Sigma_{\text{E}}^*$ for all $i \in [n]$ is called an *AMR corpus*. We refer to n as the *size of* C and to each tuple (G_i, w_i), $i \in [n]$ as an *element of* C. △

We often refer to an AMR corpus simply as *corpus*. Of course, AMR corpora are not only useful for evaluation of parsers and generators, but as well for training them. However, it is essential to not use the same data for both training and evaluation because obviously, we want a generator to perform well not only for inputs that it has already seen during training, but also for previously unknown graphs. Therefore, corpora are usually divided into several disjoint subcorpora: a sequence of *training data* used to train the parser or generator, a sequence of *development data* used e.g. for hyperparameter optimization, and a sequence of *test data* on which the quality of the chosen approach can be evaluated.

As AMR is a relatively new research topic, both the number of corpora and the number of graphs contained within these corpora is rather low compared to the number of available data for syntactic annotations like constituency trees and dependency trees. Importantly, all currently released AMR corpora consist only of AMR graphs with exactly one reference sentence per graph. Also, there is no information included with regards to how vertices and edges of the contained AMR graphs correspond to words of their realizations, i.e. no alignment between graphs and reference sentences is given.

An overview of some AMR corpora is given in Table 2. As its name suggests, the corpus *The Little Price* contains AMR graphs encoding the meaning of each sentence in the novel of the same name by Antoine de Saint-Exupéry. The Bio AMR corpus consists mostly of semantic annotations for cancer-related research papers. Both corpora released by the *Linguistic Data Consortium* (LDC), LDC2014T12 and LDC2015E86, contain

Corpus	Total Size	Size (Train / Dev / Test)	Availability
The Little Prince v1.6	1,562	1,274 / 145 / 142	general release[a]
Bio AMR v0.8	6,452	5,452 / 500 / 500	general release[a]
LDC2014T12	13,051	10,313 / 1,368 / 1,371	general release[b]
LDC2015E86	19,572	16,833 / 1,368 / 1,371	not publicly available[c]

Table 2: Overview of currently released AMR corpora. For each corpus, the total number of contained AMR graphs is listed along with the sizes of the training, development and test sets.

[a]The general releases of both The Little Prince v1.6 and Bio AMR v0.8 are available at `amr.isi.edu/download.html`.
[b]The general release of LDC2014T12 is available at `catalog.ldc.upenn.edu/LDC2014T12`.
[c]The release of LDC2015E86 is limited to participants of *Deep Exploration and Filtering of Text* (DEFT).

AMR graphs for English sentences obtained from various newswires, discussion forums and television transcripts.[10] The latter corpus is an extension of the former, containing the same development and test data but several additional AMR graphs for training.

3.4 Dependency Trees

An established way to model the syntactic structure of a sentence is through so-called *dependencies* between its words (Tesnière, 1959; Nivre, 2008). A dependency consists of a *head*, a *dependent* and a *relation* between them. While both the head and the dependent of a dependency are simply words of the analyzed sentence, their relation is usually described by a label taken from some set L_{D} of *dependency labels*.[11] To give an example, consider once more the sentence "The developer wants to sleep". The fact that "developer" is the nominal subject corresponding to the verb "wants" can be modeled through a dependency with head "wants", dependent "developer" and label "nsubj".

The main verb of a sentence is typically chosen to be its head, i.e. it is the only word that is not a dependent of any other word. As dependency relations are asymmetric and every word is the dependent of at most one head, the set of all dependencies within a sentence w can be viewed as a tree whose nodes correspond to the sentence's words and whose root is the main verb of w.

Definition 3.13 (Dependency tree) A $(L_{\mathrm{D}}, \Sigma_{\mathrm{E}})$-graph $G = (V, E, L, \prec)$ is called a *dependency tree* if it is a totally ordered tree. The set of all dependency trees is denoted by $\mathcal{G}_{\mathrm{DEP}}$. △

Let $w \in \Sigma_{\mathrm{E}}^*$ be a sentence and $G = (V, E, L, \prec)$ be a dependency tree. We call G a *dependency tree for* w if there is some bijection $b\colon V \to [|w|]$ such that for all $v, v' \in V$ and $i \in [|w|]$, it holds that $b(v) = i \Rightarrow L(v) = w(i)$ and $v \prec v' \Leftrightarrow b(v) < b(v')$.

[10]Further details on the genres and contents of the listed corpora can be found at `amr.isi.edu/download.html`.
[11]A list of all dependency labels used throughout this work along with their meanings can be found at `universaldependencies.org/u/dep`.

Example 3.14 We consider the graph $G_2 = (\{1,2,3,4,5\}, E_2, L_2, \prec_2)$ where

$$E_2 = \{(1, \text{nsubj}, 2), (1, \text{xcomp}, 3), (2, \text{det}, 4), (3, \text{mark}, 5)\}$$
$$L_2 = \{(1, \text{wants}), (2, \text{developer}), (3, \text{sleep}), (4, \text{The}), (5, \text{to})\}$$
$$\prec_2 = \{(4,2), (2,1), (1,5), (5,3)\}^+ .$$

As can easily be seen, G_2 is a dependency tree for the sentence "The developer wants to sleep"; the corresponding bijection is $b = \{(1,3), (2,2), (3,5), (4,1), (5,4)\}$. A graphical representation of G_2 can be seen in the lower half of Figure 5. △

3.5 Bigraphs

Definition 3.15 (Aligned bigraph) Let Σ be an alphabet and let L_E, L_V be sets. An *(aligned) bigraph over* (Σ, L_E, L_V) is a tuple $\mathcal{B} = (G_1, G_2, w, A_1, A_2)$ where

1. $G_1 = (V_1, E_1, L_1, \prec_1)$ and $G_2 = (V_2, E_2, L_2, \prec_2)$ are graphs with edge labels from L_E and vertex labels from L_V;
2. $w = w_1 \dots w_n \in \Sigma^*$ is a string over Σ with length $n \in \mathbb{N}$;
3. $A_1 \subseteq V_1 \times [n]$ and $A_2 \subseteq V_2 \times [n]$ are *alignments* that connect vertices of G_1 and G_2 with symbols of w. △

If we are not interested in the particular sets Σ, L_E and L_V, we refer to a bigraph over (Σ, L_E, L_V) simply as *bigraph*. Let $\mathcal{B} = (G_1, G_2, w, A_1, A_2)$ be an aligned bigraph and $G_i = (V_i, E_i, L_i, \prec_i)$ for $i \in \{1,2\}$. For $v \in V_i$, $i \in \{1,2\}$, we denote by $A_i(v)$ the set $\{j \in [|w|] \mid (v,j) \in A_i\}$ of all indices of symbols to which v is aligned. If v is only aligned to a single symbol with index $j \in [|w|]$, we sometimes identify $\{j\}$ with j. That is, we view $A_i(v)$ as being the actual number j rather than the singleton set $\{j\}$. We define two mappings $\pi^1_{\mathcal{B}} : V_1 \to \mathcal{P}(V_2)$ and $\pi^2_{\mathcal{B}} : V_2 \to \mathcal{P}(V_1)$ with

$$\pi^1_{\mathcal{B}}(v_1) = \{v_2 \in V_2 \mid (v_1, v_2) \in A_1 A_2^{-1}\}$$
$$\pi^2_{\mathcal{B}}(v_2) = \{v_1 \in V_1 \mid (v_1, v_2) \in A_1 A_2^{-1}\}$$

such that $\pi^1_{\mathcal{B}}$ assigns to each vertex v of G_1 all vertices of G_2 that are aligned to at least one symbol of w to which v is also aligned; vice versa, $\pi^2_{\mathcal{B}}$ assigns to each vertex of G_2 all vertices of G_1 connected to it through some common alignment.

Example 3.16 Let G_1 and G_2 be defined as in Example 3.9 and 3.14, respectively. We consider the bigraph $\mathcal{B} = (G_1, G_2, w, A_1, A_2)$ over $(\Sigma_{\text{E}}, L_{\text{R}} \cup L_{\text{D}}, L_{\text{C}} \cup \Sigma_{\text{E}})$ where

$$w = \text{The developer wants to sleep}$$
$$A_1 = \{(1,3), (2,2), (3,5), (4,2)\} \qquad A_2 = \{(1,3), (2,2), (3,5), (4,1), (5,4)\} .$$

A graphical representation of $\mathcal{B}$ is shown in Figure 5. The following statements are true:

$$\pi^1_{\mathcal{B}}(2) = \{2\} \qquad \pi^2_{\mathcal{B}}(2) = \{2,4\} \qquad \pi^2_{\mathcal{B}}(5) = \emptyset . \qquad \triangle$$

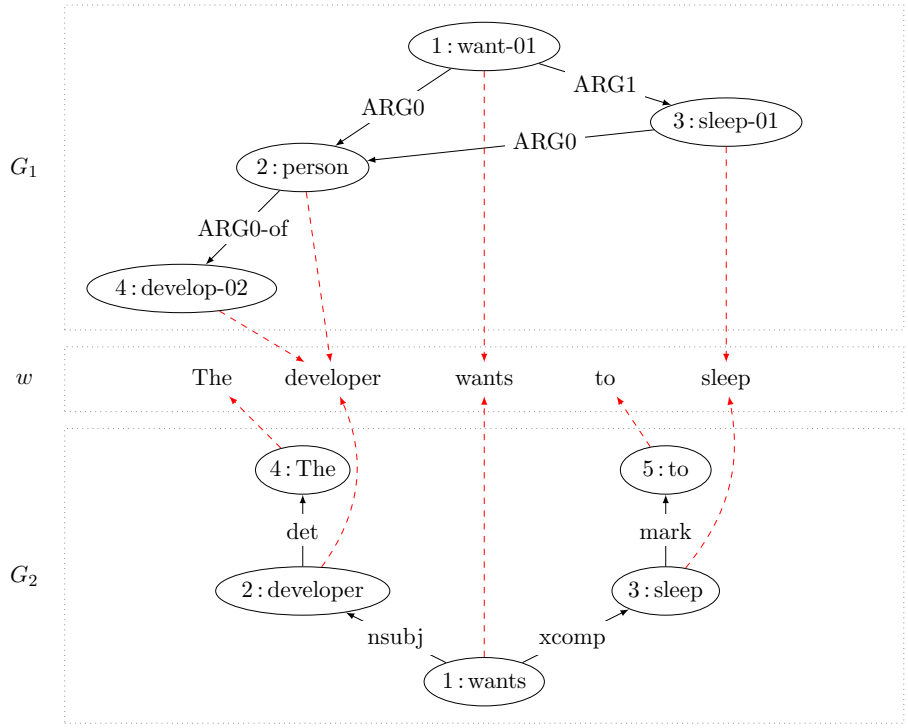


Figure 5: Graphical representation of the bigraph $\mathcal{B} = (G_1, G_2, w, A_1, A_2)$ defined in Example 3.16. For $i \in \{1, 2\}$, each node v of G_i is inscribed with $v : L_i(v)$; each alignment $(u, j) \in A_i$ is represented by a dashed arrow line connecting u and $w(j)$.

Definition 3.17 (Span) Let $\mathcal{B} = (G_1, G_2, w, A_1, A_2)$ be a bigraph, $i \in \{1, 2\}$ and let $G_i = (V_i, E_i, L_i, \prec_i)$ be an acyclic graph. The function $\mathrm{span}^i_{\mathcal{B}} : V_i \mapsto \mathcal{P}(\{1, \ldots, |w|\})$ is defined inductively for all $v \in V_i$ as

$$\mathrm{span}^i_{\mathcal{B}}(v) = A_i(v) \cup \bigcup_{v' \in \mathrm{ch}_{G_i}(v)} \mathrm{span}^i_{\mathcal{B}}(v') \,. \qquad \triangle$$

Example 3.18 We consider once more the bigraph $\mathcal{B} = (G_1, G_2, w, A_1, A_2)$ shown in Figure 5. The following holds true:

$$\mathrm{span}^1_{\mathcal{B}}(1) = \{3\} \cup \mathrm{span}^1_{\mathcal{B}}(2) \cup \mathrm{span}^1_{\mathcal{B}}(3) = \{2, 3, 5\}$$
$$\mathrm{span}^2_{\mathcal{B}}(3) = \{5\} \cup \mathrm{span}^2_{\mathcal{B}}(5) = \{4, 5\} \,. \qquad \triangle$$

3.6 Transition Systems

The key idea of this work is to define several actions – such as the deletion, merging and reordering of edges and vertices – to transform an AMR graph G into a tree structure. This structure is then turned into a realization of G through application of the yield function introduced in Definition 3.4. To embed the different kinds of required actions into a unified framework, we use the notation of *transition systems* as introduced in

Nivre (2008), but we extend the definition found therein by allowing polymorphic input and output and introducing the concept of a *finalization function*.

Definition 3.19 (Transition system) Let $\mathcal{I}$ and $\mathcal{O}$ be sets (*input space* and *output space*). A *transition system for* $(\mathcal{I}, \mathcal{O})$ is a tuple $S = (C, T, C_t, c_s, c_f)$ where

1. C is a set of *configurations* (also called *states*);
2. T is a set of *transitions*, each of which is a partial function $t \colon C \nrightarrow C$;
3. $C_t \subseteq C$ is a set of *terminal configurations*;
4. $c_s : \mathcal{I} \to C$ is an *initialization function* that maps each input from the set $\mathcal{I}$ to an *initial configuration*;
5. $c_f : C \nrightarrow \mathcal{O}$ is a *finalization function* that maps some configurations to an output from the set $\mathcal{O}$.

Let $S = (C, T, C_t, c_s, c_f)$ be a transition system for $(\mathcal{I}, \mathcal{O})$ and let $I \in \mathcal{I}$ be some input. A *partial transition sequence for I in S* is a sequence of transitions $(t_1, \ldots, t_n) \in T^*$, $n \in \mathbb{N}^+$ where

$$t_{i-1}(\ldots t_1(c_s(I))\ldots) \in \operatorname{dom}(t_i)$$

for all $i \in [n]$. Let $\tau = (t_1, \ldots, t_n)$ be a partial transition sequence for I in S. We denote by $\tau(I)$ the configuration obtained from applying the transitions $t_1, \ldots, t_n$ to $c_s(I)$, i.e.

$$\tau(I) = t_n(\ldots t_1(c_s(I))\ldots)\,.$$

If $\tau(I) \in C_t \cap \operatorname{dom}(c_f)$, we call $(t_1, \ldots, t_n)$ a *terminating transition sequence* or simply a *transition sequence*. The *output* of a terminating transition sequence τ with input I is then defined as $\mathit{out}(\tau, I) = c_f(\tau(I))$. The set of all terminating transition sequences for I in S is denoted by $\mathcal{T}(S, I)$. △

3.7 Language Modeling

A common way to improve results in natural language generation from AMR graphs is to judge each candidate realization based on two criteria: Firstly, how well does it transfer the meaning encoded by the graph? Secondly, how well does it fit into the target language? Of course, the second question can be answered regardless of the underlying graph. This is typically done using a *language model* that assigns a probability to each sentence of the target language.

Definition 3.20 (Language model) Let Σ be an alphabet. A function $p : \Sigma^* \to [0, 1]$ is called a Σ-*language model* if it is a probability distribution of Σ^*. △

Let Σ be some alphabet, $w = (w_1, \ldots, w_m)$, $m \in \mathbb{N}$ be a string over Σ and let $P(w_1, \ldots, w_n)$ denote the probability of observing this very string. The general product rule allows us to write

$$P(w_1, \ldots, w_m) = P(w_1) \cdot P(w_2 \mid w_1) \cdot \ldots \cdot P(w_m \mid w_1, \ldots, w_{m-1})\,.$$

A simplifying assumption often made is that the probability of a symbol w_i, $i \in [n]$ occurring in w does not depend on *all* previously occurring symbols w_1 to w_{i-1}, but only on a fixed number $n \in \mathbb{N}$ of previous symbols. As the first $n-1$ symbols in a sequence w do not have n previous symbols, we simply insert $n-1$ *start symbols* (denoted by $\langle s \rangle$) at the very left of the sequence. Under this assumption, we can rewrite

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_{i-n}, \ldots, w_{i-1})$$

where $w_i = \langle s \rangle$ for $i \leq 0$. A language model implementing this assumption is called an *n-gram language model*. The conditional probability $P(w_i \mid w_{i-n}, \ldots, w_{i-1})$ is often approximated by a conditional probability distribution p of Σ given Σ^n estimated from a natural language corpus $C = (w^1, \ldots, w^k) \in (\Sigma^*)^k$, $k \in \mathbb{N}$ as

$$p(w_i \mid w_{i-n}, \ldots, w_{i-1}) = \frac{\mathrm{count}_C((w_{i-n}, \ldots, w_{i-1}, w_i))}{\mathrm{count}_C((w_{i-n}, \ldots, w_{i-1}))}$$

where for all $w \in \Sigma^*$, $\mathrm{count}_C(w)$ denotes the number of occurrences of w as a substring within all strings in C. However, this simple approach suffers from the fact that whenever some sequence $(w_{i-n}, \ldots, w_{i-1}, w_i)$ does not occur at all in C, the corresponding estimated value of $p(w_i \mid w_{i-n}, \ldots, w_{i-1})$ and the probability assigned to all strings containing this sequence is equal to zero; thus, a language model trained this way is not able to handle previously unseen symbols or sequences thereof. To overcome this problem, several *smoothing* methods can be applied; the underlying idea is to subtract a small amount δ from all observed n-gram counts and to distribute it among unobserved sequences.

Example 3.21 Let $C = (\text{the man sleeps}, \text{the man and the boy}, \text{a man}) \in (\Sigma_{\mathrm{E}}^*)^3$ be an English corpus. The conditional probability $p(\text{man} \mid \text{the})$ estimated from C is

$$p(\text{man} \mid \text{the}) = \frac{\mathrm{count}_C(\text{the man})}{\mathrm{count}_C(\text{the})} = \frac{2}{3}\,. \qquad \triangle$$

A natural language corpus commonly used to train n-gram models for the English language is *Gigaword*, which consists of several million sentences obtained from various English newswire sources. As of now, five versions of Gigaword have been released, the first one being Gigaword v1 (LDC2003T05) and the newest one being Gigaword v5 (LDC2011T07).[12]

The language model used in Section 6 of this work is a 3-gram language model trained on Gigaword v1. For smoothing, we make use of a method commonly known as *Kneser-Ney smoothing*. The details of this method are beyond the scope of this work; we refer to Kneser and Ney (1995).

[12] The general releases of Gigaword v1 (LDC2003T05) and Gigaword v5 (LDC2011T07) are available at `catalog.ldc.upenn.edu/ldc2003t05` and `catalog.ldc.upenn.edu/ldc2011t07`, respectively.

3.8 Maximum Entropy Modeling

Maximum entropy modeling is a concept that can be used to estimate conditional probabilities given a set of training data (Berger et al., 1996). We will make frequent use of maximum entropy models when defining our transition system in Section 4; for example, given a configuration c and a transition t, we will use maximum entropy models to estimate $P(t \mid c)$, the probability that t is the correct transition to be applied next.

For the remainder of this section, let $\mathcal{Y}$ be a finite set of possible *outputs* and let $\mathcal{X}$ be a set of *contexts*. We will show how for all $y \in \mathcal{Y}$ and $x \in \mathcal{X}$, a maximum entropy model estimates the conditional probability of y being the correct output given context x. To this end, we use the definitions of features and maximum entropy models introduced in Berger et al. (1996) with some slight adjustments to our special use case.

Definition 3.22 (Feature function) A function $f\colon \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is called a *feature function* or, in short, a *feature*. △

Let $\mathbf{f} = (f_1, \ldots, f_n)$ be a finite sequence of features $f_i\colon \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$. The reason for introducing the concept of features is that we would like to reduce each pair $(x, y) \in \mathcal{X} \times \mathcal{Y}$ of arbitrary complexity to a real-valued vector $\mathbf{f}(x, y) = (f_1(x, y), \ldots, f_n(x, y)) \in \mathbb{R}^n$. A maximum entropy model then estimates the probability of y given x only from $\mathbf{f}(x, y)$; all information contained within x and y but not represented in $\mathbf{f}(x, y)$ is discarded.

Example 3.23 Let $\mathcal{X} = \mathcal{G}_{\text{AMR}}$ and $\mathcal{Y} = \{q, s\}$ where given an AMR graph G, the output q indicates that G represents a question and s indicates that G represents a statement. A reasonable choice of feature functions could be $\mathbf{f} = (f_1^q, f_1^s, f_2^q, f_2^s)$ where

$$f_1^y((V, E, L, \prec), y') = \begin{cases} 1 & \text{if } y = y' \wedge \exists v \in V\colon L(v) = \text{interrogative} \\ 0 & \text{otherwise} \end{cases}$$

$$f_2^y((V, E, L, \prec), y') = \begin{cases} |V| & \text{if } y = y' \\ 0 & \text{otherwise} \end{cases}$$

for all $y, y' \in \mathcal{Y}$ and $(V, E, L, \prec) \in \mathcal{G}_{\text{AMR}}$. That is, we try to decide upon whether G represents a question or a statement by considering only whether it contains a vertex with label "interrogative" and how many vertices it contains in total. △

Definition 3.24 (Maximum entropy model) A *maximum entropy model for $\mathcal{Y}$ and $\mathcal{X}$* is a conditional probability distribution p of $\mathcal{Y}$ given $\mathcal{X}$ where

$$p(y \mid x) = \frac{1}{Z_\lambda(x)} \exp\left(\sum_{i=1}^{n} \lambda_i f_i(x, y)\right)$$

with $\mathbf{f} = (f_1, \ldots f_n)$ being a finite sequence of features, $\lambda = (\lambda_1, \ldots, \lambda_n)$ being a sequence of real-valued parameters $\lambda_i \in \mathbb{R}$ for $i \in [n]$ and

$$Z_\lambda(x) = \sum_{y \in \mathcal{Y}} \exp\left(\sum_{i=1}^{n} \lambda_i f_i(x, y)\right)$$

being a normalizing factor to ensure that p is indeed a probability distribution. △

For a detailed derivation of the above definition and a discussion of the assumptions required so that $P(y \mid x)$ can be estimated by $p(y \mid x)$, we refer to Berger et al. (1996). When the sets $\mathcal{Y}$ and $\mathcal{X}$ are clear from the context, we refer to a maximum entropy model for $\mathcal{Y}$ and $\mathcal{X}$ simply as a *maximum entropy model*. While the sequence of features $\mathbf{f}$ to be used by a maximum entropy model must be specified by hand, the optimal parameter vector λ can automatically be determined given a sequence of training data for which the true output is known, i.e. a sequence $C = (x_1, y_1), \ldots, (x_m, y_m) \in (\mathcal{X} \times \mathcal{Y})^*$. The log likelihood of parameter λ given C can be calculated as

$$L(\lambda \mid C) = \log \prod_{j=1}^{m} p(y_j \mid x_j) = \sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_i f_i(x_j, y_j) - \sum_{j=1}^{m} \log Z_\lambda(x_j)$$

and the optimal parameter vector

$$\hat{\lambda} = \arg\max_{\lambda \in \mathbb{R}^n} L(\lambda \mid C)$$

can be obtained through several numerical methods such as the *Improved Iterative Scaling* (IIS) algorithm (Della Pietra et al., 1997). As the details of this process – which is also referred to as *training* of the model – are not relevant for the design of our generator, we again refer to Berger et al. (1996) for further details.

For the rest of this section, we discuss some convenient methods to turn various functions into features or feature vectors. While none of the following definitions is required for maximum entropy modeling, they simplify the notation of features used throughout this work considerably.

It is often useful to construct features by combining some information extracted only from $\mathcal{X}$ with just a single output $y \in \mathcal{Y}$. We therefore introduce a concise notation for features constructed in such a way. To this end, let $f \colon \mathcal{X} \mapsto \mathbb{R}$ and let $Y = (y_1, \ldots, y_n)$ be some enumeration of $\mathcal{Y}$. We denote by f^Y the sequence $(f^{y_1}, \ldots f^{y_n})$ where each f^{y_i}, $i \in [n]$ is a feature function with

$$f^{y_i}(x, y) = \begin{cases} f(x) & \text{if } y = y_i \\ 0 & \text{otherwise.} \end{cases}$$

As the actual order within f^Y is irrelevant as long as it is used consistently, we denote by $f^{\mathcal{Y}}$ the sequence of features obtained in the above way from some arbitrary but fixed enumeration of $\mathcal{Y}$.

Example 3.25 We consider once again the features f_2^q and f_2^s introduced in Example 3.23. For $f \colon \mathcal{G}_{\text{AMR}} \to \mathbb{R}$, defined for each $G = (V, E, L, \prec) \in \mathcal{G}_{\text{AMR}}$ by $f(G) = |V|$, it holds that $f^{(q,s)} = (f_2^q, f_2^s)$. △

Definition 3.26 (Indicator feature function) Let S be an arbitrary set. We refer to a function $s \colon \mathcal{X} \to \mathcal{P}(S)$ where $s(x)$ is finite for all $x \in \mathcal{X}$ as an *indicator feature function* or, in short, an *indicator feature*. △

Given a sequence $(x_1, \ldots, x_n) \in \mathcal{X}^n$ of training data, each indicator feature $s\colon \mathcal{X} \to \mathcal{P}(S)$ can be turned into a sequence of features as follows: Let $\{s_1, \ldots, s_m\} = \bigcup_{i=1}^{n} s(x_i)$. We first construct the ancillary sequence $f_{s_1}, \ldots, f_{s_m}$ where

$$f_{s_i}(x) = \begin{cases} 1 & \text{if } s_i \in s(x) \\ 0 & \text{otherwise} \end{cases}$$

for all $i \in [m]$. On this basis, we construct the sequence of features $\mathbf{f} = f_{s_1}^{\mathcal{Y}} \cdot \ldots \cdot f_{s_m}^{\mathcal{Y}}$.

Definition 3.27 (Indicator feature composition) Let S_1 and S_2 be sets and let $s_1\colon \mathcal{X} \to \mathcal{P}(S_1)$ and $s_2\colon \mathcal{X} \to \mathcal{P}(S_2)$ be indicator feature functions. The *composition of* s_1 *and* s_2 is the indicator feature function $s_1 \circ s_2\colon \mathcal{X} \to \mathcal{P}(S_1 \times S_2)$ with

$$(s_1 \circ s_2)(x) = \{(a, b) \in S_1 \times S_2 \mid a \in s_1(x) \wedge b \in s_2(x)\}\,. \qquad \triangle$$

Example 3.28 Let $G = (V, E, L, \prec)$ be an AMR graph. For a maximum entropy model to predict transitions, a reasonable set of contexts could be $\mathcal{X} = \mathcal{G}_{\mathrm{AMR}} \times V$ where for each tuple $(G', v) \in \mathcal{X}$, G' is the graph obtained from G so far through previously applied transitions and v is the vertex to which we want to apply the next transition. Two interesting indicator features might be $s_1\colon \mathcal{X} \to \mathcal{P}(L_{\mathrm{C}})$ and $s_2\colon \mathcal{X} \to \mathcal{P}(L_{\mathrm{C}})$ where given $G' = (V, E', L', \prec')$ and $v \in V$,

$$s_1((G', v)) = \{L'(c) \mid c \in \mathrm{ch}_{G'}(v)\} \qquad s_2((G', v)) = \{L'(p) \mid p \in \mathrm{pa}_{G'}(v)\}\,.$$

In other words, s_1 and s_2 assign to a context (G', v) the set of all labels assigned to children and parents of v in G', respectively. The composition of s_1 and s_2 is the new indicator feature function $s_1 \circ s_2\colon \mathcal{X} \to \mathcal{P}({L_{\mathrm{C}}}^2)$ where

$$(s_1 \circ s_2)((G', v)) = \{(L'(c), L'(p)) \mid c \in \mathrm{ch}_{G'}(v) \wedge p \in \mathrm{pa}_{G'}(v)\}\,. \qquad \triangle$$

4 Transition-based Generation from AMR

We now define a transition system S_{AMR} for $(\mathcal{G}_{\mathrm{AMR}}, \Sigma_{\mathrm{E}}^*)$ which we then extend to an actual generator by assigning probabilities to its transitions. For this purpose, we proceed as follows: After introducing the concept of *syntactic annotations* in Section 4.1, we define the actual transition system S_{AMR} in Section 4.2 and derive how given a probability distribution of its transitions, a generator $g\colon \mathcal{G}_{\mathrm{AMR}} \to \Sigma_{\mathrm{E}}^*$ can be built from it. To this end, we first theoretically derive the optimal output $\hat{w}$ of g given an AMR graph G. As computing this optimal output is not feasible for large graphs, we then devise an efficient algorithm to approximate $\hat{w}$. In Section 4.3, it is described how given a corpus of AMR graphs and reference realizations, the required probability distribution can be learned using several maximum entropy models. We discuss how postprocessing steps can be applied to the generated sentence for further improvement of our results in Section 4.4. Finally, we investigate in Section 4.5 how hyperparameters used throughout the generation process can be optimized using a set of development data.

4.1 Syntactic Annotations

As we have seen in Section 3.3, a lot of – mostly syntactic – information like parts of speech, number and tense gets lost in the text-to-AMR parsing process. As this information would be useful for the generation of an English sentence from an AMR graph, a key idea of this work is to annotate AMR graphs with reconstructed versions thereof. Although the desired information is arguably not purely syntactic, we refer to its reconstruction as a *syntactic annotation*. To represent syntactic annotations in a uniform way, we define a set of *syntactic annotation keys* and, for each key, a set of possible *syntactic annotation values*. A complete list of all syntactic annotation keys along with possible annotation values can be found in Table 3; exemplary syntactic annotations for vertices of an AMR graph are shown in Figure 6.[13] We denote the set of all syntactic annotation keys by $\mathcal{K}_{\mathrm{syn}} = \{\mathsf{POS}, \mathsf{DENOM}, \mathsf{TENSE}, \mathsf{NUMBER}, \mathsf{VOICE}\}$ and for each syntactic annotation key $k \in \mathcal{K}_{\mathrm{syn}}$, we refer to the set of possible annotation values as $\mathcal{V}_k$. The set of all syntactic annotation values is denoted by $\mathcal{V}_{\mathrm{syn}} = \bigcup_{k \in \mathcal{K}_{\mathrm{syn}}} \mathcal{V}_k$.

Definition 4.1 (Syntactic annotation) Let $G = (V, E, L, \prec)$ be a graph and let $v \in V$. A *syntactic annotation (for v)* is a mapping $\alpha\colon \mathcal{K}_{\mathrm{syn}} \to \mathcal{V}_{\mathrm{syn}}$ where for each $k \in \mathcal{K}_{\mathrm{syn}}$, it holds that $\alpha(k) \in \mathcal{V}_k$. The set of all syntactic annotations is denoted by $\mathcal{A}_{\mathrm{syn}}$. △

It is important to note that syntactic annotations as introduced here are strongly biased towards the English language. However, the underlying principle can easily be transfered to many other natural languages by revising the sets $\mathcal{K}_{\mathrm{syn}}$ and $\mathcal{V}_{\mathrm{syn}}$ of syntactic annotation keys and values. For example, adapting syntactic annotations to the German language may require the introduction of an additional key CASE to reflect the German case system and the redefinition of $\mathcal{V}_{\mathsf{DENOM}}$ to represent the set of German denominators.

[13] For the annotation key POS, only some exemplary values are shown in Table 3. A list of common POS tags can be found at `www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`. We use, however, only a small subset of these POS tags (see Section 4.3.2).

Key	Values	Meaning
POS	{VB, NN, JJ, CC, . . . , –}	The POS tag assigned to v
DENOM	{the, a, –}	The denominator assigned to v
TENSE	{past, present, future, –}	The tense assigned to v
NUMBER	{singular, plural, –}	The number assigned to v
VOICE	{passive, active, –}	The voice assigned to v

Table 3: Syntactic annotations used by our transition-based generator. For each syntactic annotation key $k \in \mathcal{K}_{\text{syn}}$, the set of possible values $\mathcal{V}_k$ is given and the meaning of $\alpha(k)$ for some vertex v is briefly explained.

