Approximating the Advertisement Placement Problem

Ari Freund*       Joseph (Seffi) Naor†

April 2, 2001

Abstract

The advertisement placement problem deals with space and time sharing by advertisements on the Internet. Consider a Web page containing a rectangular display area (e.g., a banner) in which advertisements may appear. The display area can be utilized efficiently by allowing several small ads to appear simultaneously side by side, as well as by cycling through a schedule of ads, allowing different ads to be displayed at different times. A customer wishing to purchase advertising space specifies an ad size and a display count, which is the number of times their ad should appear during each cycle. The scheduler may accept or reject any given advertisement, but must be able to schedule all accepted ads within the given time and space constraints. Each advertisement has a non-negative profit associated with it, and the objective is to schedule a maximum-profit subset of ads. We present a \((3 + \epsilon)\)-approximation algorithm for the general problem, as well as \((2 + \epsilon)\)-approximation algorithms for two special cases.

*Department of Computer Science, Technion, Haifa 32000, Israel. Part of this work was done while the author was visiting Bell Labs, Lucent Technologies, 700 Mountain Ave., Murray Hill, NJ 07974. E-mail: arief@cs.technion.ac.il.

†Department of Computer Science, Technion, Haifa 32000, Israel. Part of this work was done when the author was at Bell Labs, Lucent Technologies, 700 Mountain Ave., Murray Hill, NJ 07974. E-mail: naor@cs.technion.ac.il.
1 Introduction

The ad placement problem. The last decade has witnessed the advent of the World Wide Web and graphical browsers, and the consequent explosion in the use of the Internet. From a network serving mainly academia, the Internet has grown into a global communications network providing a diverse range of services including search engines, email, real-time video conferencing, software archives, etc. Virtually all of these services are conveniently accessible through the World Wide Web, and many of them are provided free of charge.

The evolution of the Internet has tracked a positive feedback loop, in which easy accessibility and low (or no) cost create an ever-expanding user base, which, in turn, attracts investors and developers, who bring about a further increase in the quality, diversity, and accessibility of services. While many of the services remain free of charge, investors still expect to make money. A lot of money. It has always been assumed that this would be achieved, at least in part, through advertising on the Web (as well as more sinister activities, such as collecting and selling information on users). Expectations notwithstanding, the quick-profit promise of the Internet has yet to be realized, and a certain amount of sobering seems to be taking place in financial circles at the time of this writing.

Still, the fact remains that nearly every commercially operated web site providing some free service contains advertising, most often in the form of a banner stretching across the top of the page. Such banners are often utilized to better advantage by periodically changing their contents, and it is also common practice to place several side-by-side advertisements in a single banner. Thus, advertising on the web combines time and space sharing between users.

We are now ready to define the the ad placement problem. We are given a schedule length of $L$ time slots and a collection of ads which we must schedule within this time frame (presumably the schedule repeats itself every $L$ time units). The ads must be placed in a rectangular display area whose contents can change every time unit. The ads all share the same height, which is the height of the display area, but may have different widths. Several ads may be displayed simultaneously (side by side), as long as their combined width does not exceed the width of the display area. In addition, each ad specifies a display count (in the range $1, \ldots, L$), which is the number of time slots during which the ad must be displayed. These time slots may be chosen arbitrarily by the scheduler, and, in particular, need not be consecutive. There are two problems to consider. In the resource minimization problem all ads must be scheduled, and the objective is to minimize the width of the display area required to enable this. In the profit maximization problem each advertisement has a non-negative profit associated with it, and the scheduler may accept or reject any given ad. The objective is to schedule a maximum-profit subset of ads within a display area of given width.

