Dotplot: a Program for Exploring Self-Similarity in Millions of Lines of Text and Code

Kenneth Ward Church
AT&T Bell Laboratories
Murray Hill, NJ 07974-2070
kwc@research.att.com

Jonathan Isaac Helfman
AT&T Bell Laboratories
Murray Hill, NJ 07974-2070
jon@research.att.com

ABSTRACT

An interactive program, *dotplot*, has been developed for browsing millions of lines of text and source code, using an approach borrowed from biology for studying homology (self-similarity) in DNA sequences. With conventional browsing tools such as a screen editor, it is difficult to identify structures that are too big to fit on the screen. In contrast, with dotplots we find that many of these structures show up as diagonals, squares, textures and other visually recognizable *features*, as will be illustrated in examples selected from biology and two new application domains: text (AP news, Canadian Hansards) and source code (5ESS®). In an attempt to isolate the mechanisms that produce these features, we have synthesized similar features in dotplots of artificial sequences. We also introduce an approximation that makes the calculation of dotplots practical for use in an interactive browser.

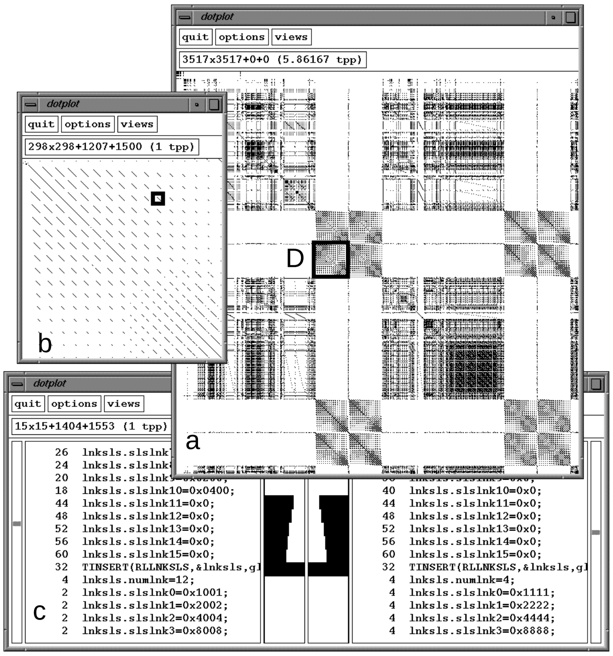

Fig. 1. Dotplot Browser

1. Introduction

We describe a graphical tool for browsing millions of lines of text and source code. It is hard to use a screen editor to conceptualize input that is much larger than the size of a screen. Following Eick (1992), who advocates the use of interactive graphical tools to help understand large software systems, we have developed a browser that can display millions of lines of input using a *dotplot*, a plot very much like those used in molecular biology for studying homology. Dotplots (not to be confused with Tukey's ''dot plot'' (1977, p.50)) are constructed by first *tokenizing* a sequence (i.e. splitting it into lines, words, characters, etc.) and then placing a dot in position i, j if the i^{th} input token is the same as the j^{th}. We believe the dotplot browser may be useful for discovering large-scale structures that may be hard to spot with conventional tools such as a screen editor: conventional tools may be too myopic to show the big picture.

Fig. 1 shows the browser in action. Three views of a source code file are presented: a) a global overview of the file in the upper right, b) a magnified view of a small portion of the file in the upper left, and c) a text view along the bottom. The views are linked together so that clicking and scrolling in one view updates the others appropriately.

Notice the fascinating *diagonals*, *squares*, and *textures* in Fig. 1. The texture labeled D will be discussed in more detail in Section 4.2. What mechanisms could be responsible for these *features*? What do the features tell us about the input sequence? This paper uses two approaches to investigate such questions. In addition to the browser, which allows us to analyze naturally occurring sequences, we also synthesize artificial sequences in an attempt to replicate features found with the browser. Both methods, analysis and synthesis, are used to study the mechanisms that might be responsible for the features.

Fig. 2 shows several synthesized dotplots. Fig. 2a, for example, was generated from the artificial sequence: ''zyxwvutsrqponmlkji''. In this case, dots appear along the main diagonal and nowhere else, because all of the input tokens are distinct. In contrast with Fig. 2a, there are two interesting diagonals in Fig. 2b, indicating that the subsequence ''abcdefghi'' is repeated. We have found that diagonals, and other features, are often symptomatic of certain potentially important patterns in the input sequence.

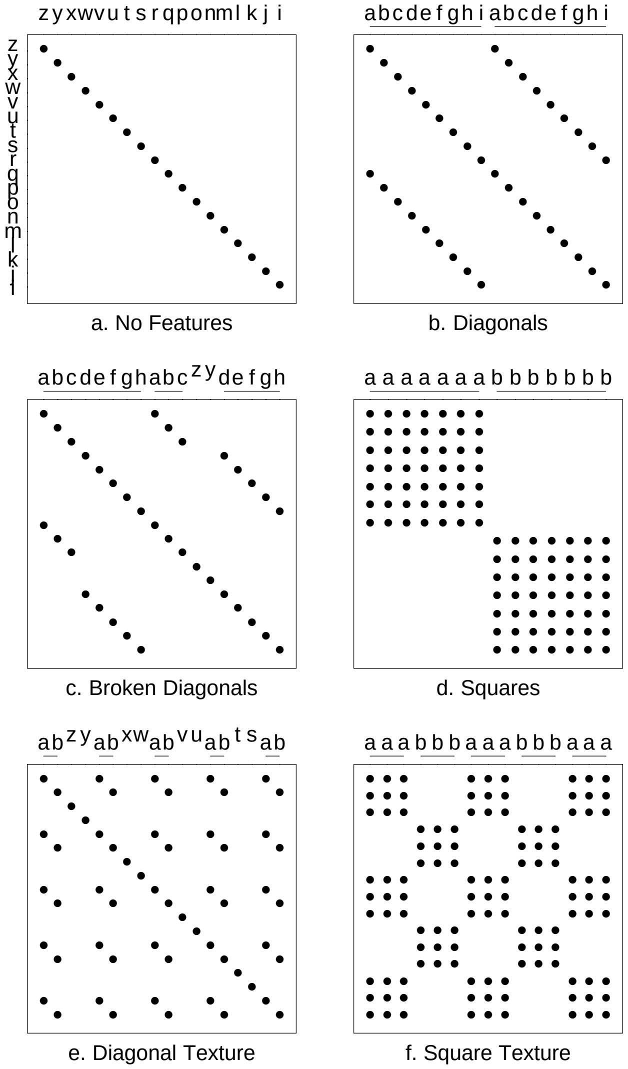

Fig. 2. Features in Synthesized Dotplots

A number of conventions will be used throughout this paper. Underlining (as in Fig. 2b) is used to emphasize tokens that repeat; raising the baseline (as in Fig. 2c) is used to emphasize tokens that do not repeat. In general, letters from the beginning of the alphabet are used to denote repeating tokens, and letters from the end of the alphabet are used to denote non-repeating tokens. Labels along the left margin are often omitted. Finally, the somewhat unusual convention of placing the origin in the upper left corner was chosen in order to conform with the fact that English text is read left to right and top to bottom. The interaction of the text views and dotplot views is more natural when the location of the origin is consistent.

