GLOBAL MULTIPLE OBJECTIVE LINE BREAKING

ALEX HOLKNER

Supervisor: DR XIAODONG LI

Honours Thesis

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

October, 2006

Abstract

The line breaking problem is as follows: given some text and a page to print to, where are the best places to start new lines within the paragraphs for the most visually appealing layout? The most well-known and used optimising typesetting software is TEX, which solves the line breaking problem according to an internal function of a paragraph's "badness."

We propose an alternative formulation of the problem in which the quality of a candidate paragraph is measured along several objective functions which are easily defined by real-world solution-space metrics. Rather than presenting a single solution to the user, our algorithm finds the Pareto-optimal set of paragraphs given a set of objective functions.

To support our algorithm, we devise a new method for determining feasible hyphenation points within a paragraph in a single pass, which is general enough to apply to and improve the original TEX algorithm.

Our results show that global multiple objective paragraph optimisation gives solutions consistently better than TEX along our real-world metrics, and that a range of solutions are returned representing a genuine trade-off in typographic quality.

Contents

1 Introduction

This paper is concerned with typesetting paragraphs for printing in an efficient, aesthetically pleasing and adjustable manner. Since 1982 this task has been almost the exclusive domain of TEX (Knuth, 1979), a program that produces output far superior to a standard word processor. It does this by considering entire paragraphs and pages at a time when deciding where to introduce line breaks.

Specifically, TEX defines a "badness" function based on the spacing between words in a line, and adds penalties if a line break would appear in a mathematical formula or by hyphenating a word, for example. The task of TEX is then to select an optimal set of line breaks that minimises this function.

A naïve approach, which calculates the badness over all possible break opportunities, is doomed to failure: for a paragraph of n words there are 2^n possible ways of breaking the paragraph. Of course, most of these ways are quite ridiculous—for example, the case in which a break is made after every word. In this case we consider only the *feasible* breakpoints: those which can be made while not exceeding the tolerances of spacing for a line. The method in which these feasible breakpoints are found and the optimal set of breakpoints determined is the critical part of TEX.

We note that TEX's badness function is comprised of a weighted sum of typographical features: penalties and the word spacing. The weights and penalties applied to this function must be input by an expert, and after much trial and error. This is something of a black art, and almost without exception users do not modify TEX's default settings.

In this paper we present an alternative formulation of the line breaking procedure, in which the badness function is vector-valued. Each element of the vector corresponds to a single typographical feature, for example, the word spacing or the number of line-breaking hyphenations. Rather than return a single optimal solution, we return to the user the Pareto-optimal set of solutions. This set of solutions are optimal in the sense that none can be improved along any dimension without decreasing the score along another. The user is then free to apply a secondary weighting procedure to the set to select one, or use their domain-specific knowledge to choose the most appropriate or visually pleasing solution.

Rather than use arbitrary weighted penalties, we use simple metrics that we show are clearly related to the quality of a paragraph. Metrics can be defined which are not possible in TEX. These metrics can be defined in just a few lines of code, and can be selected and combined at will by the user.

We pose and attempt to answer the following research questions:

- Is it possible to define meaningful metrics for paragraph layout that describe well-known typographical qualities such as looseness, hyphenation, rivers, and so on?
- Can multiple objective optimisation be applied to the line breaking problem to return multiple competing solutions? How do these solutions compare to TEX's?
- How fast does our technique perform? How much memory does it require?
- What benefits does global multiple objective line breaking give over local single optimising line breaking?

2 Background

This paper crosses several disciplinary boundaries. The most obvious, typesetting, is not an active area of research in computer science, so some time will be spent giving a brief history and introduction to the terminology and practice.

Dynamic programming is a mathematical technique for reducing the dimensionality or complexity of a problem. While it is used in countless algorithms, it is useful to examine the background of dynamic programming as it applies to problems similar to the one at hand.

Multiple objective optimisation is a field gaining much recent attention across a range of problem types. Multiple objective dynamic programming is a particularly difficult problem that has not been "solved" to date. We also examine other methods of multiple objective optimisation, such the use of evolutionary and sociological models.

2.1 Typesetting

Typesetting is the process of arranging type onto a page in an aesthetically pleasing and readable manner. In the early 1400's Johann Gutenberg invented movable type—a system of arranging metal type, *sorts*, into lines, which in turn were set in a frame by hand. Assembled type could then be pressed against ink and paper relatively quickly.

It was the typesetter's responsibility to form the type into frames from the author's manuscript. Decisions must be made regarding how far apart words should be spaced, where one line should be broken and the word placed on the next line, and where a page should end and the next one begin.

Typically lines are justified, meaning the margins are flush at both the left and right edges, as they are in this paper. This is achieved by varying the space between the words for each line. Care must be taken, though, not to space words too close together or "tight", making it difficult to see the distinctions between words, or too far apart or "loose", which looks poor and can lead the reader to skip lines. There is often enough flexibility in spacing that a typesetter can decide whether the last word of a line is kept, perhaps making it slightly tight, or moved down to the next line, making the current line loose. Words can also sometimes be hyphenated, as a compromise between the two options.

Until very recently, typesetting was a craft handed down from master to apprentices in publishing shops. As such, there is very little in the way of published material describing how typesetters came to these decisions. A review of the available literature is provided by Knuth and Plass (1981). Often the rules that were in place were vague, contradictory, or simply not followed.

With the introduction of computers came automatic typesetting systems. Barnett (1965) describes the standard computer typesetting practice at this time. Systems such as the PC6 family were able to accept marked-up source material and control dedicated photo-typesetting machines.

The algorithm employed for choosing line and page breaks is the "first-fit" method, described by Rich and Stone (1965). Words are set from left to right using the tightest allowable spacing until one does not fit, which is then moved to the next line and the current line justified to fit the margins. This method is still employed by the majority of word processors and web browsers today, though it has some serious drawbacks. Lines can be set extremely loose, since the algorithm never "looks ahead" to see if the break being considered is suitable or not.

Barnett notes that this is a serious problem, giving rise to rivers and poorly-hyphenated pages. He suggests only that a faster proofing process would enable the human operator to locate and make manual adjustments to such sections before going to print (this was before the wide adoption of raster displays for computers).

Justus (1972) considers the problem and proposes a set of rules that should be followed when typesetting. For example, "Loose lines are to be avoided." Unfortunately, the algorithm he proposes is no better than first-fit with hyphenation. In addition to the line-breaking rules, Justus considers other typographical problems such as column and page breaking, and introduces to the computer science community the terminology associated with the domain. This terminology has since been refined, for example by Rubinstein (1988), and we will use the following definitions in this paper:

Widow The final line of a paragraph appearing by itself at the top of a page or column.

Orphan The first line of a paragraph appearing by itself at the bottom of a page or column.

Cub The final word of a paragraph appearing on a line by itself.

Stack Two consecutive lines that begin with the same word, or some sequence that is the same (for example, following a hyphenation).

River Whitespace between words that forms a contiguous vertical space down the page.

These are features that can appear in set type if not attended to, and are generally frowned upon by typographers. For example, the appearance of an orphan requires the reader to turn the page during the opening phrase of a paragraph, losing its impact. Stacks can cause the reader to reread the same line several times, as the eye mis-tracks due to the duplicate cue. Rivers guide the reader's eye down the page, instead of across it, and look visually unappealing.

The severity of one of these features is of course subjective, and some are not considered detrimental at all. For example, not all publishing houses will avoid typesetting paragraphs with cubs. The other aspects typically taken into consideration are:

- The number of words that are broken with hyphenations,
- Consecutive lines that are hyphenated,
- A loose line followed by a tight line, or vice-versa.

For example, the Chicago Manual of Style (Grossman, 1993) has this to say about hyphenation:

> An abundance of hyphenated lines on one printed page is usually frowned on. Moreover, no more than three succeeding lines should be allowed to end in hyphens.

Knuth and Plass (1981) give details of an algorithm that optimises line breaks within a paragraph according to a "badness" function. This function is essentially a weighted sum of undesired characteristics, or "penalties," found in a line of type. Our algorithm is based heavily on this one, and so is described in detail in the next section.

Samet (1982) devises two heuristics for measuring the quality of a given paragraph. The first looks at the looseness of adjacent lines and sums the differences over the paragraph. While this succeeds in lowering the chance of a loose line followed by a tight line, the result is not aesthetically pleasing, and, when combined with an optimising algorithm, results in text that differs wildly in its spacing. The second heuristic simply measures the variance of looseness, and is much more effective, according to the author. We use a similar function in our own contribution.

More recently, researchers have turned their attention to the problem of pagination: determining where to break paragraphs of text between pages. This is closely related to the line-breaking problem: Knuth and Plass experimented with using essentially the same algorithm to solve both problems in separate passes (Plass and Knuth, 1982), though this functionality never appeared in TeX.

Pagination can be NP-complete, as shown by Plass (1981). Removal of widows and orphans, placement of floats (figures that lie outside the flow of text) and footnotes make the problem decidedly non-linear.

Bruggemann-Klein et al. (1995) use dynamic programming to reduce the number of page turns between the reference to a figure and the position of the float. The optimisation process requires the user to specify a minimum page fill, and cannot optimise the two objectives (hand turns and page fill) simultaneously. Their method has also been extended to handle footnotes (Brüggemann-Klein et al., 2003).

Jacobs et al. (2003) investigate the use of page templates that can be selected and rearranged by an optimising algorithm in order to reduce the number of page turns and whitespace. Unlike most approaches, theirs is designed to be completely automated, and able to reformat text for appropriate display on a variety of devices. Similar to Knuth and Plass, they use dynamic programming to optimise a "goodness" function related to the number of page turns as well as some unspecified aesthetic constraints.

In recent years we have also seen an emerging interest in automated layout engines that are not flow-based; that is, they are not governed by a main body of text that runs through the document as in a book or report. Rather, they are designed to create visually appealing layouts for pamphlets, posters and the like. Lumley et al. (2006) give a high-level overview of the various components of such a program, and discuss one approach based on hierarchical placement of containers.

Purvis (2002) and Purvis et al. (2003) present some functions that assess the aesthetic quality of a layout based on its symmetry and coverage, and use these as the fitness functions to a multiple objective genetic algorithm. They use a weighted combination of objectives, however, so do not deal with an optimal solution set as we do.

2.2 Knuth and Plass line breaking

As Knuth and Plass's method forms the basis for the work presented here, a more detailed description of its workings is warranted. The algorithm is most known for its implementation in TEX, which is extensively documented (Knuth, 1979).

A TEX source document is really a set of commands invoking a simple macro language. The result of these commands is the construction of a list of *boxes*, *glue* and *kerfs*. A box is anything that can be typeset: typically a word, but it could also be part of a mathematical formula or an inline graphic. To the layout algorithm it is unimportant how the box is rendered; all it needs to know are the width and height of the box.

Glue is the space between boxes, and has several attributes that govern its *stretchability.* The space between words typically has a nominal width as well as a minimum and maximum size that it can be stretched to in order to achieve justification.

A kerf lets the algorithm know that a line-break is possible at the given place, and is associated with a numerical *penalty*. The aim of the algorithm is to find a set of kerfs that break the lines in such a way that the sum of the penalties is minimal. Additional functions are added to the penalties to calculate the looseness and difference in looseness as appropriate.

Figure 1 shows a conceptual view of how a set of kerfs is discovered for the opening sentence of Charles Dickens' *Great Expectations*. Each node in this graph represents a line of text that can be typeset, and each edge is a decision to break the line at a certain place. Some nodes have only a single outgoing edge, meaning there is only a single way to break the line while keeping within the tolerances of the glue. Other nodes have several options, and the penalty associated with each edge should be taken into consideration.

This problem belongs to a family of shortest path problems. The optimal solution will be the path from the top of the graph to the bottom that has the minimum sum of edge weights.

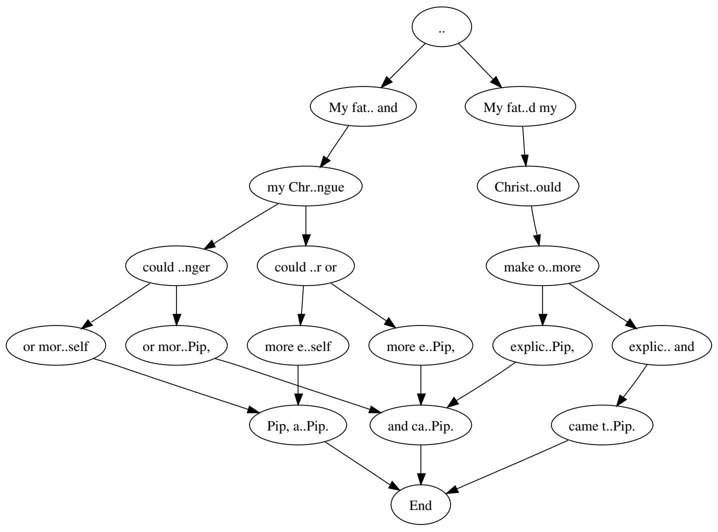


Figure 1: Conceptual view of possible line breaks for a paragraph of text.

