Poznań University of Technology
Institute of Computing Science

Combinatorial optimization problems in Internet applications

Doctoral thesis

Jakub Marszałkowski

Supervisor:
prof. dr hab. inż. Maciej Drozdowski

Poznań, 2017

Contents

1 Introduction **4**
1.1 Motivation 4
1.2 Scope and Puropose 5
1.3 Methodology 6
1.4 Common webpage-related factors 10
1.5 Outline of the Thesis 11

2 Layout Partitioning for Advertisements Fit **13**
2.1 Website's Layouts and Ad Placement 13
2.2 Problem Formulation 16
2.3 Objective Functions 19
2.3.1 Max Ad Number Function 20
2.3.2 Max Most Difficult to Pack Ad Unit Function 20
2.3.3 Min Single Ad Waste 20
2.4 Solution Method 21
2.4.1 Combining Ad Units 22
2.4.2 Valid Column Widths List 23
2.4.3 Browsing Layouts 24
2.4.4 Selecting Final Results 25
2.4.5 Example For a Small Instance 25
2.5 Benchmarks 27
2.5.1 Data Sets 27
2.5.2 Webmaster Survey 27
2.6 Computational Experiments 29
2.6.1 Input Parameters 29
2.6.2 Execution Times 31
2.6.3 Layout Partitioning Results and Discussion 31
2.7 Conclusions 35

3 Tag Cloud Construction **37**
3.1 Tag Clouds 37
3.2 Problem Analysis and Related Work Survey 38
3.2.1 Tag cloud taxonomy 38
3.2.2 Related work 40
3.2.3 Tag Cloud Usability Studies 42
3.2.4 Tag Clouds for the Web 43
3.2.5 Client Side 44
3.2.6 Analysis of Packing Problem Properties 47
3.3 Problem Formulation 48
3.4 Algorithms for Tag Cloud Optimization 50
3.4.1 Branch and Bound 51
3.4.2 Greedy Algorithms 52
3.4.3 Tabu Search 54
3.5 Computational Experiments 55
3.5.1 Test Instances 55
3.5.2 Selection of the Objective Function 56
3.5.3 Super-Fit Algorithm 57
3.5.4 Tuning Tabu Search 58
3.5.5 Branch and Bound 61
3.5.6 Comparison of the Algorithms 62
3.6 Conclusions and Future Work 65

4 CSS-sprite Packing **66**
4.1 CSS-Sprites and Loading of Web Pages 66
4.2 Practical Challenges and Problem Formulation 68
4.2.1 Geometric Challenges 68
4.2.2 Image Compression Properties 70
4.2.3 Computational Complexity 72
4.2.4 Communication Performance 76
4.2.5 Problem Formulation 79
4.3 Preliminary Tests 79
4.3.1 Packing Model 79
4.3.2 Communication Performance 82
4.4 State of the Art 86
4.5 Spritepack 91
4.5.1 Tile Classification 91
4.5.2 Geometric Packing 92
4.5.3 Merging with Image Compression 95
4.5.4 Postprocessing 96
4.6 Spritepack Evaluation 97
4.6.1 Test Instances 97
4.6.2 Initial Experiments 100
4.6.3 Spritepack Performance Comparison 106
4.6.4 End-to-End Evaluation 108
4.7 Conclusions 110

5 Summary and Final Remarks **112**

1 Introduction

1.1 Motivation

The Internet has become a key element of human civilization. It is a cornerstone of communication, trade, science, social life, entertainment, etc. To realize how much Internet changed our life's one should refer to pre-internet works SF writers and futurologist. They, in general, failed to foresee the global net and are drawing our times or even further future without the common functionalities that we have already.

Over the last two decades, World Wide Web has become a new branch of industry. The subject of engineering in designing, constructing, maintaining, improving and optimizing various web services or Internet applications and their components is an art and craft, business, scientific and engineering challenge. Engineering endeavors related to the Internet are often referred to as web engineering. However, it is hard to refrain from observing that some components of the web are constructed in an ad-hoc way. Web engineering should not be exempted from the analysis and optimization of its processes. Hence, web engineering should profit from many decades of scientific experience, including, combinatorial optimization and operations research heritage analyzing and optimizing industrial processes. And vice versa, the combinatorial optimization society should not omit this new and emerging field of applications.

This leads to this thesis motivation of conducting research in the area of web engineering and combinatorial optimization. In the next section three problems which are subject of this thesis/work/manuscript, will be outlined.

However, it should be also noted that the author conducted more research in the fields of web applications, web services and Internet advertising that went beyond the scope of the thesis. Instead, conclusions and results from these papers will be referred to as needed. This includes the paper on effectiveness of ads server [84], the paper on advertisement exchange [83], two papers on

database optimization for web services [88, 97], the paper on the role of website speed [87], the paper on Internet shopping optimization [85] and the paper on building a web platform for gamification of education [123].

1.2 Scope and Puropose

The recent astounding growth of the Internet and its applications is bringing new interesting research field to combinatorial optimization. Although Internet-related optimization problems are very novel, several elements remain the same: there is maximization of the revenue or capability, minimization of costs or resource used, etc. Often similar approaches, after specific adjustments, can be used. The challenges of computational complexity are the same. Problems are computationally hard in their nature and often requiring super fast algorithms for online results. Three specific problems have been chosen as the subject of the thesis:

I Layout Partitioning for Advertisement Fit (*LPfAF*) where a website is partitioned into columns to obtain layout capable of fitting advertisements well.

II Tag Cloud Construction Problem (*TCCP*) which consists in optimizing tag clouds for web pages to provide good readability.

III CSS-sprite Packing Problem (*CSS-SPP*) where, by bundling web page images into packages called sprites, shorter load time of web pages is achieved.

The intersection of these research problems is twofold. Firstly, all three problems are real-world challenges taken by author from his practical experience with the Internet applications. Thus, solving them can lead to improving the quality of the web pages. LPfAF can improve ads revenues but also usability of web pages. TCCP can improve readability and thus ergonomics of web applications. CSS-SPP can offload servers and network, speed up page loading and reduce the memory usage by browsers, which already is a good justification for the research.

Secondly, all three problems relay heavily on two-dimensional rectangle packing. Note that all elements of a website are actually rectangles. Regardless of their visible shapes the placement done by the browser in Document Object Model (DOM) is putting rectangular elements into rectangular space. Advertisements and slots to insert the ads are rectangular. Images might use opacity to hide from users sight parts of the rectangle, but are placed the same way. Any part of the text, would be it a single keyword or three paragraphs, are placed

with respect to their rectangular envelope. With that, several new approaches the old 2-dimensional packing problems and algorithms are developed, which is another good justification for the research.

1.3 Methodology

In this section methodologies shared in the analysis of the above three problems will be outlined. It is assumed that the reader has basic knowledge of computational complexity theory, combinatorial optimization, scheduling theory, 2-dimensional packing. In order to avoid repeating basic facts of computer science, we do not elaborate on these theories here. An interested reader may find introduction to these fields in [13, 14, 15, 36, 47, 64, 77]. Instead, we give here only a short informal outline and explanation of the guidelines used in the three problems. The guidelines define a pattern of the chapters dealing with the particular problems.

As all the three problems are real-world problems, requiring the same steps in the work with them. The common analysis procedure/pattern is as follows:

1. Literature study for state of the art in this or similar problems.
2. Mathematical modeling of the problem.
3. Analysis of the computational complexity of the problem.
4. Development of algorithms to solve the problem.
5. Validation of the algorithms.

The actual research work in these steps may vary with different problems, especially with the respect to the algorithmic approaches. However, their common factors can be discussed here.

As the research consists of three separate problems, the literature search (step 1) was performed for them separately, and the results will be presented further in the text, with only short outline here. Tag cloud construction, optimization or visualization have met some interest of researchers. To the best of the author's knowledge, the first research paper on that topic was [65]. This was continued and extended leading to many other interesting solutions, with probably Wordle [119] being the most recognized. Sadly Wordle does not offer any optimization for use on web pages motivating the author to research in this field. A survey of publications on tag clouds construction is provided in Section 3.2.1.

Although CSS-sprite optimization is a kind of popular topic, was up to date aimed rather at projects using engineering approach than scientific methods. Thus, rarely was there any optimization problem considered, and even if, then it was not going further than application of 2-dimensional packing algorithms to reduce dimensions of the sprite. Survey of projects building CSS-sprites is provided in Section 4.4.

The idea for research on LPfAF comes from the literature on ad placement optimization. Many of such problems have already been considered in the context of ad display placement, optimization or scheduling. This area of research was first proposed in [3] and [39] and up to now proliferated in so many directions, that it definitely deserves a separate review paper. However, all this research usually is considering optimization of revenues from ads that are placed in static ad spaces or slots that are given a priori. This leads to the conclusion that optimization of space for ads might be worth consideration. As the original problem of ad placement optimization for revenue maximization is mainly a motivation for the authors own project no wider survey on the original area was conducted as it was lying out of scope of this work.

All three problems considered in this thesis are to some point novel and required providing a mathematical model built from scratch (step 2). Even for the TCCP dealing with tag clouds that were tackled in numerous research papers, the author has taken completely different novel approach. In all cases, this required modeling objective functions to represent realities of the problem. This is a difficult task. In two of the problems, LPfAF and TCCP, the difficulty was catching some perception of aesthetics with mathematical equations. Whether the objective functions were modeled properly can be verified not earlier than by studying the results of the designed algorithms. While the obtained results seem to confirm a successful outcome, it will be further discussed in the following chapters.

Problems considered in this thesis, are combinatorial optimization problems and consideration of their computational complexity had to be performed (step 3). Problems of combinatorial optimization are defined on sets of discrete objects rather than on continuous domains. Problems of this nature are very often hard to solve. The only approach to give optimal solution, or often any solution is by verifying a number of solutions, which grows exponentially with the problem size (unless $\mathcal{P} = \mathcal{NP}$). Problems of this nature are called $\mathcal{NP}$-hard (see [47, 64] for precise definition). 2-dimensional packing problems are $\mathcal{NP}$-hard in general. Because all the considered problems are generalizations of known 2-dimensional packing problems, or have 2-dimensional packing as one of the sub problems, all the three problems are $\mathcal{NP}$-hard too. It is practically impossible to solve such problems by full enumeration of possible solutions. Such

approaches are called exact and are effective only for small instances of problems. Algorithms that are effective in the greater range of instances are the ones which execution times can be bounded by polynomials in the input size. These are called polynomial-time algorithms. Unfortunately, unless $\mathcal{P} = \mathcal{NP}$ we must expect that for hard combinatorial problems no effective polynomial-time algorithms can be given.

The theory of computational complexity provides a methodology of dealing with hard combinatorial problems. Several groups of algorithms have been developed to solve them (step 3):

1. Full enumeration algorithms, with main feature being searching the space of possible solutions exhaustively. It means that in the worst case all possible solutions may be visited. Their complexity may not go below exponential and execution times are prohibitively high, which limits their applicability.

2. Algorithms providing efficient search for exponential solution space can be constructed with the use of methods for pruning the solutions that are not promising. In two of three research problems such approach was used. Branch&Bound (*B&B*) algorithms [76] represent the space of solutions as a search tree, pruned by bounds on values of the solutions that a given branch might lead to. Still the size of the instance that can be solved to optimality even by best exact algorithms is limited because B&B algorithms are also worst-case exponential running time algorithms. Thus, dedicated B&B algorithm that was developed in one of the research problems was used only to solve small instances of the problems. For another of the problem, dedicated for the specific problem exact algorithm known from literature was adapted, while it solves instances of required size in acceptable time.

3. Dedicated (or tailored) heuristics are the methods, created to solve just one given combinatorial optimization problem. These run in polynomial time (making them effective) and provide correct solutions, yet without any guarantee of optimality. This can mean that, e.g., the solution given by a heuristic will be just the first feasible solution found or a local optimum of the objective function closing or even sometimes reaching the global optimum. Dedicated heuristic algorithms are using some specific characteristic of the problems they are solving, and thus it is not possible to apply them to different problems. Still, many such heuristics exist or can be designed, making them very commonly applied in solving combinatorial optimization problems [13, 36, 47]. This approach was used in all three researched problems.

4. Metaheuristics are general-purpose methods that can be adapted to many different combinatorial optimization problems. Metaheuristics provide a general framework on how the space of combinatorial problems can be searched. As they are general in their nature, they must be adapted to every actual combinatorial optimization problem. Metaheuristics include, for example tabu search, simulated annealing, genetic algorithms, a family of swarm systems and many others [1, 23, 42, 49, 50, 93, 95]. Two of the three research problems were approached with metaheuristics.aheuristics.

For all the discussed algorithms a trade-off between time complexity and quality of results must be considered in general. Usually the better results an algorithm offers, the more time it consumes. In an online environment like in TCCP the execution time is absolutely a crucial factor. For problems that are solved infrequently like in LPfAF, and in most uses of CSS-SPP, the quality of solutions would be far more important than run time. Algorithm execution time may be studied in various ways starting with the analytical approaches. For this purpose, the order of the computational complexity of the algorithm is analyzed. Informally, computational complexity function of an algorithm is the execution time in the size of the input [47]. For the majority of the algorithms developed in this thesis, the computational complexity function is analyzed and given. For more complex algorithms execution time is often verified experimentally, where the specificity of the solved problem defines a threshold of runtimes bearable for the user.

The quality of the results provided by the algorithms can be assessed in many different ways (step 5). Simpler heuristics are often analyzed to find and prove guarantees of solution quality, e.g., the ratio or the distance between the solution provided by the heuristic and the optimum objective value. This rarely is possible for algorithms solving practical problems due to their to the problem complication. It can be also observed, that quality of results that the algorithm offers typically is more important than the guarantees of quality in the worst case that almost never happens. In many cases the average distance of the solution from the optimum can be measured experimentally. For this purpose the algorithm is tested on benchmark instances that are solved also by other algorithms. This might be a comparison with exact solutions, where such solutions are known. However, due to the nature of computationally hard problems this will often require considering benchmark instances being smaller than the practical ones. Another option is to compare algorithms to other algorithms solving the same problem. Still, algorithms often may be better on

certain test cases while worse on many others. In the result, such experimental analysis may be inconclusive when the criteria of the comparison are not strictly defined.

1.4 Common webpage-related factors

As it was mentioned web engineering and development of web applications is a new branch of industry offering a host of operational research and optimization problems. A short summary of the fields of research other than the ones considered in this thesis will be given here. One of the first considered areas were Internet advertisements. Optimization of ad networks choosing ads on the basis of price, web page content, keywords related to the web page, behavioral and demographic targeting was analyzed in [75]. The first model of revenue maximization, named "side banners" was proposed in [3, 73], and extended in [5, 39, 40, 72]. Website layout or content optimization was considered in [107, 127], content analysis and fast delivery in [71, 90], while techniques for content interpretation and exploitation [113]. Finally, novel e-business applications are developed, like Internet shopping with optimization were proposed in [126] and then extended in [10, 12, 11, 80]. This by no means is a complete list of problems, rather exaples to show variety of the research on optimization problems related with Internet deserve a separate survey.

Numerous factors are common for many research problems from the area of web applications. Three examples that were shared between research problems in this thesis will be analyzed here.

Websites all over the Internet use *vertical scroll layout* [57], also known as column or grid layout. This means that there is no height limitation for content as the page can be scrolled down. Contrarily, width of the website is usually limited by the design to display well on most of the clients (browsers). More so, websites are divided into columns that are supposed to fit the content. Hence, the web pages or columns can be considered infinite stripes of given width. This will affect the research on LPfAF and TCCP where assumption of limited and exactly given width and unlimited, stretchable or scrollable height will be used.

Great deal of factors on web application design and operations depends on the client side. Thus, another problem with the research targeting web pages optimization is high fragmentation of the parameters of the client devices. It is possible to predict these parameters, at least some ranges of them, but only to some resolution. This was in a way affecting all three problems. A web designer willing to design a layout for the website, no matter whether using the solution proposed as LPfAF-solving algorithms or doing it manually with an

ad hoc approach, must consider different screen resolutions at client browsers limiting width of the layout. The optimization done by CSS-SPP must consider parameters of the Internet connections of the website user base of the website. The basic parameters of communication performance: latency, bandwidth and speedup in parallel communication will vary from website to website but also among users of the same website. This is requiring averaging or maybe clustering the groups of users into separate optimized solutions. Then, TCCP goes further with respect to the client side dependability. The sizes of tags will differ in client browsers, thus can be measured only there and thus the optimization must be performed with the use of that data.

The time a web page uses to load plays an important role. Research from 2006 [24] suggested that there is a four seconds rule, i.e. if the page loads longer than 4 seconds, then there is a good chance that a customer will direct his/her browser elsewhere. In a newer research by the Amazon experiments, every 100ms delay in page load time decreases the sales by 1 percent [103]. Similar measurements at Google showed that 500ms delay in search result display reduces revenue by 20 percent [20]. Author's own research presented in [87] was suggesting that as for 2014 Google was treating only times below 1.5 second as acceptable. More direct results of paper [87] showed that web page speed affects position in search engine results and thus also traffic and income. This speed is the factor that CSS-SPP is trying to improve, while TCCP must consider in the algorithm design process.

The list of such factors is far greater and more specific ones will be discussed further in the sections dedicated to the three research problems.

1.5 Outline of the Thesis

The rest of this thesis is organized in three chapters dedicated to each of the research problems and closed with Chapter 5 containing final summary and remarks. As the problems are different and have disjointed formulations three dedicated sets of notations will be used, that are summarized in Table 2.1 for LPfAF, Table 3.3 for TCCP and Table 4.1 for CSS-SPP.

Chapter 2 is dedicated to the problem of Layout Partitioning for Advertisement Fit. First, in Section 2.2 the LPfAF problem is formulated. Optimization criteria are discussed in Section 2.3 . Section 2.4 contains the algorithms designed for solving the problem. Benchmark datasets are presented and explained in Section 2.5. Outlines of the solutions obtained by presented method are given in Section 2.6. The last section comprises conclusions and discussion of possible future research, extensions of the model and similar problems.

Chapter 3 presents research on Tag Cloud Construction Problem. Tags and tag clouds are discussed in Section 3.2 including a survey of approaches, algorithms, design options and the choices taken in the past. This is followed by a discussion of requirements for tag clouds in the web usage. Section 3.3 provides a mathematical formulation of the Tag Cloud Construction Problem (TCCP). Algorithms solving the problem are presented in Section 3.4. In Section 3.5 results of the computational experiments are outlined. Finally, the last section summarizes the achievements in tag cloud construction.

Chapter 4 contains considerations on CSS-sprite Packing Problem. The first section, was dedicated to discussion on the realities and the challenges in sprite packing, followed by formulation of CSS-sprite Packing Problem (*CSS-SPP*). Results of preliminary empirical studies on properties of the problem are presented in Section 4.3. An extensive survey of the current advances in CSS-Sprite construction is given in Section 4.4. Holistic new method of sprite packing is given in Section 4.5 and evaluated in Section 4.6. The last section is dedicated to conclusions.

2 Layout Partitioning for Advertisements Fit

2.1 Website's Layouts and Ad Placement

In this section a problem of dividing a websites visual area into columns for future placement of advertisements will be analyzed. Internet advertising is a basic source of income on the web [32]. According to the Interactive Advertising Bureau (IAB), Internet advertising is worth €14.7 billion a year in Europe only [60]. Internet advertising is usually divided into three markets: search advertising, display advertising, and classifieds & directories. Display advertising, also called banner advertising, is the second largest online advertising market. Its European share was worth €4.4 billion in 2009 despite the economic crisis at that time [60]. Optimizing page layouts for this important part of the current industry is considered here.

In the contemporary display advertising, publishers, i.e., owners of the web sites, do not contact advertisers directly. Publishers usually join advertising networks which serve as middlemen between the advertisers and the publishers. A publisher has to prepare place on the website for fitting the ads provided by the ad networks. With each new exposition of a web page, ads are fetched from the ad network and displayed in the place provided by the publisher. The ad network chooses ads on the basis of price, web page content, keywords related to the web page, behavioral and demographic targeting [75]. An *ad unit* is a form of an agreement on the ad format between the advertiser, the publisher, and the ad network. Definition of an ad unit consists of its width and height measured in pixels. Advertisers contribute ads of the given size to the ad networks. Ad networks provide servers and code for fetching the ads. Publishers place ad units, i.e. prepare free space of the ad unit size on their web pages, and include

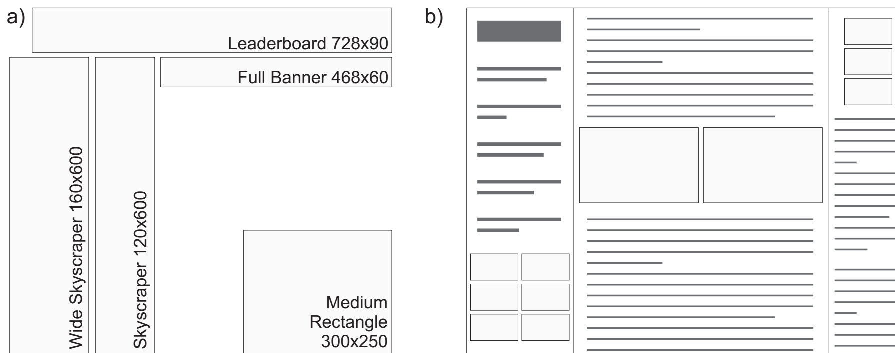

Figure 2.1: a) Five popular ad units. b) Three column layout, and ad combinations.

there the ad fetching code. Five example ad units are visualized in Fig. 2.1a. Display advertising uses many different ad units. Despite obvious advantages of unification, hundreds of ad units are in use. Benchmark ad unit datasets will be introduced in Section 2.5.

A publisher faces a problem of preparing a website such that it is commodious for diverse ad units, and aesthetically pleasing. As it was mentioned in Section 1 websites use *vertical scroll* layout with given width but scrollable down without limit. Currently, layouts comprise two, three (cf. Fig. 2.1b), or four columns. Because height for packing ads and other content is not limited as it was explained earlier, columns will be treated here as virtually infinite stripes. *Layout partitioning*, i.e. dividing a website into columns, is a critical decision with long-term consequences, because all graphical elements are adjusted to column sizes. After deciding on the number of columns, a publisher has to select column widths. The state of the art recommendations for choosing column widths are basic rules of art adapted to websites: golden ratio, rule of thirds, symmetrical or asymmetrical balance [9]. In practice websites often end up with the so-called “bread and butter design”. “Bread” is a wide column for content, and “butter” a narrow column for navigation [9]. The results that are usually achieved can be described as layouts without obvious errors (LWOE) [57]. The fact that a layman at first glance cannot suggest easy improvement is considered a sign of satisfactory layout. The above methods do not consider ad placement. In practice, layout partitioning is performed *ad hoc* with limited consideration of fitting ads. The widest column is chosen to fit the widest expected advertisement, while the remaining space is divided arbitrarily. This meets LWOE criteria, but is probably far from what could be achieved, and causes problems with three and four column layouts. If the remaining space

were again divided to fit the widest possible ad unit, then the other columns would be too narrow for anything. Alternatively, dividing the remaining space to fit the narrowest ad unit would result in a column for nothing else but that unit. The recommendations mentioned above are a few quite arbitrary choices from a wide spectrum of possibilities offering layouts of diverse quality. Hence, layout partitioning for fitting ads requires a more rudimentary study.

In this section layout partitioning for advertisement fit (*LPfAF*) will be formulated as a combinatorial optimization problem. This requires formulating constraints, and optimization criteria. Constructing a good ad placement requires grasping aesthetic aspects in a formal way, which is always difficult. However, an attempt to optimize ad placement, and limit bad-looking ad combinations will be made. Ad placement optimization has already been considered in the context of ad display scheduling. A model of "side banners" was proposed in [3, 73], and extended in [5, 39, 40, 72]. This model assumes that advertisements are packed in a side column, while column sizes are given. When this model is generalized to multicolumn ad placement, column sizes must be carefully selected, because ads may be put in any column, and ad scheduling is affected by layout partitioning. Though LPfAF problem is related to ad display scheduling, it is not per se a scheduling problem because there is no time dimension here. By a far analogy, LPfAF can be compared to factory layout optimization. To the author's best knowledge, column width selection for advertisement fit has never been considered before. Advertisement placement will be assumed in a way which is similar to guillotine cutting. Though it is possible to position ads in a different way, guillotine-cut space partition is easier for the page-building scripts. The operation of cutting a rectangle into two can be mimicked either with `<TABLE>` tag, or with `<DIV>` tag. There is a slight difference between ad placement and the guillotine cutting. For aesthetic reasons ads are not supposed to touch each other. Hence, spacing is added around the advertisements. It is achieved with the `padding` parameter, both in `<TABLE>` and `<DIV>` tags.

The rest of this chapter is organized as follows. In Section 2.2 the LPfAF problem is formulated. In Section 2.3 optimization criteria are taken into consideration. Section 2.4 is dedicated to the algorithms solving the problem. In Section 2.5 benchmark datasets are introduced. Outlines of the solutions obtained by presented method are given in Section 2.6. The last section comprises conclusions. Table 2.1 summarizes the notation.

Symbol	Definition
	parameters
a	number of columns in a layout
b_e	how many times ad unit r_e can be used in any combination
m_p	minimal width of column c_p, $p \in \{1, \dots, a\}$
n	number of ad units in the dataset
$R = \{r_1, \dots, r_n\}$	set of rectangular advertising units
t, t'	limits on the number of ads units, resp. ads, in any combination
W	page width
$w_e \times h_e$	dimensions of ad unit $r_e, e \in \{1, \dots, n\}$
X	set of ad units to be used separately
β	amount of space waste acceptable for a combination
	decision variables
c_p	width of column p, $p \in \{1, \dots, a\}$
	intermediate variables
d	number of admissible ad combinations
F	set of all possible layouts
$G_z = (c_{1z} \dots, c_{az})$	z-th layout is a vector of a column widths, $G_z \in F$
$J(c_p)$	set of ad combinations fitting in a column of width c_p, $c_p \in Y$
$K = \{K_1, \dots, K_d\}$	set of ad combinations
k_{ge}	the number of times ad unit r_e appears in the combination K_g
$K_g = (k_{g1}, \dots, k_{gn})$	vector of ad unit multiplicities in a combination, $g \in \{1, \dots, d\}$
s	number of column widths worth evaluation
u	maximum column width
V_1, V_2, V_3	objective functions
$w'_g \times h'_g$	dimensions of combination $K_g, g \in \{1, \dots, d\}$
$Y = \{y_1, \dots, y_s\}$	set of feasible column widths

Table 2.1: Summary of notation for the Layout Partitioning for Advertisement Fit problem.

2.2 Problem Formulation

In this section formulation of the LPfAF problem is presented. Let $R = \{r_1, \dots, r_n\}$ be a set of rectangular *ad units*. Ad unit r_e has dimensions $w_e \times h_e$. All dimensions are natural numbers as they represent screen pixels. We will denote by W the width of a website. As websites use vertical scroll layouts, the height is not a constraint here because advertisements can be placed one above another (cf. Fig. 2.2b). A website is divided into $a \in \{2, 3, 4, \dots\}$ columns of widths $c_1 \dots, c_a$. A *layout* is constituted by the vector $G_z = (c_{1z}, \dots, c_{az})$ of column widths. Column widths are the decision variables. Each column p can have a minimum width m_p imposed by the webmaster. If not provided, the narrowest ad unit width $\min_{1 \le e \le n}\{w_e\}$ should be used. Column widths are subject to the constraints:

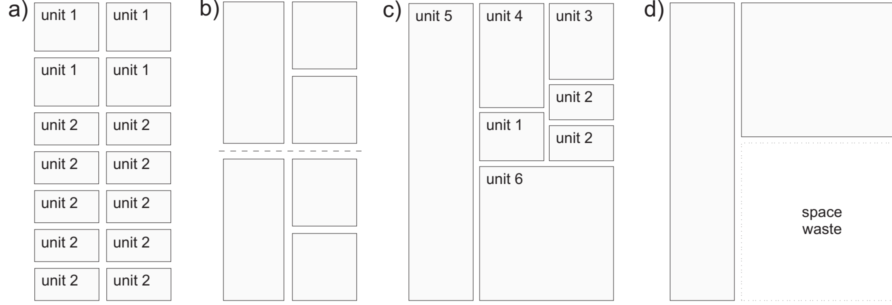

Figure 2.2: Example ad units combinations.

Constraint 1: The sum of column widths cannot extend the page width:

$$\sum_{p=1}^{a} c_p \leq W. \tag{2.1}$$

Constraint 2: To leave enough space for each column, the widest column cannot be wider than:

$$u = W - \sum_{p=1}^{a} m_p + \max_{1 \leq p \leq a} \{m_p\}. \tag{2.2}$$

By joining rectangular ad units vertically or horizontally *ad combinations* $K = \{K_1, K_2, \ldots, K_d\}$ are created. Examples of good and bad combinations are shown in Fig. 2.2. The combinations in Fig. 2.2a,b are advantageous because they are easily implemented. The combination in Fig. 2.2c is difficult to implement, and aesthetically hard to accept because it looks like a patchwork. The combination in Fig.2.2d wastes space. Each combination is represented by vector $K_g = (k_{g1}, k_{g2}, \ldots, k_{gn})$ where component k_{ge} denotes how many times ad unit r_e appears in combination K_g. The rectangular envelope of ad combination K_g has dimensions $w'_g \times h'_g$. Since set K of ad unit combinations is limited, it implicitly follows that also the set of column widths worth consideration is limited.

Ad combinations are subject to constraints mostly of aesthetic nature:
Constraint 3: No combination wider than the maximum column width could be ever placed, so for each combination K_g: $w'_g \leq u$.
Constraint 4: As mentioned earlier, two combinations can always be placed one under the other (cf. Fig. 2.2b). However, this does not change fitting capability. To avoid redundancy in combinations, the height of each combination K_g is limited to the highest ad unit: $h'_g \leq \max_{1 \leq e \leq n}\{h_e\}$.

Constraint 5: All ad units have limits $b_1, \dots, b_n$ on the number of occurrences in a combination, i.e. $k_{ge} \leq b_e$, for combination K_g and ad unit r_e. A lack of such limits, or too large limits, would lead to over-representation of the tiniest well combining ad units in the results. Such a situation will be discussed in Section 2.6.1.
Constraint 6: Certain ad units are designed for separate use, e.g., in headers, footers, or pop up windows. Such ad units can appear only as a singleton ad unit combination. Let X be a set of ad units designed for the sole use. This requests that $\forall r_e \in X$, a combination $K_g = (k_{g1}, \dots, k_{gn})$ exists such that $k_{g1} = \dots = k_{ge-1} = k_{ge+1} = \dots = k_{gn} = 0, k_{ge} = 1$.
Constraint 7: No combination can have more than t different ad units:

$$\sum_{e=1}^{n} \min\{1, k_{ge}\} \leq t. \tag{2.7}$$

This constraint is supposed to eliminate patchwork-like combinations as the one in Fig. 2.2c. It has only 7 ads, but 6 different ad units. Conversely, combination in Fig. 2.2a has 14 ads but only 2 ad units. It looks better and can be used in a narrow column.
Constraint 8: The number of ads cannot be excessive. This leads to the limit on the total number of ads in the entire combination K_g:

$$\sum_{e=1}^{n} k_{ge} \leq \min\{t', t \max_{1 \leq e \leq n} \{b_e\}\}, \tag{2.8}$$

where t' is an independent constraint value imposing stricter limit on the number of ads than it results from the earlier b_e and t parameters.
Constraint 9: To exclude patchwork-like combinations, a limit of no more than either one vertical or one horizontal join is imposed. This allows for combinations that consist of up to two columns of ads (e.g. Fig. 2.2a) or two rows of ads, and eliminates combinations like the one in Fig. 2.2c with two horizontal and four vertical joins. Let ver_g, hor_g be the number of vertical and horizontal joins performed to obtain combination K_g. Let K_g be obtained by joining combinations K'_g, K''_g. For horizontal join of K'_g, K''_g, $ver_g = \max\{ver_{g'}, ver_{g''}\}, hor_g = hor_{g'} + hor_{g''} + 1$. For vertical join of K'_g, K''_g, $hor_g = \max\{hor_{g'}, hor_{g''}\}, ver_g = ver_{g'} + ver_{g''} + 1$. Mathematically, this constraint can be expressed as: $\min\{ver_g, hor_g\} \leq 1$.
Constraint 10: Combinations wasting space, like the one in Fig. 2.2d, should be avoided. Hence, parameter β, $0 < \beta \leq 1$ is introduced to limit space waste.

Discarded are combinations satisfying:

$$\beta < 1 - (\sum_{e=1}^{n} k_{ge} w_e h_e)/(w'_g h'_g). \tag{2.10}$$

Additionally, when creating combinations, ad units are subject to padding. Empty space of size σ is created around each ad unit. This can be achieved by adding σ to w_e and h_e at the beginning of the solution process.

Combinations of ads are placed in columns of a layout as shown in Fig. 2.1b. For each admissible column width c_p a set of ad combinations $J(c_p)$ that fit in width c_p can be calculated. Column widths c_p, and combinations $K_g \in J(c_p)$, must meet the following requirements:
Constraint 11: By definition, an ad combination fits a column if it is not wider than the column width: $w'_g \leq c_p$.
Constraint 12: The ad combination should be wider than half of the column width, i.e. $w'_g > \frac{1}{2}c_p$. Otherwise, a larger combination created by joining horizontally two such combinations should fit the column to avoid space waste. See the combinations in the left and right columns in Fig. 2.1b.
Constraint 13: Every ad unit $r_1, \ldots, r_n$ must fit in at least one column of a website, i.e., $\max_{1 \leq p \leq a}\{c_p\} \geq w_e$, for $e = 1, \ldots, n$.

2.3 Objective Functions

In this section objective functions are discussed. LPfAF problem is inherently multicriterial. There are no standard measures of page layout quality. However, in the discussions with webmasters three main aspects of layout partitioning quality for advertisement fit were introduced (cf. Section 2.5.2):

1. A layout should be flexible to accommodate different ad units and their display organizations.
2. Ad units that are unwieldy should not be discriminated against.
3. Avoid space waste.

These qualitative recommendations need quantitative formulation for optimization purposes. Hence, three objective functions are proposed in the following.

2.3.1 Max Ad Number Function

This function follows a simple logic that a layout capable of comprising many combinations with lots of advertisements is commodious and flexible. Thus, the first objective is the number of possible ad units in the layout:

$$\max V_1(c_1,\ldots,c_a) = \sum_{p=1}^{a} \sum_{K_j \in J(c_p)} \sum_{e=1}^{n} k_{je} \tag{2.14}$$

2.3.2 Max Most Difficult to Pack Ad Unit Function

High value of V_1 can be built on small advertising units that are easily packable. On the contrary, units that are wide can be very difficult to fit. But still it is necessary to place them. This objective can be captured by the minimum number of fitting possibilities for any ad unit in the solution. Thus, the second objective is:

$$\max V_2(c_1,\ldots,c_a) = \min_{1\le e\le n} \{\sum_{p=1}^{a} \sum_{K_j \in J(c_p)} k_{je}\} \tag{2.15}$$

This function is used for checking constraint 13. Layouts with $V_2(c_1,\ldots,c_a) = 0$ are invalid.

2.3.3 Min Single Ad Waste

Due to the lack of ads, some columns may be filled only partially. In the worst case, only a single ad may be available to put in a column. The waste of space for a single ad placement can be defined as the remaining free horizontal space. The inner waste is the space around a sole ad unit in a column of a website. Thus, it is $c_p - w_e$ for ad unit r_e and column p. There is also outer waste that should be taken into account. This is the difference between page width and the sum of column widths $W - \sum_{p=1}^{a} c_p$, multiplied by the number of advertising units, n, as the waste is calculated for every ad unit. Ignoring outer waste would lead to a false conclusion that narrowing the layout below W reduces wasted space. In a multicolumn layout a single advertisement can be put in any of the columns. By the common sense, it should be placed in the column where the waste will be the lowest. This leads to the third objective function:

$$\min V_3(c_1,\ldots,c_a) = \sum_{e=1}^{n} \min_{1\le p\le a} \{c_p - w_e : c_p \ge w_e\} + n(W - \sum_{p=1}^{a} c_p) \tag{2.16}$$

The quality of layouts can be compared on the basis of the above three functions.

Let F be a set of all feasible layouts. The goal of layout optimization can be stated as follows:

$$\max_{G_z \in F} \{\gamma_1 V_1(c_{z1}, \ldots, c_{za}) + \gamma_2 V_2(c_{z1}, \ldots, c_{za}) - \gamma_3 V_3(c_{z1}, \ldots, c_{za})\} \qquad (2.17)$$

where $c_{z1}, \ldots, c_{za}$ are the column widths in the layout G_z, and $\gamma_1, \gamma_2, \gamma_3$ are the weights of the objective functions (cf. Section 2.5.2). As the third function should be minimized, it is subtracted from the sum of the first two functions that are maximized. The first two functions sum their values over all columns in a layout. Consequently, their partial scores can be calculated for all possible column widths once, and then added to evaluate any feasible layout.

Since representing multicriteria problem solution with a single value is as tempting as it is deceptive, it was decided to construct also Pareto frontiers for each instance of the LPfAF problem. Thus, the solution algorithm developed in the following section will give a chance of choosing from a spectrum of nondominated solutions, e.g., the solutions embodying the previously mentioned rules of art applied to layout partitioning. LPfAF problem is obviously $\mathcal{NP}$-hard because it comprises, e.g., subset sum and knapsack problems as special cases [47]. However, in the context of web pages with relatively limited widths and highly constrained solution set, it is not a key computational constraint. This will be analyzed in Section 2.6.

2.4 Solution Method

In this section methods of finding the best layout partitioning are introduced. The algorithm works in four stages:

1. Ad units are joined together to create all feasible combinations.
2. A list of valid combination widths is created and the partial scores of the first two objective functions are calculated.
3. All feasible layout partitions are enumerated. The third objective function, and the total scores are calculated.
4. The best solution and/or the member of the Pareto frontier are selected from the list of layouts.

2.4.1 Combining Ad Units

As described earlier, ad placement is similar to guillotine cutting problem. Therefore, Wang two-dimensional constrained cutting stock algorithm [122] is used for finding all feasible combinations. Note that constraints similar to 3, 4, 5 and 10 already existed in the original Wang algorithm. The algorithm uses the following steps [122]:

1. Set the initial variables $L^{(0)} = F^{(0)} = R$ and $\kappa = 1$.
2. a. Let $F^{(\kappa)}$ be a set of all combinations made by joining together combinations from $L^{(\kappa-1)}$ vertically or horizontally with respect to constraints 4, 3, 8, 7, 10 and 5 (the order is important here).

   b. Set $L^{(\kappa)} = L^{(\kappa-1)} \cup F^{(\kappa)}$. Remove duplicates from $L^{(\kappa)}$.
3. If $F^{(\kappa)}$ is nonempty, set $\kappa = \kappa + 1$ and repeat step 2. Otherwise $M = L^{(\kappa-1)}$ is a complete set of feasible ad combinations.

This algorithm is clearly exponential. In each iteration each pair of combinations can be joined vertically and horizontally. Thus, $|L^{(\kappa)}| \leq 2|L^{(\kappa-1)}|^2 + |L^{(\kappa-1)}|$. Since $F^{(0)} = R$, $|F^{(0)}| = n$, and t' is the limit on the number of ad units in a combination, the complexity of the algorithm can be bounded from above by $\mathcal{O}(2^{2^{t'}-1}n^{2^{t'}})$. However, the number of combinations d is in practice greatly limited by the imposed constraints (cf. Section 2.6).

The main computational disadvantage of this algorithm are combination duplicates. Duplicates are combinations built of the same advertising units aligned in various orders in the same direction. Ads order makes no difference for packability of a layout. Thus, the combinations that differ exclusively in the ad unit sequence are duplicates (in other words are isomorphic). Checking as fast as possible for duplicates is a key requirement for usability of the algorithm.

To avoid creating and checking for unnecessary duplicates, few improvements were introduced. When constructing $F^{(\kappa)}$ in step 2a by joining elements from $L^{(\kappa-1)}$, where $L^{(\kappa-1)} = L^{(\kappa-2)} \cup F^{(\kappa-1)}$, the results will comprise:

I. elements from $L^{(\kappa-2)}$ joined with themselves,

II. elements from $L^{(\kappa-2)}$ joined with elements from $F^{(\kappa-1)}$,

III. elements from $F^{(\kappa-1)}$ joined with elements from $L^{(\kappa-2)}$,

IV. elements from $F^{(\kappa-1)}$ joined with themselves.

Observe that construction I was done in the preceding iteration. Results of II and III are duplicates. To get only results III and IV, $F^{(\kappa-1)}$ should be joined with $L^{(\kappa-1)}$, where $L^{(\kappa-1)} = L^{(\kappa-2)} \cup F^{(\kappa-1)}$. This gives $F^{(\kappa)}$ without

duplicates built by the original Wang algorithm. For better performance, constraints should be evaluated in the order that prunes the combinations as fast as possible.

