Quantum boundaries for last line adjustments

Didier Verna

Abstract

The glue model of TEX is a powerful and elegant abstraction allowing for many typographic applications. One such application is the typesetting of a paragraph's last line, which requires special treatment. By adding infinitely stretchable glue at the end of a paragraph, the need for special-casing the final line is avoided, albeit at the cost of sub-optimal typesetting.

In conscientious typography, optimally typesetting the last line of a paragraph turns out to be a non-trivial problem, especially if one wants to take into account such aesthetic concerns as runt or full-out lines. In this paper, we present an algorithm for optimal last line adjustment, introducing the concept of "quantum boundaries": potential line breaks whose exact state remains unknown until an actual paragraph solution is observed.

1 Introduction

Absent micro-typographic adjustments [5], playing with inter-word spacing is the primary way to adjust the width of a line for justification. Inter-word blanks can be seen as elastic spaces, that is, objects of some natural width, but also with shrinking or stretching capabilities. In TEX [8], elastic spaces are reified through the concept of "glue". For example, a glue of `10pt plus2pt minus1pt` is naturally 10 points wide, but can also be stretched up to 12 points (ideally; more if necessary), or shrunk down to 9 points (and no less) if necessary. When TEX justifies a line, it stretches or shrinks all glue on that line uniformly, that is, proportionally to each glue's own elasticity.

Since the last line of a paragraph is usually too short to be justified, it deserves special treatment. TEX has a very elegant solution to this problem: it adds an infinitely stretchable glue of zero width at the end of the paragraph, while still pretending to attempt justification. This avoids the need for special-casing the end of the processing. The net effect is that if the line is naturally too long, it will be shrunk as needed (as any other line in the paragraph), but if it is short, it will remain at its natural width, since the final glue will essentially absorb all the required stretching.

The elegance of this solution has an aesthetic cost, however. As mentioned in [10], a final line at its natural width will look bad if the previous one is very tight or very loose, a defect already taken into account by TEX through its concept of "fitness classes" and its `\adjdemerits` parameter (see also [14] on the matter). Since the actual width of the final line is flexible, a naive approach to this problem would be to simply adjust it manually *afterwards* so that it looks closer to the previous one in terms of stretching or shrinking. As we will soon discover, the problem is in fact much more subtle than that, to the point that post-processing may turn out to be insufficient on many occasions. Put differently, this means that sometimes, a completely different breaking solution is required in order to improve the quality of the paragraph as a whole.

This paper is organized as follows. Section 2 presents the aesthetic considerations that we want to take into account when adjusting the appearance of a paragraph's final line. Section 3 introduces some preliminary technical considerations that are needed in order to fully understand how the final line processing is implemented. Section 4 presents the algorithm *per se*. Finally, Section 5 offers an experimental study of the algorithm's behavior with respect to the original one.

2 Aesthetic considerations

Note: the discussion in this section is the result of personal exchanges with Thomas Savary, a French compositor using LATEX in the professional publishing field (`dcao-sarl.fr`).

As previously mentioned, the first desired aesthetic improvement to the final line is to make it closer to the one before last in terms of stretching or shrinking. There are two additional considerations that should be taken into account if one aims at conscientious typography: the final line should be neither too short, nor too long. This concern is raised in existing literature, for example by Bringhurst [3].

2.1 Runt lines

A "runt line" is an excessively short final line. Runt lines have a general tendency to degrade the typographic color, sometimes to a critical point. Imagine for example two consecutive paragraphs, separated by a final line the width of which is close to the indentation width (`\parindent`). The result would be catastrophic, as I am artificially illustrating just: here.

From a more abstract point of view, runt lines are often constituted of a single word, which is also bad because it entails two micro-interruptions for the reader: the eye needs to move to the next line, only to find a single sentence-ending word. That particular case can be avoided by using an unbreakable space before the final word, but this is obviously not a general solution.

Traditionally, recommendations for runt line avoidance have been expressed as a minimum number of characters that the line should have, or as a minimum width in absolute value. This, however, does not make much sense. In the former case, the widths of the individual characters may hinder the recommendation. For example, the word *mummy* is much wider than the word *little* despite having fewer characters. In the latter case, a minimum width of say, 2 cm, would probably look fine for an 8 cm wide paragraph (such as in two-column formatting), but would be too short at 16 cm.

In our experience, it is thus preferable to express the minimum width a final line should have in proportion to the paragraph width. We call this ratio the "runt threshold". We think that 15% is a good default value, and as it turns out, it is exactly the ratio used by Thomas Savary in his `lua-typo` configuration.

To close on this subject, here is a fun fact about runt lines. In French, they are called "thief lines" ("lignes à voleur") [2]. This is reminiscent of ancient times when compositors were paid by the line. Keeping a final line so short that its content could have fit on the previous one could be seen as an abusive practice from the compositor, although most of the time, this was simply the result of `\sloppy` typesetting…

2.2 Full-out lines

Reciprocal to runt lines, a "full-out line" (sometimes called "nearly full") is a final line which is almost justified, but not quite. Such lines are equally undesirable because they give the impression that full justification was attempted, but failed (again, I am artificially demonstrating this with this line's end).

