Automated Layout of Classified Ads

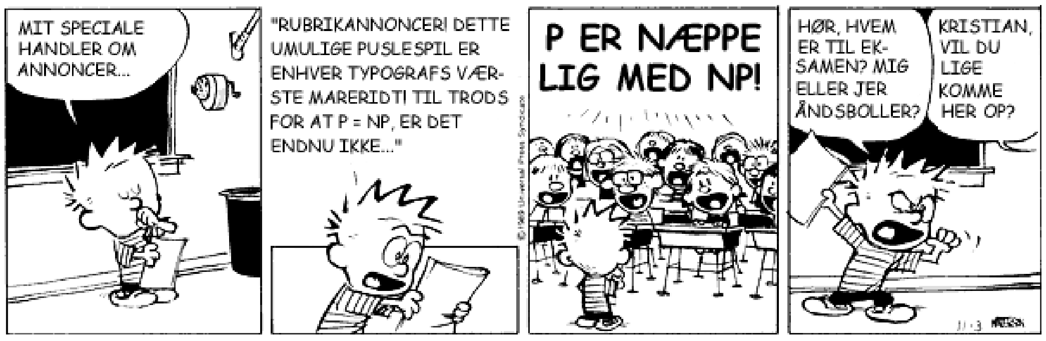
Kristian Høgsberg

M.Sc. Thesis (Specialerapport)
DAIMI, University of Aarhus, 2000

Department of Computer Science
University of Aarhus
Ny Munkegade 116, Bldg 540
DK-8000 Århus C, Denmark

Email: `hogsberg@daimi.au.dk`

Contents


MIT SPECIALE HANDLER OM ANNONCER...
"RUBRIKANNONCER! DETTE UMULIGE PUSLESPIL ER ENHVER TYPOGRAFS VÆRSTE MARERIDT! TIL TRODS FOR AT P = NP, ER DET ENDNU IKKE..."
P ER NÆPPE LIG MED NP!
HØR, HVEM ER TIL EKSAMEN? MIG ELLER JER ÅNDSBOLLER?
KRISTIAN, VIL DU LIGE KOMME HER OP?

Chapter 1

Introduction

Most newspapers have one or several sections with *classified ads.* These provide a means for the readers of the paper to communicate and advertise items for sale, personal services, or job openings. The text of a classified ad and possibly some graphics is formatted into a rectangle of a certain width given in numbers of columns and a corresponding height given in millimeters or some other unit. Depending on the amount of content, this rectangle can be anything between a one-column ad with a few lines of text and a full page ad. Before the paper can be printed, each of these rectangles must be assigned a position on a page, such that they do not overlap, and possibly, the placement should also satisfy aesthetic or costumer directed criteria. We can not expect to be able to fill the pages completely, but ideally, the amount of space that must be left blank should be minimal.

The tools, that is, computer programs, currently available for newspaper typesetting offer only minimal support for layouting classified ad sections. It is still the duty of the operator to solve the combinatorial problem of minimizing the amount of wasted space.

To some extent, though, the process of layouting the small ads can be automated; for example, car ads usually state little more than the make, model, price and color of the car along with a phone number, and this information can be typeset on a couple of lines in a one column ad. A section with thousands of these ads can be generated automatically by simple heuristics, and even if the problem is not solved to optimality, the amount of wasted space per page roughly corresponds to the average size of an ad per column. Furthermore the small ads often have to appear in some predetermined order, for example, sorted alphabetically by car make, and in this case the combinatorial problem is greatly reduced.

However, when we deal with ads with great variation in sizes, there is no simple solution. Greedy algorithms and other simple heuristics certainly produce solutions, but these are typically too far from the optimal. As for the small ads in the example above, the intuition is that the amount of wasted space per page depends on the sizes of the ads. For example, a simple algorithm could consider placing an ad on a page, and if that was not possible, it would finish that page and open a new one. When the ads are big, as compared to the page size, this is very wasteful.

1.1 Thesis Structure

In this thesis we examine the computational problems that must be dealt with in the process of automating the layout of classified ads and introduce the *skyline algorithm* as a reasonable approximation algorithm.

In Chapter 2, we state the ad packing problem and the constrained ad packing problem formally and prove that these are indeed NP-complete. We also take a look at similar problems and see how they relate to ad packing and give a survey of some of the relevant literature.

Chapter 3 describes the proposed *skyline algorithm* in detail and discusses various extensions of it, such as how to incorporate extra constraints.

To evaluate the performance of the skyline algorithm, we compare it with a recent approximation algorithm proposed by Lodi et al. in [10]. The algorithm uses a general purpose optimization technique called *tabu search*. Chapter 4 gives a brief introduction to this technique and presents the approximation algorithm.

Finally, in Chapter 5 we present results of two sets of experiments. One set seek to document the performance of the skyline algorithm as compared to the tabu search approach. The other set examines how the running time of the skyline algorithm relates to the size of the problem instance and the *skip factor*, a parameter of the algorithm.

Appendix A summarizes the notation used. Appendix B is a brief self-contained introduction to the relevant topics from computational complexity. The source code for the experimental software is included in Appendix C.

1.2 Acknowledgments

This thesis describes some of the results of a working group consisting of Anders Yeo, Riko Jacob, Gerth Stølting Brodal and the author. We have worked with the software company CCI, which produces solutions for the newspaper industry, to identify and formulate the computational problems. CCI also provided test data from one of their clients, the danish newspaper Jyllands-Posten, for the experiments described in Chapter 5.

Chapter 2

Ad Packing

As described in the introduction, the problem of layouting classified ads changes, as the sizes of the ads changes. For small ad sizes, an optimal solution, in terms of number of pages, is typically easy to produce. Sections with a mixture of small and big ads do appear, such as the car section example from the introduction, where readers put their car for sale. Auto dealers advertise in this section too, by placing big ads listing their inventory. However, in this situation we can place the big ads first using one algorithm and then fill the holes between the big ads with the small ads using another algorithm and still achieve good results. In the end, it is more a problem of generating good looking pages. To do this, we need some way of judging the aesthetic quality of the pages, or maybe the operator should manually place the big ads, and only the flowing in of the small ads should be automated. In any way, this problem is somewhat different from that which we study for the rest of this thesis, and can be solved reasonably effectively.

So, with this assertion we turn our attention to the problem, where the ads have a certain minimum size. Another way to state this is that we require that there be some upper limit on the number of ads that can be placed on a page. This restriction allows us to look to other methods, that would be otherwise infeasible.

Aesthetics are still important, but the prospect of saving a page may put aside these considerations. Thus, the problem to be solved is simple: given a set of ads, determine an arrangement of these ads on a minimal number of pages. The placement of the ads within each page is still subject to aesthetic constraints, but these are probably best left to the operator. It is not the intention to create a fully automated layout process anyway, so when reviewing the pages, that operator can make the necessary local rearrangements.

Even so, there may be other constraints that can not be ignored. For example, two ads from two competing companies may have to be placed on different pages, or maybe a customer requests a specific placement for an ad.

In Section 2.1 we first introduce the notation we will use, when reasoning about ads and pages. Then we introduce and discuss the *ad packing problem* in Section 2.2 and in Section 2.3 we describe the *constrained ad packing problem*, which incorporates a number of extra constraints. We examine lower bounds for the problems in Section 2.4 and finally in Section 2.5 we survey some related work from the literature.

2.1 Notation

Before stating the problem formally, we introduce some notation. We will usually operate with a set of ads to be layouted, and we denote this set $\mathcal{A}$, and take $n = |\mathcal{A}|$. For each ad, $a \in \mathcal{A}$, we have functions $w : \mathcal{A} \to \mathbb{N}$ and $h : \mathcal{A} \to \mathbb{N}$, giving the width and the height of the ad, respectively. Also, later on we associate a weight or a cost to each ad, which we write $c(a)$, where $c : \mathcal{A} \to \mathbb{R}$.

A page, P, is a set of *placements*, each of which is an element of $\mathcal{A} \times \mathbb{N} \times \mathbb{N}$. For a page, P, a placement $p = (a, x, y) \in P$ states that the lower left corner of a is positioned in column x at height y on P. For a placement $p = (a, x, y)$ we define functions

$$\begin{aligned} I_x(p) &= \{\, j \in \mathbb{N} \mid x \leq j < x + w(p) \,\} \text{ and} \\ I_y(p) &= \{\, j \in \mathbb{N} \mid y \leq j < y + h(p) \,\}, \end{aligned}$$

giving the intervals that the placement covers on the x- and the y-axis respectively. Also, we introduce

$$\alpha(p) = I_x(p) \times I_y(p),$$

that is, the points covered by the placement p. For a given page, P, we sometimes want to disregard the placements and only argue about the ads, and for this purpose we introduce A:

$$A(P) = \{\, a \mid (a, x, y) \in P \text{ for some } x, y \in \mathbb{N} \,\}.$$

Finally, to describe the page geometry, we give the page width in number of columns, $W \in \mathbb{N}$, and the page height, $H \in \mathbb{N}$. For page dimensions (W, H), we define the *page region*:

$$\mathcal{P} = \{\, (x, y) \mid 0 \leq x < W \wedge 0 \leq y < H \,\}.$$

With these functions, we can talk about ads and their properties in a concise way. For example, for an ad to be placed within the page boundaries, it must be the case that $\alpha(p) \subseteq \mathcal{P}$; as another example, two ad placements, p and q, overlap if and only if $\alpha(p) \cap \alpha(q) \neq \emptyset$.

2.2 Formal Description

First, we consider the the simple problem, where a number of ads must be placed on as few pages as possible:

Definition 2.1 (Ad Packing Problem) *Given a set of ads $\mathcal{A}$ and page dimensions (W, H), the* ad packing problem *is the problem of finding a minimal set of pages, S, such that*

(1) $\forall a \in \mathcal{A} : \exists! P \in S, x, y \in \mathbb{N} : (a, x, y) \in P$,

(2) $\forall p, q \in P \in S : p \neq q \Rightarrow \alpha(p) \cap \alpha(q) = \emptyset$, *and*

(3) $\forall p \in P \in S : \alpha(p) \subseteq \mathcal{P}$.

The first condition ensures that all ads are actually placed on some page, but only once. The second condition says that any two ads should not overlap, and the last condition states that ads must be placed inside the page boundary. The *ad packing decision problem* furthermore accepts a target value, m, and asks if the ads can be packed on at most m pages, satisfying the above criteria.

As stated in Definition 2.1, the ad packing problem is actually identical to the two-dimensional bin packing problem, except that here we implicitly assume that the width of the page, that is, the number of columns, is some small integer, typically less than 10.

Also, note that for $W = 1$, ad packing is just the classical (one-dimensional) bin packing problem:

Definition 2.2 (Bin Packing Problem) *Given items $a_1, a_2, \ldots, a_n$, a bin capacity C, and a cost function, c, mapping items to an integer cost, the* bin packing problem *asks for a minimal set of bins, B, forming a partition on the set of items and satisfying*

$$\forall b \in B : \sum_{a \in b} c(a) \leq C$$

The decision version of the bin packing problem furthermore takes a number of bins, m, as the target value, and asks if the items can be packed into at most this number of bins.

The bin packing problem is NP-complete, see for example Theorem 9.11 in [14], where this is proved by reduction from tripartite matching. Since ad packing can be viewed as a generalization of bin packing, ad packing is itself NP-complete:

Theorem 2.3 *Ad packing is NP-complete.*

Given an instance of the bin packing decision problem, we construct an instance of the ad packing decision problem, by taking as the set of ads $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, the width function $w(a) = 1$, the height function $h(a) = c(a)$, and page dimensions $(W, H) = (1, C)$. Immediately we get that the bin packing decision problem can be solved, if and only if the ad packing decision problem can.

2.3 Additional Constraints

Client requirements or restrictions arising from the printing machinery or other steps in the production process may further restrict the set of feasible solutions. We define the *constrained ad packing problem*, where a selection of such constraints have been added to the ad packing problem from above. The chosen constraints on one hand represent a realistic and usable set of features, but they also exemplify how to extend the skyline algorithm on various levels, which is discussed in Section 3.8.

For a number of reasons, it may be the case that two or more ads should never appear on the same page. As mentioned in the introduction to this chapter, it could be that two competing companies would not want their ads to be placed next to each other, but it could also be the case that one customer buys a number of ads and wants them to show up on different pages.

Opposite to the above constraint, we have the case where a number of ads must be placed on the same page. Some newspapers have printing equipment that can use one specific color on a page, a so called *spot color*. Ads on that page can use this color only, for example a blue line or a green logo, but it is cheaper than a full color page. To minimize production costs, ads that use the same spot color should be placed on the same page, and in this situation it is useful to require that some ads stay together.

Previously, we suggested that to satisfy aesthetic criteria the operator could rearrange the individual pages manually. This may not always be possible without disturbing the rest of the generated solution, though. Instead, the operator could restrict the allowed placements for one or several ads, so as to avoid the unpleasing layout, and have the tool generate another solution. We consider two types of placement restrictions: either an ad can be tied to one or several page borders, or it can be assigned a fixed position on the page. Tying an ad to the left page border, means that the left edge of the ad must touch the left edge of the page, and similarly for the other edges. If this still does not give satisfactory results, assigning a fixed position to one or several ads allows for layouting some of the ads by hand, and then have the tool place the rest of the ads automatically.

Before defining the constrained ad packing problem, we formalize the idea of *border constraints* and what it means for a set of pages to *satisfy* such constraints. A set of border constraints, B, is a subset of $\mathcal{A} \times \{\mathsf{B}, \mathsf{T}, \mathsf{L}, \mathsf{R}\}$. If $(a, \mathsf{T}) \in B$, then for a set of pages, S, to satisfy B, a must be placed on the top of a page. More formally, we require

$$\begin{aligned}
\forall(a,\mathsf{B}) \in B &: \forall(a,x,y) \in P \in S : y = 0 \\
\forall(a,\mathsf{T}) \in B &: \forall(a,x,y) \in P \in S : y + h(a) = H \\
\forall(a,\mathsf{L}) \in B &: \forall(a,x,y) \in P \in S : x = 0 \\
\forall(a,\mathsf{R}) \in B &: \forall(a,x,y) \in P \in S : x + w(a) = W
\end{aligned}$$

Notice that an ad may be tied to several borders, for example top and left, which effectively ties the ad to a fixed position in the top left corner of the page. It is also possible to tie an ad to the left and right borders, but that constraint can only be satisfied if the ad has width W.

We can now assemble the constraints discussed above in the following definition:

Definition 2.4 (Constrained Ad Packing) *Given a set of ads, $\mathcal{A}$, page dimensions (W, H), a set of pairs of ads, $D \subseteq \mathcal{A}^2$, that should be placed on distinct pages, a set of pairs of ads, $F \subseteq \mathcal{A}^2$, that should be placed on the same pages, a set of border constraints, B, and a set of pre-layouted pages L, the* constrained ad packing problem *is the problem of finding a minimal set of pages, S, such that conditions* (1)–(3) *from Definition 2.1 are satisfied and furthermore*

(4) $\forall(a,b) \in D, P \in S : a \in A(P) \Rightarrow b \notin A(P)$,

(5) $\forall(a,b) \in F, P \in S : a \in A(P) \Leftrightarrow b \in A(P)$,

(6) *S must satisfy B, and*

(7) $\forall P_L \in L : \exists P_S \in S : P_L \subseteq P_S$.

Condition (4) formalizes the constraint that certain ads should not appear on the same page. Notice that if $(a, b) \in D$, this implies $b \in A(P) \Rightarrow a \notin A(P)$ for all P, corresponding to $(b, a) \in D$, also. For simplicity, we will just assume that D is indeed a symmetric relation on $\mathcal{A}$.

Likewise, condition (5) formalizes the condition that certain ads must appear on the same page. In this case, we will assume that F is an equivalence relation, for the same reasons as above.

Condition (6) ensures that the border constraints are satisfied, as defined above, and the last condition states that the positions of the preplaced ads should be respected, that is, for each preplaced page there must be a page in the solution, that contains it as a subset.

The specified constraints can be unsatisfiable in many ways. For example if an ad, a, is tied to the left and right page border, but $w(a) < W$, both border constraints can not be satisfied. Also, interactions between different types of constraints can result in an inconsistent problem instance. It may be the case that ads a and b are required to appear on the same page, but at the same time a and b are both tied to the top and left borders, effectively the upper left corner. If (5) and (6) are respected, (2) is violated, since the ads overlap. An ad could have a fixed position in the middle of a page, but also be tied to one of the page borders, in which case only one of (6) and (7) can be satisfied. However, the consistency of the constraints can be checked easily, and we will assume we only deal with consistent instances.

2.4 Lower Bounds

In the context of approximation algorithms, lower bounds provide for a way to judge the results of the algorithm. If a produced solution to a minimization problem coincides with a lower bound, we know that the solution is optimal. If the solution is greater than the lower bound, the difference between the two gives an indication about the quality of the solution. Also, lower bounds can be used as a stopping condition for a local search algorithm that iteratively improves a current solution until some criteria is met. Finally, lower bounds play an important role in branch and bound frameworks, where they are used to restrict the search.

A lower bound is a measure, L, on any instance of a problem $I \in A$, such that

$$L(I) \leq c(S^*),$$

where S^* denotes an optimal solution to I. To evaluate the performance of a lower bound, we introduce the notion of *worst case performance*:

Definition 2.5 (Worst Case Performance) *Given a lower bound L, for a problem A, we define the* worst case performance*, $r(L)$, as*

$$r(L) = \inf\{L(I)/c(S^*) \mid I \in A\},$$

where S^* is an optimal solution to I.

The simplest lower bound for the ad packing problem is the *continuous lower bound*. Obviously, to layout the ads without overlap between ads, at least the total area that the ads occupy is needed, and thus, for an instance $I = (\mathcal{A}, W, H)$, we get the bound

$$L_C(I) = \left\lceil \frac{\sum_{a \in \mathcal{A}} w(a)h(a)}{WH} \right\rceil .$$

Martello and Vigo show in [11] that L_C has a worst case performance of 1/4. They also show how to tighten the bound by transforming instances, such that any lower bound for a transformed instance is also a lower bound for the original instance. Fekete and Schepers present a unified approach to this technique in [4], where they introduce a number of *dual feasible functions*:

Definition 2.6 (Dual Feasible Function) *A function, $u : [0, 1] \to [0, 1]$, is a* dual feasible function*, if for any finite set $\mathcal{S}$ of nonnegative real numbers, we have*

$$\sum_{x \in \mathcal{S}} x \leq 1 \Rightarrow \sum_{x \in \mathcal{S}} u(x) \leq 1.$$

Consider pages normalized to 1 by 1 and ads with real-valued widths and heights in the interval $[0; 1]$. Given a set of ads that fit on a page, we transform this instance by changing the widths of the ads, such that

$$w'(a) = u(w(a))$$

for some dual feasible function u. This new instance will also fit on a page, which is shown by Fekete and Schepers, see Corollary 8 in [6].

For an instance of ad packing, $I = (\mathcal{A}, W, H)$, we can obtain a transformed instance, $U(I)$, by taking $w'(a) = u_w(w(a)/W)$ and $h'(a) = u_h(h(a)/H)$, for dual feasible functions u_w and u_h. Now, for any integer k, we have:

$$\exists S \in F(I) : |S| < k \Rightarrow \exists S' \in F_1(U(I)) : |S'| < k,$$

where F_1 is the set of feasible solutions for the normalized instance. Negating this implication gives a useful result:

$$\forall S \in F_1(U(I)) : |S| \geq k \Rightarrow \forall S \in F(I) : |S| \geq k,$$

that is, any lower bound for the transformed instance $U(I)$, is in turn a lower bound for the original instance. Combining this with the continuous lower bound, we get a number of new lower bounds that in various ways account for the bulkiness of the items.

Fekete and Schepers give 3 dual feasible functions in [4]: a staircase type of function,

$$u^{(k)}(x) = \begin{cases} x, & \text{for } x(k+1) \in \mathbb{Z} \\ \lfloor (k+1)x \rfloor / k, & \text{otherwise} \end{cases},$$

where $k \in \mathbb{Z}$; a threshold type of function, mapping everything under ε to 0 and everything over $1-\varepsilon$ to 1,

$$U^{(\varepsilon)}(x) = \begin{cases} 0, & \text{for } x < \varepsilon \\ x, & \text{for } \varepsilon \leq x \leq 1-\varepsilon \\ 1, & \text{for } 1-\varepsilon < x \end{cases},$$

where $\varepsilon \in [0; \frac{1}{2}]$; and finally, a combination of the two

$$\varphi^{(\varepsilon)}(x) = \begin{cases} 0, & \text{for } x < \varepsilon \\ \frac{1}{\lfloor \varepsilon^{-1} \rfloor}, & \text{for } \varepsilon \leq x \leq \frac{1}{2} \\ 1 - \frac{\lfloor (1-x)\varepsilon^{-1} \rfloor}{\lfloor \varepsilon^{-1} \rfloor}, & \text{for } \frac{1}{2} < x \end{cases},$$

and again, $\varepsilon \in [0; \frac{1}{2}]$.

By combining the dual feasible functions as described in Remark 20 in [6], we get a lower bound, L_{2d}, which dominates the L_4 bound from [11] and is simpler to compute. By also incorporating $u^{(k)}$ into the lower bound, as suggested in the end of Section 5 of [6], we were able to further improve the computed lower bound for a number of the data sets.

2.5 Related Work

To our knowledge, the ad packing problem, where the number of columns is assumed to be some small integer, has not been dealt with previously in the literature. However, a lot of attention has been given to the more general two dimensional bin packing problem, which finds application in, for example stock cutting, where sheets of glass, steel, textile or some other material is to be cut into a number of smaller rectangles. The actual application dictates various extra constraints on the feasible solutions. For cutting applications it is often the case that the solutions must satisfy the *guillotine constraint*, which requires that the items are placed, so that they can be cut from the stock sheet using only edge-to-edge cuts. Another parameter is whether or not the items are allowed to be rotated. When cutting glass or steel this could be allowed, but for wood or textile we may want to preserve the orientation of the texture of the material. For the ad packing problem it is clear that we can not allow rotations; the height of the ad may not correspond to an integral number of columns, and of course, rotating an ad does not make sense in the first place. On the other hand, the guillotine constraint is not necessary for the ad packing problem, in fact it would only restrict the number of feasible layouts.

A wide variety of techniques have been applied to solve the two dimensional bin packing problem. One of the earliest treatments is [2] by Chung et al., where the simple *Hybrid First Fit* (HFF) approximation algorithm is described and analyzed. The HFF algorithm sorts the ads by non-increasing heights and then, in the first phase of the algorithm, places the ads in an vertical open ended strip, which is broken into pages during the second phase. Chung et al. give fairly tight bounds on the asymptotic worst case performance ratio, specifically they show that

$$2.022 \leq R^{\infty}_{HFF} \leq 2.125.$$

Berkey and Wang compare the HFF algorithm with five other simple heuristics for the two dimensional bin packing problem in [1]: finite next fit, finite first fit, finite best strip, finite bottom left, and next bottom left. The finite best strip algorithm is described in greater detail in Section 4.3. These algorithms all initially sort the items by some criteria, for example non-increasing heights, and then place them using some heuristic, for example first fit. Experiments on randomly generated problem instances indicated that finite best strip was the better heuristic, especially for the larger instances from [1].

Another approximation approach is that of *local search*, where an initial solution is generated by means of a simple polynomial time algorithm, and then improved iteratively. The local search technique is described in greater detail in Section 4.1. This type of algorithms typically produce better results than the sort-and-place type of algorithms considered by Berkey and Wang, but are difficult to analyze, with respect to both running time and worst case performance ratio.

An example of a local search algorithm is the *tabu search* algorithm presented by Lodi et al. in [10]. The tabu search improves the current solution by rearranging items between bins, using the finite best strip algorithm to actually pack the bins. We describe the tabu search algorithm in greater detail in Chapter 4. Another local search approach is [3] by Faroe et al., which uses a technique called *guided local search*, reminiscent of tabu search, for solving the three dimensional bin packing problem. An interesting aspect of the algorithm is that the local search operates directly on the solution, instead of applying a sort-and-place type of algorithm to construct the packing. Faroe et al. furthermore give a short review of previous local search algorithms for the two and three dimensional bin packing problems.

Another line of work is exact algorithms, where the problem must be solved to optimality. Martello and Vigo describe an exact algorithm in [11], using a two level branching scheme. On the outer level, items are distributed between bins, and while shifting ads around, the L_4 lower bound is used to restrict the search. This bound, which is also introduced in the article, is the maximum of a number of lower bounds, hand tailored to the two dimensional case. On the inner level of the search it is determined, whether the items assigned to a given bin actually can be packed to fit in the bin.

In a series of three articles, [5], [6] and [7], Fekete and Schepers describe an exact algorithm for solving the d-dimensional bin packing problem, and related problems, $d \geq 2$. The first article describes an enumeration technique, which uses an abstract, graph-based representation of *layout classes*. The second gives a number of lower bounds based on *dual feasible functions*, and the third describes how these results are combined to form a branch-and-bound framework.

Also related to the two dimensional bin packing problem is the two dimensional knapsack problem, where the optimal packing of a number of rectangular items from a given a set must be determined. Hadjiconstantinou and Christofides present an exact algorithm for this problem in [9] that uses an enumeration technique similar to that of the skyline algorithm, presented in the next chapter. They allow for a minimum and maximum number of occurrences of each item and use a linear programming formulation of the problem for bounding the search.

Chapter 3

The Skyline Algorithm

The algorithm we propose, uses an enumeration scheme to generate one page at the time, and by doing so repeatedly, it packs all the ads into a number of pages. The enumeration scheme works by maintaining a skyline view of the current contents of the page, that is, it builds up the page from the bottom to the top and remembers the height of the columns as it goes along. However, enumerating over all possible pages for the given set of ads is too slow, so we speed up the process, by only considering a subset of all configurations.

The page selection algorithm takes a greedy approach; it simply selects the page that maximizes the sum of the areas of the ads. This way, the pages that are generated in the beginning of the process have little wasted space, but later on it becomes increasingly difficult to pack the pages well. Furthermore, experiments show that ads of a certain width are easy to pack and are used up quickly, whereas ads of other widths tend to turn up on the last pages where a lot of space is wasted.

To avoid this typical greedy behavior, we assign a weight to each ad. Instead of maximizing the sum of the areas of the ads, we now maximize the sum of the weights of the ads. Initially the weight of an ad is just the area of the ad, but after running the algorithm we can adjust the weights based on the outcome and run the algorithm again. Using an appropriate heuristic for readjusting the weights, will force the troublesome ads to appear earlier in the process, where they can be combined with other ads to fill the page better.

