Performance Analysis of *Grappa* Parsers for Hyperedge Replacement Grammars

Mark Minas

June 9, 2017

Abstract

This document reports on some experiments on the performance of graph parsers generated by *Grappa*.[1] In particular, it compares the performance of PDT [1] and PSR [2] parsers with the more general, but — as it turns out – slower Cocke-Younger-Kasami-style parsers [4] generated by *DiaGen*.[2] All experiments have been conducted on a MacBook Pro 2013, 2,7 GHz Intel Core i7, Java 1.8.0.

Contents

[1] *Grappa* homepage: `www.unibw.de/inf2/grappa`
[2] *DiaGen* homepage: `www.unibw.de/inf2/DiaGen`

1 Nested Triangles

Consider nonterminals S and $\blacktriangle$ and the terminal $\triangle$. We use $\ell^{x_1 \cdots x_k}$ as a a shorthand for literals $\ell(x_1, \ldots, x_k)$. (Here ε denotes the empty variable sequence.) Then the rules

$$S^{\varepsilon} \rightarrow \blacktriangle^{xyz} \tag{1}$$
$$\blacktriangle^{xyz} \rightarrow \triangle^{xuv}\, \triangle^{uyw}\, \triangle^{vwz}\, \blacktriangle^{uwv} \tag{2}$$
$$\blacktriangle^{xyz} \rightarrow \triangle^{xyz} \tag{3}$$

generate a nested triangle:

$$\begin{aligned} S^{\varepsilon} \underset{1}{\Rightarrow} \blacktriangle^{123} \underset{2}{\Rightarrow} \triangle^{145}\, \triangle^{426}\, \triangle^{563}\, \blacktriangle^{465} &\underset{2}{\Rightarrow} \triangle^{145}\, \triangle^{426}\, \triangle^{563}\, \triangle^{478}\, \triangle^{769}\, \triangle^{895}\, \blacktriangle^{798} \\ &\underset{3}{\Rightarrow} \triangle^{145}\, \triangle^{426}\, \triangle^{563}\, \triangle^{478}\, \triangle^{769}\, \triangle^{895}\, \triangle^{798} \end{aligned}$$

In Fig. 1, the graphs of this derivation are drawn as diagrams.[3]

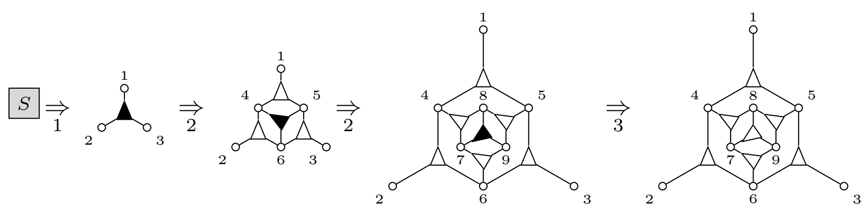
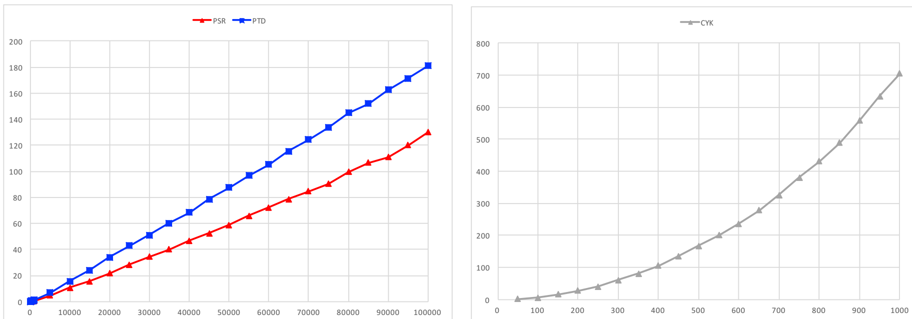


Figure 1: Diagrams of a derivation of a nested triangles. Circles represent nodes, boxes and triangles represent edges of triangle graphs, which are connected to their attached nodes by lines; these lines are ordered clockwise around the edge, starting at the sharper edge of the triangle.