2. Dotplots of DNA Sequences

The features in Figs. 2b-f can also be found in dotplots of real sequences. For example, diagonals are found in Fig. 3, a dotplot of two concatenated DNA sequences: (A) the plasmid pBR322 (Balbas et al. 1986), and (B), the plasmid pUC19 (Yanisch-Perron et. al. 1985). Dotplots are a well-known technique in biology for studying homology (e.g., Maizel & Lenk 1981, Pustell & Kafatos 1982). Biologists are very interested in diagonals which indicate, in this case, that both pBR322 and pUC19 carry the β-lactamase gene that confers ampicillin resistance. Biologists have also used dotplots to look at how sequences fold into three-dimensional structures (e.g., Quax-Jeuken et al. 1983, Blundell et al. 1987, Carrington & Morris 1987), and to investigate evolutionary questions (e.g., Laver et al. 1980, Doolittle 1981, Lake et al. 1988). A brief description of the value of this approach can be found in Argos (1987). See Vingron (1991) for a recent thesis on genetic sequence alignment.

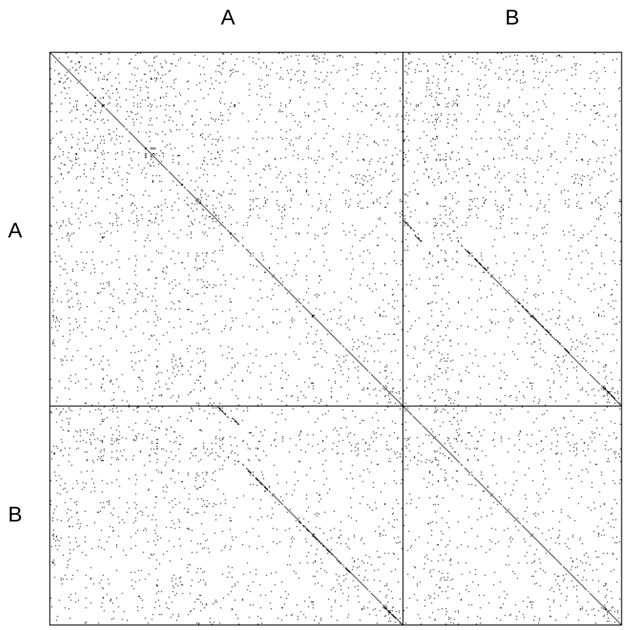

Fig. 3. Dotplot of Two DNA Sequences (7000 Nucleotides)

A few additional conventions are introduced in Fig. 3. Grid lines are used to indicate the boundaries between sequence A and sequence B. In addition, the grid box in the upper left corner is called ‘‘AA,’’ the grid box in the upper right is called ‘‘AB,’’ and so on. Grid box AA compares sequence A with itself, while grid box AB compares sequence A with sequence B. In general, the grid boxes along the main diagonal compare two identical sequences, while the other grid boxes compare two different sequences.

Note that the diagonals are broken. What does this mean? Fig. 2c (above) shows, by synthesis, that broken diagonals are caused by the insertion of non-repeating tokens (zy) into an otherwise matching subsequence (abcdefgh). In Fig. 3, the breaks probably indicate that some non-repeating nucleotides have been inserted near the beginning of B, interrupting the match between the second half of A and most of B.

3. Dotplots of Text

3.1 AP News: Broken Diagonals in Text

Broken diagonals can also be found in dotplots of text, as illustrated in Fig. 4, a dotplot of four Associated Press (AP) news stories, labeled A-D. The four stories, about Ryan White's death from AIDS, were sent over the AP wire within a few days of each other in early April of 1990. For text applications, we usually choose to tokenize the input into words.

	Date	Time	Lead	Title
A	040390	19:54	RyanWhite-Chro	
B	040990	02:47	White-Chronolo	White's Struggle Wi...
C	040390	14:03	RyanWhite	Ryan White, AIDS Be...
D	040390	04:18	RyanWhiteChron	

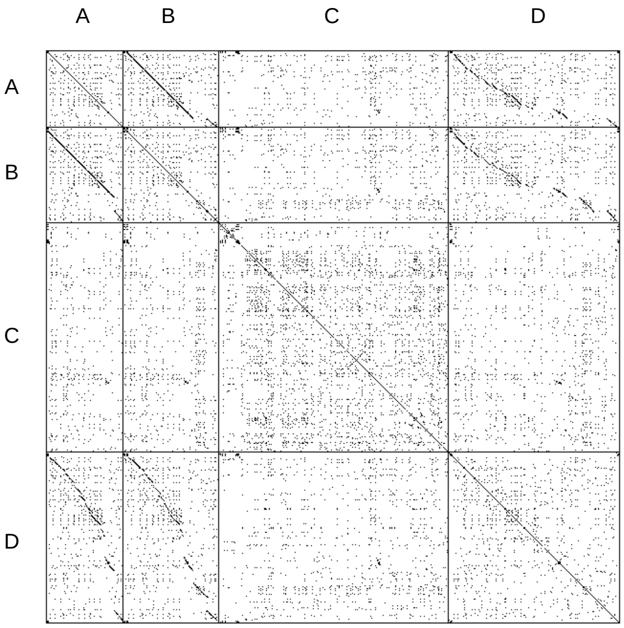

Fig. 4. Four AP News Stories (3000 Words)

What do the broken diagonals tell us about these AP news stories? We suspect that the diagonals are broken when a news story is updated with a few additional facts. Stories A, B, and D appear to be related in this way, as evidenced by the broken diagonals in grid boxes AB, BA, AD, DA, BD, DB. In contrast, story C is probably not a rewrite of the others, as evidenced by the absence of the broken diagonals in AC, CA, BC, CB, DC, CD.

Dotplots may have practical ramifications for Information Retrieval (IR) (Salton 1989). There are a number of IR systems that provide rapid access to documents in large electronic libraries. Experience with such systems has shown that users find it difficult to construct queries that cover most of the documents of interest and not too many others. This problem could be alleviated, in part, if document retrieval systems had a more effective way of handling rewrites. In particular, the user is probably only interested in one of the rewrites, e.g. story D. To date, most retrieval systems consider each document one at a time, and consequently, they would usually return either all of the rewrites or none of them. There is generally no easy way to retrieve just one of the rewrites, along with an indication that there are a few more that are nearly the same.

Thus far we have seen two examples of broken diagonals. The next section shows how diagonals can be combined with squares.

