Analysis and Solution of CSS-Sprite Packing Problem

JAKUB MARSZAŁKOWSKI, JAN MIZGAJSKI, DARIUSZ MOKWA,
and MACIEJ DROZDOWSKI, Institute of Computing Science, Poznań University of Technology

A CSS-sprite packing problem is considered in this article. CSS-sprite is a technique of combining many pictures of a web page into one image for the purpose of reducing network transfer time. The CSS-sprite packing problem is formulated here as an optimization challenge. The significance of geometric packing, image compression and communication performance is discussed. A mathematical model for constructing multiple sprites and optimization of load time is proposed. The impact of PNG-sprite aspect ratio on file size is studied experimentally. Benchmarking of real user web browsers communication performance covers latency, bandwidth, number of concurrent channels as well as speedup from parallel download. Existing software for building CSS-sprites is reviewed. A novel method, called *Spritepack*, is proposed and evaluated. Spritepack outperforms current software.

CCS Concepts: • **Information systems** → **Web interfaces**; • **Computing methodologies** → *Image compression*; • **Networks** → *Network performance modeling*; • **Mathematics of computing** → *Combinatorial algorithms;*

Additional Key Words and Phrases: CSS image sprites, load time reduction, web optimization, heuristics, image compression, JPEG, PNG, rectangle packing, web engineering

ACM Reference Format:
Jakub Marszałkowski, Jan Mizgajski, Dariusz Mokwa, and Maciej Drozdowski. 2015. Analysis and solution of CSS-sprite packing problem. ACM Trans. Web 10, 1, Article 1 (December 2015), 34 pages.
DOI: http://dx.doi.org/10.1145/2818377

1. INTRODUCTION

Short web page load time has a great importance for the Internet industry [Weinberg 2000; Marszałkowski et al. 2014]. Contemporary web pages are heavily loaded with small images (icons, buttons, backgrounds, infrastructure elements, etc.), and Jeon et al. [2012] report that 61.3% of all HTTP requests to large-scale blog servers are images, while other static content is only 10.5% of requests. Each image is a resource that 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, for example, 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

Jakub Marszałkowski acknowledges support by the FNR (Luxembourg) and NCBiR (Poland), through IShOP project, INTER/POLLUX/13/6466384.
Authors' addresses: Piotrowo 2, 60-965 Poznań, Poland; emails: Jakub.Marszalkowski@cs.put.poznan.pl, Mizgajski.Jan@gmail.com, axem.pl@gmail.com, Maciej.Drozdowski@cs.put.poznan.pl.

DOI: http://dx.doi.org/10.1145/2818377

(a)

(b)

```
<style>
.image1
{background: url("sprite.png")
0 0 no-repeat;
width:159px; height:188px
}
.image2
{background: url("sprite.png")
0 -188px no-repeat;
width:159px; height:188px
}
.image2:hover
{background: url("sprite.png")
0 -376px no-repeat;
width:159px; height:188px
}
...
</style>
```

(c)

```
<body>
...
<span class="image1"></span>

<span class="image2"
alt="image2.gif">
</span>
...
</body>
```

Fig. 1. Example of a CSS-sprite.: (a) sprite.png image, (b) part of the CSS file locating images, and (c) example of use.

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, the CSS-sprite technique is widely used in many web pages. An example of applying a CSS-sprite is shown in Figure 1. A sprite is shown in Figure 1(a). In order to extract tiles from a sprite Cascading Style Sheets (CSS) are employed in Figure 1(b). Example code using the tiles in the sprite is shown in Figure 1(c).

To the best of our knowledge, the first reference to CSS-sprite packing appeared in Staníček [2003] and was later popularized in Shea [2004]. 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, compare, for example, Marszałkowski and Drozdowski [2013] and Błażewicz and Musiał [2010]. 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 and transmission time and slowing web page rendering. In this article, we 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 problems. We tackle these problems in the following sections. In the next section, realities and the challenges in sprite packing are discussed, then the CSS-sprite packing problem is formulated. Results of preliminary empirical studies conducted to define our solution algorithm are presented in Section 3. In Section 4, current techniques for packing sprites are outlined. Our method of sprite packing is given in Section 5 and evaluated in Section 6. The last section is dedicated to conclusions. The notation used throughout the article is summarized in Table I.

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-sprite packing as an optimization problem. Given a set of images (tiles) in various file formats, we intend

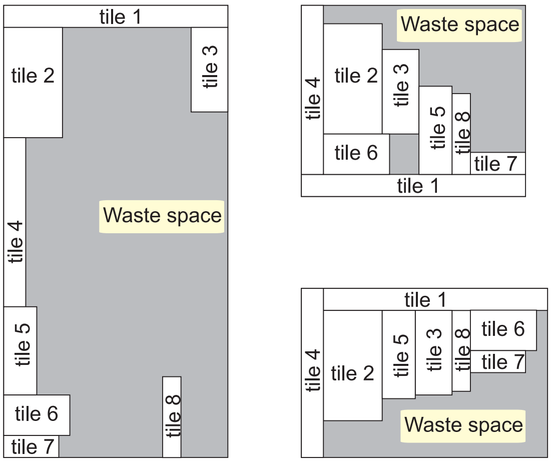


Fig. 2. Examples of CSS-sprite layouts: (a) excessive waste space, (b) vertical layout, and (c) horizontal layout.

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, and (iii) communication performance. The three factors are tightly interrelated, which will be shown in the following sections. There are certainly also other factors related to the browser (e.g., rendering efficiency), server (e.g., cache performance), and so on, but constructing a comprehensive model of their works is beyond the scope of this article, and we take them into account only implicitly.

2.1. Geometric Challenges

One of the factors affecting sprite size(s) is geometric *layout* of the tiles. By layout we mean here mutual alignment of the tiles on the plane. It determines shape, size, and location of empty spaces, and consequently, the total number of pixels in the sprite. We will call the total number of sprite pixels a 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, we deal with a class of regular two-dimensional (2D) packing problems because tiles and sprites are rectangles. Rotation of images is not allowed. Although 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 2D objects for minimum space waste has been tackled very early in glass/paper/metal sheet cutting, in packaging, factory-floor planning, VLSI design, and so on [ARC Project 2013; Christofides and Whitlock 1977; Gilmore and Gomory 1965; Lodi et al. 2002; Ntene and van Vuuren 2009]. Needless to say that 2D cutting/packing problem is computationally hard (precisely **NP**-hard). In practice, it is solved by heuristic algorithms. Unlike in the above classic applications, in sprite packing, we do not use any material sheet that (i) should be conserved and (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 Figure 2(a) has a lot of *waste space* not encoding any tile. It may be argued that the layout in Figure 2(a) is as good as the layouts in Figures 2(b) and 2(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 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 2D packing are distinguished [Lodi et al. 2002]. In the 2D 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-sprite packing because we can choose arbitrary bin sizes and using many bins due to size restrictions has no practical sense here. In the 2D *strip packing* problem (2SP), the 2D objects are put on an infinite strip with one dimension fixed: either the width or the height [ARC Project 2013; Lodi et al. 2002; Ntene and van Vuuren 2009; Steinberg 1997]. This representation is more attractive because we can use numerous algorithms proposed for 2SP. 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. We will call the former case *vertical layout* (see Figures 2(a) and 2(b)). Similarly, we will call the latter option a *horizontal layout* (see Figure 2(c)). In the *rectangle packing* problem (RP), the two dimensions are free to change [Huang and Korf 2009; Korf 2003; Korf et al. 2010; Perdeck 2011]. It is required to find the smallest area bounding box enclosing a set of rectangles. Rectangle packing seems to be closest to sprite packing. 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; and
—determining the assignment of tiles to sprites for good geometric packing.

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 article. An interested reader is recommended to begin with, for example, CompuServe Inc. [1990], International Telecommunication Union [1993], Wallace [1991], and Randers-Pehrson and Boutell [1999]. Let us note that images can be delivered to a browser as data URIs inlined in HTML or CSS text documents [Masinter 1998]. This scheme is out of scope of this article 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. We give two examples. First, 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 **NP**-hard, which can be shown by a transformation from Balanced Complete Bipartite Subgraph problem [Drozdowski and Marszałkowski 2014; Karp 1972]. Second, 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 **NP**-hard because it amounts to Hamiltonian Path problem [Drozdowski and Marszałkowski 2014; Karp 1972]. 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. We will call such a transformation 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 [Bredies and Holler 2012], 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), and (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). 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. One solution may be putting tiles with similar pixels side by side, 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 [Eckert and Bradley 1998; Popovici and Withers 2007] and a current engineering challenge [Mozilla Co. 2014; Davies et al. 2014].

