# The Self-Organizing Desktop:

An Unsupervised Content Classification System for Efficient Document Browsing

Senior Project submitted to the
Division of Science, Mathematics and Computing
of Bard College

by

Brendan Berg

Annandale-on-Hudson, New York
May 2006

# Contents

# Abstract

Document management systems like the *Windows Explorer* or OS X's *Finder* are based on real-world metaphors of files and folders. While basing a system's conceptual model on a well-understood physical analogue creates a gentle learning curve, it limits the possibilities of abstract representation. Additionally, in order to browse documents efficiently in current file managers, users are required to create their own hierarchical relationships. Manual organization is time consuming and increasingly unrealistic now that hard drives have hundred- or thousand-gigabyte capacities.

Developing a new interface for browsing a file collection requires attention to issues of categorization and representation. In this project, I focus on automatic categorization, where users aren't required to organize their documents. The categorization system I developed uses Apple's Search Kit indexing API to generate statistical data on text documents and uses a Kohonen Self-Organizing Map to visually cluster related documents on the screen. The self-organizing representation of the documents is designed with careful attention to aesthetics as well as functionality in order to be intuitive, simple, and powerful.

# 1 Introduction

The shortcomings of manual document categorization in current systems motivate this project. Present systems require the user to define the physical structure of their files and the semantic relations among those files. Manual organization becomes an increasingly time-consuming task as the number of files that a person deals with increases. Additionally, representations based on analogies derived from the real world can be too restrictive and prevent possible categorization methods.

This paper presents a discussion of current systems and hypothetical improvements, followed by implementation details for a prototype automatic document categorization system and a discussion of that system.

## 1.1 Categorization of Documents in a File System

Most current file systems are based on a hierarchical structure of folders containing sub-folders and files. This presents a simple conceptual model because people are already familiar with the concepts of files and folders in their offices. Unfortunately, this model restricts users to a system where all documents in one level of the hierarchy are equal and where documents must belong to only one directory. Commonly used workarounds such as color labels to differentiate files in a particular folder, and aliases, links, and shortcuts that show a document's additional categories are evidence of these shortcomings.

Linking the file system's modes of interaction to a well-understood idea from the real world is a brilliant decision which dramatically reduces the time needed to learn the new system. People can immediately form a conceptual model of the document structure, and that model will closely match the structure's implementation on a disk. Expert users, however, are often frustrated with the limitations of electronic documents behaving as rigidly as physical ones (Gelernter, 1998). This notion, called paradigm drag, is noticeable when the

electronic version of a well-understood physical analogue is constrained by the limitations of the physical version. Creators of conceptual structures that are closely based on familiar real-world systems should be aware of limitations they may be imposing on what could be a more versatile model.

Hierarchies turn out to be ideal for tasks such as archiving (Barreau & Nardi, 1995), but ultimately less useful for day-to-day use. A more common task is file categorization, where a user wants to group documents according to the similarity of their content, according to their common use for a particular task, or according to a common subject. A major drawback appears when a document cannot easily belong to multiple directories without one of the previously mentioned workarounds. If the same document belongs to two conceptual categories, it cannot be referenced easily in each. One solution to this problem is to tag documents with metadata, which can have a many-to-many relationship with the associated documents. Metadata, or information about a document, such as its author, its publication date and its keywords, is a useful addition to a hierarchical system. Indeed, having discrete terms with semantic value attached to a document allows for powerful search capabilities and more flexible grouping.

The Be Operating System, or BeOS, included one of the first file systems that had flexible support for document metadata. The file system allows a user or an application to specify any number of arbitrary attributes for a file (Giampaolo, 1999). The attribute data can be of string, integer, or floating-point type, or it can be raw binary data of arbitrary size. In a traditional file system, organizing documents by author is extremely difficult. The user would need to create a folder for each author and place documents in the appropriate folder. If a document has more than one author, it cannot easily be located in a folder for every author. The flexibility of a file system with metadata support proves useful when a user wants to perform this sort of categorization. If a user wants to categorize text

documents by the author's name, he or she can add an "Author" attribute containing the author's name. If the document has more than one author, an attribute can be added for each author. The user can query the file system for a particular author and see a collection of documents written by that author. A document with multiple authors will show up in queries for each of the authors.

Adding metadata for semantic associations may overcome issues of multiple categorizations (a document may have more than one author), but the problem of manual categorization remains. This is a significant shortcoming, and relying on user-defined metadata is problematic since most users are unlikely to spend the time to categorize their files (Marsden & Cairns, 2003). Another issue with metadata is that there is often too little information about a file to make meaningful judgments about document similarity. There may be enough information to group documents by their authors, but metadata alone is not sufficient for more precise clustering based on semantic content of the documents. The actual content of a text document, however, provides the necessary information, and content indexing software extracts and manages this data efficiently. Content indexing turns out to be the most straightforward way to gather semantic information about a set of documents.

Automatic categorization is a promising technique in document management systems. Its application to browsing interfaces has already been explored by a number of researchers. Harada, et al. developed a browser for a photo collection that does not require manual organization (2004). Their application clusters photos based on the date and time taken and presents a time line view for browsing the collection. Retrieval performance in user tests of automatically organized photo collections was similar to manually categorized collections in most tests and outperformed manual categorization in terms of retrieval time in several areas. Their findings support the conclusion that automatic categorization

with an appropriate interface is at least as efficient in retrieval tasks as traditional manual organization.

Similar results have been found for text categorization, specifically by Weippl (2001). The Information Landscape software he describes presents automatically clustered documents in a two-dimensional space. Browsing is not discussed, but the information landscape can be searched, highlighting relevant documents.

## 1.2 Browsing Files in a File System

Most work in the field of information retrieval is concerned with searching for a target document in a set of documents, rather than browsing. These are very different problems. In order to search, a user must have some word he or she is interested in, and the word must be both specific enough to return a manageable result set, yet general enough that it doesn't exclude anything of interest. It is difficult, however to give an adequate textual description of something one lacks precise knowledge about (Hertzum & Frøkjær, 1996). Browsing takes advantage of a person's spatial memory, in order to remember approximately where something is, and visual memory, in order to remember some identifying physical characteristic. For example, if someone is looking for a book by browsing a bookshelf, he or she may remember that the book was on a high shelf and is near other books on a similar topic. Looking at the group of books that are likely matches, visual cues like the color of the cover or the typeface used for the title prompt his or her memory. In this example, one need not remember the title or author, presenting a clear advantage over text-based search. Searching and browsing each have distinct advantages in certain cases, and interfaces that present options to switch between both may be advantageous.

Browsing is difficult to define because it encompasses so many different modes of

interaction. An enumeration of possible scenarios hints at what representations to display and interactions to allow. The following browse scenarios describe a user's goals for different types of browsing.

1. *Search Refinement* — The user is looking for a specific document. A textual search returns a large selection of items. Thumbnails, or miniature visual representations of each document's first page, and clustering of similar items (by category, file type, keyword, etc.) help the user pick the intended document.
2. *Prototype Expansion* — The user has an exemplar document and wants to see similar items. A list of relations is presented (similar content, similar use pattern, similar category), and files that are related are presented nearby.
3. *Unstructured Exploration* — The user isn't looking for anything in particular. He or she might find something interesting, find things that are related, and move on to something else. This could be similar to browsing the Web, so a record of recently viewed documents and a collection of favorite documents may be helpful.
4. *Category Exploration* — The user knows that a particular document belongs to a category of documents, and seeing all documents in that category together allows the user to ignore documents that don't belong to that category.

The browsing process allows users to navigate between documents by exploring relationships between them. The associative nature of browsing tasks takes advantage of a similar associative nature of human memory. A visual interface for browsing takes advantage of the brain's greater ability to recognize an item than to describe it, and the brain's ability to perceive approximate details at a glance (Hertzum & Frøkjær, 1996). Browsing turns out to be an important mode of interaction in current file managers, and should not be ignored as new tools such as increased use of metadata and content indexing improve searching capabilities.

## 1.3 Visual Representations for Browsing

Document management is primarily a representation problem. All operating systems provide an application, called a file manager, that provides a user interface for interacting with the file system. File managers in current operating systems use a visual interface that presents files and folders spatially, with file and folder objects that behave like their real-world counterparts. The file manager is the most frequently used software on a computer and should provide readily understandable displays and use simple, intuitive interaction.

The files on a disk are an abstraction of the physical arrangement of data on the hard drive. This abstraction is easily related to paper storage in office filing cabinets. File systems based on a hierarchical data structure are easy to visualize in many different forms, such as tree maps, outline views and the familiar desktop metaphor. This metaphor is restrictive, however, and many interfaces for file browsing do not take full advantage of possible representations for file relations.

Because of the complex and abstract nature of relationships between files, it is difficult to represent non-heirarchical file structures. For example, a graph-based approach to displaying document similarity might draw lines between individual documents. Because of the large number of relationships, this diagram would quickly become a tangle of lines, with little informative value.

One way to represent complex information is through abstraction (Mukherjea, 1999). Displaying multi-dimensional data on the two-dimensional computer screen grows "more difficult as ties of data to our familiar three-[dimensional physical] world weaken (with more abstract measures) and as the number of dimensions increases (with more complex data)" (Tufte, 1990). Although representation of complex data without the use of metaphors for familiar objects is difficult, it has the benefit of reduced paradigm drag.