2.3 Dynamic Programming

Shortest path problems have been well studied, and algorithms for solving them tend to fall into two categories: those which find the shortest path from the starting node using at most k nodes, and those which find the kth closest node to the starting node (Corley and Moon, 1985). When applied to routing problems, the textbook approach is given by Dijkstra (1959), an example of the second type.

Both approaches can be described by recursive functional equations, which can be computed efficiently using dynamic programming (Bellman, 1957). For example, for a directed graph of nodes $N = (1, \dots, n)$ and edges $A \subset N \times N$ where each edge from node i to j has a cost $a_{ij} \in A$, the least cost path has cost $\mathrm{C}(n)$, where

$$\begin{aligned} \mathrm{C}(1) &= 0 \\ \mathrm{C}(j) &= \min_{i<j}\{\mathrm{C}(j-1) + a_{ij} : j \in N, a_{ij} \in A\}. \end{aligned}$$

The requirement that a least-cost path to node j must begin with the least-cost path to node $j-1$ is satisfied by Bellman's principle of optimality. It is necessary only to compute $\mathrm{C}(a_{ij} : i, j \in N, i < j)$ (Held and Karp, 1965).

2.3.1 Multiple objective shortest path

Many real-life problems cannot be expressed with a single objective function. The classic example within the shortest path literature is of a road network in which each road is associated with both a time and a cost. Typically these functions are in opposition, and represent some trade-off. For example, a road having low cost may take a long time to traverse, and conversely, a road taking little time to traverse may have a high toll price.

In these problems we often wish to find the Pareto optimal set of solutions (also referred to in the literature as the efficient or non-dominated set). We say that a solution p^m (a vector of size m) dominates q^m if

$$p \prec q \equiv p_i \leq q_i, i = 1, \dots, m;\ \exists j \in \{1, \dots, m\}\ p_j < q_j.$$

The Pareto optimal set $E \subset P$ is the set such that

$$\begin{aligned} \forall p, q \in E \qquad & p \not\prec q \\ \forall p \in E, \forall q \in P \qquad & p \prec q. \end{aligned}$$

It should be noted that the problem is intractable in the worst-case. Hansen (1980) gives a proof of this by constructing a graph of n nodes in which E grows exponentially with n, showing that simply enumerating the solution is intractable. A pseudo-polynomial algorithm for solving bicriterion paths is given, which is suitable for the majority of problems which do not fall into the worst case.

The bicriterion shortest path problem was studied by Climaco and Martins (1982), who use Dijkstra's algorithm to enumerate all paths in non-decreasing order of one of the objectives. Once the extreme non-dominated paths have been found, the second objective can be optimised with respect to the candidates and the complete Pareto set is returned. The authors report very few efficient paths for their test problems—this result, however, is atypical, as commented on by Henig (1986).

Daellenbach and Kluyver (1980) describe an algorithm for the more general case of n-objectives, however they present results only for a bicriterion problem. The authors generate all paths and evaluate Pareto optimal set as the last step. They make use of MINSUM and MINMAX functions for further reducing the result set.

When one or more the objective functions are specialised from MINSUM, as given above, to MAXMIN or MINMAX, a polynomial algorithm exists by repeatedly applying Dijkstra's algorithm (Hansen, 1980). Martins (1984b) generalises the bicriterion case by allowing additional objective functions such as MAXMIN and MINRATIO for generating a minimal complete set of non-dominated paths. A second algorithm generates the entire non-dominated set by incrementally deleting paths from a complete graph.

The first use of dynamic programming to solve the n-objective shortest route problem is by Hartley (1985). Unfortunately the routine used examines all possible paths and is not efficient. The author introduces a weighting scheme to reduce computation at the expense of losing some non-dominated paths.

The dynamic programming approach presented in this paper is similar to the work of Henig (1986), though the author considers only a bicriterion shortest path problem. The recursive functional equation is similar to the one shown above. After evaluation of each $C(j)$, the dominated solutions within the generated set are removed. The approach will be described in more detail in the next section. Henig also presents a statistical analysis of the problem, and finds that in a network of n stages and M nodes at each stage, of the n^M total paths, $M \log n$ of those paths will be non-dominated. The expected complexity order of finding the non-dominated paths in this network is $O(M^2 n^2 \log^2 n)$.

Martins (1984a) independently presents a similar algorithm, using "labels" of nodes to convey the set of non-dominated paths. There is a correspondence between Martins' labels and Knuth's active nodes which will become apparent in the next section. A second, possibly more efficient algorithm based on the lexicographic ordering of a sub-tree is also presented, though the author admits to not having implemented it successfully.

Corley and Moon (1985) present results using an alternative recurrence relation in which the set of paths containing k or fewer nodes is solved for $k = 1, \ldots, n$. They remark that the worst-case complexity is of exponential order for the number of nodes, and that their approach is not suitable in the general case. An example of this algorithm is presented by Corley (1985).

2.3.2 Multiple objective dynamic programming applied to other problems

The use of dynamic programming for optimisation of vector values was first presented by Brown and Strauch (1965), who show that the Pareto optimal set can be found using lattice-valued functional recursion. The resulting formalism is quite general and has not been used in practice, to our knowledge.

The multiple criteria, multiple stage decision problem has been studied in detail. This is often the simplest application of dynamic programming: each stage of a process is evaluated to find the optimal configuration with respect to the state's inputs and/or outputs. The sufficient conditions for optimising the process over serial and parallel stage composition are given by Mitten (1964). Villarreal and Karwan (1982) and Yu and Seiford (1989) give algorithms for finding the Pareto optimal set, and examine the sufficient and necessary conditions of the state transition operators for which they are valid. Henig (1983) models the problem as a Markov decision process, finding a set of optimal policies that give the Pareto optimal set.

Dynamic programming has been applied to the multiple criteria, multiple objective knapsack problem. The problem is a combinatorial one of selecting elements of a set to maximise a vector-valued function according to some constraints. Five state representations of the problem, including the more difficult time-dependent case, are evaluated by Klamroth and Wiecek (2000). Villarreal and Karwan (1981) use a hybrid of functional recursion and traditional integer branch-and-bound methods to find the Pareto optimal set.

Getachew et al. (2000) examine the multiple objective shortest path problem with time-dependent objective functions. They use multiple iterations of a backward dynamic programming phase using function limits as path costs followed by a forward evaluation phase using the actual path costs. The technique, unlike many others that try to solve the same problem, reduces to a single pass when functions are not time-dependent, and does not approximate the solution.

Algorithms exist that operate on continuous multi-variable functions. Li and Haimes (1987) attempt to find the derivative of the vector objective functions and use linear interpolation to predict the optimal frontier. Liao and Li (2002) are able to reduce the dimensionality of the problem and limit the search space by approximating the bounds of the functions with a quadratic function. These techniques are not readily applicable to our problem, which is quite discontinuous.

2.3.3 Approximations to multiple objective optimisation

The multiple objective shortest path problem is NP-complete (Garey and Johnson, 1979), so it is desirable to have efficient means of approximating a solution. A common approach is to obtain an ϵ-approximate Pareto optimal set. A vector is able to dominate another if the relative error is less than some "accuracy" constant ϵ. For example, Hansen (1980) and Warburton (1987) give algorithms in which the complexity scales by $\frac{1}{\epsilon}$. Safer and Orlin (1992) give the necessary and sufficient conditions for a fast approximation scheme and apply it to the knapsack and shortest route problems.

Wakuta (2001) reformulates the shortest path problem into a multiple objective Markov decision model. A set of locally efficient policies is discovered (Sancho, 1985), followed by a second phase in which these policies are combined to determine a stochastically likely non-dominated set.

A large body of optimisation algorithms use evolutionary techniques, such as genetic algorithms and particle swarm optimisation. Multiple objective optimisation is no exception, and we investigated these techniques for some time before settling on dynamic programming for the algorithm described in the next section. An excellent overview and introduction to the current techniques used in evolutionary computing is given by Deb (2001).

2.3.4 Preference order multiple objective optimisation

The presented methods so far have all dealt with real-valued objective functions. A completely different approach, beginning with work by Mitten (1974) and Sobel (1975), is to replace these functions with a binary preference. In this case, rather than returning the entire Pareto optimal set, an optimal solution is returned based on a preference ordering function. Mitten describes how this function can be an interactive process with the decision maker and applies it to the multiple stage optimisation problem. Henig (1994) extends this work to the bicriterion shortest path problem, and presents several options for how interactivity with the decision maker can proceed.

3 Multiple objective line breaking

We reformulate the "optimal-fit" of a paragraph as defined by Knuth and Plass (1981) to incorporate multiple objectives. Rather than minimising a single "badness" value, our algorithm minimises an arbitrary number of objective functions, returning those paragraphs that cannot be improved along one objective without reducing the quality along another objective. This set is called the Pareto optimal set.

The outline of the following sections is as follows. We begin by describing the representation of text and paragraphs in our algorithm, before presenting the algorithm itself. We consider the necessary conditions on an objective function, and define some functions typical to the typesetting practice. A new method for determining hyphenation points completes the algorithm, and finally our evaluation methods are given.

3.1 Data representation

The input to the algorithm is a string of character data. In production, the character data would be marked up with character-level and paragraph-level attributes such as typeface, colour, leading and so on. These details, while complicating the implementation of the algorithm a little, do not impact on its general nature, and so for the remainder of this paper we will concern ourselves only with unformatted character data. The model described here is similar to that by Plass and Knuth (1982), but contains necessary alterations to fit with our algorithm.

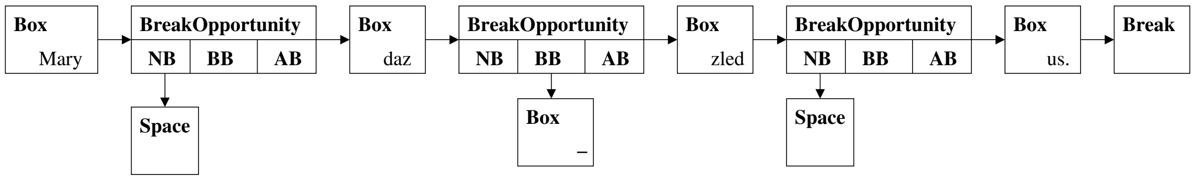
The character data is converted into a list of items. Each item can be one of three types: either it is a Box, a Space, a Break or a BreakOpportunity. A Box represents any character or pictorial data. For example, it can represent a some characters from a word, a punctuation mark, a mathematical formula or an in-line graphic. The details of the contents of the box—how it is rendered—are irrelevant to the line breaking algorithm, which needs to know only its Width, Ascent and Descent. One additional attribute, Hyphenateable, exists and will be described later.

A Space is, predictably, the space between two words when typeset. In a justified paragraph this space is not of a fixed width, so we assign it three attributes: NominalWidth, MinWidth and MaxWidth. The NominalWidth of a Space is the width it will be typeset in a ragged-left or ragged-right paragraph, and is the ideal spacing for a justified paragraph. The MinWidth and MaxWidth attributes give the minimum and maximum allowable width of the space in a justified paragraph. Some typical values for these in a technical document are 0.3, 0.2 and 0.5 em, respectively.

A Break instance signifies a forced line break. These occur, for example, at the end of every paragraph.

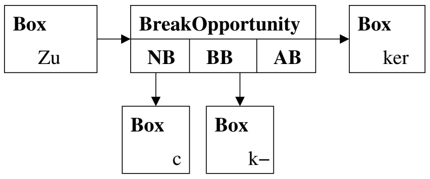
A BreakOpportunity is a composite item that lets the line breaking algorithm know that a break is permissible at that point. A BreakOpportunity has three attributes: NoBreak, BeforeBreak and AfterBreak. If the line breaking procedure determines that no break will occur at the point, the contents of NoBreak is inserted in place of the BreakOpportunity (the contents of NoBreak is a list of Box and Space instances). In the other case, where a Break takes the place of the BreakOpportunity, the contents of BeforeBreak and AfterBreak are inserted just before and after the break, respectively.

An example illustrates this representation clearly. For the paragraph “Mary dazzled us.”, we have the sequence:



Note that the word “dazzled” can be hyphenated, so the BreakOpportunity contains a Box for the hyphen in the BeforeBreak attribute. All the spaces between words represent potential breakpoints, and a Break is inserted at the end of the paragraph.