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. We will name this procedure *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 Chikuyonok [2009a, 2009b], Impulse Adventure [2007], Independent JPEG Group [2012], Silverman [2013], and Louvrier [2013].

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. Two consequences can be observed: (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, and (ii) tile-to-sprite distribution has an 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 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 [Ntene and van Vuuren 2009], 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, and
—finding satisfactory trade-off between sprite construction time and solution quality.

2.3. 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 [Simpson 2015]. 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 that 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 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 the quality of a set of sprites, we have to rely on performance models, such as flow models [Velho et al. 2013], preferably an easy-to-calculate formula, representing typical tendencies that can be reasonably traced. Thus, we face a dilemma as to 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, we consider three communication channel performance elements to estimate transfer time: (i) communication latency, (ii) available bandwidth, and (iii) number of concurrent communication channels. We assume that one sprite is transferred over one communication channel, but we abstract away 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 our model, transferring x bytes of data over one channel takes $L + x/B(1)$ seconds. Note that in our 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, our 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 3.2). This opens an opportunity to transfer sprites in parallel. We assume 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, we assume that the total bandwidth $B(c)$ is a function of the number of simultaneously open channels c. Then a single channel bandwidth share is $B(c)/c$. We will denote by $\overline{B} = [B(1), \ldots, B(c_{max})]$ a vector of aggregate bandwidths for different numbers of channels. Suppose that size of sprite i is f_i, for $i = 1, \ldots, m$. The time of transferring the set of sprites $\mathcal{S}$ over c concurrent channels is modeled by the formula:

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

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 (1) is total communication time shared fairly over c channels. The second part is a duration of the single longest communication. Formula (1) represents communications like preemptive tasks scheduled on a set of c parallel processors in the scheduling theory [McNaughton 1959]. Clearly, (1) is an approximation. We assume a simple communication time 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 (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 (1). Thus, formula (1) takes into account the trade-off between the opportunity of transfer time reduction by parallel communication and the cost of issuing a HTTP request for each sprite. Usually $B(c)$ is a nondecreasing sublinear function (see Section 3.2). Consequently, $B(c)/c$ is nonincreasing, and (1) has maximum in one of two trivial cases $c = 1$ or $c = c_{max}$. Hence, to encourage applying a mild number of parallel communication, we will use

$$T(\mathcal{S}) = \min_{c=1}^{c_{max}}\{T(\mathcal{S}, c)\} \tag{2}$$

as the objective function evaluating quality of a set of sprites. We do not take for granted that any aspect of the problem dominates download time, but by optimizing (2), we strike a balance between the number of sprites, their sizes, overheads, and parallelism. However, certain optimization versions may be handled as special cases of (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 section, 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," browser, computer platform. Moreover, many users visit the website, and each of them can be different. Specific load generated by the users also affects the website performance. Hence, we

Table I. Summary of Notation

$B(c)$	accumulated bandwidth of c concurrent communication channels
$\overline{B}$	vector $[B(1), \dots, B(c_{max})]$
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 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

have a population of visitors as well as a population of their performance indicators, and each website is unique with respect to these parameters. In order to take the full advantage of performance optimization, parameters $L, \overline{B}$ should be measured on the actual website and its viewers population. In Section 3.2, we demonstrate how this can be done in practice.

2.4. Problem Formulation

We summarize the introductory discussion by formulating CSS-sprite packing problem. Given is set $\mathcal{T} = \{T_1, \dots, T_n\}$ 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 (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 us summarize possible advantages and costs implied by the above problem formulation. By using objective function (2), we assume user-side performance perception. Applying more than one sprite allows to build better sprites and thus save on 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. We do not predetermine the number of sprites in the solution. 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 class.

3. PRELIMINARY TESTS

As discussed in the previous section, a number of decisions must be made in designing a sprite-packing solution. In this section, we report on the impact of layout choice on the efficiency of the image compression. We also present results of network communication performance evaluation.

3.1. Packing Model

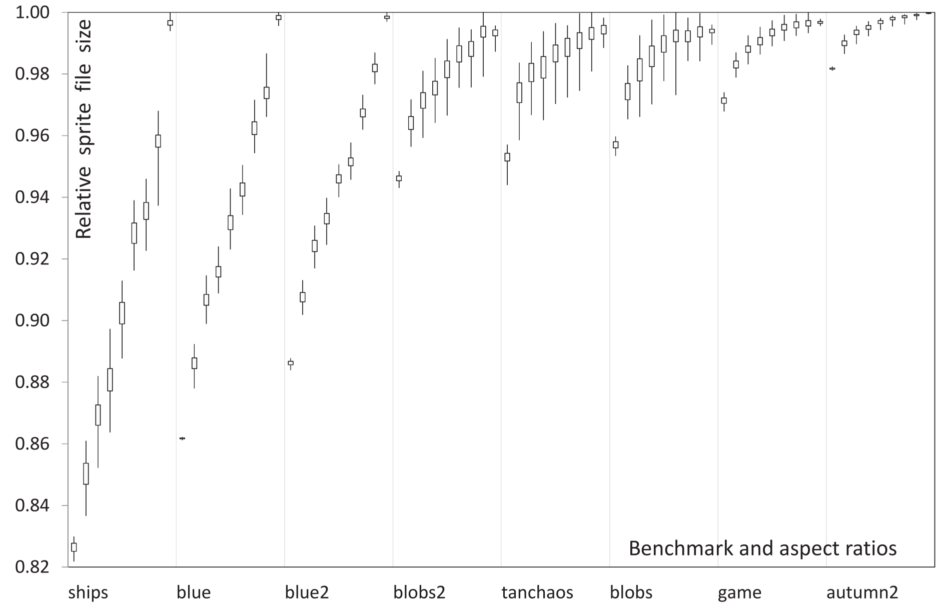
An *aspect ratio* of an image is the ratio of its vertical and horizontal sizes. Vertical and horizontal layouts may be considered the border cases of possible aspect ratios in this sense that one sprite dimension is fixed to a minimum. As noted in Section 2.2, the sprite aspect ratio may influence the 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, 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, nine aspect ratios were tested, which conventionally represent the size of a sprite as a tile array in tile units. Thus, we had 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). 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 [Boutell et al. 2013]. For PNG compression, PNG_ALL_FILTERS setting was selected, which means that in the construction of the compressed image scan line, 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 Figures 3 and 4. On the horizontal axis, different datasets are presented, and for each dataset, 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 Figures 3 and 4 have different ranges on the vertical axes. For clarity of presentation, only a subset of results is shown. It can be verified in Figures 3 and 4 that the datasets can be divided into three groups: with a preference for vertical layout (Figure 3(a)), with a preference for horizontal layout (Figure 3(b)), and datasets without any apparent preference for the aspect ratio (Figure 4). By a preference, we mean here that certain aspect ratio results in the smallest sprite sizes. Out of 36 datasets, 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 (Figures 3(a) and 3(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, for example, the designed algorithm does not need to examine swapping the same-sized tiles between their locations.

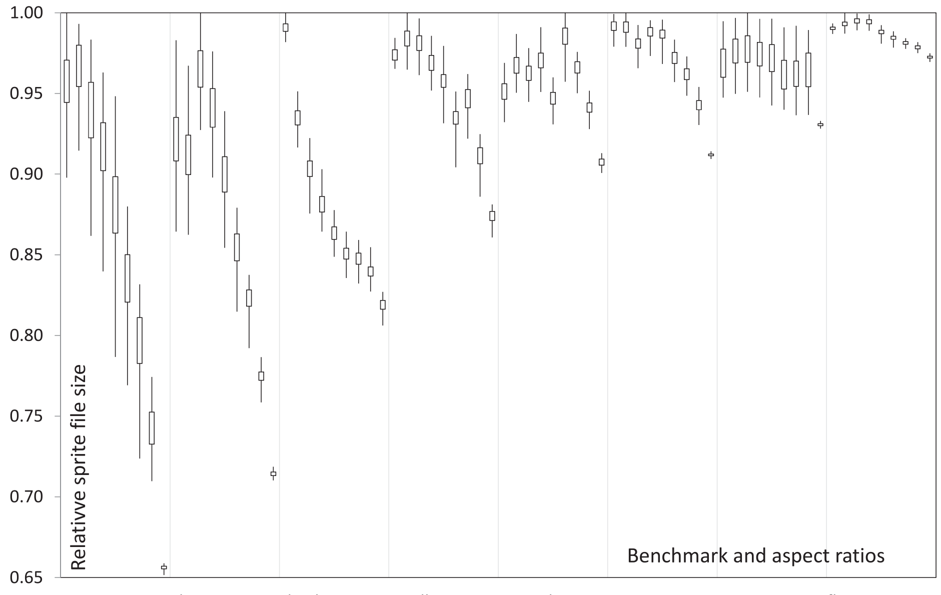
In the case of JPEG image format, no similar preference has been observed. However, size of the output sprite was strongly correlated with the sprite area (number of pixels). Therefore, in the case of JPEG images, it is advisable to eliminate unnecessary waste space.

