Grundlagen der Wissensverarbeitung
Computer Science
Universität Hamburg

Tutorial 6: Search and Parsing

Rafael Ruz Gómez
Miguel Robles Urquiza

27 November 2017

Exercise 1

(a) By what operations is the input transformed into the output? What do these operations do?

- Left-Arc: Adds an arc from the next input token to the node on top of the stack and reduces from the stack(pops).
- Right-Arc: Adds an arc from the node on top of the stack to the next input token and pushes the next input token into the stack.
- Reduce: Pops from the stack.
- Shift: Push the next input item into the stack.

(b) When does the parsing algorithm terminate?

It terminates when the list of remaining input tokens is empty, i.e. when it reaches a configuration <S,nil,A>where S represents the stack, the second element represents the empty input, and A represents the arc relation.

(c) Describe the formal properties of a dependency tree as defined in the paper.

- Single head: If there are 2 nodes having an arc to the same node, then those 2 nodes are the same node, i.e. every node has got 0 or 1 parent node, but no more than 1.
- Acyclic: For every 2 differents nodes n and n', there's never an arc from n to n' and at the same time one path(one or more arcs consecutively) from n' to n.
- Connected: For every 2 different nodes n and n', there's always a path between them (from n to n' or from n' to n).
- Projective: For every two nodes n and n' having an arc from one to the other, if there's another node n" between them (n <n" <n' means POS(n) <POS(n") <POS(n')), then there has to be a path from n or from n' to n"

For each property: Give an example dependency tree that violates the property. Note: We ask for a tree, not a sentence! Do not try to find a matching sentence to your trees, it will only distract you.

- Single head: Three nodes n0,n1 and n2. Arcs: n0->n1 , n2->n1
- Acyclic: Two nodes n0 and n1. Arcs: n0->n1 , n1->n0
- Connected: Two nodes n0 and n1. Arcs: none

- Projective: Three nodes n0,n1 and n2. Note that the list of input tokens are n0=(0,w0), n1=(1,w1) and n2=(2,w2) where the pair is consisting of a position and a word form. Then the list of arcs would be: n0->n2

Exercise 2

Try to use the proposed parser actions to produce the tree depicted in Figure 1. Write down the steps and the intermediate states.

1. Initialization . <nil, Der Mann isst eine Giraffe, ∅>
2. Shift . <Der, Mann isst eine Giraffe, ∅>
3. Left-Arc . <nil, Mann isst eine Giraffe, (Mann,Der)>
4. Shift . <Mann, isst eine Giraffe, (Mann,Der)>
5. Left-Arc . <nil, isst eine Giraffe, (Mann,Der),(isst,Mann)>
6. Shift . <isst, eine Giraffe, (Mann,Der),(isst,Mann)>
7. Shift . <eine isst, Giraffe, (Mann,Der),(isst,Mann)>
8. Left-Arc . <isst, Giraffe, (Mann,Der),(isst,Mann), (Giraffe,eine)>
9. Right-Arc . <Giraffe isst, **nil**, (Mann,Der),(isst,Mann), (Giraffe,eine), (isst,Giraffe)>

Exercise 3

The basic idea laid out in the paper by Nivre is still in use in state-of-the-art parsers such as the one coined "Parsey McParseface" by Google. While the actions are still the same, the mechanics of selecting actions are quite different. The actions are not ordered using a fixed precedence and their applicability is not restricted by lexical rules as they are introduced in the paper. Instead, search is used. If you now view parsing using the proposed actions as a search problem:

• What are the search states?

The search states are nodes composed by the set <S,I,A>as described before (S stack, I input tokens, A arcs).

• What is the start state?

The start state is a node composed by <nil, I, ∅>

• What are the end states?

The end states are those who have no more input tokens remaining, i.e. <S, nil, A>where $A \neq \emptyset$

• What are the state transitions?

The transitions are the same described in the paper :

- Left-Arc
- Right-Arc
- Reduce
- Shift

• Can the search space be created before parsing starts?

Yes, because the algorithm is deterministic. There are two kinds of transitions: Those that pops an element from the stack, and those that push an element into the stack(and then delete an element from the input). When the stack is empty, the only option is to make a "push" transition, so in the end, we will reach an end state, and so on we can create the search space before parsing starts.

• What is the advantage of the proposed algorithm in contrast to simply trying to find a good dependency tree by enumerating all possible trees and selecting the best one from them?

The efficiency in time and even more in space.

• For the search strategies discussed so far: are they a good fit for this search problem and why (not)?

It depends on the length of the input, but in fact depth-first looks like we are applying the algorithm with the priority of transitions described in the paper, so it's not a good idea to create the search space and then just do the same proposed in the paper.

Breadth-first has the same problem as always, the efficiency in time and space.

A* looks like the best option, but on the other hand I find not easy to define a good heuristic function.

• How would you design a parser using the parser actions together with an appropriate search procedure?

I would define the search space as described above and then choose A* as the search algorithm.