copyrightbox

Optimal Guillotine Layout

Graeme Gange
Dept of CSSE
University of Melbourne
Vic. 3010, Australia
ggange@cs.mu.oz.au

Kim Marriott
Clayton School of IT
Monash University
Vic. 3800, Australia
kim.marriott@monash.edu

Peter Stuckey
Dept of CSSE
University of Melbourne
Vic. 3010, Australia
pjs@cs.mu.oz.au

ABSTRACT

Guillotine-based page layout is a method for document layout commonly used by newspapers and magazines, where each region of the page either contains a single article, or is recursively split either vertically or horizontally. Suprisingly there appears to be little research into algorithms for automatic guillotine-based document layout. In this paper we give efficient algorithms to find optimal solutions to guillotine layout problems of two forms. Fixed-cut layout is where the structure of the guillotining is given and we only have to determine the best configuration for each individual article to give the optimal total configuration. Free layout is where we also have to search for the optimal structure. We give bottom-up and top-down dynamic programming algorithms to solve these problems, and propose a novel interaction model for documents on electronic media. Experiments show that our algorithms are effective for realistic layout problems.

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*Format and notation, Photocomposition/typesetting*

General Terms

Algorithms

Keywords

guillotine-based document layout, constrained optimization, dynamic programming, typography

1. INTRODUCTION

Guillotine-based page layout is a method for document layout, commonly used by newspapers and magazines, where each region of the page either contains a single article, or is recursively split either vertically or horizontally. The newspaper page shown in Figure 1(a) is an example of a guillotine-based layout where Figure 1(b) shows the series of cuts used to construct this layout.

(a) (b)

Figure 1: (a) Front page of The Boston Globe, together with (b) the series of cuts used in laying out the page. Note how the layout uses fixed width columns.

Surprisingly, there appears to have been relatively little research into algorithms for automatic guillotine-based document layout. We assume that we are given a sequence of articles $A_1, A_2, \ldots, A_n$ to layout. The precise problem depends upon the page layout model [8].

- The first model is vertical scroll layout in which layout is performed on a single page of fixed width but unbounded height: this is the standard model for viewing HTML and most web documents. Here the layout problem is to find guillotine layout for the articles which minimises the height for a fixed width.

- The second model is horizontal scroll layout in which there is a single page of fixed height but unbounded width. This model is well suited to multicolumn layout on electronic media. Here the layout problem is to find guillotine layout for the articles which minimises the width for a fixed height.

- The final model is layout for a sequence of articles in fixed height and width pages. Here the problem is to find a guillotine layout which maximises the prefix

DocEng'12, September 4–7, 2012, Paris, France.

of the sequence of articles $A_1, A_2, \ldots, A_k$ that fit on the (first) page (and then subsequently for the second, third, ... page).

We are interested in two variants of these problems. The easier variant is *fixed-cut* guillotine layout. Here we are given a guillotining of the page and an assignment of articles to the rectangular regions on the page. The problem is to determine how to best layout each article so as to minimise the overall height or width. The much harder variant is *free* guillotine layout. In this case we need to determine the guillotining, article assignment and the layout for each article so as to minimise overall height or width.

The main contribution of this paper is to give polynomial-time algorithms for optimally solving the fixed-cut guillotine layout problem and a dynamic programming based algorithm for optimally solving the free guillotine layout. While our algorithm for free guillotine layout is exponential (which is probably unavoidable since the free guillotine layout problem is NP-Hard (see Section 2), it can layout up to 13 articles in a few seconds (up to 18 if the articles must use columns of a fixed width).

Our automatic layout algorithms support a novel interaction model for viewing documents such as newspapers or magazines on electronic media. In this model we use free guillotine layout to determine the initial layout. We can fine tune this layout using fixed-cut guillotine layout in response to user interaction such as changing the font size or viewing window size. Using the same choice of guillotining ensures the basic relative position of articles remains the same and so the layout does not change unnecessarily and disorient the reader. An example of this is shown in Figure 2. However, if at some point the choice of guillotining leads to a very bad layout, such as articles that are too wide or narrow or too much wasted space, then we can find a new guillotining that is close to the original guillotining, and re-layout using this new choice.

Guillotine-based constructions have been considered for a variety of document composition problems. Photo album composition approaches [3] have a fixed document size, and must construct an aesthetically pleasing layout while maintaining the aspect ratio of images to be composed.

A number of heuristics have been developed for automated newspaper composition [6, 11] which also focus on constructing layouts for a fixed page-width. The first approach [6] considers only a single one column configuration per article, and lays out all articles to minimize height in a fixed number of columns. The second approach [11] breaks the page into a grid and considers up to 8 configurations on grid boundaries per article, It focuses on choosing which articles to place in a fixed page size, using a complex objective based on coverage. Both approaches make use of local search and do not find optimal solutions.

Hurst [7] suggested solving the fixed-cut guillotine layout problem by solving a sequence of one-dimensional minimisation problems to determine a good layout recursively. This approach was fast but not guaranteed to find an optimal layout. A genetic algorithm was also proposed for guillotine layout, using a linear approximation for content [**?**].

A closely related problem to these is the guillotine stock-cutting problem. Given an initial rectangle, and a (multi-)set S of smaller rectangles with associated values, the objective is to find a cutting pattern which gives the set $S' \subseteq S$ with maximum value. This in some sense a harder form of the third model we discuss above. A number of exact [5, 4] and heuristic [2] methods have been proposed for the guillotine stock-cutting problem. This differs from the guillotine layout problem in that each leaf region has a single configuration, rather than a (possibly large sized) disjoint set of possible configurations. It does not appear that these approaches scale to the size of problem we consider.

2. PROBLEM STATEMENT

In the rest of the paper will focus on finding a guillotine layout which minimises the height for a fixed width. It is straightforward to modify our algorithms to find a guillotine layout which minimises the width for a fixed height: we simply swap height and widths in the input to the algorithms.

We can also use algorithms for minimising height to find a guillotine layout maximising the number of articles in a fixed size page. For a particular subsequence $A_1, .., A_k$ we can use the algorithm to compute the minimum height h_k for laying them out in the page width. We simply perform a linear or binary search to find the maximum k for which h_k is less than the fixed page height. We can use the area of the articles' content to provide an initial upper bound on k.

