Transition-based parsing

Peter Ljunglöf
Department of Linguistics
Gothenburg University
`peb@ling.gu.se`

In this paper we introduce a general framework for transition-based parsing algorithms. Among the algorithms that can be described in this framework are deterministic and generalized LR-parsing (Tomita, 1986), incremental tabular parsing such as the Earley algorithm (Earley, 1970), and projective and non-projective depenency parsing (Nivre, 2008).

In the framework parsing is *not* viewed as a deductive process, as is common for tabular parsing strategies (Sikkel and Nijholt, 1997), but instead as a functional process in the spirit of recursive ascent parsing (Leermakers, 1993).

1 Automata

An automaton in this framework is a tuple $(\Sigma, \Phi, \phi_0, \Phi_F, S, R)$, where Σ is the input alphabet, Φ is a set of parsing states (forming a monoid together with the operation $\oplus$ and the zero $\bot$), $\phi_0 \in \Phi$ is the initial state, $\Phi_F \subseteq \Phi$ are the final states, and $S \in \Sigma \times \Phi \to \Phi$ and $R \in \Phi \to \Phi$ are the transition functions for the automaton (called *shift* and *reduce* respectively).

A *configuration* is a pair $\langle \beta, \phi \rangle \in \Sigma^* \times \Phi$, of the remaining input and the current state of the automaton. The basic idea of transition-based parsing is to shift one word from the remaining input to the current state, and then reduce the state any number of times. This is repeated until the input is exhausted. Reducing any number of times means that we take the union $(\oplus)$ of all possible reductions. This can be defined recursively as the auxiliary function $R^*(\phi) = \phi \oplus R^*(R(\phi))$, with the base case $R^*(\bot) = \bot$.

The parsing function $P \in \Sigma^* \times \Phi \to \Phi$ can finally be defined recursively by:

$$\begin{aligned} P(w\beta, \phi) &= P(\beta, R^*(S(w, \phi))) \\ P(\epsilon, \phi) &= \phi \end{aligned}$$

An input string $w_1 \ldots w_n$ is accepted by the automaton if $P(w_1 \ldots w_n, \phi_0) \in \Phi_F$

2 Examples

Here we give some brief examples of algorithms that have a natural formulation as a transition-based automaton. Due to space limitations, we only describe the type Φ of parsing states, and the transition functions S and R. The initial and final states, as well as correctness proofs, are left as exercises for the interested reader.

2.1 Deterministic LR(0) parsing

LR parsing uses a set Q of *LR states*, a goto relation $\mathcal{G} \subseteq \Sigma \times Q \times Q$, and a reduce relation $\mathcal{R} \subseteq Q \times \Sigma \times \mathbb{N}$. These relations are compiled from a context-free grammar (Knuth, 1965).

A parsing state $\phi \in \Phi$ is a stack of LR states, or the failure state $\bot$. When combining two stacks, $\phi \oplus \phi'$, we assume that one of them is $\bot$. Otherwise the grammar is not LR(0), and we cannot parse it deterministically. The top state of a stack ϕ is denoted $\mathrm{top}(\phi)$, and the result of popping the top state is denoted $\mathrm{pop}(\phi)$.

The shift function S is defined as $S(w, \phi) = q\phi$, if there is a q such that $\mathcal{G}(w, \mathrm{top}(\phi), q)$, $\bot$ otherwise. The reduce function R is defined as $R(\phi) = S(w, \mathrm{pop}^n(\phi))$, if there are w, n such that $\mathcal{R}(\mathrm{top}(\phi), w, n)$, $\bot$ otherwise.

2.2 Generalized LR(0) parsing

Conceptually, GLR parsing handles a set of stacks in parallel, shifting and popping on each one of them. The naive implementation leads to an exponential number of stacks, and the problem is how to store these stacks in a compact way. One way to store the stacks is to implement the parsing state $\phi \in \Phi$ as a directed acyclic graph where each node $q^{\langle i \rangle}$ is an LR state q annotated by an input position i. The topmost nodes in ϕ are denoted $\mathrm{top}(\phi)$. Popping is an operation that applies to nodes in ϕ:

$$\mathrm{pop}_\phi(q_0^{\langle k \rangle}) \quad = \quad \{q^{\langle i \rangle} \mid (q_0^{\langle k \rangle} \mapsto q^{\langle i \rangle}) \in \phi\}$$

Shifting a new token w onto the graph means that we add a new edge $[q^{\langle k+1\rangle} \mapsto q_0^{\langle k\rangle}]$ for each $q_0^{\langle k\rangle} \in \text{top}(\phi)$ and each q such that $\mathcal{G}(w, q_0, q)$:

$$\begin{aligned} S(w,\phi) &= \phi \cup \{q^{\langle k+1\rangle} \mapsto q_0^{\langle k\rangle} \mid \\ &\qquad q_0^{\langle k\rangle} \in \text{top}_\phi,\ \mathcal{G}(w, q_0, q)\} \end{aligned}$$