An abstract interface may lack the immediate recognition gained from a familiar mode of interaction, but it is not tied to concrete aspects of the real world. A self-organizing map is an interesting abstraction because it preserves readily-understood concepts of distance and similarity, while remaining free of real-world limitations. Additionally, it is extremely easy to visualize.

One of the main challenges of representation is the multi-dimensional nature of the data. Modern graphics cards and processors can handle the computation required for complex 3D graphics, but a three-dimensional view of the data may be problematic. The two-dimensionality of the computer screen prevents users from looking around or seeing inside 3D objects (Fry, 2004). Three-dimensional representations are also unfamiliar to a user who is used to 2D windows and icons on a 2D screen. For these reasons, a two-dimensional representation was chosen for this project. Additional work would be required to create a 3D representation with intuitive navigation and understandable visual structure, but with the proper considerations, users may benefit from such a representation.

Any visualization of browsing should show the relationships between files and allow context switches between multiple layers of detail. The details of the visualization chosen for this project are discussed later, and present just one method of viewing the data. With knowledge from the implementation process, it will be possible to return to the visualization portion of the project to refine to the interface.

## 1.4 Assembling the Pieces

Based on the above discussion of the drawbacks of manual file categorization and the difficulties of visualizing abstract relationships between files, this project intends to determine the feasibility of an interface for browsing automatically categorized files. Such a system should adapt to each user's collection of documents — each user will have documents that represent their personal interests, so there is no universal way of categorizing a space that can contain any conceivable document. Furthermore adaptability is important because a system that requires major adjustments between different document sets is hardly better than a system that requires manual categorization. Additionally, it should learn the semantic space of the documents and clearly present the organized files and allow browsing related documents.

# 2 Implementation

To test ideas about unsupervised file categorization, I developed an OS X application to generate content-based indices and visualize automatically laid-out representations of document similarity. The software package, called AutoGraph, uses Apple's Search Kit framework, part of Mac OS X, to index the contents of each document. Because of the computation time required for training the self-organizing map, a smaller set of terms that are most likely to be good document discriminators are automatically chosen. Given a set of vectors for each document, where each vector is made up of the normalized number of occurrences of each of the best discriminators, a Kohonen self-organizing map clusters related documents into a two-dimensional grid which is displayed on the screen. The visual representation of the Kohonen map is the basis for a file browsing interface, which provides an overview of the organized document space as a whole with clusters of documents labeled with their dominant term. A simple zooming interface provides enlargements of specific clusters of documents.

## 2.1 Content Indexing

The Search Kit framework is designed to provide efficient document content searching to OS X applications (Apple Computer, 2005a). While not used for search functionality in this case, it still provides indexing capabilities that are essential for document categorization. Search Kit constructs an index of the salient information in a set of documents where each entry in the index refers to one or more documents. The index is structured as a list of terms, or words, followed by a reference to each document in which the term occurs and the position of the term in that document. Figure 2.1.1 shows three example documents and the index created from those documents. In this example, the term "design" appears in Document 1 as the 2nd word, in Document 2 as the 1st and 8th words, and in Document 3 as the 3rd word. However, not all words are meaningful in an index. Common English function words such as articles and prepositions occur frequently in all documents and therefore

simply take up space in the index without adding any value. Indexing software typically maintains a list of these words, called stopwords, that are ignored by the index. When adding a document to an index, Search Kit automatically handles reading the contents of the file, parsing the file to remove formatting, removing stopwords, and adding terms to the index. Objective-C code for creating and populating an index is included in Appendix A.

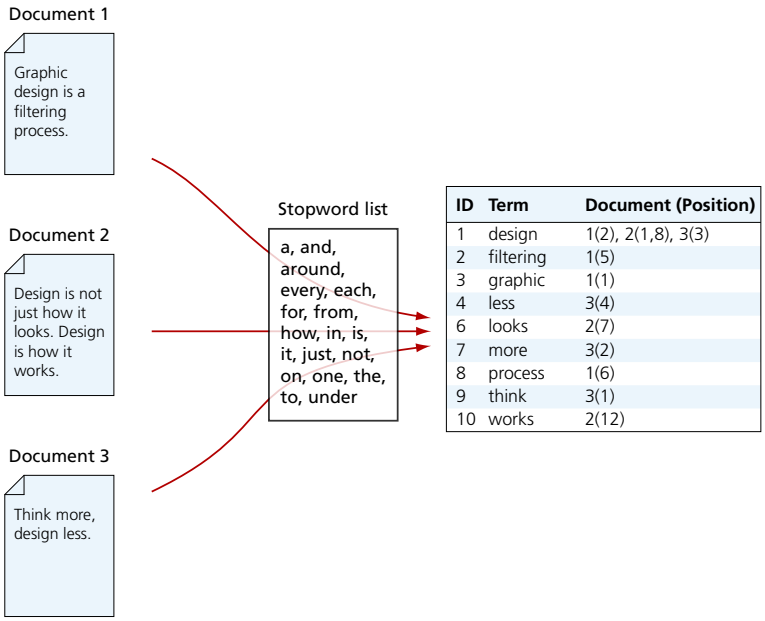



*Figure 2.1.1* A document index, showing three input documents and the stopword list.

The AutoGraph software includes an interface for managing these indexes, allowing index creation and opening and document addition and removal. Search Kit provides functions for term analysis such as the frequency of a particular term in a document, the number of documents in which a term occurs, and the number of terms in a document, which I use to select the terms that are the best document discriminators.

## 2.2 Term Selection

Although self-organizing maps are designed to reduce a high-demensional data set to a manageable number of dimensions, the number of terms in an index is so large that

using all of them in the document vector would be far too computationally intensive. A reasonably small set of documents could have a very large number of terms in the index; the test data set of just over 50 documents had more than 20,000 terms in the index. Salton, Wong and Yang discuss methods to choose only the terms most significant to a set of documents by observing changes in the density of the document space (Salton, et al., 1975). The document space can be represented by a set of vectors, where each vector represents one document, and where the $i^{th}$ element of that vector is the frequency of term *i*.

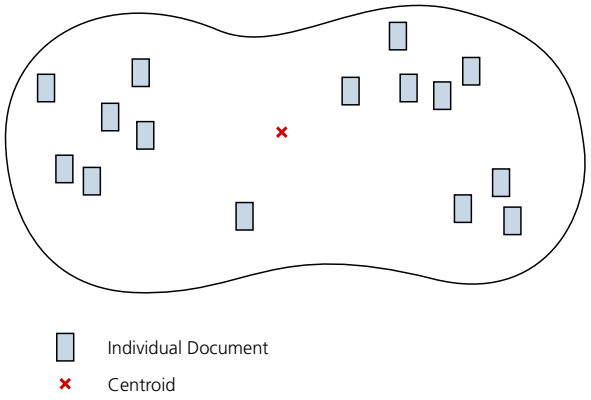



***Figure 2.2.1*** A document space showing clustered documents and the document space's centroid

The document space in Figure 2.2.1 shows various documents contained within the space and the centroid, which is defined as a vector where each element is the average of the corresponding elements of *m* document vectors. That is,

$$(2.2.1)\quad c_j = \frac{1}{m} \sum_{i=1}^{m} d_{ij},$$

where document vector $\mathbf{D}_i = \langle d_{i1}, d_{i2}, \ldots, d_{ij} \rangle$, and $\mathbf{D}_i \in K$, the set of all documents.

Document space density can be measured by computing the sum of the similarities between the main centroid and a document $\mathbf{D}_i$, for all documents in *K*, as in equation 2.2.2:

(2.2.2) $Q = \sum_{i=1}^{n} s(\mathbf{C}, \mathbf{D}_i)$

where *s* is a similarity function, typically the Euclidean distance. When Q is small, the document space is compact, and it is difficult to differentiate between individual documents. Conversely, when Q is large, the document space is spread out, and individual documents can be easily distinguished from the others.

One method of determining the quality of a term in differentiating documents from each other is the term discrimination model. The discrimination value, *DV*, of a term "measures the extent to which a given term is able to increase the differences among document vectors" (Salton, et al., 1975). A term with a high discrimination value is considered a "good" term, and decreases similarity between documents, thus increasing the document space density. The opposite is true for a term with a low discrimination value.

The discrimination value of a particular term is simply the difference in space density before and after assignment of the term. A particular term's impact on the document space density, $DV_k$, can be computed with the following equation,

(2.2.3) $DV_k = Q_k - Q$

where $Q_k$ is the size of the document space with term *k* removed from all document vectors. If the document space is more compact after the removal of term *k* (that is, $Q_k$ is less than Q), then term *k* is a good discriminator. Figure 2.2.2 shows a document space before and after the removal of a good document discriminator.

