Revenue Maximization for Multiple Advertisements Placement on a Web Banner Using a Pixel-Price Model

Edmar Langendoen[1], Flavius Frasincar[1*], Mark Riezebos[1], Vladyslav Matsiiako[1] and David Boekestijn[1]

[1]Econometric Institute, Erasmus University Rotterdam, Burgemeester Oudlaan 50, Rotterdam, P.O. Box 1738, NL-3000 DR Rotterdam, South-Holland, The Netherlands.

*Corresponding author(s). E-mail(s): frasincar@ese.eur.nl; Contributing authors: e_langendoen@hotmail.com; markriezebos97@hotmail.com; matsiiako@gmail.com; david.boekestijn@outlook.com;

Abstract

The aim of this paper is to optimize the allocation of multiple advertisements on a Web banner, where the price of an advertisement depends on the location at the banner. This problem can be defined as a two-dimensional single orthogonal knapsack problem with a location-based pixel-price model. A formulation is proposed in which the problem is specified as a 0-1 integer programming problem. As this problem is NP-complete, we mainly focus on a heuristic approach to solve the problem. We propose two new heuristic algorithms: the reactive GRASP algorithm and the partitioning left-justified algorithm. Next to that, we present an exact algorithm that is able to solve small problem instances in a reasonable time. These newly presented algorithms are compared with respect to efficiency and effectiveness to existing algorithms that solve the problem without a location-based pixel-price model. To test the quality of the algorithms, we have executed two experiments. The results of these experiments show that overall the reactive GRASP algorithm is the most effective algorithm, whereas the greedy stripping algorithm is the most efficient.

Keywords: Revenue management, Pixel advertisement, Heuristics, Two-dimensional knapsack problem, Location-based price model

1 Introduction

Worldwide, digital ad spending has dominated total media ad spending over the last few years and continues to grow at increasingly higher rates compared to non-digital ad spending [1]. The main reason for this remarkable difference is the difference in the cost of targeting between online and offline advertising [2]. The Internet is a medium that is particularly well-suited for reaching a large number of (potential) customers. There are different types of online advertising, one of them being pixel advertisement.

Pixel advertisement originates from the Million Dollar Homepage[1]. The website was developed in 2005 by an English student Alex Tew. His goal was to earn a million dollars by selling advertising space at the Million Dollar Homepage, an empty grid of 1000 by 1000 pixels. Advertisers could buy blocks of 10 by 10 pixels for 1 dollar per pixel, place an advertisement, and link it to their website. The website became a viral hit and sold out in 138 days. The last 1000 of pixels were sold at the Ebay auction. Overall, the most widely used auction system is one where various advertisers compete for a place on a banner each time a different user loads a Web page (see, e.g., [3] and [4]). Such auctions are designed to happen within seconds as the sooner the ad is shown, the more impact it can provide. This is beneficial for advertising companies as they can get more sales, but also for advertisers as when their customers are satisfied with the service, they will order more ads. Additionally, having multiple advertisements on a banner, despite not being an industry standard at the moment, is interesting as it can provide a significant business value for the advertisers. Hence, the idea of pixel advertisement, displaying several advertisements on a larger two-dimensional area, has been further investigated.

In the work of [5] the researchers aimed to incorporate pixel advertisements in the design of Web banners. Small advertisements are presented by advertisers to place on a banner. Each advertisement has a different length, height, and price per pixel. In the work of [5], several algorithms are presented to solve the '*Multiple Advertisement Allocation problem*' (MAA-problem): how to allocate these advertisements such that the revenue for the owner of the banner is maximized. Note that not every advertisement can be placed, which leads to concurrency and a higher price per pixel. As described by [5] the multiple advertisement allocation problem can be defined as a two-dimensional, single, orthogonal knapsack problem. The starting point in a knapsack problem is a set of small items (the advertisements) and a set of empty containers (the Web banner). The goal is to find a feasible allocation of a subset of these items to the containers, such that the total value of the items packed is maximized. We are dealing with a single knapsack because there is only one banner to fill. Finally, the advertisements and the banner are both two-dimensional and the sides of the advertisements must be parallel to the sides of the banner, which makes the problem orthogonal.

[1] http://www.milliondollarhomepage.com

The problem addressed in this paper is a modification of the MAA-problem. In the paper of [5] an advertiser needs to pay a certain price per pixel, independent of the location of the advertisement on the banner. It would be more realistic to charge a higher price per pixel for locations that are more frequently viewed. We define this new problem as the '*Multiple Advertisement Allocation problem with a Location-based Pixel-Price model*' (MAALP-problem). To create a realistic price model, we use results from eye-tracking studies. Technically, we model this difference in prices with respect to location as discounting the highest price per pixel (which corresponds to the most-viewed part of the screen).

[5] formulate the MAA-problem, which is in fact the same as the two-dimensional knapsack problem, as a 0-1 integer programming problem. This problem is one of the Karp's 21 NP-complete problems [6], which means that the computation time of any currently known algorithm to solve these problems exactly, increases very quickly if the size of the problem increases. As the location-based price model only changes the way the total revenue is calculated, the MAALP-problem is also NP-complete (each solution for the MAA-problem is also feasible for the MAALP-problem). This is the reason why we mainly focus on a heuristic approach to solve the problem.

The two new heuristics are the partitioning left-justified algorithm and the reactive Greedy Randomized Adaptive Search Procedure (GRASP) algorithm. The effectiveness and efficiency of these algorithms are compared to three heuristics presented by [5] that solve the MAA-problem: the left-justified algorithm, the orthogonal algorithm, and the greedy stripping algorithm. For this comparison, several simulation experiments are executed. Moreover, an exact algorithm that is able to solve small problems is also executed.

The two main contributions of this paper are as follows:

- The formulation of the MAA-problem has been extended to account for prices that are dependent on the pixel locations on a banner (MAALP-problem). This phenomenon is being modelled through discounting the highest price possible based on eye-tracking attention studies;
- Two new heuristics are proposed to solve the MAALP-problem: the reactive GRASP algorithm which is an extension of the GRASP algorithm, and the partitioning left-justified algorithm which is an extension of the left-justified algorithm.

The remainder of this paper is structured as follows. Existing related literature is discussed in Section 2. In Section 3, the multiple advertisement allocation problem with a location-based price model is formally defined and two formulations of this problem are presented. Section 4 presents an exact algorithm and two heuristics to solve this problem. The set-up and results of the simulations we run to compare the algorithms are given in Section 5. In Section 6, we draw conclusions from the results we obtained and provide suggestions for future work.

2 Related work

Advertisement optimization problems can be seen from two angles. The first one is the perspective of advertising companies that aim to maximize their revenues [7–9], and the second one is that of customers trying to maximize the efficiency of their ad placements between platforms, ad types, etc. [10–12]. In this paper, we will look at the perspective of the advertising companies looking to increase their revenues.

The MAA-problem has not been thoroughly investigated in the literature until now. In [13], the authors propose a heuristic to solve this problem. This algorithm is tested for several problem instances. [5] propose 4 different heuristic algorithms to solve the MAA-problem. Moreover, a brute force algorithm that generates an exact solution is described. The efficiency and effectiveness of the algorithms are compared by running two simulations. One simulation compares the heuristics and the exact algorithm for several small instances. The other simulation is a comparison between the heuristics for several large instances. [14] reformulate the model based only on the start location of an advertisement. Using this representation, the authors manage to improve the execution time.

On the contrary, cutting and packing problems are studied extensively in the literature. So, [15] gives a typology for different cutting and packing problems. This typology is further improved by [16]. According to this typology, the problem addressed in our paper is defined as a two-dimensional single orthogonal knapsack problem.

There is a lot of literature available in which variants of the two-dimensional knapsack problem are analyzed. In these previous studies, both exact algorithms and heuristic algorithms are described. Below we will shortly describe these works.

[17] present 4 exact algorithms to solve the two-dimensional single orthogonal constrained knapsack problem, which are based on enumeration schemes. The basis of these enumeration schemes is a natural relaxation of the two-dimensional knapsack: the one-dimensional knapsack problem, with item weights equal to the size of the item. Moreover, a (1-ϵ)-approximation algorithm is presented, which creates a feasible solution with a value of at least $1/3$ of the optimal solution, with polynomial computation time.

[18] propose an exact tree-search procedure to solve the two-dimensional single orthogonal constrained knapsack problem. The problem is formulated as a 0-1 integer programming problem. Such problems can be solved by tree-search algorithms. The speed of these algorithms depends on the goodness of the upper bound on the optimal solution of the problem. The tighter this upper bound, the faster the algorithm. In the paper, a Lagrangian relaxation of the formulation of the problem is used to obtain an upper bound. This upper bound is further reduced by a subgradient optimization procedure.

The 2-staged two-dimensional knapsack problem is analyzed by [19]. This variant of the problem requires that the maximum number of cut directions allowed to obtain each item is fixed to 2, there is no rotation allowed and the number of copies is constrained. In the paper, two integer linear programming models are presented and tested by solving them by a branch-and-bound method of the integer linear programming solver of CPLEX. Finally, several upper bound procedures for the two-dimensional knapsack problem are presented. The first upper bound is obtained by a linear relaxation of the proposed models, so the binary variables can be any value between 0 and 1. The next upper bound is found by column generation. With column generation, only the subset of the variables that are relevant is considered. Last, dual-feasible functions are used to generate an upper bound. A dual-feasible function is a function that maps a problem instance into a new problem instance such that any feasible solution for the original problem is feasible for the new problem.

In the work of [20], an exact non-guillotine cutting tree-search algorithm to solve the two-dimensional cutting problem is analyzed. This problem is defined as cutting a number of rectangular pieces from a single large rectangle, with the objective to maximize the value of the pieces cut. [18] use Lagrangian relaxation and a subgradient procedure to obtain a good upper bound; however, the models used in the papers are completely different.

Examples of metaheuristic algorithms described in the existing literature are genetic algorithms [21], simulated annealing [22] and tabu search [23]. In all these works, the algorithms are tested on several instances. We will present below some of the metaheuristics.

The genetic algorithm presented by [21] addresses several variants of the two-dimensional orthogonal knapsack problem. A genetic algorithm is very similar to natural selection. It works with 'generations' of a fixed number of solutions. Each solution is created with a layer structure. The next generation of solutions is obtained by saving the best solution of the previous generation and adding new solutions by adapting solutions from the previous generation. After this, a post-optimization procedure of the previously best-found solution is executed. This procedure tries to reduce area losses and layer borders in the solution.

In the research paper of [22], a heuristic for the two-dimensional knapsack problem of subtype 4 is presented, where the items may be rotated by 90° and guillotine cutting is not required. This work is based on a local search neighbourhood controlled by simulated annealing. In this heuristic, a sequence pair representation of a solution is used. This means that a solution is presented as a pair of sequences. In the simulated annealing part of the algorithm, a small modification is iteratively made to the sequence pair. This sequence pair is translated into a packing solution and the value of this solution is determined.

The tabu search algorithm, described by [23], is created for the two-dimensional non-guillotine cutting problem. The algorithm consists of a constructive algorithm that creates iteratively an initial solution. This solution is improved by a tabu search algorithm. The initial solution is adjusted by searching for improving moves in the same neighbourhood as the solution. This is done by removing pieces from the solution or adding pieces to the solution. A move is tabu if the combination of the value of the objective function and the smallest empty rectangle on the banner for the new solution is already included in the tabu list.