3.2. Communication Performance

In this section, we demonstrate that performance parameters $L, \overline{B}$ introduced in Section 2.3 can be obtained in practice. Before proceeding to our results, let us explain why using existing performance studies is problematic. As explained in Section 2.3,



(a)



(b)

Fig. 3. Instances with preference for (a) vertical layout ($\frac{x}{y} = \frac{1}{36}$) and (b) horizontal layout ($\frac{x}{y} = \frac{36}{1}$).

communication performance parameters should be measured on the particular web server and its user population. Consequently, latency and bandwidth results that could be obtained using tools like [WebPageTest 2015] are not adequate here because the sprites would be optimized not for the population of real users but for the benchmarking infrastructure. To the best of our 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 [Simon and Souders 2015], but

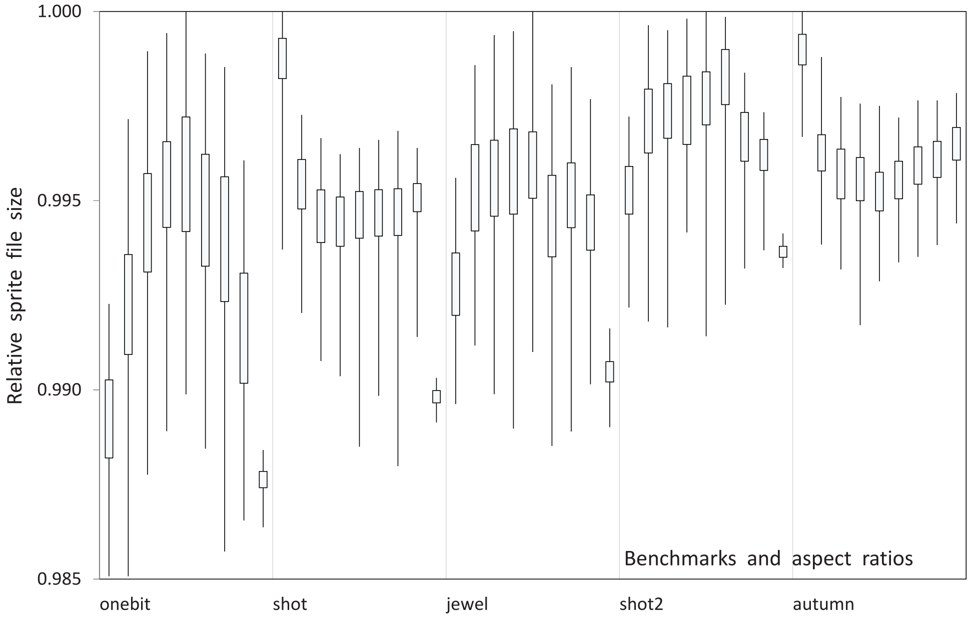
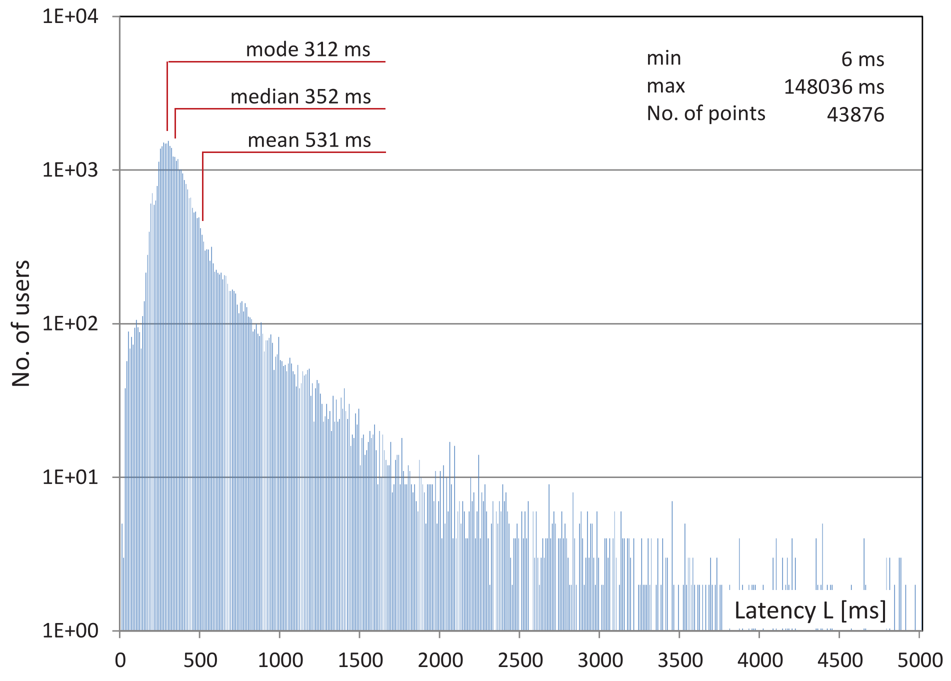


Fig. 4. Instances without strong preference for any aspect ratio.

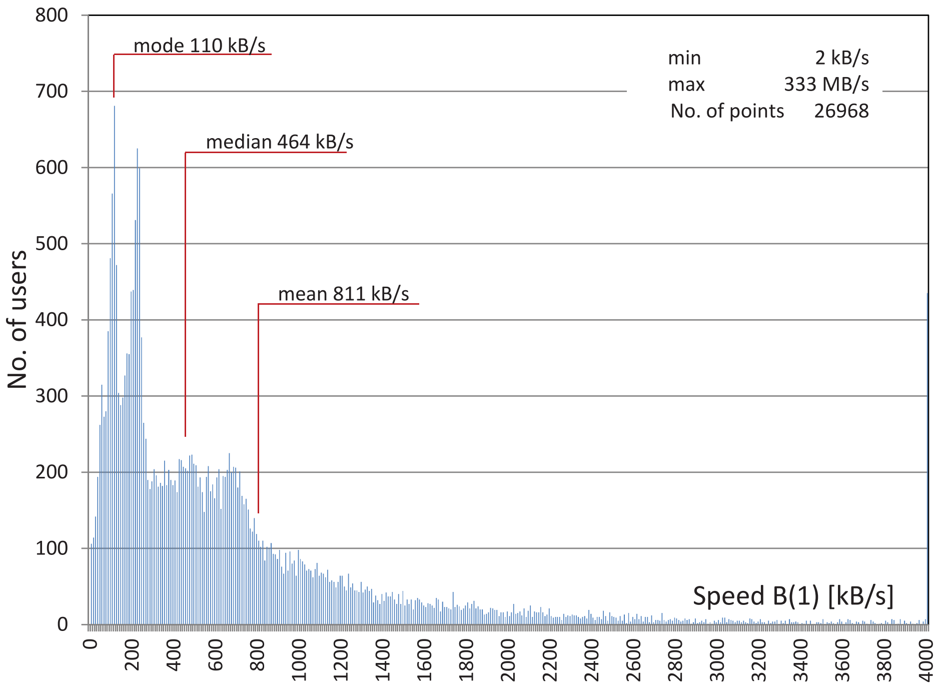
it does not translate directly to the number of parallel channels c_{max} and bandwidth scalability in $\overline{B}$ because these are determined by the server, user platforms, and the "last miles."