Duplicates can be recognized by comparing vectors of ad unit frequencies $K_g = (k_{g1}, \ldots, k_{gn})$ and dimensions $h'_g \times w'_g$. To implement it efficiently, numeric signatures are calculated from K_g where each component k_{gi} is represented as a digit. Number of bits x for such a digit should be selected to represent the largest expected value b_e. For example for $b_e = 2$ two bits per digit are required. The signatures can be stored as `int` variables. Then, depending on the size of integer in the programming environment used, for each $32/x$ or $64/x$ ad units (digits) in the input dataset only one fast `int` comparison is necessary to check them. This, plus comparison of the combination envelope sizes suffice for efficient duplicate recognition.

Another performance improvement for Wang algorithm is checking constraints in the order that prunes the combinations as fast as possible. The algorithm uses the following sequence of checking the constraints: 4, 3, 8, 7, 10, 5. The first three are the easiest to check. These constraints can be quickly checked for excluding a combination from further joining (e.g. when the width will not allow to add even the narrowest ad without exceeding page width: $w'_g > W - \min_{1 \le e \le n}\{w_e\}$). Such combination can be omitted without trying to join it with all other combinations from $L^{(\kappa-1)}$. Furthermore, if constraints 8, 7 or 5 are not satisfied on horizontal joining, then there is no need to check for vertical alignment.

The value of t' for constraint 8 can sometimes be set so high that $t' = t \max_{1 \le e \le n}\{b_e\}$. In such a case it will not exclude any of the solutions which would not be otherwise eliminated by constraints 7 or 5. Still, the constraints should be checked in the presented order. It helps to avoid testing whole branches of combinations, that would be finally discarded by constraints 7 or 5. This order provided 12.5% gain in the execution time for the largest dataset.

Finally, constraints 7 and 5 are computationally most demanding and should be checked as late as possible. The first requires enumeration by the ad units, and the second needs comparisons for each ad unit.

2.4.2 Valid Column Widths List

As a result of the previous step a set of combination widths is obtained. In this step algorithm constructs set $Y = \{y_1, \ldots, y_s\}$ of feasible column widths on the basis of the combination widths and constraints 11, 12. Violation of constraints 11, 12 effectively eliminates a combination. Objectives V_1, V_2 change values only at widths w'_g and $2w'_g$, for each distinct width w'_g for some combination

K_g, $g \in \{1, \ldots, d\}$, as below w'_g and above $2w'_g$ combination K_g cannot be placed. Objective function V_3 is monotone. By constraint 2 no column can be wider than $u \leq W$. Thus, number s of widths in Y is $s \leq \min\{2d, W\}$. In practice $s < W$ because ad unit widths are divisible by 5 and 2. This step can be done in $\mathcal{O}(ds)$ time, including precalculation of partial values of the objective functions for all the feasible widths.

2.4.3 Browsing Layouts

In this step the algorithm creates all feasible layouts using set Y of feasible column widths. As column ordering does not affect packability of the layout, column widths can be ordered non-decreasingly, i.e., $c_1 \leq ... \leq c_a$ to avoid browsing duplicate solutions. Note that minimum column widths must also follow the non-decreasing order, i.e., $m_1 \leq ... \leq m_a$, and thus they determine the lower bounds on the ordered column widths. Thanks to the column width order, all acceptable layouts can be enumerated by the following algorithm. For simplicity of presentation, superscript *old* is used to refer to values of variables obtained in the preceding iteration. Remember, that all $y_r \in Y$.

1. Set $c_p = m_p$ for each $p = 1, \ldots, a-1$.
   Set $c_a = \max_{1 \leq r \leq s}\{y_r : y_r \leq W - \sum_{p=1}^{a-1} c_p\}$. Set $q = a - 1$.

2. If $c_{a-1} \leq c_a$ record $(c_1, \ldots, c_a)$ as a proper layout and calculate values of the three component objective functions.
   Select new $c_a = \max_{1 \leq r \leq s}\{y_r : y_r < c_a^{old}\}$ and repeat step 2.
   If such c_a does not exist, proceed to 3.

3. Set new $c_q = \min_{1 \leq r \leq s}\{y_r : y_r > c_q^{old}\}$.
   For each $p > q$ set $c_p = \max\{m_p, c_{p-1}\}$.
   Set $c_a = \max_{1 \leq r \leq s}\{y_r : y_r \leq W - \sum_{p=1}^{a-1} c_p\}$.
   If $c_q \leq c_{q+1}$ set $q = a - 1$ and go to step 2.

4. Decrease q by one.
   If $q > 0$ go to step 3.

For s different column widths and a columns there are at most $\mathcal{O}(s^a)$ possible layouts. Hence, this step can be executed in time $\mathcal{O}(ads^a)$, including calculation of partial scores of the objective functions, involving at most d combinations. Since in practice $a \leq 4$, this algorithm is basically polynomial in s.

Yet, a method of fast browsing column widths worth checking is needed because the most time-consuming part of the proposed algorithm is searching in Y for the next greater or smaller value of column width.

This can be done by ordering Y by column widths, and then building a vector of references from any current width y_r to the next greater/smaller width. This vector can be built with a single pass of Y, and will not be memory-consuming as in practice its cardinality will be much smaller than page width W, where usually $W \leq 1600$ (see Section 6.1).

2.4.4 Selecting Final Results

In the preceding step the algorithm constructed a list of feasible layouts and calculated their scores in all three objective functions. For the weighted objective function (2.17), the three objectives were scaled to the common range [0,1]. Now, selecting the best weighted solution can be done by scanning the list of solutions. Constructing Pareto frontier is not much more complicated than this. Each solution has to be compared with the solutions already included in the Pareto frontier. If one of them is dominated, then it should be removed and the current solution should be added to the frontier. This procedure can be run in quadratic time of the number of feasible layouts.

2.4.5 Example For a Small Instance

For better understanding of how the algorithm works here a sample run on an instance small enough to make it traceable is presented here.

The instance consists of ad units Skyscraper (120x600), Medium Rectangle (300x250), page width is $W = 990$, there are $a = 2$ columns. Input parameters are set as described in Section 2.5.2 and Section 2.6.1, with the exception of padding set for clarity to $\sigma = 0$.

In step 1 of the algorithm (cf. Section 2.4.1) ad units are combined into six feasible ad combinations shown in Fig. 2.3. Combinations higher than 600px, e.g. two Skyscrapers joined vertically, are discarded by constraint 4. Constraint 10 excludes combinations with space waste above 10%. For example, Skyscraper joined horizontally with Medium Rectangle have 41.7% of waste. Combinations with more than two ad units of the same type are excluded by constraint 5.

In step 2 (cf. Section 2.4.2) combinations widths w'_g, and double widths $2w'_g$ serve to build the set of feasible column widths $Y = \{120, 240, 300, 480, 540, 600\}$. By Constraint 2 widths greater than 870px are excluded.

Let $\upsilon_x = \sum_{K_j(x) \in J(x)} \sum_{e=1}^{n} k_{je}$ represent partial score for function V_1 and column width x. Calculation of function V_2 is facilitated by vector ϕ_x representing ad unit multiplicity for column width x. Component e of ϕ_x is $\phi_{xe} = \sum_{K_j \in J(x)} k_{je}$. Data of combination g of width w'_g is aggregated into partial scores of υ_x, ϕ_x, for column widths x in $[w'_g, 2w'_g)$. For instance, com-

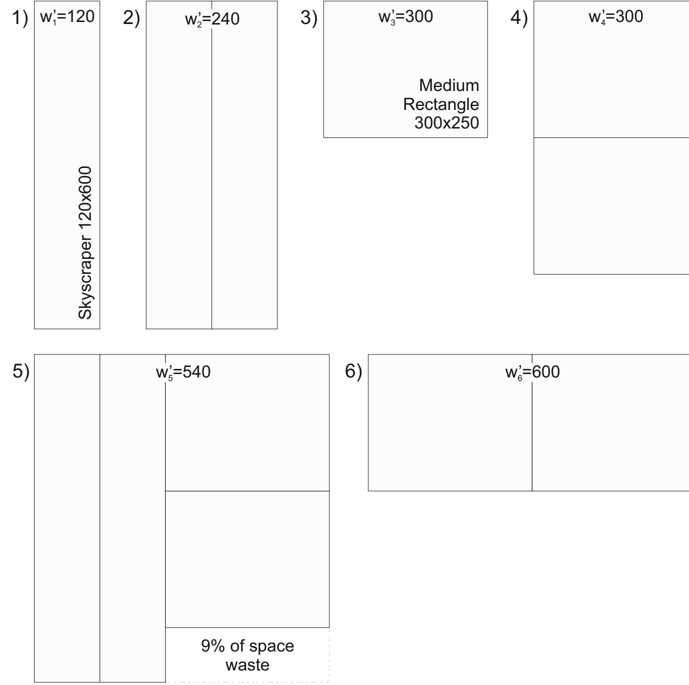

Figure 2.3: Ad combinations for the example instance.

bination $g = 2$ in Fig.2.3, has ad unit multiplicity vector $K_2 = (2, 0)$ and can be placed in columns 240px and 300px wide. Thus, values of υ_{240} and υ_{300} are increased by the total number of ad units in this combination $\sum_{e=1}^{n} k_{2e} = 2$. Furthermore, $K_2 = (2, 0)$ is added to vectors ϕ_{240} and ϕ_{300}. No other combination fits in width 240 so the partial scores remain $\upsilon_{240} = 2$ and $\phi_{240} = (2, 0)$. Combinations $g = 2, 3, 4$ in Fig.2.3 fit in column width 300. The partial scores for width 300 become $\upsilon_{300} = 5$ and $\phi_{300} = (2, 3)$. Other combinations and widths are evaluated in the analogous way.

In step 3 (cf. Section 2.4.3) feasible layouts are created by browsing combinations of the widths from set Y: 120+600=720, 120+540=660, 120+480=600, 120+300=420, 240+600=840, 240+540=780, 240+480=720, 240+300=540, 300+600=900, 300+540=840, 300+480=780, 300+300=600. Layouts not able to accommodate some ad unit are eliminated by constraint 13. For example, layout 120+240=360 has no column for Medium Rectangle (300x250). Layout objectives V_1 and V_2 are calculated from the partial scores of the column widths. For example, for layout 240+300=540: $V_1(240, 300) = \sum_{p=1}^{a} \upsilon_{c_p} = \upsilon_{240} + \upsilon_{300} = 7$, and $V_2(240, 300) = \min_{1 \le e \le n}\{\phi_{240,e} + \phi_{300,e}\} = \min\{2 + 2, 0 + 3\} = 3$. To calculate V_3 ad units are tested for fit in the layout columns. In the 240+300

layout the Skyscraper (120x600) leaves less waste, i.e. 120px, when placed in the narrower column. The Medium Rectangle (300x250) fits with waste 0 in the wider column. This layout is very narrow and leaves $2(990-540) = 900$px of outer waste. Thus, $V_3(240, 300) = -120 - 0 - 900 = -1020$.

In step 4 (cf. Section 2.4.4) the layouts are compared on the basis of V_1, V_2, V_3. Solution 300+540=840 is both Pareto optimal and best on the weighted objective function. It allows for placing ads in $V_1 = 12$ ways, each ad unit can be placed in at least $V_2 = 4$ ways. There are $V_3 = -180 - 0 - 2(990-840) = -480$ pixels of horizontal waste when placing single ads.

2.5 Benchmarks

In this section benchmark datasets are introduced. The datasets are of vital importance for practical solvability of LPfAF problem. A good column layout can be determined only with proper data on advertisements.

2.5.1 Data Sets

Ad units of three ad networks were chosen as benchmarks (see Tab. 2.2). The first network is Google AdSense [51], further referred to as Google. It offers text ads, graphical banners, and some video ads. They are automatically selected from a large database on the basis of the keywords. The second is Clicksor [33], offering an almost equally wide set of ad units, also including text ads, graphical banners and rich media banners. For most of them Clicksor is using contextual targeting. Finally, AdBrite [2] is providing the narrowest set of ad units, mostly graphical and rich media, adding to the contextual targeting some behavioral and demographic targeting. There are also other advertising methods offered by the three companies, but studying them is beyond the scope of this thesis. When it is hard to choose a single ad network, the 18 standard ad units from the IAB Unit Guidelines [61], seem to be a good choice. A subset and a superset of the above ad sets will be also used in the following discussion.

2.5.2 Webmaster Survey

To evaluate website layouts using function (2.17), it is necessary to weight the component objective functions. Instead of choosing the weights ad hoc, a survey among the webmasters maintaining websites profiting from advertising was conducted. The experts were asked to order the three objectives as important, neutral, or unimportant. Totally 21 replies were obtained. The results are as follows (cf. Fig. 2.4):

Ad unit	w (px)	h (px)	IAB	Google	Clicksor	AdBrite	Subset	b_e
Micro Bar	88	31	✓					2
Button 2	120	60	✓					2
Button 1	120	90	✓					2
Vertical Banner	120	240	✓	✓	✓			2
Skyscraper	120	600	✓	✓	✓	✓	✓	2
Square Button	125	125	✓	✓	✓			2
Wide Skyscraper	160	600	✓	✓	✓	✓	✓	2
Rectangle	180	150	✓	✓				2
Small Square	200	200		✓				2
Half Banner	234	60	✓	✓				2
Vertical Rectangle	240	400	✓					2
Square Pop-Up	250	250	✓	✓	✓			2
3:1 Rectangle	300	100	✓					2
Medium Rectangle	300	250	✓	✓	✓	✓	✓	2
Half Page Ad	300	600	✓		✓			2
Large Rectangle	336	280	✓	✓	✓			2
Full Banner	468	60	✓	✓	✓	✓	✓	2
Full page ad	550	350				✓		*
Pop-Under	720	300	✓					*
Leaderboard	728	90	✓	✓	✓	✓	✓	2
Items, n			18	12	10	6	5	

Table 2.2: Data sets. Ad unit names come from IAB where possible. * Ads designed not to be used in groups and thus by constraint 6 excluded from being combined.

1. General packability V_1 - was favored by 11 respondents and only 6 pointed it as not important.

2. Quality of fitting the most difficult ad units V_2 - found the least interest, only 5 webmasters pointed it out as important while 10 as unimportant.

3. Waste of space resulting from placing single ad units V_3 - was in the middle, it got 5 accepts and 5 rejects.

To represent the results in weights, an assumption was made that weights should start with an equal score, and should be increased or decreased accordingly for the favoring or disfavoring answers. The initial equal score was 100/3. There were 63 answers for all objective functions. Therefore, the weights were decreased, or increased, by 100/63 for each positive or negative recommendation. The results were rounded down to $\gamma_1 = 42, \gamma_2 = 25, \gamma_3 = 33$, for objectives V_1, V_2, V_3 respectively.

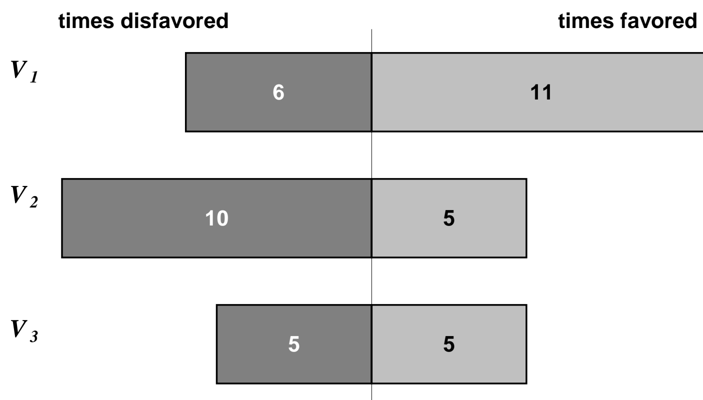

Figure 2.4: Webmaster survey results.

2.6 Computational Experiments

In this section the results of computational experiments with the above introduced algorithms and datasets are presented and discussed.

2.6.1 Input Parameters

Before starting the experiments, it was necessary to choose the input parameters: W - page width, b_e - limits to repetitions of ad units r_e in combinations, t - the limit to different ad units in a combination, β - acceptable space waste ratio, σ - ad unit padding size, and minimal column widths m_p. Such choices can be very subjective, and probably every webmaster would set them a bit differently.

For selecting page width W information about web browsers resolutions is instrumental. As of January 2011 [105] building layouts looking good in the 1024x768 resolution could be almost abandoned because less than 14% of Internet users use such resolution. Over 82% of browsers can display properly websites prepared for width of 1280px, while those capable of displaying 1440px or more, were still a minority. Let us remind, that browser elements like frames and a vertical scroll bar take at least 20-30px. Thus, $W = 990px$ was chosen for two column, and $W = 1250px$ for three and four column layouts.

The most difficult choice was the value of b_e. For some ad units b_e cannot be set to more than 2. The skyscraper ad units (see Tab. 2.2) can be joined only horizontally due to constraint 4, and putting more than two in such a way would be neither acceptable nor aesthetic. The same applies to Banner and Leaderboard, only with respect to vertical joining and constraint 3. After the

analysis of the preliminary results, it has been concluded that large changes in the results follow from small changes of b_e. Using $b_e > 2$ for some ad units ends with overflow of combinations constructed of these units, and their great over-representation in the results. An easy way to get a flood of such poor results is to set larger b_e values for the smallest ad units, e.g. the first three in Tab. 2.2. These ad units are too small to violate the combination size constraints 3 and 4, which results in production of an enormous number of combinations. Up to extra 400% of such combinations were observed in the preliminary tests. It should be noted that the three smallest ad units are not very important because they appear in IAB standard only, and are marked as obsolete. Nowadays they are used mostly for advertising exchange [83], logos, and are rarely seen in practical advertising. Thus, $b_e = 2$ was set for all ad units.

Limit $t = 4$ of different ad units in a combination was used. Several other values from 2 to 8 were tested. For values greater than 4 the algorithm produced many patchwork combinations like the one shown in Fig. 2.2c, without noticeable improvement in results quality. Space waste limit was set experimentally to $\beta = 10\%$. This was a difficult choice, as most webmasters would probably accept displaying ad combinations with far more waste. However, such small β reflects good quality layout partitioning. The value of β not only limits the waste in the final combinations, but also eliminates combinations from further joining, greatly affecting the search space, as will be shown further. The choice of padding size σ is purely aesthetic. It does not affect computational complexity, if kept reasonably small compared to ad unit sizes. Value of $\sigma = 2$ pixels was used. Minimum column widths m_p were set to the width of the narrowest ad unit in each used set. Two largest ad units, Pop-Under and Full page ad, (marked with "*" in Tab. 2.2) are included in set X. By constraint 6 they are excluded from being combined. This reflects their purpose of working outside any combinations or even outside column layout, as their names suggest.

Figure 2.5 shows the effect of parameters W, t, b_e, β on the execution time of the algorithm and the number of feasible ad combinations for the Google dataset. Parameters b_e and β heavily influence both these factors. With the increasing number of feasible combinations, also the number of layouts in the Pareto frontier is increasing. Selecting input parameters inadequately, as discussed above, can lead to very dense Pareto frontiers with solutions differing by single pixels, usually in the widest column.

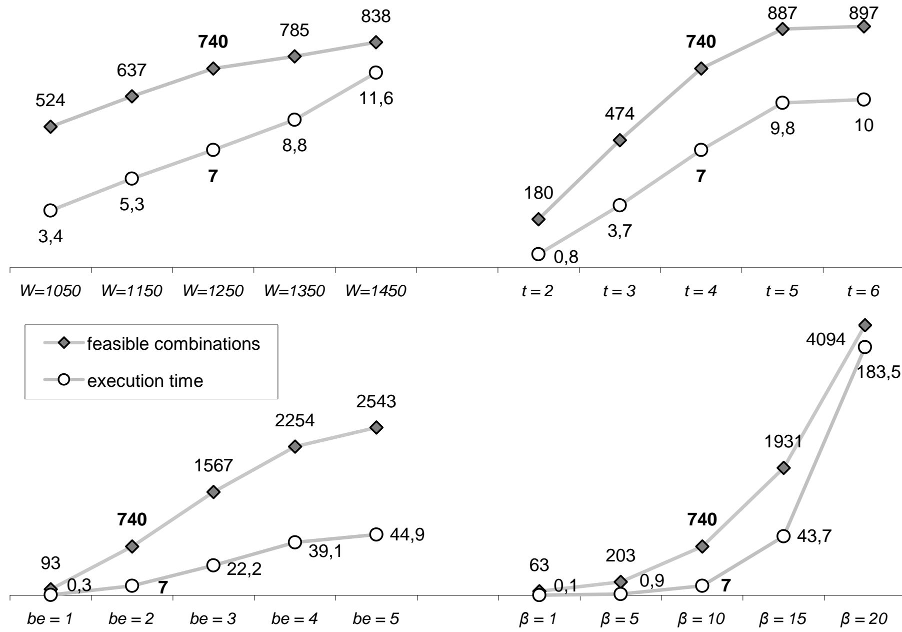

Figure 2.5: Impact of input parameters on the execution time (seconds) and the number of feasible ad unit combinations (Google dataset was used).

2.6.2 Execution Times

The algorithm presented in Section 2.4 was tested on a computer with Athlon 64 2.4Ghz processor and with RAM limited to 512MB. PHP programming language was used to allow future transforming this work into an online decision support tool. The tool was created later is available here [41]. Table 2.3 presents execution times for the most difficult case - three columns layout with $W = 1250$ for all data sets. As it can be seen, despite using exponential algorithms, the execution times are not particularly long and can be accepted in practice, especially if one takes into account that computations are performed once at the website design stage. Table 2.3 also shows the number of feasible combinations and the number of combinations that were examined.

2.6.3 Layout Partitioning Results and Discussion

Basic results for the given problem, that will be discussed here are presented in Table 2.4 and Table 2.5. Full results are attached as Appendix A, at the end of this dissertation. For each dataset and number of columns, there is a separate set of results consisting of the objective functions ranges, the best weighted result and selected results from Pareto frontier. Due to the size of most Pareto

	Stage I	Stage II	Stage III	Stage IV	Search space	Feasible combinations
subset	0.004	0.009	0.002	0.001	562	23
Adbrite	0.004	0.010	0.002	0.001	612	24
Clicksor	1.476	0.577	0.108	0.086	144 610	386
Google	5.016	1.663	0.200	0.166	530 922	740
IAB	92.200	13.661	0.510	0.383	9 340 506	3 120
Superset	140.158	20.444	0.677	0.520	14 120 750	3 836

Table 2.3: Algorithm execution times in seconds, search space size and the number of feasible combinations found.

frontiers, only the layouts with the best scores for one of the three objective functions are shown in Table 2.4. Each entry consists of column widths with the total layout width, values of the three objective functions and the value of the weighted linear function. For example, consider the results of Google dataset for three columns layout and $W = 1250$. Ranges of the three objective functions were respectively [1663,2571], [16,28], [-7539,-1723], making the range of the values 55%, 75%, 438% of the lower end (in the terms of absolute values). Thus, the objective functions are sensitive to the layout construction. Full results can be found in Appendix A.

Table 2.6 shows the cardinality of Pareto frontiers for all the datasets. The number of solutions in the Pareto frontier can be an indicator of flexibility in constructing good layouts. An interesting observation can be made on Adbrite dataset (it applies to the subset as well). Adbrite contains the fewest ad units. It has two wide units, two tall units and only one rectangle. Hence, making good combinations is very difficult. For example, there were only five feasible solutions for Adbrite, two columns, and W=990px. This resulted in only one solution in the Pareto frontier, for all Adbrite instances. Clicksor ad units offer far greater flexibility in use, while Google flexibility is even better. Observe that for two column instances, which were the least packable because of the small width, the Pareto frontier cardinality grows slower from Clicksor to the superset than the number of ad units in the datasets. Furthermore, the number of feasible combinations and the cardinality of Pareto frontiers of IAB and the superset may be a sign that these sets are too large. In effect they are difficult to use. Their subset should be more manageable in practice.

The Adbrite results for two columns are the same as for the ad hoc "bread and butter design" approach described earlier. "Bread" is set to the widest add plus padding, the "butter" is given the rest. For three and four column layouts, the Adbrite results differ from the ad hoc approach solutions, but still are close to it. The results for the richest sets: IAB in the two column instance and the superset for two and three columns are of similar nature. Clicksor and Google best weighted results show less similarity to the results of the ad hoc

subset, W=990, 2 columns $V_1 \in [42, 47], V_2 \in [7, 7], V_3 \in [-2534, -1914]$ 248+732=980; 47, 7 , -1914; 100.0	Pareto frontier: 248+732=980; 47, 7 , -1914; 100.0
subset, W=1250, 3 columns $V_1 \in [39, 55], V_2 \in [5, 7], V_3 \in [-2606, -858]$ 164+328+732=1224; 55, 7 , -858; 100.0	Pareto frontier: 164+328+732=1224; 55, 7 , -858; 100.0
subset, W=1250, 4 columns $V_1 \in [44, 50], V_2 \in [7, 7], V_3 \in [-1986, -858]$ 164+164+164+732=1224; 50, 7 , -858; 100.0	Pareto frontier: 164+164+164+732=1224; 50, 7 , -858; 100.0
AdBrite, W=990, 2 columns $V_1 \in [43, 48], V_2 \in [7, 7], V_3 \in [-2846, -2102]$ 248+732=980; 48, 7 , -2102; 100.0	Pareto frontier: 248+732=980; 48, 7 , -2102; 100.0
AdBrite, W=1250, 3 columns $V_1 \in [40, 56], V_2 \in [5, 7], V_3 \in [-3054, -1062]$ 164+328+732=1224; 56, 7 , -1062; 100.0	Pareto frontier: 164+328+732=1224; 56, 7 , -1062; 100.0
AdBrite, W=1250, 4 columns $V_1 \in [45, 51], V_2 \in [7, 7], V_3 \in [-2310, -1062]$ 164+164+164+732=1224; 51, 7 , -1062; 100.0	Pareto frontier: 164+164+164+732=1224; 51, 7 , -1062; 100.0
Clicksor, W=990, 2 columns $V_1 \in [735, 1123], V_2 \in [16, 20]$, $V_3 \in [-5833, -4493]$ 164+816=980; 1012, 20 , -5433; 64.8	Pareto frontier (selected results): 124+864=988; 1123, 17 , -5833; 48.2 164+816=980; 1012, 20 , -5433; 64.8 258+732=990; 783, 16 , -4493; 38.2
Clicksor, W=1250, 3 columns $V_1 \in [739, 1386], V_2 \in [15, 25]$, $V_3 \in [-5977, -1655]$ 184+254+812=1250; 1044, 24 , -2153; 71.5	Pareto frontier (selected results): 129+129+991=1249; 1386, 15 , -4507; 53.2 129+304+816=1249; 1082, 25 , -3282; 67.8 254+258+737=1249; 836, 17 , -1655; 44.3
Clicksor, W=1250, 4 columns $V_1 \in [743, 1147], V_2 \in [16, 22]$, $V_3 \in [-4737, -1280]$ 124+124+184+816=1248; 1026, 22 , -2073; 79.9	Pareto frontier (selected results): 124+129+129+864=1246; 1147, 19 , -3638; 65.0 124+124+184+816=1248; 1026, 22 , -2073; 79.9 129+129+254+737=1249; 813, 17 , -1280; 44.4
Google Ads, W=990, 2 columns $V_1 \in [1659, 2181], V_2 \in [18, 25]$, $V_3 \in [-7123, -5515]$ 248+742=990; 1769, 25 , -5635; 64.4	Pareto frontier (selected results): 124+864=988; 2181, 18 , -7123; 42.0 248+742=990; 1769, 25 , -5635; 64.4 258+732=990; 1807, 23 , -5515; 62.8
Google Ads, W=1250, 3 columns $V_1 \in [1663, 2571], V_2 \in [16, 28]$, $V_3 \in [-7539, -1723]$ 184+254+812=1250; 2150, 26 , -3335; 67.2	Pareto frontier (selected results): 129+129+991=1249; 2571, 16 , -6049; 50.5 129+340+762=1231; 2049, 28 , -4204; 61.8 254+258+737=1249; 1926, 24 , -1723; 61.8
Google Ads, W=1250, 4 columns $V_1 \in [1667, 2205], V_2 \in [20, 25]$, $V_3 \in [-6051, -1344]$ 124+129+254+742=1249; 1834, 25 , -1358; 70.9	Pareto frontier (selected results): 124+129+129+864=1246; 2205, 20 , -4932; 49.8 124+124+258+742=1248; 1846, 25 , -1519; 70.7 129+129+258+734=1250; 1824, 23 , -1344; 60.3
IAB, W=990, 2 columns $V_1 \in [7033, 10453], V_2 \in [37, 51]$, $V_3 \in [-11183, -8195]$ 258+732=990; 10441, 37 , -8195; 74.9	Pareto frontier (selected results): 254+732=986; 10453, 37 , -8267; 74.2 164+820=984; 8339, 51 , -9887; 55.4 258+732=990; 10441, 37 , -8195; 74.9
IAB, W=1250, 3 columns $V_1 \in [6495, 13601], V_2 \in [37, 72]$, $V_3 \in [-13567, -2990]$ 164+258+828=1250; 10301, 69 , -5275; 71.2	Pareto frontier (selected results): 129+383+732=1244; 13601, 37 , -4685; 69.7 92+313+842=1247; 10714, 72 , -9479; 62.7 254+258+737=1249; 12013, 38 , -2990; 66.3
IAB, W=1250, 4 columns $V_1 \in [6600, 11477], V_2 \in [37, 53]$, $V_3 \in [-11911, -2125]$ 92+92+254+812=1250; 9715, 50 , -3335; 76.1	Pareto frontier (selected results): 129+129+254+732=1244; 11477, 37 , -2295; 74.4 129+129+129+852=1239; 9536, 53 , -6215; 69.5 92+92+304+761=1249; 9887, 41 , -2125; 67.6
superset, W=990, 2 columns $V_1 \in [8979, 12475], V_2 \in [39, 51]$, $V_3 \in [-12221, -8901]$ 258+732=990; 12467, 39 , -8901; 74.9	Pareto frontier (selected results): 254+732=986; 12475, 39 , -8981; 74.2 164+820=984; 10277, 51 , -10781; 54.9 258+732=990; 12467, 39 , -8901; 74.9
superset, W=1250, 3 columns $V_1 \in [7904, 16708], V_2 \in [39, 72]$, $V_3 \in [-14941, -3225]$ 129+388+732=1249; 16708, 39 , -5303; 69.1	Pareto frontier (selected results): 129+388+732=1249; 16708, 39 , -5303; 69.1 92+333+824=1249; 13636, 72 , -10029; 66.2 254+258+737=1249; 14104, 40 , -3225; 63.3
superset, W=1250, 4 columns $V_1 \in [8308, 14280], V_2 \in [39, 57]$, $V_3 \in [-13101, -2419]$ 92+92+254+812=1250; 11804, 55 , -3643; 76.0	Pareto frontier (selected results): 92+92+333+732=1249; 14280, 39 , -2695; 74.1 92+92+248+816=1248; 10761, 57 , -4217; 69.7 92+92+304+762=1250; 11943, 45 , -2419; 66.9

Table 2.4: Layouts for selected test cases

AdBrite, W=990, 2 columns
$V_1 \in [22, 46], V_2 \in [3, 6], V_3 \in [-2712, -1272]$
328+632=960; 46, 6 , -1692; 90.4

Pareto frontier:
328+632=960; 46, 6 , -1692; 90.4
412+576=988; 35, 3 , -1272; 55.8

AdBrite, W=1250, 3 columns
$V_1 \in [23, 56], V_2 \in [3, 9], V_3 \in [-2962, -508]$
164+472+608=1244; 53, 9 , -564; 95.4

Pareto frontier:
124+472+632=1228; 56, 9 , -1144; 91.4
164+452+632=1248; 54, 6 , -616; 83.5
164+472+608=1244; 53, 9 , -564; 95.4
164+496+576=1236; 41, 6 , -508; 68.4
304+328+608=1240; 50, 6 , -560; 79.2

AdBrite, W=1250, 4 columns
$V_1 \in [24, 49], V_2 \in [3, 6], V_3 \in [-2342, -232]$
124+164+328+632=1248; 49, 6 , -272; 99.4

Pareto frontier:
124+164+328+632=1248; 49, 6 , -272; 99.4
164+164+328+592=1248; 40, 3 , -232; 59.9

Table 2.5: Results without Leaderboard ad unit

Dataset	No. of ad units	Feasible layouts			Pareto frontier		
		2 col.	3 col.	4 col.	2 col.	3 col.	4 col.
subset	5	5	23	7	1	1	1
AdBrite	6	5	23	7	1	1	1
Clicksor	10	102	1296	356	7	21	11
Google	12	146	2123	537	9	39	11
IAB	18	225	4069	1905	10	53	26
superset	20	264	5066	2329	12	72	21

Table 2.6: Datasets flexibility comparison - cardinality of feasible layouts and Pareto solutions.

approach. There are also many good alternative solutions in Pareto frontiers of all sets except Adbrite. The similarity to the ad hoc results is caused by the Leaderboard, the widest ad unit (728px wide). With padding $\sigma = 2$ it requires columns at least 732px wide. The subset and the AdBrite results are all built on this width, and it appears quite often in other solutions. The Leaderboard is known among webmasters as difficult to fit, but it is very popular and cannot be removed by the ad networks. However, if a webmaster choses to use it outside the column layout (e.g. in a page header or footer) then different solutions are obtained. Example results for Adbrite without the Leaderboard are shown in Table 2.5. Now column widths are completely different than for the entire dataset. Also Pareto frontiers give more options by offering 2, 5, and 2 elements for two, three, and four columns, respectively.

2.7 Conclusions

In this chapter web page layout optimization for advertisement fit has been studied. A big part of the study was dedicated to mathematical modeling of the problem including its aesthetic consideration and then providing algorithms able of solving it. Though the solution method is exponential in general, in practice it runs in acceptable time of several minutes.

As it could be expected different sets of layouts result from different page widths and datasets. Moreover, every webmaster would configure input parameters differently. Thus, it would not be valid to provide here a set of universal web page layouts. Rather, there is a need for automated tool constructing the layouts for practical use. Example of such application was implemented on basis on this research by students as their engineering thesis under supervision of the author [41].

Another direction of further research on web page partitioning might be inclusion of statistical data on real-world usage of particular ad types into similar research project. Analysis of the layouts produced with the use of such data could be a starting point to constructing some new general rules on optimized layout partitioning. Conversely, it could result in the conclusion that generalization here is not possible, and tailored optimization should be always used.

Not only layouts can be evaluated, but also the sets ad units from advertising networks. It follows from the results that AdBrite's ad unit set should be extended, while IAB ad unit set does not seem to provide additional flexibility matching its size, and thus can be cumbersome in use. A reversed research on the ad unit sets could suggest what kind of element should be added to a set, to provide the biggest improvement in usability, or what kind of element should be removed from a set without loss of flexibility. Ultimately, such algorithms could be used to construct from scratch a set of ad units that would be highly optimized for ease of fitting into web pages.

An important remark here is, that web page layout optimization can be seen as partitioning of two-dimensional space, but in one dimension only. This leads to a conclusion that problems of similar nature can be solved along the lines developed for LPfAF problem. For example, paper rolls are stripes produced by paper mills with big width dictated by the width of the production machinery. The wide rolls are cut into standardized width rolls for transportation and distribution. The standardized rolls are some kind of legacy. Instead, the widths could be optimized according to distribution of page formats cut from rolls. The roll width optimization could be done in a similar way with roll (column) widths adjusted to page formats used for printing (ad formats). Consider port container

terminals as another example. Assigning berthing space to ships would mean partitioning terminal space in the dimension perpendicular to the quay. Here quay partitioning optimization could consider parameters of the ships like width, capacity, importance, but also a time factor of arrivals or processing.

Hence, time can be also the second dimension if one would like to consider optimization of partitioning quay into berths. Berths are sections of quay used to host single ships. Both berths and ships can be considered as one-dimensional objects having only lengths. However, time of using the berth would add the second dimension typical of scheduling problem. Then, the problem of selecting berths widths can be considered as optimization problem similar to the one presented in this chapter.

In this research layout partitioning which is only one element in the process of constructing web pages was addressed. Thus, further research should be conducted into the other aspects of web page layout optimization. For example, graphical content selection, processing, and geometric positioning are interesting research topics. It can be concluded that e-commerce and web applications offer new fields of study in operations research, sometimes involving aesthetic aspects.

3 Tag Cloud Construction

3.1 Tag Clouds

In this chapter methods for building aesthetic tag clouds for use on websites will be proposed.

Basically, *tags* are phrases representing textually some set of objects. Tags can be, e.g., words and phrases summarizing content of a web page. These can be also the most frequent tags in social media, labels for best-sellers in marketing, keywords in news or in scientific publications. Each tag (a word or a phrase) has certain importance which is expressed in relation to other tags. Typically, tag importance is given as a number. A *tag cloud* is a graphical depicting of the tags projected onto a plane. A key requirement is that tags with high importance should be prominently visible in the tag cloud. Commonly, important tags are simply bigger. An example tag cloud from Flickr website is shown in Figure 3.1. There are various forms of tags and tag clouds. For instance, there are hashtags, data clouds, text clouds. A hashtag does not have to be a proper word or a phrase in some language. It can be an abbreviated word, an acronym, or any sequence of characters. Hashtags originated from tags and tagging popularized by Twitter. Hashtag was even chosen a "Word of the year 2012" by American Dialect Society [4]. Tag clouds can be built from hashtags as well. Data clouds or text clouds are specialized forms of tag clouds visualizing numerical data or word frequencies. In the further discussion, generic terms of a tag and tag clouds will be used.

The aim of this reasearch is to solve the problem of constructing visually acceptable tag clouds for web pages. The first step in tag cloud creation is preparation of tags themselves: tag/phrase selection and weighting (e.g.[46, 78]), clustering (cf. [82]), etc. Here it is assumed that the set of tags is given and their rendering in two dimensions is studied. Methods of digesting the text and extracting the tags rest in text mining area and are beyond the scope of this

research. The problem of rendering the tags into a tag cloud is formulated here as a combinatorial problem with specific objectives and constraints. Further organization of this chapter is the following. In Section 3.2 tag clouds in general and tag clouds for websites are discussed. Approaches, algorithms, design options and the choices taken in the past are surveyed. Requirements for tag clouds in the web usage are discussed. Section 3.3 provides a mathematical formulation of the Tag Cloud Construction Problem (TCCP). Algorithms solving the problem are introduced in Section 3.4. Results of the computational experiments are outlined in Section 3.5. The notation used throughout this chapter is summarized in Table 3.3.

3.2 Problem Analysis and Related Work Survey

Although tag clouds seem to be a modern invention, their origins can be traced back at least to 1976 [94]. Early tag clouds history is outlined in [118]. Around 2003 they gained a wide usage over the Internet. In 2006-2009 they became bloated, overused by many web-designers without considering whether they fit the purpose. Consequently, they were criticized and their application declined. Currently, a new generation of tag cloud approaches and applications is proposed and tag clouds can be seen where they fit well. Thus, tag clouds seem to follow the hype cycle [45] and they slowly reach the productivity stage. Many approaches to the tag cloud construction have been used in the past. In this section tag clouds are classified, as well as, the results of the studies on tag cloud formation and usability are outlined. Then, requirements for tag clouds to be used over the Internet are discussed. Finally, the requirements and status quo in web browsers as a platform for rendering tag clouds are studied.

3.2.1 Tag cloud taxonomy

There are several design choices determining appearance and usability of tag clouds. In particular, these are:

1. How tags are sorted. The options are: alphabetically, by importance, by context, randomly, packing-decided. The last means that tags may be reordered for better packing quality. Sorting by context means that tags are placed in groups connected, e.g., semantically, lexically, or in some other way specific to the field of application.

animals architecture art asia australia autumn baby band barcelona beach berlin bike bird birds
birthday black blackandwhite blue bw california canada canon car cat chicago
china christmas church city clouds color concert dance day de dog england europe fall
family fashion festival film florida flower flowers food football france friends fun
garden geotagged germany girl graffiti green halloween hawaii holiday house india
instagramapp iphone iphoneography island italia italy japan kids la
lake landscape light live london love macro me mexico model museum music nature
new newyork newyorkcity night nikon nyc ocean old paris park party people
photo photography photos portrait raw red river rock san sanfrancisco scotland sea
seattle show sky snow spain spring square squareformat street
summer sun sunset taiwan texas thailand tokyo travel tree trees trip uk unitedstates
urban usa vacation vintage washington water wedding white winter woman yellow zoo

Figure 3.1: Tag cloud from Flickr instance in Table 3.4.

Figure 3.2: Flickr instance in Table 3.4 rendered by Wordle.

2. Shape of the entire cloud. Possible options: rectangular, other regular (e.g. circular), irregular, given (e.g. given polygons, map borders used for visualization). Regular shapes may also have a ragged margin, which is often considered a typographical defect resulting from bad text justifying.

3. Shape of tag bounds. Options: rectangle, or character body. The former means that bounding boxes of the tags rendered in some given font are used. The latter means using the shapes of the characters in the given font. This allows for tighter tag alignment using free space around the letter bodies.