As discussed in Section 3.3, there is often not just one reasonable syntactic annotation for the nodes of an AMR graph. To account for this in our generator, we simply consider multiple syntactic annotations per node and assign probabilities to them. For this purpose, let $G = (V, E, L, \prec)$ be a graph and let $\alpha \colon \mathcal{K}_{\text{syn}} \to \mathcal{V}_{\text{syn}}$ be a syntactic annotation for some node $v \in V$. Furthermore, let $k_1, \ldots, k_n$ be some enumeration of $\mathcal{K}_{\text{syn}}$. We denote by $P(\alpha \mid G, v)$ the probability of α being the correct annotation for v given G and v. As a syntactic annotation, like any other function, is fully defined by the values it assigns to each element of its domain, we may write

$$P(\alpha \mid G, v) = P(\alpha(k_1), \ldots, \alpha(k_n) \mid G, v)\,, \tag{1}$$

i.e. the probability of α being the correct syntactic annotation for v is equal to the joint probability of $\alpha(k_i)$ being the correct annotation value for key k_i at vertex v for all $i \in [n]$. We note that it might be useful not to look at the syntactic annotations of all nodes in V independently; for example, the tense assigned to a node depends to a large extent on the tense assigned to its predecessors. However, ignoring these dependencies allows us to handle syntactic annotations much more efficiently as we can store the m-best syntactic annotations $\alpha_1, \ldots, \alpha_m$ for each node $v \in V$ independently.[14]

Using the general product rule, we can transform Eq. (1) into

$$\begin{aligned} &P(\alpha(k_1), \ldots, \alpha(k_n) \mid G, v) \\ &= P(\alpha(k_1) \mid G, v) \cdot P(\alpha(k_2) \mid G, v, \alpha(k_1)) \cdot \ldots \cdot P(\alpha(k_n) \mid G, v, \alpha(k_1), \ldots, \alpha(k_{n-1})) \end{aligned} \tag{2}$$

and as the above holds for any enumeration $k_1, \ldots, k_m$ of $\mathcal{K}_{\text{syn}}$, we are free to choose

$$k_1 = \mathsf{POS} \quad k_2 = \mathsf{NUMBER} \quad k_3 = \mathsf{DENOM} \quad k_4 = \mathsf{VOICE} \quad k_5 = \mathsf{TENSE}\,.$$

Importantly, there are several strong dependencies between the values assigned to different syntactic annotation keys $k_i \in \mathcal{K}_{\text{syn}}$ by α. For instance, a word that is not a

[14] As can be seen in the original assignment ("Aufgabenstellung für die Masterarbeit"), our initial idea was in fact to compute syntactic annotations top-down, allowing us to infer the syntactic annotation of a node v from the annotations of its predecessors. We dismissed this idea after some preliminary tests in which it performed poorly and instead directly integrated the determination of syntactic annotations into our transition system.

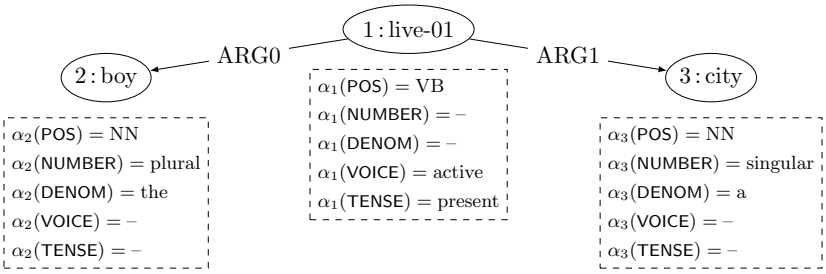


Figure 6: Exemplary syntactic annotations for an AMR graph; the annotations for each vertex are written below it and surrounded by dashed lines. A reasonable realization of the graph would be "the boys live in a city" whereas, for example, neither "the boy lives in a city" nor "the boys' life in the city" would be consistent with the given syntactic annotation.

verb should have no tense or voice assigned to it (i.e. $\alpha(\mathsf{TENSE}) = \alpha(\mathsf{VOICE}) = -$) and a plural noun can not have the article "a" as a denominator. On the other hand, it seems reasonable to assume that, for example, the tense of a verb is independent of its voice. In other words, $\alpha(\mathsf{TENSE})$ is conditionally independent of $\alpha(\mathsf{VOICE})$ given $\alpha(\mathsf{POS})$. We formulate several such conditional independence assumptions, allowing us to rewrite Eq. (2) as follows:

$$\begin{aligned} P(\alpha(k_1), \ldots, \alpha(k_n) \mid G,v) = {} & P(\alpha(\mathsf{POS}) \mid G, v) \cdot P(\alpha(\mathsf{NUMBER}) \mid G, v, \alpha(\mathsf{POS})) \\ & \cdot P(\alpha(\mathsf{DENOM}) \mid G, v, \alpha(\mathsf{POS}), \alpha(\mathsf{NUMBER})) \\ & \cdot P(\alpha(\mathsf{VOICE}) \mid G, v, \alpha(\mathsf{POS})) \cdot P(\alpha(\mathsf{TENSE}) \mid G, v, \alpha(\mathsf{POS}))\,. \end{aligned} \tag{3}$$

Finally, we estimate the above conditional probabilities using maximum entropy models p_k for each $k \in \mathcal{K}_{\text{syn}}$ and arrive at

$$\begin{aligned} P(\alpha \mid G, v) = {} & p_{\mathsf{POS}}(\alpha(\mathsf{POS}) \mid G, v) \cdot p_{\mathsf{NUMBER}}(\alpha(\mathsf{NUMBER}) \mid G, v, \alpha(\mathsf{POS})) \\ & \cdot p_{\mathsf{DENOM}}(\alpha(\mathsf{DENOM}) \mid G, v, \alpha(\mathsf{POS}), \alpha(\mathsf{NUMBER})) \\ & \cdot p_{\mathsf{VOICE}}(\alpha(\mathsf{VOICE}) \mid G, v, \alpha(\mathsf{POS})) \cdot p_{\mathsf{TENSE}}(\alpha(\mathsf{TENSE}) \mid G, v, \alpha(\mathsf{POS}))\,. \end{aligned} \tag{4}$$

Both the features extracted from G, v and α to obtain the maximum entropy models p_k and the training of these models is discussed in Section 4.3. As a final modification to the above equation, we introduce weights $w_k \in \mathbb{R}$ for each $k \in \mathcal{K}_{\text{syn}}$ and we raise each conditional probability p_k to the w_k-th power; for example, we replace $p_{\mathsf{POS}}(\alpha(\mathsf{POS}) \mid G, v)$ by $p_{\mathsf{POS}}(\alpha(\mathsf{POS}) \mid G, v)^{w_{\mathsf{POS}}}$. We denote the value obtained from $P(\alpha \mid G, v)$ through introducing these weights by $P^{\text{w}}(\alpha \mid G, v)$. While this modification is not mathematically justified, it allows our generator to decide how important it is that an applied transition actually complies with the values predicted by each of the above models. We view the weights w_k as hyperparameters; how they are obtained is described in Section 4.5.

4.2 Transition System

We now define the core part of our generator, the transition system S_{AMR}. The two main tasks to be performed by this transition system are the restructuring of the input

Key	Values	Meaning
REAL	Σ_{E}^*	The realization of v, i.e. the sequence of words that represents it in the generated sentence
DEL	$\{0, 1\}$	A flag indicating whether v needs to be deleted
INS-DONE	$\{0, 1\}$	A flag indicating whether child insertion for v is complete
LINK	V	The original vertex, if v is a copy
SWAPS	$\mathbb{Z}$	The number of times v has been swapped up ($\rho(\mathsf{SWAPS})(v) > 0$) or down ($\rho(\mathsf{SWAPS})(v) < 0$)
INIT-CONCEPT	L_{C}	The concept initially assigned to v, if it is overwritten through a MERGE transition

Table 4: Additional annotations used in the generation pipeline, assuming an AMR graph $G = (V, E, L, \prec)$. For each annotation key $k \in \mathcal{K} \setminus \mathcal{K}_{\mathrm{syn}}$, the set of possible values $\mathcal{V}_k$ is given and the meaning of $\rho(k)(v)$ for $v \in V$ is briefly explained.

AMR graph – for example by inserting and removing vertices or edges, merging multiple vertices into a single one or changing the order among them – and the determination of some additional information. The latter includes, among others, each node's syntactic annotation and its realization, i.e. a continuous sequence of words by which the node is represented in the final output of our generator. To store all additional information obtained for each node in a unified manner, we introduce the notion of an *annotation function* that generalizes the concept of syntactic annotations. We denote by

$$\mathcal{K} = \mathcal{K}_{\mathrm{syn}} \cup \{\mathsf{REAL}, \mathsf{DEL}, \mathsf{INS\text{-}DONE}, \mathsf{LINK}, \mathsf{SWAPS}, \mathsf{INIT\text{-}CONCEPT}\}$$

the set of all *annotation keys*. For each annotation key $k \in \mathcal{K} \setminus \mathcal{K}_{\mathrm{syn}}$, the set of corresponding *annotation values* $\mathcal{V}_k$ is shown in Table 4; for syntactic annotations, we refer to Table 3. While the meaning of some annotation keys might be unclear at this moment, it will become clear during the discussion of S_{AMR}. We denote by $\mathcal{V} = \bigcup_{k \in \mathcal{K}} \mathcal{V}_k$ the set of all possible annotation values.

Definition 4.2 (Annotation function) Let V be a set of vertices. An *annotation function for* V is a function $\rho \colon \mathcal{K} \to (V \nrightarrow \mathcal{V})$ such that for all $k \in \mathcal{K}$ and for all $v \in \mathrm{dom}(\rho(k))$, it holds that $\rho(k)(v) \in \mathcal{V}_k$. △

To give an example, an annotation function ρ where

$$\rho(\mathsf{POS})(v_1) = \mathrm{NN} \qquad \rho(\mathsf{REAL})(v_2) = \text{at least}$$

would indicate that the POS tag assigned to node v_1 is NN and that the realization of v_2 is the sequence "at least". As values are assigned to annotation keys incrementally during the generation process through application of transitions, we allow $\rho(k)$ to be partial for all $k \in \mathcal{K}$. Building up on the concept of annotation functions, we may now define the set of configurations used by our generator.

Definition 4.3 (Configuration for AMR generation) A *configuration for AMR generation* is a tuple $c = (G, \sigma, \beta, \rho)$ where

1. $G = (V, E, L, \prec)$ is a rooted, acyclic $(L_\mathrm{R} \cup \{\star\}, L_\mathrm{C} \cup \Sigma_\mathrm{E}^*)$-graph with $\star \notin L_\mathrm{R}$ being a special *placeholder edge label*;
2. $\sigma = (\sigma_1, \ldots, \sigma_n) \in V^*$ is a finite sequence of nodes (*node buffer*) such that for all $v \in V$, there is at most one $i \in [n]$ with $\sigma_i = v$;
3. $\beta = (\beta_1, \ldots, \beta_m) \in \mathrm{ch}(\sigma_1)^*$ is a finite sequence of nodes (*child buffer*) such that for all $v \in \mathrm{ch}(\sigma_1)$, there is at most one $i \in [m]$ with $\beta_i = v$;
4. $\rho \colon \mathcal{K} \to (V' \nrightarrow \mathcal{V})$ is an annotation function for some $V' \supseteq V$.

The set of all configurations for AMR generation is denoted by C_AMR. △

This definition is inspired by Wang et al. (2015) where configurations are defined as triples consisting of a node buffer, an edge buffer and a graph. The underlying idea is as follows: Given a configuration $c \in C_\mathrm{AMR}$, the transition to be applied next is to modify primarily the top element of the node buffer, σ_1, and, if $\beta \neq \varepsilon$, its child β_1. If this application completes the required modifications at node σ_1 (or β_1), the latter is removed from σ (or β). That way, each node contained within σ and β gets processed one at a time until they are both empty.

Definition 4.4 (S_AMR) The tuple $S_\mathrm{AMR} = (C_\mathrm{AMR}, T_\mathrm{AMR}, C_{t\mathrm{AMR}}, c_{s\mathrm{AMR}}, c_{f\mathrm{AMR}})$ is a transition system for $(\mathcal{G}_\mathrm{AMR}, \Sigma_\mathrm{E}^*)$ where

1. $T_\mathrm{AMR} = \{\textsc{Delete-Reentrance-}(v, l) \mid v \in V, l \in L_\mathrm{R}\}$
   $\cup\ \{\textsc{Merge-}(l, p) \mid l \in \Sigma_\mathrm{E}^*, p \in \mathcal{V}_\mathsf{POS}\}$
   $\cup\ \{\textsc{Swap}, \textsc{Delete}, \textsc{Keep}, \textsc{No-Insertion}\}$
   $\cup\ \{\textsc{Realize-}(w, \alpha) \mid w \in \Sigma_\mathrm{E}^*, \alpha \in \mathcal{A}_\mathrm{syn}\}$
   $\cup\ \{\textsc{Insert-}*\textsc{-}(w, p) \mid * \in \{\textsc{Child}, \textsc{Between}\}, w \in \Sigma_\mathrm{E}, p \in \{\mathsf{left}, \mathsf{right}\}\}$
   $\cup\ \{\textsc{Reorder-}(v_1, \ldots, v_n) \mid v_i \in V, i \in [n], n \in \mathbb{N}\}$ for any set V;
2. $C_{t\mathrm{AMR}} = \{(G, \varepsilon, \varepsilon, \rho) \in C_\mathrm{AMR}\}$ is the set of all configurations with both an empty node buffer and an empty child buffer;
3. $c_{s\mathrm{AMR}}(G) = (G, \sigma_G, \varepsilon, \rho)$ for all $G \in \mathcal{G}_\mathrm{AMR}$ where σ_G is some bottom-up traversal of all nodes in G and $\rho = \{(k, \emptyset) \mid k \in \mathcal{K}\}$;
4. $c_{f\mathrm{AMR}}(c) = \mathrm{yield}_{\rho(\mathsf{REAL})}(G)$ for all $c = (G, \sigma, \beta, \rho) \in C_\mathrm{AMR}$ if $G = (V, E, L, \prec)$ is totally ordered and $V \subseteq \mathrm{dom}(\rho(\mathsf{REAL}))$; otherwise, $c_{f\mathrm{AMR}}(c)$ is undefined. △

Before looking into the transitions contained within T_AMR, it is worth nothing that there is a strong connection between some of the transitions used by our generator and the transitions used by the CAMR parser of Wang et al. (2015). For example, Delete-Reentrance can be seen as a counterpart of the Insert-Reentrance transition used

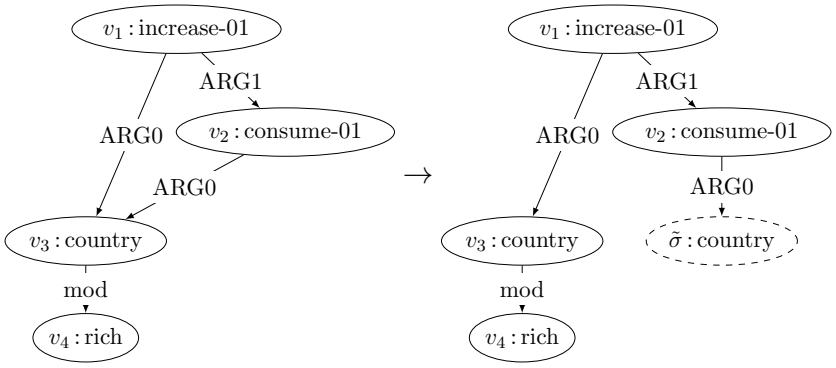


Figure 7: DELETE-REENTRANCE-$(v_2, \text{ARG0})$ transition applied to the node with label "country"; the new node $\tilde{\sigma}$ is indicated by a dashed border. The reference realization of this partial AMR graph is "rich countries increase their consumption".

in CAMR and MERGE, SWAP and DELETE transitions are used in both systems. However, other transitions such as REORDER have no direct counterpart in CAMR.

For the remainder of this section, let $G = (V, E, L, \prec)$ be an arbitrary rooted acyclic graph. If a node $v \in V$ has exactly one parent, we denote the latter by p_v. As it may be necessary to insert new nodes during the generation process, we make use of a set $V_{\text{ins}} = \{\tilde{\sigma}_i \mid i \in \mathbb{N}\}$ of *insertable nodes* for which we demand that $V \cap V_{\text{ins}} = \emptyset$. For each transition $t \in T_{\text{AMR}}$, we formally define both the actual mapping $t : C_{\text{AMR}} \nrightarrow C_{\text{AMR}}$ and $\text{dom}(t)$, the set of configurations for which t is defined. In addition, we provide a textual description and briefly justify the necessity of each class of transitions. For the more complex transitions, exemplary applications are shown in Figures 7 to 12. All AMR graphs and realizations shown in these examples are taken directly from the LDC2014T12 corpus (see Section 3.3.2) to demonstrate the actual need for the corresponding transitions.

The transitions used by our generator are defined as follows:

- DELETE-REENTRANCE-(v, l) $(v \in V,\ l \in L_{\text{R}})$

Mapping:	$(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto (G', \sigma_1{:}\tilde{\sigma}{:}\sigma, \varepsilon, \rho[\mathsf{LINK}(\tilde{\sigma}) = \sigma_1])$ where $\tilde{\sigma} \in V_{\text{ins}} \setminus V$ is some new node and $$G' = (V \cup \{\tilde{\sigma}\}, E', L \cup \{(\tilde{\sigma}, L(\sigma_1))\}, \prec)$$ $$E' = E \setminus \{(v, l, \sigma_1)\} \cup \{(v, l, \tilde{\sigma})\}\,.$$
Domain:	$\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\text{AMR}} \mid (v, l, \sigma_1) \in \text{in}_G(\sigma_1) \wedge \lvert\text{in}_G(\sigma_1)\rvert \geq 2\}$

  This transition removes the edge (v, l, σ_1); it is thus only applicable if such an edge exists and σ_1 has at least one more incoming edge. As the deleted edge may contain useful information for the generation process, a new node $\tilde{\sigma}$ is added as a

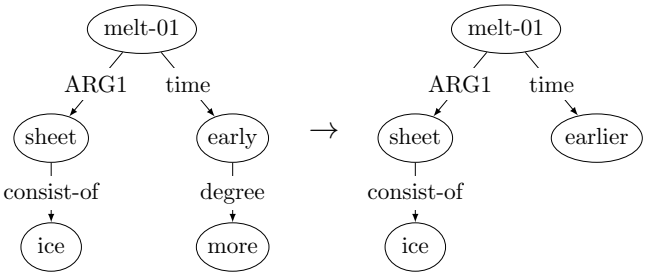


Figure 8: MERGE-(earlier,JJ) transition applied to the node with label "more". The reference realization of this partial AMR graph is "the ice sheet has melted earlier".

copy of σ_1 and connected to v. Further handling of this copy must be decided in separate transitions; therefore, $\tilde{\sigma}$ is inserted into the node buffer directly after σ_1.

Through application of DELETE-REENTRANCE, the input is stepwise converted into a tree: Whenever a node σ_1 has multiple incoming edges, all but one of these edges are successively removed using this transition. An example can be seen in Figure 7, where one of the incoming edges for the node labeled "country" gets removed and a copy of said node is added to G; the information that $\tilde{\sigma}$ is a copy of v_3 is stored in ρ by setting $\rho(\mathsf{LINK})(\tilde{\sigma}) = v_3$. To obtain the desired realization, $\tilde{\sigma}$'s realization must then be set to "their" in a subsequent transition step.

- MERGE-(l, p) $(l \in \Sigma_{\mathrm{E}}^*, p \in \mathcal{V}_{\mathsf{POS}})$

Mapping: $(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto (G', \sigma, \varepsilon, \rho')$ where $G' = (V \setminus \{\sigma_1\}, E', L', \prec)$ and

$$E' = E \setminus \{(v_1, l, v_2) \mid \sigma_1 \in \{v_1, v_2\}, l \in L_{\mathrm{R}}\} \cup \{(p_{\sigma_1}, l, v) \mid (\sigma_1, l, v) \in E\}$$

$$L' = L \setminus \{(\sigma_1, L(\sigma_1)), (p_{\sigma_1}, L(p_{\sigma_1}))\} \cup \{(p_{\sigma_1}, l)\}$$

$$\rho' = \rho[\mathsf{POS}(p_{\sigma_1}) \mapsto p, \mathsf{INIT\text{-}CONCEPT}(p_{\sigma_1}) \mapsto L(p_{\sigma_1})]$$

Domain: $\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\mathrm{AMR}} \mid |\mathrm{in}(\sigma_1)| = 1 \wedge \sigma_1 \notin \mathrm{dom}(\rho(\mathsf{DEL}))\}$

This transition merges the top element of the node buffer, σ_1, and its parent p_{σ_1} into a single node with a new vertex label $l \in \Sigma_{\mathrm{E}}^*$ and POS tag $p \in \mathcal{V}_{\mathsf{POS}}$; it is only applicable if σ_1 has exactly one incoming edge. All outgoing edges previously connected to σ_1 get reconnected to p_{σ_1}; the initial concept of p_{σ_1} is preserved in $\rho(\mathsf{INIT\text{-}CONCEPT})(p_{\sigma_1})$.

Whenever two nodes are realized by a mutual word or their realizations share at least one common word, a MERGE transition must be applied to fuse both nodes. An example can be seen in Figure 8 where the nodes labeled "early" and "more" are realized by the single word "earlier" in the reference realization.

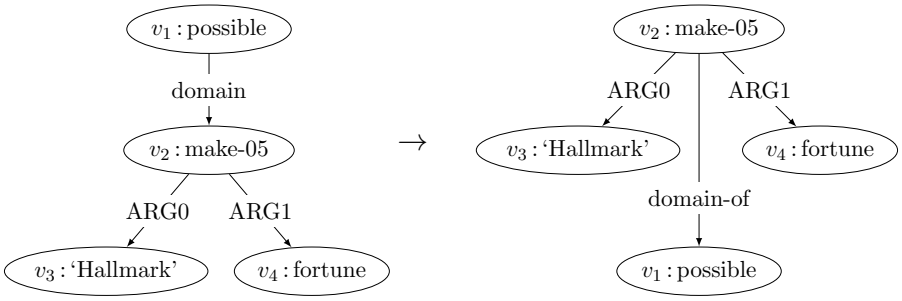


Figure 9: SWAP transition applied to the node labeled "make-05"; the edge label "domain" is converted into its inverse, "domain-of". The reference realization of this partial AMR graph is "Hallmark could make a fortune".

- SWAP

Mapping: $(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto ((V, E', L, \prec), p_{\sigma_1}{:}\sigma_1{:}(\sigma \setminus \{p_{\sigma_1}\}), \varepsilon, \rho')$ where

$$\rho' = \rho[\mathsf{SWAPS}(\sigma_1) \mapsto \mathsf{S}(\sigma_1) + 1, \mathsf{SWAPS}(p_{\sigma_1}) \mapsto \mathsf{S}(p_{\sigma_1}) - 1]$$

$$\mathsf{S}(v) = \begin{cases} \rho(\mathsf{SWAPS})(v) & \text{if } v \in \mathrm{dom}(\rho(\mathsf{SWAPS})) \\ 0 & \text{otherwise} \end{cases}$$

$$E' = E \setminus (\{(p_{\sigma_1}, l_{\sigma_1}, \sigma_1)\} \cup \{(v, l, p_{\sigma_1}) \mid v \in V, l \in L_{\mathrm{R}}\}) \cup \{(\sigma_1, l_{\sigma_1}^{-1}, p_{\sigma_1})\} \cup \{(v, l, \sigma_1) \mid (v, l, p_{\sigma_1}) \in E\}$$

and l_{σ_1} denotes the label of the edge connecting p_{σ_1} and σ_1.

Domain: $\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\mathrm{AMR}} \mid |\mathrm{in}(\sigma_1)| = 1 \wedge \sigma_1 \notin \mathrm{dom}(\rho(\mathsf{DEL}))\}$

This transition swaps the top node of the node buffer, σ_1, with its parent node. It is therefore only applicable if σ_1 has exactly one parent node p_{σ_1} and there is only one edge connecting σ_1 and p_{σ_1}. Both the direction and the label of this single incoming edge get inverted; all parents of p_{σ_1} get disconnected from p_{σ_1} and reconnected to σ_1. The information that σ_1 and p_{σ_1} were swapped is stored in ρ by incrementing $\rho(\mathsf{SWAPS})(\sigma_1)$ and decrementing $\rho(\mathsf{SWAPS})(p_{\sigma_1})$.

SWAP transitions are required due to the projectivity of $\mathrm{yield}_{\rho(\mathsf{REAL})}$ (see Definition 3.4). For instance, consider the AMR graph shown in Figure 9. If we assume that the vertices labeled "possible", "make-05", "Hallmark" and "fortune" are realized by "could", "make", "Hallmark" and "a fortune", respectively, then for the graph on the left, there is no order $\prec$ such that $\mathrm{yield}_{\rho(\mathsf{REAL})}$ produces the desired phrase "Hallmark could make a fortune". This is the case because $\rho(\mathsf{REAL})(v_1)$ cannot occur between $\rho(\mathsf{REAL})(v_3)$ and $\rho(\mathsf{REAL})(v_2)$ as v_1 is not a successor of v_2. After swapping the node labeled "possible" with the node labeled "make-05", such an order can easily be found, namely $\prec = \{(v_3, v_1), (v_1, v_2), (v_2, v_4)\}^+$.

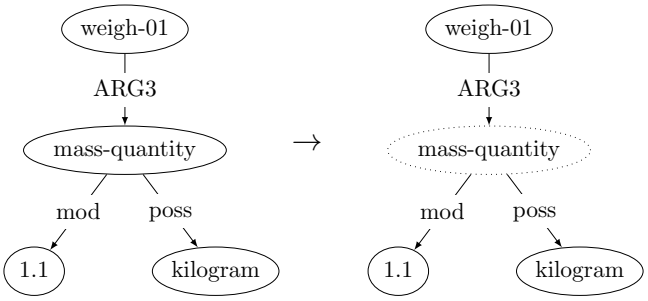


Figure 10: DELETE transition applied to the node with label "mass-quantity"; deletion is indicated by a dotted border. The reference realization of this partial AMR graph is "weighs 1.1 kilogram".

- DELETE

Mapping:	$(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto (G, \sigma_1{:}\sigma, \varepsilon, \rho[\mathsf{DEL}(\sigma_1) \mapsto 1, \mathsf{REAL}(\sigma_1) \mapsto \varepsilon])$
Domain:	$\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\text{AMR}} \mid \lvert\text{in}(\sigma_1)\rvert = 1 \wedge \sigma_1 \notin \text{dom}(\rho(\mathsf{DEL}))\}$

  Although the name may suggest otherwise, this transition does not directly remove node σ_1 from G. Instead, an application of DELETE merely indicates that node σ_1 is not represented in the generated sentence by setting the DEL flag to 1 and the realization to ε. The reason for not directly deleting σ_1 is that although it is not represented in the generated sentence, it may still provide useful information with regard to the realization and ordering of its child nodes.

  An exemplary application of DELETE is shown in Figure 10 where it is applied to the node with label "mass-quantity" as the latter has no representation in the reference realization.

- KEEP

Mapping:	$(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto (G, \sigma_1{:}\sigma, \varepsilon, \rho[\mathsf{DEL}(\sigma_1) \mapsto 0])$
Domain:	$\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\text{AMR}} \mid \lvert\text{in}(\sigma_1)\rvert = 1 \wedge \sigma_1 \notin \text{dom}(\rho(\mathsf{DEL}))\}$

  This transition serves as a counterpart to DELETE as its application indicates that the realization of node σ_1 is a part of the generated sentence. The KEEP transition also fixes the position of σ_1 with respect to its predecessors, i.e. no more MERGE or SWAP transitions can be applied to it afterwards.

  While KEEP is not an absolutely necessary transition for our transition system to work, including it allows us to make the generation process more efficient (see Section 4.2.2).

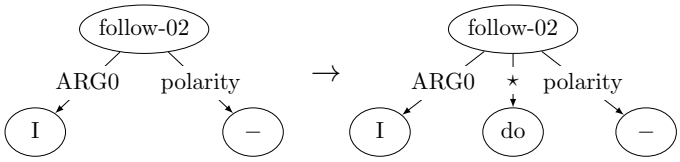


Figure 11: INSERT-CHILD-(do,left) transition applied to the node with label "follow-02". The reference realization of this partial AMR graph is "I do not follow".

- REALIZE-(w, α) $(w \in \Sigma_{\mathrm{E}}^*, \alpha \in \mathcal{A}_{\mathrm{syn}})$

Mapping:	$(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto (G, \sigma_1{:}\sigma, \varepsilon, \rho'[\mathsf{REAL}(\sigma_1) \mapsto w])$ where ρ' is obtained from ρ by setting $\rho'(k)(\sigma_1) = \alpha(k)$ for all $k \in \mathcal{K}_{\mathrm{syn}}$.
Domain:	$\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\mathrm{AMR}} \mid \rho(\mathsf{DEL})(\sigma_1) = 0 \wedge \sigma_1 \notin \mathrm{dom}(\rho(\mathsf{REAL})) \wedge (\sigma_1 \notin \mathrm{dom}(\rho(\mathsf{POS})) \vee \rho(\mathsf{POS})(\sigma_1) = \alpha(\mathsf{POS}))\}$

  REALIZE-(w, α) specifies both the syntactic annotation and the realization of node σ_1, i.e. a consecutive sequence of words w by which σ_1 is represented in the generated sentence. To give an example, reasonable transitions for a node labeled "possible" include REALIZE-(can, α_1), REALIZE-(could, α_1), REALIZE-(possible, α_2) and REALIZE-(possibility, α_3) where

$$\alpha_1 = \{(k, –) \mid k \in \mathcal{K}_{\mathrm{syn}}\}[\mathsf{POS} \mapsto \mathrm{MD}] \qquad \alpha_2 = \{(k, –) \mid k \in \mathcal{K}_{\mathrm{syn}}\}[\mathsf{POS} \mapsto \mathrm{JJ}]$$
$$\alpha_3 = \{(\mathsf{POS}, \mathrm{NN}), (\mathsf{DENOM}, \mathrm{a}), (\mathsf{TENSE}, –), (\mathsf{NUMBER}, \mathrm{singular}), (\mathsf{VOICE}, –)\}\,.$$

- INSERT-CHILD-(w, p) $(w \in \Sigma_{\mathrm{E}}, p \in \{\mathsf{left}, \mathsf{right}\})$

Mapping:	$(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto (G', \tilde{\sigma}{:}\sigma_1{:}\sigma, \varepsilon, \rho[\mathsf{DEL}(\tilde{\sigma}) \mapsto 0, \mathsf{INS\text{-}DONE}(\tilde{\sigma}) \mapsto 1])$ where $\tilde{\sigma} \in V_{\mathrm{ins}} \setminus V$ is some new node and $$G' = (V \cup \{\tilde{\sigma}\}, E \cup \{(\sigma_1, \star, \tilde{\sigma})\}, L \cup \{(\tilde{\sigma}, w)\}, \prec')$$ $$\prec' = \begin{cases} \prec \cup \{(\tilde{\sigma}, \sigma_1)\} & \text{if } p = \mathsf{left}\,, \\ \prec \cup \{(\sigma_1, \tilde{\sigma})\} & \text{if } p = \mathsf{right}\,. \end{cases}$$
Domain:	$\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\mathrm{AMR}} \mid \rho(\mathsf{DEL})(\sigma_1) = 0 \wedge \sigma_1 \in \mathrm{dom}(\rho(\mathsf{REAL})) \wedge \sigma_1 \notin \mathrm{dom}(\rho(\mathsf{INS\text{-}DONE})) \cup \mathrm{dom}(\rho(\mathsf{LINK}))\}$

  This transition inserts a new node $\tilde{\sigma}$ with label w as a child of σ_1; it also specifies whether the realization of the new node is to be left or right of σ_1 in the generated sentence. A placeholder label $\star$ is assigned to the edge connecting σ_1 and $\tilde{\sigma}$; the latter is put on top of the node buffer. To assure that the inserted node can not have children on its own, $\rho(\mathsf{INS\text{-}DONE})(\tilde{\sigma})$ is set to 1.

Commonly inserted child nodes include prepositions, articles and auxiliary verbs; an exemplary application of INSERT-CHILD-(do,left) is shown in Figure 11.

- REORDER-$(v_1, \ldots, v_n)$ ($v_i \in V$, $i \in [n]$, $n \in \mathbb{N}$)

Mapping: $(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto (G', \sigma', (v_1, \ldots, v_n) \setminus \{\sigma_1\}, \rho)$ where

$$G' = (V, E, L, \prec')$$
$$\prec' = (\prec \cup \{(v_i, v_{i+1}) \mid i \in [n-1]\})^+$$
$$\sigma' = \begin{cases} \sigma_1{:}\sigma & \text{if } n \geq 2 \\ \sigma & \text{otherwise.} \end{cases}$$

Domain: $\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\text{AMR}} \mid \{\sigma_1\} \cup \text{ch}_G(\sigma_1) = \{v_1, \ldots, v_n\}$
$\wedge\, (\sigma_1 \in \text{dom}(\rho(\textsf{INS-DONE})) \cap \text{dom}(\rho(\textsf{REAL})) \vee \rho(\textsf{DEL})(\sigma_1) = 1)$
$\wedge\, (\prec \cup \{(v_i, v_{i+1}) \mid i \in [n-1]\})^+$ is a strict order$\}$

With this transition, the order among $\text{ch}_G(\sigma_1) \cup \{\sigma_1\}$ in the realization of G is specified. After the application of REORDER, the σ_1-subgraph $G|_{\sigma_1}$ is guaranteed to be a totally ordered graph because G is processed bottom-up, i.e. for each node $v \in \text{succ}(\sigma_1)$, some instance of REORDER has already been applied.