Sections 3.1–3.6 present the enumeration algorithm in detail and discuss various heuristics for trading quality for speed. In Section 3.7 we examine some strategies for generating several pages using the enumeration algorithm, and in Section 3.8 we see how the algorithm can be generalized to accommodate the additional constraints described in Section 2.3.

3.1 Canonical Layouts

When an ad placement is given by column number and height of its lower left corner, a great number of placements are possible. Consequently, for a given set of ads there are many, essentially identical, possible layouts. Obviously, when enumerating, we should only evaluate those layouts, we consider significantly different. This raises the question of when to consider two layouts equivalent and when to consider them distinct.

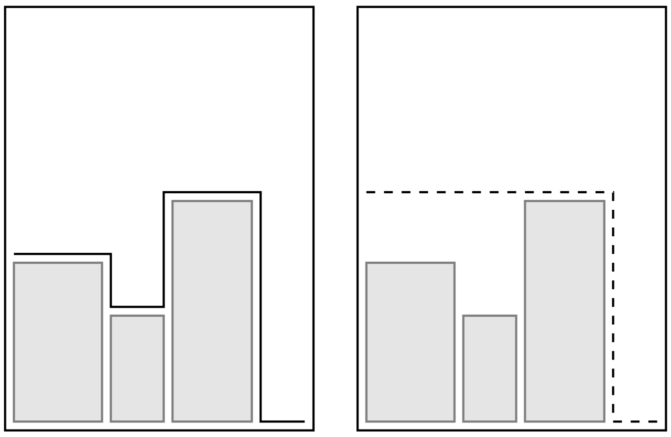
We restrict our attention to those layouts, where each ad touches some other ad or the page edges to its left and below. An ad placed this way, is placed in a *canonical position*, and layouts, where all ads are placed in canonical positions, will be called *canonical layouts.* This restriction will not discard any relevant layouts. Any layout that fits on the page can be rearranged to be on this form: given a page, while there are ads that can be pushed down or to the left, pick one and do so. Since there is a limit to how much an ad can be pushed down and left, this process stops eventually, and all ads on the resulting page will touch some other ad or the page edge to the left and below. Hadjiconstantinou and Christofides also only consider this type of layouts in their exact algorithm for the two dimensional knapsack problem [9].

Even though this restriction does not discard any relevant layouts, as seen from a enumeration point of view, a canonical layout may not be *aesthetically* pleasing. In fact, it almost never is, since most of the unused space will be at the top of the page, where it jumps into view. We will not deal with this problem, however, just assert that it can be handled by some minor rearrangements either automatically, using some heuristic that move some of the ads upwards, or simply just manually. In any case, the pages will have to be reviewed by a typographer before going to press, and thus, we are not aiming to develop a fully automatic system.

3.2 Configurations and Moves

As mentioned above, the enumeration algorithm works by maintaining a skyline or water-level view of the ads currently on the page. In fact it uses two skylines: a *real* skyline, corresponding to the actual ads placed on the page and a *virtual* skyline which is usually higher than the real skyline, and never below it (see Figure 3.1). The invariant for the moves described below is that canonical positions below the virtual skyline have been considered by the algorithm.

A *configuration* is an actual set of ads placed on the page in a canonical



(a) The *real* skyline. (b) A *virtual* skyline.

Figure 3.1: Skylines for a given set of ads.

layout, along with a virtual skyline. In a given configuration, there will be a *leftmost local minimum*, which is an interval of columns. When walking from left to right on a skyline, either virtual or real, this is the first span of columns, where both neighbor columns have a higher skyline. Formally, if $s_1, s_2, \ldots, s_W$ represents the skyline, such that s_i gives the height of the skyline in column i, and we set $s_0 = s_{W+1} = H$, then the leftmost local minimum is the interval of columns $[i_0, i_1]$, where

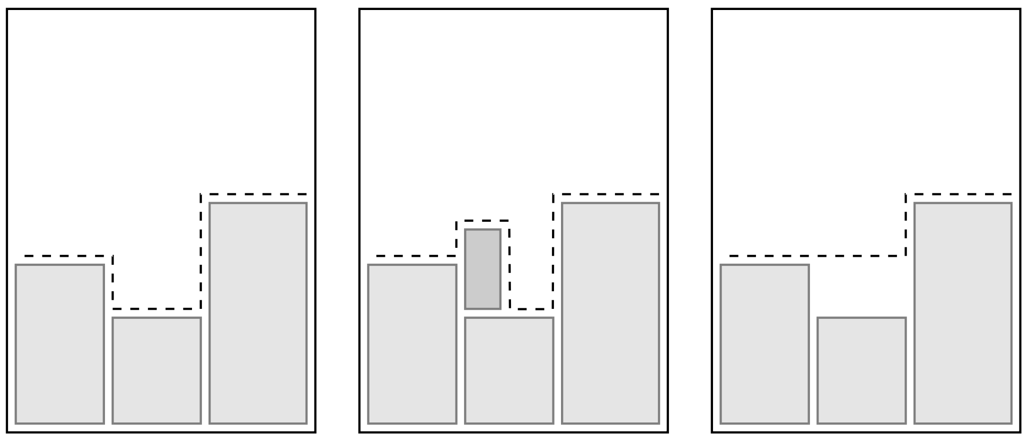
$$i_0 = \min\{\, i \mid s_{i-1} > s_i \wedge \exists k > i : (s_i < s_k \wedge \forall i \le j < k : s_j = s_i) \,\}$$

and, given i_0,

$$i_1 = \max\{\, i \mid \forall i_0 \le j \le i : s_{i_0} = s_j \,\}.$$

In Figure 3.1 (a), this is the interval on top of the middle ad, and in Figure 3.1 (b) it is the interval to the right on the bottom edge of the page. The left corner in a minimum is a *candidate canonical position*; this is where the algorithm will try to place the ads. Also, we will say that a skyline is *decreasing* within an interval of columns, when the skyline level in the columns within the interval decreases from left to right. Analogously, a skyline can be *increasing*.

Two types of moves form the basic idea of the skyline algorithm: in a given configuration, the algorithm locates the leftmost local minimum on the virtual skyline and in turn, tries to place each of the remaining ads in the lower left corner of the minimum. Whenever an ad is placed, a new configuration arises, and the virtual skyline is updated.



(a) A configuration. (b) Placing an ad. (c) Raising the skyline.

Figure 3.2: The two possible moves.

The second type of move raises the virtual skyline within the minimum to be level with the lowest of the left and right neighbor columns. Thus, the space below the raised skyline is covered, to indicate that we wish to ignore it and consider a new position. This too leads to a new configuration, except when there is no neighboring column. In this case, the skyline is a straight line across the page, that is, $s_1 = s_2 = \cdots = s_W$, and we raise the skyline to the top of the page. This closes the page, corresponding to a terminal configuration, and at this point we check to see if the generated page is better than any previous page. The two moves are illustrated in Figure 3.2.

In order to only generate pages on the form previously described, we must consider some special cases. When placing an ad, we look to the real skyline to ensure the ad actually touches some other ad to the left and some other ad below. If the virtual skyline has been raised, so that the ad would be floating as shown in Figure 3.3 (a) or if there is a gap to the left as in Figure 3.3 (b) we disallow the move.

Another situation arises when the virtual skyline is above the real skyline in the rightmost column under a newly placed ad, as in Figure 3.4 (b) and (c). In this case we will have to reconsider the ads below the virtual skyline, since the right edge of the new ad together with the upper edges of the ads below the skyline constitute new positions, that have not previously been considered. To allow the algorithm to try out these positions, we lower the virtual skyline within the current minimum. The virtual skyline is lowered to the level of the real skyline, though not where this results in a

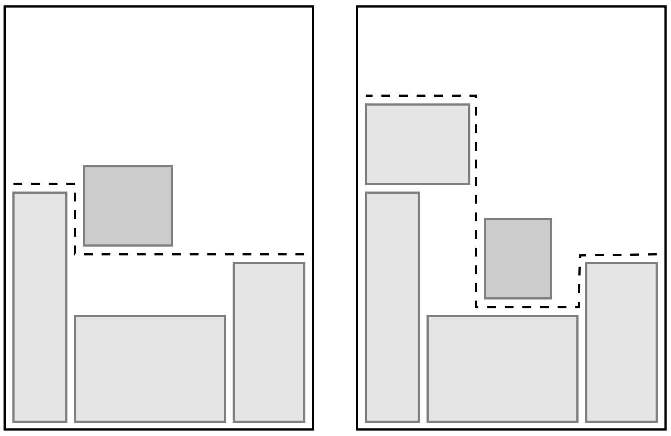

(a) No other ad below. (b) No other ad to the left.

Figure 3.3: Illegal moves.

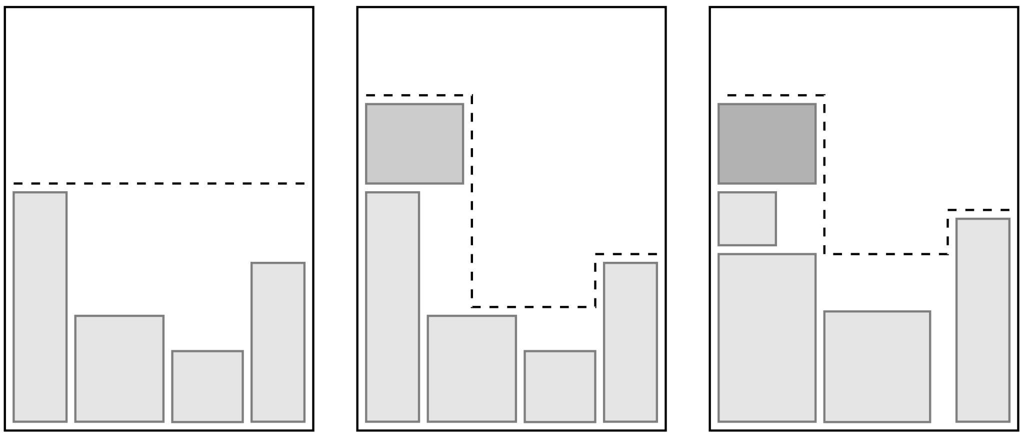

(a) Before placing the ad. (b) After placing the ad. (c) The exception.

Figure 3.4: Lowering the virtual skyline.

decreasing virtual skyline, since this would cause the algorithm to visit the same canonical position on the page several times. The move is illustrated in Figure 3.4 (a)–(b). However, if the real skyline in the rightmost column below the newly placed ad is at a higher level than the real skyline immediately to the right of the ad, then the skyline is not lowered further than this level. Again, this is to avoid visiting positions already considered, as illustrated in Figure 3.4 (c).

3.3 Pseudo-code for the Algorithm

The moves described in the previous section, are incorporated into a recursive algorithm, to implement an exhaustive depth-first search of the configuration space, as outlined in Figure 3.5. The recursive function named iterate accepts a set of ads, $\mathcal{A}$, a skyline configuration, S, and the current layout, P, as arguments. First it considers each of the ads in A in the current minimum on the skyline S, and if the placement satisfies the criteria described above, it is added to the current layout and the skyline is updated. Then, iterate is called recursively to examine the solutions reachable from this new configuration. Finally, the algorithm backtracks by taking the ad off the page and restoring the skyline.

When all the ads have been considered, the algorithm tries to raise the skyline. If this leads to a new configuration, iterate is called again to search the solutions now reachable. Otherwise, a terminal configuration has been reached, and if the sum of the weights of the ads on the page exceeds that of all other previous layouts, c_{max}, the current layout is stored in P_{max} and c_{max} is assigned the new maximum.

Initially iterate is called with the set of all ads, an initial skyline, that is, a skyline coinciding with the bottom edge of the page and an empty page.

3.4 Properties of the Algorithm

As a minimum requirement, an enumeration algorithm should consider all canonical layouts. Furthermore it is desirable that the algorithm only considers every layout once. The skyline algorithm has both these properties. First we show the following lemma.

Lemma 3.1 *In a given configuration, all candidate cannonical positions can be visited by repeatedly raising the skyline.*

When the algorithm raises the skyline, a candidate cannonical position is folded away under the skyline, and the number of such positions in the new

```
$c_{max} \leftarrow 0$
$P_{max} \leftarrow \emptyset$

**function** iterate $(\mathcal{A}, S, P)$
  **for** $a \in \mathcal{A}$ **do**
    locate leftmost local minimum $[i_0, i_1], y$
    **if** $a$ can be placed at $(i_0, y)$ **then**
      update $(S, a)$
      $P \leftarrow P \cup \{(a, i_0, y)\}$
      $\mathcal{A} \leftarrow \mathcal{A} \setminus \{a\}$

      iterate $(\mathcal{A}, S, P)$

      backtrack $(S, a)$
      $P \leftarrow P \setminus \{(a, i_0, y)\}$
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{a\}$
    **end if**
  **end for**

  raise_skyline $(S)$
  **if** $S$ changed **then**
    iterate $(\mathcal{A}, S, P)$
  **else**
    **if** $c_{max} < \sum_{b \in A(P)} c(b)$ **then**
      $c_{max} \leftarrow \sum_{b \in A(P)} c(b)$
      $P_{max} \leftarrow P$
    **end if**
  **end if**
  backtrack $(S)$
**end function**

iterate $(\mathcal{A}, \text{initial } S, \emptyset)$
```

Figure 3.5: Pseudo code for the enumeration algorithm.

configuration is one less. Having raised the skyline, the algorithm locates the new leftmost local minimum, considers the position there and moves on, eventually. Since a candidate cannonical position is defined by a pair of ads, that is, the right edge of one ad and the upper edge of another, the number of such positions is at most W, and so, in a finite number of steps, all positions will be visited.

Theorem 3.2 *The skyline algorithm examines each canonical layout exactly once.*

Given a canonical layout, we can simulate the actions of the skyline algorithm, to see how it would build this actual page. First we locate the current local minimum, which initially is the bottom edge of the page, and then we look to the given layout, to see which ad we should place there. If no ad is placed in that position in the given layout we instead choose to raise the skyline. Next we locate the new local minimum in the new configuration and repeat the process, until all ads have been placed.

Since the given layout is canonical, in every configuration at least one of the remaining ads is placed so that it touches the virtual skyline to the left and below. By Lemma 3.1 above, the skyline algorithm will eventually consider this canonical position by raising the skyline repeatedly, and thus, we will consider all ads in the given layout.

On the other hand, to prove that the algorithm examines a canonical layout at most once, consider the sequence of moves made to reach a given configuration. For example:

$$\text{place } a_7, \text{place } a_2, \text{raise skyline}, \text{place } a_3, \ldots$$

This sequence uniquely determines the layout that the algorithm will produce. But the converse is also true: given a canonical layout, there is exactly one corresponding sequence of moves that will produce this layout. If not, that would imply that two different sequences could result in the same layout, but this is not possible: the two sequences may have a common initial subsequence, but at some point they have to differ. Up until this point the generated layouts will agree, but now the different sequences will dictate different moves. Either two different ads will be placed or one sequence will raise the skyline, while the other places an ad. In both cases, the result will be two different layouts, and the remaining moves will not change this, since the position in question will not be reconsidered. The only move that could violate this claim, is when the algorithm lowers the skyline. But as explained in the last paragraph of Section 3.2, we are careful not to reintroduce previous candidate position.

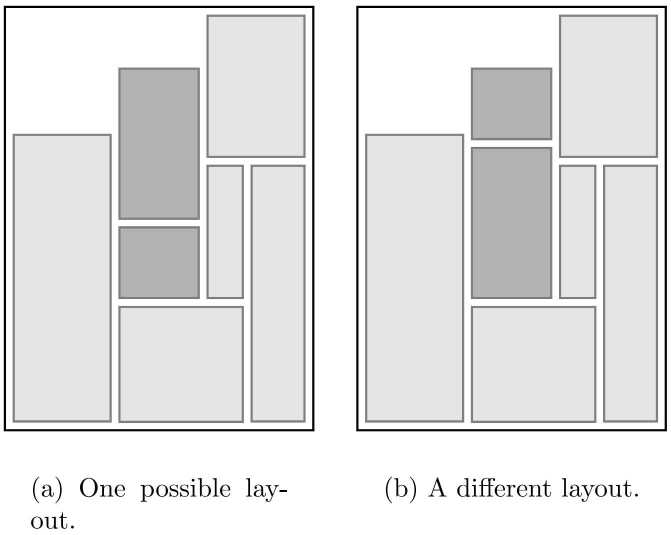

(a) One possible layout. (b) A different layout.

Figure 3.6: Example of local symmetries.

3.5 Eliminating Symmetries

As shown in the previous section, the skyline algorithm considers all canonical layouts, but this, in some sense, is still too much work. If we could incorporate symmetry considerations, many layouts could be viewed as identical, and thus ignored. We could then no longer claim to be considering all canonical layouts, but as long as we examine at least one representative from each class of symmetric layouts it will not affect the outcome. The obvious example is when the algorithm considers a page as well as the vertical and horizontal mirror images of that page. But also local symmetries could be examined: if a sub-rectangle on a page can be mirrored, only one of these combinations is worth considering. An example of local symmetries is illustrated in Figure 3.6, where the two layouts differ, only because the shaded ads have been swapped.

A systematic approach to symmetry elimination is taken by Fekete and Schepers in [5], for the n-dimensional bin packing problem. They represent a packing as a set of graphs, one for each dimension. The vertices of the graphs are the items to be packed into the bin, and there is an edge between two vertices in graph G_i, if the two corresponding items overlap when projected onto the x_i-axis. This representation has just enough structure to reflect whether the items fit in the bin, but on the other hand, it is sufficiently abstract so as to group all symmetric packings into one packing class.

However, with the skyline algorithm we take a more ad hoc approach.

We would like to detect as many symmetries as possible, but on the other hand, the algorithm should be able to check the conditions quickly. Thus, the symmetries we consider, are restricted to those that can be identified easily.

One such symmetry occurs when we place an ad on top of another ad of the same width, the situation from Figure 3.6. In general we could stack n ads of the same width on top of each other in $n!$ different ways, but only one is worth considering. To enforce this, we assign an essentially arbitrary, unique integer identifier to each ad. We then require that ads of the same widths are stacked only such that the ads appear in order of ascending identifiers from bottom to top.

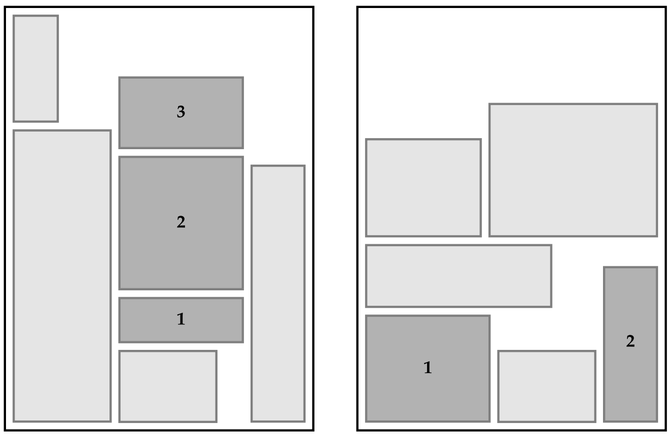
The other symmetry accounted for by the algorithm, is that of mirroring the entire page with respect to a vertical axis. Ads on the original page that do not touch some other ad on their right edge end up not touching any ad on their left edge on the mirror page. This is not a canonical layout and thus, it is not generated by the algorithm. However, the algorithm will generate an equivalent page, where all the ads are pushed as far to the left as possible, and this is the page we try to exclude. So, when we talk of a vertical mirror page, we think of the geometric mirror page with all ads pushed as far to the left as possible. To avoid generating the mirror page we again consider the integer identifier assigned to each ad and require that when placing an ad in the lower right corner, its identifier is greater than that of the ad in the lower left corner.

Figure 3.7 illustrates the symmetry rules.

3.6 Speeding up the Skyline Algorithm

Even when the skyline algorithm does not consider symmetric pages, it still has to visit a number of configurations that is exponential in the number of ads. This is still infeasible, and we now examine some ways to accelerate the search at the cost of quality of the solution. If we abandon the restriction that the algorithm should consider all pages, we can introduce optimizations that narrow the search to only a subset of all possible pages.

One way to speed up the search is to require that the biggest available ad, a, must be placed on the page. Of course, in the case that this is a full-page ad, we immediately get an optimal page. On the other hand, if the ad does not fill all of the page, the idea is that the remaining area is small, which greatly limits the expected depth of the search tree. This does not necessarily result in the optimal page for the given ads, but it does generate the optimal page with a on it. To incorporate this optimization, we extend



(a) Stacking ads of the same width.

(b) Preventing vertical symmetries.

Figure 3.7: Symmetry rules.

the algorithm, so that whenever an ad is placed on the skyline, the algorithm checks to see if the biggest ad still fits on the page, unless it is already part of the layout.

Another idea is to impose some form of granularity on the heights of the ads. There are several ways to do this; one would be to round up the ad heights to the nearest multiple of some integer, g, the granularity. If we took g to be 5 percent of the page height, there would be only 20 possible heights. By only trying one ad of each height in each configuration, we effectively limit the branching level of the search and makes it independent of the given input. The actual method that we use is sligthly different, though. Instead of using a uniform granularity, we use a *skip factor*. In a given configuration, if the algorithm manages to place an ad, a, then it regards the ads that are less than a factor f lower than a as equal and does not try to place these. This is easily realized in the algorithm by iterating through all ads of the same width at a time in order of non-increasing heights. When an ad, a, was succesfully placed on the skyline, we skip ahead until we find an ad of height at most $(1 - f)h(a)$, or if no such ad was found, we move on to a different width. This way we place at most ads of m different heights for each possible width in each recursive call, where

$$H(1 - f)^{m-1} \geq 1$$

implying

$$m \leq 1 + \log_{1/(1-f)} H.$$

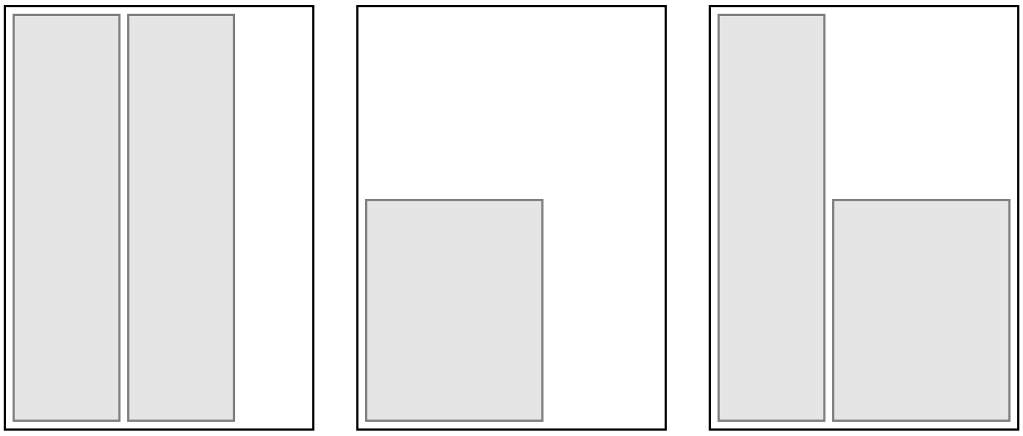
It is worth noting that while these two techniques certainly accelerate the search in most cases, they are by no means failsafe. The heuristic, where we insist on placing the biggest ad on the page, can fail, when the biggest ad is too small to significantly reduce the available space on the page. Even if we have a very high skip factor, we always have two choices in every recursive call: either we place an ad or we raise the skyline, and if the ads are sufficiently small, the search tree becomes too deep, and limiting the branching level does not help us. But, as discussed in the introduction, the problem of layouting small one-column ads can be approximated effectively using different approaches, and thus we allow ourselves certain assumptions on the input.

3.7 Filling Several Pages

Up until now, we have discussed the skyline algorithm, which approximates the optimal page for a given set of ads. To form a complete algorithm for solving the ad packing problem, we embed this functionality in some page selection framework.

The obvious choice is a greedy approach, that iteratively chooses the best page and removes the used ads from the list of available ads, until all ads have been placed. This first idea has some shortcomings, however. Consider an instance of $4n$ ads, consisting of $2n$ three-column ads of full page height and $2n$ five-column ads of height h, where $H/2 < h < 3H/5$. If we consider pages with eight columns, there are only four ways to pack these on a page: a three-column ad by itself, a five-column ad by itself, two three-column ads on a page, or a three-column ad and a five-column ad on a page. The greedy page selection scheme would choose the combination with two three-column ads for the first n pages, since it leaves the smallest amount of space unused. Then each of the five-column ads would be placed on a page by itself, leading to a solution using $3n$ pages, shown in Figure 3.8 (a), whereas the optimal solution, shown in Figure 3.8 (b), uses $2n$ pages.

But this is not only a theoretical result; the greedy behavior is apparent in the results of the experiments as well. For example, in an eight column page setting, experiments showed that the two column ads appear to be very flexible, and due to the greedy nature of the selection scheme, these are used up early in the process. On the other hand, the three column ads tend to be difficult to pack, and thus, end up on the last pages, where they are the cause of much unused space. As an illustration of this situation, consider Figure 3.9. Here a set of ads have been layouted using just the greedy selection scheme, and the problem discussed is evident from the last five pages.



(a) The solution generated by the greedy approach has n pages like the left page and $2n$ pages like the right page.

(b) The optimal solution is $2n$ pages of this type.

Figure 3.8: The solutions for the example in the text.

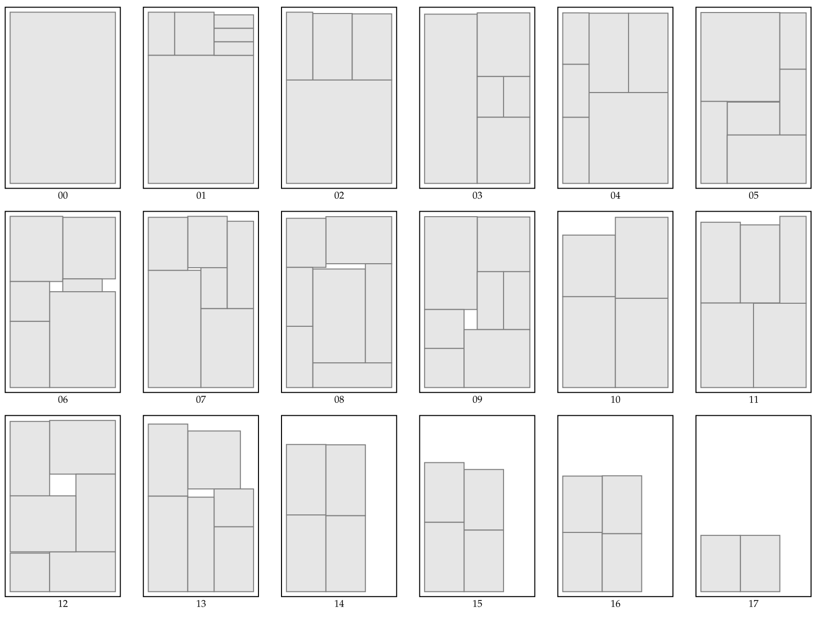


Figure 3.9: After the first run with no weight adjustments.

