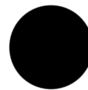


Workflow optimisation for graph illustrations

Course 213, Scripting languages and domain-specific languages

Department of Computer Science, University of Copenhagen

Henrik Stuart

3rd April 2006

Abstract

In recent years Adobe has pushed PDF heavily as the ubiquitous format for document publishing, and with great success. Since modern publishing in Computer Science is largely carried out by using LaTeX with a range of third party tools, it is fortunate that LaTeX has kept up with the industry and introduced the pdflatex compiler, which directly generates PDF documents from a LaTeX file. In many branches of Computer Science there is a prevalent need to make graph illustrations as part of articles and books, and a logical choice for these could be GraphViz, which generates graph illustrations based on simple and concise graph specifications in its own domain specific language, dot.

In this paper we look at two existing approaches for using GraphViz with pdflatex, and describe their shortcomings. We then present our own compiler that compiles dot to Ti*k*Z / PGF, which can be parsed by LaTeX using the PGF package by Till Tantau. Further we describe the results of using our compiler and how it may streamline the workflow of using GraphViz with pdflatex while minimising typographical inconsistencies. We also discuss in what areas the compiler may still be improved, and finally we briefly discuss the merits and flaws of integrating LaTeX with third party tools in this manner.

CONTENTS

List of Figures 4

1 Motivation 5

2 Existing methods 6
- 2.1 dottex 6
- 2.2 ladot 8

3 A new approach 12
- 3.1 The method 12
- 3.2 The implementation 12
- 3.3 Results 15

4 Conclusion 20

References 21

LIST OF FIGURES

Figure 1 Code and output from using dottex 7
Figure 2 An enlarged view of dottex output 7
Figure 3 The different parts of the ladot workflow 9
Figure 4 Enlarged version of the ladot output document 10
Figure 5 Workflow for gviztex 13
Figure 6 Processing of multiple path instructions 14
Figure 7 Compilation stages 15
Figure 8 The different parts of the gviztex workflow 16
Figure 9 Forstørret udsnit af gviztex dokumentet 17
Figure 10 The abstract GraphViz example graph 17
Figure 11 Font size/spacing issues 18
Figure 12 Problems with LaTeX commands 19

1 MOTIVATION

One of the most convincing ways to explain complex problems is to make illustrations that people can relate to. When we are presenting written material to our audience, it therefore becomes important that it is easy to make these illustrations. In this paper we look at graph illustrations, which is a common type of illustration in Computer Science, in conjunction with the LaTeX typesetting system. To generate our graph illustrations easily, we use GraphViz[1] that contains a domain specific language, dot, which allows us to concisely and easily describe any kind of graph. GraphViz also contains visualisation tools for processing dot-files, and it has support for a lot of different output formats. When we are typesetting documents in LaTeX we typically wish to use the PostScript output format.

However, when GraphViz generates a PostScript file with a graph visualisation, it uses the font that is specified in the source file or TimesRoman if none is specified. This font typically does not coincide with the choice of font in the LaTeX document we are writing, and furthermore, if we are using a logical font written in METAFONT then GraphViz will not know how to use it. This means inconsistencies arise between a document and its illustrations, which in turn ruins the overall aesthetics of the document's layout. This leads to our two main criteria for evaluating methods for including graph illustrations in a document:

- Ease of including illustrations in the document.
- Maintaining font consistency.

In section 2 we look at existing methods for including illustrations from GraphViz in a LaTeX document and describe their shortcomings. Then in section 3 we explain our method and implementation for making graph illustrations available and finally we review the results from using our approach and what areas that may be improved in section 3.3 and 4.

[1] GraphViz is available at www.graphviz.org.

2 EXISTING METHODS

We start by evaluating two of the existing approaches for including graph illustrations in a document according to the following criteria:

- Ease of including illustrations in the document.
- Ease of maintaining font consistency between the document and its illustrations.
- Ease of compiling the document and including illustrations when using pdfLaTeX.

The two first criteria coincide with those described in our motivation; the last is a result of many publishers preferring documents in PDF format, as Adobe has worked successfully on standardising this format in the publishing industry for the past many years. Plus, PDF is a more natural format to make available online, as the software for reading it is more ubiquitous than readers for PostScript, at least on the Microsoft Windows platform.

2.1 DOTTEX

Dottex is a LaTeX package, written by Lars Kotthoff, for specifying GraphViz specifications inside a LaTeX document using the two environments: dotpic and neatopic.[2] When the document is compiled with pdfLaTeX and shell-escape[3] is enabled, it will generate a temporary dot-file for each environment used, invoke dot or neato on them respectively and finally invoke epstopdf on the generated Encapsulated PostScript file. A typical file using dottex and its output is shown in figure 1.

