Received: 3 March 2017 | Revised: 2 December 2017 | Accepted: 21 January 2018

DOI: 10.1111/coin.12165

SPECIAL ISSUE ARTICLE

WILEY Computational Intelligence

A general LuaTEX framework for globally optimized pagination

Frank Mittelbach

LATEX3 Project, Mainz, Germany

Correspondence
Frank Mittelbach, LATEX3 Project, Mainz, Germany.
Email:
frank.mittelbach@latex-project.org

Abstract

Pagination problems deal with questions around transforming a source text stream into a formatted document by dividing it up into individual columns and pages, including adding auxiliary elements that have some relationship to the source stream data but may allow a certain amount of variation in placement (such as figures or footnotes).

Traditionally, the pagination problem has been approached by separating it into one of micro-typography (eg, breaking text into paragraphs, also known as h&j) and one of macro-typography (eg, taking a galley of already formatted paragraphs and breaking them into columns and pages) without much interaction between the two.

While early solutions for both problem areas used simple greedy algorithms, Knuth and Plass introduced in 1981 a global-fit algorithm for line breaking that optimizes the breaks across the whole paragraph. This algorithm was implemented in TEX'82 (see *Computers & Typesetting, Volume B: TeX: The Program* by Knuth in 1986) and has since kept its crown as the best available solution for this space. However, for macro-typography there has been no (successful) attempt to provide a globally optimized page layout: All systems to date (including TEX) use greedy algorithms for pagination. Various problems in this area have been researched, and the literature documents some prototype development. However, none of them have been made widely available to the research community or ever made it into a generally usable and publicly available system.

This paper is an extended version of the author's work in 2016 originally presented at the 16th ACM Symposium on Document Engineering in Vienna, Austria. It presents a framework for a global-fit algorithm for page breaking based on the ideas of Knuth/Plass. It is implemented in such a way that it is directly usable without additional executables with any modern TEX installation. It therefore can serve as a test bed for future experiments and extensions in this space. At the same time, a cleaned-up version of the current prototype has the potential to become a production tool for the huge number of TEX users worldwide.

This paper also discusses 2 already implemented extensions that increase the flexibility of the pagination process (a necessary prerequisite for successful global optimization): the ability to automatically consider existing flexibility in paragraph length (by considering paragraph variations with different numbers of lines) and the concept of running the columns on a double spread a line long or short. It concludes with a discussion of the overall approach, its inherent limitations and directions for future research.

KEYWORDS

automatic layout, global optimization, macro-typography, page breaking, pagination, typesetting

1 | INTRODUCTION

Pagination is the act of transforming a source document into a sequence of columns and pages, possibly including auxiliary elements such as floats (eg, figures and tables).

As textual material is typically read in sequential order, its arrangement into columns and pages needs to preserve the sequential property. There are applications where this is not the case or not fully the case, eg, in newspaper layout, where stories may be interrupted and "*continued on page X*," but in this paper (which is an extended version of the author's work in 2016 originally presented at the 16th ACM Symposium on Document Engineering in Vienna, Austria[1]), we limit ourselves to formatting tasks with a single textual output stream (see the work of Hailpern et al[2] for a discussion of problems related to interrupted texts).

An algorithm that undertakes the task of automatic pagination therefore has to transform the textual material into individual blocks that form the material for each column and arrange for distributing auxiliary material across all pages (thereby reducing available column heights) in a way that it best fulfills a number of (usually) conflicting goals.

This transformation is typically done as a 2-step process by first breaking the text into lines forming paragraphs and this way assembling a galley (known as hyphenation and justification or h&j for short) and then as a second step by splitting this galley into individual columns to form the pages.

However, separating line breaking and page breaking means that one loses possible benefits from having both steps influence each other. Thus, it is not surprising that this has been an area of research throughout the years, eg, other works.[3-7] The algorithm outlined in this paper implements some limited interaction to add flexibility to the page-breaking phase.

The remainder of this paper is structured as follows. We first discuss general questions related to pagination and give a short overview about attempts to automate that process and the possible limitation when using a global optimization strategy for pagination. Section 2 then describes our framework for implementing globally optimizing pagination algorithms using a TEX/LuaTEX environment. In Section 3, we have a general look at approaching the problem using dynamic programming and discuss various useful customizable constraints that can be used to influence this optimization problem. Section 4 then discusses details of the algorithms we used and gives some computational examples. This paper concludes with an evaluation of the algorithm quality and an outline of possible further research work.

Although attempts are made to introduce all necessary concepts, this paper assumes a certain level of familiarity with TEX; if necessary refer to the work of Knuth[8] for an introduction.

1.1 | Pagination rules

Rules for pagination and their relative weight in influencing the final result vary from application to application, as they are often (at least to some extent) of an aesthetic nature, but also because, depending on the given job, some primary goals may outweigh any other. It is therefore important that any algorithm for this space is configurable to support different rule sets and able to adjust the weight of each rule in contributing to the final solution.

The primary goal of nearly every document is to convey information to its audience, and thus, an undisputed “meta” goal for document formatting is to enhance the information flow or at least avoid hindering or preventing successful communication of information to the recipient. An example of a rule derived from this maxim is the already mentioned requirement of keeping the text flow in clearly understandable reading order.

Other examples are rules regarding float placement: To avoid requiring the reader to unnecessarily flip pages, floating objects should preferably be placed close to (and visible from) their main callout, and if that is not possible, they should be placed nearby on later pages (so that a reader has a clear idea where to search for them). For the same reason, they should be kept in the order of their main callouts, although that, for example, is a rule that is sometimes broken when placement rules are mainly guided by aesthetic consideration.

Other rules are more aesthetic in nature, although they too originate from the attempt to provide easy access to information, as violating them will disrupt the reading flow to some extent: have a heading always followed by a minimal number of lines of normal text, avoid widows and orphans (end or beginning line of a paragraph on its own at a column break), or do not break at a hyphenated line. An example from mathematical typesetting is to shun setting displayed equations at the top of a column, the reason being that the text before such a formula is usually an introduction to it; thus, to aid comprehension they should be kept together if possible.

Rules like the above have in common that they all reduce the number of allowed places where a column break could be taken, ie, they all generate unbreakable larger vertical blocks in the galley. Thus, finding suitable places to cut up the galley into columns of predefined sizes becomes harder, and greedy algorithms nearly always run into stumbling blocks (no pun intended), where the only path they can take is to move the offending block to the next column, thereby leaving a possibly large amount of white space on the previous one.

The second major "meta" goal, especially in publishing, is to make best use of the available space to keep the costs low. If we look only at formatting a single text stream (no floats), then it is easy to see that this goal stands in direct competition with any rule derived from the first meta goal. It is easy to prove that a greedy algorithm will always produce the shortest formatting* if the column sizes are fixed and all document elements are of fixed size and need to be laid out in sequential order (a proof is given in Lemma 1 on page 42). Thus, to satisfy both goals, one needs to allow for either

- variations in column heights,
- variations in the height of textual elements, or
- allow nonsequential ordering of elements.

In this paper, we look, in particular, at the first 2 options. The last bullet is usually not an option for text elements, except in the case of documents with short unrelated stories that can be reordered or texts that are allowed to be split and "continued." As these form their own class of documents with their own intrinsic formatting requirements, they are not addressed in this paper (see, for example, other works[9-12]). There is, however, also the possibility to introduce a certain amount of additional flexibility through clever placement of floats (such as figures or tables) as this will change the height of individual columns. We do not address the question of optimization through float placement as part of this paper but assume that floats are either absent or their placement predetermined or externally determined. The class of documents for which this can be assumed is rather large; thus, the findings in this paper are relevant even with this restriction in force. Mittelbach[13] discussed the effects of adding float streams to the optimization process and the resulting changes in complexity.

A variation in column height (typically by allowing the height to deviate by 1 line of text) is a standard trick in craft typography to work around difficult pagination situations. To hide such a change from the eyes of the reader, or at least lessen the impact, all columns of a page and, in 2-sided printing, a double spread (facing pages in the output document) needs the same treatment.† It is also best to only gradually change the column heights to avoid big differences between one double spread and the next.

The second option involves interaction between the micro- (line breaking of paragraphs, formatting of inline figures, etc) and macro-typography phase (pagination of the galley material), either by dynamically requesting micro-typography variants during pagination or by precompiling them for additional flexibility in the macro-typography phase. Examples are line breaking with suboptimal spacing (variant looseness setting in TEX's algorithm, eg, the works of Hassan and Hunter[7] and Knuth[8]) or font compression/expansion (hz-algorithm as implemented by Thánh[14]) within defined limits. Other examples are figures or tables that can be formatted to different sizes.

1.2 | Typical problems during pagination

Problems with pagination are commonplace if a greedy algorithm is used. As a typical example, Figure 1 on page 6 shows the first 6 double spreads from "Alice in Wonderland" as it would be

*The formatting is "shortest" in the sense that compared to any other pagination, it will have a lower or equal number of columns/pages, and if equal, the last column will contain a lesser or equal amount of material.
†In addition, the paper for printing should be thick enough so that the text block on the back is not shining through, as that would be a dead giveaway.

typeset by the (greedy) algorithm of LaTeX when orphans and widows are disallowed. Most of the resulting defects can easily be spotted even in the thumbnail presentation. The most glaring one is on page 7 within Figure 1, which ends up being largely empty.

In contrast, the algorithm presented in this paper is able to overcome all of these problems, producing the results shown in Figure 2 on page 7. It achieves this by manipulating the paragraphs shown in gray in both figures and by running some of the page spreads short or long.

The example may seem to be spectacular, but the reality is that these kinds of defects show up with nearly every document the moment it contains even only a small number of text blocks that cannot be broken across columns, eg, displayed formulas, headings that need to be followed by a few lines of text, disallowed widows and orphans, etc, all of which are common cases in books or journal articles.

1.3 | Global optimization

When we speak of "globally optimized pagination," we mean that out of all possible paginations for a given document, the "best" by some measure is selected. To determine this optimal pagination, we define a function that, when given a pagination as input, will return a single numerical value. By convention, a lower value indicates a better result; hence, common names for such functions in the literature are "cost function" or "objective function" as one can view this as returning the costs associated with its input. Finding the optimal solution therefore means finding the pagination that results in the lowest return value of this function.

Taking a step back from that rather abstract definition of a quality measure, what does it mean in reality and how can it be applied? Obviously, if we can measure a specific aspect of a pagination, we can attach a value to it, and this can be done in a such way that lower values correspond to better results for this particular aspect.

For example, if we are interested in a low number of pages, we could return *#pages* or $(\#pages)^2$. On the other hand, if we want to measure the quality of the white space distribution for a given pagination, we could analyze the quality of each column (obtaining a number greater than zero, if there is excess white space) and then sum these up over all columns, or sum up the squares of these values or use the maximum over all columns or …

All these examples are valid measures in the sense that they encode the quality of a certain aspect in a monotone function, but clearly, they are quite different and focus on different sub-aspects. For example, if we take the sum, then this means that a single very bad column among a lot of perfect ones is considered to be better than a few near-perfect columns, whereas summing the squares will give a better result if all values are closer to each other. Thus, even for a single quality aspect, it can turn out to be a difficult problem to define a cost function that is a reasonable approximation to the quality perception of a human looking at the paginations.

However, to make matters worse, we need to deal not with one but with several different quality aspects but still come up in real life with a single number that encompasses them all. This can then be used to determine the overall optimal solution, which is commonly done by adding up the values for each aspect after weighting each of them by a factor (a weighted sum). Very many other methods and adjustments are available for combining these values to give a single number. Each gives a new twist on the algorithm's understanding of "quality."

Summing up, an optimal solution is only optimal with respect to the quality measure that is encoded in the objective function used to determine it. If that function is defective, so will be the algorithm's result. Furthermore, correctly weighting different aspects against each other (even

Defects: Too much white space in the second column on page 2, the first column on page 6 and again in the first column of page 8 (the little bars above the columns give a visual indication of the column badness from TeX's perspective—longer means worse).
– A nearly empty column on page 7 because of the "mouse tail poem" is an unbreakable block.
– The first column on page 10 is one line too short because of a following 3-line paragraph that doesn't fit in its entirety (as widows/orphans are disallowed) and there is nothing in the column to stretch.

The gray- or green-colored paragraphs (pages 2, 5 and 6) are those that are modified by the globally optimizing algorithm to achieve the results shown in Figure 2 on the next page.

FIGURE 1 Alice in Wonderland typeset with a greedy algorithm [Color figure can be viewed at wileyonlinelibrary.com]

if only a small number of aspects are part of the equation) is a difficult art and requires some experimentation to achieve acceptable results.[‡]

[‡]During the development, the author was several times quite surprised by the changes in the solution chosen by the framework when making minor changes to some constraints. In a few cases, this revealed a hidden bug in the implementation, but usually it was due to the algorithm sacrificing the quality of one aspect in one part to get a better result for some other aspect elsewhere.

Corrective actions compared to the greedy algorithm:
– Shortened a paragraph in second column of page 2. Lengthened pages 4–7 by one line.
– On page 5 lengthened one paragraph in first column and shortened one in second column. Net effect is zero but it avoids an orphan at the end of the first column.
– Shortened two paragraphs on page 6 to allow the “mouse tail poem” to move back to page 7.
– Shortened pages 8–11 by one line to avoid some orphans and widows.

The gray- or green-colored paragraphs (pages 2, 5 and 6) are those that have been lengthened or shortened by the globally optimizing algorithm. Compare with the results from the greedy algorithm shown in Figure 1.

FIGURE 2 Alice in Wonderland typeset with an optimizing algorithm [Color figure can be viewed at wileyonlinelibrary.com]

There is thus a need for a flexible framework, one that allows the user of such an algorithm to specify their vision of quality in the form of constraints and relationships between different goals and enables them to approximate this vision as closely as possible in the objective function used during optimization. Section 1.6 gives a preliminary overview on the constraints and flexibility available in the framework described in this paper. Later sections then zoom in on individual constraints, their implementations, and discuss the relationships between them.

1.4 | Pagination strategies and related work

While Knuth and Plass[15] already introduced global optimization for micro-typography in TEX in the 1980s, pagination in today's systems is still undertaken using greedy algorithms that essentially generate column by column without looking (far) ahead.

Already in his PhD thesis, Plass[16] discussed applying the ideas behind TEX's line-breaking algorithm to the question of paginating documents containing text and floats. Since then, a number of other researchers have worked on improved pagination algorithms, eg, the work of Wohlfeil[17] addressed optimal float placement for certain types of documents in his PhD thesis and together with Brüggemann-Klein et al,[18] using dynamic programming based on the Knuth/Plass algorithm with a restricted document model. Jacobs et al[19] explored the use of layout templates that can be selected by an optimizing algorithm also based on Knuth/Plass to best fulfill a number of constraints.

Ciancarini et al[5] presented an approach (again, based on Knuth/Plass), in which the micro- and macro-typography is more tightly coupled by delaying the definite choice of line breaks and instead offering to the pagination algorithm a set of options per paragraph modeled as a flexible glue item. Using glue has the advantage that the complexity of the pagination algorithm stays low compared to the approach outlined in this paper, but the disadvantage that other aspects of the fully formatted paragraphs are unavailable to the pagination algorithm, eg, that for certain formatting, the available breaks may be of different quality. In addition, if the pagination requests that a paragraph format itself to a certain height, for example, 3.5 lines, it can only fulfill that request to the nearest line number and as these errors accumulate, it is possible that the optimal solution is missed.

A quite different approach was taken by Piccoli et al[6] to provide the necessary flexibility that enables a globally optimizing algorithm (they too use Knuth/Plass) to find solutions: They select and combine prepared page templates to find an optimal distribution of text among the template placeholders such that all pages are completely filled. The input text stream is split in chunks, and each chunk must completely fit into a template placeholder. Thus, the granularity of chunks will both determine how huge the search space will get (and therefore, how long it will take to find a solution) and how well the text gets distributed among the placeholders.

They then achieve filling the placeholders completely with text by manipulating the font size of the text for a consecutive set of placeholders that belong to what they call a flow, eg, the text following a heading on a page. For a journal with many shorter articles that needs to be assembled totally automatically, that approach may generate acceptable quality as long as the differences in text density stay really low and text that is experienced by the reader as belonging together (eg, from a single article) does not show changes in density.

The authors have implemented their algorithm as an extension to a commercial environment, thus providing a complete production environment as the final rendering is left to Adobe's InDesign® that is provided with the selected template sequence and the text chunks to be rendered as part of the template placeholders.