The main decision in the fixed-cut guillotine layout is how to break the lines of text in each article. Different choices give rise to different width/height configurations. Each article has a number of *minimal configurations* where a minimal configuration is a pair (w, h) s.t. the content in the article can be laid out in a rectangle with width w and height h but there is no smaller rectangle for which this is true. That is, for all $w' \leq w$ and $h' \leq h$ either $h = h'$ and $w = w'$, or the content does not fit in a rectangle with width w' and height h'.

Typically we would like the article to be laid out with multiple columns. One way of doing this is to allow the configuration to take any width and to compute the number of columns and their width based on the width of the configuration. We call this *article dependent* column layout. In this case for text with uniform height with W words (or more exactly, $W-1$ possible line breaks), there are up to W minimal configurations, each of which has a different number of lines. In the case of non-uniform height text, there can be no more than $O(W^2)$ minimal configurations.

The other way of computing the columns is to compute the width and number of columns based on the page width and then each article is laid out in a configuration of one, two, three etc column widths. This is, for instance, the approach used in Figure 1. We call this *page dependent* column layout. In this case the number of different configurations is much less and is simply the number of columns on the page.

We assume the minimal configurations for an article A are given as a discrete list of allowed configurations $C(A) = [\ (w_0, h_0),\ \ldots, (w_k, h_k)\ ]$, ordered by increasing width (and decreasing height). In the algorithms described in the following sections, we refer to the i^{th} entry of an ordered list L with $L[i]$ (adopting the convention that indices start at 0), and concatenate lists with $+\!+$. For a configuration c, we use $\mathsf{w}(c)$ to indicate the width, and $\mathsf{h}(c)$ for the height. Note that we can choose to not include configurations that are too narrow or too wide.

A guillotine cut is represented by a tree of cuts, where each node has a given height/width configuration. A leaf node $\textsc{cell}(A)$ in the tree holds an article A. An internal node is either: $\textsc{vert}(X, Y)$, where X and Y are its child nodes,

(a) (b)

Figure 2: Example of (a) a possible guillotine layout, and (b) the same layout adapted to a narrower display width.

representing a vertical split with articles in X to the left and articles in Y to the right; or $\text{HORIZ}(X, Y)$, representing a horizontal split with articles in X above and articles in Y below. Given a chosen configuration for each leaf node we can determine the configuration of each internal nodes as follows:

If $c(X) = (w_x, h_x)$ is the chosen configuration for X and $c(Y) = (w_y, h_y)$ is the chosen configuration for Y, then define

$$\mathsf{vert}((w_x, h_x), (w_y, h_y)) = (w_x + w_y, \max(h_x, h_y))$$
$$\mathsf{horiz}((w_x, h_x), (w_y, h_y)) = (\max(w_x, w_y), h_x + h_y)$$

and let

$$c(\text{VERT}(X, Y)) = \mathsf{vert}(c(X), c(Y))$$
$$c(\text{HORIZ}(X, Y)) = \mathsf{horiz}(c(X), c(Y)).$$

The *fixed-cut guillotine layout problem* for fixed width w is given a fixed tree T, determine the configuration of leaf nodes (and internal nodes) such that $c(T) = (w_r, h_r)$ where $w_r < w$ and h_r is minimized.

The *free guillotine layout problem* for fixed width w is given a set of articles S determine the guillotine cut T for S and configurations of leaf nodes (and internal nodes) such that $c(T) = (w_r, h_r)$ where $w_r < w$ and h_r is minimized.

We note that the free guillotine layout problem is NP-hard, by reduction from PARTITION [**?**]. Consider an instance $\{n_1, \ldots, n_k\}$ of PARTITION. We construct a free guillotine layout instance with $w = 2$, and leaves $L_1 \ldots L_n$ with configurations $C(L_k) = \{(1, n_k)\}$. The PARTITION instance is satisfiable iff the minimum height is $\frac{\sum_k L_k}{2}$.

3. FIXED-CUT GUILLOTINE LAYOUT

We will first look at solving the fixed-cut guillotine layout problem. This is a restricted form of guillotine layout, where the tree of cuts is specified, and the algorithm must pick a configuration for each article which leads to the minimum height layout. Fixed-cut guillotine layout is useful in circumstances such as online newspapers, where the layout should remain consistent, but must adapt to changes in display area. An example of this is given in Figure 2. It might also be useful in semi-automatic document authoring tools that support guillotine layout.

3.1 Bottom-up construction

Dynamic programming is a natural approach to tackle minimum height guillotine layout problems since each sub problem of the guillotine layout is again a (smaller) minimum height guillotine layout problem. The only real choices that arise in fixed-cut layout are where to place the vertical split between X and Y in a vertical cut $\text{VERT}(X, Y)$ in order to obtain the minimal height. Rather than searching for a best vertical cut, we solve this problem in the bottom-up construction by computing the *list* of minimal configurations, $C(X)$ for each subtree X of T.

Consider a node $\text{VERT}(X, Y)$, where $C(X)$ is the list of minimal configurations for X and $C(Y)$ is the list of minimal configurations for Y. To construct the list of minimal configurations for $\text{VERT}(X, Y)$ we iterate across the configurations $C(X)$ and $C(Y)$. Given a minimal configuration $\mathsf{vert}(C(X)[i], C(Y)[j])$, we want to find the next minimal configuration that is wider (and shorter). If $C(X)[i]$ is taller than $C(Y)[j]$, we can only construct a shorter configuration by picking a shorter configuration for X. In fact, the next minimal configuration is exactly $\mathsf{vert}(C(X)[i+1], C(Y)[j])$. We can use similar reasoning for the cases where $C(X)[i]$ is shorter than $C(Y)[j]$. Since $\text{VERT}(C(X)[0], C(Y)[0])$ is the narrowest minimal configuration, we can construct all minimal configurations by performing a linear scan over $C(X)$ and $C(Y)$. Pseudo-code for this is given in Figure 3.

```
vert_configs(CX,CY,w)
    C := ∅
    i := 0
    j := 0
    while (i ≤ |CX| ∧ j ≤ |CY|)
        (w_x, h_x) := CX[i]
        (w_y, h_y) := CY[j]
        if(w_x + w_y > w) break
        C := C ++[ vert(CX[i], CY[j]) ]
        if(h_x > h_y) i := i + 1
        else if(h_x < h_y) j := j + 1
        else
            i := i + 1
            j := j + 1
    return C
```

Figure 3: Algorithm for constructing minimal configurations for a vertical split from minimal configurations for the child nodes.