Using the term discrimination value for all terms in a document set, it is possible to observe

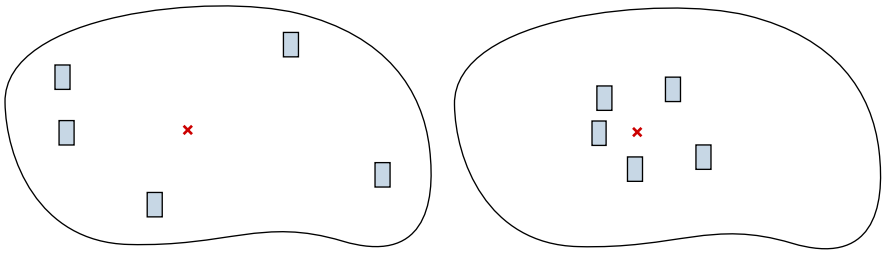


*Figure 2.2.2* A document space showing document density before (left) and after (right) the removal of a term that is a good document discriminator.

trends in the discrimination rank compared to the number of documents in which the term occurs (document frequency). Salton, et al. found that the best document discriminators were those terms with uneven frequency distribution. The addition of these terms had the effect of spreading out the document space by producing irregular changes in document vectors. Examination of the properties of good and bad discriminators in a sample collection of documents shows a trend in the discrimination rank based on document frequency alone. It was found that "the best discriminators are the 25 percent whose document frequency lies approximately between n / 100 and n / 10 for n documents" (Salton et al., 1975). Terms that appear in only one or two documents are relatively poor document discriminators, as they affect the space density only slightly. Terms with a large document frequency are even worse discriminators because they are so common.

Because computing the discrimination value for each term is a time-intensive process, rather than looking at the quality of each term as a discriminator, a certain number of document discriminators are selected from a range of document frequencies between two cut-off points. For the reasons stated above, it is assumed that a certain number of these will be good discriminators. AutoGraph automatically selects all middle-frequency terms between *n / 3* and *n / 5*, which are likely to be good discriminators. The cut-off values were chosen manually, and adjustment of the values may improve clustering performance in the next stage.

As it turns out, if there are poor discriminators in the set of selected terms, they will have little impact on the accuracy self-organizing map. Poor discriminators will naturally affect the size of the map, and therefore its speed, so an effort to remove as many of them as possible is helpful in order to reduce computation time.

After selecting terms that are good discriminators, the software generates for each document a vector containing each of the selected term's frequency in that document. The set of vectors for all of the documents is then used to train a self-organizing map.

## 2.3 Self-Organizing Feature Maps

Self-organizing maps are neural networks for automatic unsupervised learning. They are useful for their ability to cluster high-dimensional data based on similarity. Structurally, a self-organizing map consists of a network of nodes that are fully connected to a vector of arbitrary dimension from which they receive input. Vectors are $n$-tuples of real numbers in the vector space $\mathbb{R}^n$. The map of nodes is typically two dimensional, and the location of the node has semantic value because relative distances between input vectors are preserved as distances on the map of nodes. Each node has a collection of real-valued weights that connect to every element of the input vector. Therefore, if the map has $m$ nodes, and the input vector has $n$ elements, there will be $m \times n$ weights. Figure 2.3.1 shows the topology of a 5 × 5 map of output nodes receiving input from a three-dimensional input vector.

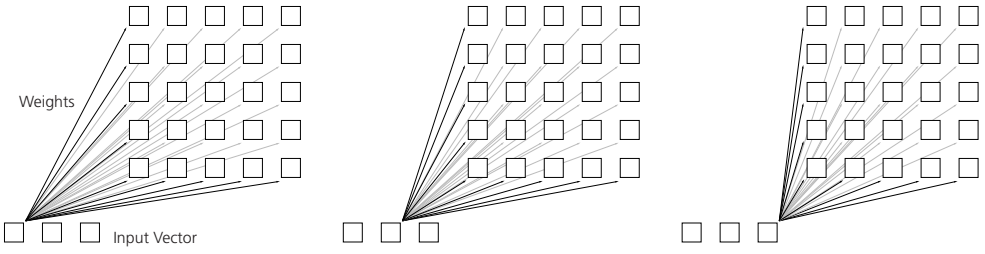



*Figure 2.3.1* Each element of the input vector is connected to each element of the map of nodes by a collection of weights that project from the input vector to the map. (The three dimensions of the input vector have been separated to avoid overlapping lines in the diagram.)

The weight vector for a node is essentially a reference marker that is compared with the currently presented input. When the algorithm searches for the best matching node, it computes the Euclidean distance between the input vector and the weight vector for each node, and finds the node whose distance is smallest.

For training, the elements of the set of input vectors are presented to the self-organizing map serially, and a particular input will activate one best matching output node on the map. The weight vector of the best matching node is adjusted to reduce the distance between it and the input vector. Additionally, the weight vectors of nodes within a certain radius are adjusted proportionally to their distance from the best matching node. The amount of adjustment and the radius of the neighborhood both decrease with each learning iteration. This process is repeated, presenting each input vector multiple times, until changes in the map are no longer discernible. (Haykin, 1994)

The learning algorithm described in Haykin (1994) is detailed below:

1. ***Initialization***. Assign random values to each element of weight vector $\mathbf{w}_k$ , $0 \leq k < m$, for each of $m$ output nodes in the map. The initial values should be in the same range as the values of the input vector elements.

2. ***Sampling***. Select an element from the input set. This can be chosen at random, or, if the input set is unordered, samples can be chosen sequentially.

3. ***Similarity Matching***. Find the best matching node for the selected input vector. This is the node with the smallest Euclidean distance from the input vector. The best matching node, $i(\mathbf{x})$, for input vector $\mathbf{x}$, is defined as:

$$(2.3.1) \quad i(\mathbf{x}) = \underset{0 \leq i < m}{\operatorname{argmin}}(\|\mathbf{w}_i - \mathbf{x}\|)$$

4. *Updating*. Adjust the weights within a certain radius of the best matching node. The amount that a node will change depends on its distance from the best matching node, where the influence decreases with distance, and the learning rate, which decreases with learning iterations. The adjustment formula for node $w_m$ is:

$$(2.3.2)\quad w_m(t+1) = \begin{cases} \mathbf{w}_m(t) + \alpha(t)\,[\,\mathbf{x} - \mathbf{w}_m(t)], & m \in \delta_{i(\mathbf{x})}(t) \\ \mathbf{w}_m(t), & \text{otherwise} \end{cases}$$

where $\alpha(t)$ is the learning rate, and $\delta_{i(\mathbf{x})}$ is the neighborhood function for the best matching unit of **x**. The learning rate decreases linearly with respect to the current iteration *t*, and the neighborhood function is the Gaussian function shown in Equation 2.3.3:

$$(2.3.3)\quad \delta_{i(\mathbf{x})} = e^{-k\,|i(\mathbf{x}) - j|^2},$$

where k is a constant scaling factor to adjust the rate the neighborhood radius decreases.

5. Repeat steps 2 through 4 until no discernible change occurs in the weight map.

The weights will eventually converge to a state where similar items are in close proximity. Convergence in one dimension was proven by Kohonen (1984) and Cottrell and Fort (1986), and proven in n-dimensions by Yin and Allinson (1995).

A simple demonstration of a self-organizing map that sorts colors may enhance understanding of the process. In this example, colors are represented as a three-dimensional vector of the percentages of the red, green and blue components that make up the color. The map in this example is a 50 × 50 grid. Fifty random colors are generated as a set of inputs, and the map of weights is initialized randomly with values in the range from 0

to 1 inclusive. A random color is selected from the input set for each of the 500 training iterations. Figure 2.3.2 shows the progress of the algorithm. Five frames out of 500 iterations are shown; the number of iterations between frames increases exponentially to show how quickly the initial stage of the learning process progresses. The top row of images is the map of weights, where each node has a particular color that it responds to. The bottom row of images shows where each input color is mapped; a square that matches the input value in color is drawn at the location of the best matching node for that input value.

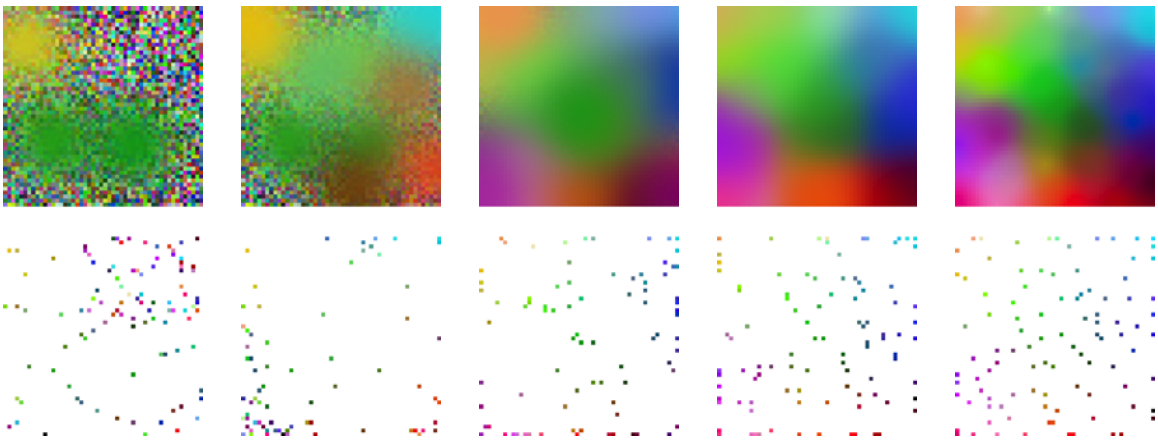


*Figure 2.3.2* Organization of a set of 50 random colors. The top row of images shows the weights for each of the nodes in the map, the bottom row shows which nodes map to the particular input samples. Shown from left to right, are iterations 3, 11, 41, 143 and 498 of 500 iterations.