Network performance observed by browsers has been tested experimentally. In order to estimate latencies and available bandwidth of user browsers, we have installed a script downloading files of size 1B and 1MB on a web page ranking popularity of over 700 other web pages. Each of the 700 ranked pages had a hyperlink to the page with our script, which users were clicking manually, causing the browser to execute our script as the page was downloaded. Thus, the test page traffic consisted of users coming from over 700 other websites. The variety of linking websites guarantees that the population of visitors was not too uniform. By viewing our web page, the visitors executed the script in their browsers and downloaded the two files using their specific browsers and Internet connections. Since the script was appended to a "production" page, we were able to gather real viewers traffic with their specific network performance features. The times of downloading the two files were collected. According to formula (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, we used t_1 as an estimate of L. Time t_2 of downloading 1MB file has a significant component related to bandwidth. We calculated speed as $B(1) = 1\text{MB}/(t_2 - t_1)$. Measurements with $t_2 \leq t_1$ were rejected. In total, measurements from 17,460 unique IP addresses were collected. Time t_1 was measured 43,876 times, 26,968 measurements with $t_2 > t_1$ were collected, and 277 measurements with $t_2 \leq t_1$ were rejected.

Results of latency measurement are shown in Figure 5(a). It can be seen that latency distribution has a long tail, but the majority of the observations are concentrated around mean, median, and the average value. Over 2/3 of the 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 Figure 5(b). Also speed distribution has a long tail, but over 76% of registered speeds are



(a)



(b)

Fig. 5. Experimental verification of network performance: (a) latency distribution (logarithmic vertical axis) and (b) user download speed distribution.

Table II. Distribution of Browser Parallel Channel Number Limit

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

Table III. Synthetic Results of Parallel Channel Experiment

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

in range [100kB/s, 2MB/s]. The histogram in Figure 5(b) 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 ($\approx$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. This, however, is beyond scope of the article.

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 $B(3) = 1\text{MB}/t(3)$. Three hundred seventy measurements from 276 unique IPs have been collected. Different types of browsers are opening various numbers of parallel connections. Hence, we first verified capability of the user infrastructure in parallel communication. The number of communication channels that 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 II, a cumulated fraction of browsers capable of using at least some number of parallel channels is shown. It can be verified in Table II that all browsers are using at least two parallel channels, in roughly 80% three channels can be used, but only 12% are using eight, nine, or more. Hence, not in all browsers can any speedup in communication be observed if, for example, eight concurrent downloads are started. To compensate for the differences in user bandwidths $B(1)$, in the further discussion we consider bandwidth speedup $B(c)/B(1)$ obtained by using c parallel communications. For the sake of giving the reader a rough impression of the obtained results, Figure 6 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, and (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 III, the results are presented in a more synthetic form. We report speedups $B(c)/B(1)$ obtained in parallel communications. 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 sublinear.

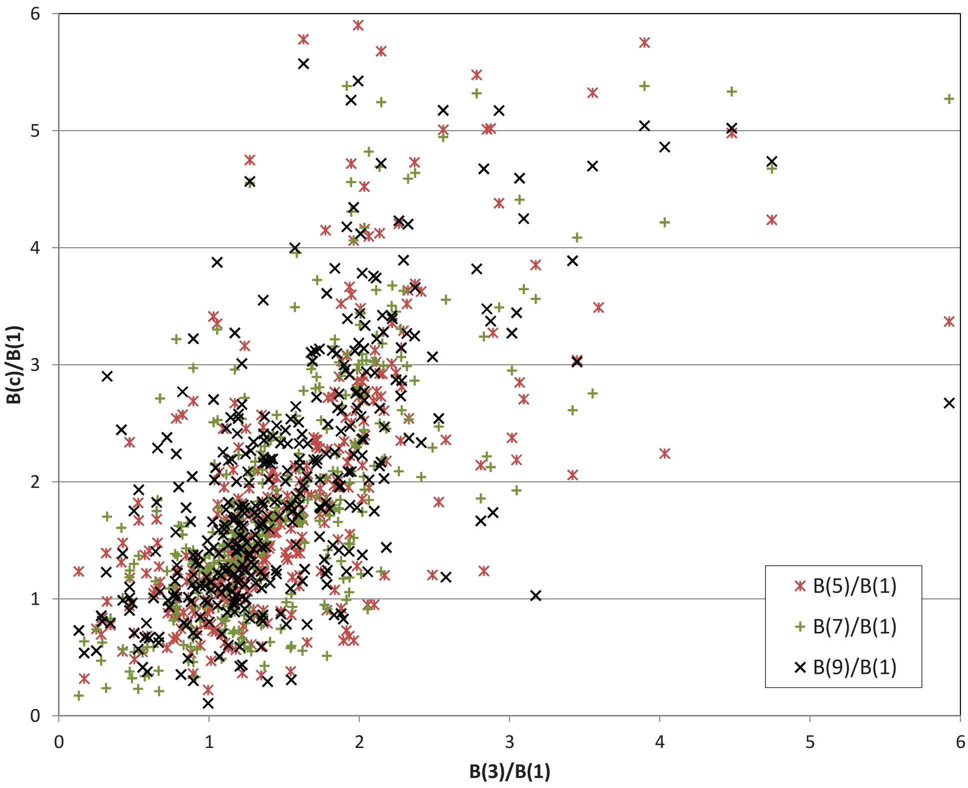


Fig. 6. Speedups $B(c)/B(1)$ vs. $B(3)/B(1)$ in using parallel channels.

4. STATE OF THE ART

Initially, CSS-sprites were constructed manually [Shea 2004]. Here we consider automatic CSS-sprite packing. Since there are many software solutions with little differing names, we will identify them by web addresses and in some cases our own short names. The index of names and addresses is given in Table V.

There are three groups of CSS-sprite generators that have been excluded from further study and evaluation (cf. Tables IV and VI). First, 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 IV. Second, there is a set of applications that could not be included in the further study because we were unable to use them. We mention such cases in Table IV. The specific situations that we encountered were failure to work after installation (Group B in Table IV), dead web applications giving no results (Group C), and sprites with overlapping tiles (Group D).

Further applications are listed in Tables VI and VII. 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 command line 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.

Table IV. Excluded CSS-Sprite Generators

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 V. Index to the CSS-Sprite Packing Solutions

Short Name	cf. Tab.	Web address
aberant	VI	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	VII	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 VI. Solutions Not Using 2D Packing Algorithms

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[T12]	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[1J]	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

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 postoptimization 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 VI, CSS-sprite generators are listed that align tiles in a single column or row. A drawback of these applications is that they construct sprites of very big dimensions and with a lot of wasted space if the number of tiles is big. As a result, sprites built by such applications are not comparable with the sprites obtained by using some geometric packing algorithm. Therefore, we consider them not suitable for real-life industrial use. This is the third set of applications we had to exclude from further comparisons.

Applications using some geometric packing algorithms are listed in Table VII. In a few lucky cases the applied 2D-packing algorithms were identified in the provided software documentation. The algorithm given in [Gordon 2011] is commonly used because its implementation is openly available. As geometric packing is **NP**-hard, most of the applications use some simple greedy heuristics.

To the best of our knowledge, all existing sprite generators build a single output sprite. No solution automatically evaluates options for distributing the tiles into several

Table VII. Solutions Using Some 2D-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 Gordon [2011].
zerocom[T7]	May 8, 2014	online	PNG, PNG8	Tries [Korf and Huang 2012] for 20 seconds. If instance is large, then uses [Chen and Chang 2006].
pypack	Jan 6, 2013	commandline multiplatform (Python)	PNG	Extension of Gordon [2011].
JSGsf[C8]	Aug 08, 2014	commandline multiplatform (Ruby)	PNG	Can be forced to use implementation of Gordon [2011].
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 Guo et al. [2001].
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 [Nuclex Framework 2009].
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 Korf [2003].
shoebox	2014	GUI, multiplatform (Adobe Air)	PNG	Unknown.

[T]Offers tile test sets. [C]Offers CSS test sets. [G]Does not read GIFs. [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.
[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 1,200×1,200px. Hangs on duplicate filenames with different extensions. Allows repacking tiles in sprites given as input.
[10]Crashes on $\geq$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.

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.

5. SPRITEPACK

In this section, we present Spritepack, our method for sprite construction. Given a 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, and (iv) postprocessing. Spritepack has been implemented in C++ using MS Visual Studio 12 and Magick++ API to ImageMagick.