Reducing means that for each $q_0^{\langle k\rangle} \in \text{top}(\phi)$ such that $\mathcal{R}(q_0, w, n)$, we pop n times to get $q_n^{\langle i\rangle}$, and add a new edge $[q^{\langle k\rangle} \mapsto q_n^{\langle i\rangle}]$, where $\mathcal{G}(w, q)$:

$$\begin{aligned} R(\phi) &= \phi \cup \{q^{\langle k\rangle} \mapsto q_n^{\langle i\rangle} \mid \\ &\qquad q_0^{\langle k\rangle} \in \text{top}_\phi,\ \mathcal{R}(q_0, s, n), \\ &\qquad q_n^{\langle i\rangle} \in \text{pop}_\phi^n(q_0^{\langle k\rangle}),\ \mathcal{G}(s, q_n, q)\} \end{aligned}$$

2.3 Earley parsing

The Earley algorithm (Earley, 1970) is often formulated in a deductive setting, but it can also be regarded as a transition-based algorithm. A parsing state $\phi \in \Phi$ is a sequence of Earley states $\langle E_0, \ldots, E_k\rangle$, where each E_j is a set of Earley items $[i, A \to \alpha \cdot \beta]$. Shifting w means that we add a new Earley state E_{k+1} constisting of the items which search for w:

$$\begin{aligned} S(w, \langle E_0, \ldots, E_k\rangle) &= \langle E_0, \ldots, E_k, E_{k+1}\rangle \\ \text{where } E_{k+1} &= \{[i, A \to \alpha w \cdot \beta] \mid \\ &\qquad [i, A \to \alpha \cdot w\beta] \in E_k\} \end{aligned}$$

This new state is created by the Earley rule *shift*. The other two inference rules, *predict* and *combine*, are used when reducing the current Earley state E_k:

$$\begin{aligned} R(\langle E_0, \ldots, E_k\rangle) &= \langle E_0, \ldots, E_k \cup E_k' \cup E_k''\rangle \\ \text{where } E_k' &= \{[k, B \to \cdot\gamma] \mid \\ &\qquad [j, A \to \alpha \cdot B\beta] \in E_k, \\ &\qquad B \to \gamma\} \\ E_k'' &= \{[i, A \to \alpha B \cdot \beta] \mid \\ &\qquad [j, B \to \gamma\cdot] \in E_k, \\ &\qquad [i, A \to \alpha \cdot B\beta] \in E_j\}) \end{aligned}$$

2.4 Projective dependency parsing

Projective dependency parsing (Nivre, 2008) is not based on a context-free grammar and does not build phrase structure trees, instead it uses a trained oracle to build dependency structures. Despite being very different from the previous examples, there is a natural formulation as a transition-based automaton.

A parsing state $\phi \in \Phi$ is a pair $\langle \sigma, A\rangle$ of a stack $\sigma \in \Sigma^*$ of input tokens and a dependency graph $A \subseteq \Sigma^2 \times \mathcal{L}$, where Σ are the nodes and the edges are labelled with dependency labels $\ell \in \mathcal{L}$.

Shifting a symbol just puts it first in the stack: $S(w, \langle \sigma, A\rangle) = \langle w\sigma, A\rangle$. The result of reducting a state ϕ depends on the oracle $\mathcal{O} \in \Phi \to \{0, 1, \bot\}$. If $i = \mathcal{O}(\phi) \in \{0, 1\}$, then we continue reducing:

$$R(\langle w_0 w_1 \sigma, A\rangle) = \langle w_i\sigma, A \cup \{w_i \overset{\ell}{\mapsto} w_{1-i}\}\rangle$$

If $i = \bot$, we stop reducing by returning $\bot$.

Since the same word form can occur in several positions, we assume that each input token is labeled by its input position. The input should thus be of the form: "*the*$_1$ *cat*$_2$ *sat*$_3$ *on*$_4$ *the*$_5$ *mat*$_6$".

3 Discussion

We have introduced a framework for transition-based parsing, which incorporates many different parsing algorithms. In the present description there is no lookahead to help decide when and how to do the reduction, which many algorithms make use of. This is not a real problem, since only small straightforward changes has to be made.

All abstract frameworks hide some implementation issues that are necessary for efficiency, and so does this. In particular, the function R^* can be implemented slightly differently depending on the underlying parsing state. However, compared to deductive approaches, our framework is nevertheless closer to a practical implementation while still being an abstract framework.

References

Jay Earley. 1970. An efficient context-free parsing algorithm. *Comm. ACM*, 13(2):94–102.

Donald E. Knuth. 1965. On the translation of languages from left to right. *Information and Control*, 8:607–639.

René Leermakers. 1993. *The Functional Treatment of Parsing*. Kluwer Academic Publishers.

Joakim Nivre. 2008. Sorting out dependency parsing. In Bengt Nordström and Aarne Ranta, editors, *GoTAL'08*, Gothenburg, Sweden.

Klaas Sikkel and Anton Nijholt. 1997. Parsing of context-free languages. In G. Rozenberg and A. Salomaa, editors, *The Handbook of Formal Languages*, volume II, pages 61–100. Springer-Verlag.

Masaru Tomita. 1986. *Efficient Parsing for Natural Language*. Kluwer Academic Press.