Tag-Cloud Drawing: Algorithms for Cloud Visualization

Owen Kaser
University of New Brunswick
Saint John, NB, Canada
o.kaser@computer.org

Daniel Lemire
Université du Québec à Montréal
Montreal, QC, Canada
lemire@acm.org

ABSTRACT

Tag clouds provide an aggregate of tag-usage statistics. They are typically sent as in-line HTML to browsers. However, display mechanisms suited for ordinary text are not ideal for tags, because font sizes may vary widely on a line. As well, the typical layout does not account for relationships that may be known between tags. This paper presents models and algorithms to improve the display of tag clouds that consist of in-line HTML, as well as algorithms that use nested tables to achieve a more general 2-dimensional layout in which tag relationships are considered. The first algorithms leverage prior work in typesetting and rectangle packing, whereas the second group of algorithms leverage prior work in Electronic Design Automation. Experiments show our algorithms can be efficiently implemented and perform well.

Categories and Subject Descriptors

H.3.3. [**Information Storage and Retrieval**]: Information Search and Retrieval; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Experimentation, Measurements

Keywords

Optimization, Tags, Visualization, Layout

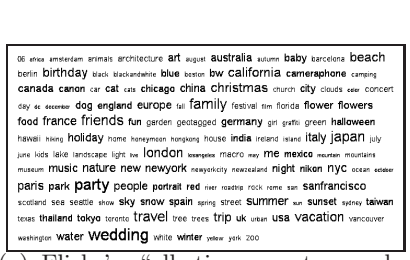
1. INTRODUCTION

Tag clouds have become a popular method to support navigation and retrieval of tagged data. This paper seeks to optimize the display of tag clouds, without concern for the origin of the tags. We follow Hassan-Montero and Herreno-Solana [14] in assuming that associated tags ought to be placed near one another.

Tag clouds conceptually resemble histograms, but whereas histograms are typically used to represent the frequencies of perhaps a dozen items, tag clouds can represent the frequencies of a hundred items. By displaying tags as hyperlinks, we obtain interaction possibilities lacking in histograms. In collaborative tagging, users are motivated to contribute tags to change the appearance of the tag cloud [25].

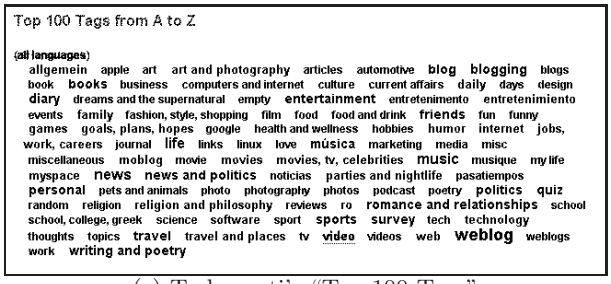
Since the font size of a displayed tag is usually used to show the relative importance or frequency of the tag, a typical tag cloud contains large and small text interspersed.

WWW2007, May 8–12, 2007, Banff, Canada.

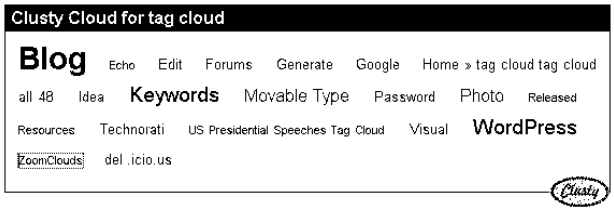


(a) Flickr's "all time most popular tags"

(b) Sample from ZoomClouds.com



(c) Technorati's "Top 100 Tags"



(d) Result from Clusty.com, search "tag cloud"

Figure 1: Tag clouds from popular Web sites.

A consequence is wasteful white space that is problematic in small-display devices, such as PDAs and cell phones, or in tight HTML designs. Moreover, clumps of white space are not, in the authors' opinion, æsthetically pleasing (see Fig. 1). As users, we prefer to work with attractive information displays, but inline text support in HTML is designed for paragraphs, not clouds. While prettier tag clouds can be generated using images, browser-specific technologies (ActiveX), plugins (Flash), or complex HTML (using absolute positioning), using only simple HTML with inline text or tables remains the commonplace approach and offers interesting challenges.

This paper tackles the two problems of wasted space and large clumps of white space identified above. It also considers grouping related tags. We solve the first problem by noting that it is essentially the same as the floorplanning/placement problem that has been tackled for at least 30 years in the field of electronic design automation (EDA). Thus, we propose the classic EDA algorithm, *min-cut placement* [3], for area minimization and clustering in tag clouds. The result looks unconventional because tags are not placed on lines, but it is supported by nested HTML tables.

The second problem's solution has a conventional appearance that does not disrupt the left-to-right, top-to-bottom order of tags. It is a hybrid of the classic Knuth-Plass algorithm [20] for text justification, and a book-placement exercise considered by Skiena [32]. In the authors' view, the resulting tag clouds are visually improved and tighter.

2. RELATED WORK

Tag clouds have been attributed [37] to Coupland [6] but have been popularized by the Web site Flickr [39] launched in 2004. They have since appeared on numerous Web sites including Technorati [36], del.icio.us [38], and so on. While a mere visual representation technique, tag clouds are commonly associated with folksonomies and social software.

Graph drawing [7] is a branch of graph theory typically concerned with the generation of two-dimensional representations of graphs that are easy to understand and pleasing to the eye. While there is no absolute metric for æsthetic, experimental evidence suggests it is important to minimize the number of edge crossing [28]. Other metrics include symmetry, orthogonality, maximization of minimum angles, and so on.

Unlike graph drawing, tag-cloud drawing has received little attention. Hassan-Montero and Herrero-Solana [14] have proposed improving tag-cloud layouts by clustering similar tags together and discarding some tags. Millen et al. [26] have proposed that the user be dynamically able to remove the less significant tags; they have also added an index so that tags can found faster in large clouds. Bielenberg [2] has proposed circular clouds, as opposed to the typical rectangular layout, where the most heavily weighted tags appear closer to the center. However, clouds are only one specific instance of tag representation. For example, Dubinko et al. [8] have proposed a model to represent tags over a time line whereas Russel [30] has proposed `cloudalicious`, a tool to study the evolution of the tag cloud over time. Jaffe et al. [16] have integrated tag clouds inside maps for displaying tags having geographical information, such as pictures taken at a given location.