It should be clear that this method of representation is flexible enough to support alternate spellings over a line break. For example, the German word, “Zucker,” is hyphenated, “Zuk-ker”. This is represented in our model as:



A similar approach can be used to deal with breaking within ligatures.

The process of converting a string of character data into a list of items is straightforward. Words are converted to Box items separated by the BreakOpportunity construct containing a Space as shown above. The resulting list is used to determine the optimal sets of line breaks.

3.2 Pareto optimal set determination

We will now describe our innovative algorithm for determining the Pareto optimal set of line breaks. The list of items is scanned sequentially, one item at a time. An *active list* of potential solutions is maintained, each *active item* storing the bottom-most line being typeset. Where a breakpoint is possible in an active item, a new active item is created representing the next line of text. This new active item contains a reference back to the previous item, so that a complete typeset paragraph can be determined by following the references back to the root item.

Figure 3.2 shows an example of the active items that might be found while setting a paragraph. The root item shows the beginning of the paragraph, and every child node is an acceptable line break. In this example, two objective functions are used. The pair of numbers in each item show the evaluated fitness for that item for each of the objective functions. Lower numbers indicate higher fitness.

The two items in the dashed box have different heritage but break at the same point in the text. This gives an opportunity for optimisation, as only non-dominated solutions need to be expanded further. In this example, the left-hand item is dominated, so it is removed from the active list and not explored.

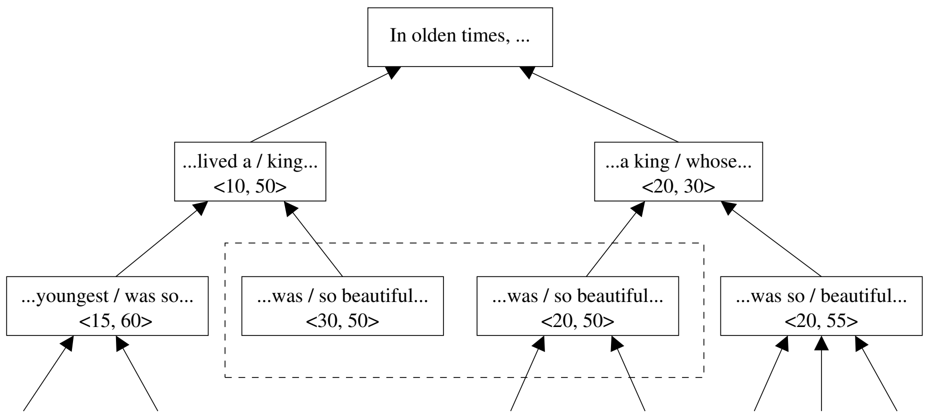


Figure 2: Active list items for a hypothetical paragraph and two objective functions.

The detail of the procedure is shown in Algorithm 1. The active list A is initialised with a sentinel. Other active items are a pair $\langle \text{previous}, \text{break} \rangle$, where previous links to the active item that precedes it, and break is the final Break or BreakOpportunity found so far. Each item b in the list items is examined in turn, and is either a Break, BreakOpportunity, Space or Box. For spaces and boxes, the minimum and maximum widths are simply added to those of each active item, as shown in Algorithm 2 (an active item begins with a minimum and maximum width of 0).

For a BreakOpportunity, each active item is checked to see if it can break at its current point. If it can, a new active item a^* is added to A^* which points back to the a which prompted its creation. Suitable attention is paid to the BeforeBreak, AfterBreak and NoBreakBox attributes to ensure the correct boxes are added in both the break and no-break cases.

When a forced Break is encountered, a new active item a^* is added for all existing active items as before, however there is no need for the feasibility check (in justified text, the last line of a paragraph is typically set ragged-right). The previous active item list A is cleared, as a forced break implies that continuing any of these lines is not permitted.

All items in the new active list A^* that are non-dominated within that list are added to the active list. The active list A is then removed of any items that have become infeasible by requiring too much width.

At the conclusion of the loop, the active list A holds the non-dominated solution set. A proof of this is straightforward. The final item in a paragraph is a Break, so in the last iteration A will be cleared and set to A^*, which is made removed of dominated solutions before exiting the loop.

3.3 Objective functions

The purpose of an objective function is to evaluate a set of breakpoints and return a numerical "score" indicating its quality. Because our algorithm supports multiple objectives, any number of objective functions can be supplied by the user. The optimal paragraphs returned represent the trade-off in each objective.

An objective function must be evaluable for an incomplete set of breakpoints. Without this condition, elimination of dominated solutions could only occur after generating all possible solutions—an intractable approach.

There is no need for an objective function to be non-decreasing, as members of the active list can

Algorithm 1 Find Pareto optimal set of line breaks

```
1: $A \leftarrow \{\langle \text{nil}, \text{nil} \rangle\}$
2: $i \leftarrow 0$
3: while $i < |\text{items}|$ do
4:     $A^* \leftarrow \{\}$
5:     $b \leftarrow \text{items}_i$
6:     if $b$ is BreakOpportunity then
7:         for $a \in A$ do
8:             $a_{\text{min}} \leftarrow a$.MinWidth $+$ $b$.BeforeBreak.MinWidth
9:             $a_{\text{max}} \leftarrow a$.MaxWidth $+$ $b$.BeforeBreak.MaxWidth
10:            if $a_{\text{min}} \leq w \leq a_{\text{max}}$ then
11:                $a^* \leftarrow \langle a, i \rangle$
12:                ADDITEM($a^*$, $b$.AfterBreak)
13:                Add $a^*$ to $A^*$
14:            end if
15:            ADDITEM($a$, $b$.NoBreakBox)
16:        end for
17:    else if $b$ is Break then
18:        for $a \in A$ do
19:            if $a$.MinWidth $\leq w$ then
20:                $a^* \leftarrow \langle a, i \rangle$
21:                Add $a^*$ to $A^*$
22:            end if
23:        end for
24:        $A \leftarrow \{\}$
25:    else if $b$ is Box or Space then
26:        for $a \in A$ do
27:            ADDITEM($a, b$)
28:        end for
29:    end if
30:    $A^* \leftarrow \{a \in A^* : \nexists b \in A^* \text{ s.t. } b \prec a\}$
31:    $A \leftarrow \{a \in A : a.\text{MinWidth} \leq w\} \cup A^*$
32:    $i \leftarrow i + 1$
33: end while
```

Algorithm 2 Procedure for adding a box or space to an active item

```
1: procedure ADDITEM($a$, $b$)
2:     $a$.MinWidth $\leftarrow$ $a$.MinWidth $+$ $b$.MinWidth
3:     $a$.MaxWidth $\leftarrow$ $a$.MaxWidth $+$ $b$.MaxWidth
4: end procedure
```

be dominated only by other items that break at the same position, and hence have the same "future." This is equivalent to allowing negative edge weights in a shortest path problem.

In the prototype application the objective functions are implemented as concrete classes, enabling them to be easily selected and enabled at runtime by the user.

Several objective functions have been demonstrated as an example of the versatility of this approach. Each objective function corresponds to one of the typographic concerns listed in the introduction.

3.3.1 Average word spacing

Tight spacing is usually desired, so this objective gives lower (better) scores to paragraphs with a tighter mean spacing. It is calculated as

$$\mathrm{L}(l) = w - \sum_{\text{box}\in l} \text{box.MinWidth}$$

$$\mu\mathrm{Looseness} = \frac{\sum_{l\in\text{lines}} \mathrm{L}(l)}{|\text{lines}|}.$$

3.3.2 Variance in word spacing

Lines should have consistent spacing. Using just the average word spacing as an objective can result in paragraphs with very tight spacing on most lines but the occasional loosely spaced line. This objective function calculates the variance in word spacing between lines, returning a lower score for paragraphs in which the word spacing is consistent.

$$\sigma^2\mathrm{Looseness} = \frac{|\text{lines}| \sum_{l\in\text{lines}} \mathrm{L}(l)^2 - (\sum_{l\in\text{lines}} \mathrm{L}(l))^2}{|\text{lines}|(|\text{lines}| - 1)}$$

3.3.3 Number of hyphenations

Hyphenations that break words between two lines can be reduced by including an objective function that simply counts them. As a lower number of hyphenations results in a lower score, a set of line-breaks that hyphenates less will be selected in preference to one that hyphenates more often.

$$\Sigma\mathrm{Hyphen} = \text{Number of hyphenations.}$$

Variations of this are possible; for example, counting the number of hyphenations that appear on consecutive lines, or three consecutive lines, and so on.

3.3.4 Number of stacks, cubs, widows and orphans

Similarly, undesirable features such as stacks, cubs, widows and orphans are easily counted and implemented as separate objective functions. In the prototype implementation we have not implemented the required functionality for pagination, so objective functions for widows and orphans are not presented.

$$\begin{aligned}\Sigma\mathrm{Stack} &= \text{Number of stacks.}\\ \Sigma\mathrm{Cub} &= \text{Number of cubs.}\end{aligned}$$

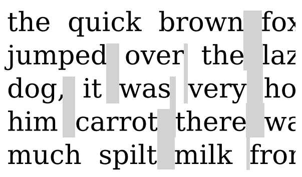


Figure 3: The area covered by rivers, as calculated by the objective function, is shown in grey.

3.3.5 Whitespace rivers

The objective function for vertical rivers of whitespace returns the two dimensional page-space area that is covered by a river. For the purposes of the function, a river is the intersection a space makes with a space in the preceding line. This is easily calculated by finding the positions of each space in a line after laying it out using the appropriate font and justification characteristics. Figure 3.3.5 shows this conceptually.

$$\Sigma\mathrm{River} = \sum \text{Area of whitespace comprising a river.}$$

3.3.6 Final line justification

This objective function is described to demonstrate the variety of effects that can be achieved using the approach described in this paper, that would otherwise be difficult or impossible using traditional techniques.

Where text is wrapped around images or figures, as in a magazine, it is desirable to have every line fully justified—including the final line of a paragraph, which is typically set ragged-right to compensate for the lack of any more words. Knuth and Plass (1981) gives an example of this extracted from *Time* magazine. In order to avoid setting the final line with extremely large word spacing, it is necessary to consider the layout of the entire paragraph with the goal of setting the final word at the right margin.

We can devise an objective function that returns a low score for layouts that can be set with justified final lines by returning the number of paragraphs that *cannot* be set within tolerances with justified final lines.

$$\Sigma\mathrm{Unjustified} = \sum \text{Paragraphs that cannot be set fully justified.}$$

3.4 Hyphenation

There are two actions involved in hyphenating text: determining the possible places of hyphenation, and deciding which, if any, hyphenation to use when breaking a line.

Like TEX, we use the Liang hyphenation algorithm (Liang, 1983) to determine valid hyphenation points within a word. This algorithm suits automatic typesetting systems well as it does not require a complete dictionary of hyphenations, allowing unknown words to be hyphenated according to the language's conventions without excessive human intervention.

However, we cannot employ TEX's method of determining when to hyphenate. TEXuses a three pass system in which hyphenation is attempted only if the first pass fails to find a set of line breaks with sufficiently low badness value. In the second pass, hyphenation of words is permitted at likely places based on the position of feasible breakpoints from the previous pass. If the optimal layout still does not meet the badness requirement, a third pass is employed in which every possible hyphenation is attempted.

This approach works well for TEXas it avoids opening any of the search space that requires a hyphenation unless strictly necessary. If hyphenation were attempted for every word in a paragraph the breadth of search would increase twofold or more.

The three pass system cannot be used with our multiple objective algorithm as it would require a vector-valued minimum to trigger each pass. This would be difficult and unnatural to specify. Additionally, the use of such a method would not permit objective functions based on the hyphenation of the text to be defined, limiting the algorithm's usefulness.

We present an efficient method for finding only the feasible hyphenation points in a paragraph in a single pass. The method is suitable for use with our multiple objective algorithm, and is general enough to be used with TEXalso (though this has not been implemented).

The modification to Algorithm 1 is shown in Algorithm 3, and replaces lines 26–28 in the original algorithm.

Algorithm 3 Alter the list of items to allow for hyphenations at feasible break points.

```
1: if $\exists a \in A$ s.t. $a$.MinWidth $\leq w \leq a$.MaxWidth $+ b$.MaxWidth then
2:     Replace items$_i$ with hyphenation of $b$
3:     $i \leftarrow i - 1$
4: else
5:     for $a \in A$ do
6:         AddItem($a, b$)
7:     end for
8: end if
```

As each Box or Space is added, it is checked with every active item for an overlap with the margin. If any active item can be broken, the Box is hyphenated. Recall that hyphenation generates several Boxes and BreakOpportunities, which are inserted into the item list in place of the current Box.