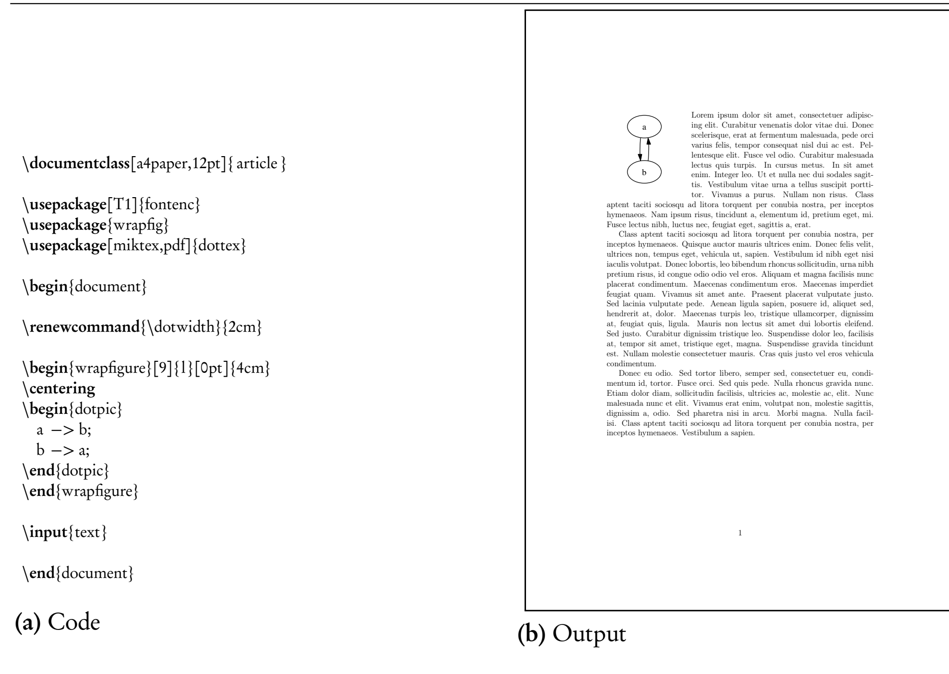
However, dottex comes with some limitations in that the dotpic environment is always a directed graph and the neatopic is always an undirected graph. This means that if we enable shell-escape we do not get to choose what layout algorithm is used to generate the illustration, as dottex chooses dot or neato for us.

With this in mind, let us see how it fares against our judgement criteria.

[2]Dottex is available at www.ctan.org/tex-archive/help/Catalogue/entries/dottex.html

[3]The shell-escape argument allows the LaTeX compiler to invoke commands on the command-line in the context of the user running the compiler.

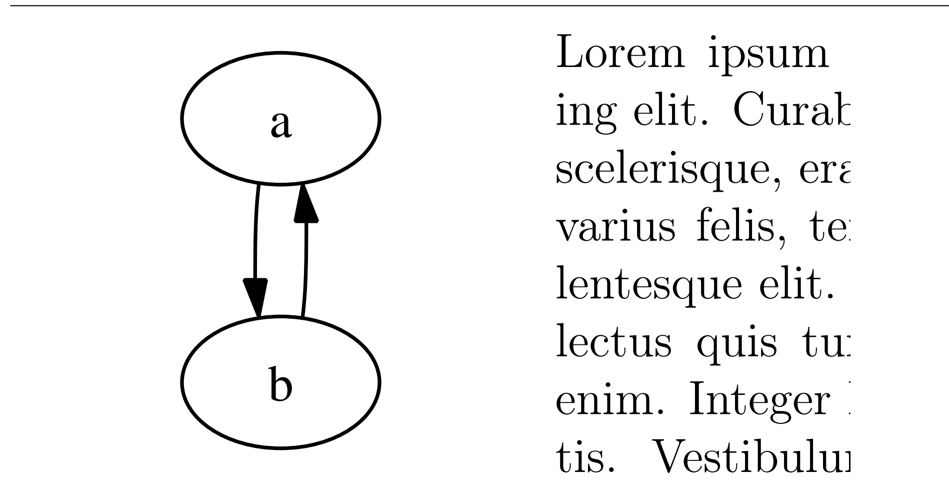
Figure 1. Code and output from using dottex



(a) Code

(b) Output

Figure 2. An enlarged view of dottex output



Inclusion:	There are many points against dottex, but it is extraordinarily easy to get graphs into the document, as they are specified directly in them.
Font consistency:	Dottex does not solve the problem of font consistency. We have to repeatedly specify the font for each dotpic or neatopic environment, and we are not able to use LaTeX-specific fonts, that is fonts that are not available at the system level. We can see the basic differences in font shapes between a default dotpic environment and a standard LaTeX document using Computer Modern in figure 2. Not exactly the best professional impression.
pdfLaTeX support:	The pdfLaTeX support is automated using epstopdf. So this is, all-in-all, fairly easy.