In order to avoid this effect, we have two possibilities in the most favorable cases: we can either fully justify the line, or make it shorter than it naturally is. In France, the recommendation from Jean-Pierre Lacroux is that the final line of a paragraph be short by at least 1em (`orthotypographie.fr`). He also argues that "beautiful typography does not allow a fully justified final line".

Thomas Savary has a slightly different view on this matter. While he admits following the "short by at least 1em" recommendation, he still likes fully justified final lines. Here, the rationale is that paragraph indentation is the modern way to visually distinguish a paragraph from the next, so in a sense, having a short final line followed by an indented first one is redundant. We tend to agree with this view.

On the other hand, our feeling about absolute measurements remains the same as in the case of runts, and we think it preferable to reason proportionally to the paragraph's width. We thus define a so-called "full-out threshold" to this aim. We think that 5% is a good default value, meaning that the final line should either remain under 95% of the paragraph width, or be fully justified. Thomas Savary has expressed his interest in this idea, and is ready to try it out.

In France, there is no official term to designate full-out lines. They are sometimes called "nearly full", but there certainly isn't any designation as picturesque as "thief lines" for runts (or as "widows" and "orphans" for that matter). I like to call them "deceptive lines" ("lignes trompeuses") in reference to the fact that they give the wrong impression of a failed justification attempt. In the aim of keeping in line with the slangish spirit of [2], it seems that Thomas would go as far as calling them "insidious", or even "vicious"…

2.3 The case of `\parfillskip`

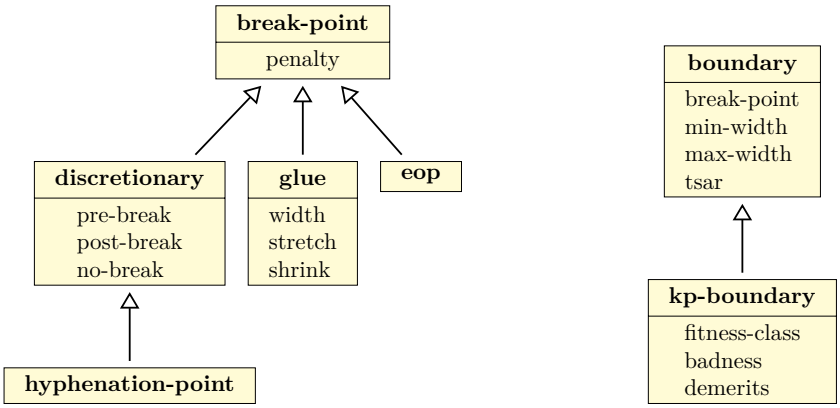
In TEX, we have control over the final glue via the primitive `\parfillskip` parameter. By modifying its default value, we can meet some of the aesthetic considerations we just described. For example, by setting it to `0pt` (without any elasticity), we can force the final line to be fully justified. By setting it to some non-zero width, we can force it to be shorter than the full-out threshold.

Nevertheless, in general it is *not* possible to let the algorithm automatically choose between full justification or shorter width on a paragraph basis; even less so to automatically adjust the amount of used elasticity based on the one-before-last line.

In the following sections, we are going to study the design and behavior of an algorithm which is not only able to adjust the spacing of the last line depending on the previous one, but also to avoid runts and full-outs. We can see already that the treatments of these two *avoid zones* differ. A runt line can only be fixed by stretching it to at least the runt threshold if possible (otherwise, a different paragraph-breaking solution needs to be found). On the other hand, a full-out line can either be shrunk to at most the full-out threshold, or be stretched to exactly the paragraph width (again, if at all possible).

3 Technical considerations

Before studying the algorithm we propose, a number of technical considerations need to be made explicit, especially since our implementation of the Knuth-Plass (KP) line-breaking algorithm [7, 9] diverges greatly from the original one.



(a) Break points (b) Boundaries

Figure 1: Break point and boundary classes

3.1 The final glue

As mentioned earlier, adjusting the length of the final line requires that we depart from TEX's original (and quite clever) idea of finishing a paragraph with an infinitely stretchable glue of zero width. This is an important modification to the algorithm, but it is in fact not a problem for us because ETAP [11, 12], our platform for typesetting experimentation (`github.com/didierverna/etap`), already treats the paragraph's end differently.

ETAP uses a set of object-oriented data structures to represent typesettable items (characters, glue, *etc.*) and algorithmic data. Figure 1(a) provides a (much) simplified view of the `break-point` hierarchy used to represent places where a line may be broken. In our implementation, penalties are expressed as properties of each and every break point, and can thus be adjusted individually. The end of the paragraph is represented by instances of a special class called `eop`. In our KP implementation, the detection of an `eop` break point triggers a specialized routine, shrinking the final line if needed, leaving it at its natural width otherwise.

3.2 Boundaries

Another important hierarchy is the `boundary` one. Boundaries represent potential line endings associated with the algorithmic properties necessary for paragraph justification. Figure 1(b) provides a simplified excerpt of this hierarchy. The base class allows storing the corresponding end of line break point, the line's minimum and maximum width given its total elasticity, and the so-called Theoretical Spacing Adjustment Ratio (TSAR). The TSAR represents the amount of available elasticity required to reach the target justification width (negative for shrinking, positive for stretching). The TSAR may be infinite if no elasticity is available.