Boxes are marked as "hyphenatable" initially, and not-hyphenatable if they were created as the result of a hyphenation, to prevent infinite recursion.

3.5 Handling failures

It is possible to present the program with a paragraph that cannot be typeset to the given constraints. For example, a long word that cannot be hyphenated appearing in a narrow column can make it impossible to set the text within the required spacing.

The algorithm as presented so far does not handle this case gracefully, as a line that cannot be set will empty the active list, presenting no options for the user. As the user is given no clue as to where the typesetting failed, it is a frustrating exercise to try to resolve such failures.

Algorithm 4 is to be inserted into Algorithm 1 immediately after lines 16 and 23. It allows typesetting to continue even when such a situation is encountered. If the active list would be reduced to nothing, each of the active items is overset (it extends outside the margin). New active items are then created, beginning a new line, and the user is warned of the failure.

While this creates unacceptable output for most purposes, it makes it quite clear to the user which line or lines are problematic, allowing them to enter an alternative hyphenation, specify looser spacing or rephrase the paragraph.

Algorithm 4 Overset lines as a failure mechanism when the active list would be empty.

```
1: if $\nexists a \in A$ s.t. $a$.MinWidth $\leq w$ and $|A^*| = 0$ then
2:     Warn user about overset line.
3:     for $a \in A$ do
4:         $a^* \leftarrow \langle a, i \rangle$
5:         Add $a^*$ to $A^*$
6:     end for
7: end if
```

3.6 Culling duplicate solutions

In practice, we have found that when a small number of objective functions are used, many solutions are returned which are non-dominated because the results of the functions are equal, however they typeset slightly differently. For example, combining $\mu\mathrm{Looseness}$ with $\Sigma\mathrm{Stack}$ often results in paragraphs identical but for a single word placement; if no stacks are present to differentiate the solutions and the mean looseness remains the same, many solutions are returned.

This caused problems during the experiments described below as a large number of intermediate solutions dramatically increases processing time, often making the problem intractable.

If we make the assumption that an end-user is not going to prefer one solution over another if, objectively, the solutions are the same, we can cull these duplicates early. The assumption is reasonable as our premise is that the user, as a domain expert, is looking at the trade-off between solutions, not for characteristics (aesthetic or otherwise) that are not embodied by the objective functions.

An arbitrary solution is selected between any two objectively identical ones after line 30 of Algorithm 1. For example, in the prototype implementation, the solution culled is the one with the numerically higher memory address.

3.7 Experimental designs

In order to test our research questions, we need to measure several aspects of the algorithm. The results of these experiments are presented in the next section.

The text used in most of these experiments is an extract of The Frog King (Grimm and Grimm, 1857), the same text used by Knuth and Plass in their publications relating to TEX. This text consists of relatively long paragraphs comprised of both long and short English words.

We will typeset the text into three different page widths: 19.5cm, 9.4cm and 6cm, corresponding to 1, 2 and 3 column layouts on A4 paper with typical margins. In every case the font used is Computer Modern at 10 point (Knuth et al., 2001), converted to PostScript Type 1 format from the original METAFONT description. Again, we are attempting to emulate the conditions under which Knuth and Plass performed their experiments with TEX, as it is the most appropriate program to use for comparison.

To ensure our objective functions are reasonable and consistent with the typographical features they aim to improve, we isolate each objective function with just one other and look at the results holistically and subjectively. For each objective function it should be clear that a reduction in score corresponds to a noticeable positive effect on the typeset paragraph.

Having shown that the objective functions have a useful real-world influence, we combine several of them at a time with varying column sizes to see the number and distribution of results returned. A low number of results could indicate that the objective functions are performing similar functions,

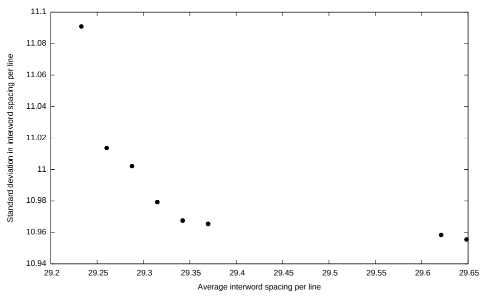


Figure 4: Example of a two dimensional result set.

and a good trade-off is not being found. On the other hand, too many similar results could be both computationally expensive and unwieldy for a user to manage.

We compare these same results with those of TEX, using its default settings. We can measure the performance of TEX along the very same objective functions we use for optimisation. This will give an indication of how well TEX's badness function approximates our multiple objective approach, and whether the added complexity of using multiple objectives is worthwhile for enhancing typography.

Finally we will look at the performance of our algorithm. It should be clear from the preceding paragraphs that the running time will be almost directly proportional to the number of items in the active list at a given time. For varying lengths of document and column sizes, we look at the maximum and average size of the active list, as well as the wall-clock running time.

4 Results

4.1 Reading a multiple objective result set

Where we need to show and compare several optimal solutions with respect to two or more objective functions, we will use the n-dimensional plot used extensively in the multiple objective evolutionary algorithm literature (Deb, 2001). For two objective functions, the plot simply maps each objective function on its own axis, as in Figure 4

Note that for every point in the solution, there is no other point which is closer to the origin in both axes. This is the key feature of a Pareto optimal set, and is characterised by the negative slope indicating the trade-off for the two functions.

Where we are plotting results for three or more solutions, it is necessary to plot each pair of objective functions separately, as projections of higher dimensional plots are generally not readable. An example is shown in Figure 5.

The disadvantage of this approach is that it is not possible to see which point corresponds to which in each of the plots. It is not even possible to tell if the result set is Pareto optimal.

We will use a small rectangular outline around a point of interest in each of the plots to highlight the position of a particular result along each dimension. This is accompanied by the actual typeset text, to make comparisons between results more intuitive

4.2 Validity of objective functions

In the previous section we claimed that each of the objective functions addresses a classical typographical concern. We need to verify that lower scoring paragraphs actually present better with respect to

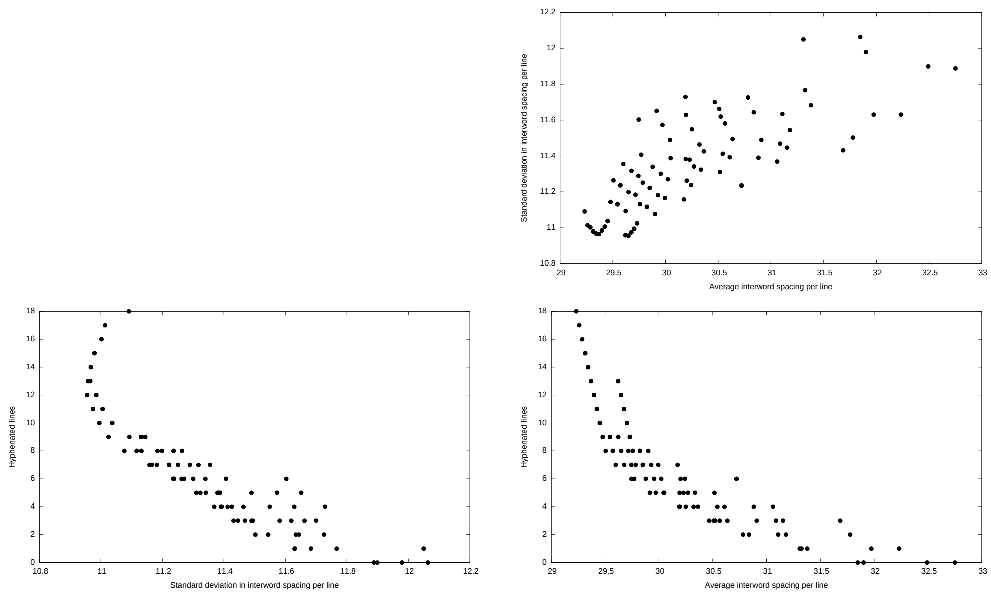


Figure 5: Example of a three dimensional result set.

the concern in question for each objective function. We do this by running the algorithm for the following pairs of objective functions:

- $\mu\mathrm{Looseness}$ and $\sigma^2\mathrm{Looseness}$
- $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{River}$
- $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$
- $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{Unjustified}$

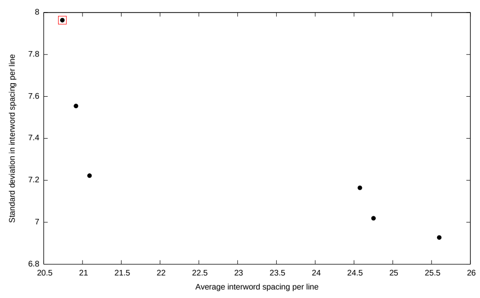
All of these tests are run with the duplication culling option enabled.

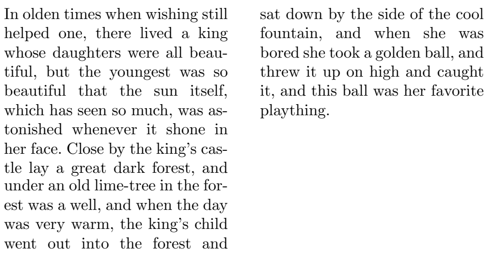
The two extreme results for $\mu\mathrm{Looseness}$ and $\sigma^2\mathrm{Looseness}$ are shown in Figure 6. For this test the maximum allowable spacing between words has been increased from 0.7 to 1.5 times the em width to make the differences in results quite clear. Figure 6a shows tight spacing on average, however it is inconsistent, and is comprised of a majority of very tight lines with several somewhat loose lines. Figure 6b shows a preferable case in which the overall spacing is slightly looser but more even.

Figure 7 shows two results for the $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{River}$ objective functions. The text is somewhat contrived to show the rivers quite clearly in the case where spacing is tight.

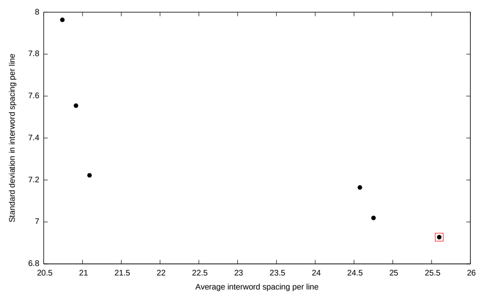
Two paragraphs are shown in Figure 8 that have been optimised using the $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$ objective functions. The first one, with 6 hyphenations, has looser spacing than the second, unhyphenated one.

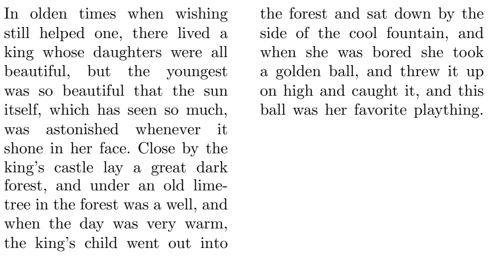
The results of using the $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{Unjustified}$ objective functions are shown in Figure 9. Three possible layouts are possible (only the two extrema are shown), corresponding to the tightest spacing when paragraphs are always, sometimes, or never justified at the final line.




In olden times when wishing still
helped one, there lived a king
whose daughters were all beau-
tiful, but the youngest was so
beautiful that the sun itself,
which has seen so much, was as-
tonished whenever it shone in
her face. Close by the king's cas-
tle lay a great dark forest, and
under an old lime-tree in the for-
est was a well, and when the day
was very warm, the king's child
went out into the forest and
sat down by the side of the cool
fountain, and when she was
bored she took a golden ball, and
threw it up on high and caught
it, and this ball was her favorite
plaything.

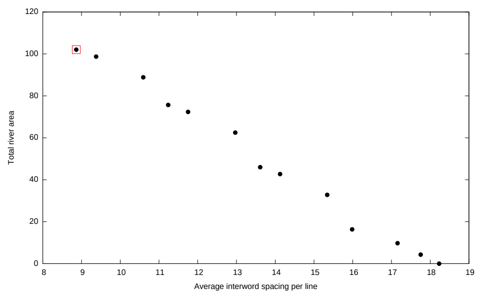
(a) μLooseness-optimal solution

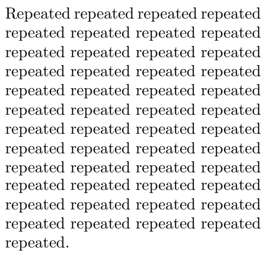



In olden times when wishing
still helped one, there lived a
king whose daughters were all
beautiful, but the youngest
was so beautiful that the sun
itself, which has seen so much,
was astonished whenever it
shone in her face. Close by the
king's castle lay a great dark
forest, and under an old lime-
tree in the forest was a well, and
when the day was very warm,
the king's child went out into
the forest and sat down by the
side of the cool fountain, and
when she was bored she took
a golden ball, and threw it up
on high and caught it, and this
ball was her favorite plaything.