However, from a typographical perspective, this approach is questionable, especially if text is set in multiple columns as the human eyes are quite capable of spotting even very small changes in vertical sizes and density.[§] This means that for continuous text, such as novels, this is not an option if the intention is to produce high-quality results.

[§]There has been 1 paragraph set on this page with a font reduced by about 0.3 mm but without changing the line spacing. As a result, that particular paragraph needs 1 line less—does it stand out? In my opinion, it does; at least, there is a slight queasy feeling when looking at the page, even if you cannot immediately pinpoint the source.

Thus, why has no widely used production system, whether it be TEX or any other, started to use a global optimizing pagination algorithm up to now?

The answer is at least 2-fold: On one hand, due to the fact that pagination has to deal with unrelated input streams, the problems in this space are much harder than those in line breaking, although superficially, they have a lot in common. As a result, most of the research work so far has focused on experimenting with certain aspects only (with the possible exception of the work carried out by Jacobs et al[20]) and was less concerned in providing a production-ready solution initially. On the other hand, typesetting requires much more than pagination, and any generally usable system implementing a new pagination either needs to also provide all the features related to micro-typography (which is a huge undertaking) or it needs to integrate into an existing system like TEX or any commercial engine.

On the commercial side, the complexity of full or even only partial optimization was so far probably considered too high compared to any resulting benefits, and the open source TEX system (while offering most aspects needed for high-quality typesetting[¶]) is monolithic and so optimized for speed that it is very difficult to extend it or replace some of its algorithms.

1.5 | The main contributions provided by this paper

This paper presents a framework for globally optimizing the paginations of documents using flexible constraints that allow the implementation of typical typographic rules. These can be weighted against each other to guide the algorithm toward a particular desired outcome. In contrast to the prototypes discussed in the literature, it has the full micro-typographic functionality of the TEX engine at its disposal and is thus able to typeset documents of any complexity.

It uses an adaption of the line-breaking algorithm by the aforementioned work,[15] which is a natural approach also deployed by other researchers. However, due to the fact that there is limited flexibility in pagination compared to line breaking, a straightforward adaptation of the Knuth/Plass algorithm would not resolve the pagination problem: It provides a globally optimizing algorithm, but one that runs out of alternatives to optimize in nearly all cases.

The other main contribution of this paper is therefore the extension of this algorithm to include 2 methods that add flexibility to the pagination process without compromising typographic quality and traditions. These are the support for the following.

Paragraph variants: Identification of paragraphs that can be typeset to different numbers of lines without much loss of quality, then using these variants as additional alternatives in the optimizing process.

Spread variants: Support for page spreads (ie, all columns of a double page) to be run short or long, thereby increasing the number of alternatives for the algorithm to optimize.

Using both extensions, enough flexibility is added to the pagination process that the globally optimizing algorithm is able to find a solution for nearly any document in acceptable time without running out of candidate solutions to optimize.

1.6 | The scope and restrictions of the framework

The framework is intended to support a wide variety of different applications but there are, of course, some assumptions that restrict it in one or the other way. It assumes that the input to

[¶] For a discussion of TEX's limitations and failures, see the work of Mittelbach[21] and for an update 23 years later.[22]

the algorithm is a sequence of precomposed textual material, intermixed with vertical spaces and that the task is to paginate this galley into columns and pages of possibly different but predefined heights. In particular, this means that the horizontal width of an object plays no role in the algorithm (everything has the same width). As a consequence, the framework cannot optimize designs that allow textual material to be formatted to the choice of different widths.

The framework also assumes that the heights of columns used in pagination is known a priori and does not depend on the content of the textual material poured into it. It is therefore not possible for the algorithm to balance textual material across several columns on a page and then restart the flow on the same page as that would be equivalent to having variable column heights that depend on signals from within the textual material.

The algorithm assumes that the textual flow is continuous and is placed into columns in a predefined order. As implemented, the writing direction is top to bottom and left to right, but other writing systems could be easily supported as that involves only a simple and straightforward transformation of the algorithm's results prior to typesetting the final document.

Other than that, the framework poses no restrictions and supports all typical typographical tasks using a system of user-specifiable constraints. In particular, it is possible to specify

- whether columns need to be filled exactly (ie, should align at the bottom);
- the alignment of columns across a spread;
- how much "extra" white space is considered acceptable in a column;
- the management of micro-typographical features such as preventing "widows and orphans" or hyphenation across columns, etc;
- the amount of paragraph length variations that is allowed;
- the importance of conserving space, ie, preferring less pages;
- design criteria, such as a preference for headings to (always) start a new column;
- requiring a full last page[#];
- any desirable or forced column breaks to affect the algorithm.

These user-specified requirements can be either absolute (in which case, the algorithm will not consider any solution that violates them) or they can be formulated as a preference, with the different constraints weighted against each other according to user specification that indicates their relative importance. This is done by attaching higher or lower cost values to individual requirements. If even more granularity is needed for experimentation and research, then any part of the objective function used for optimization can be easily adjusted.

As implemented, the framework is based on LuaTEX for reasons explained at the beginning of Section 2. However, as the algorithm for determining the optimal pagination is independent of the underlying typesetting system used, it would be possible to build a similar framework with any other typesetting engine that is capable of generating the necessary abstract representation of the galley as input for the algorithm (as discussed in Section 2.1.2). The engine should also be able to format a paragraph to variable number of lines and measure the quality of each formatting.[‖] Finally, it must be able to accept external directives while paginating the final document (Section 2.1.4).

[#] Of course, if this is specified as a strict requirement, then the algorithm may not be able to find any solution, depending on the given input. A simple example would be a short document that simply does not have enough material for a single page.

[‖] The algorithm is still capable of operating if the formatter is not capable of this, but the number of alternative solutions to optimize will be greatly reduced. As shown in Section 4.7, this will often mean that documents have no solution that adheres to the specified constraints.

1.7 | Downside of applying global optimization

While globally optimizing the pagination to further automate the typesetting process sounds like a good idea, there are a number of issues related to it that need to be taken into consideration and require further research.

First of all, global optimization means that any modification in the document source can result in pagination differences anywhere in the document. This is already now a source of concern for TEX users experiencing situations where *deleting* a word results in a paragraph getting *longer* or being broken differently across columns. By optimizing the pagination, such types of problems are moved from the localized level of micro-typography to the overall document level. Just consider a book revision where a few misspellings are corrected and instead of regenerating a handful of photographic plates for these pages, the publisher has to generate a fully reformatted book.[**]

However, there are also problems related to the interaction between globally optimized pagination and automatically generated (textual) content. If such generated content depends on the pagination, for example, if a text "see Figure 3 on the following page" changes to the much shorter text "see Figure 3 on page 13," then this generates feedback loops between micro- and macro-typography, ie, it might change the formatting, which might change the generated text and the formatting. It is not difficult given an arbitrary pagination rule set to construct a document for which there is no valid formatting possible under the conditions of this rule set. While such situations can already occur with pagination generated by greedy algorithms, they are far more likely if global optimization (especially with variant formatting for higher flexibility) is used.

2 | A GLOBALLY OPTIMIZING FRAMEWORK USING TEX

As the open source program TEX by Don Knuth is undisputedly one of the best typesetting systems in existence when it comes to micro-typography or math typesetting, it is a natural candidate for any attempt to implement improved pagination algorithms as all other aspects of typesetting are already provided with high quality and due to its large user base there are immediately many people who could benefit from an improved program.

Unfortunately, the original program by Knuth[23] is of monolithic design and highly optimized so that modifying its inner working has proven to be a serious challenge. There have been a number of such attempts though, and three of them have established themselves in the world-wide community: pdfTEX is an engine written by Thánh[14] as part of his PhD thesis that was the first engine directly generating PDF output and it also provided a number of micro-typography extensions, such as protrusion support and font expansion (hz-algorithm). These days, this TEX-extension has become the default engine in most installations, ie, the program being called, when people are processing a TEX file. The other two (still more or less under active development) are XƎTEX (see, for example, the work of Goossens[24]) and LuaTEX, implemented by the LuaTEX development team.[25]

The interesting aspect of LuaTEX is that it combines the features of a complete (and, in fact, extended) TEX engine with a full-fledged Lua interpreter that allows the execution of Lua code inside of TEX with full access to the internal TEX data structures and with the ability to hook such Lua code at various points into most TEX algorithms, enabling the code to modify or even

[**]The solution in that case would be to introduce explicit pagination commands in strategic places to keep the pagination unchanged, even if through the algorithm's eyes it is no longer optimal.

to replace them. As Lua is an interpreted language, there is no need to compile a new executable whenever some Lua code is modified; all it needs is the base LuaTEX engine to be available (which is a standard engine in all major TEX distributions such as TEX Live).

As of fall 2016, LuaTEX has reached version 1.0; thus, the development activities are expected to slow down with more focus on stability and compatibility compared to the situation in the past. Thus, this version of LuaTEX can easily serve as a very versatile test bed for developing algorithms that can be directly tried and used by a large user base. For this reason, the framework described in this paper is based on LuaTEX.

2.1 High-level workflow

The framework presented here consists of four phases and uses LuaTEX for the reasons outlined above. A high-level overview is given in Figure 3 showing the phases, their inputs and outputs, and the types of user-specifiable constraints that are applicable in the different phases.

2.1.1 Phase 1 (preprocessing)

The document, which consists of standard TEX files, is processed by a TEX engine without any modification until all implicit content (eg, table of contents, bibliography, etc) is generated, and all cross-references are resolved.[††] This implicit content is stored in auxiliary files by TEX and reused as input in Phase 2 during the galley generation and again, in Phase 4 when producing the optimally paginated document.

2.1.2 Phase 2 (galley metadata generation)

The engine is modified to interact with TEX's way of filling the main vertical list (from which, in an asynchronous way, TEX later cuts column material for pagination). An overview about the workflow during that phase is shown in Figure 4 on the following page.

Engine modification when moving material to the galley in Phase 2

In particular, whenever TEX is ready to move new vertical material to the main vertical list, this material is intercepted and analyzed. Information about each block (vertical height, depth, stretchability, if any, and penalty of a breakpoint) is then gathered and written out to an external file. In TEX, boxes (such as lines of text) have both a vertical height and a vertical depth, which is the amount of material that appears below the baseline, eg, the vertical size of descenders of letters such as "p" or "g." The total vertical size of boxes is then the sum of height and depth. This distinction is important when filling columns with material, because the depth of the last line must not be taken into consideration when determining the total vertical size occupied by the material (while the depth of all other lines is). This reflects the fact that columns and pages should align on the baseline of the last text line regardless of whether such a line has descenders.

[††] Cross-references to pages or columns in the final document can only be approximated at this stage as the final position in the optimally paginated document is not yet known. We therefore use the values produced when paginating the document with standard TEX (ie, with a greedy pagination algorithm) as placeholders and hope that they are roughly the same width. They are then replaced by the correct values in Phase 4. However, as explained in Section 1.7, documents with cross-references that depend on the pagination may not allow valid formatting and may need a manual intervention that overwrites one or the other optimization criteria.

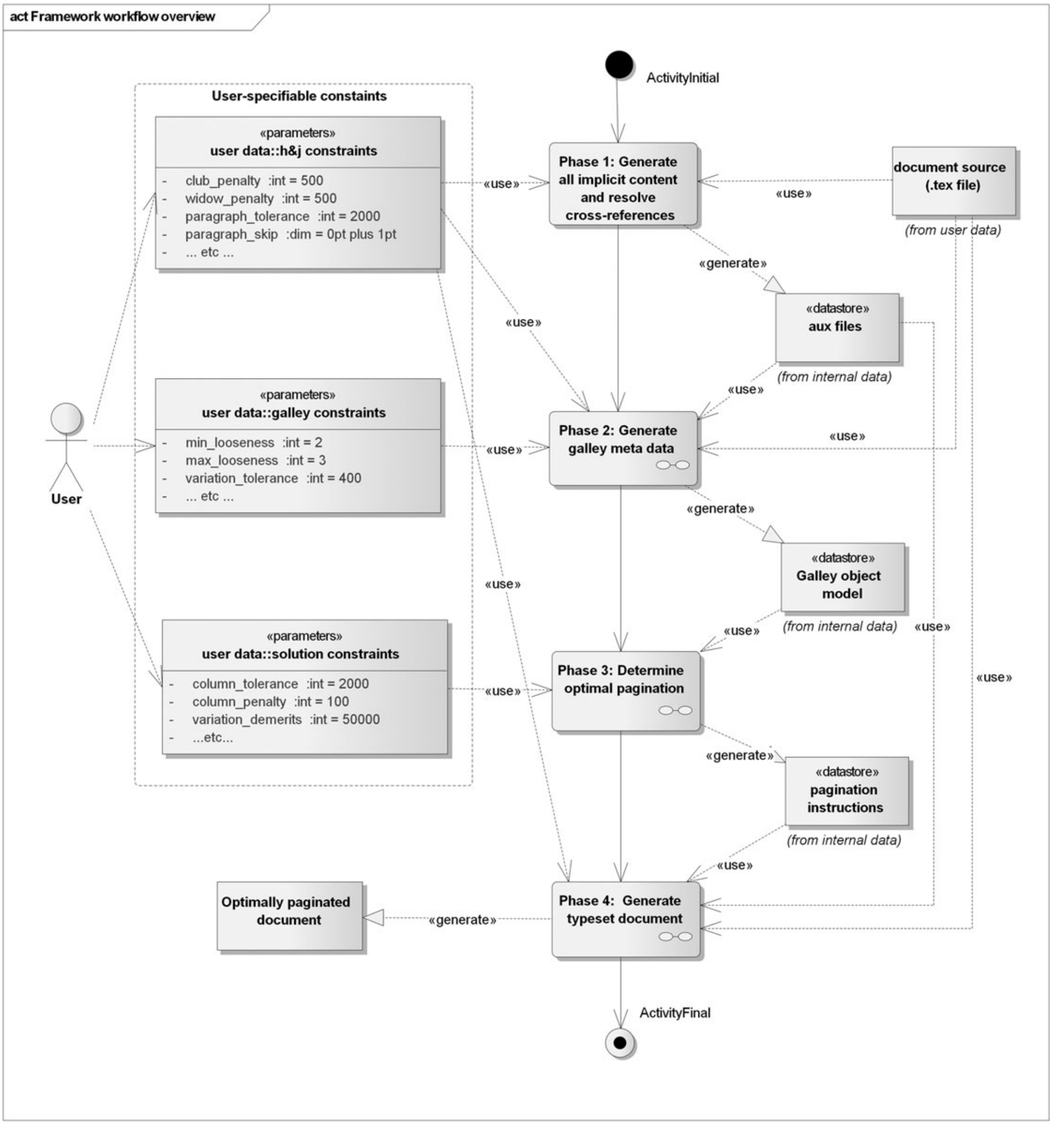

FIGURE 3 High-level framework overview (Phases 1 to 4)

If possible, data are accumulated, eg, several objects in a row without any possibility for breaking them up are written out as a single data point to reduce later processing complexity.

The modification is also able to interpret special flags (implemented as new types of "whatsit nodes" in TEX engine lingo) that can signal the start/end or switch of an explicit variation in the input source. This information is then used to structure the corresponding data in the output file for later processing.[‡‡]

[‡‡]This interface could be extended at a later stage to support controlling of the algorithm used in Phase 3 (pagination) from within the document, eg, to guide or overwrite its decisions locally.