EXAMPLE 3.1. *Consider a problem with 3 articles* $\{X, Y, Z\}$ *having configurations* $C(X) = C(Y) = [\ (1, 2), (2, 1)\ ], C(Z) = [\ (1, 3), (2, 2), (3, 1)\ ]$*, and the tree of cuts shown in Figure 4.*

Consider finding the optimal layout for $w = 3$*. First we must construct the minimal configurations for the node*

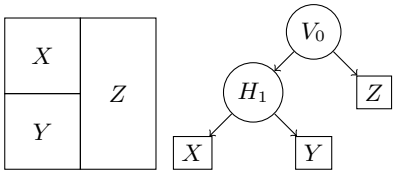

Figure 4: Cut-tree for Example 3.1.

```
horiz_configs(CX,CY,w)
    C := ∅
    minW := max(w(CX[0]), w(CY[0]))
    i := arg max_{i'} w(CX[i']) s.t. w(CX[i']) ≤ minW
    j := arg max_{j'} w(CY[j']) s.t. w(CY[j']) ≤ minW
    while (i ≤ |CX| ∨ j ≤ |CY|)
        C := C ++[ horiz(CX[i], CY[j]) ]
        (w_x, h_x) := CX[i + 1]
        (w_y, h_y) := CY[j + 1]
        if(j + 1 = |CY| ∨ w_x > w_y) i := i + 1
        else if(i + 1 = |CX| ∨ w_x < w_y) j := j + 1
        else
            i := i + 1
            j := j + 1
    return C
```

Figure 5: Algorithm for producing minimal configurations for a horizontal split from child configurations. While the maximum width is included as an argument for consistency, we don't need to test any of the generated configurations, since the width of the node is bounded by the width of the input configurations.

marked H_1. We start by picking the narrowest configurations for X and Y, giving $C(H_1) = [\ (1,4)\ ]$. We then need to select the next narrowest configuration from either X or Y. Since both have the same width, we then join both $(2,1)$ configurations, to give $C(H_1) = [\ (1,4),(2,2)\ ]$.

We then construct the configurations for V_0. We again select the narrowest configurations, $C(H_1)[0]$ and $C(Z)[0]$, giving $C(V_0) = [\ (2,4)\ ]$. Since $C(H_1)[0]$ is taller, we select the next configuration from H_1. Combining $C(H_1)[1]$ with $C(Z)[0]$ gives us $C(V_0) = [\ (2,4),(3,3)\ ]$. Since $w = 3$, we can terminate at this point, giving $(3,3)$ as the minimal configuration. If w were instead 4, we would combine $C(H_1)[1]$ with $C(Z)[1]$, giving the new configuration $(4,2)$. □

Constructing the minimal configurations for HORIZ(X,Y) is exactly the dual of the vertical case. From a minimal configuration constructed from $C(X)[i]$ and $C(Y)[j]$, we can construct a new minimal configuration by picking the narrowest of $C(X)[i+1]$ and $C(Y)[j+1]$. The only additional complexity is that (a) HORIZ$(C(X)[0], C(Y)[0])$ is not guaranteed to be a minimal configuration, and (b) we must keep producing configurations until both children have no more successors, rather than just one. Pseudo-code for this is given in Figure 5, and the overall algorithm is in Figure 6.

Consider a cut VERT(X,Y) with children X and Y. Given $C(X)$ and $C(Y)$, the algorithm described in Figures 3 to 6 computes the configurations for $C($VERT$(X,Y))$ in $O(|C(X)|+|C(Y)|)$, yielding at most $|C(X)| + |C(Y)|$ configurations (and similarly for HORIZ(X,Y)). Given a set of leaf nodes S, we construct at most $\sum_{A\in S}|C(A)|$ configurations at any node. As we perform this step $|S|-1$ times, this gives a worst-case time complexity of $O(|S|\sum_{A\in S}|C(A)|)$ for the bottom-up construction.

An advantage of the bottom-up construction method is that, if we record the lists of constructed configurations, we can update the layout for a new width in $O(\log|C| + |T|)$ time by performing a binary search on configurations of the root node using the new width, then follow the tree of

```
layout_set(T,w)
    switch (T)
        case CELL(A):
            return C(A)
        case VERT(T_1, T_2):
            return vert_configs(layout_set(T_1, w),
                                layout_set(T_2, w), w))
        case HORIZ(T_1, T_2):
            return horiz_configs(layout_set(T_1, w),
                                 layout_set(T_2, w), w))
    endswitch
```

Figure 6: Algorithm for constructing the list of minimal configurations for a fixed set of cuts.

child configurations (or $O(|T|)$ time if we use $O(w)$ space to construct a lookup table).

3.2 Top-down dynamic programming

We also consider a top-down dynamic programming approach, where subproblems are expanded only when required for computing the optimal solution. Consider a subproblem layout(HORIZ(X,Y), w). Using a top-down method, we need only to calculate subproblems layout(X, w) and layout(Y,w), rather than all configurations for the current node. The difficulty is in the case of vertical cuts, as we cannot determine directly how much of the available width should be allocated to X or Y. As such, we must compute layout$(X,\ w')$ and layout$(Y,\ w-w')$ for the set of possible cut positions w'.

A top down dynamic programming solution is almost a direct statement of the Bellman equations as a functional program, with caching to avoid repeated computation. The main difficulty is the requirement to examine every possible width when determining the best vertical split. Psuedocode is given in Figure 7, where lookup(T,w) looks in the cache to see if there is an entry $(T,w) \mapsto c$ and returns c is so, or NOTFOUNDif not; and cache(T,w,c) adds an entry $(T,w) \mapsto c$ to the cache.

This algorithm is outlined in Figure 7. Note that for simplicity we ignore the case where there is no layout of tree T with width $\leq w$. This can be easily avoided by adding an artificial configuration $(0,\infty)$ to the start of the list of configurations for each article A.

While the algorithm finds the optimal solutions quite quickly for fixed trees, there are a number of improvements to this basic algorithm which will also be useful for the free layout problem (Section 4).