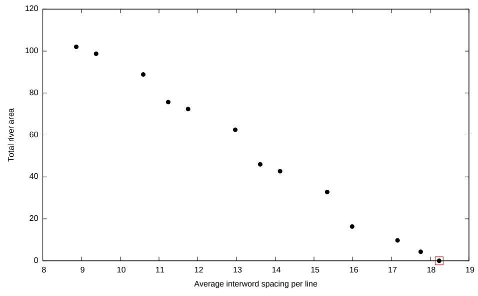
(b) σ^2Looseness-optimal solution

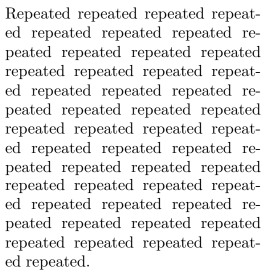
Figure 6: Extreme results for objective functions μLooseness and σ^2Looseness.




Repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated repeated repeated repeated
repeated.

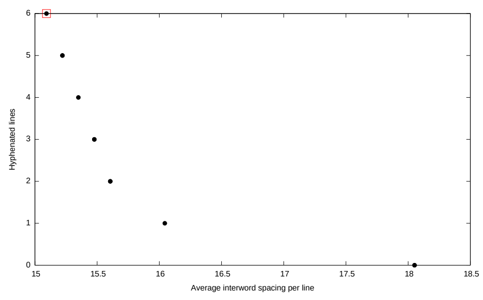
(a) μLooseness-optimal solution

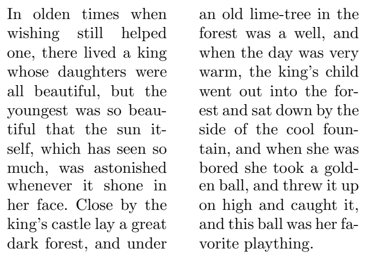



Repeated repeated repeated repeat-
ed repeated repeated repeated re-
peated repeated repeated repeated
repeated repeated repeated repeat-
ed repeated repeated repeated re-
peated repeated repeated repeated
repeated repeated repeated repeat-
ed repeated repeated repeated re-
peated repeated repeated repeated
repeated repeated repeated repeat-
ed repeated repeated repeated re-
peated repeated repeated repeated
repeated repeated repeated repeat-
ed repeated.

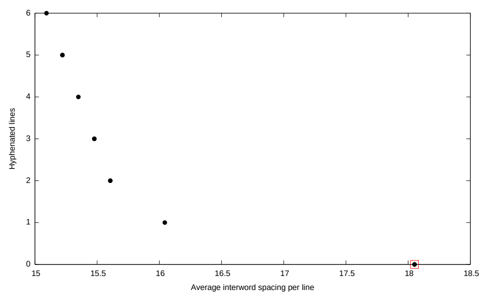
(b) ΣRiver-optimal solution

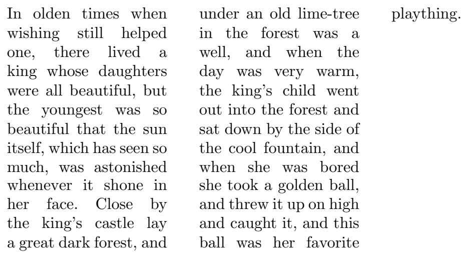
Figure 7: Extreme results for objective functions μLooseness and ΣRiver.




In olden times when
wishing still helped
one, there lived a king
whose daughters were
all beautiful, but the
youngest was so beau-
tiful that the sun it-
self, which has seen so
much, was astonished
whenever it shone in
her face. Close by the
king's castle lay a great
dark forest, and under
an old lime-tree in the
forest was a well, and
when the day was very
warm, the king's child
went out into the for-
est and sat down by the
side of the cool foun-
tain, and when she was
bored she took a gold-
en ball, and threw it up
on high and caught it,
and this ball was her fa-
vorite plaything.

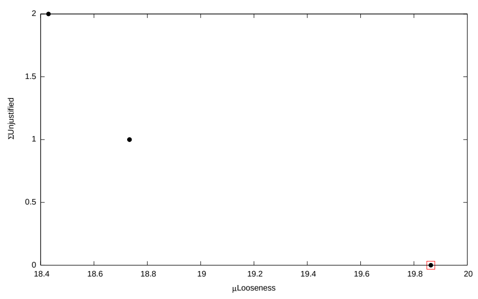
(a) μLooseness-optimal solution

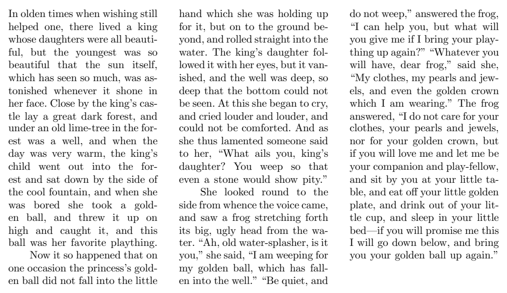



In olden times when
wishing still helped
one, there lived a
king whose daughters
were all beautiful, but
the youngest was so
beautiful that the sun
itself, which has seen so
much, was astonished
whenever it shone in
her face. Close by
the king's castle lay
a great dark forest, and
under an old lime-tree
in the forest was a
well, and when the
day was very warm,
the king's child went
out into the forest and
sat down by the side of
the cool fountain, and
when she was bored
she took a golden ball,
and threw it up on high
and caught it, and this
ball was her favorite
plaything.

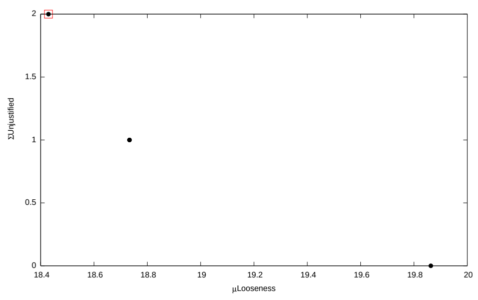
(b) ΣHyphen-optimal solution

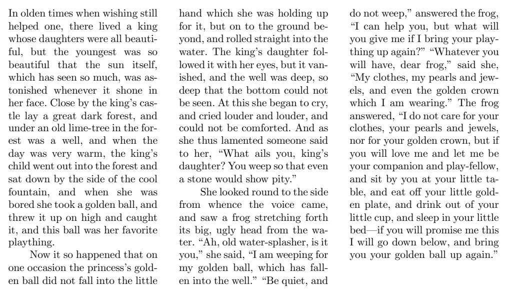
Figure 8: Extreme results for objective functions μLooseness and ΣHyphen.




In olden times when wishing still
helped one, there lived a king
whose daughters were all beauti-
ful, but the youngest was so
beautiful that the sun itself,
which has seen so much, was as-
tonished whenever it shone in
her face. Close by the king's cas-
tle lay a great dark forest, and
under an old lime-tree in the for-
est was a well, and when the
day was very warm, the king's
child went out into the for-
est and sat down by the side of
the cool fountain, and when she
was bored she took a gold-
en ball, and threw it up on
high and caught it, and this
ball was her favorite plaything.
Now it so happened that on
one occasion the princess's gold-
en ball did not fall into the little
hand which she was holding up
for it, but on to the ground be-
yond, and rolled straight into the
water. The king's daughter fol-
lowed it with her eyes, but it van-
ished, and the well was deep, so
deep that the bottom could not
be seen. At this she began to cry,
and cried louder and louder, and
could not be comforted. And as
she thus lamented someone said
to her, "What ails you, king's
daughter? You weep so that
even a stone would show pity."
She looked round to the
side from whence the voice came,
and saw a frog stretching forth
its big, ugly head from the wa-
ter. "Ah, old water-splasher, is it
you," she said, "I am weeping for
my golden ball, which has fall-
en into the well." "Be quiet, and
do not weep," answered the frog,
"I can help you, but what will
you give me if I bring your play-
thing up again?" "Whatever you
will have, dear frog," said she,
"My clothes, my pearls and jew-
els, and even the golden crown
which I am wearing." The frog
answered, "I do not care for your
clothes, your pearls and jewels,
nor for your golden crown, but
if you will love me and let me be
your companion and play-fellow,
and sit by you at your little ta-
ble, and eat off your little golden
plate, and drink out of your lit-
tle cup, and sleep in your little
bed—if you will promise me this
I will go down below, and bring
you your golden ball up again."

(a) μLooseness-optimal solution




In olden times when wishing still
helped one, there lived a king
whose daughters were all beauti-
ful, but the youngest was so
beautiful that the sun itself,
which has seen so much, was as-
tonished whenever it shone in
her face. Close by the king's cas-
tle lay a great dark forest, and
under an old lime-tree in the for-
est was a well, and when the
day was very warm, the king's
child went out into the forest and
sat down by the side of the cool
fountain, and when she was
bored she took a golden ball, and
threw it up on high and caught
it, and this ball was her favorite
plaything.
Now it so happened that on
one occasion the princess's gold-
en ball did not fall into the little
hand which she was holding up
for it, but on to the ground be-
yond, and rolled straight into the
water. The king's daughter fol-
lowed it with her eyes, but it van-
ished, and the well was deep, so
deep that the bottom could not
be seen. At this she began to cry,
and cried louder and louder, and
could not be comforted. And as
she thus lamented someone said
to her, "What ails you, king's
daughter? You weep so that even
a stone would show pity."
She looked round to the side
from whence the voice came,
and saw a frog stretching forth
its big, ugly head from the wa-
ter. "Ah, old water-splasher, is it
you," she said, "I am weeping for
my golden ball, which has fall-
en into the well." "Be quiet, and
do not weep," answered the frog,
"I can help you, but what will
you give me if I bring your play-
thing up again?" "Whatever you
will have, dear frog," said she,
"My clothes, my pearls and jew-
els, and even the golden crown
which I am wearing." The frog
answered, "I do not care for your
clothes, your pearls and jewels,
nor for your golden crown, but if
you will love me and let me be
your companion and play-fellow,
and sit by you at your little ta-
ble, and eat off your little gold-
en plate, and drink out of your
little cup, and sleep in your little
bed—if you will promise me this
I will go down below, and bring
you your golden ball up again."

(b) ΣUnjustified-optimal solution

Figure 9: Extreme results for objective functions μLooseness and ΣUnjustified.

4.3 Size and spread of result set

We are interested in the number and diversity of Pareto optimal solutions that will be returned for a given configuration of the algorithm. The result of this experiment impacts the usability of our approach: too many similar results and it may be unwieldy for an end-user. Too few results mean some of the benefit of the multiple objective approach is lost. We will measure and report on the number and distribution of solutions returned for column widths of 260pt, 360pt and 460pt with the following combinations of objective functions:

- $\mu\mathrm{Looseness}$ and $\sigma^2\mathrm{Looseness}$
- $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$
- $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{River}$
- $\mu\mathrm{Looseness}$, $\sigma^2\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$
- $\mu\mathrm{Looseness}$, $\sigma^2\mathrm{Looseness}$ and $\Sigma\mathrm{River}$
- $\mu\mathrm{Looseness}$, $\sigma^2\mathrm{Looseness}$, $\Sigma\mathrm{Hyphen}$ and $\Sigma\mathrm{River}$

For all other tests the duplicate culling option has been enabled.

We also show the performance of TEX with respect to each of the objective functions for each of the page widths and compare it to the multiple solutions returned by our algorithm.

Figure 10 shows the two-dimensional plot of solutions in objective space using the $\mu\mathrm{Looseness}$ and $\sigma^2\mathrm{Looseness}$ objective functions for each of the three columns widths. All three widths give the familiar Pareto trade-off curve, however the narrowest column does so with 26 solutions, compared with only 5 for each of the wider columns.

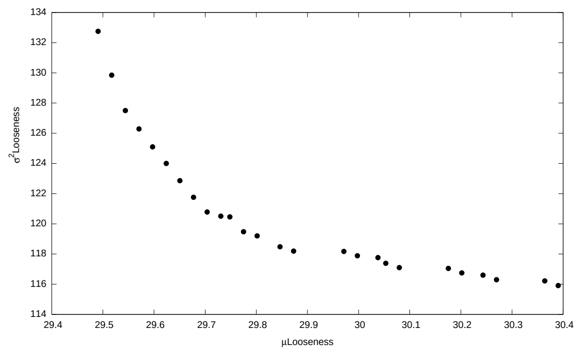
In Figure 10d, the three column widths are superimposed on the same plot along with the position of LATEX's solutions. In every column width, every optimal solution provided by our algorithm measured better than the LATEX solution along these objectives by a wide margin.