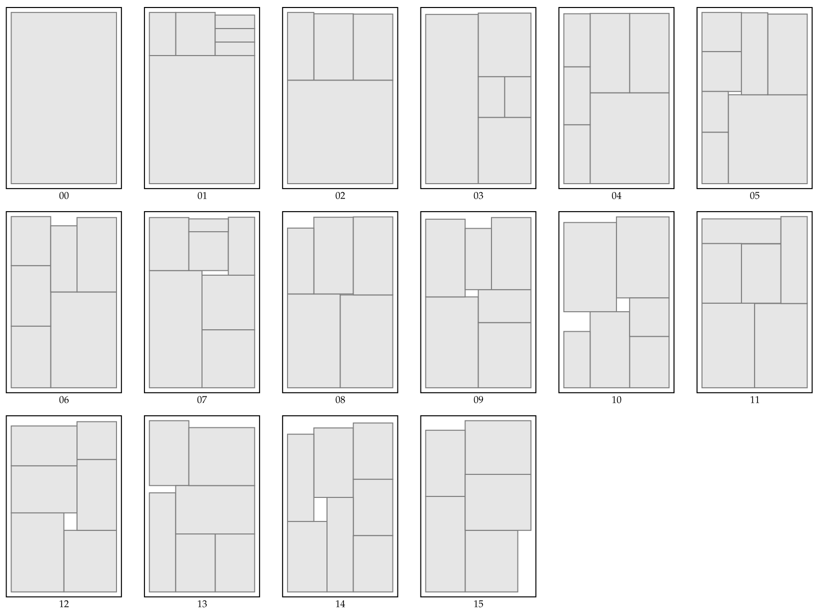


Figure 3.10: Adjusting the weights based on the previous run.

In general, it seems to be the case that some ads are easily utilized by the algorithm, while others are put off for later, causing trouble in the end of the layout process. Furthermore, this degree of usefulness apparently depends on the widths of the ads. This behavior must be contributed to the fact that the number of columns, and thus the number of possible widths, is a small integer. Under these conditions, ads with different widths have different features, and in particular, we would not expect this behavior in general two-dimensional bin packing.

A way to counter the greedy behavior would be to force the algorithm to use the troublesome ads earlier in the process. This way, the troublesome ads are used, when there is still a wide variety of ads available, and the page can be packed tightly, if not optimally. Similarly, if the algorithm can be made to save the flexible ads for later, these will ease the process of packing the last pages.

This idea can be implemented, by assigning a weight to each ad, given by $c(a)$, and then have the skyline algorithm search for the page that maximizes the sum of the *weights* of the ads on the page. By setting these weights to be the area of the corresponding ad, that is,

$$c(a) = w(a)h(a),$$

we get the original algorithm. But if we adjust the weights of the ads slightly, we can account for the preferences of the algorithm. If we adjust the weights, such that the troublesome ads get a higher weight and the flexible ads get a lower weight, we get the behavior outlined above. For example, even though a page is not the optimal page with respect to utilized area, it may be optimal with respect to this new measure, because it uses one or several ads that would otherwise cause problems later on, and thus maximizes the sum of the weights of the ads. On the other hand, a page, which is optimal with respect to utilized area, may be rejected, because it contains too many ads that have had their weight lowered, and consequently, these ads are saved for later.

The question remains how to actually adjust the weights. One strategy would be to increase the weight for the ads on the last, say, twenty percent of the pages. We could multiply the weights of these ads by some small constant, larger then one, and consequently the ads would have a higher weight and appear earlier in new layout. This approach has some shortcomings, though. The adjustment factor is fixed and does not account for the quality of the page, which would seem like a good idea. Also, looking at a fixed percentage of the pages may cause the readjustment process to consider too many or too few pages.

Alternatively, we could judge the *sparseness* of a page as the total page area compared to the amount of space occupied by ads,

$$Q(P) = \frac{WH}{\sum_{a \in A(P)} w(a)h(a)}.$$

To have the weight adjustments correspond to the quality of the page that the ad appear on, we could instead adjust the weight of an ad by multiplying it by $Q(P)$. In this way, ads appearing on near full pages receive almost no adjustment, whereas ads placed on sparse pages have their weights significantly increased. Furthermore, we can use this technique for all pages instead of arbitrarily only considering the last twenty percent, since we expect the first pages to be fairly compact, and thus, the adjustments to the weights of the ads on those pages will be minimal.

But none of these adjustment schemes, make use of the observation that the degree of usefulness of the ads seems to depend on the widths. Consider again the pages in Figure 3.9. We could adjust the weights of the ads, so that the three-column ads from the last five pages would be used earlier by the algorithm. But then the three-column ads from the first pages would be put off for later by the algorithm. Instead we average the adjustment factor over all ads of the same width. This corresponds to interpreting an ad appearing on a sparse page as an indication that ads of that particular width are hard to make use of, not just the actual ad itself.

So, given a solution, S, we compute the weighted average over page sparse-

m	1	2	3	4	5	6	7	8
$Q_m(S)$	-	1.017	1.275	1.032	1.029	1.008	-	1.004

Table 3.1: Adjustments for the example from Figure 3.9.

ness for pages containing m-column ads:

$$Q_m(S) = \frac{\sum_{P \in S} \sum_{a \in A(P), w(a)=m} Q(P)}{|\{ a \in \mathcal{A} \mid w(a) = m \}|},$$

Given the weight function, c, used to produce the solution S, we get a new weight function, c' as

$$c'(a) = c(a) Q_{w(a)}(S).$$

As an example, the adjustments for the pages in Figure 3.9 can be seen in Table 3.1. Adjusting the weights and running the algorithm again gives the pages in Figure 3.10. The adjustment scheme works quite well in practice, though it does not necessarily converge. The typical behavior is that during the first couple of runs of the algorithm, the generated number of pages decreases steadily. After that the number of pages does not change for a couple of runs, and then it increases erraticly (see Figure 5.5 in Section 5.4).

The skip factor can also be adjusted between the iterations, so that the first iterations are done with a high skip factor. These first runs of the algorithm will be fast and accomplish an initial rough adjustment of the weights. Lowering the skip factor in the later iterations enables the algorithm to generate tighter layouts when the weights have been adjusted.

3.8 Constrained Ad Packing

Finally we discuss how the enumeration algorithm and the rest of the framework can be generalized to accommodate the extra constraints formulated in the Constrained Ad Packing Problem, Definition 2.4. Conditions 4 and 5 from can be incorporated with only minimal changes to the algorithm, whereas 6 and to a greater extent 7 require that the enumeration algorithm be adjusted.

First we consider condition 4. We are given an symmetric relation D on $\mathcal{A}$, and it is required that no pair of ads a and b, with $(a, b) \in D$, is placed on the same page.

One way to do this is to, as we place an ad a, remove those ads that can not appear on a page with a from the set that the algorithm is currently considering. For each ad, a, we define the set

$$D_a = \{ b \in \mathcal{A} \mid (a, b) \in D \}.$$

Furthermore, for every ad, a, we maintain a count, δ_a, of how many ads from D_a is currently part of the current configuration. Initially this count is zero. Whenever the algorithm decides to place b on the skyline, D_b is subtracted from the set of remaining ads, and for each $c \in D_b$, δ_c is increased by one. Later, when the algorithm backtracks and removes b, δ_c is decreased correspondingly for every c in D_b. If δ_c reaches 0 for some c, c is put back into the set of remaining ads. This way we will never place ads a and b on the same page, if $(a, b) \in D$.

Alternatively we could mark an ad as placed, whenever it is part of the current configuration. Then, before placing an ad, a, we could check every ad in D_a, and if one of these were marked as placed, we would reject a. However, using the former technique we only consider the ads in D_a when placing a, whereas with the latter technique we would consider and reject them in each subsequent recursive call.

Condition 5 requires that any pair of ads a and b should appear on the same page, if $(a, b) \in F$, where F is assumed to be an equivalence relation. Again, we consider the set of ads related to an ad, a, by F:

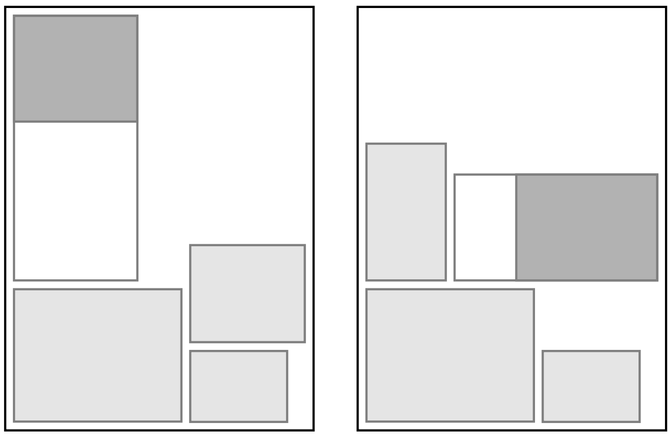
$$F_a = \{\, b \in \mathcal{A} \mid (a, b) \in F \,\}.$$

Here, F_a is the equivalence class containing a, and all these ads should be placed on the same page for the solution to be feasible. In any given configuration, we may have placed a number of ads from a number of different equivalence classes, and for this configuration to end in a feasible page it should be possible to fit the remaining ads from all classes on the page. Now, we can not decide this exactly, but we can get a quick negative answer. If, in some later configuration, the total area of the remaining ads exceeds the amount of free space, that is, the space above the virtual skyline, we just abort the search.

Also, before placing an ad, a, as the first ad from F_a, we can check if the total area of the ads in F_a exceeds the amount of free space in the current configuration. If this is the case, we immediately reject a.

Both of these checks can be done in constant time, if we precompute the total area of the ads in each equivalence class, and if we maintain the amount of free space in the current configuration and, for each equivalence class, the total area of the ads from that class that have been placed so far.

The constraints specified by condition 6, restrict the allowed placements of an ad on a page. If an ad is tied to the left or the bottom edge, the algorithm will naturally place the ad in a position that satisfies the constraint during the enumeration. Whenever this is the case, the algorithm is allowed to place the ad, otherwise the placement is rejected. However, when an ad is tied to



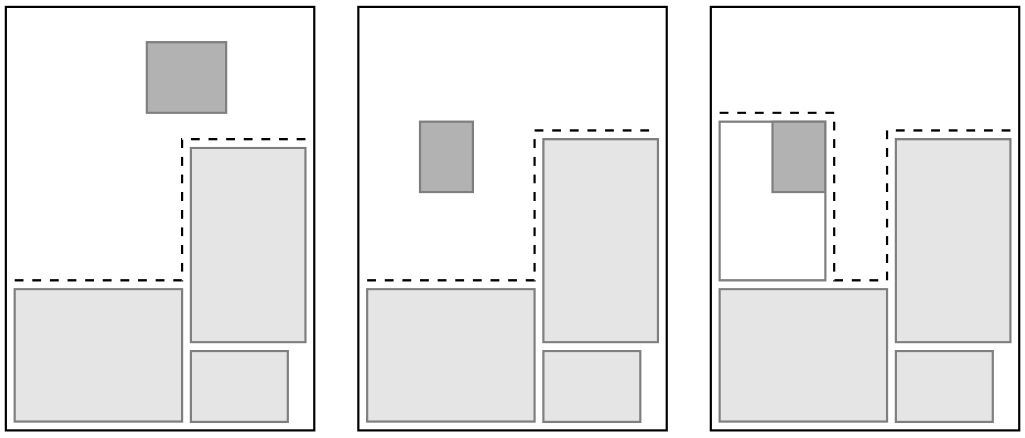
(a) Top border. (b) Right border.

Figure 3.11: Placing ads tied to the top or right borders.

the top or right edge, the algorithm does not necessarily generate a page that respects the constraint, since the algorithm only produces layouts where the ads are pushed down and to the left. In particular, when an ad is tied to the top edge, it is unlikely that there will be a set of ads, such that the sum of their heights equals H. Instead, when an ad is tied to the top edge, we pretend that its height of the ad is exactly such that it touches the upper page edge, as illustrated in Figure 3.11 (a). The skyline is updated as usual, and consequently the space below the ad is left unoccupied. This may seem wasteful, but otherwise the algorithm would consider the same configuration twice. Presumably, in the next iteration the algorithm will try another ad, and then in a later configuration place the ad tied to the top, yielding a better page than that of Figure 3.11 (a), which still satisfies condition 6.

We use the same technique when dealing with ads tied to the right page edge, though, if the current local minimum does not extend all the way to the right edge of the page we reject the placement altogether. Also, if the ad is too narrow it may be rejected, much like the situation in Figure 3.3 (b). The skyline is updated as if the ad actually was that wide. In particular the real skyline is raised to the level of the upper edge of the ad for all columns in the local minimum.

Preplaced ads, as specified by condition 7, are dealt with in much the same way as ads tied to the borders, as we also in this case treat the relevant ads as being bigger than they really are. When we consider a preplaced ad, a, assigned the placement p, we first check wether the ad is placed above the



(a) In this situation, the ad is rejected.

(b) A preplaced ad above the current local minimum.

(c) Updating the skyline.

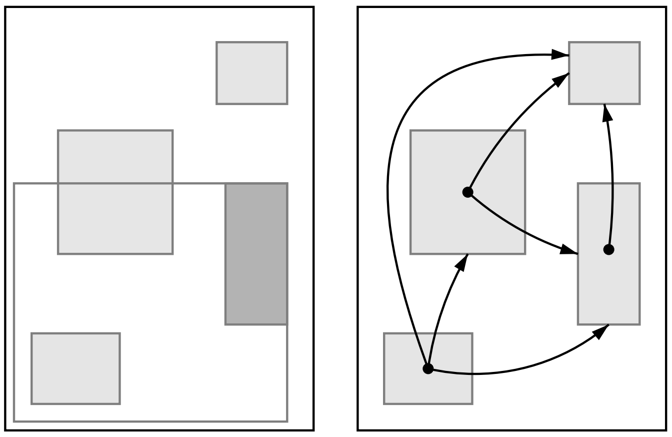
Figure 3.12: Handling preplaced ads.

current local minimum. That is, if the minimum is the interval $[i_0, i_1]$, we require that

$$\alpha(p) \subseteq [i_0, i_1] \times [s_{i_0}, H],$$

where, as before, s_i is the level of the virtual skyline in column i. If this is not the case, as in Figure 3.12 (a), the ad is rejected, but it will be considered in a later configuration. If the ad is placed above the current local minimum, as in Figure 3.12 (b), the placement is allowed. When placing the ad, we update the skyline, as if the ad had been expanded as much as possible downwards and to the left, as suggested by Figure 3.12 (c).

We could incorporate this placement technique into the main loop, so that in every recursive call, we try to place all preplaced ads from a given prelayouted page, $P \in L$, and in the end we reject those pages, where not all ads from $A(P)$ have been placed. However, such a set of ads can only be placed in a certain sequence. Consider Figure 3.13 (a), where a number of preplaced ads are shown. If the shaded ad was placed first, as indicated, the extension of the shaded ad would overlap with two of the other preplaced ads, and these could not be placed in any subsequent configurations. Consequently, no feasible page could be generated, if we chose to place the shaded ad first. To avoid this situation, we define a partial ordering on the ads. We say that an ad placement, $p = (a_p, x_p, y_p)$, is *dominated horizontally* by another



(a) Placing preplaced ads in the wrong order.

(b) Domination relation.

Figure 3.13: Order of placement for preplaced ads.

placement, $q = (a_q, x_q, y_q)$, if

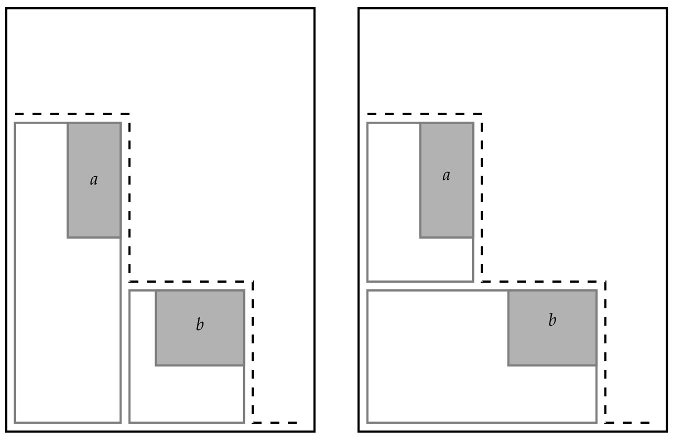
$$x_p + w(a_p) \leq x_q \text{ and } y_p < y_q + h(a_q),$$

or in other words, a_p is placed left of a_q and intersects the extension of a_q downward and to the left. Conversely, p is *dominated vertically* by q, when

$$y_p + h(a_p) \leq y_q \text{ and } x_p < x_q + w(a_q).$$

Finally, p is *dominated* by q, when it is either vertically or horizontally dominated by q or both. Figure 3.13 (b) shows the domination graph for the ads in Figure 3.13 (a). If we only place an ad, a, when all ads dominated by a have been placed, a will never overlap with other preplaced ads.

However, with this extension the skyline algorithm no longer visits each configuration exactly once. Consider the pages in Figure 3.14. From the initial configuration, we can choose to place a and then b, which leads to the configuration shown in Figure 3.14 (a). Placing the ads in the opposite order, gives the configuration in Figure 3.14 (b), which is identical to the other. But in fact both combinations are necessary: a tall, narrow ad could only be placed to the right of a and to the left of b by placing a first. Correspondingly, a low, wide ad could only be placed under a and over b by placing b first.



(a) Placing a first, then b.

(b) Placing b first, then a.

Figure 3.14: Two ways to reach the same configuration.

Chapter 4

Tabu Search Algorithm

We consider a recent approximation algorithm for the two dimensional bin packing problem, proposed by Lodi et al. in [10]. As the algorithm solves the two dimensional bin packing problem, it is immediately applicable to the ad packing problem. The algorithm uses a generic optimization strategy called *tabu search*.

In Section 4.1 we briefly review the area of tabu search. It is not the intention to cover the topic in great detail, rather we seek to introduce the basic idea and explain the concepts present in the algorithm.

In Sections 4.2–4.4 we describe the tabu search algorithm for two dimensional bin packing problem: first, in Section 4.2, we explain the initialization step, then, in Section 4.3, a simple algorithm used as a subroutine in the search, and finally, in Section 4.4 the actual search.

4.1 Tabu Search Overview

As defined in Appendix B.4, an optimization problem asks for a solution, s^*, that maximizes or minimizes $c(s)$ over the set of feasible solutions, $F(x)$, for a given problem instance, x. One way to approximate the optimal solution is to generate an initial feasible solution, and then iteratively improve this solution until some stopping criteria is met. In each iteration, we consider solutions in the *neighborhood*, $N(s)$, of the current solution, s. The neighborhood is the set of solutions reachable by one of a number of *moves*, each of which modifies the current solution by making some local changes, yielding a new solution. By some simple heuristic we choose a solution in $N(s)$; either the solution representing the biggest improvement, a random solution, or just the first solution we consider.

This technique is called *neighborhood search* or *local search*. While conceptually simple and intuitive, this approach has a serious drawback. If the search finds its way to a local optimum, that is, a solution, s, where all neighbor solutions reduces $c(s)$, it necessarily terminates. This local optimum can be very far from the global optimum.

An algorithm guiding the local search so as to explore the neighborhood in a more intelligent manner is a *meta-heuristic.* For example, *genetic algorithms* are based on an abstract model of natural evolution and maintain a number of simultaneous solutions. During the search, the best of these solutions are combined pairwise to form new solutions, replacing older ones. As another example, *Simulated annealing* has its origins in statistical mechanics and models the cooling process of solids. The basic idea is to pick a random neighbor, s' to the current solution, s. The simulated annealing procedure accepts s' as the new current solution, if s' constitute an improvement over s, or else by random choice, depending on $c(s') - c(s)$ and the current *temperature.* The temperature is gradually lowered, and the probability of accepting worse solutions declines.

Tabu search uses memory to guide the search. A tabu search algorithm maintains a list of attributes of recently visited solutions. That is, only part of a solution is remembered, for example, an edge that was recently added to a subgraph, but enough to prevent the search from visiting solutions that were considered just recently. This list is called a *tabu list* and the attributes remembered in the list are *tabu active.* Moves involving a tabu active attribute are prohibited, and thus, the tabu list restricts the search to a subset $N^*(s) \subseteq N(s)$ of the neighborhood of the current solution, s. An attribute remains tabu active for a number of iterations; this period of time is called the *tabu tenure.* This can be a fixed number of iterations, chosen at random within an interval, or more advanced, possibly problem specific, tenure policies can be employed. Tabu lists can be realized by bit vectors, circular lists, or search trees, depending on the number of attributes and the tenure policy. If the attributes correspond to edges in a graph, say, a bit-vector could be used, but if the attributes in question were real valued measures on the solution, a search tree would be better. A tabu search can operate with several types of attributes, and for each type of attribute, a number of tabu lists can be maintained.

Sometimes a tabu rule can be overridden, if certain *aspiration criteria* are met. It could be that $N^*(s) = \emptyset$, that is, all moves contain tabu active attributes, and for the search to be able to continue, the algorithm must revoke the tabu status of some attributes. Another type of aspiration criteria allows a move, when it results in a solution better than any other previous solution.

More advanced tabu search algorithms also incorporate use of long term memory. The long term memory remembers *elite solutions* encountered during the search. These can be used to restart the search elsewhere in the solution space or as a means to revisit promising neighborhoods and study them closer.

For a full treatment on tabu search we refer to the book [8] of Glover and Laguna. Osman and Kelly give an overview of the most successful metaheuristics in [13].

4.2 Initial Feasible Solution

We now turn to the tabu search algorithm by Lodi et al. for the two dimensional bin packing problem. To start the tabu search algorithm, an initial feasible solution is generated by the following simple polynomial time approximation algorithm, IH, described in the article. The IH algorithm is shown to have a worst case performance ratio of 4, which is quite bad, considering that the HFF algorithm, discussed in Section 2.5, has an upper bound of $2\frac{1}{8}$ on its asymptotic worst case performance ratio. However, Lodi et al. argue that the solutions generated by the IH algorithm work better, in that they are easier to modify by the tabu search, and thus, they result in an overall better outcome.

The IH algorithm starts by grouping ads into classes according to their heights. All ads that are higher than half the page height go into class 0, all ads higher than a quarter of the page height, but lower than or equal to half the page height, are put in class 1 and so forth. More formally we have

$$\mathit{class}(a) = \lfloor \log_2(H/h(a)) \rfloor.$$

For a class of ads, r, we define the set of *admissible vertical coordinates*:

$$V(r) = \{\, t\lfloor H/2^r \rfloor \mid 0 \leq t < 2^r \,\}.$$

When an ad, a, is placed at (x, y) it we say that it *occupies* the coordinates

$$\{\, (x', y') \in \mathbb{N}^2 \mid x \leq x' < x + w(a) \land y \leq y' < y + \lfloor H/2^r \rfloor \,\},$$

where $r = \mathit{class}(a)$. In particular, any space above the ad up until the next admissible y-coordinate is occupied.

Assume the ads, $a_1, a_2, \ldots, a_n$ have been ordered by non-decreasing class. The algorithm now proceeds to place the ads in an open ended horizontal strip, as outlined in Figure 4.1. This procedure iteratively places the ads in the leftmost, lower, unoccupied admissible position. The variable $\bar{x}$ is the x-position the algorithm currently considers, and X is the set of possible

```
$\bar{x} \leftarrow 0$
$X \leftarrow \emptyset$

**for** $j \leftarrow 1$ **to** $n$ **do**
    **if** $(\bar{x}, y)$ is occupied for all $y \in V(class(a_j))$ **then**
        $\bar{x} \leftarrow \min(X)$
        $X \leftarrow X \setminus \{\bar{x}\}$
    **end if**

    $\bar{y} \leftarrow \min\{\, y$ is unoccupied $\mid y \in V(class(a_j))\,\}$
    place $a_j$ with its lower left corner at $(\bar{x}, \bar{y})$
    $X \leftarrow X \cup \{\bar{x} + w(a_j)\}$
**end for**
```

Figure 4.1: Pseudo code for the IH algorithm.

future x-positions. When all the ads have been placed, the resulting strip is subdivided along the x-axis into intervals of width W. For each interval, the algorithm creates a page with the ads that are entirely contained within that interval, and if any ads cross over the right boundary, these are placed on a second page. Figure 4.2 shows the result of packing ten ads into a strip. The numbers give the order that the ads were placed in. When the strip is divided into intervals of length W, as indicated, ad 1 will be placed on a page by itself. Ads 2 and 3 cross the boundary between the first and second interval, so these are placed on a second page. Only ad 5 is entirely contained in the second interval, and thus it goes on a page by itself. Finally, ads 4, 6 and 7 cross the interval boundary, and these are placed on a page together, while ads 8 and 9 are contained within the third interval and are put on a page together.

4.3 Finite Best Strip Algorithm

The basic idea of the tabu search algorithm is to move ads between pages to produce a new solution. To determine whether an ad can be added to a page the *finite best strip* algorithm (FBS) by Berkey and Wang from [1] is used. The FBS algorithm is a simple polynomial time approximation algorithm, similar to the HFF algorithm described in Section 2.5. Initially the ads are sorted in order of non-increasing heights and packed into page-wide blocks of a open ended vertical strip. In this context, a block is a rectangle as wide as the page and as high as the highest ad in the block. As the ads are placed

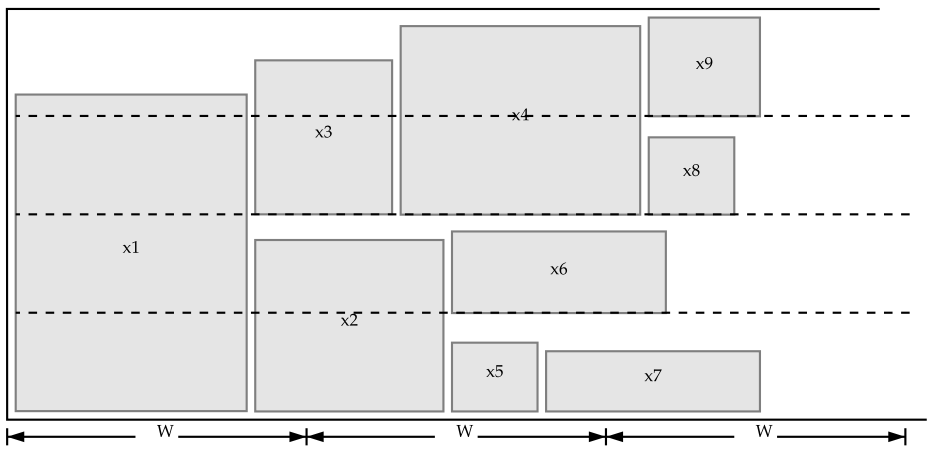


Figure 4.2: Before breaking up the strip.

in non-increasing order, the highest ad is always the first, which is placed at the leftmost position in the block. Subsequent ads are placed from left to right, always on the bottom edge of the block. If no existing block can hold the ad, a new block is added to the stack.