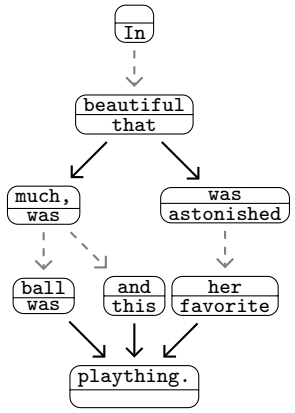


Figure 2: Line breaking graph example

This base class is used in various greedy algorithms, but our KP implementation defines a subclass called `kp-boundary` which is used to store additional properties, in particular, the line's fitness class, badness, and local demerits (depending on the badness, break point penalty, and line penalty).

3.3 Sharing

One final, yet extremely important technicality that needs to be stated is about the sharing of data structures. Our boundaries (which are more or less the equivalent of "nodes" in KP terminology) contain line information, but should remain independent of the surrounding context (typically, the previous lines).

Remember that paragraph justification may be seen as a "Single Pair Shortest Path" problem [4, 6]. The possible solutions for breaking a paragraph into lines are represented in the form of a graph (as illustrated in Figure 2), in which every possible line break is a node, and the ability to go from one line break

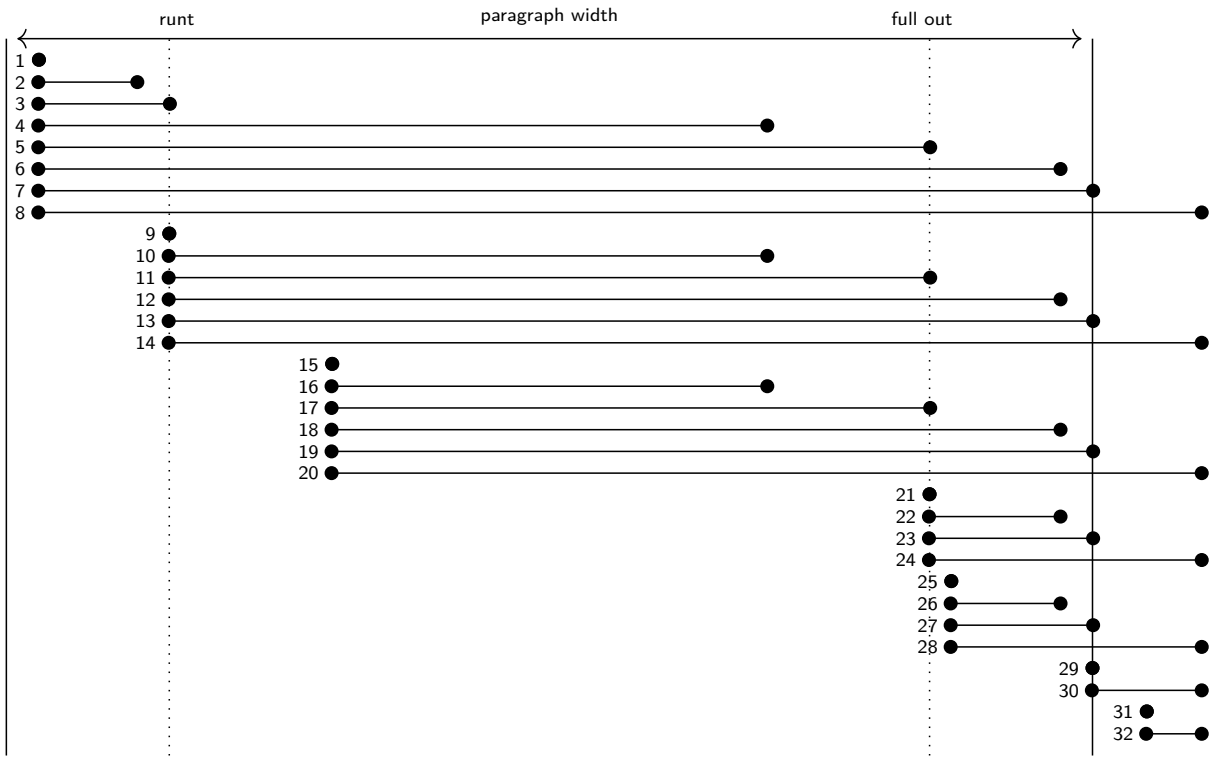


Figure 3: Last line morphology: 32 cases

to the next is represented by an edge connecting two nodes. In the figure, each node advertises the corresponding end-of-line, and the beginning of the next. The problem thus boils down to finding the best route (referred to as "shortest path" in graph theory) from the top node to the bottom one.

In order to perform efficiently, the KP algorithm uses a "dynamic programming" [1] optimization technique which allows it to never construct the full (potentially huge) graph in memory. As it progresses through the paragraph's text, TEX maintains branches representing the best possible solutions (note the plural) *so far*, but also gets rid of branches which are provably going to be sub-optimal in the end.