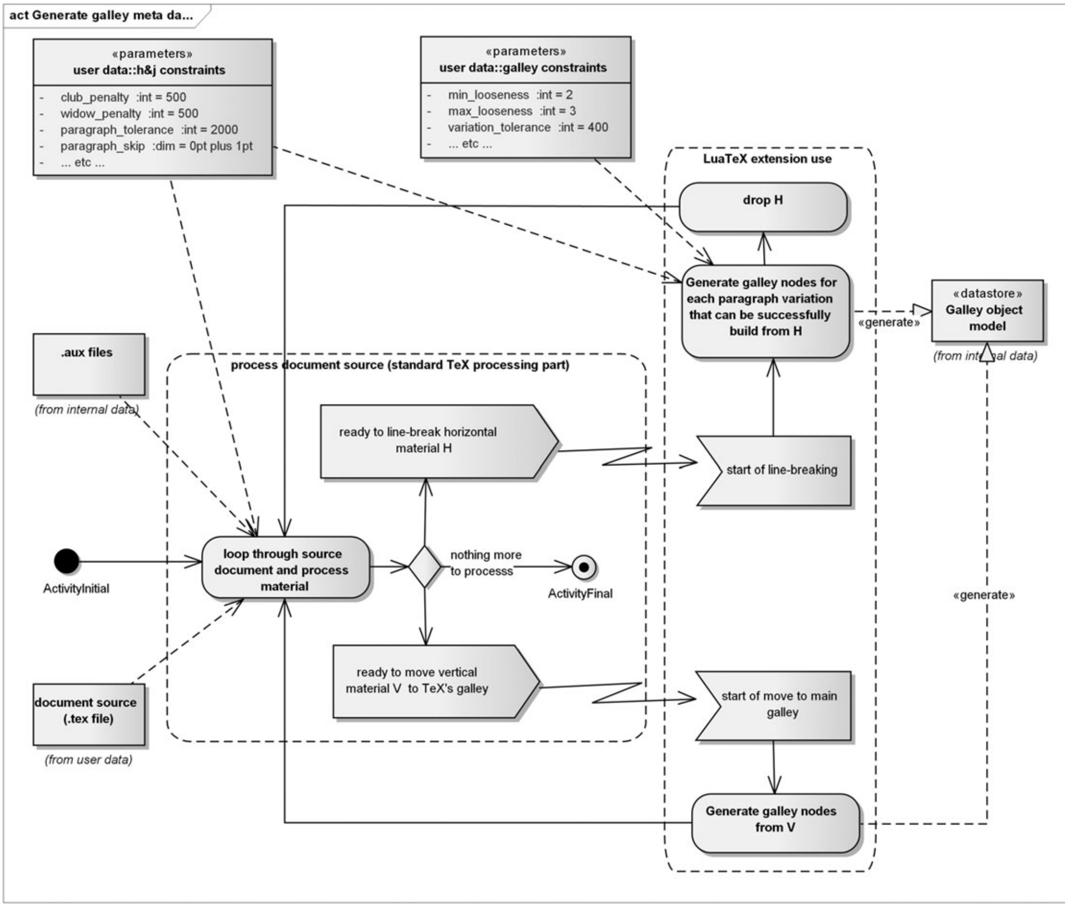

FIGURE 4 Galley metadata generation (Phase 2)

Engine modification when generating paragraphs in Phase 2

The second modification to the engine is to intercept the generation of paragraphs targeted for the main galley[§§] prior to TEX applying line breaking.

For each horizontal list that is passed to the line-breaking algorithm, the framework algorithm determines the possible variations in "looseness" within the specified parameter settings (galley constraints parameters). An example of a paragraph reformatted to a different number of lines is shown in Figure 5 on the next page.

For this, the paragraph is first broken into lines according to the given h&j constraints resulting in a paragraph with ℓ lines. Then, TEX is asked to try again and line-break the horizontal list to generate paragraphs with lines between $\ell -$ `min_looseness` and $\ell +$ `max_looseness`. For these attempts a special `variation_tolerance` is used, which can be set to a value different from the `paragraph_tolerance` used on normal paragraphs.

[§§]Paragraph variations in other places, eg, inside float boxes, marginal notes, footnotes, etc, are currently not considered. Thus, those objects always have their natural (fixed) dimensions. Extending the framework in that direction would be possible but would considerably complicate the mechanism without a lot of gain.

'I won't indeed!' said Alice, in a great hurry
to change the subject of conversation. 'Are you–
are you fond–of–of dogs?' The Mouse did not
answer, so Alice went on eagerly: 'There is such
a nice little dog near our house I should like to
show you! A little bright-eyed terrier, you know,
with oh, such long curly brown hair! And it'll
fetch things when you throw them, and it'll sit
up and beg for its dinner, and all sorts of things–
I can't remember half of them–and it belongs to
a farmer, you know, and he says it's so useful,
it's worth a hundred pounds! He says it kills all
the rats and–oh dear!' cried Alice in a sorrowful
tone, 'I'm afraid I've offended it again!' For the
Mouse was swimming away from her as hard as
it could go, and making quite a commotion in
the pool as it went.

optimal line breaks

```
looseness = 0
tolerance = 4000
```

'I won't indeed!' said Alice, in a great hurry to
change the subject of conversation. 'Are you–are
you fond–of–of dogs?' The Mouse did not answer,
so Alice went on eagerly: 'There is such a nice lit-
tle dog near our house I should like to show you!
A little bright-eyed terrier, you know, with oh,
such long curly brown hair! And it'll fetch things
when you throw them, and it'll sit up and beg for
its dinner, and all sorts of things–I can't remem-
ber half of them–and it belongs to a farmer, you
know, and he says it's so useful, it's worth a hun-
dred pounds! He says it kills all the rats and–oh
dear!' cried Alice in a sorrowful tone, 'I'm afraid
I've offended it again!' For the Mouse was swim-
ming away from her as hard as it could go, and
making quite a commotion in the pool as it went.

run short (tight)

```
looseness = -1
tolerance = 500
```

'I won't indeed!' said Alice, in a great hurry
to change the subject of conversation. 'Are you–
are you fond–of–of dogs?' The Mouse did not
answer, so Alice went on eagerly: 'There is such
a nice little dog near our house I should like
to show you! A little bright-eyed terrier, you
know, with oh, such long curly brown hair!
And it'll fetch things when you throw them,
and it'll sit up and beg for its dinner, and
all sorts of things–I can't remember half of
them–and it belongs to a farmer, you know,
and he says it's so useful, it's worth a hun-
dred pounds! He says it kills all the rats and–
oh dear!' cried Alice in a sorrowful tone, 'I'm
afraid I've offended it again!' For the Mouse was
swimming away from her as hard as it could go,
and making quite a commotion in the pool as
it went.

run long (loose)

```
looseness = 1
tolerance = 500
```

'I won't indeed!' said Alice, in a great hurry
to change the subject of conversation. 'Are
you–are you fond–of–of dogs?' The Mouse did
not answer, so Alice went on eagerly: 'There
is such a nice little dog near our house I
should like to show you! A little bright-eyed
terrier, you know, with oh, such long curly
brown hair! And it'll fetch things when you
throw them, and it'll sit up and beg for
its dinner, and all sorts of things–I can't re-
member half of them–and it belongs to a
farmer, you know, and he says it's so use-
ful, it's worth a hundred pounds! He says
it kills all the rats and–oh dear!' cried Al-
ice in a sorrowful tone, 'I'm afraid I've of-
fended it again!' For the Mouse was swimming
away from her as hard as it could go, and
making quite a commotion in the pool as it
went.

run very long—too loose!

```
looseness = 2
tolerance = 4000
```

Explanation: This paragraph from Alice in Wonderland is shown in in Figure 1 on page 6. For this paragraph the globally optimizing pagination algorithm selected the variant with a `looseness` of `−1` as shown in Figure 2 on page 7. The paragraph appears in the second column of page 5 in both figures.

In small column measures one should set the default tolerance fairly high (4000) to ensure that TEX finds a solution in situations where line breaking is problematic. This particular paragraph, however, breaks nearly perfectly, thus a tolerance of 200 would have produced the same result.

Trials with a non-zero looseness should use a smaller tolerance to avoid bad results when diverging from the optimum number of lines, e.g., with this paragraph two extra lines are only possible with a tolerance of 4000 or higher. With the standard constraint settings this variant would therefore be considered a failure.

FIGURE 5 A paragraph from Alice under different `looseness` settings

The paragraph tolerance defines whether lines are considered acceptable to be part of an optimal solution, thus with a higher tolerance TEX will have more possibilities to chose from. It will still use only lines with low tolerance if that leads to a solution, but it may not find any solution at all if the tolerance is set too low and line breaking is difficult (eg, in narrow columns). Therefore, allowing lines with high badness in emergencies might be better than overfull lines, because no solution was found.

The situation with paragraph variants, however, is different: Variants are intended to provide some additional flexibility for the pagination process, but they should only be used if their quality is sufficiently high. Therefore, the line-breaking trial for variants should not consider lines with high badness as acceptable, which means that the tolerance in these trials should be set to a much lower value.

However, for positive values of looseness, it is not enough to check if TEX could build a paragraph matching it, as TEX by default uses a fairly naive approach that would always result in the last line containing only a single word or even only part of a single word whenever the paragraph is lengthened.

It is therefore important to first manipulate the horizontal material to prevent this from happening and ensure that "loosened" paragraphs stay visually acceptable to the human eyes.[¶¶]

The approach is to add extra penalties to the line break possibilities near the end of the paragraph (including those due to hyphenation) so that TEX will prefer to break elsewhere unless there is no good alternative. Thus, if feasible, TEX will put at least a few words into the last line. The behavior can be observed in Figure 5, where the loose setting still has two words on the last line, but the very loose setting ends with a single word, because, otherwise, the paragraph would have been even worse.

[¶¶]As the paragraph variations with this manipulation applied are used as input to the optimizing algorithm used in Phase 3, we will later have to reapply the manipulation in exactly the same way in Phase 4 (typesetting) on any variant paragraph that has been chosen in Phase 3 as being part of the optimal solution to ensure that it is typeset accurately.

For each possible variation, the paragraph-breaking trial then determines the exact sequence of lines, vertical spaces, and associated penalties under that specific “looseness” value.

Such trials with a special looseness may fail, either because the requested looseness cannot be attained at all, or only with a tolerance value exceeding the `variation_tolerance` constraint, or because the resulting paragraph has overfull lines (like this paragraph).

Solutions with overfull lines can happen because the trials are typeset with a special tolerance value. Under this tolerance, the paragraph may not have any acceptable solution (ie, without overfull lines). Starting from such a “bad” paragraph breaking as the best possibility, TeX might report success in making it even shorter because that does not make it worse in TeX’s eyes (ie, they are considered to be equally bad).[##]

The results of each successful trial are then externally recorded together with the associated “looseness” value of that variation. For a given looseness value, the line breaking chosen by TeX will be optimal in the work of Knuth,[8(p103)] but, of course, it will usually be of a lower quality compared to the optimal line breaking with ℓ lines (ie, the number of lines with looseness 0). The algorithm accounts for this by applying a user-customizable cost factor whenever such a variant is chosen in Phase 3 below.

Finally, instead of adding a vertical list representing the formatted paragraph on TeX’s main vertical list, the material is dropped, and a single special node is passed so that the paragraph material is not collected again by the first modification described above.

As already indicated, the user-specifiable constraints used in this phase are those dealing with the break costs during h&j (eg, handling of widows and orphans, breaks before headings, etc), specifications of flexible vertical spaces (eg, skips between paragraphs, before headings, around lists, etc), and the galley variation constraints that describe what kind of paragraph variations are deemed acceptable and what additional costs to add if a variation with a lower paragraph quality is chosen.

As the result of this phase, the external file will hold an abstraction of the document galley material (the galley object model in Figure 4), including marked-up variations for each paragraph for which valid variations exist.

2.1.3 | Phase 3 (determining the optimal pagination)

The result of Phase 2 (ie, the galley object model) is then used as input to a global optimizing algorithm modeled after the Knuth/Plass algorithm for line breaking that uses dynamic programming to determine an optimal sequence of column and page breaks throughout the whole document. Compared to the line-breaking algorithm, this page-breaking algorithm provides the following additional features.

- Support for variations within the input: This is used to automatically manage variant break sequences resulting from different paragraph breakings calculated in Phase 2 but could also be used to support, for example, variations of figures or tables in different sizes or similar applications.
- Support for shortening or lengthening the vertical height of double spreads to enable better column/page breaks across the whole document.

[##]This behavior caused some surprise during the implementation of the algorithm until it was understood that an explicit check for overfull lines is needed at this point.

- Global optimization is guided by parameters that allow a document designer to balance the importance of individual aspects (eg, avoiding widows against changing the spread height or using suboptimal paragraphs) against each other.

The influence of such user-specified constraints is discussed in Section 3. Details of the algorithm are then described in Section 4.

The result of this phase will be a sequence of optimal column break positions within the input together with length information for all columns for which it applies. Also recorded is which of the variants have been chosen when selecting the optimal sequence.

2.1.4 | Phase 4 (typesetting)

This phase again uses a modified TeX engine that is capable of interpreting and using the results of the previous phases. For this, it hooks into the same places as the modifications in Phase 2, but this time, applying different actions (see Figure 6).

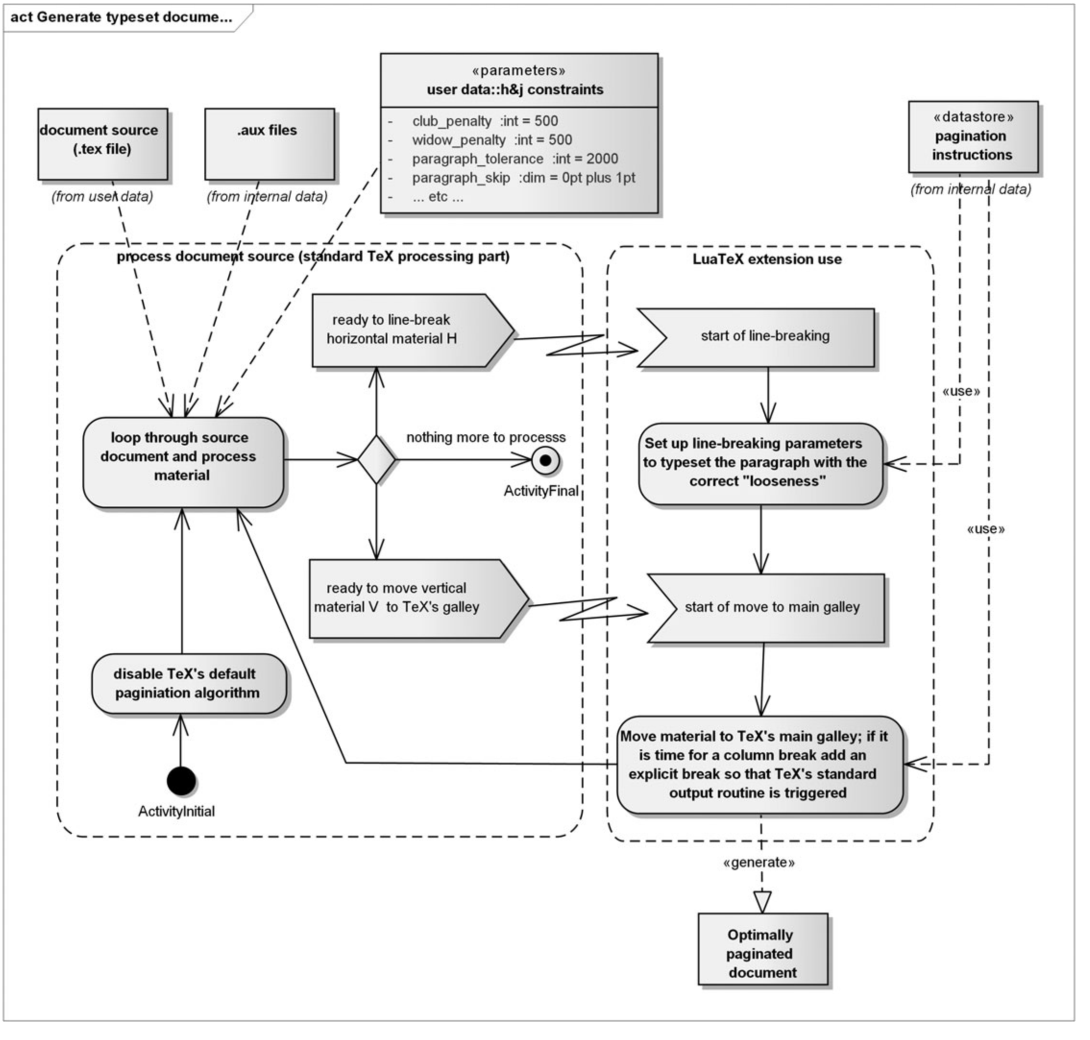

FIGURE 6 Generate the optimally paginated document (Phase 4)

To begin with, the vertical target size for gathering a complete column will be artificially set to the largest legal dimension so that by itself, the TEX algorithm will not mistakenly break up the galley at an unwanted place due to some unusual combination of data.[‖]

Engine modification when generating paragraphs in Phase 4

Whenever TEX gets ready to apply line breaking to paragraph material for the main vertical list, the modification looks up with which "looseness" this paragraph should be typeset and adjusts the necessary parameters so that TEX generates the lines corresponding to the variation selected in the optimal break sequence for the whole document determined in Phase 3 (pagination). At this point, the algorithm also reapplies the modification described in Section 2.1.2 page 15 on any paragraph for which a variation was chosen.