The problem of improving the layout of HTML pages through special-purpose algorithms has received some attention: Hurst et al. [15] showed that it is possible to make HTML tables significantly more appealing. Generally, there is ongoing work to improve the layout of text in HTML pages using Cascading Style Sheet (CSS) [11].

3. BACKGROUND

The current paper builds on previous work in automated typesetting and previous work in EDA. Minimal EDA background is required to appreciate our claim that tag-cloud layout can be accomplished with EDA tools, but more is required for a deeper understanding of the details and limitations of our approach.

3.1 Typesetting

Automatic typesetting systems, such as TEX [19], must quickly fit text onto the page. The result must be visually attractive. We should break lines so that there is an even amount of space between words.

A greedy approach fits as many words per line as possible, beginning a new line whenever further words cannot be placed on the current line, with the possibility of sometimes slightly squeezing the spaces between words and letters or hyphenating a word. Sneep [33] reports that this is the approach used by most Web browsers (Microsoft Internet Explorer, Firefox, Apple's Safari, and Opera) and most word processors. We know of no Web browser that can hyphenate text. Our own investigations of the Firefox 2.0 browser lead us to believe that, for English text, line breaking is achieved using a simple greedy approach, since no squeezing of spaces between words or letters was observed, and text justification of a sequence of inline elements is achieved by inserting unreported pixels between some elements. An advantage of the greedy approach is it can be done on-line, without waiting for the end of a paragraph. Indeed, a browser should start displaying content before the page has been completely loaded. Unfortunately, the greedy approach can (and frequently does) produce suboptimal solutions.

For TEX, Knuth and Plass [20] compute an optimal solution elegantly, using dynamic programming. Given a line of text, the difference between the preferred width of the text as dictated by the chosen font and the page (or column) width is used to compute the *badness* of the fit. Additional penalties handle hyphenated words and variations in the tightness of lines. Their *total-fit* algorithm minimizes the sum of the squares of each line's badness. Excluding hyphenation and penalties, we summarize their algorithm. We label the words of a paragraph from 1 to n. Let $b_{k,j}$ be the badness measure resulting from a line containing the words k to j inclusively with the convention that $b_{k,j} = 0$ when $k > j$. Let t_j be the minimal possible sum of squares of the line badnesses when the j^{th} word ends a line with the convention that $t_0 = 0$. We have that $t_j = \min_{k \le j}(t_k + b^2_{k+1,j})$ with an exception if $j = n$: the last line can be shorter without the same type of penalty. For $j > 1$, let $K_j = \arg\min_k(t_k + b^2_{k+1,j})$ be the last word of the line prior to the one ending with the j^{th} word. We can compute K_j for all possible $j = 1, \ldots, n$ in time $O(n^2)$ and $O(n)$ space. We can then reconstruct the optimal solution recursively with the following line breaks: $n, K_n, K_{K_n}, \ldots, 0$.

If our tags must be presented in a given order, and if all tags have the same height, then this approach can be used to lay out a cloud optimally. However, clouds have tags of various heights which can be reordered, colored, etc.

3.2 EDA: Physical Design

Techniques for electronic design automation (EDA) have received much research attention in the past few decades. Within the EDA field, *physical design* of VLSI refers to the process of translating from high-level logical circuit descriptions down to a specification of the locations and shapes of individual transistors, wires, and so forth. Today, designs are frequently composed of a mixture of custom-designed blocks of circuitry and licensed "Intellectual Property (IP)

blocks" of pre-designed circuitry. See Lengauer [21] for more information on physical design.

Placement and *floorplanning* are two closely related stages during many physical design flows. Both concern the assignment of blocks of circuitry to locations on the chip. For instance, two submodules in a design might include (rectangular) IP-blocks for a ROM and a shift register. Placement/floorplanning might decide that the ROM should have its lower-left corner at (0,0) on the chip, rotated by 90 degrees, and the shift register should be rotated 180 degrees and have its lower-left corner at (200,200). This decision avoids module overlap and leaves enough blank space for the interconnection wires between them that the subsequent *routing* phase can succeed, while not leaving excessive space between the items, as small chips are preferred.

While it is sometimes observed [29] that, mathematically, floorplanning and placement solve the same problem, from a practical viewpoint they are applied differently. Floorplanning is often done early in the design stage, sometimes before the designs of the submodules are begun. Using estimates of the area required for submodules (and constraints on the aspect ratio), during floorplanning we not only choose module locations, but we also choose module shapes. Then the modules can be custom designed according to the required shapes. As module design progresses, more accurate shape estimates may require that floorplanning be re-done. Floorplanning gives a "bird's eye" view of the layout, based on incomplete area and wiring estimates. Placement, on the other hand, is typically done with complete knowledge of module shapes, the locations of interconnect "pins" on the boundaries of the modules, and so forth.

The scenario presented assumed that floorplanning is done with *soft modules* whose aspect ratios can vary as needed. IP blocks give rise to *hard modules*, whose shape cannot be adjusted. A further case arises in floorplanning when a collection of logically equivalent hard modules are available.

A final distinction between floorplanning and (final) layout is that the former is iterated, often while a human designer is exploring design alternatives. Thus, floorplanning must be fast. In contrast, during final placement, the quality of solution is more important than the running time.

Despite the conventional distinction of floorplanning from placement, recent tools [29] blur the distinction.

3.2.1 Placement Approaches in EDA

Placement problems are typically NP-hard: even 2-d packing problems that ignore routing are intractable [23]. Therefore many heuristics have been proposed. Approaches include force-directed placement (e.g., considered recently by Kennings and Vorwerk [17]), where modules are attracted to modules with which they are strongly interconnected, and repulsed by nearby modules in general (to try to reduce overlap). Force-directed methods have been adapted for graph drawing [10, 13]. When solution quality is more important than speed, metaheuristics such as simulated annealing [18] are often used to guide semi-exhaustive searches. Such approaches would be justified for clouds computed once and accessed many times. Consider a tagging site's list of hot tags for the previous month, optimized for a common display size.

For speed, *min-cut placement* [3] is often chosen. Since we envision tag clouds generated on-the-fly by a server, we adapt min-cut placement to tag-cloud display.

4. MODELS FOR CLOUD OPTIMIZATION

We consider two æsthetic models, one for tags as inline text and one for tags in nested HTML table.