Each triangle graph consists, for some positive integer n, of $3n$ nodes and $3n-2$ edges. Fig. 2a shows the runtime of the PSR and PTD parsers when processing triangle graphs with varying values of n. Runtime has been measured in milliseconds on the y-axis while n is shown on the x-axis. Note the apparent linear behavior of the PSR parser and the, slightly slower, PTD parser. Fig. 2b shows the corresponding diagram for the CYK parser. Note that the runtime of the CYK parser is not linear in the size of the triangle graph. Note also that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 700ms to parse a triangle graph with $n = 1000$ whereas the PTD parser needs just 0.97ms, and the PSR parser just 0.44ms.

[3]Thanks to Berthold Hoffmann for this description of nested triangles.



(a) (b)

Figure 2: Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for nested triangles. Note that the scales in (a) and (b) differ.

2 Nassi-Shneiderman Diagrams

We also conducted experiments with the more complicated language of Nassi-Shneiderman diagrams that represent structured programs with conditional statements and while loops. Fig. 3 shows such diagrams. Each diagram can be modelled by a graph where statement, condition, and while blocks are represented by edges of type *stmt*, *cond*, and *while*, respectively. Diagram D_1 in Fig. 3, for instance, is represented by a graph $\mathit{cond}^{abcd}\,\mathit{stmt}^{cefg}\,\mathit{stmt}^{edgh}$. The language of all *Nassi-Shneiderman graphs* is defined by an HR grammar with the following rules:

$$
\begin{array}{rcl}
S^{\varepsilon} & \to & \mathit{NSD}^{xyuv} \\
\mathit{NSD}^{xyuv} & \to & \mathit{NSD}^{xyrs}\,\mathit{Stmt}^{rsuv} \;\mid\; \mathit{Stmt}^{xyuv} \\
\mathit{Stmt}^{xyuv} & \to & \mathit{stmt}^{xyuv} \;\mid\; \mathit{cond}^{xyrs}\,\mathit{NSD}^{rmun}\,\mathit{NSD}^{msnv} \;\mid\; \mathit{while}^{xyrsut}\,\mathit{NSD}^{rstv}
\end{array}
$$

We use the shorthand notation $L \to R_1 \mid R_2$ to represent rules $L \to R_1$ and $L \to R_2$ with the same left-hand side.

Runtime of the different parsers has been measured for Nassi-Shneiderman graphs D_n with varying values of n. Fig. 3 recursively defines these graphs D_i for $i = 1, 2, 3, \ldots$ and also shows D_3 as an example. Each diagram D_i consists of $2 + 6i$ nodes and $3i$ edges.

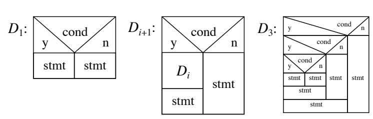
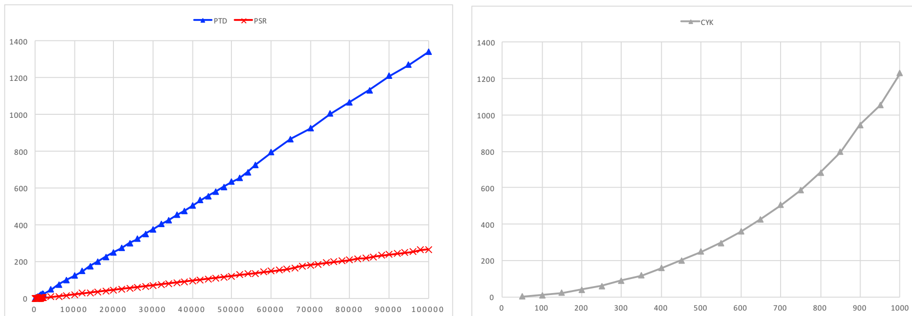


Figure 3: Nassi-Shneiderman diagrams D_i, $i = 1, 2, 3, \ldots$.



(a) (b)

Figure 4: Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for Nassi-Shneiderman graphs built as shown in Fig. 3. Note that the scales in (a) and (b) differ.

Fig. 4a shows the runtime of the PSR and the PTD parser for graphs D_n with n being shown on the x-axis and the runtime in milliseconds on the y-axis. Fig. 4b shows the corresponding diagram for the CYK parser. The PSR parser and the CYK parser have been generated from the HR grammar presented above. For generating the PTD parser, a slightly modified grammar with *merging rules* [1] had to be used because the presented grammar is not PTD.

Note that the runtime of the PSR parser and the slower PTD parser is linear in the size of the input graph whereas the runtime of the CYK parser is not linear. Note again that the scales in the diagrams shown in Fig. 4a and b differ and that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 1.2s to parse D_{1000} whereas the PTD parser needs just 12ms, and the PSR parser just 1.0ms.