Engine modification when moving material to the galley in Phase 4

While TEX is moving objects to the main vertical list the algorithm keeps track of the galley blocks seen so far, and when it is time for a column break, according to the optimal solution, it will explicitly place a suitable forcing penalty onto the main vertical list so that TEX is guaranteed to use this place to end the current column or page. Again, as a safety measure, other penalties seen at this point that should not result in a column break will be either dropped or otherwise rendered harmless so that TEX's internal (greedy) page-breaking algorithm is not misinterpreting them as a "best break" by mistake.

Finally, whenever TEX has finished a column (due to the fact that we have added an explicit penalty in the previous step), we will arrange for the correct target dimensions for the current column according to the data from Phase 3 (pagination). This is done immediately after TEX has decided what part of the galley it will pack up for use in its "output routine" (which is a set of TEX macros) but before, this routine is actually called.[***]

The result is a paginated document with globally optimized column breaks according to the user-specified constraints.

2.2 | Notes on the workflow phases

Phase 1 (preprocessing 2.1.1) is necessary to generate all implicit content so that it will be considered in the following phases. Without this phase, the page-breaking step in Phase 3 would base its evaluation on the wrong input.

Phases 2 (galley generation 2.1.2) and 4 (typesetting 2.1.4) will require a modified/extended TEX engine. The workflow uses the LuaTEX engine for this purpose as it internally provides a Lua interpreter to implement the modifications and the necessary callbacks into the TEX algorithms so that the new code can easily take control and provide the necessary changes.

The algorithm used in Phase 3 (pagination 2.1.3) is also implemented in Lua. As this phase is executed without any direct involvement of a TEX engine processing the source document, this code could have been written in any computer language (and could probably be faster, depending on language choice and implementation). Nevertheless, the use of Lua was deliberate, as it allows

[‖] As long as the calculation for deciding on a column break used by TEX and the one used by the algorithm deployed in Phase 3 are exactly the same, this is actually not necessary. However, requiring a 100% correspondence is not a useful restriction; thus, this is a safety measure against deliberate or unintentional differences in this place.

[***] This way the engine modifications are largely transparent for the TEX macro level and the modification will work with some small adjustments with any macro flavor of TEX, eg, LATEX, plain TEX, etc.

to use the LuaTEX engine[†††] in all phases and this means that the workflow can be executed using a standard TEX installation, ie, is out of the box available for the millions of LATEX-users and other TEX flavors without the need to install any additional software programs.[‡‡‡]

While the typesetting phase (Phase 4 Section 2.1.4) claims that the result is a globally optimized formatted document, it does not actually claim that it is a correctly formatted document and as explained in Section 1.7, this may, in fact, not be the case. The mechanisms available in LATEX will detect this situation, but the framework currently makes no attempt to resolve this problem if it arises. Depending on the exact nature of the issue a further run through Phases 2 to 4 might resolve it. However, if the formatted result oscillates between two or more states then manual intervention is necessary.

As indicated in Figure 3 on page 13 the behavior of the framework is customizable through parameterization during all four phases. The next sections will show that more complex customizations can be carried out as well, by providing alternative code written in Lua that modifies the framework algorithms.

3 | THE CONSTRAINT MODEL USED FOR GLOBALLY OPTIMIZED PAGINATION

In this section, we discuss the constraints necessary to implement common and less common typographic design criteria for pagination. In part, these are already provided by TEX (though used there by its greedy pagination algorithm and not in the context of global optimization). Additional ones are added for exclusive use by the globally optimizing pagination algorithm discussed in Section 4.

A number of user-specified constraints available as parameters of the TEX engine define the set B of breakpoints available in a given document galley and the numerical "costs" associated with such breakpoints (see Section 3.1). With P, we denote the set of all partitions of the galley along such breakpoints, ie, the set of all subsets of B.

Furthermore, user-specifiable constraints are implemented by defining a suitable objective function $\mathcal{F}$ that numerically measures the "inverse quality" (lower values are better) of a partition $p = \{b_0, \dots, b_n\} \in P$, ie, how well p adheres to the given constraints. By attaching different weights or using different formulas in the objective function or when calculating the breakpoint costs, it is possible to adjust the relationships between different (possibly conflicting) constraints and favor some over others. Some examples are given below.

Thus, abstractly speaking, the act of globally optimizing the pagination of a galley means finding a $p \in P$ for which $\mathcal{F}(p)$ is minimal. Doing this by evaluating $\mathcal{F}(p)$ for every possible partition is impractical as most of the partitions will result in an impossible or ridiculously bad pagination (ie, with overfull or nearly empty columns). Furthermore, the number of partitions grows exponentially in the number of breakpoints so that even for small galleys the number of cases to evaluate will exceed the capabilities of any computer. It is therefore important to reduce number of evaluations significantly while still ensuring that

$$\min_{p \in P} \mathcal{F}(p) \tag{1}$$

[†††] LuaTEX can be run as a standalone Lua interpreter by calling it under the name `texlua`.

[‡‡‡] Lua code is interpreted and available in the form of ASCII files. It can therefore be easily provided as part of the standard TEX distributions or (with older installations) manually downloaded and installed.

will be found. As we will see, this problem can be solved with dynamic programming techniques as long as the objective function has certain characteristics.

3.1 | Constraining the available breakpoints

TeX already provides a sophisticated breakpoint model to describe different typographic requirements. In TeX, it is used to guide its greedy pagination algorithm but clearly all of these constraints make sense for a globally optimizing algorithm as well; thus, we use those unchanged. The breakpoints of a galley are modeled in TeX as follows.

- Breaks are implicitly possible in front of vertical spaces if such spaces directly follow a box (eg, a line of text). Such breaks are normally neutral, ie, have no special costs associated with them (although there is a parameter to change that). As lines of a paragraph are always separated by spaces (to make them line up at a distance of a "baselineskip"), this means that it is usually possible to break after each line of text.
- Breaks are also possible at the so-called penalty nodes that can be explicitly added through macros (such as a heading command) or implicitly by TeX through parameters in certain situations. The value of the penalty defines the "cost" to break at this point: A negative value means there is an incentive to break here, and a positive value means a break at this point is less desirable.
- However, a value of 10 000 or higher means a break is totally forbidden, thus by adding a penalty with that value a break at a certain point can be prevented.
- In the opposite direction a value of −10 000 or less means that a break is forced, ie, TeX will always break at this point.
- A number of typographical conventions are modeled by TeX through parameters that generate penalty nodes, eg, between the first and second lines of a paragraph TeX adds a penalty with the value of `\clubpenalty` (to model orphans) and between the last and the second last it adds a penalty of `\widowpenalty`. If a line ends in a hyphen it adds `\brokenpenalty`, etc.

Thus, by setting such penalty parameters to appropriate values, certain breakpoints can be made more or less attractive (or can be totally forbidden). For example, LaTeX by default adds a penalty of −300 in front of a section heading, thus breaking in front of headings is encouraged. In the opposite directions, widows and orphans are frowned upon therefore `\widowpenalty` and `\clubpenalty` have a default value of 150 and many journal designs even require 10 000, ie, totally forbid orphans and widows.

3.2 | Constraining the column "badness"

As mentioned in the introduction, one quality factor for a good pagination is the white space distribution in the columns, ie, how far this distribution deviates from the optimal distribution as specified by the design for the document. If every vertical space s in a document has a natural height $\mathcal{H}_s$ and an acceptable[§§§] stretch $\mathcal{S}_s^+$ and shrink $\mathcal{S}_s^-$, then it is possible to define a function that calculates a "badness" for a column that contains a certain amount of material.

Intuitively speaking, that badness should be 0 if all spaces in the column are set exactly to their natural height, and it should increase if the spaces have to be stretched or need to be compressed

[§§§] Acceptable, in the eyes of the designer of the particular document design.

to fit all material into the column. Compression beyond a certain amount should not be possible (to avoid overlapping material in the column); thus, a natural limit would be disallowing more than the available shrink.

Stretching beyond the available stretch amount is certainly also undesirable. However, experiments show that with many documents it is not possible to find any solution at all if the allowed space distribution is handled too rigidly. A practical badness function should therefore penalize such loose columns heavily, but not totally disallow them.

The badness function used by default in the algorithm outlined in this paper is given by

$$\underset{\mathrm{col}\,i}{badness}(\langle\text{material}\rangle) = \begin{cases} 0, & \text{if there is infinite stretch within } \langle\text{material}\rangle \\ \infty, & \text{for } r < -1, \text{ ie, column is overfull} \\ 100|r|^3, & \text{for } r \geqslant -1, \end{cases} \tag{2}$$

where r is the ratio of "space needed to fill column i" and stretch available, ie, $\sum_s \mathcal{S}_s^+$ (in case of stretching) or the ratio of "shrink amount necessary to fit the material into column i" and the shrink available, ie, $\sum_s \mathcal{S}_s^-$. If stretching or shrinking is required but no stretch or shrink available, we set $r = \infty$; thus, the badness too will become ∞ in the above formula.

The precise form of the badness formula in Equation (2) is admittedly an arbitrary choice,[¶¶¶] but due to its cubic form, it does penalize larger deviations from the desired state more strongly and thus models the general expectation quite well. Nevertheless, experimenting with different functions to see how that influences, the algorithm behavior could be an interesting study in itself. This is supported by the framework by providing the badness function as a user-redefinable Lua function.

With a badness function like the one given in (2), we can specify a customizable constraint that drops inadequate solutions by defining a constant $c_{\text{tolerance}}$ such that any solution containing a column with a badness higher than $c_{\text{tolerance}}$ is rejected.

Thus, if $p = \{b_0, \ldots, b_n\}$ is a partition of the document into n columns (with b_0 and b_n the document start and end, respectively, and the other b_i the chosen breakpoints) and if c_i is the cost associated with breakpoint b_i we can define a simple objective function $\mathcal{F}$ as follows:

$$\mathcal{F}(p) = \begin{cases} \sum_{i=1}^{n} \underset{\mathrm{col}\,i}{badness}(b_{i-1}, b_i) + c_i, & \text{if } \forall i : \underset{\mathrm{col}\,i}{badness}(b_{i-1}, b_i) \leqslant c_{\text{tolerance}} \\ \infty, & \text{otherwise.} \end{cases} \tag{3}$$

The above objective function can be improved in several directions. In its current form, it does not distinguish solutions with different numbers of columns if the additional columns have badness and break costs both zero or canceling each other. However, minimizing the number of columns is an often required constraint. It also has the disadvantage of favoring solutions with a few really bad columns or really high costs over an overall lower badness and cost; in other words, it is not minimizing the overall badness and cost values at all.

The first problem can be resolved by adding a customizable c_{column} penalty that will be added for each column, ie, n times and the second problem by doing the summation over squares or cubes of the badness and costs.

[¶¶¶] Modeled after the badness function used by TeX to define the badness of lines in line breaking and the badness of pages when deciding where its greedy algorithm should cut the next page.

Thus, an improved definition for $\mathcal{F}$ is given by

$$\mathcal{F}(p) = \begin{cases} \sum_{i=1}^{n} \delta_i, & \text{if } \forall i : \underset{\text{col } i}{badness}(b_{i-1}, b_i) \leqslant c_{\text{tolerance}} \\ \infty, & \text{otherwise,} \end{cases} \tag{4}$$

with

$$\delta_i = c_{\text{column}} + \begin{cases} \left(\underset{\text{col } i}{badness}(b_{i-1}, b_i)\right)^2 + c_i^2, & \text{for } c_i > 0 \\ \left(\underset{\text{col } i}{badness}(b_{i-1}, b_i)\right)^2 - c_i^2, & \text{for } -10\,000 < c_i < 0 \\ \left(\underset{\text{col } i}{badness}(b_{i-1}, b_i)\right)^2, & \text{otherwise (forced break).} \end{cases}$$

Just like the badness function in Equation (2), the above objective function is only one possible way to constrain the solution space, but one that has proven to produce high-quality results by providing a good balance between trying to minimize the badness over all columns while putting also some weight onto a certain level of uniformity across all columns.

The influence of column badness compared to the influence of the break costs could be adjusted by changing either definitions in (2) or (4) or changing the cost values for the individual breakpoints, with the latter being the most flexible approach.

3.3 | Constraining the use of paragraph variations

For the paragraph variation extension, we provide user constraints for defining the range of variations that are tried (`min_looseness` and `max_looseness`) and the minimal quality all lines of a variant paragraph must have (`variation_tolerance`) to be considered at all.

To find the optimal line breaking for a paragraph, TeX calculates a numerical cost value for each possible line breaking. For each of the variations, this cost value will be obviously higher than the one calculated for the optimal result. Thus, we can use the difference Δ between the two and add it (multiplied with a user-customizable factor $c_{\text{para variation}}$) to the column costs if a particular variant paragraph is being used in the pagination solution. This factor then allows the user to specify how much the algorithm should disfavor solutions that use paragraph variants, ie, diverge from the optimal micro-typographic solutions when searching for a suitable pagination.

To incorporate the paragraph variation extension into the objective function $\mathcal{F}$, we have to add in another term that sums up the additional costs generated in each column due to selecting paragraph variants instead of the optimal paragraphs.

Thus, a suitable form that includes this extension would take the following form:

$$\mathcal{F}(p) = \begin{cases} \sum_{i=1}^{n} \delta_i + \sum_{i=1}^{n} \alpha_i, & \text{if } \forall i : \underset{\text{col } i}{badness}(b_{i-1}, b_i) \leqslant c_{\text{tolerance}} \\ \infty, & \text{otherwise,} \end{cases} \tag{5}$$

with

$$\alpha_i = c_{\text{para variation}} \cdot \sum_{\substack{\text{para variations} \\ \text{used in col } i}} \Delta \quad \text{(the } \Delta \text{ values depend on the para variations!).}$$

3.4 | Constraining the use of spread variations

For the double spread variation, we provide $c_{\text{spread variation}}$ as a user-specifiable constant to penalize running a column long or short. Note that with the spread variation, we introduce a dependency between columns; as for typographical reasons, all columns of a spread should be handled in the same way. That is, the first spread has only 1 page, whereas all the later ones have 2 pages. Thus, if each page has k columns and we denote by $\sigma = \{\sigma_1, \ldots, \sigma_n\}$ the modifications we make to each column, then we have $\sigma_1 = \cdots = \sigma_k, \sigma_{k+1} = \cdots = \sigma_{3k}, \sigma_{3k+1} = \cdots = \sigma_{5k}$, etc.

The objective function from (4) can then be augmented to cover both extensions as follows (note that $\mathcal{F}$ now takes both p and σ as input):

$$\mathcal{F}(p,\sigma) = \begin{cases} \sum_{i=1}^{n} (\delta_i + \alpha_i + \gamma_i), & \text{if } \forall i : \underset{\text{col}\, i}{badness}(b_{i-1}, b_i) \leqslant c_{\text{tolerance}} \\ \infty, & \text{otherwise}, \end{cases} \tag{6}$$

with

$$\gamma_i = \begin{cases} c_{\text{spread variation}}, & \text{if } \sigma_i \neq 0 \text{ (short or long spread)} \\ 0, & \text{otherwise.} \end{cases}$$

More complex definitions for γ_i are possible, for example, by providing different constraints for long and short spreads and by adding some extra costs if the sequence changes directly from long to short or vice versa, eg, a definition such as

$$\gamma_i = c_{\text{incompatible spread}} \cdot (|\sigma_i - \sigma_{i-1}| - 1) + |\sigma_i| \cdot \begin{cases} c_{\text{short spread}}, & \text{if } \sigma_i < 0 \text{ (short spread)} \\ c_{\text{long spread}}, & \text{if } \sigma_i > 0 \text{ (long spread)} \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

However, at the moment, the algorithm described in Section 4 uses the simpler definition from Equation (6).

3.5 | Finding the minimum $\mathcal{F}(p, \sigma)$

As mentioned above, given m breakpoints in a galley, we have a total of 2^m possible partitions in P to consider. This makes a simple enumeration approach, therefore, clearly intractable. However, as long as the objective function used has certain characteristics, we will see that the problem can be solved efficiently though dynamic programming techniques.

To successfully apply dynamic programming, we need to be able to formulate the problem as a number of subproblems that share common subsubproblems, and it is necessary that the problem exhibits optimal substructure. By this, we mean that an optimal solution to the problem consists of optimal solutions to its subproblems.

By solving the common subsubproblems only once and remembering an optimal solution for them, we can successively build optimal solutions to larger subproblems, knowing that due to the optimal substructure, an optimal solution to a subproblem will only contain optimal subsubproblems. Thus, we do not have to remember any of the nonoptimal solutions to subsubproblems, thereby considerably reducing the solution space to evaluate.