In order for us to collect interesting experimental data (Section 5) ETAP provides two versions of KP: one using the same dynamic programming optimization technique as in the original version, and one constructing the full graph in memory. Whether optimized or not, sharing nodes (boundaries) in the graph is a property that we want to retain in order to save resources: given the (theoretically) exponential complexity of the problem, creating a tree of solutions would be unrealistic in practice.

Technically, there are two situations in which it is not possible to share nodes. In those situations, we need different nodes for same break point. The first case is that of a non-rectangular paragraph, as the shape of the lines (and whether a line would be feasible at all) depends on the line number. The second case is the handling of adjacent demerits, which are obtained by comparing the current line with the previous one. As adjacent demerits do not impact the feasibility of a line (only its aesthetic cost), it is not a problem in the unoptimized version of the algorithm, but it has to be taken into account in the dynamic programming version: two lines beginning at the same position but preceded by different fitness classes need to be differentiated.

As we will soon discover, adjusting the shape of the final line entails more complication when it comes to sharing the boundaries in the graph of solutions.

4 The final-line adjustment algorithm

We are now ready to study our final line adjustment algorithm. Meticulous examination of the different situations reveals that there are no less than 32 individual cases that need to be addressed. Fortunately, many of those have equivalent semantics, or can be handled in the same way.

Figure 3 depicts the morphology of those 32 cases. Each segment represents a final line's width range, from minimum to maximum (that informa-

tion is available from boundary instances; see Figure 1(b) again). The range in question depends on the algorithm's tolerance setting. For cases 1 to 8, the minimum width lies somewhere below the runt threshold. For cases 9 to 14, the minimum width is exactly the runt width, *etc.* In cases 1, 9, 15, 21, 25, 29, and 31, the line has no elasticity so the minimum and maximum width are the same.

Now, the algorithm proceeds as usual, examining the feasibility of break points successively, except that when an `eop` (end of paragraph) break point is detected, the corresponding boundary is created in a way that depends on the 32 cases exhibited in Figure 3.

4.1 Simple cases

We start with the simple cases where regular boundaries are sufficient.

4.1.1 Runts

In cases 1, 2, 3, and 9, we compute the boundary's TSAR for a target width of `runt`, indicating that the line should be stretched to the minimum tolerable. The treatment is the same, but the outcome is different. In cases 1 and 2, this leads to an underfull which is eliminated. In case 3, this leads to the only acceptable solution. In case 9, there's nothing to be done.

4.1.2 Full-outs

Cases 21 and 22: We compute the boundary's TSAR for a target width of `full-out`, indicating that the line should be shrunk to the maximum tolerable width. In case 21, there is nothing to be done. In case 22, this leads to the only acceptable solution.

4.2 Full justification

Cases 27, 28, 29, 30, 31, and 32: We compute the boundary's TSAR for full justification to avoid not only overfull lines, but also the full-out zone. The treatment is the same, but the outcome is different. In cases 27, 28, and 30, this leads to the only acceptable solution. In cases 31 and 32, this leads to an overfull which is eliminated. In case 29, there is nothing to be done.

4.3 Quantum boundaries

The remaining cases are more complicated because they involve several possible target widths (whether feasible or not). We thus need new data structures to record those multiple possibilities, while not making a premature decision, since we still want these boundaries to be shared in the solutions graph.

By analogy with quantum mechanics (really, for marketing purposes), we call these "quantum boundaries" to illustrate the fact that these boundaries can be in multiple states at the same time, until a specific breaking solution is observed, and they can be frozen with a definitive TSAR.

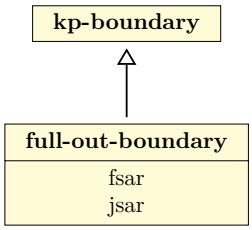


Figure 4: Quantum full-outs

4.3.1 Full-outs

In cases 23 and 24, we use a new boundary class depicted in Figure 4 to record the fact that the corresponding line could be set to either the full-out width, or the paragraph width. The Full-out Spacing Adjustment Ratio (FSAR) records the amount of elasticity required to set the line to the full-out width, and the Justification Spacing Adjustment Ratio (JSAR) to the paragraph width respectively.

Full-out boundaries inherit from all the properties in their super-hierarchy (Figure 1(b)), so they also have a regular TSAR. This property is initialized to $\min(|\text{FSAR}|, |\text{JSAR}|)$ so as to remain as close as possible to natural spacing.

4.3.2 Intolerable full-outs

Cases 25 and 26 are treated like 23 and 24, but the outcome is different: both the FSAR and the JSAR are above the tolerance. In other words, these boundaries are both overfull (with respect to the full-out width), and underfull (with respect to the paragraph width). Also, the regular TSAR is initialized to the JSAR, rather than to the FSAR. The reason for this choice is that with some elasticity, underfull lines get a (very large) numerical badness instead of an infinite one. In the unoptimized version of the algorithm, this allows us to sort the set of solutions (by increasing total demerits) with more accuracy.

4.3.3 Interlude: About the TSAR

At this point, the reader may wonder if it really makes sense to keep the original TSAR property around, especially since we seem to initialize it to a value that may not be the final one, and since the value in question is already available somewhere else (it is either the FSAR or the JSAR, after all). There are three reasons for this.

