Satisficing Scrolls: A shortcut to satisfactory layout

Nathan Hurst
Adobe Systems Inc.
345 Park Ave.
San Jose, CA 95110
hurst@adobe.com

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

ABSTRACT

We present at a new approach to finding aesthetically pleasing page layouts. We do not aim to find an optimal layout, rather the aim is to find a layout which is not obviously wrong. We consider vertical scroll-like layout with floating figures referenced within the text where floats can have alternate sizes, may be optional, move from one side to the other and change their order. We also allow pagination. Our approach is to use a randomised local search algorithm to explore different configurations of floats, i.e. choice of floats and relative ordering. For a particular float configuration we use an efficient gradient projection-like continuous optimization algorithm. The resulting system is fast and provides an efficient warm start option to improve interactive support.

Categories and Subject Descriptors

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

General Terms

Algorithms

Keywords

optimisation techniques, floating figure, multi-column layout

1. INTRODUCTION

Look at almost any web page and you will see a vertical scroll layout. Without the need for separate pages it makes sense to display documents in a fixed width space based on the available screen space. A common problem in such a layout is the placement of floating figures, i.e. floats. Existing algorithms for scroll layout either use a first-fit algorithm to place floats, such as those used in web browsers or use an exhaustive graph search to find an optimal layout.

First-fit placement is fast and people are experienced with the output that results. However, it does not always adapt well to different viewing areas. Graph search algorithms can potentially handle arbitrary layout rules and optimisation criteria [13, 3, 10, 12], but generally require exponential run time for all but the most restrictive kinds of layout. This search cost can be reduced by using incomplete heuristic search or windowing [12, 10] at the cost of optimality. But, if we are willing to consider incomplete search, how can we decide what is good enough?

Humans rarely seek a global optimal solution to a problem. Instead, they seek a solution which satisfies the core constraints, and is sufficient—a *satisficing solution* in the words of Herbert Simon [15]. For layout we prefer the term *layouts without obvious errors* (LWOE): in other words, a solution to which a human would not be able to immediately suggest an improvement.

As an exploration of this idea, we propose an alternative approach to layout which tries to emulate a human searching for, and through, candidate solutions. When constructing a document by hand, a designer will first sketch an initial layout using something similar to a first fit algorithm, then make a series of refinements to the layout. Sometimes, under refinement the layout becomes clearly wrong. At this point the human considers rearranging things by making local changes to the layout, swapping floats, resizing elements, and adding or removing optional images.

This paper presents a method which uses a randomised local search algorithm to explore different configurations of floats, i.e. choice of floats and relative ordering. While local search algorithms may not return a global solution, they do return a local optimum. Such local optimality implies LWOE, since it means that there is no simple way to change the layout and improve the quality. Relaxing the requirement to find a global optimum to one of find a local optimal means that we can increase the generality of the model to allow complex constraints and objectives but still keep efficiency. For a particular float configuration we use an efficient gradient projection-like continuous optimization algorithm. This is related to techniques used for graph layout [4] and relies on text behaving like a liquid [8].

2. MODEL

We first describe our vertical scroll layout model. A document consists of a single narrative thread or `body` of text with linked floats. The body text is formatted text providing different fonts, multiple language support and other standard typographic features.

The underlying layout model is that the canvas has a fixed width and infinitely expandable height. Floats are placed on the canvas and must not overlap. In general, floats can be left aligned, right aligned or centered, but the author can

DocEng'08, September 16–19, 2008, São Paulo, Brazil.

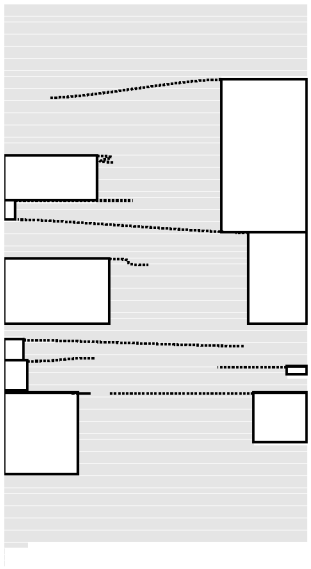


Figure 1: Example vertical scroll layout. The dashed lines show the anchor point connected to each float.

specify for each float the allowed placement styles. The body text flows into the space not occupied by floats. However, we do not allow narrow stretches of text so as to avoid bad line-breaking. The designer can specify a minimum width for text, and any point where the width is less than this is filled with white space. (In this case perhaps a better layout can be found by letting a float expand, or become centred).

Floats do not necessarily have a fixed order. However, the floats may have a partial order $\leq$ on their placement in the document. For example, Weitzman and Wittenburg [16] examine the layout of a DIY guide. In such a guide the construction order is important, but may not need to be strongly tied to the text. We require that the layout respects the partial order. That is to say, for all floats f, f', if $f \leq f'$ then the top of f is above or equal to the top of f'.

The floats are connected to *anchor regions* defined in the body. Currently an anchor region is required to be a single point in the body text although in the future we plan to allow anchor regions to be a paragraph, section or range. Many floats have a one-to-one relationship with anchors, but we also allow a set of floats to be associated with an anchor. In this case, the layout can contain a subset of floats from the set chosen to meet some aesthetic criteria such as having an equal amount of space for floats and text. For example, in an article on castles we may have fifty good pictures to choose from, some tall, some wide, some square, and we pick a few so as to provide "colour" but not overwhelm the layout.

Not only should floats be placed, but we want them to resize as well. We want to be able to specify alternatives for each float, with a penalty associated with the different choices.