To conclude then dottex automates the inclusion of graph illustrations very well, but we are overly limited in choosing what visualisation algorithms are used. The worst part is, however, that dottex has no support for maintaining font consistency between LaTeX and GraphViz, so it fails at our second criteria.

2.2 LADOT

Ladot is a Perl script, written by Brighten Godfrey, that automates typesetting LaTeX maths expressions inside GraphViz specifications.[4] Ladot is constrained to only using the dot visualisation tool, and it only processes data between $'s in the graph specification. Since ladot cannot always properly estimate the width of the LaTeX text, we can specify a character string length after the last $ in the pair, as we can see in figure 3b. When we run ladot it will automatically generate a .dot-file from our source file, and from this generate a .ps and a .tex file with psfrag[5] information in it. The .tex file is shown in figure 3c. To use the generated files we first input the .tex-file and after that include the .ps file, as shown in figure 3a.

The steps to use ladot and get the final output as shown in figure 3d are as follows:

1. Run ladot on the .ladot-file.
2. Run latex on the document that includes the generated .tex and .ps files.
3. Run dvips to get a PostScript version of the document.

[4]Ladot is available at brighten.bigw.org/projects/ladot/

[5]Psfrag is a LaTeX package for replacing text in Encapsulated PostScript. It is available at www.ctan.org/tex-archive/help/Catalogue/entries/psfrag.html

Figure 3. The different parts of the ladot workflow

```
\documentclass[a4paper,11pt]{article}

\usepackage{psfrag,graphicx,wrapfig}
\usepackage[T1]{fontenc}

\begin{document}

\begin{wrapfigure}[10]{l}[0pt]{3cm}
  \input{test}
  \includegraphics{test.ps}
\end{wrapfigure}

\input{../dottex/text}

\end{document}
```

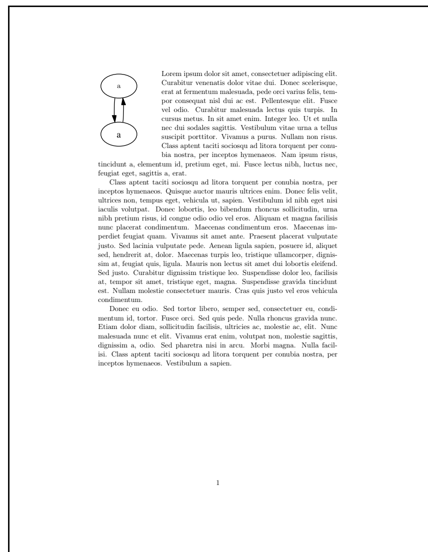
(a) Source Document

```
digraph {
  a [ label = "$\textrm{a}$(1)" ];
  c [ label = "a" ];
  a -> c;
  c -> a;
}
```

(b) Graph specification

```
\psfrag{G}[cc][cc]{$\textrm{a}$}
```

(c) Generated Code for the Graph



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur venenatis dolor vitae dui. Donec scelerisque, erat at fermentum malesuada, pede orci varius felis, tempor consequat nisl dui ac est. Pellentesque elit. Fusce vel odio. Curabitur malesuada lectus quis turpis. In cursus metus. In sit amet enim. Integer leo. Ut et nulla nec dui sodales sagittis. Vestibulum vitae urna a tellus suscipit porttitor. Vivamus a purus. Nullam non risus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Nam ipsum risus, tincidunt a, elementum id, pretium eget, mi. Fusce lectus nibh, luctus nec, feugiat eget, sagittis a, erat.