The effectiveness of the discussed metaheuristic algorithms in comparison to GRASP has been studied extensively in previous works. For example, [24] compared tabu search to GRASP for the switch allocation problem and finds that GRASP obtains better results than tabu search. Additionally, for the flexible job-shop scheduling problem with various constraints, [25] and [26] found that GRASP is able to provide better solutions than a genetic algorithm.

Because of the demonstrated effectiveness of the GRASP algorithm for optimization problems, we contribute to the literature by proposing a reactive GRASP algorithm as an extension thereof. For our second approach, we extend the left-justified algorithm to the partitioning left-justified algorithm. We propose these approaches to solve the MAALP-problem, an extension of the MAA-problem that additionally accounts for prices that depend on the pixel locations on a banner, modelled through discounting the highest price possible based on eye-tracking attention studies.

3 Problem definition

In this section, we define the MAALP-problem formally and give two 0-1 integer linear programming formulations for this problem. First, we provide a formal definition of the problem in Section 3.1. In Section 3.2 a model based on a formulation for the two-dimensional cutting problem is modified such that it is applicable for the MAALP-problem. A second formulation, by adjusting the model given in Section 3.2 for a popular tool, is presented in Section 3.3.

3.1 Formal definition

The formal definition of the MAALP-problem is very similar to the definition of the MAA-problem, as described by [5]. To this end, we will first give the definition of the MAA-problem and afterwards point out the differences between the MAA- and MAALP-problem.

In the MAA-problem, there is a banner B with a width W and height H in which we need to allocate advertisements from the set A. The properties of an advertisement $a_i \in A$ are its width (w_i), its height (h_i) (both measured in pixels), and the price per pixel the advertiser is willing to pay (pp_i), for each $i \in \{1, ..., |A|\}$. The 'starting point' of an ad is defined as (p, q). This means that the left top of the advertisement is at the p^{th} row and the q^{th} column of the banner if we start counting banner pixels from the left top of the banner. The objective of the problem is formulated as maximizing the value of all allocated ads in the banner, such that the ads do not overlap and fit in the banner.

The difference between the MAA-problem and the MAALP-problem is the price model. The price per pixel an advertiser needs to pay depends on the location of the advertisement in the banner. We capture this dependence by giving a location-based discount on pp_i. We reformulate pp_i as mpp_i, the maximum price per pixel an advertiser is willing to pay because the real price per pixel might be lower, which is caused by the discount. For this reason, we can define the MAALP-problem as a two-dimensional single orthogonal knapsack problem with a location-based price model.

3.2 Formulation

As stated in Section 1, we can formulate the MAALP-problem as a 0-1 integer linear programming problem. The formulation given in this section is based on the models described by [5, 20]. [5] consider the MAA-problem for which they give a formulation, although the constraints in their model are non-linear. Using the decision variables of [20], the model can be linearized. Moreover, the function to calculate the total price of a banner (the objective function that needs to be maximized) must be changed, to take into account the location of the ad. Next to the sets and parameters as defined in Section 3.1, we need to define extra sets, parameters, and decision variables to formulate the 0-1 integer programming problem. These will first be given, and afterwards the full model is presented.

3.2.1 Sets

We have already defined one set in the formal definition: A, the set of all advertisements. Let us define the new sets X and Y as the set of the columns and rows of the banner. Mathematically, we can display this as follows:

$$X = \{x \mid 1 \leq x \leq W\}$$

$$Y = \{y \mid 1 \leq y \leq H\}.$$

Let X_i (Y_i) be the set of all possible starting columns (rows) of a_i. X_i and Y_i are subsets of X and Y because a_i cannot start at each point in the banner, without violating the boundaries of the banner, which is visualized in Figure 1 below. This can be stated as:

$$X_i = \{x \mid 1 \leq x \leq W - w_i + 1\}$$

$$Y_i = \{y \mid 1 \leq y \leq H - h_i + 1\}.$$

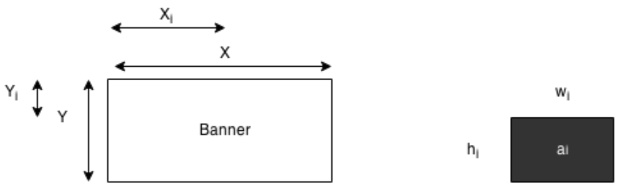


Fig. 1. Visualization of the sets X, Y, X_i and Y_i, for advertisement a_i on Banner.

3.2.2 Parameters and decision variables

In the model, the value of the parameters v_{ipq}, the price of placing advertisement i on location (p, q) is calculated using Equation (1).

$$v_{ipq} = \sum_{r=p}^{p+h_i-1} \sum_{s=q}^{q+w_i-1} mpp_i \cdot (1 - discount_{rs}) \tag{1}$$

Here, $discount_{rs}$ is a parameter that denotes the amount of discount you get on the maximum price at location (r, s). So, for each location occupied in the banner by a_i, if starting at (p, q), we discount the maximum price. Finally, we take the sum over all these discounted prices. In Section 5, we will further specify how we determine the discount (which is problem-dependent). Additionally, let

$$a_{ipqrs} = \begin{cases} 1 & \text{if } p \leq r \leq p + h_i - 1 \text{ and } q \leq s \leq q + w_i - 1 \\ 0 & \text{otherwise.} \end{cases}$$

In other words, a_{ipqrs} is equal to 1 if a_i cuts through point (r, s) when it starts at point (p, q), and equals 0 otherwise. These parameters exist $\forall p \in Y_i$, $\forall q \in X_i$, $\forall r \in Y$, $\forall s \in X$ and $\forall i \in A$.

We furthermore define our decision variables as:

$$x_{ipq} = \begin{cases} 1 & \text{if the left top of } a_i \text{ is allocated on position (p,q) of the banner} \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, these variables exist $\forall p \in Y_i$, $\forall q \in X_i$, and $\forall i \in A$.

3.2.3 Model

The 0-1 integer programming problem can be modelled as follows

$$\text{Max.} \sum_{i \in A} \sum_{p \in Y_i} \sum_{q \in X_i} v_{ipq} x_{ipq} \tag{2}$$

$$\text{s.t.} \sum_{i \in A} \sum_{p \in Y_i} \sum_{q \in X_i} a_{ipqrs} x_{ipq} \leq 1 \qquad \forall r \in Y, \forall s \in X \tag{3}$$

$$\sum_{p \in Y_i} \sum_{q \in X_i} x_{ipq} \leq 1 \qquad \forall i \in A \tag{4}$$

$$x_{ipq} \in \mathbb{B} \qquad \forall i \in A, \forall p \in Y_i, \forall q \in X_i \tag{5}$$

The objective function (2) maximizes the value of all the allocated advertisements. The set of constraints (3) ensure that each location (r, s) on the banner is occupied by at most one advertisement, so overlapping ads are not allowed. Constraint set (4) ensures that each ad is allocated at most once on the banner. Finally, the decision variables are required to be binary by constraint set (5).

3.3 Formulation adapted to the MATLAB solver

Various programming languages such as MATLAB[2], AIMMS[3] or Excel[4] have built-in solvers to solve an integer linear programming problem. Some of these solvers are very flexible (e.g., AIMMS) and allow you to enter complex constraints as in the first formulation. However, other built-in solvers are more restrictive, and can only solve (mixed) integer linear programming problems in a standard form.

The second formulation of the MAALP-problem we present here does not differ in essence from the first formulation. However, it adds practicality by providing an alternative specification for the MATLAB community. It is a rewritten version of the first formulation to this standard form, as MATLAB can only solve problems in this form. To be able to do this, we define new sets, parameters, and decision variables. Note that we will not use the second formulation in our implementation of the exact algorithm due to speed limitations in the MATLAB environment.

3.4 Sets

We define two new sets

$$\begin{aligned} J &= \{j \mid j = (i,p,q), i \in A, p \in Y_i, q \in X_i\} \\ K &= \{k \mid k = (r,s), r \in Y, s \in X\}. \end{aligned}$$

The set J consist of all possible starting points for all advertisements, so the cardinality of J is $\sum_{i \in A} |Y_i| \cdot |X_i|$. The set K contains all points in the banner, so $|K| = H \cdot W$.

3.5 Parameters and decision variables

Using the new set J, we can vectorize the parameters v_{ipq} and variables x_{ipq} to v_j and x_j, with $j \in J$. We can define parameter a_{ipqrs} as a_{jk}. Moreover, we define a new parameter

$$aa_{ji} = \begin{cases} 1 & \text{if index } j \text{ belongs to } a_i \\ 0 & \text{otherwise.} \end{cases}$$

[2]http://nl.mathworks.com/help/optim/ug/mixed-integer-linear-programming-algorithms.html

[3]http://www.aimms.com/aimms/overview/

[4]http://www.solver.com/excel-solver-help

3.6 Model

We can rewrite Formulation 1 as the 0-1 integer programming problem like so.

$$\text{Max.} \sum_{j \in J} v_j x_j \tag{6}$$

$$\text{s.t.} \sum_{j \in J} a_{jk} x_j \leq 1 \qquad \forall k \in K \tag{7}$$

$$\sum_{j \in J} aa_{ji} x_j \leq 1 \qquad \forall i \in A \tag{8}$$

$$x_j \in \mathbb{B} \qquad \forall j \in J \tag{9}$$

The order of the objective function and restrictions (6) - (9) is the same as that of (2) - (5). Restrictions (7) - (8) ensure that the ads do not overlap and that each ad can only be placed once, respectively. Restrictions (9) set a binary domain for all decision variables. This formulation can easily be written in the standard matrix notation. We create two vectors f and x, with elements v_j, respectively x_j, with $j \in J$. Let the restriction matrix A_{rest} consist of two sub-matrices $A1$ and $A2$. Then the elements of $A1$ are a_{jk}, and the elements of $A2$ are aa_{ji}. Finally, we create a column vector b with all elements equal to one, with a length of $|K| + |A|$. This gives us the following formulation.

$$\text{Min.} \ -f^t x \tag{10}$$

$$\text{s.t. } A_{\text{rest}} \cdot x \leq b \text{ with } A_{\text{rest}} = \left[\frac{A1}{A2}\right] \tag{11}$$

$$0 \leq x \leq 1 \tag{12}$$

$$x \in \mathbb{Z} \tag{13}$$

4 Algorithms

This section describes the different algorithms which have the purpose to give an efficient and effective solution for this problem—a solution is efficient when it is fast to compute; a solution is effective when its quality is high (the generated revenue is high in our context). We have implemented an exact algorithm and two heuristic algorithms to solve the MAALP-problem: the reactive GRASP algorithm and the partitioning left-justified algorithm. The algorithms will be compared to the heuristics that solve the MAA-problem, as described by [5].

4.1 Exact algorithm

The exact algorithm that solves the MAALP-problem is implemented in the programming language Java and makes use of the IBM CPLEX Optimizer. We have used a Java wrapper for CPLEX as we would like the code to be usable by Web applications which are often developed in Java. This optimization tool is able to solve the problem formulated in (2) - (5).

To solve the MAALP-problem exactly, first the parameters need to be generated. With the set of advertisements A, the empty banner B, and the matrix *discount*, the parameters are generated according to the formulas in Section 3.2.2. Now, we perform the so-called exact algorithm which entails going through all the possible pixels and ad sets.