Some floats, inline or floating, may be better placed in a group and formatted together, rather than independently. For example, if there are three pictures of the same height, it would look good to put all three in a line, spaced evenly. In this case our algorithm treats them as a single float.

The layout algorithm should, as far as possible, avoid whitespace at the bottom of the scroll, moving floats further up the document to push the text in line with the last float.

We need a penalty function to penalize the distance between floats and anchors. We use the quadratic penalty function

$$\sum_{f \in F} f_w (f_y - a_y^f)^2$$

where F is the set of floats in the layout, f_w reflects the importance of placing the float close to its anchor (we allow a 0 weight for delocalised floats), f_y is the vertical position of the float's top and a_y^f is the vertical position of the top of the anchor of f.

Allowing floats to be reordered means that our layout problem is NP-hard:

VERTICAL SCROLL LAYOUT WITH PARTIALLY ORDERED FLOATS (VSL)

INSTANCE: Rectangular floats F with a partial ordering $\leq$, where for each float $f \in F$ has a non-empty set of allowed placement styles: left aligned, right aligned or centered, and a weight $w_f \geq 0$; text T which contains an anchor point a^f for each float $f \in F$; scroll width $W \geq 0$ and penalty $S \geq 0$. QUESTION: Is there a placement for the floats on a rectangular canvas of width W s.t. the floats do not overlap, the layout respects the partial ordering, the placement style for each float is an allowed style and, if the text T is laid out from the top of the scroll flowing around the floats,

$$\sum_{f \in F} f_w (f_y - a_y^f)^2 \leq S?$$

PROPOSITION 2.1. *The VSL problem is NP-hard.*

Proof: There is a simple reduction from the PARTITION problem. Recall that in this problem we have a finite set P and a size $s(p) > 0$ for each $p \in P$, and must determine if there is a subset $P' \subseteq P$ s.t.

$$\sum_{p \in P'} s(p) = \sum_{p \in P \setminus P'} s(p).$$

We can model this using VSL as follows. The scroll width is 2. There are a top and bottom float T and B with height 1 and width 2 as well as a float f^p for each $p \in P$ with width 1 and height $s(p)$. The partial ordering is given by $B \leq f^p$ and $f^p \leq T$ for all $p \in P$. Each f^p can be right or left aligned and has weight 0 while floats T and B are centered and have weight w_T and w_B respectively where $w_T >> w_B$. The text contains a single word w of height 1 and length 2 which is the anchor point for all of the floats. It follows from the construction that a layout of minimal penalty will look like that shown in Figure 2. Furthermore, the penalty will be $w_T + (1 + h)w_B$ where the floats f^p are arranged in two side-by-side columns of height h. If we choose the penalty to be

$$S = w_T + (1 + \frac{\sum_{p \in P} s(p)}{2})w_B$$

we have that the original PARTITION problem is satisfied iff this VSL problem is satisfiable. Clearly the reduction is polynomial, so the VSL problem is NP-hard since PARTITION is NP-hard [6]. $\square$

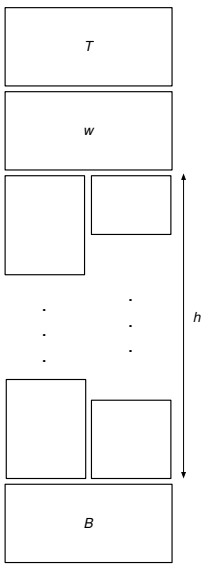


Figure 2: Encoding PARTITION as a VSL problem.

3. FLOAT PLACEMENT WITH SEPARATION CONSTRAINTS

It follows from Proposition 2.1 that, unless $P = NP$, there is no polynomial algorithm for finding an optimal vertical scroll layout. However, the reason for the NP-hardness is that floats can be placed in a number of different placement styles, can be reordered and also we can choose a subset of floats to include in the layout. In this section, we first consider a simpler problem: float placement where the choice of floats and placement style for each float is fixed and we know the relative vertical ordering between floats that have a horizontal overlap. We call this a *float configuration*.

Finding an optimal layout for a given float configuration is possible in polynomial-time with certain reasonable restrictions on the allowed layout using a dynamic programming algorithm [9, 12]. However, such an approach is quite slow and does not allow us to explore multiple different float configurations in a reasonable time. Instead, we have developed a fast algorithm for finding a near-optimal layout for a particular float configuration. This uses an iterative gradient projection-like optimization algorithm [1] and is related to constrained graph layout techniques [4].

Since the choice of floats and their placement style is fixed, the only variables in the layout are the vertical position, f_y, for each float $f \in F$. Since we know the relative vertical ordering between floats that have a horizontal overlap we can generate a set C of so-called *separation constraints* of form $f_y + g \leq f'_y$ that specify a minimum gap g between pairs of floats to ensure that the relative ordering is preserved and that the floats do not overlap. Assume that we have a current guess at the placement of the anchor region a^f_y for each float. We want to find values for the f_y which minimize $\sum_{f \in F} f_w(f_y - a^f_y)^2$ subject to the separation constraints C. Such a problem is a kind of quadratic program and can be solved efficiently using VPSC, an active-set method described in Dwyer et al [5]. This works by merging variables whose constraints are violated into larger and larger "blocks" of variables constrained with equality. By performing this operation in an efficient order, and having a closed form for both the minimisation of equality constraints and the computation of the Lagrangian, the algorithm is very efficient.