If we look at the pagination problem of finding $p \in P$ with minimal $\mathcal{F}(p, \sigma)$ for some given σ, we can easily see that it can be formulated as a problem with overlapping subproblems and that with an objective function like the one given in Equation (6), it exhibits an optimal substructure.

Assuming each page has k columns and given a partition $p = \{b_0, \dots, b_n\}$ and a spread variation sequence $\sigma = \{\sigma_1, \dots, \sigma_n\}$, then this partition will generate $s = \lfloor (n-1+k)/2k \rfloor + 1$ spreads the last one possibly not fully filled with columns. If we denote by n' the column number of the last column in the second-last spread, ie, $n' = (2(s-1)-1)k$, then $p' = \{b_0, \dots, b_{n'}\}$ defines a subpartition of p that generates all but the last spread.

Now, we have

$$\begin{aligned}\mathcal{F}(p,\sigma) &= \sum_{i=1}^{n}(\delta_i, +\alpha_i + \gamma_i)\\ &= \sum_{i=1}^{n'}(\delta_i, +\alpha_i + \gamma_i) + \sum_{i=n'+1}^{n}(\delta_i, +\alpha_i + \gamma_i)\\ &= \mathcal{F}(p',\sigma') + \sum_{i=n'+1}^{n}(\delta_i + \alpha_i + \gamma_i),\end{aligned} \tag{8}$$

where $\sigma' = \{\sigma_1, \dots, , \sigma_{n'}\}$. The sum $\sum_{i=n'+1}^{n}(\delta_i + \alpha_i + \gamma_i)$ can be thought of the extra costs added by the columns in the last spread. Now, the term δ_i in this sum is independent of the breakpoints in p' and the spread variations σ' of the first spreads (it depends on $\sigma_{n'+1} = \cdots = \sigma_n$ though). The paragraph variation term α_i always depends only on the situation in column i and the term γ_i for the columns of the last spread is independent of σ' as we have made the split after the last column of a spread.

Thus, if p is an optimal solution for paginating the document into n columns under a given spread variation σ, then p' must be an optimal solution for paginating the partial document from b_0 to $b_{n'}$, ie, into s fully filled spreads. If p' would not be optimal, we could replace it with an optimal pagination p'', ie, one with $\mathcal{F}(p'', \sigma') < \mathcal{F}(p', \sigma')$, which would contradict that $\mathcal{F}(p, \sigma)$ is minimal.

By the same argument, we can see that $\mathcal{F}(p', \sigma')$ is, in fact, an optimal solution regardless of the chosen spread variations, ie, replacing σ' by some other σ'' cannot improve the result. However, this argument only works if we choose the simpler definition for γ_i from Equation (6). If we use the definition from (7) instead, then the γ_i from the columns of the last spread have a dependency on $\sigma_{n'}$, which is part of $\mathcal{F}(p', \sigma')$. Thus, in that case, an algorithm would need to work harder and keep more of the potential partial solutions in memory.

On the other hand, if we keep σ fixed, then the above argument is true for any value of n' as then, the terms in the right-hand sum are always independent of $\mathcal{F}(p', \sigma')$.

To find the optimal solution, it is therefore sufficient to first find all breakpoints that can end the first column within the given constraints and all possible values for σ_1. Then, starting from those breakpoints, find all breakpoints that provide a solution to end the second column using $\sigma_1 = \sigma_2$. This continues until we reach the end of a spread, in which case, we only need to remember the best way to reach this point, ie, the one with $\mathcal{F}(b_0, \dots, b_{n'}, \sigma)$ for any σ because of Equation (8).

Then, the process reiterates, trying all possible values for $\sigma_{n'+1}$ as we are at the start of a new spread. This continues until we finally reach the end of the document with b_n as our last breakpoint. This breakpoint may have been reached several times using different values for σ_n, and the optimal solution for the pagination of the whole document is then simply the sequence of breakpoints through which we reached that last breakpoint with minimal $\mathcal{F}$; something that can be easily obtained by backtracking through the partial solutions remembered along the way.

As we will see below, this approach will result in an algorithm that has a quadratic runtime behavior in the number of breakpoints; in fact, if all columns have the same height, it will even run in linear time.

4 | AN ALGORITHM FOR GLOBALLY OPTIMIZED PAGINATION

In the following, we discuss a slightly simplified version of the algorithms used in the pagination phase (Phase 3) of the framework. As mentioned before, the base algorithm is a variation of the Knuth/Plass algorithm for line breaking suitably changed and adjusted for the pagination application. In particular, it uses a somewhat different object model to account for the pagination peculiarities and to support the extensions.

On a very high level of abstraction, one can build an object correspondence between the algorithms as follows: words in Knuth/Plass correspond to paragraphs in pagination; hyphenation points in words to lines that allow column breaks; and spaces between words to (stretchable) vertical spaces between paragraphs or other objects on the galley. However, while words or partial words have only a width that is used by the Knuth/Plass algorithm, objects for the pagination algorithm have both a height and a depth as we see later and both need to be separately accounted for.

While the double spread extension (Section 4.4) has no natural application in line breaking, the variation support extension (Section 4.5) could be incorporated back into a line-breaking algorithm: Individual variation paths would become alternate words or phrases and a global optimizing line-breaker would then pick and choose among them, to best satisfy other requirements, such as the desired number of lines, number of hyphenated words, tightness of white space, etc. This would, for example, support and simplify the approach outlined by Kido et al[26] on layout improvements through automated paraphrasing.

4.1 | Preliminary definitions

The input for the pagination algorithm is the galley object model generated in Phase 2. This is a sequence of objects $x_1, x_2, \dots, x_m$, where each x_i is either a “text” block t_i that will always be present in the final paginated document (eg, textual material) or a “breakpoint/space” block b_i at which the galley may get split during pagination.

Usually a text block represents a single line of text in the galley. However, if there is no legal breakpoint between 2 or more lines, then all such lines and any intermediate spaces are combined by the process in Phase 2 into a single text block. For example, if widows and orphans are disallowed, then a 3-line paragraph would have no legal breakpoint and, thus, would form a single block. Other examples are multiline equations or code fragments that are marked as unbreakable in the source.

In a similar fashion, consecutive vertical spaces in the source will be combined into a single breakpoint block as the galley can only be broken in front of the first of such spaces. If, however, a space in the source is followed by an (explicit) penalty, then this starts a new breakpoint block to represent the additional breakpoint. Thus, without loss of generality, we can assume that the block sequence alternates between single text blocks and one or more consecutive breakpoint blocks.

If a break happens at b_i, then that block gets discarded (in particular, it does not contribute to the height of the columns on either side of the break).[###] In addition, all directly following breakpoint blocks $b_{i+1}, b_{i+2}, \ldots$ will also be discarded. This reflects the fact that spaces between textual elements are expected to "vanish" at column/page breaks.

Each block x_i has an associated height $\mathcal{H}_{x_i}$, stretch $\mathcal{S}^+_{x_i}$, and shrink $\mathcal{S}^-_{x_i}$ component that describe the block's contribution to the galley and in case of breakpoint blocks, also an associated penalty $\mathcal{P}_{b_i}$, indicating the cost of breaking at this block.

Additionally, each text block t_i has an associated depth component $\mathcal{D}_{t_i}$ that holds the size of the descenders in the last line of the block. This value is not directly incorporated into the $\mathcal{H}_{t_i}$ as it should not participate in height calculations if t_i is the last block before a break, as explained earlier. For breakpoint blocks, $\mathcal{D}_{b_i}$ is always 0.

For any $i < j$, we define $\mathrm{col}_{i,j}$ to be the material between the 2 breakpoints b_i and b_j, ie, the sequence of all blocks $x_{\mathrm{after}(i)}, \ldots, x_{j-1}$, where $x_{\mathrm{after}(i)}$ is the first text block with an index greater than i (as all breakpoint blocks directly following a break are dropped). We call this a "column candidate" as it may be the material that gets placed into a column by the algorithm. The natural height of its content is

$$\mathcal{H}_{\mathrm{col}_{i,j}} = \sum_{k=\mathrm{after}(i)}^{j-2} \left(\mathcal{H}_{x_k} + \mathcal{D}_{x_k}\right) + \mathcal{H}_{x_{j-1}}, \tag{9}$$

its depth is $\mathcal{D}_{\mathrm{col}_{i,j}} = \mathcal{D}_{x_{j-1}}$, its stretch is $\mathcal{S}^+_{\mathrm{col}_{i,j}} = \sum_{k=\mathrm{after}(i)}^{j-1} \mathcal{S}^+_{x_k}$, and its shrink $\mathcal{S}^-_{\mathrm{col}_{i,j}}$ is defined in the same way.

If C_k is the target height for column k in the final document, then we denote by $Q^k_{i,j}$ the cost value (the inverse quality, ie, lower values mean higher quality) calculated for placing the material $\mathrm{col}_{i,j}$ into column k. Its definition is given by

$$Q^k_{i,j} = \begin{cases} \infty, & \text{if } \mathcal{H}_{\mathrm{col}_{i,j}} - \mathcal{S}^-_{\mathrm{col}_{i,j}} > C_k \\ f\left(C_k, \mathcal{H}_{\mathrm{col}_{i,j}}, \mathcal{S}^+_{\mathrm{col}_{i,j}}, \mathcal{S}^-_{\mathrm{col}_{i,j}}, \mathcal{P}_{b_j}\right), & \text{otherwise.} \end{cases} \tag{10}$$

If there is no way to squeeze the material into the available space (ie, when the column is overfull after applying all available shrink), we have $Q^k_{i,j} = \infty$. Otherwise, the function f is used to provide a measure for how well the content sequence fills the column, eg, how much space is left unused. For its precise definition, many possibilities are available, provided the function has no dependencies on breakpoint choices made earlier, or if it does, only needs to look back through a fixed number of earlier breakpoints to ensure applicability for dynamic programming. By default, the framework currently uses the "badness" function that is also used by TEX's greedy algorithm for page breaking, ie, the badness function discussed in Section 3.2, ie,

$$Q^k_{i,j} = \begin{cases} \infty, & \text{if } \mathcal{H}_{\mathrm{col}_{i,j}} - \mathcal{S}^-_{\mathrm{col}_{i,j}} > C_k \\ \delta_k, & \text{otherwise.} \end{cases} \tag{11}$$

However, this could be altered and made more flexible.

[###] In the remainder of this paper, we therefore usually talk about "the breakpoint b" rather than "the breakpoint at breakpoint block b" if there is no confusion possible.

If $badness_{\mathrm{col}\,k}(b_i, b_j) \leqslant c_{\mathrm{tolerance}}$ for some customizable parameter $c_{\mathrm{tolerance}}$, we call $\mathrm{col}_{i,j}$ a feasible solution for column k in the final document (or, if all columns have the same target height, a feasible solution for all columns), otherwise, an infeasible one that we ignore.‖‖‖

The goal of the algorithm can now be formulated as the quest to find the best sequence of breakpoints $b_0, b_2, \ldots, b_n$ through the document such that all $Q^k_{b_{k-1},b_k}$ are feasible and

$$D_{b_0,\ldots,b_n} = \sum_{\ell=1}^{n} Q^{\ell}_{b_{\ell-1},b_\ell} \tag{12}$$

is minimized (with b_0 and b_n representing start and finish of the document, respectively). In the TEX world, D is usually called the *demerits* of the solution.

To solve this, it is not necessary to calculate $Q^k_{i,j}$ for all possible combinations of k, i, and j because $Q^k_{i,j} = \infty$ implies $Q^k_{i,j+1} = \infty$. Furthermore, if $b_0, \ldots, b_k$ and $b_0, b'_1, \ldots, b'_{k-1}, b_k$ are 2 breakpoint sequences ending at the same place, the algorithm only needs to remember the best of the 2 partial solutions, because extending the sequences to b_{k+1} means adding $Q^{k+1}_{b_k,b_{k+1}}$; thus, the relationship between the extended sequences will stay the same.

The algorithm therefore loops through the sequence of all x_i, thereby building up all partial breakpoint sequences $b_0, \ldots, b_k$ that are possible candidates for the best sequences, ie, applying the pruning possibilities outlined above. For this, we maintain a list active nodes $A = a_1, a_2, \ldots$ where each a_i is a data structure that represents the last breakpoint b_k in some candidate sequence plus some additional data.**** This list is initialized with a single active node representing the document start.

While looping through x_i, we maintain information about total height, stretch, and shrink from the start of the document up to x_i. In the data structure for an active node a, we record the column number k that ended in this node and the total height, stretch, and shrink from the start of the document to $b_{\mathrm{after}(a)}$ so that calculating, for example, $\mathcal{H}_{\mathrm{col}_{a,b_j}}$ becomes a simple matter of subtracting the total height recorded in a from the total height at b_j.

D_a is defined to be the smallest demerits value that leads up to a break at a, ie, $D_a = D_{b_0,\ldots,b_k}$ for some sequence of $k+1$ breakpoints with $break(a) = b_k$. Recording this value in the active node data structure for a makes it easy to prune those active nodes that cannot become part of the final solution and to arrive at Equation (12) eventually, as we have $D_{a'} = D_a + Q^{k+1}_{b_k,b_{k+1}}$ for a newly created active node a' at breakpoint b_{k+1}.

Whenever we encounter new possible candidate sequences, we compare them and add corresponding active nodes for the best of them. In addition, when an active node a is so far away from the current block x_i such that $Q^{k+1}_{a,x_i} = \infty$, we remove the active node from the list as the partial sequence represented by a can no longer be extended to become the best solution.

4.2 | Details of the base algorithm

The overall algorithm is detailed out in Figure 7 and works as follows: We start by initializing the active list with a single node representing the start of the document. For $i = 1, \ldots, m$, ie, all

‖‖‖There are cases where it is necessary to consider infeasible solutions as well, but these are boundary cases that we ignore for the discussion here.

****Again, it is convenient later on to talk about "the breakpoint a" instead of "the breakpoint b that is associated with the active node a" if there is no possible confusion.

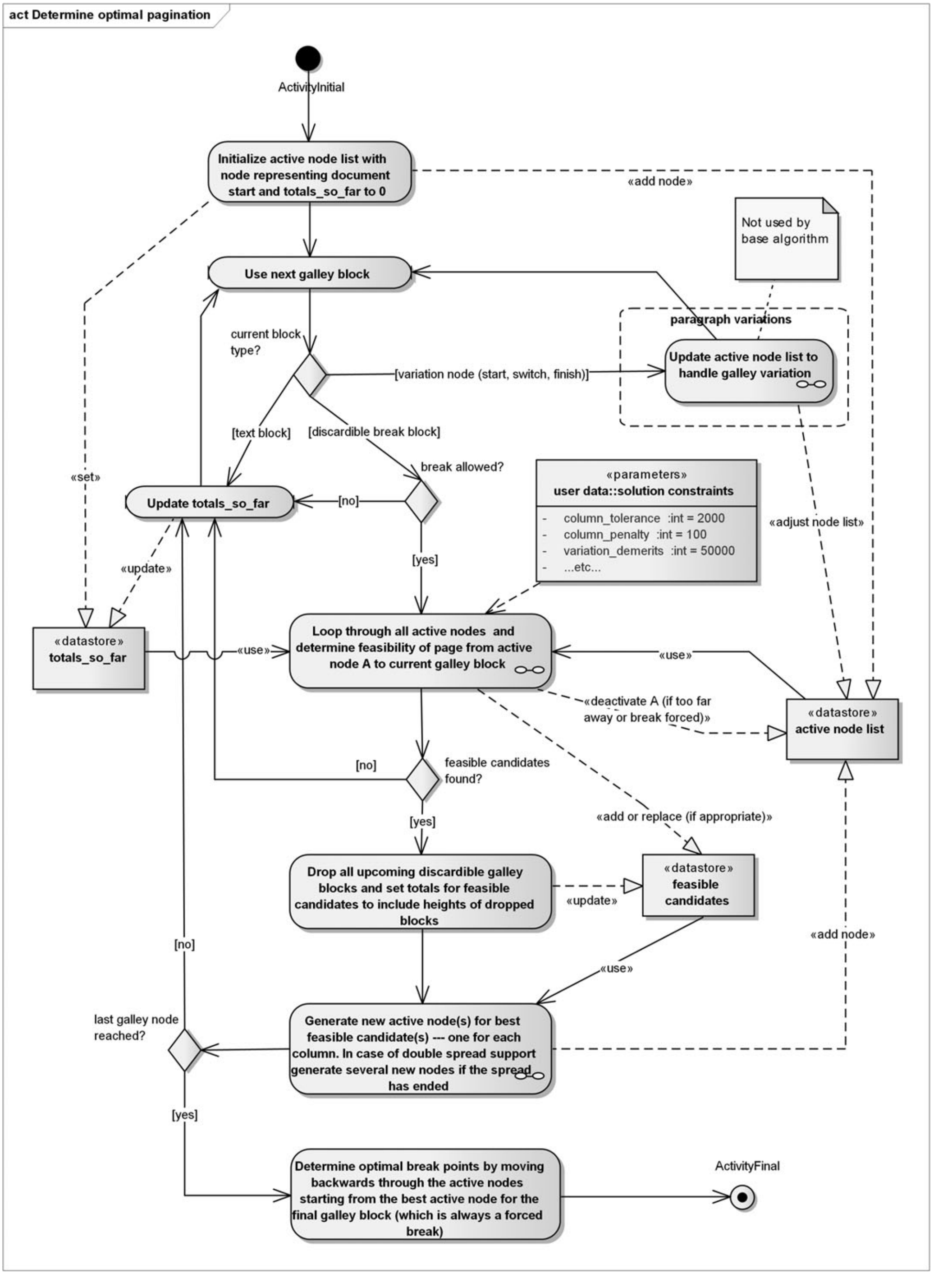

FIGURE 7 The main algorithm (Phase 3)

Activities when encountering a break block are further detailed in Figure 8 on the facing page.
Generating new active nodes with or without double spread support is outlined in Figure 9 on page 32.
Finally, the handling of paragraph variations is further detailed in Figure 10 on page 34.