With these parameters, the objective function from (2) and the constraints from (3) and (4) are added to the model, such that the CPLEX Optimizer is able to find the exact solution. Finally, the solution found by the CPLEX Optimizer is translated to a representation of the banner in matrix form.

As said before in Section 1, the MAALP-problem is NP-complete. This means that the computation time of an exact algorithm, as presented in this section, increases exponentially with the size of the problem, unless P=NP. As a result of this fact, this exact algorithm is impractical for real MAALP-problems in our online context.

Research by [27] has shown that majority of the people on the Web break their sessions after 15 seconds of waiting for a result. Hence, taking into account the desired use of the algorithm on the Web similar to the one of [28], the computation time of an algorithm should not be long. However, since the problem is, as described by [5], very specific, and due to the experiments performed, we set the maximum computation time to 30 seconds.

For these reasons, we only solve relatively small instances exactly. These instances are described in detail in Section 5.2.1. To solve more realistic instances, we only use heuristics.

4.2 Reactive GRASP algorithm

The reactive GRASP algorithm we present is based on the algorithm described by [29]. The authors of this paper propose a Greedy Randomized Adaptive Search Procedure (GRASP) algorithm for the constrained two-dimensional non-guillotine cutting problem. This algorithm has been modified slightly and we added some extra options to make it suitable for solving the MAALP-problem. We start by giving an overview of a generic GRASP algorithm in Section 4.2.1. Subsequently, we describe the specific construction phase (Section 4.2.2), and the improvement phase (Section 4.2.3) used in the reactive GRASP algorithm. Finally, we give an overview of the complete reactive GRASP algorithm in Section 4.2.4.

4.2.1 GRASP

The GRASP algorithm was first described by [30] as an iterative randomized sampling technique for solving combinatorial optimization problems. The generic structure is displayed in Algorithm 1. A GRASP is an iterative algorithm that executes two phases in each iteration: a construction phase, which creates an initial solution, and an improvement phase, in which this solution is possibly improved using a local search algorithm. If the found solution improves on the best solution found in the previous iterations, this solution is saved.

Algorithm 1 Pseudo-code of the generic GRASP algorithm

```
for iteration = 1: maximum iterations do
    Run construction phase
    Run improvement phase
    if solution better than best found solution then
        Update best found solution
    end if
end for
return: Best found solution
```

The construction phase of a GRASP algorithm is iterative as well. In each iteration of the construction phase, one element is added to the solution created thus far. The determination of this element to add is done by ordering the set of all possible elements according to a greedy function. The adaptive component of the heuristic is that in each iteration, this ordering is done again, and might be adjusted because of the addition of the previous element. The element chosen to be added does not need to be the best option from this ordered list. The element is chosen randomly from the best options.

The initial solution, created in the construction phase, is not necessarily locally optimal. During the improvement phase, a local search algorithm tries to find a better solution in the neighbourhood of the found solution.

4.2.2 Construction phase

The pseudo-code of the specific construction phase we use to build an initial solution is displayed in Algorithm 2. This construction phase is an iterative process as well and considers the (partly filled) banner as a set of empty rectangles that need to be filled (L). Before the iterations, the algorithm sorts the set of advertisements A.

We sort the ads in decreasing order according to the *maximum price per pixel* (mp) or the *maximum total price of the ad* ($w \times h \times mp$). We think these are the most effective sorting criteria because allocating advertisements with a high maximum (total) price will lead to a large increase in the total value of the banner. Ties for one criterion will be broken by a second sorting criterion. For this second sorting criterion, we use one proposed by [5]: the *width* (w), *height* (h), *size* ($w \times h$), *flatness* (w/h) and *proportionality* ($|log(w/h)|$) of an ad. So for each primary sorting criterion, we have 10 different secondary sorting criteria (5 different criteria ascending and descending). Hence, there are in total 20 different orderings for each set of advertisements A.

After this ordering, we try to fill the banner with advertisements, by allocating ads from the set A to empty rectangles in the banner. The set of all empty rectangles in the banner L first needs to be sorted. We use three options for this, of which only the first option was used in the original GRASP algorithm:

1. Sort the set L ascending according to size;

2. Sort the set L ascending according to the average discount per pixel in the rectangle;
3. 'Sort' the set L randomly.

The reason why only option one was considered by [29] is that if a large rectangle is filled with a small piece at the beginning, the resulting rectangles might be useless for large pieces that still need to be cut. We added the second option because we think it might be profitable to allocate the advertisement with the highest (total) price to rectangles with a low discount percentage. The last option is added to see whether the ordering of the rectangles has indeed an influence on the total price of the banner. After sorting A and L, the iterative process starts. In each iteration, we iterate through the sorted list L, until we find a rectangle in which an advertisement from A fits. If there is no such rectangle left, we stop and return the initial solution. If there is an advertisement a_i that does fit in a rectangle, we add a_i and all the other ads which fit in the same rectangle to the set A_{fitting}.

We determine the ad to place from this set A_{fitting} in three ways:

1. Select the ad at random from the set $S = \{j \mid v_j \geq d \times v_{max}\}$;
2. Select the ad at random from the $100(1-d)\%$ 'best' ads;
3. Select the 'best' ad.

Algorithm 2 Pseudo-code of the construction phase

```
 1: function CONSTRUCTION_PHASE($A, B, d, L, C, S, discount$)
Require: $A$ = Set of advertisements to allocate and their properties
Require: $B$ = Matrix of size $(H, W)$, which represents the banner
Require: $d \in [0.1, 0.2, ..., 0.9]$
Require: $L$ = Set of empty rectangles in the banner and their properties
Require: $C$ = Set of advertisements already allocated and their properties
Require: $S$ = Price of banner B
Require: $discount$ = Matrix of size $(H, W)$, which represents the discount for each
    pixel of the banner
 2:     Order set of advertisements $A$
 3:     $done$ = false;
 4:     while $done$ == false do
 5:         Order set of empty rectangles $L$  %% Option 1 %%
 6:         $fitting\_piece\_found$ = false; $A_{\text{fitting}}$ = []; $j = 1$;
 7:         while $fitting\_piece\_found$ == false && $j \leq |L|$ do
 8:             if $rect_j \in L$ is not marked as unusable then
 9:                 for all $a_i \in A$ do
10:                     if $a_i$ fits in $rect_j$ then
11:                         $rectangle = rect_j$;
12:                         add $a_i$ to $A_{\text{fitting}}$;
13:                         $fitting\_piece\_found$ = true;
14:                     end if
15:                 end for
16:                 if $fitting\_piece\_found$ == false then
17:                     mark $rect_j$ as unusable;  %% no ad fits in rectangle %%
18:                 end if
19:             end if
20:             if $fitting\_piece\_found$ == false then
21:                 $j = j+1$;  %% check next rectangle %%
22:             end if
23:         end while
```

```
24:         if $fitting\_piece\_found$ == false then
25:             $done$ = true;
26:             break; %% stop if there is no usable empty rectangle left %%
27:         end if
28:         Select $ad$ from $A_{\text{fitting}}$ to add to the $rectangle$ and calculate the
    discounted $price$ of $ad$ %% Option 2 %%
29:         Place $ad$ in banner $B$ %% Option 3 %%
30:         $S = S + price$;
31:         Remove $ad$ from $A$; remove $rectangle$ from $L$
32:         Add $ad$ to set of allocated ads $C$
33:         Check if there are new rectangles created for $L$, by placing the $ad$
34:         if new rectangles for $L$ then
35:             try to merge them with existing rectangles in $L$
36:         end if
37:         Add new rectangles to L
38:     end while
39:     return: $A, B, C, L, S$;
40: end function
```

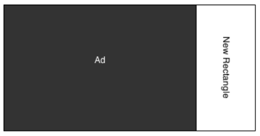
In the first option, v_{max} is the value of the primary sorting criterion of the first ad placed in A_{fitting}, so we choose ads from a set in which the value of the primary sorting criterion deviates at most $100(1-d)\%$ from the highest value of the primary sorting criteria. In options two and three, the 'best' ads, are the ones placed first in the set A_{fitting}. Options one and two are the probabilistic factor of the GRASP algorithm, whereas option three leads to a deterministic constructive phase. Therefore, option three is only used in the improvement phase, which will be explained in Section 4.2.3. The parameter d is a value between 0.1 and 0.9, and is chosen randomly before the construction phase. How the parameter d is chosen, will be elaborated on in Section 4.2.4.

If the rectangle that will be filled, and the ad that will be allocated are chosen, this ad will be placed on the banner, and removed from A. The chosen rectangle is removed from L. The advertisement will always be placed in a corner because this results in the largest new empty rectangles. We use two different approaches to decide in which corner of the rectangle the advertisement is placed:

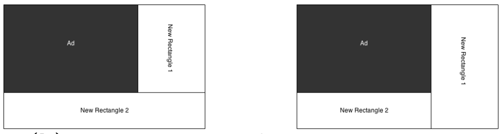
1. Place the ad in the corner which is nearest to a corner of the banner;
2. Place the ad in the corner which yields the highest price for the ad.

Under option one, once the ad has been placed in the corner of a rectangle, the ad is then moved as close as possible to the corresponding corner of the banner (in order to maximize the empty space in the center of the large rectangle). Only option one is used by [29], and is expected to result in a higher fill rate of the banner. I.e., the new empty rectangles are centered in the middle of the banner and can thus be more easily merged with existing ones, so more ads can be allocated. On the contrary, the reason for adding the second option is that it leads to a higher price of the ad. If the ad is placed on the banner, the total price of the banner S is updated, by adding the discounted price of the placed ad.

An advertisement can fill a rectangle totally or partially. If the width (height) of the ad is equal to that of the rectangle, but the height (width) is smaller, there arises only one new rectangle, see Figure 2.



(a) One new rectangle.



(b) Two new rectangles give two options.

Fig. 2. Examples of partially filled rectangles.

If both the size and the width of the ad are smaller than the rectangle, there are two new rectangles. There are two possibilities to choose the rectangles. We use the option where the next ad of the sorted set A fits in the largest rectangle, i.e., the rectangle with the largest size. For the newly created rectangles, we check whether it is beneficial to merge one of them with existing empty rectangles in L. Again, the option where the first ad of the sorted set A fits in the biggest rectangle is chosen. These new rectangles are added to the set L and the original rectangles are deleted.

The different options we use to sort L, select an ad to place, and place an ad, are displayed in Table 1. We added some new options that are not used in the research paper of [29], which take the location-based price model into account (for example *Option 1.2* and *Option 3.2*).

Table 1. Summary of the different options in the construction phase.

Identifier	**Description**	**Possibilities**
Option 1	Sorting L	1. Sort L ascending w.r.t. size 2. Sort L ascending w.r.t. average discount per pixel 3. 'Sort' L randomly
Option 2	Selecting ad to place	1. Select ad from $S = \{j \mid v_j \geq d \times v_{max}\}$ 2. Select ad from $100(1-d)\%$ 'best' ads
Option 3	Placing ad	1. Place in corner nearest to a corner of the banner 2. Place in the corner which yields the highest price for the ad

4.2.3 Improvement phase

The improvement phase of the GRASP algorithm takes as input the output of the construction phase. The pseudo-code of the improvement phase is displayed in Algorithm 3. The initial solution created in the construction phase is adapted by removing several advertisements from the banner.

The removed ads will be put back into the set A, and this set is ordered again using the same sorting criteria employed in the construction phase.