3.2.1 Restricting vertical split positions

The algorithm given in Figure 7, on a vertical split, must iterate over all possible values of w' to find the optimal cut position. Let w^T_{min} indicate the narrowest possible configuration for T. Since we are only interested in feasible layouts, for a node VERT(T_1,T_2) we need only consider cut positions in $[w^{T_1}_{min}, w - w^{T_2}_{min}]$. We can improve this by using a binary cut to eliminate regions that cannot contain the optimal solution, keeping track of the range *low..high* where the optimal cut is.

Consider the cut show in Figure 8(a). Let $h^{w'}_A$ = layout(A,w') and $h^{w-w'}_B$ = layout$(B, w-w')$. In this case, $h^{w'}_A > h^{w-w'}_B$. As the resulting configuration has height $\mathbf{max}(h^{w'}_A, h^{w-w'}_B)$, the only way we can reduce the overall height is by adopt-

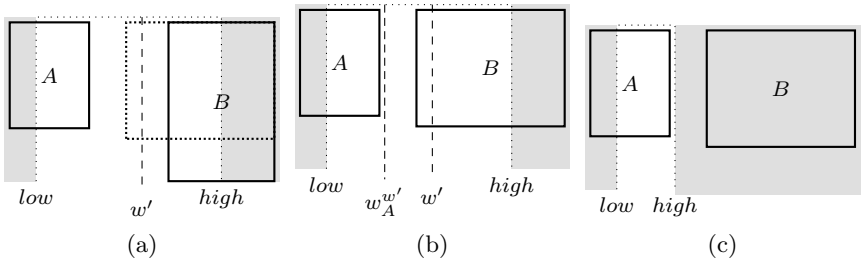

Figure 9: If the optimal layout for layout(A, w') has width smaller than w', then we may lay out B in all the available space, using $w - w_A^{w'}$, rather than $w - w'$. If B is still taller than A, we know the cut must be moved to the left of $w_A^{w'}$ to find a better solution.

```
layout(T,w)
    c := lookup(T,w)
    if c ≠ NotFound return c
    switch (T)
    case cell(A):
        c := C(A)[i] where i is maximal s.t. w(C(A)[i]) ≤ w
    case horiz(T1, T2):
        c := horiz(layout(T1, w), layout(T2, w))
    case vert(T1, T2):
        c := (0, ∞)
        for w′ = 0..w
            c′ := vert(layout(T1, w′), layout(T2, w − w′))
            if (h(c′) < h(c)) c := c′
        endfor
    endswitch
    cache(T,w,c)
    return c
```

Figure 7: Pseudo-code for the basic top-down dynamic programming approach, returning the minimal height configuration $c = (w_r, h_r)$ for tree T such that $w_r \leq w$.

ing a shorter configuration for A – by moving w' further to the right. Normally we would set $low = w'$ as shown in Figure 8(b). In fact, we can move low to $\mathbf{max}(w', w - w_B^{w-w'})$ as shown in Figure 8(c), since moving w' right cannot increase the overall height until B shifts to a narrower configuration.

We can improve this further by observing that, if configurations are sparse, we may end up trying multiple cuts corresponding to the same configuration. If we construct a layout for A with cut position w', but A does not fill all the available space (so $w_A^{w'} < w'$), we can use that additional space to lay out B. If B is still taller than A (as shown in Figure 9), we know that the cut can be shifted to the left of $w_A^{w'}$, rather than just w'.

The case for vert(T_1, T_2) in Figure 7 can then be replaced with the following:

```
c := (0, ∞)
low := w_min^T1
high := w − w_min^T2
while (low ≤ high)
    w′ := ⌊(low + high)/2⌋
    c1 := layout(T1,w′)
```

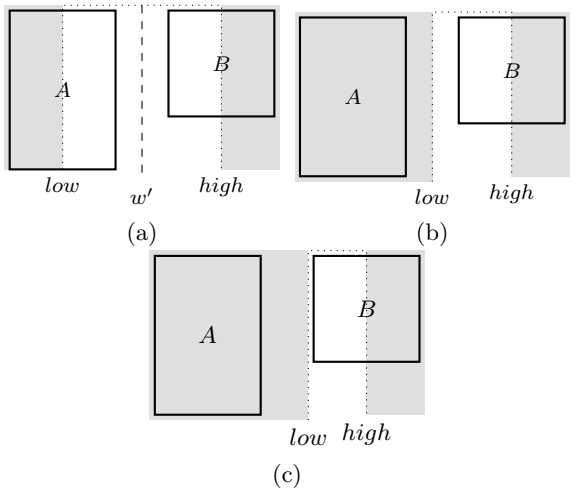

Figure 8: Illustration of using a binary chop to improve search for the optimal cut position. If $h_A^{w'} > h_B^{w-w'}$ as shown in (a), we cannot improve the solution by moving the cut to the left. Hence we can update (b) $low = w'$. Since B will retain the same configuration until the cut position exceeds $w - w_B^{w-w'}$, we can (c) set $low = w - w_B^{w-w'}$.

$c_2 := \mathsf{layout}(T_2, w - \mathsf{w}(c_1))$
$c' := \mathsf{vert}(c_1, c_2)$
if $(\mathsf{h}(c') < \mathsf{h}(c))$ $c := c'$
if $(\mathsf{h}(c_1) \leq \mathsf{h}(c_2))$ $high := \mathsf{w}(c_1) - 1$
if $(\mathsf{h}(c_1) \geq \mathsf{h}(c_2))$ $low := \max(w' + 1, w - \mathsf{w}(c_2))$

EXAMPLE 3.2. *Consider again the problem described in Example 3.1. The root node is a vertical cut, so we must pick a cut position. Since* $w^{H_1}_{min} = w^{Z}_{min} = 1$*, the cut must be in the range* $[1, 2]$.