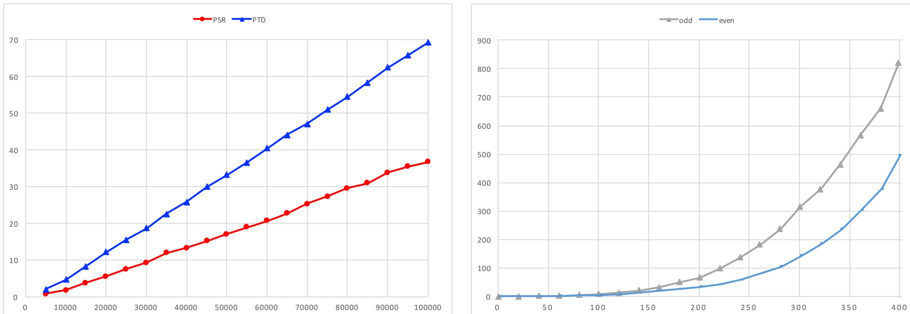
3 Palindromes

We consider palindromes, i.e., words that read the same backward as forward, over the alphabet $\{a, b\}$ and model them by string graphs using an HR grammar with the following rules:

$$\begin{array}{rcl} S^{\varepsilon} & \to & P^{xy} \\ P^{xy} & \to & a^{xu}\,a^{vy}\,P^{uv} \;\mid\; b^{xu}\,b^{vy}\,P^{uv} \;\mid\; a^{xu}\,a^{uy} \;\mid\; b^{xu}\,b^{uy} \;\mid\; a^{xy} \;\mid\; b^{xy} \end{array}$$

Note that the string language of palindromes is not deterministic and cannot be parsed by an $LL(k)$ or $LR(k)$ parser, but its string graph language is PTD as well as PSR.

In the experiment, we considered palindromes w_n of length n starting with letter a and alternating letters as long as possible, i.e., $w_1 = a, w_2 = aa, w_3 =$



(a) (b)

Figure 5: Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for palindromes. Note that the scales in (a) and (b) differ and that the upper and lower graphs in (b) show the runtime for palindromes of odd and even length, respectively.

$aba, w_4 = abba, w_5 = ababa, w_6 = abaaba, w_7 = abababa, \ldots$, and measured the runtime of the PSR parser, the PTD parser, and the CYK parser. Fig. 5a shows the runtime of the PSR and the PTD parser for palindromes w_n with n being shown on the x-axis and the runtime in milliseconds on the y-axis. Note that the runtime of the PSR parser and the slower PTD parser is linear in the size of the input graph.

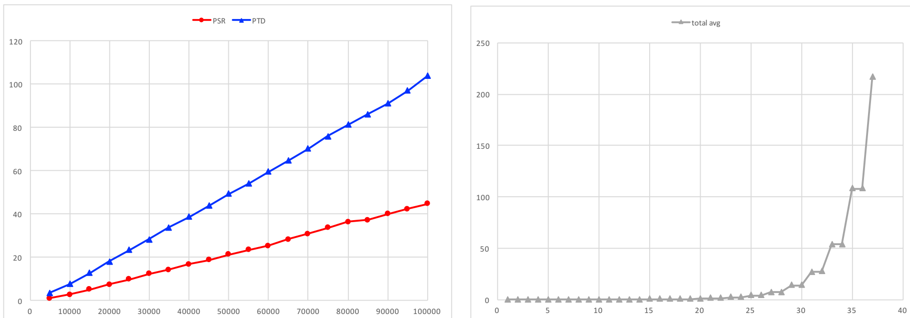
Fig. 5b shows the parsing time for the CYK parser as two graphs: the upper graph shows the parsing time for palindromes w_n where n is odd and the lower graph for n being even. This is so because the CYK parser must follow many more possible reverse derivations leading into dead ends for odd values of n than for even values of n. Note again that the scales in the diagrams shown in Fig. 4a and b differ and that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 820ms and 500ms to parse w_{399} and w_{400}, respectively, whereas the PTD parser needs 0.16ms, and the PSR parser just 44μs for w_{399} and w_{400}.