5.1. Tile Classification

With the goal of grouping tiles with similar sets of colors and to retain as low of 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 bits 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), and
(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 was converted to minimum necessary color depth PNG image using Magick++ and saved on file. Only then was the tile reopened and assigned to the appropriate class. A similar procedure was applied to JPEG images. If the JPEG image converted to PNG had a smaller size, then the PNG version was used in the further manipulations. Images with 1, 2, and 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 [Stefanov 2008].

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 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. By a *group,* we will understand 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. The performance of Spritepack under various k settings is discussed in Section 6. Geometric packing is a simple hyperheuristic [Chakhlevitch and Cowling 2008] 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: 2D strip packing (2SP) and rectangle packing (RP). The 2SP comes in two flavors of either horizontal or vertical layout. Since the 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 (FFDH, computational complexity $\mathcal{O}(n \log n)$),
—First-Fit Decreasing Height with Two-Fit (FFDH2F, $\mathcal{O}(n^3 \log n)$),
—Best-Fit Decreasing Height (BFDH, $\mathcal{O}(n \log n)$),
—Best-Fit Decreasing Height with Two-Fit (BFDH2F, $\mathcal{O}(n^3 \log n)$),
—Bottom-Left (BL, $\mathcal{O}(n^2)$),
—Modified Bottom Left (MBL, $\mathcal{O}(n^3)$).

For the RP model algorithm, Variable Height Left Top (VHLT, $\mathcal{O}(n^2 w_0)$) is available. In the following, we give a short description of the above heuristics. A more detailed account can be found, for example, in ARC Project [2013], Lodi et al. [2002], Ntene and van Vuuren [2009], and Perdeck [2011].

In the coming description of 2SP algorithms, we assume vertical layout. It means that we have a strip of the width equal to the widest tile, and in the process of packing, the occupied area extends upward. The heuristics FFDH, FFDH2F, BFDH, BFDH2F are so-called shelf algorithms. It means that they pack the tiles as if on shelves cut from the strip: the bottom lines of the tiles are aligned to the bottom of the shelf. The height of a shelf is determined by the highest rectangle on the shelf. It is required that total width of the rectangles on no shelf exceeds the width of the strip. Thus, shelf algorithms are 2D renditions of 1D bin-packing methods. The above shelf algorithms consider tiles in the order of decreasing height. First-Fit algorithms (FFDH, FFDH2F) place the current tile on the first shelf that can accommodate the width of the tile. Best-Fit algorithms (BFDH, BFDH2F) place the tile on the shelf on which the remaining width is smallest. When placing the current tile closes a shelf (i.e., no single remaining

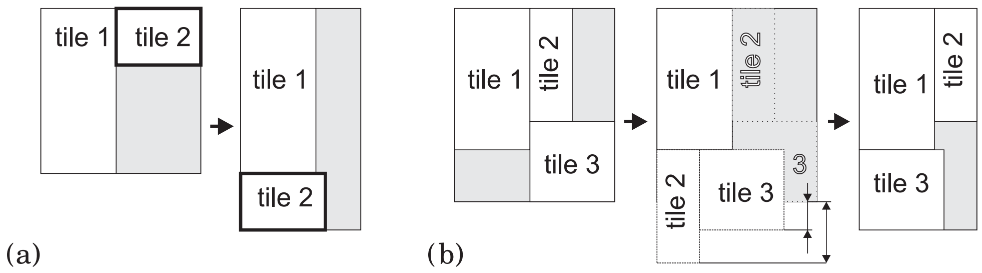


Fig. 7. Increasing bounding box height in VHLT after (a) successful and (b) unsuccessful packing.

tile is able to use the shelf), the Two-Fit algorithms (FFDH2F, BFDH2F) search among the remaining tiles for a pair wider than the current tile and still able to fit on the shelf. BL algorithm [Chazelle 1983] places tiles as close to the bottom and as close to the left edge of the strip as possible. In our implementation (MBL) of BL, 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 that 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 [Perdeck 2011] is inspired by Korf [2003]. In the original description [Perdeck 2011], a horizontal layout is used. Hence, the Left-Top could equally well be referred to as Bottom-Left in the vertical layout rendering. However, in the subsequent description, we stick to the original horizontal setting. The 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 Perdeck [2011] 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, then testing bounding box (w, h) may be skipped and w is decreased again. In Perdeck [2011], 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. Figure 7(a)); and (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 (Figure 7(b)). 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.

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 formula (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 that 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 that 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 include the following: (i) for PNG format, minimum color depth is selected and all filters are tested; and (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, and JPEG compression quality are input parameters of Spritepack.

5.4. Postprocessing

As described in Section 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 is too time-consuming. Therefore, Spritepack takes the opportunity of optimizing sprites as a postprocess 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 postprocessors is customizable and builds on the examples from Louvrier [2013]. In further experiments, postprocessors `pngout` [Silverman 2013] with the option of using its KFlate algorithm and `jpegtran` [Independent JPEG Group 2012] 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).

6. SPRITEPACK EVALUATION

In this section, we report 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 our method and its efficiency but also into the status quo in the web. Unless stated 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, the 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 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 $L = 352\text{ms}$, which is the median in Figure 5(a). 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 the median speed in Figure 5(b) and bandwidth speedups in Table III with additional curve-fitting.

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 VIII, a concise summary on the dataset is collected in Table IX, and further details are provided in Marszałkowski et al. [2015]. 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 IX provide numbers of tiles in GIF, PNG, and 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. We analyzed test sets offered together with the alternative sprite generators described in Section 4. 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 our benchmark making a total of 32 test sets.

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.

6.2. Initial Experiments

In this section we report on performance of Spritepack on a corpus of tile sets (Table IX). 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 (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 twofold consequences. On the one hand,

Table VIII. Test Instance Index

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&section=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

reducing k also reduces processing time because fewer groups of tiles are evaluated in merging with image compression (Section 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.

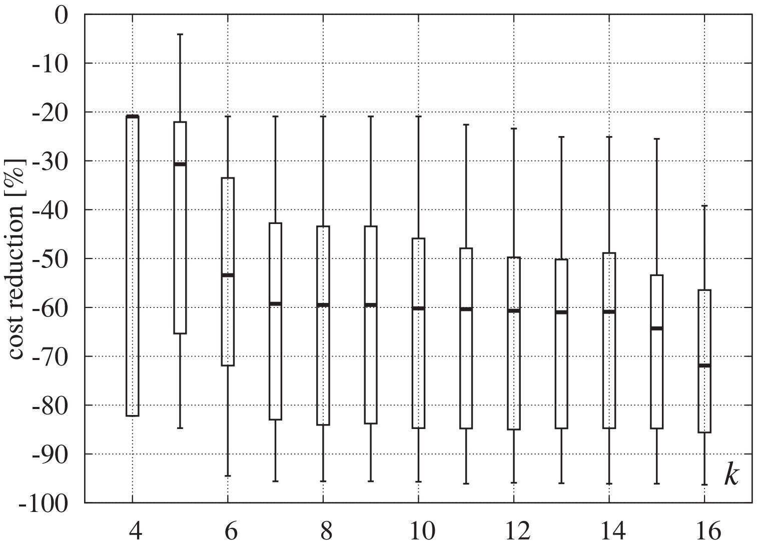
The instances from Table IX have been solved for $k = 4, \ldots, 16$. Since k can be neither greater than the number of tiles n, nor can it be smaller than the number of tile classes plus two (cf. Section 5.2), 320 test instances have been solved in total. The results of this series of experiments are collected in Figures 8–10 and in Tables X and XI. In Figure 8, reduction of the goal function (2) vs. k is shown. The reduction is expressed relative to the value of the goal function $T(\mathcal{T}, 1)$, that is, 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 Figure 8(a), goal function reduction obtained solely by Spritepack is shown, and in Figure 8(b), the reduction obtained in postprocessing 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% to 4%. With growing k, the reductions are better, which is a result of two processes. Indeed there are six test