4.1 Tag Clouds with Inline Text

A tag cloud with inline text is a paragraph (block) made exclusively of inline HTML elements such as `span`, `font`, `em`, `b`, `i`, `strong`, `a`, and `br`. A tag, even one with spaces, must remain on a single line. White space outside the tags is in a given default font and font size. Any area outside a tag, but inside the tag cloud will be referred to as "white", irrespective of the background color. The fonts and font sizes corresponding to different tags are enforced using inline elements with, for example, the HTML `style` attribute. The width available to the tag cloud is also determined depending on the page layout, but the height of the tag cloud is assumed to be a free parameter. Naturally, the fonts and font sizes as well as the tag-cloud width are determined by the Web browser as well as by the page content. While the CSS properties `letter-spacing` and `word-spacing` allow us to change the width of phrases, there are implementation-specific limitations. Our primary view has the width and height of each tag fixed, although we consider relaxing this restriction in Sect. 5.2.4. Similarly, the horizontal space between tags must be at least as large as the normal space in the default font. Hence, we will not include a penalty for squeezing tags or spaces. This is in line with the current breed of Web-browser layout engines.

While tags are commonly ordered alphabetically in clouds, we find no evidence that users actually browse tag clouds alphabetically. For large clouds, a simple ECMAScript search box highlighting tags starting with some text can make searching specific tags convenient [26].

Let the height and width (in pixels) of the k tags on some line be w_i, h_i for i ranging from 1 to k. The height h of the line is determined by the tallest tag in the line ($h = \max h_i$) whereas w, the width of the cloud, is fixed. For each line of the tag cloud, there might be extra horizontal white space $\omega = w - \sum w_i - (k-1)W$ where W is the normal width of a white space. Hence, there is at least $h \times \omega$ extra white-space area on a line. Because we fix w but not the maximal width of a tag, we must permit ω to be negative (but only when a very wide tag is alone on a line). Similarly, lines in text are typically separated by some white space (dictated by the `line-height` property in CSS), but it does not enter into our model. However, when a tag is shorter than the tallest tag on its line ($h_i < h$), this introduces some (extra) vertical space above the tag having area $(h - h_i)w_i$. Therefore, in our model (see Example 1), we define the badness of a line as $h \times |\omega| + \sum_i (h - h_i)w_i$ where the sum is over the tags on the line. Hence, the badness of a line is only a function of the set of tag dimensions (w_i, h_i).

This badness measure does not take into account symmetry or homogeneity. In fact, the exact placement of the tags on the line is not measured: tags can be left aligned, centered or justified. Lines can be permuted without changing the badness. The alignment of tags across lines, as a measure of orthogonality, is also not taken into account. Finally, for clouds with inline text, the order of text is presumed either fixed or unimportant.

EXAMPLE 1. *Suppose that the tags on a line have the following sizes in (width, height) format: (32,14), (45,16),*

(24,12) with a specified tag cloud width w of 128 pixels and an expected white-space width of 4 pixels between tags. The line height is $h = \max\{14, 16, 12\} = 16$. *There is extra (horizontal) white space on the line,* $128 - 2 \times 4 - 32 - 45 - 24 = 19$, *contributing to the badness by* $19 \times 16 = 304$. *The first and last tags have lesser heights than the second tag, and they contribute respectively* $32(16 - 14) = 64$ *and* $24(16 - 12) = 96$ *to the badness. The total line badness is thus* $304 + 64 + 96 = 464$. *As another example, if we have a single tag with dimension (130,16), then the (overfull) line has badness* $16(130 - 128) = 32$.

In the spirit of the Knuth-Plass total-fit algorithm [20], we might define the overall badness of a tag cloud as the sum of the squares of the badnesses of each line. Merely summing the line badnesses, without taking the squares, is also an option. Summing the squares of the badness has the benefit of penalizing more heavily solutions with some very bad lines, whereas a straight summation might tend to produce shorter clouds. Or we might minimize the maximum badness across all lines, but this might generate very tall clouds because if even a single line is forced into having a large badness, then all other lines can have the same badness without prejudice to the overall measure. Recall that the l_p norm of a vector v is defined as $\|v\|_p = \sqrt[p]{\sum_i |v_i|^p}$ when $1 \leq p < \infty$ and as $\max_i |v_i|$ when $p = \infty$. The three aggregates above can be described by the l_2, l_1, and l_∞ norms respectively.

4.2 Tag Clouds with Arbitrary Placement

Our model for this section assumes that

1. tags may be reordered and placed arbitrarily (but without overlap or rotation) in the plane;
2. tag relationships are known, and strongly related tags should be in close proximity;
3. tag-cloud width has an upper bound;
4. tag-cloud height should be small, to reduce scrolling;
5. (optional) tags may be deformed slightly (made shorter but wider, for instance), so long as tag area remains (nearly) constant;
6. (optional) large clumps of white space are bad.

In contrast to clouds with only inline text, there is no analogue to a "line" of tags when arbitrary placement is allowed. Therefore, when adapting the model from Sect. 4.1, it is not clear how to combine the various undesirable white areas that are in excess of a tag and its small surrounding border. A simple and appealing method is to sum this bad area, which is equivalent to minimizing the area occupied by the tag cloud.

Another (possibly conflicting) goal is to obtain spatial clustering of semantically related tags. If we form a graph with tags as vertices and edge weights indicating the strength with which two tags are linked, a reasonable measure of (undesirable) spatial non-proximity is

$$\sum_{p,q} w(p,q)d(p,q), \tag{1}$$

where p and q are placed tags linked with strength $w(p,q)$ and separated spatially by distance $d(p,q)$. Small values indicate better clustering. In experiments, we used Euclidean distance for computing $d(p,q)$.

4.3 Tag Relationships

The previous subsection assumed a graph-based model with a binary tag-similarity relation. Yet, higher-degree relations may also make sense, loading to a hypergraph-based model.

One method of determining tag relationships counts co-occurrences [14], when a pair of tags have been assigned to the same resource (e.g., a photo). Viewed this way, relationships are binary and can be modelled as edges in a graph. Another view is that each resource corresponds to a hyperedge in a hypergraph, whose members consist of the tags. Translation between these views can be achieved by replacing each hyperedge by a clique.

For example, consider a resource (perhaps a photo) tagged "baby, tears, bottle, diaper", another tagged "bottle, gas, beer", another tagged "beer, rioting, sports", and a fourth tagged "gas, tear gas, tears, rioting". (See Fig. 2(a).) The second view has 4 hyperedges, whereas the first view has $\binom{4}{2} + \binom{4}{2} + \binom{3}{2} + \binom{3}{2}$ edges. For instance, the hyperedge {bottle, gas, beer} from the first view would correspond to the edges { (bottle, gas), (bottle, beer), (gas, beer) } in the second view.