4 Trees

We also conducted experiments with trees built by an HR grammar with the following rules [1]:

$$\begin{aligned} S^{\varepsilon} &\rightarrow T^{x} \\ T^{x} &\rightarrow \varepsilon \mid edge^{xu}\, T^{x}\, T^{u} \end{aligned}$$

For the experiment, we considered binary trees T_n with n nodes. Each tree T_n has all of its levels but the last one completely filled; the last level is filled up from left to right in order to obtain a binary tree with n nodes.



(a) (b)

Figure 6: Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for binary trees T_n. Note that the scales in (a) and (b) differ.

Fig. 6a shows the runtime of the PSR and PTD parsers when processing T_n with varying values of n. Runtime has been measured in milliseconds on the y-axis while n is shown on the x-axis. Note the apparent linear behavior of the PSR parser and the, slightly slower, PTD parser. Fig. 7b shows the corresponding diagram for the CYK parser. Note that the runtime of the CYK parser is not linear in the size of the triangle graph. The “steps” in the graph are a result of the ambiguity of the grammar and the varying numbers of different derivation trees of T_n when n varies.

Note also that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 220ms to parse tree T_{37} whereas the PTD parser needs 26μs, and the PSR parser just 9.5μs.

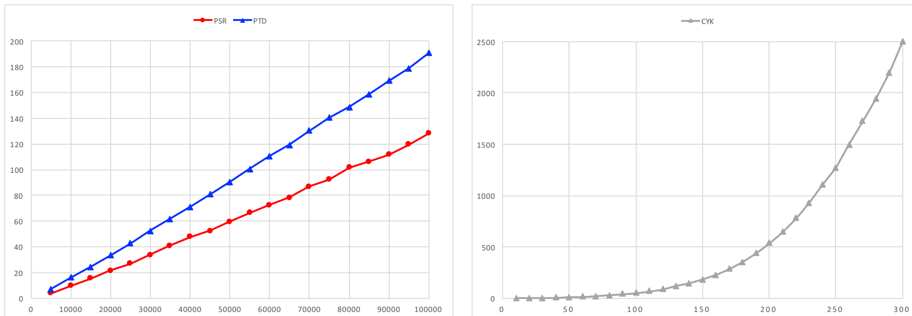
5 $a^n b^n c^n$ Language

We now consider the string language $\{a^n b^n c^n \mid n = 1, 2, 3, \ldots\}$, which is not context-free. However, when modelled by string graphs, it is the graph language of an HR grammar with the following rules [3]:

$$\begin{array}{rcl} S^{\varepsilon} & \to & a^{xu}\, b^{uy}\, c^{yz} \mid a^{xu}\, b^{vy}\, c^{yw}\, A^{uvwz} \\ A^{xyqz} & \to & a^{xu}\, b^{vy}\, c^{qw}\, A^{uvwz} \mid a^{xu}\, b^{uy}\, c^{qz} \end{array}$$

Grappa requires HR grammars to have a unique start rule to be PSR. We therefore used an equivalent HR grammar with the following rules:

$$\begin{array}{rcl} S^{\varepsilon} & \to & Z^{xyz} \\ Z^{xyz} & \to & a^{xu}\, b^{uy}\, c^{yz} \mid a^{xu}\, b^{vy}\, c^{yw}\, A^{uvwz} \\ A^{xyqz} & \to & a^{xu}\, b^{vy}\, c^{qw}\, A^{uvwz} \mid a^{xu}\, b^{uy}\, c^{qz} \end{array}$$



(a) (b)

Figure 7: Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for $a^n b^n c^n$ string graphs. Note that the scales in (a) and (b) differ.

$$
\begin{array}{rcl}
S^{\varepsilon} & \to & \mathit{Tree}^{xy} \\
\mathit{Tree}^{xy} & \to & \mathit{pair}^{xy} \mid \\
 & & \mathit{Child}^{xyu}\ \mathit{Tree}^{xy} \\
\mathit{Child}^{xyu} & \to & \mathit{edge}^{xyuv}\ \mathit{Tree}^{uv}\ \mathit{Next}^{xyu} \\
\mathit{Next}^{xyu} & \to & \mathit{Child}^{xyu} \mid \\
 & & \varepsilon
\end{array}
$$

Figure 8: Blowball graph grammar.