1. When the regular TSAR is initialized, so is the boundary's badness. This means that if the corresponding break point is infeasible, the original algorithm will be able to filter the boundary out

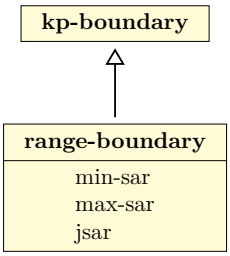


Figure 5: Quantum ranges

before even trying to make a final decision on the line's appearance, thus avoiding unnecessary computation.

2. In the case of a completely infeasible paragraph, there is a chance that this boundary be used as an emergency boundary, in which case a decent initial value for the TSAR will make sure that the (bad) solution still looks not so bad.
3. If the last line of the paragraph also happens to be the first, there is no previous line to compare to, so again, it makes sense to initialize the TSAR to the most sensible value.

4.3.4 Ranges

All the remaining cases except 15, for which there is nothing to do, require a different data structure, as they offer a full range of solutions, plus in some cases the possibility to fully justify the line. The new data structure in question is depicted in Figure 5, and used for cases 4–8, 10–14, and 16–20.

A final line belonging to one of these cases can extend between $\max(\textit{min-width}(\textit{boundary}), \textit{runt})$ and $\min(\textit{max-width}(\textit{boundary}), \textit{full-out})$. The boundary's minimum and maximum Spacing Adjustment Ratios (SARs) (*min-sar* and *max-sar*) are set accordingly.

In cases 4–6, 10–12, and 16–18, the line cannot be justified, so the JSAR is set to `nil`. On the other hand, in cases 7, 8, 13, 14, 19, and 20, it is set to the appropriate numerical value.

The only remaining question is about initializing the TSAR to a sensible value (see Section 4.3.3). We essentially make sure that the value in question is within $[\textit{min-sar}, \textit{max-sar}]$ or equal to JSAR when possible, while minimizing the distortion from natural spacing. More specifically:

- lines naturally shorter than the runt will initially be stretched to it,
- lines naturally overfull will be initially shrunk to justification,
- lines naturally within the full-out range will be either shrunk to full-out or fully justified when possible, whichever is closer to natural spacing,
- all other lines will remain at their natural width.

4.4 Finalization

We now have the ability to represent the paragraph breaking solutions space as a graph of boundaries, with some (the quantum boundaries) carrying a level of uncertainty as to their final state. Finalizing their state is in fact fairly straightforward: we want their TSAR to be as close as possible to the previous line's one, within the range of acceptable values.

Since the final state of a quantum boundary depends on the one-before-last line, they cannot be ultimately shared, so we need fresh copies in each and every breaking solution.

4.5 Unoptimized version

In the unoptimized version of the algorithm, the solutions graph is constructed in its entirety first. All the solutions are then computed and sorted by increasing demerits. A particular solution is essentially a sequence of (shared) boundaries from the graph, leading to the end of the paragraph. If the graph contains a quantum boundary, it is cloned into the solution being created instead of used directly. This ensures that the finalization of its TSAR does not affect the other solutions.

4.5.1 Optimized version

In our implementation of the optimized KP, the active boundaries (nodes) are stored in a hash table. Recall that different boundaries for the same break point are already needed sometimes, namely in the case of a non-rectangular paragraph, or in the case of different fitness classes leading to it. This is done by using hash table keys consisting of the breakpoint itself, the line number, and the incoming fitness class.

In order to prevent the sharing of quantum boundaries across different solutions, the hash table keys for accessing quantum boundaries are made to automatically include the previous node as well. This ensures that all active paths lead to a different end-of-paragraph quantum object.

4.6 Final badness

In the original algorithm, a naturally overfull line is shrunk to justification, while all other lines remain at their natural width. This means that final lines have non-zero badness only if they absolutely *need* to be shrunk.

The situation is now different, because a naturally fitting line may be stretched or shrunk, in order to get closer to the previous line, or to one of the thresholds. Thus the question of taking a final line's badness into account needs to be considered: if we artificially adjust the final line's width for aesthetic

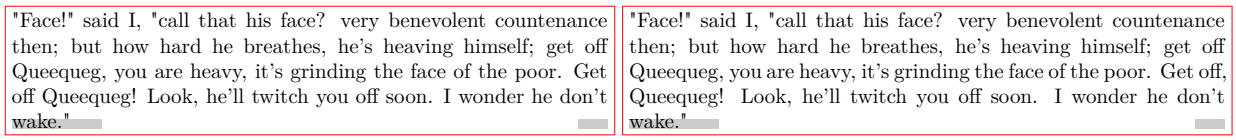


Figure 6: Additional failure example

reasons, what sense does it have to also increase its badness?

We think that the final line's badness should indeed be counted in full, for several reasons. The first one is that the original algorithm counts it in for naturally overfull lines, so it's a matter of staying consistent across all situations.

The second reason is that the badness of a line is a *local* aesthetic measure, mostly interested in the inter-word space distortion, regardless of the target justification width. In other words, if a very loose line looks bad in the middle of a paragraph, so will a final line, even if it occupies only half the width.