Algorithm 3 Pseudo-code of the improvement phase

```
 1: function IMPROVEMENT_PHASE($A, B, L, C, S$)
Require: $A$ = Set of advertisements to allocate and their properties
Require: $B$ = Matrix of size $(H, W)$, which represents the empty banner; all
    elements are equal to zero
Require: $L$ = Set of empty rectangles in the banner and their properties
Require: $C$ = Set of advertisements already allocated and their properties
Require: $S$ = Price of banner B
 2:     Remove $\beta$% of the allocated ads from banner $B$ and set $C$;
    %% Option improve %%
 3:     Add removed ads to set $A$; add empty rectangles to $L$;
 4:     Update $S$;
 5:     Order set of advertisements $A$;
 6:     Try to merge rectangles in $L$;
 7:      %% execute deterministic construction phase ($d$ has no influence) %%
 8:     $d = 0$;
 9:     $[\sim, B2, C2, \sim, S2]$ = construction_phase($A, B, d, L, C, S1, discount$);
10:     if $S2 < S$ then  %% if not improved, return values from constr. phase %%
11:         $S2 = S$; $B2 = B$; $C2 = C$;
12:     end if
13:     $filled$ = sum of the sizes of all ads in $C2$;
14:     $alloc\_count = |C2|$;
15:     return: $B2, S2, filled, alloc\_count$;
16: end function
```

Removing the ads results in new empty rectangles, which are added to L. We try to merge each rectangle in L with another rectangle in L, according to the method described in Section 4.2.2. If we succeed in merging, we iterate through L again. We stop until it is not beneficial to merge any of the rectangles in L anymore. This partial solution is extended by executing a deterministic construction phase. In this phase, the selection of the ad is done deterministically, as described in Section 4.2.2. Only if the price of the solution is improved, the solution is updated. The method of how to remove the ads from the banner is one of those displayed in Table 2. Under the method which removes β% of the ads at random from the banner, we first remove ads that are adjacent to empty spaces thus enabling to create bigger empty spaces.

Table 2. Summary of the different options in the improvement phase.

Identifier	Description	Possibilities
Option improve	Removing ads	1. Remove the β% last added ads from the banner 2. Remove β% of the ads at random from the banner
β		5, 10, 15

4.2.4 The main reactive GRASP

The reactive GRASP algorithm (Algorithm 4) is a specific version of the GRASP algorithm. During the reactive GRASP, the probability of choosing d from a set D (for selecting an ad to place) is updated after a certain amount of iterations. We define the set D as [0.1, 0.2, ... , 0.9]. At first the probability of choosing d is equal for all values of this set. The total price of the best-found solution ($Sbest$) and the total price of the worst found solution ($Sworst$) over all iterations are updated if necessary. Moreover, we keep track of the sum of the total price of all the solutions obtained by using the chosen d by the parameter $sumS_d$. Each time the number of iterations is a multiple of $maxIter/5$, we update the probability for each possible d, p_d. So, if for a certain d good results are obtained, the probability to choose this d will increase.

We propose another method to update the parameter p_d compared to the one used by [29], which is shown in Equations (14) - (16). As we use the mean price per pixel for a given d ($meanpp_d = \frac{mean_d}{W \times H}$), the price per pixel in the worst solution ($Sworstpp = \frac{Sworst}{W \times H}$) and the price per pixel in the best solution ($Sbestpp = \frac{Sbest}{W \times H}$), the evaluation of the banners does not depend on the size of the banner as was the case in the original method. Moreover, because we take into account the previous value of p_d, we can never generate a probability of 0, which could be the case in the method of [29]. In case $Sbestpp = Sworstpp$, $eval_d$ is not possible to compute and, hence, $eval2_d$ will not be updated.

$$eval_d = \frac{meanpp_d - Sworstpp}{Sbestpp - Sworstpp} \tag{14}$$

$$eval2_d = p_d + \frac{eval_d}{\sum_{d \in D} eval_d} \tag{15}$$

$$p_d = \frac{eval2_d}{\sum_{d \in D} eval2_d} \tag{16}$$

If the maximum number of iterations is reached, the banner with the highest price found over all iterations is returned. A big advantage of the reactive GRASP algorithm is that you can choose the maximum number of iterations yourself. For a more effective solution, a higher maximum number of iterations may be desired; for a more efficient, but probably less accurate solution, one could decrease the maximum. Moreover, because of the randomness in the construction phase, there is a smaller risk of getting stuck in a local optimum. The iterations are independent, so in each iteration the algorithm starts from the beginning. Due to its superior flexibility compared to the GRASP algorithm, we will consider in the evaluation only the reactive GRASP algorithm.

Algorithm 4 Pseudo-code of the reactive GRASP algorithm

```
1: function REACTIVE_GRASP($A, B, discount, maxIter, options$)
Require: $A$ = Set of advertisements to allocate and their properties
Require: $B$ = Matrix of size $(H, W)$, which represents the empty banner; all
    elements are equal to zero
Require: $discount$ = Matrix of size $(H, W)$, which represents the discount for each
    pixel of the banner
Require: $maxIter$ = Maximum number of iterations in the reactive GRASP
Require: $options$ = Set of options according to which to sort L, select the ad, place
    the ad and improve the solution.
    %% initialisation %%
    $D = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]$;
    $Sbest = 0$;
    $Sworst = \text{inf}$;
    $n_d = 0$; $sumS_d = 0$; $p_d = 1/|D|\ \forall d \in D$;
    $iter = 0$;
    while $iter < maxIter$ do
        %% initialisation of an iteration %%
        Choose $d^*$ from $D$, with probability $p_{d^*}$ %% $d^*$ is the current $d$ %%
        $n_{d^*} = n_{d^*} + 1$;
        $L = B$;
        $C = [\ ]$;
        $S1 = 0$;
        $[Anew, Bnew, Cnew, Lnew, S1] =$
                construction_phase($A, B, d^*, L, C, S1$);
        $[B2, S2, filled, alloc\_count] =$
                improvement_phase($Anew, Bnew, Cnew, S1, L$);
        if $S2 > Sbest$ then
            $Sbest = S2$;
            $bannerbest = B2$;
            $fillrate = filled/size(B)$;
            $alloc\_countbest = alloc\_count$;
        end if
        if $S2 < Sworst$ then
            $Sworst = S2$;
        end if
        $sumS_{d^*} = sumS_{d^*} + S2$;
        if mod($iter$,$maxIter/5$) $== 0$ then
            $mean_d = sumS_d/n_d\ \forall d \in D$;
            $mean\_temp = sum(sumS_d)/iter$;
            for all $d \in |D|$ do
                if $isnan(mean_d)$ then %% if $d$ is not used before %%
                    $mean_d = mean\_temp$;
                end if
            end for
            $meanpp_d = mean_d/size(B)\ \forall d \in D$;
            $Sworstpp = Sworst/size(B)$;
            $Sbestpp = Sbest/size(B)$;
            $eval_d = (meanpp_d - Sworstpp)/(Sbestpp - Sworstpp)\ \forall d \in D$;
            $eval2_d = p_d + eval_d/sum(eval_d)\ \forall d \in D$;
            $p_d = eval2_d/sum(eval2_d)\ \forall d \in D$;
        end if
    end while
    return: $bannerbest, Sbest, fillrate, alloc\_countbest$;
end function
```

4.3 Partitioning left-justified algorithm

The partitioning left-justified algorithm is a modification of the left-justified algorithm from the research of [5]. The left-justified algorithm iterates for each advertisement through all the places in the banner, until there is a free location found in which the advertisement fits. The algorithm starts scanning in the left top of the banner and checks all rows before scanning the next column.

The pseudo-code of the partitioning left-justified algorithm is displayed in Algorithm 5. In the partitioning left-justified algorithm the same approach as in the original left-justified algorithm is used, however, the algorithm starts

Algorithm 5 Pseudo-code of the partitioning left-justified algorithm

```
1: function LEFT-_JUSTIFIED_PARTITIONING($A, B, discount$)
Require: $A$ = Set of advertisements to allocate and their properties
Require: $B$ = Matrix of size $(H, W)$, which represents the empty banner; all
    elements are equal to zero
Require: $discount$ = Matrix of size $(H, W)$, which represents the discount for each
    pixel of the banner
2:     Order parts of the banner $rectangles$; Order set of advertisements $A$;
3:     $S = 0$; $filled = 0$; $alloc\_count = 0$;
4:     for all $a_i \in A$ do
5:         $j = 1$;
6:         $finished$ = false; %% Boolean which indicates whether checking $a_i$ is
                               finished %%
7:         $finished\_rectangle$ = false; %% Boolean which indicates whether
                                          checking $rectangle_j$ is finished %%
8:         while $(j \leq |rectangles|)$ && ($finished$ == false) do
9:             $row = upper\_row(rectangle_j)$; %% start at left top of rectangle %%
10:            $col = left\_col(rectangle_j)$;
11:            while ($finished\_rectangle$ == false) && ($finished$ == false) do
12:                if $B$ is empty on $(row, col)$ then
13:                    if $a_i$ fits on $B$ on $(row, col)$ then
14:                        Allocate $a_i$ on $(row, col)$
15:                        $alloc\_count = alloc\_count + 1$;
16:                        $filled = filled + size(a_i)$;
17:                        $S = S + price$;
18:                        $finished$ = true; %% check next ad %%
19:                    else if no space for $a_i$ on $(row, col)$ (not out of bounds) then
                   %% ad would fit in the empty banner on this location%%
20:                        if there is a next row in $rectangle_j$ then
21:                            $row = row+1$;
22:                        else
23:                            if there is a next column in $rectangle_j$ then
24:                                $row = upper\_row(rectangle_j)$;
25:                                $col = col+1$;
26:                            else
27:                                $finished\_rectangle$ = true;
28:                            end if
29:                        end if
30:                    else if $a_i$ goes vertically out of bounds then
31:                        if $row == upper\_row(rectangle_j)$ then
32:                            $finished\_rectangle$ = true;
33:                        else
34:                            if there is a next column in $rectangle_j$ then
35:                                $row = upper\_row(rectangle_j)$;
36:                                $col = col+1$;
37:                            else
38:                                $finished\_rectangle$ = true;
39:                            end if
40:                        end if
41:                    else if $a_i$ goes horizontally out of bounds then
42:                        $finished\_rectangle$ = true;
43:                    end if
44:                else %% $B$ is not empty on $(row, col)$ (line 13) %%
45:                    if there is a next row in $rectangle_j$ then
46:                        $row = row+1$;
47:                    else
```

```
                    if there is a next column in $rectangle_j$ then
                        $row = upper\_row(rectangle_j)$;
                        $col = col+1$;
                    else
                        $finished\_rectangle$ = true;
                    end if
                end if
            end if
            if $finished\_rectangle$ == true then
                $j = j+1$; %% check next rectangle %%
            end if
        end while
    end while
end for
$fillrate = filled/size(B)$;
return: $B, Sbest, fillrate, alloc\_count$;
end function
```

at the left top of the best (least discounted) rectangle of the banner. So we sort the parts of the banner *rectangles* ascending with respect to the discount factor, breaking ties by choosing the largest one first (not all parts have to be a square). Figure 3 gives an example of such an ordering.