Our float placement with separation constraints (FPSC) algorithm essentially works as follows. Find an initial position for floats and text say using a first-fit algorithm. Update the position of the floats using VPSC and the current position of the anchor regions as the desired position for each float, now flow text around the floats to find the new position of the anchor regions and the layout penalty. This step is repeated until changing the layout does not lead to sufficient improvement.

compute initial values for f_y and a^f_y
repeat
  use VPSC to compute new values f'_y for f_y
  (using a^f_y as the desired position for f_y)
  place floats at f'_y
  layout text around floats to compute new values of a^f_y
  compute penalty
until insufficient improvement

Figure 3: The Optimistic FPSC Algorithm.

For efficiency, when computing the new positions for the anchor regions a^f_y we do not actually lay the text out but rather approximate the text by a continuous liquid with the same area as the text [8]. The algorithm is given in Figure 3.

Let us consider an idealised example of a single float in a column of width 1. Figure 5 shows the execution of the Optimistic FPSC Algorithm. Figure 5(a) shows eight steps in the execution with a float of 0.1 wide and 1 unit long with the float initially placed a long way from the anchor. The small circle shows the (continuous) anchor position. The algorithm converges nicely[1], and in fact steps almost to the correct position in the first step. This is not surprising as VPSC steps exactly to the solution for a float of width 0.

However, Figure 5(b) shows the execution for a float of width 0.5. In this case the algorithm doesn't converge, and in between these two widths we have worsening convergence—0.4 oscillates a lot before reaching the minimum. If we step through with exact arithmetic it confirms that the algorithm cycles, as does any larger float width. We also see that the penalty of the successive layouts can increase.

We can tackle this by forcing the algorithm to make progress. At every update we look to see whether the true error increased, and if so, take a smaller step. This is a common approach in multidimensional non-linear minimisation algorithms, called a backtracking line search [1]. Figure 4 shows the modifications required and result in the Careful FPSC Algorithm. Figures 5(c) and 5(d) shows the execution of this modified algorithm on wider floats—convergence is now comparable to that of the narrow floats, and surprisingly, in some cases finds the exact solution in just a few steps.

Finally, what if the float fills the whole column? In this case there are in fact two equally bad solutions, one with the anchor just before the float and one with the anchor just after. Furthermore, allowing floats this wide means that the penalty is no longer continuous in the float placement, since

[1]It actually converges linearly, but with an error improvement of 3×10^{-3} at each step.

compute initial values for f_y and a_y^f
repeat
  use VPSC to compute new values f'_y for f_y
  (using a_y^f as the desired position for f_y)
  $\alpha = 1$
  repeat
    place floats at $\alpha f'_y + (1-\alpha) f_y$
    layout text around floats to compute new values of a_y^f
    compute penalty
    $\alpha = \alpha/2$
  until α very small or sufficient improvement
until insufficient improvement

Figure 4: The Careful FPSC Algorithm.

moving a float infinitesimally upwards can move an anchor from above the float to below it. The problem is that our text liquid cannot fit next to the float. However, this suggests a simple, if crude, fix. We simply insert an infinitesimal gap around the float and solve with the same algorithm. We can actually handle this gap without resorting to tiny ϵ changes to the sizes by keeping track of the infinitesimal gap in the VPSC code. Figure 5(e) shows steps taken by the Careful FPSC Algorithm with the attendant cyclic behaviour[2], Figure 5(f) shows the result of adding an infinitesimal gap. It is tempting to instead allow multiple discrete anchor points above and below the figures for such problems, but doing so creates an additional discrete search problem and doesn't handle the case where two figures coincidentally result in completely filled column.

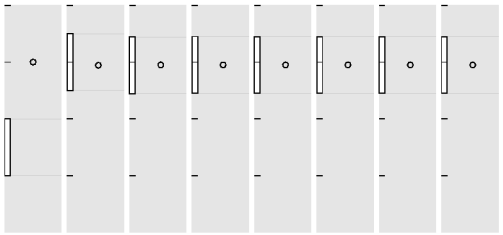
For efficiency, in our actual implementation of the Careful FPSC algorithm we do not update the entire layout at each step. The point is that if a "block" of floats is separated by complete lines of text from the other floats then the floats in the block can be optimized in isolation. In our implementation we consider the blocks in turn starting from the top of the layout. The difficulty is that during optimization a block may merge with a previously placed block requiring the entire group of floats to be optimized at once.

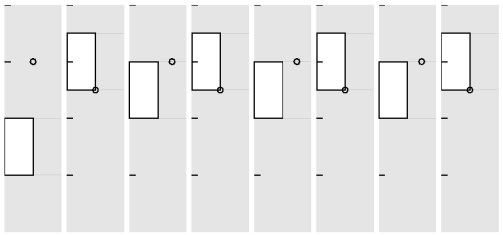
This algorithm improves on HTML's float placement in a number of ways. Firstly, it allows groups of floats to spread more evenly around around their anchor points. Secondly, it allows us to add weights to floats to give priority to important references.

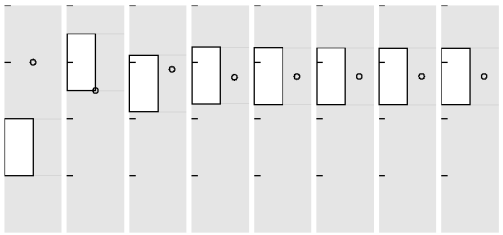
We can include strict vertical boundaries such as the top of the page (we don't want floats cut off at the top) and any fixed-position content (perhaps a logo cut out of the top right side) as infinite[3] weight variables with goal values at the top and bottom of the page.