We choose the initial cut as $w' = 1$*. The sequence of calls made is as follows:*

$f(V_0, 3)$
  $w' = 1$
    $f(H_1, 1)$
      $f(X, 1)$
        $\rightarrow (1, 2)$
      $f(Y, 1)$
        $\rightarrow (1, 2)$
      $\rightarrow (1, 4)$
    $f(Z, 2)$
      $\rightarrow (2, 2)$
    $\rightarrow (3, 4)$

The best solution found so far is $(3, 4)$*. Since the height of* H_1 *is greater than the height of* Z*, we know an improved solution can only be to the right of the current cut. We update* $low := 2$*, and continue:*

  $w' = 2$
    $f(H_1, 2)$
      $f(X, 2)$
        $\rightarrow (2, 1)$
      $f(Y, 2)$
        $\rightarrow (2, 1)$
      $\rightarrow (2, 2)$
    $f(Z, 1)$
      $\rightarrow (1, 3)$
    $\rightarrow (3, 3)$
  $\rightarrow (3, 3)$

Finding the optimal solution at $w' = 2$*, giving configuration* $(3, 3)$. □

4. FREE GUILLOTINE LAYOUT

In this section we consider the more difficult problem of free guillotine layout. Given a set of leaves (say, newspaper articles), we want to construct the optimal tree of cuts such that all leaves are used, and the overall height is minimized. Both the top-down and bottom-up construction methods given in the last section for fixed-cut guillotine layout can be readily adapted to solving the free layout problem.

The structure of the bottom-up algorithm remains largely the same. To compute the minimal configurations for a set S', we try all binary partitionings of S' into S'' and $S' \setminus S''$. We then generate the configurations for $\textsc{vert}(S' \setminus S'', S'')$ and $\textsc{horiz}(S' \setminus S'', S'')$ as for the fixed problem. However, we must then eliminate any non-minimal configurations that have been generated. This is done by merge, which merges two sets of minimal configurations. Pseudo-code for this process is given in Figure 10. As we need to generate all configurations for all $2^{|S|}$ subsets of S, we construct the results for subsets in order of increasing size.

```
layout_free_bu(S,w)
  for(c ∈ {2,...,|S|})
    for(S' ⊆ S, |S'| = c)
      C(S) := ∅
      e := min i ∈ S'
      for(S'' ⊂ S' \ {e})
        C(S') := merge(C(S'),
                       horiz_configs(C(S' \ S''), C(S''), w))
        C(S') := merge(C(S'),
                       vert_configs(C(S' \ S''), C(S''), w))
  return C(S)[|C(S)| − 1]

merge(CX, CY)
  C := []
  i := 0
  j := 0
  while (i ≤ |CX| ∧ j ≤ |CY|)
    (w_x, h_x) := CX[i]
    (w_y, h_y) := CY[j]
    if(w_x < w_y)
      if(h_x > h_y)
        C := C ++[ CX[i] ]
        i := i + 1
      else j := j + 1
    else if(w_x > w_y)
      if(h_x < h_y)
        C := C ++[ CY[j] ]
        j := j + 1
      else i := i + 1
    else % w_x = w_y
      if(h_x ≤ h_y) j := j + 1
      else i := i + 1
  while (i ≤ |CX|)
    C := C ++[ CX[i] ]
    i := i + 1
  while (j ≤ |CY|)
    C := C ++[ CY[j] ]
    j := j + 1
  return C
```

Figure 10: Pseudo-code for a bottom-up construction approach for the free guillotine-layout problem for articles S. The configurations $C(S')$ for $S' \subseteq S$ are constructed from those of $C(S' \setminus S'')$ and $C(S'')$ where $S' \setminus S''$ and S'' are non empty and the first set is lexicographically smaller than the second.

```
layout_free(S,w)
    c := lookup(S,w)
    if c ≠ NotFound return c
    if (S = {A})
        c := C(A)[i] where i is maximal s.t. w(C(A)[i]) ≤ w
    else
        e := min(S)
        c := (0, ∞)
        for S' ⊂ S \ {e}
            L := {e} ∪ S'
            R := S \ L
            % Try a horizontal split
            c' := horiz(layout_free(L, w), layout_free(R, w))
            if(h(c') ≤ h(c)) c := c'
            % Find the optimal vertical split
            low := w^L_min
            high := w − w^R_min
            while(low ≤ high)
                w' := ⌊(low+high)/2⌋
                c_l := layout_free(L, w')
                c_r = layout_free(R, w − w(c_l))
                c' := vert(c_l, c_r)
                if(h(c') ≤ h(c)) c := c'
                if(h(c_l) ≤ h(c_r)) high := w(c_l) − 1
                if(h(c_l) ≥ h(c_r))
                    low := max(w' + 1, w − w(c_r))
    cache(S,w,c)
    return c
```

Figure 11: Basic top-down dynamic programming for the free guillotine layout problem.

For the top-down method, at each node we want to find the optimal layout for a given set S and width w. To construct the solution, we try all binary partitions of S. Consider a partitioning into sets S' and S''. As there are a large number of symmetric partitionings, we enforce that the minimal element of S must be in S'. We then try laying out both $\textsc{vert}(S', S'')$ and $\textsc{horiz}(S', S'')$, picking the best result.

Pseudo-code for the top-down dynamic programming approach is given in Figure 11. The structure of the algorithm is very similar to that for the fixed layout problem, except it now includes additional branching to choose binary partitions of S and try both cut directions. As before, w^S_{min} indicates narrowest feasible width for laying out S. This is calculated by taking the the widest minimum configuration width for any node in S.

4.1 Bounding

The dynamic program as formulated has a very large search space. We would like to reduce this by avoiding exploring branches containing strictly inferior solutions. We can improve this if we can calculate a lower bounds $\mathsf{lb}(S, w)$ on the height of any configuration for S in width w. If h_{max} is the best height so far and $\mathsf{lb}(S, w) \geq h_{max}$, we know the current state cannot be part of any improved optimal solution, so we can simply cut-off search early with the current bound. This is a form of bounded dynamic programming [10].

For the minimum-height guillotine layout problem, we compute the minimum area used by some configuration of each leaf. This allows us to determine a lower bound on the area required for laying out the set of articles S. Since any valid layout must occupy at least $\mathsf{area}(S)$, a layout with a fixed width of w will have a height of at least $\left\lceil \frac{\mathsf{area}(S)}{w} \right\rceil$.

We can also use the area approximation to reduce the set of vertical splits that must be explored. If we have a current best height h_{max}, any cut for $\textsc{vert}(X, Y)$ where $w' \leq \left\lceil \frac{\mathsf{area}(X)}{h_{max}} \right\rceil$ or $w' \geq w - \left\lceil \frac{\mathsf{area}(Y)}{h_{max}} \right\rceil$ cannot give an improved solution. Pseudo-code for the bounded dynamic programming approach is given in Figure 12. Note that configurations are now given as a triple (w_i, h_i, e_i), where $e_i \in \{\mathsf{true}, \mathsf{false}\}$ indicates whether the configuration is exact ($e_i = \mathsf{true}$), or a lower bound ($e_i = \mathsf{false}$). We use $\mathsf{e}(c)$ to extract the third component of a configuration.