Although the prime motivation for the ad placement problem has to do with advertising on the Web, we can also consider continuous time and arbitrary display durations (rather than integral display counts). This setting models any scheduling situation in which jobs require the use of a certain amount of resource (or machines) over a certain amount of time. All jobs have a common release-date and a common due-date, and preemption and migration are allowed at no cost. This model has been studied quite extensively in the scheduling community [7], in particular the profit maximization version, in both the single and the multiple machine model [5, 4, 6]. (In both models a job is assumed to perform on only one machine.) However, the results obtained do not carry over to our setting, where ads may have different widths. This corresponds to a scheduling model where the number of machines required for a job is not fixed, but rather job-dependent.
Previous work. The ad placement problem was introduced and studied by Adler et al. [1]. They considered both resource minimization and profit maximization. Nearly all of their results pertain to the special case of divisible ad widths, i.e., when the ad widths are $h_1, \ldots, h_n$ such that there exist positive integers $k_2, \ldots, k_n$ such that $h_{i+1} = h_i/k_{i+1}$ for all $1 \leq i < n$, and in the context of profit minimization there must also be a positive integer $k_1$ such that $h_1 = 1/k_1$. For the resource minimization problem Adler et al. presented an efficient algorithm that finds an optimal schedule in the case of divisible ad widths. This algorithm implies an efficient 2-approximation algorithm for general ad widths (by rounding up the widths to the nearest power of 2). For the profit maximization problem, Adler et al. only considered the special case where the ad widths are divisible and the profit of each ad is proportional to its “volume” (width times display count). They devised a 2-approximation algorithm for this case. They also obtained several results for the on-line version of the profit maximization problem. The work of Adler et al. [1] was implemented and a Java demonstration is available at http://www.bell-labs.com/project/collager.

Our results. Our main result is a $(3 + \epsilon)$-approximation algorithm for the profit maximization problem with arbitrary ad widths and profits. The complexity of our algorithm is polynomial in the problem parameters and $1/\epsilon$. Our approach is opposite to that of Adler et al. [1] in that they consider ads for assignment in non-increasing order of width, whereas we consider them in non-decreasing order. Although non-increasing order has a better chance of yielding optimal solutions, (indeed, as pointed out by Adler et al., it results in an assignment of all ads if such an assignment exists), the opposite order maintains a more even distribution of ads in time slots, making for a better performance guarantee.

In addition to the general problem, we also consider two special cases, for which we show $(2 + \epsilon)$-approximation algorithms. The special cases are: (1) ad widths do not exceed one half of the display area width, and (2) the only ads whose width exceeds half of the display area width are ads that occupy the entire display area. Note that the second special case contains the case of divisible widths. It also contains the first special case, but we consider them separately because our solution for the first case is much simpler than for the second. In fact, the $(2 + \epsilon)$ approximation factor for the first case is achieved by a slight modification of our algorithm for the general problem, and this modified algorithm is then used as a stepping stone to obtain the $(2 + \epsilon)$ performance guarantee for the second special case.

Finally, we demonstrate that the integrality gap of the straightforward linear programming relaxation of the ad placement problem is 3. Actually, we show a stronger result: we observe that the familiar knapsack problem can be obtained from the ad placement problem by relaxing some, but not all, of the integrality constraints, and show that this relaxation incurs a gap of 3.

Organization of this paper. In Section 2 we define the profit maximization problem precisely and introduce some notation and terminology. In Section 3 we present the main ideas of our algorithm, and in Section 4 we describe the algorithm and our treatment of the two special cases. In Section 5 we show that the integrality gap is 3.

----

1 A simpler way to achieve a factor 2 approximation is Graham’s algorithm for scheduling on identical machines [3]. In our context, this algorithm becomes what we (and Adler et al.) refer to as scheduling by the LF heuristic.
2 Problem Statement, Notation, and Terms Used

For simplicity, we develop the algorithm in terms of the following discrete bins formulation, which is essentially the formulation given by Adler et al. [1]. This formulation is equivalent to the informal description of the ad placement problem in the previous section. In Section 4.2 we discuss briefly how our algorithm may be adapted to the more general scheduling problem that results when time is continuous and ad widths are arbitrary rationals.

The discrete bins formulation of the problem follows. The input consists of $L > 0$ unit size bins numbered $\{0, \ldots, L-1\}$ and $n$ jobs labeled $j_1, \ldots, j_n$. Each job $j_i$ is defined by a triplet $(h_i, l_i, w_i)$, where $0 < h_i \leq 1$ is the job's height, $l_i \in \{1, \ldots, L\}$ is its length, and $w_i > 0$ is its weight. In terms of the ad scheduling problem discussed earlier, job heights correspond to ad widths and job lengths correspond to display counts. A feasible solution is a subset of jobs $S \subseteq \{j_1, \ldots, j_n\}$ together with a feasible assignment for $S$. A feasible assignment for $S$ is defined as an assignment of $l_i$ bins to each job $j_i \in S$ such that the total height of jobs to which any given bin is assigned does not exceed 1. More formally, a feasible assignment of $S$ is a mapping $\alpha : S \rightarrow P(L)$ (where $P(L)$ is the power set of $\{0, \ldots, L-1\}$) such that:

1. For all $j_i \in S$, $|\alpha(j_i)| = l_i$.
2. For all $b \in \{0, \ldots, L-1\}$, \[ \sum_{j_i \in S \land \alpha(j_i) = b} h_i \leq 1. \]

The weight of a feasible solution $(S, \alpha)$ is the total weight of jobs in $S$, denoted $w(S)$. The goal is to find a maximum-weight feasible solution.