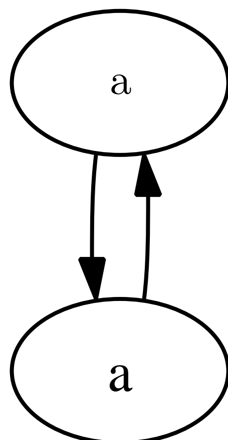
Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Quisque auctor mauris ultrices enim. Donec felis velit, ultrices non, tempus eget, vehicula ut, sapien. Vestibulum id nibh eget nisi iaculis volutpat. Donec lobortis, leo bibendum rhoncus sollicitudin, urna nibh pretium risus, id congue odio odio vel eros. Aliquam et magna facilisis nunc placerat condimentum. Maecenas condimentum eros. Maecenas imperdiet feugiat quam. Vivamus sit amet ante. Praesent placerat vulputate justo. Sed lacinia vulputate pede. Aenean ligula sapien, posuere id, aliquet sed, hendrerit at, dolor. Maecenas turpis leo, tristique ullamcorper, dignissim at, feugiat quis, ligula. Mauris non lectus sit amet dui lobortis eleifend. Sed justo. Curabitur dignissim tristique leo. Suspendisse dolor leo, facilisis at, tempor sit amet, tristique eget, magna. Suspendisse gravida tincidunt est. Nullam molestie consectetuer mauris. Cras quis justo vel eros vehicula condimentum.

Donec eu odio. Sed tortor libero, semper sed, consectetuer eu, condimentum id, tortor. Fusce orci. Sed quis pede. Nulla rhoncus gravida nunc. Etiam dolor diam, sollicitudin facilisis, ultricies ac, molestie ac, elit. Nunc malesuada nunc et elit. Vivamus erat enim, volutpat non, molestie sagittis, dignissim a, odio. Sed pharetra nisi in arcu. Morbi magna. Nulla facilisi. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Vestibulum a sapien.

1

(d) Output

Figure 4. Enlarged version of the ladot output document



Unfortunately psfrag only supports raw PostScript drivers like dvips, which means that we cannot use it with pdfLaTeX. So if we wish to use ladot with pdfLaTeX, we will need to create a separate LaTeX document with all our font settings, and code to include the illustration, then we compile it using ladot, latex, dvips and ps2pdf, and then include the resulting PDF in our main document. This makes it, of course, even more elaborate to use than GraphViz alone, but we do get the advantage of having some of our labels typeset using LaTeX and its fonts. It does, however, require some care to ensure that all illustrations have consistent fonts. This is clearly illustrated in figure 4, where we can see the distinction between the a in the top node, which is typeset using LaTeX, and the a in the lower node, which is typeset using the default GraphViz font, TimesRoman. We furthermore see that the upper a is identical to any of the a's on the right.[6] With this in mind let us see how it fares at our criteria:

Inclusion:	If we use a PostScript-based workflow, inclusion is fairly easy. We just have to remember to input the generated psfrag instructions first and then include the PostScript file.
Font consistency:	We are able to use LaTeX fonts if we construct all our labels as $ pairs, but if we forget this we will still use the default GraphViz font, unless we specify otherwise in the GraphViz file. For the default GraphViz font we will still not be able to use the LaTeX fonts. Lastly, it may prove a problem that there is a requirement for $ pairs as a lot of visualisation software

[6] If the a in the graph seems smaller it is due to an optic illusion, it is in fact the same size.

does not generate GraphViz output using these.

pdfLaTeX support: Ladot has no immediate support for pdfLaTeX, so if we wish to use pdfLaTeX we will have to do an extraordinary amount of work to bring our illustration into our PDF document. This is probably one of the biggest problems of ladot, but it is not caused by ladot, but rather by its dependence on the psfrag package, which only works on Encapsulated PostScript.

So all-in-all we have moved closer to font consistency, but we have drastically worsened our workflow having to do extra manual steps for every illustration we need to include in our document.

3 A NEW APPROACH

The two existing approaches we saw in the last section have neither excelled nor been abhorrently bad according to our criteria, but they still leave room for improvement. In this section we describe a method for translating a GraphViz specification to code that can be parsed by a TEX / LATEX package for complete, native inclusion in a document. First we describe the general method for the conversion, then how it is implemented, and lastly we describe the results of using our method.

3.1 THE METHOD

Whereas the existing models we have seen merely include the Encapsulated PostScript that GraphViz generates we have taken a different approach and written a compiler that translates post-processed dot-files, to code that can be successfully executed by the Ti*k*Z / PGF LATEX package.[7] This gives us the benefit that our entire figure is typeset by LATEX and there are no further inclusion issues to take into account, and considering PGF is available for both PostScript and PDF we may use our generated output in both a PostScript and PDF workflow.