In EDA, the rôle of tags is played by modules and the natural relation between modules is "have a wire that interconnects several modules", leading to a hypergraph model. We argue in Sect. 5.2.5 that tag-cloud display should instead use a graph model.

5. SOLUTIONS

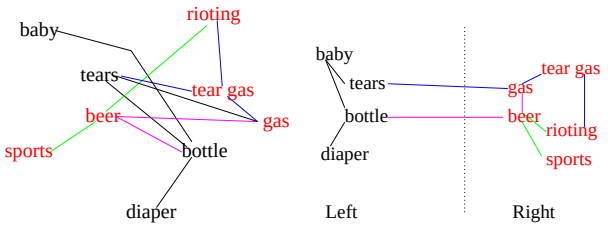
We propose different approaches to the two major problems. For inline text, dynamic programming or shelf-packing heuristics can be applied. For arbitrary placement, we use the min-cut placement algorithm from EDA.

5.1 Cloud Layout with Inline Text

Our first breed of algorithms take an ordered list of tags and choose where to break lines. We first designed a simple greedy algorithm: tags are added to the current line one by one, inserting a white space between them, until the line is full. It runs in $O(n)$ time and matches what is done by most browsers. When a tag is too wide to fit on even an empty line, a new line is created for this tag alone. Second, we implemented a dynamic-programming solution. Our algorithm is nearly identical to the $O(n^2)$ time and $O(n)$ space Knuth-Plass algorithm [20] given in Sect. 3.1, except that:

- the last line is not an exception: it cannot be half empty without penalty;
- if, and only if, a tag exceeds the maximal width, then it will be given a line of its own; no other overfull lines are allowed.

The second breed of algorithms reorders tags, attempting to decrease the badness. Finding an optimal ordering is NP-hard: when the required horizontal white space between tags is zero, we have the NP-hard Strip Packing Problem (SPP) [23]. As a rough heuristic to assess the influence of order, we randomly shuffle tags several times (10 in our experiments), apply the dynamic-programming algorithm to place the tags optimally, and keep only the best solution. Other simple heuristics are based on approximation algorithms for SPP, although SPP is only a special case of our problem. We use Next Fit Decreasing Height (NFDH)



(a) Bipartitioning: before (b) Bipartitioning: after

Figure 2: Bipartitioning (hypergraph view). This cut includes two hyperedges, or five edges.

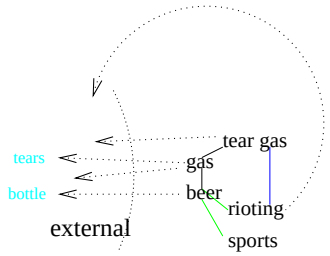
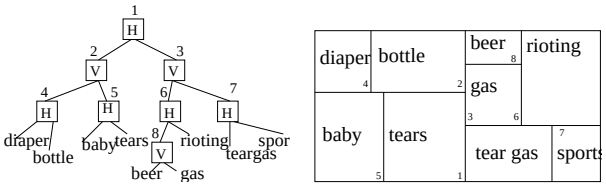


Figure 3: In future horizontal partitioning there will be a bias to encourage all tags (except `sports`) to the left side of their area.



(a) Slicing tree. Numbers next to nodes relate to areas in the slicing floorplan. (b) Slicing floorplan

Figure 4: Slicing tree and associated slicing floorplan. Cut-lines numbers indicate slicing-tree nodes.

and First Fit Decreasing Height (FFDH) from Coffman et al. [5]. They are SPP 2-optimal and SPP 17/10-optimal, respectively, and they run in $O(n \log n)$ time [23]. Both first sort tags by non-increasing height. NFDH is then the application of the simple greedy algorithm described above. FFDH places each new tag on the first available line, starting from the first line ever created, and creating a new line at the end whenever necessary. Tags exceeding the maximal width are placed on a line of their own. Because we typically have several tags with the same height, but different width, we further refined FFDH to our First Fit Decreasing Height, Weight heuristic (FFDHW). With FFDHW, tags continue to be primarily sorted by (non-increasing) height, but ties are broken by (non-increasing) width.

We could better assess the heuristics if we could obtain optimal solutions to this NP-hard reordering problem. However, for interesting clouds (e.g., $n = 100$), the ordering search space is huge: $n! = 100! \approx 9.33 \times 10^{158}$. We suspect branch and bound, or other sophisticated enumerative approaches, are too slow even for experimental work.

5.2 Cloud Layout with Arbitrary Placement

Fast arbitrary tag placement is achieved with min-cut placement, optionally followed by floorplan sizing.

5.2.1 Min-cut Placement

Min-cut placement [3] recursively decomposes a collection of tags by *bipartitioning*: splitting the tags into a "Left" group and a "Right" group. Then each group is recursively split, probably into "Top" and "Bottom" groups, although re-splitting into "Left" and "Right" may also occur. Ideally, the bipartition must be fairly balanced —the number of tags or total areas of tags, for instance, must be similar for the two groups. Also, the cut size (the number — or perhaps total weight — of edges/hyperedges containing tags in both groups) should be small. Since these two goals may conflict, various approaches can be considered. For instance, we specify a balance constraint and then try to minimize the cut. (See Fig. 2 for an example.) While bipartitioning is NP-hard, well-known heuristics exist.

During partitioning of a group of tags, there should be an influence of "outside" tags. Therefore, we track how strongly each tag is connected to external tags known to be above, below, leftward, and rightward of the group of tags being bipartitioned. See Fig. 3, where we see that even though tags `beer` and `sports` are connected, tag `beer` has an external leftward pull but `sports` does not. This may encourage partitions where these two tags are separated. Integrating external pulls into bipartitioning is often handled in min-cut placement by creating two dummy tags, "externalLeft" and "externalRight" and insisting that these dummy tags cannot change location [9]. Tag "externalLeft" is a surrogate for all external tags to the left of the nodes being partitioned. "ExternalTop" and "externalBottom" are similar.

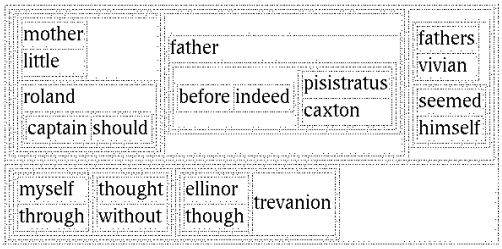
Under reasonable assumptions about tag sizes and bipartition balance requirements, given n tags with $m \in \Omega(n)$ relationships, min-cut placement can run in $O(m \log n)$ time if we use the Fiduccia-Mattheyses bipartitioning heuristic [12].