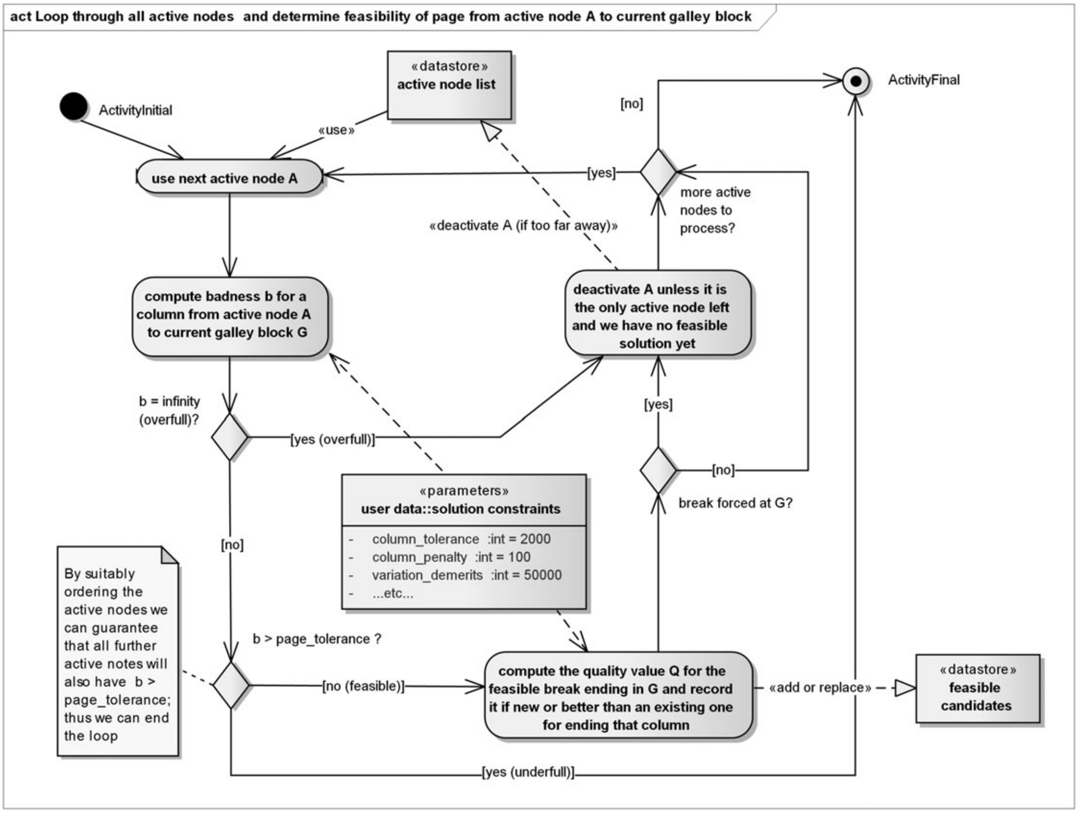

FIGURE 8 Handle a break block (Phase 3)

blocks in the galley model, we then have to distinguish the following cases.

Case $x_i = t_i$: We update the totals seen so far by adding $\mathcal{H}_{t_i}$, $\mathcal{S}^{+}_{t_i}$, and $\mathcal{S}^{-}_{t_i}$, respectively. The depth $\mathcal{D}_{t_i}$ is not yet added at this point.

Case $x_i = b_i$: A possible breakpoint; the detailed workflow for that case is shown in Figure 8.

We loop through all active nodes $a \in A$ and evaluate

$$\beta = \underset{column(a)+1}{badness}\,(a, b_i)$$

using the badness function from (2) to see how well it works to form the column $column(a)+1$ with the material between a and b_i, ie, with col_{a,b_i}.

If $\beta \leqslant c_{\text{tolerance}}$, we remember col_{a,b_i} as one feasible way to end column $column(a) + 1$ at breakpoint b_i.

Otherwise, if $c_{\text{tolerance}} < \beta < \infty$, we consider col_{a,b_i} an infeasible way to end the column that we normally ignore.[††††] By suitably ordering the active nodes, we can ensure that all further active nodes will also have $c_{\text{tolerance}} < \beta$.

[††††]Exception: If the current break is forced and we have not seen a feasible solution so far, we need to keep the best of the infeasible ones, as otherwise, the active list would be empty afterward.

This is achieved by grouping all active nodes of the same column class[‡‡‡‡] together and within each group ordering the active nodes by their distance from the start of the document (earlier ones first). Given a badness function as the one defined by (2), this means that if the material that is put into the column is already been stretched out, the badness will get worse if we put less material in.

Thus, as long as $\mathcal{P}_{b_i}$ is not a forced break and we are already stretching we do not need to consider any further active node in the current column class as it will have a worse badness. This will speed up the algorithm considerably, especially if there is only a single-column class.

Otherwise, if $\beta = \infty$, we remove the active node a as it is too far away from b_i and cannot form a feasible solution with this or any later breakpoint.

In either case, if $\mathcal{P}_{b_i}$ is a forced break, we also remove the active node a as it cannot form a column with a later breakpoint. Because of this necessary housekeeping, we cannot end the loop prematurely in case of forced breaks.

Then, we move to the next active node unless the loop ended prematurely above, in which case we move to the first active node in the next column class if any.

Once all active nodes are processed, we determine $b_{\text{after}(i)}$ so that the total height, stretch, and shrink from the beginning of the document to this breakpoint can be calculated for any newly created active nodes associated with b_i.

We then look at all the newly collected candidate solutions ending in b_i, and for each different column k, we select the best candidate (having the smallest value of $\sum_{\ell=1}^{k} Q$) and record a new active node for it. Infeasible candidate solutions with $c_{\text{tolerance}} < \beta \leqslant \infty$ will normally be thrown away at this point unless they are the only way to proceed, ie, if without one of them the active list would end up being empty.

Finally, we update the totals seen so far by adding the new height and the previous depth $(\mathcal{H}_{b_i} + \mathcal{D}_{x_{i-1}})$, and the stretch and shrink $\mathcal{S}^{+}_{b_i}$ and $\mathcal{S}^{-}_{b_i}$. This has to happen after generating new active nodes as the material is not part of the current column if a break is taken at b_i.

Finally, after having processed x_m, which is the last node in the document and supposed to be a forcing breakpoint, we have only active nodes left that correspond to x_m (but possibly to different columns/pages). Out of those we select, the one with the smallest D_a as the best solution. From this active node, we can move backward through the active nodes that lead to it, to obtain the complete breakpoint sequence of the optimal solution.[§§§§]

4.3 | Complexity and search space

In the algorithm as described, the cost function Q that weights the different constraints against each other is used to obtain the final solution (by minimizing $\sum Q$) but to limit the search space only the status with respect to the column badness is evaluated. This means that solutions with a column badness higher than $c_{\text{tolerance}}$ are disregarded even if their Q-value may be lower than others that are being considered.

One can informally describe this behavior as follows: Different constraints can be weighted against each other but only as long as one constraint is not violated too badly.

[‡‡‡‡] Abstractly speaking, a column class is defined by the sequence of heights for all future columns that still need to be built. Thus, if all column heights are equal there is only one column class. Otherwise, if 2 active nodes end different columns, their column class will usually be different. In case of double spread support, the column class may even differ if the active nodes end the same column (since the following column may be run short or long, ie, differ in height).

[§§§§] From an implementation point of view this means that we cannot throw active nodes away when they get removed from the active list as their info may still be necessary in this step.

A different approach would be to limit the search space by requiring that Q is lower than a certain constant, but from a user interface perspective, it is much harder to understand the significance of the cutoff point if it is a combination of several constraints is used to generate its value.

Technically, although the 2 ways are identical, as it is always possible to provide a definition for f that results in the behavior as implemented; thus, it is more a matter of user interface style than anything else.

If the column heights vary throughout the document, then the complexity of the base algorithm is of order $O(m^2)$, where m is the total number of blocks x_i in the document. If the algorithm would calculate all $\mathrm{col}_{i,j}$ then this would be $m(m-1)/2$ computations; thus, this gives us an upper bound. However, since many of them will be naturally infeasible, the number of calculations actually needed to be carried out can be reduced a lot, making the problem computable even for larger values of m.

The main loop has to be executed for each block and for $x_i = b_i$, which can be assumed to happen about half of the time and one needs to calculate col_{a,b_i} for all $a \in A$ at this point. Now, the number of active nodes a with $column(a) = k$ in that list is bounded by the first line in Equation (10) as active nodes get deactivated, once they are too far away from the current breakpoint, ie, more than C_k plus any available shrink in the material. Thus, assuming the column target height C_k is bounded (which it had better be in a real life scenario), as well as the ability for material to shrink, then the maximum number of active nodes a with $column(a) = k$ will be smaller than $c \cdot n$ with c as a small constant and n the number of breakpoints possible in material of height $\max_k(C_k)$.

However, due to the variation introduced by the ability of material to stretch and shrink, the active list will not contain just nodes related to a single column, but over time will grow and contain nodes related to different columns. If we assume, for example, that there is ±5% flexibility generally, then after looking at breakpoints for roughly 20 columns worth of material, we may find active nodes ending at column 19 (material was always stretched) or 21 (material was always compressed) beside those for column 20 which would be the natural length. Thus, with m growing the length of the active node list A will grow proportionally to it and although that factor of growth would be very small, it will give us a complexity bound of $O(m^2)$.

However, there is a very common subclass of layouts, in which the situation is much better: If the target column heights C_k are equal for all columns or all columns after a certain index and if the cost function Q only depends on general characteristics such as a common column size,[¶¶¶¶] Then, it is possible to collapse different feasible solutions for a given breakpoint to one even if they are for different columns. This will reduce the search space that the algorithm has to walk through considerably, and the complexity will be reduced to $O(m)$ as now the maximum length of the active list is bounded by a constant.

4.4 | Double spread support

Providing support for shortening or lengthening the columns of a double spread means that if the active node a represents a column break for the last column k on a double spread, then the calculation of $Q^{k+1}_{a,b}$ in the main loop needs to be done 3 times with different values for the height

¶¶¶¶ This is the case for Q in the base algorithm as it only takes C_k as column-related input. In the double spread extension, the variant height V_a and, implicitly, the column type T_a (which is a function of k) are additional inputs to Q and so individual active nodes need to be maintained for any combination of their values. Here, collapsing could happen for all columns that share the same type and the same height variation value.

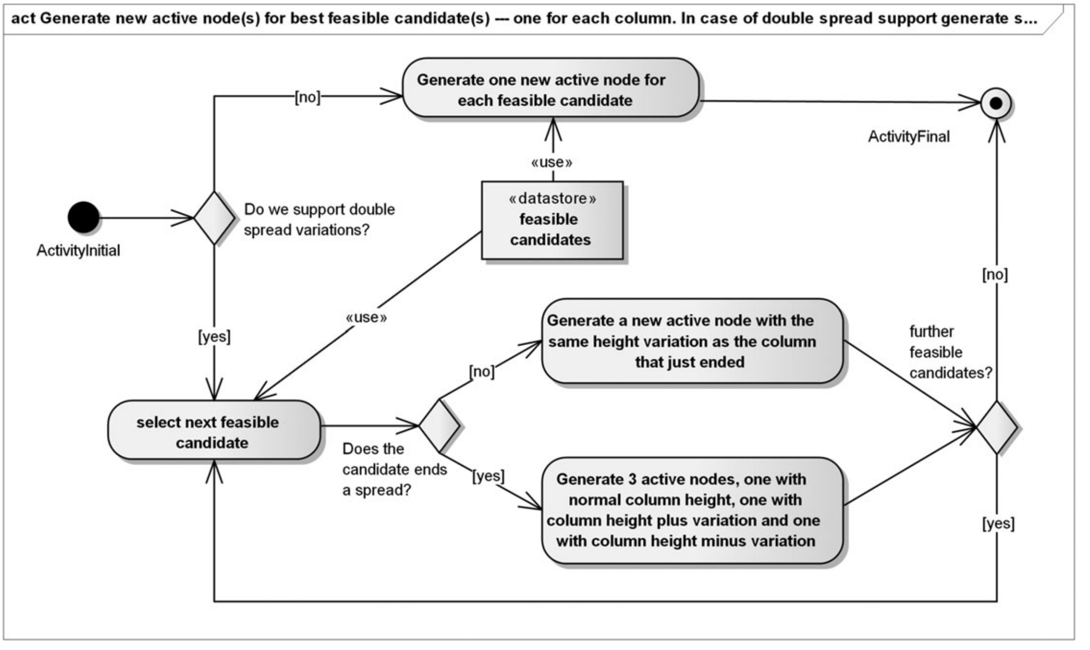

FIGURE 9 Double spread support (Phase 3)

of column $k+1$, namely, C_{k+1} and $C_{k+1} \pm \mathit{variation}$ and we can only deactivate a once Q is ∞ for all different column heights. Furthermore, for column $k+1$, we now need to generate a new active node for each combination of $k+1, C_{k+1} + \mathit{variation}$ for which there exists a feasible candidate.

On the other hand, if such a new active node a' has been created for column $k+1$, then whatever height has been used for column $k+1$ needs to be reused when evaluating $Q^{k+2}_{a',b'}$ for some breakpoint b'. In addition, the same happens for all further columns of that spread. Thus, the target height as input to f is no longer just depending on the current column but also on the situation on previous column(s). It can be varied if we are starting a new spread or it needs to be whatever the previous column was if we are on any other column of the spread.

To support this efficiently, we extend the active node data structure to keep track of the type of column T_a that will start at a (ie, a function of k) and the amount of height adjustment V_a that should be used on that column.[####] Then, at the point in the algorithm where we are generating new active nodes from feasible candidates (see Figure 9), we check the type of column that has started by the current active node a and ended at the current breakpoint as follows.

- If $T_a = \texttt{last}$, then the next column has flexibility and can be run a line long or short. We model this by generating at this point not 1 but 3 new active nodes that are identical except for the variation amount V_a to be used on the next column: This is set to 0, or $\pm\mathit{baselineskip}$, respectively.
- If, on the other hand, $T_a \neq \texttt{last}$, then the variation amount is predetermined by the value specified in the active node a that was used in the feasible candidate. For each group of feasible candidates (with the same value of k and V_a), we therefore generate a single new active node a' and set $V_{a'} = V_a$.

[####] Think of T_a as recording “column x out of y” so that each column is identifiable, and we can test if we are in the last column of a spread.

It is important to reiterate that the above means that partitioning of the feasible candidates in groups is not just based on the values for column k but on k and V_a (the latter only if $T_a \neq \texttt{last}$) and that for each such group one needs to generate a new active node (or a set of active nodes).

In case C_i is constant, partitioning needs to happen only on T_a and V_a, which reduces the complexity but still means that, as compared with the base algorithm, a noticeable number of extra active nodes need to be generated and processed.