- INSERT-BETWEEN-(w, p) ($w \in \Sigma_{\text{E}}$, $p \in \{\textsf{left}, \textsf{right}\}$)

Mapping: $(G, \sigma_1{:}\sigma, \beta_1{:}\beta, \rho) \mapsto (G', \sigma', \beta, \rho[\textsf{REAL}(\tilde{\sigma}) \mapsto w])$ where $\tilde{\sigma} \in V_{\text{ins}} \setminus V$ is some new node, l_{β_1} denotes the label of the edge connecting σ_1 with β_1 and

$$G' = (V \cup \{\tilde{\sigma}\}, E', L \cup \{(\tilde{\sigma}, w)\}, \prec')$$
$$E' = E \setminus \{(\sigma_1, l_{\beta_1}, \beta_1)\} \cup \{(\sigma_1, l_{\beta_1}, \tilde{\sigma}), (\tilde{\sigma}, \star, \beta_1)\}$$
$$\prec' = (\prec \cup \prec'' \cup \{(v, \tilde{\sigma}) \mid (v, \beta_1) \in \prec\} \cup \{(\tilde{\sigma}, v) \mid (\beta_1, v) \in \prec\})^+$$
$$\prec'' = \begin{cases} \prec \cup \{(\tilde{\sigma}, \beta_1)\} & \text{if } p = \textsf{left} \\ \prec \cup \{(\beta_1, \tilde{\sigma})\} & \text{if } p = \textsf{right} \end{cases} \qquad \sigma' = \begin{cases} \sigma_1{:}\sigma & \text{if } \beta \neq \varepsilon \\ \sigma & \text{otherwise.} \end{cases}$$

Domain: $\{(G, \sigma_1{:}\sigma, \beta_1{:}\beta, \rho) \in C_{\text{AMR}} \mid \rho(\textsf{DEL})(\sigma_1) = 0\}$

This transition inserts a new node $\tilde{\sigma}$ with label w and realization w between σ_1, the top element of the node buffer, and β_1, the top element of the child buffer; it also specifies whether the realization of $\tilde{\sigma}$ should be left or right of β_1 in the generated sentence. As INSERT-BETWEEN-(w, p) specifies both the realization and the position of the inserted node, the latter is already completely processed right

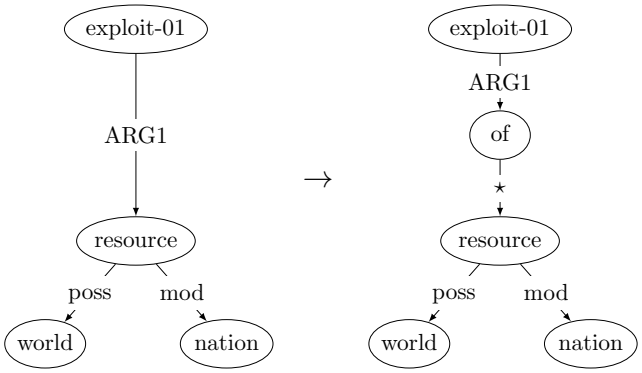


Figure 12: INSERT-BETWEEN-(of,left) transition applied to the nodes with labels "exploit-01" and "resource". The reference realization of this partial AMR graph is "the exploitation of the world's national resources".

after its insertion and therefore does not need to be put onto the node buffer. The placeholder edge label $\star$ is assigned to the new edge connecting $\tilde{\sigma}$ and β_1.

INSERT-BETWEEN transitions are mostly used to insert adpositions (e.g. "of", "to", "in", "for", "on") between two nodes; an example can be seen in Figure 12.

- NO-INSERTION

Mapping:	$(G, \sigma_1{:}\sigma, \varepsilon, \rho) \mapsto (G, \sigma_1{:}\sigma, \varepsilon, \rho[\mathsf{INS\text{-}DONE}(\sigma_1) \mapsto 1])$
	$(G, \sigma_1{:}\sigma, \beta_1{:}\beta, \rho) \mapsto (G, \sigma', \beta, \rho)$ where $\sigma' = \begin{cases} \sigma_1{:}\sigma & \text{if } \beta \neq \varepsilon \\ \sigma & \text{otherwise.} \end{cases}$
Domain:	$\{(G, \sigma_1{:}\sigma, \varepsilon, \rho) \in C_{\mathrm{AMR}} \mid \rho(\mathsf{DEL})(\sigma_1) = 0 \wedge \sigma_1 \in \mathrm{dom}(\rho(\mathsf{REAL})) \wedge \sigma_1 \notin \mathrm{dom}(\rho(\mathsf{INS\text{-}DONE}))\} \cup \{(G, \sigma_1{:}\sigma, \beta_1{:}\beta, \rho) \in C_{\mathrm{AMR}}\}$

NO-INSERTION serves as counterpart to both INSERT-BETWEEN and INSERT-CHILD and indicates that no node needs to be inserted. In case the edge buffer is not empty, this transition removes the top element β_1; otherwise, it leaves the graph and both buffers unchanged, but sets the INS-DONE flag of σ_1 to 1.

This concludes our discussion of T_{AMR}. For each transition $t \in T_{\mathrm{AMR}}$, we denote by $\mathcal{C}(t)$ the *class* to which it belongs; this class is obtained by simply removing all parameters from t. To give a few examples, $\mathcal{C}(\text{INSERT-BETWEEN-(of, left)}) = \text{INSERT-BETWEEN}$ and $\mathcal{C}(\text{MERGE-(earlier,JJ)}) = \text{MERGE}$. We extend this definition to subsets T of T_{AMR} and denote by $\mathcal{C}(T)$ the set $\{\mathcal{C}(t) \mid t \in T\}$; in particular, $\mathcal{C}(T_{\mathrm{AMR}})$ denotes the set of all classes of transitions used in our transition system S_{AMR}.

4.2.1 Modeling

We now turn the transition system S_{AMR} into an actual generator; in other words, we derive from it a function $g\colon \mathcal{G}_{\text{AMR}} \to \Sigma_{\text{E}}^*$ that assigns to each AMR graph G some realization $\hat{w} = g(G)$. Given an AMR graph G as input, our key idea is to rank all possible transition sequences according to some score. We then take the sentence generated by the highest scoring transition sequence to be the output of our generator:

$$\hat{w} = \text{out}(\hat{t}, G) \quad \text{where } \hat{t} = \operatorname*{arg\,max}_{t \in \mathcal{T}(S_{\text{AMR}}, G)} \text{score}(t, G)\,. \tag{5}$$

We define the score of a transition sequence $t = (t_1, \ldots, t_n)$, $n \in \mathbb{N}$ to be a linear combination of a score assigned to its output by some language model, denoted by score_{LM}, and a score assigned to the individual transitions t_i, $i \in [n]$, denoted by score_{TS}:

$$\text{score}(t, G) = \theta_{\text{LM}} \cdot \text{score}_{\text{LM}}(\text{out}(t, G)) + \sum_{i=1}^{n} \theta_{\mathcal{C}(t_i)} \cdot \text{score}_{\text{TS}}(t_i, t, G)\,. \tag{6}$$

In the above equation, $\theta_{\text{LM}} \in \mathbb{R}^+$ and $\theta_\tau \in \mathbb{R}^+$, $\tau \in \mathcal{C}(T_{\text{AMR}})$ are hyperparameters; how they are obtained is described in Section 4.5. We may theoretically define score_{LM} using an arbitrary language model p_{LM} (see Definition 3.20) but we explicitly assume here an n-gram model and set

$$\text{score}_{\text{LM}}(w) = \log p_{\text{LM}}(w) \cdot |w|^{-1} \tag{7}$$

where the additional factor of $|w|^{-1}$ is used to compensate for the fact that n-gram language models tend to favor sentences with only few words. We finally set

$$\text{score}_{\text{TS}}(t_i, t, G) = \log P(t_i \mid t_1, \ldots, t_{i-1}, G) \tag{8}$$

where $P(t_i \mid t_1, \ldots, t_{i-1}, G)$ denotes the probability of t_i being the correct transition to be applied next when the input to the transition system is G and the previously applied transitions are t_1 to t_{i-1}. We assume that this probability depends only on the current configuration and not on all previously applied transitions, allowing us to simplify

$$P(t_i \mid t_1, \ldots, t_{i-1}, G) = P(t_i \mid c) \tag{9}$$

where $c = (t_1, \ldots, t_{i-1})(G)$ denotes the configuration obtained from applying $t_1, \ldots, t_{i-1}$ to $c_{s\text{AMR}}(G)$ (see Definition 3.19). If t_i does not belong to one of the classes Realize and Reorder, we simply estimate the above conditional probabilities $P(t_i \mid c)$ using a maximum entropy model, i.e. we assume

$$P(t_i \mid c) = p_{\text{TS}}(t_i \mid c) \tag{10}$$

where p_{TS} is a maximum entropy model for T_{AMR} and C_{AMR}; the features used by p_{TS} will be described in Section 4.3 where we will also discuss the training procedure.

We now consider the two special cases of Realize and Reorder transitions. For this purpose, let $c = (G, \sigma_1{:}\sigma, \beta, \rho) \in C_{\text{AMR}}$ be a configuration for AMR generation

where $G = (V, E, L, \prec)$. Furthermore, let $w \in \Sigma_{\mathrm{E}}^*$ and $\alpha \in \mathcal{A}_{\mathrm{syn}}$. Using the law of total probabilities, we derive

$$P(\textsc{Realize-}(w, \alpha) \mid c) = \sum_{\alpha' \in \mathcal{A}_{\mathrm{syn}}} P(\alpha', \textsc{Realize-}(w, \alpha) \mid c) \tag{11}$$

where $P(\alpha', t \mid c)$ denotes the joint probability of α' being the right annotation for σ_1 and t being the correct transition to be applied next given c. As this transition must assign the right syntactic annotation to σ_1, we argue that $P(\alpha', \textsc{Realize-}(w, \alpha) \mid c) = 0$ for all $\alpha' \neq \alpha$, allowing us to simplify Eq. (11) to

$$\begin{aligned} P(\textsc{Realize-}(w, \alpha) \mid c) &= P(\alpha, \textsc{Realize-}(w, \alpha) \mid c) && (12) \\ &= P(\alpha \mid c) \cdot P(\textsc{Realize-}(w, \alpha) \mid c, \alpha) && (13) \end{aligned}$$

where Eq. (13) is obtained from Eq. (12) using the general product rule.

We make the simplifying assumption that $P(\alpha \mid c)$ depends only on G and σ_1, but we replace $P(\alpha \mid G, \sigma_1)$ with its weighted version $P^{\mathrm{w}}(\alpha \mid G, \sigma_1)$ as introduced in Section 4.1. Furthermore, we use a maximum entropy model $p_{\textsc{Real}}$ for T_{AMR} and $C_{\mathrm{AMR}} \times \mathcal{A}_{\mathrm{syn}}$ to estimate $P(t \mid c, \alpha)$ and obtain

$$P(\textsc{Realize-}(w, \alpha) \mid c) = P^{\mathrm{w}}(\alpha \mid G, \sigma_1) \cdot p_{\textsc{Real}}(\textsc{Realize-}(w, \alpha) \mid c, \alpha)\,. \tag{14}$$

For Reorder transitions, we use an approach similar to the one of Pourdamghani et al. (2016). Let c and G be defined as above. Furthermore, let $s = (v_1, \ldots, v_n)$, $n \in \mathbb{N}$ be a sequence of vertices from V such that $c \in \mathrm{dom}(\textsc{Reorder-}(v_1, \ldots, v_n))$. Then there is some $k \in [n]$ such that $s = (v_1, \ldots, v_{k-1}, \sigma_1, v_{k+1}, \ldots, v_n)$. Let

$$\lessdot = \{(v_i, v_j) \mid 1 \leq i < j \leq n\}$$

denote the total order such that s is the $(\mathrm{ch}(\sigma_1) \cup \{\sigma_1\})$-sequence induced by $\lessdot$. As applying $\textsc{Reorder-}(v_1, \ldots, v_n)$ has the effect of adding $\lessdot$ to $\prec$, we rewrite

$$P(\textsc{Reorder-}(v_1, \ldots, v_n) \mid c) = P(\lessdot \mid c) \tag{15}$$

where $P(\lessdot \mid c)$ denotes the probability of $\lessdot$ being the correct order among $\mathrm{ch}(\sigma_1) \cup \{\sigma_1\}$ given c. We extract from $\lessdot$ three disjoint sets

$$\begin{aligned} \lessdot_* &= \{(v_1, v_2) \in \lessdot \mid v_1 = \sigma_1 \vee v_2 = \sigma_1\} \\ \lessdot_l &= \{(v_i, v_j) \in \lessdot \mid 1 \leq i < j \leq k-1\} \\ \lessdot_r &= \{(v_i, v_j) \in \lessdot \mid k+1 \leq i < j \leq n\} \end{aligned}$$

such that $\lessdot_*$ contains all tuples from $\lessdot$ involving σ_1, $\lessdot_l$ contains all tuples for which both vertices are left of σ_1 and $\lessdot_r$ contains all tuples for which both vertices are right of σ_1. We note that $\lessdot = (\lessdot_* \cup \lessdot_r \cup \lessdot_l)^+$ and assume

$$P(\lessdot \mid c) = P(\lessdot_*, \lessdot_r, \lessdot_l \mid c)\,. \tag{16}$$

Under the further assumption that the order among the vertices left of σ_1 is independent of the order among those right of σ_1, we can use the general product rule to obtain

$$P(\lessdot \mid c) = P(\lessdot_* \mid c) \cdot P(\lessdot_r \mid c, \lessdot_*) \cdot P(\lessdot_l \mid c, \lessdot_*)\,. \tag{17}$$

We finally assume that firstly, the elements contained within $\lessdot_*$ are conditionally independent of one another given c and that secondly, for all $1 \leq i < j \leq n$ with $k \notin \{i, j\}$, the probability of v_i occurring before v_j depends only on c and the relative position of both v_1 and v_2 with respect to σ_1. This allows us to transform Eq. (17) into

$$\begin{aligned} P(\lessdot \mid c) = &\prod_{i=1}^{k-1} P(v_i \lessdot \sigma_1 \mid c) \cdot \prod_{i=k+1}^{n} P(\sigma_1 \lessdot v_i \mid c) \\ &\cdot \prod_{i=1}^{k-2} \prod_{j=i+1}^{k-1} P(v_i \lessdot v_j \mid c, v_i \lessdot \sigma_1, v_j \lessdot \sigma_1) \\ &\cdot \prod_{i=k+1}^{n-1} \prod_{j=i+1}^{n} P(v_i \lessdot v_j \mid c, \sigma_1 \lessdot v_i, \sigma_1 \lessdot v_j)\,. \end{aligned} \tag{18}$$

We note that as $\lessdot$ is a total order, for all $v, v' \in \mathrm{ch}(\sigma_1) \cup \{\sigma_1\}$ we must either have $v \lessdot v'$ or $v' \lessdot v$. We can thus rewrite

$$P(v \lessdot v' \mid c) = 1 - P(v' \lessdot v \mid c)\,.$$

Using this identity, slightly reordering the terms from Eq. (18) and estimating all required probabilities through maximum entropy models p_*, p_l and p_r, respectively, we arrive at our final equation

$$\begin{aligned} &P(\textsc{Reorder-}(v_1, \ldots, v_n) \mid c) \\ &= \prod_{i=1}^{k-1} \left(p_*(v_i \lessdot \sigma_1 \mid c) \cdot \prod_{j=i+1}^{k-1} p_l(v_i \lessdot v_j \mid c, v_i \lessdot \sigma_1, v_j \lessdot \sigma_1) \right) \\ &\quad \cdot \prod_{i=k+1}^{n} \left((1 - p_*(v_i \lessdot \sigma_1 \mid c)) \cdot \prod_{j=i+1}^{n} p_r(v_i \lessdot v_j \mid c, \sigma_1 \lessdot v_i, \sigma_1 \lessdot v_j) \right)\,. \end{aligned} \tag{19}$$

Like for the other classes of transitions, the details of training the maximum entropy models from the above equation are described in Section 4.3.

4.2.2 Decoding

Unfortunately, finding the solution to Eq. (5) by simply trying all possible transition sequences $t \in \mathcal{T}(S_{\mathrm{AMR}}, G)$ is far from being feasible for large AMR graphs G. Therefore, the aim of this section is to find a good approximation $\tilde{w}$ of $g(G)$ that can efficiently be computed. We then use this approximation $\tilde{w}$ as the output of our generator.

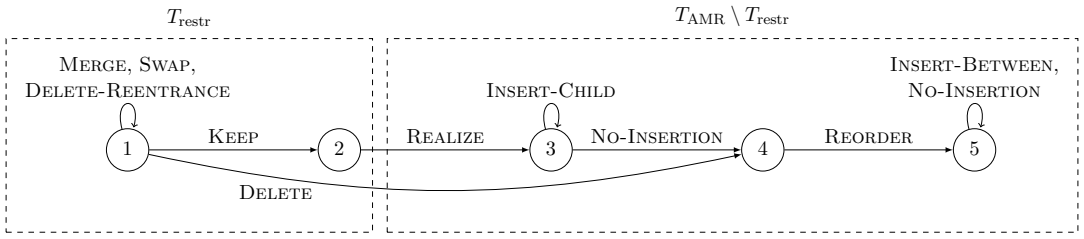


Figure 13: Graphical representation of the order in which transitions can be applied to a node

An obvious first approach to approximate $g(G)$ would be to start with the initial configuration $c_{s\text{AMR}}(G)$ and then continuously apply the most likely transition until a terminal configuration $c_t \in C_{t\text{AMR}}$ is reached. This idea is implemented in Algorithm 1, which is the equivalent of the parsing algorithm used by Wang et al. (2015); we will refer to it as the *greedy generation algorithm* and denote the obtained terminal configuration c_t by generateGreedy(G).

Algorithm 1: Greedy generation algorithm

Input: AMR graph $G = (V, E, L, \prec)$
Output: terminal configuration $c \in C_{t\text{AMR}}$

```
function generateGreedy(G)
    c ← c_sAMR(G)
    while c ∉ C_tAMR do
        T* ← {t ∈ T_AMR | c ∈ dom(t)}
        t* ← arg max_{t∈T*} P(t | c)
        c ← t*(c)
    return c
```

While this first algorithm is both extremely simple and efficient, it suffers from the obvious problem that it does not in any way integrate the language model into the generation process and thus approximates the best solution to Eq. (5) rather poorly. A simple fix for this problem might be to consider for each configuration not just one, but the n-best applicable transitions $t_1, \ldots, t_n$, $n \in \mathbb{N}$ and to rerank all so-obtained transition sequences using the language model. However, even for low values of n this approach is unfeasible as for $n > 1$, the number of transition sequences to consider grows exponentially with the number of vertices.

Yet another approach would be to directly take the language model into account at each transition step. It is, however, not clear how a partial transition sequence or a single transition might be scored by our language model; even more so if said transition does not directly effect the realization of a node. Our solution to this problem stems from an observation shown in Figure 13: The transitions in T_{AMR} are applied to each node v of our input graph G in a very specific order; this order can roughly be divided into five stages (numbered 1 to 5 in Figure 13). First, Merge, Swap and Delete-Reentrance

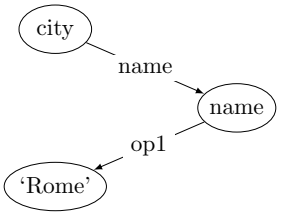


Figure 14: AMR representation of Rome

transitions modify the relation between v and its predecessors (1). Afterwards, it is decided whether v is deleted or kept; in the latter case, a realization must be determined and child nodes may be inserted (2, 3). Irrespective of whether v was deleted, an order among its children must be determined in the next stage (4) before finally, insertions between v and its children are applied (5).

In accordance with these five stages, we partition the set T_{AMR} into two disjoint sets of consecutive transitions (denoted by T_{restr} and $T_{\text{AMR}} \setminus T_{\text{restr}}$, respectively). We choose this partition in such a way that the first set is restricted to transitions for which we believe that a language model is not helpful in rating them; the second one contains all remaining transitions. Each set can then be processed separately: In a first processing phase, we modify the input AMR graph using only transitions from T_{restr} and completely ignoring the language model. In a second phase, we run a modified version of our generation algorithm on the output of the previous run, this time using only transitions from $T_{\text{AMR}} \setminus T_{\text{restr}}$, considering multiple possible transition sequences for each vertex and scoring them using the language model. As indicated in Figure 13, we set

$$T_{\text{restr}} = \{t \in T_{\text{AMR}} \mid \mathcal{C}(t) \in \{\textsc{Delete-Reentrance}, \textsc{Merge}, \textsc{Swap}, \textsc{Delete}, \textsc{Keep}\}\}\,.$$

The reason for this specific choice is that all these transitions are applied to a node *before* its realization is determined. Therefore, it often takes several subsequent transition steps until their effects on the generated sentence become clear; this makes it difficult to assign language model scores to them. While this is not entirely true for the Delete transition – which does have a direct impact on the realizations of nodes – a language model would still hardly be useful in rating it. For an example, consider the concepts "city" and "name" as used in Figure 14. Possible realizations of the corresponding AMR graph include "the city with name Rome" and simply "Rome". In most cases, we would prefer the latter realization over the first; thus, Delete transitions should be applied to the vertices labeled "name" and "city". However, as both "city" and "name" are frequent English words, it is likely that

$$\text{score}_{\text{LM}}(\text{the city with name Rome}) > \text{score}_{\text{LM}}(\text{Rome})$$

and thus, the language model strongly favors applying Keep to both vertices.

For the first phase of our generation algorithm – in which only transitions from T_{restr} are applied –, we slightly modify the definition of Delete and Keep transitions such

that the top element σ_1 is removed from the node buffer whenever one of them is applied. We denote the result of applying this modified version of the greedy generation algorithm to some input graph G by $\text{generateGreedy}_{\text{restr}}(G)$.

For the second phase of our two-phase approach, we must define how a partial transition sequence with transitions only from $T_{\text{AMR}} \setminus T_{\text{restr}}$ can be scored by a language model. As a starting point towards this goal, we first introduce the concept of *partial transition functions*.

Definition 4.5 (Partial transition function) Let $G = (V, E, L \prec)$ be a rooted acyclic graph. A *partial transition function (for G)* is a function $b\colon V \cup V_{\text{ins}} \nrightarrow (T_{\text{AMR}} \times [0, 1])^*$ that assigns to some nodes $v \in V \cup V_{\text{ins}}$ a sequence of transitions to be applied when v is the top element of the node buffer along with their probabilities. The set of all partial transition functions is denoted by $\mathcal{T}^{\text{par}}_{\text{AMR}}$. △

Using this notion of a partial transition function b, we derive Algorithm 2 that, given some configuration $c = (G, \varepsilon, \varepsilon, \rho) \in C_{\text{AMR}}$, applies to each node v of G exactly those transitions specified by b; we refer to this algorithm as the *partial generation algorithm* and denote the result of its application by $\text{generatePartial}(c, b)$.

Algorithm 2: Partial generation algorithm

Input: configuration $c = (G, \varepsilon, \varepsilon, \rho) \in C_{\text{AMR}}$ where $G = (V, E, L, \prec)$ is rooted and acyclic, partial transition function $b \in \mathcal{T}^{\text{par}}_{\text{AMR}}$
Output: configuration $c_r \in C_{\text{AMR}}$, the result of partially processing c with b

```
1  function generatePartial(c, b)
2      let σ be a bottom-up traversal of all nodes in G
3      c ← (G, σ, ε, ρ)
4      while c ∉ C_tAMR do
5          let c = (G', σ1:σ', β, ρ')
6          if σ1 ∈ dom(b) ∧ b(σ1) ≠ ε then
7              let b(σ1) = (t1, s1) · ... · (tn, sn)
8              i ← 1
9              while i ≤ n ∧ c ∈ dom(ti) do
10                 c ← ti(c)
11                 i ← i + 1
12             b(σ1) ← (ti, si) · ... · (tn, sn)
13         else
14             c ← (G', σ', ε, ρ')
15     return c
```

The partial generation algorithm allows us to process a graph even if the required transitions for some vertices are still unknown; it does so by simply ignoring these vertices. However, we are still unable to actually assign language model scores to partial transition functions. This is because we must apply $c_{f\text{AMR}}$ to obtain a sentence from

a configuration, but $c_{f\mathrm{AMR}}$ can only be applied to states whose first component is a totally ordered graph G and whose annotation function ρ assigns a realization to each node contained within said graph; otherwise, $\mathrm{yield}_{\rho(\mathsf{REAL})}(G)$ would not be defined. We therefore generalize yield to a *partial yield function* which allows for arbitrary acyclic graphs and partial realization functions.

Definition 4.6 (Partial yield) Let $G = (V, E, L, \prec)$ be an acyclic graph. Furthermore, let Σ be an alphabet, $V \subseteq V'$ and $\rho : V' \nrightarrow \Sigma^*$. The function $\mathrm{yield}^{\mathrm{par}}_{(G,\rho)} : V \to \Sigma^*$ is defined for each $v \in V$ as

$$\mathrm{yield}^{\mathrm{par}}_{(G,\rho)}(v) = \begin{cases} * & \text{if } \prec \text{ is a total order on } \mathrm{ch}(v) \cup \{v\} \text{ and } v \in \mathrm{dom}(\rho) \\ \varepsilon & \text{otherwise.} \end{cases}$$

where

$$* := \mathrm{yield}^{\mathrm{par}}_{(G,\rho)}(c_1) \cdot \ldots \cdot \mathrm{yield}^{\mathrm{par}}_{(G,\rho)}(c_k) \cdot \rho(v) \cdot \mathrm{yield}^{\mathrm{par}}_{(G,\rho)}(c_{k+1}) \cdot \ldots \cdot \mathrm{yield}^{\mathrm{par}}_{(G,\rho)}(c_{|\mathrm{ch}(v)|})$$

and $(c_1, \ldots, c_k, v, c_{k+1}, \ldots, c_{|\mathrm{ch}(v)|})$, $k \in [|\mathrm{ch}(v)|]_0$ is the $(\mathrm{ch}(v) \cup \{v\})$-sequence induced by $\prec$. If G is rooted, we write $\mathrm{yield}^{\mathrm{par}}_{\rho}(G)$ as a shorthand for $\mathrm{yield}^{\mathrm{par}}_{(G,\rho)}(\mathrm{root}(G))$. △

From the above definition it is easy to see that $\mathrm{yield}^{\mathrm{par}}_{(G,\rho)}(v)$ behaves almost like $\mathrm{yield}_{(G,\rho)}(v)$, the only difference being that the partial yield function sets the realization of all unprocessed nodes to ε and ignores all v'-subtrees of $G|_v$ for which no total order among $\mathrm{ch}(v') \cup \{v'\}$ is specified.

We are now able to make the desired generalization of our score function so that it is not only applicable to terminating transition sequences, but also to partial transition functions given an initial configuration. For this purpose, let c be a configuration and b be a partial transition function. Furthermore, let $\mathrm{generatePartial}(c, b) = (G, \sigma, \beta, \rho)$ and $v \in V$. We define the *partial score of b at v given c* to be

$$\mathrm{score}^{\mathrm{par}}(c, b, v) = \theta_{\mathrm{LM}} \cdot \mathrm{score}_{\mathrm{LM}}(\mathrm{yield}^{\mathrm{par}}_{(G,\rho(\mathsf{REAL}))}(v)) + \sum_{v' \in \mathrm{dom}(b)} \mathrm{score}^{\mathrm{par}}_{\mathrm{TS}}(b(v')) \qquad (20)$$

where

$$\mathrm{score}^{\mathrm{par}}_{\mathrm{TS}}(s) = \sum_{i=1}^{n} \theta_{\mathcal{C}(t_i)} \cdot \log p_i$$

for all $s = (t_1, p_1) \cdot \ldots \cdot (t_n, p_n) \in (T_{\mathrm{AMR}} \times [0, 1])^*$ and for all $\tau \in \mathcal{C}(T_{\mathrm{AMR}})$, θ_τ denotes the hyperparameter by the same name introduced in Eq. (6).

Example 4.7 We consider the partial transition function $b_1 \colon V \nrightarrow (T_{\mathrm{AMR}} \times [0, 1])^*$ where $\mathrm{dom}(b_1) = \{1, 2\}$ and

$$\begin{aligned} b_1(1) = {} & (\textsc{Realize-}(\mathrm{wants}, a_1), 0.75) \cdot (\textsc{No-Insertion}, 0.8) \cdot (\textsc{Reorder-}(2, 1, 3), 0.01) \\ & \cdot (\textsc{No-Insertion}, 0.9) \cdot (\textsc{Insert-Between-}(\mathrm{to}, \mathsf{left}), 0.4) \\ b_1(2) = {} & (\textsc{Realize-}(\mathrm{he}, a_2), 0.9) \cdot (\textsc{No-Insertion}, 0.95) \cdot (\textsc{Reorder-}(2), 1) \\ a_1 = {} & \{(\mathsf{POS}, \mathrm{VB}), (\mathsf{DENOM}, -), (\mathsf{TENSE}, \mathrm{present}), (\mathsf{NUMBER}, -), (\mathsf{VOICE}, \mathrm{active})\} \\ a_2 = {} & \{(\mathsf{POS}, \mathrm{PRP}), (\mathsf{DENOM}, -), (\mathsf{TENSE}, -), (\mathsf{NUMBER}, -), (\mathsf{VOICE}, -)\}\,. \end{aligned}$$

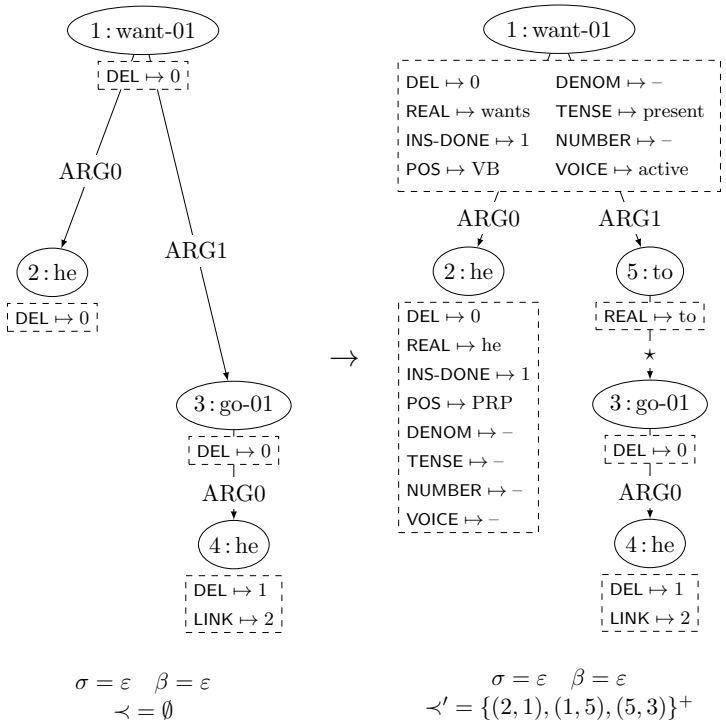


Figure 15: Application of Algorithm 2 where b is the partial transition function described in Example 4.7, c is shown on the left and the resulting configuration generatePartial(c, b) is shown on the right.

Additionally, we consider the state $c = (G, \sigma, \beta, \rho)$ shown in Figure 15 where $G = (V, E, L, \prec)$ and ρ is represented as follows: For each $k \in \mathcal{K}$ and each $v \in \mathrm{dom}(\rho(k))$, the box directly below the graphical representation of v is inscribed with $k \mapsto \rho(k)(v)$. The result of applying the partial generation algorithm, generatePartial$(c, b) = (G', \sigma', \beta', \rho')$ with $G' = (V', E', L', \prec')$ is shown in the right half of Figure 15. It holds that

$$\begin{aligned}
\mathrm{yield}^{\mathrm{par}}_{\rho'(\mathsf{REAL})}(G') &= \mathrm{yield}^{\mathrm{par}}_{(G', \rho'(\mathsf{REAL}))}(2) \cdot \rho'(\mathsf{REAL})(1) \cdot \mathrm{yield}^{\mathrm{par}}_{(G', \rho'(\mathsf{REAL}))}(5) \\
&= \rho'(\mathsf{REAL})(2) \cdot \rho'(\mathsf{REAL})(1) \cdot \rho'(\mathsf{REAL})(5) \cdot \mathrm{yield}^{\mathrm{par}}_{(G', \rho'(\mathsf{REAL}))}(3) \\
&= \rho'(\mathsf{REAL})(2) \cdot \rho'(\mathsf{REAL})(1) \cdot \rho'(\mathsf{REAL})(5) \cdot \varepsilon = \text{he wants to}\,.
\end{aligned}$$

Let $\theta_\tau = 1$ for all $\tau \in \mathcal{C}(T_{\mathrm{AMR}})$. Then

$$\mathrm{score}^{\mathrm{par}}(c, b, 1) = \theta_{\mathrm{LM}} \cdot \mathrm{score}_{\mathrm{LM}}(\text{he wants to}) + \mathrm{score}^{\mathrm{par}}_{\mathrm{TS}}(b(1)) + \mathrm{score}^{\mathrm{par}}_{\mathrm{TS}}(b(2))$$

where

$$\begin{aligned}
\mathrm{score}^{\mathrm{par}}_{\mathrm{TS}}(b(1)) &= \log 0.75 + \log 0.8 + \log 0.01 + \log 0.9 + \log 0.4 \\
\mathrm{score}^{\mathrm{par}}_{\mathrm{TS}}(b(2)) &= \log 0.9 + \log 0.95 + \log 1\,.
\end{aligned}$$

△

While we are now able to compute scores for partial transition sequences, it is still unclear how a good such sequence for a given input $G = (V, E, L, \prec)$ can efficiently be found. Our approach is to create a set of candidate partial transition functions for each v-subgraph of G bottom-up, factoring in the language model at each step. More formally, we successively construct a function $\text{best}\colon V \to \mathcal{P}(\mathcal{T}^{\text{par}}_{\text{AMR}} \times \mathbb{R})$ such that for each $v \in V$, $\text{best}(v) = \{(b_1, s_1), \ldots, (b_n, s_n)\}$ contains partial transition functions $b_1, \ldots b_n$ that specify transitions for exactly the nodes of $G|_v$, i.e. $b_i\colon \text{succ}(v) \cup \{v\} \to (T_{\text{AMR}} \times [0,1])^*$ for all $i \in [n]$; each number s_i is the partial score of the corresponding partial transition function b_i. Before we give an actual algorithm to calculate $\text{best}(v)$, we define two important functions of which we will make use in said algorithm.