3.2 Hansards: Combinations of Sparse Features

Fig. 5 is a dotplot of 37 million words of Canadian *Hansards*, parliamentary debates, which are available in both English and French. The input is constructed by concatenating 3 years of debates in English (37/2 million words) followed by the French equivalent (the remaining 37/2 million words). Consequently, there is a lag of approximately 37/2 million words between an English sentence and its French translation. 37 million is such a large amount of data that the dots in Fig. 5 represent the relative number of matches per pixel, rather than the existence or non-existence of a particular match.

Note the diagonals and large dark squares in Fig. 5. We have seen examples of these features in isolation, but what mechanism could explain the combination? Figs. 6 and 7 present a two step solution: first, sparse versions of the diagonals and squares are synthesized, as illustrated in Figs. 6b and 6d, and then the interesting halves of Figs. 6b and 6d are interleaved to produced the desired combination in Fig. 7.

How does the synthetic sequence in Fig. 7 relate to the Hansards? Let the a's denote English words, e.g. *government*, the b's denote French words, e.g. *gouvernement*, and the c-k's denote words that are the same in both English and French, e.g. proper nouns, dates, times, numbers, etc. We hypothesize that the square in the upper left is formed because there are many a's matching a's or English words matching English words. Similarly, the square in the lower right is probably formed because there are many b's matching b's or French words matching French words. The diagonals indicate how the English text should be aligned with the French. There is a good chance of a dot contributing to the diagonal when the two texts are so aligned because there are a fair number of proper nouns, dates, times, numbers, etc. that will match when text is compared with its translation.

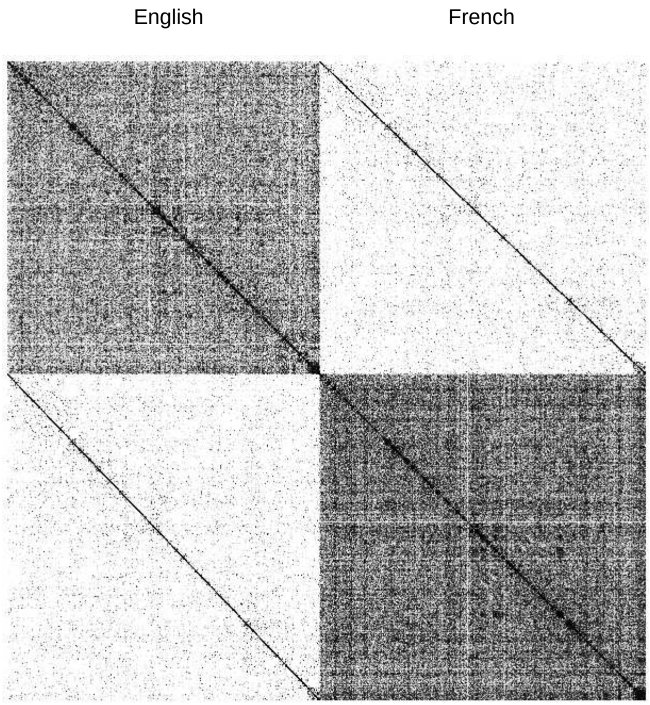

Fig. 5 Three Years of Hansards (37 Million Words)

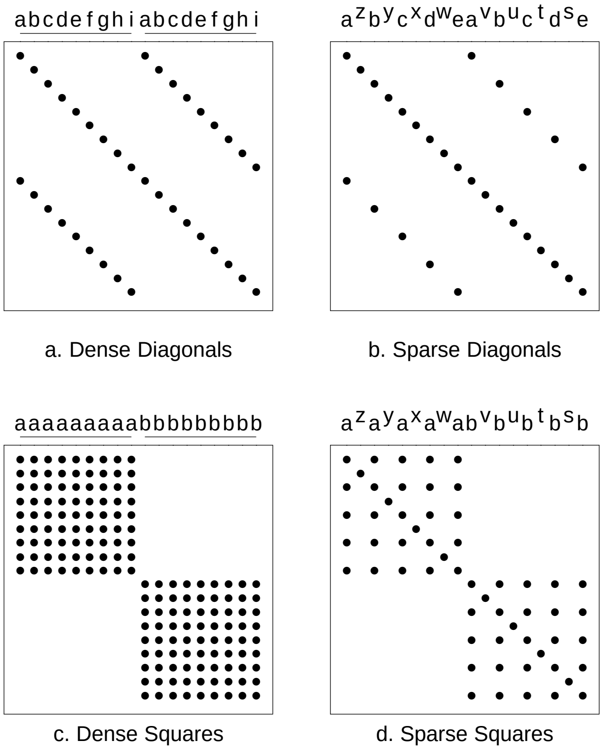

Fig. 6. Dense vs. Sparse Features

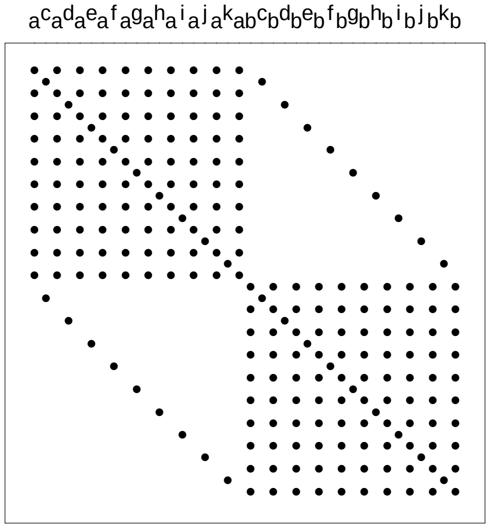

Fig. 7. Combination of Diagonals and Squares

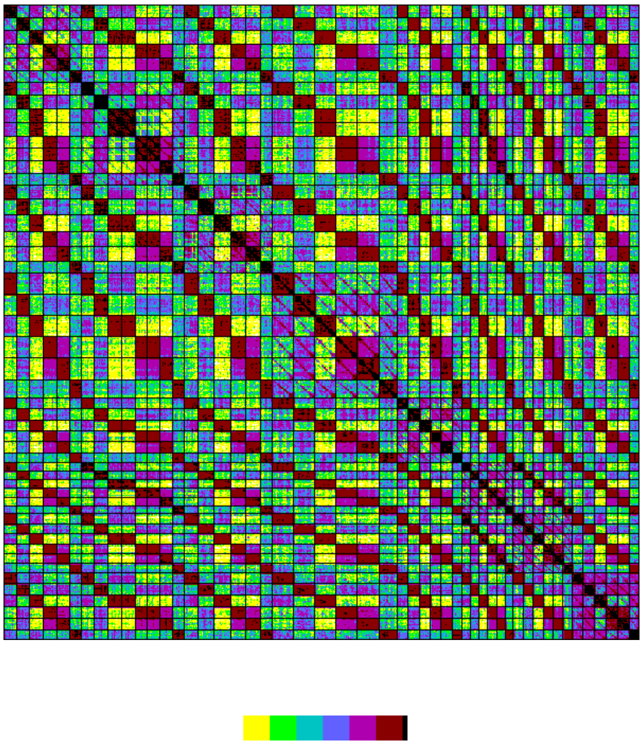

Fig. 8 Six Chapters of Microsoft Manuals in Seven Languages (3.3 Million Words) with Colormap

The combination of diagonals and squares is also apparent in Fig. 8, a dotplot used by translators at AT&T Language Line Services to help maintain six Chapters of Microsoft manuals in seven languages (Dutch, French, German, Italian, Spanish, Swedish). When a file is compared with its translation, there is a good chance of finding a diagonal, whereas when a file is compared with another file in the same language, there is a good chance of finding a dark square.