A final optimization is to note that if we find a configuration c which has height equal to the lower bound ($\mathsf{h}(c) = \left\lceil \frac{\mathsf{area}(S)}{w} \right\rceil$) we can immediately return this solution.

5. UPDATING LAYOUTS

In the interaction model proposed in the introduction, we suggested using a fixed-cut layout to relayout an article during user interaction, until the current fixed-cut leads to a very bad layout. Bad layout can be for two reasons. The first reason is that current choice of guillotining does not allow a layout for the desired width while a different choice of guillotining will. The second reason is that the choice of guillotine leads to quite un-compact layout and so to a page height that is unnecessarily large. We note that un-compact layout is typically more of a problem for page dependent column layout. In the case that the current fixed-cut leads to bad layout we wish to modify the guillotining to give a layout close to the current layout.

First, we must determine how bad a layout can be before we relayout the document.

Given a set of articles S, we can precompute the optimal layout for a set of given widths W using `layout_free_bu` or `layout_free`. We can then build a piecewise linear approximation $\mathsf{approx\_height}(S, w)$ to the minimal height for free layout of S for width w for all possible widths. However, since the optimal layout is generally close to the area bound, we can use the simpler approximation $\mathsf{approx\_height}(S, w) = \left\lceil \frac{\mathsf{area(S)}}{w} \right\rceil$. We use this function to determine when to change guillotine cuts during user interaction. Assume the current layout of S is T, then if $\mathsf{layout}(T, w) > \alpha \times \mathsf{approx\_height}(S, w)$ we know that the fixed-cut is giving poor layout. We use $\alpha = 1.1$.

When we are generating a new guillotine cut for S, we want to ensure that the new layout is "close" to the current cut T. Our approach is to try and change the guillotining only at the bottom of the current cut T. Define the tree height of a tree T as follows: $\mathsf{theight}(\textsc{cell}(A)) = 0$, $\mathsf{theight}(\textsc{vert}(T_1, T_2)) = \max(\mathsf{theight}(T_1), \mathsf{theight}(T_1)) + 1$, $\mathsf{theight}(\textsc{horiz}(T_1, T_2)) = \max(\mathsf{theight}(T_1), \mathsf{theight}(T_1)) + 1$. We first try to modify only subtrees with tree height 1 (that is parents of leaf nodes). If that fails to improve the current layout enough we modify subtrees of tree height 2, etc. Psuedo-code for the interactive layout problem is given in Figure 13.

6. EXPERIMENTAL RESULTS

To evaluate the methods described in Sections 3 to 4, we required a set of documents suitable for guillotine layout.

```
layout_free_bnd(S,w,h_max)
    c := lookup(S,w)
    if(c ≠ NotFound)
        if(e(c) ∨ h(c) ≥ h_max)
            return c
    % If the bound is greater than
    % h_max, we can stop early
    if (⌈area(S)/w⌉ ≥ h_max)
        c := (w, ⌈area(S)/w⌉, false)
    if (S = {A})
        i := maximal i' s.t. w(C(A)[i']) ≤ w
        c := (w(c'), h(c'), h(c') ≤ h_max) where c' = C(A)[i]
    else
        e := min(S)
        c := (0, ∞)
        for S' ⊂ S \ {e}
            L := {e} ∪ S'
            R := S \ L
            A_l := area(L)
            A_r := area(R)
            % Try a horizontal split
            c_l := layout_free_bnd(L, w, h_max − ⌈A_r/w⌉)
            c_r := layout_free_bnd(R, w, h_max − h(c_l))
            if(h(c_l) + h(c_r) ≤ h(c))
                c := (max(w(c_l), w(c_r)), h(c_l) + h(c_r), e(c_l) ∧ e(c_r))
                if(h(c) = ⌈area(S)/w⌉) break
            % Ensure a vertical split is feasible
            if(w^L_min + w^R_min > w)
                continue
            h^vert_min := max(⌈A_l/(w − w^R_min)⌉, ⌈A_r/(w − w^L_min)⌉)
            if(h^vert_min > h_max)
                if (h^vert_min < hc) c := (w, h^vert_min, false)
                continue
            % Find the optimal vertical split
            low := max(w^L_min, ⌈A_l/h_max⌉)
            high := w − max(w^R_min, ⌈A_r/h_max⌉) + 1
            while(low < high)
                w' := ⌊(low+high)/2⌋
                c_l := layout_free_bnd(L, w', h_max)
                if(h(c_l) ≥ h_max)
                    c_r := (w − w(c_l), ⌈A_r/(w − w(c_l))⌉, false)
                else
                    c_r := layout_free_bnd(R, w − w(c_l), h_max)
                if(max(h(c_l), h(c_r)) ≤ h(c))
                    c := (w(c_l) + w(c_r), max(h(c_l), h(c_r)), e(c_l) ∧ e(c_r))
                    if(h(c) ≤ h_max) h_max := h(c)
                if(h(c_l) ≤ h(c_r)) high := h(c_l) − 1
                if(h(c_l) ≥ h(c_r))
                    low := max(w' + 1, w − w(c_r))
                if(h(c) = ⌈area(S)/w⌉) break for
    cache(S,w,c)
    return c
```

Figure 12: Pseudo-code for the bounded top-down dynamic programming approach. Note that while bounding generally reduces search, if a previously expanded state is called again with a more relaxed bound, we may end up partially expanding a state multiple times.

```
interact(T,w)
    c := layout(T,w)
    S := articles in T
    k := 1
    while(h(c) > α × approx_height(S, w))
        c := relayout(T,w,k)
        k := k + 1
    return c

relayout(T,w,k)
    c := lookup(T,w)
    if(c ≠ NotFound) return c
    if(theight(T) ≤ k)
        S := set of articles appearing in T
        return layout_free(S,w)
    switch (T)
    case cell(A):
        c := C(A)[i] where i is maximal s.t. w(C(A)[i]) ≤ w
    case horiz(T_1, T_2):
        c := horiz(relayout(T_1, w, k), relayout(T_2, w, k))
    case vert(T_1, T_2):
        c := (0, ∞)
        for w' = 0..w
            c' := vert(relayout(T_1, w', k), relayout(T_2, w − w', k))
            if (h(c') < h(c)) c := c'
        endfor
    endswitch
    cache(T,w,c)
    return c
```

Figure 13: Pseudo-code for the basic top-down dynamic programming relayout, where we can change configuration for subtrees with tree height less than or equal to k.

n	td	td+b	BU
10	**< 0.01**	**< 0.01**	**< 0.01**
20	**< 0.01**	**< 0.01**	**< 0.01**
30	0.02	0.01	**< 0.01**
40	0.03	0.02	**< 0.01**
50	0.06	0.05	**< 0.01**
60	0.13	0.10	**< 0.01**
70	0.18	0.14	**< 0.01**
80	0.29	0.22	**< 0.01**
90	0.41	0.33	**< 0.01**

Table 1: Results for the fixed-cut minimum-height guillotine layout problem with $w = w_{min} + 0.1(w_{max} - w_{min})$.