The final reason, and perhaps the most important one, is that by counting the badness of the final line in, we give the algorithm more opportunities to look for alternative solutions. Remember in particular that KP only has four fitness classes, which gives only a very coarse measurement of a paragraph's blank homogeneity [14]. Concretely, when adjusting the width of the final line, there is a good chance that its fitness class will remain the same, effectively increasing the paragraph's total demerits. Whether this degradation can be so important that a completely different breaking solution becomes preferable is an interesting question.

5 Experimentation

In order to study the impact of our last line adjustment algorithm on the behavior of KP as a whole, we ran both versions on 1279 paragraphs from the novel *Moby Dick* (`gutenberg.org/ebooks/2701`) with a paragraph width of 284 pt (approximately 10 cm), and with the Latin Modern Roman 10 pt font. All original KP parameters were left at their default settings, and all other extensions of ours [13, 14] were deactivated.

5.1 Failures

The original algorithm fails to typeset 33 paragraphs out of the 1279 (2.58%). With final line adjustments, we observe 4 additional failures, all due to a runt final line that cannot be fixed. Figure 6 illustrates one such case. The runt and full-out avoid zones are grayed out in the background.

In this particular situation, KP would find only two acceptable solutions in total, both ending with the same runt final line with no elasticity. Notice how the second solution is even worse than the first one: there are two similarities, ending lines 2 and 3, and beginning lines 3 and 4.

5.2 Identical solutions

In 1099 cases (85.92%) both algorithms came up with the same breaking solution, meaning that it is enough to post-process the (same) final line. This is a remarkable figure, as it means that in the vast majority of the cases, the original solution is not only readily adjustable, but also remains the best choice after adjustment.

5.2.1 Acceptable lines

Further analysis reveals that among those 1099 cases, 978 (88.98%) already end with an acceptable final line. It is thus only a matter of slightly adapting its stretching or shrinking ratio. An example is given in Figure 7. Notice how the final line in the rightmost version has been slightly stretched in order to better match the previous line.

5.2.2 Adjustable runts

In only *one* case out of 1099, the original solution is runt, yet adjustable. The final line is "end boat.", thus containing only one glue item. Its natural width is 41.94 pt and needs to be stretched to 42.6 pt in order to reach the runt threshold. In doing so, the TSAR difference from the previous line happens to

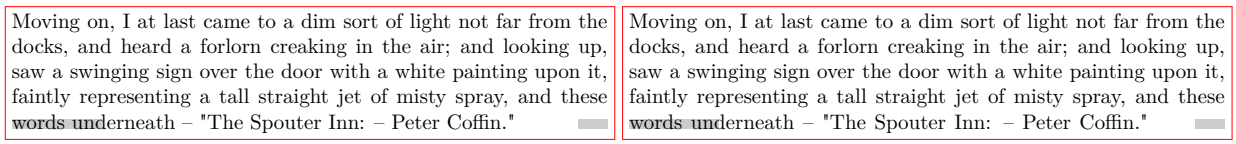


Figure 7: Acceptable solution example

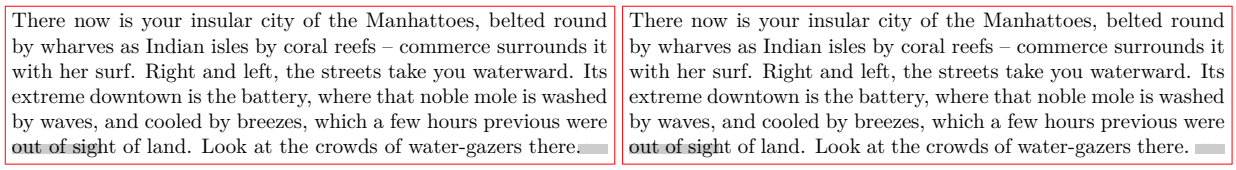


Figure 8: Adjustable full-out example

increase, but both lines remain decent. Reaching the minimum width takes precedence over adjacency considerations. This case is not displayed here as the difference between the two solutions (0.66 pt) is hardly visible.

The extreme scarcity of adjustable runts is not particularly surprising. Runt lines being very short, they are likely to be rigid (a single word) or barely elastic (very few glue items).

5.2.3 Adjustable full-outs

70 of the 1099 identical solutions (6.37%) naturally fall in the full-out avoid zone, but have enough elasticity to be adjusted. Remember that the adjustment can go both ways: by shrinking to at most the full-out width, or by stretching to the paragraph width.

Figure 8 gives an example. In the original version, the final line is slightly too long (not by much). In the adjusted solution, it has been shrunk more than absolutely necessary. That is because it was possible to match the shrinking ratio of the previous line exactly.

5.2.4 Adjustable overfull lines

In the remaining 50 cases (4.55%), the final line is naturally overfull but is elastic enough to be shrunk to full justification. In that situation, the final appearance is exactly the same in both versions, since the original algorithm needs to shrink the line as well.

In theory, some naturally overfull final lines could have enough elasticity to be made even shorter than the full-out threshold. This would be possible in cases 8, 14, 20, and 24 from Figure 3. Although we did not check that specifically in our experiments, it is extremely unlikely to happen, as the amount of required shrinking would be quite high, and also would occur only in order to match an extremely tight one-before-last line.