After that, the set advertisements A is sorted in the same way as in the initialization of the heuristics by [5], where the authors use 6 different sorting criteria (ascending and descending): the *maximum price per pixel* (mp) and the *width* (w), *height* (h), *size* ($w \times h$), *flatness* (w/h) and *proportionality* ($|log(w/h)|$) of an ad. As in the reactive GRASP algorithm, we add a seventh sorting criteria: the *maximum total price of the ad* ($w \times h \times mp$). We want to choose 2 sorting criteria from a set of 14 (all 7 criteria ascending and descending). The number of different orderings can be defined as P_2^{14}, i.e., 2-permutations of 14. However, we do not use the permutations where the first and second criteria are the same but ascending and descending. The number of such permutations is 14 (for each of the 7 sorting criteria twice). This gives a total number of $\frac{14!}{12!} - 14 = 168$ different orderings of A in the partitioning left-justified algorithm.

If the rectangles of the banner are sorted, the algorithm checks for each ad if it can be placed on a location of the banner, starting at the pixel in the left top of the first rectangle in the ordered set *rectangles*. Each column in each rectangle is checked from top to bottom. This is done for all locations in the banner until the ad is placed, or all rectangles are checked.

5 Simulations

In this section, we present a realistic location-based price model for the advertisements. Moreover, the set-up of the different types of simulations we run is also described. The experiments regarding the exact algorithm are implemented in Java SE 8 using CPLEX Optimization Studio v12.8 (without modifying any default option), and all other experiments are implemented in MATLAB R2017a. All experiments are run on an Intel(R) Core (TM) i7-6500U

0	0.05	0.12
0.05	0.09	0.20

(a) The discount per part of the banner.

1	2	5
3	4	6

(b) The order of the parts of the banner.

Fig. 3. Example of ordering the parts in the banner for the partitioning left-justified algorithm.

CPU at 2.50 GHz with 12 GB RAM. The results of the simulations are analyzed from multiple perspectives. Last, we mention some concluding remarks regarding the results of the experiments.

5.1 Price model

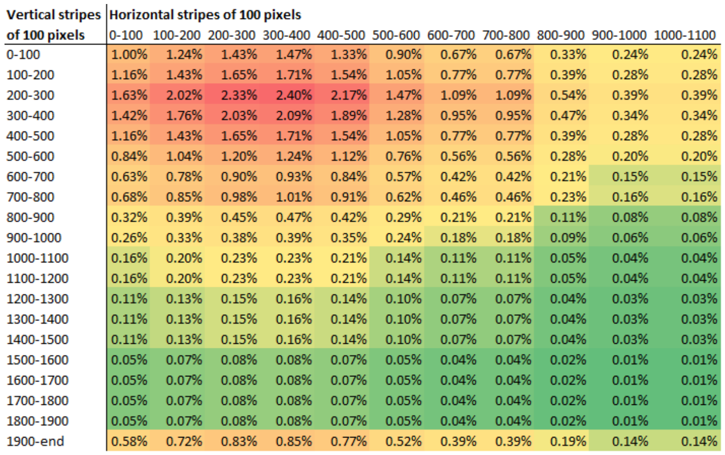
One part of the location-based price model is already presented in Section 3, where we defined the price of an advertisement i on location (p, q) in Equation (1). We did not yet specify how to calculate $discount_{rs}$, the discount an advertiser gets if the ad fills location (r, s) on the banner. To calculate the discount percentages on the banner, we use Equation (17). With this equation, we compute the discount for a block of 100 by 100 pixels. The value of $perc_{ij}$ for a block, is the percentage of viewing time of people on an average Web page, according to the eye-tracking research works [31, 32]. These articles present the results from researches about the horizontal and vertical distribution of attention of people on an average Web page per strips of 100 pixels. From these results, it is possible to make a heatmap with respect to the attention of people on a Web page per blocks of 100 by 100 pixels, as shown in Figure 4.

The value of the discount of a block is assigned to each pixel in the block. The values used for $\max(perc)$ and $\min(perc)$ are the highest and lowest percentages inside the banner. This results in a 0% discount on the maximum price per pixel in the block which is viewed the most inside the banner, and a 20% discount on the maximum price per pixel in the block which is viewed the least. The pixels in the other blocks have a discount between 0 and 20%.

$$disc\_per\_100_{ij} = 0.2 \cdot \frac{\max(perc) - perc_{ij}}{\max(perc) - \min(perc)} \tag{17}$$

5.2 Experiment set-up

We test the algorithms using the two experiment set-ups described by [5]. The first one compares the heuristics to the exact algorithm for a small-sized problem. The second one solves a more realistic problem using the existing and newly proposed heuristics. Existing heuristics are taken from [5]: the



Vertical stripes	Horizontal stripes of 100 pixels										
of 100 pixels	0-100	100-200	200-300	300-400	400-500	500-600	600-700	700-800	800-900	900-1000	1000-1100
0-100	1.00%	1.24%	1.43%	1.47%	1.33%	0.90%	0.67%	0.67%	0.33%	0.24%	0.24%
100-200	1.16%	1.43%	1.65%	1.71%	1.54%	1.05%	0.77%	0.77%	0.39%	0.28%	0.28%
200-300	1.63%	2.02%	2.33%	2.40%	2.17%	1.47%	1.09%	1.09%	0.54%	0.39%	0.39%
300-400	1.42%	1.76%	2.03%	2.09%	1.89%	1.28%	0.95%	0.95%	0.47%	0.34%	0.34%
400-500	1.16%	1.43%	1.65%	1.71%	1.54%	1.05%	0.77%	0.77%	0.39%	0.28%	0.28%
500-600	0.84%	1.04%	1.20%	1.24%	1.12%	0.76%	0.56%	0.56%	0.28%	0.20%	0.20%
600-700	0.63%	0.78%	0.90%	0.93%	0.84%	0.57%	0.42%	0.42%	0.21%	0.15%	0.15%
700-800	0.68%	0.85%	0.98%	1.01%	0.91%	0.62%	0.46%	0.46%	0.23%	0.16%	0.16%
800-900	0.32%	0.39%	0.45%	0.47%	0.42%	0.29%	0.21%	0.21%	0.11%	0.08%	0.08%
900-1000	0.26%	0.33%	0.38%	0.39%	0.35%	0.24%	0.18%	0.18%	0.09%	0.06%	0.06%
1000-1100	0.16%	0.20%	0.23%	0.23%	0.21%	0.14%	0.11%	0.11%	0.05%	0.04%	0.04%
1100-1200	0.16%	0.20%	0.23%	0.23%	0.21%	0.14%	0.11%	0.11%	0.05%	0.04%	0.04%
1200-1300	0.11%	0.13%	0.15%	0.16%	0.14%	0.10%	0.07%	0.07%	0.04%	0.03%	0.03%
1300-1400	0.11%	0.13%	0.15%	0.16%	0.14%	0.10%	0.07%	0.07%	0.04%	0.03%	0.03%
1400-1500	0.11%	0.13%	0.15%	0.16%	0.14%	0.10%	0.07%	0.07%	0.04%	0.03%	0.03%
1500-1600	0.05%	0.07%	0.08%	0.08%	0.07%	0.05%	0.04%	0.04%	0.02%	0.01%	0.01%
1600-1700	0.05%	0.07%	0.08%	0.08%	0.07%	0.05%	0.04%	0.04%	0.02%	0.01%	0.01%
1700-1800	0.05%	0.07%	0.08%	0.08%	0.07%	0.05%	0.04%	0.04%	0.02%	0.01%	0.01%
1800-1900	0.05%	0.07%	0.08%	0.08%	0.07%	0.05%	0.04%	0.04%	0.02%	0.01%	0.01%
1900-end	0.58%	0.72%	0.83%	0.85%	0.77%	0.52%	0.39%	0.39%	0.19%	0.14%	0.14%

Fig. 4. Heatmap of an average Web page, with the percentage of viewing time of blocks of 100 by 100 pixels.

orthogonal algorithm, the left-justified algorithm, and the greedy stripping algorithm.

5.2.1 Experiment set-up with a small number of instances

In the first part of the first experiment, two banners need to be filled. Banner B_1 has a height $H = 4$ and a width $W = 4$, and banner B_2 has a height $H = 4$ and a width $W = 5$. There are two sets of advertisements that can be allocated to these banners: A_1 and A_2. The width w_i, height h_i and maximum price per pixel mpp_i of all a_i contained in these sets are displayed in Table 3 and 4. To calculate the percentages of discount, we use the percentages of viewing time, as shown in Figure 4. We assume that the left top of the banner starts at position (100, 100) and we use the percentages per block of 100 by 100 pixels for each location in the banner. These percentages are displayed in Table 5 and Table 7. Using Equation (17), we calculate the discount matrices. The resulting discount matrices for both banners are displayed in Table 6 and Table 8. The instances we used can be accessed through https://github.com/VladyslavMatsiiako/PMAWB.

As mentioned before in Section 4.1, the computation time of the exact algorithm increases exponentially as the size of the problem increases. In order to examine the behaviour of the exact algorithm and see where its limits are, in the second part of the first experiment, we try to solve the MAALP-problem for five types of standard banners (leader board, half banner, square button, skyscraper and large rectangle) of increasing size, using the exact algorithm. For each banner, a set of advertisements together with their properties is simulated 100 times. The number of ads in the set is determined such that the sum of the sizes of the ads is approximately twice as large as the size of the banner. The width and height of an ad are obtained by applying Equation

(18). The random number *rand* in this equation is drawn from the standard normal distribution.

$$w_i, h_i = \max(1, \min(\min(W, H), \lceil(\min(W, H)/4 \times |rand|\rceil)) \tag{18}$$

The maximum price per pixel an advertiser needs to pay, can differ because of bargaining between the advertiser and the owner of the banner. This maximum price is determined by picking a number from a uniform distribution between 9.0 and 11.0, with a step of 0.1. The real price per pixel an advertiser needs to pay is calculated by subtracting the discount of the pixel from the maximum price.

Table 3. Properties of ads in A_1.

i	w_i	h_i	mpp_i
1	1	1	9.1
2	2	3	9.3
3	1	2	9.5
4	1	1	9.7
5	3	2	9.9
6	2	1	10.1
7	1	1	10.3
8	2	2	10.5
9	3	1	10.7
10	1	3	10.9

Table 4. Properties of ads in A_2.

i	w_i	h_i	mpp_i
1	1	1	9.1
2	2	3	9.3
3	1	2	9.5
4	1	1	9.7
5	3	2	9.9
6	2	1	10.1
7	1	1	10.3
8	2	2	10.5
9	3	1	10.7
10	1	3	10.9
11	1	1	11.0

Table 5. Percentages from Figure 4 of B_1.

1.43	1.65	1.71	1.54
2.02	2.33	2.40	2.17
1.76	2.03	2.09	1.89
1.43	1.65	1.71	1.54

Table 6. Discount matrix of B_1.

0.20	0.15	0.14	0.18
0.08	0.02	0.00	0.05
0.13	0.08	0.06	0.11
0.20	0.15	0.14	0.18

Table 7. Percentages from Figure 4 of B_2.

1.43	1.65	1.71	1.54	1.05
2.02	2.33	2.40	2.17	1.47
1.76	2.03	2.09	1.89	1.28
1.43	1.65	1.71	1.54	1.05

Table 8. Discount matrix of B_2.