Another benefit that our compilation confers is that we are able to utilise the output of all the GraphViz visualisation tools, whereas ladot only supported dot, and dottex only supported dot and neato for directed and undirected graphs respectively. However, unlike ladot we do not support specifying the 'real' label width for mathematical expressions, so typesetting LATEX expressions is not natively supported. We have made this decision by design, as our primary motive for making this compiler is to eliminate typographical inconsistencies and not so much typesetting LATEX expressions in GraphViz, but we will elaborate on this choice when we discuss our results.

The workflow for using our method is illustrated in figure 5.

3.2 THE IMPLEMENTATION

We have implemented our compiler, gviztex, in C# 2 using Malcolm Crowe's lexer and parser generators.[8] We parse almost all of the the GraphViz specification, but only generate code for a subset of the full specification and the well-defined attributes. Table 1 specifies what attributes we generate code for.

[7] Ti*k*Z / PGF is available at www.ctan.org/tex-archive/help/Catalogue/entries/pgf.html
[8] The lexer and parser generator is available at cis.paisley.ac.uk/crow-ci0/.

Figure 5. Workflow for gviztex

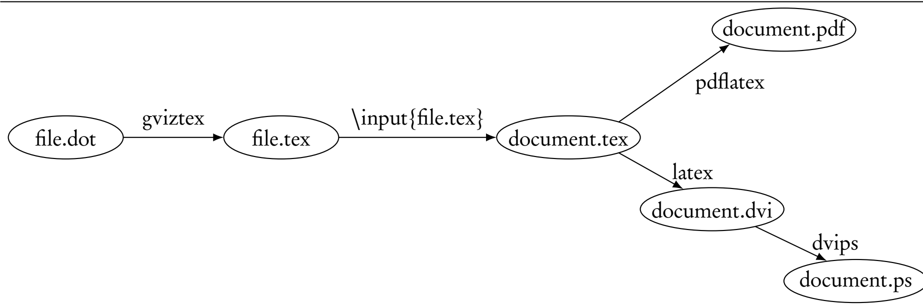


Table 1 Attribute support in gviztex

Attribute	*Entities*
pos	node, edge
color	node, edge
fillcolor	node
label	node
style[I]	node
shape[II]	node
width	node
height	node

[I] The only style parameter that we check is ‘filled’.
[II] Only ellipse and rectangle are currently supported.

Figure 6. Processing of multiple path instructions

```
graph {
  a -- b -- c -- d;
}
```

(a) Original dot-file

```
graph {
  node [label="\N"];
  graph [bb="0,0,54,252"];
  a [pos="27,234", width="0.75", height="0.50"];
  b [pos="27,162", width="0.75", height="0.50"];
  c [pos="27,90", width="0.75", height="0.50"];
  d [pos="27,18", width="0.75", height="0.50"];
  a -- b [pos="27,216 27,205 27,191 27,180"];
  b -- c [pos="27,144 27,133 27,119 27,108"];
  c -- d [pos="27,72 27,61 27,47 27,36"];
}
```

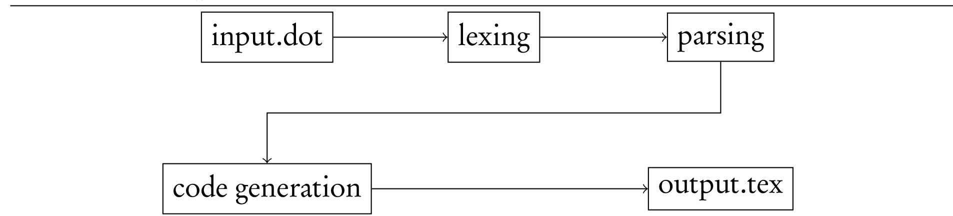
(b) Post-processed dot-file

GraphViz contains the functionality to process dot-files and generate corresponding dot-files where nodes and edges are placed in a coordinate system and graphs and sub graphs are assigned bounding boxes. We utilise this processing capability and parse what we will call post-processed dot-files, which is dot-files where a GraphViz visualisation tool has processed it and generated a dot-file. Our code generator is implemented as a reflective visitor pattern, which is based on the more commonly known visitor pattern described in Erich Gamma et al. (1995). This allows us to easily tweak the code for disparate parts of the abstract syntax tree, and it is easy to add new output formats if we wish, for instance to create a pre-processor for manipulating the dot-file before we process it using GraphViz.