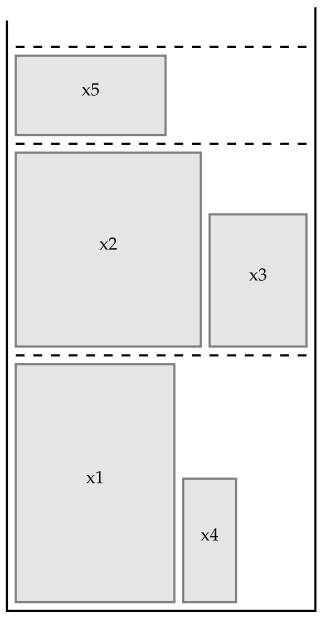
The FBS algorithm uses a *best fit* heuristic for choosing the block for a given ad a, more specifically, it chooses the block where a minimizes the remaining horizontal space. This phase of the algorithm is illustrated in Figure 4.3 (a). Again, the numbers indicate the order, in which the ads were placed. The best fit approach is evident from the fact that ad 3 is placed in the second block; a first fit approach, such as HFF, would place it in the first block.

When all ads have been placed in the strip, the resulting blocks are packed into pages in a similar best fit manner. For the strips in the example, we get the pages in Figure 4.3 (b).

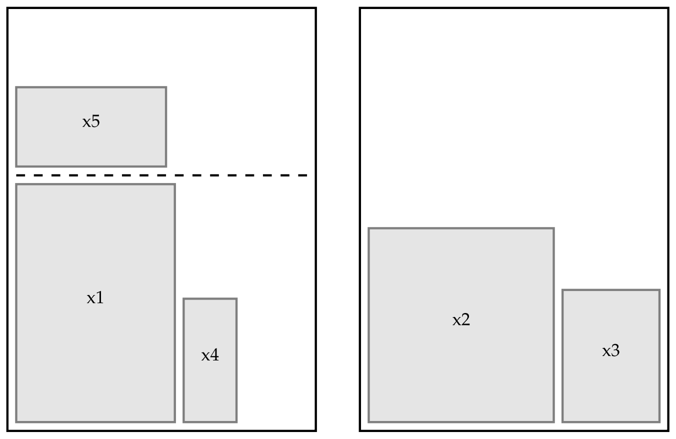
In both phases of the algorithm, the search for the best fit position can be done in time $O(\log n)$, and thus, the overall running time of the algorithm is $O(n \log n)$. In [1], Berkey and Wang experimentally compare FBS to five other simple heuristics for two dimensional bin packing, and FBS is found to be superior in both packing efficiency and time consumption for large data sets.

4.4 Tabu Search Algorithm

Given an initial solution, the actual tabu search improves this by searching two types of neighborhoods alternatingly. Both neighborhoods are defined



(a) Strip packing.



(b) The resulting pages.

Figure 4.3: Finite best strip illustrated.

by moves that try to empty a specific page by moving ads from that page to other pages. The page in question is the *weakest page*, which is defined as the page that minimizes the measure

$$\varphi(P) = \alpha \frac{\sum_{a \in A(P)} w(a)h(a)}{WH} - \frac{|P|}{n},$$

which reflects how easily the page can be emptied. The intuition is that pages, where a large fraction of the area is occupied by ads, are difficult to empty. On the other hand, if the page contain many ads, these must be relatively small, and thus for each of these, it should be easy to find another page that can hold it. The factor α is a pre-fixed nonnegative parameter and provides for a means to adjust the relative importance of these two elements.

The outer loop of the search alternates between searching the two different neighborhoods, as shown in Figure 4.4. The first neighborhood is defined by a move that tries to combine an ad from the weakest page with ads from another page. Ads that were recently moved are marked as tabu active for τ_1 iterations, in which they can not participate in other moves. So, for each non-tabu active ad, a, on the weakest page, P_W, we consider

$$R = \text{FBS}(\{a\} \cup A(P)),$$

for each of the pages, P, in the rest of the current solution. When a page is found, such that $R = 1$, a is moved to this page, marked as tabu active, and the search continues for the next ad in P_W. If the weakest page has only one ad, an aspiration criteria allows the search to try to move this ad off the page, even if it is tabu active. If the search manages to move all ads of the weakest page, the new weakest page is located, and the search continues in the first neighborhood. Otherwise, if no move is possible, the algorithm turns to the second neighborhood. Pseudo-code for the exploration of the first neighborhood is shown in Figure 4.5.

The second neighborhood consists of solutions reachable by recombining an ad from the weakest page with ads from two other pages. Some moves can be performed immediately, after which the algorithm returns to the first neighborhood. But if no such move is available from the current solution, we rate the possible moves with a *score* and in the end choose the best move, if any. The second tabu list records scores for recent moves, and moves that have a tabu active score are prohibited. For each ad, a, on the weakest page, the algorithm considers each pair of pages, P_1 and P_2, from the rest of the current solution. The ads are placed using the FBS algorithm,

$$R = \text{FBS}(\{a\} \cup A(P_1) \cup A(P_2)),$$

and depending on the outcome different actions are taken.

If $R = 1$, the FBS heuristic managed to pack the ads onto one page. This type of move will save one or two pages, depending on whether a was the only ad on P_W. The move is performed at once, after which the search returns to explore the first neighborhood.

In the case where $R = 2$, P_1 and P_2 could be rearranged to also hold a. If a was the last ad in P_W, this move will save a page, and in this situation the search continues in the first neighborhood. Otherwise, P_W may no longer be the weakest page. If it is not, the search continues in the first neighborhood, but if P_W still is the weakest bin, the search continues in the second neighborhood.

When $R = 3$, the FBS heuristic could not rearrange the ads so as to fit on two pages. The ads from the weakest of the three new pages, P', are combined with the remaining ads from P_W, again using the FBS heuristic:

$$R' = \mathrm{FBS}(A(P') \cup A(P_W) \setminus \{a\}).$$

If $R' > 1$, the size of the solution would be increased, and thus, the move is rejected. Otherwise, the solution stay the same size, and then we compute $\varphi(\bar{P})$, where $\bar{P}$ is the page that holds the ads from P' and P_W excluding a. This is the *score* of the move, and if the computed score is not tabu, and furthermore, smaller than all previous scores, the move is remembered as the currently best move.

Finally, if $R > 3$, the move is immediately rejected as the size of the solution would increase.

When all of the second neighborhood has been explored, the best move is performed, and its score marked as tabu active for τ_2 iterations. After this, the search returns to the first neighborhood. Pseudo-code for the exploration of the second neighborhood is shown in Figure 4.6.

If no move was found during the exploration of either neighborhoods, or if the solution was not improved for μ iterations, a *restart action* is triggered. For a solution, S, the $|S|/2$ weakest pages are discarded, and the ads from those pages are placed again, using the IH algorithm.

```
$S \leftarrow \mathrm{IH}(\mathcal{A})$
$P_W \leftarrow \emptyset$
while no stopping criteria is met do
    Search first neighborhood
    Search second neighborhood
end while
```

Figure 4.4: Pseudo code for the tabu search main loop.

```
while first neighborhood moves are possible do
    $P_W \leftarrow P \in S$ minimizing $\varphi(P)$
    for $a \in A(P_W) \setminus \mathrm{TabuList}_1$ do
        if $\exists P \in S \setminus \{P_W\} : \mathrm{FBS}(\{a\} \cup A(P)) = 1$ then
            move $a$ from $P_W$ to $P$
        end if
    end for

    Aspiration criteria
    if $A(P_W) = \{a\} \wedge \exists P \in S \setminus \{P_W\} : \mathrm{FBS}(\{a\} \cup A(P)) = 1$ then
        move $a$ from $P_W$ to $P$
    end if
end while
```

Figure 4.5: Pseudo code for searching the first neighborhood.

```
for $a \in A(P_W)$ do
  for $P1, P2 \in S \setminus \{P_W\}, P_1 \neq P_2$ do
    if FBS($\{a\} \cup A(P_1) \cup A(P_2)) = 1$ then
      perform this move and return to first neighborhood
    end if

    if FBS($\{a\} \cup A(P_1) \cup A(P_2)) = 2$ then
      perform this move
      if $P_W = \emptyset$ then
        return to first neighborhood
      else
        $P' \leftarrow P \in S$ minimizing $\varphi(P)$
        if $P_W \neq P'$ then
          return to first neighborhood
        end if
      end if
    end if

    if FBS($\{a\} \cup A(P_1) \cup A(P_2)) = 3$ then
      $P' \leftarrow$ weakest page of the new pages
      if FBS($A(P') \cup A(P_W) \setminus \{a\}) > 1$ then
        reject the move
      else
        remember the move and its *score*, which is $\varphi(\bar{P})$,
        where $\bar{P}$ is the page that holds $A(P') \cup P_W \setminus \{a\}$
      end if
    end if
end for

perform the move with minimum non-tabu score if any,
mark this score as tabu active for $\tau_2$ iterations, and
return to the first neighborhood
```

Figure 4.6: Pseudo code for searching the second neighborhood.

Chapter 5

Experiments

The skyline algorithm and the tabu search have been implemented, and in this chapter we summarize our experiences from implementing and testing the algorithms. Section 5.1 give a brief overview of the experimental setup, and in Section 5.2 we analyze the test instances from Jyllands-Posten. Section 5.3 presents the results of experiments comparing the skyline algorithm and the tabu search algorithm. Finally, in Section 5.4 we examine the influence of various parameters on the running time of the skyline algorithm.

5.1 Implementation Overview

The skyline algorithm and the tabu search have been implemented in C, within a common framework. Data types, such as pages, layout descriptions and page lists have been implemented, and the lower bound functions and the HFF, FBS and IH heuristics are also part of the framework. The test instances are stored as XML files, and code for reading and writing these files are shared between the two implementations as well. A validate tool for ensuring that the produced solutions are indeed feasible has been implemented, also using the framework.

The data from Jyllands-Posten was converted into the XML format using a small script, and another script was used to visualize the generated solutions, stored in XML files, as PostScript. Figures 3.9 and 3.10 were generated using this script.

All the tools maintain a change log in the XML file they operate on, as can be seen in Figure 5.1, which gives an example of the data file format. The change log makes it easy to keep track of the changes and reproduce results. The change log in the data file in the example records that the file was generated by the `extract.pl` script the 26th of July, and on the 26th of

```
<?xml version="1.0"?>

<adlist>
  <history>
    <entry tool="extract.pl" version="1.7"
           date="Wed Jul 26 22:12:10 CEST 2000">
      extracting data from ToDaimi.000327.tar.gz
    </entry>
    <entry tool="skyline" version="0.1"
           date="Sun Nov 26 17:30:26 2000">
      Number of pages: 16
      Lower bound: 16
      Skip factor: 0.10
      Skip factor step: 0.05
    </entry>
  </history>
  <ad width="8" height="520" id="JP03">
    <placement page="00" x="0" y="0" type="automatic"/>
    <annotation id="adid">3IHALV</annotation>
  </ad>
  <ad width="6" height="75" id="JP14">
    <placement page="11" x="0" y="257" type="automatic"/>
    <annotation id="adid">VMH7SL</annotation>
  </ad>

  ...

</adlist>
```

Figure 5.1: The XML file format.

November the skyline algorithm generated a solution consisting of 16 pages using a skip factor of 0.10.

The implementation source code is listed in Appendix C for reference and browsing, when technical details are not addressed in the text. Also, the complete experimental setup can be downloaded from *the internet* at this address:

`http://www.daimi.au.dk/~hogsberg/thesis.html`.

5.2 Test Instances

Typically, the performance of an approximation algorithm is evaluated by running it on test instances from the literature and comparing with previous results. In the case of the two dimensional bin packing problem, quite a few randomly generated sets of instances have been suggested. Berkey and Wang proposes 6 classes of instances in [1], and these, among others, are often used. However, we can not make use of these instances, since the ad packing problem assumes a small number of columns.

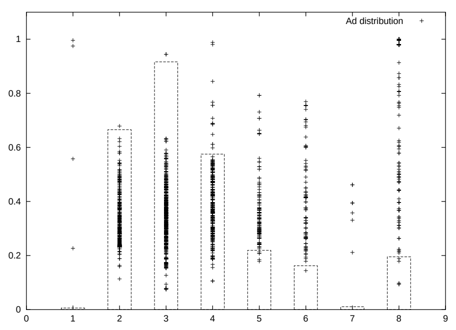
For the purpose of benchmarking the skyline algorithm, Jyllands-Posten and CCI kindly provided data sets corresponding to the vacant positions section in the paper. The dimensions of these ads meet our assumptions regarding minimum ad size, and are layouted in an eight column page setting.

A total of 24 instances have been considered, and the number of ads in each instance varies greatly: from 9 ads in the smallest instance to 186 ads in the biggest instance. The average ad size is 18% of the page size, but this also varies greatly from full page ads to one column ads at 20% of the page height. Most randomly generated instances from the literature are instances with item dimensions that are uniformly distributed over some interval. The distribution of the dimensions of the ads from Jyllands-Posten are far from uniform, though. The bars in the plot in Figure 5.2 (a) shows the distribution on widths for the ads. To give an indication of the range and distribution on heights for the eight different widths, each ad has been plotted as a cross. In Figures 5.2 (b)–(d) the distribution on heights for the three, six and eight column ads are shown in greater detail. As can be seen, the height distribution varies greatly between the different widths; for the three column ads appear to be normally distributed, but it is less obvious what goes on for the six and eight column ads.

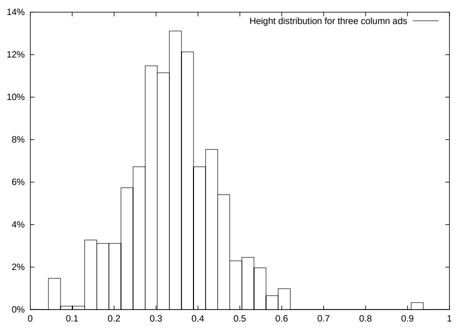
5.3 Benchmarks

The skyline algorithm and the tabu search algorithm have been benchmarked by running each of the algorithms on the 24 data sets. The results are shown in Table 5.3, which also give some further details on the data sets. The first three columns give the name of the instance, the number of ads in the instance, and the average ad area for the instance. The L_{2d} column gives the lower bound for the instance, computed using the technique described in Section 2.4.

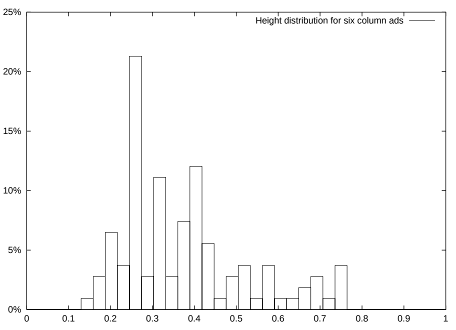
The results for the tabu search algorithm are shown in the column TS. The tabu search was allowed to run for 200 seconds, after which it was terminated. The column t_{TS} gives the running time for the instances that were solved to optimality within the time limit. The parameters for the tabu search were taken from [10], that is, we used $\alpha = 5.0$, tabu tenures of $\tau_1 = 3$ and $\tau_2 = 5$, and restart threshold of $\mu = 30$.



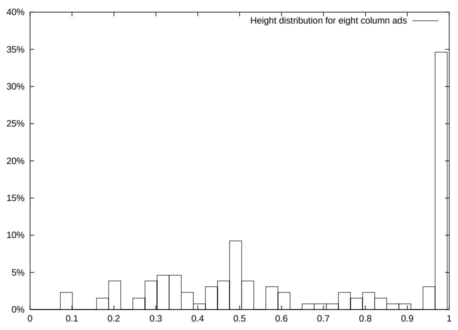
(a) Distribution of all ads from all data sets.



(b) Height distribution of three column ads.



(c) Height distribution of six column ads.



(d) Height distribution of eight column ads.

Figure 5.2: Distribution of the ads.

Instance	n	avg.	L_{2d}	TS	t_{TS}	SL	t_{SL}
`1999-10-27.xml`	140	0.186	27	28	199.66	27*	1.45
`2000-03-27.xml`	89	0.176	16	17	199.82	17	7.08
`2000-04-05.xml`	124	0.185	24	25	199.88	24*	9.31
`2000-05-16.xml`	120	0.187	23	24	199.69	24	11.66
`2000-06-04.xml`	101	0.205	21	22	199.71	22	4.10
`2000-06-07.xml`	55	0.212	12	13	199.88	13	3.34
`2000-06-11.xml`	74	0.214	17	17*	1.57	17*	3.12
`2000-06-14.xml`	67	0.214	16	16*	17.32	16*	1.09
`2000-06-18.xml`	116	0.181	22	23	199.98	22*	7.09
`2000-06-21.xml`	83	0.180	16	16*	0.52	16*	2.35
`2000-06-25.xml`	85	0.188	17	18	199.89	17*	0.26
`2000-06-28.xml`	74	0.195	15	16	199.82	15*	0.50
`2000-07-02.xml`	61	0.165	11	11*	3.20	11*	4.53
`2000-07-05.xml`	52	0.145	8	9	199.86	8*	2.20
`2000-07-09.xml`	40	0.170	7	8	199.97	7*	0.11
`2000-07-12.xml`	22	0.150	4	4*	0.01	4*	0.02
`2000-07-19.xml`	17	0.157	3	4	199.92	3*	0.04
`2000-07-23.xml`	13	0.228	4	4*	0.01	4*	0.02
`2000-07-26.xml`	9	0.153	2	2*	0.01	2*	0.01
`2000-07-30.xml`	48	0.162	8	9	199.88	8*	2.64
`2000-08-02.xml`	36	0.150	6	6*	0.03	6*	0.73
`2000-08-06.xml`	186	0.195	37	39	200.07	38	86.76
`2000-08-09.xml`	136	0.203	28	30	199.99	29	110.86
`2000-08-13.xml`	83	0.201	17	18	199.98	18	47.03
Provably optimal solutions				8		17	

Table 5.1: Performance of the skyline algorithm and the tabu search. Solutions marked with a * are optimal.

Similarly, for the skyline algorithm, columns SL and t_{SL} give the number of pages produced and the running time in seconds. The weight adjustment process was allowed to do at most five iterations, with an initial skip factor of 0.38, which was decreased by 0.05 after each iteration. These parameters were determined experimentally and give good results fairly quickly.

Of the 24 instances, the tabu search was able to determine a provably optimal solution for 8 of the instances. The skyline algorithm, on the other hand, determined a provably optimal solution for 17 of 24 instances, and in all cases it is at least as good as the tabu search. Furthermore, the solutions produced by the skyline algorithm were never more than one page above the lower bound.

n	L_{2d}	SL	t_{SL}
20	5	5*	0.03
40	10	10*	0.17
60	13	13*	0.19
80	17	18	12.50
100	22	23	7.54
120	25	26	16.81
140	28	29	61.43
160	32	34	46.23
180	36	36*	551.71
200	40	41	112.27
220	40	41	87.67
240	42	42*	567.32
260	49	50	332.19
280	54	57	305.64
300	59	61	83.21
320	61	64	192.15
340	66	68	198.95
360	70	72	316.52
380	74	76	343.68
400	78	81	171.55

Table 5.2: Performance of the skyline algorithm on randomly sampled ads.

5.4 Skyline Behavior

In this section we examine the dynamic behavior of the skyline algorithm. In particular, we look at how the running time is affected by the number of ads and the skip factor, and how the number of pages evolve during the iterations.

To test the dependency on number of ads, we have created test instances with up to 400 ads. The instances have been created by uniformly sampling n ads from the set of all ads from all instances. Table 5.4 shows the results of running skyline on these instances; again L_{2d} denote the lower bound, SL the number of pages produced by the skyline algorithm, and t_{SL} the running time in seconds. The running time as a function of n has been plotted in Figure 5.3.

To see how the skip factor affects the running time, we have chosen a subset of the instances and run the skyline on these for increasing skip factors. The results can be seen in Figure 5.4. As expected, the running time tends to decrease, but otherwise, it is rather unpredictable.

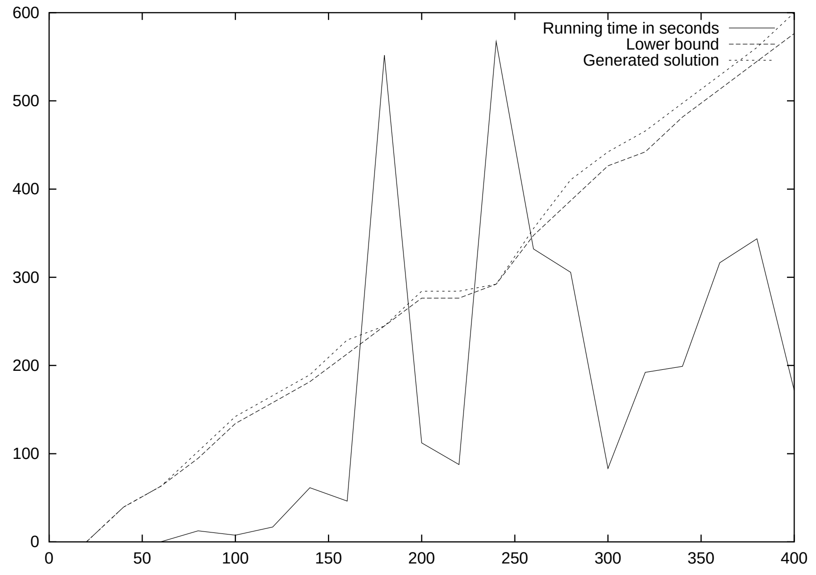


Figure 5.3: Running time as a function of n.

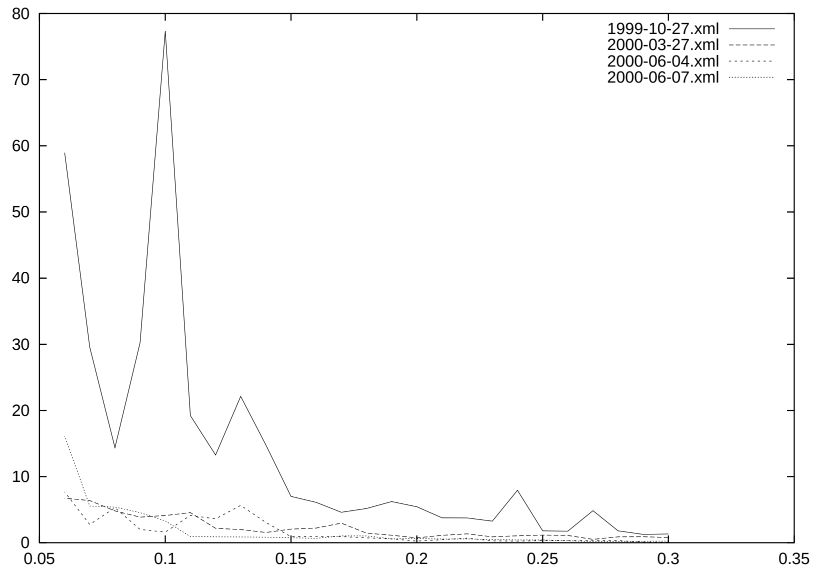


Figure 5.4: Running time as a function of the skip factor.

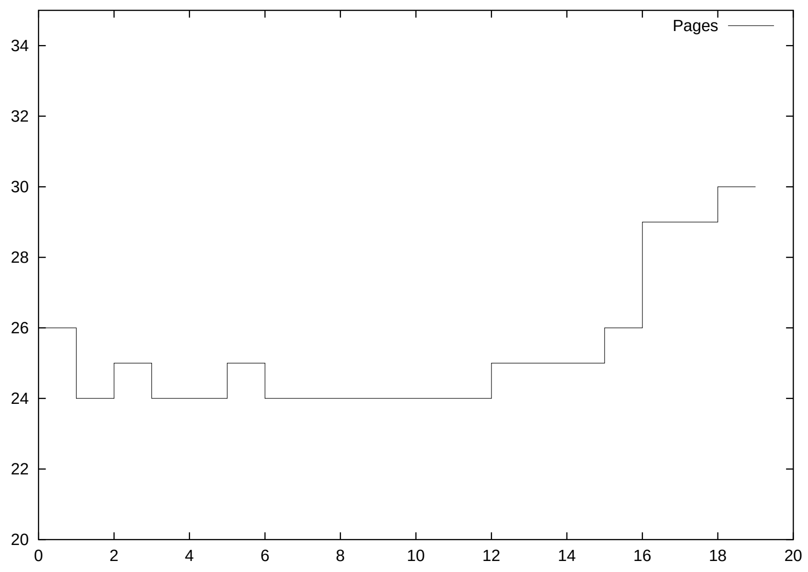


Figure 5.5: A typical behavior of the weight adjustment process.

Finally, in Figure 5.5, we can see how the number of pages change during the iterations, for the problem instance `2000-04-05.xml`. As mentioned in Section 3.7, the number of pages initially decrease, but after that it increases erratically.

Chapter 6

Conclusion

The layout of classified ads in newspapers is a time-consuming and tedious task. Existing tools automate this only to some extent, and typically only for listing-like sections of small ads. When the average ad size increase, the problem of packing the pages becomes puzzle-like, and existing tools only assist by allowing the operator to manually move the ads around.

6.1 Ad Packing

We have stated the simple ad packing problem, which asks for a minimal layout, in terms of number of pages, of a set of ads. This problem is similar to the two-dimensional bin packing problem, and both problems are NP-complete. By also incorporating a number of constraints to control the structure of the generated solution we get the constrained ad packing problem. The constraints considered include restrictions on ad positions and restrictions on which ads can appear on a page together.