Definition 4.8 (All) The mapping $\text{all}\colon C_{\text{AMR}} \to \mathcal{P}(T_{\text{AMR}} \times \mathbb{R})$, defined by

$$\text{all}(c) = \{(t, p) \in T_{\text{AMR}} \times \mathbb{R} \mid c \in \text{dom}(t) \wedge p = P(t \mid c)\}$$

for all $c \in C_{\text{AMR}}$, assigns to each configuration c the set of all applicable transitions along with their probabilities. △

Definition 4.9 (Prune) Let A be a set, $S = \{(a_1, p_1), \ldots, (a_m, p_m)\} \in \mathcal{P}(A \times \mathbb{R})$ be a set, $n \in \mathbb{N}$ and $r \in \mathbb{R}_0^+$. The set $\text{prune}_n(S)$ is defined recursively by

$$\text{prune}_n(S) = \begin{cases} \emptyset & \text{if } S = \emptyset \vee n = 0 \\ \{\hat{s}\} \cup \text{prune}_{n-1}(S \setminus \{\hat{s}\}) & \text{otherwise} \end{cases}$$

where $\hat{s} = \arg\max_{(a,p) \in S} p$. In other words, $\text{prune}_n(S)$ is the set obtained from S by including only the $k = \min(n, m)$ pairs (a_i, p_i) with the highest scores p_i. We define

$$\text{prune}_{(n,r)}(S) = \{(a, p) \in \text{prune}_n(S) \mid p \geq p_{\max} - r\}$$

where $p_{\max} = \max_{(a,p) \in S} p$. That is, $\text{prune}_{(n,r)}(S)$ is obtained from $\text{prune}_n(S)$ by retaining only pairs for which the score is lower than $p_{\max}$ by at most r. △

Example 4.10 Let $A = \{\alpha, \beta, \gamma, \delta\}$ and $S = \{(\alpha, 0.9), (\beta, 0.3), (\gamma, 0.8), (\delta, 0.45)\}$. The following holds true:

$$\begin{aligned} \text{prune}_n(S) &= S \text{ for } n \geq 4 \\ \text{prune}_3(S) &= \{(\alpha, 0.9), (\gamma, 0.8), (\delta, 0.45)\} \\ \text{prune}_{(3,\,0.15)}(S) &= \{(\alpha, 0.9), (\gamma, 0.8)\}\,. \end{aligned}$$

△

With the help of the above definitions, we can now formulate Algorithm 3 that, given an initial state $c \in C_{\text{AMR}}$, a node $v \in V$ and a partial function $\text{best}\colon V \nrightarrow \mathcal{P}(\mathcal{T}^{\text{par}}_{\text{AMR}} \times \mathbb{R})$ with $\text{succ}(v) \subseteq \text{dom}(\text{best})$, computes the set $\text{best}(v)$ containing an approximation of the best transition sequences for $\text{succ}(v) \cup \{v\}$. We call this algorithm the *best transition sequence algorithm* and refer to its output given the above input by $\text{getBest}(v, c, \text{best})$. Note that this algorithm makes use of hyperparameters $h_i = (n_i, r_i) \in \mathbb{N}^+ \times \mathbb{R}_0^+$, $i \in [5]$. These tuples are used in several places for pruning the number of transitions to be considered; the maximum size of $\text{best}(v)$ is determined by n_5.

Algorithm 3: Best transition sequence algorithm

Input: configuration $c = (G, \varepsilon, \varepsilon, \rho) \in C_{\mathrm{AMR}}$ with $G = (V, E, L, \prec)$,
vertex $v \in V$ with $\rho(\mathsf{DEL})(v) = 0$ and $v \notin \mathrm{dom}(\rho(\mathsf{REAL}))$,
function best: $V \nrightarrow \mathcal{P}(\mathcal{T}^{\mathrm{par}}_{\mathrm{AMR}} \times \mathbb{R})$ such that $\mathrm{succ}(v) \subseteq \mathrm{dom}(\mathrm{best})$

Output: n_5-best transition sequences for $\mathrm{succ}(v) \cup \{v\}$

function getBest(v, c, best)
  $c \leftarrow (G, v, \varepsilon, \rho)$
  $\mathrm{best}(v) \leftarrow \emptyset$
  for $(t_{\mathsf{real}}, s_{\mathsf{real}}) \in \mathrm{prune}_{h_1}(\mathrm{all}(c))$ **do**
    $\mathrm{hist} \leftarrow (t_{\mathsf{real}}, s_{\mathsf{real}})$
    $c_{\mathsf{real}} \leftarrow t_{\mathsf{real}}(c)$
    repeat
      $T^* \leftarrow \{t \in T_{\mathrm{AMR}} \mid c_{\mathsf{real}} \in \mathrm{dom}(t)\}$
      $t^* \leftarrow \arg\max_{t \in T^*} P(t \mid c_{\mathsf{real}})$
      $\mathrm{hist} \leftarrow \mathrm{hist} \cdot (t^*, P(t^* \mid c_{\mathsf{real}}))$
      $c_{\mathsf{real}} \leftarrow t^*(c_{\mathsf{real}})$
      if $t^* \neq$ No-Insertion **then**
        let $c_{\mathsf{real}} = (G', (\tilde{\sigma}, v), \varepsilon, \rho')$
        $\mathrm{best}(\tilde{\sigma}) \leftarrow \mathrm{getBest}(\tilde{\sigma}, c_{\mathsf{real}}, \mathrm{best})$
        $c_{\mathsf{real}} \leftarrow (G', v, \varepsilon, \rho')$
    until $t^* =$ No-Insertion
    for $(t_{\mathsf{reor}}, s_{\mathsf{reor}}) \in \mathrm{prune}_{h_2}(\mathrm{all}(c_{\mathsf{real}}))$ **do**
      $\mathrm{hist} \leftarrow \mathrm{hist} \cdot (t_{\mathsf{reor}}, s_{\mathsf{reor}})$
      $c_{\mathrm{reor}} \leftarrow t_{\mathsf{reor}}(c_{\mathsf{real}})$
      let $c_{\mathrm{reor}} = (G', \sigma, (\beta_1, \ldots, \beta_n), \rho')$
      $b_0 \leftarrow \{(v, \mathrm{hist})\}$
      $\mathrm{best}_{\leq 0}(v) \leftarrow \{(v, \{(b_0, 1)\})\}$
      for $i \leftarrow 1, \ldots, n$ **do**
        $c_i \leftarrow (G', \sigma, \beta_i, \rho')$
        $\mathrm{best}_{\leq i}(v) \leftarrow \emptyset$
        for $b \in \mathrm{best}_{\leq i-1}(v)$ **do**
          for $b_i \in \mathrm{best}(\beta_i)$ **do**
            for $(t_{\mathsf{insb}}, s_{\mathsf{insb}}) \in \mathrm{prune}_{h_3}(\mathrm{all}(c_{\mathsf{reor}}))$ **do**
              $b_{\mathrm{new}} \leftarrow b[v \mapsto b(v) \cdot (t_{\mathsf{insb}}, s_{\mathsf{insb}})] \cup b_i$
              $s_{\mathrm{new}} \leftarrow \mathrm{score}^{\mathrm{par}}(c, b_{\mathrm{new}}, v)$
              $\mathrm{best}_{\leq i}(v) \leftarrow \mathrm{prune}_{h_4}(\mathrm{best}_{\leq i}(v) \cup \{(b_{\mathrm{new}}, s_{\mathrm{new}})\})$
      $\mathrm{best}(v) \leftarrow \mathrm{prune}_{h_5}(\mathrm{best}(v) \cup \mathrm{best}_{\leq n}(v))$
  return $\mathrm{best}(v)$

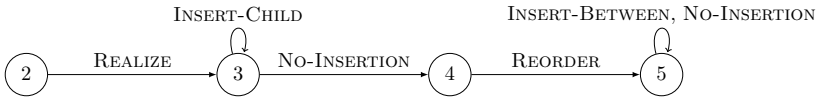


Figure 16: Representation of the order in which transitions from $T_{\text{AMR}} \setminus T_{\text{restr}}$ can be applied

As the best transition sequence algorithm is far more complex than the ones previously shown, we give a more detailed explanation. For this purpose, we again consider the five stages of processing a node shown in Figure 13; the stages relevant for Algorithm 3 are recapped in Figure 16. Algorithm 3 processes the input node v from stage 2 to stage 5, each time considering multiple possible transitions:

- Line 2 – 3: Configuration c is slightly modified as we are interested in the sequence of transitions to apply when v is on top of the node buffer; best(v) is set to $\emptyset$.
- Line 4: Given $c = (G, v, \varepsilon, \rho)$, all applicable transitions belong to the class Realize; this follows directly from the fact that $\rho(\mathsf{DEL})(v) = 0$ and there is no realization assigned to v. The n_1-best Realize-(w, α) transitions are obtained through all(c).
- Line 5 – 6: The currently chosen Realize-(w, α) transition t_{real} is stored in a sequence hist and applied to c; we thereby move from stage 2 to stage 3.
- Line 7 – 16: The most likely Insert-Child transitions are greedily applied until the best transition is No-Insertion. For each newly inserted vertex $\tilde{\sigma}$, the set of best transition sequences best$(\tilde{\sigma})$ is determined. Through application of No-Insertion, we move from stage 3 to stage 4.
- Line 17: Given configuration c_{real}, only Reorder transitions can be applied; we obtain the n_2-best Reorder-$(v_1, \ldots, v_n)$ transitions from all(c_{real}).
- Line 18 – 19: The current Reorder-$(v_1, \ldots, v_n)$ transition t_{reor} is stored in hist and applied to c_{real}; the final stage of processing v is reached.
- Line 22 – 31: We successively construct sets $\text{best}_{\leq i}(v) \subseteq \mathcal{T}^{\text{par}}_{\text{AMR}} \times \mathbb{R}$, $i \in [n]$ that, given state c_{reor}, store the best partial transition sequences for v, its children $\beta_1, \ldots, \beta_i$ and their successors. Accordingly, $\text{best}_{\leq 0}(v)$ contains only transitions previously applied to v; these transitions are inferred from hist. The set $\text{best}_{\leq i}(v)$ is obtained by iterating over all partial transition functions in both $\text{best}_{\leq i-1}(v)$ and best(β_i) as well as the n_3-best Insert-Between (or No-Insertion) transitions for v and β_i, computing the corresponding partial transition function b_{new} along with its score and collecting the n_4-best so-obtained functions. In other words, we combine the best partial transition functions for $\{v\} \cup \bigcup_{j=1}^{i-1}(\{\beta_j\} \cup \text{succ}(\beta_j))$ with the best partial transition functions for $\{\beta_i\} \cup \text{succ}(\beta_i)$ and the best applicable transitions when v is on top of the node buffer and β_i is on top of the child buffer.
- Line 32: For each considered Realize-(w, α) and Reorder-$(v_1, \ldots, v_n)$ transition, the set $\text{best}_{\leq n}(v)$ is added to best(v) which is then pruned to obtain only the n_5-best partial transition functions.

This concludes our discussion of the best transition sequence algorithm. We note that this algorithm is currently only defined for vertices v where $\rho(\mathsf{DEL})(v) = 0$. However, it can easily be extended to support also vertices with $\rho(\mathsf{DEL})(v) = 1$. We do not explicitly write down this extension, but it can be derived from Algorithm 3 by simply skipping both the realization of v and all possible insertions, i.e. only considering possible reorderings. Whenever we refer to $\text{getBest}(v, c, \text{best})$ in the future, we explicitly mean this modified version that works for each vertex v regardless of $\rho(\mathsf{DEL})(v)$.

In a last step, we combine Algorithms 1 to 3 and construct Algorithm 4, our final generation algorithm that takes as input an AMR graph G and outputs $\tilde{w}$, the desired approximation of $\hat{w}$ as defined in Eq. (5): We first apply the restricted version of Algorithm 1 to G, resulting in a state of the form $c = (G', \varepsilon, \varepsilon, \rho)$. Subsequently, we compute the sets $\text{best}(v)$ for each node v in G' bottom-up using Algorithm 3. Finally, Algorithm 2 is applied to c using $\hat{b}$, the best partial transition function found for the root of G'. Note that $\hat{b}$ is guaranteed to assign a REALIZE and REORDER transition to every node of G', so we can apply $c_{f\text{AMR}}$ to the resulting configuration.

Algorithm 4: Generation algorithm

Input: AMR graph $G = (V, E, L, \prec)$
Output: generated sentence $\tilde{w} \in \Sigma_{\text{E}}^*$

function generate(G)
  $c = (G', \varepsilon, \varepsilon, \rho) \leftarrow \text{generateGreedy}_{\text{restr}}(G)$
  let $\sigma = (\sigma_1, \ldots, \sigma_n)$ be a bottom-up traversal of all nodes in G'
  $\text{best} \leftarrow \emptyset$
  for $i \leftarrow 1, \ldots, n$ **do**
    $\text{best} \leftarrow \text{best} \cup \{(\sigma_i, \text{getBest}(\sigma_i, c, \text{best}))\}$
  $(\hat{b}, \hat{s}) \leftarrow \arg\max_{(b,s) \in \text{best}(\text{root}(G'))} s$
  $\hat{c} \leftarrow \text{generatePartial}(c, \hat{b})$
  $\tilde{w} \leftarrow c_{f\text{AMR}}(\hat{c})$
  return $\tilde{w}$

4.2.3 Complexity Analysis

We derive a theoretical upper bound for the number $N(G)$ of operations required to compute $\tilde{w} = \text{generate}(G)$ for an AMR graph G using Algorithm 4. Before we derive this upper bound, we add several constraints to our transition system, limiting the number of possible transitions. For example, the number of INSERT-CHILD transitions that can be applied to a vertex is currently unlimited, resulting in $N(G)$ being unbounded; we therefore set the maximum number of INSERT-CHILD transitions per vertex to some constant $C_{\text{ins}} \in \mathbb{N}$. We additionally demand that SWAP is never applied to vertices added through DELETE-REENTRANCE transitions and, as is done in Wang et al. (2015), that SWAP can not be reversed; that is, if a SWAP transition was applied to some vertex v with parent p_v, it may not be applied to p_v with parent v in a subsequent step. For

our study of Algorithm 4, let $G = (V, E, L, \prec)$ be the input AMR graph. Furthermore, let $G' = (V', E', L', \prec')$ be the graph constructed in line 2 and $\hat{c} = (\hat{G}, \varepsilon, \varepsilon, \hat{\rho})$ with $\hat{G} = (\hat{V}, \hat{E}, \hat{L}, \hat{\prec})$ be the configuration obtained in line 8.

Finding a bottom-up traversal of all vertices in G' (line 3) requires us to completely process all nodes therein once; it therefore takes $\mathcal{O}(|V'|)$ steps. Similarly, computing $c_{f\mathrm{AMR}}(\hat{c})$ (line 9) requires $\mathcal{O}(|\hat{V}|)$ steps. As for each $v \in \mathrm{dom}(\mathrm{best})$, $|\mathrm{best}(v)| \leq n_5$ where n_5 is the hyperparameter introduced in Algorithm 3, finding the arg max (line 7) requires $\mathcal{O}(n_5)$ steps. We will see below that all these operations are negligible compared to the number of steps required by the subroutines called in lines 2, 6 and 8. For each of these three subroutines, we assume all operations performed therein to require only a constant number of atomic steps and we denote the number of executed such operations by N_1, N_2 and N_3, respectively.

We first discuss the complexity of $\mathrm{generateGreedy}_{\mathrm{restr}}(G)$ as called in line 2 of the generation algorithm. As the restricted version of the greedy generation algorithm only considers transitions from the set T_{restr}, we can derive

$$N_1 \in \mathcal{O}(\sum_{\tau \in \mathcal{C}(T_{\mathrm{restr}})} N_1'(\tau))$$

where for each $\tau \in \mathcal{C}(T_{\mathrm{restr}})$, $N_1'(\tau)$ is an upper bound for the number of transitions from τ applied during the processing of G. As each Delete-Reentrance transition removes an edge and no other transition from T_{restr} increases the number of edges, we can easily derive the upper bound $N_1'(\textsc{Delete-Reentrance}) = |E|$. Similarly, each Merge transition removes a vertex and as Delete-Reentrance may add up to $|E|$ new vertices, we obtain the upper bound $N_1'(\textsc{Merge}) = |V| + |E|$. For each pair of vertices, at most one Swap transition can be applied and vertices inserted by Delete-Reentrance can not be swapped; therefore, $N_1'(\textsc{Swap}) = |V|^2$ is an upper bound for the number of Swap transitions. Finally, we derive $N_1'(\textsc{Delete}) + N_1'(\textsc{Keep}) = |V| + |E|$ from the fact that each vertex is either kept or deleted and this is decided exactly once. From these considerations, we can conclude that $N_1 \in \mathcal{O}(|E| + |V|^2)$. Furthermore, we can easily derive $|V'| \leq |V| + |E|$.

We now consider the subroutine $\mathrm{getBest}(\sigma_i, c, \mathrm{best})$ called in line 6. For this purpose, let $C_{\max} = \max_{v \in V} |\mathrm{ch}_G(v)|$ be the maximum number of children for all nodes in G. A straightforward analysis of the for-loops in Algorithm 3 gives

$$N_2 \in \mathcal{O}(n_1 \cdot (C_{\mathrm{ins}} \cdot N_{\mathrm{ins}} + n_2 \cdot (C_{\max} + C_{\mathrm{ins}}) \cdot n_4 \cdot n_5 \cdot n_3))$$

where the term $C_{\mathrm{ins}} \cdot N_{\mathrm{ins}}$ comes from the fact that up to C_{ins} Insert-Child transitions may be applied and for each inserted child $\tilde{\sigma}$, routine getBest is called recursively, requiring up to N_{ins} additional operations. However, as inserted vertices have no children of their own and Insert-Child transitions are not applicable to them, N_{ins} is in $\mathcal{O}(n_1)$. Due to our assumption of C_{ins} being a constant, we can further simplify

$$N_2 \in \mathcal{O}(n_1^2 + C_{\max} \cdot \prod_{i=1}^{5} n_i)\,.$$

We must take into account that getBest$(\sigma_i, c, \text{best})$ is computed once for each node $v \in V'$ and, as shown before, $|V'| \leq |V| + |E|$. However, for vertices $\tilde{\sigma}$ added through Delete-Reentrance transitions, only $\mathcal{O}(n_1)$ operations are required to compute the set best$(\tilde{\sigma})$; the reasoning is the same as above in the case of vertices added through Insert-Child transitions. Therefore, the number of operations required for executing lines 5 to 6 of the generation algorithm is

$$N_2' \in \mathcal{O}(|V| \cdot N_2 + |E| \cdot n_1)\,.$$

To compute generatePartial$(c, \hat{b})$ as called in line 8, a constant number of transitions needs to be applied to each vertex; the number of vertices is bounded by $|V| + |E|$. Additionally, up to $C_{\max}$ Insert-Between or No-Insertion transitions are applied for each vertex with at least one child; in total, however, the number of such transitions is also bounded by $|V| + |E|$ as each node is at most once the top element of the child buffer β. The resulting number of operations for the partial generation algorithm is therefore

$$N_3 \in \mathcal{O}(|V| + |E|)\,.$$

As the number of transitions applied is constant in the number of vertices and so is the number of added vertices per transition, it follows directly that $|\hat{V}| \in \mathcal{O}(|V| + |E|)$.

Combining all of the above considerations, we arrive at the sought-after upper bound

$$N(G) \in \mathcal{O}(N_1 + N_2' + N_3) = \mathcal{O}(|E| + |V| \cdot (|V| + n_1^2 + C_{\max} \cdot \prod_{i=1}^{5} n_i))$$

for the number of operations required by the generation algorithm with input G. As can be seen from the above equation, this number depends tremendously on the values chosen for hyperparameters n_1 to n_5. However, it is worth noting that in practice, the actual number of required operations is often well below this upper bound. For example, the number of Swap transitions required to process an AMR graph from one of the corpora discussed in Section 3.3.2 is rarely higher than 3, whereas our upper bound is quadratic in the number of vertices. We will further discuss the performance of Algorithm 4 from a practical point of view in Section 6.

4.3 Training

The aim of this section is to describe how the maximum entropy models introduced in Sections 4.1 and 4.2.1 can be trained given an AMR corpus $C = ((G_1, w_1), \ldots, (G_n, w_n))$. We proceed as follows: As a first step, we derive in Section 4.3.1 how an AMR corpus can be converted into the structure we use for our training process. In Section 4.3.2, we describe how the models required to estimate the probabilities of syntactic annotations can be learned. Finally, we show in Section 4.3.3 how sequences of training data $(c, t) \in C_{\text{AMR}} \times T_{\text{AMR}}$ where t is the right transition to be applied when c is the current configuration can be extracted from C to train the remaining maximum entropy models required for our transition system. We also describe the sequences of features to be used by all these models.

4.3.1 Preparations

Let $C = ((G_1, w_1), \ldots, (G_n, w_n))$ be an AMR corpus. We extend this corpus to a sequence C_{ext} from which both syntactic annotations and required transition steps can be inferred more easily. Let $(G, w) \in \mathcal{G}_{\text{AMR}} \times \Sigma_{\text{E}}^*$ be some element of C and let $G = (V_G, E_G, L_G, \prec_G)$. As a first preparation step, we convert w to lower case and remove all punctuation from it, resulting in a new string $w' = w_1 \ldots w_m$, $m \in \mathbb{N}$, $w_i \in \Sigma_{\text{E}}$ for $i \in [m]$. We then utilize a dependency parser to generate the corresponding dependency tree $D = (V_D, E_D, L_D, \prec_D)$ as well as an alignment $A_D \subseteq V_D \times [m]$. As each vertex $v \in V_D$ corresponds to exactly one word of w, A_D is guaranteed to be a bijective function. Next, we use a POS tagger to annotate each word w_i, $i \in [m]$ with its part of speech $p_i \in \mathcal{V}_{\mathsf{POS}}$; we abbreviate the obtained sequence $(w_1, p_1) \ldots (w_m, p_m)$ by w^{POS}.

As a final step, we try to obtain an alignment $A_G \subseteq V_G \times [m]$ that links each vertex $v \in V_G$ to its realization. To this end, we make use of two methods: Firstly, we use the aligner by Pourdamghani et al. (2014) which bijectively converts AMR graphs into strings and aligns the latter to realizations using the word alignment model described in Brown et al. (1993); the so obtained string-to-string alignment can then easily be converted into the desired format, resulting in the first candidate alignment $A_{\text{wa}} \subseteq V_G \times [m]$. Secondly, we use the rule-based greedy aligner by Flanigan et al. (2014) to obtain another candidate alignment $A_{\text{rb}} \subseteq V_G \times [m]$. An important difference between these two approaches is that the aligner of Flanigan et al. (2014) aligns each vertex to a *contiguous* sequence of words. In other words, for each $v \in V_G$ that is aligned to at least one word, there are some $k, l \in \mathbb{N}$ such that

$$A_{\text{rb}}(v) = \{k, k+1, k+2, \ldots, k+l-1, k+l\}\,.$$

This property is useful for our generator as the realization assigned to each vertex through Realize transitions is as well a contiguous sequence of words. Therefore, we also enforce this property upon A_{wa} by removing from it for each vertex v all tuples (v, i) that do not belong to the first contiguous sequence aligned to v, beginning from the left; we denote the resulting alignment by A'_{wa}. As it is desirable for our generator that as many words as possible are aligned to some vertex, we construct a joint alignment A by fusing both alignments. To this end, we take A'_{wa} as a baseline; for every vertex that is not aligned to any word, we adopt the alignment assigned by A_{rb}, resulting in the alignment

$$A = A'_{\text{wa}} \cup \{(v, i) \in A_{\text{rb}} \mid \nexists j \in [m] \colon (v, j) \in A'_{\text{wa}}\}\,.$$

We further improve upon this alignment by adding a small number of handwritten rules. For example, for unaligned vertices $v \in V_G$ whose concept consists of several words separated by hyphens (such as "at-least"), we search for a contiguous sequence of precisely those words in the reference realization. If such a subsequence $w_i \ldots w_{i+j}$ of w' is found and none of the corresponding words is already aligned to some vertex, we add $\{(v, k) \mid i \leq k \leq i+j\}$ to A. Also, we remove alignments to articles, auxiliary verbs and adpositions as these words should almost always be handled through Insert-Child and Insert-Between transitions and thereby get assigned their own, new vertices.

For a complete list of all handwritten alignment rules, we refer to Section 5.3.2. We denote by A_G the alignment obtained from A by applying all handwritten rules to it. The components obtained during the preparation process can be joined together into a bigraph $\mathcal{B} = (G, D, w^{\mathsf{POS}}, A_G, A_D)$. Doing so for all elements of C results in the desired extended corpus

$$C_{\text{ext}} = ((G_1, D_1, w_1^{\mathsf{POS}}, A_{G_1}, A_{D_1}), \ldots, (G_n, D_n, w_n^{\mathsf{POS}}, A_{G_n}, A_{D_n}))$$

which we require for our training process.

4.3.2 Syntactic Annotations

Throughout this section, let $\mathcal{B} = (G, D, w^{\mathsf{POS}}, A_G, A_D)$ be an element of the extended corpus C_{ext} as defined above where $G = (V_G, E_G, L_G, \prec_G)$ and $w^{\mathsf{POS}} = (w_1, p_1) \ldots (w_m, p_m)$. In the following, we first derive how for each vertex $v \in V_G$, the *gold syntactic annotation* $\alpha_v \in \mathcal{A}_{\text{syn}}$ can be obtained from $\mathcal{B}$ and then describe how a maximum entropy model can be trained from the resulting sequence of tuples $(v, \alpha_v) \in V_G \times \mathcal{A}_{\text{syn}}$.

In order to assign to some vertex $v \in V_G$ a meaningful syntactic annotation α_v, the latter should somehow be inferred from the words to which v is aligned; if there are no such words, i.e. $A_G(v) = \emptyset$, we ignore vertex v during the training process. If there are multiple such words, i.e. $|A_G(v)| \geq 2$, and these words differ with regards to their syntactic properties, we must somehow decide from which of them to infer the syntactic annotation of v. We do so in a very simple way by using a function $\text{bestPrefix}_{\mathcal{B}} : V_G \times \mathcal{P}([m]) \nrightarrow [m]$ that, given a vertex v and a nonempty set of word indices $S \subseteq [m]$, returns the index $i \in S$ such that w_i has the longest common prefix with $L_G(v)$; if multiple such indices exist, the lowest one is chosen.

Example 4.11 Let $\mathcal{B}_1 = (G_1, D_1, w_1^{\mathsf{POS}}, A_{G_1}, A_{D_1})$ be an element of the extended corpus C_{ext} where $G_1 = (V, E, L, \prec)$, $V = \{v_1, v_2, v_3, v_4\}$ and

$$L = \{(v_1, \text{person}), (v_2, \text{develop-02}), (v_3, \text{delight-01}), (v_4, -)\}$$
$$w_1^{\mathsf{POS}} = (\text{the}, \text{DT})(\text{developer}, \text{NN})(\text{is}, \text{VB})(\text{not}, \text{RB})(\text{delighted}, \text{JJ})\,.$$

The following statements are true:

$$\text{bestPrefix}_{\mathcal{B}_1}(v_2, \{2, 5\}) = 2 \quad \text{bestPrefix}_{\mathcal{B}_1}(v_2, \{4, 5\}) = 5 \quad \text{bestPrefix}_{\mathcal{B}_1}(v_1, \{1, 2\}) = 1\,.$$

Note that the last of the above statements is true although the longest common prefix of $L(v_1)$ with both w_1 and w_2 is equal to ε because index 1 is lower than 2. △

For the syntactic annotation key `POS`, we consider only a subset of the POS tags used in the *Penn Treebank Project* (Marcus et al., 1993).[15] This subset is obtained by aggregating POS tags whenever a distinction between them is not relevant to our use case or can be inferred from the value assigned to some other syntactic annotation key.

[15] A list of all POS tags used in the Penn Treebank Project can be found at `www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`.

The function simplify$\colon \mathcal{V}_{\mathsf{POS}} \to \mathcal{V}_{\mathsf{POS}}$ that maps each POS tag to the simplified version we are interested in is defined by

$$\text{simplify}(p) = \begin{cases} \text{NN} & \text{if } p \in \{\text{NN, NNS, NNP, NNPS, FW}\} \\ \text{VB} & \text{if } p \in \{\text{VB, VBD, VBP, VBZ}\} \\ \text{JJ} & \text{if } p \in \{\text{JJ, JJR, JJS, RB, RBR, RBS, WRB}\} \\ p & \text{otherwise.} \end{cases}$$

In order to obtain gold syntactic annotations, we will sometimes be required to check whether a word w is close to another word from some set $S \subseteq \Sigma_{\mathrm{E}}$; for example, to find out a noun's denominator, we must check whether it has one of the words "the", "a" and "an" to its left. However, this word is not necessarily directly adjacent to w. We therefore define the mapping $\text{left}_S^{\mathcal{B}}\colon [m] \mapsto \{\text{true}, \text{false}\}$ as

$$\text{left}_S^{\mathcal{B}}(i) = \begin{cases} \text{true} & \text{if } w_{i-1} \in S \vee (w_{i-2} \in S \wedge \text{simplify}(p_{i-1}) = \text{JJ}) \\ \text{false} & \text{otherwise} \end{cases}$$

so that $\text{left}_S^{\mathcal{B}}(i)$ is true if and only if w_i has some word from the set S to its left, possibly with some adjective or adverb between them.