4. Tag rotation: none, free to rotate, allowed with limited degrees of freedom.

Source	1. Tag ordering	2. Cloud shape	3. Tag shape	4. Tag rotation	5. Vertical alignment
Flickr (2004)	alphabetical	ragged rectangle	rectangle	none	baseline
[65]Kaser (2007)	packing	rectangle	rectangle	none	free
[65]Kaser (2007)	context	ragged rectangle	rectangle	none	limited
[74]Kuo (2007)	alphabetical	ragged rectangle	rectangle	none	baseline
[46]Fujimura (2008)	context	irregular	rectangle	none	limited
[108]Seifert (2008)	packing	ragged polygon	rectangle	none	free
[119]Wordle (2009)	random or alphabetical	irregular	font body	from none to free	free
[37]Cui (2010)	context	irregular	rectangle	none	free
[98]Nguyen (2010)	alphabetical	given borders	rectangle	none	background
[66]Kim (2011)	packing	irregular	rectangle	none	baseline
[21]Burch (2013)	context	ragged regular	rectangle	none	limited
[27]Cheng (2014)	context	irregular	rectangle	none	limited
[78]Lohmann (2015)	context	elipsoid	rectangle	none	free
[28]Chi (2015)	random	given borders	rectangle	from none to free	free
This research	packing	rectangle	rectangle	none	baseline

Table 3.1: Summary of packing choices in tag clouds (See Section 3.2 for details)

5. Vertical tag alignment. Options: sticking to the typographical baselines, limited by the algorithm properties (e.g. some tags are grouped), free - possibly leading to 2-dimensional packing, forced by the tag cloud background (e.g. a given heat map).

The consequences of the design decisions can be compared in Figures 3.1 and 3.2 (cf. the design decisions outlined in Table 3.2.1). There are still further design-choices possible. For instance, it is possible to use differing colors or fonts (typefaces, sizes, weights and styles). Here it will be assumed that fonts are determined in the tag preparation step (for example, chosen by the web designer), and hence given as input. Note that use of colors to distinguish tags may be a bad idea for users with color-impaired sight. Thus,it can be assumed that tags are essentially monochromatic on a contrasting background (e.g. black on white).

3.2.2 Related work

Since the very start tag clouds construction attracted interest of researchers. Kaser and Lemiere [65] experimented with two types of tag clouds. Firstly, they considered building baseline tag clouds with the criteria of height and badness. The badness was similar to the measure developed for TeX [67] which is a sum of the free space at end of the line but also above the tags of smaller height. They were testing variants of Next Fit Decreasing and First Fit Decreasing [128] against random and alphabetical placement. Secondly, they proposed an idea of adapting min-cut placement floor planning algorithm minimizing cloud height for fixed width. They used nested tables for slicing floor plans, with grouping of strongly related tags. The criteria were area of bounding box and distance of

related tags. Kuo et al. [74] presented application of a tag cloud to summarize results of a query over a database. Their tag placement algorithm is very simple, tags are placed alphabetically and automatic line breaking shapes it into ragged rectangle. Fujimura et al. [46] proposed a metaphor of generating a topographical map to visualize large tag clouds (5000 tags, 10000x10000 pixels), where height on the map reflected tag importance. They used a genetic algorithm while the rank of each tag defines equivalent of its repulsion force. Seifert et al. [108] worked on fitting tag clouds into convex polygons, allowing limited font changes and tag truncation. Their algorithm partitioned the polygon area on every tag placement, and a technique of prioritization was used to keep these regions compact. Four versions of the algorithm were tested for best usability and aesthetic effects. Wordle [119] is a well-known web-based text visualizing tool. Wordle allows to choose several parameters like rotation or sorting, but always justifies tags to shapes of characters and outputs irregular tag clouds. Wordle uses randomized greedy algorithm that starts with tags placement at positions determined by its internal rules and user input settings. Then, starting from the tags in the middle of the cloud it iterates by colliding tags and searching for feasible positions on a spiral path around the cloud. To speed up the collision detection it uses hierarchical bounding boxes for tags, spatial index for tags already placed and ordered checking of collisions starting with tags that collided in the previous check. Cui et al.[37] used tag clouds to illustrate temporal evolution of tags and their contexts (e.g. semantic relations such as common word roots). They proposed tag clouds keeping the tags linked by the context together and coloring tags to visualize trends in time (such as tag appearing and disappearing over time). Their algorithm uses Delaunay Triangulation to build a graph for the initial tag placement and adaptation of force-directed algorithm to compact the cloud while preserving contextual links. Nguyen and Schumann [98] applied tag clouds fitting into map shapes to support exposition and exploration of geo-tagged data. They placed median tag in a calculated center of the shape, and the remaining tags in rows to fit the silhouette of the shape vertically. They allowed scaling and flexible adaptation of tag sizes, with limitations, to improve fitting. Kim et al. [66] applied graphs, where both nodes and edges were tag clouds, to show relationships between entities of text corpora. Their algorithm places tags in the free space as close as possible to the centers of nodes for node clouds, and as close as possible to centers of edges for edge clouds, respectively. While positioning tags the algorithm partitioned the remaining space into up to four rectangles which were later used for placing the following tags. Burch et al. [21] introduced the idea of prefix tag clouds. They built prefix trees to group words that share roots. The prefix trees were first visualized as tag clouds stretching to the right from the root prefix. Then

prefix tree clouds were positioned along a spiral path and moved as far as needed from the spiral center to avoid overlapping the previously placed tags. Cheng et al. [27] applied tag clouds with a similar goal of grouping synonyms. Their algorithm placed synonyms as a column around the most popular word from the group. The most popular synonym was visualized as the largest. Then, the groups were bundled into the cloud. Lohmann et al. [78] proposed word clouds on concentric circles where appearance of words in sections visualizes relationships between documents. Tag placement is done along concentric circles starting from the center with words ordered by decreasing frequencies. It was allowed to omit words that cannot be placed well. Chi et al. [28] are placing sets of tags into evolving or morphing silhouettes to present how text or data is changing in time. Tags are placed by use of rigid body dynamics with a set of constraints to obey the required bonduaries, orientation, position, uniformity, etc. Tag design choices made in the above papers are summarized in the Table 3.2.1.

3.2.3 Tag Cloud Usability Studies

Results of the studies on the effect of tag clouds on the user experience and productivity have been reported in [8, 54, 79, 106, 108, 120]. According to [106], the tasks supported by tag clouds are: searching, browsing, impression formation, and recognition/matching. The last one means verifying whether the tag cloud is representing a particular subject. Note that only searching is a goal-oriented task, while the remaining ones are rather free browsing tasks. Outcomes from an experiment measuring time necessary to find a certain tag, are reported in [54]. It has been found that alphabetically ordered lists are actually faster to search than tag clouds. The authors also concluded that users scan rather than read tag clouds. A different approach has been used in [108]. The users have been asked to point out three most important tags and the coincidence of their decisions with the actual tag ranking has been measured. Although the results were partially inconclusive because they depended on the variant of their algorithm, it can be judged that simple visualization methods, few tags, and pruning less important tags help in shortening reaction time. Yet, the resulting clouds are not necessarily considered beautiful. In [106] it has been concluded that font size and tag location affect low-level memory processes, while layout impacts the high-level ones, such as impression formation. In the study described in [8] font-related parameters were tested, leading to the conclusion that larger and stronger fonts draw more users attention. Font color, though well recognized, incurs difficulties in assessing importance. Authors of [79] conducted a study on the performance of executing certain tasks on various cloud layouts. They

confirm the earlier findings of [54] that locating a specific tag is fastest with alphabetical sorting and that users are scanning rather than reading. Yet, their other experiments show that for finding the most important tags, recalling tags, etc. cloud layout plays an important role. Walhout et al. [120] compared navigation on tag clouds with hierarchical menus by capturing eye movements and logging task performance. They found that tag clouds lead to more focused search without impairing task performance, resulting in fewer page revisits.

The above presented research was focused on goal-orientated tasks, which are easier to measure, as opposed to free browsing. Browsing is an important application of tag clouds in the web.

3.2.4 Tag Clouds for the Web

In the above account on the state of the art only two papers consider factors important for website use. Tag clouds for websites have to meet a set of additional requirements. Website space is always rectangular and scarce so it should be used wisely. All tag clouds in the web, even the irregular ones, are finally displayed on a rectangular space of some computer interface. Thus, fancy non-rectangular shapes waste space around the cloud. This gives a preference to tag clouds filling a rectangular envelope well. As websites usually use column layout [86], horizontal size of a tag cloud (i.e. width) is usually fixed, while the vertical size can be changed by moving website components below the tag cloud a little up or down.

A tag cloud for a website should use standard technologies, making a reasonable trade-off between fancy looks and the simplicity of the code. This has two reasons: Firstly, it is a matter of the ease of implementation. Secondly, not only humans read websites and making website content accessible to the robots is of great importance in search engine ranking [87]. Thus, tags must be robot-readable. Consequently, tag clouds must be available as text on the web page. Using HTML, CSS and JavaScript (JS) for coding tag clouds is a natural decision here, because these are the technologies commonly used in web page development. This implies some of the further choices: Though the use of exact tag shapes or tag rotation are possible in most modern browsers, they are not standard and cannot be guaranteed to work well and look in the same way for every client, especially mobile one. Wordle which is using exact font shapes and rotated tags can output tag clouds only as images. Making such tags clickable (i.e. assigning links to tag areas) on a website would be a real challenge. Hence, the use of exact font shapes and tag rotation should be discouraged. The same argument can be applied in preferring the alignment to the baselines over the freedom of arbitrary 2-dimensional packing. Tags on a baseline will be consid-

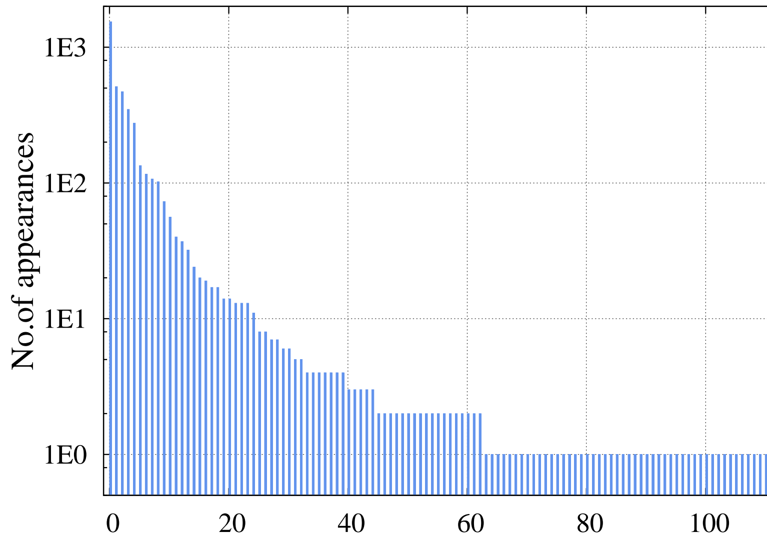

Figure 3.3: Tag sizes distribution measured on Internet users.

ered just as a line of text by the robots. Considering the results of the studies demonstrating that users scan lines of the clouds (see Section 3.2.1), the use of baselines will make reading tags easier and faster.

The next issue is the choice of tag ordering. It was already mentioned that clouds with alphabetically ordered tags perform worse in terms of search time than simple lists of phrases. Moreover, alphabetical ordering significantly restricts flexibility of packing: Firstly, as tags cannot be reordered the only remaining option is to choose where to put a line break. Secondly, use of different font sizes incurs big waste spaces because tags in the smallest font cannot be moved away from the very tall tags in the biggest font size. To achieve any reasonable visual quality tags have to be rearranged, i.e. the sequence of tags should follow from the packing algorithm.

3.2.5 Client Side

A number of challenges is posed by the target platform of tag cloud exposition. In times of more and more personalized web content each user can get a different set of tags. But there is more than that to significantly affect packing of the tags. Namely, clients may have different sizes (in pixels) of the same tag depending on the browser, system and fonts installed. A study was conducted into browser font rendering consistency to verify this expectations. A tailored set of 6 benchmark tag sets testing different methods defining look of text with CSS properties: fonts, font stacks, sizes and weights has been constructed (cf. Table 3.2). A script measuring tag sizes was installed on a production website and in the course of two days responses from 4201 different clients were registered.

```
<a href="#" style="
        font-family: sans-serif;
        font-weight: bold;
        font-style: italic;
        font-size: medium;      ">
shorts
</a>

<a href="#" style="
        font-family: Cambria; 'Hoefler Text'; Utopia;
    'Liberation Serif'; 'Nimbus Roman No9 L Regular'; Times;
    'Times New Roman'; serif;
        font-weight: normal;
        font-style: italic;
        font-size: 14pt;        ">
Neutral space
</a>

<a href="#" style="
        font-family: monospaced;
        font-weight: 800;
        font-style: normal;
        font-size: large;       ">
aero-moon.com
</a>

<a href="#" style="
        font-family: Helvetica; Verdana; sans-serif;
        font-weight: 400;
        font-style: '';
        font-size: x-large;     ">
waffle filling
</a>

<a href="#" style="
        font-family: Frutiger;'Frutiger Linotype'; Univers;
    Calibri; 'Gill Sans'; 'Gill Sans MT'; 'Myriad Pro';
    Myriad; 'DejaVu Sans Condensed'; 'Liberation Sans';
    'Nimbus Sans L'; Tahoma; Geneva; 'Helvetica Neue';
    Helvetica; Arial; sans-serif;
        font-weight: '';
        font-style: '';
        font-size: 16pt;        ">
Long Tail Marketing
</a>

<a href="#" style="
        font-family: serif;
        font-weight: 900;
        font-style: '';
        font-size: small;       ">
CREAM
</a>
```

Table 3.2: Tags set used to test distribution of tag sizes on client side.

In the gathered data it was possible to identify 112 distinct combinations of sizes for the benchmark tags. The results are shown in the Figure 3.3. As could be expected it was found that the distribution of tag sizes follow the power law: $popularity \approx 1532 \times pos^{-1.297}$, where pos is the rank of a position, with fit quality $R^2 = 0.984$. On the one end, the three most popular font combinations are found in, respectively, 36.61%, 12.21% and 11.19% of client platforms. Most of users use browser/system platforms that render the tags in less than a dozen of popular sizes. On the opposite end, sizes with popularity smaller than 1% form a long tail of 101 different values. Tag sizes on mobile devices differ much more than on desktop/laptop computers (even two-three times). These results lead to the conclusion that tag cloud construction must be adjusted to the tag sizes measured at the client side. Furthermore, it means that building tag clouds must be moved to the client side.

Algorithmic building of tag clouds on the client side has to meet a few further requirements. The algorithm must be deterministic because the tag cloud must look the same way for the given user. Thus, randomized algorithms, or algorithms linking their stopping criteria with the runtime must be excluded. The implementation has to use JavaScript (JS). Although other choices are possible, only JS has sufficient market penetration. Moreover, JS works on the elements of the Document Object Model (DOM) structure, thus supporting readability of the tag cloud for the robots. A disadvantage is that the algorithm constructing a tag cloud must run in very limited time, i.e. in the order of tenths of a second. There is plenty of research showing that users do not want to wait for downloading web page content and rendering it, because they quickly lose interest. An up-to-date survey given in [87] suggests time below 1.5 second for the whole page. Since the performance of the client browser is unknown, the algorithm must be fast and simple enough to give a valid solution in tight time limits even on slow browsers.

Before proceeding to the final problem formulation, let us summarize the design requirements: 1) Cloud shape is rectangular, 2) tags are rectangular boxes, 3) tags are reordered with packing, 4) rotation is not allowed, 5) tags are packed on shelves and aligned to the baselines, 6) minimum waste of the rectangular area is desired, 7) tonal weight distribution should be as even as possible, 8) a tag cloud must be text, not graphics, 10) constructed on the client side, 11) in tenths of of a seconds, 12) using fonts available on the client side, 13) by a deterministic algorithm, 14) implemented in JavaScript.

Although it may seem that in most cases simplifying choices were made, we still end up with a perplexing $\mathcal{NP}$-hard combinatorial problem. Thus, it can be expected that optimum solutions (e.g. in the sense of used area) can be delivered only by exponential-time algorithms. Solving a problem formulated

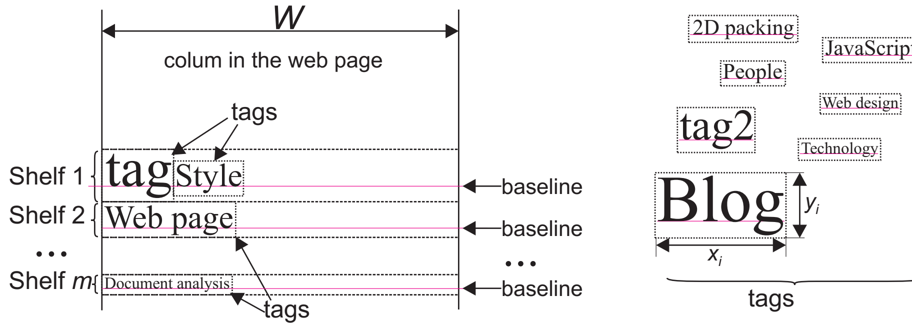

Figure 3.4: The relationship between tags and shelves.

according to the above recommendations encompasses Bin Packing or Strip Packing problems which can be practically solved by use of, e.g., shelf algorithms or metaheuristics[22].

3.2.6 Analysis of Packing Problem Properties

Let's start with determining the type of packing problem we are dealing with. As discussed in Section 3.2.4: 1) cloud shape is rectangular, 2) tags are rectangular boxes, 3) tags are reordered with packing, 4) rotation is disallowed, 5) tags are aligned to baselines. Fulfilling the first two requirements requires solving a 2-dimensional packing problem [13, 56, 77, 99]. However, not all 2-dimensional packing formulations are applicable here. For example, in the *2-dimensional Bin Packing* (2BP) formulation [77] both dimensions of the cloud should be fixed. A difficulty is that it is an $\mathcal{NP}$-complete problem to decide whether certain cloud dimensions are feasible to accommodate set $\mathcal{T}$, i.e. if the tags fit in the given box on the web page. Contrarily, in the rectangle packing formulation it is required to determine the smallest area bounding box enclosing the tags [56]. A disadvantage is that this formulation makes cloud dimensions variable and the web-designer would not be able to safely position the cloud on the web page. In the *2-dimensional Strip Packing* (2SP) formulation, tags are put on an infinite strip with one dimension fixed. This formulation is more practical because it is always possible to fit tags on such a strip if its length is floating. Thus, in the presented formulation it is conventionally assumed that tags are put in a column of width W and flexible height H. All three presented approaches to 2-dimensional packing are $\mathcal{NP}$-hard which implies that according to the current state of knowledge they can be solved either to optimality by an exponential running time algorithm or by a polynomial-time heuristic.

Let us now return to the canons of typography for typesetting beautiful text. One of such rules is the use of the baselines. This rule has already been introduced as the requirement 5. A conjunction of the strip packing and the requirement of baselines situates the analyzed problem as the so-called level or *shelf packing* [7, 77, 99].

Shelf packing means that algorithm packs the tiles as if on shelves cut from the strip: The bottom lines of the tiles are aligned to the bottom of the shelf. In each row, rectangles are aligned and the highest rectangle determines the bottom line of the next row (see Fig.3.4, compare also Fig.3.1). It is required that total width of the rectangles on no shelf exceeds the width of the strip. Thus, shelf algorithms are 2-dimensional renditions of 1-dimensional Bin Packing methods. By convention we will be referring to the rows as to *shelves*.

Other recommendation from typography is balancing tonal weight distribution on the page (also called the typographic color distribution). This means an even distribution of the mass of gray in the case of black letters on white background [19, 43]. A human typographer usually has to squint to asses tonal weight distribution. An advantage of this idea is that color can be measured in HTML/JavaScript by reading pixels using canvas. Then, tonal weights of the entire cloud, or its sections, can be calculated from the weights and sizes of the tags. Dispersion of the tonal weight will be the objective guiding the construction of a tag cloud.

3.3 Problem Formulation

In this section Tag Cloud Construction Problem (*TCCP*) is formulated. What is novel in presented approach, is resorting to the canons of typography used to typeset readable and aesthetic text. Unfortunately, mathematical models for the canons of beauty are rare. Still, tag clouds construction will be modelled as a discrete optimization problem with a particular objective function.

Let's assume that set of tags $\mathcal{T} = \{t_1, ..., t_n\}$ is given. Tag t_i is defined by the advertised phrase, font (i.e. font-family, style, size, weight, etc.), and hyperlink url address. This research abstracts away from how the importance of the phrase is transformed into the font attributes. They are assumed to be given. Phrase and font determine sizes x_i and y_i of tag t_i as well as its tonal weight a_i. In practice, sizes x_i, y_i and tonal weight a_i can be read with JavaScript, or its extensions. Tags are basically monochromatic which means that colors are not used to distinguish the tags.

Symbol	Definition	Role
a_i	tonal weight of tag t_i	input parameter
α_j	tonal weight of shelf j	
d_i	density of tag t_i	input parameter
H	height of the tag cloud	
m	number of shelves	
m_i	mass of tag t_i	input parameter
n	number of tags in $\mathcal{T}$	input parameter
O	dispersion in tonal weights between shelves	objective function
s_i	shelf of tag t_i (can be expressed as Z_j)	decision variable
$\mathcal{T}$	set of tags	input parameter
W	width of the tag cloud	input parameter
x_i, y_i	sizes of tag t_i	input parameter
Z_j	set of tags on shelf j (can be expressed as s_i)	decision variable

Table 3.3: Summary of the notation for the Tag Cloud Construction Problem. Variables α_j, O derived from tonal weight a_i are real numbers, but coming from a finite set. Remaining variables are discrete.

In more detail, for the pixel at coordinates x, y weight is calculated from:

$$b[x,y] = 1 - \frac{R[x,y] + G[x,y] + B[x,y]}{3 * 255} \tag{3.1}$$

where $R[x,y], G[x,y], B[x,y]$ are values of the pixel color components in bytes. For the sake of simplicity the above equation is used as an averaging method of producing grayscale from RGB. Other methods like lightness or luminosity can be used easily where desired. For the time being, two ways of defining contribution of the tag to the tonal weight will be considered. Let mass of tag t_i be

$$m_i = \sum_{1 \le x \le x_i, 1 \le y \le y_i} b[x,y], \tag{3.2}$$

and let density of tag be

$$d_i = \frac{m_i}{x_i y_i}. \tag{3.3}$$

Let's denote by s_i the index of the shelf where tag t_i is placed and Z_j the set of tags placed on shelf j, i.e.: $Z_j = \{t_i : s_i = j\}$. It is required that tags assigned to shelf j do not exceed width of the tag cloud W:

$$\sum_{t_i \in Z_j} x_i \le W \qquad \forall\, j = 1, \ldots, m. \tag{3.4}$$

Given set Z_j of the tags, the tonal weight of shelf j can be calculated as:

$$\alpha_j = \frac{\sum_{t_i \in Z_j} a_i}{h_j W}, \tag{3.5}$$

where $h_j = \max\{y_i | t_i \in Z_j\}$ is the height of shelf j and a_i is tonal weight of tag t_i equal either to its density d_i or to its mass m_i. For the time being two ways of defining the tonal weight will be allowed: either as mass m_i (3.2), or as density d_i (3.3). The final choice will be made in Section 3.5.2. Note that free space in the shelf will impact value of α_j. For example, if tags have large differences in their heights or the shelf is hardly filled, then large empty areas shall result in small α_j. A solution, i.e. a tag cloud, is the set $c = \{Z_1, \ldots, Z_m\}$ of tag assignments to shelves $1, \ldots, m$. The objective function guiding the dispersion of tonal weights between shelves will be:

$$O(c) = \sum_{j=1}^{m} (1 - \alpha_j)^k = \sum_{j=1}^{m} \left(1 - \frac{\sum_{t_i \in Z_j} a_i}{\max\{y_i | t_i \in Z_j\} W} \right)^k, \tag{3.6}$$

where exponent $k > 0$ is constant and c is a tag cloud. Function $O(c)$ will be minimized by changing decision variables:

m – the number of shelves, and Z_j – the tag-to-shelf assignment, subject to constraints (3.4). It can be intuitively expected that objective (3.6) favors solutions with fewer shelves m and bigger values of α_j, hence, more densely packed shelves.

Note that in this formulation neither shelves ordering nor tags ordering on the shelves is assumed. These can be rearranged after the packing, in the post-process step. Basically there will be three options of sorting shelves that are highest, top to bottom, bottom to top or middle to borders. With that, shelves with bigger (i.e. more important) tags can be moved to the areas more frequently scanned by users. Let us also observe that the goal of obtaining a good tonal weight embodied by irregular objective (3.6) makes or TCCP different from the classic 2-dimensional Bin Packing problems.

3.4 Algorithms for Tag Cloud Optimization

This section outlines algorithms proposed for constructing tag clouds. All these algorithms must meet the requirement of very light computational demands imposed by the browser platform. Before proceeding to the details of the algorithms let us explain their position in the tag preparation workflow (see Fig.3.5). The tags and their weights are obtained by periodically analyzing the documents, or other data sources for the considered field of application. A web designer composes a web page, and in particular, chooses font attributes of the tags in the tag cloud. The font attributes have the form of the CSS classes for the tags of certain weight. The tags and the CSS classes are merged in a web

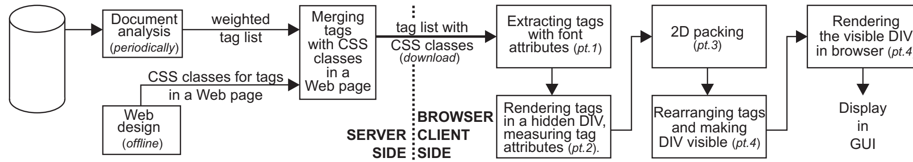

Figure 3.5: Tag preparation and exposition workflow.

page to be downloaded by the client browsers. The above steps are conducted on the server side in the web page preparation process. On the client side, a generic method constructing a tag cloud progresses in the following steps:

1. reading set of tags $\mathcal{T}$;
2. measuring tags sizes, masses, densities to obtain tonal weights;
3. optimizing the assignment of the tags to shelves;
4. delivering tag cloud c in the target format.

As explained in the previous section attributes of the tags rendered in a browser (sizes, mass, tonal weight) are read by use of JavaScript and canvas element in HTML. The output format of the tag cloud c is a sequence of HTML elements representing shelves, e.g. <div> elements, comprising tags as text elements most often with <a> hyper-reference (url address). Output must also include proper CSS formatting. In the following alternative algorithms are introduced for solving TCCP.

3.4.1 Branch and Bound

Branch and bound (B&B) is an exhaustive search method. Unfortunately, for $\mathcal{NP}$-hard problems (as ours) exhaustive search algorithms run in time exponential in the size of the problem. Consequently, a B&B algorithm can be applied only for instances of limited number of tags n, as a reference allowing to measure optimality gap of other algorithms. A B&B algorithm is defined by the branching and bounding schemes. The branching scheme determines the way of enumerating possible solutions. It can be envisaged as construction of and search in a tree. In this case a tag cloud, i.e. a solution, is defined by an assignment of tags t_i to shelves. Consider some node η_i of the B&B tree representing a partial solution with $0 \leq i < n$ tags already assigned to μ_i shelves, where $\mu_0 = 0$. Tag t_{i+1} can be assigned to one of the μ_i shelves (if it fits in the remaining widths) or a new (μ_i+1)th shelf may be opened for t_i. Thus, there are μ_i+1 offspring nodes of η_i representing the assignments of t_{i+1} to the shelves. There are at most $|\mathcal{T}| = n$ shelves and at most $n!$ solutions may be visited. Since size

of the B&B search tree grows very quickly with n, various bounding methods are applied to prune the tree. In this case offspring of node η_i were pruned if μ_i exceeded the number of shelves m for some complete solution. Values of m and O were obtained, and updated, whenever the search reached leaves of the tree, i.e. all n tags have been assigned to some shelf.

3.4.2 Greedy Algorithms

A generic greedy shelf algorithm is formulated as follows:

GENERIC GREEDY SHELF PACKING
INPUT: tag sorting rule TSR, shelf choice rule SCR, set $\mathcal{T}$ of tags;
OUTPUT: tag to shelf assignment c;
1: c=NEW shelf assignment; // initialize tag cloud
2: l=tags in $\mathcal{T}$ sorted according to TSR; // order tags
3: WHILE $l \neq \emptyset$ // $\emptyset$ means here an empty list
3.1: t = first tag from l; remove t from l;
3.2: s = shelf for t from c according to SCR; // find best shelf for t
3.3: IF s=NIL
3.3.1: s=NEW shelf; // open a new shelf for t
3.3.2: $c = c + s$; // append the new shelf to assignment c
3.4: ENDIF;
3.5: $s = s + t$; // put tag t on shelf s
4: ENDWHILE
5: RETURN c.

Particular implementations of the greedy shelf packing are defined by the tag sorting and shelf selection rules TSR, SCR. These are presented in the following.

Tag Sorting Rules TSR Tags can be sorted according to one of the following orders:

- increasing or decreasing masses m_i of the tags (denoted im, dm, respectively),
- increasing or decreasing densities d_i of the tags (id, dd),
- increasing or decreasing widths x_i of the tags (iw, dw),
- increasing or decreasing heights y_i of the tags (ih,dh).

The above tag sorting rules generalize ordering by decreasing item height considered, e.g., in [22, 34]. All tag sorting rules have complexity $\mathcal{O}(n \log n)$.

Shelf Choice Rules SCR One of the following rules to choose shelves for tags was applied:

- use the shelf with the smallest remaining horizontal space after inserting the tag (Best Fit, BF);
- use the shelf with the biggest remaining horizontal space after inserting the tag (Worst Fit, WF);
- use the shelf with the smallest tonal weight (3.5) after inserting the tag (Smallest Tonal Weight, STW);
- use the shelf with the largest tonal weight (3.5) after inserting the tag (Largest Tonal Weight, LTW);

There are two additional options for assigning a tag which, if chosen to act, override the above rule.
Fit Zero (f0). If there is such a shelf j that after inserting the tag, the shelf will have at most ε free horizontal space, then choose j. If such a shelf is found, then shelf selection is finished. Here it was applied $\varepsilon = 0$ in pixels. f0 can be run in $\mathcal{O}(\log m) < \mathcal{O}(\log n)$ time per tag, which does not increase the overall order of complexity of the packing algorithms.
Fit Two (f2) rule is applied after choosing shelf s_i for tag t_i according to one of the above BF, ..., LTW methods. Let δ be the horizontal space remaining on s_i after inserting t_i. If $\delta < \min_{t_i \in l}\{x_i\}$ then check for a pair of tags t_a, t_b such that $x_i < x_a + x_b < x_i + \delta$. In other words, if putting t_i on s_i would leave less horizontal space than required by the narrowest remaining tag, then check if there are two tags t_a, t_b which both fit on s_i but leave less free space than t_i. If t_a, t_b are found, then they are placed on s_i, and tag t_i is returned to the list of tags l. f2 rule, if applied exhaustively, would require complexity $\mathcal{O}(n^2)$ for checking all pairs of tags. In order to reduce the complexity of f2 a simplified search for t_a, t_b was implemented. The tags are sorted according to their widths x_i. We look for two tags t_a, t_b in the list of unassigned tags l such that x_a is the biggest width not greater than $\delta/2$ and x_b is the smallest width not smaller than $\delta/2$. If $x_i < x_a + x_b < x_i + \delta$ satisfy the above conditions, then we are done. If not, then the rule fails to find t_a, t_b. In this way t_a, t_b are searched in $\mathcal{O}(\log n)$ time and the additional sorting tags according to their widths is done once in time $\mathcal{O}(n \log n)$.

Altogether there are $8 \times 4 \times 2^2$ implementations possible of the greedy shelf packing algorithm. It is feasible to run all the above implementations of the greedy algorithms and choose the best result. Such a combination of greedy shelf packing algorithms will be referred to as to *Super-Fit* (SF) algorithm. An advantage of SF is that it lessens the impact of the worst-case instances existing for the component algorithms.

3.4.3 Tabu Search

Tabu Search (TS) [49] is a local search method. Local search algorithms improve solutions by searching their neighborhoods. A neighborhood of a solution is defined by moves which can be applied on the current solution to obtain a new, hopefully better, solution. Tabu Search allows to escape local optima by using a so-called tabu list (actually a queue) of forbidden solutions. Our implementation of the TS is outlined below:

```
TABU SEARCH
INPUT: tag cloud c1;
OUTPUT: tag cloud c2;
1: i=0; c2 = c1, tabu = ∅;            // initialize data structures
2: WHILE i < IterLimit                 // is iteration number exceeded?
2.1:  r=best move for c1;              // the best move r must not be tabu
2.2:  IF r=NIL
2.2.1: RETURN c2;                      // no feasible move in c2
2.3:  ENDIF;
2.4:  execute move r on c1;            // c1 is updated
2.5:  IF |tabu| = TabuSizeLimit        // is tabu list size exceeded?
2.5.1: remove the oldest element from tabu;
2.6:  ENDIF;
2.7:  append c1 to tabu;
2.8:  IF O(c1) < O(c2)
2.8.1: c2 = c1;                        // record the best solution found so far
2.9:  ENDIF;
2.10: i = i + 1;                       // increase iteration counter
3: ENDWHILE;
4: RETURN c2.
```

The algorithm starts from the best solution c_1 obtained by some other algorithm. In this case c_1 is a solution constructed by the Super-Fit algorithm, i.e. the best from the $8 \times 4 \times 2^2$ greedy shelf packing algorithms. According to the above description TS method has control parameters: $IterLimit$ – the limit on the number of iterations, $TabuSizeLimit$ – the limit on the length of the tabu list. Tuning these parameters is subject of Section 3.5. Two types of moves are evaluated in line 2.1: *simple move*, and *swap* move. In the simple move each tag $t_i, i = 1, \ldots, n$ is tested for a relocation from its current shelf s_i to a different shelf. In the swap move, all pairs of tags $t_i, t_j, i \neq j$ are exchanged on their shelves. A move which was not tabu and offering the smallest value of the objective function of the new solution is applied. In order to reduce the time of verifying whether some solution is tabu, a dedicated hash function has been

Short name	Reference	Date accessed	No.of tags	No.of styles
Collection A (training set)				
Amazon	`http://amazon.com`	Jun 13, 2015	100	100
Chir.ag	`http://chir.ag/tags/`	Sep 15, 2015	100	95
Flickr	`https://www.flickr.com/photos/tags/`	Sep 15, 2015	142	131
WeDeWa	`http://webdesignerwall.com/`	Jun 13, 2015	25	25
Wykop	`http://www.wykop.pl/`	Jun 13, 2015	35	31
Collection B (test set)				
NatDir	`http://nationaldirectory.co.uk/mod/tagcloud/`	Mar 10, 2016	70	5
VecMe	`http://pl.vector.me/tags`	Mar 9, 2016	100	18
WordPress	`https://en.wordpress.com/tags/`	Mar 9, 2016	188	178
ProfOWeb	`http://www.professionalontheweb.com/`	Feb 29, 2016	55	6
Metafilter	`http://www.metafilter.com/tags/`	Feb 29, 2016	150	13

Table 3.4: Test instances.

designed. Thus, when verifying tabu status of a potential new solution only hash values of the old solutions stored in *tabu* list and of the new one are checked. Computational complexity of one TS iteration is $\mathcal{O}(n^2 + TabuSizeLimit)$.

3.5 Computational Experiments

This section reports on the experiments in solving TCCP. Test instances are introduced first. The desirable objective function is elected. Tuning of the tabu method is outlined next. Finally, the performance of the heuristics in quality and runtime is compared. Unless stated to be otherwise, all tests were performed on a PC with Intel i7-3840@2.8GHz CPU, 32GB of RAM, Windows 7.0, Chrome 48.0.2564.116 browser. Greedy algorithms and Tabu Search were implemented in JavaScript. B&B algorithm has been implemented in Java 1.8.0_45 SE.

3.5.1 Test Instances

The experiments were performed on two collections, A and B, of test instances. Each collection comprises 5 sets $\mathcal{T}$ of tags gathered in Internet sources. Short names and characteristics of the instances are given in Table 3.4. For each set of tags $\mathcal{T}$, 11 test instances were created by setting tag cloud widths W from x_{max} to $2 * x_{max}$ with 10% progresses, where $x_{max} = \max_{t_i \in \mathcal{T}}\{x_{max}\}$ is the width of the widest tag in $\mathcal{T}$. Examples of the sets of tags are shown in Fig.3.6. Collection A has been used in tuning of the Tabu Search, while collection B has been applied in the final tests of all the algorithms. In this way training and testing algorithms on the same data was avoided.

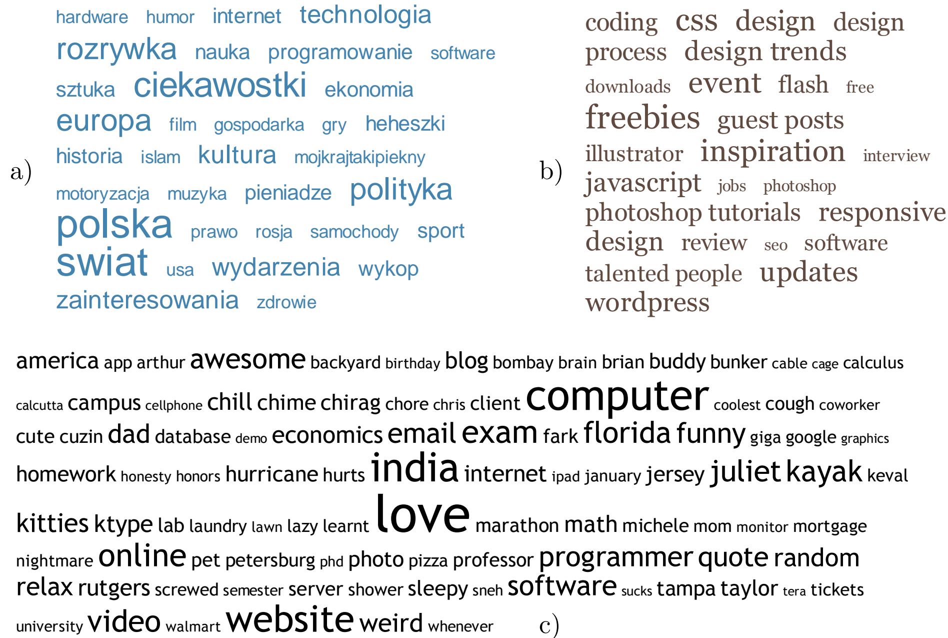

Figure 3.6: Examples of tag cloud instances. a) Wykop (resized, the biggest tag is 25pt, the smallest is 11pt), b) WeDeWa (the biggest tag is twice bigger than the smallest), c) Chir.ag (the biggest tag is 46pt, the smallest is 11pt).

3.5.2 Selection of the Objective Function

In Section 3.3 objective function (3.6) has been introduced which in its generality allows for two methods of defining tonal weights (either by tag mass (3.2) or by tag density (3.3)) and alternative ways of directing the tonal weight dispersion between the tags by different exponents k. Unfortunately, the above defining features of the objective function cannot be chosen by an analytical study of function (3.6) because human perception of tag cloud quality comes into play here. In order to omit this difficulty and select a sensible form of function (3.6) we decided that human experts will choose the most desirable objective function. Five experts evaluated all different tag clouds obtained by the B&B algorithm for a subset of $n = 16$ tags, for all 55 instances defined in collection A, two ways of defining tag tonal weight (equation (3.2) vs (3.3)), for exponents $k = 0.5, 1, 2$ in (3.6). Thus, each expert had to evaluate 55×6 tag clouds. The size of tag set $\mathcal{T}$ has been limited to $n = 16$ due to the computational complexity of the B&B constructing optimum solutions. For each instance in collection A tag clouds were constructed using 6 alternative objective functions and the 6 tag clouds were shown together to the experts. Experts voted for combinations of k and tonal weight quantifying method by choosing from the 6 clouds the one which was most aesthetically pleasing. The results of the voting are collected

Tonal weight	Exponent	E1	E2	E3	E4	E5	Sum
density	$k = 0.5$	33	22	23	18	29	125
density	$k = 1$	31	22	28	16	28	125
density	$k = 2$	31	23	22	19	26	121
mass	$k = 2$	8	17	16	22	16	79
mass	$k = 0.5$	13	19	13	20	9	74
mass	$k = 1$	12	18	13	21	9	73

Table 3.5: Goal function selection: votes awarded to goal functions by five experts.

in Table 3.5.2. Not in all cases were the tag clouds different. If tag clouds were the same for some combinations of k and tonal weight calculation method, and some expert chose one of such clouds, then each of the combinations of k and tonal weight calculation methods that built the identical cloud, received a vote. Consequently, the number of votes in Table 3.5.2 does not sum to 55 times the number of experts. It can be seen that experts clearly chose tag density (3.3) as the base of calculating tonal weights. This strong support may seem surprising because all tag clouds had the same number of shelves for the considered instance (combination of $\mathcal{T}, W$) and correlation between the number of shelves and values of the objective O is very strong (coefficient of correlation over 0.99 in all cases). And still, experts apparently chose density over mass. This has at least two consequences. Firstly, density seems more sensitive than mass to big vs small gradients of tonal weight present in high vs small tags. Secondly, the distinction between density and mass proves that the objective function grasps more than just the number of shelves. For the density as a measure of tonal weight, $k = 0.5, 1$ were preferred a bit over $k = 2$. Different values of k result in various pressure on uniformity set of α_j values. For $k < 1$, minimization of α_j dispersion between the shelves can be expected by minimizing (3.6). Hence, $k = 0.5$ has been chosen.