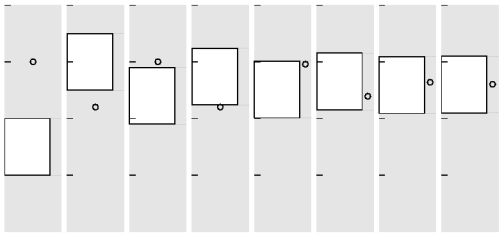
It is not unusual in first-fit based layout of scrolls to have floats protruding from the end of the text. We can elegantly remove this space by adding an invisible full-width zero-height figure at the bottom of the document with a fixed order and a doubly infinite[4] weight anchor at the end of the text. As shown in Figure 6, this forces any floats protruding

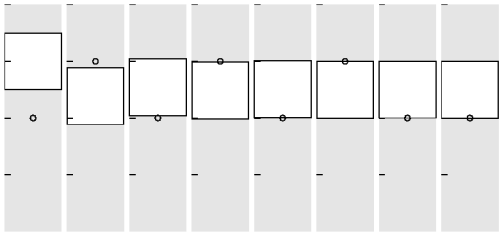
[2]Although the last step appears to have converged, the next step results in another large step.
[3]In practice we let the weight be M, a large enough number that its perturbation is insignificant when active.
[4]M^2. TEX uses a similar powers-of-infinity approach for the glue strengths.


(a) 0.1 width, optimistic.


(b) 0.5 width, optimistic.


(c) 0.5 width, careful.


(d) 0.8 width, careful.


(e) 1 width, careful.

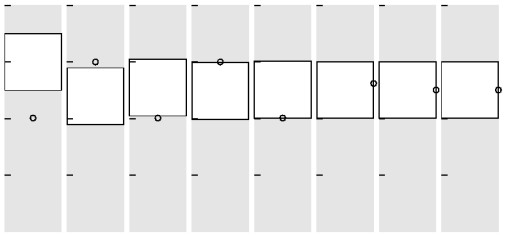
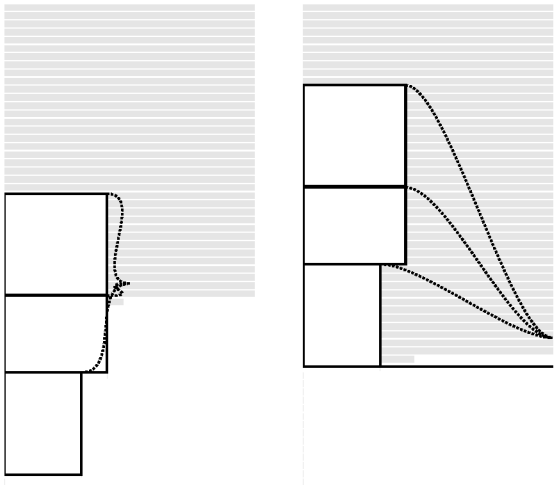

(f) $1-\epsilon$ width, careful.

Figure 5: Comparative convergence of the Optimistic and Careful FPSC Algorithms.



(a) loose (b) tight

Figure 6: Tightening the layout. The dotted lines show the anchor points by connecting each float with its anchor point in the text.

back into the text until the text is in line with the bottom of the last float. This sort of penalty due to overflow is difficult to handle with graph search, as the penalty cannot easily be determined until the layout is complete.

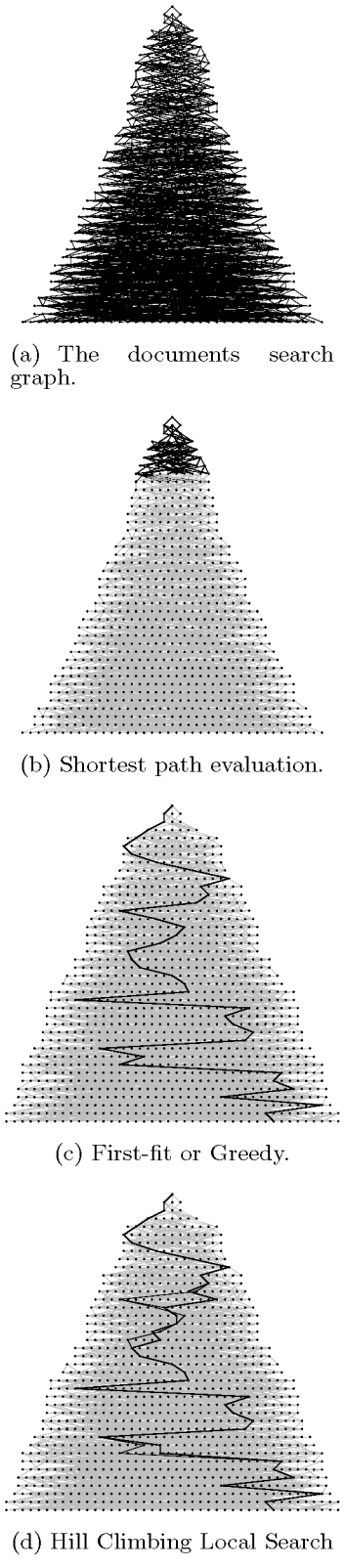
4. LOCAL SEARCH

The algorithm we have given in the previous section requires that we have chosen a particular float configuration. Determining the layout configuration is a discrete problem requiring us to choose which floats to include, their placement style and their relative order.