With seven languages instead of just two, there is an opportunity to see matches between similar languages. Color makes it easier to see these matches. The relative number of matches per pixel is assigned to a cell in a colormap using a *histogram equalization* technique described in Section 5.3. Fig. 8 uses a colormap (shown immediately below Fig. 8) in which groups of consecutive color cells all map to the same color (e.g. cells 1-16 are yellow, 17-32 light green, 33-48 blue, 49-64 medium blue, 65-80 purple, 81-96 dark red, and the rest black). Matches between two files in the same language show up in dark red (a relatively high level), matches between two files in very different languages (e.g., Spanish and German) show up in yellow (a relatively low level), and matches between two files in similar languages (e.g., Spanish and Italian) show up in an intermediate color between yellow and dark red (e.g., medium-blue, purple). Note that the ninth and eleventh files are copies of the eighth and tenth as shown by dark red boxes with black diagonals.

Of all the dotplots in this paper, Fig. 5 has the highest *data-ink ratio*. (Tufte 1983, p.93). 37 million words is at least four orders of magnitude more than could be seen with a conventional text editor. The tremendous compression factor of Fig. 5 increases the apparent density of features that are actually quite sparse.

To summarize, we have seen three examples of text dotplots, AP news, Hansards, and Microsoft Manuals, and two types of features, diagonals and squares. Diagonals indicate regions of ordered similarity (e.g. matches, alignments, and translations), while squares indicate regions of unordered similarity (e.g. documents in similar languages). We have also seen how features can be preserved despite changes in density, and how this property allows us to interleave combinations of sparse features. These ideas will be further developed in the next section which introduces the source code application.

4. Dotplots of Source Code

4.1 Diagonals in Source Code

The source code examples in this paper are taken from the 5ESS® switch, a large program that handles much of the world's long distance telephone service. For source code applications, we usually choose to tokenize the input into lines of code.

Fig. 9 shows a dotplot of eight source code files labeled A-H. Two features are of interest: (1) a long broken diagonal starting at AE and extending down to DH (and, by symmetry, another long broken diagonal starting at EA and extending down to HD), and (2) 56 short diagonals, each starting in the upper left corner of a different grid box.

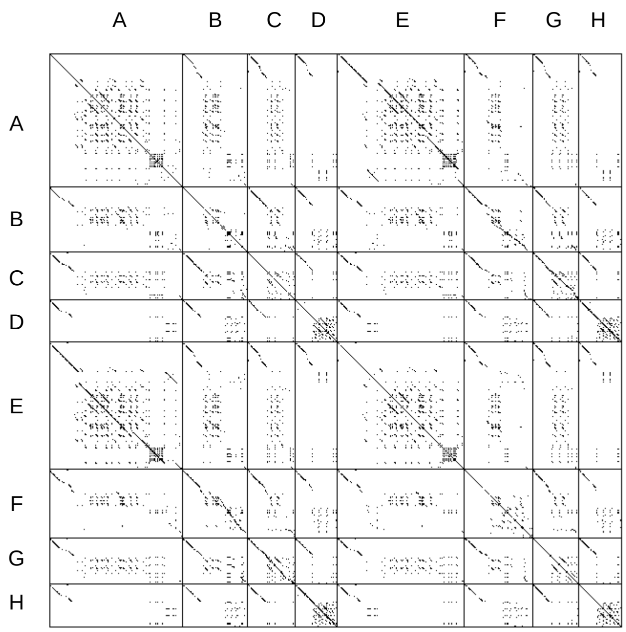

Fig. 9. 3000 Lines of Code

What could cause these features? The first feature, the long broken diagonals, indicates that files A, B, C, and D are similar to files E, F, G, and H, respectively. This observation is further supported by the striking pattern in the names of the files, as shown below. Perhaps these files were copied to maintain a parallel version of the software.

First 4 Files		Second 4 Files	
A	P.ISqf3.c	E	P.ISqf4.c
B	P.ISqf3_hold.c	F	P.ISqf4_hold.c
C	P.ISqf3_hr.c	G	P.ISqf4_hr.c
D	P.ISqf3_rr.c	H	P.ISqf4_rr.c

The second feature, the 56 short diagonals, has a different explanation. Each of the eight files starts with a highly structured comment of the form:

```
/*
 * File:    ...
 *
 * Data:    ...
 *
 * Name:    ...
```

The comments also include a number of additional fields: `Abstract`, `Loadable Package`, `Usage`, `Parameters`, `Externals`, etc. The 56 short diagonals are caused by similarities in the eight comments.

4.2 Textures in Source Code

A relatively small number of comments in Fig. 9 generate a relatively large number of diagonals. In general, n copies of a subsequence generate $n(n-1)$ diagonals. Consequently, even a relatively small number of copies will generate such a large number of diagonals that they form a texture. A good example can be found in Fig. 1a (near the label D), shown in more detail in Fig. 10.

This texture consists of a large number of diagonals of varying lengths. Consider the upper left corner where diagonals are shrinking. Fig. 11 shows, by synthesis, that diagonals shrink as a repeating subsequence is diluted with increasing numbers of non-repeating tokens.