In the AutoGraph application, the set of inputs contains the term frequency vectors for all documents in the index. Document vectors are presented sequentially, however memory management issues prevented the application from completing the required iterations for full training of the self-organizing map. The computational complexity of the algorithm is another, which is discussed in greater depth with the rest of the results.

## 2.4 Visual Display of Clustered Documents

Presentation of the data set generated by the self-organizing map involves both the field of information visualization and that of user interface design. These are perhaps the

most subjective aspects of this project because of their association with visual aesthetics. Information visualization is concerned with the clear and concise graphical presentation of abstract data, while interface design is concerned with intuitive and easy-to-understand interactions between the user and the software package. Because abstract data like relationships between documents lacks an inherent visual appearance, information visualization is often more challenging than scientific visualization. Additionally, interaction with such visualizations can be difficult because representations of abstract relationships can be disorienting (Mukherjea, 1999). Major work in both fields tends to focus on easily-quantifiable aspects that are often purely functional. For example, user interface testing tends to measure the time a user takes to complete a specific task. Only recently have researchers given surveys on subjective qualities such as the enjoyability of the experience. Focus on aesthetic qualities in either field remains secondary to functional qualities (Fry, 2004). In this project, I intend to design an interface that is meaningful, easy to use and visually appealing.

One of the benefits of self-organizing maps is that they are easy to visualize. Self-organizing maps have the convenient property that their structure can be easily mapped directly to the screen. Their dimensional reduction abilities make it easy to present high-dimenisonal data (like document contents) on a lower-dimensional surface. If the map of nodes is two-dimensional, nodes can simply have a one-to-one mapping with regions of an on-screen grid.

In designing AutoGraph's interface, I chose to place document icons in a grid based on the location of the document vector's best matching unit. This presents an interesting overall view of document organization, where similar documents are in close proximity to each other. A screen shot of the initial display of the document space is shown in Figure 2.4.1. This sort of arrangement is a departure from the traditional grid-based representation

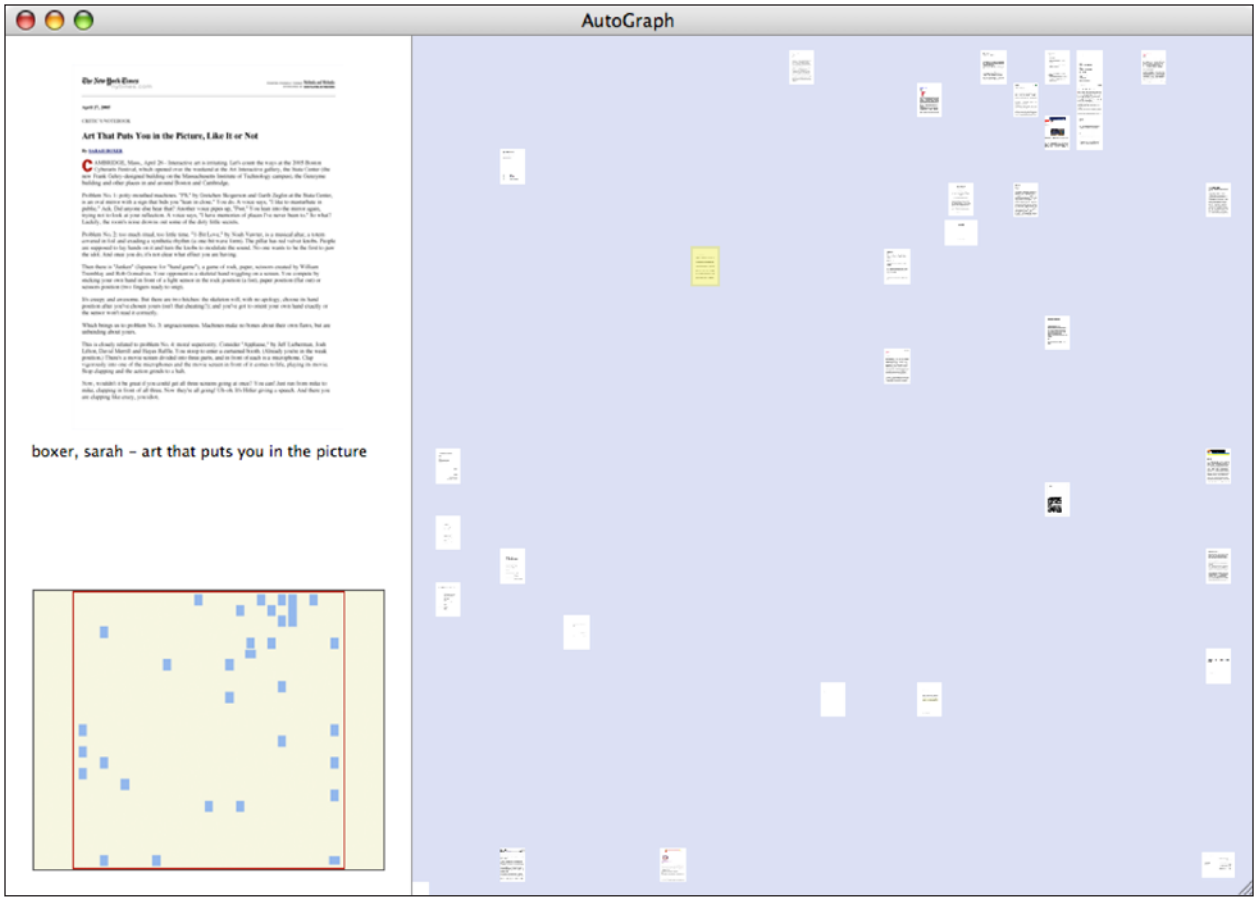
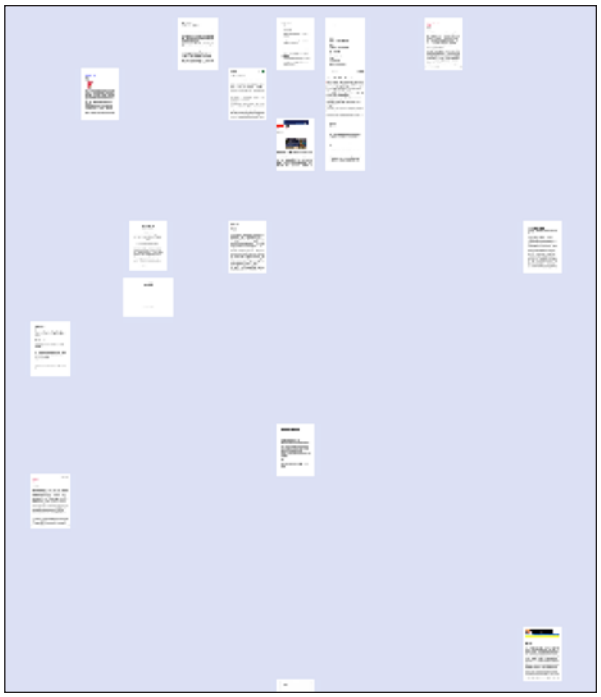



*Figure 2.4.1* A screenshot of the opening view (above), showing the clustered documents with an enlarged thumbnail of the selected document shown in the sidebar. Detail of the detailed information for the selected document (below, left) and of the clustered documents (below, right).

that wastes little space between documents, and it is interesting to note that in this representation unused space has semantic value, as distance between documents directly represents the degree of difference in their contents.

The initial view shows the overall space of clustered documents, with text labels denoting the categories that have been automatically found. Category labels are generated by finding the dominant term in each document vector.

Each document is represented by a miniature representation of its first page, or thumbnail, allowing quick visual identification of the document. While icons can successfully prompt identification of a file type, they do not indicate document contents. With large numbers of documents of the same type, icons are nearly useless for file identification (Vaillancourt, 2000). Thumbnails are already used regularly in visual identification of photo collections, and are slowly being applied to additional types of documents. Text documents very often have a distinct macrostructure that can be identified quickly even at reduced size. The actual content of the document is less of a memory prompt than the size of the first few paragraphs, location of images, or arrangement of text. At a large enough size, it is even possible to identify the document by its thumbnail alone, without additional information such as its title or authors.

## 2.5 An Interactive Browsing Interface

Since browsing is an interactive process, the ideal display of clustered documents is not static. The overall view of the document space is useful in the beginning stages of browsing in order to get a general sense of the layout of the space, but it does not present enough detailed information on smaller clusters of documents or single documents to be useful. Focus and context techniques are helpful for visualizing large amounts of data at multiple layers of detail. This approach to representation involves displaying in detail the information

that currently holds the user's interest while simultaneously showing general context information (Mukherjea, 1999). The small size and limited resolution of computer screens makes this difficult to present in one static view. Therefore, I implemented a simple zoom function to allow more detailed views of smaller regions of the overall map, and a sidebar that contains both additional information about a currently selected document and a smaller representation of the initial macro-view with a rectangular 'viewfinder' that denotes the section of the overall view that is shown in detail. Figure 2.4.2 shows a screen shot of this view.