Table IX. Classification of the Images in Test Instances

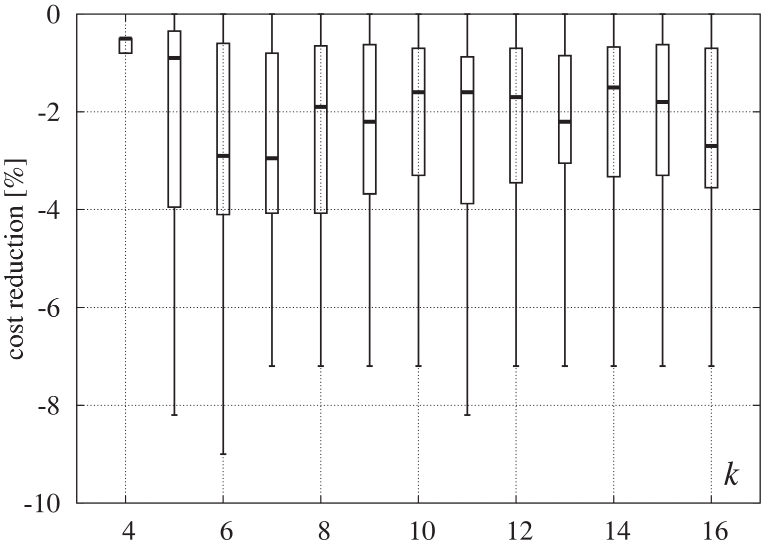
	Original tiles			Spritepack tile classification							Total
Instance name	PNG	GIF	JPEG	PNG8i	PNG8it	PNG8g	PNG8gt	PNG24	PNG32t	JPEG	n
4images_travelphoto	9	41	7	42	8	0	0	1	0	6	57
acoderin	20	0	0	9	6	0	0	4	1	0	20
concrete5_coffee	0	1	14	0	1	0	0	1	0	13	15
dotnetnuke_bright	2	0	34	0	31	0	0	0	1	4	36
drupal_fervens	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	1,028
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	1,456	1,428	248	464	2,193	9	11	39	285	131	3,132

sets where increasing k decreases the objective function as could be expected due to a greater sprite combination flexibility. However, a set of instances that can be applied for a given k also has influence in Figure 8(a). Remember 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 that can be packed with a given k grows from 2 for $k = 4$ to 30 instances for $k = 7, \ldots, 9$ 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 6.1. This observation applies also to Figures 9 and 10. 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 X. In all tests, the biggest number of 10 sprites has been constructed for vbulletin_darkness instance, which had 1,028 tiles. Hence, in the further tests, we used $k = 10$ because it is not restricting the choice of the final sprite number. It can also be observed that Spritepack uses moderate numbers of sprites comparable with the number of browser download channels (see Table II).

As mentioned above, sprite file sizes and the total area are additional performance indicators. Changes in file size are presented in Figure 9(a) for Spritepack alone and in Figure 9(b) for postprocessing. Along the vertical axis, the fraction of the total initial



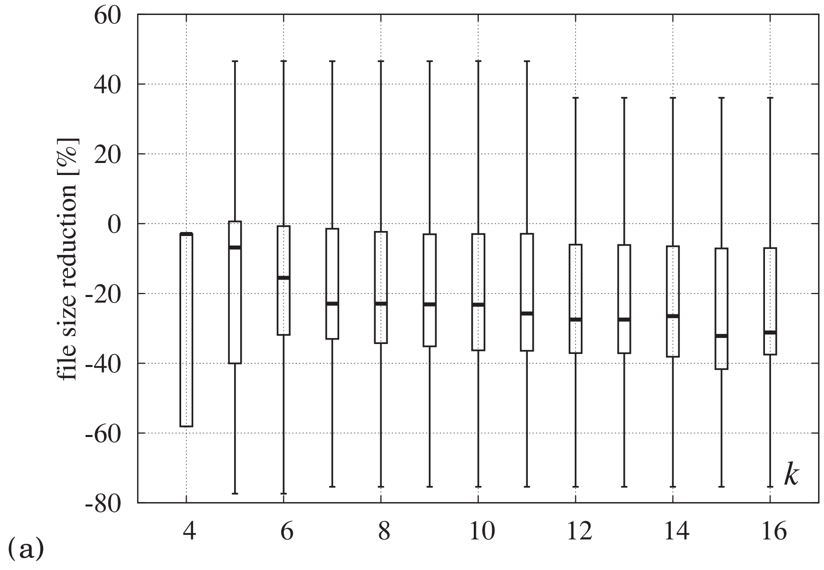
(a)



(b)

Fig. 8. Reduction of communication time estimation (2): (a) Spritepack and (b) postprocessing. Lower is better.

tile sizes by which the Spritepack sprite(s) are smaller is shown. Negative values represent reduction in file size. As shown in Figure 9(b), postprocessing reduces file size by approximately 4% to 7%, which is a useful complement to Spritepack. It can be seen in Figure 9(a) 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 Figure 9(a) 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 Figure 10(a) showing relative increase in image area. It can be seen in Figure 10(a) that usually image area is not increasing



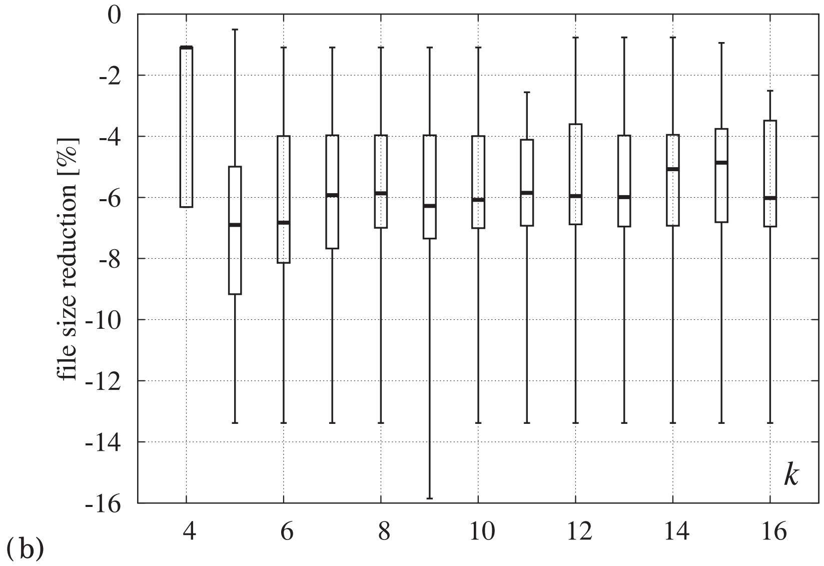
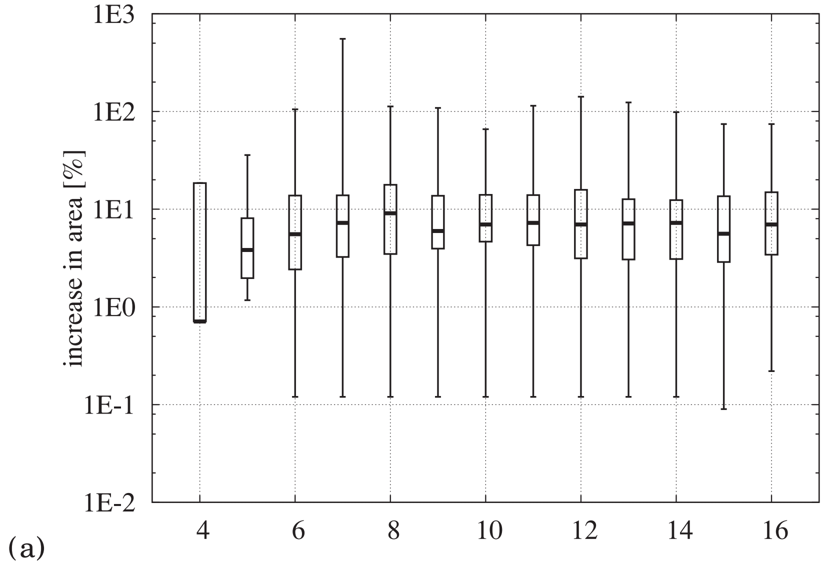


Fig. 9. Reduction in file size: (a) Spritepack and (b) postprocessing. Lower is better. Positive values represent increased file sizes.

more than by 10% to 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 article. Still, Spritepack is