5.3 Different solutions

In the remaining 143 cases (11.18%), the final solution is different from the original one. Among these, 119 are obtained within the same pass, and 24 involve another pass of the algorithm.

5.3.1 Out of necessity

In the majority of the cases, that is, 108 out of 143 (75.52%), the algorithm needs to look for another solution because the original one is not adjustable. But perhaps the more interesting result here is that the cause for failure is a runt final line in *every single case*. This, again, is explained by the fact that runt lines are very short, so they are likely to have no, or very little stretchability. Figure 9 provides an example of that situation, with the original solution and the alternative one side by side. Notice how the alternative version has been loosened starting on line 3, in order to fill more of the final line.

5.3.2 Out of choice

There are also situations in which the original solution could have been adjusted, yet the algorithm decided on a completely different one. Among these, we observed 27 paragraphs with an already acceptable final line, one paragraph with an adjustable runt line, and 7 paragraphs with an adjustable full-out line. This amounts to roughly 25% of the cases where a different solution is chosen, so this is definitely not negligible.

Choosing an alternative layout when the original one could have been adjusted can only be due to one reason: taking a stretched final line's badness into account (see Section 4.6), which never occurs in the original algorithm since final lines are never stretched.

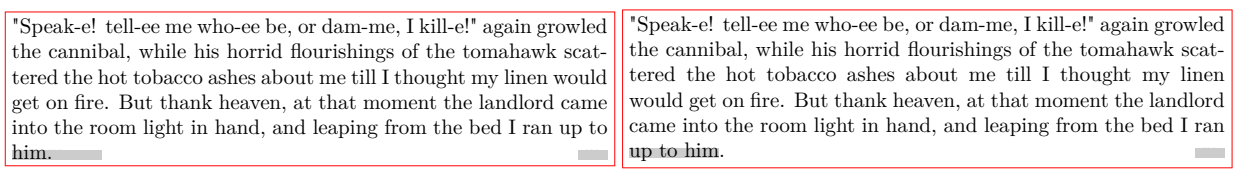


Figure 9: Necessary alternative example

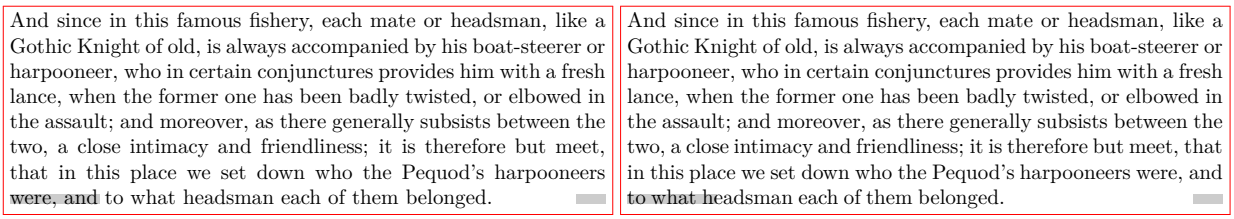


Figure 10: Arbitrary alternative example

From that perspective, it is not surprising that a naturally adjustable overfull final line never leads to choosing an alternative layout: in both versions of the algorithm, the shrinking badness is taken into account, so the total demerits remain the same in both cases (and minimal).

Let us explain in detail what happens for a specific example. Figure 10 shows a perfectly fine layout on the left, yet a different final choice on the right. In the original solution, the final line's badness is 0 since it remains at its natural spacing. The one-before-last line is loose; it has a TSAR of approximately 0.90. No adjacent demerits are involved, and the total demerits of this paragraph amount to approximately 14 543.

Stretching the final line to 0.90 puts its badness at approximately 72.5. The line becomes loose and, as previously, no adjacent demerits are involved. But this time, the total demerits increase to approximately 21 249. This is in fact more than for the paragraph on the right, the total demerits of which only amount to 18 594. As a matter of fact, the original solution, once adjusted, has become the algorithm's second choice only.

Interestingly enough (and in all honesty), the original solution here is probably still better than the new one, although it is *not* the fault of the algorithm. Look at the one- and two-before-last lines in the alternative version. They both end with a single word after punctuation (`, that` and `, end`) which is unfortunate. Yet, it is not the fault of the algorithm because that kind of defect is simply not taken into account, in either version. The usual solution to this problem is to penalize the undesirable line breaks more heavily.

5.4 Additional figures

We conclude this section with three quick final observations.

5.4.1 Pruning ratio

Recall that because of the runt and full-out thresholds, some acceptable breaking solutions in the original algorithm cannot be adjusted, and hence become unacceptable in the new one. During our experiments, we recorded the total number of available solutions in both cases, and found that in average, the additional constraints put on the final line eliminate 7% of the original possibilities.

This, however, is unlikely to have any noticeable impact on the performance of the algorithm, since the culling in question occurs only for the final line, so quite late in the processing.

5.4.2 Lookup depth

When the refined algorithm chooses an alternative layout rather than simply adjusting the original one, this layout also exists as a possible choice in the original version (modulo the final line adjustment). It is just not the first one. In our experiments, we recorded the ranking of the alternative layouts in the sorted list that the original algorithm produced. We found the average rank of the alternative solution to be 2.11. This means that when an alternative layout is preferable, it is not far from the original one; on average, it would be the third best choice.