To construct this data-set, we randomly select a set of n of articles from the REUTERS-21578 news corpus [1], then use a modified version of the binary search described in [9] to determine the set of available configurations for each article. All times are given in seconds, and all experiments are run with a time limit of 600 seconds. Times given are averages over 10 instances of each problem size. Bold entries indicate the best result for each problem size.

For convenience in generating the dataset, we assume the use of a fixed-width font. While A-series page sizes have a $1 : \sqrt{2}$ aspect ratio, fixed-width fonts fit approximately equal number of lines as characters per line.

All experiments were conducted on a 3.00Ghz Core2 Duo with 2 Gb of RAM running Ubuntu GNU/Linux 8.10. td denotes the top-down dynamic programming approach, and td+b is top-down dynamic programming with bounding. bu denotes bottom-up construction.

6.1 Fixed-Cut Layout

Instances for fixed-cut guillotine layout were constructed with a random tree of cuts, selecting horizontal and vertical cuts with equal probability. Initially, we selected the instance width as a linear combination of the minimum and maximum width configurations for the instance. Results given Table 1 are constructed with $w = w_{min} + 0.1(w_{max} - w_{min})$, where w_{min} is the overall width when each article takes the narrowest feasible configuration (and similarly for w_{max}). Clearly, the top-down methods degrade quite rapidly compared to the bottom-up method. This appears to be due more to the rapidly increasing width than the increasing number of articles; the instances with 90 articles are laid out on a page that is 3000 to 5000 characters wide. This is illustrated in Table 2, where we calculated $w = w_{min} + 300$. Although the top-down methods are still distinctly slower than the bottom-up approach, they now scale far more gracefully.

6.2 Free Layout

For the free layout problem, we constructed instances for each size between 4 and 15. The instance width was selected as $\left\lceil \sqrt{(1+\alpha)\mathsf{area}(S)} \right\rceil$, to approximate a layout on an A-series style page with α additional space. For these experiments, we selected $\alpha = 0.2$. Times given in Table 3 denote the average time for solving the 10 instance of the indicated size.

As before, td performs significantly worse than the other

n	td	td+b	BU
10	**< 0.01**	**< 0.01**	**< 0.01**
20	**< 0.01**	**< 0.01**	**< 0.01**
30	0.01	**< 0.01**	**< 0.01**
40	0.01	**< 0.01**	**< 0.01**
50	0.02	0.01	**< 0.01**
60	0.03	0.01	**< 0.01**
70	0.03	0.01	**< 0.01**
80	0.04	0.01	**< 0.01**
90	0.05	0.02	**< 0.01**

Table 2: Results for the fixed-cut minimum-height guillotine layout problem with $w = w_{min} + 300$.

n	td	td+b	bu
4	**< 0.01**	**< 0.01**	**< 0.01**
5	**< 0.01**	**< 0.01**	**< 0.01**
6	0.01	**< 0.01**	**< 0.01**
7	0.03	**< 0.01**	0.01
8	0.12	**0.01**	0.04
9	0.42	**0.03**	0.15
10	1.62	**0.11**	0.50
11	5.81	**0.41**	1.64
12	22.33	**1.50**	5.02
13	106.46	**6.09**	17.51
14	413.63	**24.43**	53.11
15	—	**74.84**	143.83

Table 3: Results for the free minimum-height guillotine layout problem. Times (in seconds) are averages of 10 randomly generated instances with n articles.

n	td	td+b	bu
4	**< 0.01**	**< 0.01**	**< 0.01**
5	**< 0.01**	**< 0.01**	**< 0.01**
6	**< 0.01**	**< 0.01**	**< 0.01**
7	**< 0.01**	**< 0.01**	**< 0.01**
8	**< 0.01**	**< 0.01**	**< 0.01**
9	0.01	**< 0.01**	0.01
10	0.04	**< 0.01**	0.05
11	0.12	**< 0.01**	0.16
12	0.39	**0.01**	0.47
13	1.23	**0.04**	1.53
14	3.85	**0.10**	4.38
15	13.10	**0.37**	15.50
16	41.65	**0.50**	48.32
17	168.19	**1.17**	157.96
18	590.76	**3.80**	442.19

Table 4: Results for the free minimum-height guillotine layout problem using page dependent column-based layout. Times (in seconds) are averages of 10 randomly generated instances with n articles.

methods. Unlike the fixed layout problem, these instances have much narrower page widths, and the search space arises largely from the selection of binary partitions. As a result, bounding provides a substantial improvement – td+b is consistently around twice as fast as bu on these instances.

In this first experiment we did not use column-based layout. However, in practice column-based layout is preferable so as to avoid long text measures. We generated test data for page dependent column-based layouts in a similar manner to the other guillotine layouts; having selected a column width, we calculate the number of lines required for the article body, and use this to determine the dimensions given a varying number of columns. This is combined with the layout for the article title (calculated as before).

We select a column width of 38 characters, chosen as being typical of print newspapers. Page width is selected as before, then rounded up to the nearest number of columns. Results for this dataset are given in Table 4. The results for this case differ substantially to those for the non column-based instances – since the number of possible vertical cuts is much smaller (even the large instances generally have only 4 columns) fewer subproblems need to be expanded at each node during the execution of the dynamic programming approaches. In this case, td actually slightly outperforms bu, but td+b is substantially faster than both.