We will find it convenient to think of feasible assignments as placing copies of jobs in bins rather than assigning subsets of bins to jobs. In other words, instead of assigning bins 0, 3, 4, and 7 to job $j_9$ we will speak of placing the four copies of $j_9$ in bins 0, 3, 4, and 7 (here we have assumed $l_9 = 4$). Indeed, we formulate the algorithm in terms of progressively placing job copies in bins. Taking this metaphor one step further we introduce the notions of volume and capacity. The volume of a single copy of job $j_i$ is $h_i$ and the volume of the entire job is $h_i \cdot l_i$. The capacity of each bin is one volume unit. Also, since our algorithm places job copies one at a time, it is useful to define the occupancy of a bin at a given moment as the total volume of job copies present in the bin at that moment.

3 Elements of the Algorithm

Our algorithm combines two key elements: knapsack relaxation and Smallest Size Least Full (SSLF) heuristic.

Knapsack Relaxation. Consider a feasible solution. It consists of selecting a subset of jobs and distributing all of their copies among the available bins. Clearly, the solution must obey the condition that the total volume of jobs selected does not exceed the total capacity of all bins, namely $L$. Thus, the familiar knapsack problem may be viewed as a relaxation of our problem: for a given input instance $I$ we define an instance of knapsack in which the knapsack capacity is $L$ and the objects are the jobs in $I$. The size and profit of each object are equal to the volume and weight of the corresponding job, as defined by $I$. Then every feasible solution to our problem defines a
feasible solution, with the same weight/profit, to the \textit{knapsack} instance. Thus, the optimum for the \textit{knapsack} relaxation is an upper bound on the optimum for our problem. It is well known that a fully polynomial time approximation scheme (FPTAS) exists for \textit{knapsack} (see, e.g. [2, Chapter 2, pp. 69-74]). We use this fact in our algorithm.

**Smallest Size Least Full (SSLF) Heuristic.** The SSLF heuristic actually consists of two separate heuristics governing the order in which jobs are considered and the order in which bins are considered.

**Smallest size\(^2\) first (SS).** Sort the jobs by height in non-decreasing order. The jobs will be considered for placement in the bins in this order.

**Least full first (LF).** When a job is up for placement, place its copies in the following manner.

Let \(l\) be the length of the job. Repeat the following \(l\) times: place one copy of the job in the currently least full bin (the bin with minimum occupancy, breaking ties arbitrarily) among the bins that do not yet contain a copy of the job.

The salient property of the LF heuristic is expressed in the next lemma, which is slightly more general than Claim 2.8 in [1] and is implied by its proof. We provide a proof for completeness.

**Lemma 1** Suppose jobs are placed in bins according to the LF heuristic (regardless of the order in which the jobs are considered, and allowing bins to overflow). At any given moment during the assignment let \(h_{\text{max}}\) be the maximum height of a job considered so far, and denote by \(f(b)\) the occupancy of bin \(b\). Then \(f(b') - f(b'') \leq h_{\text{max}}\) for all bins \(b'\) and \(b''\).

**Proof.** The proof is by induction. Consider a pair of bins \(b'\) and \(b''\). When the algorithm starts, both bins are empty and the claim holds. Suppose the claim is true after the first \(k - 1\) jobs are processed, and consider the placement of the \(k\)th job, whose height we denote \(h\). The claim can only be violated if \(f(b') - f(b'')\) increases (since \(h_{\text{max}}\) can only increase), and this can only happen if the algorithm places a copy of the \(k\)th job in \(b'\) but not in \(b''\). However, in this case, the LF heuristic implies that \(f(b') \leq f(b'')\) before the assignment, and thus \(f(b') - f(b'') \leq h \leq h_{\text{max}}\) afterwards. A simple extension of this argument shows that the claim holds at all times, not just between jobs. \(\blacksquare\)

4 The Algorithm

The algorithm proceeds in three steps.

