Matthew Marting

Dependency Tree Linearization Literature Review

"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.

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.