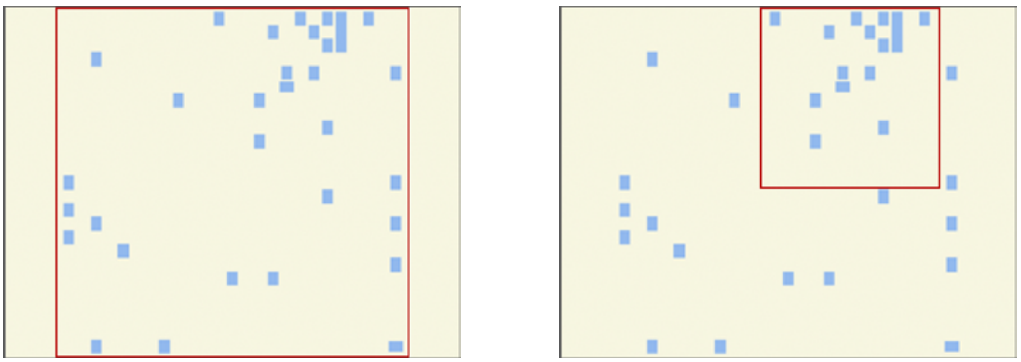


*Figure 2.4.2* The 'viewfinder' portion of the window. The red rectangle shows the area of the document space displayed in the window. Left, a viewfinder showing that the whole space is shown in the window, and right, a viewfinder showing that only the upper left portion of the space is shown.

The visual representation of the data was the intended focus of this project. Being the last step, information design and interface design often get the least attention in application development. The process of development from raw data to visual meaning is ideally handled by people with a deep understanding of all steps involved (Fry, 2004). This is the opposite of the traditional method of application development in which designers are asked to "make things pretty" at the end. If the designer has a good understanding of the data analysis and representation, the visual representation will be more coherent. Additionally, a designer with adequate understanding of the preceding steps can make changes to those stages, changing the data analysis or representation to make more sense in the visualization stage.

# 3 Results

This section presents a quantitative and qualitative analysis of the effectiveness of the AutoGraph application. Self-organizing map performance is analyzed based on the quality of the automatically generated categories, and the visualization and interaction are analyzed according to theories of information display presented by Fry (2004) and Tufte (1983, 1990).

## 3.1 Analysis of Categorization Quality

One method of determining the effectiveness of a self-organizing map is to compute a similarity map using the weight vectors for each node in the map. A similarity map shows how similar a given node is to the surrounding nodes by computing the average Euclidean distance between itself and the surrounding nodes. The algorithm is outlined below:

1. Iterate over all nodes in the map of nodes. For each node, compute the Euclidean distance to each surrounding node within *k* nodes in each direction, where *k* is a similarity weight; larger values of *k* observe more surrounding nodes. The average of all distances from the current node to the surrounding nodes is the similarity value for the current node.

3. Normalize all similarity values from the previous step to values between 0 and 1.

The similarity values are normalized to values between 0 and 1, inclusive. If the weight vectors around a node are similar to it, the distances between that node and the surrounding nodes will be small and the similarity value will be close to 0. Vectors that are different will have larger distances, and the normalized similarity value will be closer to 1.

AutoGraph saves a similarity matrix before and after the self-organization process. By

comparing the two snapshots, it is possible to analyze the quality of organization. Similarity matrices for a trial run on the set of 53 documents used for testing are shown in Figure 3.1.1. (The same trial run produced the screenshots in the previous section.)

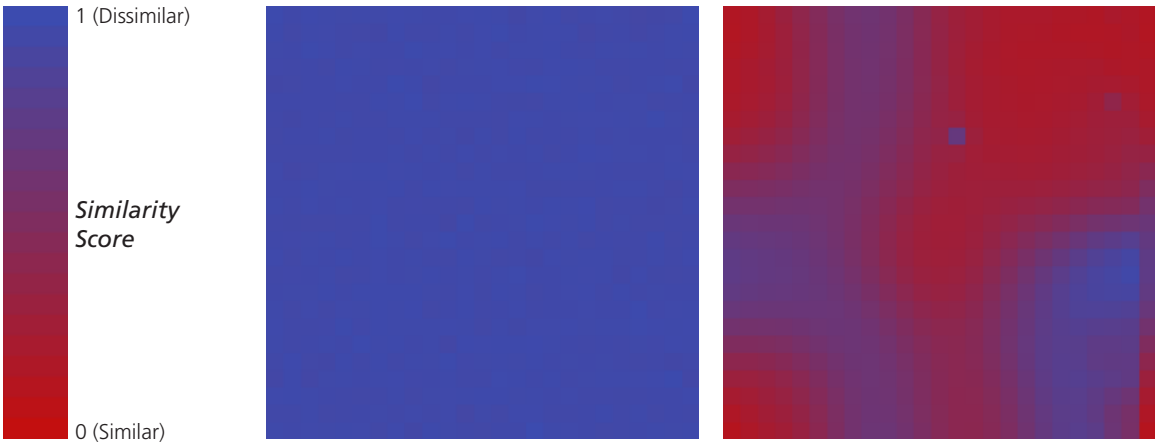



*Figure 3.1.1* Similarity matricies for the document space before (left), and after (right) the self-organization process. This performance metric shows an initially random space, where all nodes are different becoming more structured, where regions of similar items are separated by ridges of dissimilar ones.

The similarity matrix for the unorganized map shows values that are mostly 1, meaning that the entire map is made up of weight vectors that are different from their neighbors. This is to be expected given that the map is initialized with random values. The similarity matrix for the organized map shows the underlying structure of the clustered documents. Ridges where nodes are different from the surrounding nodes separate regions of high similarity. Each corner has similar items around it, with the upper left corner showing a large cluster of similar items.

Even though memory errors prevented the algorithm from running to completion, the map shows significantly more order than the original random configuration. Categorization quality should increase if the algorithm is allowed to run to completion.

## 3.2 Analysis of Visual Representation

Issues of graphic design play an important role in the quality of a representation of abstract data. Graphic design is not simply an exercise in making a representation more attractive — it is about creating a representation that is accurate and easy to understand (Fry, 2004). Because of the non-quantifiable nature of aesthetic improvements to a particular representation, work in the field of information visualization generally lacks an emphasis on high-quality visual design. Good design removes all non-essential information while presenting the important data with clarity and simplicity. Graphical excellence, as defined by Tufte, is "the well-designed presentation of interesting data" (Tufte, 1983).

> Graphical excellence consists of complex ideas communicated with clarity, precision, and efficiency. Graphical excellence is that which gives to the viewer the greatest number of ideas in the shortest time with the least ink in the smallest space. (Tufte, 1983)

The representation of documents in this project borrows two key ideas from Tufte's analysis of graphical excellence. Interesting in one regard is the multiple graphical purposes of a document's thumbnail. The thumbnails are both representations of the document itself and points in a larger display of document similarity. Multifunctioning graphical elements such as these show several different pieces of data at once (Tufte, 1983). The document similarity map acts as a scatterplot, where each point represents a particular document. The distances between documents have meaning in the scatterplot; closer points are more similar, while unrelated documents are outliers in the plot. Additional information is added when each point in the scatterplot is represented by a miniature image of the document's first page. By displaying thumbnails that serve two functions, more information is being communicated in the same amount of space, thereby increasing the value of the screen real estate taken up by the graphic.

The multifunctioning data points discussed above have the added benefit of adding a

second layer of scale to the representation. The overall structure of the clustered data points presents a global reading of the data, while the texture of each thumbnail presents small, detailed readings, showing structures present on the level of the individual document (Tufte, 1990). The power of the combination of general and detailed views is its flexibility. With multiple layers of context, comparisons can be simultaneously made on the largest and smallest of scales. Middle-range structures present "emergent regions," or similarity groupings that have additional context within elements of each region. The increased detail on all levels gives users more context than a more traditional representation where the arrangement of documents has no semantic significance.

It is interesting to note that the regular size of document thumbnails helps users distinguish documents even when looking at small thumbnails. The same postage-stamp size is repeated so that comparisons can be made at the same scale (Tufte, 1990). Regularity among many similar items helps the brain notice differences between the items. The white background of the page itself forms a frame for the data, and repetition of this data frame emphasizes small changes in the structure and layout of the individual pages.

Color plays an additional role in the presentation of documents in this project. The blue background increases contrast between the white page and the space around it, eliminating the need for distracting borders around the thumbnails. Moreover, the background's light value softens what would otherwise be a bright-white expanse; it has the effect of reducing glare on the screen (Tufte, 1990). Color is also used to separate annotations from data itself. Highly contrasting colors such as yellow and red are used to show the currently selected document in the map and to show the context of the self-organizing map pane when it is zoomed to a detailed view.

## 3.3 Analysis of Interaction

Tufte's discussions primarily analyze static graphical representations. The changing nature of a collection of files requires dynamic information displays. While there is a large amount of research in general usability, very little of it focuses on interacting with dynamic information visualizations.