5.2.2 Slicing Floorplans

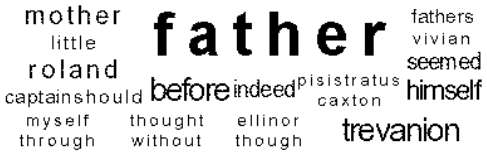
Recursive bipartitioning's effect can be represented in a *slicing tree* [34]. See, for instance, Fig. 4(a). Leaves store tags. Internal nodes specify the relative placements of tags in the subtrees, and they are labelled **H**orizontal or **V**ertical, depending how they divide tags. Each internal node is naturally associated with a placement area into which all tags in its subtree will be stored. The node also slices its placement area, either horizontally or vertically, assigning each of its subtrees to one of the sub-areas. At the finest level, each tag has been assigned a particular area into which it, and only it, may be placed. The resulting subdivision of the placement area is a *slicing floorplan* (see Fig. 4(b)) and can be used for placement: a straightforward tree traversal can assign precise locations to a "tightest possible" placement that corresponds to the slicing floorplan.

5.2.3 Nested Tables for Slicing Floorplans

Given a slicing tree, it is simple to make the browser render the tag cloud. (See Fig. 5.) We use a trick: each internal node in the slicing tree corresponds to a 2-element table in HTML. The table is either 2×1 or 1×2, depending whether the slicing-tree node is tagged 'H' or 'V'. If the node's children are not leaves, then the table's cells contain sub-tables. For example, node 6 in Fig. 4(a) leads to



(a) Displaying table borders



(b) Displayed with appropriate CSS

Figure 5: Slicing floorplan shown as nested tables.

```
<table><tr>
  <td> <table>
      <tr><td>beer</td></tr>
      <tr><td>gas</td></tr>
      </table></td>
  <td>rioting</td>
</tr></table>
```

CSS (e.g., `border-spacing:0px`) then reduces whitespace.

5.2.4 *Choosing Aspect Ratios*

Although the orientations chosen for the cuts in the slicing tree have perhaps the largest effect on the eventual shape of the layout, floorplanning can also choose *which* precise shape to use. In VLSI, there may be a tall skinny implementation of a ROM or a functionally equivalent square implementation. With tag clouds, we may be willing to stretch or squash a tag somewhat, as long as its total area remains more-or-less constant. This can be accomplished using CSS's `font-stretch` [35]. Unfortunately, few browsers act on this property yet; to simulate it, we adjusted `font-family`, as well as `letter-spacing`, `font-weight` and `font-size`.

This *floorplans sizing* can be done efficiently [31, 34] for slicing floorplans. In particular, it runs in $\Theta(s \log s)$ time if s represents the sum, taken over the number of shape options for each tag. Yet for general floorplans this problem is intractable [34].

5.2.5 *EDA Placement Is Not (Quite) Tag Placement*

Overall, the EDA problem of placement/floorplanning and our problem of tag-cloud layout are almost the same. Can we simply feed our tag-cloud data to an EDA placement tool and then extract a final placement? The answer is a qualified "yes": we have essentially done this, but we found it appropriate to modify the EDA tool.

Long tags can have aspect ratios that would be unusual for EDA. In placement or floorplanning, it is often permissible for cells to be rotated by 90 degrees. With rotation, every tall, thin module can become a short, wide module and thus there is a symmetry. With tags, a 90 degree rotation might be artistically interesting but hard to read. Thus we forbid such rotation. Rarely do we see a tag that is taller than it is wide; moreover, we have constrained the cloud width (but not height). Thus, we must handle widths and heights asymmetrically.

In some EDA design styles, the interconnect wiring must run between modules, rather than atop them. Thus, adequate white space must be allocated throughout the layout to accommodate wiring. Superficially, tag clouds are similar: we should not abut two tags without leaving some white space between them. However, in EDA the amount of white space at any particular area depends on the number of wires that must pass though that area. This is much more complicated than with tags, where a fixed boundary, or perhaps one proportional to the font size, is appropriate.

In both floorplans and tag clouds, strongly coupled items should be close to one another. For EDA, coupling comes from *nets*, which are best modelled as hyperedges in a hypergraph. Each net is a subset of modules, and electrical connectivity will eventually be achieved by finding a spanning (or Steiner) tree over the modules belonging to the net. The transitivity of electrical connections is a factor: consider the wiring necessary when a single net includes two tightly packed clusters of 10 modules each, found at opposite ends of the layout. A *single* wire can traverse the long distance; min-cut should count a cost of 1 for this net when bipartitioning. However, this behaviour is intuitively wrong when considering one cluster of 20 related tags. Each tag in the cluster is related to every other tag, and thus dividing them should be much more expensive: we are splitting a clique in an ordinary graph.

Finally, the expected input sizes and acceptable running times and solution quality levels may be different. Placement problems would frequently have thousands of elements, more than any reasonable tag cloud. Also, obtaining a high-quality solution would be more important than obtaining a solution quickly. On the other hand, any technique for on-demand tag-cloud creation for servers must necessarily be fast. "Fast floorplanners" (such as McFarland describes [24]) are used interactively with only a coarse subdivision of the design into top-level modules. These would more closely match the input sizes and response-time requirements of tag-cloud placement.

6. EXPERIMENTAL RESULTS

To evaluate our methods, we obtained test data from several source, implemented the methods, and analyzed the results they obtained.

6.1 Test Data

Tags and their accompanying importance levels (0-9) were obtained from ZoomClouds and Project Gutenberg; on average, clouds had 93 tags. For each of the 10 importance levels, we defined CSS classes with corresponding style choices: font sizes ranged from 8 pt to 44 pt and the selected font family was arial. However, our techniques need the size of each tag's bounding box, and we chose not to limit ourselves to monospace fonts. Experience showed insufficient accuracy from predictions based on knowing the text, font size, and various CSS parameters. Therefore, our programs are given tag bounding-box sizes as part of their input. These were obtained using ECMAScript and the DOM attributes `offsetWidth` and `offsetHeight` applied to an HTML `span` element.

Our requirement for browser-specific display information means that practical use of these techniques is perhaps best done on the client in ECMAScript, although server-side processing (using AJAX, for instance) is not impossible. Our experimental program for in-line text was written in Java, whereas the program for arbitrary placement was written in C; thus layout times may reflect a server environment.