Example 4.12 We consider once again the bigraph $\mathcal{B}_1 = (G_1, D_1, w_1^{\mathsf{POS}}, A_{G_1}, A_{D_1})$ as introduced in Example 4.11 where

$$w_1^{\mathsf{POS}} = (\text{the}, \text{DT})(\text{developer}, \text{NN})(\text{is}, \text{VB})(\text{not}, \text{RB})(\text{delighted}, \text{JJ}) = (w_1, p_1) \dots (w_5, p_5)\,.$$

The statements $\text{left}_{\{\text{the, a, an}\}}^{\mathcal{B}_1}(2)$ and $\text{left}_{\{\text{is}\}}^{\mathcal{B}_1}(5)$ are both true. The first statement is true because $w_{i-1} = w_1 \in \{\text{the, a, an}\}$; the second statement is true because $w_{i-2} = w_3 \in \{\text{is}\}$ and $\text{simplify}(p_{i-1}) = \text{simplify}(\text{RB}) = \text{JJ}$. △

Using the above prerequisites, we now describe how the gold syntactic annotation α_v for each vertex $v \in V_G$ can be obtained from $\mathcal{B}$. For this purpose, let $v \in V_G$ be a vertex that is aligned to at least one word, i.e. $A_G(v) \neq \emptyset$, and let $i = \text{bestPrefix}_{\mathcal{B}}(v, A_G(v))$. Furthermore, let

$$\begin{aligned} \langle \text{be} \rangle &= \{\text{be, am, is, are, was, were, being, been}\} \\ \langle \text{have} \rangle &= \{\text{have, has, had, having}\} \end{aligned}$$

be two sets containing all forms of the verbs "be" and "have", respectively. The gold syntactic annotation values $\alpha_v(k)$ for all syntactic annotation keys $k \in \mathcal{K}_{\text{syn}}$ can be determined independently as follows:

- POS: We assign to v the POS tag $\text{simplify}(p_i)$; the only exception to this rule is that when w_i is a participle and has some form of "be" or "have" to its left, we treat v like an actual verb:

$$\alpha_v(\mathsf{POS}) = \begin{cases} \text{VB} & \text{if } p_i \in \{\text{VBN, VBG}\} \wedge \text{left}_{\langle \text{be} \rangle \cup \langle \text{have} \rangle}^{\mathcal{B}}(i) \\ \text{simplify}(p_i) & \text{otherwise.} \end{cases}$$

- NUMBER: The number of v can be inferred from its non-simplified POS tag:

$$\alpha_v(\textsf{NUMBER}) = \begin{cases} \text{singular} & \text{if } p_i \in \{\text{NN, NNP, FW}\} \\ \text{plural} & \text{if } p_i \in \{\text{NNS, NNPS}\} \\ – & \text{otherwise.} \end{cases}$$

- VOICE: To determine whether a vertex has passive voice, we check whether its realization is a past participle that has some form of the verb "be" close to its left:

$$\alpha_v(\textsf{VOICE}) = \begin{cases} \text{active} & \text{if simplify}(p_i) = \text{VB} \\ \text{passive} & \text{if } p_i = \text{VBN} \wedge \text{left}^{\mathcal{B}}_{\langle \text{be} \rangle}(i) = 1 \\ – & \text{otherwise.} \end{cases}$$

- TENSE: To determine the tense of a vertex, we must take into account both its non-simplified POS tag and its left context:

$$\alpha_v(\textsf{TENSE}) = \begin{cases} \text{present} & \text{if } p_i \in \{\text{VBP, VBZ}\} \\ \text{past} & \text{if } p_i = \text{VBD} \\ \text{future} & \text{if } p_i = \text{VB} \wedge \text{left}^{\mathcal{B}}_{\{\text{will}\}}(i) = 1 \\ – & \text{otherwise.} \end{cases}$$

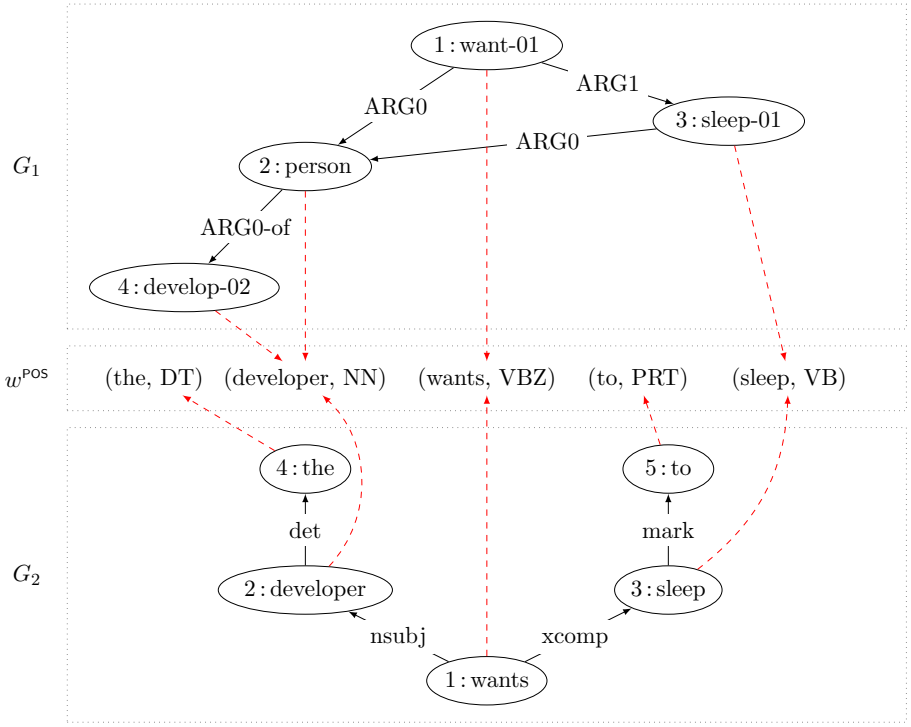
- DENOM: We devise two different approaches to assign a denominator to a vertex. While the first approach is purely based upon the AMR graph and the reference realization, the second one makes use of the dependency tree D. For the first approach, we simply check whether the currently considered vertex represents a noun and, if so, whether some article can be found close to its left:

$$\alpha_v(\textsf{DENOM}) = \begin{cases} \text{the} & \text{if simplify}(p_i) = \text{NN} \wedge \text{left}^{\mathcal{B}}_{\{\text{the}\}}(i) \\ \text{a} & \text{if simplify}(p_i) = \text{NN} \wedge \text{left}^{\mathcal{B}}_{\{\text{a, an}\}}(i) \\ – & \text{otherwise.} \end{cases}$$

  For the second approach, let $D = (V_D, E_D, L_D, \prec_D)$. We consider $v' = A_D^{-1}(i)$, the vertex of the dependency tree that corresponds to w_i, and simply check whether one of its children is an article:

$$\alpha_v(\textsf{DENOM}) = \begin{cases} \text{the} & \text{if } \exists v'' \in \text{ch}_D(v') \colon L_D(v'') = \text{the} \\ \text{a} & \text{if } \exists v'' \in \text{ch}_D(v') \colon L_D(v'') \in \{\text{a, an}\} \\ – & \text{otherwise.} \end{cases}$$

An example of how gold syntactic annotations can be obtained using the above procedures can be seen in Figure 17, where the gold syntactic annotations extracted from a POS-annotated version of the bigraph introduced in Example 3.16 are shown.



(a) Graphical representation of the bigraph $\mathcal{B} = (G_1, G_2, w^{\mathsf{POS}}, A_1, A_2)$, a POS-annotated version of the bigraph introduced in Example 3.16. For $i \in \{1, 2\}$, each node v of G_i is inscribed with $v : L_i(v)$; each alignment $(u, j) \in A_i$ is represented by a dashed arrow line connecting u and $w^{\mathsf{POS}}(j)$.

α_1	$\alpha_2 = \alpha_4$	α_3
POS $\mapsto$ VB	POS $\mapsto$ NN	POS $\mapsto$ VB
NUMBER $\mapsto$ –	NUMBER $\mapsto$ singular	NUMBER $\mapsto$ –
VOICE $\mapsto$ active	VOICE $\mapsto$ –	VOICE $\mapsto$ active
TENSE $\mapsto$ present	TENSE $\mapsto$ –	TENSE $\mapsto$ –
DENOM $\mapsto$ –	DENOM $\mapsto$ the	DENOM $\mapsto$ –

(b) Gold syntactic annotation α_i for each vertex $i \in \{1, 2, 3, 4\}$ of graph G_1 shown above

Figure 17: A bigraph and the gold syntactic annotations inferred from it

By extracting the correct syntactic annotation α_v for each $v \in V_G$ and doing so for every graph contained within our extended corpus C_{ext}, we obtain a sequence of training data that can be used to train the maximum entropy models p_k, $k \in \mathcal{K}_{\text{syn}}$ required in Section 4.1; the only remaining task is to specify the sequence of features used by these models. To fulfill this task, we first define a set $\mathcal{F}$ of *feature candidates* where each feature candidate is itself a sequence of features. We then automatically select the best working feature candidates using a greedy algorithm that works as follows:[16] We start with an empty sequence of features $\mathbf{f}_0 = \varepsilon$ and check for each of the feature candidates $\mathbf{f} \in \mathcal{F}$ whether and by how much adding the contained features to $\mathbf{f}_0$ improves the number of vertices correctly annotated by the fully trained model on a development data set. We then update $\mathbf{f}_0$ by adding to it the best performing feature candidate $\hat{\mathbf{f}}$ to obtain $\mathbf{f}_1 = \hat{\mathbf{f}}{:}\mathbf{f}_0$ and set $\mathcal{F} \leftarrow \mathcal{F} \setminus \{\hat{\mathbf{f}}\}$. We continue this procedure to obtain $\mathbf{f}_2, \ldots, \mathbf{f}_n$ until either $\mathcal{F} = \emptyset$ or no more feature candidate is found which improves the result and we take the resulting sequence $\mathbf{f}_n$ as the feature vector of our maximum entropy model. Before describing how $\mathcal{F}$ is obtained, we require two auxiliary definitions.

Definition 4.13 (Gold parent) Let $G = (V, E, L, \prec)$ be a rooted, acyclic graph and $v \in V \setminus \{\text{root}(G)\}$. The *gold parent of* v, denoted by $\widehat{\text{pa}}_G(v)$, is defined as

$$\widehat{\text{pa}}_G(v) = \operatorname*{arg\,min}_{v' \in \text{pa}_G(v)} \text{dist}(\text{root}(G), v')$$

where for all $v_1, v_2 \in V$, $\text{dist}(v_1, v_2) = 0$ if $v_1 = v_2$ and otherwise, $\text{dist}(v_1, v_2)$ denotes the number of vertices in the shortest walk starting at v_1 and ending at v_2. △

Definition 4.14 (Empirical POS tag) Let $l \in L_{\text{C}}$ be an AMR concept. The *empirical POS tag of* l, denoted by $\overline{\text{pos}}(l)$, is defined as

$$\overline{\text{pos}}(l) = \begin{cases} \texttt{PROP} & \text{if } l \text{ is a PropBank frameset} \\ \widehat{\text{pos}}(l) & \text{otherwise} \end{cases}$$

where $\widehat{\text{pos}}(l)$ denotes the POS tag observed most often for concept l in a set of training data. △

Table 5 lists the indicator features from which $\mathcal{F}$ is derived. Most of these features are parametrized with a single vertex v; when computing the feature vector for some vertex v', we set this parameter not only to v', but also to $\widehat{\text{pa}}_G(v')$ and $\widehat{\text{pa}}_G(\widehat{\text{pa}}_G(v'))$, if they exist. In other words, we extract features not only from vertex v' itself, but also from its gold parent and grandparent. We collect all so-obtained indicator features in a set $S = \{s_1, \ldots, s_m\}$, $m \in \mathbb{N}$. The set $\mathcal{F}$ of feature candidates is then derived in a one-to-one manner from the indicator features in S and all pairwise combinations $s_i \circ s_j$, $1 \leq i < j \leq m$ thereof; the details of this composition and the conversion from indicator features to actual features can be found in Section 3.8.

[16] Feature selection is also performed through the training algorithm itself by setting corresponding weights to zero. We nonetheless narrow down the choice of feature candidates to improve efficiency.

Indicator Feature	**Value**
Concept(v)	$L(v)$
Concept$_S$(v), $S \subseteq L_{\mathrm{C}}$	A flag indicating whether $L(v) \in S$
Lemma(v)	$L(v)$ with all PropBank sense tags removed
WordNetPos(v)	The most likely POS tag for Lemma(v) according to the *use count* provided by WordNet (Miller, 1995; Fellbaum, 1998)
Pos(v)	The POS tag assigned to v, if already determined
Number(v)	The number assigned to v, if already determined
InLabel(v)	If $v \neq \mathrm{root}(G)$, this is the label of the edge connecting $\widehat{\mathrm{pa}}_G(v)$ and v; otherwise, it is set to a special value `ROOT`
InLabelInv(v)	A flag indicating whether InLabel(v) ends with -of
InLabelArg(v)	A flag indicating whether InLabel(v) starts with ARG
HasChild$_l$(v), $l \in L_{\mathrm{C}}$	A flag indicating whether there is some $v' \in \mathrm{ch}_G(v)$ with $L(v') = l$
HasEdge$_l$(v), $l \in L_{\mathrm{R}}$	A flag indicating whether there is some $v' \in V$ such that $(v, l, v') \in E$
OutSize(v)	$\lvert\mathrm{ch}_G(v)\rvert$
OutEmpty(v)	A flag indicating whether $\lvert\mathrm{ch}_G(v)\rvert = 0$
OutLabels(v)	$\{l \in L_{\mathrm{R}} \mid \exists v' \in V\colon (v, l, v') \in E\}$
InLabels(v)	$\{l \in L_{\mathrm{R}} \mid \exists v' \in V\colon (v', l, v) \in E\}$
OutLabelsPos(v)	$\{(l, p) \in L_{\mathrm{R}} \times \mathcal{V}_{\mathsf{POS}} \mid \exists v' \in V\colon (v, l, v') \in E \wedge \overline{\mathrm{pos}}(L(v')) = p\}$
InLabelsPos(v)	$\{(l, p) \in L_{\mathrm{R}} \times \mathcal{V}_{\mathsf{POS}} \mid \exists v' \in V\colon (v', l, v) \in E \wedge \overline{\mathrm{pos}}(L(v')) = p\}$
Children(v)	$\{L(v') \mid v' \in \mathrm{ch}_G(v)\}$
Parents(v)	$\{L(v') \mid v' \in \mathrm{pa}_G(v)\}$
OutLabelsChildren(v)	$\{(l_r, l_c) \in L_{\mathrm{R}} \times L_{\mathrm{C}} \mid \exists v' \in V\colon (v, l_r, v') \in E \wedge L(v') = l_c\}$
NonLinkChildren(v)	$\{L(v') \mid v' \in \mathrm{ch}_G(v) \wedge v = \widehat{\mathrm{pa}}_G(v')\}$
ChildrenPos(v)	$\{\overline{\mathrm{pos}}(L(v')) \mid v' \in \mathrm{ch}_G(v)\}$
Name(v)	The name assigned to v, if name $\in$ OutLabels(v)
Mod(v)	$\{L(v') \mid v' \in V, (v, \mathrm{mod}, v') \in E\}$
ModPos(v)	$\{\overline{\mathrm{pos}}(L(v')) \mid v' \in V, (v, \mathrm{mod}, v') \in E\}$
Height(v)	The height of $G\lvert_v$, if the latter is a tree
Depth(v)	The length of the shortest path from root(G) to v
NrOfArgs(v)	$\lvert\{e \in E \mid \exists v' \in V, i \in \mathbb{N}\colon e = (v, \mathrm{ARG}i, v')\}\rvert$
ArgFlags(v)	$\{(\mathrm{ARG}i, *(i)) \mid 1 \leq i \leq 5\}$ where $*(i)$ is a flag indicating whether v has an outgoing edge labeled ARGi
ArgLinkFlags(v)	$\{(\mathrm{ARG}i, *(i)) \mid 1 \leq i \leq 5\}$ where $*(i)$ is a flag indicating whether v has an outgoing edge $(v, \mathrm{ARG}i, v')$ such that $v = \widehat{\mathrm{pa}}_G(v')$
ArgOfFlags(v)	$\{(\mathrm{ARG}i\text{-of}, *(i)) \mid 1 \leq i \leq 5\}$ where $*(i)$ is a flag indicating whether v has an incoming edge labeled ARGi-of
AllEdgeLabels	$\{l \in L_{\mathrm{R}} \mid \exists v_1, v_2 \in V\colon (v_1, l, v_2) \in E\}$
AllCombinedLabels	$\{(l_r, l_c) \in L_{\mathrm{R}} \times L_{\mathrm{C}} \mid \exists v_1, v_2 \in V\colon (v_1, l_r, v_2) \in E \wedge L(v_2) = l_c\}$

Table 5: Indicator features used for modeling the probability of syntactic annotations given an AMR graph $G = (V, E, L, \prec)$. For $v \in V$ and $l \in L_{\mathrm{C}}$, $\widehat{\mathrm{pa}}_G(v)$ denotes v's gold parent and $\overline{\mathrm{pos}}(l)$ denotes the empirical POS tag of l. For each indicator feature s, the value $s(G)$ is either explained textually or formally defined. If $s(G)$ is a singleton, delimiting brackets are omitted.

4.3.3 Transitions

We now describe how the parameters required for estimating the probability distribution $P(t \mid c)$ for $t \in T_{\mathrm{AMR}}$, $c \in C_{\mathrm{AMR}}$ with maximum entropy models can be obtained from an extended corpus C_{ext} as defined in Section 4.3.1. To this end, we first show how each element of C_{ext} can be turned into a sequence of training data $T = (c_1, t_1), \ldots, (c_m, t_m) \in (C_{\mathrm{AMR}} \times T_{\mathrm{AMR}})^*$ consisting of configurations and corresponding *gold transitions*.

We again focus on one element $\mathcal{B} = (G, D, w^{\mathsf{POS}}, A_G, A_D)$ of C_{ext}. To extract the desired sequence T from $\mathcal{B}$, we require two auxiliary procedures: Firstly, we need a function $\mathrm{gold}_{\mathcal{B}} \colon C_{\mathrm{AMR}} \setminus C_{t\mathrm{AMR}} \to T_{\mathrm{AMR}}$ that maps each non-terminal configuration c to the correct transition $\mathrm{gold}_{\mathcal{B}}(c)$ to be applied next; we call this function an *oracle*. Secondly, we require a procedure to update $\mathcal{B}$ whenever some transition t is applied to c in order to reflect this application on $\mathcal{B}$. We denote the result of updating the bigraph according to this procedure by $\mathrm{update}(\mathcal{B}, c, t)$. Using these procedures, the sequence T can be obtained through Algorithm 5, a simple modification of Algorithm 1 to which we refer as the *training data algorithm*. At the very end of the current section, a comprehensive exemplary application of the training data algorithm and the subroutines used therein is given.

Algorithm 5: Training data algorithm

Input: bigraph $\mathcal{B} = (G, D, w^{\mathsf{POS}}, A_G, A_D)$ from C_{ext}
Output: sequence of training data $T \in (C_{\mathrm{AMR}} \times T_{\mathrm{AMR}})^*$

```
function trainingData(B)
    T ← ε
    c ← c_sAMR(G)
    while c ∉ C_tAMR do
        t* ← gold_B(c)
        T ← (c, t*) : T
        B ← update(B, c, t*)
        c ← t*(c)
    return T
```

In the following, we first devise an algorithm to determine $\mathrm{gold}_{\mathcal{B}}(c)$ and then describe the procedure required to obtain $\mathrm{update}(\mathcal{B}, c, t)$. Given a configuration $c \in C_{\mathrm{AMR}}$, we compute $\mathrm{gold}_{\mathcal{B}}(c)$ by first checking for each class $\tau \in \mathcal{C}(T_{\mathrm{AMR}})$ whether some instance thereof, i.e. some transition t such that $\mathcal{C}(t) = \tau$, needs to be applied. As soon as a class τ is found of which an instance needs to be applied, we distinguish two cases: If τ is not parametrized, i.e. $\tau \in \{\textsc{Keep}, \textsc{Delete}, \textsc{Swap}, \textsc{No-Insertion}\}$, then τ is returned immediately. Otherwise, the actual instance of τ that needs to be applied is determined by calling yet another subroutine $\mathrm{gold}'_{\mathcal{B}} \colon \mathcal{C}(T_{\mathrm{AMR}}) \times C_{\mathrm{AMR}} \nrightarrow T_{\mathrm{AMR}}$ that is defined such that $\mathrm{gold}'_{\mathcal{B}}(\tau, c)$ always belongs to class τ.[17] The only exception to this

[17] In the definition of $\mathrm{gold}'_{\mathcal{B}}(\tau, c)$, we will sometimes use nondeterminism. It is therefore not a function in the strict mathematical sense; we will view it as a function nonetheless.

rule is that if $\tau \in \{\textsc{Insert-Child}, \textsc{Insert-Between}\}$, we also allow $\text{gold}'_{\mathcal{B}}(\tau, c)$ to be a No-Insertion transition. The idea outlined above is implemented in Algorithm 6, to which we will refer as the *oracle algorithm*.

Algorithm 6: Oracle algorithm

Input: configuration $c = (G, \sigma_1{:}\sigma, \beta, \rho) \in C_{\text{AMR}}$ where $G = (V, E, L, \prec)$,
bigraph $\mathcal{B} = (G, D, w^{\mathsf{POS}}, A_G, A_D)$ from C_{ext}
Output: gold transition $t \in T_{\text{AMR}}$

function $\text{gold}_{\mathcal{B}}(c)$
  if $\sigma_1 \notin \text{dom}(\rho(\mathsf{DEL}))$ **then**
    if $|\text{in}_G(\sigma_1)| \geq 2$ **then**
      return $\text{gold}'_{\mathcal{B}}(\textsc{Delete-Reentrance}, c)$
    let $\text{pa}_G(\sigma_1) = \{p_{\sigma_1}\}$
    if $A_G(\sigma_1) = \emptyset$ **then**
      return Delete
    else if $A_G(\sigma_1) \cap A_G(p_{\sigma_1}) \neq \emptyset$ **then**
      return $\text{gold}'_{\mathcal{B}}(\textsc{Merge}, c)$
    else if
     $A_G(p_{\sigma_1}) \neq \emptyset \wedge \forall i \in \text{span}^1_{\mathcal{B}}(p_{\sigma_1})\colon \min(\text{span}^1_{\mathcal{B}}(\sigma_1)) \leq i \leq \max(\text{span}^1_{\mathcal{B}}(\sigma_1))$
     then
      return Swap
    else
      return Keep
  else if $\sigma_1 \notin \text{dom}(\rho(\mathsf{REAL}))$ **then**
    return $\text{gold}'_{\mathcal{B}}(\textsc{Realize}, c)$
  else if $\sigma_1 \notin \text{dom}(\rho(\mathsf{INS\text{-}DONE})) \wedge \rho(\mathsf{DEL})(\sigma_1) = 0$ **then**
    return $\text{gold}'_{\mathcal{B}}(\textsc{Insert-Child}, c)$
  else if $\beta = \varepsilon$ **then**
    return $\text{gold}'_{\mathcal{B}}(\textsc{Reorder}, c)$
  return $\text{gold}'_{\mathcal{B}}(\textsc{Insert-Between}, c)$

We now describe how the subroutine $\text{gold}'_{\mathcal{B}}\colon \mathcal{C}(T_{\text{AMR}}) \times C_{\text{AMR}} \nrightarrow T_{\text{AMR}}$ is defined. For some classes $\tau \in \mathcal{C}(T_{\text{AMR}})$, we devise two different approaches for obtaining the best transition: one that is purely based upon the AMR graph, its realization and the alignment between them and one that additionally makes use of dependency trees.

Let $\mathcal{B} = (G, D, w^{\mathsf{POS}}, A_G, A_D)$ be an element of C_{ext} as above, $c = (G, \sigma_1{:}\sigma, \beta, \rho) \in C_{\text{AMR}}$, $G = (V_G, E_G, L_G, \prec_G)$, $D = (V_D, E_D, L_D, \prec_D)$ and $w^{\mathsf{POS}} = (w_1, p_1) \ldots (w_n, p_n)$. For $i \in [n]$, we denote w_i also by $w(i)$ and p_i also by $p(i)$. The required gold transitions can be obtained as follows:

- $\text{gold}'_{\mathcal{B}}(\textsc{Delete-Reentrance}, c)$: A *gold incoming edge* $\hat{e} \in \text{in}_G(\sigma_1)$ for vertex σ_1 is determined; we view this edge as the only incoming edge that is not to be removed.

Given $\hat{e}$, some non-gold edge $(v, l, \sigma_1) \in \text{in}(\sigma_1) \setminus \{\hat{e}\}$ is chosen nondeterministically and the transition DELETE-REENTRANCE-(v, l) is returned. We are guaranteed that such an edge exists as $|\text{in}_G(\sigma_1)| \geq 2$.

For our first approach – which makes no use of D –, we simply take the edge connecting v and its gold parent $\widehat{\text{pa}}_G(v)$ (see Definition 4.13) as the gold incoming edge $\hat{e}$. If there are multiple such edges, we choose any of them but we favor edges with non-inverted labels. We note that this approach does not even make use of w^{POS} or A_G. Therefore, $\hat{e}$ can also unambiguously be inferred from an AMR graph G during test time.

For the second approach, we use D to compute a set of candidates $C \subseteq V_G$ containing every parent of σ_1 for which some corresponding dependency tree vertex is also a parent of some dependency tree vertex corresponding to σ_1:

$$C = \{p_{\sigma_1} \in \text{pa}_G(\sigma_1) \mid \exists p_{\text{dep}} \in \pi^1_{\mathcal{B}}(p_{\sigma_1}), \sigma_{\text{dep}} \in \pi^1_{\mathcal{B}}(\sigma_1) \colon p_{\text{dep}} \in \text{pa}_D(\sigma_{\text{dep}})\}\,.$$

If C consists of only one parent candidate $\hat{p}$ and there is exactly one edge $\hat{e}$ connecting $\hat{p}$ and σ_1, we simply take $\hat{e}$ to be the gold incoming edge. Otherwise, we determine $\hat{e}$ using the first approach, but with the additional constraint that it must originate from some vertex contained within C.

- $\text{gold}'_{\mathcal{B}}(\text{MERGE}, c)$: Whenever this subroutine is called, we are guaranteed that σ_1 has exactly one parent; we denote this parent by p_{σ_1}. As the alignments $A_G(\sigma_1)$ and $A_G(p_{\sigma_1})$ are contiguous and $A_G(\sigma_1) \cap A_G(p_{\sigma_1}) \neq \emptyset$, their union $A = A_G(\sigma_1) \cup A_G(p_{\sigma_1})$ must as well be contiguous. Let $(a_1, \ldots, a_n)$ be the A-sequence induced by $<_{\mathbb{N}}$. The gold transition returned is MERGE-(real, pos) where $\text{real} = w(a_1) \ldots w(a_n)$ and $\text{pos} = \text{simplify}(p(a_1))$.

- $\text{gold}'_{\mathcal{B}}(\text{REALIZE}, c)$: Let $(a_1, \ldots, a_n)$ be the $A_G(\sigma_1)$-sequence induced by $<_{\mathbb{N}}$. We set $\text{real} = w(a_1) \ldots w(a_n)$ and return REALIZE-$(\text{real}, \alpha_{\sigma_1})$ where α_{σ_1} is the gold syntactic annotation for node σ_1 as derived in Section 4.3.2.

- $\text{gold}'_{\mathcal{B}}(\text{REORDER}, c)$: We adapt the method by Pourdamghani et al. (2016) to obtain the *gold order* among $\text{ch}_G(\sigma_1) \cup \{\sigma_1\}$. To this end, all children of σ_1 are first divided into a left and right half:

$$\begin{aligned} \text{left} &= \{v \in \text{ch}_G(\sigma_1) \mid \text{med}(\text{span}^1_{\mathcal{B}}(v)) \leq \text{med}(A_G(\sigma_1))\} \\ \text{right} &= \text{ch}_G(\sigma_1) \setminus \text{left} \end{aligned}$$

where med denotes the median of a set of natural numbers and $\text{med}(\emptyset) = -\infty$. For all $S \in \{\text{left}, \text{right}\}$, let

$$\lessdot_S = \{(v_1, v_2) \in S \times S \mid \text{med}(\text{span}^1_{\mathcal{B}}(v_1)) < \text{med}(\text{span}^1_{\mathcal{B}}(v_2))\}\,.$$

We turn $\lessdot_S$ into a total order $\lessdot_S{}'$ on S by fixing some arbitrary order among all nodes $v_1, v_2 \in S$ with $\text{med}(\text{span}^1_{\mathcal{B}}(v_1)) = \text{med}(\text{span}^1_{\mathcal{B}}(v_2))$. Let x_S denote the S-sequence induced by $\lessdot_S{}'$. We return REORDER-$(x_{\text{left}} \cdot \sigma_1 \cdot x_{\text{right}})$.

- $\text{gold}'_{\mathcal{B}}(\textsc{Insert-Child}, c)$: For the approach disregarding D, we restrict ourselves to left insertions and utilize a handwritten set $\Sigma_{\text{IC}} \subseteq \Sigma_{\text{E}}$ of allowed concepts for child insertions. This set consists mostly of auxiliary verbs and articles; for details, we refer to Section 5.3.5. We require that articles can only be inserted as children of nouns whereas auxiliary verbs can only be assigned to verbs and adjectives. Let $i = \min(A_G(\sigma_1))$ and let $k \in \mathbb{N}$ be some hyperparameter. For $j = i-1, i-2, \ldots, i-k$ we check whether w_j is an element of Σ_{IC} and the following conditions hold:

  $$(\nexists v' \in V_G \colon j \in A_G(v')) \wedge (\nexists j' \in \mathbb{N} \colon j < j' < i \wedge \text{simplify}(p_{j'}) = \text{simplify}(p_i))\,.$$

  In other words, we only consider such words as candidates for Insert-Child transitions that are not aligned to any vertex and we demand that each such word is inserted as a child of the vertex aligned to the closest word to its right with fitting POS tag. As soon as some j is found such that all of the above conditions hold, $\textsc{Insert-Child-}(\text{lem}(w_j), \mathsf{left})$ is returned where for each $e \in \Sigma_{\text{E}}$, $\text{lem}(e)$ denotes the base form of e; for example, lem(is) = be and lem(houses) = house. If no such j is found, we return No-Insertion.

  For our alternative approach using the dependency tree D, we consider the set

  $$C = \{v \in V_D \mid \exists v' \in \pi^1_{\mathcal{B}}(\sigma_1) \colon v \in \text{ch}_D(v')\}$$

  of dependency tree vertices that are children of some vertex corresponding to σ_1. For all $v \in C$, we note that $\pi^2_{\mathcal{B}}(v) = \emptyset$ means that the word at index $A_D(v)$ has no representation in the AMR graph. Therefore, we assume

  $$I = \{i \in [n] \mid \exists v \in C \colon \pi^2_{\mathcal{B}}(v) = \emptyset \wedge i = A_D(v)\}$$

  to be the set of indices of all words that need to be inserted as children of σ_1. If $I = \emptyset$, we return No-Insertion. Otherwise, let $j = \min(I)$. We return $\textsc{Insert-Child-}(\text{lem}(w(j)), d)$ where lem is defined as above and

  $$d = \begin{cases} \mathsf{left} & \text{if } j < \min(A_G(\sigma_1)) \\ \mathsf{right} & \text{otherwise.} \end{cases}$$

  For both approaches, if $\text{gold}'_{\mathcal{B}}(\textsc{Insert-Child}, c) \neq \textsc{No-Insertion}$, we denote by $\text{ind}_{\mathcal{B}}(\textsc{Insert-Child}, c)$ the index j of the word which triggered the insertion.

- $\text{gold}'_{\mathcal{B}}(\textsc{Insert-Between}, c)$: As $\beta \neq \varepsilon$ whenever this subroutine is called, we are guaranteed that there are $\beta_1 \in \text{ch}_G(\sigma_1)$ and $\beta' \in \text{ch}_G(\sigma_1)^*$ such that $\beta = \beta_1{:}\beta'$.

  For the first approach, we again make use of a handwritten set $\Sigma_{\text{IB}} \subseteq \Sigma_{\text{E}}$ of allowed concepts, this time consisting mostly of adpositions, and we consider only cases where $\min(A_G(\sigma_1)) < \min(A_G(\beta_1))$. Furthermore, we require that the word to be inserted is located between the phrase corresponding to σ_1 and the phrase corresponding to β_1 in the reference realization. That means, we consider only words with indices in the range $(\max(A_G(\sigma_1)), \min(A_G(\beta_1))$ as insertion candidates. From right to left, we check for each index i in the above range whether

w_i is not aligned to any vertex (i.e. $\{v \in V_G \mid (v,i) \in A_G\} = \emptyset$) and $w_i \in \Sigma_{\text{IB}}$. If this is the case, we return Insert-Between-(w_i, left); if no such index is found, we return No-Insertion. However, as soon as we encounter some word w_i that is aligned to some other child β' of σ_1 (i.e. $\beta' \in \{v \in \text{ch}_G(\sigma_1) \mid (v,i) \in A_G\}$) while iterating over i, we assume that all words to the left of w_i should be inserted between σ_1 and β' rather than between σ_1 and β_1 and immediately return No-Insertion.

For our alternative approach, we use the dependency tree D to align edges to corresponding insertions in advance and store these alignments in a set $A_{\text{IB}} \subseteq E \times [|w^{\mathsf{POS}}|]$. This is done as follows: For each vertex $v \in V_D$ with $\text{pa}_D(v) \neq \emptyset$ and $\text{ch}_D(v) \neq \emptyset$ that does not correspond to any vertex of G, i.e. $\pi^2_{\mathcal{B}}(v) = \emptyset$, we check whether there is some pair $(p_v, c_v) \in \text{pa}_D(v) \times \text{ch}_D(v)$ such that the AMR vertices corresponding to p_v and c_v are connected through some edge. In other words, we search for some edge $e = (v_1, l, v_2) \in E_G$ such that

$$\exists (p_v, c_v) \in \text{pa}_D(v) \times \text{ch}_D(v) \colon v_1 \in \pi^2_{\mathcal{B}}(p_v) \wedge v_2 \in \pi^2_{\mathcal{B}}(c_v)\,.$$

If such an edge is found, then we add $(e, A_D(v))$ to A_{IB} and continue with the next dependency tree vertex. Otherwise, we check whether some edge $e' = (v_2, l, v_1)$ with the required property exists and, if so, add $(e', A_D(v))$ to A_{IB}. If this is also not the case, we extend our search radius and consider not only all parents and children of v, but also its grandparents and grandchildren. At runtime, we must then simply check whether the edge e connecting σ_1 and β_1 is aligned to some word index i through A_{IB}. If this is not the case, No-Insertion is returned; otherwise, we return Insert-Between-$(w(i), d)$ where

$$d = \begin{cases} \mathsf{left} & \text{if } i < \min(A_G(\beta_1)) \\ \mathsf{right} & \text{otherwise.} \end{cases}$$

For both approaches, if $\text{gold}'_{\mathcal{B}}(\text{Insert-Between}, c) \neq$ No-Insertion, we denote by $\text{ind}_{\mathcal{B}}(\text{Insert-Between}, c)$ the index i of the word which triggered the insertion.

This concludes our discussion of the oracle algorithm; we are now able to extract the correct transition to be applied next from a bigraph $\mathcal{B}$ of the extended corpus and a corresponding configuration c. As a next step, we describe how the bigraph $\mathcal{B}$ is updated after applying this gold transition. For this purpose, let $\mathcal{B} = (G, D, w^{\mathsf{POS}}, A_G, A_D)$, $c = (G, \sigma_1{:}\sigma, \beta, \rho) \in C_{\text{AMR}}$, $t \in T_{\text{AMR}}$ and $G = (V, E, L, \prec)$. Furthermore, let $t(c) = (G', \sigma', \beta', \rho')$ where $G' = (V', E', L', \prec')$. Then

$$\text{update}(\mathcal{B}, c, t) = (G', D, w^{\mathsf{POS}}, A'_G, A_D)$$

where depending on the class $\mathcal{C}(t)$ of the transition applied, the new alignment A'_G between G' and w^{POS} can be obtained by distinguishing the following cases:

- If $\mathcal{C}(t) =$ Merge, then σ_1 must have exactly one parent p_{σ_1} and the application of t merges σ_1 and p_{σ_1} into a single vertex. To reflect this in the alignment, we set

$$A'_G = A_G \setminus \{(\sigma_1, i) \mid i \in [|w^{\mathsf{POS}}|]\} \cup \{(p_{\sigma_1}, i) \mid (\sigma_1, i) \in A_G\}\,.$$

- If $\mathcal{C}(t) \in \{\textsc{Insert-Child}, \textsc{Insert-Between}\}$, then a new vertex is inserted into the graph, so $V' = V \cup \{\tilde{\sigma}\}$ for some vertex $\tilde{\sigma} \in V_{\text{ins}}$. This vertex must be aligned to the word which triggered its insertion. We set

$$A'_G = A_G \cup \{(\tilde{\sigma}, \text{ind}_{\mathcal{B}}(\mathcal{C}(t), c))\}\,.$$

- If $\mathcal{C}(t) \notin \{\textsc{Merge}, \textsc{Insert-Child}, \textsc{Insert-Between}\}$, i.e. none of the above cases applies, we leave the alignment unchanged and set $A'_G = A_G$.