Finally, we have reviewed related work from the literature. A lot of attention has been given to the two- and three-dimensional bin packing problems, but no articles directly address the ad packing problem. However, any algorithm that solves the former two problems can also be used to solve the ad packing problem. A variety of approaches has been used, ranging from simple fast heuristics, over different local search algorithms, to exact branch-and-bound frameworks.

6.2 The Skyline Algorithm

We proposed the skyline algorithm as a realistic algorithm for solving the constrained ad packing problem. The algorithm uses a greedy approach for iteratively generating a near optimal page using the remaining ads. This works well for the first pages, but later the quality of the pages decrease. Certain types of ads appear to be difficult to place, and these are put off for last by the algorithm, and consequently, the last pages generated have much unused space. If we associate weights to the ads and optimize the sum of the weights of the ads on a page, we can counter this typical greedy behavior, by adjusting the weights based on results from previous runs.

The extra requirements considered in the constrained ad packing problem can be incorporated into the algorithm by restricting the set of pages that the search visits, to those that satisfy the requirements. This is accomplished by checking a number of criteria before placing an ad or by placing the ad slightly differently.

6.3 Experiments

The proposed algorithm has been implemented and compared experimentally with a tabu search approach by Lodi et al., presented in [10]. For benchmarking the algorithms we have used test instances corresponding to actual classified ads sections from the newspaper Jyllands-Posten. Instances from the literature have not been used, since these are designed for the general two-dimensional bin packing problem, and also, the uniform distribution typically used for generating these instances, does not reflect the actual distribution of ad dimensions.

For the type of instances we consider, the skyline algorithm performs very well, but the experiments also reveal a problem with the algorithm; the skip factor allows for adjusting a time/quality tradeoff, and in general a lower skip factor means longer running time, in return for better solutions. But sometimes the skip factor interacts badly with the problem instance, resulting in unreasonable long running times, as can be seen in Figure 5.4.

Bibliography

[1] J. O. Berkey and P. Y. Wang. Two dimensional finite bin packing algorithms. *Journal of Operational Research Society*, 38:423–429, 1987.

[2] F. R. K Chung, M. R. Garey, and D. S. Johnson. On packing two-dimensional bins. *SIAM Journal on Algebraic and Discrete Methods*, 3:66–76, 1982.

[3] Oluf Faroe, David Pisinger, and Martin Zachariasen. Guided local search for the three-dimensional bin packing problem. Technical Report 99/12, DIKU, University of Copenhagen, 2000.

[4] Sándor P. Fekete and Jörg Schepers. New classes of lower bounds for bin packing problems. In *Integer Programming and Combinatorial Optimization*, volume 1412 of *Springer Lecture Notes in Computer Science*, pages 256–270. Springer, 1998.

[5] Sándor P. Fekete and Jörg Schepers. On more-dimensional packing I: Modelling. Technical Report 97-288, Angewandte Mathematik und Informatik Universität zu Köln, `http://www.zpr.uni-koeln.de`, 2000.

[6] Sándor P. Fekete and Jörg Schepers. On more-dimensional packing II: Bounds. Technical Report 97-289, Angewandte Mathematik und Informatik Universität zu Köln, `http://www.zpr.uni-koeln.de`, 2000.

[7] Sándor P. Fekete and Jörg Schepers. On more-dimensional packing III: Exact algorithms. Technical Report 97-290, Angewandte Mathematik und Informatik Universität zu Köln, `http://www.zpr.uni-koeln.de`, 2000.

[8] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[9] Eleni Hadjiconstantinou and Nicos Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83:39–56, 1995.

[10] A. Lodi, Silvano Martello, and Daniele Vigo. Approximation algorithms for the oriented two-dimensional bin packing problem. *European Journal of Operational Research*, 112:158–166, 1999.

[11] S. Martello and D. Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Science*, 44:388–399, 1998.

[12] John C. Martin. *Introduction to Languages and the Theory of Computing.* McGraw-Hill, Inc., 1991.

[13] Ibrahim H. Osman and James P. Kelly. Meta-heuristics: An overview. In *Meta-Heuristics: Theory & Applications*, pages 1–21. Kluwer Academic Publishers, 1996.

[14] Christos H. Papadimitiou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

Appendix A

Notation Overview

Symbol	Meaning
$\mathcal{A}$	The set of ads
n	The number of ads, $n = \lvert\mathcal{A}\rvert$
$w(a)$	Function $w : \mathcal{A} \to \mathbb{N}$, giving the width of the ad a
$h(a)$	Function $h : \mathcal{A} \to \mathbb{N}$, giving the height of the ad a
$c(a)$	Function $c : \mathcal{A} \to \mathbb{R}$ giving the weight of the ad a
$I_x(p)$	Function giving the interval of columns spanned by the placement p
$I_y(p)$	Function giving the interval on the y-axis spanned by the placement p
$\alpha(p)$	Function giving the set of points covered by the placement p
S	A solution to the ad packing problem
P	A page, that is, a subset of $\mathcal{A} \times \mathbb{N} \times \mathbb{N}$
$\mathcal{P}$	The set of points in a page, that is, $\{ (x, y) \mid 0 \leq x < W \wedge 0 \leq y < H \}$
W	The width of a page in number of columns
H	The height of a page
$A(P)$	The set of ads on a page, $A(P) = \{ a \mid (a, x, y) \in P \text{ for some } x, y \in \mathbb{N} \}$

Appendix B

Basic Theory

In this appendix we offer a review of some essential topics from computational complexity. In Sections B.1 and B.2 we define the Turing machine and the important generalization; the nondeterministic Turing machine. Furthermore, we consider time measurements and the concept of complexity classes. In Section B.3 we introduce reductions and the notion of completeness. Finally, in Section B.4, we examine optimization problems and approximation algorithms.

The treatment below is based on Chapters 16, 17 and 23 from [12] and Chapters 2, 7, 8 and 13 from [14].

B.1 Turing Machines

The Turing machine is a simple construction, originally conceived by Alan Turing in the late 1930's. Though it may seem awkward and inexpressive, it captures what we think of as computability. The Church-Turing thesis states that any "algorithmic procedure" that a person can carry out, can be performed on a Turing machine. This, of course, is not a formal theorem one can prove, but it certainly is the intuition.

The Turing machine can be though of as a finite automaton with an associated infinite sequential access read/write memory. The memory is modeled as an infinite string of symbols (the tape) and a current position in the string (the head). The Turing machine has a finite set of states, and transitions between these are guided by the current state and the contents of the tape. Formally the Turing machine is defined as follows:

Definition B.1 (Turing Machine) *A* Turing machine *is a 4-tuple $M = (K, \Sigma, \delta, s)$. K is the finite set of states, Σ is the finite alphabet, that is, the symbols allowed on the tape, δ is a partial function from $K \times \Sigma$ to $(K \cup \{h\}) \times$*

$\Sigma \times \{\leftarrow, \rightarrow, \cdot\}$ *defining the valid transitions of the machine, and* $s \in K$ *is the initial state. It is assumed that* $h \notin K$.

The tape is thought to have a left end (position 0) that the head can not move past, but no right end: the tape extends infinitely to the right, with uninitialized positions containing the symbol $\sqcup$ (blank). To see how the Turing machine works, we consider a *configuration* of a machine. For a given Turing machine, $M = (K, \Sigma, \delta, s)$, a configuration is a triple (q, x, y) that completely describes the machine at a given point in time: $q \in K$ is the current state, $x \in \Sigma^*$ is the contents of the tape up to the head, and $y \in \Sigma^*$ is the rest of the tape contents, with the tape head being positioned at the first symbol in y. Even though we consider the tape to be infinite, x and y are always finite strings.

Given a configuration (q, x, y), with $x = wb$ and $y = az$ for $a, b \in \Sigma$ and $z, w \in \Sigma^*$, suppose that δ is defined for (q, a) and that $\delta(q, a) = (r, c, d)$. The interpretation is that M moves to state r, replaces a with c on the tape and moves the head left or right one symbol or does not move it, depending on whether d is $\leftarrow$, $\rightarrow$ or $\cdot$. Formally we write this as

$$\begin{aligned}
(q, x, y) &\vdash_M (r, w, bcz) \text{ when } d \text{ is } \leftarrow \\
(q, x, y) &\vdash_M (r, xc, z) \text{ when } d \text{ is } \rightarrow \\
(q, x, y) &\vdash_M (r, x, cz) \text{ when } d \text{ is } \cdot
\end{aligned}$$

The geometry of the tape translates into certain restrictions on the transition function. We require that $\vdash_M$ be undefined for any configuration (q, ε, x), when δ would require the head to move to the right. Also, when $\delta(q, \sqcup) = (r, c, d)$ the transition function should allow moves of the form

$$\begin{aligned}
(q, x, \varepsilon) &\vdash_M (r, w, bc) \text{ when } d \text{ is } \leftarrow \\
(q, x, \varepsilon) &\vdash_M (r, xc, \varepsilon) \text{ when } d \text{ is } \rightarrow \\
(q, x, \varepsilon) &\vdash_M (r, x, c) \text{ when } d \text{ is } \cdot
\end{aligned}$$

in order to realize the infinite tape.

We denote by $\vdash_M^*$ the transitive closure of $\vdash_M$, so that when M can go from configuration (q, x, y) to (r, w, z) in zero or more steps we write

$$(q, x, y) \vdash_M^* (r, w, z)$$

The computation stops when there is no possible move, that is, when δ is not defined in the current configuration, or when the head tries to move left in a configuration of the form (q, ε, x). Whenever the machine moves to state h it will stop, since δ is not defined for any (h, a), $a \in \Sigma$. When this is the case, we say that the machine has *halted.* If by contrast the machine stops in any other state than h, we say that it *crashed.* In the event that the computation does not stop we say that the machine *loops.*

The machine is initialized by means of an *input string*, x. Given M and x the initial configuration is $(s, \triangleright, x)$. The $\triangleright$ is provided for convenience as a beginning of string marker, and could be excluded.

With this setup we can now define the output of a machine, M, on a given input string x. Provided that M halts, the output is the contents of the tape. When M crashes or loops we say that the computation diverges and do not consider the output. We write this as

$$\begin{aligned} M(x) &= y \text{ when } (s, \triangleright, x) \vdash_M^* (h, \triangleright, y) \\ M(x) &= \uparrow \text{ otherwise} \end{aligned}$$

and thus, M can be viewed as a partial function from Σ^* to Σ^*.

As an example, think of a machine, M, that accepts an encoding of a graph, and then decides if the graph has a certain property. The graph could be encoded using adjacency list or an incidence matrix, and we could have M compute 1 if the graph had the desired property or 0 otherwise. Another way to think of this is as M recognizing a language, that is, the subset of Σ^* containing only strings that are valid representations of graphs with the property.

Definition B.2 (Language Recognition) *Given a Turing machine, M, we say that M* recognizes *the language $L \subseteq \Sigma^*$, if $M(x) = 1$ whenever $x \in L$, and $M(x) = 0$ otherwise.*

Notice that we require the machine to explicitly reject strings not in the language by computing $M(x) = 0$. Another notion is that of acceptance: *M accepts L*, provided that $M(x) = y$, for some $y \in \Sigma^*$ when $x \in L$, and $M(x) = \uparrow$ otherwise. However, in the context of computational complexity, language *recognition* is by far the most used criteria.

To measure the execution time of a machine M on input x we will use the number of steps taken by the machine from the initial configuration $(s, \triangleright, x)$ to the final result $(h, \triangleright, y)$. This notion can be extended to entire languages; suppose that M recognizes a language L. If furthermore there exists a function $f : \mathbb{N} \rightarrow \mathbb{N}$, such that for all $x \in \Sigma^*$, M finishes the computation in time $f(|x|)$ or less, we say that *M recognizes L in time f.* The set of languages for which f is a polynomial, is the complexity class P.

B.2 Nondeterministic Machines

A very important generalization of the original Turing machine introduces nondeterminism in the way the machine chooses its actions. In the deterministic machine the transition function δ uniquely determines the next move, whereas in the nondeterministic machine, we want to allow several possible moves from a single configuration.

Definition B.3 (Nondeterministic Turing Machine) *We define a* nondeterministic Turing machine *to be a 4-tuple, $M = (K, \Sigma, \delta, s)$, much like the deterministic Turing machine. However, the transition function δ, now maps $K \times \Sigma$ to a subset of $(K \cup \{h\}) \times \Sigma \times \{\leftarrow, \rightarrow, \cdot\}$. If, for some $(q, a) \in K \times \Sigma$ no move is possible $\delta(q, a)$ will be the empty set, otherwise $\delta(q, a)$ is a set of triples representing the possible transitions.*

Configurations for a nondeterministic Turing machine are defined just as for the deterministic machine. The $\vdash_M$-relation between configurations is defined slightly differently, though. In a given configuration (q, x, y), with $x = wb$ and $y = az$ for $a, b \in \Sigma$ and $z, w \in \Sigma^*$, the nondeterministic Turing machine may take the following actions:

$$\begin{aligned}
(q, x, y) &\vdash_M (r, w, bcz) \text{ when } (r, c, \leftarrow) \in \delta(q, a) \\
(q, x, y) &\vdash_M (r, xc, z) \text{ when } (r, c, \rightarrow) \in \delta(q, a) \\
(q, x, y) &\vdash_M (r, x, cz) \text{ when } (r, c, \cdot) \in \delta(q, a)
\end{aligned}$$

and we see how there may be several possible moves. Again we denote by $\vdash_M^*$ the transitive closure of $\vdash_M$; when there exist a sequence of zero or more steps taking (q, x, y) to (r, w, z), we write $(q, x, y) \vdash_M^* (r, w, z)$.

It does not make sense, however, to talk about the output of a nondeterministic computation, since the machine could halt in several different configurations. Still, though, we can define what it means for a machine to recognize a language:

Definition B.4 (Language Recognition) *Given a nondeterministic Turing machine, M, we say that M* recognizes *the language $L \in \Sigma^*$, when $(s, \triangleright, x) \vdash_M^* (h, \triangleright, 1)$ if and only if $x \in L$. That is, if there from the start configuration exists at least one sequence of steps that causes the machine to halt with 1 on the tape.*

As with deterministic machines, we consider the number of steps taken by the machine as the time measure. However, we must be a bit more precise here, since there may be several computation paths. So, we will say that a machine operate in time k, if *all* computation paths have length at most k, that is

$$(s, \triangleright, x) \vdash_M^\ell (q, y, z) \ \Rightarrow \ \ell \leq k.$$

If a machine M recognizes a language L, and furthermore operate in time $f(|x|)$ for all $x \in \Sigma^*$, we say that *M recognizes L in time f*. The set of languages for which f is a polynomial, is the complexity class NP.

Being a generalization of the deterministic Turing Machine, it is clear that the nondeterministic machine can recognize any language recognized by the deterministic machine, even within the same time bounds. It follows that P $\subseteq$ NP. On the other hand, by simulating all the possible computation traces of the nondetermistic machine in a breadth-first manner, the deterministic machine can perform exactly the same computations as the nondeterministic machine, and thus recognize the same languages. However, the breadth-first search scheme must visit a number of configurations that is exponential in the length of the computation path. Therefore, this construction does not allow us to conclude anything, with regards to whether NP $\subseteq P$, which turns out to be an open problem, if not *the* open problem in computer science.

B.3 Reductions and Completeness

Consider a problem, A, known to be in NP and a problem, B, from P. After several failed attemps to come up with a polynomial time algorithm solving A, one might suspect that somehow A is more difficult than B, and we would like to express this intuition formally. If we knew that A was in NP, but not in P, this would be a strong statement on the relative difficulty of A and B. Furthermore, this would prove that NP $\not\subseteq$ P, but so far, no problems have been proved to lie solely in NP. Instead, to relate problems we say that A is at least as difficult as B, if any instance of B can be solved in terms of A, or more formally:

Definition B.5 (Reduction) *Given languages L_1 and L_2, we say that L_1 is* polynomial time reducible *to L_2, if there exists a deterministic Turing machine, computing R, a function from strings to strings, in time polynomial in the length of the input, such that $x \in L_1$ if and only if $R(x) \in L_2$. We say that R is a* reduction *from L_1 to L_2.*

The restrictions on R are crucial for the definition to be reasonable: if R could be any function, we could actually solve the problem we were reducing from as part of the reduction, and output a simple instance of the problem we were reducing to with the right properties.

Also, notice that if we compose two reductions, R from L_1 to L_2 and R' from L_2 to L_3, the resulting function, $R \circ R'$, is again a reduction; clearly, $x \in L_1$ if and only if $R'(R(x)) \in L_3$, and $R \circ R'$ can be computed in polynomial time by first computing R on x and then computing R' on the output. It follows that reductions are transitive, and thus induce an order on problems, as desired.

To really justify the difficulty of a problem, say A from NP, we could show that A is as hard as any problem in NP, that is, that any problem in NP can be reduced to A. In this case we say that A is NP-complete:

Definition B.6 (Completeness) *Given L, from the language class $\mathcal{C}$, if all $L' \in \mathcal{C}$ are reducible to L, we say that L is* complete *for $\mathcal{C}$, or just $\mathcal{C}$-complete.*

A problem can be proved to be NP-complete, essentially in two ways: either by reducing some other problem already known to be NP-complete to it, or by showing that any problem in NP can be reduced to it. The latter can be achieved, for example, by showing that any nondeterministic Turing machine, M, deciding some $L \in$ NP, with a given input string, x, can be expressed as an instance, $T(M, x)$, of the problem in question, say L', such that $(s, \triangleright, x) \vdash_M^* (h, \triangleright, 1)$ if and only if $T(M, x) \in L'$. Provided that T is computable in time polynomial in x, this constitutes a reduction from L to L'.

If it turns out that $P \subset NP$, then NP-complete languages can not be decided by deterministic machines in polynomial time. Suppose by contradiction that $P \subset NP$ and that M is a deterministic machine deciding the NP-complete language L in polynomial time. It follows that L is also in P. Any problem in NP can be reduced to L, so consider such $L' \in$ NP, reducible to L by R, computed by M'. The composition of M' and M forms a deterministic machine deciding L' in polynomial time, and thus $L' \in$ P. This is a contradiction, since it was assumed that $P \subset NP$.

While the theory of completeness relates *decision problems*, we are usually also interested in the structure of the solutions. For example, the ad packing problem defined in Chapter 2 asks for a minimal set of pages, and thus, apparently we can not argue about this problem. However, we can adjust the problem formulation to get a decision problem: given a *target value* N, does the ads fit on N pages? As shown in Chapter 2 this is indeed an NP-complete problem. If we were to find a polynomial algorithm, solving the ad packing problem, we would trivially obtain an polynomial algorithm solving the decision problem, and thus contradicting the NP-completeness of that problem, or proving P = NP.

B.4 Approximation Algorithms

Optimization problems ask for the maximum or minimum of some measure, for example the weight of a certain subgraph, or the number of pages required to place a set of ads. For an instance, x, of a given optimization problem, A, we denote by $F(x)$ the set of *feasible solutions*. A feasible solution is a solution to the problem instance that has the required properties. For such a solution, $S \in F(x)$, we can evaluate the cost, $c(S)$, a positive integer, which is the measure we want to maximize or minimize. So, in other words, an optimization problems asks for a solution, S^*, maximizing or minimizing $c(S)$ over $F(x)$.

However, many interesting optimization problems have NP-complete corresponding decision problems. As discussed in the previous section, if a problem has been proved to be NP-complete, there is little hope that we can come up with a polynomial time algorithm. On the other hand, this justifies alternative approaches, such as approximation algorithms, the topic of this section, and exhaustive search, the process of considering all possible solutions.

Instead of insisting on finding the optimal solution, an approximation algorithm looks for a solution that is still feasible, but not necessary optimal.

Definition B.7 (Approximation Algorithm) *If for any instance, x, of a minimization problem, A, the algorithm $f(x)$ computes a feasible solution in polynomial time on a deterministic Turing machine, such that*

$$c(f(x)) \leq R \cdot c(S^*),$$

*for a fixed $R \geq 1$, and S^* is the optimal solution, we say that f is an R-*approximation algorithm*. Correspondingly, if A is a maximization problem we require that*

$$c(f(x)) \geq c(S^*)/R.$$

The factor R is the performance ratio *of the algorithm.*

While it certainly makes sense to approximate difficult problems from P with faster polynomial algorithms, the above definition is really geared towards arguing about approximating NP-complete problems using deterministic polynomial time algorithms. After all, any problem in P is 1-approximable. A bit surprising perhaps, not all NP-complete problems can be approximated, assuming P $\neq$ NP. The traveling salesman problem, which asks for a minimum weight tour of the nodes in a graph, is one such problem. Given that we had an R-approximation algorithm for the traveling salesman problem, we could use it to solve the Hamilton cycle problem exactly in polynomial deterministic time. However, the Hamilton cycle problem is also NP-complete, which contradicts the existence of such an algorithm, given that P $\neq$ NP.

Appendix C

Implementation

This appendix contains source code for the implementation of the skyline algorithm and the tabu search. The files `basic-types.h`, `basic-types.c`, `xml.h` and `xml.c` establish a basic framework for loading and saving data files and operating on ads, layouts and pages. The two algorithms have been implemented within this framework, and so has a validate program, which has been used to automate the test phase. Various scripts used to transform between the original data file format, XML and PostScript have been omitted, but is available with the rest of the implementation from `http://www.daimi.au.dk/~hogsberg/thesis.html`.

A cross index of all functions defined in the source code can be found in Section C.7 and should be useful when studying the implementation.

C.1 Basic Data Structures

The `basic-types.h` and `basic-types.c` files implement basic types such as ads, layout descriptions, pages and page lists, as well as a number of convenience functions operating on these. Many of the datastructures are based on the `GList` doubly chained list structure, which is part of the GLib convenience library for C (see `http://www.gtk.org`).

basic-types.h

```
#ifndef BASIC_TYPES_H
#define BASIC_TYPES_H

#include <glib.h>
#include <gnome-xml/parser.h>

#define PAGE_HEIGHT 520
#define PAGE_WIDTH 8
#define PAGE_AREA (PAGE_HEIGHT * PAGE_WIDTH)

typedef struct Page Page;
typedef enum AdFlags AdFlags;
typedef struct Ad Ad;
typedef struct Layout Layout;
typedef struct AdGroup AdGroup;

struct Page {
  char *id;
  int number;
  Layout *layout;
  void (*destroy) (Page *p);
  Page *(*copy) (Page *p);
};

enum AdFlags {
  AD_ALIGN_LEFT   = 1,
  AD_ALIGN_RIGHT  = 2,
  AD_ALIGN_TOP    = 4,
  AD_ALIGN_BOTTOM = 8,
  AD_PREPLACED = 16
};

struct Ad {
  int width, height;
  int weight, key, class;
  char *id;
  AdGroup *group;
  AdFlags flags;
  int min_level[PAGE_WIDTH];
  int preplaced_x, preplaced_y;

  xmlNode *node;
};

struct AdGroup
{
  GList *members;
  int total_area;
  int placed_area;
};

struct Layout {
  Ad *ad;
  int x, y;
  Layout *next;
};

Layout *layout_new (Ad *ad, int x, int y, Layout *next);
void layout_free (Layout *layout);
void layout_free_all (Layout *layout);
GList *layout_validate (Layout *layout);
Layout *layout_copy (Layout *layout);
Layout *layout_remove_ad (Layout *layout, Ad *ad);
int layout_weight (Layout *layout);
int layout_area (Layout *layout);

Page *base_page_new (int number, const char *id, Layout *layout);
Page *base_page_copy (Page *page);

void page_destroy (Page *page);
void base_page_destroy (Page *page);
void page_remove_ad (Page *page, Ad *ad);
void page_merge (Page *page, Page *extra);
Page *page_copy (Page *page);
GList *page_get_ads (Page *page);
double page_compute_sparseness (Page *page);

void page_list_free (GList *list);
GList *page_list_remove (GList *list, Page *p);
GList *page_list_get_ads (GList *list);
void page_list_enumerate (GList *list);
GList *page_list_copy (GList *list);

Ad *ad_list_get_ad (GList *list, const char *id);
void ad_list_dump (GList *list);
```

```
Ad *ad_list_get_biggest_ad (GList *list);
GList *ad_list_compact (GList *list, Layout *layout);
GList *ad_list_subtract (GList *list, GList *remove);

AdGroup *ad_group_new (GList *members);
void ad_group_join (AdGroup *g1, AdGroup *g2);

GList *g_list_split (GList *list, int index);

#endif
```