3.5.3 Super-Fit Algorithm

Performance of the SF algorithm is reported in Table 3.5.3. Due to high computational complexity of the B&B algorithm the relative distance to the optimum has been calculated only for the instances comprising the first $n = 16$ tags. It can be observed that objective O is numerically very close to the optimum if the same number of shelves is used by the SF and in the optimum solution. Only for instances Chir.ag at $W = 1.5 \times x_{max}$ and WordPress at $W = 1.2 \times x_{max}$ was SF not able to construct minimum number of shelves. In the first case SF used 6 shelves instead of 5 and in the second – 8 instead of 7. It increased the relative distance from the optimum to $\approx$ 20% and $\approx$14%, respectively. This

Test set	Relative distance to optimum	Time [ms]			
		Median	Max	Average per tag	Std.dev. per tag
A	0.3639362%	17	30	0.25	0.16
A+B	0.3119139%	17	57	0.24	0.10

Table 3.6: Performance of the Super-Fit Algorithm.

situation happened twice in all tests A and B. The longest registered runtime 57ms appeared for Flickr instance ($n = 142$), where the average for this instance was 19ms. The biggest instance WordPress with $n = 188$ has been solved in at most 51ms (40ms on average). Thus, execution times of SF are low which allows to use it as a constructor of a starting solution for Tabu Search. SF execution times and solution quality will be discussed in more detail in Section 3.5.6.

An interesting aspect of SF method is verifying which component greedy algorithms returned the best solutions most often. This is shown in Fig. 3.7 for test sets A and B together. If more than one algorithm returned the best solution, then each of the algorithms won on a given instance. Only algorithms which won at least twice are shown in Fig. 3.7. The names of the component algorithms shown along the horizontal axis follow the short-hand notation introduced in Section 3.4.2. For example, the algorithm with TSR decreasing tag height (dh) and shelf selection rule smallest tonal weight (STW) without fit 0 or fit 2 is used most often. It can be seen in Fig. 3.7 that the most frequently chosen tag sorting rules are based on the tag height (in 53% of cases). The next most frequently used tag sorting rules use tag mass (27%). The options f0, f2, were applied in 16% and 8% of the winning algorithms. Out of the 128 possible algorithms only 52 have ever provided the best solution of the SF. The algorithms with various combinations of f0, f2 mostly failed to win.

3.5.4 Tuning Tabu Search

Our implementation of TS has two control parameters: $IterLimit$ – the iterations number limit and $TabuSizeLimit$ – the limit of the tabu list length. The quality measures applied in TS tuning are runtime and objective function O. Tuning of TS control parameters is intended to keep the runtime low, while securing noticeable improvements in the objective function O.

Each instance from set A has been solved for $IterLimit = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$ and for tabu list size limit $TabuSizeLimit = IterLimit \times \{0.1, 0.2, 0.5, 0.8, 0.9, 1\}$. It means that the tabu list held from $0.1 \times IterLimit$ up to $IterLimit$ of the last moves. Thus, each instance from collection A has been solved with 19 different iteration limits

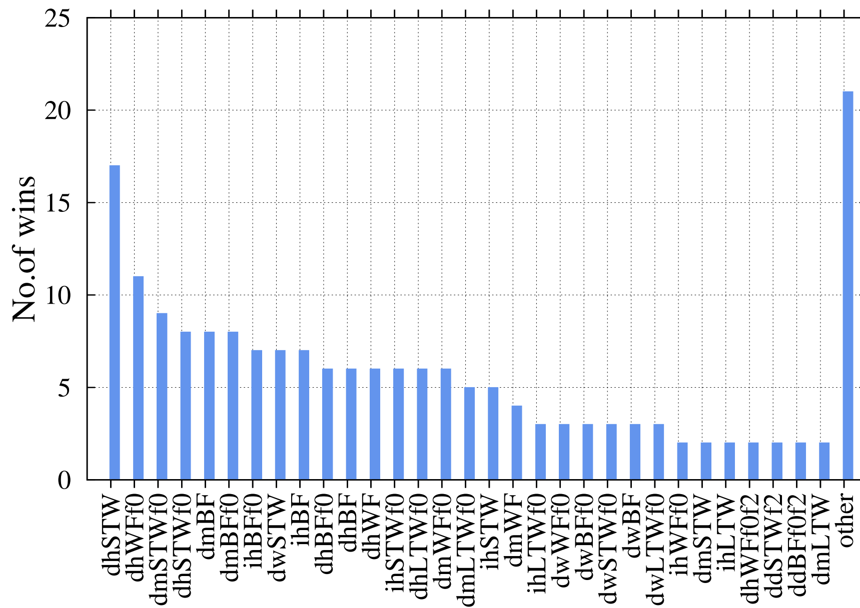

Figure 3.7: Super-Fit: frequency of giving best results by the component algorithms.

and 6 different tabu list lengths. Quality of the solutions generated by TS has been compared against the quality of B&B solutions. Since the execution times of the B&B are quite long, instances comprising the first $n = 16$ tags were used. Still, execution times of TS are presented for complete instances. It may be argued that time and quality performance were evaluated on different test sets, but it was observed that on instances of size $n = 16$ even for $IterLimit = 1000$ the median runtime was below 17ms, and the longest execution time observed (an outlier) was below 200ms. These times are too optimistic indication of the runtime especially as TS will be used on complete instances with up to hundreds of tags. Therefore, it was concluded that runtime on complete instances is a better indicator for TS tuning. Results of $IterLimit$ evaluation are shown in Fig.3.8 and $TabuSizeLimit$ in Fig.3.9. In Fig. 3.8a quartiles of TS execution times are shown. It can be seen in Fig. 3.8a that execution time of TS grows linearly with the number of iterations and for $IterLimit = 300$ median execution time is below 170ms. Note that this time includes 30ms taken on average by SF constructing the initial solution (see Table 3.5.3). Changes of solution quality with the number of iterations are shown in Fig. 3.8b. The average distance from the optimum solution obtained by the B&B method is shown on the vertical axis. The range of values of the objective function is very narrow in the numerical sense. For example, the distance from the optimum at the left end of Fig. 3.8b

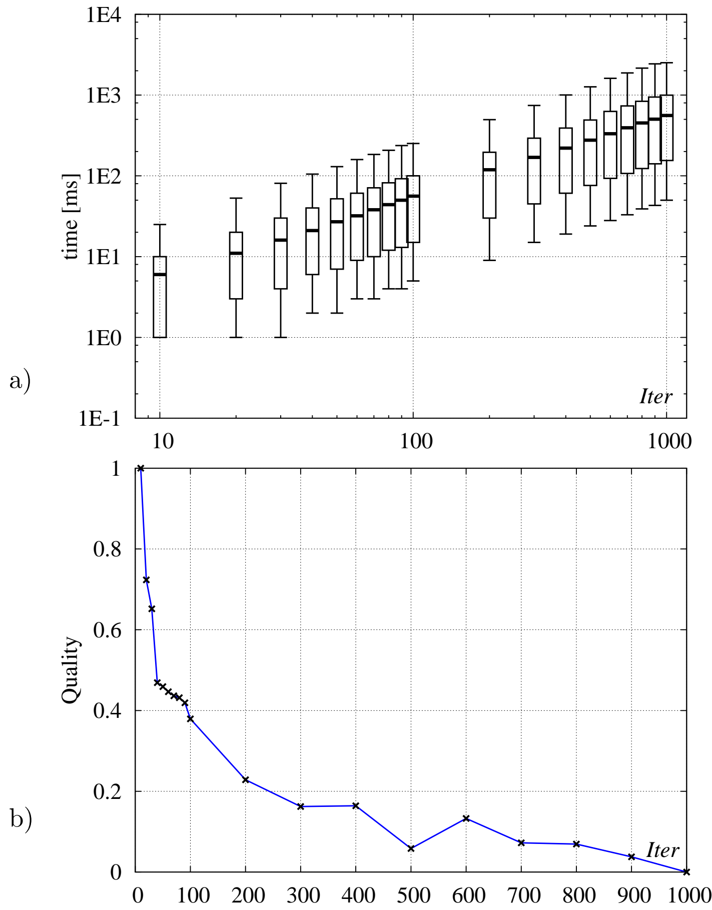

Figure 3.8: Tuning Tabu Search. a) Runtime vs $IterLimit$, b) Changes of objective O vs $IterLimit$ (distance from the optimum, scaled to range).

(at $IterLimit = 10$) is 0.363777% away from the optimum while the right end (at $IterLimit = 1000$) is 0.363746% away from the optimum. Hence, for better visibility, labels on the vertical axes of Fig. 3.8b, 3.9b are scaled to the range of the observed values. It can be seen that the value of the objective function is improved with increasing $IterLimit$, but around $IterLimit = 300$ the rate of changes slows down. The impact of $TabuSizeLimit$ on TS execution time is negligible on average (cf. Fig.3.9a). It can be concluded that the method of verifying tabu status of a solution, using a hash function, performs effectively. For the convergence of the objective $TabuSizeLimit \geq 0.5$ is sufficient (see Fig.3.9b). In the following tests $IterLimit = 300$ and $TabuSizeLimit = 0.5$ were used as a compromise between solution quality and runtime.

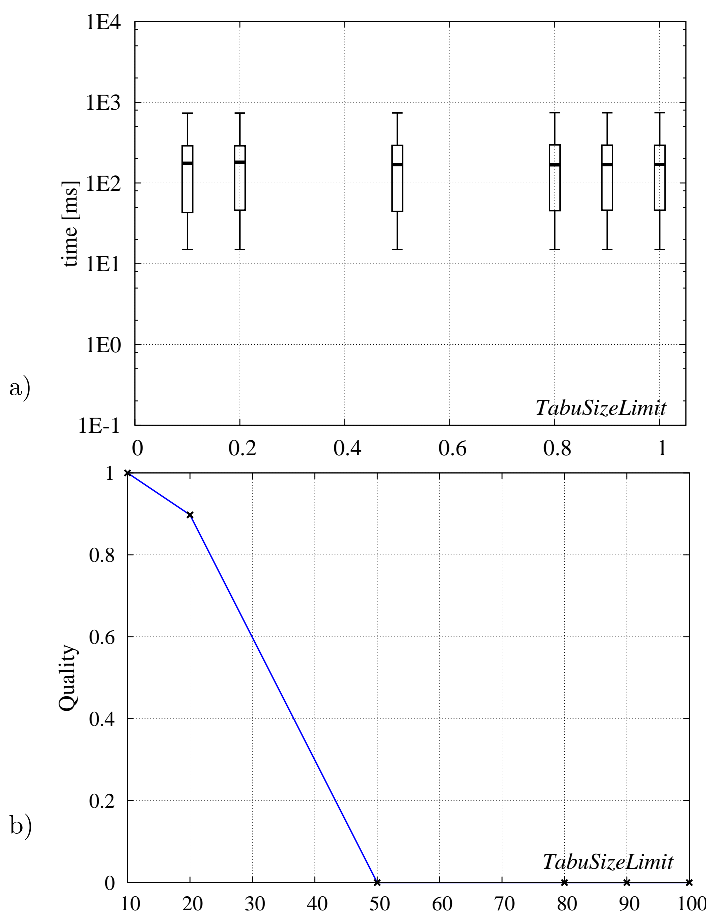

Figure 3.9: Tuning Tabu Search at $IterLimit = 300$. a) Runtime vs $TabuSizeLimit$, b) Changes of objective O vs $TabuSizeLimit$ (distance from the optimum, scaled to range).

3.5.5 Branch and Bound

The B&B runtimes are reported here to give the reader impression on the computational requirements of this algorithm. Time performance of the B&B is depicted in Fig.3.10. As it can be seen in Fig.3.10, at $n = 7$ computational costs related with enumeration of solutions outweigh the fixed overheads and quickly grow with n. On average, the B&B algorithm was solving instances with 20 tags in 62s. Unfortunately, solving a few worst-case instances for $n = 20$ required time exceeding 4 hours. It can be concluded that B&B cannot be applied to construct tag clouds in the client's browser because it takes too much time. Still, B&B can be applied as a provider of reference solutions for small-size instances.

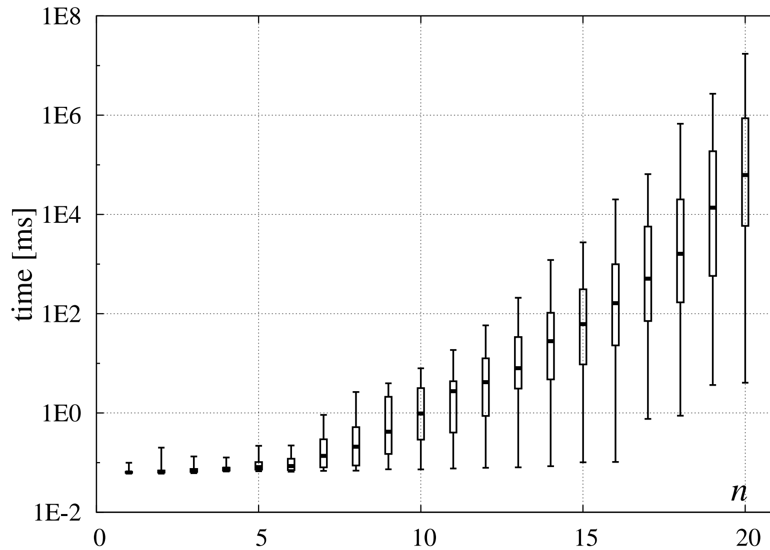

Figure 3.10: Running time of Branch&Bound algorithm.

3.5.6 Comparison of the Algorithms

In this section the trade-off between quality and runtime made by the algorithms are analyzed. In order to avoid giving advantage to the algorithms tuned on test set A the performance of all algorithms has been evaluated on test set B. The results of the comparison are depicted in Fig.3.11. Along the horizontal axis execution time of the algorithms is shown. Quality is shown along the vertical axis as the relative distance from the optimum solution for $n = 16$ (Fig.3.11a) or from the best obtained solution for the complete set of instances (Fig.3.11b). The range of the objective function is small in numerical sense if the minimum number of shelves is used. Therefore, relative distance in quality is shown on the vertical axis with 1E-10 as the unit of relative distance. Value 1 represents the optimum, or the best solution. Each algorithm is represented by a box and a median point (Q2) in time$\times$quality space. Boxes represent ranges between the first (Q1) and third (Q3) quartile of the execution time (horizontally) and quality (vertically).

The runtimes and solution quality scores for every greedy algorithm run separately on all instances have been recorded and the obtained population of results is presented under label "Greedy Algorithms". It can be observed that running all of the greedy algorithms together and choosing the best solution, which is done in Super-Fit (SF) algorithm, on average increases the runtime ≈ 60 times and decreases the distance from the best solution by two ($n = 16$) to four orders of magnitude (complete instances). The upper end of the box

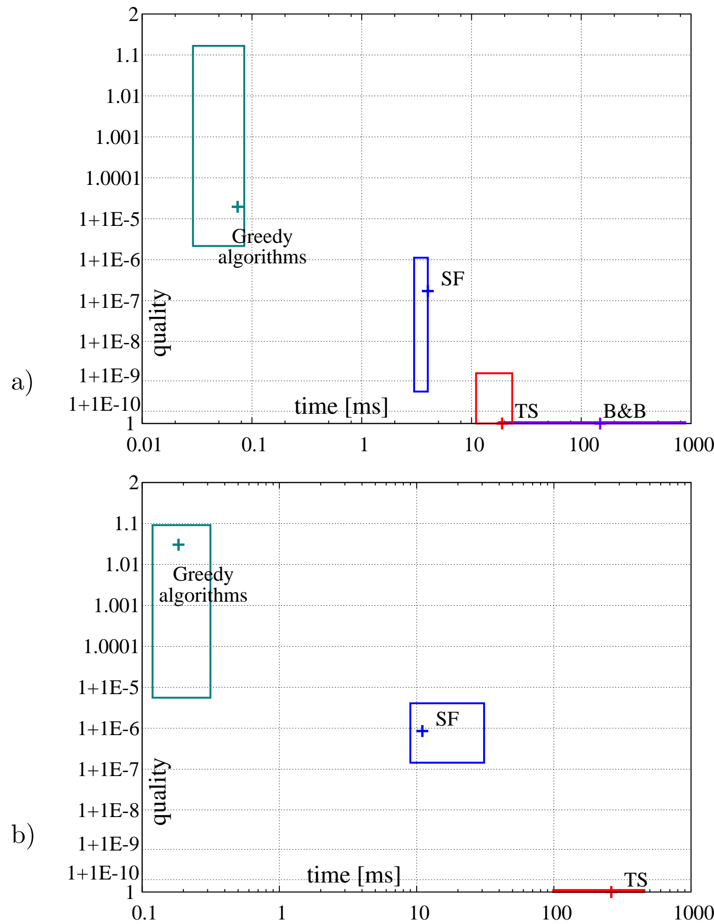

Figure 3.11: Juxtaposition of all algorithm performance: quality vs runtime. a) Instances limited to first $n = 16$ tags, b) complete instances. Log scales. 1E-10 is a unit of distance in quality.

for all greedy algorithms represents solutions with more than the minimum number of shelves. As the objective function heavily penalizes using additional shelves, the results of greedy algorithms are much worse with respect to the objective function, than the for the other algorithms. In the instances of set B for $n = 16$ more than 43% of solutions of greedy algorithms used more shelves than the minimum. For complete instances over 60% of greedy solutions used more shelves than the minimum observed. Contrarily, greedy algorithms are very fast. Super-Fit algorithm delivers the best of the greedy solutions at acceptably low runtime (Q2=11ms for complete instances, and the worst observed runtime was 48ms). Tabu Search (TS) improves the results of SF on average by at least two orders of magnitude. Execution time of TS is acceptable on average (Q2=262ms for complete instances) though it has increased a bit compared to

Figure 3.12: Tag clouds built by the presented algorithms. a) Wykop by SuperFit, ordering of the highest rows from the middle. b) WeDeWa and c) Chir.ag by Tabu Search the highest rows from the top.

test set A. This could be expected because test set B comprises instances with more tags than the training set A. The worst-case observed runtime for TS was 1436ms which is hardly acceptable in practice. A partial solution may be retuning of TS if instances bigger than considered here are encountered in practice. Unfortunately, standard computer systems (i.e. non-real-time ones) give no guarantees of the limits on the process/thread execution times and runtime distributions have long-tails. Consequently, it is not possible to eliminate all such worst-case runtimes altogether. On the other hand, TS is the second best in delivering quality solutions after B&B.

The advantages and disadvantages of the proposed methods revolve around the two dimensions of the runtime and the solution quality. Fig.3.11 is a good illustration of the trade-off between them. It can be observed that proposed algorithms form non-dominated groups of methods with respect to the quality-runtime trade-off: Greedy algorithms are fastest but their solutions are the worst on average. However, choosing the best of them (as in SF) already significantly improves solution quality. TS provides still better solutions at increased cost. Finally, B&B offers optimum solutions albeit at computational cost unacceptable for the applications considered here.

For the end of this section let us present visual examples of tag clouds constructed by the proposed algorithms. The tag clouds built from instances presented in Fig. 3.6 are shown in Fig. 3.12. Observe that apart from improving aesthetics and readability of tag clouds their height is reduced as a beneficial side effect.

3.6 Conclusions and Future Work

In this chapter online construction of the tag clouds for the websites has been considered. While other tag cloud building problems met some interest in the past, the website application had only a few ad hoc approaches. The tag cloud construction has been formulated here as a 2-dimensional Strip Packing problem with irregular objective function embodying canons of typography to control aesthetics of the generated clouds. Moreover, requirements and restrictions of the field of application force building tag clouds on the client side which introduced further restrictions of the computing platform (the browser) and the runtime. Two algorithms developed here – Super-Fit and Tabu Search – have practical applicability. They run in dozens of milliseconds (SF) or a few hundreds of milliseconds (TS), hardly ever use more shelves for tags than the minimum, and they build visually acceptable tag clouds.

It can be concluded that from the algorithmic point of view, tag cloud construction problem is virtually solved. However, as stated earlier mathematical models for the rules of beauty are rare and difficult. Hence, better representation of tag cloud aesthetics (even beauty) in closed-form expressions such as objective functions should be recommended as the subject of the further study. Devising such functions condensing in a low computational complexity expression the connection between tag cloud features and human perception is a challenge in itself. The methodologies developed here can be applied to solve problems of similar nature, such as procedural web page layout and content construction, infographics construction, game content generation, etc. These can be guided by optimization processes using aesthetics models expressed as objective functions. All this is doable online in the heterogeneous and volatile medium of information delivery: the browser.

4 CSS-sprite Packing

4.1 CSS-Sprites and Loading of Web Pages

In this chapter novel methods for the generation of CSS-Sprites to offload web servers and speed up web pages loading will be provided.

Short web page load time has a great importance for the Internet industry [87, 125]. Contemporary web pages are heavily loaded with small images (icons, buttons, backgrounds, infrastructure elements, etc.) and it is reported in [63] that 61.3% of all HTTP requests to large scale blog servers are images. Other static content constitutes only 10.5% of requests. Each image is a resource which must be downloaded from a web server. The interaction with a web server has a relatively long constant delay (a.k.a. latency, startup time) resulting from, e.g., traversing network stack by the messages carrying the request, request processing at the server, locating resources in server caches, etc. Fetching many images separately multiplies such fixed overheads and results in extensive web page loading time. CSS-sprite packing is a technique used in web design to overcome disadvantageous repetition of web interactions and improve performance of displaying web pages. The many small images, called *tiles*, are bundled into a single picture called a tile set, a sprite sheet, or simply a *sprite*. The sprite is loaded once and hence the constant delay elapses only once. An additional advantage can be taken in preloading images used in the web page interaction animations. In such animations appearance of a graphical element can be changed in almost no time because there is no communication delay of downloading a different view of the element. Sprites improve performance of the web servers too. Each interaction with a browser requires an overhead at the server. Reducing the number of the interactions by supplying a sprite once lowers the server load. Consequently, CSS-sprite technique is widely used in many web pages. An example of applying a CSS-sprite is shown in Fig.4.1. A

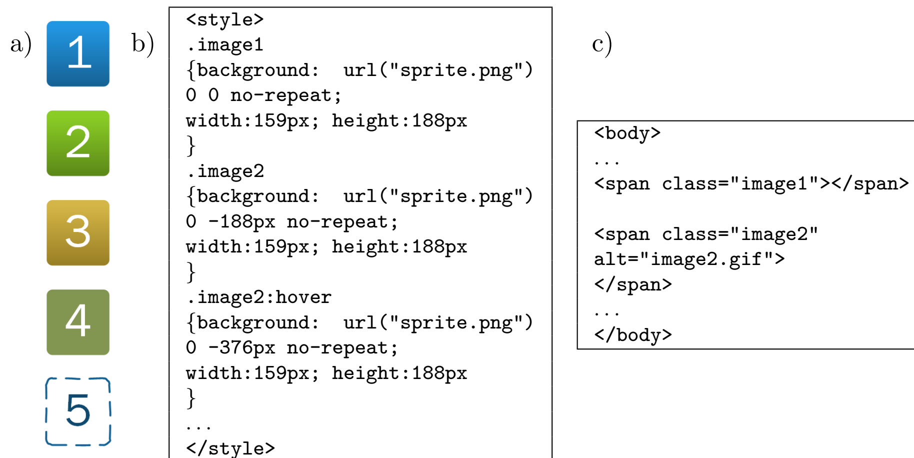

Figure 4.1: Example of CSS-sprite. a) sprite.png image, b) part of the CSS file locating images, c) example of use.

sprite is shown in Fig.4.1a. In order to extract tiles from a sprite Cascading Style Sheets (CSS) are employed in Fig.4.1b. Example code using the tiles in the sprite is shown in Fig.4.1c.

To the best of the author's knowledge the first reference to CSS-sprite packing appeared in [114] and it has been later popularized in [109]. CSS-sprite packing rests in the area of web development practice rather than in the sphere of scientific research. It seems quite common situation in web engineering, as discussed in the introduction and, e.g. in [16, 86]. Contemporary CSS-sprite generators pack all tiles into a single sprite, optimizing geometric area, if anything. This indeed reduces the number of server interactions, but at the risk of increasing file size, transmission time and slowing web page rendering. One of the main ideas behind this research is to allow to pack website tiles into multiple sprites for optimization of loading time. CSS-sprite packing is a practical problem with multiple facets involving image compression, complex distributed system modeling, solving combinatorial optimization problems. These problems are tackled in the following sections. In the next section realities and the challenges in sprite packing are discussed, then the CSS-sprite Packing Problem (*CSS-SPP*) is formulated. Results of preliminary empirical studies conducted to define the solution algorithm are presented in Section 4.3. In Section 4.4 current techniques for packing sprites are outlined. A new method of sprite packing is given in Section 4.5 and evaluated in Section 4.6. The last section is dedicated to conclusions. The notation used throughout this chapter is summarized in Table 4.1.

4.2 Practical Challenges and Problem Formulation

Before formulating the CSS-sprite Packing Problem let us discuss our goals and technical constraints. This analysis serves representing CSS-SPP as an optimization problem. Given a set of images (tiles) in various file formats, the intention is to combine them into a set of sprites for minimum browser downloading time. Factors determining the downloading time can be arranged into groups of: (i) geometric packing, (ii) image compression, (iii) communication performance. The three factors are tightly interrelated which will be shown in the following discussion. There are certainly also other factors related to the browser (e.g. rendering efficiency), server (e.g. cache performance), etc., but constructing a comprehensive model of their works is beyond the scope of this research and is taken into account only implicitly.

4.2.1 Geometric Challenges

One of the factors affecting sprite size(s) is geometric *layout* of the tiles. The layout means here mutual alignment of the tiles on the plane is meant here. It determines shape, size and location of empty spaces, and consequently, the total number of pixels in the sprite. The total number of sprite pixels will be called sprite *area*. Sprite area (in px) strongly correlates with the size (in bytes) of the sprite converted to a file or a message. When optimizing sprite area it can bee seen as a class of regular 2-dimensional packing problems because tiles and sprites are rectangles. Rotation of images is not allowed. Though it is technically possible to rotate images using CSS, tile rotation has not been used in CSS-sprite packing so far for the lack of compatibility with older browsers.

The problem of optimizing a layout of 2-dimensional objects for minimum space waste has been tackled very early in glass/paper/metal sheet cutting, in packaging, factory-floor planning, VLSI design, etc. [6, 31, 48, 77, 99]. Needless to say that 2-dimensional cutting/packing problem is computationally hard (precisely $\mathcal{NP}$-hard). In practice, it is solved by heuristic algorithms. Unlike in the above classic applications, in sprite packing one does not use any material sheet which (i) should be conserved, (ii) would impose a *bounding box*. Hence, it may seem that arbitrary tile layout is as good as any other. For example, the sprite in Fig.4.2a has a lot of *waste space* not encoding any tile. It may be argued that the layout in Fig.4.2a is as good as the layouts in Figs 4.2b,c because algorithms used in image compression are capable of dealing with such waste, i.e. with repeating equal pixels. In reality it is more complicated because various

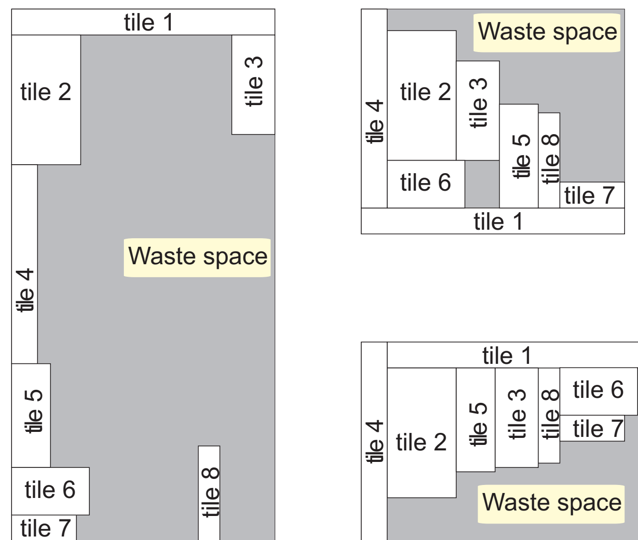

Figure 4.2: Examples of CSS-sprite layouts. a) excessive waste space, b) vertical layout, c) horizontal layout.

compression strategies used for this purpose have diverse efficiency. Encoding equal pixels is not completely costless because the information about the pixels must be stored to reconstruct them. Moreover, sprites must be decompressed to a bitmap in the browser. Consequently, waste space drains memory. Excessive memory usage affects browser performance. Hence, there are advantages in not wasting space in the sprites.

Another geometric factor determining sprite area is its *bounding box*. It is possible to restrict sizes in both, in one, or none of the dimensions. Accordingly, three variants of 2-dimensional packing are distinguished [77]. In the 2-dimensional Bin Packing problem (2BP) both sizes of the box (the bin) are fixed and it is required to minimize the number of used bins. The 2BP is furthest from CSS-SPP because arbitrary bin sizes can be chosen and using many bins due to size restrictions has no practical sense here. In the 2-dimensional *strip packing* problem (2SP) the 2-dimensional objects are put on an infinite strip with one dimension fixed: either the width or the height [6, 77, 99, 116]. This representation is more attractive because numerous algorithms proposed for 2SP can be used. Moreover, there are two intuitive ways of defining the fixed dimension of the strip: either as the width of the widest tile, or as the height of the highest tile. The former case will be called *vertical layout* (see Fig.4.2a,b). Similarly, the the latter option will called a *horizontal layout* (see Fig.4.2c). In the *rectangle packing* problem (RP) the two dimensions are free to

change [56, 68, 70, 101]. It is required to find the smallest area bounding box enclosing a set of rectangles. Rectangle packing seems to be closest to CSS-SPP. A disadvantage is a smaller set of known algorithms for the RP problem.

The geometric challenges in sprite-packing can be summarized as follows:

- determining packing model (RP vs 2SP),
- determining bounding box, respectively, the strip fixed size,
- selecting packing algorithms,
- determining the assignment of tiles to sprites for good geometric packing.

4.2.2 Image Compression Properties

Image compression techniques and standards (GIF, PNG, JPEG) are essential elements of this study. However, introducing computer graphics compression technology is beyond the scope of this research. An interested reader is recommended to begin with, e.g., [35, 62, 104, 121]. Let us note that images can be delivered to a browser as data URIs inlined in HTML or CSS text documents [91]. This scheme is out of scope of this research and requires an independent study.

Methods of image compression introduce complex interactions impacting sprite size. Combining tiles for the best image compression is computationally hard in general. There are two examples given: Firstly, PNG and GIF image formats permit indexed colors. When the number of image colors is limited a color palette can be used. Then, for each pixel an index of a color in a palette is recorded. The number of bits per pixel can be smaller than if the colors were encoded independently for each pixel, while keeping *color depth* of the image. Consequently, images sharing a palette of colors, when combined into a sprite, can be stored with fewer bits per pixel. This requires determining the set of images sharing an indexed palette. Assume that set $\mathcal{T}$ of tiles is given and a subset $\mathcal{T}' \subseteq \mathcal{T}$ which can share a palette of some fixed size l must be determined. Determining maximum cardinality $\mathcal{T}'$ is $\mathcal{NP}$-hard which will be shown in the next Section 4.2.3. Secondly, compression algorithms in PNG and GIF formats analyze images line by line. If two tiles aligned horizontally have the touching border areas in the same colors then such pictures compress better than if the colors were different. Aligning tiles for maximum length of constant color is again $\mathcal{NP}$-hard as it will be shown in Section 4.2.3. Since selecting and aligning tiles for good graphical compression is computationally hard, we are bound to heuristics choosing the set of tiles and constructing the layout.

Lossy JPEG compression adds another dimension of difficulty: When a JPEG tile is supplied for sprite-packing, it must be converted to a bitmap, and then may be stored in a JPEG sprite. Such a transformation will be called JPEG *repacking*. Repacking and any other conversions into a JPEG file inevitably reduce image quality. The change may remain unnoticeable for a nonprofessional user if the compression ratio is small, but a high compression ratio results in various discernible artifacts. There are methods of artifact-free decompression [18], but still cartoon-like smoothing or staircasing effects are problems remaining to be solved. Chroma subsampling allows to reduce image size by lowering chromatic resolution. Thus, it is easy to build a JPEG sprite of small size by trimming image quality. However, it has two undesirable consequences: (i) It is hard to determine acceptable lossy compression settings, e.g. a threshold of compression ratio. (ii) Fair comparison of various software for sprite-packing is challenging because in most cases settings of lossy image compression are undocumented (cf. Section 4.4). Therefore, it is hard to assess whether small sprite sizes of some sprite-packing software are obtained at the cost of image quality, or by effectively exploiting opportunities for good geometric packing or for compression without quality loss. In JPEG compression pixels of touching tiles influence each other which may distort pictures reconstructed from a sprite. Some solution may be putting side by side tiles with similar pixels, which again is computationally hard (as discussed above for PNG/GIF), and its effects are unpredictable. Aligning tiles to JPEG block sizes can be only a partial solution because filling the blocks with some dummy pixels may result in the so-called ringing artifacts and eliminating them is a research subject [44, 102] and a current engineering challenge [38, 96].

Given some images, their sizes quite often can be further reduced by use of compression optimizers. Here it means that the sprites can be further processed for minimum size. This procedure will be called *postprocessing*. Compression optimizers reduce image headers, remove metadata, and most importantly, experiment with compression settings. For example, in JPEG there is a choice between the baseline and the progressive compression, for the latter different image divisions can be used. For PNG one of five filters can be applied to each pixel row, which gives numerous possible combinations. Both formats use Huffman compression which is impacted by the choices of frame size and methods of searching for repetitions (PNG 1.2 offers four). Some tools for PNG use LZMA or Zopfli algorithms as alternatives to Huffman coding. Since the settings resulting in the smallest file are data-dependent and hence a priori unknown, various compression arrangements are checked by brute-force or by some heuristic. This is an extensively experimental area and its chicanery is partially described in sources like [29, 30, 58, 59, 81, 110].

Choosing the bounding box or the width of a strip in the geometric packing may limit chances of putting some tiles together. Thus, the geometric packing implicitly affects image compression efficiency. Observe two consequences: (i) Building many sprites may be profitable because some pictures do not combine well and putting them in one sprite gives worse results than keeping them separated. (ii) Tile to sprite distribution has effect both on geometric packing and on image compression. Hence, the two aspects are mutually related: It may be profitable to use worse geometric packing for the benefit of better image compression or vice versa. However, the overall effect cannot be predicted.

The difficulties resulting from unpredictability of geometric packing and image compression can be overcome by trying many alternative solutions and choosing the best one. This may take several forms: trying various geometric packing methods (cf. Section 4.2.1), verifying alternative tile to sprite distributions, experimenting with different image compression settings. However, the process of image compression is time-consuming and limits the number of compression attempts that can be made. For example, it seems barely acceptable to verify a few hundred alternative ways of packing and compressing the tiles, but it would be far better if only a few dozens of such attempts were made. Furthermore, there are many fast algorithms for geometric tile packing [99], but it seems impractical to verify all possible sprites resulting from such geometric packings due to the computational complexity of image compression. Thus, there is a trade-off between achievable sprite size and the time needed to construct it.

The main challenges related to image compression can be summed up as follows:

- determining the assignment of tiles to sprites for good image compression,
- choosing satisfactory compression settings for each compression standard,
- finding satisfactory trade-off between sprite construction time and solution quality.

4.2.3 Computational Complexity

Here complexity of two subproblems of the CSS-SPP will be analyzed. First Max Pictures for Shared Palette problem and then Picture Alignment for Maximum Length of Constant Color.

Max Pictures for Shared Palette.

Informally, the problem of the maximum set of pictures for a shared palette consists in selecting as many pictures as possible from a given set such that their colors are covered by the shared palette of a limited size. Here $\mathcal{NP}$-completeness of this problem will be shown.

Consider set $\mathcal{I}$ of n images. Image i has palette (i.e. set) of colors p_i from some spectrum U of size $|U| = | \cup_{i=1}^{n} p_i|$. Thus, a palette of size at most $|U|$ is needed to index all colors of $\mathcal{I}$. Given is a limit $l < |U|$ on the shared palette size. The problem is formulated as follows:

Max Pictures for Shared Palette
Input: Set $\mathcal{I}$ of images with palettes $p_1, \dots, p_n$, shared palette size l, positive integer m.
Question: Is there a subset $\mathcal{I}' \subseteq \mathcal{I}$ such that $| \cup_{i \in \mathcal{I}'} p_i| \leq l$, and $|\mathcal{I}'| \geq m$, i.e. is it possible to cover at least m pictures from $\mathcal{I}$ by palette of size l?

Theorem 1 *Max Pictures for Shared Palette is $\mathcal{NP}$-complete.*

Proof. Max Set of Pictures for Shared Palette is in $\mathcal{NP}$ because NDTM can guess set $\mathcal{I}'$ in time $O(m) \leq O(|\mathcal{I}|)$, and verify whether $|\cup_{i \in \mathcal{I}'} p_i| \leq l$ in $O(|\mathcal{I}||U|)$ time.

Next, it will be shown that Balanced Complete Bipartite Subgraph (problem GT24 in [64]) polynomially transforms to our problem. The former problem is defined as follows:

Balanced Complete Bipartite Subgraph (BCBS)
Input: Bipartite graph (V_1, V_2, E), where V_1, V_2 are disjoint sets of vertices, E is set of the edges and positive integer k.
Question: Are there two disjoint sets $X_1 \subseteq V_1$, $X_2 \subseteq V_2$ such that $|X_1| = |X_2| = k$ and such that $u \in X_1, v \in X_2$ implies $\{u, v\} \in E$.

Thus, the question in BCBS asks for a biclique $K_{k,k}$. Let $n(i)$ denote neighbors of node $i \in V_1$. In the transformation from BCBS to Max Pictures for Shared Palette nodes of V_1 correspond with the pictures of $\mathcal{I}$ and nodes in V_2 with colors in U. Thus, we have: $|\mathcal{I}| = |V_1|$, $|U| = |V_2|$. Let's assume palette $p'_i = V_2 \setminus n(i)$ consisting of colors not used by image i, i.e. palette p'_i is a complement of the neighbors in $n(i)$. The question is if it is possible to cover $m = k$ pictures with a palette of size $l = |V_2| - k$. The transformation can be done in polynomial time $O(|E|)$.

Suppose the answer to BCBS is positive and the required sets X_1, X_2 exist. We construct set $\mathcal{I}'$ using pictures corresponding to the nodes of X_1. The palette p' has colors in $V_2 \setminus X_2$ and size $l = |V_2| - k$. Note that picture $i \in \mathcal{I}'$ uses colors

in $V_2 \setminus n(i)$ and hence *no* colors from X_2. Since $\forall i \in X_1, j \in X_2, \{i,j\} \in E$ the picture corresponding to i is using no colors from X_2 and a palette of size $|V_2| - k = l$ is sufficient to cover all pictures in $\mathcal{I}'$.

Suppose the answer to Max Pictures for Shared Palette is positive and set $\mathcal{I}'$ of $m = k$ pictures covered by a palette of size $l = |V_2| - k$ exists. It means that $|V_2| - l = k$ colors are not used by any picture in $\mathcal{I}'$ and have been eliminated from the palette. Since picture i uses colors which are complement of $n(i)$, the instance of BCBS has an edge from each node corresponding to a picture in $\mathcal{I}'$ to each node corresponding to the colors absent from the palette. Hence, nodes of X_1 corresponding to $\mathcal{I}'$ and the nodes of X_2 corresponding to unused colors form biclique $K_{k,k}$ and the answer to BCBS is positive. $\square$

Picture Alignment.

The problem of picture alignment may be formulated as follows: given a set of pictures align them horizontally for maximum overlap of colors on neighboring sides. Picture alignment problem has practical motivation. When packing pictures into a CSS-sprite some pictures will be in direct horizontal contact, i.e. their vertical edges touch each other. If a pair of neighboring pictures have edges of different colors then more data is stored to encode the different neighboring colors, than if the colors were the same. The best alignment of the pictures minimizes the number of color changes. It will be shown here that picture alignment problem is $\mathcal{NP}$-complete.

More formally picture alignment problem may be formulated as follows. Given is set $\mathcal{I}$ of n rectangular images. For the sake of conciseness picture features and graphical compression are very simple. Only pixels on the vertical sides of a picture matter for packing efficiency. Therefore, picture i is defined by the sequence of pixel colors l_i on the left side and the sequence of pixel colors r_i on the right side as if the pictures were 2 pixels wide. Both l_i and r_i are arrays, and color of x-th pixel, e.g., in l_i can be referred to as $l_i[x]$. We assume that all pictures have the same height. If $r_i[y] = l_j[y]$, i.e. y-th right-edge pixel of some picture y has the same color as the y-th left-edge pixel of picture j, then the cost of encoding the pair is 1. Otherwise, $r_i[y] \neq l_j[y]$ and the cost of encoding them is equal 2.