The only other modification that is still needed, is to extend the definition of the cost function Q from Equation (10), as it now needs to incorporate the value of V_a

$$Q_{i,j}^{k,V_a} = \begin{cases} \infty, & \text{if} \quad \mathcal{H}_{\text{col}_{i,j}} - \mathcal{S}^{-}_{\text{col}_{i,j}} > C_k + V_a \\ f\left(C_k, \mathcal{H}_{\text{col}_{i,j}}, \mathcal{S}^{+}_{\text{col}_{i,j}}, \mathcal{S}^{-}_{\text{col}_{i,j}}, \mathcal{P}_{b_j}, V_a\right), & \text{otherwise.} \end{cases} \tag{13}$$

The check whether or not the material fits the column is adjusted to include the height variation and the function f is extended to accept V_a as input so that deviations from the norm ($V_a \neq 0$) can be appropriately penalized by adding to the value returned by Q. By default, the framework uses the following definition for f:

$$f\left(C_k, \mathcal{H}_{\text{col}_{i,j}}, \mathcal{S}^{+}_{\text{col}_{i,j}}, \mathcal{S}^{-}_{\text{col}_{i,j}}, \mathcal{P}_{b_j}, V_a\right) = \begin{cases} \delta_{k+V_a} + c_{\text{spread variation}}, & \text{for} \quad V_a \neq 0 \\ \delta_k, & \text{otherwise.} \end{cases}$$

This way, if a column is run long or short, the constant $c_{\text{spread variation}}$ is added to the demerits; thus, by changing the value for this constant, one can make it more or less likely that the height of double spreads get changed by the algorithm.

4.5 | Variation support

Support for variants in galley material (eg, paragraphs with different line breaks resulting in different number of lines, or in a different distribution hyphenation points) is handled by introducing new types of control elements c_i in the input stream that signal "start," "switch," and "end" of a variation set. Start and switch controls have an associated penalty $\mathcal{P}_{c_i}$ that is used to penalize the choice of that particular variation.

One difficulty introduced by variations is that they provide different amounts of material along their variation paths. Thus, the distance from the start of the document to any breakpoint b after the variation block is no longer a single well-defined value. Instead, it depends on the route through which b has been reached. By supporting multipath variations as well as variations within variations, this can get arbitrarily complicated. In the algorithm, this is resolved by manipulating the data stored in active nodes essentially by pretending that the document has started on an earlier or later point. This way, it becomes transparent for the calculation of $\text{col}_{a,b}$ through which variation paths b has been reached.

The paths from all variation sets are uniquely labeled so that every possible way to move from the start to the end of the document can be uniquely described by simply concatenating the path labels.‖‖‖‖

‖‖‖‖The precise method is of no importance, as long as it is possible to exactly reconstruct the selection made to achieve the best solution. In the prototype implementation, sequential numbers for both the variation sets and the individual paths within them have been used. For example, `1-2;2-2;3-1` means that the second path was taken in the first and second variation set, whereas the first path was taken in the third variation set.

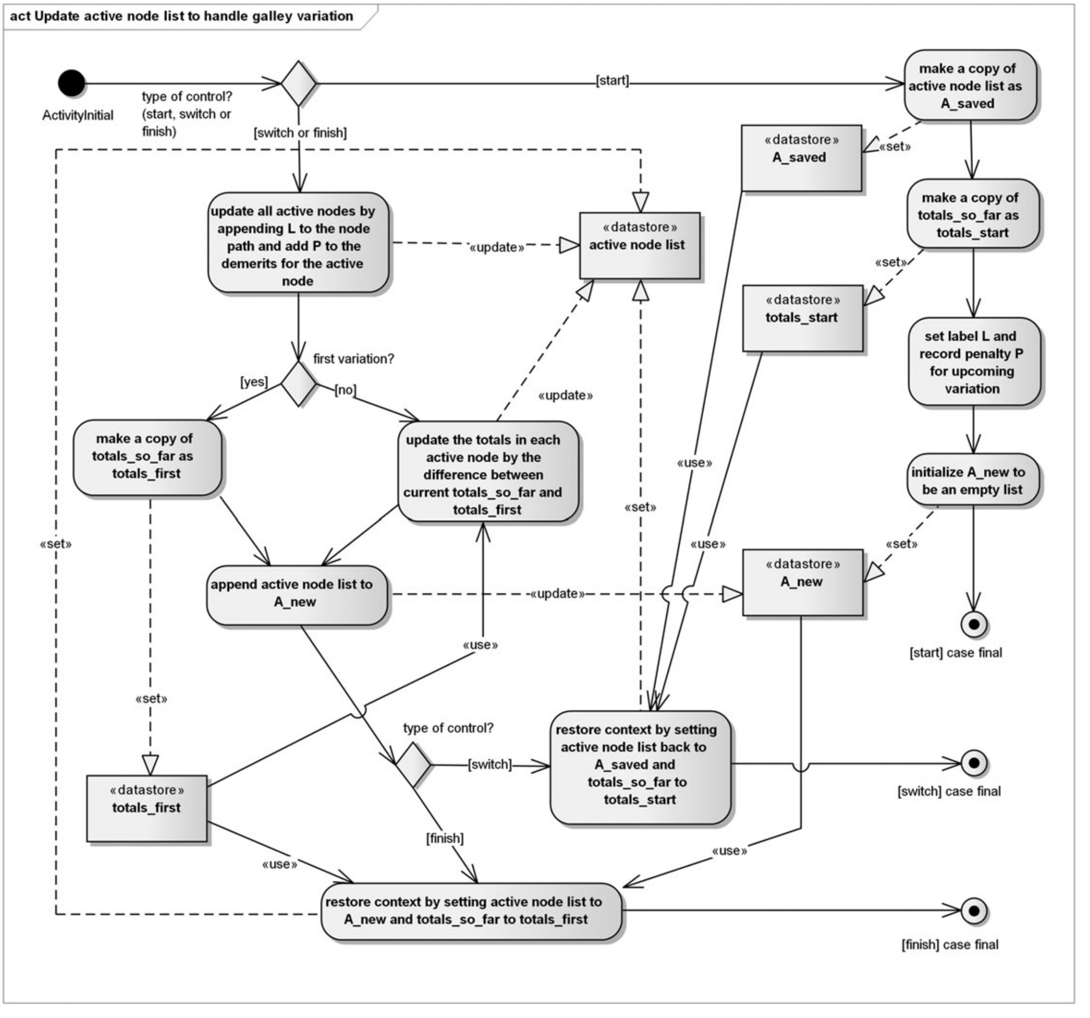

FIGURE 10 Handle paragraph variation data (Phase 3)

The active node data structure is extended to record in *path*(a), the cumulated path through all variations up to the break point associated with a.

For variation support, the main loop of the base algorithm is then extended as outlined in Figure 10 by managing the following additional cases.

Case $x_i = c_i$ **with** $type(c_i) = \texttt{start}$ This signals the start of a variation set. We make a copy of the active node list $A_{\text{saved}} \leftarrow A$, and we also file away the totals $\bar{H}_{\text{start}}$, $\bar{S}^{+}_{\text{start}}$, and $\bar{S}^{-}_{\text{start}}$ from the beginning of the document to the current position for later use. A label L for the current variation path is chosen and $P = \mathcal{P}_{c_i}$ is saved as the penalty to add to the demerits in case this path is chosen.[*****] Then, we proceed with the next block x_{i+1}.

[*****]The penalty $\mathcal{P}_{c_i}$ has been calculated in Phase 2 from the difference in quality between the optimal paragraph breaking and the paragraph breaking with a nonzero looseness value (TEX returns a numerical value for the line-breaking quality). This difference is then multiplied with the user-constraint $c_{\text{paravariation}}$ to obtain that penalty.

Case $x_i = c_i$ **with** *type*(c_i) = `switch` In this case, we reached the end of a variation path. All nodes currently in the active node list A are either on the current variation path (because they have been only recently created) or they are from before the variation block, but we have evaluated the breakpoints on the variation path against them.

Thus, we update all $a \in A$ by appending the label for the finished variation to the path in each a, ie, $path(a) \leftarrow path(a); L$ and add P to $demerits(a)$.

If this is the first variation in the variation block, we save the totals from the beginning of the document to the current point in $\bar{H}_{\text{first}}$, $\bar{S}^{+}_{\text{first}}$, and $\bar{S}^{-}_{\text{first}}$ for later use.

Otherwise, we also update the totals stored in all $a \in A$ with the height difference between the current and the first variation path, ie, $\bar{H}_{\text{first}} - \bar{H}_{c_i}$, etc. This way, later on $\mathcal{H}_{\text{col}_{a,b}}$ for some breakpoint b after the variation block can still be simply calculated by subtracting the totals at b from the totals at a, ie, the calculation is transparent to the path by which b was reached.

Finally, we save away the updated active node list A. We then restore the context we were in before the first variation, ie, $A \leftarrow A_{\text{saved}}$ and we restore $\bar{H}_{\text{start}}$, $\bar{S}^{+}_{\text{start}}$, and $\bar{S}^{-}_{\text{start}}$ as the current totals. We then select a new label L and set $P \leftarrow P_{c_i}$ for the next variation path. Then, we proceed to x_{i+1}.

Case $x_i = c_i$ **with** *type*(c_i) = `finish` The end of the variation block has been reached. We update the active node list as described in the switch case and then combine it with the active node lists saved earlier. This will then form the complete new active node list going forward. All that remains to do otherwise, is to restore the totals to the values at the end of the first variation (as the active nodes in all other variations have been adjusted to pretend this is correct). These values have been previously recorded as $\bar{H}_{\text{first}}$, $\bar{S}^{+}_{\text{first}}$, and $\bar{S}^{-}_{\text{first}}$.

Starting from the final active node a when finishing the algorithm, we arrive at the optimal solution for the whole document by determining the list of active breaks that leads to this node and examining all selected variation paths as recorded in $path(a)$. The latter is an integral part of the solution as many variation blocks will end up between 2 chosen breakpoints, yet it is important to know which path was used in the construction since we have to replicate that decision in the typesetting phase (Phase 4).

4.6 | Complexity of the extensions

It is easy to see that both extensions do not change the overall complexity of the algorithm, ie, it stays $O(m^2)$ in the general case and $O(m)$ if the column height is constant after a certain point.

In the double spread case, the maximum length of the active list will have an additional factor of 3 × *number of spread columns* due to the variability when starting a new spread and the fact that we need to distinguish active nodes for different columns on a spread.

The situation with variation blocks is worse, as the number of active nodes depends on the number of paths through the variation sets seen along the way. The number of different paths through variations $v_1, \dots, v_\ell$ is $\prod_{i=1}^{\ell} w_i$ with w_i being the number of "ways" through variation set v_i. Thus, this is exponentially growing in ℓ, but, fortunately, ℓ is bounded by the number of breakpoints that can fit on a single column. Therefore, this product is actually also of complexity $O(1)$, although unfortunately, with a much larger constant if we have columns with many variable paragraphs; see Section 4.7.

In case of constant column heights C_i, the overall complexity is $O(m)$ for the base algorithm, because it is possible to collapse all active nodes associated with breakpoint b into one, regardless of the column they did end. This limits the maximum length of the active node list so that it becomes a constant in the complexity calculation. This argument also holds true for the variation set extension (with the small practical problem that the actual constant is fairly large).

In case of the double spread extension, we have dependencies between different columns; therefore, a solution for column one is not necessarily a solution for all other columns. Nevertheless, collapsing is also possible with the only difference that we have to keep the best feasible candidate for each combination of T_a and V_a in the running. Again, this makes the length of the active node list independent from m so that the overall complexity of the algorithm drops to $O(m)$.

4.7 | Computational experience

To gain experience with the behavior of the algorithm and its extensions, it was tested on different types of novels. A few of these documents and the respective findings are listed in Table 1. They differ in size, average paragraph length, frequency of headings, complexity, and other aspects and are a good representation of the complete set of documents tested. These documents are “Alice's Adventures in Wonderland” by Lewis Carroll, “Call of the Wild” by Jack London, “Fairy Tales” by the brothers Grimm translated into English by Edgar Taylor and Marian Edwardes, “Pride and Prejudice” by Jane Austen, and “The Old Curiosity Shop” by Charles Dickens.

All documents have been set in 2 columns with a width of 8 cm. Each column could hold 46 lines of text and the paragraph requirements have been fairly strict: No widows or orphans and only a small amount of flexibility (+1pt) for the paragraph separation. This means that in each column, one could gain a flexibility of up to 2 lines (but only when there are 8 or more paragraphs in the column and we accept a stretch of up to 3 times the nominal value, which corresponds to a badness of 2700).

This type of paragraph flexibility (indicated by “flex” in the table) is rather uncommon when typesetting novels, as there one usually tries to keep all text lines on a grid. However, it is often used with technical documentation that contains objects of sizes that differ from a normal text line height. It is therefore the default used by LaTeX.

For comparison, all trials have also been run without allowing for any flexible space between paragraphs (this is indicated by “strict” in the table). Obviously, this means that the pagination algorithm has (even) fewer options to choose from. Thus, with LaTeX’s greedy algorithm, we see more “ugly” columns and with the optimizing algorithm, we see additional cases where the algorithm is unable to find any solution at all.

The first row for each document in the table gives the results when processing the document using the standard LaTeX pagination, ie, a greedy algorithm and in all cases, there are a large number of bad column breaks that would require manual attention (between 4% (Carroll) and 16% (London) of all columns).

The second row for each document then shows the results from LaTeX’s greedy algorithm but without allowing any flexibility between paragraphs. In this case, the number of issues rises up to 40% of all columns (Carroll) and between 15% and 25% for all others.

The base algorithm (ie, optimizing across the whole document but without adding any additional flexibility through an extension) shows a maximum active list length of 37, 9(!), 22, 42, and 41 (flex) and 6, 2, 20, 2, and 2 (strict), respectively. As a column with 46 lines would have at most that many breakpoints, these values are in line with expectations, ie, the inherent galley flexibility contributes only a very small factor; thus, only with Austen and Dickens,

TABLE 1 Document performance using different algorithm extensions

	Document		Active List		Paragraphs		Available Looseness[a]					Vertical Badness[b]			Run
	Columns	Blocks	Max	Average	Total	Variable	−1/0	−1/1	−1/2	0/1	0/2	Good	Bad	Ugly	time, s
Alice in Wonderland	72				833							69	0	**2+1**	< 1
greedy, strict	72											40	4	**1+27**	
base, flex	–	6943	37	12								*no solution*[c]			
base, strict	–	6943	6	1								*no solution*[c]			
+ spread, flex	–	6943	432	122								*no solution*[c]			
+ spread, strict	–	6943	10	2								*no solution*[c]			
+ variations, flex	74[d]	9473	602	55		111	6	15	0	89	1	73	1	–	≈ 6
+ variations, strict	–	9473	261	19								*no solution*[c]			
+ variations, spread, flex	72	9473	7076	496								71	1	–	≈ 10
+ variations, spread, strict	70[d]	9473	1566	169								70	–	–	≈ 5
Call of the Wild	78				340							64	1	**9+4**	< 1
greedy, strict	78											62	0	**0+16**	
base, flex	–	9148	9	2								*no solution*[c]			
base, strict	–	9148	2	1								*no solution*[c]			
+ spread, flex	78	9148	263	134								78	–	–	≈ 4
+ spread, strict	79[d]	9148	41	14								79	–	–	≈ 4
+ variations, flex	78	14 970	263	68		139	11	3	0	124	1	78	–	–	≈ 6
+ variations, strict	78	14 970	263	63								78	–	–	≈ 5
+ variations, spread, flex	78	14 970	3156	704								78	–	–	≈ 12
+ variations, spread, strict	78	14 970	3156	637								78	–	–	≈ 11
Grimm's Fairy Tales	236				1041							212	6	**6+12**	< 2
greedy, strict	236											198	1	**0+37**	
base, flex	–	27 907	22	4								*no solution*[c]			
base, strict	–	27 907	20	1								*no solution*[c]			
+ spread, flex	234[d]	27 907	485	319								234	–	–	≈ 14
+ spread, strict	–	27 907	146	12								*no solution*[c]			
+ variations, flex	239[d]	59 110	437	92		441	10	50	21	318	42	239	–	–	≈ 15
+ variations, strict	236	59 110	422	86								236	–	–	≈ 14
+ variations, spread, flex	236	59 110	5532	1030								236	–	–	≈ 67
+ variations, spread, strict	237[d]	59 110	4980	968								237	–	–	≈ 55

(Continues)

TABLE 1 (Continued)

	Document		Active List		Paragraphs		Available Looseness[a]					Vertical Badness[b]			Run
	Columns	Blocks	Max	Average	Total	Variable	−1/0	−1/1	−1/2	0/1	0/2	Good	Bad	Ugly	time, s
Pride and Prejudice	315				2127							291	8	7+9	< 2
greedy, strict	315											229	14	0+73	
base, flex	316[d]	34 645	42	14								316	–	–	≈ 10
base, strict	–	34 645	2	1									*no solution*[c]		
+ spread, flex	312[d]	34 645	486	347								312	–	–	≈ 17
+ spread, strict	–	34 645	25	4									*no solution*[c]		
+ variations, flex	315[d]	56 861	633	73		483	10	51	6	397	19	315	–	–	≈ 16
+ variations, strict	314[d]	56 861	633	59								314	–	–	≈ 14
+ variations, spread, flex	314[d]	56 861	7596	837								314	–	–	≈ 58
+ variations, spread, strict	314[d]	56 861	7596	766								314	–	–	≈ 45
The Old Curiosity Shop	554				4097							524	12	9+9	< 3
greedy, strict	554											421	12	0+121	
base, flex	555[d]	60 480	41	24								555	–	–	≈ 17
base, strict	–	60 480	2	1									*no solution*[c]		
+ spread, flex	556[d]	60 480	487	359								556	–	–	≈ 32
+ spread, strict	–	60 480	22	3									*no solution*[c]		
+ variations, flex	558[d]	91 547	1084	74		768	65	33	2	653	15	558	–	–	≈ 25
+ variations, strict	556[d]	91 547	676	63								556	–	–	≈ 22
+ variations, spread, flex	560[d]	91 547	10 932	851								560	–	–	≈ 116
+ variations, spread, strict	555[d]	91 547	8832	799								555	–	–	≈ 100

[a] A count of paragraphs that can be affected by setting specific looseness values. For example, the column of −1/2 counts all paragraphs that could be shortened by 1 line and extended by up to 2 lines.