basic-types.c

```
#include <glib.h>
#include "basic-types.h"

static Layout *layout_free_list;

Layout *
layout_new (Ad *ad, int x, int y, Layout *next)
{
  Layout *l;

  if (layout_free_list != NULL) {
    l = layout_free_list;
    layout_free_list = layout_free_list->next;
  }
  else
    l = g_new (Layout, 1);

  l->ad = ad;
  l->x = x;
  l->y = y;
  l->next = next;

  return l;
}

void
layout_free (Layout *layout)
{
  layout->next = layout_free_list;
  layout_free_list = layout;
}

void
layout_free_all (Layout *layout)
{
  Layout *l;

  if (layout == NULL)
    return;

  l = layout;
  while (l->next != NULL)
    l = l->next;

  l->next = layout_free_list;
  layout_free_list = layout;
}

Layout *
layout_copy (Layout *layout)
{
  Layout *l, *copy;

  copy = NULL;
  for (l = layout; l != NULL; l = l->next)
    copy = layout_new (l->ad, l->x, l->y, copy);

  return copy;
}

int
layout_weight (Layout *layout)
{
  Layout *l;
  int weight;

  for (l = layout, weight = 0; l != NULL; l = l->next)
    weight += l->ad->weight;

  return weight;
}

int
layout_area (Layout *layout)
{
  Layout *l;
  int area;

  for (l = layout, area = 0; l != NULL; l = l->next)
    area += l->ad->width * l->ad->height;

  return area;
}
```

```
GList *
layout_validate (Layout *layout)
{
  GList *errors;
  Layout *l1, *l2;

  errors = NULL;
  for (l1 = layout; l1 != NULL; l1 = l1->next)
    {
      /* Check for overlap between l1->ad and all the other ads.  This
       * is just a simple loop through the ads.  Presumably this could
       * be in time n log n using some clever scheme, but this is
       * intentionally kept simple.
       */
      for (l2 = layout; l2 != NULL; l2 = l2->next)
        {
          if (l1 != l2 &&
              l1->x < l2->x + l2->ad->width &&
              l2->x < l1->x + l1->ad->width &&
              l1->y < l2->y + l2->ad->height &&
              l2->y < l1->y + l1->ad->height)
            {
              errors = g_list_prepend (errors, l1->ad);
              break;
            }
        }

      /* If l1->ad didn't overlap any other ads, check that it's
       * actually placed within the page
       */
      if (l2 == NULL)
        {
          if (l1->x < 0 || l1->x + l1->ad->width > PAGE_WIDTH ||
              l1->y < 0 || l1->y + l1->ad->height > PAGE_HEIGHT)
            errors = g_list_prepend (errors, l1->ad);
        }
    }

  return errors;
}

Layout *
layout_remove_ad (Layout *layout, Ad *ad)
{
  Layout *l, *next;

  if (layout->ad == ad)
    {
      l = layout->next;
      layout_free (layout);
      return l;
    }
  else
    for (l = layout; l != NULL; l = l->next)
      {
        if (l->next != NULL && l->next->ad == ad)
          {
            next = l->next;
            l->next = next->next;
            layout_free (next);
            break;
          }
      }

  return layout;
}

void
layout_dump (Layout *layout, FILE *fp)
{
  Layout *l;

  for (l = layout; l != NULL; l = l->next)
    fprintf (fp, "Ad ‘%s’ at (%d,%d)\n", l->ad->id, l->x, l->y);
}

Page *base_page_new (int number, const char *id, Layout *layout)
{
  Page *page;

  page = g_new (Page, 1);
  page->number = number;
  if (id == NULL)
    page->id = g_strdup_printf ("%d", number);
  else
    page->id = g_strdup (id);
  page->layout = layout;
  page->destroy = base_page_destroy;
  page->copy = base_page_copy;

  return page;
}

void
base_page_destroy (Page *page)
{
  layout_free_all (page->layout);
  g_free (page->id);
  g_free (page);
}

Page *
base_page_copy (Page *page)
{
  Page *copy;

  copy = g_new (Page, 1);
  copy->number = page->number;
  copy->id = g_strdup (page->id);
  copy->layout = layout_copy (page->layout);
  copy->copy = base_page_copy;
  copy->destroy = base_page_destroy;

  return copy;
}

void
page_destroy (Page *page)
{
  page->destroy (page);
}


Page *
page_copy (Page *page)
{
  return page->copy (page);
}

void
page_remove_ad (Page *page, Ad *ad)
{
  page->layout = layout_remove_ad (page->layout, ad);
}

void
page_merge (Page *page, Page *extra)
{
  Layout *l;

  for (l = extra->layout; l != NULL; l = l->next)
    page->layout = layout_new (l->ad, l->x, l->y, page->layout);
}

double
page_compute_sparseness (Page *page)
{
  Layout *l;
  int area;

  area = 0;
  for (l = page->layout; l != NULL; l = l->next)
    area += l->ad->width * l->ad->height;

  return (double) PAGE_AREA / area;
}

GList *
page_get_ads (Page *page)
{
  GList *ads;
  Layout *l;

  ads = NULL;
  for (l = page->layout; l != NULL; l = l->next)
    ads = g_list_prepend (ads, l->ad);

  return ads;
}

void
page_list_free (GList *list)
{
  GList *c;

  for (c = list; c != NULL; c = c->next)
    page_destroy (c->data);

  g_list_free (list);
}

GList *
page_list_remove (GList *list, Page *p)
{
  list = g_list_remove (list, p);
  page_destroy (p);

  return list;
}

void
page_list_enumerate (GList *list)
{
  GList *c;
  int i;
  for (c = list, i = 0; c != NULL; c = c->next, i++)
    {
      Page *p = c->data;

      if (p->id != NULL)
        g_free (p->id);
      p->id = g_strdup_printf ("%02d", i);
      p->number = i;
    }
}

GList *
page_list_copy (GList *list)
{
  GList *c, *copy;

  copy = NULL;
  for (c = list; c != NULL; c = c->next)
    copy = g_list_prepend (copy, page_copy (c->data));

  return g_list_reverse (copy);
}

GList *
page_list_get_ads (GList *list)
{
  GList *c, *ads;

  ads = NULL;
  for (c = list; c != NULL; c = c->next)
    ads = g_list_concat (ads, page_get_ads (c->data));

  return ads;
}

Ad *
ad_list_get_ad (GList *list, const char *id)
{
  GList *c;

  for (c = list; c != NULL; c = c->next)
    {
      Ad *ad = (Ad *) c->data;

      if (strcmp (ad->id, id) == 0)
        return ad;
    }

  return NULL;
```

```
}

Ad *
ad_list_get_biggest_ad (GList *list)
{
  GList *c;
  Ad *ad, *biggest_ad;

  biggest_ad = NULL;

  for (c = list; c != NULL; c = c->next)
    {
      ad = (Ad *) c->data;

      if (ad == NULL)
        continue;

      if (biggest_ad == NULL ||
          biggest_ad->width * biggest_ad->height < ad->width * ad->height)
          biggest_ad = ad;
    }

  return biggest_ad;
}

static int compare_ad_id (Ad *ad1, Ad *ad2)
{
  return ad1->id - ad2->id;
}

GList *
ad_list_subtract (GList *list, GList *remove)
{
  GList *c;

  for (c = remove; c != NULL; c = c->next)
    list = g_list_remove (list, c->data);

  return list;
}

AdGroup *
ad_group_new (GList *members)
{
  AdGroup *group;
  GList *c;
  int area;

  group = g_new (AdGroup, 1);
  group->members = g_list_copy (members);

  for (c = members; c != NULL; c = c->next)
    {
      Ad *ad = c->data;

      if (ad->group != NULL)
        ad_group_join (group, ad->group);
      else
        {
          group->members = g_list_prepend (group->members, ad);
          ad->group = group;
        }
    }

  area = 0;
  for (c = group->members; c != NULL; c = c->next)
    {
      Ad *ad = c->data;

      area += ad->width * ad->height;
    }

  group->total_area = area;
  group->placed_area = 0;

  return group;
}

void
ad_group_join (AdGroup *g1, AdGroup *g2)
{
  GList *c;

  if (g1 == g2)
    return;

  for (c = g2->members; c != NULL; c = c->next)
    {
      Ad *ad = c->data;

      ad->group = g1;
      g1->members = g_list_prepend (g1->members, ad);
    }

  g_free (g2);
}

GList *
```

```
g_list_split (GList *list, int index)
{
  GList *p;

  p = g_list_nth (list, index);
  p->prev->next = NULL;
  p->prev = NULL;

  return p;
}
```

C.2 Data File Functions

The `xml.h` and `xml.c` files implement the input and output functions for the data files. The actual parsing and writing of XML files is done by the library `libxml`, available from `www.xmlsoft.org`. The functions in the files below convert the data structures returned by `libxml` to the data structures defined in `basic-types.h` and vice versa.

xml.h

```
#ifndef XML_H
#define XML_H

#include <glib.h>
#include <gnome-xml/parser.h>
#include "basic-types.h"

GList *xml_get_ads (xmlDoc *doc);
GList *xml_get_pages (xmlDoc *doc, GList *ads, const char *plc_type);

void xml_update_ads (GList *ads);
void xml_update_pages (GList *pages);
void xml_add_history_entry (xmlDoc *doc,
                            const char *tool,
                            const char *version,
                            const char *content);

#endif /* XML_H */
```

xml.c

```
#include <stdlib.h>
#include <tree.h>
#include <xmlmemory.h>
#include <time.h>
#include <unistd.h>
#include <glib.h>
#include "basic-types.h"

static int
xml_node_get_int_prop (xmlNode *node, const char *name, int default_val)
{
  char *val = xmlGetProp (node, name);
  int ret;

  if (val != NULL)
    {
      ret = atoi (val);
      xmlFree (val);
    }
  else
    ret = default_val;

  return ret;
}

static char *
xml_node_get_str_prop (xmlNode *node, const char *name,
                       const char *default_val)
{
  char *val = xmlGetProp (node, name);
  char *ret;

  if (val != NULL)
    {
      ret = g_strdup (val);
      xmlFree (val);
    }
  else
    ret = default_val ? g_strdup (default_val) : NULL;

  return ret;
}

static void
xml_node_set_int_prop (xmlNode *node, const char *name, int val)
{
  char buf[100];

  g_snprintf (buf, 100, "%d", val);
  xmlSetProp (node, name, buf);
}

/* Given an xmlNode, node, look for the child element with name name.
 * If id is non-null, also check that the element has an attribute
 * called id, whose value matches id.
 */

static xmlNode *
xml_node_get_child (xmlNode *node, const char *name, const char *id)
{
  xmlNode *n;

  for (n = node->childs; n != NULL; n = n->next)
    if (strcmp (n->name, name) == 0)
      {
        if (id != NULL)
          {
            char *actual_id = xml_node_get_str_prop (n, "id", NULL);

            if (actual_id != NULL && strcmp (actual_id, id) == 0)
              return n;
          }
        else
          return n;
      }

  return NULL;
}

static xmlNode *
xml_node_set_child (xmlNode *node, const char *name, const char *id,
                    const char *content)
{
  xmlNode *n;

  n = xml_node_get_child (node, name, id);
  if (n == NULL)
    {
      n = xmlNewChild (node, NULL, name, content);
      xmlSetProp (n, "id", id);
    }
  else
    xmlNodeSetContent (n, content);

  return n;
}

static Page *
get_page (GList **pages, const char *id)
{
  GList *c;
  Page *p;

  for (c = *pages; c != NULL; c = c->next)
    {
      Page *p = c->data;
      if (strcmp (p->id, id) == 0)
        return p;
    }

  p = base_page_new (0, id, NULL);
  *pages = g_list_prepend (*pages, p);

  return p;
```

```
}

static Ad *
ad_new_from_xml_node (xmlNode *node)
{
  Ad *ad;
  static int id;
  char buf[100], *p;

  ad = g_new (Ad, 1);

  ad->width = xml_node_get_int_prop (node, "width", 0);
  ad->height = xml_node_get_int_prop (node, "height", 0);
  ad->weight = xml_node_get_int_prop (node, "weight", ad->width * ad->height);
  ad->key = id;
  g_snprintf (buf, 100, "%d", id);
  ad->id = xml_node_get_str_prop (node, "id", buf);
  ad->node = node;
  ad->group = NULL;
  id++;

  ad->flags = 0;
  p = xml_node_get_str_prop (node, "valign", NULL);
  if (p != NULL)
    {
      if (strcmp (p, "top") == 0)
        ad->flags |= AD_ALIGN_TOP;
      else if (strcmp (p, "bottom") == 0)
        ad->flags |= AD_ALIGN_BOTTOM;

      g_free (p);
    }

  p = xml_node_get_str_prop (node, "halign", NULL);
  if (p != NULL)
    {
      if (strcmp (p, "left") == 0)
        ad->flags |= AD_ALIGN_LEFT;
      else if (strcmp (p, "right") == 0)
        ad->flags |= AD_ALIGN_RIGHT;

      g_free (p);
    }

  return ad;
}

GList *
xml_get_ads (xmlDoc *doc)
```

```
{
  GList *c;
  xmlNode *node;
  Ad *ad;

  c = NULL;
  for (node = doc->root->childs; node != NULL; node = node->next)
    {
      if (strcmp (node->name, "ad") != 0)
        continue;
      ad = ad_new_from_xml_node (node);
      c = g_list_prepend (c, ad);
    }

  return g_list_reverse (c);
}

/* if plc_type is NULL we want all placements */

GList *
xml_get_pages (xmlDoc *doc, GList *ads, const char *plc_type)
{
  GList *c;
  xmlNode *node, *plc;
  Page *p;
  Ad *ad;
  char *type, *id, *page_id;
  int x, y;

  c = NULL;
  for (node = doc->root->childs; node != NULL; node = node->next)
    {
      if (strcmp (node->name, "ad") != 0)
        continue;
      plc = xml_node_get_child (node, "placement", NULL);
      if (plc != NULL)
        {
          type = xml_node_get_str_prop (plc, "type", NULL);
          if (plc_type == NULL ||
              (type != NULL && strcmp (type, plc_type) == 0))
            {
              page_id = xml_node_get_str_prop (plc, "page", 0);
              x = xml_node_get_int_prop (plc, "x", 0);
              y = xml_node_get_int_prop (plc, "y", 0);
              id = xml_node_get_str_prop (node, "id", NULL);
              g_assert (id != NULL);

              p = get_page (&c, page_id);
              ad = ad_list_get_ad (ads, id);
              g_free (page_id);
              p->layout = layout_new (ad, x, y, p->layout);
            }
          g_free (type);
        }

    }

  return c;
}

void
xml_update_ads (GList *ads)
{
  GList *c;

  for (c = ads; c != NULL; c = c->next)
    {
      Ad *ad = c->data;

      if (ad->weight != ad->width * ad->height)
        xml_node_set_int_prop (ad->node, "weight", ad->weight);
    }
}

static void
update_layout (Page *page, gpointer null)
{
  Layout *l;
  xmlNode *plc;

  for (l = page->layout; l != NULL; l = l->next)
    {
      char buf[100];

      if (xml_node_get_child (l->ad->node, "annotation", "adid") == NULL)
        xml_node_set_child (l->ad->node, "annotation", "adid", l->ad->id);
      plc = xml_node_set_child (l->ad->node, "placement", NULL, NULL);
      g_snprintf (buf, 100, "%02d", page->number);
      xmlSetProp (plc, "page", buf);
      if (!(l->ad->flags & AD_PREPLACED))
        xmlSetProp (plc, "type", "automatic");
      xml_node_set_int_prop (plc, "x", l->x);
      xml_node_set_int_prop (plc, "y", l->y);
    }
}

void
```

```
xml_update_pages (GList *pages)
{
  g_list_foreach (pages, (GFunc) update_layout, NULL);
}

void
xml_add_history_entry (xmlDoc *doc,
                       const char *tool,
                       const char *version,
                       const char *content)
{
  xmlNode *node, *entry;
  char *time_string;
  time_t t;

  for (node = doc->root->childs; node != NULL; node = node->next)
    {
      if (strcmp (node->name, "history") == 0)
        break;
    }

  if (node == NULL)
    {
      node = xmlNewChild (doc->root->childs, NULL, "history", NULL);
      xmlAddNextSibling (doc->root->childs, node);
    }

  t = time (NULL);
  time_string = ctime (&t);

  entry = xmlNewChild (node, NULL, "entry", content);
  xmlSetProp (entry, "tool", tool);
  xmlSetProp (entry, "version", version);
  xmlSetProp (entry, "date", g_strchomp (time_string));
}
```

C.3 Skyline Algorithm Implementation

The `skyline.h` and `skyline.c` files implement the skyline algorithm. The function `skyline_iterate` corresponds roughly to the pseudo-code in Figure 3.5, `skyline_place_ad` implements the test for wether an ad can be placed and also the updating of the skyline data structure. The function `skyline_raise` implements the process of raising the skyline and the function `skyline_update_minimum` locates the leftmost local minimum for a given skyline.

The `skyline-main.c` file handle argument parsing and reading and writing datafiles. Also in this file is the function `adjust_weights`, which is responsible for adjusting the weights of the ads as described in Section 3.7.

skyline.h

```
#ifndef SKYLINE_H
#define SKYLINE_H

#include "basic-types.h"

typedef struct Skyline Skyline;
typedef struct SkylinePage SkylinePage;

struct Skyline {
  /* Location of leftmost local minimum. min_start is first
     colum inside the minimum, min_end is first column
     outside. */
  int min_start, min_end;

  /* Length of non-real prefix of skyline at minimum. */
  int virtual_length;

  /* This is the height of the gap from the baseline of the
     minimum to the lower edge of the top ad to the left. */
  int gap;

  /* The real and the virtual skyline. */
  int real_skyline[PAGE_WIDTH], virtual_skyline[PAGE_WIDTH];

  /* The actual layout corresponding to the current skyline. */
  Layout *layout;

  /* The current weight and occupied area. */
  int weight, area, waste;

  /* In columns where the real and the virtual skyline
```

```
     coincide, this array holds the address of the topmost
     ad.  Otherwise the corresponding entry is NULL. */
  Ad *top_ads[PAGE_WIDTH];
};

struct SkylinePage {
  Page page;
  Layout *preplaced_ads;
  int weight, waste;
  int initial_skyline[PAGE_WIDTH], column_height[PAGE_WIDTH];
};

SkylinePage *skyline_page_new (int number);
void         skyline_page_calculate_skyline (SkylinePage *p);
void         skyline_page_print (SkylinePage *p, FILE *fp);
void         skyline_page_free (SkylinePage *page);

void         skyline_init (Skyline *skyline);
void         skyline_update_minimum (Skyline *skyline);
int          skyline_raise (Skyline *skyline, SkylinePage *page);
int          skyline_ad_fits (Skyline *skyline, SkylinePage *page, Ad *ad);
void         skyline_iterate (SkylinePage *page, GList *list,
                              Skyline *skyline);
GList       *skyline_layout_pages (GList *ad_list, GList *preplaced,
                                   int iteration, double skip_factor,
                                   FILE *progress);
void         skyline_backtrack (Skyline *skyline,
                                Skyline *original_skyline, Ad *ad);

#endif
```

skyline.c

```
#include <glib.h>
#include <stdlib.h>
#include <math.h>
#include <values.h>
#include <gnome-xml/parser.h>
#include <string.h>
#include "skyline.h"

static double skip_factor ;
static Ad *first_ad;
static Layout *preplaced_table[PAGE_WIDTH][PAGE_WIDTH];
static int preplaced_area, current_preplaced_area;

static void
preplaced_ads_tabulate (Layout *layout)
```

```
{
  Layout *l;
  int i, w;

  preplaced_area = 0;
  for (l = layout; l != NULL; l = l->next)
    {
      for (i = 0; i < PAGE_WIDTH; i++)
        {
          for (w = 0; w < PAGE_WIDTH; w++)
            if (i < l->x + l->ad->width &&
                l->x < i + w + 1 &&
                i + w + 1 <= PAGE_WIDTH)
              {
                preplaced_table[i][w] =
                  layout_new (l->ad, l->x, l->y, preplaced_table[i][w]);
              }
        }
      preplaced_area += l->ad->width * l->ad->height;
    }


}

void
preplaced_ads_free (void)
{
  int i, w;

  for (i = 0; i < PAGE_WIDTH; i++)
    for (w = 0; w < PAGE_WIDTH + 0; w++)
      {
        layout_free_all (preplaced_table[i][w]);
        preplaced_table[i][w] = NULL;
      }
  preplaced_area = 0;
}

static void skyline_page_destroy (SkylinePage *page);

SkylinePage *
skyline_page_new (int number)
{
  SkylinePage *page;
  int i;

  page = g_new (SkylinePage, 1);

  page->page.number = number;
  page->page.layout = NULL;
  page->weight = 0;
  page->preplaced_ads = NULL;
  page->page.id = g_strdup_printf ("%d", number);

  for (i = 0; i < PAGE_WIDTH; i++)
    {
      page->initial_skyline[i] = 0;
      page->column_height[i] = PAGE_HEIGHT;
    }

  page->page.destroy = (void (*)(Page *)) skyline_page_destroy;

  return page;
}

void
skyline_page_destroy (SkylinePage *page)
{
  layout_free_all (page->preplaced_ads);
  layout_free_all (page->page.layout);
  g_free (page->page.id);
  g_free (page);
}

void
skyline_init (Skyline *skyline)
{
  int i;

  skyline->min_start = 0;
  skyline->min_end = PAGE_WIDTH;
  skyline->virtual_length = 0;
  skyline->gap = 0;
  skyline->weight = 0;
  skyline->area = 0;
  skyline->layout = NULL;

  for (i = 0; i < PAGE_WIDTH; i++)
    {
      skyline->top_ads[i] = NULL;
      skyline->real_skyline[i] = 0;
      skyline->virtual_skyline[i] = 0;
    }

  skyline_update_minimum (skyline);
}

/* Find the leftmost local minimum of the new skyline */
```

```
void
skyline_update_minimum (Skyline *skyline)
{
  int level, i;

  skyline->min_start = 0;
  level = skyline->virtual_skyline[0];

  for (i = 1; i < PAGE_WIDTH; i++)
    if (skyline->virtual_skyline[i] < level) {
      skyline->min_start = i;
      level = skyline->virtual_skyline[i];
    }
    else if (skyline->virtual_skyline[i] > level)
      break;
  skyline->min_end = i;

  /* Calculate the length of the initial not-real skyline of the new
   * minimum. */

  for (i = skyline->min_start; i < skyline->min_end; i++)
    if (skyline->virtual_skyline[i] == skyline->real_skyline[i])
      break;
  skyline->virtual_length = i - skyline->min_start;
}

/* Lower the virtual skyline in the interval [start; end) so it forms
 * an increasing staircase.  lower_edge is the y-coordinate of the
 * lower edge of the ad that make up the left border of the minimum,
 * which is used to calculate the new gap.
 */

void
skyline_lower (Skyline *skyline, int start, int end, int lower_edge)
{
  int level, i;

  level = skyline->real_skyline[start - 1];
  for (i = start; i < end; i++)
    {
      if (skyline->real_skyline[i] > level)
        level = skyline->real_skyline[i];
      skyline->virtual_skyline[i] = level;
    }

  skyline->gap = lower_edge - skyline->virtual_skyline[start];
}
```

```
static int
skyline_place_ad (Skyline *skyline, SkylinePage *page, Ad *ad)
{
  Layout *l;
  int i, level, x0, x1, aw;
  int lower_edge, left_edge;

  x0 = skyline->min_start;
  x1 = skyline->min_end;
  aw = ad->width;
  left_edge = x0;
  level = skyline->virtual_skyline[x0];
  lower_edge = level;

  if (ad->flags != 0)
    {
      /* Check that AD_TIE_BOTTOM and AD_TIE_LEFT are satisfied */

      if ((ad->flags & AD_ALIGN_BOTTOM) && level > 0)
        return FALSE;

      if ((ad->flags & AD_ALIGN_RIGHT) && x0 > 0)
        return FALSE;

      /* AD_TIE_RIGHT is implemented by pretending the ad extends to
       * the right edge.  */

      if (ad->flags & AD_ALIGN_LEFT)
        left_edge = PAGE_WIDTH - aw;

      /* AD_TIE_TOP is implemented by placing the ad at the top. */

      if (ad->flags & AD_ALIGN_TOP)
        lower_edge = PAGE_HEIGHT - ad->height;

      if (ad->flags & AD_PREPLACED)
        {
          if (level < ad->min_level[x0])
            return FALSE;
          left_edge = ad->preplaced_x;
          lower_edge = ad->preplaced_y;
        }
    }

  /* See if the ad fits in the minimum.  The ad shouldn't be wider
     than the minimum, it should rest on part of the real skyline and
     it should be taller than the gap. */

  if (left_edge + aw > x1)
```

```
    return FALSE;

  if (lower_edge + ad->height > PAGE_HEIGHT)
    return FALSE;

  if (left_edge + aw <= x0 + skyline->virtual_length)
    return FALSE;

  if (lower_edge + ad->height <= level + skyline->gap)
    return FALSE;

  /* Check for symmetry conditions only if the ad hasn't got any
   * border constraints, e.g. if the ad with the smallest key was tied
   * to the top it would never be placed.  Also, we dont want to
   * reject preplaced ads for symmetry reasons. */

  if (ad->flags == 0)
    {
      /* Stack ads in order of bottom up increasing ad->key only. */

      if (skyline->top_ads[x0] != NULL &&
          aw == skyline->top_ads[x0]->width &&
          skyline->top_ads[x0]->key > ad->key)
        return FALSE;

      /* More symmetry considerations: did this pair occur earlier? */

      if (level == 0 && first_ad != NULL &&
          x0 + aw == PAGE_WIDTH &&
          first_ad->key > ad->key)
        return FALSE;
    }

  /* Does this ad overlap with some preplaced ad? */

  if (!(ad->flags & AD_PREPLACED))
    {
      l = preplaced_table[x0][left_edge + aw - x0 - 1];
      while (l != NULL)
        {
          if (level < l->y + l->ad->height &&
              l->y < lower_edge + ad->height)
            return FALSE;
          l = l->next;
        }
    }

  /* OK, we can place the ad.  Update the skyline accordingly.  If we
   * cut the virtual skyline, then lower it and calculate new gap.
   * Otherwise just remove the gap.  We should do this before updating
   * the real skyline, since the lowering function needs to know
   * real_skyline as it were before placing the ad. */

  if (x0 + aw < PAGE_WIDTH && skyline->real_skyline[x0 + aw - 1] < level)
    skyline_lower (skyline, x0 + aw, x1, lower_edge);
  else
    skyline->gap = lower_edge - level;

  for (i = x0; i < left_edge; i++)
    {
      skyline->virtual_skyline[i] = lower_edge + ad->height;
      skyline->top_ads[i] = NULL;
    }
  for (i = left_edge; i < left_edge + aw; i++)
    {
      skyline->area += lower_edge + ad->height - skyline->real_skyline[i];
      skyline->virtual_skyline[i] = lower_edge + ad->height;
      skyline->real_skyline[i] = lower_edge + ad->height;
      skyline->top_ads[i] = NULL;
    }
  skyline->top_ads[left_edge] = ad;

  if (level == 0 && x0 == 0)
    first_ad = ad;

  skyline->weight += ad->weight;
  skyline->layout = layout_new (ad, left_edge, lower_edge, skyline->layout);

  if (ad->flags & AD_PREPLACED)
    current_preplaced_area += ad->width * ad->height;

  skyline_update_minimum (skyline);

  return TRUE;
}

/* Raise the skyline for current minimum, so it has the same height as
 * its lowest neighbour column.  If the skyline was raised TRUE is
 * returned otherwise (if the minimum spans the whole page) FALSE is
 * returned. */

int
skyline_raise (Skyline *skyline, SkylinePage *page)
{
  int new_level, level, i, width;
  int x0, x1;

  x0 = skyline->min_start;
  x1 = skyline->min_end;

  /* Determine level to raise to by looking at neighbour columns, if
   * any.  It doesn't matter wether we look at the virtual or the real
   * skyline, as they coincide in this case. */

  new_level = MAXINT;
  if (x0 > 0)
    new_level = skyline->virtual_skyline[x0 - 1];
  if (x1 < PAGE_WIDTH)
    new_level = MIN (new_level, skyline->virtual_skyline[x1]);

  level = skyline->virtual_skyline[x0];
  width = x1 - x0;

  if (new_level != MAXINT)
    {
      for (i = x0; i < x1; i++)
        {
          skyline->virtual_skyline[i] = new_level;
          skyline->top_ads[i] = NULL;
        }

      skyline_update_minimum (skyline);

      return TRUE;
    }
  else
    {
      /* This page is now closed.  Update best page if necessary */

      if (skyline->weight > page->weight)
        {
          layout_free_all (page->page.layout);
          page->page.layout = layout_copy (skyline->layout);
          page->weight = skyline->weight;
        }
      return FALSE;
    }
}

/* Remove the ad from the page.  The page is restored by copying the
 * saved state back, after freeing the layout cell. */

void
skyline_backtrack (Skyline *skyline, Skyline *original_skyline, Ad *ad)
{
  if (ad != NULL)
    layout_free (skyline->layout);
  if (ad == first_ad)
    first_ad = NULL;
  if (ad->flags & AD_PREPLACED)
    current_preplaced_area -= ad->width * ad->height;

  *skyline = *original_skyline;
}

int
skyline_ad_fits (Skyline *skyline, SkylinePage *page, Ad *ad)
{
  int i, j, *vs;
  int area;

  if (ad == NULL)
    return TRUE;

  area = ad->width * ad->height;
  if (preplaced_area > 0)
    return skyline->area +
      (preplaced_area - current_preplaced_area) + area <= PAGE_AREA;

  if (skyline->area + area > PAGE_AREA)
    return FALSE;

  vs = skyline->virtual_skyline;
  for (i = 0; i < PAGE_WIDTH - ad->width; i++)
    {
      for (j = i; j < i + ad->width; j++)
        if (vs[j] + ad->height > PAGE_HEIGHT)
          break;
      if (j == ad->width)
        return TRUE;
    }

  return FALSE;
}

/* Before iterating through the ads we save the real and virtual
 * skylines.  Then we loop through all ads placing each one and
 * calling recursively.  Finally we try to raise the skyline, and if
 * the configuration changed, we call recursively.
 */

static Ad *big_ad;
static int big_ad_placed;

void
skyline_iterate (SkylinePage *page, GList *ads, Skyline *skyline)
```

```
{
  GList *c;
  Skyline new_skyline;
  Ad *ad, *prev_ad;

  new_skyline = *skyline;
  prev_ad = NULL;

  for (c = ads->next; c->data != NULL; c = c->next)
    {
      ad = (Ad *) c->data;

      if (!(ad->flags & AD_PREPLACED) &&
          prev_ad != NULL && prev_ad->width == ad->width &&
          prev_ad->height - ad->height < skip_factor * prev_ad->height)
        continue;

      if (!skyline_place_ad (&new_skyline, page, ad))
        continue;

      c->prev->next = c->next;
      c->next->prev = c->prev;

      if (ad == big_ad)
        big_ad_placed = TRUE;

      if (big_ad_placed || skyline_ad_fits (&new_skyline, page, big_ad))
        skyline_iterate (page, ads, &new_skyline);
      skyline_backtrack (&new_skyline, skyline, ad);

      c->prev->next = c;
      c->next->prev = c;

      if (ad == big_ad)
        big_ad_placed = FALSE;

      prev_ad = ad;
    }

  if (skyline_raise (&new_skyline, page))
    skyline_iterate (page, ads, &new_skyline);
}

static int
compare_ads (const void *p1, const void *p2)
{
  int res;
  Ad *ad1, *ad2;
  ad1 = (Ad *) p1;
  ad2 = (Ad *) p2;

  res = ad1->width - ad2->width;

  if (res == 0)
    return ad2->height - ad1->height;
  else
    return res;
}

static void
skyline_print_progress (int iteration, int current, int total,
                int length, FILE *progress)
{
  int i, scaled, p;
  char buffer[80];

  scaled = current * length / total;
  p = g_snprintf (buffer, 80, "\rIteration %3d: [", iteration);
  for (i = 0; i < length && p < 80; i++, p++)
    buffer[p] = i < scaled ? '#' : '.';
  g_snprintf (buffer + p, 80 - i, "] %d/%d", current, total);
  fprintf (progress, "%s", buffer);
}

static void
ad_ring_make (GList *list1, GList *list2)
{
  GList *t;

  list1 = g_list_concat (list1, list2);
  t = g_list_prepend (list1, NULL);
  t->prev = g_list_last (list1);
  t->prev->next = t;
}

static void
ad_ring_break (GList *ring, GList *list)
{
  GList *t;

  t = ring->prev;
  if (list == NULL)
    list = t;
  t->next->prev = NULL;
  t->next = NULL;
  list->prev->next = NULL;
  list->prev = NULL;
  g_list_free (list);
}

GList *
skyline_layout_pages (GList *ad_list, GList *preplaced_list,
                      int iteration, double sf, FILE *progress)
{
  int i, total;
  GList *new_pages, *ads, *c, *tmp;
  SkylinePage *page1, *page2;

  new_pages = NULL;
  ads = g_list_sort (g_list_copy (ad_list), compare_ads);

  for (c = ads, i = 0; c != NULL; c = c->next, i++)
    ((Ad*)c->data)->key = i;

  total = g_list_length (ads);
  skip_factor = sf;

  for (i = 0; ads != NULL; i++)
    {
      Skyline skyline;

      big_ad = ad_list_get_biggest_ad (ads);

      page1 = skyline_page_new (i);
      ad_ring_make (ads, NULL);
      skyline_init (&skyline);
      skyline_iterate (page1, ads->prev, &skyline);
      ad_ring_break (ads, NULL);

      if (preplaced_list != NULL)
        {
          Page *preplaced_page = preplaced_list->data;
          GList *preplaced_ads;

          page2 = skyline_page_new (i);
          skyline_init (&skyline);

          preplaced_ads_tabulate (preplaced_page->layout);
          preplaced_ads = page_get_ads (preplaced_page);
          ad_ring_make (ads, preplaced_ads);
          skyline_iterate (page2, ads->prev, &skyline);
          preplaced_ads_free ();
          ad_ring_break (ads, preplaced_ads);
        }
      else
        page2 = NULL;
      if (page2 == NULL || page1->weight > page2->weight)
        {
          tmp = page_get_ads ((Page *) page1);
          ads = ad_list_subtract (ads, tmp);
          g_list_free (tmp);
          new_pages = g_list_prepend (new_pages, page1);
          if (page2 != NULL)
            page_destroy ((Page *) page2);
        }
      else
        {
          tmp = page_get_ads ((Page *) page2);
          ads = ad_list_subtract (ads, tmp);
          g_list_free (tmp);
          new_pages = g_list_prepend (new_pages, page2);
          page_destroy ((Page *) page1);
          preplaced_list = preplaced_list->next;
        }

      if (progress != NULL)
        skyline_print_progress (iteration, total - (g_list_length (ads)),
                                total, 40, progress);
    }

  new_pages = g_list_concat (g_list_copy (preplaced_list), new_pages);

  if (progress)
    fprintf (progress, ", %d pages\n", g_list_length (new_pages));

  return new_pages;
}
```