Picture Alignment
Input: Set $\mathcal{I}$ of n images of width 2 and height k with pixels l_i, r_i, for $i = 1, \ldots, n$, on the left and the right side respectively, positive integer f.
Question: Is there a sequence of images such that their cost of packing is not greater than f?

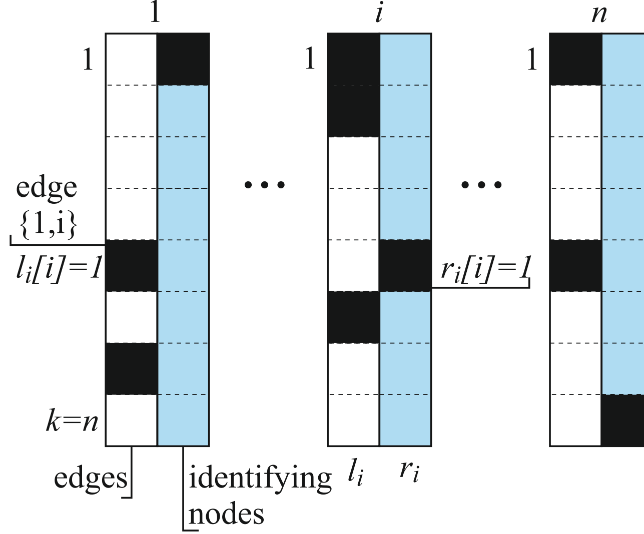

Figure 4.3: Pictures in the picture alignment problem

Theorem 2 *Picture alignment is $\mathcal{NP}$-complete.*

Proof. The problem is in $\mathcal{NP}$ because NDTM can guess the sequence of length n and calculate the cost of packing in time $O(nk)$. Next, we give polynomial time transformation from Hamiltionian Path problem [47, 64]:

Hamiltionian Path (HP)
Input: Graph (V, E), where V is the set of vertices, E is the set of edges.
Question: Is there a Hamiltonian path in G, i.e. a path visiting each node of G once?

In the transformation nodes of HP correspond with pictures. We will use three colors: 0, 1, 2. The set of the right pixels r_i serves for identifying nodes. Hence, $|\mathcal{I}| = n = |V|$, $k = n$, $r_i[i] = 1, \forall j \neq i, r_i[j] = 2$, for pictures $i = 1, \ldots, n$. Colors in l_i make for edges, $\forall \{i, j\} \in E, l_j[i] = 1$ and otherwise $l_j[i] = 0$ (cf. Fig.4.3). The question is whether it is possible to pack the pictures in $\mathcal{I}$ with the cost not greater than $f = (n-1)(2k-1)$. If $\{i, j\} \in E$ then pictures i aligned on the left of j have one neighboring equal pixel and cost of packing j after i is $2k-1$. Otherwise $\{i, j\} \notin E$ and all k pixels are different and the cost of packing j after i is $2k$.

If a Hamiltionian path exists in HP, then the sequence of pictures corresponding to the sequence of G nodes is used. The cost of packing is $f = (n-1)(2k-1)$. If a packing of cost not greater than $(n-1)(2k-1)$ exists then it means that the number of different colors in each aligned pair is $k-1$ because it is not

possible to have a smaller difference. This means that between each pair of nodes corresponding to the pair of pictures in the sequence there is an edge in G. Hence, Hamiltonian path also exists. □

4.2.4 Communication Performance

Since quality of a set of sprites should be measured as the downloading time, sprite(s) can be constructed to take advantage of communication channel characteristics. For example, a large constant delay in communication time encourages packing tiles in one sprite. Hence, the primary rule of web performance optimization has always been to minimize the number of HTTP requests. Still, if parallel communication is possible, then it may be advantageous to construct a few sprites and send them in parallel [112]. As mentioned above, in the ideal case downloading time measures sprite(s) quality. However, a number of circumstances make it close to impossible. Let us consider limitations to the perception of communication performance. Downloading time is determined by a chain of components: the browser, network communication stacks, network devices on the path from the client to the server, web-server queuing and buffering. A variety of browser, communication, server platforms exist which deal with messages in various ways. All these components are shared by activities with unknown arrival times and durations. Diverse scheduling strategies are used to dispatch them. Consequently, communication time is unpredictable and nondeterministic, which materializes in dispersion of performance parameters (see Section 4.3.2). It is not possible to use detailed methods of packet-level simulation to calculate sprite transfer time because such methods are too time-consuming to be called hundreds of times in the optimization process. Hence, in evaluating quality of a set of sprites it must be relied on performance models, such as flow models [117], preferably an easy to calculate formula, representing typical tendencies which can be reasonably traced. Thus, occurs a dilemma how to represent essential determinants of the transfer time in the tractable way. Our approach is detailed in the following.

Given a set of sprites sizes, three communication channel performance elements are considered to estimate transfer time: (i) communication latency, (ii) available bandwidth, (iii) the number of concurrent communication channels. It is assumed that one sprite is transferred over one communication channel but the this abstracts away from the specific packet exchanges. Communication latency (startup time) L is the constant overhead emerging in a sprite transfer time. Bandwidth $B(1)$ (e.g. in bytes per second) is the speed of transferring data between the web-server and the browser using one communication channel. Thus, according to this model, transferring x bytes of data over one channel

takes $L+x/B(1)$ seconds. Note that in this representation L implicitly covers all constant overheads, both in the communication channel and in the web-server. Similarly, bandwidth accounts for the speed of the communication channel and the server. Consequently, this network performance model encompasses all communication layers from the physical to the application layer. Browsers allow for opening a few concurrent communication channels to the web-server (cf. Section 4.3.2). This opens an opportunity to transfer sprites in parallel. It is assumed that one channel may transfer several sprites sequentially. The performance for parallel communications is ruled by sequencing them in the browser, packet scheduling in the network, sharing the communication path and bandwidth with other communications and with network protocols signaling. Hence, the total bandwidth is not increasing linearly with the number of used channels. Instead it is assumed that the total bandwidth $B(c)$ is a function of the number of simultaneously open channels c. Then, a single channel bandwidth share is $B(c)/c$. A vector of aggregate bandwidths for different numbers of channels will be denote by $\overline{B} = [B(1), \dots, B(c_{max})]$ Suppose that size of sprite i is f_i, for $i = 1, \dots, m$. The time of transferring the set of sprites $\mathcal{S}$ over c concurrent channels is modeled by the formula:

$$T(\mathcal{S}, c) = \max \left\{ \frac{1}{c} \sum_{i=1}^{m} (L + \frac{f_i}{B(c)/c}), \max_{i=1}^{m} \{L + \frac{f_i}{B(c)/c})\} \right\}. \quad (4.1)$$

In the above formula $L + f_i/(B(c)/c)$ is communication time of sprite i transferred via one of c channels. The first part of (4.1) is total communication time shared fairly over c channels. The second part is a duration of the single longest communication. Formula (4.1) represents communications like preemptive tasks scheduled on a set of c parallel processors in the scheduling theory [92]. Clearly formula (4.1) is an approximation. A simple communication time is assumed model because, as discussed above, the actual scheduling of communications is unknown. More detailed models of the transfer time (e.g. accepting certain sequencing of sprites in channels) are not justified without further disputable assumptions. An advantage of formula (4.1) is that it can be easily calculated in $O(m)$ time from sprite sizes without a need for more complex algorithms or simulations. Note that increasing the number of sprites m means increasing the number of HTTP requests. This is represented by mL in the first part of formula (4.1). Thus, (4.1) takes into account the trade-off between the opportunity of transfer time reduction by parallel communication and the cost of issuing an HTTP request for each sprite. Usually $B(c)$ is a nondecreasing sublinear function (see Section 4.3.2). Consequently, $B(c)/c$ is nonincreasing and (4.1) has maximum in one of two trivial cases $c = 1$ or $c = c_{max}$. Hence, to encourage

Symbol	Definition
$B(c)$	accumulated bandwidth of c concurrent communication channels
$\overline{B}$	vector $[B(1), \ldots, B(c_{max})]$
c	number of concurrent communication channels
c_{max}	maximum admissible number of concurrent communication channels
f_i	size of sprite i in bytes
k	number of intermediate tile groups (cf. Section 4.5.2)
L	communication latency (startup time)
m	number of sprites
n	number of tiles
$\mathcal{S}$	set of sprites
$\mathcal{T}$	set of tiles
$T(\mathcal{S}, c)$	communication time as a function of the set of sprites $\mathcal{S}$ and number of used communication channels c

Table 4.1: Summary of notation for the CSS-sprite Packing Problem.

applying a mild number of parallel communication function

$$T(\mathcal{S}) = \min_{c=1}^{c_{max}} \{T(\mathcal{S}, c)\} \tag{4.2}$$

will be used as the objective function evaluating quality of a set of sprites. It is not taken for granted that any aspect of the problem dominates download time, but by optimizing (4.2) a balance between the number of sprites, their sizes, overheads, and parallelism is stroken. However, certain optimization versions may be handled as special cases of (4.2). For $L = 0, B(c) = 1, c_{max} = 1$ total size of transferred data is minimized. Similarly, for $L = \infty, B(c) = 1, c_{max} = 1$ the number of communications is minimized, i.e. one sprite will be created.

For the end of this discussion let us note that communication performance has a "demographic" aspect. The website performance perceived by its user is impacted not only by the server, but also by factors on the user side such as the "last mile", the browser, the computer platform. Moreover, not one user visits the website but many and each of them can be different. Hence, there is a population of visitors and population of performance indicators. Members of the population create a specific profile of loading the server with communications. Thus, each website is unique with respect to its users population. In order to take the full advantage of performance optimization, parameters $L, \overline{B}$ should be measured on the actual web site and its viewers population. Section 4.3.2 demonstrates how this can be done in practice.

4.2.5 Problem Formulation

Let's summarize the introductory discussion by formulating CSS-sprite Packing Problem (*CSS-SPP*). Given is set $\mathcal{T} = \{T_1, \ldots, T_n\}$ of n tiles (images in standard image formats such as JPEG, PNG, GIF), communication link with latency L and bandwidths vector $\overline{B}$ of length c_{max}. Construct a set of sprites $\mathcal{S}$ such that objective function $T(\mathcal{S})$ as defined in (4.2) is minimum. Rotation of tiles is not allowed. Each tile is comprised in only one sprite. Each sprite is transferred in one communication channel.

Let's summarize possible advantages and costs implied by the above problem formulation. By using objective function (4.2) user-side performance perception is assumed. Applying more than one sprite allows to build better sprites and thus save on the total transferred data size and memory usage in browsers. Employing many sprites offers faster downloading by parallelizing communication at the cost of establishing many connections on the server. The interplay of communication performance and the sprite(s) determines efficiency of the solution. Hence, sprite construction is guided by the actual data: $n, \overline{B}, L$, tiles sizes and features. The number of sprites in the solution is not predetermined. Depending on the actual set of tiles and the performance data it may be a single or a few sprites. As observed in the previous section, a single sprite will be constructed if additional latencies outweigh benefits of parallel connections. It is also justified to consider separating significantly different classes of user browsers (e.g. mobile vs wired) and constructing different sprite(s) for each user class.

4.3 Preliminary Tests

As discussed in the previous section, a number of decisions must be made in designing a sprite-packing solution. This section reports on the impact of layout choice on the efficiency of the image compression and also presents results of network communication performance evaluation.

4.3.1 Packing Model

An *aspect ratio* of an image is the ratio of its vertical and horizontal sizes. Vertical and horizontal layouts may be considered the border cases of possible aspect ratios in this sense that one sprite dimension is fixed to a minimum. As noted in Section 4.2.2 the sprite aspect ratio may influence the efficiency of image compression. In order to examine the extent of such relationship, an experiment has been conducted. 36 sets of 36 rectangular tiles representing web icons,

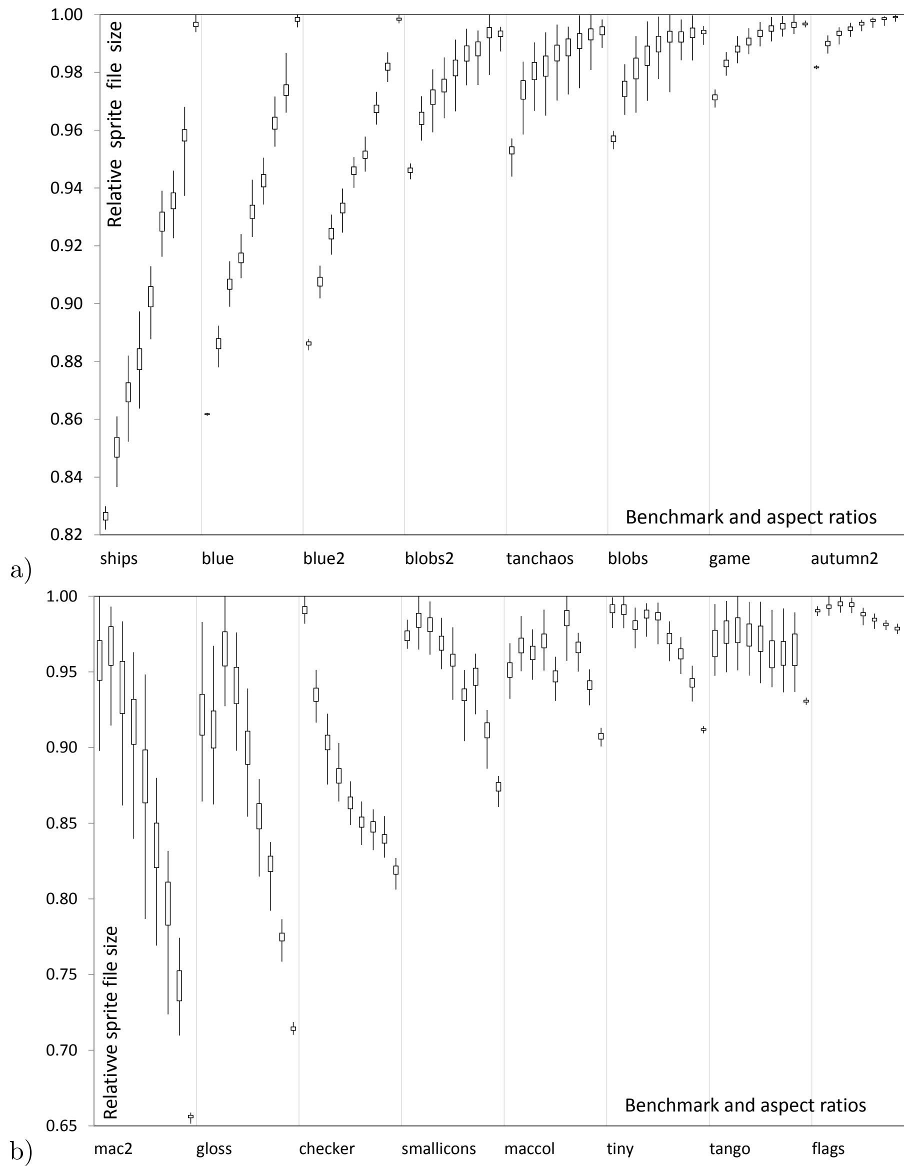

Figure 4.4: Instances with preference for a) vertical layout ($\frac{x}{y} = \frac{1}{36}$) b) horizontal layout ($\frac{x}{y} = \frac{36}{1}$).

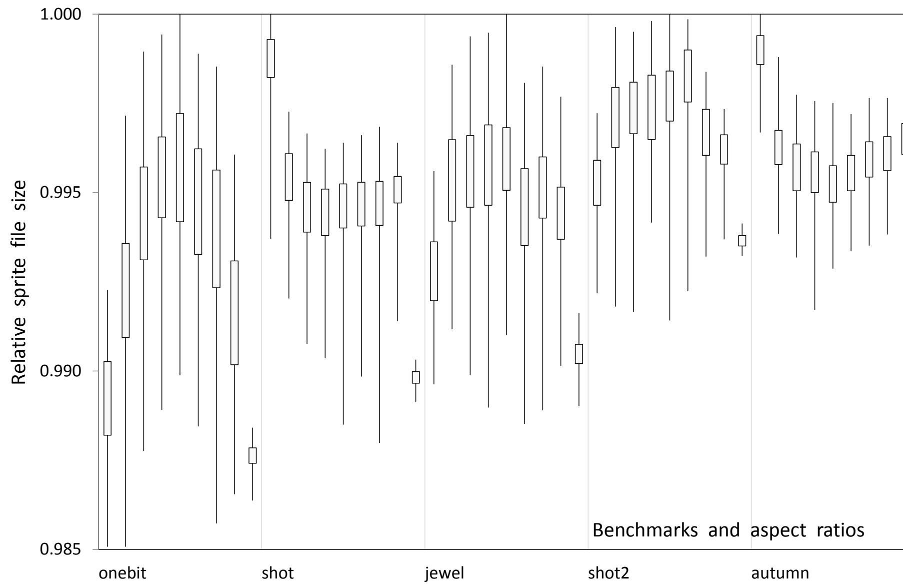

Figure 4.5: Instances without strong preference for any aspect ratio.

buttons and similar elements were collected from websites offering stock images. The sets had various colors, backgrounds, visual styles and sizes. In addition to sets with images coming from a single origin and hence with similar visual style, sets comprising images from different sources, distorted images and blank tiles to simulate wasted space were tested. Since in each test set tile sizes were equal, it was possible to pack them without real waste. The only waste was introduced intentionally in the test by using blank tiles. For 36 rectangles 9 aspect ratios were tested which conventionally represent the size of a sprite as a tile array in tile units. Thus, aspects ($\frac{x}{y}$): $\frac{1}{36}$ (a vertical layout), $\frac{2}{18}$, $\frac{3}{12}$, $\frac{4}{9}$, $\frac{6}{6}$, $\frac{9}{4}$, $\frac{12}{3}$, $\frac{18}{2}$, $\frac{36}{1}$ (a horizontal layout) have been examined. Since the mutual arrangement of the tiles may alter results of image compression, 200 random permutations of tiles were generated for each aspect ratio. Image manipulations were performed with GD Graphics library [17]. For PNG compression PNG_ALL_FILTERS setting was selected which means that in the construction of the compressed image scanline all compression filters were tried and the most effective compression filter was applied. Images were compressed with the strongest level 9 of DEFLATE method.

Results of the experiments with PNG images are shown in Fig.4.4-4.5. On the horizontal axis different data sets are presented, and for each data set aspect ratios are shown from $\frac{1}{36}$ to $\frac{36}{1}$. Along the vertical axis sizes of sprite files in relation to the size of the biggest sprite created for the given test set are given. The results from 200 permutations are shown as boxplots with minimum, first quartile (Q1), third quartile (Q3), and maximum. Note that Fig.4.4-4.5 have

different ranges on the vertical axes. For clarity of presentation only a subset of results is shown. It can be verified in Fig.4.4-4.5 that the data sets can be divided into three groups: with a preference for vertical layout (Fig.4.4a), with a preference for horizontal layout (Fig.4.4b), and data sets without any apparent preference for the aspect ratio (Fig.4.5). A preference means here that certain aspect ratio results in the smallest sprite sizes. Out of 36 data sets 17 had preference for horizontal layout, 14 for vertical layout, and 5 demonstrated no aspect preference. In the instances with preference of the layout the sprite sizes could be reduced by 2% to 35% from the worst to the best aspect ratio (Fig.4.4a,b). In the case of no correlation of file size with the aspect ratio, sprite file sizes could be reduced by less than 1.5% by selecting the aspect ratio. The above results give a strong argument that in case of PNG images it is justified to focus the examination of the geometric packing models on strip packing with vertical and horizontal layouts. Moreover, for the preferred aspect ratios the impact of tile permutations was always within 2%. It can be concluded that the neighborhood of the tiles has a relatively small impact on the sprite size and, e.g., the CSS-SSP algorithm does not need to examine swapping the same-sized tiles between their locations.

In the case of JPEG image format no similar preference has been observed. However, size of the output sprite was strongly correlated with the sprite area (number of pixels). Therefore, in the case of JPEG images it is advisable to eliminate unnecessary waste space.

4.3.2 Communication Performance

In this section it is demonstrated that performance parameters $L, \overline{B}$ introduced in Section 4.2.4 can be obtained in practice. Before proceeding to the results let us explain why using existing performance studies is problematic. As explained in Section 4.2.4 communication performance parameters should be measured on the particular web server and its user population. Consequently, latency and bandwidth results which could be obtined using tools like [124] are not adequate here becasue the sprites would be optimized not for the population of real users but for the benchmarking infrastructure. To the best of the auhthor's knowledge data on bandwidth scalability, here expressed in vector $\overline{B}$, is not available in the open sources. The number of per-domain parallel connections a browser may open is well studied [111], but it does not translate directly to the number of parallel channels c_{max} and bandwidth scalability in $\overline{B}$ because these are determined by the server, user platform, and the "last mile".

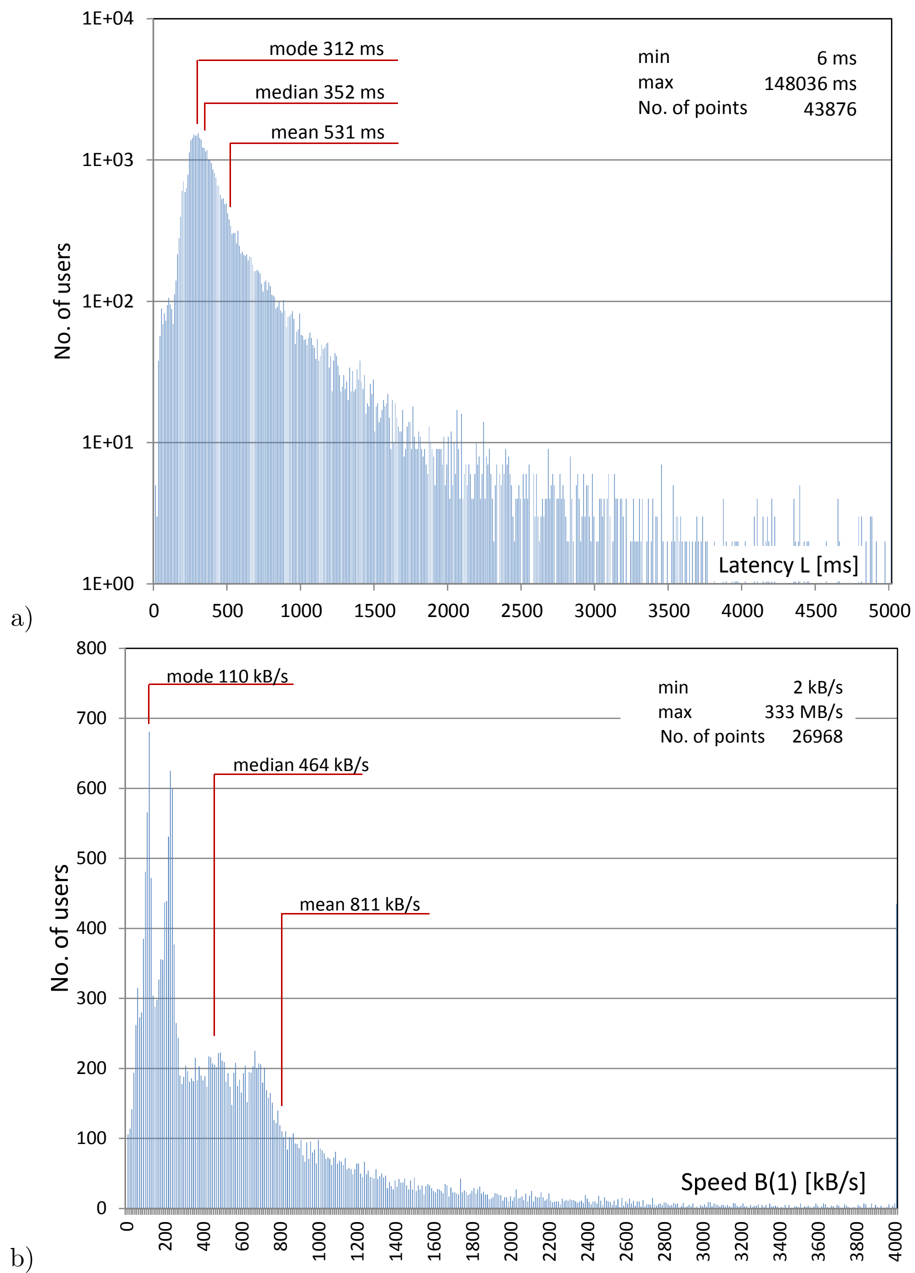

Figure 4.6: Experimental verification of network performance: a) latency distribution (logarithmic vertical axis), b) user download speed distribution.

Number of channels	≥ 2	≥ 3	≥ 4	≥ 5	≥ 6	≥ 7	≥ 8	≥ 9
Accumulated frequency	100%	81%	68%	65%	61%	57%	12%	6%

Table 4.2: Distribution of browser parallel channel number limit.

Network performance observed by browsers has been tested experimentally. In order to estimate latencies and available bandwidth of user browsers a script downloading files of size 1B and 1MB has been installed on a web page ranking popularity of other web pages. The test page is linked to from over 700 other websites using hyperlinks which users may click manually. The variety of linking websites guarantees that the population of visitors is not too uniform. By viewing this web page the visitors executed the scripts in their browsers and downloaded the two files using their specific browsers and Internet connections. Since the scripts were appended to a "production" page, the it was possible to gather real viewers traffic with their specific network performance features. The times of downloading the two files were collected. According to formula (4.1) transferring x bytes of data without using parallel channels takes $L + x/B(1)$ units of time. Time t_1 of downloading 1B file is dominated by communication latency L. Hence, t_1 as an estimate of L is used. Time t_2 of downloading 1MB file has a significant component related to bandwidth. The speed is calculated as $B(1)$ =1MB/$(t_2 - t_1)$. Measurements with $t_2 \leq t_1$ were rejected. In total, measurements from 17460 unique IP addresses were collected. Time t_1 was measured 43876 times, 26968 measurements with $t_2 > t_1$ were collected, 277 measurements with $t_2 \leq t_1$ were rejected.

Results of latency measurement are shown in Fig.4.6a. It can be seen that latency distribution has a long tail, but majority of the observations are concentrated around mean, median and the average value. Over 2/3 observations are concentrated in range [200ms,500ms]. It can be concluded that performance optimization should focus on typical values of the latency. Distribution of speeds is shown in Fig.4.6b. Also speed distribution has a long tail, but over 76% of registered speeds are in range [100kB/s,2MB/s]. The histogram in Fig.4.6b demonstrates that measurements aggregate around particular speeds (in bits/s): 1Mb/s, 2Mb/s, 4Mb/s, 6Mb/s. The number of clients with speeds greater than 6Mb/s (≈750kB/s) quickly decreases with increasing speed. Therefore, it may be advisable to divide users into classes and optimize performance for a particular speed representing a given user class. Such classes could be established by ranges of IP addresses assigned by Internet service providers to client connection type classes, or by separating mobile device browsers or by clustering users according to their connection performance. This, however, is beyond the scope of this study.

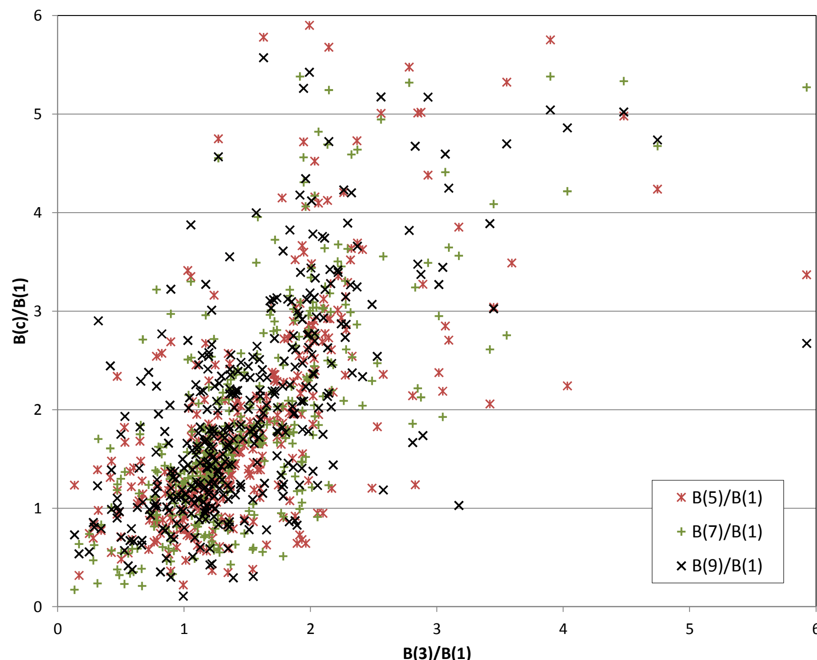

Figure 4.7: Speedups $B(c)/B(1)$ vs $B(3)/B(1)$ in using parallel channels.

Speedups	$\frac{B(3)}{B(1)}$	$\frac{B(5)}{B(1)}$	$\frac{B(7)}{B(1)}$	$\frac{B(9)}{B(1)}$
Medians	1.36	1.56	1.66	1.77
SIQR	0.39	0.60	0.61	0.68

Table 4.3: Synthetic results of parallel channel experiment.

In order to evaluate opportunities for parallel communication a very similar script downloading 1MB of data over $c = 1, 3, 5, 7, 9$ channels has been designed. For example, for $c = 3$ three files of size 1/3MB have been downloaded by a browser executing the script. The downloading time of the last of the files $t(3)$ has been recorded to calculate bandwidth as $B(3)$ =1MB/$t(3)$. 370 measurements from 276 unique IPs have been collected. Different types of browsers are opening various numbers of parallel connections. Hence, the first capability of the user infrastructure in parallel communication has been verified first. The number of communication channels which can be effectively simultaneously open has been determined as the number of communications overlapping in time. If a of communications were performed at least partially in parallel, while the $(a + 1)$-th communication was executed after one of the earlier a communications, then a was recorded as the number of available channels. In Table 4.2 cumulated fraction of browsers capable of using at least some number of parallel channels is shown. It can be verified in Table 4.2 that all browsers are using

at least two parallel channels, in roughly 80% three channels can be used, but only 12% are using 8, 9 or more. Hence, not in all browsers can any speedup in communication be observed if, e.g., 8 concurrent downloads are started. To compensate for the differences in user bandwidths $B(1)$, in the further discussion bandwidth speedup $B(c)/B(1)$ obtained by using c parallel communications is considered. For the sake of giving the reader a rough impression of the obtained results Fig.4.7 shows speedups $B(c)/B(1)$ as sets of points. On the horizontal axis speedup $B(3)/B(1)$ is shown, along the vertical axis speedups $B(c)/B(1)$ for $c = 5, 7, 9$ are shown. It can be seen that: i) indeed there is some acceleration of communication by use of parallel channels because most of the observations are located above a diagonal line, ii) the acceleration has a great deal of dispersion, iii) the results form one cluster, iv) there are cases for which no gain (no speedup) has been observed. In roughly 19% of measurements parallel communication resulted in longer communication time. In Table 4.3 the results are presented in a more synthetic form. Speedups $B(c)/B(1)$ obtained in parallel communications are reported. The first line presents medians of the speedups. The second line provides SIQR (semi-interquartile range) as an index of dispersion. A moderate speedup increasing with the number of open channels c can be seen. Clearly, the speedups are sub-linear.

4.4 State of the Art

Initially CSS-sprites were constructed manually [109]. Here only tools for an automatic CSS-sprite packing are considered. Since there are many software solutions with little differing names, they will be identified by web addresses and in some cases their own short names. The index of names and addresses is given in Table 4.4.

There are three groups of CSS-sprite generators which have been excluded from further study and evaluation (cf. Table 4.4 and Table 4.6). Firstly, there is a group of tools bound to web pages developed in a specific technology stack and software framework. These tools were created with the intention of generating sprites applicable only in certain technology ecosystem and not as independent files for external use. Applications in this set are marked as group A in Table 4.4. Secondly, there is a set of applications which could not be included in the further study because the it was not possible to use them. Such cases are mentioned in Table 4.4. The specific situations which were encountered were: failure to work after installation (group B in Table 4.4), dead web applications giving no results (C), sprites with overlapping tiles (D).

Reason	Web address
A. bound to websites created in certain technology stack and framework	aspnet.codeplex.com/releases/view/65787 compass-style.org/reference/compass/helpers/sprites/ contao.org/en/extension-list/view/cssspritegen.en.html docs.typo3.org/TYPO3/SkinningReference/BackendCssApi/SpriteGeneration drupal.org/project/sprites github.com/northpoint/SpeedySprite github.com/shwoodard/active_assets requestreduce.org spriterightapp.com
B. failed to install and work properly	search.cpan.org/perldoc?CSS::SpriteMaker yostudios.github.io/Spritemapper/
C. online with dead website or scripts	css-sprit.es spritifycss.co.uk
D. produce results with overlapping tiles	mobinodo.com/spritemasterweb spritepad.wearekiss.com timc.idv.tw/canvas-css-sprites/en/

Table 4.4: Excluded CSS-sprite generators.

Further applications are listed in Table 4.6 and Table 4.7. In the third column of the tables (application type) the way of using a generator is described. CSS-sprite generators are usually used in two ways: as an online or as a commandline application. In both cases tiles and sprites are files. A few exceptions exist. SpriteMe and mod_ps read web page background images and convert them into sprites. Moreover, mod_ps is an Apache server module and does it in web pages it serves. IHLabs and selaux are scripts without commandline support, parameters (e.g. input images) are set by code modification. Applications using script languages (e.g. Ruby or Python) often require additional packages, sometimes quite hard to install. The set of user options for the output sprite is described in the fourth column. PNG denotes a 32bpp truecolor PNG image with transparency. PNG8 is an 8bpp PNG image with or without transparency. It can be observed that the set of output formats is usually limited and if there is any option, then the responsibility rests on the user to choose reasonable settings. Some applications admit using postprocessing to further reduce the sprites. However, such post-optimization cannot undo bad decisions made earlier. Hence, there is a need for some decision support in selecting minimum color depths and in optimizing output format. In Table 4.6 CSS-sprite generators are listed which align tiles in a single column or row. A drawback of these applications is that they construct sprites of very big dimensions if the number of tiles

Short Name	cf. Tab.	Web address
aberant	4.6	github.com/aberant/css-spriter
cbrewer		codebrewery.blogspot.com/2011/01/cssspriter.html
cssscom		csssprites.com
csssorg		csssprites.org
elentok		github.com/elentok/sprites-gen
fsgen		freespritegenerator.com
IHLabs		github.com/IndyHallLabs/css-sprite-generator
insts		instantsprite.com
JWwsg		github.com/jakobwesthoff/web-sprite-generator
perforgsg		spritegen.website-performance.org
mod_ps		developers.google.com/speed/pagespeed/module/ filter-image-sprite
selaux		github.com/selaux/node-sprite-generator
spriteme		spriteme.org
acoderin	4.7	acoderinsights.ro/sprite/
cdplxsg		spritegenerator.codeplex.com
codepen		codepen.io/JFarrow/full/scxKd
csgencom		css.spritegen.com
csssnet		cssspritesgenerator.net
glue		glue.readthedocs.org/
isaccc		codeproject.com/Articles/140251/ Image-Sprites-and-CSS-Classes-Creator
JSGsf		github.com/jakesgordon/sprite-factory/
pypack		jwezorek.com/2013/01/sprite-packing-in-python/
txturepk		codeandweb.com/texturepacker
simpreal		simpreal.org.ua/csssprites/
shoebox		renderhjs.net/shoebox/
spcanvas		cssspritegenerator.net/canvas
stitches		draeton.github.io/stitches/
sstool		leshylabs.com/apps/sstool/
zerocom		zerosprites.com

Table 4.5: Index to the CSS-sprite packing solutions.

Short Name	last update	application type	output options	2D packing mode
aberant[T]	Mar 24, 2011	commandline multiplatform (Ruby)	PNG	One row
elentok[T]	Nov 5, 2011	commandline multiplatform (Python)	PNG, JPEG	One row
fsgen	unknown	online	PNG	One column
spriteme[1T2]	Aug 29, 2014	Bookmarklet. Analyzes a web page	PNG, color mode	One column
cbrewer	Jan 2, 2011	windows executable	PNG, JPEG	One column
IHLabs[C]	Aug 22, 2008	code to modify and run (PHP)	PNG, JPEG, GIF	One column
cssscom	unknown	online, single file upload	PNG, no opacity	One column or row with padding
csssorg[TC3]	Feb 14, 2014	commandline multiplatform (JAVA)	PNG, automatic color depth	One column or row with padding
insts[T]	Oct 30, 2014	online	PNG, GIF	One column or row with padding
perforgsg[4]	Jan 22, 2010	online, upload of zip file (filename bugs)	PNG, JPEG, GIF, number of colors and loss rate	Columns or rows with padding
mod_ps[1 J]	Aug 28, 2014	Apache module	PNG, GIF	One column
JWwsg[C]	Mar 27, 2010	commandline multiplatform (PHP)	PNG	Multiple rows with pictures of similar colors
selaux	Aug 12, 2014	code to modify and run (JavaScript)	PNG	One column, row or diagonal line

[T] Offers tile test sets. [C] Offers CSS test sets. [J] Does not read JPEGs.
[1] Accepts only background PNG and GIF images from a web page.
[2] Simple decision support based on predefined rules.
[3] Reads images from CSS file, requires manual annotation of the files.
[4] Possible postprocess: OptiPNG.

Table 4.6: Solutions not using 2-dimensional packing algorithms.

Short Name	Last Update	Application Type	Output Options	2D-Packing Method
csssnet[5]	2014	online	PNG	Unknown
codepen[6]	?	online	PNG	Unknown. Choice of: tile sorting, sprite dimensions.
glue	?	commandline multiplatform (Python)	PNG, PNG8	Implementation of [52].
zerocom[T7]	May 8, 2014	online	PNG, PNG8	Tries [69] for 20 seconds. If instance is large then uses [26].
pypack	Jan 6, 2013	commandline multiplatform (Python)	PNG	Extension of [52].
JSGsf[C8]	Aug 08, 2014	commandline multiplatform (Ruby)	PNG	Can be forced to use implementation of [52].
acoderin[G9]	Jan 22, 2010	online, upload of zip file	PNG, JPEG	Some variation of guillotine split heuristic.
csgencom[10]	May 2014	online	PNG, JPEG, GIF, loss rate	Unknown.
cdplxsg[11]	Sep 10, 2010	windows executable	PNG	Implementation of [53].
txturepk[12]	Oct 27, 2014	GUI for Windows, MacOs, Linux	PNG, and many other formats	Best result of the heuristics: MaxRects, ShortSideFit, LongSideFit, AreaFit, BottomLeft, ContactPoint.
stitches[T13]	May 4, 2013	online	PNG	Unknown.
sstool[14]	May 29, 2014	online	PNG	Unknown local search.
isaccc[G]	Feb 17, 2013	windows command line	PNG	ArevaloRectanglePacker [100].
simpreal[15]	Feb 25, 2013	online	PNG, JPEG, GIF, BMP, Base64	Many options: heuristics, column or row mode, groups of images, tile sorting.
spcanvas[16]	?	online	PNG	Implementation of [68].
shoebox	2014	GUI, multiplatform (Adobe Air)	PNG	Unknown.

[T] Offers tile test sets. [C] Offers CSS test sets. [G] Does not read GIFs.

[5] Forces padding. Fails on spaces in the input filenames, and files larger than 30kB.

[6] Not fitting tiles are discarded without warning.

[7] Filename limitations. Postprocess: PngOpt. High computational complexity.

[8] Failed to work with rmagick package, but works with chunkypng instead. Possible postprocess: pngcrush.

[9] Creates more than one sprite if bounding box exceeds 1200px×1200px. Hangs on duplicate filenames with different extensions. Allows repacking tiles in sprites given as input.

[10] Crashes on ≥ 73 tiles.

[11] Fails on spaces in the filenames and duplicate filenames.

[12] Possible postprocess: PngOpt.

[13] 2D-packing places pictures instantly, but unexpectedly continues computations for some more time.

[14] Optimization feature randomly repacks sprite. High computational complexity.

[15] Rich interface with many options. Hard to use.

[16] Bounding box can be resized, which sometimes leads to tile overlapping.

Table 4.7: Solutions using some 2-dimensional packing algorithms.

is big and with a lot of wasted space if the tile aspect ratios differ. As a result, sprites built by such applications are not comparable with the sprites obtained by using some geometric packing algorithm. Therefore, they are considered not suitable for real-life industrial use. This is the third set of applications excluded from further comparisons.

Applications using some geometric packing algorithms are listed in Table 4.7. In a few lucky cases the applied 2-dimensional packing algorithms were identified in the provided software documentation. Algorithm [52] is commonly used because its implementation is openly available. As geometric packing is $\mathcal{NP}$-hard most of the applications use some simple greedy heuristics.

To the best of the author's knowledge all existing sprite generators build a single output sprite. No solution automatically evaluates options for distributing the tiles into several sprites for better matching tile types and to optimize communication time. Only one solution uses a set of rules to optimize image color depths and compression settings.