6.1.1 ZoomClouds

ZoomClouds [1] is a Web site using the Yahoo! Content Analysis API to produce historical tag clouds for any given RSS feed using some content-processing heuristic. They make available a REST API producing an XML description of a tag cloud including tag names and weights. None of their tag clouds had more than 100 tags. We retrieved 65 different tag clouds with an average of 94 tags per cloud. For each tag cloud, we normalized the weights with a linear function so that they were integers between 0 and 9. We chose most tag clouds randomly with the random sources function of the Web site, but we also included major Web sites such as USA Today, Slashdot, the New York Times, L.A. Times, as well as major blogs such as Scobleizer and Boing Boing.

6.1.2 Project Gutenberg E-books

Test data, including tag relationships, were also derived from word co-occurrences in 20 e-books produced by Project Gutenberg [27]. Initial processing removed all nonalphabetic characters, converted all characters to lower case, and removed short words (those with 5 letters or less). The remaining words became tags. Only the most frequent k tags were kept (we used $k = 20$, 50, 100 and 200) in our tests. The importance i of tag T was determined as $i = \lfloor 10\frac{t-r}{f-r+1} \rfloor$, where f, r and t are respectively the frequencies of the most frequent tag, the least frequent retained tag, and the tag T.

Word co-occurrences determined the relationship strength between tags, as in recent work on tag-cloud display [14]. Two consecutive words form a *(distance 0) co-occurrence*. Each pair of tags had a relationship of strength s if there were $s \geq 2$ such co-occurrences in the e-book.

6.2 Tag Clouds with In-line Text

Our in-line text algorithms were implemented in Java 1.5. On a 2.16 GHz Intel Core 2 Duo processor, we ran 5000 tests on a large `del.icio.us` tag cloud [38] (140 tags, presented in Fig. 1): the average wall-clock running time for one tag cloud optimization was well under 1 ms for all algorithms (except the 10-random-shuffle heuristic), and under 0.2 ms for the greedy algorithms. Our code was not particularly optimized for speed.

We tested our algorithms on both the 65 ZoomClouds tag clouds and the 80 Project Gutenberg tag clouds. Fig. 6 presents a visual example of the result of 4 heuristics applied to one tag cloud. Alphabetically-sorted tags are, on average, 40% larger than weight-sorted tags. Dynamic programming does not reduce the area of the tag clouds for weight-sorted tags, but offers a reduction of about 3% for alphabetically-sorted tags. The random-shuffling algorithm does worse than sorting by weight[1]. The NFDH heuristic gives about the same average tag-cloud height as does the weight-sorted greedy algorithm, but the FFDH and FFDHW heuristics offer an average reduction of about 3% in the height of

[1]even trying 5000 shuffles (not shown).

(a) Alphabetically sorted tags, greedy algorithm

(b) Alphabetically sorted tags, dynamic programming

(c) Tags sorted by weight, greedy algorithm

(d) FFDH heuristic

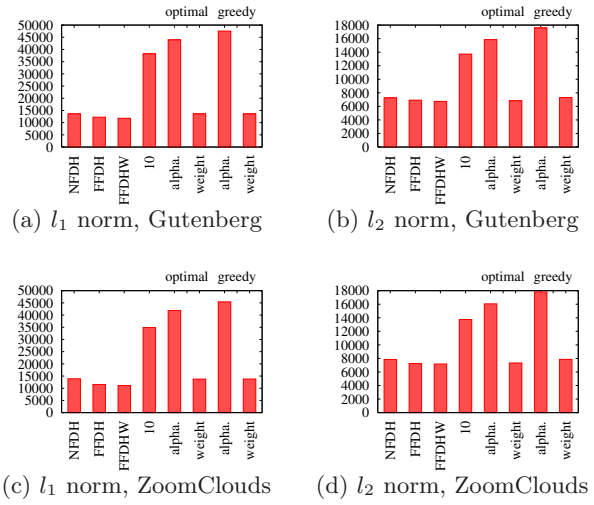
Figure 6: Screen shots of a tag cloud optimized using different algorithms: the greedy algorithm is similar to normal browser display.

the ZoomClouds tag clouds, and of 1% and 2% respectively for the Project Gutenberg tag clouds. Varying our badness model aggregates used by the dynamic-programming algorithms shows that using the maximum line-badness aggregate (l_∞) can generate unacceptably tall tag clouds (3 times taller than normal); however, the difference in height between the sum (l_1) and sum of squares (l_2) aggregates is well below 1%, though the l_1 aggregate has a small edge, as expected. While tighter clouds do not have large clumps of white space, they do not necessarily appear more symmetric.

The results we obtain with the badness measures are similar (see Fig. 7). The most competitive algorithms are FFDH, FFDHW and either the greedy or dynamic-programming algorithms applied to weight-sorted tags. The random-shuffles or alphabetically-sorted-tags algorithms are not competitive. When considering the l_1 norm of the line badnesses, the FFDH and FFDHW improve over the weight-sorted algorithms by 24% for the ZoomClouds data set, and by 11% and 15% respectively for the Project Gutenberg data set. When considering the l_2 norm, the main difference is that dynamic programming suddenly improves over the greedy algorithm (for weight-sorted tags) by 7% whereas FFDH and FFDHW only manage to improve over dynamic programming by 1% or 2%. In short, if the l_1 norm is chosen, the FFDHW heuristic is the clear winner and dynamic programming is not worth the effort, whereas if the sum of squares is preferred, it is a close race, with dynamic programming applied to weight-sorted tags a competitive solution.

6.3 Tag Clouds with Arbitrary Placement

We began with a simple EDA tool, a straightforward min-cut floorplanner previously implemented in C by one of the



(a) l_1 norm, Gutenberg (b) l_2 norm, Gutenberg

(c) l_1 norm, ZoomClouds (d) l_2 norm, ZoomClouds

Figure 7: Average aggregated badnesses (in pixels), the 10-random-shuffles heuristic has the label "10" while the optimal solution was computed by dynamic programming.

Figure 8: Large tag cloud generated from a Project Gutenberg e-text.