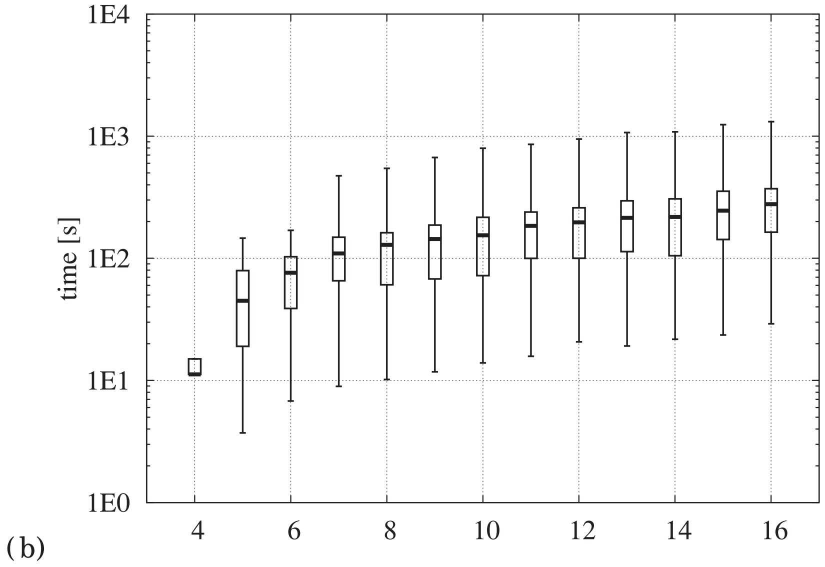


Fig. 10. (a) Change in image area, (b) Processing time. Logarithmic scales. Lower is better.

able to deal with such big tile sets on the basis of web performance. It is demonstrated in Figure 10(a) 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 other things, 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 key factors is image

Table X. Number of Tests vs. the Number of Final Sprites

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 XI. Usage of Geometric Packing Heuristics

MBL(V)	MBL(H)	VHLT	FFDH2F(V)	BL(V)	BL(H)	FFDH2F(H)	BFDH2F(V)	FFDH(V)
24,135	5,381	2,829	208	68	21	17	8	5

area and color depth. In our test sets, tiles had various sizes and color depths. Average execution time per tile in all our test sets was 4.59s 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. Figure 10(b) 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 website 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%, and 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, we registered frequencies of using certain geometric packing algorithms. The results are shown in Table XI. The first line of Table XI contains names of heuristics whose 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 XI 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 (FFDH, FFDH2F, BFDH2F) are hardly ever used. The BFDH method mentioned in Section 5.2 has not been used at all. It seems that reducing the set of geometric packing algorithms to just MBL, VHLT, FFDH2F may be a reasonable option to curb Spritepack complexity in production systems. Contrarily, to obtain better results the geometric packing algorithms should outperform the MBL.

6.3. Spritepack Performance Comparison

In this section we compare Spritepack with alternative sprite generators. For the reasons discussed in Section 2, comparing sprite generators rigorously and fairly is not easy. Moreover, a great number of sprite generators exist. Therefore, we applied the following procedure. 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 that have been most reliable, versatile, and provided smallest sprites. In the second series of tests, the selected generators have been compared with Spritepack in generating sprites for all instances from Table IX.

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 (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 XII. 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 XII. Comparison of Sprite Generators on Size of Output
Lower is better. Spritepack is 100%. Spritepack was forced to create a single file.

instance name:	acoderin	modx_ creatif	Sprite Creator	squirrel mail _outlook	joomla_busi nes14a	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]	7,274	395,393	28,663	69,714	190,145	–

Table XII. 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. The line labeled "input" expresses size of the input tiles relative to the single Spritepack sprite. An empty entry in Table XII means that a certain generator has not been able to construct a sprite. Four alternative sprite generators that 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 (2), and size of the output sprites on a complete set of instances from Table IX. However, it turned out that Spritepack outperformed the alternative generators, and their results were extremely bad. For example, the shoebox generator, which was best in the previous set of tests, returned sprites, which had objective (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 (1,028 tiles), shoebox stopped reacting (hang) on tile 666. There are various reasons for such situations, 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), and (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 our evaluation may be deemed unfair. Therefore, to

Table XIII. Evaluation of Best Sprite Generators on 32 Test Instances. Lower is better. Spritepack is 100%

	shoebox	spcanvas	simpreal	isaccc
objective function (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

make the conditions of the comparison more compatible with the above assumptions and easier for the alternative generators, we limited (only for them) the set of the tiles subjected to sprite construction to the tiles of file size below 10kB. As a result, each tile set has been split into a number of tiles that have not been combined into a sprite and a set of tiles that 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 (2) has been calculated in the same way as in the Spritepack. In this experiment, Spritepack still operated on the whole datasets comprising all the tiles and produced as many sprites as it found effective.

The results of this series of experiments are collected in Table XIII. The four alternative sprite generators have been compared in two criteria: objective function (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 (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 (2), it is not surprising that some other method performs better on the sprite size criterion.

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, our communication performance model, and objective function (2), to evaluate the advantages of applying sprites in general and Spritepack in particular. Furthermore, we compared Spritepack and shoebox generator performance. 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 3.2 has been designed and inserted into a web page analyzed in Section 3.2. By viewing the page, users downloaded the tiles in the above three alternative ways consecutively: first all of them separately, next as a single shoebox sprite, and 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 3.2 and had been applied to build sprites by Spritepack. Detailed parameters of the test instances are shown in Table XIV. The instances were chosen to represent a spectrum of possible situations: from modx_ecolife tile set of size smaller

Table XIV. Sprites in End-to-End Test of Sprite Generators

	input		shoebox		Spritepack	
Instance name	files	size [B]	sprites	size [B]	sprites	size [B]
magneto_hardwood	9	373,610	1	482,828	3	294,128
modx_ecolife	10	50,947	1	366,663	3	48,891
mojoportal_thehobbit	39	218,993	1	726,364	7	154,486
oscommerce_pets	203	1,201,692	1	1,683,872	6	673,785

Table XV. Time Results of the End-to-End Evaluation in Real-World Setting

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

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 XV. For oscommerce_pets, the biggest tile set with over 203 tiles, 2,274 measurements were collected. For the remaining tile sets, the number of measurements exceeded 4,000, and, for example, for modx_ecolife, 5,057 samples were collected. The second and fifth columns in Table XV ("input") represent all the tiles sent independently (i.e., not sprited). It can be seen that using a single sprite, as in shoebox, may halve the download time. Yet, such reductions do 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 to 4. In absolute terms, it was from approximately 350ms to 2.4s (medians of differences), while the reduction from shoebox single sprite download time was 140ms to 800ms. It can be concluded that judiciously 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 (2) as a model of the download time. The coefficient of correlation between the medians of download times and the objective function (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 XV, function (2) can be considered an effective guide in sprite optimization process.

7. CONCLUSIONS

In this article, the problem of effective construction of CSS-sprites for web applications has been considered. This problem poses a number of theoretical and practical challenges. On the theoretical side, it is a matter of constructing effective heuristics when the evaluation of one solution is time-consuming. It is also difficult to grasp in a tractable way the complexity of the network communication performance. On the practical side, it is a matter of, for example, tuning the algorithms for particular tile datasets, choosing image compression setting, obtaining network performance indicators, and finding a good trade-off between solution quality and processing time. We have proposed and implemented in Spritepack an approach that 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. Our 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. As the key ingredients of Spritepack, we consider (i) the geometric packing method, which is a fast hyperheuristic operating on low-level geometric packing algorithms; (ii) verifying many options for effective image compression; and (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 **NP**-hard problem, it can be concluded that our method builds quality sprites in reasonable time and compares well with the alternative methods. Spritepack source code is available in Marszałkowski et al. [2015].

It seems technically feasible to improve Spritepack, for example, 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 [HTTPbis Working Group 2015] 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.

REFERENCES

ARC Project. 2013. Survey on Two-Dimensional Packing. Retrieved from http://cgi.csc.liv.ac.uk/∼epa/survey.pdf.

Jacek Błażewicz and Jedrzej Musiał. 2010. E-commerce evaluation-multi-item internet shopping, optimization and heuristic algorithms. In *Operations Research Proceedings*, B. Hu et al. (Ed.). Springer-Verlag, Berlin, 149–154.

Thomas Boutell, Pierre Joye, and PHP.net. 2013. GD Graphics Library. Retrieved from http://libgd.bitbucket.org/.

Kristian Bredies and Martin Holler. 2012. A total variation-based JPEG decompression model. *SIAM J. Imaging Sci.* 5, 1 (2012), 366–393.