5.5 Paragraph length

Choosing an alternative layout may also have an impact on the paragraph's length. In our experiments we also recorded that information. It turns out that alternative layouts are one line shorter than the original one in 33% of the cases for the same pass of the algorithm, and 20% when an additional pass is required. The remaining cases are of equal length: an alternative layout is *never* longer than the original one.

Again, this might not be very surprising, as long final lines usually have a lot of elasticity to play with. On the other hand, this also means that in a fair amount of situations, runt lines cannot be adjusted and their contents needs to be incorporated in the previous lines somehow.

6 Conclusion

Fine-tuning the appearance of a paragraph's final line in conscientious typography presents interesting challenges. Post-processing an already acceptable

solution is not trivial since we need to depart from TEX's original model (the final, infinitely stretchable glue). Avoiding runt or full-out lines cannot always be done by simply adjusting their length: a completely different breaking solution is sometimes required.

In this paper, we have presented an algorithm which is able to do all that with minimal modifications to KP. In addition to harmonizing the spacing ratio of the final line to that of the previous one, the algorithm defines two adjustable parameters, the runt and full-out thresholds, specifying at which point a final line should be considered too short, or too long, and thus avoided. Since a final line's width essentially becomes a moving target, the algorithm introduces the concept of quantum boundaries, allowing us to carry the uncertainty until the last minute, and to preserve the node-sharing property in the graph-based representation of the problem.

Experimentation with this algorithm gave us a number of interesting results. First of all, the additional aesthetic constraints on the final line lead to only 0.3% additional failures (where some constraints need to be relaxed). Secondly, in roughly 86% of the cases, the breaking solution remains the same, modulo the adjustment of the final line's width. Experience also shows that natural runt lines are very rarely adjustable, which is explained by their lack of elasticity. Thirdly, in the remaining 11% of the cases where a different breaking solution is required, the most frequent cause (75%) is that the original solution is not adjustable enough. In the remaining 25% of the cases, another solution is arbitrarily chosen because the revised algorithm takes a final line's stretching badness into account.

Finally, our experiments lead us to discover that the additional constraints put on the final line reduce the number of possible solutions by 7%, and lead to shorter paragraphs in 20% to 33% of the cases where the solution is different. That represents only 2.5% of the total number of feasible paragraphs.

While taking the final line's stretching badness into account could still be debated, we do think it is a good idea, be it only because it gives more room to the algorithm for exploring alternative layouts. Even though the provided example (Figure 10) is not particularly conclusive (see Section 5.3.2), it is very likely that the outcome would be generally improved in combination with a more subtle handling of adjacent demerits. As a matter of fact, we already have that development in place [14]. Its influence on the final line adjustments described in this paper remains to be studied.

References

[1] R. Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954. doi.org/10.1090/S0002-9904-1954-09848-8

[2] E. Boutmy. *Dictionnaire de l'Argot des Typographes*. Bibebook, 1883.

[3] R. Bringhurst. *The Elements of Typographic Style*. Hartley & Marks Inc., 2nd ed., 1996.

[4] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec. 1959. doi.org/10.1007/BF01386390

[5] Hàn Thế Thành. Micro-typographic extensions to the TEX typesetting system. *TUGboat* 21(4), 2000. tug.org/TUGboat/tb21-4/tb69thanh.pdf

[6] P. Hart, N. Nilsson, B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[7] D.E. Knuth, M.F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981. doi.org/10.1002/spe.4380111102

[8] D.E. Knuth. *The TEXbook*, vol. A of *Computers and Typesetting*. Addison-Wesley, 1986.

[9] D.E. Knuth. *TEX: The Program*, vol. B of *Computers & Typesetting*. Addison-Wesley, 1986.

[10] F. Mittelbach. E-TEX: Guidelines for future TEX extensions — revisited. *TUGboat* 34(1), 2013. tug.org/TUGboat/tb34-1/tb106mitt.pdf

[11] D. Verna. ETAP: Experimental typesetting algorithms platform. In *15th European Lisp Symposium*, pp. 48–52, Porto, Portugal, Mar. 2022. doi.org/10.5281/zenodo.6334248

[12] D. Verna. Interactive and real-time typesetting for demonstration and experimentation: ETAP. *TUGboat* 44(2):242–248, 2023. doi.org/10.47397/tb/44-2/tb137verna-realtime

[13] D. Verna. Similarity problems in paragraph justification: an extension to the Knuth-Plass algorithm. In *Proceedings of the ACM Symposium on Document Engineering 2024*, DocEng'24, pp. 127–130, New York, NY, USA, Aug. 2024. Association for Computing Machinery. doi.org/10.1145/3685650.3685666

[14] D. Verna. Towards more homogeneous paragraphs. In *Proceedings of the ACM Symposium on Document Engineering 2025*, DocEng'25, pp. 112–121, New York, NY, USA, Sept. 2025. Association for Computing Machinery. doi.org/10.1145/3704268.3742696

◇ Didier Verna
EPITA Research Lab
didier (at) didierverna.net
www.didierverna.net/
ORCID 0000-0002-6315-052X