The procedures used by the training data algorithm are now fully specified. In order to obtain a complete sequence T_{comp} of training data, we join together the sequences $T = \text{trainingData}(\mathcal{B})$ for each element $\mathcal{B}$ of C_{ext}. As probabilities for Realize and Reorder transitions are modeled slightly different from the rest, two final modifications must be made to this sequence T_{comp}: Firstly, each tuple $(c, \textsc{Realize-}(w, \alpha))$ is removed from T_{comp} and the tuple $((c, \alpha), \textsc{Realize-}(w, \alpha))$ is added to a new sequence $T_{\textsc{Real}}$. This is done because the probabilities of Realize transitions are estimated by a separate maximum entropy model $p_{\textsc{Real}}$ introduced in Eq. (14) and in accordance with this model, we may assume the correct syntactic annotation for Realize transitions to be known. Secondly, we remove each pair (c, t) with $\mathcal{C}(t) = \textsc{Reorder}$ from T_{comp} and extract from it the sequences of training data required for training the maximum entropy models introduced in Eq. (19). To this end, let $t = \textsc{Reorder-}(v_1, \ldots, v_n)$ and $c = (G, \sigma_1{:}\sigma, \beta, \rho)$. Then there is some $k \in [n]$ such that $\sigma_1 = v_k$. The following sets containing pairs of contexts and corresponding outputs are extracted from (c, t):

$$\begin{aligned}
S_* &= \{(c, v_i \lessdot \sigma_1) \mid 1 \leq i < k\} \cup \{(c, \sigma_1 \lessdot v_i) \mid k < i \leq n\} \\
S_l &= \{((c, v_i \lessdot \sigma_1, v_j \lessdot \sigma_1), v_i \lessdot v_j) \mid 1 \leq i < j < k\} \\
S_r &= \{((c, \sigma_1 \lessdot v_i, \sigma_1 \lessdot v_j), v_i \lessdot v_j) \mid k < i < j \leq n\}
\end{aligned}$$

For $i \in \{*, l, r\}$, the sets S_i extracted from all tuples in T_{comp} of the above form are collected and joined to a new sequence T_i; this sequence is then used to train the maximum entropy model p_i introduced in Eq. (19). Analogously, the sequence $T_{\textsc{Real}}$ is used to train $p_{\textsc{Real}}$. For the maximum entropy model p_{TS} introduced in Eq. (10), which handles all remaining transitions, the tuples remaining in T_{comp} are used as training data.

To train all of the above maximum entropy models, we proceed exactly the same as for the syntactic annotation models (see Section 4.3.2). That is, we specify a set of indicator features from which we extract feature candidates that are then greedily composed to a final feature sequence with which the model is trained. As indicator features, we use the same features as for our syntactic annotation models (see Table 5) as well as some additional ones. These additional indicator features can be found in Table 6; all of them are parametrized with some vertex v. It is important to note that both the relevance and the definiteness of all our features depends heavily on the transitions whose probability is to be obtained. For instance, we may be interested in properties of both the node σ_1 on top of the node buffer and its parent when considering Merge transitions, whereas for Insert-Between transitions, properties of σ_1 and the node β_1 on top of the child

Indicator Feature	Value
$\mathrm{Rho}_k(v)$, $k \in \mathcal{K}$	$\rho(k)(v)$
RealizationLemma(v)	The base form of $\rho(\mathsf{REAL})(v)$
RelativePosition(v)	If $v \prec p_v$ and $\rho(\mathsf{DEL})(p_v) = 0$, this is set to "left". Otherwise, if $p_v \prec v$ and $\rho(\mathsf{DEL})(p_v) = 0$, this is set to "right". If none of the above holds, this feature is set to "del".
$\mathrm{OutLabels}_S(v)$, $S \subseteq L_\mathrm{R}$	A flag indicating whether $\mathrm{OutLabels}(v) \subseteq S$
SameSideSize(v)	$\lvert\{v' \in V \mid p_v = p_{v'} \wedge (v \prec p_v \Leftrightarrow v' \prec p_v)\}\rvert$
SameSideLabels(v)	$\{l \in L_\mathrm{R} \mid \exists v' \in V\colon (p_v, l, v') \in E \wedge (v \prec p_v \Leftrightarrow v' \prec p_v)\}$
SameSideLabelsPos(v)	$\{(l, p) \in L_\mathrm{R} \times \mathcal{V}_\mathsf{POS} \mid \exists v' \in V\colon (p_v, l, v') \in E \wedge \overline{\mathrm{pos}}(L(v')) = p \wedge (v \prec p_v \Leftrightarrow v' \prec p_v)\}$
SameSidePos(v)	$\{\overline{\mathrm{pos}}(L(v')) \mid v' \in V \wedge p_v = p_{v'} \wedge (v \prec p_v \Leftrightarrow v' \prec p_v)\}$
Mergeable(v)	A flag indicating whether some MERGE transition has been applied to any vertex with the same concept and parent concept as v during training
ComplexPos(v)	For $\rho(\mathsf{POS})(v) \notin \{\mathrm{NN}, \mathrm{VB}\}$, this is equal to $\rho(\mathsf{POS})(v)$. For nouns, the value of $\rho(\mathsf{NUMBER})(v)$ is added and for verbs, this feature is a composition of $\rho(\mathsf{TENSE})(v)$, $\rho(\mathsf{VOICE})(v)$, $\mathrm{HasChild}_l(v)$ for all grammatical mood indicators l and the most likely grammatical number $n \in \mathcal{V}_\mathsf{NUMBER}$ for the first child of v connected through an edge with label ARGi, $i \in \mathbb{N}$, if such a child exists.

Table 6: Additional indicator features used for modeling the probabilities of transitions $P(t \mid c)$ where $c = (G, \sigma, \beta, \rho)$ with $G = (V, E, L, \prec)$. For $v \in V$ and $l \in L_\mathrm{C}$, p_v denotes the parent of v if $|\mathrm{pa}_G(v)| = 1$ and $\overline{\mathrm{pos}}(l)$ denotes the empirical POS tag of l (see Definition 4.14). For each indicator feature s, the value $s(G)$ is either explained textually or formally defined. If $s(G)$ is a singleton, delimiting brackets are omitted.

buffer are of relevance. Furthermore, available context information varies due to the order in which transitions are applied. For example, the POS tag assigned to a vertex is only known *after* its realization has been determined; it can therefore only be used as an indicator feature for transitions applied to it after a REALIZE transition. To handle both problems, we use varying sets of parameters for each parametrized indicator feature, depending on the considered transition; as is done by Wang et al. (2015), we also set each indicator feature to a special value NONE whenever it is not relevant or not properly defined in the current context. The actual list of relevant features for each class of transitions $\tau \in \mathcal{C}(T_\mathrm{AMR})$ can be found in the implementation (see Section 5.3.4).

We are now able to train all maximum entropy models required to estimate $P(t \mid c)$, but we make one final modification to the training procedure: To compensate for errors made by our model p_TS in an early stage of processing a node, we carry out the training procedure twice. In a first iteration, we train all models exactly as described above. In a second iteration, we slightly modify Algorithm 5: Whenever the transition to be applied next is contained within the set T_restr, we replace the call to $\mathrm{gold}_\mathcal{B}(c)$ in line 5 with

$$t^* \leftarrow \operatorname*{arg\,max}_{t \in T_\mathrm{restr}\colon c \in \mathrm{dom}(t)} P(t \mid c)$$

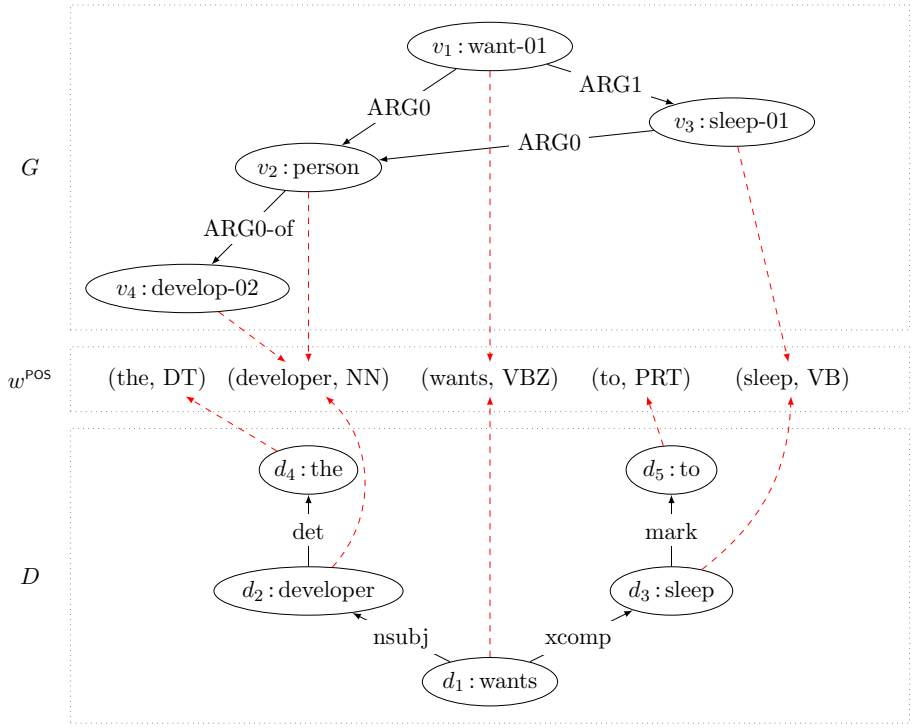


Figure 18: Graphical representation of the bigraph $\mathcal{B} = (G, D, w^{\text{POS}}, A_G, A_D)$ introduced in Example 4.15. For $i \in \{G, D\}$, each node $v \in V_i$ is inscribed with $v : L_i(v)$; each alignment $(u, j) \in A_i$ is represented by a dashed arrow line connecting u and $w^{\text{POS}}(j)$.

where P is estimated by the model trained in the first iteration. In other words, we replace gold transitions from T_{restr} with the actual output of our pretrained model. We then fuse the so-obtained training data sequence with the sequence obtained in the first run and retrain all maximum entropy models using this combined sequence.

We conclude this section with a comprehensive exemplary application of the training data algorithm; this application also includes several runs of the oracle algorithm. As this requires frequent switching between both algorithms, we abbreviate each line l of an algorithm a by $(a{:}l)$; for example, (6:3) refers to the third line of Algorithm 6.

Example 4.15 We consider a POS-annotated and lowercased version of the bigraph $\mathcal{B}_1$ introduced in Example 3.16. For reasons of consistency with the notation used throughout this section, we additionally rename its components and obtain the bigraph $\mathcal{B} = (G, D, w^{\text{POS}}, A_G, A_D)$ with $G = (V_G, E_G, L_G, \prec_G)$ and $D = (V_D, E_D, L_D, \prec_D)$ shown in Figure 18. We walk through Algorithm 5 with $\mathcal{B}$ as an input step by step and show how the set trainingData($\mathcal{B}$) is obtained.

The first step of the training data algorithm is to initialize $T = \varepsilon$ and to compute

$$c_{\text{sAMR}}(G) = (G, (v_4, v_2, v_3, v_1), \varepsilon, \rho) \text{ where } \rho = \{(k, \emptyset) \mid k \in \mathcal{K}\}$$

which is stored in a variable c (5:3). As c is not a terminal state, the algorithm calls routine gold$_{\mathcal{B}}(c)$ to obtain the gold transition to be applied next. In this subroutine, it is

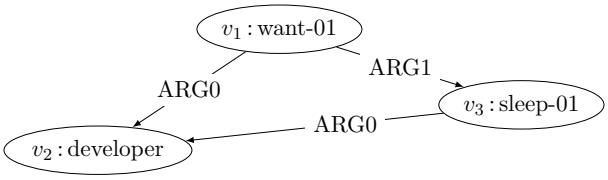


Figure 19: Graphical representation of the AMR graph $G_1 = (V_{G_1}, E_{G_1}, L_{G_1}, \prec_{G_1})$. Each node $v \in V_{G_1}$ is inscribed with $v : L_{G_1}(v)$.

first determined that node v_4 has only one parent and thus, no DELETE-REENTRANCE transition needs to be applied (6:3). Also, as v_4 is aligned to some word, it must not be deleted (6:6). It is then tested whether v_4 and its parent node v_2 have a common realization (6:8). As this is the case, the gold transition to be applied next belongs to the class MERGE and as $\text{gold}'_{\mathcal{B}}(\text{MERGE}, c) = \text{MERGE-}(\text{developer}, \text{NN})$, the value returned by $\text{gold}_{\mathcal{B}}(c)$ is likewise $t^* = \text{MERGE-}(\text{developer}, \text{NN})$. The training tuple (c, t^*) is appended to T (5:6), $\mathcal{B}$ is updated by removing all alignments involving v_4 (5:7) and c is updated by applying t^* (5:8), resulting in the new configuration

$$c \leftarrow (G_1, (v_2, v_3, v_1), \varepsilon, \rho_1)$$

where $\rho_1 = \rho[\mathsf{POS}(v_2) \mapsto \text{NN}, \mathsf{INIT\text{-}CONCEPT}(v_2) \mapsto \text{person}]$ and G_1 is shown in Figure 19.

As c is still no terminal configuration, the next transition is determined by calling $\text{gold}_{\mathcal{B}}(c)$. Because v_2 has two parent nodes, v_1 and v_3, a DELETE-REENTRANCE transition needs to be applied (6:3). For both the text-based and the dependency-tree-based approach, $\text{gold}'_{\mathcal{B}}(\text{DELETE-REENTRANCE}, c)$ returns $\text{DELETE-REENTRANCE-}(v_3, \text{ARG0})$, indicating that $e = (v_3, \text{ARG0}, v_2)$ needs to be removed from E_{G_1}. For the text-based approach, this is the case because the path from v_3 to $\text{root}(G_1)$ is longer than the path from v_1, making v_1 the gold parent of v_2 (see Definition 4.13). For the approach using D, the reason is that d_2, the dependency tree vertex corresponding to v_2, is a child of d_1 (which corresponds to v_1), but not a child of d_3 (which corresponds to v_3). After $t^* = \text{DELETE-REENTRANCE-}(v_3, \text{ARG0})$ is returned, (c, t^*) is added to the sequence T of training data (5:6), $\mathcal{B}$ is updated (5:7) and by application of t^* (5:8), the new configuration

$$c \leftarrow (G_2, (v_2, \tilde{v}_1, v_3, v_1), \varepsilon, \rho_2)$$

is obtained where $\rho_2 = \rho_1[\mathsf{LINK}(\tilde{v}_1) = v_2]$ and G_2 is shown in Figure 20 on the left.

In the next iteration, neither DELETE-REENTRANCE nor DELETE transitions are applicable for the same reasons as in the very first iteration. There is no need for a MERGE transition as v_2 and v_1 do not have a common realization (6:8). No SWAP is required because no word aligned to v_1 is between two words belonging to the span of v_2 (6:10). The oracle algorithm therefore returns $t^* = \text{KEEP}$ (6:13). Again, (c, t^*) is added to T, the bigraph is updated and t^* is applied whereby the new configuration

$$c \leftarrow (G_2, (v_2, \tilde{v}_1, v_3, v_1), \varepsilon, \rho_3)$$

with $\rho_3 = \rho_2[\mathsf{DEL}(v_2) \mapsto 0]$ is obtained; as KEEP only modifies the DEL flag, this configuration is almost identical to the previous one.

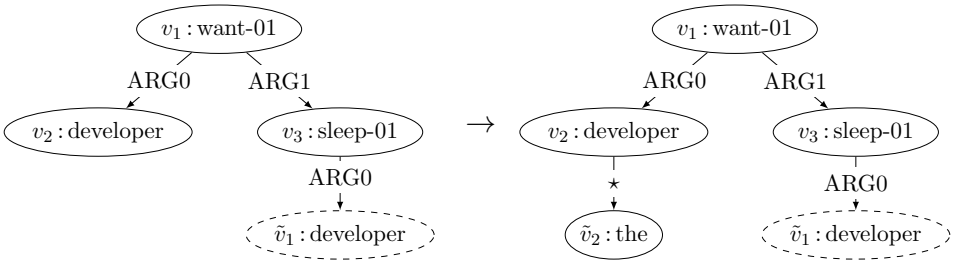


Figure 20: Graphical representation of the AMR graph $G_2 = (V_{G_2}, E_{G_2}, L_{G_2}, \prec_{G_2})$ and the graph $G_3 = (V_{G_3}, E_{G_3}, L_{G_3}, \prec_{G_3})$ obtained from G_2 through a INSERT-CHILD-(the, left) transition. For $i \in \{2, 3\}$, each node $v \in V_{G_i}$ is inscribed with $v : L_{G_i}(v)$.

At its next call, the oracle algorithm returns $t^* = \text{REALIZE-}(\text{developer}, \sigma_{v_2})$ where in accordance with Figure 17 (Section 4.3.2),

$$\sigma_{v_2} = \{(\mathsf{POS}, \text{NN}), (\mathsf{DENOM}, \text{the}), (\mathsf{TENSE}, -), (\mathsf{NUMBER}, \text{singular}), (\mathsf{VOICE}, -)\}$$

is the gold syntactic annotation for v_2. The tuple (c, t^*) is added to T, $\mathcal{B}$ is updated and t^* is applied, resulting in the configuration

$$c \leftarrow (G_2, (v_2, \tilde{v}_1, v_3, v_1), \varepsilon, \rho_4)$$

where ρ_4 is obtained from $\rho_3[\mathsf{REAL}(v_2) \mapsto \text{developer}]$ by setting $\rho_4(k)(\sigma_1) = \sigma_{v_2}(k)$ for all $k \in \mathcal{K}_{\text{syn}}$. Yet another call of the oracle algorithm returns $t^* = \text{INSERT-CHILD-}(\text{the}, \mathsf{left})$, regardless of which approach for $\text{gold}'_{\mathcal{B}}(\text{INSERT-CHILD}, c)$ is chosen (6:17). For the text-based approach, this is the case because w_1 ("the") is not aligned to any vertex and occurs directly left of w_2 ("developer"), the first word aligned to v_2 in the reference realization. For the approach using D, the sets

$$C = \{v \in V_D \mid \exists v' \in \pi^1_{\mathcal{B}}(v_2) \colon v \in \text{ch}_D(v')\} = \{d_4\}$$
$$I = \{i \in [n] \mid \exists v \in C \colon \pi^2_{\mathcal{B}}(v) = \emptyset \wedge i = A_D(v)\} = \{1\}$$

are computed and $t^* = \text{INSERT-CHILD-}(\text{lem}(w(j)), d)$ is returned where $j = \min(I) = 1$, $\text{lem}(w(1)) = \text{lem}(\text{the}) = \text{the}$ and $d = \mathsf{left}$ as $1 < \min(A_G(v_2)) = 2$.

As before, we update T and $\mathcal{B}$ and apply t^* to obtain

$$c \leftarrow (G_3, (\tilde{v}_2, v_2, \tilde{v}_1, v_3, v_1), \varepsilon, \rho_5)$$

where $\rho_5 = \rho_4[\mathsf{DEL}(\tilde{v}_2) \mapsto 0, \mathsf{INS\text{-}DONE}(\tilde{v}_2) = 1]$ and G_3 is shown in Figure 20 on the right. We leave further study of the remaining steps to the reader, but we provide in Table 7 a list of all gold transitions returned by the oracle algorithm in subsequent calls, assuming that in each call of $\text{gold}'_{\mathcal{B}}$, the approach which makes no use of the dependency tree D is chosen to obtain the gold transition whenever two alternative approaches are defined. △

σ	β	**Gold Transition**
$\tilde{v}_2 : (v_2, \tilde{v}_1, v_3, v_1)$	ε	REALIZE-(the, $\sigma_{\tilde{v}_2}$) where $\sigma_{\tilde{v}_2} = \{$(POS, DT), (DENOM, –), (TENSE, –), (NUMBER, –), (VOICE, –)$\}$
$\tilde{v}_2 : (v_2, \tilde{v}_1, v_3, v_1)$	ε	REORDER-($\tilde{v}_2$)
$v_2 : (\tilde{v}_1, v_3, v_1)$	ε	NO-INSERTION
$v_2 : (\tilde{v}_1, v_3, v_1)$	ε	REORDER-($\tilde{v}_2, v_2$)
$v_2 : (\tilde{v}_1, v_3, v_1)$	$\tilde{v}_2$	NO-INSERTION
$\tilde{v}_1 : (v_3, v_1)$	ε	DELETE
$\tilde{v}_1 : (v_3, v_1)$	ε	REORDER-($\tilde{v}_1$)
$v_3 : (v_1)$	ε	KEEP
$v_3 : (v_1)$	ε	REALIZE-(sleep, σ_{v_3}) where $\sigma_{v_3} = \{$(POS, VB), (DENOM, –), (TENSE, –), (NUMBER, –), (VOICE, active)$\}$
$v_3 : (v_1)$	ε	NO-INSERTION
$v_3 : (v_1)$	ε	REORDER-($\tilde{v}_1, v_3$)
$v_3 : (v_1)$	$\tilde{v}_1$	NO-INSERTION
v_1	ε	KEEP
v_1	ε	REALIZE-(wants, σ_{v_1}) where $\sigma_{v_1} = \{$(POS, VB), (DENOM, –), (TENSE, present), (NUMBER, –), (VOICE, active)$\}$
v_1	ε	NO-INSERTION
v_1	ε	REORDER-(v_2, v_1, v_3)
v_1	$v_2 : (v_3)$	NO-INSERTION
v_1	v_3	INSERT-BETWEEN-(to, left)
ε	ε	–

Table 7: Gold transitions returned by the oracle algorithm when processing the configuration $c = (G_3, (\tilde{v}_2, v_2, \tilde{v}_1, v_3, v_1), \varepsilon, \rho_5)$. The contents of the node buffer σ and the child buffer β before application of each transition are specified.

4.4 Postprocessing

To further improve the quality of the realizations produced by our generator, we carry out several postprocessing steps. For doing so, we make use of both the actual realization $\tilde{w} = \text{generate}(G)$ obtained from the input AMR graph G and the final configuration from which this realization is inferred. While there may be several more useful postprocessing steps, we restrict ourselves here to revising inserted articles, adding punctuation and removing duplicate words from the realization.

In the following, let $\hat{c} = (\hat{G}, \varepsilon, \varepsilon, \hat{\rho})$ with $\hat{G} = (\hat{V}, \hat{E}, \hat{L}, \hat{\prec})$ be the final configuration obtained in line 8 of Algorithm 4 for input G. As a first postprocessing step, we revise all inserted articles and check whether further articles need to be inserted. It makes sense to perform this revision as articles are added through CHILD-INSERTION transitions; at the time these transitions are applied to a node, its context (i.e. the words to its left and right in the final realization) is generally still unknown. We therefore simply check for each $v \in \hat{V}$ with $\hat{\rho}(\mathsf{POS})(v) = \text{NN}$ whether removing or inserting an article improves the score assigned to $f_{\text{AMR}}(\hat{c})$ through our language model. To this end, we first remove from $\hat{G}$ each child of v whose label is an element of the set $\langle \text{art} \rangle = \{\text{a}, \text{an}, \text{the}\}$. We then compute a linear combination of the language model score and the syntactic annotation probabilities of the so-obtained graph $\hat{G}'$ and compare this score with the scores of the graphs obtained from $\hat{G}'$ by inserting a new vertex $\tilde{v}$ with some realization from the set $\langle \text{art} \rangle$ as the leftmost child of v. From all of these graphs, we choose the one with the highest score and update the final configuration $\hat{c}$ accordingly.

Since all punctuation marks are removed from the AMR corpus during preparation in Section 4.3.1, our generator does not learn to insert them. To fix this problem, we use a rather simple, non-probabilistic approach for which we consider the set

$$\hat{R} = \begin{cases} \text{ch}_{\hat{G}}(\text{root}(\hat{G})) & \text{if } \hat{L}(\text{root}(\hat{G})) = \text{multi-sentence} \\ \{\text{root}(\hat{G})\} & \text{otherwise} \end{cases}$$

that, in most cases, just contains the root of $\hat{G}$. However, some AMR graphs encode not just one, but multiple sentences; this is indicated through a special concept "multi-sentence" for the root node. Therefore, whenever the root of $\hat{G}$ is labeled "multi-sentence", we process the subgraphs $\hat{G}|_v$ for all $v \in \text{ch}_{\hat{G}}(\text{root}(\hat{G}))$ as if they were separate graphs. For every vertex $v \in \hat{R}$, we define two predicates

$$\phi_v(\texttt{?}) = \exists v' \in \text{ch}_{\hat{G}}(v) \colon \hat{L}(v') \in \{\text{interrogative}, \text{amr-unknown}\}$$
$$\phi_v(\texttt{,}) = v \neq \text{root}(\hat{G}) \wedge \exists v' \in \text{ch}_{\hat{G}}(\text{root}(\hat{G})) \colon v \mathrel{\hat{\prec}} v'$$

from which we infer the punctuation mark for the subgraph $\hat{G}|_v$ as follows:

$$\text{punc}(v) = \begin{cases} ? & \text{if } \phi_v(\texttt{?}) \\ , & \text{if } \neg\phi_v(\texttt{?}) \wedge \phi_v(\texttt{,}) \\ . & \text{if } \neg\phi_v(\texttt{?}) \wedge \neg\phi_v(\texttt{,}) \wedge |\hat{V}| \geq 5 \\ \varepsilon & \text{otherwise.} \end{cases}$$

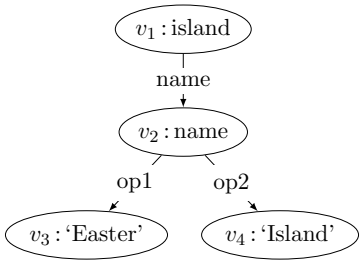


Figure 21: AMR representation of Easter Island

In other words, we assign to each subgraph $\hat{G}|_v$ the punctuation mark "?" if v has a child labeled "interrogative" or "amr-unknown" as these are the concepts used by AMR to indicate questions. We assign the punctuation mark "," if $\hat{G}|_v$ does not encode a question and its span does not contain the rightmost word of the generated sentence. If none of the above conditions holds and $\hat{G}$ has at least five vertices, the punctuation mark "." is assigned to it. We do not append a full stop to AMR graphs with less than five vertices because these often do not represent complete sentences.

Using the above definitions, we construct a new terminal configuration c' that includes the punctuation marks to be inserted. To this end, we require a set of new vertices $V_{\text{punc}} = \{v_{\text{punc}} \mid v \in \hat{R}\}$ such that $V_{\text{punc}} \cap \hat{V} = \emptyset$. We set the realization of each vertex v_{punc} to the punctuation mark assigned to $\hat{G}|_v$ and modify $\hat{\prec}$ such that this punctuation mark is the rightmost word of the subgraph's realization. More formally, we define $c' = (G', \varepsilon, \varepsilon, \rho')$ where

$$
\begin{aligned}
G' &= (\hat{V} \cup V_{\text{punc}}, E', L', \prec') \\
E' &= \hat{E} \cup \{(v, \star, v_{\text{punc}}) \mid v \in \hat{R}\} \\
L' &= \hat{L} \cup \{(v_{\text{punc}}, \text{punc}(v)) \mid v \in \hat{R}\} \\
\prec' &= (\hat{\prec} \cup \{(v', v_{\text{punc}}) \mid v \in \hat{R}, v' \in \text{ch}_{\hat{G}}(v) \cup \{v\}\})^+ \\
\rho' &= \hat{\rho}[\mathsf{REAL} \mapsto \hat{\rho}(\mathsf{REAL}) \cup \{(v_{\text{punc}}, \text{punc}(v)) \mid v \in \hat{R}\}]
\end{aligned}
$$

and compute $\tilde{w} = f_{\text{AMR}}(c')$.

As a final postprocessing step, we remove duplicate words from $\tilde{w}$. That is, whenever a word appears twice in a row in $\tilde{w}$, one of both instances is discarded. Such realizations with duplicate words are occasionally generated by our system due to named instances whose concept shares a common word with its name. An example of such a named instance can be seen in Figure 21, where the English word "island" is both the concept of vertex v_1 and part of its name, possibly resulting in the lower-case realization "easter island island" for the whole AMR graph.

4.5 Hyperparameter Optimization

Throughout the previous sections, we have introduced several hyperparameters. These parameters include, for example, real-valued weights θ_τ, $\tau \in \mathcal{C}(T_{\text{AMR}})$ for transitions

and tuples $(n, r) \in \mathbb{N}^+ \times \mathbb{R}_0^+$ for pruning. In this section, we will give a short overview on how these parameters can be obtained.

To simplify the optimization task, we regard each k-dimensional hyperparameter $\theta \in \mathbb{R}^k$, $k \in \mathbb{N}$, as a sequence of k one-dimensional hyperparameters. Let $n \in \mathbb{N}$ be the total number of such one-dimensional hyperparameters used in our generation pipeline. As $\mathbb{N} \subseteq \mathbb{R}$, we can write each possible assignment of values to all hyperparameters as a sequence $\Theta = (\theta_1, \ldots, \theta_n) \in \mathbb{R}^n$. To evaluate a particular such assignment Θ, we simply use the development set of an AMR corpus and calculate the Bleu score that the generation algorithm achieves if for all $i \in [n]$, the i-th hyperparameter is set to θ_i; we denote the obtained score by $\text{score}_{\text{Bleu}}(\Theta)$. We are then interested in the highest-scoring assignment

$$\hat{\Theta} = \arg\max_{\Theta \in \mathbb{R}^n} \text{score}_{\text{Bleu}}(\Theta) \,.$$

Two commonly used algorithms to approximate the solution to the above equation are *grid search* and *random search*. While the first algorithm defines a set $V_i = \{v_i^1, \ldots, v_i^m\}$, $m \in \mathbb{N}$ of possible values for each hyperparameter i and then performs an exhaustive search over all possible assignments, the latter samples random assignments for a predefined number of times. As reported by Bergstra and Bengio (2012), random search is in general the more efficient of both approaches, especially if the number of hyperparameters is high or the evaluation of a hyperparameter set is an expensive operation. We therefore first perform a random search and then try to locally optimize single hyperparameters in the best assignment found during random search.

To reduce the search space, we introduce for each $i \in [n]$ an interval $r_i = [\min_i, \max_i]$ with $\min_i \leq \max_i$ and $\min_i, \max_i \in \mathbb{R}$ that specifies both the minimum and the maximum value that can be assigned to the i-th hyperparameter. We then sample several uniformly distributed vectors $(\theta_1, \ldots, \theta_n) \in r_1 \times \ldots \times r_n$ and take the highest-scoring such vector $(\hat{\theta}_1, \ldots, \hat{\theta}_n)$ as an initial assignment. Afterwards, we iterate over all $i \in [n]$ and look whether the total score of vector $(\hat{\theta}_1, \ldots, \hat{\theta}_n)$ can be improved by changing only $\hat{\theta}_i$. To this end, we introduce yet another parameter $s \in \mathbb{N}^+$ and try replacing $\hat{\theta}_i$ by all values contained within the set

$$V_i = \{\min_i + j \cdot \frac{\max_i - \min_i}{s} \mid 0 \leq j \leq s\} \,.$$

In other words, we try $s + 1$ values uniformly distributed between $\min(i)$ and $\max(i)$.

For a list of all required hyperparameters and further details on the implementation of this hyperparameter optimization algorithm, we refer to Section 5.3.3.

5 Implementation

We now describe our implementation of the transition-based generator. This implementation is written entirely in Java, a relatively fast high-level programming language that is also used by most of the external libraries required by our generator. It is worth nothing that our implementation occasionally differs to some extent from the algorithms and formal definitions given in Section 4. While some modifications actually improve the output of our generator, the vast majority thereof is solely due to reasons of efficiency. For example, we do not train a single maximum entropy model p_{TS} to estimate $P(t \mid c)$ for all transitions $t \in T_{\text{AMR}}$ with $\mathcal{C}(t) \notin \{\textsc{Reorder}, \textsc{Realize}\}$, but instead train independent models for each of the stages identified in Figure 13 (Section 4.2.2); this makes the training process both faster and more memory efficient by reducing the number of training data per model. However, the most important changes in terms of the generator's actual output are that firstly, we enforce several constraints with regards to the applicability of transitions and secondly, we provide *default realizations* in order to cope with AMR concepts not seen during training.