Unfortunately, being the last step in the process, the interface design received the least attention in this project. Two interactive features for additional information about the documents are implemented. One is a detailed information pane for the current document, shown on the left side of the application window. This displays an enlarged thumbnail along with additional information such as filename, size, and date created. The second is a zoom interface for more detailed views of portions of the clustered documents. It is important for any interface to present enough choices so the user can navigate without frustration while limiting unnecessary and confusing options (Fry, 2004). This is especially true for "zoomable" user interfaces that are often awkward to use. It is easy to overshoot while zooming, so the AutoGraph application employs pre-determined discrete levels of zoom.

For larger document maps, it may be helpful to automatically zoom and center. For example, if a user were interested in documents on minimalist sculpture, the documents that matched the search criteria would be highlighted and the display would automatically center on that cluster. In addition, the display could zoom to a point where any documents that were outside the result set would be outside the edges of the result window.

One aspect which requires further testing is that of dynamically updating arrangements of items on the screen. Traditional thought in interface design emphasizes the importance of muscle memory in finding a particular item on the screen (Hansen, 1971). An object that

always appears in the same place can be found more quickly than one that moves because the user has a spacial memory of where the object is, as well as a physical memory of the movements required to reach that object. Users very often get frustrated when something moves out of their grasp. The dynamic placement of items in the similarity map in this project presents a similar problem. As documents are added and removed from the index, items may shift unexpectedly. This should be explored further with user testing.

# 4 Future Directions

The application written for this project is a first step in the development of a full-fledged document management system. Many issues need to be addressed, such as usability, efficiency, and graphical accuracy. As Fry notes, development of dynamic information visualization applications is an iterative process (2004). Addition and removal of features and improvements to both the visual design and user interaction will occur in each step towards a finished application. Some issues of immediate importance are discussed below.

## 4.1 User Testing

AutoGraph has not been subjected to any form of usability study. The interactive aspects of the application could be improved with knowledge of how users interact with the application in its current state. One drawback to usability testing is that time between iterations is considerably higher, as an application needs to reach a testable state, studies need to be conducted, results analyzed, and changes proposed. Paper prototyping may be a viable alternative. With paper prototyping, a mock-up of the interface made with printouts and paper cut-outs is tested for usability. This method eliminates the time required for development between usability studies, and can be done concurrently with application development. Additionally, a researcher with a deeper knowledge of psychology may be able to provide feedback that has not been considered in the project so far.

## 4.2 Algorithmic Efficiency

One of the drawbacks of the self-organizing map algorithm is its computational complexity. When tested, the learning process was by far the most time consuming process in the application, taking between three and seven minutes on the test set of 53 documents and approximately 20,000 terms. Some improvements to the algorithm are proposed by Roussinov and Chen (1998). The sparsity of the document vectors is the primary opportunity for optimization. Document vectors will contain a number of zeros, which

represent the non-existance of a term in a particular document. Because the terms that are automatically selected by AutoGraph are guaranteed to occur in only a fraction of the documents, the number of zeros in a document vector will be significant. Roussinov & Chen also propose a method for updating the weights  in time proportional to the number of non-zero elements in the document vectors. Also discussed is a method for computing distances to each node in time proportional to non-zero elements. These improvements are experimentally benchmarked to show dramatic increases in speed.

## 4.3 Additional Representations

Far more time could be given to the design of visual representations. Three-dimensional representations were given little thought because of the limitations of a two-dimensional screen, however, the use of additional dimensions may allow more interesting displays of information. Rather than simply adding another dimension to the self-organizing map's grid of weights, an interesting possibility is to map the weights onto the surface of a sphere. While this representation adds additional navigation issues, it resolves the issues of edge behavior in the self-organizing map. Observation of the learning process in the self-organizing color map shows colors being pushed towards the edges during the initial phase of learning. Changing the topology to a torus would eliminate these effects, but also present challenges to visualization on a two-dimensional screen. A sphere seems to be a more appropriate solution because it has a continuous surface and is easy to visualize.

# 5 Conclusion

This project shows that automatic document classification coupled with a good user interface has the potential to compete with traditional modes of document organization. While it is increasingly clear that automatic classification is superior to manual classification, other methods of categorization have not been explored. AutoGraph is a first step in the development of better tools for computer users. It demonstrates that there are alternatives to the traditional method of document organization which has changed little over the past thirty years.

# References


Apple Computer. (2005 a). *Search Kit programming guide*. Cupertino, CA: Author. Retrieved February 22, 2005, from http://developer.apple.com/documentation/UserExperience/Conceptual/SearchKitConcepts/SearchKitConcepts.pdf

Apple Computer. (2005 b). *Search Kit reference*. Cupertino, CA: Author. Retrieved February 22, 2005, from http://developer.apple.com/documentation/UserExperience/Reference/SearchKit/SearchKit_Reference.pdf

Barreau, D., & Nardi, B. A. (1995, July). Finding and reminding: File organization from the desktop. *SIGCHI Bulletin*, *27*(3), 39-43. Retrieved March, 2006, from ACM Digital Library database.

Cottrell, M., & Fort, J.-C. (1987). Étude d'un processus d'auto-organisation [Study of a process of self-organization]. *Annales de l'institut Henri Poincaré (B) Probabilités et Statistiques* [Annals of the Henri Poincaré Institute], *1*, 1-20. Retrieved April 24, 2006, from Numdam database: http://www.numdam.org/item?id=AIHPB_1987__23_1_1_0

Fry, B. J. (2004). *Computational information design*. Unpublished doctoral dissertation, Massachusetts Institute of Technology, Cambridge, MA. Retrieved November 14, 2005, from http://acg.media.mit.edu/people/fry/phd/dissertation-050312b-acrobat.pdf

Gelernter, D. H. (1998). Beyond the desktop. In *Machine beauty*: Elegance and the heart of technology (pp. 87 - 117). New York: BasicBooks.

Giampaolo, D. (1999). *Practical file system design with the Be File System*. San Francisco: Morgan Kaufmann.

Hansen, W. J. (1971). User engineering principles for interactive systems. In *Proceedings of the Fall Joint Computer Conference* (pp. 523-532). Mondale, NJ: AFIPS Press.

Harada, S., Naaman, M., Song, Y. J., Wang, Q. Y., & Paepcke, A. (2004). Lost in memories: Interacting with photo collections on PDAs. *Proceedings of the 4th ACM/IEEE-CS Joint Conference on Digital Libraries*, 325-333. Retrieved November, 2005, from ACM Digital Library database.

Haykin, S. (1994). *Neural networks: A comprehensive foundation*. Upper Saddle River, NJ: Prentice-Hall.

Hertzum, M., & Frøkjær, E. (1996, June). Browsing and querying in online documentation: A study of user interfaces and the interaction process. *ACM Transactions on Computer-Human Interaction*, *3*(2), 136-161. Retrieved April 17, 2006, from ACM Digital Library database.

Kohonen, T. (1984). *Self-organization and associative memory*. Berlin: Springer-Verlag.

Marsden, G., & Cairns, D. E. (2003). Improving the usability of the hierarchical file system. *Proceedings of SAICSIT*, 122-129. Retrieved November, 2005, from ACM Digital Library database.

Mukherjea, S. (1999, December). Information visualization for hypermedia systems. *ACM Computing Surveys*, *31*(4es), article 6. Retrieved November, 2005, from ACM Digital Library database.

Roussinov, D. G., & Chen, H. (1998). A scalable self-organizing map algorithm for textual classification: A neural network approach to thesaurus generation. *Communication and Cognition in Artificial Intelligence*, *15*(1-2), 81-111. Retrieved April 19, 2006, from dLIST database: http://dlist.sir.arizona.edu/

Salton, G., Wong, A., & Yang, C. S. (1975, November). A vector space model for automatic indexing. *Communications of the ACM*, *18*(11), 613-620. Retrieved November, 2005, from ACM Digital Library database.

Tufte, E. R. (1983). *The visual display of quantitative information*. Cheshire, CT: Graphics Press.

Tufte, E. R. (1990). *Envisioning information*. Cheshire, CT: Graphics Press.

Vaillancourt, A. D. M. G. (1998). Generated glyphs as memorable desktop icons for documents. *Proceedings of the 1998 Workshop on New Paradigms in Information Visualization and Manipulation*, 9-12. Retrieved April 15, 2006, from ACM Digital Library database.

Weippl, E. (2001). Visualizing content based relations in texts. *Proceedings of the 2nd Australasian Conference on User Interface*, 34-41. Retrieved November, 2005, from ACM Digital Library database.

Yin, H., & Allinson, N. M. (1995). On the distribution and convergence of feature space in self-organising maps. *Neural Computation*, *7*(6), 1178-1187. Retrieved April 16, 2006, from University of Manchester, School of Electrical and Electronic Engineering Web site: http://soft.ee.umist.ac.uk/hujun/mypublications/p_nc1.ps

# A Content Indexing Code

The following Objective-C code shows the functions used for content indexing in the AutoGraph application. The code relies on Apple's Cocoa API for OS X application development.