4.5 Spritepack

This section presents Spritepack, a method for sprite construction. Given set of sprites $\mathcal{T}$, communication parameters $L, \overline{B}$ Spritepack progresses in four steps: i) tile classification, ii) geometric packing, iii) packing with image compression, iv) postprocessing. Spritepack has been implemented in C++ using MS Visual Studio 12 and Magick++ API to ImageMagick.

4.5.1 Tile Classification

With the goal of grouping tiles with similar sets of colors and to retain as low color depth in sprites as possible, input tiles are first classified according to their color depth. The following image classes have been distinguished:

1. 8 bit per pixel (bpp) indexed color PNG without transparency (denoted as PNG8i),
2. 8 bpp indexed color PNG with transparency (PNG8it),
3. 8 bpp gray-scale PNG without transparency (PNG8g),
4. 8 bpp gray-scale PNG with transparency (PNG8gt),
5. 24 bpp truecolor PNG without transparency (PNG24),
6. 32 bpp truecolor PNG with transparency (PNG32t),
7. JPEG images (jpeg).

Each tile is included in the class with minimum color depth greater than or equal to the original tile color depth. Since the original image information may specify higher depth than actually existing, images may be attributed to wrong classes. To avoid such a situation each input tile were converted to minimum necessary color depth PNG image using Magick++ and saved on file. Only then was the tile re-opened and assigned to the appropriate class. Similar procedure was applied to JPEG images. If the JPEG image converted to PNG had smaller size, then the PNG version was used in the further manipulations. Images with 1,2,4 bits per pixel are currently relatively rare, and therefore are included in PNG8i, or PNG8it. For similar reasons PNG tiles with 16 bits per color channel were not considered. All GIF images were converted to PNG8i or PNG8it which sometimes reduces image size [115].

4.5.2 Geometric Packing

The goals of geometric only packing are twofold. The first objective is to identify tiles which have similar sizes and can be put together in one sprite with little waste. It should also filter out tiles with odd shapes which should not be combined into a sprite to avoid excessive waste. The second purpose is reducing Spritepack runtime. As noted in Section 4.2.2 image compression is time-consuming, and full evaluation of each intermediate sprite would take too much time. Hence, geometric packing is a form of fast proxy to the full version of the algorithm, or a preprocessing step reducing the number of sprite candidates for complete evaluation. The algorithms for geometric packing operate on tile bounding boxes, that is on rectangles, rather than on bitmaps. A *group* will understood here a set of tentatively assembled tiles. The procedure for geometric packing is given in the following pseudocode.

Geometric Packing
input: set $\mathcal{T}$ of tiles
1: Create a group for each input tile;
2: **while** number of groups is bigger than k
2.1: $bp_1, bp_2 \leftarrow$ **nil**; $bw \leftarrow \infty$; // create an empty group pair with waste bw
2.2: **for all** unevaluated group pairs g_1, g_2 with equal image classes
2.2.1: join g_1, g_2 into a new group g_3;
2.2.2: apply to g_3 all geometric packing strategies;
record the packing with minimum geometric waste w_3;
2.2.3: **if** $w_3 < bw$ **then** $bp_1 \leftarrow g_1, bp_2 \leftarrow g_2, bw \leftarrow w_3$;
2.3: **endfor**;

2.4: create a new group from $bp_1 \cup bp_2$, remove bp_1, bp_2,
reduce number of groups by 1;

3: **endwhile**

Geometric packing is a one-pass method merging in each iteration the best pair of groups. Note that in geometric packing only tiles of the same class may be merged (step 2.2). In this way premature upgrading tiles to higher color depths is avoided. Thus, dealing with the uncertainties of image compression efficiency is delayed to the next step of Spritepack. The above procedure finishes with k groups of tiles. Value of k is a control parameter of Spritepack. Yet, limits on k exist. On the one hand, k cannot be greater than the number of tiles, which is important for small sets $\mathcal{T}$. On the other hand, k cannot be smaller than the number of tile classes identified in set $\mathcal{T}$ plus 2. The offset of two groups has been established experimentally. Without such a margin all tiles from a given class end up in one group. Consequently, very different tile shapes are combined, thus invalidating the first purpose of the geometric packing step. Performance of Spritepack under various k settings is discussed in Section 4.6. Geometric packing is a simple hyperheuristic [23] because it guides a set of low-level heuristics referred to as geometric packing strategies in step 2.2.2. The strategies involve packing model and packing algorithm. Two packing models are possible: 2-dimensional strip packing (2SP) and rectangle packing (RP). The 2SP comes in two flavors of either horizontal or vertical layout. Since geometric phase may involve hundreds of tiles and packing algorithms may be called hundreds of times and more, therefore only fast heuristics are acceptable here. Packing algorithms are dedicated to each type of packing model. For 2SP the following low-level heuristics are available:

- First-Fit Decreasing Height (dhFF, computational complexity $\mathcal{O}(n \log n)$),
- First-Fit Decreasing Height with Fit Two (dhFFf2, $\mathcal{O}(n^3 \log n)$),
- Best-Fit Decreasing Height (dhBF, $\mathcal{O}(n \log n)$),
- Best-Fit Decreasing Height with Fit Two (dhBFf2, $\mathcal{O}(n^3 \log n)$),
- Bottom-Left (BL, $\mathcal{O}(n^2)$),
- Modified Bottom Left (MBL, $\mathcal{O}(n^3)$).

For RP model algorithm Variable Height Left Top (VHLT, with complexity $\mathcal{O}(n^2 w_0)$) is available. In the following a short description of the above heuristics is given. A more detailed account can be found, e.g., in [6, 77, 99, 101].

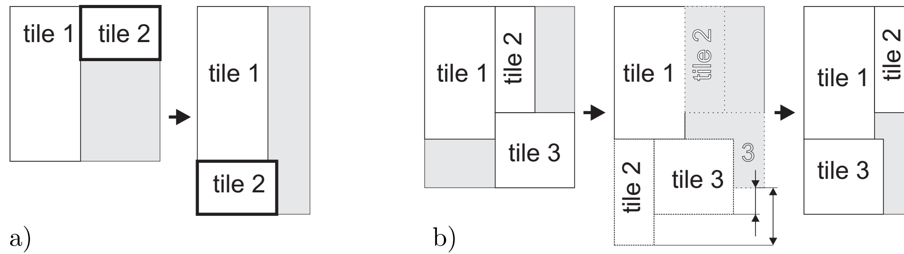

Figure 4.8: Increasing bounding box height in VHLT after a) successful, b) unsuccessful packing.

In the coming description of 2SP algorithms the vertical layout is assumed. It means that there is a strip of the width equal to the widest tile and in the process of packing the occupied area extends upward. Heuristics dhFF, dhFFf2, dhBF, dhBFf2 are shelf packing. As it was explained in Section 3.2.6, in the shelf packing, rectangles are put on the stripe in rows aligned to bottom of the shelf and the height of a shelf is determined by the highest rectangle on the shelf.

The above shelf algorithms consider tiles in the order of decreasing height. First-Fit algorithms (dhFF,dhFFf2) place the current tile on the first shelf which can accommodate the width of the tile. Best-Fit algorithms (dhBF, dhBFf2) place the tile on the shelf on which the remaining width is smallest. When placing the current tile closes a shelf, that is no single remaining tile is able to use the shelf, the Fit Two algorithms (dhFFf2, dhBFf2) search among the remaining tiles for a pair wider than the current tile and still able to fit on the shelf. The Fit Two option has been explained in more detail in section 3.4.2. BL algorithm [25] places tiles as close to the bottom and as close to the left edge of the strip as possible. In this implementation of BL (MBL) tiles are considered in the order of nonincreasing width and holes (empty areas not accessible from above) are not considered. In each iteration MBL tests all available tiles for their placement. The tile which can be put closest to the bottom is chosen. The versions of the algorithms for horizontal packing are defined analogously by swapping the roles of widths and heights.

Implementation of VHLT [101] is inspired by [68]. In the original description of VHTL [101] a horizontal layout is used. Hence, the Left-Top could equally well be referred to as Bottom-Left in the vertical layout rendering. However, the subsequent description sticks to the original horizontal setting. VHLT algorithm iterates over admissible widths w and heights h of the bounding box, verifies feasibility of packing in the given (w, h) using Left-Top algorithm, and returns the bounding box with the smallest total area. A special data structure has been proposed in [101] to represent available space. The iteration starts from

the rectangle of dimensions (w_0, h_0) obtained by Left-Top for horizontal layout. Suppose that the current bounding box (w, h) is feasible, then the width w is decreased by 1px. If the new rectangle is feasible, then w is decreased again. If it is infeasible, then h is increased by one. Moreover, if $w \times h$ is smaller than the area of tiles, then the bounding box is infeasible and h is increased until the rectangle is feasible. If $w \times h$ is bigger than the smallest area of a feasible bounding box, i.e. of a known feasible solution, then testing bounding box (w, h) may be skipped and w is decreased again. In [101] the following rules tailored to Left-Top have been used: i) After a successful packing the next narrower bounding box must be higher at least by the height of the highest tile touching the right edge of the bounding box (cf. Fig.4.8a). ii) After an unsuccessful packing the next narrower bounding box must be higher at least by the smaller of the values: the height of the first rectangle which could not fit, or the minimum extra height allowing rectangles neighboring horizontally be put on one another (Fig.4.8b). The advantages of VHLT are that dimensions of the bounding box are not fixed and that holes are considered. A disadvantage is VHLT complexity. Since each possible width may be verified VHLT is pseudopolynomial, that is VHLT has exponential running time in the length of w_0 encoding. In practice this may be less severe because the initial width w_0 usually does not exceed a few thousand pixels and only a subset of possible widths is really tested by VHLT.

4.5.3 Merging with Image Compression

Merging with image compression is a core of Spritepack. It is based on a similar idea as geometric packing, but takes into account size of the obtained sprites after image compression and the resulting load time estimation defined in (4.2). The procedure for merging with image compression is given in the following pseudocode.

Merging with Image Compression
input: k groups of tiles

1: Create a sprite for each input tile group; record current set of sprites as solution $\mathcal{S}$ and as the best solution $\mathcal{S}^*$ with objective $T^* = \min_{c=1}^{c_{max}} T(\mathcal{S}, c)$;
2: **while** number of sprites is bigger than 1
2.1: $bs_1, bs_2, bs_3 \leftarrow$ **nil**; $bS \leftarrow \infty$; // create an empty sprite pair and empty sprite junction with size bS

2.2: **for all** unevaluated sprite pairs s_1, s_2
2.2.1: apply to the tiles in $s_1 \cup s_2$ all strategies of merging with image compression; record as s_3 the sprite with minimum size S_3;
2.2.2: **if** $S_3 < bS$ **then** $bs_1 \leftarrow s_1, bs_2 \leftarrow s_2, bs_3 \leftarrow s_3; bS \leftarrow S_3$;

2.3: **endfor**;
2.4: $\mathcal{S} \setminus \{bs_1 \cup bs_2\} \cup bs_3$; calculate objective $T = \min_{c=1}^{c_{max}} T(\mathcal{S}, c)$
2.5: **if** $T < T^*$ **then** $\mathcal{S}^* \leftarrow \mathcal{S}$; $T^* \leftarrow T$;
3: **endwhile**;

Merging with image compression is again a greedy sprite merging procedure. In each iteration (while loop in lines 2-3) a pair of sprites which can be packed in minimum size (measured in bytes) is selected in line 2.2.2. Note that in the progress from the initial set of k sprites to just one sprite each intermediate set of sprites $\mathcal{S}$ is a valid solution. The set of intermediate sprites which minimizes the objective function is selected in line 2.5. A key ingredient of merging with image compression are the strategies applied in line 2.2.1. A strategy is defined here by a combination of geometric packing strategy and image compression method. Geometric packing strategies were discussed in the previous section. All geometric packing strategies are verified in line 2.2.1 on the set of tiles included in s_1, s_2. It means that the tiles in $s_1 \cup s_2$ are once again arranged geometrically, and their layouts existing in s_1, s_2 are not passed to s_3. Image compression methods are: i) for PNG format minimum color depth is selected and all filters are tested, ii) if both sprites s_1, s_2 comprise only JPEG tiles or it is allowed to change PNG type tiles to JPEG then JPEG formats with the baseline and progressive compression are tested. The set of admissible PNG filters, the option for changing a PNG class tile into a JPEG class tile, JPEG compression quality are input parameters of Spritepack.

4.5.4 Postprocessing

As it was described in Section 4.2.2 image sizes may be reduced by applying different compression settings. It is not possible to verify alternative image compression settings directly in the earlier step because it its too time-consuming. Therefore, Spritepack takes the opportunity of optimizing sprites as a post-process to the images obtained in the previous stage. This means that sprites obtained in the merging with image compression step are further processed for minimum size. The set of Spritepack post-processors is customizable and builds on the examples from [81]. In further experiments postprocessors `pngout` [110] with the option of using its KFlate algorithm and `jpegtran` [59] with the option of verifying progressive and baseline compression have been applied.

For the end of this section let us note that the CSS-style sheets generated by Spritepack take into account not only the position of a tile in a sprite, but also which sprite comprises the tile (if there are more than one sprite).

4.6 Spritepack Evaluation

This section reports on testing Spritepack. Performance of Spritepack is compared against other existing applications for sprite generation. The results give insight not only into the internal workings of this method and its efficiency, but also into the status quo in the web. Unless stated to be otherwise all tests were performed with the use of ImageMagick 6.8.7-10-Q16-x64 on a typical PC with i5-3450 CPU (3.10GHz), 8GB of RAM and Windows 7. For PNG Compression zlib compression level has been set to 7. All feasible filter types (0-4) have been always tested for a given PNG-type sprite, and the resulting sprite with minimum size was always preserved (cf. Section 4.5.3). For JPEG images quality has been set to 89 in ImageMagick. Combining a non-JPEG tile into a JPEG sprite has been disallowed. Latency has been set to L=352ms which is median in Fig.4.6a. Aggregate bandwidth vector has been set to $\overline{B} = [464, 557, 631, 685, 723, 750, 770, 791, 821]$ in kB/s which has been calculated from median speed in Fig.4.6b and bandwidth speedups in Table 4.3 with additional curve-fitting.

4.6.1 Test Instances

In order to evaluate Spritepack 30 test sets were collected first. The tiles in the test sets are skins and other reusable GUI elements of popular open source web applications. An index to instance names is given in Table 4.6.1, a concise summary on the dataset are collected in Table 4.6.1, further details are provided in [89]. Instance names come from the name of the originating software package and graphical theme name (if there was any). The second through fourth columns in Table 4.6.1 provide numbers of tiles in GIF, PNG, JPEG formats. Animated GIFs and tiles with improperly assigned file extensions were excluded. The following seven columns specify tile classes assigned by Spritepack. Spritepack moved all GIFs to PNG format. Also some JPEG tiles have been transferred to PNG classes because this reduced their sizes. It can be observed that gray-scale tiles are rare and classes PNG8g, PNG8gt hardly ever appear. Test sets offered together with the alternative sprite generators described in Section 4.4 have been also analyzed. Unfortunately, most of them are too simple, consisting of a few tiles with identical shapes. Therefore, only acoderin and SpriteCreator test sets were included in this benchmark making a total of 32 test sets.

Instance name	URL	Accessed on
4images_travelphoto	http://www.themza.com/4images/travel-photography-template.html	Nov 14, 2012
acoderin	http://acoderinsights.ro/sprite/sample/img.zip	Aug 26, 2014
concrete5_coffee	http://www.smartwebprojects.net/concrete5-themes/morningcoffee/	Dec 6, 2012
dotnetnuke_bright	http://www.freednnskins.com/FreeSkins/tabid/152/Article/88/bright.aspx	Jan 1, 2013
drupal_fervens	http://kahthong.com/2009/12/fervens-drupal-theme	Dec 6, 2012
drupal_garden	http://drupal.org/project/gardening	Dec 6, 2012
e107_race	http://www.themesbase.com/e107-Themes/7106_Race.html	Dec 6, 2012
joomla_ababeige	http://www.themesbase.com/Joomla-Templates/7232_Aba-Beige.html	Nov 14, 2012
joomla_busines14a	http://jm-experts-25-templates.googlecode.com/files/busines14a_bundle_installer.zip	Nov 14, 2012
magneto_hardwood	http://www.themesbase.com/Magento-Skins/7396_Hardwood.html	Dec 6, 2012
mambo_partyzone	http://www.themza.com/mambo/party-zone-template.html	Nov 14, 2012
mediawiki_bookjive	http://www.themesbase.com/Mediawiki-Skins/7487_BookJive.html	Nov 14, 2012
modx_creatif	http://modxd.com/creatif-template.html	Dec 6, 2012
modx_ecolife	http://modxd.com/eco-life-template.html	Dec 6, 2012
mojoportal_thehobbit	http://mojoportal.codeplex.com/downloads/get/534280	Jan 1, 2013
moodle_university	http://www.themza.com/moodle/online-university-theme.html	Jan 1, 2013
myadmin_cleanstrap	https://github.com/phpmyadmin/themes/tree/master/cleanstrap/img	Jan 1, 2013
opencart_choco	http://www.opencart.com/index.php?route=extension/extension/info&extension_id=9853&filter_search=cakes	Jan 1, 2013
oscommerce_pets	http://www.themesbase.com/osCommerse-Templates/7195_pets.html	Nov 14, 2012
phpbb_wow	http://www.themesbase.com/phpBB-Themes/8124_WoW5thAniversary.html	Nov 14, 2011
phpfusion_skys	http://www.themesbase.com/PHP-Fusion-Themes/6839_Skys.html	Dec 6, 2012
phpnuke_dvdfuture	http://www.themesbase.com/PHPNuke-Themes/1809_sb-dvd-future-7.html	Dec 6, 2012
prestashop_matrice	http://dgcraft.free.fr/blog/index.php/themes-prestashop/matrice-themes-prestashop-1-3-1-gratuits/	Jan 1, 2013
smf_classic	http://www.themesbase.com/SMF-Themes/7339_Classic.html	Dec 6, 2012
SpriteCreator	http://www.codeproject.com/KB/HTML/SpritesAndCSSCreator/SpriteCreator_v2.0.zip	Jun 30, 2015
squirrelmail_outlook	http://sourceforge.net/projects/squirreloutlook	Jan 1, 2013
textpattern_mistylook	http://txp-templates.com/template/mistylook-for-textpattern	Dec 6, 2012
tinymce_bigreason	http://thebigreason.com/blog/2008/09/29/thebigreason-tinymce-skin	Dec 6, 2012
vbulletin_darkness	http://www.bluepearl-skins.com/forums/index.php?app=core&module=attach§ion=attach&attach_id=2809	Nov 14, 2012
wordpres_creamy	http://www.themesbase.com/WordPress-Templates/9831_Creamy.html	Jun 19, 2015
xoops_bellissima	http://www.themesbase.com/XOOPS-Themes/6849_Bellissima.html	Nov 14, 2012
zencart_artshop	http://www.themesbase.com/Zen-cart-templates/7405_Artstore.html#	Nov 14, 2012

Table 4.8: Test instance index

Instance name	Original tiles			Spritepack tile classification							
	PNG	GIF	JPEG	PNG8i	PNG8it	PNG8g	PNG8gt	PNG24	PNG32t	JPEG	Total n
4images_travelphoto	9	41	7	42	8	0	0	1	0	6	57
acoderin	20	0	0	9	6	0	0	4	1	0	20
concrete5_coffee	0	1	14	0	1	0	0	1	0	13	15
dotnetnuke_bright	2	0	34	0	31	0	0	0	1	4	36
drupal_fervens	5	0	0	2	2	0	0	1	0	0	5
drupal_garden	37	7	4	2	40	0	1	0	1	4	48
e107_race	13	16	17	14	19	2	0	2	0	9	46
joomla_ababeige	10	0	4	7	2	0	0	1	0	4	14
joomla_busines14a	110	1	1	23	82	0	0	0	7	0	112
magneto_hardwood	3	5	1	2	6	0	0	0	0	1	9
mambo_partyzone	2	13	1	14	1	0	0	0	0	1	16
mediawiki_bookjive	6	8	1	1	11	0	0	0	2	1	15
modx_creatif	7	0	17	7	0	0	0	1	6	10	24
modx_ecolife	0	4	6	4	0	0	0	0	0	6	10
mojoportal_thehobbit	11	19	9	9	22	0	0	1	0	7	39
moodle_university	8	246	3	13	240	0	0	2	0	2	257
myadmin_cleanstrap	210	2	0	22	155	7	10	0	18	0	212
opencart_choco	27	0	0	5	19	0	0	1	2	0	27
oscommerce_pets	1	131	71	46	111	0	0	13	0	33	203
phpbb_wow	81	39	10	6	56	0	0	2	58	8	130
phpfusion_skys	8	31	3	18	22	0	0	0	1	1	42
phpnuke_dvdfuture	0	11	3	3	9	0	0	0	0	2	14
prestashop_matrice	37	122	21	61	110	0	0	6	2	1	180
smf_classic	62	254	1	14	283	0	0	0	19	1	317
SpriteCreator	56	0	0	0	1	0	0	0	55	0	56
squirrelmail_outlook	16	57	0	29	43	0	0	0	1	0	73
textpattern_mistylook	1	7	3	5	4	0	0	0	0	2	11
tinymce_bigreason	5	1	0	3	2	0	0	0	1	0	6
vbulletin_darkness	660	355	13	92	833	0	0	3	89	11	1028
wordpres_creamy	28	0	0	3	18	0	0	0	7	0	28
xoops_bellissima	19	2	1	0	7	0	0	0	14	1	22
zencart_artshop	2	55	3	8	49	0	0	0	0	3	60
Total files	1456	1428	248	464	2193	9	11	39	285	131	3132

Table 4.9: Classification of the images in test instances

Number of sprites	1	2	3	4	5	6	7	8	9	10
Number of cases	11	77	52	68	51	31	12	14	3	1

Table 4.10: Number of tests vs the number of final sprites

A disadvantage of the evaluation using a test set collection is some inflexibility in choosing parameters of the tests. Nevertheless, this test set collection represents tiles existing in practical applications and allows examining Spritepack in a realistic setting.

4.6.2 Initial Experiments

This section reports on performance of Spritepack on a corpus of tile sets (Table 4.6.1). The experiments evaluated goal function optimization, sprite sizes and numbers, Spritepack processing time. This series of experiments allows to choose number k of tile groups passed from geometric packing stage and the set of usable geometric packing algorithms.

Before discussing the results let us remind that Spritepack is minimizing goal function (4.2) which is a model of communication time. Total size of the sprites (e.g. in bytes) is not directly minimized and it can be used only as a secondary criterion for comparisons. In the process of combining tiles into sprites some space may be wasted. This results in the increased total area of the sprites compared to the initial area of the tiles (expressed e.g. in px). Consequently, more memory may be needed to represent tiles in the browser than if the tiles were downloaded independently. Hence, the increase in sprite area is an additional evaluation criterion. In the experiments a range of parameter k is swept which has two-fold consequences. On the one hand, reducing k also reduces processing time because fewer groups of tiles are evaluated in merging with image compression (Section 4.5.3). On the other hand, increasing k gives more possibilities of combining groups of tiles into sprites. Thus, k should be neither too big, nor too small.

The instances from Table 4.6.1 have been solved for $k = 4, \dots, 16$. Since k can be neither greater than the number of tiles n, nor can it be smaller than the number of tile classes plus two (cf. Section 4.5.2), 320 test instances have been solved in total. The results of this series of experiments are collected in Fig.4.9-Fig.4.11 and in Tables 4.6.2 and 4.6.2. In Fig.4.9 reduction of the goal function (4.2) vs k is shown. The reduction is expressed relative to the value of the goal function $T(\mathcal{T}, 1)$, i.e. as $(T(\mathcal{S})/T(\mathcal{T}, 1) - 1) \times 100\%$. $T(\mathcal{T}, 1)$ is the cost of transferring the initial tile set $\mathcal{T}$ over one communication channel without packing into any sprite. In Fig.4.9a goal function reduction obtained solely by Spritepack is shown and in Fig.4.9b the reduction obtained in postprocessing

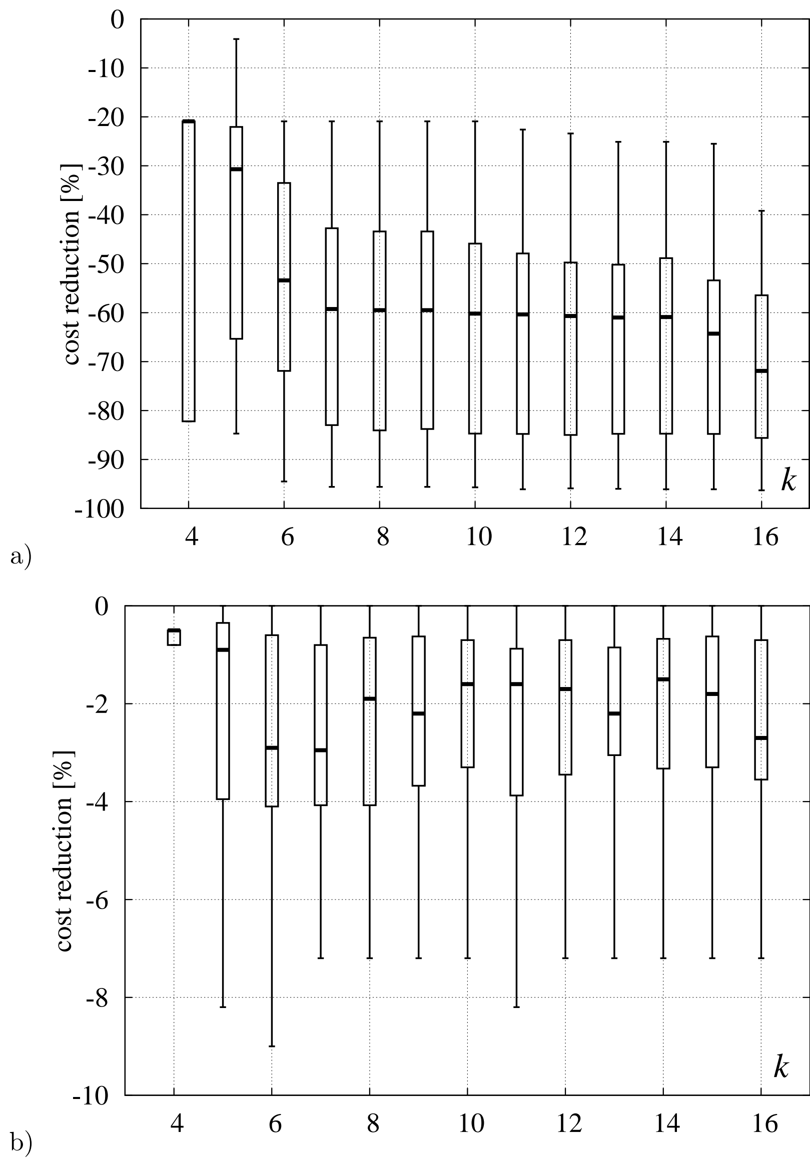

Figure 4.9: Reduction of communication time estimation (4.2). a) Spritepack, b) postprocessing. Lower is better.

is shown. It can be seen that typically Spritepack is able to reduce the goal function by 60% and postprocessing further reduces it by roughly 0.5-4%. With growing k the reductions are better, which is a result of two processes. Indeed there are 6 test sets where increasing k decreases the objective function as could be expected due to a greater sprite combination flexibility. However, a set of instances which can be applied for a given k also has influence in Fig.4.9a. Let us remind that k cannot be greater than the number of tiles nor can it be smaller than the number of tile classes plus 2. Consequently, the number of instances which can be packed with a given k grows from 2 for $k = 4$ to 30 instances for $k = 7, \dots, 9$ and then decreases to 23 test sets for $k = 16$. Therefore, the reduction in the goal function is also a result of changing set of test cases. It is an unavoidable consequence of using real-world test sets as mentioned in Section 4.6.1. This observation applies also to Figs 4.10, 4.11. It can be concluded that for average set of tiles appearing over Internet $k \geq 7$ is sufficient. This should be juxtaposed with the number of the sprites finally constructed shown in Table 4.6.2. In all tests the biggest number of 10 sprites has been constructed for vbulletin_darkness instance which had 1028 tiles. Hence, in the further tests $k = 10$ has been used because it is not restricting the choice of the final sprite number. It can be also observed that Spritepack uses moderate numbers of sprites comparable with the number of browser download channels (see Table 4.2).

As mentioned above, sprite file sizes and the total area are additional performance indicators. Changes in file size are presented in Fig.4.10a for Spritepack alone and in Fig.4.10b for postprocessing. Along the vertical axis the fraction of the total initial tile sizes by which the Spritepack sprite(s) are smaller is shown. Negative values represent reduction in file size. As shown in Fig.4.10b postprocessing reduces file size approximately by 4-7%, which is a useful complement to Spritepack. It can be seen in Fig.4.10a that in general Spritepack reduces total file size by more than 20% (cf. medians). However, for approximately 1/6 of all the cases file size increased, which is shown in Fig.4.10a as positive values. Some increase in file size should not be surprising because merging tiles into a sprite may waste some space and this results in bigger sprite files. It is further confirmed in Fig.4.11a showing relative increase in image area. It can be seen in Fig.4.11a that usually image area is not increasing more than by 10–20%. Yet, there have been cases when area increased by more than 100% for $k = 7$. The impact of enlarged sprite area can be reduced by increasing k even beyond $k = 10$. The most problematic tile sets (prestashop_matrice, moodle_university) have over 180 diverse tiles corresponding to different functionalities of the services from which they come. Tile sets covering such scattered areas of application should be merged into separate sprites according to the system functionalities.

Otherwise, some tiles may be preloaded in some sprite and never used. This may be done effectively by the web-designer on the basis of tile application area. Partitioning tile sets according to their function and frequency of use is beyond the scope of this thesis. Still, Spritepack is able to deal with such big tile sets on the basis of web performance. It is demonstrated in Fig.4.11a for $k \geq 10$ where Spritepack mitigates the worst area increments. Therefore, in the case of tile sets with hundreds of images, possibly representing varied functionalities, Spritepack should be allowed to check also $k > 10$.

Spritepack processing time depends, among the other, on the number of tiles n and group number k. The coefficient of correlation between processing time and the number of tiles observed for $k = 10$ was 0.438 with p-value (probability of obtaining such correlation randomly) equal $\approx$0.0175. Hence, the dependence on n is statistically strong, yet it involves a great deal of dispersion. Such a situation is natural because timing of graphical image compression depends on many factors. One of the key factors is image area and color depth. In the test set tiles had various sizes and color depths. Average execution time per tile in all test sets was 4.59s per tile at $k = 10$. It should not be forgotten that it is only a rough indication of the execution time and real execution times may change very much depending on size of tiles and their complexity. Fig.4.11b gives an impression of Spritepack processing time (including postprocessing). As it can be seen most of the test sets have been processed in at most a couple of minutes. This should be acceptable considering that sprites are built once at the web-site construction stage. Spritepack processing time is split between tile classification, geometric packing, merging with image compression, and postprocessing. The four stages consumed on average 5%, 1%, 81%, 13% of the total processing time, respectively. Thus, merging with image compression is the most time-consuming step. The geometric packing step is very short and it is worth its computational effort as a preparatory step before merging with compression.

In the course of experiments frequencies of using certain geometric packing algorithms were registered. The results are shown in Table 4.6.2. The first line of Table 4.6.2 contains names of heuristics which output has been used at least once. Letters H and V refer to the horizontal and the vertical layouts, respectively. The second line in Table 4.6.2 is the number of times results of some heuristic have been used. The most frequently used heuristics MBL and VHLT cover 99% of all use cases. The shelf packing heuristics (dhFF, dhFFf2, dhBFf2) are hardly ever used. The dhBF method mentioned in Section 4.5.2 has not been used at all. It seems that reducing the set of geometric packing algorithms to just MBL, VHLT, dhFFf2 may be a reasonable option to curb Spritepack complexity in production systems. Contrarily, to obtain better results the geometric packing any new algorithms should outperform the MBL.

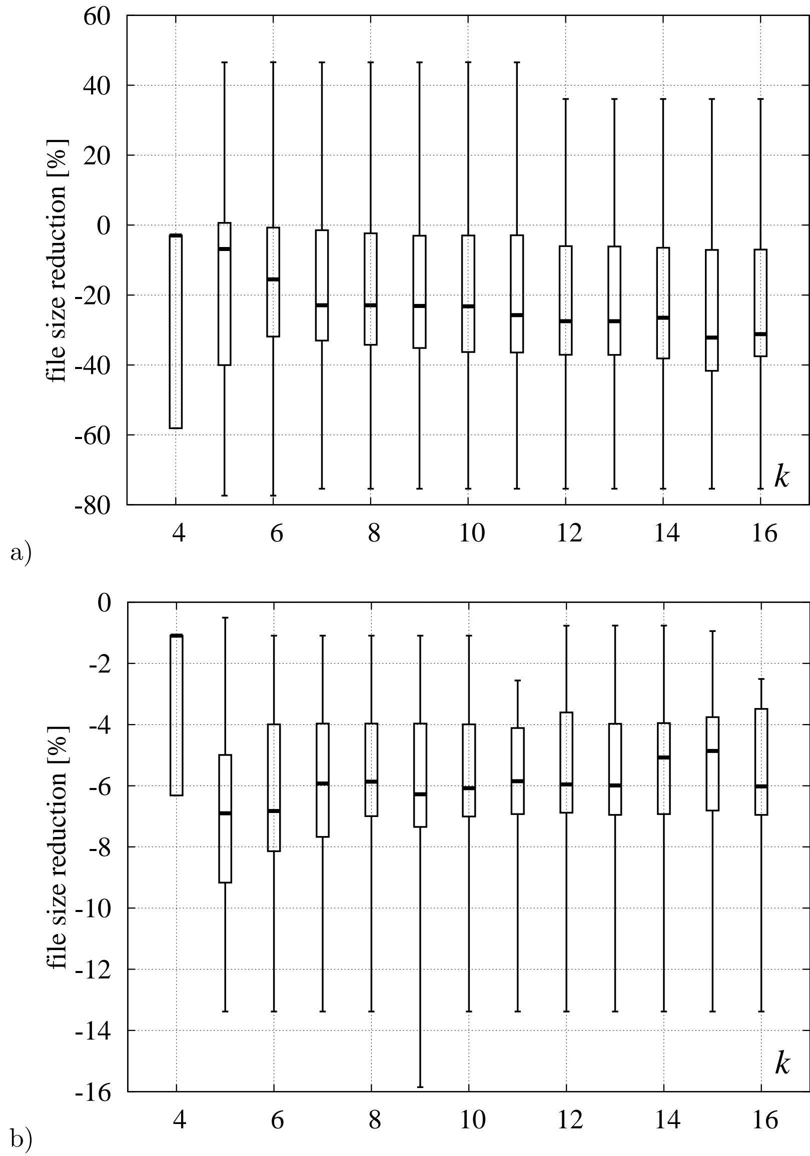

Figure 4.10: Reduction in file size. a) Spritepack, b) postprocessing. Lower is better. Positive values represent increased file sizes.

MBL(V)	MBL(H)	VHLT	dhFFf2(V)	BL(V)	BL(H)	dhFFf2(H)	dhBFf2(V)	dhFF(V)
24135	5381	2829	208	68	21	17	8	5

Table 4.11: Usage of geometric packing heuristics

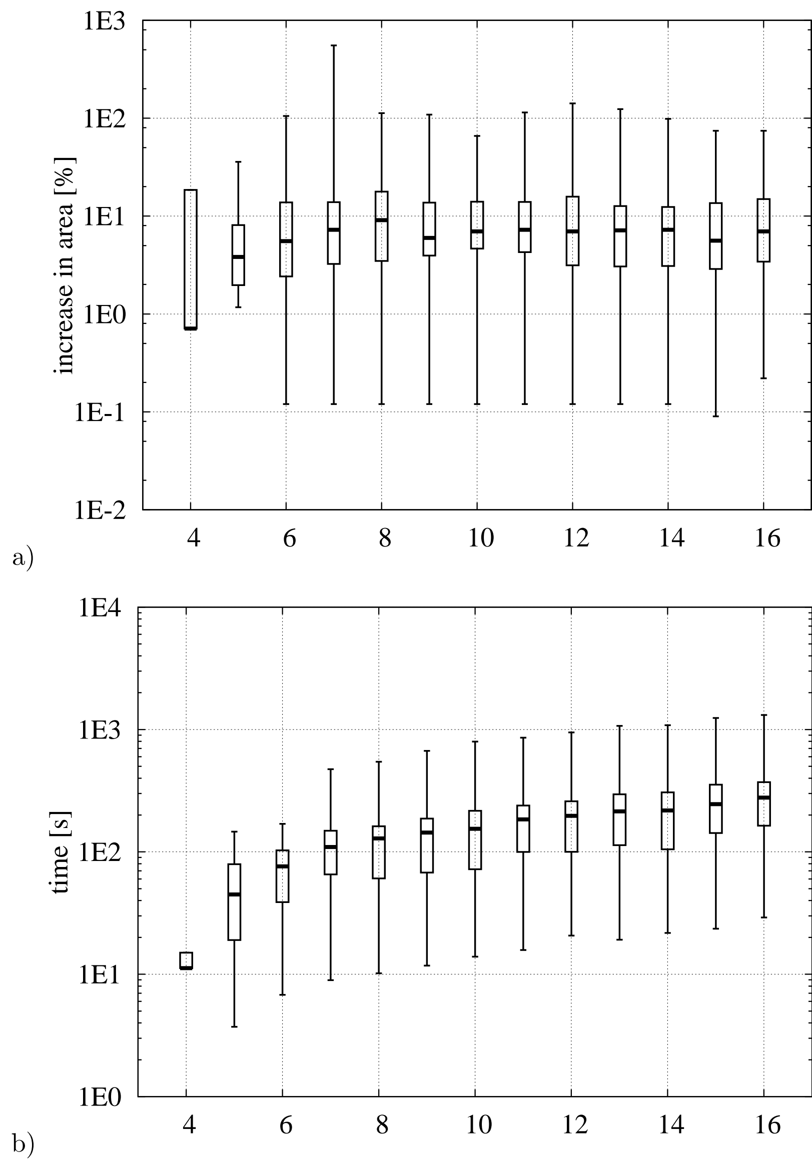

Figure 4.11: a) Change in image area, b) Processing time. Logarithmic scales. Lower is better.

4.6.3 Spritepack Performance Comparison

In this section Spritepack is compareds with alternative sprite generators. For the reasons discussed in Section 4.2 comparing sprite generators rigorously and fairly is not easy. Moreover, a great number of sprite generators exist. Therefore, the following procedure was applied. In the first experiment a big set of sprite generators has been compared on a small set of test instances. As a result, a few solutions have been singled out which have been most reliable, versatile and provided the smallest sprites. In the second series of tests the selected generators have been compared with Spritepack in generating sprites for all instances from Table 4.6.1.

As mentioned above sizes of the sprites built by the alternative generators have been evaluated first. Test instances with moderate number of tiles n have been used. Since not all generators were able to deal with JPEG tiles all tiles have been converted to PNG image format. The alternative sprite generators construct one sprite, while Spritepack builds a number of sprites which minimizes objective function (4.2). In order to make the comparison possible Spritepack code has been modified to extract the single sprite constructed in the last iteration of merging with image compression. The results of the evaluation are collected in Table 4.6.3. The table head gives names of the test instances. Sizes of the sprites constructed by Spritepack (in bytes) are reported in the last line of Table 4.6.3. Except for the last line results are expressed in % relative to the size of the single sprite constructed by Spritepack. Each line gives results for a certain generator. Line labeled "input" expresses size of the input tiles relative to the single Spritepack sprite. An empty entry in Table 4.6.3 means that certain generator has not been able to construct a sprite. Four alternative sprite generators which have given the smallest sprites on average have been selected for the next round of performance comparison. Although Spritepack was not built for creating one sprite with the smallest file size it still outperforms most of the competitors and only one application in a single case produces better results.

In the second round of comparisons the selected sprite generators have been evaluated with respect to the values of the objective function (4.2), and size of the output sprites on a complete set of instances from Table 4.6.1. However, it turned out that Spritepack outperformed the alternative generators and their results were extremely bad. For example, the shoebox generator, which was the best in the previous set of tests, returned sprites which had objective (4.2) equal on average 235% of the Spritepack's (and 642% in the worst case). Similarly, file sizes were on average 376% of the Spritepack's sprite sizes (883% in the worst case). In the case of vbulletin_darkness (1028 tiles) shoebox stopped reacting

Instance name:	acoderin	modx_creatif	SpriteCreator	squirrelmail_outlook	joomla_busines14a	Average
Input	198	100	236	122	87	148
csssnet	211		205	159		192
codepen	199	140	157	122	128	149
glue	154	114	174	157	146	149
zerocom	136	117	191	159	137	148
pypack	149	120	182	146	141	148
JSGsf	161	114	162	156	135	146
acoderin	136	118	170	161	143	145
csgencom	145	116	173			144
cdplxsg	135	140	192	129	115	143
txturepk	132	112	166	128	149	137
stitches	126	139	168	121	117	134
sstool	134	132	174	112	116	134
isaccc	114	153	155	121	123	133
simpreal	123	136	177	107	121	133
spcanvas	137	135	164	116	112	133
shoebox	107	120	143	106	96	114
Spritepack [bytes]	7274	395393	28663	69714	190145	–

Table 4.12: Comparison of sprite generators on size of output. Lower is better. Spritepack is 100%. Spritepack was forced to create a single file.