In the following, we will first discuss all enforced transition constraints in Section 5.1 and the embedding of default realizations in Section 5.2. Subsequently, we provide a description of the implementation's overall structure and selective Java classes in Section 5.3. An overview of external libraries used by our generator is given in Section 5.4. For a more quick and practical introduction on how to use the generator, we refer to the instructions found in the implementation's `README.html` file (see Appendix B).

5.1 Transition Constraints

For each class $\tau \in \mathcal{C}(T_{\text{AMR}})$, we implement several constraints limiting the number of configurations given which transitions from τ are applicable. For our discussion of these constraints, let $c = (G, \sigma_1{:}\sigma, \beta, \rho)$ be the current configuration of our transition system where $G = (V, E, L, \prec)$. If σ_1 has only a single parent node, we denote the latter by p_{σ_1}. The constraints for each class of transitions are as follows:

- Swap: We allow this transition only if σ_1 is not a copy of some other node, i.e. $\sigma_1 \notin \text{dom}(\rho(\textsf{LINK}))$. We do so because copies created through Delete-Reentrance transitions can not have any children of their own and thus, the projectivity of yield does not constitute a problem. Furthermore, we demand that σ_1 is not a named entity; this can be verified by checking whether there is some $v \in \text{ch}_G(\sigma_1)$ with $L(v) = \text{name}$. As a final constraint, we demand that σ_1 and p_{σ_1} have not already been swapped in any previous transition step.

- Merge: During training, we store for each pair (p_{σ_1}, σ_1) of merged vertices all concepts and POS tags assigned to them. From these data, we construct a lookup table

  $$L_{\text{M}} \colon L_{\text{C}} \times L_{\text{C}} \nrightarrow \Sigma_{\text{E}}^* \times \mathcal{V}_{\textsf{POS}}$$

  mapping each pair of parent and child labels to the tuple of concept and POS tag observed most often. For instance, the lookup table obtained from training with

the LDC2014T12 corpus (see Section 3.3.2) contains, among others, the following entries:

$$L_{\mathrm{M}}(\text{early}, \text{more}) = (\text{earlier}, \text{JJ}) \qquad L_{\mathrm{M}}(\text{likely}, -) = (\text{unlikely}, \text{JJ})$$
$$L_{\mathrm{M}}(\text{thing}, \text{achieve-01}) = (\text{achievement}, \text{NN}) \qquad L_{\mathrm{M}}(\text{person}, \text{hunt-01}) = (\text{hunter}, \text{NN})$$

We then restrict the number of allowed MERGE transitions as follows: Whenever $(L(p_{\sigma_1}), L(\sigma_1)) \notin \mathrm{dom}(L_{\mathrm{M}})$, i.e. vertices with the same labels as σ_1 and p_{σ_1} have never been merged during training, we disallow all kinds of MERGE transitions. Otherwise, we allow only MERGE-$L_{\mathrm{M}}(L(p_{\sigma_1}), L(\sigma_1))$, the MERGE transition observed most often for the given pair of labels. As in the case of SWAP transitions, we additionally disallow MERGE transitions whenever σ_1 is a copy of some other node or a named entity.

- DELETE: Again, we disallow DELETE transitions for named entities. Although copies created through DELETE-REENTRANCE are often not represented in the generated sentences, we also disallow DELETE transitions if $\sigma_1 \in \mathrm{dom}(\rho(\mathsf{LINK}))$. This is because the realization of such copies is handled exclusively through default realizations as described in Section 5.2.

- REALIZE: We implement several restrictions with regards to syntactic annotations; the main purpose of these restrictions is to make the process of computing and storing syntactic annotations more efficient. Whenever a REALIZE-(w, α) transition is applied, the following must hold:

$$\begin{aligned} \alpha(\mathsf{POS}) \neq \text{VB} &\Rightarrow \alpha(\mathsf{TENSE}) = \alpha(\mathsf{VOICE}) = - \\ \alpha(\mathsf{POS}) \neq \text{NN} &\Rightarrow \alpha(\mathsf{NUMBER}) = \alpha(\mathsf{DENOM}) = - \\ \alpha(\mathsf{NUMBER}) = \text{plural} &\Rightarrow \alpha(\mathsf{DENOM}) \neq \text{a}\,. \end{aligned}$$

  To further improve the efficiency of our implementation, whenever the concept represented by σ_1 is not a PropBank frameset,[18] we require that $\alpha(\mathsf{POS}) = \widehat{\mathrm{pos}}(L(\sigma_1))$, i.e. we assign to σ_1 the POS tag most frequently observed for concept $L(\sigma_1)$ during training (see Definition 4.14). This restriction stems from the observation that for most concepts which are not PropBank framesets, almost all reasonable realizations have the same simplified part of speech. For example, it is almost always the case that instances of the concepts "boy", "city" and "world" are realized as nouns and instances of "early", "rich" and "fast" are realized as adverbs or adjectives. If $\sigma_1 \in \mathrm{dom}(\rho(\mathsf{LINK}))$, we only allow REALIZE-$(w, \alpha)$ if w is one of the default realizations assigned to c and α (see Section 5.2).

  In our implementation of Algorithm 3, we do not consider all possible syntactic annotations when computing the n_1-best REALIZE transitions. Instead, we only consider the n_k-best values for each syntactic annotation key $k \in \mathcal{K}_{\mathrm{syn}}$ where $n_k \in \mathbb{N}$ is some hyperparameter.

[18] Whether a vertex $v \in V$ represents a PropBank frameset can easily be determined by checking whether $L(v)$ matches the regular expression `[A-z]+-[0-9]+`.

- INSERT-CHILD: We allow at most one INSERT-CHILD transition per vertex and we only allow vertices to be inserted left of σ_1; both restrictions are purely on grounds of efficiency. Furthermore, we manually handle insertions of articles and auxiliary verbs required by passive constructions as these can directly be inferred from the syntactic annotation values $\rho(\mathsf{DENOM})(\sigma_1)$ and $\rho(\mathsf{VOICE})(\sigma_1)$, respectively.

- REORDER: As the number of possible reorderings for some vertex v grows super-exponentially with the number of its children, we implement several constraints to reduce the number of reorderings to be considered. Let REORDER-$(v_1, \ldots, v_n)$ be the REORDER transition whose applicability is to be checked and let

$$\lessdot = \{(v_i, v_j) \mid 1 \leq i < j \leq n\}$$

  denote the total order such that $(v_1, \ldots, v_n)$ is the $(\mathrm{ch}(\sigma_1) \cup \{\sigma_1\})$-sequence induced by $\lessdot$. If σ_1 has some child c_{σ_1} with $L(c_{\sigma_1}) \in \{\text{the, a, an}\}$, we demand that c_{σ_1} occurs before σ_1 and all of its other children, i.e. $c_{\sigma_1} = v_1$. For enumerations and listings, we require that the order defined through edge labels of the form OPi, $i \in \mathbb{N}$ be preserved. In other words, if σ_1 has children $c_1, \ldots, c_m$ where each child c_i is connected to σ_1 through an edge with label OPi, we demand that $c_j \lessdot c_k$ for all $1 \leq j < k \leq m$. We implement several more such restrictions; for a full list thereof, we refer to Section 5.3.3.

- INSERT-BETWEEN: We restrict the allowed labels for vertices inserted through left and right INSERT-BETWEEN transitions to two handwritten sets W_{left} and W_{right}, containing the insertions observed most frequently during training as well as common English prepositions (see Section 5.3.5). As children connected to σ_1 through an edge with label "domain" almost always require a INSERT-BETWEEN-(w, right) transition with $w \in \langle \text{be} \rangle$, we handle this special case manually.

5.2 Default Realizations

As some AMR concepts are either not observed at all during training or only some specific forms thereof are observed (for example, a verb may occur in the training corpus only in past tense), we provide default realizations $\tilde{r}_{(c,\alpha)}$ for some pairs $(c, \alpha) \in C_{\text{AMR}} \times \mathcal{A}_{\text{syn}}$. Given some configuration $c = (G, \sigma_1{:}\sigma, \varepsilon, \rho)$ in which REALIZE transitions are applicable, we then set

$$P(\text{REALIZE-}(\tilde{r}_{(c,\alpha)}, \alpha) \mid c, \alpha) = \tilde{p}$$

for all $\alpha \in \mathcal{A}_{\text{syn}}$ where $\tilde{p} \in [0, 1]$ is some hyperparameter; in order to assure that P is still a valid probability measure, we subtract a small amount δ from the probabilities of all other applicable REALIZE transitions.

Let the current configuration be of the form $c = (G, \sigma_1{:}\sigma, \varepsilon, \rho)$ with $G = (V, E, L, \prec)$ and let $\alpha \in \mathcal{A}_{\text{syn}}$ be a syntactic annotation for σ_1. If σ_1 is a noun, verb, adjective or adverb according to α and not a copy of some other node, i.e. $\alpha(\mathsf{POS}) \in \{\text{NN, VB, JJ}\}$ and $\sigma_1 \notin \mathrm{dom}(\rho(\mathsf{LINK}))$, we determine $\tilde{r}_{(c,\alpha)}$ as follows: If $L(\sigma_1)$ is a PropBank frameset, we first remove the frameset id from it; for example, we turn the instances "want-01" and

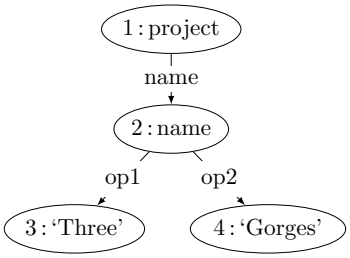


Figure 22: AMR representation of the "Three Gorges" project

"develop-02" into "want" and "develop", respectively. Let l_{σ_1} denote the so-obtained truncated label. We query WordNet (Fellbaum, 1998; Miller, 1995) to find out whether a word with lemma l_{σ_1} and POS tag $\alpha(\mathsf{POS})$ exists; if this is not the case, no default realization $\tilde{r}_{(c,\alpha)}$ can be found. Otherwise, we use SimpleNLG (Gatt and Reiter, 2009) to turn l_{σ_1} into the required word form according to α. This is done by first instantiating a *phrase* consisting only of l_{σ_1} and then specifying *features* of this phrase. For example, the number of a noun can be set to some value `num` as follows:

```
phrase.setFeature(Feature.NUMBER, num);
```

The so-obtained word is then returned as a default realization $\tilde{r}_{(c,\alpha)}$. For $\alpha(\mathsf{POS}) = \text{JJ}$, if l_{σ_1} can serve as both an adjective and an adverb, both forms are used as default realizations with probabilities of $\tilde{p}/2$ each. For example, given $l_{\sigma_1} = \text{quick}$, both "quick" and "quickly" are returned.

If $\alpha(\mathsf{POS}) \notin \{\text{NN, VB, JJ}\}$, we check whether l_{σ_1} is a pronoun and if so, we provide both the corresponding personal pronoun and possessive pronoun forms as default realizations, each with probability $\tilde{p}/2$. Importantly, this is also done if σ_1 is a copy of some other vertex, but in this case, we make use of yet another hyperparameter $p_\varepsilon \in [0, \tilde{p}]$, set the probabilities of both realizations to $(\tilde{p} - p_\varepsilon)/2$ and add ε as another default realization with probability p_ε. If none of the above applies and $\sigma_1 \in \text{dom}(\rho(\mathsf{LINK}))$, we return only ε as a default realization.

Apart from this basic handling of unknown instances and pronouns, we also provide special realization rules for named entities (i.e. vertices with a child labeled "name"), dates and numbers. For named entities, we remove all vertices encoding the name from the AMR graph and keep only the concept itself, for which we allow three different kinds of default realizations: nothing but the name, the name followed by the concept and the concept followed by the name. For instance, consider the AMR graph shown in Figure 22. As this graph represents a named entity, we remove from it all vertices but the root, for which we provide the three default realizations "Three Gorges", "Three Gorges project" and "project Three Gorges". If the named entity has already been observed during training, we choose from these three candidates the realization assigned to it most often to be the default realization. Otherwise, if at least the concept of the named entity has already been observed during training, we choose the arrangement observed most often for this concept. If neither the name nor the concept were observed during

training, we take only the name itself as the default realization. An exception to the above rules are countries, world regions and continents, for which the default realizations are both the name and the corresponding adjective, each with probability $\tilde{p}/2$.[19] For example, an instance of the AMR concept "country" with name "France" gets assigned the default realizations "France" and "French".

Date entities are converted to month-day-year format, resulting in strings like "April 2 2016" or "July 24 2011". Finally, numbers that are not part of a date are converted to ordinal numbers if their parent is an instance of the concept "ordinal-entity" and otherwise left as is, but if they end with six or nine zeros, the latter are replaced by the string "million" or "billion", respectively.

5.3 Packages

Our implementation of the transition-based generator is divided into five packages `main`, `dag`, `ml`, `gen` and `misc`. For each of these packages, we discuss here only the most important classes contained therein and the functionality they provide; for a thorough description of all classes and functions, we refer to the *Javadoc* documentation available in the `javadoc` subdirectory of our implementation.

5.3.1 `main`

The `main` package consists only of the two classes `PathList` and `AmrMain`. While the former contains nothing but string constants referring to the paths of training, development and test data, trained maximum entropy models and various external resources, the latter provides wrapper functions for the most important tasks to be performed by our implementation: Generation, training and hyperparameter optimization can be performed using the methods `generate()`, `train()` and `optimizeHyperparams()`, respectively. While the first method can be called with an arbitrary list of AMR graphs as parameter, the other methods require the training and development corpora to be found in the directories specified in `PathList`. Assuming that they are stored in official AMR format,[20] AMR graphs can be read from a file using the `loadAmrGraphs()` function.

To train the generator using `train()`, each subdirectory of the training directory (specified in `PathList.AMR_SUBDIRECTORIES` and `PathList.TRAINING_DIR`, respectively) must contain all information required to build an extended corpus (see Section 4.3.1), but this information is to be distributed among several files. These files must go by the following names specified in `PathList` and should contain the following information:

- `PathList.AMR_FILENAME`: This file must contain a list of aligned and tokenized AMR graphs, separated by empty lines and encoded using the official AMR format. The alignments must be stored in the format used by Flanigan et al. (2014).[21]

[19] The adjective forms corresponding to countries and nations are extracted from `en.wikipedia.org/wiki/List_of_adjectival_and_demonymic_forms_for_countries_and_nations`.

[20] See `github.com/amrisi/amr-guidelines/blob/master/amr.md` for a description of this format.

[21] See `github.com/jflanigan/jamr/blob/Generator/docs/Alignment_Format.md` for a description of this format.

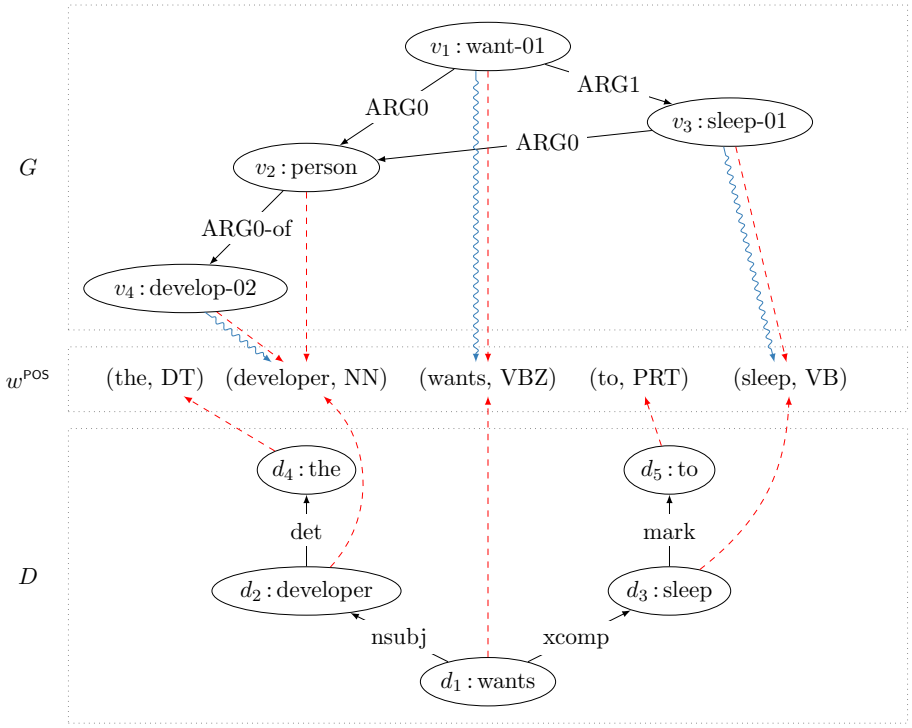


Figure 23: Graphical representation of the bigraph $\mathcal{B} = (G, D, w^{\text{POS}}, A_G, A_D)$ as described in Example 4.15. For $i \in \{G, D\}$, each node $v \in V_i$ is inscribed with $v : L_i(v)$; each alignment $(u, j) \in A_i$ is represented by a dashed arrow line connecting u and $w^{\text{POS}}(j)$. An additional alignment $A'_G \subseteq V_G \times [|w^{\text{POS}}|]$ is indicated through wavy arrow lines.

Above each AMR graph, there must be a line starting with `# ::tok` containing a tokenized reference realization and a line starting with `# ::alignments` containing the alignments. Additional annotations – such as the non-tokenized reference realization – are allowed, but ignored during the training procedure. For example, the AMR graph shown in Figure 23, its reference realization and the corresponding alignment A_G may be represented like this:

```
# ::tok the developer wants to sleep
# ::alignments 1-2|0.0+0.0.0 2-3|0 4-5|0.1
(v1 / want-01
      :ARG0 (v2 / person
            :ARG0-of (v4 / develop-02))
      :ARG1 (v3 / sleep-01
            :ARG0 v2))
```

- `PathList.DEPENDENCIES_FILENAME`: This file must contain a list of dependency trees which correspond to the AMR graphs found in the above file in a one-to-one manner. The dependency trees must be separated by empty lines and encoded in

Stanford dependencies (SD) format.[22] To give an example, the dependency tree shown in Figure 23 can be encoded as follows:

```
root(ROOT-0, wants-3)
nsubj(wants-3, developer-2)
xcomp(wants-3, sleep-5)
det(developer-2, the-1)
mark(sleep-5, to-4)
```

- `PathList.POS_FILENAME`: This file should contain a newline-separated list of POS sequences where POS tags are separated by spaces. The i-th sequence of POS tags must correspond to the reference realization of the i-th AMR graph found in the `PathList.AMR_FILENAME` file. The following entry corresponds to the reference realization shown in Figure 23:

```
DT NN VBZ PRT VB
```

- `PathList.EM_ALIGNMENTS_FILENAME`: This file should contain a newline-separated list of alignments in the format used by the string-to-string aligner described in Pourdamghani et al. (2014).[23] The i-th alignment must correspond to the reference realization of the i-th AMR graph found in the `PathList.AMR_FILENAME` file. For example, the entry encoding the additional alignment A'_G shown in Figure 23 may look like this:

```
1-1.1.1 2-1 4-1.2
```

The training procedure requires at least 8GB of RAM and may take several hours to days, depending on the used hardware. It is important to note that when training the generator with the `train()` method on a different corpus than LDC2014T12, some of the resources found in directory `res` must also be rebuilt using the corresponding methods provided by `misc.StaticHelper`. For more information on this process, we refer to the Javadoc documentation of the latter class and to `README.html` (see Appendix B).

Our implementation also supports the command-line based generation of English sentences from AMR graphs. For generation using the command line, the following parameters may be specified:

- `--input` (`-i`): The file in which the input graphs are stored in official AMR format. If this parameter is not specified, it is assumed that the required AMR graphs can be found in the subdirectories of the `PathList.TEST_DIR` file.

- `--output` (`-o`): The file in which the generated sentences should be saved. This is the only mandatory parameter.

[22] See `nlp.stanford.edu/software/stanford-dependencies.shtml` for a description of this format.
[23] Note that this format differs slightly from the one used by Flanigan et al. (2014).

- `--bleu` (`-b`): If this flag is set, the Bleu score achieved by the generator on the given data set is printed to the standard output stream. This is only possible if the AMR graphs are stored with tokenized reference realizations in the input file.
- `--show-output` (`-s`): If this flag is set, pairs of reference realizations and corresponding generated sentences are printed to the standard output stream once the generator is finished. Again, this can only be done if the AMR graphs are stored with tokenized reference realizations in the input file.

As the generation process requires around 8GB of RAM, the generator should always be run with parameter `-Xmx8g`. For example, the command

```
java -jar -Xmx8g AmrGen.jar --input in.txt --output out.txt --bleu
```

can be used to generate sentences from all AMR graphs found in `in.txt`, write them to `out.txt` and print the obtained Bleu score to the standard output stream.

5.3.2 `dag`

This package contains classes that are closely related to labeled ordered graphs as introduced in Definition 3.1. Most importantly, the class `DirectedGraph` is used to model actual graphs; their vertices and edges are represented by instances of `Vertex` and `Edge`, respectively.

Although they could theoretically be modeled using just the above classes, a wrapper class `DependencyTree` is used to represent dependency trees and a class `Amr` is used to represent AMR graphs. Bigraphs are not explicitly modeled; instead, AMR graphs simply store a reference to the corresponding dependency tree. If given, the `Amr` class also stores the reference realization of the graph and the corresponding alignment as well as POS tags. Furthermore, it provides some convenient methods and functions for the handling of AMR graphs. For example, the `calculateSpan()` method can be used to calculate the span of each vertex and `yield()` implements both $\text{yield}_{(G,\rho)}$ and $\text{yield}^{\text{par}}_{(G,\rho)}$. Another important method provided by this class is `prepare()` and its subroutines `prepareForTesting()` and `prepareForTraining()`, which prepare an AMR graph either for training or testing; this preparation includes, among others, collapsing named entities into a single node for more efficient processing, converting the reference realization to lower case and computing the span of each vertex. The `prepareForTraining()` method also defines all alignment rules mentioned in Section 4.3.1.

In addition to the above functionality, the package `dag` provides two classes `AmrFrame` and `DependencyTreeFrame` which provide means of visualizing both dependency trees and AMR graphs; these classes are also capable of showing alignments between graphs and their realizations as well as annotations assigned to vertices.

5.3.3 `gen`

This package constitutes the core of our generator. The actual generation algorithm is implemented in the classes `FirstStageProcessor` and `SecondStageProcessor`. The

former contains a method `processFirstStage()` which implements the restricted version of the greedy generation algorithm, applying only transitions from the set T_{restr} to its input; the latter contains the rest of the logic required by the generation algorithm. Most importantly, it contains a function `getBest()`, which is a straightforward implementation of Algorithm 3, the best transition sequence algorithm. Default realizations as defined in Section 5.2 and required by this method are provided by the `getDefaultRealizations()` function of class `DefaultRealizer`. A full list of restrictions for REORDER transitions can be found in class `PositionHelper`, which also contains a method to compute n-best reorderings. Finally, the `postProcess()` method of class `PostProcessor` can be used to perform postprocessing as described in Section 4.4.

For training the various maximum entropy models required by our generator, the non-instantiable classes `GoldSyntacticAnnotations` and `GoldTransitions` contain static methods to obtain gold syntactic annotation values and gold transitions, respectively. These classes implement all approaches devised in Sections 4.3.2 and 4.3.3, with the sole exception of DELETE-REENTRANCE transitions, for which only the text-based approach is implemented. This is the case because a qualitative analysis of several dozen AMR graphs from the LDC2014T12 corpus showed both approaches to give almost identical results, but this approach performed slightly better than the dependency-tree-based approach and is much easier to implement.

Hyperparameters used throughout the generation process are managed by the classes `Hyperparam` and `IntHyperparam`; the former also contains methods to perform random search and grid search as explained in Section 4.5. For a list of all hyperparameters and a short explanation thereof, we refer to the documentation of the `Hyperparams` class.

5.3.4 `ml`

This package contains all classes related to maximum entropy modeling. As mentioned before, we do not use a single maximum entropy model p_{TS} to estimate $P(t \mid c)$ for all transitions $t \in T_{\text{AMR}}$, but instead train independent such models for each stage identified in Figure 13 (Section 4.2.2). On grounds of efficiency, we additionally use two different maximum entropy models for INSERT-BETWEEN transitions: The model implemented by `ArgInsertionMaxentModel` is queried whenever the vertex on top of the node buffer is connected to its child through a PropBank semantic role (i.e. the edge connecting both vertices has a label of the form ARGi for some $i \in \mathbb{N}$); in all other cases, we use the model implemented by `OtherInsertionMaxentModel`.

All classes representing maximum entropy models can be identified by their common suffix `MaxentModel`; they are subclasses of either `OpenNlpMaxentModelImplementation`, an implementation of maximum entropy models based on the `GISModel` class provided by OpenNLP, or `StanfordMaxentModelImplementation`, an implementation using the Stanford Classifier.[24] The `IndicatorFeature` interface and its two implementations `StringFeature` and `ListFeature` provide means of representing features.

[24] For further details on OpenNLP and the Stanford Classifier, we refer to `opennlp.apache.org` and `nlp.stanford.edu/software/classifier.shtml`, respectively.

5.3.5 `misc`

The package `misc` contains miscellaneous classes whose methods are used in various places throughout the implementation. For example, the class `PosHelper` provides the simplify mapping defined in Section 4.3.2 and `PrunedList` implements the function prune_n as introduced in Definition 4.9. The class `StaticHelper` contains functions for generating additional resources required by the generator, such as the lookup table L_{M} for MERGE transitions introduced in Section 5.1. The `WordNetHelper` class provides an interface to WordNet (Fellbaum, 1998; Miller, 1995). Importantly, the class `WordLists` contains several collections of words required by the generator; for example, the words allowed for INSERT-BETWEEN and INSERT-CHILD transitions are defined therein.

5.4 External Libraries

Our implementation makes use of several external libraries for various purposes such as POS tagging, language modeling, maximum entropy modeling and computing Bleu scores. Below, we list all external libraries embedded into our generator and briefly explain how they are used:

- The *Extended Java WordNet Library* (available at `extjwnl.sourceforge.net`) is used to access WordNet (Miller, 1995; Fellbaum, 1998) which, in turn, is required for default realizations and to compute some features of our maximum entropy models.
- We use both the *Apache OpenNLP* library (available at `opennlp.apache.org`) and the *Stanford Classifier* (available at `nlp.stanford.edu/software/classifier.shtml`) for maximum entropy modeling; while the training procedure provided by the former library is both faster and more memory-efficient, we achieved slightly better results using the latter.
- The *Berkeley Language Model* (Pauls and Klein, 2011) is used for computing score_{LM}, the language model score assigned to generated sentences. It provides methods for efficiently loading and accessing large n-gram language models.
- For POS tagging of our training and development data, we use the *Stanford Log-linear Part-Of-Speech Tagger* (Toutanova et al., 2003), a part of the *Stanford CoreNLP* toolkit (Manning et al., 2014).
- *SimpleNLG* (Gatt and Reiter, 2009) is used to determine default realizations.
- We use the `BleuMetric` implementation of *Phrasal* (Spence Green and Manning, 2014) to compute the Bleu score obtained by our generator.
- To graphically display AMR graphs and dependency trees, we use several classes provided by *JGraphX* (available at `github.com/jgraph/jgraphx`).
- For parsing command line options, we make use of *JCommander* (available at `jcommander.org`).

6 Experiments

We evaluate our approach by studying the results of several experiments conducted using the implementation described in Section 5. For carrying out these experiments, a single machine with 8GB of RAM and a 2.40GHz Intel® Core™i7-3630QM CPU with eight cores was used; the operating system was Ubuntu 16.10.

All experiments reported in this section were performed using the LDC2014T12 corpus, containing 10,313 training AMR graphs, 1,368 development AMR graphs and 1,371 test AMR graphs (see Table 2, Section 3.3.2). The reference realizations of all AMR graphs in the training and development set were tokenized using *cdec* (Dyer et al., 2010) and annotated with POS tags using the *Stanford Log-linear Part-of-Speech Tagger* (Toutanova et al., 2003); dependency trees were obtained using the BLLIP parser (Charniak, 2000; Charniak and Johnson, 2005) and subsequently converted into the format required by our generator using the *Stanford Dependencies Converter*.[25] Alignments between AMR graphs and reference realizations were obtained using the methods by Flanigan et al. (2014) and Pourdamghani et al. (2014) and fused as described in Section 4.3.1. For language modeling, we used a 3-gram model with Kneser-Ney smoothing trained on Gigaword v1 (LDC2003T05).[26] The corresponding language model file in binary format can be found in the file `res/lm.binary` of our implementation.

We manually compared the quality of gold annotations and transitions returned by the alternative approaches devised in Sections 4.3.2 and 4.3.3 on a small number of development AMR graphs; in the vast majority of cases, both approaches returned exactly the same. However, using dependency trees to determine gold denominators turned out to be slightly more error-prone, the reason being that the automatically generated dependency trees for some realizations were themselves erroneous. For INSERT-CHILD and INSERT-BETWEEN transitions, it happened occasionally that one of both approaches returned nonsensical transitions, but it was very rarely the case that both approaches failed simultaneously. Therefore, in all of the experiments discussed below, we used the purely text-based approach to obtain gold denominators during training; for INSERT-CHILD and INSERT-BETWEEN transitions, we used both approaches concurrently, thus doubling the number of available training data. Hyperparameter optimization was performed as described in Section 4.5 with parameter $s = 15$, resulting in the configuration found in the file `res/hyperparams.txt`.

As a first experiment, we used the fully trained system to generate realizations for all AMR graphs in the development and test set of LDC2014T12 and computed the corresponding Bleu scores.[27] Our approach achieves a Bleu score of 27.4 on both the development and test set. A comparison of these results with the scores achieved by all other currently published approaches can be seen in Table 8; therein and throughout the remainder of this section, we abbreviate the tree-transducer-based approach of Flanigan

[25] For further details on the Stanford Dependencies format and the conversion process, see `nlp.stanford.edu/software/stanford-dependencies.shtml`.

[26] The used Gigaword n-gram counts are available at `www.keithv.com/software/giga/`.

[27] Throughout this section, we implicitly mean the case-insensitive 1…4-gram Bleu score with scaling factor $s = 100$, rounded to the first decimal place, whenever we speak of Bleu scores.

System	LM[a]	Corpus	$l_{\max}$	Dev	Test
Our approach	3-gram	LDC2014T12	∞	27.4	27.4
			30	28.3	28.9
JAMR-gen (2016)	5-gram	LDC2014T12	∞	22.7	22.0
PBMT-gen (2016)	5-gram	LDC2014T12	∞	27.2	26.9
TSP-gen (2016)	4-gram	LDC2015E86	30	21.1	22.4
SNRG-gen (2017)	4-gram	LDC2015E86	30	25.2	25.6
NEUR-gen (2017)	–	LDC2014T12, LDC2011T07	∞	–	29.7

Table 8: Comparison of our approach with other generators. The "LM" column lists the kind of language model used, the "Corpus" column contains the used corpora and the "$l_{\max}$" column contains the maximum number of words in the reference realization for an AMR graph to be considered for Bleu score computation. The "Dev" and "Test" columns show the Bleu scores obtained on the development and test sets, rounded to the first decimal place.

[a]All language models are trained on Gigaword; our language model is trained on Gigaword v1 (LDC2003T05) whereas JAMR-gen, TSP-gen and SNRG-gen use Gigaword v5 (LDC2011T07). For PBMT-gen, the version of Gigaword used to build the language model is not specified.

et al. (2016) by JAMR-gen, the phrase-based generator of Pourdamghani et al. (2016) by PBMT-gen, the approach of Song et al. (2016) based on a traveling salesman problem solver by TSP-gen, the synchronous node replacement grammar approach of Song et al. (2017) by SNRG-gen and the generator of Konstas et al. (2017) using a neural network architecture by NEUR-gen. Whenever available, Table 8 lists the results obtained with the LDC2014T12 corpus as this is the corpus used for our experiments, thus allowing for better comparisons than LDC2015E86.

In terms of Bleu scores, our approach performs much better than JAMR-gen, TSP-gen and SNRG-gen and slightly better than PBMT-gen, but worse than NEUR-gen. For the comparison with the TSP-gen and SNRG-gen generators, we must take into account that these systems were both trained using the LDC2015E86 corpus; while the test and development sets in this corpus are exactly the same as for LDC2014T12, it contains 6,520 additional training AMR graphs, thus giving TSP-gen and SNRG-gen a noticeable advantage. It is also important to note that the scores reported in Song et al. (2016, 2017) were obtained after removing from the development and test sets all AMR graphs whose reference realizations have more than $l_{\max} = 30$ words; this is especially relevant as longer AMR graphs are, generally speaking, more difficult to process. After removal of all AMR graphs with more than 30 words, our approach achieves scores of 28.3 and 28.9 on the development and test set, respectively, whereas TSP-gen achieves scores of 21.1 and 22.4 and SNRG-gen achieves scores of 25.2 and 25.6.

Except for NEUR-gen, the above-mentioned generators all make use of language mod-

els trained on Gigaword; however, JAMR-gen, TSP-gen, SNRG-gen and PBMT-gen use 4- or 5-gram models trained on Gigaword v5 whereas we consider only 3-grams and use Gigaword v1. As higher-order n-grams can cope with more complex sentence structures and are thus more powerful than a 3-gram model, we believe that our approach would perform even better if we replaced our 3-gram model by some higher-order model. Unfortunately, we are not able to verify this claim as neither Gigaword nor higher-order n-gram models trained on it are available free of charge; we thus have to resort to a freely available 3-gram language model trained on Gigaword v1.

