Performance Analysis of \textit{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 \textit{Grappa}\footnote{Grappa homepage: \url{www.unibw.de/inf2/grappa}}. In particular, it compares the performance of PDT\footnote{DiaGen homepage: \url{www.unibw.de/inf2/DiaGen}} and PSR\footnote{DiaGen homepage: \url{www.unibw.de/inf2/DiaGen}} parsers with the more general, but — as it turns out — slower Cocke-Younger-Kasami-style parsers\footnote{DiaGen homepage: \url{www.unibw.de/inf2/DiaGen}} generated by \textit{DiaGen}\footnote{DiaGen homepage: \url{www.unibw.de/inf2/DiaGen}}. All experiments have been conducted on a MacBook Pro 2013, 2.7 GHz Intel Core i7, Java 1.8.0.

Contents

1 Nested Triangles 2

2 Nassi-Shneiderman Diagrams 3

3 Palindromes 4

4 Trees 5

5 \(a^n b^n c^n\) Language 6

6 Blowballs 8
1 Nested Triangles

Consider nonterminals $S$ and ▲ and the terminal △. We use $ℓ^{x_1...x_k}$ as a shorthand for literals $ℓ(x_1,...,x_k)$. (Here $ε$ denotes the empty variable sequence.) Then the rules

\[
S^ε \rightarrow ▲^{xyz} \quad (1)
\]

\[
▲^{xyz} \rightarrow △^{xuv} △^{uyw} △^{vwz} ▲^{uvw} \quad (2)
\]

\[
▲^{xyz} \rightarrow △^{xyz} \quad (3)
\]

generate a nested triangle:

\[
S^ε \Rightarrow ▲^{123} \Rightarrow △^{145} △^{26} △^{563} △^{465} \Rightarrow △^{145} △^{426} △^{563} △^{478} △^{769} △^{895} △^{798}
\]

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

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. 2 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. 2 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.

Thanks to Berthold Hoffmann for this description of nested triangles.
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 $cond_{abcd}^{stmt_{efg}stmt_{dgh}}$. The language of all Nassi-Shneiderman graphs is defined by an HR grammar with the following rules:

\[
\begin{align*}
S^c & \rightarrow NSD^{xyuv} \\
NSD^{xyuv} & \rightarrow NSD^{xys} Stmt^{suv} | Stmt^{xyuv} \\
Stmt^{xyuv} & \rightarrow stmt^{xyuv} | cond^{xyrs} NSD^{rsmn} NSD^{mnuv} | while^{xyrsut} NSD^{rsatu}
\end{align*}
\]

We use the shorthand notation $L \rightarrow R_1 \mid R_2$ to represent rules $L \rightarrow R_1$ and $L \rightarrow R_2$ 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 $D_i$, $i = 1, 2, 3, \ldots$
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. 4 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:

\[
\begin{align*}
S^x & \rightarrow P^{xy} \\
P^{xy} & \rightarrow a^xu \ a^yv \ P^{uv} \ | \ b^xu \ b^yv \ P^{uv} \ | \ a^xu \ a^yv \ | \ b^xu \ b^yv \ | \ a^y \ | \ b^y
\end{align*}
\]

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., $w_1 = a, w_2 = aa, w_3 =$
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 $44\mu s$ for $w_{399}$ and $w_{400}$.

4 Trees

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

$$
S^c \rightarrow T^x
$$

$$
T^x \rightarrow \varepsilon \mid edge^{xu} T^x T^u
$$

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.
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]:

$$
\begin{align*}
S^e & \rightarrow a^{xu} b^{uy} c^{yz} \mid a^{xu} b^{uy} c^{yw} A^{uvwz} \\
A^{xyz} & \rightarrow a^{xu} b^{vy} c^{qw} A^{uvwz} \mid a^{xu} b^{uy} c^{qz}
\end{align*}
$$

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

$$
\begin{align*}
S^e & \rightarrow Z^{xyz} \\
Z^{xyz} & \rightarrow a^{xu} b^{vy} c^{yz} \mid a^{xu} b^{vy} c^{yw} A^{uvwz} \\
A^{xyz} & \rightarrow a^{xu} b^{vy} c^{qw} A^{uvwz} \mid a^{xu} b^{uy} c^{qz}
\end{align*}
$$
Fig. 7 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.

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.

Figure 8: Blowball graph grammar.

Figure 9: Blowball graph $B_{10}$. 
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 9$\mu$s, and the PSR parsers (both versions) just 5$\mu$s.

References