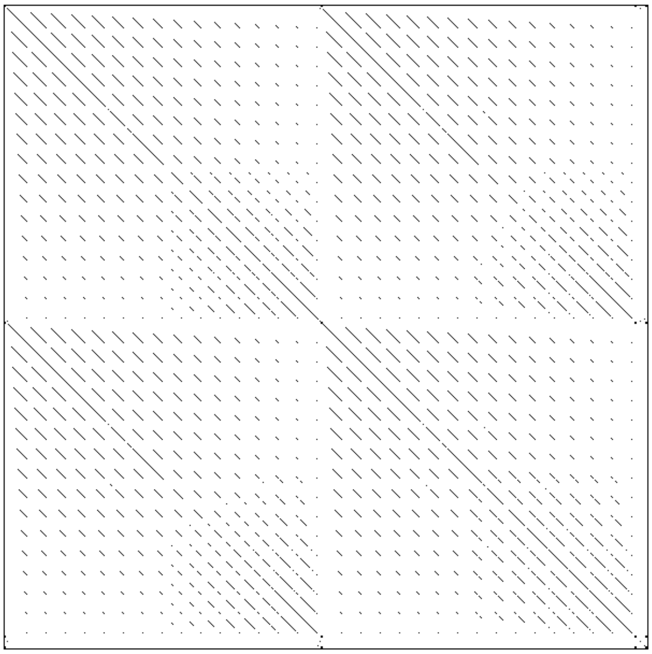

Fig. 10. 600 Lines of Code (Detail of Fig. 1)

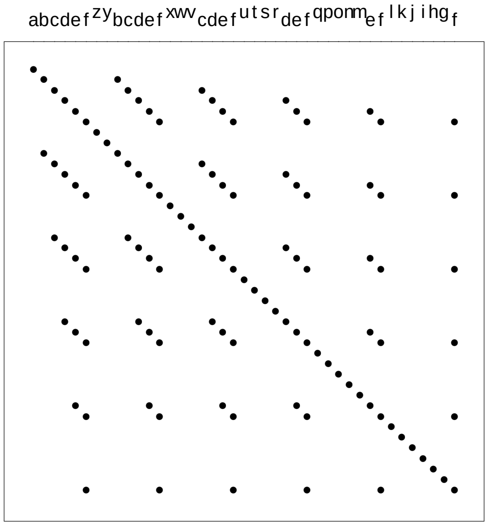

Fig. 11. Shrinking Diagonals

How does the artificial sequence in Fig. 11 relate to the actual input sequence for Fig. 10? The texture in Fig. 10 is generated by two clauses of an *if* statement. Each clause consists of 16 groups of 18 lines of code. The first group initializes all but the first field of a structure to 0; the second initializes all but the first two fields to 0; the third initializes all but the first three fields to 0; etc. The repeating sequence of initializations to 0 is diluted with increasing numbers of non-zero initializations resulting in the ‘‘shrinking diagonals’’ texture. There is also another pattern in the code that causes diagonals to grow longer toward the lower right corner of the texture.

This example shows that dotplots highlight a level of structure that would have been very difficult to discover using traditional tools, such as a screen editor, because the pattern extends over several hundred lines of code, much more than could possibly fit on a screen. In addition, this structure would also be difficult to appreciate with a dynamic programming approach such as the UNIX™ `diff` program. Such programs attempt to find a single match, and are therefore unable, in principle, to find the rich texture of multiple overlapping matches. In addition, the `diff` program would have trouble in this case because many of the matches aren't exact. To handle cases like this, Baker (1992) introduced an inexact matching criterion, *parameterized match*, which overcomes this difficulty by equating two lines that are the same up to the names of the parameters and the values of the constants. But unfortunately this equivalence relation would also miss the pattern of ‘‘shrinking diagonals,’’ because it depends crucially on the names of the parameters and the values of the constants.

However, in other cases, equivalence relations have proved to be extremely powerful. Consider, Fig. 12, for example, where it appears that several large sections of code were copied verbatim (white space and all), as evidenced by the long diagonals. Suppose we wanted to understand more about the copied code: Who copied it? When? Why?

4.3 Attributes

Author Attributes

Author	Code
carlson	`/*`
carlson	` * Name:  RTgeninit`
kedzierski	` *`
veach	`#feature ( 5E2_2G )`
martin	` * Module:  RTmain`
martin	` *`
ahmad	`#endfeature ( 5E2_2G )`

One approach to answering these kinds of questions makes use of an equivalence relation we call *attributes*. Large software development projects typically maintain a database that associates each line of code with various attributes such as the author's name, modification dates, etc. The table above illustrates the author attributes for the first few lines of code that were used to generate Fig. 12.

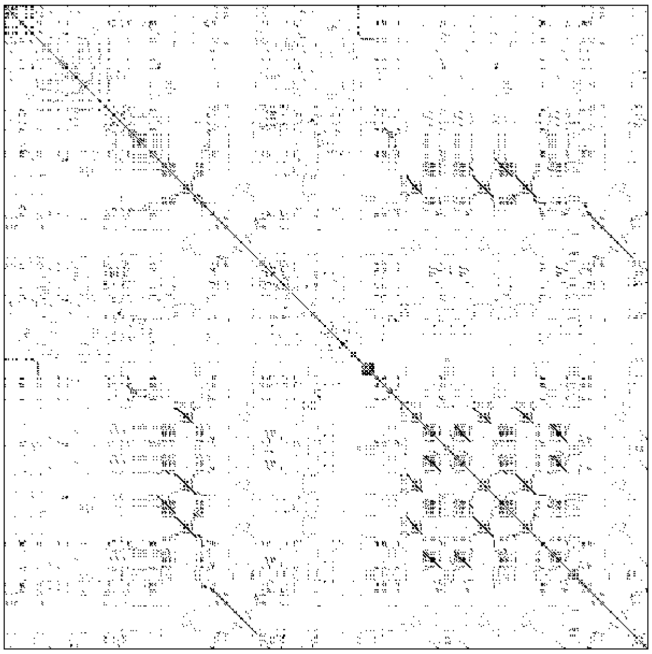

Fig. 12. 3400 Lines of Code

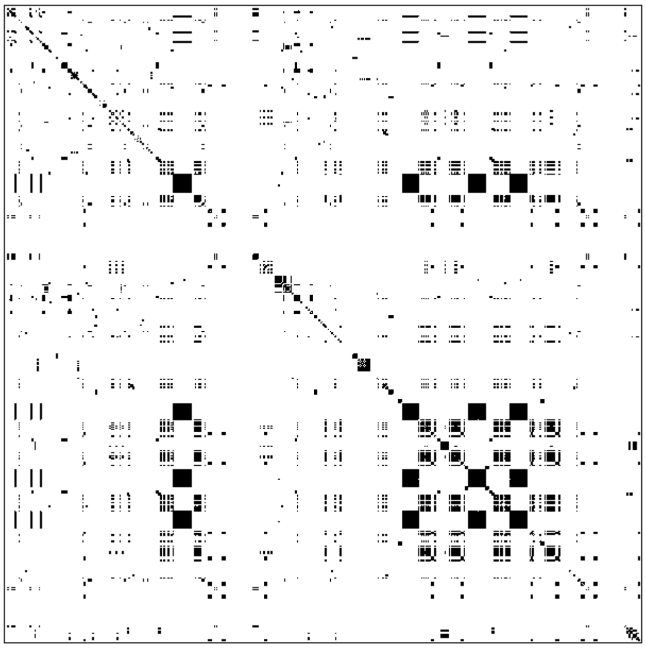

Fig. 13. 3400 Author Attributes

A dotplot can be generated from any input. Fig. 13, for example, is just like Fig. 12 except that it uses the author attributes (column 1) as input instead of the code (column 2). It is interesting to compare Figs. 12 and 13 in order to see if there might be a pattern between the code and the authors. In this case, at least, it appears that many of the diagonals in Fig. 12 coincide with squares in Fig. 13. Why? We suspect that the copies were created by the same author, at the same time, and for the same reason. This conjecture can be further tested by looking at a dotplot of modification dates, and seeing if the features in that dotplot line up with those in Figs. 12 and 13.

