Matthew Marting

Dependency Tree Linearization Literature Review

In this paper, I review two papers concerning linearization of dependency trees. The first introduces the concept and describes necessary data structures. The second outlines a fast algorithm and briefly touches on training.

"The First Surface Realisation Shared Task: Overview and Evaluation Results"

The purpose of the Surface Realisation (SR) Task was to compare SR systems. An SR system produces a sentence, or realization, given as an input a representation thereof. At least here, all the representations comprised nodes and edges. Nodes corresponded to lexical units. The edge of one node to another described the nature of the first node's dependence on the second.

To compare SR systems, the team organizing the SR Task established two common input formats: shallow and deep. As opposed to prior tests of SR systems, the SR Task's common input formats ensured that, within each kind of input representation (shallow or deep), each system received the same amount of information. That is, each *shallow* SR system would receive the same amount of information as all the other *shallow* SR systems; all the *deep*, the same as all the other *deep*. The team "assessed three criteria in the human evaluations: Clarity, Readability and Meaning Similarity" (Belz et al. 221), and they also scored SR systems according to "the following well-known automatic evaluation metrics: [. . .] BLEU, [. . .] NIST, [. . .] METEOR, [. . .] TER" (Belz et al. 220). Differences in systems' realization qualities would depend only on the systems themselves.

As the SR Task parsed randomly-selected sentences within a corpus, it retained their original forms and contexts. A sentence's original form would serve as a kind of "human topline" (Belz et al. 217). Perhaps we could use such a topline for training?

The shallow input format was "a more 'surfacey', syntactic representation of the sentence" (Belz et al. 218), while the deep format was "closer to a semantic, more abstract, representation" (Belz et al. 218). As mentioned above, both representations comprised nodes and edges, where nodes corresponded to lexical units. As such, each node comprised the lemma; also present for both formats were, if applicable, the tense, number, "tense and participle features and a sense tag id (as a suffix to the lemma)" (Belz et al. 218). The shallow format's nodes, unlike those of the deep format, contain a part-of-speech (POS) tag. Together with the edges they comprise a dependency tree. Shallow edges have labels that signify the grammatical role of the node they point to to the node they originate from, such as `OBJ` (for object of a verb) or `SBJ` (for subject) (Natural Language Technology Group 3). These syntactic edges may have more than one label. The strict arrangement of nodes and edges into a tree poses an issue for sentences such as "He commissions and splendidly interprets fearsome contemporary scores" (Natural Language Technology Group 3). "scores" is the object of both "commissions" and "interprets", but since only one edge may point to each node, it is only the object of "commissions" (Natural Language Technology Group 4). This is a definite disadvantage of using a tree structure, but one could parse to a tree and then perform "a post-processing step which adds some of these missing relations, producing graph representations with multiple headed dependencies" (Natural Language Technology Group 3).

Potential Implementation. One could make a `Node` class and an `Edge` class. `Node` would have a `std::wstring lemma_`. One could store the POS as either an `enum class` (given Apertium's recent adoption of the not-so-recent C++11 standard) or a `std::wstring`. If the set of course tags remains constant enough to hard-code (as in the SR task, where it was fixed), the first option would be better by far, since one could switch on the POS tag's value. Such a switch could compile to a couple comparisons to check the range and then a jump table, since letting the compiler determine the enumerators' values would make them contiguous; therefore, there would be no need to check for invalid intermediate values in the switch. This would pose an issue were it necessary to store the POS in a file, since a change in the set of enumerators might change their values, which would corrupt all existing files unless one stored the POS in a different way, such as with a `std::wstring`. One could also just set the enumerators' values manually.

Tense and number, among other attributes, would need to be `Optional`. It would not be more efficient to write a custom `Optional` for `Node`, since, while one could base whether an optional field such as tense exists off the POS tag's value, to do so would require calling an additional function, which would always be slower than just comparing a pointer with `nullptr`. One would also still need to allocate memory, which could be done rather easily in a constructor; it would call the POS function once to determine which `Optional` fields to initialize, and the fields would allocate their own memory.

Making a separate `Node` class for each POS is out of the question, since there will be so many POS tags. Should the set of POS tags change quickly enough, the second option, storing the POS in a `std::wstring`, would be better, since one could read in a new set of tags instead of recompiling. To be useful, the POS-tag-input format must include what tags would have which attributes; otherwise, this too would have to be hard-coded, but in a less-efficient (than a switch) if-else chain of string comparisons.

Each `Node` would need to have a list of all the edges that originate from it. Since the order of these edges does not matter, fast random insertion and removal are not important and would serve only to slow down other actions and use more memory. Only random access is important, so one should use a `std::vector`, which supports that and fast insertion and removal only at the end. We will need to determine whether we need to traverse the structure in reverse and, if so, how often. Either each node must have a pointer to the edge that points to it or, if we adopt a graph representation as opposed to a tree, a `std::vector` of pointers or we must implement a function to find this or these edges. Since multiple-headed nodes would be rare, the former would require memory approximately on the order of the number of nodes; most edge pointer vectors would have one element, or, if we adopt a tree representation, each node would have only one additional pointer. The complexity of the later function would depend on the number of edges originating from each node; at worst, it would be of the order of the number of nodes; at best, of the order of the logarithm of the number of nodes. Since we do not yet know the number of sentences that may be in memory at once or the order a reverse-traversal function might be called on, we cannot yet say which option is better. I suspect, however, that our SR system will work on only one sentence at a time, so the additional memory from edge pointers would be negligible.