(hang) on tile 666. There are various reasons for such situation, mostly some tacit assumptions made while designing the alternative generators. It can be inferred that most of the alternative generators assume that (i) there are no large JPEG tiles (like backgrounds or page headers), (ii) tiles have minimum possible color depth, (iii) there is no advantage in special treatment of tiles with odd dimensions, (iv) all tiles sizes are small (icons, buttons), (v) there is no advantage in parallel communication.

A consequence of the first four assumptions is that big savings that could have been made by optimizing big images for color depth, alternative compression, geometric layout are not realized. Still, some of the above assumptions may be considered reasonable in certain applications and the presented evaluation may be deemed unfair. Therefore, to make the conditions of the comparison more compatible with the above assumptions and easier for the alternative generators limitation was made (and only for them) on the set of the tiles subjected to sprite construction to the tiles of file size below 10kB. As a result each tile set

	shoebox	spcanvas	simpreal	isaccc
objective function (4.2)				
min	101	101	101	101
median	132	131	137	134
max	248	284	272	291
file size				
min	82	82	82	83
median	138	141	143	143
max	382	379	386	397

Table 4.13: Evaluation of best sprite generators on 32 test instances. Lower is better. Spritepack is 100%.

has been split into a number of tiles which have not been combined into a sprite and a set of tiles which have been. The obtained set of files, i.e. a sprite and a set of untouched tiles, has been treated as an output tile set $\mathcal{S}$ and the objective function (4.2) has been calculated in the same way as in the Spritepack. In this experiment Spritepack still operated on the whole data sets comprising all the tiles and produced as many sprites as it found effective.

The results of this series of experiments are collected in Table 4.6.3. The four alternative sprite generators have been compared in two criteria: objective function (4.2) and sprite file size. Since the tests have been done on a set of 32 instances, three statistics are reported: minimum, median and maximum values in the population. These three measures are given in % relative to the results provided by Spritepack. It can be seen that the four alternative generators on average build solutions worse than Spritepack by roughly 30% with respect to the objective function (4.2) and 40% with respect to file sizes. There has been only one instance phpfusion_skys when the alternative generators have constructed a solution with smaller overall file size. In this case Spritepack included a JPEG tile with chroma subsampling into a PNG sprite. Since Spritepack is not optimizing sprite size, but the objective function (4.2), it is not surprising that some other method performs better on the sprite size criterion.

4.6.4 End-to-End Evaluation

The end-to-end tests were conducted to verify in a real setting the validity of using multiple sprites, the communication performance model and objective function (4.2), to evaluate the advantages of applying sprites in general and Spritepack in particular. Furthermore, the Spritepack and shoebox generator

Instance name	input		shoebox		Spritepack	
	files	size [B]	sprites	size [B]	sprites	size [B]
magneto_hardwood	9	373610	1	482828	3	294128
modx_ecolife	10	50947	1	366663	3	48891
mojoportal_thehobbit	39	218993	1	726364	7	154486
oscommerce_pets	203	1201692	1	1683872	6	673785

Table 4.14: Sprites in end-to-end test of sprite generators.

performance have been compared. Shoebox has been selected as an alternative generator because in the preceding tests it demonstrated high reliability and solution quality.

In the experiment, the times of downloading all the tiles separately, as a single sprite constructed by shoebox, and as the sprites constructed by Spritepack were measured on the clients' side and reported back to the server. For this purpose a similar script as mentioned in Section 4.3.2 has been designed and inserted into a web page analyzed in Section 4.3.2. By viewing the page, users downloaded the tiles in the above three alternative ways: first all of them separately, next as a single shoebox sprite, finally as a set of Spritepack sprites. Note that in this experimental setup the same communication performance parameters were experienced as had been measured in Section 4.3.2 and had been applied to build sprites by Spritepack. Parameters of the test instances are shown in Table 4.6.4. The instances were chosen to represent a spectrum of possible situations: from modx_ecolife tile set of size smaller than 50kB to oscommerce_pets with 203 tiles and over 1.1MB total size. It can be seen that Spritepack, by using a few sprites, was able to reduce the total size of transferred data. Shoebox, with single sprites, achieves much bigger file sizes, which is in line with the results reported in the previous section.

The results of time measurements are collected in Table 4.6.4. The second and fifth columns ('input') represent all the tiles sent independently, i.e. not sprited. For oscommerce_pets, the biggest tile set with over 203 tiles, 2274 measurements were collected. For the remaining tile sets the number of measurements exceeded 4000 and, e.g., for modx_ecolife 5057 samples were collected. It can be seen that using a single sprite, as in shoebox, may halve the download time. Yet, such reductions not always materialize because in some cases one sprite is not as effective in keeping small file size as Spritepack or even not spriting at all. Despite using a few sprites, which incur additional interactions with the server, Spritepack was able to reduce the download time of tiles sent individually by a factor of 2.5-4. In absolute terms it was from approx. 350ms to 2.4s (medians of differences) while the reduction from shoebox single sprite download time was 140-800ms. It can be concluded that judiciously

Instance name	medians [ms]			SIQR [ms]		
	input	shoebox	Spritepack	input	shoebox	Spritepack
magneto_hardwood	1723	764	574	1597	441	330
modx_ecolife	685	727	244	1502	427	119
mojoportal_thehobbit	776	954	302	456	539	204
oscommerce_pets	3653	1831	931	1453	872	537

Table 4.15: Time results of the end-to-end evaluation in real-world setting.

chosen multiple sprites are not an obstacle to short download times. Overall, it can be concluded that Spritepack fares very well compared to the alternative generators.

Finally, let us comment on the validity of objective (4.2) as a model of the download time. The coefficient of correlation between the medians of download times and the objective function (4.2) was 0.952 and its p-value was below 2E-06. Though these results should be taken with caution, because of big SIQRs in Table 4.6.4, function (4.2) can be considered an effective guide in sprite optimization process.

4.7 Conclusions

The problem of effective construction of CSS-sprites for web applications (CSS-SPP) has been considered in this chapter. This problem poses a number of theoretical and practical challenges. On the theoretical side it is a matter of constructing effective heuristics when evaluation of one solution is time-consuming. It is also difficult to grasp in a tractable way complexity of the network communication performance. On the practical side it is a matter of, e.g., tuning the algorithms for particular tile datasets, choosing image compression setting, obtaining network performance indicators, finding a good trade-off between solution quality and processing time. The method Proposed and implemented Spritepack significantly extends current methods of sprite construction. A typical approach in sprite packing is to take all small images building page layout and combine them into one CSS-sprite. Spritepack approach allows to take all static images, including the ones normally not considered for spriting, and let the algorithm decide how to combine them on the basis of communication performance. Consequently, the overall number of web interactions for one page can be reduced. The key ingredients of Spritepack there are considered: (i) geometric packing method which is a fast hyperheuristic operating on low-level geometric

packing algorithms, (ii) verifying many options for effective image compression, (iii) constructing many sprites for better file size and faster network transfer. Spritepack performance has been compared against alternative solutions on a set of benchmark instances. Though Spritepack is not constructing guaranteed optimum sprites, because it is a heuristic for an $\mathcal{NP}$-hard problem, it can be concluded that this method builds quality sprites in reasonable time and compares well with the alternative methods. Spritepack source code is available at [89].

It seems technically feasible to improve Spritepack, e.g., by more extensive combinatorial search in the stage of merging with image compression or by verifying alternative compression strategies in this stage. Such a step would allow for more effective discovery of tile combinations and for avoiding singular bad cases. However, there is a trade-off between solution quality and processing time. The area of image compression is constantly evolving and thus, new algorithms may be tested in the merging with image compression or in the postprocessing steps. Spritepack has been constructed as a research tool, not an industry-grade product. Hence, the CSS stylesheets produced by Spritepack may be extended by an automatic analysis and update of the existing web pages. Future technologies such as the upcoming HTTP 2.0 [55] or growing popularity of SVG encoding may change the context of sprite packing. Nevertheless, it does not seem that these new technologies will make Spritepack irrelevant and the techniques introduced here can be adapted to the new circumstances.

5 Summary and Final Remarks

In this thesis three combinatorial optimization problems were analyzed. Firstly, in Layout Partitioning for Advertisement Fit where a website layout was partitioned into columns and where by optimization of widths of these columns capability of fitting advertisements was improved. Secondly, in Tag Cloud Construction Problem the tags were packed on shelves of the cloud to provide good readability on the web pages. And finally, in CSS-sprite Packing Problem web page images were packed into a set of bigger sprite-images offloading a server and speeding up page loading. The three presented research problems share the fact that they are solving real-world optimization tasks taken from the field of the Internet and web applications. Another common factor was 2-dimensional cutting/packing which was one of the subproblems of all three analyzed problems and thus certain approaches or even algorithms could be shared between them.

There are many novel elements in all three presented problems and their solutions. Optimization of layout partitioning is a new idea as a whole. While similar problems are subject to computational optimization in text processing or in advertisement placement, the step of webpage layout preparation was usually preformed ad-hoc by a web designer. The algorithms provided for LPfAF allow for optimization of this process and with the wide spectrum of input parameters that can be set by chosen by the user, the results can be well customized for anyone's needs. Finally, the output in the form of weighted best partition or Pareto frontier of nondominated solutions allows finding usable layout even with further conditions set by the web designers. Construction of tag clouds relies on a novel idea of using rules of typography, namely the rule of typographical color. Building the objective function on the basis of the rules of art, where such rules are available, was proven to be valid as a more general idea of optimization of the aesthetic feel. Using the provided mathematical model for Tag Cloud Construction Problem, even the simplest algorithms proposed here produce tag clouds looking better than original ones. Moreover, tag clouds produced by

the algorithms solving TCPP are meeting requirements dictated by the website usage. CSS-sprite optimization introduces a long list of novel ideas, starting with allowing more than one sprite, exploiting the speedups in parallel download or tailoring solution to specific traffic measured on site. The algorithms facilitate properties of the compression methods including the aspect ratio of PNG file property analyzed for the first time in this thesis. Moreover, the objective used is not one but two steps ahead of state of the art solutions optimizing only image dimensions. While the first step was minimizing image file size, then the second was optimization of the download time. As a whole, the software suite built for the CSS-sprite Packing Problem is providing a complex framework, from measuring the traffic parameters on the website, through the analysis of the images used to build its layout, to optimization of the sprites, including an intelligent decision on their number.

As all the three problems were taken from real-world applications it was important to prove that all of the algorithms and their results are usable in practice. Firstly, as the three problems are $\mathcal{NP}$-hard and optimum solutions can be expected in polynomial time (unless $\mathcal{P} = \mathcal{NP}$), the execution times of the algorithms developed had to fit into acceptable time frames for each problem. Secondly, the results provided should be meaningfully better than the state of the art. Algorithms provided for Layout Partitioning Problem for Advertisement Fit are solving the given instances in time acceptable for tasks done once, at the stage of the creation or redesigning of a web page. The constructed layouts, both the one suggested as optimal and the set of nondominated ones, are substantially different from what an ad hoc partitioning would result in. Moreover, their scores for the three given objective function seem to show, that optimized layouts are better. A variety of algorithms developed for tag clouds construction is fulfilling both qualitative requirements, and meet the range of runtimes acceptable for usable web pages. It is difficult to measure whether the beauty goal was achieved, but experiment with experts scoring the solutions seems to confirm this. Similarly to LPfAF, the CSS-sprite optimization algorithm is doing the required work in the non-negligible time, but acceptable at the phase of creation or redesigning a web page. Then, the quantitative results of CSS-sprite optimization are firstly provided analytically, by comparison of the objective function's estimations and then confirmed by measurement in a real-world experiment. These quantitative results are of great importance as they demonstrate that application of the proposed CSS-sprite optimization on websites can offer significant performance gains both on the server side by offloading and infrastructure and client side speeding up loading web pages and reducing memory usage.

Possible extensions of the research topics considered here are immense. The method of solving Layout Partitioning Problem for Advertisement Fit together with parts of the algorithms can be transferred to such remote areas like logistics optimization problems including harbor or general yard organization problems and very specific ones like paper rolls factory. Both Tag Cloud Construction Problem and CSS-sprite Packing Problem are functioning in rapidly changing technological context. Achievements of TCCP can be used both on websites but also in other areas suitable for tag clouds, while its general idea in an even wider area of data visualization. CSS-SPP is closest to being market-ready and the proof of concept produced should see more work on a working prototype to become an industry-grade product.

Appendix A: Complete Results of Partitioning

Due to their size complete computational results are presented in this supplement. Instances are defined by a benchmark set of ad units, webpage width W, and the number of columns. Objective functions ranges for all feasible layouts are included. For each instance there the set of results consists of the best weighted solution and the complete Pareto frontier. Each solution is given as column widths with the total layout width, values of the three objective functions, and the value of the weighted linear function.

subset, W=990, 2 columns
$V_1 \in [42, 47], V_2 \in [7, 7], V_3 \in [-2534, -1914]$
248+732=980; 47, 7 , -1914; 100.0
Pareto frontier:
248+732=980; 47, 7 , -1914; 100.0

subset, W=1250, 3 columns
$V_1 \in [39, 55], V_2 \in [5, 7], V_3 \in [-2606, -858]$
164+328+732=1224; 55, 7 , -858; 100.0
Pareto frontier:
164+328+732=1224; 55, 7 , -858; 100.0

subset, W=1250, 4 columns
$V_1 \in [44, 50], V_2 \in [7, 7], V_3 \in [-1986, -858]$
164+164+164+732=1224; 50, 7 , -858; 100.0
Pareto frontier:
164+164+164+732=1224; 50, 7 , -858; 100.0

AdBrite, W=990, 2 columns
$V_1 \in [43, 48], V_2 \in [7, 7], V_3 \in [-2846, -2102]$
248+732=980; 48, 7 , -2102; 100.0
Pareto frontier:
248+732=980; 48, 7 , -2102; 100.0

AdBrite, W=1250, 3 columns
$V_1 \in [40, 56], V_2 \in [5, 7], V_3 \in [-3054, -1062]$
164+328+732=1224; 56, 7 , -1062; 100.0
Pareto frontier:
164+328+732=1224; 56, 7 , -1062; 100.0

AdBrite, W=1250, 4 columns
$V_1 \in [45, 51], V_2 \in [7, 7], V_3 \in [-2310, -1062]$
164+164+164+732=1224; 51, 7 , -1062; 100.0
Pareto frontier:
164+164+164+732=1224; 51, 7 , -1062; 100.0

Clicksor, W=990, 2 columns
$V_1 \in [735, 1123], V_2 \in [16, 20], V_3 \in [-5833, -4493]$
164+816=980; 1012, 20 , -5433; 64.8
Pareto frontier:
124+864=988; 1123, 17 , -5833; 48.2
129+861=990; 1075, 17 , -5783; 44.3
164+816=980; 1012, 20 , -5433; 64.8
184+806=990; 922, 20 , -5233; 60.0
248+742=990; 796, 18 , -4593; 49.6
253+737=990; 779, 17 , -4543; 42.8
258+732=990; 783, 16 , -4493; 38.2

Clicksor, W=1250, 3 columns
$V_1 \in [739, 1386], V_2 \in [15, 25], V_3 \in [-5977, -1655]$
184+254+812=1250; 1044, 24 , -2153; 71.5
Pareto frontier:
129+129+991=1249; 1386, 15 , -4507; 53.2
129+164+948=1241; 1279, 16 , -4286; 50.5
129+184+936=1249; 1246, 16 , -4122; 49.6
129+254+866=1249; 1168, 20 , -3632; 58.3
129+254+864=1247; 1173, 20 , -3638; 58.5
129+258+862=1249; 1152, 20 , -3604; 57.4
129+304+816=1249; 1082, 25 , -3282; 67.8
129+308+812=1249; 1073, 25 , -3254; 67.5
164+184+898=1246; 1194, 18 , -3257; 57.8
164+258+828=1250; 1056, 21 , -2797; 59.9
164+258+816=1238; 1059, 24 , -2845; 67.2
184+184+876=1244; 1147, 18 , -2533; 60.3
184+184+866=1234; 1152, 18 , -2583; 60.2
184+184+864=1232; 1157, 18 , -2593; 60.5
184+254+812=1250; 1044, 24 , -2153; 71.5
184+258+806=1248; 969, 24 , -2143; 66.7
184+293+773=1250; 891, 21 , -1958; 55.6
184+304+761=1249; 893, 20 , -1908; 53.6
184+308+756=1248; 878, 19 , -1893; 50.2
254+254+742=1250; 855, 18 , -1665; 48.0
254+258+737=1249; 836, 17 , -1655; 44.3

Clicksor, W=1250, 4 columns
$V_1 \in [743, 1147], V_2 \in [16, 22], V_3 \in [-4737, -1280]$
124+124+184+816=1248; 1026, 22 , -2073; 79.9
Pareto frontier:
124+124+184+816=1248; 1026, 22 , -2073; 79.9
124+129+129+866=1248; 1142, 19 , -3632; 64.5
124+129+129+864=1246; 1147, 19 , -3638; 65.0
124+129+184+812=1249; 1018, 22 , -1988; 79.8
124+129+254+742=1249; 829, 18 , -1290; 50.2
129+129+129+862=1249; 1129, 19 , -3604; 63.4
129+129+164+828=1250; 1033, 19 , -2692; 62.2
129+129+164+816=1238; 1036, 22 , -2740; 74.5
129+129+184+806=1248; 946, 22 , -1978; 72.4
129+129+254+737=1249; 813, 17 , -1280; 44.4
129+164+184+773=1250; 860, 21 , -1773; 61.3

Google Ads, W=990, 2 columns
$V_1 \in [1659, 2181], V_2 \in [18, 25], V_3 \in [-7123, -5515]$
248+742=990; 1769, 25 , -5635; 64.4
Pareto frontier:
124+864=988; 2181, 18 , -7123; 42.0
129+861=990; 2123, 18 , -7063; 38.6
129+848=977; 2077, 19 , -7063; 38.4
164+816=980; 2032, 22 , -6643; 54.1
184+806=990; 1908, 22 , -6403; 49.1
204+762=966; 1812, 22 , -6163; 46.3
248+742=990; 1769, 25 , -5635; 64.4
253+737=990; 1756, 24 , -5575; 61.0
258+732=990; 1807, 23 , -5515; 62.8

Google Ads, W=1250, 3 columns
$V_1 \in [1663, 2571], V_2 \in [16, 28], V_3 \in [-7539, -1723]$
184+254+812=1250; 2150, 26 , -3335; 67.2
Pareto frontier:
124+258+864=1246; 2313, 22 , -5667; 53.2
124+313+812=1249; 2214, 27 , -5111; 62.2
129+129+992=1250; 2567, 16 , -6046; 50.3
129+129+991=1249; 2571, 16 , -6049; 50.5
129+164+952=1245; 2351, 16 , -5746; 42.0
129+164+948=1241; 2377, 16 , -5758; 43.1
129+184+936=1249; 2307, 16 , -5554; 41.1
129+254+864=1247; 2301, 22 , -4930; 56.8
129+258+862=1249; 2295, 22 , -4888; 56.8
129+308+812=1249; 2209, 27 , -4438; 65.8
129+313+806=1248; 2092, 27 , -4396; 60.6
129+340+778=1247; 2042, 28 , -4156; 61.7
129+340+773=1242; 2043, 28 , -4171; 61.7
129+340+762=1231; 2049, 28 , -4204; 61.8
164+184+898=1246; 2271, 19 , -4619; 50.9
164+204+864=1232; 2223, 19 , -4515; 49.3
164+258+828=1250; 2161, 23 , -4011; 57.6
164+258+816=1238; 2164, 26 , -4059; 63.8
164+313+773=1250; 1977, 27 , -3571; 60.0
164+313+762=1239; 1983, 27 , -3615; 60.0
184+184+864=1232; 2215, 19 , -3915; 52.3
184+204+862=1250; 2203, 19 , -3685; 53.1
184+254+812=1250; 2150, 26 , -3335; 67.2
184+258+806=1248; 2040, 26 , -3317; 62.2
184+304+762=1250; 1969, 27 , -2985; 62.9
184+308+756=1248; 1910, 27 , -2967; 60.3
204+204+842=1250; 2113, 20 , -3007; 54.9
204+204+816=1224; 2085, 23 , -3163; 58.9
204+238+806=1248; 1961, 23 , -2815; 55.2
204+254+792=1250; 1936, 26 , -2707; 60.9
204+258+778=1240; 1937, 26 , -2743; 60.7
204+258+773=1235; 1938, 26 , -2773; 60.6
204+258+762=1224; 1944, 26 , -2839; 60.5
204+288+756=1248; 1883, 27 , -2515; 61.6
238+254+756=1248; 1864, 26 , -2201; 60.4
254+254+742=1250; 1930, 25 , -1731; 64.1
254+258+737=1249; 1926, 24 , -1723; 61.8
258+258+734=1250; 1932, 23 , -1731; 60.0
258+258+732=1248; 1939, 23 , -1747; 60.2

Google Ads, W=1250, 4 columns
$V_1 \in [1667, 2205], V_2 \in [20, 25], V_3 \in [-6051, -1344]$
124+129+254+742=1249; 1834, 25 , -1358; 70.9
Pareto frontier:
124+124+258+744=1250; 1842, 25 , -1503; 70.5
124+124+258+742=1248; 1846, 25 , -1519; 70.7
124+129+129+864=1246; 2205, 20 , -4932; 49.8
124+129+184+812=1249; 2054, 24 , -3172; 70.4
124+129+254+742=1249; 1834, 25 , -1358; 70.9
129+129+129+862=1249; 2187, 20 , -4888; 48.7
129+129+129+848=1235; 2101, 21 , -4930; 46.7
129+129+164+816=1238; 2056, 24 , -3954; 65.1
129+129+184+806=1248; 1932, 24 , -3152; 61.0
129+129+254+737=1249; 1818, 24 , -1348; 64.8
129+129+258+734=1250; 1824, 23 , -1344; 60.3

IAB, W=990, 2 columns
$V_1 \in [7033, 10453], V_2 \in [37, 51]$,
$V_3 \in [-11183, -8195]$
258+732=990; 10441, 37 , -8195; 74.9
Pareto frontier:
129+852=981; 8512, 51 , -10517; 50.5
164+824=988; 8413, 48 , -9887; 50.9
164+820=984; 8339, 51 , -9887; 55.4
184+806=990; 8146, 47 , -9527; 49.8
216+773=989; 7872, 42 , -8951; 43.9
216+761=977; 8220, 41 , -8951; 46.4
248+742=990; 9195, 39 , -8375; 61.1
253+737=990; 9990, 38 , -8285; 70.1
254+732=986; 10453, 37 , -8267; 74.2
258+732=990; 10441, 37 , -8195; 74.9

IAB, W=1250, 3 columns
$V_1 \in [6495, 13601], V_2 \in [37, 72]$,
$V_3 \in [-13567, -2990]$
164+258+828=1250; 10301, 69 , -5275; 71.2
Pareto frontier:
92+313+844=1249; 10707, 72 , -9477; 62.7
92+313+842=1247; 10714, 72 , -9479; 62.7
92+313+834=1239; 10716, 69 , -9487; 60.5
92+330+828=1250; 10704, 69 , -9187; 61.4
92+340+816=1248; 10768, 52 , -9019; 50.2
92+382+776=1250; 11459, 43 , -8303; 50.0
92+383+773=1248; 11842, 42 , -8288; 51.6
92+396+761=1249; 12193, 41 , -8066; 53.7
124+288+836=1248; 10037, 71 , -6543; 67.1
124+313+812=1249; 10794, 50 , -6213; 57.6
124+368+756=1248; 11160, 40 , -5503; 54.9
124+383+742=1249; 13005, 39 , -5303; 65.7
129+293+828=1250; 10410, 70 , -5729; 71.2
129+304+816=1249; 10551, 52 , -5603; 59.5
129+308+812=1249; 10783, 50 , -5555; 59.6
129+313+808=1250; 10889, 47 , -5489; 58.3
129+328+792=1249; 10675, 44 , -5315; 55.5
129+340+780=1249; 10786, 43 , -5171; 55.8
129+340+764=1233; 11089, 41 , -5267; 55.9
129+340+761=1230; 11119, 41 , -5285; 56.0
129+362+756=1247; 11136, 40 , -4919; 56.6
129+378+742=1249; 12901, 39 , -4715; 66.9
129+383+737=1249; 13285, 38 , -4655; 68.7
129+383+734=1246; 13590, 37 , -4673; 69.7
129+383+732=1244; 13601, 37 , -4685; 69.7
164+254+828=1246; 10313, 69 , -5347; 71.1
164+258+828=1250; 10301, 69 , -5275; 71.2
164+308+778=1250; 10289, 43 , -4725; 54.3
164+313+773=1250; 10605, 42 , -4670; 55.6
164+313+764=1241; 10923, 41 , -4733; 56.6
164+313+761=1238; 10953, 41 , -4754; 56.7
164+330+756=1250; 10943, 40 , -4483; 56.8
184+248+816=1248; 9285, 52 , -4687; 54.9
184+254+812=1250; 10263, 50 , -4611; 59.5
184+258+808=1250; 10030, 47 , -4571; 56.1
184+288+778=1250; 9805, 43 , -4271; 52.9
184+293+773=1250; 9855, 42 , -4221; 52.6
184+304+761=1249; 10435, 41 , -4119; 55.6
184+308+756=1248; 10645, 40 , -4087; 56.2
238+244+768=1250; 8496, 42 , -4073; 45.0
238+248+764=1250; 9114, 41 , -4037; 48.1
238+248+761=1247; 9144, 41 , -4064; 48.2
238+254+756=1248; 10100, 40 , -4001; 53.3
244+244+761=1249; 8937, 41 , -3565; 48.5
244+248+756=1248; 9188, 40 , -3543; 49.3
254+254+742=1250; 12061, 39 , -3007; 67.3
254+254+734=1242; 12330, 37 , -3095; 67.2
254+254+732=1240; 12341, 37 , -3117; 67.2
254+258+737=1249; 12013, 38 , -2990; 66.3
254+258+734=1246; 12318, 37 , -3023; 67.3
254+258+732=1244; 12329, 37 , -3045; 67.3
258+258+734=1250; 12306, 37 , -2995; 67.3
258+258+732=1248; 12317, 37 , -3017; 67.3

IAB, W=1250, 4 columns
$V_1 \in [6600, 11477], V_2 \in [37, 53]$,
$V_3 \in [-11911, -2125]$
92+92+254+812=1250; 9715, 50 , -3335; 76.1
Pareto frontier:
92+92+244+820=1248; 8469, 51 , -3903; 65.0
92+92+248+816=1248; 8737, 52 , -3907; 68.8
92+92+254+812=1250; 9715, 50 , -3335; 76.1
92+92+304+761=1249; 9887, 41 , -2125; 67.6
92+92+308+756=1248; 10097, 40 , -2175; 67.6
92+92+308+754=1246; 10444, 39 , -2203; 69.0
92+92+313+744=1241; 11056, 39 , -2318; 73.8
92+92+313+742=1239; 11068, 39 , -2346; 73.9
92+124+254+780=1250; 9417, 43 , -2591; 65.1
92+124+258+776=1250; 9397, 43 , -2587; 64.9
92+129+184+844=1249; 8996, 52 , -4582; 68.8
92+129+184+842=1247; 9003, 52 , -4598; 68.8
92+129+253+776=1250; 9578, 43 , -2976; 65.2
92+129+254+773=1248; 9718, 42 , -2473; 66.5
92+129+254+761=1236; 10066, 41 , -2605; 67.5
92+129+258+764=1243; 10024, 41 , -2520; 67.4
92+129+258+761=1240; 10054, 41 , -2553; 67.6
124+124+184+816=1248; 8767, 51 , -4387; 65.9
124+129+254+742=1249; 10881, 39 , -2250; 72.6
129+129+129+852=1239; 9536, 53 , -6215; 69.5
129+129+164+820=1242; 9363, 52 , -5121; 70.1
129+129+254+737=1249; 11161, 38 , -2240; 73.5
129+129+254+734=1246; 11466, 37 , -2273; 74.4
129+129+254+732=1244; 11477, 37 , -2295; 74.4
129+129+258+734=1250; 11454, 37 , -2221; 74.5
129+129+258+732=1248; 11465, 37 , -2243; 74.5

superset, W=990, 2 columns
$V_1 \in [8979, 12475], V_2 \in [39, 51]$,
$V_3 \in [-12221, -8901]$
258+732=990; 12467, 39 , -8901; 74.9
Pareto frontier:
129+852=981; 10487, 51 , -11481; 50.5
164+824=988; 10356, 48 , -10781; 49.6
164+820=984; 10277, 51 , -10781; 54.9
164+812=976; 10320, 49 , -10781; 51.3
184+806=990; 10043, 49 , -10381; 51.9
204+780=984; 9832, 47 , -9981; 49.2
216+773=989; 9891, 46 , -9741; 50.2
216+762=978; 10245, 45 , -9741; 52.4
248+742=990; 11162, 41 , -9101; 61.4
253+737=990; 11957, 40 , -9001; 69.9
254+732=986; 12475, 39 , -8981; 74.2
258+732=990; 12467, 39 , -8901; 74.9

superset, W=1250, 3 columns
$V_1 \in [7904, 16708], V_2 \in [39, 72]$,
$V_3 \in [-14941, -3225]$
129+388+732=1249; 16708, 39 , -5303; 69.1
Pareto frontier:
92+333+824=1249; 13636, 72 , -10029; 66.2
92+340+816=1248; 13735, 57 , -9897; 55.7
92+340+812=1244; 13783, 55 , -9901; 54.4
92+362+792=1246; 13299, 48 , -9481; 47.9
92+382+776=1250; 14450, 47 , -9097; 53.7
92+388+768=1248; 14946, 46 , -8985; 55.7
92+396+762=1250; 15330, 45 , -8831; 57.2
124+288+836=1248; 12112, 71 , -7461; 65.4
124+313+812=1249; 12887, 55 , -7081; 58.0
124+333+792=1249; 13247, 48 , -6781; 55.3
124+368+756=1248; 14148, 43 , -6261; 57.3
124+383+742=1249; 16007, 41 , -6031; 65.3
129+254+852=1235; 12478, 64 , -7263; 62.4
129+258+852=1239; 12470, 64 , -7183; 62.6
129+293+828=1250; 12460, 70 , -6627; 68.6
129+304+816=1249; 12596, 57 , -6479; 59.9
129+308+812=1249; 12876, 55 , -6423; 59.8
129+313+808=1250; 12893, 51 , -6347; 57.1
129+328+792=1249; 13157, 48 , -6143; 56.7
129+340+780=1249; 13691, 47 , -5975; 58.9
129+340+773=1242; 13743, 46 , -6017; 58.3
129+340+764=1233; 14067, 45 , -6071; 58.9
129+340+762=1231; 14097, 45 , -6083; 59.0
129+362+756=1247; 14080, 43 , -5679; 58.6
129+378+742=1249; 15897, 41 , -5443; 66.4
129+383+737=1249; 16287, 40 , -5373; 67.7
129+388+732=1249; 16708, 39 , -5303; 69.1
164+254+828=1246; 12359, 69 , -6253; 68.5
164+258+828=1250; 12351, 69 , -6173; 68.6
164+313+773=1250; 12655, 46 , -5458; 54.7
164+313+764=1241; 12979, 45 , -5521; 55.3
164+313+762=1239; 13009, 45 , -5535; 55.4
164+330+756=1250; 13441, 43 , -5237; 56.8
164+340+746=1250; 14379, 41 , -5107; 60.1
164+340+744=1248; 14640, 41 , -5121; 61.3
164+340+742=1246; 14652, 41 , -5135; 61.3
164+340+734=1238; 14934, 39 , -5191; 61.0
164+340+732=1236; 14970, 39 , -5205; 61.1
184+248+816=1248; 11309, 57 , -5565; 56.3
184+254+812=1250; 12352, 55 , -5477; 60.0
184+258+808=1250; 12034, 51 , -5429; 55.6
184+293+773=1250; 11905, 46 , -5009; 52.4
184+304+762=1250; 12491, 45 , -4877; 54.8
184+308+756=1248; 12644, 43 , -4845; 54.1
184+333+732=1249; 14828, 39 , -4537; 62.3
204+204+842=1250; 11154, 52 , -5359; 52.3
204+216+828=1248; 11114, 49 , -5245; 50.2
204+216+820=1240; 11023, 52 , -5317; 51.8
204+238+808=1250; 10773, 50 , -4985; 50.1
204+244+802=1250; 10639, 49 , -4919; 48.9
204+254+792=1250; 11878, 48 , -4809; 54.3
204+288+756=1248; 12191, 43 , -4453; 53.0
216+216+816=1248; 11025, 50 , -5221; 50.6
216+216+812=1244; 11073, 50 , -5257; 50.7
216+254+780=1250; 11830, 47 , -4785; 53.4
216+254+773=1243; 11882, 46 , -4848; 52.7
216+258+776=1250; 11814, 47 , -4741; 53.4
216+258+773=1247; 11874, 46 , -4768; 52.9
216+258+764=1238; 12198, 45 , -4849; 53.5
216+258+762=1236; 12228, 45 , -4867; 53.5
238+244+768=1250; 10597, 46 , -4321; 48.1
238+248+764=1250; 11228, 45 , -4281; 50.4
238+248+762=1248; 11258, 45 , -4301; 50.5
238+254+756=1248; 12174, 43 , -4241; 53.5
244+244+762=1250; 11044, 45 , -3803; 50.9
244+248+756=1248; 11245, 43 , -3789; 50.4
254+254+742=1250; 14148, 41 , -3245; 64.2
254+254+732=1240; 14466, 39 , -3365; 63.9
254+258+737=1249; 14104, 40 , -3225; 63.3
254+258+732=1244; 14458, 39 , -3285; 64.1
258+258+734=1250; 14414, 39 , -3229; 64.0
258+258+732=1248; 14450, 39 , -3253; 64.1

superset, W=1250, 4 columns
$V_1 \in [8308, 14280], V_2 \in [39, 57]$,
$V_3 \in [-13101, -2419]$
92+92+254+812=1250; 11804, 55 , -3643; 76.0
Pareto frontier:
92+92+248+816=1248; 10761, 57 , -4217; 69.7
92+92+254+812=1250; 11804, 55 , -3643; 76.0
92+92+304+762=1250; 11943, 45 , -2419; 66.9
92+92+308+756=1248; 12096, 43 , -2485; 65.0
92+92+313+744=1241; 13044, 41 , -2635; 68.4
92+92+313+742=1239; 13056, 41 , -2665; 68.4
92+92+328+737=1249; 13520, 40 , -2650; 70.3
92+92+330+734=1248; 13849, 39 , -2683; 71.2
92+92+333+732=1249; 14280, 39 , -2695; 74.1
92+124+254+780=1250; 11396, 47 , -2867; 64.4
92+124+258+776=1250; 11380, 47 , -2863; 64.3
92+129+253+776=1250; 11540, 47 , -3247; 64.3
92+129+254+773=1248; 11764, 46 , -2746; 66.0
92+129+254+762=1237; 12118, 45 , -2878; 66.7
92+129+258+764=1243; 12080, 45 , -2798; 66.7
92+129+258+762=1241; 12110, 45 , -2822; 66.8
124+129+254+742=1249; 12865, 41 , -2490; 67.6
129+129+254+737=1249; 13145, 40 , -2475; 68.2
129+129+254+732=1244; 13499, 39 , -2535; 69.1
129+129+258+734=1250; 13455, 39 , -2455; 69.1
129+129+258+732=1248; 13491, 39 , -2479; 69.3

Appendix B: Polskie streszczenie

W pracy zaproponowano i rozwiązano trzy nowatorskie zagadnienia badawcze, których głównymi wspólnymi cechami było pochodzenie z rzeczywistych zastosowań aplikacji internetowych oraz użycie optymalizacji kombinatorycznej, a w szczególności algorytmów pakowania dwuwymiarowego, dla poprawy jakości działania. Elementów wspólnych jest jednak znacznie więcej: wszystkie rozpatrywane problemy są obliczeniowo trudne, należą do klasy problemów NP-trudnych i jako takie w praktyce muszą być rozwiązywane algorytmami heurystycznymi. Wspomniane trzy zagadnienia badawcze opisano poniżej.

Optymalizacja układu szerokości kolumn strony internetowej w celu umieszczania reklam

Jednostka reklamowa to format reklamy uzgodniony między reklamodawcą, wydawcą a siecią reklamową. Jest to prostokąt o zadanych wymiarach. Wydawca musi przygotować strony internetowe tak, żeby mieć możliwie wiele dogodnych możliwości umieszczania jednostek reklamowych, ale też by zachować estetyczny wygląd całości. Witryny mają układ kolumn z przewijaniem pionowym, w którym szerokości są zadane na stałe, zaś wysokość jest właściwie nieograniczona. Obecnie stosowana jest metoda dobierania szerokości kolumn ad-hoc. Podział całkowitej szerokości strony na kolumny sformułowano jako problem optymalizacyjny. Reklamy mogą być umieszczane w kolumnach w grupach, które przez ograniczenia HTML podobne są do problemu cięcia gilotynowego. Szerokości kolumn, które w modelu matematycznym są zmiennymi decyzyjnymi, będą dostosowywane do możliwie wszechstronnego mieszczenia takich grup reklam.

W optymalizacji użyto trzech funkcji celu, które odzwierciedlają kolejno: 1) Elastyczność układu kolumn, tj. jak wiele różnych kombinacji reklam jest on w stanie zmieścić. 2) Zdolność mieszczenia najmniej wygodnej, tzn. najszerszej jednostki reklamowej. 3) Minimalizację marnowanego miejsca, przy pesymistycznym założeniu umieszczania tylko jednej reklamy. Aby właściwie odzwierciedlić wielokryterialność problemu, jako wyniki podawano zarówno wszystkie rozwiązania niezdominowane w sensie Pareto, jak i jedno rozwiązanie najlepsze, uzyskanie przez nadanie funkcjom celu wag. Dla wyznaczenia wag przeprowadzono ankietę wśród 21 ekspertów, profesjonalistów zajmujących się stronami internetowymi.

Zaproponowany model ma aż 13 ograniczeń, począwszy od tak oczywistych jak nie przekraczanie ograniczeń szerokości strony, poprzez ograniczające redundancję rozwiązań, skończywszy na takich które mają odzwierciedlać wymagania estetyczne w łączeniu reklam w grupy, czy pozostawiania pustej prze-

strzeni. Model poprzez wartości wejściowe parametrów używanych w ograniczeniach pozwala na znaczne dostosowanie wyników do potrzeb konkretnego web-developera. Zaproponowany algorytm działa w czterech etapach:

1. Jednostki reklamowe są łączone, tworząc wszystkie możliwe kombinacje.
2. Tworzona jest lista dopuszczalnych szerokości kombinacji i częściowe wartości dla późniejszego wyliczenia funkcji celu.
3. Generowane są wszystkie dopuszczalne układy szerokości kolumn i ostateczne wyniki funkcji celu.
4. Z tej listy wybierane są najlepsze rozwiązanie i listy rozwiązań Pareto-optymalnych.

Etapy 2. i 4. to w gruncie rzeczy dość trywialne przeglądanie listy wartości dostarczonych przez poprzednie etapy. Dla etapu 1. użyto zmodyfikowanego algorytmu Wanga dla dwuwymiarowego rozkroju z ograniczeniami. Modyfikacje polegały na dostosowaniu go do potrzeb rozważanego problemu, oraz na poprawkach wydajnościowych, w szczególności ograniczeniu generowania duplikatów i sprawniejszego ich wyszukiwania i usuwania. Dla etapu 3. skonstruowano algorytm przeglądający układy kolumn w możliwie sprawny sposób, przez wykorzystanie ograniczeń zbioru możliwych szerokości. Algorytm ma złożoność wykładniczą, jednak wobec ograniczonych rozmiarów szerokości strony, jest to rozwiązaniem akceptowalnym.

Dla potrzeb eksperymentów obliczeniowych, zaproponowano jako benchmarki zestawy jednostek reklamowych trzech popularnych sieci reklamowych oraz zestaw rekomendacji jednostek reklamowych Internet Advertising Bureau. Wartości wejściowe parametrów zostały dobrane tak, by w miarę możliwości jak najlepiej odzwierciedlały rzeczywiste warunki. Algorytm na przeciętnym komputerze biurowym generował wyniki w czasach od kilku milisekund o 160 sekund, co jest rezultatem bardzo dobrym dla czynności, która wykonywana jest tylko raz przy tworzeniu strony internetowej. Czas działania algorytmu może się zmieniać w zależności od parametrów wejściowych, co zostało przebadane.