4.4 Summary

In summary, we have explored the properties of a variety of features, primarily diagonals, but also squares and textures (Fig. 2). We have seen that features can appear in many variations: they can be broken (Fig. 2c), dense or sparse (Fig. 6), and they can appear in various combinations (Fig. 7). We have also seen applications of dotplots in biology (Fig. 3), as well as the two new applications: text (Fig. 4-8) and source code (Fig. 9-13). Both analysis and synthesis have been used to learn more about the relationship between features and corresponding patterns in input sequences.

Dotplots may have a number of practical ramifications for source code applications. First, dotplots might be useful for identifying large structures in a program, especially during *discovery*, the process of reading code for the first time. Secondly, dotplots might help developers find undesirable duplication so that it can be removed. In some cases, for example, it is possible to replace multiple copies with a single subroutine, as suggested in Baker (1992). In other cases, the ability to identify multiple copies can be useful for maintenance. In particular, if a bug is found in one of the copies, then there is a good chance that the others might require attention, as well. Thus, dotplots appear to be useful for identifying large structures, removing undesirable duplication when possible, and coping more effectively with duplication that cannot be removed.

Some users might believe that redundancy is always indicative of a weakness of some kind. For example, one user has started using the browser to identify C constructions, such as `switch` statements, which are often associated with a texture generated by repeated `break` statements. In this way, dotplots have been used to help design a new programming language that avoids many of these ‘‘wordy’’ constructions.

Should redundancy be considered ‘‘harmful’’? Following a policy like Dijkstra’s stand on `goto`s, one might suggest that redundancy should be eliminated, whenever possible (Dijkstra 1968). Unfortunately, such a policy would also remove a number of very useful structures such as the structured comments in Fig. 9. Carried to its logical extreme, such a policy would reduce a structured program to a random string, a string whose shortest description is itself. As in good writing, repetition can be a powerful rhetorical device for conveying emphasis, parallelism, etc. It would be a mistake to discourage such practices in a futile attempt to eliminate ‘‘wordiness’’ and other forms of ‘‘bad’’ writing.

5. Software Design

The next three subsections describe the implementation of the browser. First, the input data is tokenized into a sequence of N tokens. Secondly, this sequence is used to construct the *f-image*, an array of floating point values. Finally, these values are quantized into the *q-image*, an array which is suitable for displaying on a color or greyscale monitor.

$$\text{tokens} \rightarrow \text{f-image} \rightarrow \text{q-image}$$

5.1 Tokenization

The program begins by tokenizing the input and applying the appropriate equivalence relations, if any. Equivalence relations were discussed briefly in sections 4.2 and 4.3; they can be used to remove white space, simulate a parameterized match, replace a token with one of its attributes, etc. The details of the tokenizer depend on the particular application. In the text application, for example, we have tended to tokenize the input text into words, whereas in the source code application, we have tended to tokenize the input code into lines.

Before discussing the next topic, the calculation of the f-image, it might be worthwhile to clarify a potential source of confusion between the terms *type* and *token*. Consider, for example, the English phrase, ‘‘to be or not to be,’’ which contains 6 words, but only 4 of them are distinct. We say that the sentence contains 6 tokens, but only 4 types. By convention, we denote the number of tokens in the input data with the variable N, and we denote the number of types in the input data with the variable V (for ‘‘vocabulary size’’).

One might normally choose to represent types as strings. That is, it would be natural to represent the word, *to*, as the string "to", and the line of code, ‘‘`for(i=1; i<N; i++)`’’, as the string "for(i=1; i<N; i++)". For computational convenience, we have decided not to represent types as strings, but rather as contiguous integers in the range of 0 to $V-1$. The strings are converted to numbers using standard hashing techniques. Representing tokens as integers has several advantages. In particular, it makes it easy to test whether or not the type of the i^{th} token is the same as the type of the j^{th} token: `if(tokens[i] == tokens[j])`. If we had used strings instead of integers, then we would have had to use `strcmp` instead of `==`, which would have been much less efficient.

5.2 Computing the F-image

After the input data has been parsed into a sequence of tokens, the token are then converted into a floating point image, the f-image. In the simplest case, this is accomplished by placing a dot in `fimage[i][j]` if the type of the i^{th} token is the same as the type of the j^{th} token. In other words:

```
float fimage[N][N];

for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    if(tokens[i] == tokens[j])
      fimage[i][j] = 1;
    else fimage[i][j] = 0;
```

This N^2 algorithm can be improved by making use of three observations: (1) tokens should be weighted to adjust for the fact that some matches are more surprising than others, (2) dotplots with large values of N may consume too much space, and (3) dotplots with large values of N may also consume too much time. We address these issues in the following three subsections: (1) *Weighting*, (2) *Compression*, and (3) *Approximation*.

5.2.1 Weighting

The N^2 algorithm can be improved by replacing ‘‘`fimage[i][j] = 1;`’’ with ‘‘`fimage[i][j] = weight(tokens[i]);`’’, where the function `weight` returns a value between 0 and 1, depending on how surprising it is to find that `tokens[i] == tokens[j]`. There are quite a number of reasonable functions to use for `weight`. The weighting concept is illustrated below, using the natural suggestion of weighting each match inversely by the frequency of the type. In this way, frequent types (e.g., the English word *the* or the line of C-code ‘‘`}`’’) do not contribute very much to the f-image because matches among such frequent types are not very surprising.

```
/* Initialize freq */
float freq[V] = {0};
for(i=0; i<N; i++)
  freq[tokens[i]]++;

for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    if(tokens[i] == tokens[j])
      fimage[i][j] = 1/freq[tokens[i]];
    else fimage[i][j] = 0;
```

5.2.2 Compression

If N is large, it becomes impractical to allocate N^2 storage, and therefore it becomes necessary to compress the image in some way. Suppose that we wanted to compress the f-image from N by N, down to n by n, for some $n << N$. Then we could simply aggregate values that fall into the same n by n cell as shown below. Of course, it is recommended that the signal be filtered appropriately before compression in order to avoid aliasing (Gonzalez & Wintz 1987, p.94). Filtering may also be useful if there are too many dots in the f-image as is well known in the biology application (Maizel & Lenk 1981, Pustell & Kafatos 1982). In general, various well-known signal processing techniques might be useful for enhancing features of interest.

```
/* Initialize f-image */
float fimage[n][n] = {0};
/* Map x from token coordinates
   into f-image coordinates */
#define CELL(x) (((x) * n) / N)

for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    if(tokens[i] == tokens[j])
      fimage[CELL(i)][CELL(j)] +=
        weight(tokens[i]);
```

5.2.3 Approximation

In practice, if N is very large, it becomes impractical to perform the N^2 comparisons and it is therefore useful to introduce an approximation. Recall the dotplot of the Canadian Hansards shown in Fig. 5. If we had tried to compute this figure with the N^2 algorithm, the calculation would have required $37{,}000{,}000^2$ steps, which is utterly impractical. Even if each step took only a micro-second, the N^2 algorithm would require more than 40 years.