Traditional approaches for handling discrete choices in document layout include: first-fit, greedy, graph search and stochastic search. We have proposed that humans actually just want a LWOE layout, that is, one that doesn't have any obvious problems. To find a solution that is LWOE we look to local search methods. Local search works by starting with a reasonable state such as returned by first-fit and then repeatedly exploring neighbouring states to find one that improves the quality or satisfaction. Local search will return a solution that is locally optimal in the sense that no neighbouring state is better. If neighbours in the state space correspond to "obvious" layout changes then local optimality implies that a human will not see any obvious way to improve the layout.

Figure 7 shows a simulated document graph 7(a) with real algorithms applied to it[5]. Graph search algorithms systematically work through the graph until a *finish* node is found. Depending on the graph structure, different problems can have nearly linear time such as found by Knuth and Plass [11, 13], or slowly growing exponential time. Finding such efficient algorithms is quite hard in practice. Figure 7(b) shows the steps taken by Dijkstra's shortest path algorithm. A*-heuristic search looks similar but with a less regimented frontier.

[5]In practice, depending on the neighbourhood structure, this graph tends to be very wide and would not fit legibly in this document.



(a) The documents search graph.

(b) Shortest path evaluation.

(c) First-fit or Greedy.

(d) Hill Climbing Local Search

Figure 7: Different searching strategies. The Hill Climbing Local Search and Shortest Path evaluations are shown after the same amount of work.

First-fit and greedy, shown in Figure 7(c), find a single path from the start node in a document to a finish node. For graphs with the *matroid* structure, greedy in fact finds the optimal solution; but for documents this is also hard to ensure. In practice, first-fit has been a very successful strategy, and numerous heuristics have been developed to encourage it to find a good solution. However, often enough first fit is not satisfactory, as is evidenced by the amount of time people spend tweaking documents.

In this section we consider the most basic local search technique *hill climbing*. This starts from some initial solution and the search proceeds by repeatedly choosing a neighbour with lower penalty until no such neighbour exists in which case we have reached a local optimum. Hill-climbing must terminate assuming the graph is finite.

Unlike the previous methods, hill-climbing local search explores the graph sideways, changing configurations slightly to neighbouring configurations. Figure 7(d) shows a few steps from the greedy solution using hill-climbing. There are graphs in which hill-climbing finds the global optimum which we might call *convex* in reference to the similarity local search holds to numerical optimisation techniques. We are not sure exactly what is required to make a graph convex, but there are some basic guidelines we developed by experimentation: Penalise constraints rather than using hard constraints; use quadratic penalties rather than Manhattan distance; avoid permutations.

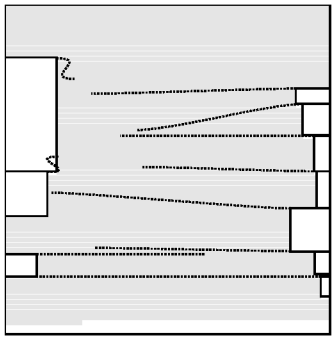
So let us consider allowing floats to move from one side of a document to the other. We add to the state used in the previous section a flag for each float indicating which side of the column the float should appear on. Our neighbourhood consists of all new states which have at most one float changing side. We embed the solver from the previous section inside a new search function:

```
def local_search(state, *params):
    best ← (state.cost(), state)
    while True:
        trial ← (best[0], None)
        for n ∈ best[1].neighbours(*params):
            n.solve()
            c ← n.cost()
            if c < trial[0]:
                trial ← (c, n)
        if trial[1] = None:
            return best[1]
        else:
            best ← trial
```

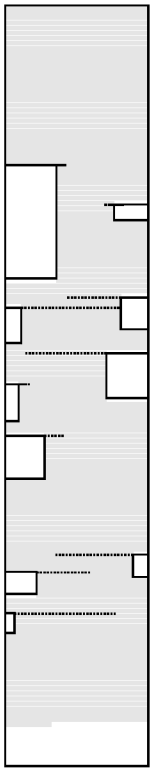
Figure 8: Hill-climbing local search.

Figure 9 shows the result of this. In a narrow document the preferred solution is to alternate floats down the page to give a more pleasing look, but when the floats become bunched up, the algorithm tries to make the two sides roughly equally sized. When the floats are separated the choice of side has no effect on other elements, so the local search can operate on each group of floats independently, reducing the computational cost significantly.

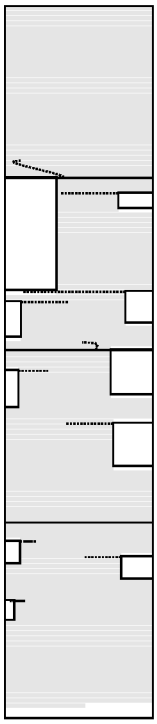
Anyone who has printed a web page with floats (or tables) has discovered a drawback to vertical scroll layout. Web browsers generally just cut the scroll into equal sized pieces and send them off to the printer splitting floats over



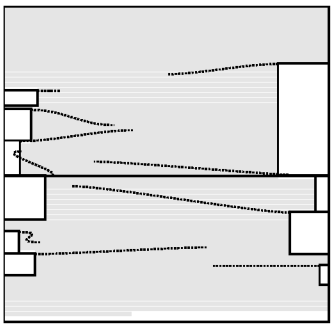
(a) wide with alternating sides



(b) narrow with alternating sides



(c) narrow with alternating sides and page breaks



(d) wide with alternating sides and page breaks

Figure 9: Floats change sides and switch pages to improve the flow of the document.