Another benefit of the post-processed dot-file is that GraphViz reduces some of the complexities in the original dot-file. This can, for instance, be seen in dot-files that contain multiple path instructions in one statement, as illustrated in figure 6a. After processing this file we get separate descriptions of the nodes and each of the edges, as seen in figure 6b. This means that we can ignore some extra complications in processing edges in our code generation visitor. If we were to write a pre-processor instead, the visitor implementing it could not count on these simplifications.

When we generate LaTeX code we must remember that LaTeX's special characters differ from most other systems. This means we have to carefully translate the contents of labels to something LaTeX will accept. A further complication is that Ti*k*Z / PGF doesn't accept multi-line content, so in order to emulate that, we will need to wrap the label text in a LaTeX **\parbox**. We have only made a very rudimentary translation, so it is possible to write labels that will generate errors when we try to compile the LaTeX document. Our current translation only wraps labels in a **\parbox** when they contain a newline, \n, or a 'displaymath' sequence, \[... \]. In practice there are many other commands that require paragraph mode in LaTeX, but making an exhaustive list of these is virtually impossible.

Figure 7. Compilation stages



For completeness sake we have illustrated the different parts of the compilation process in figure 7. Otherwise there are no interesting aspects to the implementation; it is merely a simple conversion from dot to Ti*k*Z / PGF. It makes little sense to enumerate what the primitives in dot maps to in the Ti*k*Z / PGF 'language'. For full insight into the mapping, see the source code.[9]

To invoke the compiler, we execute it with 'gviztex dot file.dot', and it generates a file.tex by first running dot on file.dot, and then parsing its output and generating Ti*k*Z / PGF code to file.tex. This means it may completely replace dot, neato, etc. in our document's makefile, if we use one such, but apart from that there are no further issues in using it to generate LaTeX code. From LaTeX we only need to use the Ti*k*Z package and input the generated LaTeX file at the appropriate place.

3.3 RESULTS

Using an equivalent example to the earlier ones, we see the source code for the graph in figure 8a, and the generated code, slightly edited to fit on the page, in figure 8b. Finally we see the LaTeX document that includes our generated code in figure 8c, and the generated PDF can be seen in figure 8d.

In order to see whether we have gained the font consistency we sought, we see an enlarged part of the document in figure 9, where it is clearly evident that both the a and b are identical to the a's and b's in the document text. We may also generate large graphs and input them natively after gviztex has been run, as we see on the example graph from GraphViz in figure 10. Due to the lack of a colour printer we will not illustrate our ability to compile graphs with colours in this paper.

This means that we have more or less successfully managed to fulfil our three criteria: our illustration uses the same font as our main document, it is easy to include in our document, and we can use it together with pdfLaTeX easily. However, we have only 'more or less' succeeded, because there are still issues with synchronising between GraphViz and LaTeX, and some of these issues still con-

[9]The source code for gviztex is available at www.hstuart.dk/files/school/dsl/gviztex.zip

Figure 8. The different parts of the gviztex workflow

```
digraph {
  a -> b;
  b -> a;
}
```

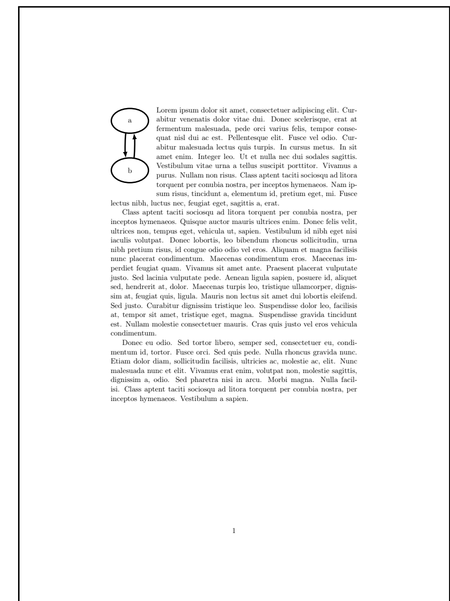
(a) Graph specification

```
\begin{tikzpicture}[line width=2bp]
\pgfsetarrowsend{latex}
% a
\begin{scope}
\pgfpathellipse{\pgfpoint{27bp}{90bp}}{\pgfpoint{27bp}{0bp}}
  {\pgfpoint{0bp}{18bp}}
\pgfusepath{stroke}\pgftransformshift{\pgfpoint{27bp}{90bp}}
\pgfnode{rectangle}{center}
  {\color{black} \parbox{27bp}{\centering a}}
  {}{\pgfusepath{}}
\end{scope}

% b
\begin{scope}
\pgfpathellipse{\pgfpoint{27bp}{18bp}}{\pgfpoint{27bp}{0bp}}
  {\pgfpoint{0bp}{18bp}}
\pgfusepath{stroke}\pgftransformshift{\pgfpoint{27bp}{18bp}}
\pgfnode{rectangle}{center}
  {\color{black} \parbox{27bp}{\centering b}}
  {}{\pgfusepath{}}
\end{scope}

% a -> b
\pgfpathmoveto{\pgfpoint{21bp}{72bp}}
\pgfpathcurveto{\pgfpoint{20bp}{64bp}}
  {\pgfpoint{20bp}{55bp}}{\pgfpoint{20bp}{46bp}}
\pgfpathlineto{\pgfpoint{21bp}{36bp}}
\pgfusepath{stroke}
% b -> a
\pgfpathmoveto{\pgfpoint{33bp}{36bp}}
\pgfpathcurveto{\pgfpoint{34bp}{44bp}}
  {\pgfpoint{34bp}{53bp}}{\pgfpoint{34bp}{62bp}}
\pgfpathlineto{\pgfpoint{33bp}{72bp}}
\pgfusepath{stroke}
\end{tikzpicture}
```