Before presenting the approximation, it is convenient to introduce the concept of a *posting*, a precomputed data structure that indicates where a particular type can be found in the input sequence. Thus, for the input sequence, ''to be or not to be,'' there are two postings for the type ''to'': one at position 0 and the other at position 4. One can compute the dots for the type ''to'' in this example by placing a dot in positions: (0, 0), (0, 4), (4, 0), and (4, 4). In general, for a word with frequency f, there are f^2 combinations of postings that need to be considered. The algorithm below simply iterates through all f^2 combinations for each of the V types in the vocabulary.

```
for(type=0; type<V; type++) {
  w = weight(type);
  f = freq[type];
  postings = get_postings(type);
  for(p1=0; p1 < f; p1++) {
    i = postings[p1];
    for(p2=0; p2 < f; p2++) {
      j = postings[p2];
      fimage[CELL(i)][CELL(j)] += w;}}}
```

We now come to the key approximation. If we assume that types with large frequencies ($f \geq T$, for some threshold T) have vanishingly small weights, then we don't need to iterate over their postings. This approximation produces significant savings since it allows us to ignore just those types with large numbers of postings. In fact, the resulting computation takes less than $V\ T^2$ iterations.

```
for(type=0; type<V; type++) {
  w = weight(type);
  f = freq[type];
  /* the key approximation */
  if(f < T) {
    postings = get_postings(type);
    for(p1=0; p1 < f; p1++) {
      i = postings[p1];
      for(p2=0; p2 < f; p2++) {
        j = postings[p2];
        fimage[CELL(i)][CELL(j)] += w;}}}}
```

In practice, we have found that T can often be set quite small. The Hansard dotplot shown in Fig. 5, for example, was computed with $T = 20$, so that the entire calculation took less than $400V \approx 52{,}000{,}000$ steps and completed in only 25 seconds of real time on a Silicon Graphics Personal Iris workstation 4D/35.

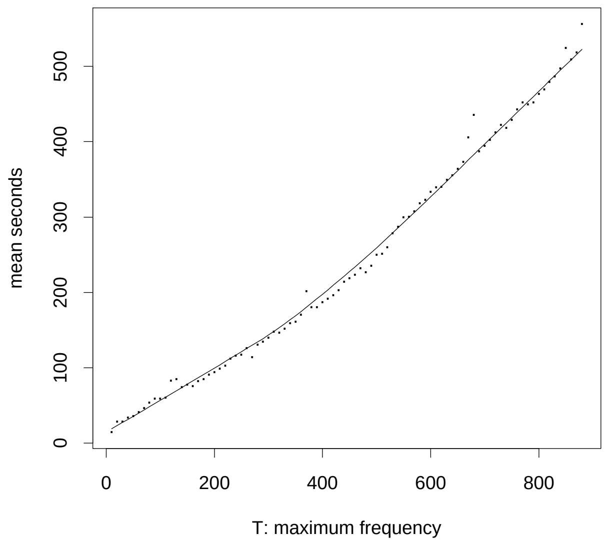

Fig. 14. Computing Hansard Plot with Different Values of T

How much does the choice of T affect the calculation time? Fig. 14 attempts to address this question for the Hansard data by plotting elapsed time as a function of T. Each point in Fig. 14 shows a mean of 10 trials for each value of T. The line shows a *lowess* smooth (Cleveland 1979) of the plotted points. Over this range, it appears that increasing the threshold by 1 increases the computation time by less than a second; dotplots with T's up to a few hundred can be computed in a few minutes of real time.

5.3 Computing the Q-image

After computing the f-image, the floating point values are quantized to conform to the available display hardware. Suppose, for example, that the hardware is designed to handle at most C colors, where $C \approx 256$. An obvious quantization technique is linear interpolation. Unfortunately, we have found that the values in the f-image often belong to an extremely skewed distribution, as shown in Fig. 15. Using linear interpolation on such a highly skewed distribution would introduce serious quantization errors.

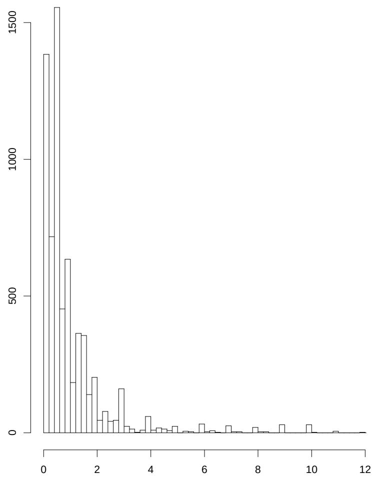

Fig. 15. Histogram of Values in Fig. 9's F-image

We have had more success with a non-parametric approach: histogram equalization (Gonzalez & Wintz 1987, pp. 146-152), which quantizes the values in the f-image into C quantiles, one for each color. Unfortunately, even histogram equalization has difficulties when the input is highly quantized. We have found empirically that many of the f-image values are small integers and ratios of small integers. This might be expected in the text application where Zipf's Law would predict most word frequencies to be small integers; it also appears to hold in the other applications, as well. In order to avoid assigning multiple colors to the same integer, we have found it useful to remove duplicate values before applying histogram equalization.

5.4 User Interface

Finally, the q-image is converted into an image suitable for displaying in a window as a component of the interactive dotplot browser (see Fig. 1). In a color X Windows implementation (Scheifler & Gettys 1986), this final step establishes a mapping from values in the q-image to cells in the X server's default colormap. We use a separate program that maintains these color cells and allows switching between various, pre-defined colormaps. The colormap typically used for black and white dotplots is *binary*: color 0 is white (indicating relatively few matches), while all other colors are black (otherwise).

Alternatively, grey-scale colormaps that vary smoothly (i.e. with perceptually even differences) from white to black, indicate the total number of weighted matches per pixel. Darker pixels indicate the locations of more interesting matches, while lighter pixels indicate less interesting matches. Grey-scale is particularly useful for very dense dotplots, which may appear completely black with a binary colormap.

One problem with grey-scale is that it is difficult to distinguish between adjacent values in the colormap. We have found that by carefully adding color to grey-scale colormaps (e.g. a range from white, through yellow and orange, to dark red), we can enhance the contrast between adjacent cells. This makes it easier to assess the relative number of weighted matches at different locations within a dotplot. We have also found that this effect is enhanced by limiting the number of colors (see Fig. 8).