Konstantin Chakhlevitch and Peter Cowling. 2008. Hyperheuristics: Recent developments. In *Adaptive and Multilevel Metaheuristics*, C. Cotta et al. (Ed.). Studies in Computational Intelligence, Vol. 136. Springer-Verlag, Berlin, 3–29.

Bernard Chazelle. 1983. The bottom-left bin-packing heuristic: An efficient implementation. *IEEE Trans. Comput.* 32, 8 (1983), 697–707.

Tung-Chieh Chen and Yao-Wen Chang. 2006. Modern floorplanning based on B*-tree and fast simulated annealing. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 25 (2006), 637–650.

Sergey Chikuyonok. 2009a. Clever JPEG Optimization Techniques. Retrieved from http://www.smashingmagazine.com/2009/07/01/clever-jpeg-optimization-techniques/.

Sergey Chikuyonok. 2009b. Clever PNG Optimization Techniques. Retrieved from http://www.smashingmagazine.com/2009/07/15/clever-png-optimization-techniques/.

N. Christofides and C. Whitlock. 1977. An algorithm for two-dimensional cutting problems. *Oper. Res.* 25, 1 (1977), 30–44.

CompuServe Inc. 1990. Graphics Interchange Format. Retrieved from http://www.w3.org/Graphics/GIF/spec-gif89a.txt.

Andy Davies, Gregor Fabritius, Neil Jedrzejewski, Alessandro Lenzen, Claus Meteling, André Roaldseth, Christian Schäfer, and Yoav Weiss. 2014. Adept - The Adaptive JPG Compressor. Retrieved from https://github.com/technopagan/adept-jpg-compressor/.

Maciej Drozdowski and Jakub Marszałkowski. 2014. *On the Complexity of Sprite Packing*. Technical report. RA-07/2014, Institute of Computing Science, Poznań University of Technology.

Michael Eckert and Andrew Bradley. 1998. Perceptual quality metrics applied to still image compression. *Signal Processing* 70 (1998), 177–200.

P. C. Gilmore and R. E. Gomory. 1965. Multistage cutting stock problems of two and more dimensions. *Oper. Res.* 13, 1 (1965), 94–120.

Jake Gordon. 2011. Binary Tree Bin Packing Algorithm. Retrieved from http://codeincomplete.com/posts/2011/5/7/bin_packing/.

Pei-Ning Guo, Toshihiko Takahashi, Chung-Kuan Cheng, and Takeshi Yoshimura. 2001. Floorplanning using a tree representation. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 20, 2 (2001), 281–289.

HTTPbis Working Group. 2015. Hypertext Transfer Protocol Version 2. Retrieved from https://tools.ietf.org/html/draft-ietf-httpbis-http2-17.

Eric Huang and Richard E. Korf. 2009. New improvements in optimal rectangle packing. In *Proceedings of the 21st International Jont Conference on Artificial Intelligence (IJCAI'09)*. 511–516.

Impulse Adventure. 2007. What Is an Optimized JPEG? Retrieved from http://www.impulseadventure.com/photo/optimized-jpeg.html.

Independent JPEG Group. 2012. Jpegtran. Retrieved from http://jpegclub.org/jpegtran/.

International Telecommunication Union. 1993. Recommendation T.81: Information Technology - Digital Compression and Coding of Continuous-Tone Still Images - Requirements and Guidelines. Retrieved from http://www.w3.org/Graphics/JPEG/itu-t81.pdf.

Myeongjae Jeon, Youngjae Kim, Jeaho Hwang, Joonwon Lee, and Euiseong Seo. 2012. Workload characterization and performance implications of large-scale blog servers. *ACM Trans. Web (TWEB)* 6, 4 (2012), 16.

Richard M. Karp. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher (Ed.). Plenum Press, New York, 85–103.

Richard E. Korf. 2003. Optimal rectangle packing: Initial results. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS'03)*. American Association for Artificial Intelligence, Palo Alto, CA, 287–295.

Richard E. Korf and Eric Huang. 2012. Optimal rectangle packing: An absolute placement approach. *J. Artif. Intell. Res.* 46 (2012), 47–87.

Richard E. Korf, Michael D. Moffitt, and Martha E. Pollack. 2010. Optimal rectangle packing. *Ann. Oper. Res.* 179, 1 (2010), 261–295.

Andrea Lodi, Silvano Martello, and Michele Monaci. 2002. Two-dimensional packing problems: A survey. *Eur. J. Oper. Res.* 141, 2 (2002), 241–252.

Cédric Louvrier. 2013. Optimisation Web (Images, Performance). Retrieved from http://css-ig.net/.

Jakub Marszałkowski and Maciej Drozdowski. 2013. Optimization of column width in website layout for advertisement fit. *Eur. J. Oper. Res.* 226, 3 (2013), 592–601.

Jakub Marszałkowski, Jedrzej M. Marszałkowski, and Maciej Drozdowski. 2014. Empirical study of load time factor in search engine ranking. *J. Web Eng.* 13, 1&2 (2014), 114–128.

Jakub Marszałkowski, Jan Mizgajski, Dariusz Mokwa, and Maciej Drozdowski. 2015. Spritepack Resources. Retrieved from http://www.cs.put.poznan.pl/mdrozdowski/spritepack/.

Larry Masinter. 1998. RFC 2397: The "Data" URL Scheme. Retrieved from https://www.ietf.org/rfc/rfc2397.txt.

Robert McNaughton. 1959. Scheduling with deadlines and loss functions. *Manage. Sci.* 6, 1 (1959), 1–12.

Mozilla Co. 2014. Mozilla JPEG Encoder Project. Retrieved from https://github.com/mozilla/mozjpeg/.

Nthabiseng Ntene and Jan H. van Vuuren. 2009. A survey and comparison of guillotine heuristics for the 2D oriented offline strip packing problem. *Discrete Optim.* 6, 2 (2009), 174–188.

Nuclex Framework. 2009. Rectangle Packing. Retrieved from http://nuclexframework.codeplex.com/wikipage?title=Rectangle.

Matt Perdeck. 2011. Fast Optimizing Rectangle Packing Algorithm for Building CSS Sprites. Retrieved from http://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu.

Irina Popovici and Wm. Douglas Withers. 2007. Locating edges and removing ringing artifacts in JPEG images by frequency-domain analysis. *IEEE Trans. Image Process.* 16, 5 (2007), 1470–1474.

Glenn Randers-Pehrson and Thomas Boutell. 1999. PNG (Portable Network Graphics) Specification. http://www.libpng.org/pub/png/spec/1.2/PNG-Contents.html. (1999).

Dave Shea. 2004. CSS Sprites: Image Slicing's Kiss of Death. Retrieved from http://www.alistapart.com/articles/sprites.

Ken Silverman. 2013. Ken Silverman's Utility Page. Retrieved from http://advsys.net/ken/utils.htm.

Lindsey Simon and Steve Souders. 2015. Browserscope. Retrieved from http://www.browserscope.org/?category=network&v=1.

Kyle Simpson. 2015. Obsessions: HTTP Request Reduction. Retrieved from http://blog.getify.com/obsessions-http-request-reduction/.

Petr Staníček. 2003. CSS Technique: Fast Rollovers Without Preload. Retrieved from http://wellstyled.com/css-nopreload-rollovers.html.

Stoyan Stefanov. 2008. Image Optimization, Part 3: Four Steps to File Size Reduction. Retrieved from http://yuiblog.com/blog/2008/11/14/imageopt-3/.

A. Steinberg. 1997. A strip-packing algorithm with absolute performance bound. *SIAM J. Comput.* 26, 2 (1997), 401–409.

Pedro Velho, Lucas Mello Schnorr, Henri Casanova, and Arnaud Legrand. 2013. On the validity of flow-level TCP network models for grid and cloud simulations. *ACM Trans. Model. Comput. Simul.* 23, 4 (2013), 23.

Gregory K. Wallace. 1991. The JPEG still picture compression standard. *Commun. ACM* 34, 4 (1991), 30–44.

WebPageTest. 2015. WebPageTest. Retrieved from http://www.webpagetest.org/.

Bruce D. Weinberg. 2000. Don't keep your Internet customers waiting too long at the (virtual) front door. *J. Interact. Marketing* 14, 1 (2000), 30–39.

Received November 2014; revised August 2015; accepted August 2015