(b) Generated graph code

```
\documentclass[a4paper,11pt]{article}

\usepackage{tikz,wrapfig}
\usepackage[T1]{fontenc}

\begin{document}

\begin{wrapfigure}[10]{l}[0pt]{2cm}
  \input{test}
\end{wrapfigure}

\input{../dottex/text}

\end{document}
```

(c) Document



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur venenatis dolor vitae dui. Donec scelerisque, erat at fermentum malesuada, pede orci varius felis, tempor consequat nisl dui ac est. Pellentesque elit. Fusce vel odio. Curabitur malesuada lectus quis turpis. In cursus metus. In sit amet enim. Integer leo. Ut et nulla nec dui sodales sagittis. Vestibulum vitae urna a tellus suscipit porttitor. Vivamus a purus. Nullam non risus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Nam ipsum risus, tincidunt a, elementum id, pretium eget, mi. Fusce lectus nibh, luctus nec, feugiat eget, sagittis a, erat.

Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Quisque auctor mauris ultrices enim. Donec felis velit, ultrices non, tempus eget, vehicula ut, sapien. Vestibulum id nibh eget nisi iaculis volutpat. Donec lobortis, leo bibendum rhoncus sollicitudin, urna nibh pretium risus, id congue odio odio vel eros. Aliquam et magna facilisis nunc placerat condimentum. Maecenas condimentum eros. Maecenas imperdiet feugiat quam. Vivamus sit amet ante. Praesent placerat vulputate justo. Sed lacinia vulputate pede. Aenean ligula sapien, posuere id, aliquet sed, hendrerit at, dolor. Maecenas turpis leo, tristique ullamcorper, dignissim at, feugiat quis, ligula. Mauris non lectus sit amet dui lobortis eleifend. Sed justo. Curabitur dignissim tristique leo. Suspendisse dolor leo, facilisis at, tempor sit amet, tristique eget, magna. Suspendisse gravida tincidunt est. Nullam molestie consectetuer mauris. Cras quis justo vel eros vehicula condimentum.

Donec eu odio. Sed tortor libero, semper sed, consectetuer eu, condimentum id, tortor. Fusce orci. Sed quis pede. Nulla rhoncus gravida nunc. Etiam dolor diam, sollicitudin facilisis, ultricies ac, molestie ac, elit. Nunc malesuada nunc et elit. Vivamus erat enim, volutpat non, molestie sagittis, dignissim a, odio. Sed pharetra nisi in arcu. Morbi magna. Nulla facilisi. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Vestibulum a sapien.