No. Tags	Iters	Finish	Size	C-hard	C-soft
20	3.0	38.5	1.0	24.5	339.0
50	2.2	43.1	0.9	38.0	865.5
100	1.1	39.2	1.1	39.0	1765.0
200	1.0	64.6	1.4	59.0	6429.5

Table 1: Times (ms) to make and size the slicing tree. Times for block-packer COMPASS **are also shown.**

authors. The floorplanner used techniques described by various authors [21, 22, 24, 40]. Various modifications addressed the concerns of Sect. 5.2.5.

We first modified our program to perform graph bipartitioning rather than hypergraph partitioning. With 12 or fewer tags, bipartitioning is done exhaustively. The two parts must be somewhat balanced in their total tag areas: the larger part's tag area may be at most twice the tag area of the smaller part. Subject to this constraint, the total weight of edges crossing the partition is minimized. With more than 12 tags, the system uses the Fiduccia-Mattheyses heuristic [12], taking the best result of 10 runs, each starting with a different random bipartition. With these larger problems, we require more balanced partitions. Again, balance is based on total tag area, but with the constraint that the size difference must be no larger than the area of the largest tag in the set being bipartitioned.

The original floorplanner assumed a routing model where routing area needs to be reserved in the areas around each cell [40]. Estimating the correct amount of "padding" area is not required for tag placement. We replaced this complex code by an estimate that a 2-pixel horizontal space was required on the left sides of a tag, except where the tag was on the left edge of the layout area.

The original floorplanner also preferred square layouts. However, for tag clouds, we have a fixed width bound that should not be exceeded. This bias affected the cut orientations chosen for the slicing tree, since during placement simple heuristics monitor the estimated aspect ratios of each floorplan area. Originally, areas were divided by vertical cuts when they were wider than they were high. This is a relative decision and does not enforce an absolute width bound. Hence, we added an estimate of the absolute width of a floorplan area. Comparing this estimate against the widths of the tags for that area, we may determine that, despite a possibly non-square floorplan area, a vertical cut cannot be permitted. Now, for large clouds, near the roots of our slicing trees there continues to be frequent switching between horizontal and vertical cuts. However, near the leaves, horizontal cuts predominate.

Fig. 5 shows an example, a 20-tag cloud that we obtain from Bulwer-Lytton's *The Caxtons* [27, etext 7605]. The word 'father' has been chosen in a shorter/wider variation, whereas the word 'before' was chosen to have a taller but narrower shape than the default. Although it may not be possible to read the smaller tags in the 200-tag cloud (Fig. 8) from Rodenbough's 1885 text on the Anglo-Russian dispute [27, etext 7320], the cloud shows the effect of tag resizing ('afghanistan' being vertically compressed and 'kandahar' being horizontally compressed). The organization of the smaller tags into (local) columns also confirms that horizontal cut lines predominate near the slicing-tree leaves.

6.3.1 Results

Results are shown in Tables 1–3. In the first table, the 'Finish' column gives the over-all time, in milliseconds on a 1.7 GHz, Pentium 4-based machine. Interestingly, the results were obtained faster for 100-tag clouds than for 50-tag clouds: a tuning parameter was used in the modifications made so that the floorplanner would not exceed a 550 pixel width. If set incorrectly, the placement can be much narrower, or perhaps wider, than 550 pixels. If this is detected, the program adjusts the parameter and tries again. For the smallest tags, an average of 3 attempts per cloud were made

No. Tags	Greedy Min-cut	(sorted)	(random)	compaSS
20	31	37	46	**29**
50	63	62	85	**59**
100	111	99	139	**98**
200	192	**165**	231	170

Table 2: Average area (kilopixels) for the bounding boxes of tag clouds.

No. Tags	Greedy Min-cut	(sorted)	(random)	compaSS
20	**61**	124	120	65
50	**166**	282	271	180
100	**296**	465	482	382
200	**438**	693	765	654

Table 3: Average total weighted distance ($\times 10^3$) using Equation 1. Distances were computed between the lower-left corners of tags.

(column 'Iters'). From the 'Size' column, we see that floorplan sizing was only a small part of the over-all time. For comparison, we ran the block-packing program compaSS [4] against our data. It does not consider tag proximity, but simply seeks a tight layout. The 'C-hard' column uses only the normal sizes of tags, whereas the 'C-soft' column shows that unacceptably long runtimes are required for the resizing variant, where tag areas are fixed but each tag's aspect ratio is continuously variable over a range. (compaSS was fast when when given a set of 3 aspect variations per tag. Unfortunately, it assumes exactly identical area for each variation. However, with indirect control over how the browser renders the tag variations, the three areas are not quite identical. Thus, we cannot compare solution qualities fairly.)

The solution area was examined in Table 2, and it was compared against two row-based tag layouts: first, when the tags were given in descending order (by height); second, when the tags were randomly ordered. The last column shows the solution obtained by compaSS.

The sorted greedy heuristic used 2–19% less area than our min-cut heuristic (although the random greedy heuristic used 20–48% more area than ours). Compared with compaSS, min-cut used 7–13% more area. These results are not surprising: compaSS and the sorted greedy heuristic seek only a tight packing, whereas min-cut also seeks to group together strongly related tags. With 200 tags, it is remarkable that the more sophisticated compaSS approach was not as good as the greedy heuristic. This might be attributed to special characteristics of our data, such as the non-squareness of most tags. For the greedy approaches and compaSS, only the default shape of each tag was used. When tags were instead available with continuously variable aspect ratios (over the range used by the 3 choices available to our min-cut program), compaSS was able to reduce area by approximately 12%, compared with its performance when only the default shape was allowed. (However, Table 1 shows that this required more than 6 s on large clouds.)

The min-cut approach clearly (and unsurprisingly) outperformed greedy approaches and compaSS when we tested proximity for semantically related tags (see Table 3). It considers this factor, unlike the others.

Although compaSS and the sorted greedy approach both packed tightly, and although both are oblivious to tag relationships, compaSS is apparently better at grouping than the sorted greedy heuristic. This is counterintuitive and reveals a weakness in using Equation 1 to assess grouping: a tight, square packing will score better than a loose or rectangular packing. It appears that compaSS often uses far less than its maximum 550 pixel width. By nature, the greedy approach leads to widths of almost 550 pixels; hence, its small clouds have large aspect ratios. On small clouds, our min-cut floorplanner seeks a square layout, assuming that each tag is itself approximately square. The effect is that small min-cut clouds tend to have aspect ratios similar to a typical tag: in other words, their aspect ratios often lie between compaSS-produced clouds and clouds produced by the greedy heuristic. With more tags, the width bound begins to affect all heuristics similarly, so the effects due to aspect-ratio differences are reduced.