0.14	0.11	0.10	0.13	0.20
0.06	0.01	0.00	0.03	0.14
0.10	0.06	0.05	0.08	0.17
0.14	0.11	0.10	0.13	0.20

5.2.2 Experiment set-up with a large number of instances

The second experiment consists of rather large problems. The banners which need to be filled are standard banners. Their names and sizes are shown in Table 9. In this manner, we have a total of 5 banners. Once more, we assume that the left top of the banner is placed 100 pixels from the top and 100 pixels from the left side of the homepage so we can construct the discount matrix using the price model outlined in Section 5.1. We conduct a sensitivity analysis about the trade-off between effectiveness and efficiency with regard to the reactive GRASP algorithm, obtained by varying the maximum number of iterations. Based on this analysis, we determine the maximum number of iterations.

Table 9. Standard banners and their sizes.

Name	W	H
Leader board	728	90
Half banner	234	60
Square button	125	125
Skyscraper	120	600
Large rectangle	336	280

For each type of banner, a set of advertisements together with their properties is simulated 10 times, in the same manner as in the second part of the first experiment. The only difference with the second part of the first experiment is that the width and height (measured in pixels) of an ad are obtained by applying Equation (19) instead of Equation (18). The random number *rand* in Equation (19) is also drawn from the standard normal distribution. According to [5], the distribution of the width and height of the advertisements on the Million Dollar Homepage was approximately normally distributed, with a minimum of 10 pixels. The values generated by Equation (19) are multiples of 10, but can not be larger than the banner dimensions. Moreover, the obtained values are also approximately normally distributed.

$$w_i, h_i = \max(10, \min(\min(W,H), \lceil(\min(W,H)/40 \times |rand|\rceil \times 10)) \qquad (19)$$

The maximum price per pixel an advertiser needs to pay is determined the same as in the second part of the first experiment. The results in Section 5.3.2 are averages over the 10 simulations for each banner type. Again, the instances that were used for each banner can be found at https://github.com/VladyslavMatsiiako/PMAWB.

5.3 Results

Each problem instance in the first experiment is solved by the exact algorithm, and all different versions by the newly presented and existing heuristics. The results of all the different versions of the algorithms are analyzed in the next sections from different perspectives. The effectiveness of an algorithm depends on the price of the banner, whereas the efficiency of an algorithm is measured by the computation time. The fill rate, the percentage of the banner filled by advertisements, is a good indication of the effectiveness. However, the different prices per pixel for ads and the location-based discount influence the total price of the banner as well.

Table 10. The best results for each problem instance in experiment with a small number of instances with respect to price.

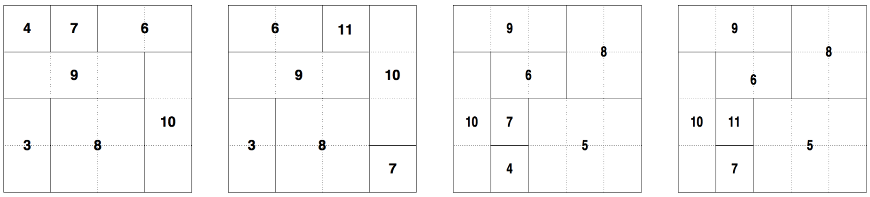
Problem instance	Algorithm	Prim. sort	Sec. sort[5]	Comp. time (s)	Price banner	Fillrate	Options[6]
(B1, A1)	Exact	-	-	0.054	**146.8183**	1	
	Reactive GRASP	maxpp. desc.	price desc.	0.194454	**146.8183**	1	(1, 2, 2, 1, 5)
	Orthogonal	maxpp. desc.	*	0.000565	146.5373	1	
	Part. left just.	prop. desc.	maxpp. desc.	0.000601	146.5262	1	
	Left just.	prop. desc.	price desc.	0.000540	146.4427	1	
	Greedy str.	price desc.	*	0.000173	108.7585	0.75	
(B1,A2)	Exact	-	-	0.038	**146.5778**	1	
	Reactive GRASP	price desc.	prop. asc.	0.0860	**146.5778**	1	(1, 1, 2, 2, 15)
	Part. left just.	prop. desc.	price desc.	0.000343	146.2517	1	
	Left just.	prop. desc.	price desc.	0.000178	146.1818	1	
	Orthogonal	prop. desc.	price desc.	0.000727	146.1728	1	
	Greedy str.	price desc.	*	0.000239	108.394	0.75	
(B2,A1)	Exact	-	-	0.050	**185.2725**	1	
	Reactive GRASP	maxpp. desc.	price desc.	0.1033	**185.2725**	1	(2, 2, 2, 2, 5)
	Part. left just.	height desc.	maxpp. desc.	0.000310	182.3079	1	
	Left just.	width desc.	prop. asc.	0.000201	181.2404	1	
	Orthogonal	width desc.	prop. asc.	0.000383	181.2404	1	
	Greedy str.	height desc.	price desc.	0.000289	143.7303	0.8	
(B2,A2)	Exact	-	-	0.126	**184.766**	1	
	Reactive GRASP	price desc.	width desc.	0.0716	**184.766**	1	(3, 1, 1, 1, 15)
	Left just.	width desc.	maxpp. desc.	0.000187	181.1422	1	
	Orthogonal	width desc.	maxpp. desc.	0.000484	181.1422	1	
	Part. left just.	size desc.	maxpp. asc.	0.000284	178.8797	1	
	Greedy str.	height desc.	price desc.	0.000324	144.0608	0.8	

5.3.1 Experiment with a small number of instances

In Table 10, for the first part of the first experiment, the best results (with respect to the price) for each problem instance are presented. We show for each algorithm the sorting criteria for which the solution is obtained with the highest price of the banner. For the reactive GRASP, algorithm we also present the options that lead to this best result (how is the set of rectangles sorted, in which way is the ad to place selected). This is done by presenting numbers that refer to the possibilities in Table 1 and Table 2. Next to that, the computation time in seconds to find the solution (*Comp. time (s)*), the total price of the solution (*Price banner*), and the percentage of the banner which is filled in the solution (*Fillrate*) are presented for each algorithm.

The best solution per problem instance provided by the exact algorithm is displayed in Figure 5. We observe in Table 10 that the only heuristic which finds the optimal solution for all 4 instances is the reactive GRASP algorithm. So for each instance, the reactive GRASP algorithm is the most effective algorithm. We see that the solutions rendered by the partitioning left-justified algorithm are for all instances more effective than the greedy stripping algorithm, but approximately evenly effective as the left-justified and orthogonal algorithm. The greedy stripping algorithm performs the worst with respect to effectiveness. It is the only algorithm which creates banners that are not fully allocated.

On the other hand, if we compare the algorithms with respect to efficiency, the performance of the left-justified algorithm is the best for most instances. For the reactive GRASP algorithm it even takes more time to find the solution than for the exact algorithm.



(a) B_1, A_1 **(b)** B_1, A_2 **(c)** B_2, A_1 **(d)** B_2, A_2

Fig. 5. The exact solutions for the 4 different problem instances.

From this first part of the experiment with a small number of instances, we can conclude that for small problem instances the reactive GRASP algorithm is the most effective, but also the least efficient algorithm. The greedy stripping algorithm is the least effective but a relatively efficient algorithm. The sizes of the problem instances are too small to draw general conclusions about the quality of the algorithms. Nevertheless, the experiment gives a useful indication of the quality of the results, compared to the optimal solutions.

In Table 11, for the second part of the first experiment, the average computation times over 100 simulations of the exact algorithm are presented for five types of standard banners. Here, we observe that the largest sizes for which the exact algorithm can solve the MAALP-problem in less than 30 seconds are 9×9 pixels, 56×7 pixels, 28×7 pixels, 9×45 pixels, and 6×5 pixels for the square button, leader board, half banner, skyscraper, and large rectangle, respectively. These results illustrate the need for heuristics to solve realistic instances (recall that because of the use on the Web, the computation time of an algorithm is desired to be less than 30 seconds).

Table 11. Average computation times in seconds over 100 simulations of the exact algorithm for different banner sizes.

	Size	Comp. time (s)
Square button	9×9	16.701
(125×125)	10×10	75.612
Leader board	56×7	22.521
(728×90)	64×8	60.396
Half banner	28×7	5.848
(234×60)	32×8	45.011
Skyscraper	9×45	21.127
(120×600)	10×50	48.928
Large rectangle	6×5	0.089
(336×280)	12×10	185.531

5.3.2 Experiment with a large number of instances

We analyze the results of the experiment with a large number of instances from more perspectives than the experiment with a small number of instances, because the larger problems are more similar to real-time MAALP-problems than the smaller problems in the first experiment. To be able to compare the different sized banners, we analyze the properties per pixel of the banner, instead of the properties of the whole banner. In this way, the results do not depend on the size of the banner. To do this, we define the price per pixel of the banner as $Price\ pp = \frac{Price\ banner}{W \times H}$ and the computation time per pixel as $Comp.\ time\ pp = \frac{Comp.\ time}{W \times H}$.

We first conduct a sensitivity analysis about the trade-off between effectiveness and efficiency with regard to the reactive GRASP algorithm, obtained by varying the maximum number of iterations. This analysis is done regardless of the sorting criteria and type of banner. We do not make a distinction between the different options of the reactive GRASP algorithm either. In fact, for every value of the maximum number of iterations, we aggregate all different versions of the algorithm. For each of these values of the maximum number of iterations we compute the mean of *Price pp* and the mean of *Comp. time pp*. These results are displayed in Figure 6.

There is a sudden increase in the mean of *Comp. time pp* between a maximum number of iterations of 40 and 50. Also, as we reach a maximum number of iterations of 40, the mean of *Price pp* is growing relatively slow as the maximum number of iterations increases. Therefore, we set the maximum number of iterations for the reactive GRASP algorithm to 40. This means that for the largest banner we consider (large rectangle), the total computation time is 3.221 seconds on average, which is below 30 seconds, as desired. Note that there is no optimal choice for the maximum number of iterations, as one may want to have higher effectiveness (efficiency) at the expense of lower efficiency (effectiveness).

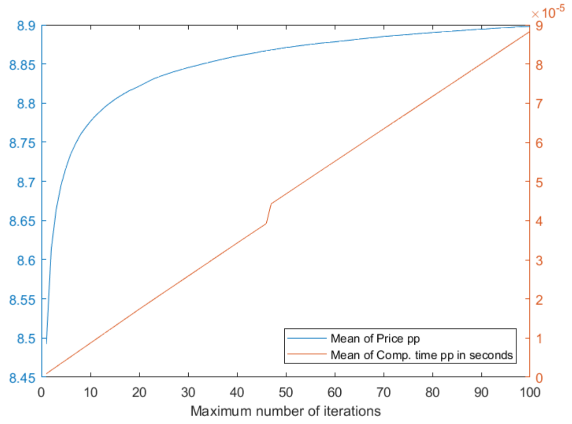


Fig. 6. The trade-off between effectiveness and efficiency with regard to the reactive GRASP algorithm.

As in the first analysis, the second analysis is done regardless of the sorting criteria and type of banner. For the reactive GRASP algorithm, we again do not make a distinction between the different options. Here, we aggregate all different versions of the algorithm and compare their properties by using the *five-number summary* and the mean. The five-number summary is a descriptive statistic that consists of the five most important percentiles of a data-set: the minimum, the first quartile, the median, the third quartile, and the maximum. The values of these descriptive statistics for the price and the computation time per pixel are displayed in Tables 12 and 13.

On almost every point in Table 12, the reactive GRASP algorithm performs the best. The least effective results are generated by the greedy stripping algorithm and the partitioning left-justified algorithm. However, the highest maximum value of the price per pixel is generated by the partitioning left-justified algorithm. Apparently, if the partitioning left-justified algorithm is used on the right banner with the right sorting criteria, it generates a very good solution.