In a monochrome implementation, the q-image step is unnecessary since the f-image can be converted directly into black and white using various standard techniques such as thresholding, dithering, error diffusion, etc.

In addition to the dotplot views discussed thus far, there are also text views, as shown in Fig. 1c. A text view consists of two panes so that two subsequences of the input can be presented side-by-side. The text view is linked to a dotplot view, so that clicking the mouse on a point in the dotplot corresponding to the pair of tokens x,y causes the left pane to be centered around x and the right pane to be centered around y.

6. Conclusion

Dotplots, which have been used to study homology in biology, are also useful for discovering potentially important patterns in text and source code. In the software application, for example, we have seen that dotplots can be used to discover large-scale structures, remove undesirable duplication when possible, and cope more effectively with duplications that cannot be removed. Similarly, there are also a number of practical ramifications of these patterns in the text application.

We have seen that many of these potentially important patterns are often associated with certain features in the dotplot: diagonals, squares, textures and combinations thereof. There was a considerable discussion of a number of mechanisms that explain some of these associations. Much of the discussion used the browser to analyze a feature in a real sequence, and then tried to replicate the feature in a synthesized dotplot. For example, the browser was used to find broken diagonals in AP stories (Fig. 4), a combination of squares and diagonals in the Hansards (Fig. 5), and ''shrinking diagonals'' in a large program (Fig. 10). Each of these features were replicated in a synthesized dotplot: broken diagonals in Fig. 2c, the combination of squares and diagonals in Figs. 6 and 7, and ‘‘shrinking diagonals’’ in Fig. 11. The discussion then concluded with a speculation of the underlying mechanism. In the AP news, for example, the diagonals were probably broken by the insertion of a few extra facts into a rewrite. Similarly, the shrinking diagonals in the software example were probably caused by a repeating sequence of initializations to 0 being diluted with increasing numbers of non-zero initializations.

In many cases, the patterns are much easier to find with a dotplot than with an alternative such as a text editor or the UNIX `diff` program. A text editor, for example, is ill-suited for identifying structures that extend well beyond the size of the screen. Similarly, the `diff` program is ill-suited for identifying a texture such as the ‘‘shrinking diagonals’’ pattern discussed in Section 4.2, because the `diff` program attempts to find a single alignment path and therefore can’t deal effectively with the rich structure of multiple overlapping matches.

The final section of the paper described the implementation of dotplots, with an emphasis on weighting, compression and approximation. These steps make it possible compute dotplots quickly enough for use in an interactive browser.

Acknowledgements

We would like to thank Joe Kruskal for suggesting the connections with the biology literature, Mike Zuker for a number of extremely helpful references, Justina Voulgaris for access to the pBR322 and pUC18 data, and Eric Sumner and Joe Steffen for access to the 5ESS code and an appreciation of the realities of programming in the large. The X implementation benefited considerably from Doug Blewett’s invaluable knowledge. We also appreciate the enthusiastic response from early users of the browser, especially Chris Ramming. William Eddy and an anonymous reviewer contributed several helpful comments to an earlier version of this paper, which is to appear in the Proceedings of the INTERFACE-92 Graphics and Visualization Symposium.

REFERENCES

Argos, P. 1987. A Sensitive Procedure to Compare Amino Acid Sequences. Journal of Molecular Biology 193.385-396.

Baker, B. S. March 1992. A Program for Identifying Duplicated Code. INTERFACE ’92. Interface Foundation of North America.

Balbas, P.; X. Soberon; E. Merino; M. Zurita; H. Lomeli; F. Valle; N. Flores; and F. Bolivar. 1986. Plasmid Vector pBR322 and its Special-Purpose Derivatives — A Review. Gene 50.3-40.

Blundell, T. L.; B. L. Sibanda; M. J. E. Sternberg; and J. M. Thornton. March, 1987. Knowledge-Based Prediction of Protein Structures and the Design of Novel Molecules. Nature 326:26.347-352.

Carrington, J. C., and T. J. Morris. 1987. Structure and Assembly of Turnip Crinkle Virus IV. Analysis of the Coat Protein Gene and Implications of the Subunit Primary Structure. Journal of Molecular Biology 194.265-276.

Cleveland, W. S. December 1979. Robust Locally Weighted Regression and Smoothing Scatterplots. JASA 74:368.829-836.

Dijkstra, E. W. March 1968. Go to Statement Considered Harmful. Communications of the ACM 11:3.147-148.

Doolittle, R. F. October 1981. Similar Amino Acid Sequences: Chance or Common Ancestry?. Science 214:9.149-159.

Eick, S. G. March 1992. Dynamic Graphics for Software Visualization. INTERFACE ’92. Interface Foundation of North America.

Gonzalez, R. C., and P. Wintz. 1987. Digital Image Processing. Addison-Wesley. Second Edition.

Lake, J. A.; V. F. de la Cruz; P. C. G. Ferreira; C. Morel; and L. Simpson. July 1988. Evolution of Parasitism: Kinetoplastid Protozoan History Reconstructed from Mitochondrial rRNA Gene Sequences. Proceedings of the National Academy of Science, USA 85.4779-4783. Evolution.

Laver, W. G.; G. M. Air; T. A. Dopheide; and C. W. Ward. January 1980. Amino Acid Sequence Changes in the Haemagglutinin of A/Hong Kong (H3N2) Influenza Virus During the Period 1968-77. Nature 283:31.454-457.

Maizel, J. V., and R. P. Lenk. December 1981. Enhanced Graphic Matrix Analysis of Nucleic Acid and Protein Sequences. Proceedings of the National Academy of Science, USA 78:12.7665-7669. Genetics.

Pustell, J., and Fotis C. Kafatos. 1982. A High Speed, High Capacity Homology Matrix: Zooming Through SV40 and Polyoma. Nucleic Acids Research 10:15.4765-4782.

Quax-Jeuken, Y. E. F. M.; W. J. Quax; and H. Bloemendal. June 1983. Primary and Secondary Structure of Hamster Vimentin Predicted from Nucleotide Sequence. Proceedings of the National Academy of Science, USA 80.3548-3552. Biochemistry.

Salton, G. 1989. Automatic Text Processing. Addison-Wesley.

Scheifler, R. W., and J. Gettys. April, 1986. The X Window System. ACM Transactions on Graphics 5:2.79-109.

Tufte, E. R. 1983. The Visual Display of Quantitative Information. Graphics Press.

Tukey, J. W. 1977. Exploratory Data Analysis. Addison-Wesley.

Vingron, M. 1991. Multiple Sequence Alignment and Applications in Molecular Biology. Heidelberg University dissertation.

Yanisch-Perron, C.; J. Vierira; and J. Messing. 1985. Improved M13 Phage Cloning Vectors and Host Strains: Nucleotide Sequences of the M13mp18 and pUC19 Vectors. Gene 33.103-119.