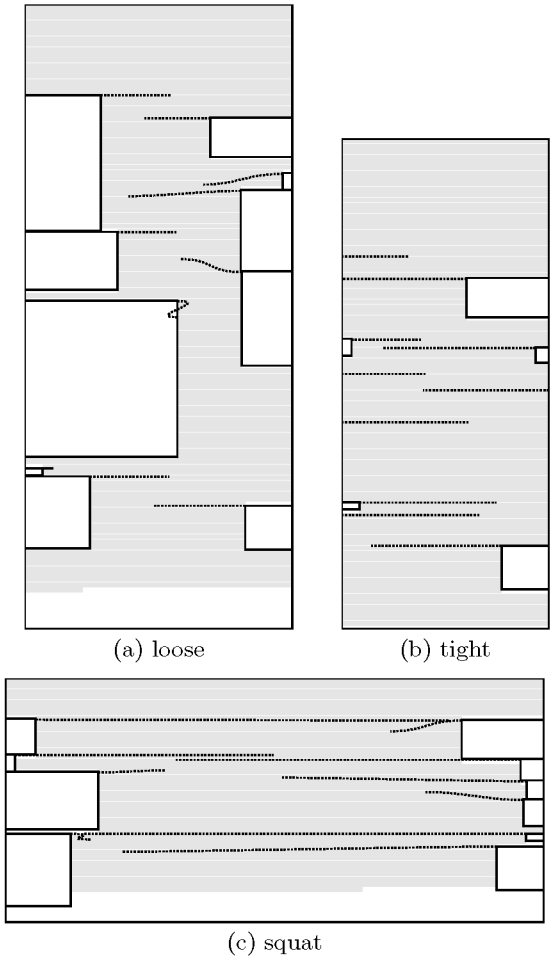


Figure 10: Alternative sizes for floats.

the page breaks. We can use local search to produce a more appealing solution. We cut the document into pages by associating floats with pages. Our document state now consist of the previous state and a list of lists of floats detailing which floats are on each page. Our neighbourhood relation is extended to allow the first float on a page to migrate to the previous page, and the last float to migrate to the next. Figures 9(c) and 9(d) show the result of this. We can also handle full column width figures (including groups of multiple figures) with the same strategy.

Another requested feature for layout is alternative float sizes. We can handle this in essentially the same way. We include an integer representing the alternative chosen for each float and associate a fixed penalty with each choice. We extend the neighbourhood to allow floats to step to a taller float, a shorter float, a wider float and a narrower float (often these will be just a pair of options). Figure 10 shows three layouts of a document whose floats have alternative sizes. The left figure is loose with whitespace at the end, the right figure has been considerably reduced in area[6]. The figure at the bottom shows that without any additional code the program also can find a squat solution when the closeness of the reference themselves encourages smaller floats.

[6]It is hard to see with the greeking, but in fact all the text is included.

5. RANDOMISED LOCAL SEARCH

It is well understood in the discrete optimisation community that randomisation can result in significant improvements in both runtime and solution quality of non-complete searches. We investigated this as a simple modification of the previous hill-climbing local search strategy. A few papers have considered stochastic search for guillotine layouts [7] and for searching the space of all possible box placements [14].

The motivation for this is to improve the efficiency of reordering floats. Floats with a low anchor weight or large anchor-point ranges in the text can move out of the way of floats with high weight (and hence high locality to the text). We tried very hard to pick the right sized neighbourhood for reordering floats, to try and match what a human would consider, but we found that such choices have vastly different neighbourhood sizes. For example, if we consider swapping consecutive pairs, there are $\binom{n}{1} = n$ neighbours in the neighbourhood. If we consider swapping all pairs, there are $\binom{n}{2} = O(n^2)$ neighbours. This jump in the neighbourhood size made the algorithm rather slow compared to the randomised approach — especially in the beginning when there are many good steps to choose.

We use a variant of randomised local search where we consider small random neighbour sets at first, until we find a state in which none of the neighbours lead to an improvement. At that point, we expand the neighbourhood size by a factor of two and continue. Whenever we get an improvement, we reduce the size of the neighbourhood again by a factor of two. The neighbour function lazily generates randomised candidates and stops when all candidates have been emitted. We try to avoid choosing neighbours that are easily detected as infeasible. For example, for resizing floats we immediately reject all float sizes which are wider than the page. There are a number of other simple consistency tests performed before a neighbour is optimised. This means that the algorithm tends towards a depth first greedy algorithm until it gets stuck in a local minimum, at which point it expands its horizon and continues to search. Figure 11 shows the modifications to local search for the adaptive random searching.

We also keep a number of older solutions (a restricted candidate list) around for restarting points to hopefully escape deeper local minima. We used ideas taken from the GRASP strategy ([2]) for updating this list. We also add a few randomised neighbours that allow the algorithm to jump out of local minima, but also surprisingly often allow the algorithm to converge faster to a good local minimum.

A significant downside of using randomised searching is that the results can be unpredictable. We can fix the random seed, but small change in size can result in a different neighbour being selected, resulting in effectively a different random sequence.

PROPOSITION 5.1. *RLS on termination is locally optimal.*

Proof: The only return from `randomise_local_search` is when all neighbours have been examined (`complete` = True) and the current node is the best node (`trial[1]` == None). □

```
def randomised_local_search(state, *params):
    best ← (state.cost(), state)
    focus ← 1
    while True:
        trial ← (best[0], None)
        seen ← 0
        complete ← True
        for n ∈ best[1].neighbours(*params):
            n.solve()
            c ← n.cost()
            if c < trial[0]:
                trial ← (c, n)
            seen ← seen + 1
            if seen ≥ focus:
                complete ← False
                break
        if trial[1] = None:
            if complete:
                return best[1]
            else:
                focus ← focus * 2
        else:
            focus ← max(focus/2, 1)
            best ← trial
```

Figure 11: Randomised local search.