According to the numbers in Table 13, the greedy stripping algorithm is the most efficient algorithm. The least efficient algorithm is the reactive GRASP algorithm. From this we can conclude there is a trade-off between the effectiveness and efficiency of the algorithms: in general, the more efficient an algorithm is, the less effective its results are.

Table 12. The five-number summary and the mean of the *Price pp.*

Algorithm	Minimum	Q1	Median	Q3	Maximum	Mean
Reactive GRASP	7.5840	8.5588	9.0301	9.2296	9.6577	8.8566
Left just.	5.7739	7.6332	8.5688	8.8846	9.6839	8.2275
Orthogonal	5.7606	7.6434	8.5888	8.8821	9.6839	8.2272
Part. left just.	5.9451	7.3669	8.0585	8.7750	9.6850	7.9985
Greedy str.	5.2074	7.3541	8.2605	8.6633	9.5463	7.9901

Table 13. The five-number summary and the mean of the *Comp. time pp* in seconds.

Algorithm	Minimum	Q1	Median	Q3	Maximum	Mean
Greedy str.	1.554E-08	4.266E-08	5.867E-08	8.848E-08	3.780E-07	6.529E-08
Left just.	5.874E-07	2.445E-06	3.924E-06	8.038E-06	3.661E-05	5.905E-06
Orthogonal	1.099E-06	3.206E-06	6.851E-06	1.014E-05	2.099E-05	7.459E-06
Part. left just.	1.157E-06	3.201E-06	5.551E-06	1.539E-05	3.780E-05	1.094E-05
Reactive GRASP	5.040E-06	2.002E-05	3.123E-05	4.781E-05	0.0029	3.374E-05

In the previous analysis, there was no distinction made between the different types of the banners or the sorting criteria. We now do distinguish the different sorting criteria and compare the best results per banner for each algorithm. These results are displayed in Tables 14-18. In 4 of the 5 banner types, the reactive GRASP algorithm creates a solution with the highest fillrate. Except for the *leader board* and the *skyscraper*, the reactive GRASP algorithm generates solutions with the highest price per pixel for all other cases. The drawback of this algorithm is that the computation time per pixel is for almost each banner the highest. For each type of banner, the most efficient algorithm is the greedy stripping algorithm.

We considered all different versions of the reactive GRASP algorithm. In Tables 14-18, we show the best options. As in the results of the experiment with a small number of instances, the options are represented by the numbers stated in Table 1 and Table 2. It is notable that in almost each best solution the algorithm selects the ad from the $100(1-d)\%$ 'best' ads (*Option 2.2*) and places the selected ad in the corner of the rectangle which yields the highest price for the ad (*Option 3.2*). The choice about which rectangle to fill in the construction phase (*Option 1*) and the choice about which ads to remove in the improvement phase (*Option improve*), do not seem to influence the effectiveness. The different options of the reactive GRASP algorithm are further analyzed in the next paragraphs.

Table 14. Best results *leader board* (728x90).

Algorithm	Prim. sort	Sec. sort	Comp. time pp (s)	Price pp	Fillrate	Options
Part. left just.	maxpp. desc.	price desc.	4.48260E-06	9.6850	0.9890	
Left just.	maxpp. desc.	price desc.	3.8466E-06	9.6839	0.9890	
Orthogonal	maxpp. desc.	price desc.	6.6666E-06	9.6839	0.9890	
Reactive GRASP	maxpp. desc.	price desc.	6.9035E-05	9.6577	0.9890	(1, 2, 2, 1, 15)
Greedy str.	maxpp. desc.	size asc.	7.5998E-08	9.5463	0.9774	

Table 15. Best results *half banner* (234x60).

Algorithm	Prim. sort	Sec. sort	Comp. time pp (s)	Price pp	Fillrate	Options
Reactive GRASP	maxpp. desc.	size desc.	5.3945E-05	9.3071	0.9829	(2, 2, 2, 1, 10)
Part. left just.	maxpp. desc.	price desc.	1.7150E-06	9.2798	0.9829	
Orthogonal	maxpp. desc.	price desc.	2.7342E-06	9.2389	0.9829	
Left just.	maxpp. desc.	price desc.	1.1059E-06	9.2392	0.9829	
Greedy str.	maxpp. desc.	flatn. asc.	1.0701E-07	9.1547	0.9779	

Table 16. Best results *square button* (125x125).

Algorithm	Prim. sort	Sec. sort	Comp. time pp (s)	Price pp	Fillrate	Options
Reactive GRASP	price desc.	height desc.	2.0486E-05	8.0132	1	(2, 2, 2, 1, 10)
Part. left just.	price desc.	maxpp. desc.	2.1729E-06	7.6966	0.9158	
Left just.	price desc.	maxpp. desc.	2.4424E-06	7.6909	0.9158	
Orthogonal	price desc.	maxpp. desc.	2.7942E-06	7.6698	0.9171	
Greedy str.	width desc.	maxpp. desc.	5.0799E-08	7.4864	0.8960	

Table 17. Best results *skyscraper* (120x600).

Algorithm	Prim. sort	Sec. sort	Comp. time pp (s)	Price pp	Fillrate	Options
Orthogonal	maxpp. desc.	prop. asc.	7.1995E-06	9.3057	0.9997	
Reactive GRASP	maxpp. desc.	size desc.	3.3084E-05	9.2689	0.9992	(1, 2, 2, 2, 15)
Left just.	maxpp. desc.	price desc.	3.0352E-06	9.2364	0.9968	
Greedy str.	maxpp. desc.	width desc.	4.7787E-08	8.9760	0.9671	
Part. left just.	maxpp. desc.	prop. desc.	5.2536E-06	8.4226	0.8903	

Table 18. Best results *large rectangle* (336x280).

Algorithm	Prim. sort	Sec. sort	Comp. time pp (s)	Price pp	Fillrate	Options
Reactive GRASP	price desc.	flatn. desc.	5.7904E-06	9.1036	0.9778	(3, 1, 2, 1, 10)
Left just.	price desc.	maxpp. desc.	2.8398E-06	8.9557	0.9727	
Orthogonal	price desc.	maxpp. desc.	3.8659E-06	8.9546	0.9731	
Part left just.	price desc.	height desc.	2.7337E-05	8.7751	0.9582	
Greedy str.	height desc.	price desc.	1.9339E-08	8.3776	0.9229	

To analyze the different options of the reactive GRASP algorithm, we aggregate the results for the different sorting criteria and different banners. This is done by averaging the computation time and the price per pixel over all different versions of a specific combination of options. The ten best and worst results are presented in Tables 19 and 20. In Table 19, the combinations of options are ordered descending with respect to the mean computation time, and in Table 20 they are sorted descending with respect to the mean price per pixel.

Table 19. The average results of the reactive GRASP algorithm sorted descending on the mean computation time (efficiency).

Options	Mean Comp. time	Mean Price pp
(2, 2, 1, 1, 15)	3.4089	8.8135
(1, 1, 2, 1, 15)	2.8074	8.8553
(1, 2, 1, 1, 15)	2.2127	8.9224
(3, 2, 1, 1, 15)	2.0902	8.8943
(1, 2, 2, 1, 15)	2.0775	8.9412
(2, 2, 2, 1, 15)	2.0569	8.9300
(1, 1, 1, 1, 15)	1.9221	8.8408
(3, 2, 2, 1, 15)	1.9036	8.9251
(3, 1, 1, 1, 15)	1.8225	8.8050
(2, 1, 2, 1, 15)	1.7833	8.8448
	...	
(1, 1, 2, 1, 5)	1.2649	8.8501
(2, 1, 1, 2, 5)	1.2616	8.7188
(2, 1, 1, 1, 5)	1.2480	8.7239
(2, 1, 2, 1, 5)	1.2464	8.8359
(1, 1, 1, 2, 5)	1.2454	8.8282
(3, 1, 1, 2, 5)	1.2406	8.7962
(1, 1, 2, 2, 5)	1.2296	8.8502
(2, 1, 2, 2, 5)	1.2132	8.8373
(3, 1, 2, 1, 5)	1.1756	8.8275
(3, 1, 2, 2, 5)	1.1644	8.8213

Table 20. The average results of the reactive GRASP algorithm sorted descending on the mean price per pixel (effectiveness).

Options	Mean Comp. time	Mean Price pp
(1, 2, 2, 2, 15)	1.5814	8.9416
(1, 2, 2, 1, 15)	2.0775	8.9412
(1, 2, 2, 1, 10)	1.6667	8.9403
(1, 2, 2, 2, 10)	1.4672	8.9390
(1, 2, 2, 1, 5)	1.3886	8.9359
(1, 2, 2, 2, 5)	1.3454	8.9316
(2, 2, 2, 2, 15)	1.5677	8.9309
(2, 2, 2, 1, 15)	2.0569	8.9300
(2, 2, 2, 2, 10)	1.4473	8.9269
(2, 2, 2, 1, 10)	1.6666	8.9263
	...	
(3, 1, 1, 2, 15)	1.5544	8.8013
(3, 1, 1, 2, 10)	1.3920	8.7995
(3, 1, 1, 1, 5)	1.2653	8.7970
(3, 1, 1, 2, 5)	1.2406	8.7962
(2, 1, 1, 1, 15)	1.5260	8.7278
(2, 1, 1, 1, 10)	1.3840	8.7273
(2, 1, 1, 1, 5)	1.2480	8.7239
(2, 1, 1, 2, 10)	1.3937	8.7231
(2, 1, 1, 2, 15)	1.5120	8.7220
(2, 1, 1, 2, 5)	1.2616	8.7188

We see that most of the versions of the reactive GRASP algorithm which have the highest computation time, use the improvement phase where the last added β% of the ads are removed (*Option improve.1*). The higher this value for β, the higher the computation time is. This is because a larger number of ads are removed when β increases, and thus partially filled banner can be filled with more ads, which takes longer. As can be seen in Table 20, this increase in computation time does not necessarily lead to more effective results. However, it is observable that a combination of selecting an ad from the $100(1-d)$% 'best' ads (*Option 2.2*) and placing the selected ad in the corner which yields the highest price (*Option 3.2*) leads to effective results, whereas the combination of selecting an ad from the set of ads with a value which deviates at most $d\%$ from the best value (*Option 2.1*) and placing the selected ad in the corner which is nearest to a corner of the banner (*Option 3.1*) leads to the least effective results.

Option 2 determines the number of ads from which one is selected. The set of the $100(1-d)\%$ 'best' ads is apparently better to choose from. This can be explained by the fact that relatively more 'best' ads are contained in this set, than in the set of ads with a value that deviates at most $100(1-d)\%$ from the best value. So a 'better' ad (an ad that generates more revenue) will be chosen. The difference in the results caused by the placement strategy can be explained by the following. Placing ads in the corner of the rectangle which is nearest to a corner of the banner should lead to larger empty rectangles in the middle of the banner which should be easier to fill. Placing an ad in the corner which yields the highest price, will scatter the ads more over the banner, leading to smaller empty rectangles. According to the results, a higher price for an ad is more important than larger empty rectangles. How to choose the rectangle in which an ad will be placed, seems to influence neither the effectiveness, nor the efficiency of an algorithm.

6 Conclusion and future work

In this paper we addressed the MAALP-problem, an extension of the multiple advertisement allocation problem where a pixel-price model is used to determine the price of an advertisement on a banner. In Section 3, we presented a formal definition of the problem and gave two 0-1 integer programming formulations that specify the problem.