skyline-main.c

```
#include <glib.h>
#include <stdlib.h>
#include <math.h>
#include <values.h>
#include <gnome-xml/parser.h>
#include <string.h>

#include "skyline.h"
#include "lower-bound.h"
#include "xml.h"

void
adjust_weights (GList *pages)
```

```
{
  GList *c;
  int count[8], i;
  double badness[8], fraction;
  Layout *l;

  for (i = 0; i < 8; i++)
    {
      count[i] = 0;
      badness[i] = 0;
    }

  /* We skip the last page, otherwise the ads there would be unfairly
   * readjusted.  */

  for (c = pages->next; c != NULL; c = c->next)
    {
      Page *page = c->data;

      fraction = page_compute_sparseness (page);

      for (l = page->layout; l != NULL; l = l->next)
        {
          badness[l->ad->width - 1] += fraction;
          count[l->ad->width - 1]++;
        }
    }

  for (c = pages; c != NULL; c = c->next)
    {
      Page *page = c->data;

      for (l = page->layout; l != NULL; l = l->next)
        {
          int width = l->ad->width - 1;

          /* If there only was one ad of this width and it occurred on
           * the last page, count[width] will be 0.  If this happens,
           * we just ignore the ad. */

          if (count[width] == 0)
            continue;

          if (l->ad->flags & AD_PREPLACED)
            l->ad->weight *= page_compute_sparseness (page);
          else
            l->ad->weight = l->ad->weight * badness[width] / count[width];
        }
    }
```

```
}

void
calculate_preplace_constraints (GList *pages)
{
  GList *c;
  Layout *l, *m;
  int i;

  for (c = pages; c != NULL; c = c->next)
    {
      Page *p = c->data;

      for (l = p->layout; l != NULL; l = l->next)
        {
          for (i = 0; i < PAGE_WIDTH; i++)
            l->ad->min_level[i] = 0;
          l->ad->preplaced_x = l->x;
          l->ad->preplaced_y = l->y;
          l->ad->flags |= AD_PREPLACED;

          for (m = p->layout; m != NULL; m = m->next)
            {
              if (l == m)
                continue;

              if ((m->x + m->ad->width < l->x &&
                   m->y < l->y + l->ad->height) ||
                  (m->y + m->ad->height < l->y &&
                   m->x < l->x + l->ad->width))
                for (i = 0; i < m->x + m->ad->width; i++)
                  l->ad->min_level[i] = MAX (l->ad->min_level[i],
                                             m->y + m->ad->height);

            }
        }
    }
}

int
main (int argc, char *argv[])
{
  GList *ad_list, *page_list, *new_pages, *preplaced_ads;
  GString *entry;
  xmlDoc *doc;
  int i, iterations;
  int lb_dual_feasible, lb_continuous, lb;
  double skip_factor, step;
```

```
if (argc < 2) {
  fprintf (stderr, "usage: %s [-quiet] <adfile> "
           "[skip-factor [iterations]]\n", argv[0]);
  return 1;
}

if (argc >= 3)
  skip_factor = atof (argv[2]);
else
  skip_factor = 0;

if (skip_factor < 0.0001)
  skip_factor = 0.0001;

if (argc >= 4)
  iterations = atoi (argv[3]);
else
  iterations = 4;

if (argc >= 5)
  step = atof (argv[4]);
else
  step = 0;

doc = xmlParseFile (argv[1]);
if (doc == NULL)
  {
    fprintf (stderr, "no such file: ‘%s’\n", argv[1]);
    return 1;
  }

ad_list = xml_get_ads (doc);
page_list = xml_get_pages (doc, ad_list, "fixed");

lb_continuous = lower_bound_continuous (ad_list);
lb_dual_feasible = lower_bound_dual_feasible (ad_list);
lb = MAX (lb_continuous, lb_dual_feasible);

fprintf (stderr, "Skip factor: %.02f\n", skip_factor);
fprintf (stderr, "Dual feasible lower bound: %d pages\n", lb_dual_feasible);
fprintf (stderr, "Continuous lower bound: %d pages\n", lb_continuous);

preplaced_ads = page_list_get_ads (page_list);
calculate_preplace_constraints (page_list);
ad_list = ad_list_subtract (ad_list, preplaced_ads);
g_list_free (preplaced_ads);

new_pages = NULL;
for (i = 0; i < iterations; i++)
```

```
    {
      double current_sf;

      current_sf = skip_factor + (iterations - i - 1) * step;
      if (step > 0)
        fprintf (stderr, "Skip factor: %f\n", current_sf);

      new_pages = skyline_layout_pages (ad_list, page_list,
                                        i, current_sf, stderr);
      adjust_weights (new_pages);
      if (g_list_length (new_pages) == lb)
        break;
    }

  new_pages = g_list_reverse (new_pages);
  page_list_enumerate (new_pages);
  xml_update_pages (new_pages);

  entry = g_string_new (NULL);
  g_string_sprintfa (entry, "\n      Number of pages: %d\n",
                     g_list_length (new_pages));
  g_string_sprintfa (entry, "      Lower bound: %d\n", lb);
  if (step == 0)
    g_string_sprintfa (entry, "      Skip factor: %.02f\n    ",
                       skip_factor);
  else
    {
      g_string_sprintfa (entry, "      Skip factor: %.02f\n",
                         skip_factor);
      g_string_sprintfa (entry, "      Skip factor step: %.02f\n    ",
                         step);
    }
  xml_add_history_entry (doc, "skyline", "0.1", entry->str);
  g_string_free (entry, TRUE);

  xmlDocDump (stdout, doc);

  return 0;
}
```

C.4 Tabu Search Implementation

The `tabu-search.h` and `tabu-search.c` files implement the tabu search algorithm. The exploration of the first neighborhood is implemented by the functions `empty_weakest_page` and `tabu_search_neighborhood1`, while the function `tabu_search_neighborhood2` implements the exploration of the second neighborhood. The restart move is implemented by `tabu_search_restart`.

Argument parsing and other details are handled in `tabu-search-main.c`, where we also handle the stopping criteria by setting up a timer.

tabu-search.h

```
#ifndef TABU_SEARCH_H
#define TABU_SEARCH_H

#include "basic-types.h"
#include "xml.h"

#define AD_TENURE 3
#define SCORE_TENURE 5
#define RESTART_PERIOD 30

typedef struct TabuList TabuList;
struct TabuList
{
  Ad *ads[AD_TENURE];
  int ad_head, ad_tail, ad_length;

  double scores[SCORE_TENURE];
  int score_head, score_tail, score_length;
};

GList *tabu_search_layout_pages (GList *ads, int goal, FILE *progess);
void tabu_search_stop (void);

#endif TABU_SEARCH_H
```

tabu-search.c

```
#include <stdlib.h>
#include <math.h>
#include <glib.h>
#include <values.h>

#include "tabu-search.h"
#include "strip.h"
```

```
#include "ih.h"

static double alpha = 5.0;
static int number_of_ads = 0;
static int iteration;

static double
compute_phi (Page *page)
{
  Layout *l;
  int area, count;

  area = 0;
  count = 0;
  for (l = page->layout; l != NULL; l = l->next)
    {
      area += l->ad->width * l->ad->height;
      count++;
    }

  return alpha * (double) area / PAGE_AREA - (double) count / number_of_ads;
}

static Page *
find_weakest_page (GList *page_list)
{
  GList *c;
  double phi, min_phi;
  Page *weakest_page;

  weakest_page = NULL;
  min_phi = 0;

  for (c = page_list; c != NULL; c = c->next)
    {
      Page *page = c->data;

      phi = compute_phi (page);
      if (weakest_page == NULL || phi < min_phi)
        {
          weakest_page = page;
          min_phi = phi;
        }
    }

  return weakest_page;
}

/* Combine the ads from page1 and page2 together with the ad extra
```

```
 * into a GList.  Any of page1 and page2 can be NULL, in which case
 * they are just ignored.
 */

static GList *
combine_pages_with_ad (Page *page1, Page *page2, Ad *extra)
{
  Layout *l;
  GList *list;
  GList *new_pages;

  list = g_list_prepend (NULL, extra);

  if (page1 != NULL)
    for (l = page1->layout; l != NULL; l = l->next)
      list = g_list_prepend (list, l->ad);

  if (page2 != NULL)
    for (l = page2->layout; l != NULL; l = l->next)
      list = g_list_prepend (list, l->ad);

  new_pages = fbs_layout_pages (list);
  g_list_free (list);

  return new_pages;
}

/* Combine the ads from page1 and page2 into a GList, but leave out
 * the ad exclude, if it is contained in page2.  Any of page1 and
 * page2 can be NULL, in which case they are just ignored.
 */

static GList *
combine_pages_without_ad (Page *page1, Page *page2, Ad *exclude)
{
  Layout *l;
  GList *list;
  GList *new_pages;

  list = NULL;
  if (page1 != NULL)
    for (l = page1->layout; l != NULL; l = l->next)
      list = g_list_prepend (list, l->ad);

  if (page2 != NULL)
    for (l = page2->layout; l != NULL; l = l->next)
      {
        if (l->ad != exclude)
          list = g_list_prepend (list, l->ad);
      }

  new_pages = fbs_layout_pages (list);
  g_list_free (list);

  return new_pages;
}

void
tabu_list_init (TabuList *tl)
{
  tl->score_head = 0;
  tl->score_tail = 0;
  tl->score_length = 0;

  tl->ad_head = 0;
  tl->ad_tail = 0;
  tl->ad_length = 0;
}

void
tabu_list_add_ad (TabuList *tl, Ad *ad)
{
  tl->ads[tl->ad_tail] = ad;
  tl->ad_tail = (tl->ad_tail + 1) % AD_TENURE;

  if (tl->ad_length == AD_TENURE)
    tl->ad_head = (tl->ad_head + 1) % AD_TENURE;
  else
    tl->ad_length++;
}

int
tabu_list_ad_is_tabu (TabuList *tl, Ad *ad)
{
  int i, j;

  for (i = tl->ad_head, j = 0; j < tl->ad_length; i++, j++)
    {
      if (tl->ads[i % SCORE_TENURE] == ad)
        return TRUE;
    }

  return FALSE;
}

void
tabu_list_add_score (TabuList *tl, double score)
{
  tl->scores[tl->score_tail] = score;
  tl->score_tail = (tl->score_tail + 1) % SCORE_TENURE;

  if (tl->score_length == SCORE_TENURE)
    tl->score_head = (tl->score_head + 1) % SCORE_TENURE;
  else
    tl->score_length++;
}

int
tabu_list_score_is_tabu (TabuList *tl, double score)
{
  int i, j;

  for (i = tl->score_head, j = 0; j < tl->score_length; i++, j++)
    {
      if (fabs (tl->scores[i % SCORE_TENURE] < 1e-6))
        return TRUE;
    }

  return FALSE;
}

static void
empty_weakest_page (GList *pages, Page *wp, TabuList *tl)
{
  GList *c, *new_pages;
  Layout *l, *remaining;
  int length, single_ad_on_page;

  single_ad_on_page = (wp->layout != NULL && wp->layout->next == NULL);

  remaining = NULL;
  for (l = wp->layout; l != NULL; l = l->next)
    {
      if (tabu_list_ad_is_tabu (tl, l->ad) && !single_ad_on_page)
        {
          remaining = layout_new (l->ad, l->x, l->y, remaining);
          continue;
        }

      for (c = pages; c != NULL; c = c->next)
        {
          Page *p = c->data;

          if (p == wp)
            continue;

          new_pages = combine_pages_with_ad (p, NULL, l->ad);
          length = g_list_length (new_pages);

          if (length == 1)
            {
              page_destroy (p);
              c->data = new_pages->data;
              g_list_free (new_pages);
              tabu_list_add_ad (tl, l->ad);
              iteration++;
              break;
            }
          else
            page_list_free (new_pages);
        }

      if (c == NULL)
        remaining = layout_new (l->ad, l->x, l->y, remaining);
    }

  layout_free_all (wp->layout);
  wp->layout = remaining;
}

GList *
tabu_search_neighborhood1 (GList *pages, TabuList *tl)
{
  Page *wp;

  while (1)
    {
      wp = find_weakest_page (pages);
      empty_weakest_page (pages, wp, tl);
      if (wp->layout == NULL)
        pages = page_list_remove (pages, wp);
      else
        break;
    }

  return pages;
}

typedef struct Move Move;
struct Move
{
  double score;
  Page *h;
  Page *k;
  GList *pages;
  Ad *ad;
```

```
};

GList *
tabu_search_neighborhood2 (GList *pages, TabuList *tl)
{
  Page *wp, *new_wp;
  GList *new_pages, *c, *d;
  int length;
  Move best_move;
  Layout *l, *next;

  wp = find_weakest_page (pages);
  best_move.pages = NULL;

  for (l = wp->layout; l != NULL; l = next)
    {
      next = l->next;

      for (c = pages; c != NULL; c = c->next)
        {
          Page *h = (Page *) c->data;

          if (h == wp)
            continue;

          for (d = pages; d != NULL; d = d->next)
            {
              Page *k = (Page *) d->data;

              if (k == wp || k == h)
                continue;

              new_pages = combine_pages_with_ad (h, k, l->ad);
              new_wp = find_weakest_page (new_pages);
              length = g_list_length (new_pages);

              if (length == 1)
                {
                  pages = page_list_remove (pages, h);
                  pages = page_list_remove (pages, k);
                  pages = g_list_concat (new_pages, pages);
                  page_remove_ad (wp, l->ad);

                  if (wp->layout == NULL)
                    pages = page_list_remove (pages, wp);
                  iteration++;

                  return pages;
                }
```

```
else if (length == 2)
  {
    pages = page_list_remove (pages, h);
    pages = page_list_remove (pages, k);
    pages = g_list_concat (new_pages, pages);
    page_remove_ad (wp, l->ad);
    iteration++;

    if (wp->layout == NULL)
      return page_list_remove (pages, wp);
    else if (compute_phi (new_wp) < compute_phi (wp))
      return pages;
    else
      {
        /* The current weakest page wasn't updated, so
         * the search continues in this neighborhood.
         * However, if we rearranged pages or ads that
         * were part of the best move, we must
         * invalidate it. */

        if (best_move.pages != NULL)
          if (best_move.ad == l->ad ||
              best_move.h == h ||
              best_move.h == k ||
              best_move.k == h ||
              best_move.k == k)
            {
              page_list_free (best_move.pages);
              best_move.pages = NULL;
            }
        goto next_ad;
      }
  }
else if (length == 3)
  {
    GList *rest = combine_pages_without_ad (new_wp, wp, l->ad);

    if (g_list_length (rest) > 1)
      {
        page_list_free (new_pages);
        page_list_free (rest);
      }
    else
      {
        double score = compute_phi (rest->data);

        if (!tabu_list_score_is_tabu (tl, score) &&
            (best_move.pages == NULL || score < best_move.score))
          {
             new_pages = page_list_remove (new_pages, new_wp);
             new_pages = g_list_concat (new_pages, rest);

             if (best_move.pages != NULL)
               page_list_free (best_move.pages);

             best_move.pages = new_pages;
             best_move.h = h;
             best_move.k = k;
             best_move.score = score;
             best_move.ad = l->ad;
            }
           else
            {
             page_list_free (new_pages);
             page_list_free (rest);
            }
          }

        }
       else
        page_list_free (new_pages);
      }
    }
  next_ad:
  }

 if (best_move.pages != NULL)
  {
   pages = page_list_remove (pages, best_move.h);
   pages = page_list_remove (pages, best_move.k);
   pages = page_list_remove (pages, wp);
   pages = g_list_concat (best_move.pages, pages);
   tabu_list_add_score (tl, best_move.score);
   iteration++;
  }

 return pages;
}

static int
compare_weakness (Page *p1, Page *p2)
{
 double diff;

 diff = compute_phi (p1) - compute_phi (p2);
 if (diff < 0)
  return -1;
 else
```

```
    return 1;
}

GList *
tabu_search_restart (GList *pages)
{
  GList *weak_pages, *ads, *new_pages;
  int length;

  pages = g_list_sort (pages, (GCompareFunc) compare_weakness);
  length = g_list_length (pages);

  weak_pages = pages;
  pages = g_list_split (pages, length / 2);
  ads = page_list_get_ads (weak_pages);
  page_list_free (weak_pages);
  new_pages = ih_layout_pages (ads);
  g_list_free (ads);

  return g_list_concat (new_pages, pages);
}

static int stop = FALSE;

void
tabu_search_stop (void)
{
  stop = TRUE;
}

static void
tabu_search_print_progress (FILE *progess, int len)
{
  static int radar_index;
  char kit_radar[7] = "#.....";

  strcpy (kit_radar, "......");
  radar_index = (radar_index + 1) % 10;
  if (radar_index < 6)
    kit_radar[radar_index] = '#';
  else
    kit_radar[10 - radar_index] = '#';

  fprintf (progess, "\r[%s] Incumbent solution: %d pages",
           kit_radar, len);
}

GList *
tabu_search_layout_pages (GList *ads, int goal, FILE *progess)
```

```
{
  TabuList tl;
  int len, prev_len, prev_change, prev_iter;
  int current_minimum;
  GList *best_solution, *pages;

  tabu_list_init (&tl);
  current_minimum = MAXINT;
  len = MAXINT;
  iteration = 0;

  number_of_ads = g_list_length (ads);
  prev_len = 0;
  prev_change = 0;
  stop = FALSE;
  pages = ih_layout_pages (ads);
  best_solution = NULL;

  while (!stop && (goal > 0 && len > goal))
    {
      prev_iter = iteration;
      pages = tabu_search_neighborhood1 (pages, &tl);
      pages = tabu_search_neighborhood2 (pages, &tl);
      len = g_list_length (pages);

      if (len < current_minimum)
        {
          tabu_search_print_progress (stderr, len);
          current_minimum = len;
          if (best_solution != NULL)
            page_list_free (best_solution);
          best_solution = page_list_copy (pages);
        }

      if (prev_iter == iteration ||
          (prev_len == len && prev_change + RESTART_PERIOD <= iteration))
        pages = tabu_search_restart (pages);

      if (prev_len != len)
        {
          prev_len = len;
          prev_change = iteration;
        }

      if ((iteration & 63) == 0)
        tabu_search_print_progress (stderr, current_minimum);
    }

  fprintf (stderr, "\n");
  page_list_free (pages);

  return best_solution;
}
```

tabu-search-main.c

```
#include <glib.h>
#include <stdlib.h>
#include <math.h>
#include <values.h>
#include <gnome-xml/parser.h>
#include <string.h>
#include <signal.h>
#include <sys/time.h>

#include "ih.h"
#include "lower-bound.h"
#include "tabu-search.h"

void
sigint_handler (int signum)
{
  fprintf (stderr, "\nSIGINT caught, finishing current iteration.\n");
  signal (SIGINT, SIG_DFL);
  tabu_search_stop ();
}

void
sigprof_handler (int signum)
{
  fprintf (stderr, "\nSIGPROF caught, finishing current iteration.\n");
  signal (SIGPROF, SIG_DFL);
  tabu_search_stop ();
}

static void
initialize_signals (int timeout)
{
  struct itimerval new;

  if (timeout > 0)
    {
      new.it_interval.tv_usec = 0;
      new.it_interval.tv_sec = 0;
      new.it_value.tv_usec = 0;
      new.it_value.tv_sec = timeout;
      setitimer (ITIMER_PROF, &new, NULL);
      signal (SIGPROF, sigprof_handler);
    }
  signal (SIGINT, sigint_handler);
}

static void
reset_signals (void)
{
  signal (SIGPROF, SIG_DFL);
  signal (SIGINT, SIG_DFL);
}

int
main (int argc, char *argv[])
{
  GList *ads, *pages;
  xmlDoc *doc;
  char *filename;
  GString *entry;
  int goal, timeout, lb_continuous, lb_dual_feasible;

  if (argc < 2)
    {
      fprintf (stderr, "usage: %s AD-FILE [TIME-BOUND [GOAL]]\n", argv[0]);
      return -1;
    }

  filename = argv[1];

  if (argc > 2)
    timeout = atoi (argv[2]);
  else
    timeout = 0;

  if (argc > 3)
    goal = atoi (argv[3]);
  else
    goal = 0;

  doc = xmlParseFile (filename);
  ads = xml_get_ads (doc);

  lb_continuous = lower_bound_continuous (ads);
  lb_dual_feasible = lower_bound_dual_feasible (ads);

  if (goal == 0)
    goal = MAX (lb_continuous, lb_dual_feasible);
  fprintf (stderr, "Dual feasible lower bound: %d pages\n", lb_dual_feasible);
  fprintf (stderr, "Continuous lower bound: %d pages\n", lb_continuous);
  fprintf (stderr, "Goal: %d\n\n", goal);
  if (timeout > 0)
    fprintf (stderr, "Time bound: %d seconds\n", timeout);

  initialize_signals (timeout);
  pages = tabu_search_layout_pages (ads, goal, stderr);
  reset_signals ();

  page_list_enumerate (pages);
  xml_update_pages (pages);

  entry = g_string_new (NULL);
  g_string_sprintfa (entry, "\n      Number of pages: %d\n    ",
                     g_list_length (pages));
  xml_add_history_entry (doc, "tabu-search", "0.1", entry->str);
  g_string_free (entry, TRUE);

  xmlDocDump (stdout, doc);

  return 0;
}
```

C.5 HFF, FBS and IH implementations

The HFF and FBS heuristics are implemented in the `strip.h` and `strip.c` files. A lot of code is shared between the two algorithms; only the search for a vacant position differs between the two implementations.