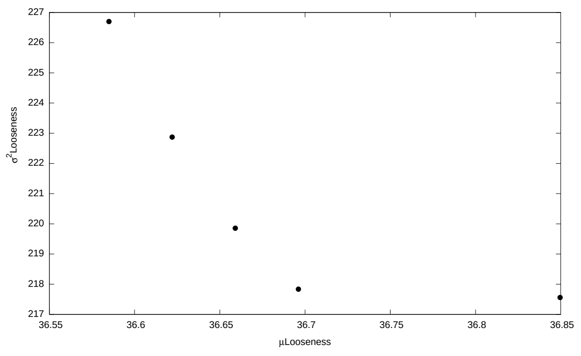
The results shown in Figure 11 for $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$ are similar: all column widths show an even spread with between 19 and 45 results each. There is an obvious and useful trade-off between the number of hyphens and the tightness of the spacing. The LATEXsolutions contain no hyphenations, and do not have as tight spacing as any of those presented by our algorithm.

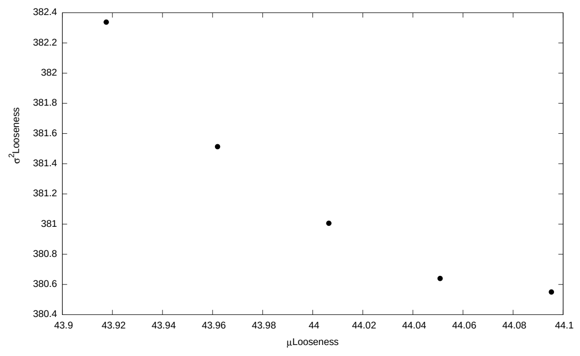
A large number of solutions are presented when $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{River}$ are used together: 65 for the 460pt column. River areas (measured in the horizontal direction only) range from approximately 320pt in the widest column to almost 550pt in the narrowest. These are consistent with expectations: typesetting within a narrow column is more likely to produce rivers due to the limited choice of break points. When compared with LATEX, our algorithm produces text with tighter spacing and approximately half the river area, according to our metric.

We now proceed to combine three objective functions together. The first experiment is with $\mu\mathrm{Looseness}$, $\sigma^2\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$, with the aim of producing tight, evenly spaced paragraphs with a minimum of hyphenations.

The results for the 260pt column are shown in 3-dimensional form in Figure 13. There are over 90 solutions, with solutions having between 0 and 16 hyphenations, a small range of average looseness and relatively small range in the variance of looseness. We can see that identifying the trade-off curve is difficult due to the multi-dimensional nature of the data. In comparison, the 360pt and 460pt columns have only 20 and 9 solutions, respectively (not shown).


(a) 260pt wide column


(b) 360pt wide column


(c) 460pt wide column

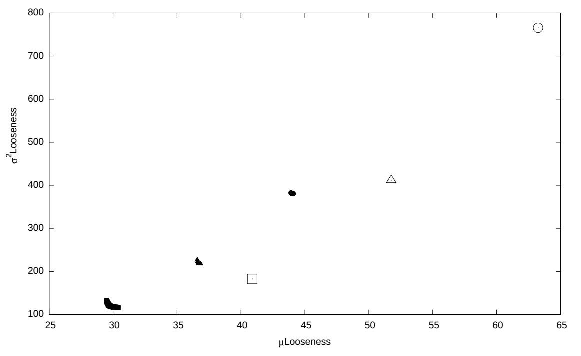
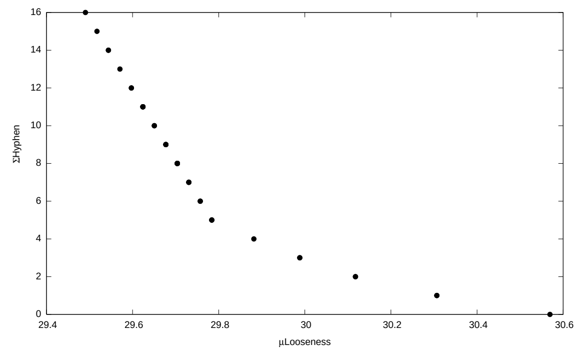
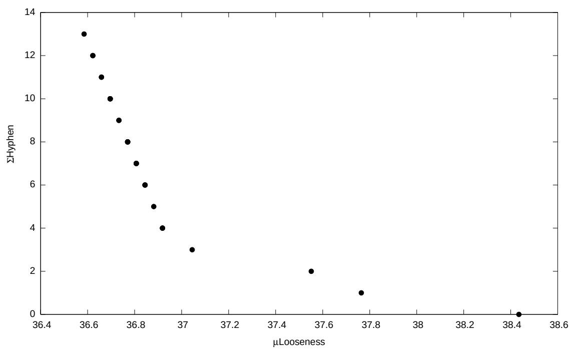

(d) Comparison with LaTeX

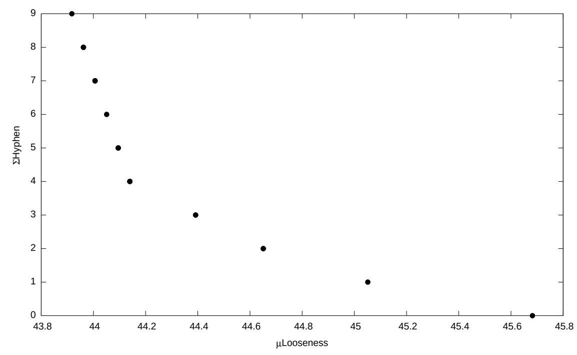
Figure 10: Solutions for $\mu\mathrm{Looseness}$ and $\sigma^2\mathrm{Looseness}$ objective functions for three different column widths. Plot 10d superimposes all three column widths on the same axes, with the 260pt, 360pt and 460pt columns marked as squares, triangles and discs, respectively. The LaTeX solution for each is shown with an unfilled square, triangle or disc.



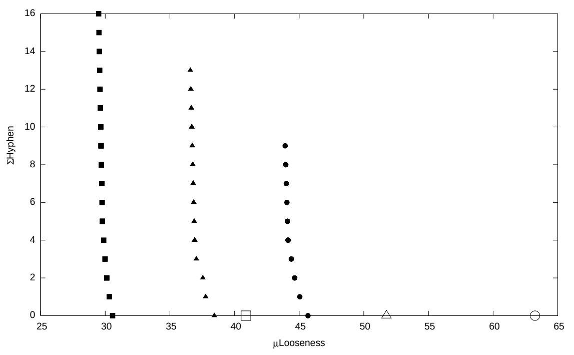
(a) 260pt wide column



(b) 360pt wide column

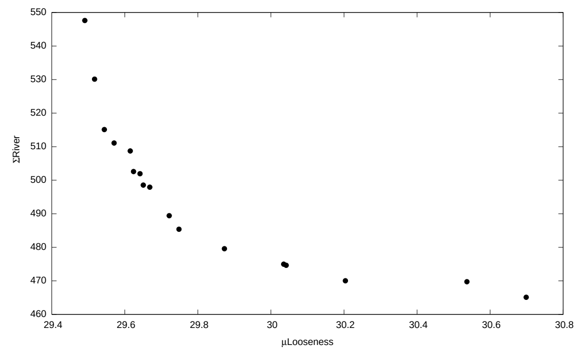


(c) 460pt wide column

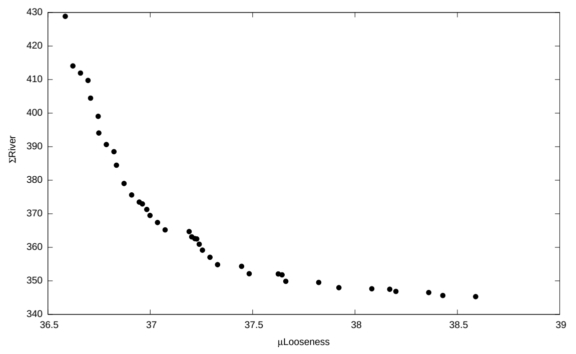


(d) Comparison with LaTeX

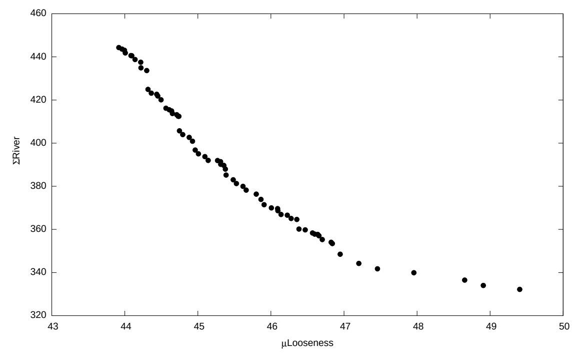
Figure 11: Solutions for $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$ objective functions for three different column widths. Plot 11d superimposes all three column widths on the same axes, with the 260pt, 360pt and 460pt columns marked as squares, triangles and discs, respectively. The LaTeX solution for each is shown with an unfilled square, triangle or disc.



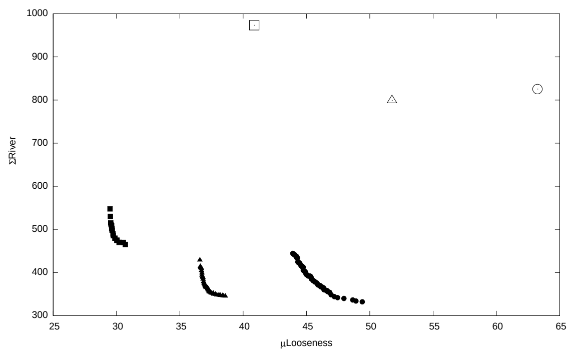
(a) 260pt wide column



(b) 360pt wide column



(c) 460pt wide column



(d) Comparison with LaTeX

Figure 12: Solutions for $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{River}$ objective functions for three different column widths. Plot 12d superimposes all three column widths on the same axes, with the 260pt, 360pt and 460pt columns marked as squares, triangles and discs, respectively. The LaTeX solution for each is shown with an unfilled square, triangle or disc.

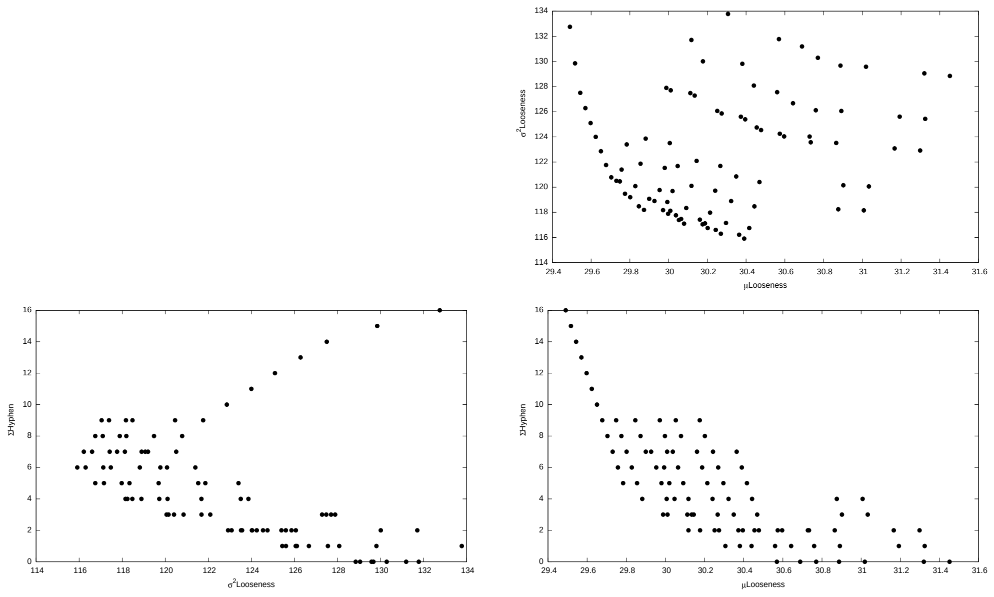


Figure 13: Solution set for the $\mu\mathrm{Looseness}$, $\sigma^2\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$ objective functions in a 260pt column.