The NEUR-gen system does not include a language model at all; instead, sentences from Gigaword v5 (LDC2011T07) are annotated with AMR graphs using the text-to-AMR parser described in Konstas et al. (2017) and directly embedded into the system as additional training data (see Section 2). However, only such sentences from Gigaword are used which contain exclusively words that also occur in LDC2014T12. To obtain the Bleu score of 29.7 on the LDC2014T12 test set, Konstas et al. (2017) use two million such sentences, increasing the number of training data by a factor of roughly 153. Although many of the automatically generated AMR graphs are likely to contain at least some errors, it is reasonable to assume that the improvement in Bleu score compared to other approaches is mainly due to this enormous enlargement of the training corpus. This claim is supported by the fact that using the LDC2015E86 corpus, the test set results reported by Konstas et al. (2017) lie between 22.0, when only the AMR graphs from LDC2015E86 are used, and 33.8, when 20 million annotated sentences from Gigaword are factored into the training process. For LDC2014T12, Konstas et al. (2017) unfortunately do not report the scores for the development set or for any number of included Gigaword sentences other than two million. Naturally, it would make sense to investigate whether including annotated sentences from Gigaword into the training process of our system leads to comparable improvements of our results. As mentioned above, however, Gigaword is not free of charge, making us unable to carry out this investigation.

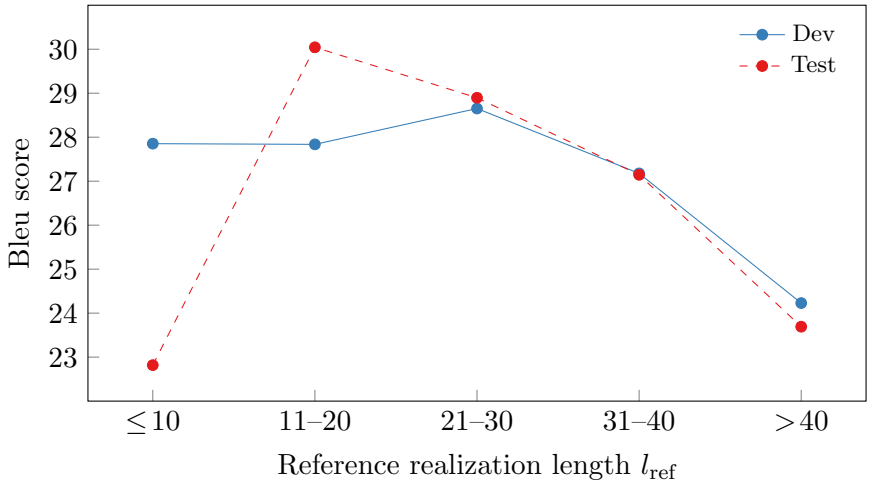
As another experiment, we evaluated our generator on several subsets of our development and test sets that contain only AMR graphs for which the number of tokens l_{ref} in the reference realization lies within a certain interval. We chose the set of intervals

$$\{[0, 10], (10, 20], (20, 30], (30, 40], (40, \infty)\}$$

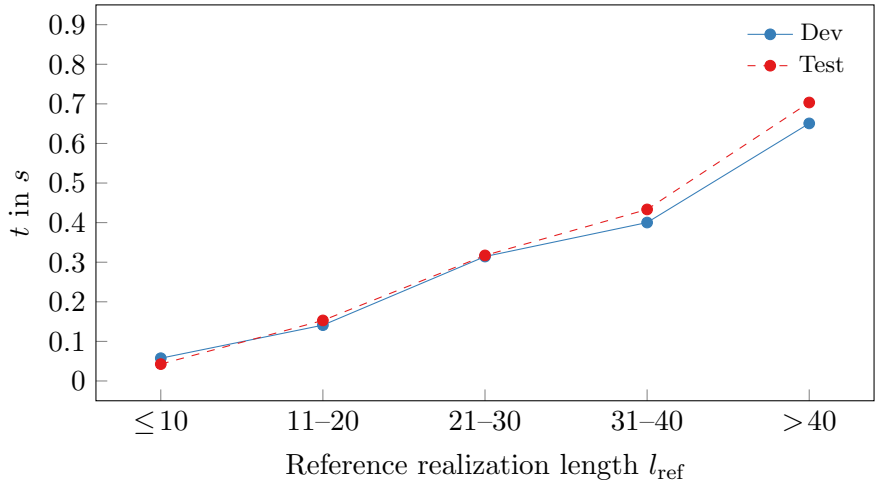
and computed the Bleu score and the average time required to process a single graph for each interval.[28] The results can be seen in Figure 24a and 24b; Figure 24c lists the number of graphs in the LDC2014T12 corpus for each of the above intervals.

Not surprisingly, the processing of AMR graphs takes more time the longer the reference realizations are, with about 0.05s required for graphs with $l_{\text{ref}} \leq 10$ and up to 0.7s required for graphs with $l_{\text{ref}} > 40$. However, it is worth noting that our implementation is by no means optimized with respect to algorithmic efficiency. For example, the processing of large graphs could massively be improved through parallelization as for vertices v and v' with $v \notin \text{succ}(v')$ and $v' \notin \text{succ}(v)$, the sets $\text{best}(v)$ and $\text{best}(v')$ required by Algorithm 4 can be computed independently.

[28] The time measurements do not include the time required to load the language model and all required maximum entropy models into memory.



(a) Case-insensitive $1, \ldots, 4$-gram Bleu score achieved by our generator on the development and test set when only AMR graphs with reference realization lengths l_{ref} in the given intervals are considered



(b) Average time required to generate a sentence from a single AMR graph in the development and test set when only AMR graphs with reference realization lengths l_{ref} in the given intervals are considered.

	Reference realization length l_{ref}				
	≤ 10	11–20	21–30	31–40	>40
Dev AMRs	255	485	374	162	92
Test AMRs	299	441	333	173	125

(c) Number of development and test AMR graphs for some values of l_{ref}

Figure 24: Performance of our transition-based generator when considering only AMR graphs for which the number l_{ref} of tokens in the reference realization is within a certain interval

With regards to the Bleu scores reported in Figure 24a, it is noteworthy that the results for $l_{\text{ref}} > 40$ are well below average, supporting our claim that a 4- or 5-gram language model might improve the Bleu score achieved by our generator as such higher order n-gram models are especially helpful for long sentences. Interestingly, however, the Bleu score of 22.8 achieved on the test set for $l_{\text{ref}} \leq 10$ is even lower than for $l_{\text{ref}} > 40$. A qualitative analysis of all AMR graphs whose reference realizations have at most ten tokens shows that this low score is mainly due to wrongly guessed punctuation marks – which can have a great impact on the Bleu score for sentences with relatively few words –, wrong date formats and errors made by our syntactic annotation models. To illustrate this, consider the following examples, where for each $i \in \mathbb{N}$, w_{r}^i denotes a reference realization provided in the LDC2014T12 test set and w_{g}^i denotes the output of our generator for the corresponding AMR graph:

$w_{\text{r}}^1 =$ 2004-12-19 $\quad$ $w_{\text{r}}^2 =$ a kathmandu police officer reports

$w_{\text{g}}^1 =$ december 19 2004 $\quad$ $w_{\text{g}}^2 =$ a report by the kathmandu police officers .

For w_{r}^1 and w_{g}^1, there are no matching n-grams at all; for w_{r}^2 and w_{g}^2, only three unigrams and one bigram match. Nonetheless, w_{g}^1 and w_{g}^2 are about equally good realizations of the corresponding AMR graphs as w_{r}^1 and w_{r}^2.

Our generator works best for AMR graphs whose reference realizations have between 11 and 30 tokens; for an example, consider the following pairs of reference realizations w_{r}^i and outputs w_{g}^i:

$w_{\text{r}}^3 =$ the story is based on the final report of the attorney general 's office .

$w_{\text{g}}^3 =$ the story is based on the attorney general 's office final report .

$w_{\text{r}}^4 =$ wen stated that the chinese government supports plans for peace in the middle east and remains firmly opposed to violent retaliation .

$w_{\text{g}}^4 =$ wen stated that the chinese government supports the plan for peace in the middle east and remains in firm opposition to the violent retaliation .

However, if there are long range dependencies, our generator often fails to find syntactically correct realizations that transfer the meaning of the corresponding graphs. This is especially the case for AMR graphs with long reference realizations, as can be seen in the below example:

$w_{\text{r}}^5 =$ the performance of the female competitors of the chinese diving team , mingxia fu and bin chi , in the first 6 rounds of the 10 - meter platform diving competition at the seventh world swimming championships held here today was ideal , and hopes of entering the heats are in sight .

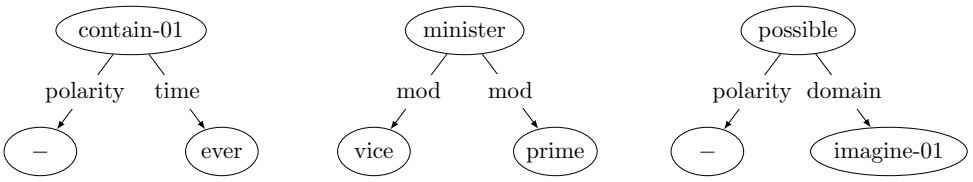
$w_{\text{g}}^5 =$ the ideal female competitors mingxia fu and bin chi of chinese diving team performance 6 first round of preliminary competition of the 10 meter platform diving at the seventh world swimming championships were held here today and hope to enter the heat is in sight .

Figure 25: Confusion matrix for transitions performed in the first phase of our generation algorithm; DELETE-REENTRANCE transitions are not included as they are always applied correctly.

As a last experiment, we looked into the individual syntactic annotations and transitions used by our generator and investigated how well the prediction of these annotations and transitions works. In accordance with our generation algorithm, we discuss the results of this investigation separately for transitions from the set T_{restr} and all remaining transitions.

For transitions contained within T_{restr}, the confusion matrix shown in Figure 25 compares the transitions applied by our generator during the processing of all development AMR graphs of LDC2014T12 with the respective gold transitions. Each entry in a row with label t_{a} and column with label t_{g} denotes the number of times a transition of class t_{a} was applied when the gold transition would have been in t_{g}; accordingly, diagonal entries correspond to correctly applied transitions. For example, 707 MERGE transitions were applied correctly and 70 MERGE transitions were applied when according to $\text{gold}_{\mathcal{B}}$, a KEEP transition should have been applied. As can be concluded from Figure 25, SWAP is by far the most error-prone transition for the first stage: It is only applied correctly in 75 cases whereas in 332 cases, a KEEP transition is applied when a SWAP transition would actually be required.

With regards to MERGE, it is noteworthy that our definition of this transition – which only allows merging nodes with their parents – makes it impossible for our generator to transform several graphs into their reference realizations. This can be seen in the three exemplary partial AMR graphs from LDC2014T12 illustrated in Figure 26: The graph in Figure 26a requires a MERGE transition among the two neighboring nodes with labels "−" and "ever" to obtain the reference realization; similarly, merging the nodes with labels "vice" and "prime" is necessary for the graph shown in Figure 26b. Even more problematic is the graph illustrated in Figure 26c, which would require us to merge all three vertices simultaneously. These examples suggest that revising the definition of MERGE transitions might be a way to improve the results obtained by our generator.



w_r = never contained
w_g = not ever contained
(a)

w_r = vice-prime minister
w_g = vice prime minister
(b)

w_r = unimaginable
w_g = can not imagine
(c)

Figure 26: Partial AMR graphs from LDC2014T12 requiring MERGE transitions among neighbors. The corresponding reference realization w_r and the output of our generator w_g in the respective contexts is given below each partial graph.

Reorderings	Dev	Test
p_*	85.34%	83.90%
p_l	84.38%	83.96%
p_r	83.26%	78.11%

Insertions	Dev	Test
p_{TS} (Stage 3)	86.32%	84.78%
p_{TS} (Stage 5)	89.71%	89.55%

Realizations	Dev	Test
$p_{\textsc{pos}}$	76.58%	74.90%
$p_{\textsc{denom}}$	80.61%	81.65%
$p_{\textsc{tense}}$	74.79%	72.49%
$p_{\textsc{number}}$	84.80%	86.00%
$p_{\textsc{voice}}$	93.35%	93.84%
$p_{\textsc{real}}$	82.28%	81.83%

Table 9: Percentage of times in which the maximum entropy models used by our generator assign the highest probability to the correct outputs when processing the development and test sets of LDC2014T12. Situations in which the correct transition or annotation is uniquely determined through the transition constraints defined in Section 5.1 are excluded.

We finally turn to an evaluation of the maximum entropy models used for syntactic annotations and all remaining transitions. Table 9 shows the percentage of times in which the transition with the highest probability according to our models was in fact the gold transition to be applied, divided into three groups. The first of these groups, headed "Reorderings" in Table 9, lists the number of times the maximum entropy models p_*, p_l and p_r assigned the highest probability to the right order between two vertices. The group captioned "Insertions" lists the percentage of correctly predicted transitions in stages 3 and 5 of Figure 13 (Section 4.2.2). We recall that in stage 3, only INSERT-CHILD and NO-INSERTION transitions can be applied whereas in stage 5, only INSERT-BETWEEN and NO-INSERTION transitions are applicable. The last group, titled "Realizations", subsumes the results obtained by all syntactic annotation models p_k, $k \in \mathcal{K}_{\text{syn}}$ and the model p_{REAL} for REALIZE transitions. The vast majority of values shown in Table 9 is above 80%, indicating that in general, the features used to train our models are well-chosen. The percentage of correctly determined POS tags on both the development and test set is comparably low; however, as can be seen in the example outputs w_g^2 and w_g^4 shown before, this does not necessarily result in bad realizations.

7 Conclusion

We have devised a novel approach for the challenging task of AMR-to-text generation. Our core idea was to turn input AMR graphs into ordered trees from which sentences can easily be inferred through application of the yield function. We chose the principle component of our approach to be the transition system S_{AMR}, whose set of transitions T_{AMR} defines how the transformation from AMR graphs to suitable trees can be performed. Some transitions contained within this set, such as MERGE, SWAP and DELETE, have an equivalent in the likewise transition-based text-to-AMR parser by Wang et al. (2015), which served as a model for our approach.

In order to turn S_{AMR} into a generator, we assigned probabilities to transitions and defined the score of a transition sequence to be a linear combination of the probabilities of all its transitions and the probability assigned to the resulting sentence by a language model. We approximated these probabilities using maximum entropy models that were trained with a set of gold transitions extracted from a large corpus of AMR graphs and corresponding realizations. As an exhaustive search for the highest-scoring transition sequence given some input would be far too time-consuming, we developed an algorithm that approximates this sequence in two phases: In a first phase, only transitions from a subset T_{restr} of T_{AMR} are greedily applied without taking the language model into consideration; in a second phase, the output of this first phase is processed bottom-up, considering multiple partial transition sequences at each step and factoring in the language model. Through parametrized pruning, we restricted the number of sequences to be considered, allowing us to find a good balance between required time and quality of the generated sentences. We introduced the concepts of syntactic annotations and default realizations to help our system decide which transition to apply next. To further improve our results, we defined some postprocessing steps – such as the insertion of punctuation marks – to revise the tree structure obtained from our transition system.

In experiments carried out using a Java-based implementation of our generator, we obtained a lower-cased $1 \ldots 4$-gram Bleu score of 27.4 on the LDC2014T12 test set, the second best result reported so far and the best without using parsed sentences from an external source such as Gigaword (LDC2011T07) as additional training data. This result strongly suggests that our transition-based transformation of AMR graphs into ordered tree structures is indeed quite a promising approach for the AMR-to-text generation task.

Throughout this work, we have highlighted a number of ways in which the results obtained by our system may further be improved upon. As outlined in Section 6, one promising way that could easily be implemented, but would require access to Gigaword, would be to replace the used 3-gram language model with some higher-order model. One could also follow the idea of Konstas et al. (2017) and annotate Gigaword sentences with AMR graphs using a parser to augment the number of available training data; as pointed out in Section 6, it is reasonable to assume that implementing this idea would have a major impact on the quality of our generator.

Another possible modification shown to be promising in Section 6 is the redefinition of MERGE transitions to allow for a merging of neighboring vertices. It is also conceivable

to modify this transition in a way that allows for vertex groups of arbitrary size to be merged. In this context, one may also investigate whether the generator could further be tweaked by revising other classes of transitions. Of course, such a revision does not have to be limited to the formal definitions of the transitions themselves, but may also be extended to the extraction of gold transitions from a training corpus as done by the oracle algorithm introduced in Section 4.3.3.

While we have put plenty of effort into the selection of suitable features for the training of our maximum entropy models, one could of course also try to improve our generator's output by adding new features extracted from the given contexts. In addition, it should be investigated whether the conditional probability $P(t \mid c)$ of a transition t given a configuration c and the various conditional probabilities of syntactic annotations can be predicted more reliably by a model more powerful than maximum entropy models. In view of recent advances in AMR generation and parsing made with neural network architectures (see van Noord and Bos, 2017; Konstas et al., 2017), especially probabilistic neural networks come to mind.

A further way to improve results may be to extend or revise the postprocessing steps introduced in Section 4.4. For instance, the assignment of punctuation marks could be refined – or even be integrated into the actual transition system – as the current output of punctuation marks by our generator shows some room for improvement, especially with respect to the placement of commas.

Yet another possibility for enhancing the quality of our generator lies in editing the current implementation in order to make it more resource-friendly and time-efficient; as outlined in Section 6, the latter could be achieved through parallelization. A time-optimized implementation may also lead to better results in terms of Bleu score, as it would allow us to both drop some of the transition constraints introduced in Section 5.1 and increase the maximum values allowed for performance-relevant hyperparameters used by the best transition sequence algorithm.

Finally, it would also be interesting to investigate in how far our results are, as claimed in Section 1, in fact transferable to other languages. As indicated in Section 4.1, this would require us to revise the concept of syntactic annotations to properly reflect the linguistic peculiarities of the considered language. Unfortunately, however, such an investigation is not feasible at present, as no sufficiently large AMR corpus is available for any other language than English.

References

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 *Proc. Linguistic Annotation Workshop*, pages 178–186.

Berger, A. L., Della Pietra, V. J., and Della Pietra, S. A. (1996). A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.

Brown, P. F., Della Pietra, V. J., Della Pietra, S. A., and Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311.

Buys, J. and Blunsom, P. (2017). Robust incremental neural semantic graph parsing. arXiv:1704.07092 [cs.CL].

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 (Volume 2: Short Papers)*, pages 748–752.

Charniak, E. (2000). A maximum-entropy-inspired parser. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*, pages 132–139.

Charniak, E. and Johnson, M. (2005). Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 173–180.

Della Pietra, S., Della Pietra, V., and Lafferty, J. (1997). Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393.

Dyer, C., Weese, J., Setiawan, H., Lopez, A., Ture, F., Eidelman, V., Ganitkevitch, J., Blunsom, P., and Resnik, P. (2010). cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proceedings of the ACL 2010 System Demonstrations*, pages 7–12.

Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. MIT Press.

Flanigan, J., Dyer, C., Smith, N. A., and Carbonell, J. (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*, pages 731–739.

Flanigan, J., Thomson, S., Carbonell, J. G., 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*, pages 1426–1436.

Gatt, A. and Reiter, E. (2009). SimpleNLG: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation*, pages 90–93.

Huang, L., Knight, K., and Joshi, A. (2006). Statistical syntax-directed translation with extended domain of locality. In *Proceedings of Association for Machine Translation in the Americas*, pages 66–73.

Jones, B., Andreas, J., Bauer, D., Hermann, K. M., and Knight, K. (2012). Semantics-based machine translation with hyperedge replacement grammars. In *Proceedings of the 24th International Conference on Computational Linguistics*, pages 1359–1376.

Kingsbury, P. and Palmer, M. (2002). From TreeBank to PropBank. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation*, pages 1989–1993.

Kneser, R. and Ney, H. (1995). Improved backing-off for m-gram language modeling. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 181–184.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., Dyer, C., Bojar, O., Constantin, A., and Herbst, E. (2007). Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, pages 177–180.

Konstas, I., Iyer, S., Yatskar, M., Choi, Y., and Zettlemoyer, L. (2017). Neural AMR: Sequence-to-sequence models for parsing and generation. arXiv:1704.08381 [cs.CL].

Langkilde, I. and Knight, K. (1998). Generation that exploits corpus-based statistical knowledge. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1*, pages 704–710.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S., and McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Miller, G. A. (1995). WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41.

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

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

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 311–318.

Pauls, A. and Klein, D. (2011). Faster and smaller n-gram language models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, pages 258–267.

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*, pages 32–41.

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*, pages 425–429.

Pourdamghani, N., Knight, K., and Hermjakob, U. (2016). Generating English from abstract meaning representations. In *Proceedings of the 9th International Natural Language Generation Conference*, pages 21–25.

Pust, M., Hermjakob, U., Knight, K., Marcu, D., and May, J. (2015). Parsing English into abstract meaning representation using syntax-based machine translation. In *Conference on Empirical Methods in Natural Language Processing*, pages 1143–1154.

Puzikov, Y., Kawahara, D., and Kurohashi, S. (2016). M2L at SemEval-2016 task 8: AMR parsing with neural networks. In *Proceedings of the 10th International Workshop on Semantic Evaluation*, pages 1154–1159.

Shen, D. and Lapata, M. (2007). Using semantic roles to improve question answering. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 12–21.

Song, L., Peng, X., Zhang, Y., Wang, Z., and Gildea, D. (2017). AMR-to-text generation with synchronous node replacement grammar. arXiv:1702.00500 [cs.CL].

Song, L., Zhang, Y., Peng, X., Wang, Z., and Gildea, D. (2016). AMR-to-text generation as a traveling salesman problem. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2084–2089.

Spence Green, D. C. and Manning, C. D. (2014). Phrasal: A toolkit for new directions in statistical machine translation. In *Proceedings of the 9th Workshop on Statistical Machine Translation*, pages 114–121.

Tesnière, L. (1959). *Eléments de syntaxe structurale*. Librairie C. Klincksieck.

Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, pages 173–180.

van Noord, R. and Bos, J. (2017). Neural semantic parsing by character-based translation: Experiments with abstract meaning representations. arXiv:1705.09980 [cs.CL].

Wang, C., Xue, N., and Pradhan, S. (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*, pages 366–375.

Zhou, J., Xu, F., Uszkoreit, H., Qu, W., Li, R., and Gu, Y. (2016). AMR parsing with an incremental joint model. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 680–689.

A List of Symbols

Symbol	Meaning	Page
$\mathcal{A}_{\text{syn}}$	the set of all syntactic annotations	23
$\mathcal{C}(T)$	the classes to which the transitions in T belong	34
C_{AMR}	the set of all configurations for AMR generation	27
$c_{f\text{AMR}}$	the finalization function used by S_{AMR}	27
$c_{s\text{AMR}}$	the initialization function used by S_{AMR}	27
$C_{t\text{AMR}}$	the set of terminal configurations used by S_{AMR}	27
$\mathcal{G}_{\text{AMR}}$	the set of all AMR graphs	11
$\mathcal{G}_{\text{DEP}}$	the set of all dependency trees	15
$\mathcal{K}$	the set of all annotation keys	26
$\mathcal{K}_{\text{syn}}$	the set of all syntactic annotation keys	23
L_{C}	the set of all AMR concept labels	10
L_{D}	the set of all dependency labels	15
L_{R}	the set of all AMR relation labels	10
S_{AMR}	our transition system used for AMR-to-text generation	27
$\mathcal{T}(S, I)$	the set of all terminating transition sequences for I in S	18
T_{AMR}	the set of transitions used by S_{AMR}	27
T_{restr}	the restricted set of transitions used in the first phase of our generation algorithm	39
$\mathcal{V}$	the set of all annotation values	26
$\mathcal{V}_k$	the set of annotation values for key $k \in \mathcal{K}$	23, 26
V_{ins}	the set of insertable vertices	28
$\mathcal{V}_{\text{syn}}$	the set of all syntactic annotation values	23
$\pi^1_{\mathcal{B}}$, $\pi^2_{\mathcal{B}}$	mappings through which vertices of a bigraph $\mathcal{B}$ with common alignments are linked	16
Σ_{E}	the set of all English words, including numbers and punctuation marks	6

B Readme File

The following is the content of the `README.html` file included in the implementation. This content is largely identical to the description of our implementation in Section 5, but places a particular emphasis on the setup and practical use of the generator.

Transition-based AMR Generator

This is a Java-based implementation of the AMR-to-text generator introduced in "Transition-based Generation from Abstract Meaning Representations". For a detailed description of all relevant classes, please refer to the Javadoc documentation found in the `javadoc` subdirectory. Running the generator requires Java Version 8 or newer.

Generation

There are two ways of generating sentences from AMR graphs using this generator: You may either use the precompiled and pretrained (using the LDC2014T12 corpus) generator's command line interface, which requires almost no time to set up but is not very flexible, or you may set up the generator as described in section *Setup* and then use the methods `loadAmrGraphs(String directory, boolean forTesting)` and `generate(List<Amr> amrs)` of class `main.AmrMain`.

For using the command line interface, the following parameters may be specified:

- `--input` (`-i`): The file in which the AMR graphs are stored in official AMR format. The AMR graphs must be separated by empty lines and there must be *two* line breaks after the last graph. If this parameter is not specified, it is assumed that the required AMR graphs can be found in the subdirectories `bolt`, `consensus`, `dfa`, `proxy` and `xinhua` of `corpus/test` (as is the case for LDC2014T12).
- `--output` (`-o`): The file in which the generated sentences should be saved. This is the only required parameter.
- `--bleu` (`-b`): If this flag is set, the Bleu score achieved by the generator on the given data set is output to the standard output stream. This is only possible if the AMR graphs are stored with tokenized reference realizations (indicated by a line beginning with `# ::tok` right above each actual AMR graph) in the input file.
- `--show-output` (`-s`): If this flag is set, pairs of (reference realization, generated sentence) are printed to the standard output stream when the generator is finished. Again, this is only possible if the AMR graphs are stored with tokenized reference realizations in the input file.

Important: Note that the generation process requires around 8GB of RAM. Therefore, the generator should always be run with `-Xmx8g` or more.

Examples

Following is the content of the file `in.txt` (line breaks are indicated through ↩):

```
(v1 / want-01 ↩
      :ARG0 (v2 / person ↩
         :ARG0-of (v4 / develop-02)) ↩
      :ARG1 (v3 / sleep-01 ↩
         :ARG0 v2)) ↩
↩
```

It is an encoding (in official AMR format) of an AMR graph used extensively in the Master's thesis. The following command generates an English sentence from this graph:

```
java -jar -Xmx8g AmrGen.jar --input in.txt --output out.txt
```

Running this command creates a new file `out.txt` which contains only a single line with content "the developer wants to sleep".

The following command generates sentences from all AMR graphs found in `some/directory/input.txt`, writes them to `some/other/directory/output.txt` and outputs the obtained Bleu score to the standard output stream:

```
java -jar -Xmx8g AmrGen.jar --input some/directory/input.txt --output
    some/other/directory/output.txt --bleu
```

The following command generates sentences from all AMR graphs found in the subdirectories `bolt`, `consensus`, `dfa`, `proxy` and `xinhua` of `corpus/test`, writes them to `some/directory/output.txt` and outputs both the Bleu score and pairs of reference realizations and generated sentences to the standard output stream:

```
java -jar -Xmx8g AmrGen.jar -o some/directory/output.txt -b -s
```

Setup

To set up the AMR generator, simply build the Maven project using `pom.xml`, which automatically loads all dependencies.

Setup using IntelliJ IDEA

Using IntelliJ IDEA (tested with IntelliJ IDEA Ultimate 2016.3 under Ubuntu 16.10, Windows 10 and OS X 10.10.5), the project can be set up as follows:

- Select **File** | **New** | **Project from Existing Sources...**
- In the "Select File or Directory to Import" dialogue, select the root folder of the implementation and click **Ok**.
- In the "Import Project" dialogue, click **Next** several times and then **Finish**.

Training

After performing the steps described above, the maximum entropy models required by the generator can be retrained using the `train()` method provided by `main.AmrMain`. This assumes that the development and training AMR graphs can be found in the subdirectories `bolt`, `consensus`, `dfa`, `proxy` and `xinhua` of `corpus/dev` and `corpus/training`, respectively. Each of these subfolders should contain the following four files:

- `data.amr.tok.aligned`: A list of aligned and tokenized AMR graphs, separated by newlines. The file must end with two line breaks. To obtain the reported results, the alignments should be created using JAMR. Above each AMR graph, there should be a line starting with `# ::tok` containing a tokenized reference realization and a line starting with `# ::alignments` containing the alignments. For example, an AMR graph may be represented like this:

```
# ::tok the developer wants to sleep
# ::alignments 1-2|0.0+0.0.0 2-3|0 4-5|0.1
(v1 / want-01
      :ARG0 (v2 / person
          :ARG0-of (v4 / develop-02))
      :ARG1 (v3 / sleep-01
          :ARG0 v2))
```

- `data.amr.tok.charniak.parse.dep`: A list of dependency trees which correspond to the AMR graphs found in the above file in a one-to-one manner. The dependency trees must be separated by empty lines and encoded in Stanford dependencies format. For example, the dependency tree corresponding to the sentence encoded by the above AMR graph may look like this:

```
root(ROOT-0, wants-3)
nsubj(wants-3, developer-2)
xcomp(wants-3, sleep-5)
det(developer-2, the-1)
mark(sleep-5, to-4)
```

- `pos.txt`: A newline-separated list of POS sequences, where POS tags are separated by spaces and each sequence corresponds in a one-to-one manner to the reference realizations of the AMR graphs in the above file. The following entry corresponds to the sentence represented by the above AMR graph:

```
DT NN VBZ PRT VB
```

- `alignments.txt`: A list of additional alignment sequences, where each sequence corresponds in a one-to-one manner to the AMR graphs in the above file. To obtain

the reported results, these alignments must be encoded in the format used by the aligner of Pourdamghani et al. (2014) found at `isi.edu/~damghani/papers/Aligner.zip` and should be obtained using this very aligner. For example, the alignment `1-2|0.0+0.0.0 2-3|0 4-5|0.1` shown above in JAMR format should be encoded as follows:

```
1-1.1 1-1.1.1 2-1 4-1.2
```

To change the naming conventions, edit the corresponding entries in `main.PathList`. To retrain only specific models, use the `setUp(List<Models> modelsToTrain, boolean stopAfterFirstStage)` method provided by `main.AmrMain`.

Important: Note that the training process requires around 8GB of RAM and may take several hours to days. Therefore, it should always be run with `-Xmx8g` or more.

Important: Note that retraining the AMR generator on a different dataset may also require you to rebuild some of the files described in section External Resources. For these files, the functions required to rebuild them are given below.

Hyperparameter Optimization

After training the classifier, hyperparameter optimization may be performed using the `optimizeHyperparams()` method provided by `main.AmrMain`. This assumes that the development AMR graphs can be found in the subdirectories `bolt`, `consensus`, `dfa`, `proxy` and `xinhua` of `corpus/dev`. For randomized hyperparameter optimization, the various kinds of update functions provided by `gen.Hyperparam` can be used.

External Resources

All external resources used by our implementation of the transition-based generator can be found in the subdirectory `res`. The paths to all of these files are defined in `main.PathList`. The external resources have the following contents:

- **res/lm.binary**: The language model to be used by the generator. This language model should be compatible with the Berkeley LM. For efficient generation, it should be in binary format. By default, this file contains a 3-gram language model trained on Gigaword (LDC2003T05) which can be found at `www.keithv.com/software/giga`.
- **res/english-bidirectional-distim.tagger**: A model file for the Stanford POS tagger used to annotate reference realizations and unknown words with POS tags.
- **res/morph-verbalization.txt**: A file containing tuples of verbs and corresponding nouns, e.g. (develop,development) or (pray,prayer). This file is obtained from `amr.isi.edu` and used for determining default realizations.

- **res/verbalization.txt**: A file containing nouns and corresponding AMR graph realizations using PropBank framesets, e.g. (`actor`, `person :ARG0-of act-01`). It is obtained from `amr.isi.edu` and used during the preparation of AMR graphs.
- **res/concepts.txt**: This file contains all concepts observed during training. It can be refilled using the `getConceptList(List<Amr> amrs)` method provided by `misc.StaticHelper`.
- **res/bestpostags.txt**: This file maps each non-PropBank concept to the POS tag observed most often in the training data of LDC2014T12. It was obtained using the `getBestPosTagsMap(List<Amr> amrs)` method of `misc.StaticHelper`.
- **res/mergemap.txt**: For each pair of vertices that has been merged during training, this file contains the resulting (realization,pos)-tuple observed most often, e.g. `(long,more)` → `(longer,JJ)`. It was obtained using the `getMergeMap(List<Amr> amrs)` method of `misc.StaticHelper`.
- **res/namedentities.txt**: This file stores realizations observed for named entities during training along with the number of times these realizations have been observed.
- **res/hyperparams.txt**: This file contains the current configuration for all hyperparameters. For more details, please refer to the Javadoc documentation of `gen.Hyperparam` and `gen.Hyperparams`.