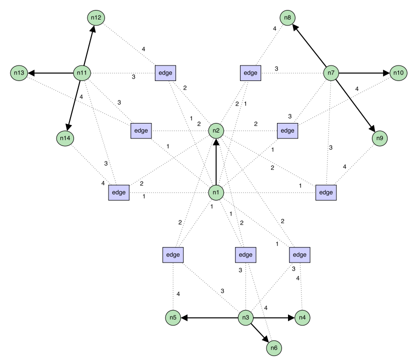
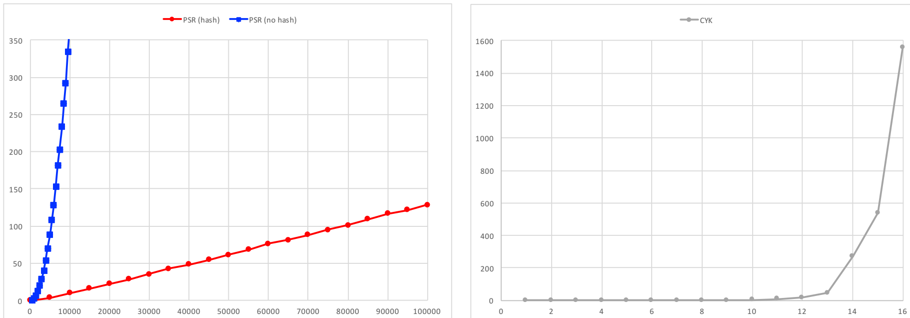


Figure 9: Blowball graph B_{10}.

Fig. 7a shows the runtime of the PSR and PTD parsers when processing $a^n b^n c^n$ string graphs with varying values of n. Runtime has been measured in milliseconds on the y-axis while n is shown on the x-axis. Note the apparent linear behavior of the PSR parser and the, slightly slower, PTD parser. Fig. 7b shows the corresponding diagram for the CYK parser. Note that the runtime of the CYK parser is not linear in the size of the triangle graph. Note also that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 2.5s to parse the string graph for $a^{300}b^{300}c^{300}$ whereas the PTD parser needs 0.27ms, and the PSR parser just 0.11ms.



(a) (b)

Figure 10: Runtime (in milliseconds) of the PSR parser (a) with hash tables (faster) and without hash tables (slower) and the CYK parser (b) for blowball graphs B_n. Note that the scales in (a) and (b) differ.

6 Blowballs

The PSR parsers described above make use of determining nodes and, therefore, do not require hash tables to obtain linear parsing time. In order to demonstrate the speed-up produced by hash tables, we constructed an HR grammar (see Fig. 8), called *blowball grammar* because of the shapes of its graphs. Its PSR parser must perform some edge look-ups without determining nodes. *Grappa* has been used to generate two versions of a PSR parser: Version *PSR (hash)* uses hash tables to speed up these edge look-ups, whereas version *PSR (no hash)* iterates over lists of candidates instead. Moreover, a PTD and a CYK parser have been generated. For the experiments, we considered blowball graphs B_n, $n \geq 1$, like B_{10} shown in Fig. 9: B_n consists of n *pair* edges (represented by arrows in Fig. 9), one in the center and the rest forming stars where the number of edges in each star is as close to the number of stars as possible. Runtime of the different parsers has been measured for these graphs B_n with varying values n. Fig. 10a shows the results of the two PSR parsers. The *PSR (no hash)* parser has quadratic parsing time and is much slower than the *PSR (hash)* parser with linear parsing time. For instance, *PSR (no hash)* needs 360ms to parse B_{10000}, whereas *PSR (hash)* needs just 10ms. Parsing time of the PTD parser is similar to the *PSR (no hash)* parser and is not shown here. Fig. 10b shows the results of the CYK parser, which is again by several orders of magnitude slower than the other parsers. For instance, the CYK parser needs 1.6s to parse B_{16} whereas the PTD parser needs just 9μs, and the PSR parsers (both versions) just 5μs.

References

[1] F. Drewes, B. Hoffmann, and M. Minas. Predictive top-down parsing for hyperedge replacement grammars. In F. Parisi-Presicce and B. Westfechtel,

editors, *Graph Transformation - 8th International Conference, ICGT 2015. Proceedings*, volume 9151 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015.

[2] F. Drewes, B. Hoffmann, and M. Minas. Predictive shift-reduce parsing for hyperedge replacement grammars. In J. de Lara and D. Plump, editors, *Graph Transformation - 10th International Conference, ICGT 2017. Proceedings*, Lecture Notes in Computer Science. Springer, 2017. To appear.

[3] A. Habel. *Hyperedge Replacement: Grammars and Languages*. Number 643 in Lecture Notes in Computer Science. Springer, 1992.

[4] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.