6. WARM START AND INCREMENTALITY

A key advantage of local search over complete search methods is that local search works from an initial state and heads towards a good state. In particular, this means that we can use any previously determined good solution as a starting point. An interesting result is that by searching from the current state when things change, local search by its nature tends to find a layout with few changes (although it does not find an optimal such improvement). This is advantageous in document layout as it helps maintain a connection with the text reducing re-reading and perhaps improving comprehension.

The warm start nature of local search is also useful for incrementally editing a document.

Editing the text: WYSIWYG and word processor like editing is preferred by many users over LaTeXand HTML's batch formatter approach. Graph search layout algorithms are hard to use in conjunction with interactive editing, whereas with local search the user can drag objects and adjust strengths, immediately seeing how their actions affects the layout. Cutting and pasting regions of a document can carry the appropriate portion of the layout for more efficient re-solve.

Adding a float: Local search can handle the addition of a single float with a little work. Essentially, the float is greedily placed in the float order nearest the anchor point. The algorithm then considers those blocks affected by the addition and propagates outwards.

Resizing the column width: If a column is resized slightly, the floats will only need to be moved at the FPSC level and tremendously fast (<1ms) update is possible — much faster than the text flow that will follow. Once some blocks change we must use the local search or randomised local search to clean up the solution.

Another important advantage of local search over graph search algorithms is that at every step we have a best solution. This is called an *anytime* algorithm and is very useful in the context of interactive display. For example, as the worst case run time of the algorithm is unknown, a viewer can decide to allocate a fixed time to layout generation, stopping the algorithm when that time has elapsed. In the context of tabbed browsing it is common for users to load interesting links in new tabs as they read a page. These tabs can load and start running their layout optimisation in the background.

A more compelling use of such an anytime algorithm is to find a good solution to the portion of the page that is visible to the user, and fix these choices, whilst continuing to search to find a good layout for the remainder of the document.

7. EMPIRICAL EVALUATION

In this section we examine the performance of our layout algorithms. For comparison we compared the quality of greedily constructed layouts such as are generated by web browsers and word processors with the output from our algorithms. We also compare the solution quality with the optimal layout found using exhaustive search for some small, randomly generated problems. To give an idea of the impracticality of an exhaustive search, consider the example in figure 1. This has a solution space of 8.7×10^{15} as a result of the various alternative sizes (product of figure size choices), choice of figure side(2^n) and figure ordering ($O(n!)$).

The code is written in just 700 lines of Python. The code has been optimised with a satisficing approach — we only spent time working on things that were slow enough to become tedious in the development process. As a result, there is great opportunity for optimisation, particularly in smarter pruning of the neighbourhood.

As hill-climbing and RLS only store a constant number of layout trials, memory consumption is essentially linear with document size. This compares favourably with graph search algorithms which are notorious for their exponential memory consumption. Furthermore, it is conceptually easy to parallelise RLS by running the same algorithm on multiple processors with a different seed. We have not investigated this approach, however there is much active research in this area.

Local search has the interesting property that independent regions in the state description tend to optimise in parallel. That is, a document where the floats are spread out requires very little time to solve. For example, with our most general model and a document with 100 floats the times vary wildly for different float densities. If the floats are all far enough apart that they never interact the algorithm takes negligible time (roughly 0.1s), whereas when the same 100 floats all crowded together around the same anchor point the hill-climbing local search algorithm takes takes over half an hour to find a local minimum. It turns out that a strong predictor of run time is the sum of the run times of the interacting blocks. In Table 1 we look at the run times of the various cumulative extensions discussed in this paper. We use the same sequence of randomly generated floats (and alternatives) in each case averaged over 5 runs by 5 random seeds for the configuration. All the floats are forced to cluster within a single block so as to determine the worst case behaviour. The number in parentheses is the number of states considered in the search.

Clearly there is super-linear growth in the search time. This is not surprising considering the search space itself is exponential. It is not particularly useful to ask for the time to termination for a randomised search, instead we look at

No. Floats	Optimal FPSC	sides	pages	reorder resize
5	0.010s	0.34s(15)	0.3s(66)	0.38s(39)
10	0.066s	1.26s(40)	1.82s(102)	6.62s(350)
20	0.155s	4.11s(60)	16.4s(636)	73.8s(1856)
50	0.042s	87.7s(500)	294.s(8544)	
100	0.083s	892.s(5400)		

Table 1: Run times for documents with tightly clustered floats using hill-climbing local search.

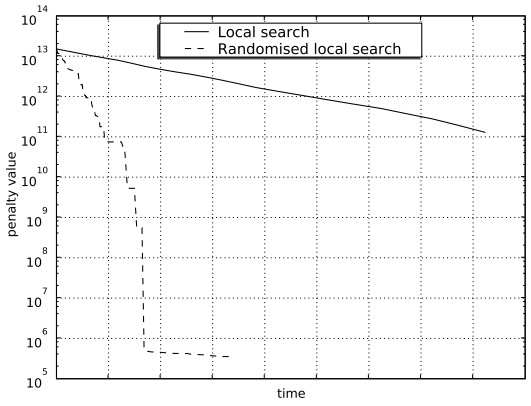


Figure 12: Comparing convergence of hill-climbing local search and randomised local search.

the convergence rate. Figure 12 shows the convergence rate of hill-climbing local search and randomised local search on a 50 element cluster document. Both searches start with the greedy solution, with a penalty of 1.5×10^{13}.