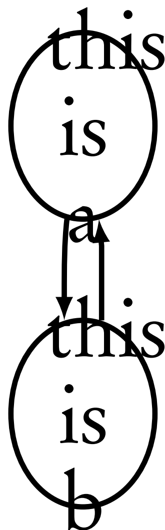
1

(d) Output

Figure 9. Forstørret udsnit af gviztex dokumentet

Figure 10. The abstract GraphViz example graph

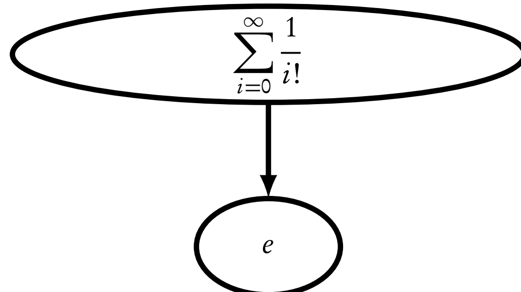
Figure 11. Font size/spacing issues



cern font consistency, but they are a lot harder to fix. The issue manifests itself when we start to change the LaTeX font size and spacing, because GraphViz still computes the sizes of the vertices on the basis that we are using 12 point TimesRoman—GraphViz's standard font and font size—and this, of course will pose a problem if we are using URW Garamond No. 8 in point size 30. In particular the problem manifests itself in that our label goes over the boundary of the vertex, as illustrated in figure 11. These font sizes are, of course, vastly overemphasised as to what one would normally use in an illustration, but they do illustrate the lack of consistency between GraphViz and LaTeX, when GraphViz computes the extent of its labels. The inconsistency has thus changed from being a problem with synchronising the fonts to being a genuine integration problem.

To handle this we would have to integrate LaTeX and GraphViz much more closely, possibly drawing on the concept that dottex uses, where we generate the dot-file from inside LaTeX, including a specification of the font size/spacing, but this may pose a problem for the tools we use that generate dot code. Furthermore, this may not entirely solve the problem as the font we decide to use may not exist as far as GraphViz is concerned, so the 'non-existent' font may have a far greater character width and overshoot the boundaries of the vertex even though we have changed the font size/spacing. The core problem is, of course, that we have no apparent way to instruct GraphViz of the exact extent of the label, we are only able to specify the exact width and height of a vertex, which in most cases isn't sufficient.

Figure 12. Problems with LaTeX commands



A further complication arises if we wish to typeset LaTeX expressions inside vertices, which is, in particular, interesting for typesetting maths. The problem, as we have seen with ladot, is that GraphViz doesn't understand the LaTeX commands and thus estimates the width of the vertex far too wide and the height too small, typically. This is illustrated in figure 12, where we try to typeset the expression `\[\sum_{i=0}^\infty \frac{1}{i!}\]` in the top node, and `$e$` in the lower node. As we can see in the top node then the width fits fairly well with the width and height of the expression, but, alas, not with the width and height of the evaluated expression.

Like with the font size problem, the core problem is caused by us having no way to instruct GraphViz on the extent of the label in its post-evaluated form. We have already seen a partial solution to this with how ladot processes labels, namely if we instruct ladot to typeset the label `$\sum_{i=0}^\infty i^2$(4)`, then the (4) instructs ladot that it should generate a random character sequence of four characters, because that will be the approximate width of the label, however that doesn't take care of the extra height 'Sigma' would be typeset with in a 'display-math' setting—that is where the 'Sigma' has the bounds typeset at the top and bottom rather than at the side. Ladot's approach is just an approximated solution, and while it may prove adequate for practical purposes it is infeasible for drawings that are auto-generated, so the only way we see that we can accomplish solving this problem, would be through a much tighter integration between LaTeX and GraphViz, most likely with bidirectional communication so GraphViz can query LaTeX for widths and heights, and LaTeX can query GraphViz when a graph illustration is to be included. There is, however, no apparent way to achieve this integration with the way GraphViz is constructed today. We will leave a solution of this problem to future work.

4 CONCLUSION

Typographical inconsistencies are luckily manageable to avoid, but they still occur occasionally in articles, journals and books throughout the self-publishing and professional publishing industry. These inconsistencies are rarely a result of ill will or laziness, but an expression that our tools do not adequately address the issue of integrating properly with each other. As we have seen several times during the course of this paper, there have been a number of approximations to making the workflow of integrating graphs made by GraphViz with LaTeX more manageable, but either they limit our freedoms of choosing visualisation tools excessively, like dottex, or they require us to insert label widths explicitly and take a lot of extra care when working with PDF, like ladot.

For these purposes we believe that our compiler, gviztex, adequately solves the problem of typesetting pure text in vertices directly inside our PostScript or PDF workflow in LaTeX. There is still room for improvement, including support for a lot more attributes, and support for integrating LaTeX commands and synchronising font size/spacing. However, these two last problems were not ones we set out to solve initially; they were uncovered along the way as interesting problems as they illustrate the dichotomy between LaTeX and a long range of third party tools we may wish to use with it.

In closing, we have not solved the entire problem of integrating GraphViz with LaTeX, but we have made a good solution for converting dot-files to a format that can be parsed and understood by LaTeX. From this it is fairly easy to make the last tweaks and alterations, if we are not entirely happy with the way GraphViz has placed our vertices or edges.

REFERENCES

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.