Prezentowane wyniki składają się z najlepszego układu kolumn i zestawu układów Pareto-optymalnych dla kolejnych benchmarków i różnych szerokości stron. Zakresy wartości przyjmowanych przez trzy składowe funkcje celu świadczą o ich czułości. Z wyników można też wyciągnąć dodatkowe wnioski. Na przykład, zestaw reklam Adbrite zawierający tylko pięć jednostek jest zbyt mały i jego użycie jest utrudnione. Z kolei dla wielu innych zestawów wyniki wykazują, że zmienienie układu kolumn o kilka pixeli może znacznie poprawić elastyczność umieszczania reklam.

Wnioski końcowe wskazują dodatkowe przyszłe kierunki badań. Możliwa jest próba zastosowania danych o częstości stosowania jednostek reklamowych, np. w danym kraju, w celu znalezienia układów kolumn mających pewną wszechstronną stosowalność. Z kolei odwrócenie problemu tj. badanie zestawów jednostek reklamowych, pozwoliłoby stwierdzić które z nich mają braki, jakie jednostki należałoby dodać dla najlepszego efektu, nawet pozwoliłyby na pokuszenie się na stworzenie od podstaw kompletnego zestawu. Badanie można też uogólniać do problemu podziału dwuwymiarowej przestrzeni w jednym wymiarze, czyli na kolumny. Takie zagadnienie może znaleźć zastosowanie w planowaniu struktury portu, czy cięciu szerokich bel papieru produkowanych przez fabryki na węższe, dające się transportować i składować.

Budowanie chmur tagów dla zastosowań internetowych

Tag to inaczej fraza reprezentująca tekstowo jakiś obiekt. Tagi mają przypisaną wartość istotności w relacji do innych tagów. Chmura tagów to ułożenie tagów na płaszczyźnie z wizualizacją graficzną ich istotności zazwyczaj przez większy rozmiar. Celem badania było rozwiązanie problemu konstrukcji estetycznych i czytelnych chmur tagów.

Na potrzeby przeglądu literaturowego zaproponowana została taksonomia klasyfikacji chmur tagów w oparciu o 5 parametrów i zakresy ich wartości:

1. Sortowanie tagów. Dostępne opcje to: alfabetycznie, według znaczenia, kontekstowo, losowo, bądź kolejność układana przez algorytm pakowania.
2. Kształt całej chmury. Możliwe opcje: prostokątny, inny kształt regularny (np. okrągły), nieregularny, zadany (np. zadany wielobok, granice mapy).
3. Kształt samych tagów. Opcje: prostokąt, lub kształt znaków.
4. Obracanie znaczników. Opcje: brak, swobodne, dozwolone z ograniczeniami.
5. Wyrównanie w pionie. Opcje: użycie typograficznych linii podstawowych, ograniczone przez właściwości algorytmu (np. grupowanie tagów), swobodne.

Przeanalizowano 14 chmur tagów z literatury, zarówno w zakresie tych parametrów jaki użytych algorytmów oraz zastosowań chmury. Przeanalizowano również literaturę dotyczącą badań użyteczności chmur tagów.

Następnie przeprowadzono analizę wymagań i zaleceń dla chmur tagów, które mają być używane na stronach internetowych. Wymagania i zalecenia te wynikają zarówno z ograniczeń zasad konstrukcji i technologii (np. HTML,

CSS) samych stron internetowych, fragmentacji rynku klientów jak i tego, że strony mają być czytelne zarówno dla ludzi, jak i dla robotów indeksujących. Wynikające z analizy rekomendacje i decyzje dotyczące chmur tagów dla WWW są następujące: 1) chmura jest prostokątna, 2) tagi są traktowane jak prostokąty, 3) algorytm pakowania ustala kolejność tagów, 4) obracanie tagów nie jest dopuszczalne, 5) znaczniki umieszczane są na linii bazowej (na półkach), 6) realizowana winna być minimalizacja marnowanej przestrzeni w prostokącie chmury. Chociaż może się wydawać, że w większości przypadków dokonano wyborów najprostszych, nadal wyjściowy problem optymalizacyjny jest NP-trudny, jako że jest szczególną wersją problemów bin- lub strip-packing.

Dla uzyskania estetycznego wyglądu chmury zastosowano regułę typograficznej równomiernego koloru typograficznego, tj. tekst tak ułożony powinien wizualizować się np. jako możliwie jednorodna masa szarości. Osobnym wymaganiem jest uruchamianie algorytmu konstruowania chmur po stronie użytkownika. Wykazało to badanie rozmiarów tagów przeprowadzone na 4201 użytkownikach rzeczywistej strony internetowej, w którym zidentyfikowano 112 kombinacji rozmiarów tagów wynikających z różnych czcionek dostępnych na urządzeniach i różnić w ich renderowaniu. Dodatkowo chmury powinny za każdym razem być generowane w tym samych wyglądzie, aby nie powodować konsternacji dla użytkownika, a więc przez algorytm deterministyczny, oraz ze względu na wymagania szybkości wyświetlania stron internetowych w czasie rzędu dziesiątych części sekundy.

Przeprowadzona została analiza problemu pakowania rozwiązywanego przy budowaniu chmur tagów. Mamy tu do czynienia z problemem typu strip-packing, w którym szerokość prostokąta jest ustalona, zaś wysokość może być zmieniana w miarę potrzeb przez przesuwanie elementów strony www znajdujących się pod chmurą. Następnie przedstawione zostało matematyczne sformułowanie problemu. Dla każdego tagu możliwe jest wyliczenie jego zaczernienia, a z tagów także dla każdej półki. Minimalizowana winna być potęga k różnicy między zaczernieniem półki a maksymalnym możliwym, sumowana po wszystkich półkach w chmurze. Wykładnik potęgi k został wyznaczony eksperymentalnie. Drugim elementem do dobranym eksperymentalnie był sposób reprezentowania zaczernienia półek, który może być masą, czyli sumą czarności pixeli w tagach lub gęstością, czyli masą dzieloną przez powierzchnię.

Eksperymenty obliczeniowe przeprowadzono za pomocą specjalnie zaprojektowanego algorytmu typu Branch and Bound, pozwalającego rozwiązać problem do optymalności. Ponieważ jednak algorytm ten jest wykładniczy, może rozwiązywać w akceptowalnym czasie ograniczone rozmiary instancji, wstępne testy ograniczono do 16 tagów. Wyniki zostałuy wykorzystane dla zmierzenia dystansu od rozwiązań optymalnych oraz dostrojenia funkcji celu. Strojenie

przeprowadzono przez wygenerowanie 55 testowych chmur tagów w 6 kombinacjach parametrów i poddanie ich ocenie pięciu ekspertów. Na tej podstawie oceniono, że gęstość jest bardziej czułym parametrem niż masa, zaś dla gęstości najlepiej ocenianą wratością k jest 0,5. W dalszych pracach użyto więc tych parametrów funkcji celu.

Dla rozwiązania problemu zaproponowano też specjalny algorytm zachłanny. W oparciu o jego bazową wersję z wykorzystaniem 8 różnych reguł sortowania tagów (po masie, gęstości, wysokości i szerokości) oraz 4 różnych reguł wybierania półek (best fit, worst fit, najmniejsza oraz największa masa tonalna) oraz dwóch dodatkowych modyfikacji możliwe było 8*4*2*2 wersji algorytmu zachłannego. Całość postanowiono wykorzystać jako algorytm Super Fit, którego główną zaletą jest mniejsze prawdopodobieństwo wpadnięcia w pułapkę przypadków pesymistycznych. Jako ostatnią metodę rozwiązania problemu algorytmu przygotowano algorytm Tabu Search. Algorytm startuje z najlepszego rozwiązania z Super Fit i następnie przeszukuje przestrzeń lokalnie wykorzystując tablicę ruchów tabu, w celu ominięcia już odwiedzanych rozwiązań.

Do eksperymentów obliczeniowych wykorzystano rzeczywiste chmury tagów pobrane z działających stron internetowych. Algorytm Super Fit rozwiązuje instancje do 142 tagów w czasie do 57ms (średnio 17ms). Parametry algorytmów zachłannych zastosowanych w Super Fit powodowały różną jego efektywność, 52 z 128 nigdy nie wyprodukowało rozwiązania, które byłoby lepsze od rozwiązań innych algorytmów i mogłyby zostać usunięte z pakietu, gdyby była taka potrzeba. Algorytm Tabu Search po strojeniu, wykonując 300 iteracji, działał średnio w czasie 170ms. Różnice numeryczne w wartości funkcji celu są niewielkie, jeżeli algorytm uzyska minimalną możliwą liczbę półek.

Pakowanie CSS-sprite

CSS-sprite to technika umieszczenia wielu grafik stanowiących elementy strony WWW na jednym obrazku (nazywanym właśnie CSS-sprtite) w celu zmniejszenia liczby zapytań do serwera. Fragmenty tego obrazka są wyświetlane za pomocą reguł CSS w miejscu oryginalnych grafik.

Rozwiązanie problemu rozpoczęto od analizy wyzwań związanych z pakowaniem CSS-sprite. Wyzwania natury geometrycznej związane są z układaniem grafik na CSS-sprite czyli z problemem pakowania. Występujący tu problem pakowania jest nietypowej natury ponieważ przestrzeń, do której będą pakowane elementy nie ma zadanych żadnych wymiarów. Zamiast tego poszukiwana jest najmniejsza powierzchnia w której upakowane mogą zostać elementy. Techniki kompresji grafiki stwarzają kolejne wyzwania, wpływając na rozmiar plików docelowych w sposób nie możliwy do przewidzenia. Pliki mają różne

głębie kolorów, odzwierciedlane za pomocą różnej liczby bitów na pixel. Dodatkowo kompresja PNG będzie osiągać lepsze rezultaty, jeżeli w obrazku będą dłuższe ciągi punktów o jednakowym kolorze. A to może zależeć od wzajemnego położenia grafik. Z kolei kompresja JPEG jest stratna, ale też pixele z sąsiadujących grafik na jednym obrazku mogą na siebie wzajemnie wpływać. Oba formaty nadają się też lepiej dla różnych typów grafik i mają wiele innych parametrów decydujących o rozmiarach obrazków i inych własnościach. Dalsze wyzwania są natury obliczeniowej. W pracy przedstawiony został dowód, że zarówno problem wyboru zbioru grafik dla wspólnej palety kolorów o ograniczonym rozmiarze jak i problem umiejscowienia obok siebie grafik tak, by maksymalizować sąsiadowanie takich samych kolorów są NP-trudne. Wreszcie wydajność komunikacji, przesyłu CSS-sprite między serwerem a przeglądarką nie jest znana. Wpływa na nią wiele czynników począwszy od parametrów serwera, łącza i klienta, poprzez parametry przesyłanych plików, a skończywszy na użytym algorytmie szeregowania pakietów. Jako przybliżenie tego procesu zaproponowana została zależność wykorzystująca algorytm McNaughtona. Zaproponowana metoda wyliczenia czasu komunikacji ma jednocześnie niski koszt akceptowalny dla zastosowania w praktyce, jak i akceptowalną dokładność.

Następnie problem został sformułowany jako model matematyczny. Dany zestaw grafik ma być umieszczonych na CSS-sprite, a umiejscowienie grafik, jak i liczba CSS-sprite są zmiennymi decyzyjnymi. Funkcja celu zakłada minimalizowanie czasu przesyłu CSS-sprite dla zadanych parametrów łącza, w tym przyspieszenia wynikającego z zrównoleglenia przesyłania.

Przed przystąpieniem do dalszych prac przeprowadzono szereg dodatkowych eksperymentów. W pierwszym z nich testowano wpływ kształtu CSS-sprite i wzajemnego umiejscowienia grafik na rozmiar plików. Przygotowane grafiki układano na CSS-sprite o wszystkich możliwych kształtach, począwszy od bardzo długich ale niskich, przez zbliżone do kwadratu, a skończywszy na bardzo wysokich ale wąskich. Jednocześnie testowano 200 permutacji wzajemnego ułożenia grafik. Z 36 testowych zestawów grafik 17 silnie preferowało dla układ długi, 14 układów wysoki, a 5 nie wykazało preferencji. Przez dobór właściwego układu rozmiar CSS-sprite może być zmniejszony o 2% do 35%. Nie znaleziono czynników pozwalających określać preferencję inaczej niż eksperymentalnie. Jednakże po znalezieniu właściwego układu permutacje kolejności grafik pozwalały na zysk mniejszy niż 1,5%. Uznano, że dla grafik typu PNG istotne będzie testowanie obu układów, zaś kolejność ułożenia grafik można pominąć. Nie stwierdzono podobnych zależności w plikach JPEG.

Drugi eksperyment miał na celu znalezienie przykładowych parametrów wydajności komunikacji i sprawdzenie przyspieszenia wynikającego z równoległego pobierania danych. W tym celu skonstruowano skrypt do pomiarów, który

umieszczono na działającej stronie internetowej z rzeczywistym ruchem i zebrano pomiary z 17460 unikatowych adresów IP. Wykazane zostało, że przeglądarki są zdolne do równoległego pobierania. Co najmniej dwa kanały stwierdzono w 100% z nich, zaś przeszło połowa miała ich więcej niż 7. Z kolei przyspieszenie z pobierania równoległego wynosiło średnio od 36% dla trzech kanałów do 77% dla 9 kanałów.

Następnie przystąpiono do analizy dostępnych rozwiązań generujących CSS Sprite. Zidentyfikowano przeszło 30 gotowych programów. Część z nich nie mogła zostać włączona do dalszych eksperymentów, ponieważ były zamkniętymi rozwiązaniami dostępnymi tylko dla konkretnej technologii, np. serwera, bądź nie dało się ich uruchomić, np. przez niedziałające strony internetowe. W pozostałej grupie można było jeszcze wyróżnić rozwiązania nie stosujące żadnych algorytmów pakowania, układających grafiki po prostu jedna obok drugiej. Wreszcie ostatnia grupa to narzędzia do tworzenia CSS-sprite, używające algorytmów pakowania, czasem dość zaawansowanych, dla minimalizowania wymiarów CSS-sprite. Wszystkie znalezione rozwiązania generują dokładnie jednego CSS-sprite, nie biorą pod uwagę odkrytych zależności formatów kompresji, nie optymalizują rozmiaru pliku, nie optymalizują czasu pobierania, które to cechy są głównymi nowościami proponowanego rozwiązania. Niektóre ze znalezionych programów używają postprocessingu algorytmów pakowania dla możliwego poprawienia kompresji i zmniejszenia rozmiaru wyjściowego pliku.

Jako alternatywę zaproponowano algorytm SpritePack działa w czterech etapach:

1. klasyfikacja grafik – w którym testowane są ich parametry takie jak głębia kolorów i podatność na kompresję,

2. pakowanie geometryczne - w którym obrazki są wstępnie grupowane w zadaną liczbę k grup na podstawie pasowania do siebie wymiarami w pakowaniu geometrycznym i zbieżnopści parametrów grafik poznanych w klasyfikacji,

3. pakowanie z kompresją obrazu – grupy z poprzedniego kroku są łączone dalej, pakowane 2-wymiarowo i testowo kompresowane, łączenie w grupy ma charakter algorytmu zachłannego, a funkcją celu jest oszacowanie czasu pobierania,

4. postprocessing – wykonywane jest dodatkowe ulepszenie kompresji.

W procesie pakowania 2D testowane były algorytmy: First-Fit Decreasing Height (oraz w wersji z Fit2), Best-Fit Decreasing Height (oraz w wersji z Fit2), Bottom-Left, Modified Bottom Left oraz Variable Height Left Top. W praktyce

dwa ostatnie radziły sobie najlepiej w 99% przypadków, choć oczywiście wiązało się to z pewnym kosztem czasowym, zwłaszcza w porównaniu do algorytmów zachłannych otwierających tą listę. Docelowo do tej dwójki postanowiono dołączyć jeszcze First-Fit Decreasing Height Two-Fit, który radził sobie najlepiej w większości instancji niezdominowanych przez tamte dwa.

Na potrzeby eksperymentów obliczeniowych przygotowano 32 instancje testowe będące zestawami grafik ze skórek dla popularnych aplikacji webowych w otwartym kodzie. Wstępne testy posłużyły to strojenia parametrów pracy SpritePacka, w szczególności parametr k, który wydatnie wpływa na czas wykonania najkosztowniejszego trzeciego etapu. Eksperymentalnie ustalono $k = 10$. Następnie przeprowadzono na pięciu zestawach testowych porównanie z dostępnymi rozwiązaniami zmuszając SpritePack (dla uzyskania zgodności formy rezultatów) do generowania dokładnie jednego CSS-sprite. Tylko dla jednego z zestawów testowych tylko jedno z konkurencyjnych rozwiązań dało lepszy wynik. Do dalszych testów wybrano cztery najlepsze rozwiązania, które były gorsze od SpritePack śrefunkcji celudnio o 14-33%. Tym razem porównywano funkcję celu, a więc czas pobierania CSS-sprite, ale także rozmiary plików CSS-sprite na wszystkich 32 instancjach testowaych. Dla rozmiarów plików, optymalizowanych przez SpritePack pośrednio, był on lepszy średnio o 38-43%, jednak zdarzały się przypadki, w których konkurencyjne rozwiązania dawały rozwiązania lepsze o do 18%. W funkcji celu SpritePack był średnio lepszy od każdego z konkurentów o przynajmniej 31% i nigdy nie wystąpił przypadek, żeby konkurencja była lepsza od SpritePack. Na koniec przeprowadzono jeszcze eksperyment z wygenerowanymi CSS-sprite na rzeczywistym serwerze porównując CSS-sprite generowane przez SpritePack z zestawem obrazków przesyłanych bez użycia CSS-sprite i z rozwiązaniem generowanym przez najlepszego z konkurentów. Na czterech testowanych zestawach grafik Spritepack zmniejszał czasy pobierania o 350ms do 2.4s w porównaniu z brakiem CCC-sprite, podczas gdy najlepsza z rozwiązań konkurencyjnych zaledwie o 140-800ms. Eksperyment ten potwierdził też, że użyty model, w szczególności prognozowanie czasów pobierania za pomocą funkcji McNaughtona oraz zmierzone przyspieszenie pobierania równoległego, są poprawne. Współczynnik korelacji między medianami zmierzonego czasu pobierania a prognozowanymi przez funkcję celu wynosił 0,952, a jego wartość p była poniżej 2E-06.

Bibliography

[1] E. H. Aarts and J. H. Korst, *Simulated annealing*, ISSUES, 1 (1988), p. 16.

[2] adBrite, *adbrite exchange*. [on-line] http://www.adbrite.com/, 2011.

[3] M. Adler, P. B. Gibbons, and Y. Matias, *Scheduling space-sharing for internet advertising*, Journal of Scheduling, 5 (2002), pp. 103–119.

[4] American Dialect Society, *"Hashtag" is the 2012 word of the year*. http://www.americandialect.org/hashtag-2012, 2013.

[5] A. Amiri and S. Menon, *Efficient scheduling of internet banner advertisements*, ACM Transactions on Internet Technology (TOIT), 3 (2003), pp. 334–346.

[6] ARC Project, *Survey on two-dimensional packing*. http://cgi.csc.liv.ac.uk/~epa/survey.pdf, 2013.

[7] B. S. Baker and J. S. Schwarz, *Shelf algorithms for two-dimensional packing problems*, SIAM J. Comput., 12 (1983), pp. 508–525.

[8] S. Bateman, C. Gutwin, and M. Nacenta, *Seeing things in the clouds: the effect of visual features on tag cloud selections*, in Proceedings of the nineteenth ACM conference on Hypertext and hypermedia, ACM, 2008, pp. 193–202.

[9] J. Beaird, *The principles of beautiful web design*, SitePoint, Collingwood, Vic. :, 1st ed. ed., 2007.

[10] J. Blazewicz, P. Bouvry, M. Kovalyov, and J. Musial, *Erratum to: Internet shopping with price-sensitive discounts*, 4OR, 12 (2014), pp. 403–406.

[11] ——, *Internet shopping with price sensitive discounts*, 4OR, 12 (2014), pp. 35–48.

[12] J. Blazewicz, N. Cheriere, P.-F. Dutot, J. Musial, and D. Trystram, *Novel dual discounting functions for the internet shopping optimization problem: new algorithms*, Journal of Scheduling, (2014), pp. 1–11.

[13] J. Błażewicz, M. Drozdowski, B. Soniewicki, and R. Walkowiak, *Two-dimensional cutting problem: Basic complexity results and algorithms for irregular shapes*, Foundations of Control Engineering, 14 (1989), pp. 137–159.

[14] J. Błażewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, *Handbook on scheduling: from theory to applications*, Springer Science & Business Media, 2007.

[15] ——, *Scheduling computer and manufacturing processes*, Springer Science & Business Media, 2013.

[16] J. Błażewicz and J. Musiał, *E-commerce evaluation-multi-item internet shopping, optimization and heuristic algorithms*, in Operations Research Proceedings, B. H. et al, ed., Berlin Heidelberg, 2010, Springer-Verlag, pp. 149–154.

[17] T. Boutell, P. Joye, and PHP.net, *GD graphics library.* http://libgd.bitbucket.org/, 2013.

[18] K. Bredies and M. Holler, *A total variation-based JPEG decompression model*, SIAM Journal on Imaging Sciences, 5 (2012), pp. 366–393.

[19] R. Bringhurst, *The elements of typographic style*, CRC Studio, 1996.

[20] J. Brutlag, *Speed matters for google web search.* http://www.isaacsunyer.com/wp-content/uploads/2009/09/test_velocidad_google.pdf, 2009.

[21] M. Burch, S. Lohmann, D. Pompe, and D. Weiskopf, *Prefix tag clouds*, in 17th International Conference Information Visualisation, IEEE, 2013, pp. 45–50.

[22] E. K. Burke, M. R. Hyde, and G. Kendall, *Evolving bin packing heuristics with genetic programming*, in Parallel Problem Solving from Nature-PPSN IX, LNCS 4193, T. Runarsson, H.-G. Beyer, E. Burke, J. Merelo-Guervós, J. L. Darrell Whitley, and X. Yao, eds., Springer, 2006, pp. 860–869.

[23] K. Chakhlevitch and P. Cowling, *Hyperheuristics: Recent developments*, in Adaptive and Multilevel Metaheuristics, C. C. et al, ed., vol. 136 of Studies in Computational Intelligence, Springer-Verlag, Berlin Heidelberg, 2008, pp. 3–29.

[24] G. Charlton, *Eight second rule for e-commerce websites now halved.* http://econsultancy.com/uk/blog/500-eight-second-rule-for-e-commerce-websites-now-halved, 2006.

[25] B. Chazelle, *The bottom-left bin-packing heuristic: An efficient implementation*, IEEE Transactions on Computers, 32 (1983), pp. 697–707.

[26] T.-C. Chen and Y.-W. Chang, *Modern floorplanning based on b*-tree and fast simulated annealing*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25 (2006), pp. 637–650.

[27] C. Cheng, T. Angustia, M. H. Ching, C. A. Cristobal, and G. M. Gabuyo, *Synonym based tag cloud generation*, in DLSU Research Congress, 2014.

[28] M.-T. Chi, S.-S. Lin, S.-Y. Chen, C.-H. Lin, and T.-Y. Lee, *Morphable word clouds for time-varying text data visualization*, IEEE transactions on visualization and computer graphics, 21 (2015), pp. 1415–1426.

[29] S. Chikuyonok, *Clever JPEG optimization techniques*. http://www.smashingmagazine.com/2009/07/01/clever-jpeg-optimization-techniques/, 2009.

[30] ——, *Clever png optimization techniques*. http://www.smashingmagazine.com/2009/07/15/clever-png-optimization-techniques/, 2009.

[31] N. Christofides and C. Whitlock, *An algorithm for two-dimensional cutting problems*, Operations Research, 25 (1977), pp. 30–44.

[32] E. Clemons, *Monetizing the internet: Surely there must be something other than advertising*, in System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on, 2009, pp. 1 –10.

[33] Clicksor, *Online contextual advertising and behavioral marketing services*. [on-line] http://www.clicksor.com/, 2011.

[34] E. G. J. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan, *Performance bounds for level-oriented two-dimensional packing algorithms*, SIAM J. Comput., 9 (1980), pp. 808–826.

[35] CompuServe Inc., *Graphics interchange format*. http://www.w3.org/Graphics/GIF/spec-gif89a.txt, 1990.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, vol. 6, MIT press Cambridge, 2001.

[37] W. Cui, Y. Wu, S. Liu, F. Wei, M. X. Zhou, and H. Qu, *Context preserving dynamic word cloud visualization*, in Pacific Visualization Symposium (PacificVis), IEEE, 2010, pp. 121–128.

[38] A. Davies, G. Fabritius, N. Jedrzejewski, A. Lenzen, C. Meteling, A. Roaldseth, C. Schäfer, and Y. Weiss, *Adept - the adaptive JPG compressor*. https://github.com/technopagan/adept-jpg-compressor/, 2014.

[39] M. Dawande, S. Kumar, and C. Sriskandarajah, *Performance bounds of algorithms for scheduling advertisements on a web page*, Journal of Scheduling, 6 (2003), pp. 373–394.

[40] ——, *Scheduling web advertisements: a note on the minspace problem*, Journal of Scheduling, 8 (2005), pp. 97–106.

[41] M. Domański, W. Piasecki, P. Płatek, M. Witczak, J. Marszałkowski, and M. Drozdowski, *Aplikacja wspierająca wybór układu stron www dla celów reklamowych*. http://www.cs.put.poznan.pl/jmarszalkowski/optymalizator_layoutu/, 2012.

[42] R. Eberhart and J. Kennedy, *A new optimizer using particle swarm theory*, in Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on, IEEE, 1995, pp. 39–43.

[43] R. Eckersley, R. Angstadt, C. M. Ellertson, and R. Hendel, *Glossary of typesetting terms*, University of Chicago Press, 2008.

[44] M. Eckert and A. Bradley, *Perceptual quality metrics applied to still image compression*, Signal Processing, 70 (1998), pp. 177–200.

[45] J. Fenn, *When to leap on the hype cycle*. http://www.cata.ca/_pvw522C275E/files/PDF/Resource_Centres/hightech/reports/indepstudies/Whentoleaponthehypecycle.pdf, 1995.

[46] K. Fujimura, S. Fujimura, T. Matsubayashi, T. Yamada, and H. Okuda, *Topigraphy: visualization for large-scale tag clouds*, in Proceedings of the 17th international conference on World Wide Web, ACM, 2008, pp. 1087–1088.

[47] M. R. Garey and D. S. Johnson, *Computers and intractability: A Guide to the Theory of NP-Completeness*, W.H.Freeman and Co., 1979.

[48] P. Gilmore and R. Gomory, *Multistage cutting stock problems of two and more dimensions*, Operations Research, 13 (1965), pp. 94–120.

[49] F. Glover, *Tabu search, part I*, ORSA Journal on Computing, 1 (1989), pp. 190–206.

[50] D. E. Goldberg, *Genetic algorithms*, Pearson Education India, 2006.

[51] Google, *Adsense*. [on-line] http://adsense.google.com, 2011.

[52] J. Gordon, *Binary tree bin packing algorithm*. http://codeincomplete.com/posts/2011/5/7/bin_packing/, 2011.

[53] P.-N. Guo, T. Takahashi, C.-K. Cheng, and T. Yoshimura, *Floorplanning using a tree representation*, IEEE Transaction On Computer-Aided Design of Integrated Circuits And Systems, 20 (2001), pp. 281–289.

[54] M. J. Halvey and M. T. Keane, *An assessment of tag presentation techniques*, in Proceedings of the 16th international conference on World Wide Web, ACM, 2007, pp. 1313–1314.

[55] HTTPbis Working Group, *Hypertext transfer protocol version 2*. https://tools.ietf.org/html/draft-ietf-httpbis-http2-17, 2015.

[56] E. Huang and R. E. Korf, *New improvements in optimal rectangle packing*, in Proceedings of the 21st International Jont Conference on Artificial Intelligence IJCAI'09, 2009, pp. 511–516.

[57] N. Hurst and K. Marriott, *Satisficing scrolls: a shortcut to satisfactory layout*, in Proceeding of the eighth ACM symposium on Document engineering, DocEng '08, New York, NY, USA, 2008, ACM, pp. 131–140.

[58] Impulse Adventure, *What is an optimized JPEG?* http://www.impulseadventure.com/photo/optimized-jpeg.html, 2007.

[59] Independent JPEG Group, *Jpegtran.* http://jpegclub.org/jpegtran/, 2012.

[60] Interactive Advertising Bureau, *Adex 2009 european online advertising expenditure*, 2009.

[61] ——, *IAB ad unit guidelines.* [on-line] http://www.iab.net/media/file/IAB-Ad-Unit-Guidelines-Update-20091029.pdf, 2009.

[62] International Telecommunication Union, *Recommendation t.81: Information technology - digital compression and coding of continuous-tone still images - requirements and guidelines.* http://www.w3.org/Graphics/JPEG/itu-t81.pdf, 1993.

[63] M. Jeon, Y. Kim, J. Hwang, J. Lee, and E. Seo, *Workload characterization and performance implications of large-scale blog servers*, ACM Transactions on the Web (TWEB), 6 (2012), p. 16.

[64] R. M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. M. J. W. Thatcher, ed., Plenum Press, New York, 1972, pp. 85–103.

[65] O. Kaser and D. Lemire, *Tag-cloud drawing: Algorithms for cloud visualization*, arXiv preprint cs/0703109, (2007).

[66] K. Kim, S. Ko, N. Elmqvist, and D. S. Ebert, *Wordbridge: Using composite tag clouds in node-link diagrams for visualizing content and relations in text corpora*, in 44th Hawaii International Conference on System Sciences (HICSS), IEEE, 2011, pp. 1–8.

[67] D. E. Knuth and M. F. Plass, *Breaking paragraphs into lines*, Software: Practice and Experience, 11 (1981), pp. 1119–1184.

[68] R. E. Korf, *Optimal rectangle packing: Initial results*, in Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling ICAPS'03, Palo Alto, USA, 2003, American Association for Artificial Intelligence, pp. 287–295.

[69] R. E. Korf and E. Huang, *Optimal rectangle packing: An absolute placement approach*, Journal of Artificial Intelligence Research, 46 (2012), pp. 47–87.

[70] R. E. Korf, M. D. Moffitt, and M. E. Pollack, *Optimal rectangle packing*, Annals of Operations Research, 179 (2010), pp. 261–295.

[71] M. Kudelka, V. Snasel, Z. Horak, A. E. Hassanien, A. Abraham, and J. D. Velásquez, *A novelapproach for comparing web sites by using microgenres*, Engineering Applications of Artificial Intelligence, 35 (2014), pp. 187–198.

[72] S. Kumar, M. Dawande, and V. Mookerjee, *Optimal scheduling and placement of internet banner advertisements*, Knowledge and Data Engineering, IEEE Transactions on, 19 (2007), pp. 1571–1584.

[73] S. Kumar, V. S. Jacob, and C. Sriskandarajah, *Scheduling advertisements on a web page to maximize revenue*, European Journal of Operational Research, 173 (2006), pp. 1067 – 1089.

[74] B. Y. Kuo, T. Hentrich, B. M. Good, and M. D. Wilkinson, *Tag clouds for summarizing web search results*, in Proceedings of the 16th international conference on World Wide Web, ACM, 2007, pp. 1203–1204.

[75] M. Langheinrich, A. Nakamura, N. Abe, T. Kamba, and Y. Koseki, *Unintrusive customization techniques for web advertising*, Computer Networks, 31 (1999), pp. 1259–1272.

[76] E. L. Lawler and D. E. Wood, *Branch-and-bound methods: A survey*, Operations research, 14 (1966), pp. 699–719.

[77] A. Lodi, S. Martello, and M. Monaci, *Two-dimensional packing problems: A survey*, European Journal of Operational Research, 141 (2002), pp. 241–252.

[78] S. Lohmann, F. Heimerl, F. Bopp, M. Burch, and T. Ertl, *Concentri cloud: Word cloud visualization for multiple text documents*, in 2015 19th International Conference on Information Visualisation, IEEE, 2015, pp. 114–120.

[79] S. Lohmann, J. Ziegler, and L. Tetzlaff, *Comparison of tag cloud layouts: Task-related performance and visual exploration*, in Human-Computer Interaction–INTERACT 2009, Springer, 2009, pp. 392–404.

[80] M. C. Lopez-Loces, J. Musiał, J. E. Pecero, H. J. Fraire-Huacuja, J. Blazewicz, and P. Bouvry, *Exact and heuristic approaches to solve the internet shopping optimization problem with delivery costs*, International Journal of Applied Mathematics and Computer Science, 26 (2016), pp. 391–406.

[81] C. Louvrier, *Optimisation web (images, performance)*. `http://css-ig.net/`, 2013.

[82] M. Mahdavi, M. H. Chehreghani, H. Abolhassani, and R. Forsati, *Novel meta-heuristic algorithms for clustering web documents*, Applied Mathematics and Computation, 201 (2008), pp. 441–451.

[83] J. Marszałkowski, *The importance of advertising exchange for marketing browser games*, Homo Ludens, 3 (2011).

[84] J. Marszałkowski, *Prototype of high performance scalable advertising server with local memory storage and centralised processing*, in Information and Communication Technologies: 18th EUNICE/ IFIP WG 6.2, 6.6 International Conference, EUNICE 2012, Budapest, Hungary, August 29-31, 2012. Proceedings, R. Szabó and A. Vidács, eds., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 194–203.

[85] ——, *Budgeted internet shopping optimization problem (b-isop)*, in Proceedings of 7th Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2015), B. M. P. v. Zdenek Hanzálek, Graham Kendall, ed., 2015, pp. 885 – 887.

[86] J. Marszałkowski and M. Drozdowski, *Optimization of column width in website layout for advertisement fit*, European Journal of Operational Research, 226 (2013), pp. 592–601.

[87] J. Marszałkowski, J. M. Marszałkowski, and M. Drozdowski, *Empirical study of load time factor in search engine ranking*, Journal of Web Engineering, 13 (2014), pp. 114–128.

[88] J. Marszałkowski, J. M. Marszałkowski, and J. Musiał, *Database scheme optimization for online applications*, Foundations of Computing and Decision Sciences, 36 (2011), pp. 121–129.

[89] J. Marszałkowski, J. Mizgajski, D. Mokwa, and M. Drozdowski, *Spritepack resources*. `http://www.cs.put.poznan.pl/mdrozdowski/spritepack/`, 2015.

[90] J. Marszałkowski, J. Mizgajski, D. Mokwa, and M. Drozdowski, *Analysis and solution of CSS-sprite packing problem*, ACM Transactions on the Web, 10 (2016), p. article No.1.

[91] L. Masinter, *Frc 2397: The "data" URL scheme*. `https://www.ietf.org/rfc/rfc2397.txt`, 1998.

[92] R. McNaughton, *Scheduling with deadlines and loss functions*, Management Science, 6 (1959), pp. 1–12.

[93] Z. Michalewicz, *Gas: What are they?*, in Genetic algorithms+ data structures= evolution programs, Springer, 1994, pp. 13–30.

[94] S. Milgram and D. Jodelet, *Psychological maps of Paris*, in Environmental Psychology: People and Their Physical Settings, H. Proshansky, W. Ittelson, and L. Rivlin, eds., Holt, Reinehart and Winston, New York, 1976.

[95] M. Mitchell, *An introduction to genetic algorithms*, MIT press, 1998.

[96] Mozilla Co., *Mozilla jpeg encoder project*. `https://github.com/mozilla/mozjpeg/`, 2014.

[97] J. Musiał and J. Marszałkowski, *Propozycja poprawy wydajności bazy danych dla nowoczesnych aplikacji internetowych*, Zeszyty Naukowe Uniwersytetu Szczecińskiego. Ekonomiczne Problemy Usług, (2011), pp. 404–411.

[98] D.-Q. Nguyen and H. Schumann, *Taggram: Exploring geo-data on maps through a tag cloud-based visualization*, in 14th International Conference Information Visualisation, IEEE, 2010, pp. 322–328.

[99] N. Ntene and J. H. van Vuuren, *A survey and comparison of guillotine heuristics for the 2d oriented offline strip packing problem*, Discrete Optimization, 6 (2009), pp. 174–188.

[100] Nuclex Framework, *Rectangle packing*. `http://nuclexframework.codeplex.com/wikipage?title=Rectangle`, 2009.

[101] M. Perdeck, *Fast optimizing rectangle packing algorithm for building CSS sprites*. http://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu, 2011.

[102] I. Popovici and W. Withers, *Locating edges and removing ringing artifacts in JPEG images by frequency-domain analysis*, IEEE Transactions on Image Processing, 16 (2007), pp. 1470–1474.

[103] R. L. R. Kohavi, *Online experiments: Lessons learned*, Computer 40, no. 9, (2007), pp. 103–105.

[104] G. Randers-Pehrson and T. Boutell, *PNG (portable network graphics) specification*. http://www.libpng.org/pub/png/spec/1.2/PNG-Contents.html, 1999.

[105] Refsnes Data, *Browser display statistics*. [on-line] http://www.w3schools.com/browsers/browsers_resolution_higher.asp, 2011.

[106] A. W. Rivadeneira, D. M. Gruen, M. J. Muller, and D. R. Millen, *Getting our head in the clouds: toward evaluation studies of tagclouds*, in Proceedings of the SIGCHI conference on Human factors in computing systems, ACM, 2007, pp. 995–998.

[107] H. Q. Saremi, B. Abedin, and A. M. Kermani, *Website structure improvement: quadratic assignment problem approach and ant colony metaheuristic technique*, Applied Mathematics and Computation, 195 (2008), pp. 285–298.

[108] C. Seifert, B. Kump, W. Kienreich, G. Granitzer, and M. Granitzer, *On the beauty and usability of tag clouds*, in 12th International Conference Information Visualisation, IEEE, 2008, pp. 17–25.

[109] D. Shea, *CSS sprites: Image slicing's kiss of death*. http://www.alistapart.com/articles/sprites, 2004.

[110] K. Silverman, *Ken Silverman's utility page*. http://advsys.net/ken/utils.htm, 2013.

[111] L. Simon and S. Souders *et al*, *Browserscope*. http://www.browserscope.org/?category=network&v=1, 2015.

[112] K. Simpson, *Obsessions: Http request reduction*. http://blog.getify.com/obsessions-http-request-reduction/, 2015.

[113] E. Spyrou and P. Mylonas, *A survey on flickr multimedia research challenges*, Engineering Applications of Artificial Intelligence, 51 (2016), pp. 71–91.

[114] P. Staníček, *CSS technique: Fast rollovers without preload*. http://wellstyled.com/css-nopreload-rollovers.html, 2003.

[115] S. Stefanov, *Image optimization, part 3: Four steps to file size reduction*. http://yuiblog.com/blog/2008/11/14/imageopt-3/, 2008.

[116] A. Steinberg, *A strip-packing algorithm with absolute performance bound*, SIAM Journal on Computing, 26 (1997), pp. 401–409.

[117] P. Velho, L. M. Schnorr, H. Casanova, and A. Legrand, *On the validity of flow-level tcp network models for grid and cloud simulations*, ACM Transactions on Modeling and Computer Simulation, 23 (2013), p. 23.

[118] F. B. Viégas and M. Wattenberg, *Timelines: Tag clouds and the case for vernacular visualization*, ACM Interactions, 15 (2008), pp. 49–52.

[119] F. B. Viégas, M. Wattenberg, and J. Feinberg, *Participatory visualization with Wordle*, IEEE Transactions on Visualization and Computer Graphics, 15 (2009), pp. 1137–1144.

[120] J. Walhout, S. Brand-Gruwel, H. Jarodzka, M. van Dijk, R. de Groot, and P. A. Kirschner, *Learning and navigating in hypertext: Navigational support by hierarchical menu or tag cloud?*, Computers in Human Behavior, 46 (2015), pp. 218–227.

[121] G. K. Wallace, *The JPEG still picture compression standard*, Communications of the ACM, 34 (1991), pp. 30–44.

[122] P. Y. Wang, *Two algorithms for constrained two-dimensional cutting stock problems*, Operations Research, 31 (1983), pp. 573–586.

[123] J. Wawrzyniak and J. Marszałkowski, *Gamifikacja w edukacji: przegląd wymagań dla platformy gamifikacyjnej*, Homo Ludens, 7 (2015), pp. 229–247.

[124] *Webpagetest*. `http://www.webpagetest.org/`, 2015.

[125] B. D. Weinberg, *Don't keep your internet customers waiting too long at the (virtual) front door*, Journal of Interactive Marketing, 14 (2000), pp. 30–39.

[126] A. Wojciechowski and J. Musial, *A customer assistance system: Optimizing basket cost*, Foundations of Computing and Decision Sciences, 34 (2009), pp. 59–69.

[127] P.-Y. Yin and Y.-M. Guo, *Optimization of multi-criteria website structure based on enhanced tabu search and web usage mining*, Applied Mathematics and Computation, 219 (2013), pp. 11082–11095.

[128] M. Yue, *A simple proof of the inequality ffd (l)? 11/9 opt (l)+ 1,? l for the ffd bin-packing algorithm*, Acta Mathematicae Applicatae Sinica (English Series), 7 (1991), pp. 321–331.