Alpino chart generator design

Table of Contents

Introduction

This document provides a quick overview of the Alpino chart generator, and the design choices that were involved. The generator creates realizations from abstract dependency trees (ADTs). Generation is performed in the following steps:

- ADT processing
- Lexical lookup
- Chart generation
- Unpacking
- Fluency ranking

Note

The generator is still very much work in progress, so substantial parts of the design can still change.

ADT construction

Abstract dependency trees (ADTs) are constructed from a normal Alpino dependency tree, by removing certain information. Currently, the DT is transformed in an ADT in the following manner

- Particles that have the *svp* relation and that are in the frame of their head in the dependency relation are removed.
- Lexical item positions (that determine adjacency) are removed.
- Frames are converted to a part of speech tag and a set of attribute/value pairs. Some attributes can be underspecified.

- Optional punctuation is removed.

ADT processing

Before lexical lookup of lexical items is performed, the ADT is enriched with information that is useful for subsequent steps.

MWU concatenation

Words from multi-word units are represented as separate nodes a DT and ADT. But these separate nodes are constructed as a post-processing step during parsing. To be able to look up multi word units in the lexicon, they have to be combined into one item again.

Bitcoding

Every lexical item in the tree is assigned a unique bitcode. This is a single bit that uniquely identifies lexical items (e.g. the first lexical item gets bitcode *1*, the second *10*, the third *100*, etc). Bitcodes are used as a very fast check during generation to see if to-be constructed items have no overlapping lexical items.

Some lexical items need special handling, and particles in particular, since particles can be combined or separated from their head. In this case we look up how many particles an item has, and assign that many bitcodes. In a later step, where we introduce lexical items, we spread the bitcodes across the head and its particles for the separated forms.

Frame lookup

The ADT specifies lexical items as a root, a part of speech tag, and a set of attribute/value pairs. But for the lexical lookup, we need a frame for an item. During this step, we look up all possible frames for a given root/tag/attributes combination.

Feature structure construction for dependency relations

During the generation we will want to restrict the combination of items by the dependency relations specified by the ADT. Rather than checking such dependency relations when combining an edge, we inject information about the expected dependency relations into the feature structures. In that manner, combination of items is checked through unification. During this step, we simply extract the dependency relations, and build a feature structure for every head that lists its dependants. Dependants are represented at this point by their root and bitcode.

The resulting feature structures with dependency information are unified with the feature structures found for the lexical items during lexical lookup.

Lexical lookup

During lexical lookup, we collect the lexical items from the ADT and store them. Afterwards, particles and *with_dt* items are added. As a final step, the feature structures are constructed for every lexical item.

Particles

Particles are introduced for lexical items with verb frames, where the frame indicates the precense of a particle. All particles are collected from the frame, and introduced with bitcodes snooped from the additional bits assigned to the verb. After introducing the particles for a given lexical item, the bitcode for the lexical item is updated to reflect that the particle is separated.

with_dt structures

Some lexical items (usually consisting of multiple words) require specification of a more extensive dt (dependency tree) feature structure. These so-called *with_dt* entries require special handling to be recognized in the ADT. We do this by determining for category nodes whether a sublist of the roots that the category node dominates, is a sequence of roots of a with_dt entry. The with_dt inverse dictionary has roots that are sorted to be able to look up with_dt entries quickly.

Frame filtering

TODO

Chart generation

The main generator is a chart generator. At the beginning of the generation process an agenda is initialized with initial items based on the lexical items that were collected during the lexical lookup. After that, the generator processes each item on the agenda. Items can be *active* (the item still has uncompleted daughters) or *inactive* (all daughters have been completed). Inactive items can cause rule invocation (where a rule is activated to create an active or inactive item). Inactive items can also cause dot movement, where it completes a daughter, creating a new item. All new items are put on the agenda, items on the agenda that are processed are put on the chart. The item being processed at a given point can only interact with items on the chart.

To be able to apply dot movement or rule invocation, an inactive item should unify with the next daughter to be processed for a given active item. Additionally, the bitcodes of the items are combined should have no overlapping bits (otherwise the items represent overlapping lexical items).

No dot movement or rule invocation is attempted for an inactive agenda item that has semantics that are subsumed by an existing inactive item on the chart. Since the existing item subsumes the current item, it can represent that item. In such a case the item is forgotten, and the history of the item is attached to that of the subsuming item.

Packing

As described in the previous section, items for which a subsuming item is found on the chart, are packed into the existing item. Each item has a unique identifier. A history item contains the edge identifier to which the history is attached, and a list of histories that represent the original rule daughters (as item identifiers).

Unpacking

Derivations with a top category and expected bitcode and dependency structure are unpacked after generation. Since an arbitrary amount of punctuation can be introduced, we only allow the minimum number of punctuation signs required to get at least one realization (through iterative deepening).

A beam can be used to selectively unpack only the most fluent realizations from the forest. This selection is applied to histories representing tree nodes with a maximum projection. We also memoize the partial derivation trees with a maximum projection of the root node.

Fluency ranking

For a given ADT, there are often many realizations that are allowed by the grammar. But not all realizations are equally fluency. For this reason a fluency model is required to rank sentences by fluency. Currently there are two fluency models: an n-gram language model and a maximum-entropy model.

N-gram language model

The n-gram language model is trained on word trigrams from a gold standard corpus. Since quite much data is required to get proper estimations of the conditional n-gram probabilities, the n-gram model is normally trained on a relatively large corpus of written text of a good quality.

Even when training on large corpora, many word trigrams will never occur on the training data. The n-gram language model uses linear interpolation smoothing to interpolate the trigram probability from uni/bi/trigram probabilities. The probability of unknown unigrams is estimated using Witten-Bell smoothing.

Maximum entropy language model

While the n-gram language model provides a good baseline, it is relatively limited because it can only estimate the probability of a sentence using the surface string. Maximum entropy models allow us to integrate other information, such as structural information from the derivation tree.

Features for the maximum entropy model can be extracted using the *train_fluency* end-hook. Currently, the following feature templates are used, which are derived from Velldal and Oepen (2005) and Velldal (2007):

- *ngram_lm*: The n-gram language model score for a sentence.
- *lds*: Local derivation subtrees are extracted, consisting of the rule identifier of a node and the identifiers of the node daughters. This can be used with optional grandparenting, and daughter skewedness information.
- *ldsb*: Back-off for local derivation subtrees, where only one daughter is listed. This is also used with grandparenting.
- *tngramw*: Frame n-grams (N < = 3), with the surface of the last token of the n-gram.
- *tngram*: Frame n-grams (N < = 3), without the surface of the last token of the n-gram.
- *lds_dl*: Derivation tree node identifiers with binned frequencies of the number of lexical items each daughter dominates over.
- *lds_skew*: Derivation tree node identifiers with binned standard deviations of the number of lexical items each daughter dominates over.

Other feature templates that are used:

- *lds_deps*: Derivation tree node identifiers with the dependency relations listed in their dts, ordered by head positions.
- Various syntactic features from parse disambiguation.

Optimal feature weights are estimated using the TADM tool. Infrastructure for automatic training is provided in the *Generation/fluency/MaxEnt* directory.