## A.1 Excerpts from IndexController.m

```
- (void) createIndexWithPath: (NSString *) pathString {

        if (selectedIndex != NULL) {
                SKIndexClose(selectedIndex);
        }

        SKLoadDefaultExtractorPlugIns();

        NSString *indexName = @"Browser";

        SKIndexType type = kSKIndexInverted;

        NSNumber *minTermLength = [NSNumber numberWithInt: 3];
        NSNumber *maxTerms = [NSNumber numberWithInt: 0];

        NSSet * stopwords = [NSSet setWithObjects:
                @"i", @"a", @"about", @"an", @"are", @"as", @"at", @"and",
                @"be", @"by", @"for", @"from", @"how", @"in", @"is", @"it",
                @"its", @"it's", @"of", @"on", @"or", @"that", @"the",
                @"this", @"to", @"was", @"what", @"when", @"where", @"who",
                @"will", @"with", nil
        ];

        NSDictionary * properties = [NSDictionary dictionaryWithObjectsAndKeys:
                minTermLength, @"kSKMinTermLength",
                (CFSetRef) stopwords, @"kSKStopWords",
                maxTerms, @"kSKMaximumTerms",
                //@"", @"kSKStartTermChars",   // additional starting-characters for terms
                @"-_@.'", @"kSKTermChars",   // additional characters within terms
                //@"", @"kSKEndTermChars",     // additional ending-characters for terms
                nil
        ];

        selectedIndex = SKIndexCreateWithURL (
                                        (CFURLRef) [NSURL fileURLWithPath: pathString],
                                        (CFStringRef) indexName,
                                        type,
                                        (CFDictionaryRef) properties
        );

        NSDictionary *temp = (NSDictionary *) SKIndexGetAnalysisProperties(selectedIndex);
        NSArray *tempArray = [NSArray arrayWithArray:
                                        [[temp objectForKey:@"kSKStopWords"] allObjects]];
        NSString *tempStr = [[NSString alloc] init];
        int i;
```

```
        for(i = 0; i < [tempArray count]; i++) {
                tempStr = [[tempStr stringByAppendingString:
                                [tempArray objectAtIndex:i]] stringByAppendingString:@" "];
        }
        NSLog(tempStr);
}



- (void) openIndexWithPath: (NSString *) pathString  {

        if (selectedIndex != NULL) {
                SKIndexClose(selectedIndex);
        }

        SKLoadDefaultExtractorPlugIns();

        NSString *indexName = @"Browser";

        selectedIndex = SKIndexOpenWithURL (
                                (CFURLRef) [NSURL fileURLWithPath: pathString],
                                (CFStringRef) indexName, true);

        NSDictionary *temp = (NSDictionary *) SKIndexGetAnalysisProperties(selectedIndex);
        NSArray *tempArray = [NSArray arrayWithArray:
                                        [[temp objectForKey:@"kSKStopWords"] allObjects]];
        NSString *tempStr = [[NSString alloc] init];
        int i;
        for(i = 0; i < [tempArray count]; i++) {
                tempStr = [[tempStr stringByAppendingString:
                                [tempArray objectAtIndex:i]] stringByAppendingString:@" "];
        }
        NSLog(tempStr);

}



- (void) addToIndexDocumentsWithPath: (NSString*) path {

        NSURL *url = [NSURL fileURLWithPath: path];
        SKDocumentRef doc = SKDocumentCreateWithURL ((CFURLRef) url);

        [(id) doc autorelease];

        SKIndexAddDocument (selectedIndex, doc, NULL, true);

        PDFDocument *pdfDoc;
        pdfDoc = [[PDFDocument alloc] initWithURL: url];

        ThumbnailView *t = [[ThumbnailView alloc] initWithDocument: pdfDoc];
        [t drawRect: [t pageBounds]];
}



- (BOOL) addAllDocumentsFromIterator: (SKIndexDocumentIteratorRef)iterator
                                toArray: (NSMutableArray*)array {
```

```
        if (iterator) {
                CFRetain(iterator);
        } else {
                iterator = SKIndexDocumentIteratorCreate(selectedIndex,NULL);
        }

        SKDocumentRef doc;
        BOOL iteratorIsEmpty = TRUE;

        while (doc = SKIndexDocumentIteratorCopyNext(iterator)) {

                SKIndexDocumentIteratorRef subIter = SKIndexDocumentIteratorCreate(
                                                                selectedIndex,doc);

                if (subIter) {

                        // If the sub-iterator is empty (addAllDocumentsFromIterator
                        // returns true), we've reached a leaf node. Add the document.

                        if([self addAllDocumentsFromIterator:subIter toArray:array]) {
                                [array addObject:
                                [NSMutableDictionary dictionaryWithObjectsAndKeys:
                                        [NSNumber numberWithInt:
                                                SKIndexGetDocumentID(selectedIndex, doc)],
                                        @"documentID",
                                        (NSString *) SKDocumentGetName(doc),
                                        @"documentName",
                                        nil
                                ]];
                        }

                        CFRelease(subIter);
                }

                CFRelease(doc);
                iteratorIsEmpty = FALSE;
        }

        CFRelease(iterator);
        return iteratorIsEmpty;
}
```

# B Kohonen Self-Organizing Map Code

The following Objective-C code is the Self-Organizing Map code used in the AutoGraph application. The code relies on Apple's Cocoa API for OS X application development.

## B.1 SelfOrganizingMap.h

```
#import <Cocoa/Cocoa.h>


@interface SelfOrganizingMap : NSObject {

        int iteration;
        int timeLimit;

        NSSize mapDimensions;
        int inputVectorArity;

        float neighborhoodRadius;

        NSArray **weights;
}

- (id) initWithMapSize: (NSSize) mapSize andVectorSize: (int) arity;

- (bool) performIterationWithSample: (NSArray *) sample;
- (NSPoint) getBestMatchingUnit: (NSArray *)inVect;
- (void) scaleNeighborsOfUnit: (NSPoint *) unit withValue: (NSArray *) sample;

- (float) computeSimilarityMap;

- (float) distanceFromVector: (NSArray *) v1 toVector: (NSArray *) v2;
- (float) distanceSquaredFromVector: (NSArray *) v1 toVector: (NSArray *) v2;
- (float) distanceFromPoint: (NSPoint *) p1 toPoint: (NSPoint *) p2;
- (float) distanceSquaredFromPoint: (NSPoint *) p1 toPoint: (NSPoint *) p2;
- (NSArray *) multiplyVector: (NSArray *) v1 byVector: (NSArray *) v2;
- (NSArray *) multiplyVector: (NSArray *) v1 byScalar: (NSNumber *) s;
- (NSArray *) addVector: (NSArray *) v1 toVector: (NSArray *) v2;

- (NSArray **) weights;
- (int) mapLength;
- (NSSize) mapSize;
- (int) mapDepth;

- (int) iteration;

- (float) percentComplete;

- (int) timeLimit;
- (void) setTimeLimit: (int) limit;

@end
```