The IH algorithm described in Section 4.2 is implemented in the files `ih.h` and `ih.c`.

strip.h

```
#ifndef STRIP_H
#define STRIP_H

#include "basic-types.h"

GList *hff_layout_pages (GList *ads);
GList *fbs_layout_pages (GList *ads);

#endif /* HFF_F */
```

strip.c

```
#include <stdlib.h>
#include <stdio.h>
#include "basic-types.h"

typedef struct Strip Strip;
struct Strip {
  int height, width;
  GList *ads;
};

typedef struct StripPage StripPage;
struct StripPage {
  Page page;
  int size;
  GList *strips;
};

Strip *
strip_new (Ad *ad)
{
  Strip *strip;

  strip = g_new (Strip, 1);
  strip->height = ad->height;
  strip->width = ad->width;
  strip->ads = g_list_prepend (NULL, ad);

  return strip;
}

void
strip_free (Strip *strip)
{
  g_list_free (strip->ads);
  g_free (strip);
}

void
strip_layout (Strip *strip, int level, StripPage *page)
{
  GList *c;
  int x;

  x = 0;

  for (c = strip->ads; c != NULL; c = c->next)
    {
      Ad *ad = c->data;

      page->page.layout = layout_new (ad, x, level, page->page.layout);
      x += ad->width;
    }
}

GList *
strip_list_add_first_fit (GList *list, Ad *ad)
{
  GList *c;
  Strip *strip;

  for (c = list; c != NULL; c = c->next)
    {
      strip = (Strip *) c->data;
      if  (strip->width + ad->width <= PAGE_WIDTH)
        {
          strip->ads = g_list_prepend (strip->ads, ad);
          strip->width += ad->width;
          return list;
        }
    }

  return g_list_append (list, strip_new (ad));
}
```

```
GList *
strip_list_add_best_fit (GList *list, Ad *ad)
{
  GList *c;
  Strip *strip, *best;

  best = NULL;
  for (c = list; c != NULL; c = c->next)
    {
      strip = (Strip *) c->data;
      if (strip->width + ad->width <= PAGE_WIDTH &&
           (best == NULL || strip->width > best->width))
        best = strip;
    }

  if (best == NULL)
    return g_list_append (list, strip_new (ad));
  else
    {
      best->ads = g_list_prepend (best->ads, ad);
      best->width += ad->width;
      return list;
    }
}

static void strip_page_destroy (StripPage *page);

StripPage *
strip_page_new (Strip *strip)
{
  StripPage *page;
  static int number = 1;

  page = g_new (StripPage, 1);
  page->page.number = number++;
  page->page.layout = NULL;
  page->page.id = NULL;
  page->size = strip->height;
  page->strips = g_list_prepend (NULL, strip);

  page->page.destroy = (void (*) (Page*)) strip_page_destroy;
  page->page.copy = (Page *(*) (Page*)) base_page_copy;

  return page;
}

static void
strip_page_destroy (StripPage *page)
```

```
{
  layout_free_all (page->page.layout);
  g_free (page->page.id);
  g_list_foreach (page->strips, (GFunc) strip_free, NULL);
  g_list_free (page->strips);
  g_free (page);
}

static GList *
strip_page_list_add_first_fit (GList *list, Strip *strip)
{
  StripPage *page;
  GList *c;

  for (c = list; c != NULL; c = c->next)
    {
      page = (StripPage *) c->data;
      if (page->size + strip->height <= PAGE_HEIGHT)
        {
          page->size += strip->height;
          page->strips = g_list_prepend (page->strips, strip);
          return list;
        }
    }

  return g_list_append (list, strip_page_new (strip));
}

static GList *
strip_page_list_add_best_fit (GList *list, Strip *strip)
{
  StripPage *page, *best;
  GList *c;

  best = NULL;
  for (c = list; c != NULL; c = c->next)
    {
      page = (StripPage *) c->data;
      if (page->size + strip->height <= PAGE_HEIGHT &&
          (best == NULL || page->size > best->size))
        best = page;
    }

  if (best == NULL)
    return g_list_append (list, strip_page_new (strip));
  else {
    best->size += strip->height;
    best->strips = g_list_prepend (best->strips, strip);
    return list;
  }
}

static int
compare_ads (Ad *ad1, Ad *ad2)
{
  if (ad2->height == ad1->height)
    return ad2->width - ad1->width;
  else
    return ad2->height - ad1->height;
}

static GList *
hff_generate_pages (GList *ads)
{
  GList *c, *pages, *strips;

  ads = g_list_sort (g_list_copy (ads), (GCompareFunc) compare_ads);

  strips = NULL;
  for (c = ads; c != NULL; c = c->next)
    strips = strip_list_add_first_fit (strips, c->data);

  pages = NULL;
  for (c = strips; c != NULL; c = c->next)
    pages = strip_page_list_add_first_fit (pages, c->data);

  g_list_free (strips);
  g_list_free (ads);

  return pages;
}

GList *
hff_layout_pages (GList *ads)
{
  GList *pages, *c, *d;
  Strip *strip;
  int level;

  pages = hff_generate_pages (ads);
  for (c = pages; c != NULL; c = c->next)
    {
      StripPage *page = c->data;

      level = 0;
      for (d = page->strips; d != NULL; d = d->next)
        {
          strip = d->data;
          strip_layout (strip, level, page);
          level += strip->height;
          strip_free (strip);
        }
      g_list_free (page->strips);
      page->strips = NULL;
    }

  return pages;
}

static GList *
fbs_generate_pages (GList *ads)
{
  GList *c, *pages, *strips;

  ads = g_list_sort (g_list_copy (ads), (GCompareFunc) compare_ads);

  strips = NULL;
  for (c = ads; c != NULL; c = c->next)
    strips = strip_list_add_best_fit (strips, c->data);

  pages = NULL;
  for (c = strips; c != NULL; c = c->next)
    pages = strip_page_list_add_best_fit (pages, c->data);

  g_list_free (strips);
  g_list_free (ads);

  return pages;
}

GList *
fbs_layout_pages (GList *ads)
{
  GList *pages, *c, *d;
  Strip *strip;
  int level;

  pages = fbs_generate_pages (ads);
  for (c = pages; c != NULL; c = c->next)
    {
      StripPage *page = c->data;

      level = 0;
      for (d = page->strips; d != NULL; d = d->next)
        {
          strip = d->data;
          strip_layout (strip, level, page);
          level += strip->height;
          strip_free (strip);
        }
      g_list_free (page->strips);
      page->strips = NULL;
    }

  return pages;
}
```

ih.h

```
#ifndef IH_H
#define IH_H

#include "basic-types.h"

GList *ih_layout_pages (GList *ads);

#endif /* IH_F */
```

ih.c

```
#include <stdlib.h>
#include <math.h>
#include <glib.h>
#include <values.h>

#include "basic-types.h"

/* OK, this thing got a bit more complicated than it could have been.
 * The y coordinates are relative to the class the slot or ad belongs
 * to.  That is, to get the actual y coordinate on the page you do:
 *
 *   real_y = y * PAGE_HEIGHT / (1 << class)
 *
 * Thus, if you just want to compare two y coordinates from two
 * different classes you can do:
 *
 *   y1 << class2 < y2 << class1 */

#define SLOT_VPOS(s)     ((s)->y * PAGE_HEIGHT / (1 << (s)->class))

typedef struct Slot Slot;
struct Slot {
  int x, y;
  int class;
  Ad *ad;
};

Slot *
slot_new (int x, int y, int class)
{
  Slot *slot;

  slot = g_new (Slot, 1);
  slot->x = x;
  slot->y = y;
  slot->class = class;
  slot->ad = NULL;

  return slot;
}

void
slot_list_insert (GList *list, Slot *slot)
{
  GList *c;

  for (c = list; c != NULL; c = c->next)
    {
      Slot *s = c->data;

      if (slot->x < s->x ||
          (slot->x == s->x &&
           slot->y << s->class < s->y << slot->class))
        {
          g_list_prepend (c, slot);
          break;
        }
    }

  if (c == NULL)
    g_list_append (list, slot);
}

/* The algorithm is implemented by maintaining a list of vacant slots
 * in the strip, sorted by x and then y coordinate of the lower left
 * corner.  The ads are sorted non-decreasingly by class, so its just
 * a matter of fitting the first ad in the first slot.  Now the list
 * of slots must be updated: first a new slot with the same class as
 * the ad is allocated to the right of the ad.  Second, if the ad
 * belonged to a greater class than the slot (ie. it's less than half
 * the height of the slot) the slot is split into n - 1 new slots,
 * where n is 1 << (ad-class - slot-class).
 */
```

```
static GList *
fill_strip (GList *ads)
{
  GList *slots, *c, *si;
  Slot *s, *new_slot;
  int diff, i;

  slots = g_list_prepend (NULL, slot_new (0, 0, 0));

  for (c = ads, si = slots; c != NULL; c = c->next)
    {
      Ad *ad = c->data;

      s = si->data;
      diff = ad->class - s->class;
      s->ad = ad;
      s->class = ad->class;
      s->y = s->y << diff;

      new_slot = slot_new (s->x + ad->width, s->y, ad->class);
      slot_list_insert (si, new_slot);

      for (i = (1 << diff) - 1; i > 0; i--)
        {
          /* We could use slot_list_insert () here, but since the new
           * slots preceede all other slots we just use
           * g_list_prepend ().
           */

          new_slot = slot_new (s->x, s->y + i, s->class);
          si->next = g_list_prepend (si->next, new_slot);
        }
      si = si->next;
    }

  /* Free the remaining unused slots. */

  si->prev->next = NULL;
  g_list_foreach (si, (GFunc) g_free, NULL);
  g_list_free (si);

  return slots;
}

static GList *
get_pages_from_slots (GList *slots)
{
  GList *si, *pages;
  Slot *s;
  Page *p0, *p1;
  int x, n;

  x = 0;
  n = 1;
  p0 = base_page_new (n++, NULL, NULL);
  p1 = base_page_new (n++, NULL, NULL);
  pages = NULL;

  for (si = slots; si != NULL; si = si->next)
    {
      s = (Slot *) si->data;

      if (x + PAGE_WIDTH <= s->x)
        {
          x += PAGE_WIDTH;
          if (p0->layout != NULL)
            {
              pages = g_list_prepend (pages, p0);
              p0 = base_page_new (n++, NULL, NULL);
            }
          if (p1->layout != NULL)
            {
              pages = g_list_prepend (pages, p1);
              p1 = base_page_new (n++, NULL, NULL);
            }
        }

      if (s->x + s->ad->width <= x + PAGE_WIDTH)
        p0->layout = layout_new (s->ad, s->x - x, SLOT_VPOS (s), p0->layout);
      else
        p1->layout = layout_new (s->ad, 0, SLOT_VPOS (s), p1->layout);
    }

  if (p0->layout == NULL)
    page_destroy (p0);
  else
    pages = g_list_prepend (pages, p0);

  if (p1->layout == NULL)
    page_destroy (p1);
  else
    pages = g_list_prepend (pages, p1);

  return g_list_reverse (pages);
}

static int
compare_classes (Ad *ad1, Ad *ad2)
```

```
{
  return ad1->class - ad2->class;
}

GList *
ih_layout_pages (GList *ads)
{
  GList *c, *slots, *pages, *list;

  for (c = ads; c != NULL; c = c->next)
    {
      Ad *ad = c->data;
      double frac;

      frac = (double) PAGE_HEIGHT / ad->height;
      ad->class = floor (log (frac) / M_LN2);
      g_assert (ad->node != NULL);
    }

  list = g_list_copy (ads);
  list = g_list_sort (list, (GCompareFunc) compare_classes);

  slots = fill_strip (list);
  pages = get_pages_from_slots (slots);

  g_list_free (list);
  g_list_foreach (slots, (GFunc) g_free, NULL);
  g_list_free (slots);

  return pages;
}
```

C.6 Miscellaneous

The `validate.c` file implements the validate tool. Most of the work is done by the function `layout_validate` from `basic-types.c`.

The `lower-bounds.h` and `lower-bounds.c` implement the dual feasible functions, the continuous lower bound and the L_{2d} lower bound described in Section 2.4.

validate.c

```
#include <stdio.h>
#include "basic-types.h"
#include "xml.h"

int
compare_pages (Page *p1, Page *p2)
{
  return p1->number - p2->number;
}

int
compage_ads (Ad *ad1, Ad *ad2)
{
  return strcmp (ad1->id, ad2->id);
}

int
validate_geometry (GList *pages)
{
  GList *c, *d;
  int return_code;

  return_code = TRUE;
  fprintf (stderr, "** validating geometry\n");
  for (c = pages; c != NULL; c = c->next)
    {
      Page *p = (Page *) c->data;
      GList *errors;

      errors = layout_validate (p->layout);
      if (errors != NULL)
        {
          fprintf (stderr, "page %s: ", p->id);
          for (d = errors; d != NULL; d = d->next)
            {
              Ad *a = (Ad *) d->data;
              fprintf (stderr, "%s%s",
                       a->id, d->next != NULL ? ", " : "\n");
            }
          g_list_free (errors);
          return_code = FALSE;
        }
    }

  return return_code;
}

int
validate_against_original (GList *ads, char *filename)
{
  GList *missing, *extra, *c, *d, *orig_ads;
  xmlDoc *orig;
  int return_code;

  fprintf (stderr, "** validating against ‘%s’\n", filename);

  orig = xmlParseFile (filename);
  orig_ads = xml_get_ads (orig);
  orig_ads = g_list_sort (orig_ads, (GCompareFunc) compage_ads);

  missing = NULL;
  extra = NULL;
  c = orig_ads;
  d = ads;

  while (c != NULL && d != NULL)
    {
      int res;

      res = strcmp (((Ad *)c->data)->id, ((Ad *)d->data)->id);
      if (res < 0)
        {
          missing = g_list_prepend (missing, c->data);
          c = c->next;
        }
      else if (res > 0)
        {
          extra = g_list_prepend (extra, d->data);
          d = d->next;
        }
      else
        {
          c = c->next;
          d = d->next;
        }
    }

  missing = g_list_concat (c, missing);
  extra = g_list_concat (d, extra);

  if (missing != NULL)
    {
      int len;

      len = g_list_length (missing);
      fprintf (stderr, "new layout miss %d ad%s: ",
               len, len > 1 ? "s" : "");
      for (c = missing; c != NULL; c = c->next)
        {
          Ad *a = (Ad *) c->data;

          fprintf (stderr, "%s%s", a->id, c->next != NULL ? ", " : "\n");
        }
    }

  if (extra != NULL)
    {
      int len;

      len = g_list_length (extra);
      fprintf (stderr, "new layout has %d extra ad%s: ",
               len, len > 1 ? "s" : "");
      for (c = extra; c != NULL; c = c->next)
        {
          Ad *a = (Ad *) c->data;

          fprintf (stderr, "%s%s", a->id, c->next != NULL ? ", " : "\n");
        }
    }

  if (missing != NULL || extra != NULL)
    return_code = 1;
  else
    return_code = 0;

  g_list_free (missing);
  g_list_free (extra);

  return return_code;
}


int
main (int argc, char *argv[])
```

```
{
  GList *ads, *pages;
  xmlDoc *doc;
  char *filename;
  int return_code;

  if (argv[1] == NULL)
    {
      fprintf (stderr, "usage: %s AD-FILE [ORIGINAL]\n", argv[0]);
      return -1;
    }
  else
    filename = argv[1];

  return_code = 0;

  doc = xmlParseFile (filename);
  if (doc == NULL)
    {
      fprintf (stderr, "file ‘%s’ not found\n", filename);
      exit (-1);
    }

  ads = xml_get_ads (doc);
  ads = g_list_sort (ads, (GCompareFunc) compage_ads);

  pages = xml_get_pages (doc, ads, NULL);
  pages = g_list_sort (pages, (GCompareFunc) compare_pages);

  if (!validate_geometry (pages))
    return_code = 1;

  if (argv[2] != NULL &&
      validate_against_original (ads, argv[2]) == 1)
    return_code = 1;

  return return_code;
}
```

lower-bound.h

```
#ifndef LOWER_BOUND_H
#define LOWER_BOUND_H

int lower_bound_continuous (GList *ads);
int lower_bound_dual_feasible (GList *ads);

#endif /* LOWER_BOUND_H */
```

lower-bound.c

```
#include <math.h>
#include "basic-types.h"

/* Dual feasible functions.  */

/* The granularity of x is 1/520 ~ 2e-3, so we can use 1e-6 as a fudge
 * factor to eliminate numerical instability, without disturbing the
 * result of the comparison.
 * It doesn’t make sense to use ’=’ or ’<=’, but we can say
 *
 *   ’x < y - FUDGE’ when we mean ’x < y’ and
 *   ’x < y + FUDGE’ when we mean ’x <= y’
 */

#define FUDGE 1e-6

#define AD(p) ((Ad *) ((p)->data))
#define AD_WIDTH(p) (AD (p)->width)
#define AD_HEIGHT(p) (AD (p)->height)
#define AD_WIDTH_TO_UNIT(w) ((double) w / 8)
#define AD_HEIGHT_TO_UNIT(h) ((double) h / PAGE_HEIGHT)
#define AD_UNIT_WIDTH(a) (AD_WIDTH_TO_UNIT (AD_WIDTH (a)))
#define AD_UNIT_HEIGHT(a) (AD_HEIGHT_TO_UNIT (AD_HEIGHT (a)))

static double
dual_feasible_id (double x, double dummy)
{
  return x;
}

static double
dual_feasible_u (double x, int k)
{
  if (fabs (floor (x * (k + 1)) - x * (k + 1)) < FUDGE)
    return x;
  else
    return floor ((k + 1) * x) / k;
}

static double
dual_feasible_u1 (double x, double dummy)
{
  if (x < 0.5 - FUDGE)
    return 0;
  else if (x < 0.5 + FUDGE)
    return x;
  else
```

```
    return 1;
}

static double
dual_feasible_U (double x, double epsilon)
{
  if (x < epsilon - FUDGE)
    return 0;
  else if (x < 1 - epsilon + FUDGE)
    return x;
  else
    return 1;
}

static double
dual_feasible_phi (double x, double epsilon)
{
  if (epsilon == 0)
    {
      if (x < 0.5 + FUDGE)
        return 0;
      else
        return  x;
    }
  else
    {
      if (x < epsilon - FUDGE)
        return 0;
      else if (x < 0.5 + FUDGE)
        return 1 / floor (1 / epsilon);
      else
        return 1 - floor ((1 - x) / epsilon) / floor (1 / epsilon);
    }
}

int
lower_bound_continuous (GList *ads)
{
  GList *c;
  int area;

  area = 0;
  for (c = ads; c != NULL; c = c->next)
    {
      Ad *ad = c->data;

      area += ad->width * ad->height;
    }
  return (area + PAGE_AREA - 1) / PAGE_AREA;
}

static int
compare_width_dist_from_middle (Ad *ad1, Ad *ad2)
{
  int d1, d2;

  if (ad1->width < 4)
    d1 = ad1->width;
  else
    d1 = 8 - ad1->width;

  if (ad2->width < 4)
    d2 = ad2->width;
  else
    d2 = 8 - ad2->width;

  return d1 - d2;
}

static int
compare_height_dist_from_middle (Ad *ad1, Ad *ad2)
{
  int d1, d2;

  if (ad1->height < (PAGE_HEIGHT / 2))
    d1 = ad1->height;
  else
    d1 = PAGE_HEIGHT - ad1->height;

  if (ad2->height < (PAGE_HEIGHT / 2))
    d2 = ad2->height;
  else
    d2 = PAGE_HEIGHT - ad2->height;

  return d1 - d2;
}

static int
dual_feasible_map_widths (GList *ads,
                          double (*w1) (double, double),
                          double (*w2) (double, double), double e2)
{
  GList *l, *c;
  double lb, lb_max;
  double old_contrib, new_contrib;
  int ie, new_ie;
  l = g_list_sort (g_list_copy (ads),
                   (GCompareFunc) compare_width_dist_from_middle);

  lb = 0;
  ie = 0;
  for (c = l; c != NULL; c = c->next)
    lb = lb + w1 (AD_UNIT_WIDTH (c), 0) * w2 (AD_UNIT_HEIGHT (c), e2);

  lb_max = lb;
  c = l;
  while (c != NULL)
    {
      new_ie = MIN (AD_WIDTH (c), 8 - AD_WIDTH (c)) + 1;

      while (c != NULL && MIN (AD_WIDTH (c), 8 - AD_WIDTH (c)) < new_ie)
        {
          old_contrib = w1 (AD_UNIT_WIDTH (c), AD_WIDTH_TO_UNIT (ie)) *
            w2 (AD_UNIT_HEIGHT (c), e2);
          new_contrib = w1 (AD_UNIT_WIDTH (c), AD_WIDTH_TO_UNIT (new_ie)) *
            w2 (AD_UNIT_HEIGHT (c), e2);
          lb = lb - old_contrib + new_contrib;
          c = c->next;
        }

      lb_max = MAX (lb_max, lb);
      ie = new_ie;
    }

  g_list_free (l);

  return ceil (lb_max);
}


static int
dual_feasible_map_heights (GList *ads,
                           double (*w1) (double, double), double e1,
                           double (*w2) (double, double))
{
  GList *l, *c;
  double lb, lb_max;
  int new_ie, ie;
  double old_contrib, new_contrib;

  l = g_list_sort (g_list_copy (ads),
                   (GCompareFunc) compare_height_dist_from_middle);

  lb = 0;
  ie = 0;
  for (c = l; c != NULL; c = c->next)
    lb = lb + w1 (AD_UNIT_WIDTH (c), e1) * w2 (AD_UNIT_HEIGHT (c), 0);

  lb_max = lb;
  c = l;
  while (c != NULL)
    {
      new_ie = MIN (AD_HEIGHT (c), PAGE_HEIGHT - AD_HEIGHT (c)) + 1;

      while (c != NULL &&
             MIN (AD_HEIGHT (c), PAGE_HEIGHT - AD_HEIGHT (c)) < new_ie)
        {
          old_contrib = w1 (AD_UNIT_WIDTH (c), e1) *
            w2 (AD_UNIT_HEIGHT (c), AD_HEIGHT_TO_UNIT (ie));
          new_contrib = w1 (AD_UNIT_WIDTH (c), e1) *
            w2 (AD_UNIT_HEIGHT (c), AD_HEIGHT_TO_UNIT (new_ie));
          lb = lb - old_contrib + new_contrib;
          c = c->next;
        }

      lb_max = MAX (lb_max, lb);
      ie = new_ie;
    }

  g_list_free (l);

  return ceil (lb_max);
}

static int
dual_feasible_map_both (GList *ads,
                        double (*w1) (double, double),
                        double (*w2) (double, double))
{
  GList *l, *c;
  int lb, lb_max;
  int ie;

  l = g_list_sort (g_list_copy (ads),
                   (GCompareFunc) compare_height_dist_from_middle);

  ie = 0;
  lb_max = 0;
  c = l;

  while (c != NULL && ie < PAGE_HEIGHT / 2)
    {
      lb = dual_feasible_map_widths (ads, w1, w2, AD_HEIGHT_TO_UNIT (ie));
      lb_max = MAX (lb_max, lb);
      ie = MIN (AD_HEIGHT (c), PAGE_HEIGHT - AD_HEIGHT (c)) + 1;
      while (c != NULL &&
             MIN (AD_HEIGHT (c), PAGE_HEIGHT - AD_HEIGHT (c)) < ie)
        c = c->next;
    }

  g_list_free (l);

  return lb_max;
}

int
map_u (GList *ads)
{
  GList *c;
  double lb, lb_max;
  int k;

  lb_max = 0;
  for (k = 0; k < PAGE_WIDTH + 1; k++)
    {
        lb = 0;
         for (c = ads; c != NULL; c = c->next)
          lb = lb + dual_feasible_u (AD_UNIT_WIDTH (c), k) * AD_UNIT_HEIGHT (c);
        lb_max = MAX (lb, lb_max);

        lb = 0;
         for (c = ads; c != NULL; c = c->next)
          lb = lb + AD_UNIT_WIDTH (c) * dual_feasible_u (AD_UNIT_HEIGHT (c), k);
        lb_max = MAX (lb, lb_max);
    }

  return ceil (lb_max);
}


int
lower_bound_dual_feasible (GList *ads)
{
  int lb, b;

  lb = dual_feasible_map_heights (ads, dual_feasible_id, 0, dual_feasible_id);

  b = dual_feasible_map_heights (ads, dual_feasible_u1, 0, dual_feasible_U);
  lb = MAX (lb, b);
  b = dual_feasible_map_widths (ads, dual_feasible_U, dual_feasible_u1, 0);
  lb = MAX (lb, b);
  b = dual_feasible_map_heights (ads, dual_feasible_u1, 0, dual_feasible_phi);
  lb = MAX (lb, b);
  b = dual_feasible_map_widths (ads, dual_feasible_phi, dual_feasible_u1, 0);
  lb = MAX (lb, b);

  b = dual_feasible_map_heights (ads, dual_feasible_id, 0, dual_feasible_U);
  lb = MAX (lb, b);
  b = dual_feasible_map_widths (ads, dual_feasible_U, dual_feasible_id, 0);
  lb = MAX (lb, b);

  b = dual_feasible_map_both (ads, dual_feasible_phi, dual_feasible_phi);
  lb = MAX (lb, b);

  b = map_u (ads);
  lb = MAX (lb, b);

  return lb;
}
```

C.7 List of Functions