`Edge` itself would need to have a pointer to the node that the edge that it represents points to. Under the same circumstances and for the same reasons as nodes having pointers to the edges that point to them, an edge would have a pointer to the node that it originates from or a function to find the node. A node pointer vector would not be useful, since by definition an edge originates from exactly one node. One could store an edge's relation in the same way as a node's POS: as either an `enum class` or a `std::wstring`, and for the same reasons.

“Sentence Realization with Unlexicalized Tree Linearization Grammars”

An unlexicalized linearization grammar comprises a set of linearization rules for a particular local configuration and their associated probabilities. The grammar is unlexicalized in that it does not regard lemmata. A local configuration comprises a node, all of its dependencies, and the node’s respective relations to them. For example, the `SBJ` of a `VERB` could be a `NOUN`, and the `OBJ` could be a `NOUN` likewise. A linearization rule, which describes how to order all of a local configuration’s nodes, could be `SBJ VERB OBJ`. Say this has a probability of 0.9. Another rule could be `OBJ VERB SBJ` with a probability of 0.1. Together, these would comprise the unlexicalized linearization grammar for `VERB`s with two dependencies: a `NOUN` `SBJ` and a `NOUN` `OBJ`. One could also use fine-grained POS tags.

Linearization Algorithm. The paper describes an $\mathcal{O}(\mathrm{n})$ algorithm to linearize dependency trees, where n is the number of nodes. It appears to use memory on the order of n^2. Since the algorithm is “fully generative” (Wang and Zhang 1302), it generates the n-best linearizations of the dependency tree.

There are six subroutines of in the algorithm, three of which the paper outlines (Wang and Zhang 1303–1304). The main one, `linearize-node`, takes a node root and a natural number n as parameters and returns the n-best linearizations of the dependency tree root is the head of. It does so by calling two other subroutines, `hypothesize-node` and `instantiate-hypothesis`, in a loop it executes n times. Each turn of the loop pushes the next-worse linearization of the dependency tree onto the back of the return vector. `hypothesize-node` takes root and a whole number i as parameters. `linearize-node` initializes i to zero and increments it after each call to `hypothesize-node`, which finds the ith-best (where 0^{th}-best is the best) linearization of the dependency tree in terms of the rank of each of the local configurations. Recall that for each local configuration there is a set of rules and associated probabilities: an unlexicalized tree linearization grammar. A zero would indicate that the rule with the highest probability was applied to a particular node and its dependencies; a one, the second-highest; etc. The subroutine `sorted-rules` fetches this set sorted from highest probability to lowest. A `hypothesis` stores the index of the current node's linearization in its rule set and that of all its dependencies in the vector `indices`. It also stores the associated hypotheses of all its dependent nodes in the vector `daughters`.

A node's best hypothesis is the one for which the node itself and all its dependencies use their best linearization, recursively. To find the next-worse linearization, `hypothesize-node` calls the subroutine `advance-indices`, which returns a list of possible `indices`. `hypothesize-node` then recursively finds the ith-best linearization for each of its dependencies. If the node is final, the hypothesis has an empty `indices`; if it is semi-final, its `indices`' sole element is the index of the ith-best unlexicalized linearization rule, or i. It passes each hypothesis to the subroutine `score-hypothesis`, which calculates the probability of each prospective linearization. One should probably implement this as a recursive function. The probability of any final node would be 1.0; the probability of any other would be the probability of the particular rule selected in the `indices` times the product of the probabilities of all the node's dependencies. The procedure `new-hypothesis` calls `score-hypothesis` and indexes it in the node's `agenda` by probability. Since `hypothesize-node` always pops off the hypothesis from the `agenda` with the highest probability, it always executes in incremental, ascending order of i, and for each i it inserts the next-worse group of hypotheses, the hypothesis it pops off the `agenda` will always match i. It stores this hypothesis in `hypotheses[i]` and eventually returns it.

Conclusion

The algorithm above has no need of reverse-traversal, so nodes need only hold some kind of reference to their dependencies. We will need to implement a lexical unit class, and this would have the POS. Therefore, it seems logical for a node to have some kind of reference to a lexical unit. These lexical units could be in a `std::vector` to represent a sentence. During training, one could iterate through each node, determine the order of its dependencies, and increment the frequency of the respective rule. (Recall that a rule is an ordering of a particular set of dependencies.) To determine the order would require some kind of comparison, which is impossible with references. One would either need to use pointers to the lexical units or their indices in the sentence vector. These two papers provide insight into both a data structure and algorithm for linearizing dependency trees. Besides an algorithm for linearizing dependency graphs, the only major front untouched is parsing.

Works Cited

Belz, Anja, et al. “The First Surface Realisation Shared Task: Overview and Evaluation Results.” *13th European Workshop on Natural Language Generation (ENLG)*, Sept. 2011, Nancy, France, Association for Computational Linguistics, 2011, pp. 217–226, www.aclweb.org/anthology/W11-2832. Accessed 29 Nov. 2016.

“Generation Challenges 2011 Surface Realisation Shared Task: Documentation and Instructions for Participants.” *Natural Language Technology Group*, University of Brighton, 19 Apr. 2011, www.itri.brighton.ac.uk/home/Anja.Belz/pdf/SR-Task-2011-Doc.pdf. Accessed 29 Nov. 2016.

Wang, Rui, and Yi Zhang. “Sentence Realization with Unlexicalized Tree Linearization Grammars.” *24th International Conference on Computational Linguistics (COLING 2012)*, Dec. 2012, Mumbai, India, Association for Computational Linguistics, 2012, pp. 1301–1310, www.aclweb.org/anthology/C12-2127. Accessed 1 Dec. 2016.