1. Use an FPTAS to solve the \textit{knapsack} relaxation. Obtain a set of jobs \(S\) such that \(w(S)\) is within a factor of \(1 + \epsilon/3\) of the optimum for \textit{knapsack}.

2. Place the copies of the jobs in \(S\) in the bins according to the SSLF heuristic. Do so until some bin overflows. Let \(j_k\) be the offending job. Partition \(S\) into three subsets \(S_1 = \{\text{all jobs placed successfully}\}, S_2 = \{j_k\}, \text{ and } S_3 = \{\text{all remaining jobs in } S\}. \) If no overflow occurs then \(S_1 = S\) and \(S_2 = S_3 = \emptyset\).

\(^2\)Following [1], we use the term \textit{size} rather than \textit{height}.
3. Return the maximum-weight set among $S_1$, $S_2$, and $S_3$, together with an assignment. If $S_1$ is returned, the assignment is the one found in Step 2; if $S_2$ is returned, the assignment is trivial; if $S_3$ is returned, the assignment is constructed as described below.

By design, the weight of the solution returned is at least $w(S)/3$, and hence it is $(3+\epsilon)$-approximate.

It remains to show how to construct an assignment for $S_3$. Let us examine the state of the bins just before the algorithm attempts to place job $j_i$ (the offending job). Let $k$ be the number of bins whose occupancy is $1-h_i$ or less. We know that $k < h_i$, for otherwise job $j_i$ would not cause an overflow. Furthermore, since the placement of $j_i$ does cause an overflow, we know that the occupancy of at least one bin is strictly greater than $1-h_i$. Thus, by Lemma 1 the occupancy of every bin is greater than $1-2h_i$ (because the SSLF heuristic ensures that jobs are considered in non-decreasing order of height and thus $h_{\text{max}} = h_i$). Thus, we have $k$ bins with occupancy greater than $1-2h_i$ and $L-k$ bins with occupancy greater than $1-h_i$. Hence the total volume of the jobs in $S_1$ is more than $(L-k)(1-h_i) + k(1-2h_i) > L(1-h_i) - kh_i$. Now, the total volume of jobs in $S$ is at most $L$ (by the knapsack relaxation) and the volume of $j_i$ is $kh_i$, so the total volume of jobs in $S_3$ is less than $Lh_i$. Finally, by the SSLF heuristic, the height of every job in $S_3$ is at least $h_i$, hence the total length of these jobs is less than $L$. Placing a set of jobs with total length less than $L$ in $L$ bins is easy: simply place each job copy in a bin of its own.

### 4.1 Two Special Cases

As we show in Section 5, the analysis of our algorithm is tight (up to $\epsilon$) for general input. There are, however, special cases in which our algorithm can be used as a basis to achieving an approximation factor of $2+\epsilon$. We present two such cases next.

The first special case we consider is the case in which $h_i \leq 1/2$ for all $i$. Suppose we run the algorithm and obtain $S_1$, $S_2$, and $S_3$. Observe that the total length of jobs in $S_2 \cup S_3$ is less than $2L$ since the contribution of $S_3$ is less than $L$ (as we have seen in the previous section), and $S_2$ consists of a single job, whose length is at most $L$ (by definition). Since no job height is greater than $1/2$, we can construct an assignment for $S_2 \cup S_3$ as follows. Create a list of the job copies such that the copies of each job appear consecutively and place the $i$th copy on the list in bin number $i \mod L$. We obtain a $(2+\epsilon)$-approximate solution by choosing the better solution between $S_1$ and $S_2 \cup S_3$.

The second special case is when every job of height greater than $1/2$ has height 1. We combine our algorithm with the dynamic programming approach of the standard knapsack FPTAS in order to find a combination of full height jobs (i.e., jobs of height 1) and small jobs (jobs of height at most $1/2$). We only sketch the idea here because the bulk of the details involved are simply the details of the knapsack FPTAS, and once those are mastered the remaining details can be filled in quite easily.

Recall that the knapsack FPTAS scales the job weights and rounds them to integers. It then calculates an upper bound $\omega_{\text{max}}$ on the optimum weight (with respect to the new job weights), and uses dynamic programming to find $S(\omega)$ for $\omega = 0, 1, \ldots, \omega_{\text{max}}$, where $S(\omega)$ is a minimum volume subset of jobs with total weight $\omega$. Let $L(S(\omega))$ be the total volume of the jobs in $S(\omega)$. The FPTAS returns $S(\omega^*)$, where $\omega^*$ is maximum such that $L(S(\omega^*))$ does not exceed the knapsack’s volume.