6.3 Updating Layouts

In practice, for non page dependent column-based layouts, a fixed optimal cutting remains near-optimal over a wide range of width values. To illustrate this, we took a document with 13 articles from the set used in Section 6, and computed the optimal cutting for $w = 200$. Figure 14 shows the height given by laying out this fixed cutting using layout with widths between 40 and 200. We compare this with the height given by the area bound and the optimal layout for each width. While the fixed layout is quite close to the optimal height over a wide range of values, it begins to deviate as we decrease the viewport width. For widths 40 and 50, this fixed layout is infeasible, and we are forced to compute a new tree of cuts.

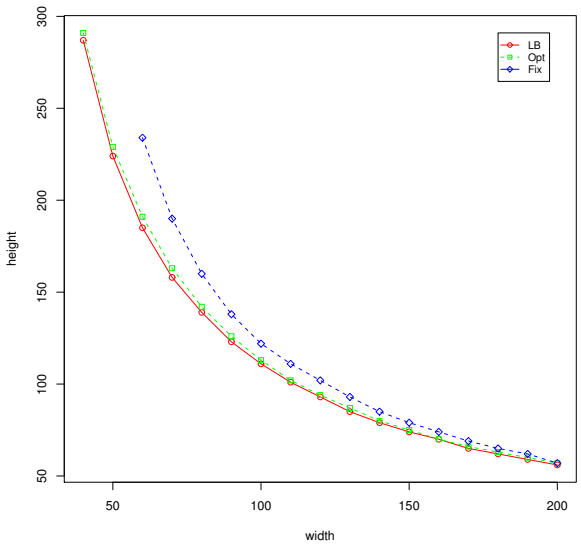

Figure 14: Layout heights for a 13-article document used in Section 6. LB the lower bound at the given width, and OPT is the minimum height given by layout_free_bnd. For FIX, we computed the optimal layout for $w = 200$, and adjusted the layout to the desired with using layout.

To test the performance of the relayout algorithm, we consider again the set of 13-article documents used in the previous experiment. We computed the optimal layout for page widths between 40 and 200 characters, in 5 character intervals. We compared this with adapting the fixed layout computed for $w = 40$, and progressively used relayout at each width. relayout was implemented with the bounded top-down methods for both the fixed and free components.

The average runtime for layout_free_bnd over the varying documents and widths was 7.48s. Runtime for layout was less than 0.01s in all cases, but deviated from the minimal height by up to 40%. Average runtime for relayout (with $\alpha = 1.1$) was 0.02s, and deviated from the minimal height by at most 10%. Results for page dependent column-based layout are similar. For documents with 16 articles, layout generated layouts up to 32% taller than the optimum; layout_free_bnd took 0.48s on average, compared to less than 0.01 for relayout (and $\alpha = 1.1$).

7. CONCLUSION

Guillotine-based layouts are widely used in newspaper and magazine layout. We have given algorithms to solve two variants of the automatic guillotine layout problem: the fixed cut guillotine layout problem in which the choice of guillotine cuts is fixed and the free guillotine layout problem in which the algorithm must choose the guillotining. We have shown that the fixed guillotine layout problem is solvable in polynomial time while the free guillotine layout problem is NP-Hard.

We have presented bottom-up and top-down methods for the minimum-height guillotine layout problem. For fixed-cut guillotine layout, the bottom-up method is far superior, as complexity is dependent only on the number of leaf configu-

rations, rather than the page width; the bottom-up method can optimally layout reasonable sized graphs in real-time.

For the free guillotine layout problem, which has smaller width and larger search space, the bounded top-down method was substantially faster than the other methods. On instances with arbitrary cut positions, the bounded top-down method could solve instances with up to 13 articles in a few seconds; when restricted to page dependent column-based layouts, we can quickly produce layouts for at least 18 articles.

We have also suggested a novel interaction model for viewing on-line documents with a guillotine-based layout in which we solve the free guillotine layout problem to find an initial layout and then use the fixed cut guillotine layout to adjust the layout in response to user interaction such as changing the font size or viewing window size.

Currently our implementation only handles text. Future work will be to incorporate images.

8. ACKNOWLEDGEMENTS

The authors acknowledge the support of the ARC through Discovery Project Grant DP0987168.

9. REFERENCES

[1] Reuters-21578, Distribution 1.0. `http://www.daviddlewis.com/resources/testcollections/reuters21578`.

[2] R. Alvarez-Valdés, A. Parajón, and J. M. Tamarit. A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems. *Computers & OR*, 29(7):925–947, 2002.

[3] C. B. Atkins. Blocked recursive image composition. In *Proceedings of the 16th International Conference on Multimedia 2008*, pages 821–824, 2008.

[4] N. Christofides and E. Hadjiconstantinou. An exact algorithm for orthogonal 2-d cutting problems using guillotine cuts. *European Journal of Operational Research*, 83(1):21–38, 1995.

[5] N. Christofides and C. Whitlock. An algorithm for two-dimensional cutting problems. *Operations Research*, pages 30–44, 1977.

[6] J. González, J. Merelo, P. Castillo, V. Rivas, and G. Romero. Optimizing web newspaper layout using simulated annealing. In J. Mira and J. Sanchez-Andres, editors, *Engineering Applications of Bio-Inspired Artificial Neural Networks*, volume 1607 of *Lecture Notes in Computer Science*, pages 759–768. Springer Berlin / Heidelberg, 1999.

[7] N. Hurst. *Better Automatic Layout of Documents*. PhD thesis, Monash University, Department of Computer Science, May 2009.

[8] N. Hurst, W. Li, and K. Marriott. Review of automatic document formatting. In *Proceedings of the 9th ACM symposium on Document engineering*, pages 99–108. ACM, 2009.

[9] N. Hurst, K. Marriott, and P. Moulder. Minimum sized text containment shapes. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 3–12, New York, NY, USA, 2006. ACM.

[10] J. Puchinger and P. Stuckey. Automating branch-and-bound for dynamic programs. In R. Glück and O. de Moor, editors, *Proceedings of the ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation (PEPM '08)*, pages 81–89. ACM, 2008.

[11] T. Strecker and L. Hennig. Automatic layouting of personalized newspaper pages. In B. Fleischmann, K.-H. Borgwardt, R. Klein, and A. Tuma, editors, *Operations Research Proceedings 2008*, pages 469–474. Springer Berlin Heidelberg, 2009.