[b] Badness of columns: “Good” means the column material is stretched within the specified limits ($b < 2000$); “bad” means a noticeable stretch ($2000 \leq b < 4000$) and “ugly” means that the space in the column is stretched more than 3.4 times its available flexibility ($4000 \leq b$) or is infinitely bad in TEX’s eyes ($b = 10\,000$) indicated by the second value.

[c] The pagination algorithm ran out of options (active list empty) and produced one or more overfull columns as an emergency fix.

[d] The optimized solution has a different number of columns compared to the default LATEX solution.

we see a maximum close to 46. This is due to the fact that these documents are fairly long (several hundred columns) and have relatively short paragraphs so that the paragraph flexibility accumulates and some breakpoints end up being candidates for ending different columns. At the same time, we see that the maximum drops sharply if the paragraph flexibility is removed.

The very low flex value for London is due to the fact that this document has very long paragraphs (average of 4 per column) and thus is unable to build up any significant flexibility that makes the active list grow toward its boundary.

It is therefore also not surprising that the base algorithm does not find a solution (except with Austen and Dickens when using flex), as the number of alternatives to consider are not high enough to resolve all obstacles resulting from widows and orphans.

When applying the spread extension, the length of the active list gets bounded by $46 \times 3 \times 4 = 552$; thus, again, the observed maxima of 432, 263, 485, 486, and 487 (flex) are in line with expectations. Without flex, the maxima are also higher than before but nowhere close to the highest possible value. It may appear surprising that this additional flexibility does not result in a solution for Carroll, but this is due to the fact that this document contains an unbreakable object of nearly the height of a column so that it requires a much higher amount of flexibility to move this out of a break position. Figure 1 on page 6 shows these problematic pages.

As discussed in Section 4.6, the factor by which the active node list can increase in case of paragraph variations is basically the product $\prod_{i=1}^{\ell} w_i$, where the w_i is the number of different ways one can get through the variation set v_i, and ℓ is the number of variation sets in the current column. The majority of the variation sets in the texts by Carroll and London have $w = 2$ and only a few 3. Grimm and Austen on the other hand have 113 and 76 variation sets with $w = 3$ or 4, respectively. However, Carroll's paragraphs are much shorter on average, thus more fit on a page and larger values for ℓ are likely. Thus, seeing a factor of 16, 30, 20, 16, and 26, respectively, for the 5 documents again fits with expectations.

With the double spread extension, we vary the column height by 1 line and given 46 lines per column introduce an additional flexibility of roughly ±2.2%. The important aspect is that in contrast to variation sets this flexibility will be available on all columns and thus the change in the active node list length should be fairly uniform across all documents. In contrast, the paragraph variation extension will only make a noticeable difference in that length when several variable paragraphs are close together. Again, we can observe this difference: with the spread extension, the average and the maximum are fairly close to each other, whereas the average length when applying paragraph variations is noticeably smaller.

When running the algorithm in its current prototype implementation with both extensions applied, we can see a time increase of a factor of 15 to 100 compared to a run using standard LaTeX. While this sounds large, we have to realize that this means less than a second per page for a globally optimized document. When the author started to work with TeX, processing time for a single page was often 30 seconds and more. Thus, global optimization, even with additional bells and whistles added, has become a workable option.

5 | CONCLUSION AND FURTHER WORK

The main contribution of this paper is the definition and implementation of a general framework for experimenting with globally optimized pagination algorithms. This framework will enable researchers to quickly test out new strategies for pagination and make them available to a larger audience with ease.†††††

All constraints are parameterized so that experimenting with different value combinations is a simple matter of adjusting the values in a configuration file. More complex adjustments, such as supplying different objective functions or a different method to calculate the column badness, can be done by replacing a Lua-function with new code. As Lua is an interpreted language that code can be loaded at runtime, ie, as part of the configuration as well.

Experiments with a base algorithm for globally optimized pagination have shown that the relative performance hit, as compared with a greedy algorithm, is neglectable with today's powerful computer systems (ie, processing time increases by a factor of < 8, which means 10 instead of 1.3 seconds for a document such as Austen). However, with many documents (that do not contain enough flexible vertical space), the algorithm will run out of alternatives to optimize and thus manual correction, just as with the greedy algorithm, will still be necessary.‡‡‡‡‡

For successful global optimization, it is therefore important to develop methods that add additional flexibility to the pagination process. In this paper, we introduced 2 such methods. The approach of running columns on double spreads 1 line short or long and the use of variants in the text. The latter was implemented by automatically providing all paragraph variants (ie, paragraphs formatted with different numbers of lines), whenever this can be done without compromising the quality on the micro-typography level beyond a specified tolerance.

When applying the algorithm with the extensions, we add enough additional flexibility to fully optimize (nearly) every document without any manual intervention.§§§§§ In addition, the price to pay is acceptable if it avoids hours of iterative tinkering that are otherwise necessary when manually optimizing the results of a greedy algorithm.

Moreover, in fact, what is typically been done to manually resolve such issues is precisely what the extensions will automatically integrate into the algorithm: redoing some paragraphs to make them longer or shorter and running some columns long or short combined with placing explicit breaks in strategic places.

The base algorithm outlined in this paper does not handle additional auxiliary input streams such as floats (which of course raises the complexity further). As there are quite different models possible (some of them touched upon in Section 1.4), such work should be provided as extensions to the base algorithm, to enable easy comparison between different approaches. Results from such types of extensions are presented in the work of Mittelbach.[13]

Other interesting research topics are alternative approaches for limiting the search space in meaningful ways, or strategies that only locally consider variants if the pagination runs out of good options.

The current algorithm assumes that columns have a defined size (which can vary from column to column but is otherwise fixed) and that these columns are filled sequentially. This means that filling strategies that balance material across columns are not supported and cannot be optimized by the algorithm. It would therefore be interesting and important to develop alternative or extended algorithms that support these important types of designs as well.

†††††The framework will eventually become part of the standard TeX distributions.

‡‡‡‡‡There is still a huge advantage: The number of issues will be noticeably smaller and resolving them normally does not require an iterative process, which is the case with the greedy algorithm.

§§§§§It is certainly possible to construct documents that cannot be optimized even then. However, for most documents even using just one of the extensions will be sufficient.

ORCID

Frank Mittelbach http://orcid.org/0000-0001-6318-1230

REFERENCES

1. Mittelbach F. A general framework for globally optimized pagination. In: Proceedings of the 2016 ACM Symposium on Document Engineering DocEng '16; ACM; New York, NY: 2016:11-20. http://doi.acm.org/10.1145/2960811.2960820
2. Hailpern J, Venkata ND, Danilevsky M. Pagination: it's what you say, not how long it takes to say it. In: Proceedings of the 2014 ACM Symposium on Document Engineering DocEng '14; ACM; New York, NY: 2014:147-156. http://doi.acm.org/10.1145/2644866.2644867
3. Mittelbach F, Rowley C. The pursuit of quality: how can automated typesetting achieve the highest standards of craft typography? In: Vanoirbeek C, Coray G, eds. EP92: Proceedings of Electronic Publishing 1992, International Conference on Electronic Publishing, Document Manipulation, and Typography, Swiss Federal Institute of Technology, Lausanne, Switzerland, April 7-10, 1992; 1992:261-273.
4. Holkner A. Global Multiple Objective Line Breaking [Master's thesis]. Melbourne, Australia: School of Computer Science and Information Technology, RMIT University; 2006.
5. Ciancarini P, Di Iorio A, Furini L, Vitali F. High-quality pagination for publishing. *Softw Pract Exp*. 2012;42(6):733-751. https://doi.org/10.1002/spe.1096
6. Piccoli R, Oliveira J, Manssour I. Optimal pagination and content mapping for customized magazines. *J Brazilian Comput Soc*. 2012;18(4):331-349. https://doi.org/10.1007/s13173-012-0066-6
7. Hassan T, Hunter A. Knuth-plass revisited: flexible line-breaking for automatic document layout. In: Proceedings of the 2015 ACM Symposium on Document Engineering DocEng '15. ACM; New York, NY: 2015:17-20. http://doi.acm.org/10.1145/2682571.2797091
8. Knuth DE. *Computers & Typesetting, Volume A: The TEXbook*. Reading, MA: Addison-Wesley; 1986.
9. Hurlburt A. *The Grid: A Modular System for the Design and Production of Newspapers, Magazines and Books*. New York: Van Nostrand Reinhold; 1978.
10. Harrower T. *The Newspaper Designer's Handbook*. Dubuque, IA: Wm. C. Brown Publishers; 1991.
11. Enlund NES. Electronic Full-Page Make-Up of Newspaper in Perspective: Gutenberg-Gesellschaft, Internationale Vereinigung für Geschichte und Gegenwart der Druckkunst e.V., Mainz, Germany; 1991.
12. Gange G, Marriott K, Stuckey P. Optimal guillotine layout. In: Proceedings of the 2012 ACM Symposium on Document Engineering, DocEng '12; ACM; New York, NY: 2012:13-22. http://doi.acm.org/10.1145/2361354.2361359
13. Mittelbach F. Effective floating strategies. In: Proceedings of the 2017 ACM Symposium on Document Engineering, DocEng '17; ACM; New York, NY: 2017:29-38. http://doi.acm.org/10.1145/3103010.3103015
14. Thánh HT. Micro-typographic extensions to the TEX typesetting system [PhD dissertation]. Faculty of Informatics, Masaryk University: Brno, Czech Republic; 2000.
15. Knuth DE, Plass MF. Breaking paragraphs into lines. *Softw Pract Exp*. 1981;11(11):1119-1184.
16. Plass MF. Optimal Pagination Techniques for Automatic Typesetting Systems [PhD thesis]. Department of Computer Science, Stanford University. Stanford, CA. Report No. STAN-CS-81-970; 1981.
17. Wohlfeil S. On the Pagination of Complex Book-Like Documents [PhD thesis]. Hagen, Germany: Fernuniversität Hagen; 1998.
18. Brüggemann-Klein A, Klein R, Wohlfeil S. *Computer Science in Perspective*. New York, NY: Springer-Verlag New York, Inc.; 2003;49-68. http://dl.acm.org/citation.cfm?id=865449.865455
19. Jacobs C, Li W, Schrier E, Bargeron D, Salesin D. *Adaptive Grid-Based Document Layout*. Association for Computing Machinery, Inc; 2003. http://research.microsoft.com/apps/pubs/default.aspx?id=69470
20. Jacobs C, Li W, Salesin DH. Adaptive document layout via manifold content. *Second International Workshop on Web Document Analysis (wda2003)*. Liverpool, UK; 2003.
21. Mittelbach F. E-TEX: guidelines for future TEX extensions. *TUGboat*. 1990;11(3):337-345.
22. Mittelbach F. E-TEX: guidelines for future TEX extensions—revisited. *TUGboat*. 2013;34(1):47-63.
23. Knuth DE. *Computers & Typesetting, Volume B: TEX: The Program*. Reading, MA: Addison-Wesley; 1986.

24. Goossens M. The XƎTEX Companion: TEX meets OpenType and Unicode, Switzerland. 2010. http://xml.web.cern.ch/XML/lgc2/xetexmain.pdf

25. LuaTEX Development Team. 2017. http://www.luatex.org/svn/trunk/manual/luatex.pdf

26. Kido Y, Hikaru Y, Goran T, Akiko A. Document layout optimization with automated paraphrasing. 2015. http://doi.acm.org/10.1145/2682571.2797095

How to cite this article: Mittelbach F. A general LuaTeX framework for globally optimized pagination. *Computational Intelligence*. 2018;1–43 https://doi.org/10.1111/coin.12165

APPENDIX: PROOF THAT THE GREEDY PAGINATION IS THE SHORTEST PAGINATION ASSUMING A SUITABLE MEASURE

In Section 1.1, we remarked that it is easy to prove that the greedy pagination of a document (ie, the one that always uses the maximum amount of material possible) will be the shortest pagination possible under a suitable definition of "shortest." The proof of this statement is given in the following Lemma.

Lemma 1. *Let $b_0, b1, \dots, b_m$ be the sequence of all possible breakpoints of some document with the restriction that the distance between 2 consecutive breakpoints is always strictly positive (ie, $b_i < b_{i+1}$). Then, the greedy pagination of this document will always be the "shortest" pagination in the sense that it will either has less pages than any other pagination or it will have the same number of pages but a last page with less material.*

Proof. Let $g_0, g_1, \dots, g_n$ be the greedy pagination and let us assume that there exists a non-greedy pagination $x_0, x_1, \dots, x_{n'}$ that is shorter in the above sense. We have $b_0 = g_0 = x_0$ (start of document) and $b_m = g_n = x_{n'}$ (end of document).

If $n' < n$, ie, if the greedy pagination has more pages than the x-pagination, then there must exist an index k with $x_k \leqslant g_k < g_{k+1} < x_{k+1}$, ie, there exists a page in the x-pagination that contains at least 2 breakpoints from the greedy pagination. However, this contradicts the assumption that the g-pagination was greedy as g_{k+1} could be replaced by x_{k+1} and thus extending page $k + 1$ in the g-pagination.

Given that $x_0 = g_0$ the same argument applies if $n = n'$ and $x_{n-1} > g_{n-1}$ (ie, the last page has less material in the x-pagination). In that case there also exists at least one index k with $x_k \leqslant g_k$ and $x_{k+1} > g_{k+1}$, which, again, contradicts that the g-partition was greedy.

If $n = n'$ and $x_{n-1} = g_{n-1}$, then the last page is identical in both partitions, and we can confine ourselves to the shorter document ending in $x_{n-1} = g_{n-1}$. By repeatedly applying the earlier argument, we will then eventually show that the g-partition is not greedy or that both partitions are identical in all breakpoints, ie, are, in fact, the same partition. □

This result may appear somewhat surprising, as it seems to be contradicted by the fact that different line breakings of a paragraph (which is an analogue problem) can result in different paragraph heights and there the greedy line breaking may not produce the paragraph with the shortest height. However, this is only true because we apply a different measure in that case: We are looking at the height of a paragraph, which is the sum of the heights of all lines (plus their separations) and not at the number of lines produced. As elements in the paragraph may have different heights it is possible that changes in line-breaking change the overall height and thus the result, if the height is used as the measure.

However, if we translate the measure defined in Lemma 1 to the line-breaking scenario, then it would state that the greedy algorithm results in a line breaking with a minimal number of lines and in case of equal number of lines with less material in the last line and for that measure the statement from Lemma 1 is also true for line breaking.

This measure for the length of a pagination as used in the above Lemma is reasonable, as it means that we assume that the width of all text blocks in the galley are of identical width, or more precisely, that their width is not exceeding the page width, ie, that the length of the pagination is not affected by the width of individual pages.