## B.2 SelfOrganizingMap.m

```
#import <Cocoa/Cocoa.h>


#import “SelfOrganizingMap.h”
#import <math.h>
#define SIMILARITY_WEIGHT 3

@implementation SelfOrganizingMap

- (id) initWithMapSize: (NSSize) size andVectorSize: (int) arity {
        int weightsLength;
        self = [super init];

        iteration = 0;
        timeLimit = 95;

        neighborhoodRadius = 60;

        mapDimensions = size;
        inputVectorArity = arity;

        weights = malloc(sizeof(NSArray*) * mapDimensions.width * mapDimensions.height);
        weightsLength = mapDimensions.width * mapDimensions.height;

        srand48(time(NULL));

        int i, j;
        NSMutableArray *temp;

        for (i = 0; i < weightsLength; i++) {
                temp = [[NSMutableArray alloc] init];
                for (j = 0; j < inputVectorArity; j++) {
                        [temp addObject:[NSNumber numberWithFloat: drand48()]];
                }
                weights[i] = temp;
        }

        return self;
}



// - (bool) performIterationWithSample: (NSArray *) sample;
// ------------------------------------------------------------------
// The performIteration method is given a randomly selected input
// vector and if the current iteration is less than the time limit it
// finds the best matching unit and scales the units in that
// neighborhood. The method returns true if the current iteration is
// less than the time limit and returns false otherwise.

- (bool) performIterationWithSample: (NSArray *) sample {
        if (iteration < timeLimit) {
                NSPoint bmu = [self getBestMatchingUnit: sample];
                NSLog(@”BMU %d, %d”, (int)bmu.x, (int)bmu.y);
                [self scaleNeighborsOfUnit: &bmu withValue: sample];
                iteration++;
```

```
                return true;
        } else {
                return false;
        }
}



// - (NSPoint *) getBestMatchingUnit: (NSArray *) inVect;
// ---------------------------------------------------------------------
// The getBestMatchingUnit method searches the map for the unit whose
// weight vector is closest to the input vector. It returns an NSPoint
// contains the closest unit's coordinates in the map.

- (NSPoint) getBestMatchingUnit: (NSArray *)inVect {
        NSPoint temp;
        temp.x = -1;
        temp.y = -1;

        float distance, minimum = FLT_MAX;

        int i, j;
        if ([inVect count] == inputVectorArity) {
                for (i = 0; i < mapDimensions.width; i++) {
                        for (j = 0; j < mapDimensions.height; j++) {
                                distance = [self distanceSquaredFromVector: inVect
                                                toVector: weights[i *
                                                (int)mapDimensions.width + j]];
                                if (distance < minimum) {
                                        minimum = distance;
                                        temp.x = i;
                                        temp.y = j;
                                }
                        }
                }
        }

        return temp;

}



// - (void) scaleNeighborsOfUnit: (NSPoint *) unit withValue: (NSArray *) sample;
// ---------------------------------------------------------------------
// The scaleNeighbors method scales the neighbors of unit closer to the
// value of sample. The scale function is time dependent, and both the
// radius of influence and the amount each unit learns decreases over
// time (the instance variable iterations).

- (void) scaleNeighborsOfUnit: (NSPoint *) unit withValue: (NSArray *) sample {
        float percentComplete = (float) iteration / (float) timeLimit;
        int r = (int) round( neighborhoodRadius * (1 - percentComplete) / 2 );

        NSPoint zero;
        zero.x = 0;
        zero.y = 0;

        NSPoint outer;
```

```
        outer.x = r;
        outer.y = r;

        float maxDistance = r * r;
        float distance, a;

        NSArray *product1, *product2, *old;

        int i, j;
        for (i = -r; i < r; i++) {
                for (j = -r; j < r; j++) {
                        if ((unit->x + i) >= 0 &&
                            (unit->x + i) < mapDimensions.width &&
                            (unit->y + j) >= 0 && (unit->y + j) < mapDimensions.height) {

                                // Get distance from center point and normalize it
                                outer.x = i;
                                outer.y = j;
                                distance = fminf(
                                        [self distanceSquaredFromPoint: &zero
                                                               toPoint: &outer], r * r);
                                distance /= maxDistance;

                                // Get the scale factor
                                a = expf(-1 * distance / 0.15);

                                // The amount a neuron learns decreases with time
                                a /= percentComplete * 4.0 + 1.0;

                                // Scale the selected unit parametrically
                                old = weights[((int)unit->x + i) *
                                        (int)mapDimensions.width + ((int)unit->y + j)];

                                product1 = [self multiplyVector:weights[((int)unit->x + i) *
                                                (int)mapDimensions.width +
                                                ((int)unit->y + j)]
                                        byScalar:[NSNumber numberWithFloat:1 - a]];

                                product2 = [self multiplyVector:sample
                                                byScalar:[NSNumber numberWithFloat:a]];

                                weights[((int)unit->x + i) *
                                          (int)mapDimensions.width +
                                          ((int)unit->y + j)] = [self addVector: product1
                                                                       toVector: product2];

                                [old release];
                                [product1 release];
                                [product2 release];
                        }
                }
        }
        //[pool release];
}


// mapDimensions must be greater than HEIGHT = 1 and WIDTH = 1.
- (float) computeSimilarityMap {
        float* similarity_map;
```

```
        NSArray* center;
        similarity_map = malloc(sizeof(float) * mapDimensions.height *
                                                    mapDimensions.width);

        NSString* output = [NSString stringWithString: @"float[][] similarityMap = { "];

        int i, j, k, l, itemsInAverage;
        float total, maxDistance = 0.0f;

        for (i = 0; i < mapDimensions.width; i++) {
                for (j = 0; j < mapDimensions.height; j++) {
                        center = weights[i * (int)mapDimensions.width + j];

                        itemsInAverage = 0;
                        total = 0.0f;

                        for (k = -SIMILARITY_WEIGHT; k <= SIMILARITY_WEIGHT; k++) {
                                for (l = -SIMILARITY_WEIGHT; l <= SIMILARITY_WEIGHT; l++) {
                                        if ( (k + i >= 0) &&
                                             (k + i < mapDimensions.width) &&
                                             (l + j >= 0) &&
                                             (l + j < mapDimensions.height)) {

total += [self distanceFromVector: weights[(k + i) * (int)mapDimensions.width + (l + j)]
                toVector: center];
                                                itemsInAverage++;
                                        }
                                }
                        }

                        total /= itemsInAverage - 1;

                        if (total > maxDistance) {
                                maxDistance = total;
                        }

                        similarity_map[i * (int)mapDimensions.width + j] = total;
                }
        }

        total = 0.0f;

        for (i = 0; i < mapDimensions.width - 1; i++) {
                output = [output stringByAppendingString: @"{"];
                for (j = 0; j < mapDimensions.height - 1; j++) {

output = [output stringByAppendingFormat: @"%f, ",
                similarity_map[i * (int)mapDimensions.width + j] / maxDistance];
total += similarity_map[i * (int)mapDimensions.width + j];

                }

                output = [output stringByAppendingFormat: @"%f}, ",
                similarity_map[i * (int)mapDimensions.width +
                                        (int)mapDimensions.height - 1] / maxDistance];

                total += similarity_map[i * (int)mapDimensions.width +
                                                (int)mapDimensions.height - 1];
        }
```

```
        output = [output stringByAppendingString: @"{"];
        for (j = 0; j < mapDimensions.height - 1; j++) {
                output = [output stringByAppendingFormat: @"%f, ",
                        similarity_map[(int)mapDimensions.width - 1 *
                                (int)mapDimensions.width + j] / maxDistance];
                total += similarity_map[(int)mapDimensions.width - 1 *
                                                (int)mapDimensions.width + j];
        }
        output = [output stringByAppendingFormat: @"%f} };",
                        similarity_map[(int)mapDimensions.width - 1 *
                                (int)mapDimensions.width +
                                (int)mapDimensions.height - 1] / maxDistance];
        total += similarity_map[(int)mapDimensions.width - 1 *
                                (int)mapDimensions.width +
                                (int)mapDimensions.height - 1];

        NSLog(output);

        return total;
}


// Linear algebra methods

- (float) distanceFromVector: (NSArray *) v1 toVector: (NSArray *) v2 {
        float temp = -1;

        if ([v1 count] == [v2 count]) {
                temp = 0;

                int i;
                for (i = 0; i < [v1 count]; i++) {
                        temp += pow([[v1 objectAtIndex:i] floatValue]
                                                - [[v2 objectAtIndex:i] floatValue], 2);
                }

                temp = sqrt(temp);
        }

        return temp;
}

- (float) distanceSquaredFromVector: (NSArray *) v1 toVector: (NSArray *) v2 {
        float temp = -1;

        if ([v1 count] == [v2 count]) {
                temp = 0;

                int i;
                for (i = 0; i < [v1 count]; i++) {
                        temp += pow([[v1 objectAtIndex:i] floatValue]
                                                - [[v2 objectAtIndex:i] floatValue], 2);
                }
        }

        return temp;
}
```

```
- (float) distanceFromPoint: (NSPoint *) p1 toPoint: (NSPoint *) p2 {
	float temp;
	temp = pow(p1->x - p2->x, 2);
	temp += pow(p1->y - p2->y, 2);
	return sqrt(temp);
}


- (float) distanceSquaredFromPoint: (NSPoint *) p1 toPoint: (NSPoint *) p2 {
	float temp;
	temp = pow(p1->x - p2->x, 2);
	temp += pow(p1->y - p2->y, 2);
	return temp;
}


- (NSArray *) multiplyVector: (NSArray *) v1 byVector: (NSArray *) v2 {
	NSMutableArray* temp = [[NSMutableArray alloc] init];

	int i;
	for (i = 0; i < [v1 count]; i++) {
		[temp addObject:[NSNumber numberWithFloat:
		[[v1 objectAtIndex:i] floatValue] * [[v2 objectAtIndex:i] floatValue]]];
	}

	return temp;
}

- (NSArray *) multiplyVector: (NSArray *) v1 byScalar: (NSNumber *) s {
	NSMutableArray* temp = [[NSMutableArray alloc] init];
	float scalar = [s floatValue];

	int i;
	for (i = 0; i < [v1 count]; i++) {
		[temp addObject:[NSNumber numberWithFloat:
		[[v1 objectAtIndex:i] floatValue] * scalar]];
	}

	return temp;
}

- (NSArray *) addVector: (NSArray *) v1 toVector: (NSArray *) v2 {
	NSMutableArray* temp = [[NSMutableArray alloc] init];

	int i;
	for (i = 0; i < [v1 count]; i++) {
		[temp addObject:[NSNumber numberWithFloat:
		[[v1 objectAtIndex:i] floatValue] + [[v2 objectAtIndex:i] floatValue]]];
	}

	return temp;
}
```

```
// Getters and setters for the self organizing map

- (NSArray **) weights {
	return weights;
}

- (int) mapLength {
	return (mapDimensions.width * mapDimensions.height);
}

- (NSSize) mapSize {
	return mapDimensions;
}

- (int) mapDepth {
	return inputVectorArity;
}

- (int) iteration {
	return iteration;
}

- (float) percentComplete {
	return (float) iteration / (float) timeLimit;
}

- (int) timeLimit {
	return timeLimit;
}

- (void) setTimeLimit: (int) limit {
	if (limit > 0) timeLimit = limit;
}

@end
```