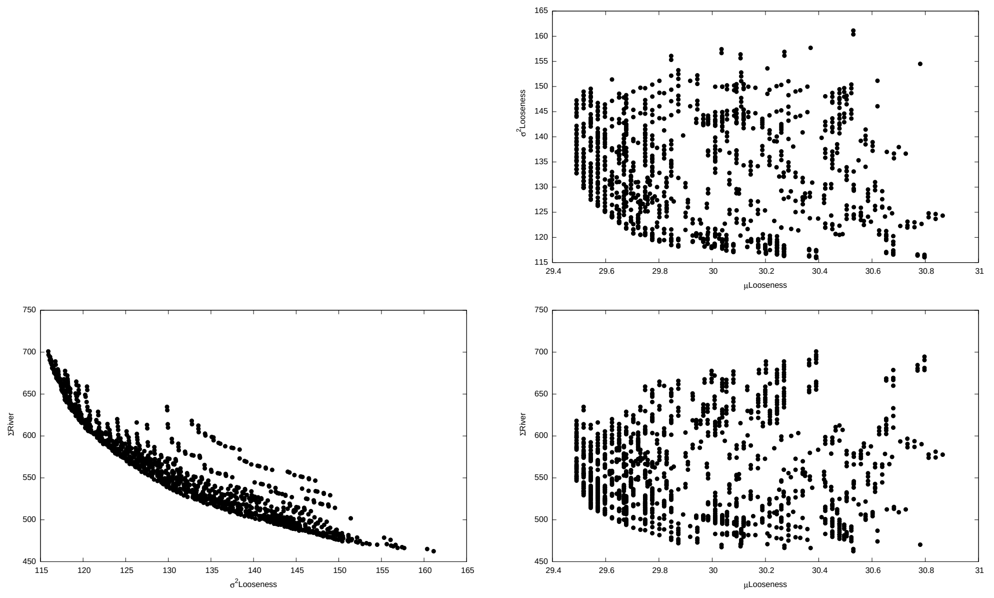


Figure 14: Solution set for the $\mu\mathrm{Looseness}$, $\sigma^2\mathrm{Looseness}$ and $\Sigma\mathrm{River}$ objective functions in a 260pt column.

Column width	Hyphenations	Ratio
260pt	1004	71%
360pt	945	67%
460pt	856	60%

Table 1: Number of hyphenations attempted for varying column widths, and as a ratio of the total word count for the extract.

Combining μLooseness, σ^2Looseness and ΣRiver results in far more solutions: 1176 at 260pt (shown in Figure 14), 322 at 360pt and 363 at 460pt.

We were unable to complete the experiment combining all four objective functions due to the expected long running time, as discussed later.

4.4 Effectiveness of hyphenation algorithm

We claim that our hyphenation algorithm improves performance by not having to examine the hyphenation of each word in the paragraph, only those that are feasible. We test this claim by reporting the ratio of hyphenations examined to the word count. A low ratio indicates the benefit of the algorithm over a naïve implementation.

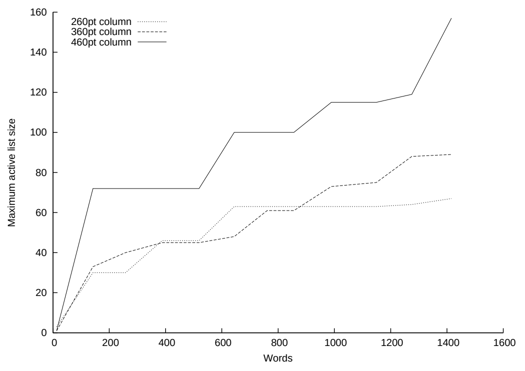
The results are for three column widths are shown in Table 1. Setting different objective functions had no effect on the number of hyphenations attempted. It is clear that using our algorithm to determine possible hyphenation points is more efficient than simply attempting hyphenation on every word. It is possible that the overhead of the algorithm may outweigh this benefit.

4.5 Performance concerns

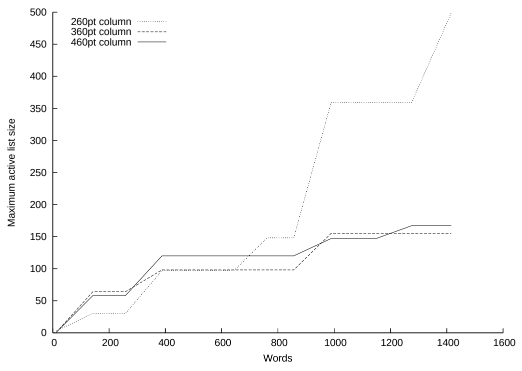
For the algorithm to be practical, it needs to have a competitive running time to that of the status quo. Both the word count and the number and characteristics of the objective functions being used affect performance. Running time is closely related to the number of items in the active list at any given point in time. For column widths 260pt, 360pt and 460pt and increasing document sizes we measure the maximum intermediate size of the active list as well as the wall-clock running time for these combinations of objective functions:

- μLooseness
- μLooseness and σ^2Looseness
- μLooseness and ΣHyphen
- μLooseness and ΣRiver
- μLooseness, σ^2Looseness and ΣHyphen
- μLooseness, σ^2Looseness and ΣRiver

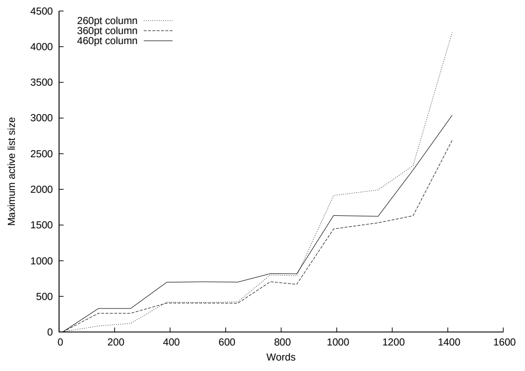
The tests were run on an AMD Athlon 64 3500+ at 2.2GHz with 2 GB RAM under Linux 2.6.17. The prototype implementation is written in Python, an interpreted language. An optimised, native machine implementation could be expected to run 100 times faster, but these figures should be indicative of the trends and tractability of the tests.



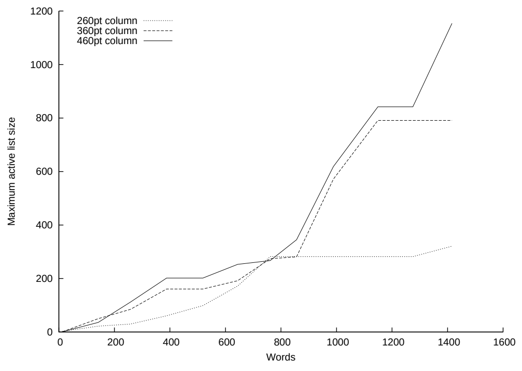
(a) μLooseness



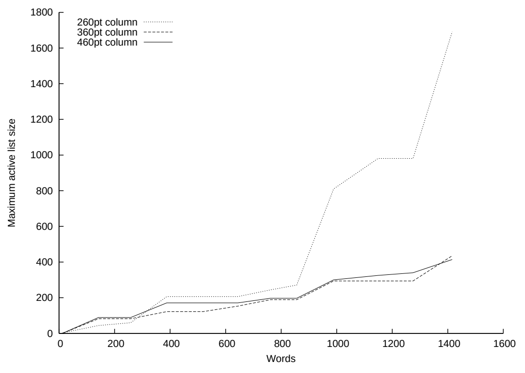
(b) μLooseness and σ^2Looseness



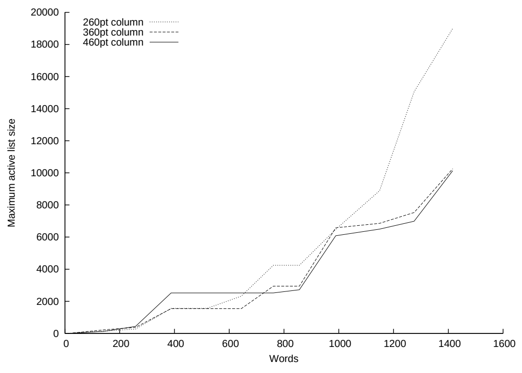
(c) μLooseness and ΣHyphen



(d) μLooseness and ΣRiver

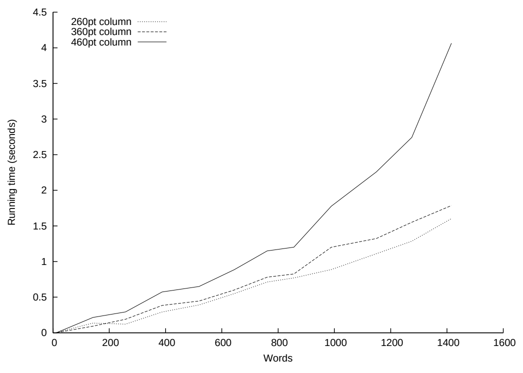


(e) μLooseness, σ^2Looseness and ΣHyphen

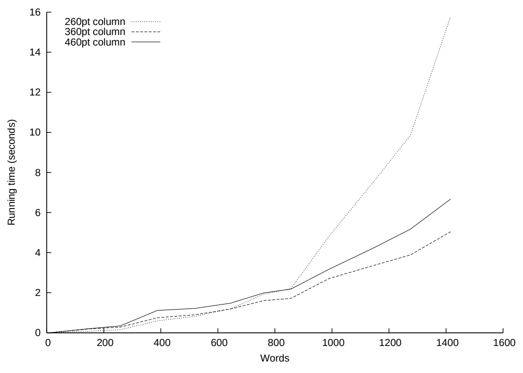


(f) μLooseness, σ^2Looseness and ΣRiver

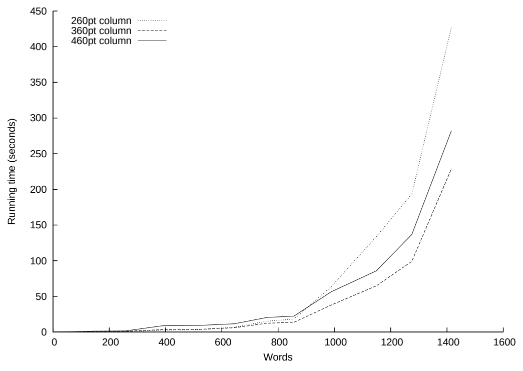
Figure 15: Maximum intermediate size of the active list for increasing document length, for various combinations of objective functions and at three column widths.



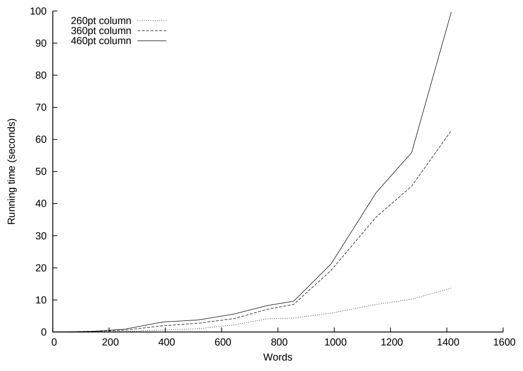
(a) μLooseness



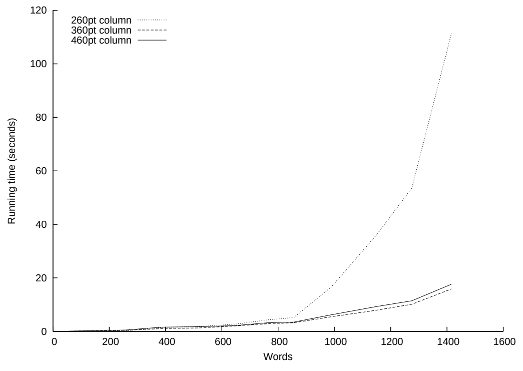
(b) μLooseness and σ^2Looseness



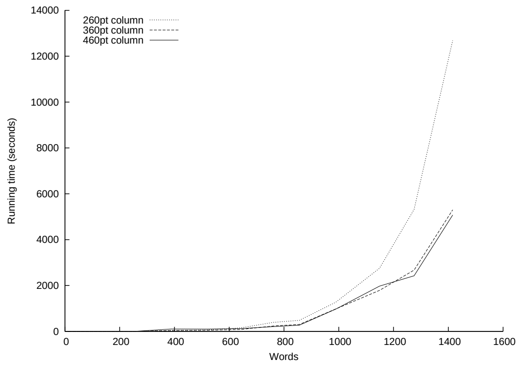
(c) μLooseness and ΣHyphen



(d) μLooseness and ΣRiver



(e) μLooseness, σ^2Looseness and ΣHyphen



(f) μLooseness, σ^2Looseness and ΣRiver

Figure 16: Running time for increasing document length, for various combinations of objective functions and at three column widths.

The maximum size of the active list for each test is shown in Figure 15, and the running time in Figure 16. There is a strong correlation between the size of the active list and the running time, as was predicted.

Larger documents take a longer amount of time to process than shorter ones, and have a higher active list size. This is to be expected, and is easily proven mathematically. It is interesting, however, that in every case the running time and active list size increase at a faster rate as the word count increases, implying a polynomial or exponential trend.

The choice of objective functions is a large factor in the performance of the algorithm, sometimes in surprising and unpredictable ways. For example, the narrowest column is the most computationally intensive case for the $\mu\mathrm{Looseness}$ and $\sigma^2\mathrm{Looseness}$ functions, but the widest column is the most expensive for the $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{River}$ functions. For $\Sigma\mathrm{Hyphen}$ the difference is less pronounced and there is no discernible pattern relating to the column widths.