8. FURTHER WORK

Layout without obvious errors is easily extensible, and adding features is not expensive unless the elements interact. Thus, it is easy to experiment. Clearly improved performance is key. A very significant performance cost is the lack of incrementality in the core routines. In developing the prototype we used the published VPSC source code, which does not include incremental removal of constraints, and due to time constraints, we did not have the opportunity to add them. Thus, the existing code constructs a new VPSC instance for each step. This automatically incurs an linear cost with increasing floats and is relatively easy to fix.

The second performance cost is in the neighbour generation. Computing the true Lagrangian values requires closer interaction with the VPSC algorithm, and rather than calling VPSC in a loop, it would be much more efficient to calculate the float placement and Lagrangians directly[7]. Given these more accurate lagrangians it is possible to prune many more neighbours than we currently do, potentially removing another linear complexity cost.

Heuristic starting configurations: As GRASP runs multiple short optimisation passes, it is easy to seed the search with different heuristically directed algorithms, such as approximate knapsack and windowed dynamic programming. GRASP will then naturally follow up good alternatives.

User interface layout: We are intending to look at how these techniques may be applied to user interface development in tools such as Adobe FLEX. As the requirements for user interfaces and documents merge more sophisticated techniques are becoming mandatory for good user interfaces. We feel that LWOE could be a key development.

Templates: As well as directly implementing template-based layout using the ideas presented in this paper, it is also intriguing to consider allowing the designer to suggest templates implicitly as examples to the penalty function. The penalty function would consist of the existing rules, but also include the distance to similar templates to encourage these solutions.

[7]Although the Lagrangians are difficult to compute for FPSC, active constraints in VPSC do correspond to active constraints in FPSC, allowing some minor pruning

9. CONCLUSION

The idea of looking for *layouts without obvious errors* (LWOE) rather than a complete search for an optimal solution opens the possibilities of support for considerably more powerful layout constraints. The most interesting feature of LWOE is that it allows mixing of many types of existing constraint families without the combinatorial explosion of complete searches. Instead, the onus is on the layout system designer to consider what transformations a user would consider when assessing a layout and so specify the neighbourhood for local search.

We demonstrated a system consisting of three layers of constraint solving: a gradient projection-like technique for float placement, hill-climbing local search to find solutions without obvious errors, and randomised searching to increase the chance that we get a globally good solution in the face of enormous search spaces. The algorithm is anytime, allowing quick presentation whilst refining the document in a background thread.

While we have only considered single-column vertical scroll layout, it seems possible to extend our approach to handle multi-column layout. It is straightforward to extend the Careful FPSC Algorithm to handle multiple-column layout: we have only to extend the separation constraint generation algorithm to consider floats that penetrate the column boundaries. And the use of local search methods only requires us to define a suitable neighbourhood relation.

10. ACKNOWLEDGEMENTS

We would like to thank Tim Cole, Paul Harrison and Grayson Lang for their suggestions and criticisms.

11. REFERENCES

[1] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[2] S. Binato, H. Jr, and M. Resende. Greedy randomized adaptive path relinking.

[3] A. Brüggemann-Klein, R. Klein, and S. Wohlfeil. Pagination reconsidered. In *Electronic Publishing*, volume 8, pages 139–152, 1995.

[4] T. Dwyer, Y. Koren, and K. Marriott. IPSep-CoLa: An incremental procedure for separation constraint layout of graphs. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):821–828, 2006.

[5] T. Dwyer, K. Marriott, and P. J. Stuckey. Fast node overlap removal. In *GD2005: 13th International Symposium of Graph Drawing 2005*, volume 3843 of *Lecture Notes in Computer Science*, pages 153–164. Springer Berlin, 2006.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[7] E. Goldenberg. Automatic layout of variable-content print data. Technical Report 286, Hewlett-Packard Laboratories, Oct. 2002.

[8] N. Hurst and K. Marriott. Approximating text by its area. In *DocEng '07: Proceedings of the 2007 ACM symposium on Document engineering*, New York, NY, USA, 2007. ACM Press.

[9] N. J. Hurst. *Better automatic layout of documents*. PhD thesis, Monash University, pending.

[10] C. Jacobs, W. Li, E. Schrier, D. Bargeron, and D. Salesin. Adaptive grid-based document layout. *ACM Trans. Graph.*, 22(3):838–847, 2003.

[11] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. In *Software—Practice and Experience, 11(11)*, pages 1119–1184, Nov. 1982.

[12] K. Marriott, P. Moulder, and N. Hurst. Automatic float placement in multi-column documents. In *DocEng '07: Proceedings of the 2007 ACM symposium on Document engineering*, New York, NY, USA, 2007. ACM Press.

[13] M. F. Plass. *Optimal pagination techniques for automatic typesetting systems*. PhD thesis, Stanford University, June 1981.

[14] L. Purvis, S. Harrington, B. O'Sullivan, and E. C. Freuder. Creating personalized documents: an optimization approach. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*, pages 68–77, New York, NY, USA, 2003. ACM Press.

[15] H. A. Simon. A behavioral model of rational choice. *The Quarterly Journal of Economics*, 69(1):99–118, 1955.

[16] L. Weitzman and K. Wittenburg. Automatic generation of multimedia documents using relational grammars. In *Proceedings of 2nd ACM Conference on Multimedia*, 1994.