Using simulation experiments, we found that for each type of banner, the greedy stripping algorithm is the most efficient algorithm. The most effective algorithms with their properties are displayed in Table 21 for each type of banner. The newly proposed reactive GRASP algorithm is the most effective algorithm for three of the five banners. Overall, the newly proposed partitioning left-justified algorithm performs worse than the existing left-justified algorithm and orthogonal algorithm qua effectiveness, but for some specific problem instances it does perform better.

A natural phenomenon that is visible in the experiments is the trade-off between efficiency and effectiveness of an algorithm. The algorithms compared in this paper all render an acceptable solution in a reasonable time span. Which algorithm to choose, depends on your preferences with respect to efficiency and effectiveness. If you care more about the effectiveness of a solution than about the efficiency, it is possible to improve the solution of the reactive GRASP algorithm even further. We defined the maximum number of iterations as 40, but by increasing this number, the algorithm will render more solutions and is possible to find an even better solution. The disadvantage of this is the increase in computation time.

Table 21. The most effective algorithms and their properties for each type of banner.

Banner	Algorithm	Prim. sort	Sec. sort	Options
Leader board (728×90)	Part. left just.	maxpp. desc.	price desc.	
Half banner (234×60)	Reactive GRASP	maxpp. desc.	size desc.	(2, 2, 2, 1, 10)
Square button (125×125)	Reactive GRASP	price desc.	height desc.	(2, 2, 2, 1, 10)
Skyscraper (120×600)	Orthogonal	maxpp. desc.	prop. asc.	
Large rectangle (336×280)	Reactive GRASP	price desc.	flatn. desc.	(3, 1, 2, 1, 10)

Overall, the adaptation of the multiple advertisement allocation problem to account for the location differences is making this problem more real and useful for industry practitioners. The newly presented algorithms (reactive GRASP and partitioning left-justified algorithm) are able to find solutions with better price per pixel outcomes under similar time constraints, when compared to the orthogonal algorithm, the left-justified algorithm, and the greedy stripping algorithm. This would be quite beneficial for companies like Google, Facebook, Snapchat, etc. Here, on the one side are the advertising companies and their ads with corresponding bids. On the other side are the advertisers that need to select which ads to show and how to allocate them optimally. The goal is that both the revenues are optimised and the customers are satisfied with the ad performance. Our work can be used in existing revenue management models [33] by taking pixel advertisement into account.

Despite the extension of the multiple advertisement allocation problem we discussed in this paper, there are still unexplored directions which are interesting to investigate. One of these could be adding a time component to the problem. A simplified variant of this is the ad placement problem. In this problem, a banner needs to be filled with multiple advertisements for different time slots. The banner can have a different allocation in each time slot, but it is also possible to allocate ads in more than one time slot. The simplifying factor is the assumption that each advertisement has the same height as the banner, which makes it a one-dimensional knapsack problem. In the existing literature, there are several algorithms that solve this problem, for example via column generation [34] and Lagrangian decomposition [35] or with a hybrid genetic approach [36]. It would be interesting to extend these algorithms or even create new ones to solve the ad placement problem in the two-dimensional case, as there is no known literature about it thus far.

Another limitation of our paper is that in the real world, there may be cases of ads with non-rectangular shapes. In fact, every such advertisement consists of pixels which are essentially squares. Considering the rectangles together to combine them into certain shapes is another point for possible further research.

The methods in our problems can also be improved. The price model we use is based on the eye-tracking research done by [31, 32]. These researches focus on the gazing time of people on an average website. This is the most useful data now available on where people spend most attention on a Web page. However, the researchers do not mention anything about advertisement banners. To obtain a more realistic price model for the MAALP-problem, it should be investigated where people pay attention to with respect to banners specifically. Moreover, the improvement phase used in the reactive GRASP algorithm could be refined. We remove a couple of ads from the banner and use the deterministic construction phase to place new ads. In this construction phase, the 'best' advertisement which fits, according to the sorting criteria, is placed. If multiple placement strategies are considered (so not only placing the 'best' ads) and the best solution with the highest price of the banner is saved, the improvement can be higher. However, this can lead to an increase in computation time.

Compliance with ethical standards

The authors declare no conflicts of interest and this research was performed without involvement of human participants and/or animals.

References

[1] Insider Intelligence: Worldwide Digital Ad Spending 2021. https://www.insiderintelligence.com/content/worldwide-digital-ad-spending-2021. Accessed 9 March 2023

[2] Goldfarb, A.: What is Different about Online Advertising? Review of Industrial Organization **44**(2), 115–129 (2014)

[3] Google: Google Ads. https://support.google.com/google-ads/answer/142918. Accessed 9 March 2023

[4] Microsoft: Microsoft Advertising. https://ads.microsoft.com. Accessed 9 March 2023

[5] Boskamp, V., Knoops, A., Frasincar, F., Gabor, A.: Maximizing Revenue with Allocation of Multiple Advertisements on a Web Banner. Computers & Operations Research **38**(10), 1412–1424 (2011)

[6] Karp, R.M.: Reducibility among Combinatorial Problems. In: Complexity of Computer Computations. The IBM Research Symposia Series, pp. 85–103 (1972). Springer

[7] Manik, P., Gupta, A., Jha, P., Govindan, K.: A Goal Programming Model for Selection and Scheduling of Advertisements on Online News Media. Asia-Pacific Journal of Operational Research **33**(02), 1650012 (2016)

[8] Malthouse, E.C., Hessary, Y.K., Vakeel, K.A., Burke, R., Fudurić, M.: An Algorithm for Allocating Sponsored Recommendations and Content: Unifying Programmatic Advertising and Recommender Systems. Journal of Advertising **48**(4), 366–379 (2019)

[9] Kim, G., Moon, I.: Online Banner Advertisement Scheduling for Advertising Effectiveness. Computers & Industrial Engineering **140**, 106226 (2020)

[10] Dayanik, S., Parlar, M.: Dynamic Bidding Strategies in Search-Based Advertising. Annals of Operations Research **211**(1), 103–136 (2013)

[11] Rodríguez, I., Rubio, F., Rabanal, P.: Automatic Media Planning: Optimal Advertisement Placement Problems. In: 2016 IEEE Congress on Evolutionary Computation (CEC 2016), pp. 5170–5177 (2016). IEEE

[12] Yang, H., Wang, T., Tang, X., Yu, H., Liu, F., Song, H.: Dynamically Optimizing Display Advertising Profits under Diverse Budget Settings. IEEE Transactions on Knowledge and Data Engineering **35**(1), 362–376 (2021)

[13] Wojciechowski, A., Kapral, D.: Allocation of Multiple Advertisement on Limited Space: Heuristic Approach. In: Future Multimedia Networking, pp. 230–235 (2009). Springer

[14] Kaul, A., Aggarwal, S., Gupta, A., Dayama, N., Krishnamoorthy, M., Jha, P.: Optimal Advertising on a Two-dimensional Web Banner. International Journal of System Assurance Engineering and Management **9**(1), 306–311 (2018)

[15] Dyckhoff, H.: A Typology of Cutting and Packing Problems. European Journal of Operational Research **44**(2), 145–159 (1990)

[16] Wäscher, G., Haußner, H., Schumann, H.: An Improved Typology of Cutting and Packing Problems. European Journal of Operational Research **183**(3), 1109–1130 (2007)

[17] Caprara, A., Monaci, M.: On the Two-Dimensional Knapsack Problem. Operations Research Letters **32**(1), 5–14 (2004)

[18] Hadjiconstantinou, E., Christofides, N.: An Exact Algorithm for General, Orthogonal, Two-Dimensional Knapsack Problems. European Journal of Operational Research **83**(1), 39–56 (1995)

[19] Lodi, A., Monaci, M.: Integer Linear Programming Models for 2-Staged Two-Dimensional Knapsack Problems. Mathematical Programming **94**(2-3), 257–278 (2003)

[20] Beasley, J.E.: An Exact Two-Dimensional Non-Guillotine Cutting Tree Search Procedure. Operations Research **33**(1), 49–64 (1985)

[21] Bortfeldt, A., Winter, T.: A Genetic Algorithm for the Two-Dimensional Knapsack Problem with Rectangular Pieces. International Transactions in Operational Research **16**(6), 685–713 (2009)

[22] Egeblad, J., Pisinger, D.: Heuristic Approaches for the Two- and Three-Dimensional Knapsack Packing Problem. Computers & Operations Research **36**(4), 1026–1049 (2009)

[23] Alvarez-Valdes, R., Parreño, F., Tamarit, J.M.: A Tabu Search Algorithm for a Two-Dimensional Non-Guillotine Cutting Problem. European Journal of Operational Research **183**(3), 1167–1182 (2007)

[24] Benavides, A., Machado, M.S., Costa, A.M., Ritt, M., Buriol, L., Garcia, V., Franca, P.: A Comparison of Tabu Search and GRASP for the Switch Allocation Problem. 41st Brazilian Operational Research Symposium (SBPO 2009) (2009)

[25] Rajkumar, M., Asokan, P., Vamsikrishna, V.: A GRASP Algorithm for Flexible Job-shop Scheduling with Maintenance Constraints. International Journal of Production Research **48**(22), 6821–6836 (2010)

[26] Rajkumar, M., Asokan, P., Anilkumar, N., Page, T.: A GRASP Algorithm for Flexible Job-shop Scheduling Problem with Limited Resource Constraints. International Journal of Production Research **49**(8), 2409–2423 (2011)

[27] Shaikh, J., Fiedler, M., Collange, D.: Quality of Experience from User and Network Perspectives. Annals of Telecommunications **65**(1-2), 47–57 (2010)

[28] Knoops, A., Boskamp, V., Wojciechowski, A., Frasincar, F.: Single Pattern Generating Heuristics for Pixel Advertisements. In: 10th International Conference on Web Information Systems Engineering (WISE 2009), pp. 415–428 (2009). Springer

[29] Alvarez-Valdes, R., Parreño, F., Tamarit, J.M.: A GRASP Algorithm for Constrained Two-Dimensional Non-Guillotine Cutting Problems. Journal of the Operational Research Society **56**(4), 414–425 (2005)

[30] Feo, T.A., Resende, M.G.C.: Greedy Randomized Adaptive Search Procedures. Journal of Global Optimization **6**(2), 109–133 (1995)

[31] Nielsen, J.: Horizontal Attention Leans Left. http://www.nngroup.com/articles/horizontal-attention-leans-left/. NN Group (2010)

[32] Nielsen, J.: Scrolling and Attention. http://www.nngroup.com/articles/scrolling-and-attention/. NN Group (2010)

[33] Roels, G., Fridgeirsdottir, K.: Dynamic Revenue Management for Online Display Advertising. Journal of Revenue and Pricing Management **8**(5), 452–466 (2009)

[34] Valério de Carvalho, J.M.: Exact Solution of Bin-packing Problems Using Column Generation and Branch-and-Bound. Annals of Operations Research **86**, 629–659 (1999)

[35] Menon, S., Amiri, A.: Scheduling Banner Advertisements on the Web. INFORMS Journal on Computing **16**(1), 95–105 (2004)

[36] Kumar, S., Jacob, V.S., Sriskandarajah, C.: Scheduling Advertisements on a Web Page to Maximize Revenue. European Journal of Operational Research **173**(3), 1067–1089 (2006)