7. CONCLUSIONS

Future work should include browser-based implementations. For in-line text, our cloud-badness model is probably incomplete since it ignores some basic symmetry issues: some lines may only have a few short tags, whereas taller lines are densely packed. It is also incomplete because it does not take into account tag similarities, but it is not necessarily easy to take existing tag clouds and infer an interesting similarity measure between tags. Tag-cloud coloring is also open to optimization.

Despite the differences between tag-cloud layout and EDA placement, we plan to test an industrial strength min-cut placement tool such as Capo [29], to see how well it places tags. However, a better metric is needed for assessing clustering of related tags, and optimizing according to some new metric would likely require substantial changes to an existing EDA tool.

HTML and its presentation counterpart, CSS, will probably never directly account for representations such as tag clouds. However, CSS3 [11] may introduce some new instructions which may alleviate some problems. For example, while it is possible to justify an inline tag cloud with the `text-align` property, the last line is typically not justified, a limitation addressed by the upcoming `text-align-last` property. Also, the new `hyphenate` property might encourage the use of slightly more sophisticated line-breaking algorithms in browsers.

8. ACKNOWLEDGMENTS

The first author was supported in part by NSERC grant 155967, and the second author was supported in part by NSERC grant 261437 and FQRNT grant 112381.

9. REFERENCES

[1] AR Networks. ZoomClouds, 2007. [Online; accessed 22-01-2007].

[2] K. Bielenberg. Groups in social software: Utilizing tagging to integrate individual contexts for social navigation. Master's thesis, Universität Bremen, 2005.

[3] M. A. Bruer. Min-cut placement. *Journal of Design Automation and Fault-Tolerant Computing*, 1(4):343–362, 1977.

[4] H. Chan and I. L. Markov. Practical slicing and non-slicing block packing without simulated annealing. In *Proceedings, Great Lakes Symposium on VLSI*, pages 282–287, 2004. see also `http://vlsicad.eecs.umich.edu/BK/BLoBB/`.

[5] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Comput.*, 9(4):808–826, 1980.

[6] D. Coupland. *Microserfs*. Flamingo, 1996.

[7] G. Di Battista et al. *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall, 1999.

[8] M. Dubinko, R. Kumar, J. Magnani, J. Novak, P. Raghavan, and A. Tomkins. Visualizing tags over time. In *15th International World Wide Web Conference*, pages 193–202. ACM Press New York, NY, USA, 2006.

[9] A. E. Dunlop and B. W. Kernigan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4:92–98, Jan. 1985.

[10] P. Eades. A heuristic for graph drawing. In *Congressus Numeratium*, pages 149–160, 1984.

[11] E. J. Etemad (Editor). CSS3 text effects module. `http://www.w3.org/TR/css3-text/`, last checked on 24/01/2007, 2005. W3C Working Draft 27 June 2005.

[12] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th Design Automation Conference (DAC-82)*, 1982.

[13] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software — Practice and Experience*, 21(11):1129–1164, 1991.

[14] Y. Hassan-Montero and V. Herrero-Solana. Improving tag-clouds as visual information retrieval interfaces. In *International Conference on Multidisciplinary Information Sciences and Technologies (InSciT2006)*, Mérida, Spain, Oct. 2006.

[15] N. Hurst, K. Marriott, and D. Albrecht. Solving the simple continuous table layout problem. In *Proceedings, DocEng '06*, pages 28–30, 2006.

[16] A. Jaffe, M. Naaman, T. Tassa, and M. Davis. Generating summaries and visualization for large collections of geo-referenced photographs. In *MIR '06*, pages 89–98, 2006.

[17] A. Kennings and K. P. Vorwerk. Force-directed methods for generic placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2076–2087, 2006.

[18] S. Kirkpatrick, C. D. Gelatt, Jr, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[19] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[20] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software — Practice & Experience*, 11(11):1119–1184, 1982.

[21] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley and Sons, New York, 1990.

[22] T. Lengauer and R. Muller. A robust framework for hierarchical floorplanning with integrated global wiring. In *Proc. International Conference on Computer-Aided Design (ICCAD-90)*, pages 148–151, 1990.

[23] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[24] M. C. McFarland, SJ. A fast floor planning algorithm for architectural evaluation. In *Proc. International Conference on Computer Design (ICCD-89)*, pages 96–99, 1989.

[25] D. Millen, J. Feinberg, and B. Kerr. Social bookmarking in the enterprise. *Queue*, 3(9):28–35, 2005.

[26] D. R. Millen, J. Feinberg, and B. Kerr. Dogear: Social bookmarking in the enterprise. In *CHI '06*, pages 111–120, 2006.

[27] Project Gutenberg Literary Archive Foundation. Project Gutenberg. `http://www.gutenberg.org/`, 2006. checked 2006-10-17.

[28] H. Purchase. Effective information visualisation: a study of graph drawing aesthetics and algorithms. *Interacting with Computers*, 13(2):147–162, 2000.

[29] J. A. Roy, S. N. Adya, D. A. Papa, and I. L. Markov. Min-cut floorplacement. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(7):1313–1326, 2006.

[30] T. Russell. cloudalicious: folksonomy over time. In *JCDL'06*, pages 364–364, 2006.

[31] W. Shi. An optimal algorithm for area minimization of slicing floorplans. In *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 480–484, 1995.

[32] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.

[33] M. Sneep. A short comparison of various typesetting engines. `http://www.nat.vu.nl/~sneep/ars/type/comparison.pdf`, last checked on 24/01/2007, 2005.

[34] L. Stockmeyer. Optimal orientation of cells in slicing floorplan designs. *Information and Control*, 57(2–3), 1983.

[35] M. Suignard and C. Lilley (editors). CSS3 module: Fonts. `http://www.w3.org/TR/css3-fonts/`, last checked on 28/01/2007, 2005. W3C Working Draft 2 August 2002.

[36] Technorati Inc. Technorati, 2007. [Online; accessed 22-01-2007].

[37] Wikipedia. Tag cloud — Wikipedia, the free encyclopedia, 2004. [Online; accessed 4-January-2007].

[38] Yahoo! Inc. del.icio.us, 2007. [Online; accessed 22-01-2007].

[39] Yahoo! Inc. Flickr, 2007. [Online; accessed 22-01-2007].

[40] G. Zimmerman. A new area and shape function estimation technique for VLSI layouts. In *Proc. 25th Design Automation Conference (DAC-88)*, pages 60–65, 1988.