We are not interested in the solution returned by the FPTAS, but rather in the sets $S(\omega)$. The key property of these sets is that the total weight of jobs in $S(\omega)$ with respect to the original weights...
is within a factor of $1 + \epsilon$ of the optimum (with respect to the original weights) for a knapsack whose volume is $L(S(\omega))$. We make use of this as follows. We run the FPTAS on the full-height jobs (using a knapsack of volume $L$) and obtain the sets $S(\omega)$. For every $\omega$ we use $L(S(\omega))$ bins to place the copies of the jobs belonging to $S(\omega)$, and then run our $(2 + \epsilon)$-approximation algorithm on the small jobs with the remaining $L - L(S(\omega))$ bins. Thus, we obtain for each $\omega$ a feasible solution that is a combination of full-height jobs and small jobs. It is easy to see that the best of these solutions is $(2 + \epsilon)$-approximate.

4.2 Polynomial-Time Implementation

A straightforward implementation of our algorithm runs in super-polynomial time, since the placement of jobs in bins according to the SSLF heuristic requires $\Omega(L)$ time. This cannot be avoided if the contents of each bin must be specified explicitly in the output. However, a more compact representation is possible, which results in a polynomial-time implementation of the algorithm.

Recall that when the algorithm selects a bin according to the LF heuristic, it chooses the currently least full bin, breaking ties arbitrarily. Let us not break ties arbitrarily, but rather always select the lowest numbered bin among the contestants. Doing so leads to each job being placed in “consecutive runs” of bins. Thus, after processing $i - 1$ jobs the set of bins can be partitioned into at most $i$ super bins, which are simply sets of consecutive bins, such that the contents of all bins belonging to the same super bin are identical. Let us define the occupancy of a super bin to be the occupancy of any one of its bins. Then, to assign the $i$th job we repeatedly select a super bin of minimum current occupancy and place a copy of the job in each of its constituent bins. If necessary, we split the last super bin in order to maintain the invariant that all the bins constituting a given super bin have identical contents. Thus, we can formulate the algorithm entirely in terms of super bins and dispense with the original bins completely. The running time and output size in a straightforward implementation of this algorithm depend polynomially on $n$ and $1/\epsilon$ (the output size depends only on $n$), and are independent of $L$.

It is now clear that the requirement that $L$ and the $l_i$’s be integers serves no real purpose. Our algorithm can deal just as easily with the generalized problem in which $L$ and the $l_i$’s are arbitrary rationals, and jobs may be split into “pieces” at arbitrary points along their $l_i$ axis and the resulting pieces may be placed anywhere in the interval $[0, L]$.

5 The Integrality Gap of the Knapsack Relaxation

For simplicity, we return to the discrete bins version of the problem, although everything we say in this section applies to the general problem (as defined in the last subsection) as well.

The feasibility constraints in our problem contain the following two types of integrality constraints (and, in fact, the problem can be formulated as an integer programming problem in which these two sets of constraints translate into integrality constraints on two different sets of 0-1 variables): (1) every job must either be accepted in its entirety or rejected completely, and (2) every job copy must be either present in its entirety in a given bin or absent from it completely (i.e., it is impossible to place one half of a job copy in one bin and the other half in another bin). Relaxing the constraints of the second type yields the knapsack problem. Hence the term integrality gap is appropriate in connection with the knapsack relaxation.
As we have shown in the previous section, the integrality gap is at most 3. We now demonstrate that it is at least 3 as well. Consider the following problem instance. The input consists of three identical jobs, each with length \( L/2 + 1 \), height \( 1/2 + \epsilon \) (for sufficiently small \( \epsilon \)), and weight \( w \). Clearly, the total volume of all three jobs is roughly \( 3L/4 \) (for sufficiently large \( L \)) and thus the optimum for the knapsack relaxation is \( 3w \). On the other hand, no feasible assignment exists for more than one job, since every such assignment would have to place at least one pair of job copies in the same bin, and that is not allowed because their combined height exceeds 1. Thus, the optimum for our problem is \( w \), and hence the integrality gap for this instance is 3. This also shows that the analysis of our algorithm is tight (up to \( \epsilon \)), for if the input consists of the above three jobs, as well as a fourth job of height 1, length \( L \), and weight \( 3w \), then our algorithm might return one of the first three jobs rather than the optimal solution, which is the fourth.

References