Also surprising is the change in performance as more objective functions are added. When $\sigma^2\mathrm{Looseness}$ is added to $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{Hyphen}$ performance improves—by more than 10 times for the wider two columns. On the other hand, when $\sigma^2\mathrm{Looseness}$ is added to $\mu\mathrm{Looseness}$ and $\Sigma\mathrm{River}$ performance degrades so badly that some of the tests had to be stopped when they took more than six hours to complete.

One could conceive of various typographical reasons why this is the case. For example, creating a trade-off between consistent spacing ($\sigma^2\mathrm{Looseness}$) and reducing systematic rivers is expensive as the concepts are perhaps in opposition. $\Sigma\mathrm{Hyphen}$ and $\sigma^2\mathrm{Looseness}$ are not necessarily in a trade-off situation, so the solution set is smaller as it collapses into two dimensions.

5 Conclusion

We have presented a new formulation of the line breaking problem, in which typographical features are measured along separate dimensions and multiple optimal solutions are presented to the user.

5.1 Response to research questions

Is it possible to define meaningful metrics for paragraph layout that describe well-known typographical qualities such as looseness, hyphenation, rivers, and so on?

Not only is it possible to define numerical metrics for paragraph layout, it is surprisingly simple. We used the mean and variance of word spacing to find paragraphs that have tight, consistent spacing. Simple counters were implemented to favour paragraphs with fewer hyphens, stacks and cubs. A novel approach to measuring the area of whitespace rivers in a paragraph is shown to produce results consistent with expectations.

These metrics are more familiar to typographers and general users alike, representing real-world quantities rather than arbitrary penalties.

Can multiple objective optimisation be applied to the line breaking problem to return multiple competing solutions? How do these solutions compare to TEX's?

Multiple objective optimisation returns an excellent spread of solutions along each dimension for the sample text using various column widths and objective functions. When combining three or more objective functions, there are perhaps an excessive number of solutions returned. In every measured case, the results of our algorithm outperformed $\TeX$ on the objective functions measured.

It has become apparent through our research that while $\TeX$ returns the optimal paragraph according to its weighted sum measurement, this measurement is not optimal with respect to our metrics. Having shown that our metrics represent real-world typographic qualities, we can say that $\TeX$ can be improved upon, and that our method does this.

How fast does our technique perform? How much memory does it require?

Time and memory (active list size) requirements have positive growth for increasing document length and, more significantly, increasing objectives. The time to process a 1000 word document varies between under a second and several hours, depending on the configuration. Some objective functions work faster together (e.g., $\Sigma\mathrm{Hyphen}$ and $\sigma^2\mathrm{Looseness}$) than others ($\Sigma\mathrm{River}$).

The current implementation is by no means optimised, and has been written in an interpreted language. Nevertheless, it is clear that significant modifications must be made to the algorithm if performance approaching interactive speeds is required.

What benefits does global multiple objective line breaking give over local single optimising line breaking?

Besides the aforementioned improvement in typographic quality, multiple objective line breaking provides multiple competing solutions that represent a genuine trade-off in quality along multiple dimensions. Due to the global nature of the algorithm, it should be straightforward to apply it to pagination and other page-level concerns that cannot currently be solved optimally by existing systems. Performance problems are the main limiting factor in implementing this.

5.2 User interface concerns

While we have not discussed the issue of user interfaces so far, this has been a topic of significant interest while we carried out the research. We envisage an application making use of our algorithm could present several objective functions to the user before typesetting a document, who could pick and choose the ones they feel are relevant. For example, a publishing house printing a novel will want to optimise for elimination of hyphens and rivers, whereas a technical journal will optimise for tight spacing and reduced page count.

Presentation of results could proceed in several ways. For the expert user interested in typography, it would be most beneficial to show the multi-dimensional plots and allow them to select, view and cull the results until they have found one they are most satisfied with. For more ordinary users, a preference order of the objectives, or objective-space weighted function could automate this task with sensible defaults created by expert typographers. Note that weighting the results in objective space is far more desirable than the penalty weighting in decision space that $\TeX$ performs.

5.3 Future work

We have identified several areas that need more research. Most pressing is the issue of performance. The performance seriously deteriorates as more objective functions are added, but there is nothing to prevent a user from combining several objective functions with a weighting function. This is not ideal, as it somewhat defeats the main argument of this research, but for some cases it may not be completely detrimental. For example, combining $\Sigma\mathrm{Cub}$, $\Sigma\mathrm{Stack}$ and $\Sigma\mathrm{Hyphen}$ into a single objective function measuring the number of "typographic anomalies" may work well enough.

Speed is also affected by document length, particularly because we optimise the entire document at once, rather than a paragraph at a time, as TeX does. We perform this global optimisation because no other solution allows us to properly collate the multiple solutions generated at each paragraph. It also has the perceived future benefit of allowing pagination to be optimised simultaneously. It may be possible to perform the optimisation in two passes, first individual paragraphs, then optimising pagination for paragraphs of different line counts—it is not immediately apparent if this is feasible.

Our optimising algorithm can perhaps be improved by making use of the multiple objective approximations that have been discussed in recent years. This is a non-trivial problem, but could solve both the performance problem as well as that of returning too many solutions to the user.

We have not investigated any objective functions beyond the paragraph level line breaking ones. Objective functions are easily devised to optimise widows, orphans, placement of floats and footnotes, page count, and so on. These should be implemented and tested for consistency with the expected typographical output. Further avenues of research could include allowing floats and margins to be dynamically resized by the optimisation process in order to find an optimal fit.

Finally, prototype user interfaces need to be devised and users interviewed for feedback on their usability, productivity and the quality of output.

References

Barnett, M. (1965). *Computer typesetting: experiments and prospects*. MIT Press.

Bellman, R. (1957). *Dynamic Programming*. Dover Publications.

Brown, T. and Strauch, R. (1965). Dynamic programming in multiplicative lattices. *Journal of mathematical analysis and applications*, 12:364–370.

Bruggemann-Klein, A., Klein, R., and Wohlfeil, S. (1995). Pagination reconsidered. *Electronic Publishing—Origination, Dissemination, and Design*, 8(2):139–152.

Brüggemann-Klein, A., Klein, R., and Wohlfeil, S. (2003). On the pagination of complex documents. In Wegner, L., Six, H.-W., Ottman, T., and Klein, R., editors, *Computer Science in Perspective*, pages 49–68. Springer.

Climaco, J. and Martins, E. (1982). A bicriterion shortest path algorithm. *European Journal of Operational Research*, 11:399–404.

Corley, H. (1985). Some multiple objective dynamic programs. *IEEE Transactions on Automatic Control*, 30(12):1221–1222.

Corley, H. and Moon, I. (1985). Shortest paths in networks with vector weights. *Journal of Optimization Theory and Applications*, 46(1):79–86.

Daellenbach, H. and Kluyver, C. (1980). Note on multiple objective dynamic programming. *The Journal of the Operational Research Society*, 31(7):591–594.

Deb, K. (2001). *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman & Co. New York, NY, USA.

Getachew, T., Kostreva, M., and Lancaster, L. (2000). A generalization of dynamic programming for pareto optimization in dynamic networks. *RAIRO Operations Research*, 34:27–47.

Grimm, J. L. K. and Grimm, W. K. (1857). The frog king. In *Complete Fairy Tales of the Brothers Grimm*. Addison-Wesley.

Grossman, J. (1993). *The Chicago manual of style*. The University of Chicago Press.

Hansen, P. (1980). Bicriterion path problem. *Multiple Criteria Decision Making: Theory and Applications, LNEMS*, 177:109–127.

Hartley, R. (1985). Vector optimal routing by dynamic programming. In Serafini, P., editor, *Mathematics of Multiobjective Optimization*, volume 289 of *CISM Courses and Lectures*, pages 215–224. International Centre for Mechanical Sciences, Udine, Italy.

Held, M. and Karp, R. (1965). The construction of discrete dynamic programming algorithms. *IBM Journal of Research and Development*, 4(2):136–147.

Henig, M. (1983). Vector-valued dynamic programming. *SIAM Journal of Control and Optimization*, 21(3):490–499.

Henig, M. (1986). Shortest path problem with two objective functions. *European Journal of Operational Research*, 25(2):281–291.

Henig, M. (1994). Efficient interactive methods for a class of multiattribute shortest path problems. *Management Science*, 40(7):891–897.

Jacobs, C., Li, W., Schrier, E., Bargeron, D., and Salesin, D. (2003). Adaptive grid-based document layout. *ACM Transactions on Graphics*, 22(3):838–847.

Justus, P. E. (1972). There is more to typesetting than setting type. *IEEE Transactions on Professional Communications*, PC-15:13–16.

Klamroth, K. and Wiecek, M. (2000). Dynamic programming approaches to the multiple criteria knapsack problem. *Naval Research Logistics*, 47(1):57–76.

Knuth, D. (1979). *TEX and METAFONT: New directions in typesetting*. American Mathematical Society Boston, MA, USA.

Knuth, D., Blue Sky Research, and Y&Y, Inc. (2001). Computer Modern PostScript Fonts (Adobe Type 1 format). Accessible online at `http://www.ams.org/tex/type1-fonts.html`.

Knuth, D. E. and Plass, M. F. (1981). Breaking paragraphs into lines. *Software, Practice and Experience*, 11(11):1119–1184.

Li, D. and Haimes, Y. Y. (1987). The envelope approach for multiobjective optimization problems. *IEEE Transactions on Systems, Man and Cybernetics*, 17(6):1026–1038.

Liang, F. (1983). *Word Hy-phen-a-tion by Comput-er*. PhD thesis, Ph. D. thesis, Stanford University, Stanford.

Liao, L. and Li, D. (2002). Adaptive differential dynamic programming for multiobjective optimal control. *Automatica*, 38(6):1003–1015.

Lumley, J., Gimson, R., and Rees, O. (2006). Extensible layout in functional documents. *Proceedings of SPIE*, 6076:177–188.

Martins, E. (1984a). On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245.

Martins, E. (1984b). On a special class of bicriterion path problems. *European Journal of Operational Research*, 17:85–94.

Mitten, L. (1964). Composition principles for synthesis of optimal multistage processes. *Operations Research*, 12(4):610–619.

Mitten, L. (1974). Preference order dynamic programming. *Management Science*, 21(1):43–46.

Plass, M. and Knuth, D. (1982). Choosing better line breaks. *Document Preparation Systems: A Collection of Survey Articles*, pages 221–242.

Plass, M. F. (1981). *Optimal pagination techniques for automatic typesetting systems*. PhD thesis, Standford University.

Purvis, L. (2002). A genetic algorithm approach to automated custom document assembly. In *Second international workshop on Intelligent systems design and application*, pages 131–136, Atlanta, GA, USA. Dynamic Publishers, Inc.

Purvis, L., Harrington, S., O'Sullivan, B., and Freuder, E. (2003). Creating personalized documents: an optimization approach. *Proceedings of the 2003 ACM symposium on Document engineering*, pages 68–77.

Rich, R. and Stone, A. (1965). Method for hyphenating at the end of a printed line. *Communications of the ACM*, 8(7):444–445.

Rubinstein, R. (1988). *Digital Typography*. Addison-Wesley, first edition.

Safer, H. M. and Orlin, J. B. (1992). *Fast approximation schemes for multi-criteria combinatorial optimization*. PhD thesis, Massachusetts Institute of Technology, Sloan School of Management.

Samet, H. (1982). Heuristics for the line division problem in computer justified text. *Communications of the ACM*, 25(8):564–570.

Sancho, N. (1985). Routing problems and markovian decision processes. *Journal of Mathematical Analysis and Applications*, 105(1):76–83.

Sobel, M. (1975). Ordinal dynamic programming. *Management Science*, 21(9):967–975.

Villarreal, B. and Karwan, M. (1981). Multicriteria integer programming: A (hybrid) dynamic programming recursive approach. *Mathematical Programming*, 21(1):204–223.

Villarreal, B. and Karwan, M. (1982). Multicriteria dynamic programming with an application to the integer case. *Journal of Optimization Theory and Applications*, 38(1):43–69.

Wakuta, K. (2001). A multi-objective shortest path problem. *Mathematical Methods of Operations Research (ZOR)*, 54(3):445–454.

Warburton, A. (1987). Approximation of pareto optima in multiple-objective, shortest-path problems. *Operations Research*, 35(1):70–79.

Yu, P. and Seiford, L. (1989). Multistage decision problems with multiple criteria. In *Multiple Criteria Analysis*, pages 235–244.