Models and Techniques for the Visualization of Labeled Discrete Objects *

Dinesh P. Mehta †‡ Sartaj Sahni†

Abstract

A general technique for the visualization of discrete objects is presented. This technique which consists of identifying similar substructures within the structure and color coding them is demonstrated by applying it to linear strings, circular strings, binary trees, and series-parallel graphs.

The problem of *display conflicts* which is encountered while applying this technique is described and methods to deal with it are suggested. Some of these methods are interactive in nature. Queries that would be supported in such an environment are described. Efficient algorithms to implement some of the queries are developed. The performance of these algorithms is studied, both theoretically and experimentally. We also demonstrate that our algorithms are well-suited for implementation using the object oriented paradigm.

The application of our techniques to such diverse areas as molecular biology, text analysis, analysis of numerical sequences, computer vision, computer graphics, computer-aided design, data compression, algorithm animation, and debuggers is outlined.

*This research was supported in part by the National Science Foundation under grant MIP 86-17374.

†Dept. of Computer and Information Sciences, University of Florida, Gainesville, FL 32611

‡Dept. Computer Science, University of Minnesota, Minneapolis, MN 55455

1 Introduction

The objective of visualization is to extract useful and relevant information from data and represent it so that it can be easily understood and assimilated by humans. This enables specialists in application areas to observe trends and patterns in the data. This could lead to better understanding of various phenomena and provide insights resulting in theories or hypotheses, which can subsequently be proved (or disproved) by formal methods or by further experimentation.

A notion that is useful in the understanding of objects is that of *similarity*. Multiple occurrences of the same pattern in data which represents the outcome or result of some process indicates the presence of many instances of the same "effect". Analyzing the set of circumstances associated with these occurrences could yield a plausible "cause". For example, multiple occurrences of the same flaw in a paper roll might reveal the faulty component in the paper production process.

Similarly, multiple occurrences of the same patterns in an object whose structure is being studied indicates the presence of many instances of the same "cause". Observing the phenomena which occur in the presence of these patterns could shed some light on the "effect" of the patterns. For example, the presence of multiple occurrences of patterns in DNA strands in organisms could manifest themselves as common characteristics shared by the organisms.

This linkage of cause and effect is a fundamental goal in many scientific disciplines. Most work in visualization so far, which attempts to facilitate the scientific goals outlined above, consists of choosing methods to display *individual units of data*, so that patterns and trends become visually obvious when the data is seen in its entirety [1, 2]. However, the onus of detecting patterns and trends still lies on the user or the specialist. This becomes more crucial when the amount of data is large and the user's perceptual faculties are overburdened. Consequently, errors of omission (not seeing patterns which are actually there) become more likely.

Our work attempts to shift the responsibility of detecting patterns from the human to the computer. This is done by devising algorithms to detect patterns in the data; and then making these patterns available to the user for further scrutiny. Multiple occurrences of the same pattern can be made visually explicit by color coding them with the same color. Other methods could also be used, such as "flashing" occurrences of the same pattern on the screen. If visual schemes are

not appropriate, recurring patterns could simply be provided as a list of occurrences which the user goes through. In the context of a visualization environment, this technique could be used either in a stand-alone manner or as a supplement to other visualization methods.

For the technique outlined above to be useful, the following principles on displaying patterns should be adhered to:

Principle 1: Two patterns should be displayed to *look* similar iff they *are* similar.

Principle 2: The *degree of similarity* between the displays of two patterns should be proportional to the *actual similarity* of the patterns.

A complete specification of a visualization problem requires one to provide the following five items. These are illustrated using linear string visualization as an example. Henceforth, the word "string" shall refer to linear strings.

(1) Structure of the Data to be Visualized: Does the data represent a string, a series-parallel graph, a binary tree etc.

In the string visualization example, the data is a string, S of length n, whose characters are chosen from a fixed alphabet, Σ, of constant size.

(2) Structure of Patterns: This depends largely on the structure of the data. If the data represents a string, then the structure of the pattern could be a substring (contiguous sequence of string elements), a subsequence (a non-contiguous sequence of string elements), etc. Patterns can have other constraints imposed upon them. For example, a pattern may be required to be of a minimum size.

In the string visualization example, the pattern is a substring of S, defined uniquely by its start and end positions.

(3) Maximality of Patterns: If a pattern is repeated in the data, then any subpattern of the pattern is also repeated. For example, if abc repeats in a string, so do ab, bc, a, b, and c . In particular, if all occurrences of ab occur in the context of abc, then attempting to distinguish between ab and abc does not serve any useful purpose. Defining maximality and restricting the users attention to maximal patterns helps to simplify the display.

In the string visualization example, a pattern is said to be maximal iff all its occurrences are

Figure 1: Highlighting Displayable Entities

not all preceded by the same letter, nor all followed by the same letter. Consider the string $S =$ $abczdefydefxabc$. Here, abc and def are the only maximal patterns. The occurrences of def are preceded by different letters (z and y) and followed by different letters (y and x). The occurrences of abc are not preceded by the same letter (the first occurrence does not have a predecessor) nor followed by the same letter. However, de is not maximal because all its occurrences in S are followed by f.

(4) Measure of Similarity (MS): A measure of similarity would consist of attaching numerical values to pairs of maximal patterns which indicate the degree of similarity between the two patterns.

In the string visualization example, the measure of similarity could be defined as follows: If two patterns are identical, then **MS** = 1. Otherwise, **MS** = 0. In other words, two patterns are defined to be similar iff they are identical. There is no concept of "degree of similarity" in this definition.

(5) Display Models: This addresses the issues of which patterns are displayed and how they are displayed. While choosing display models, Principles 1 and 2 should be kept in mind.

In the string visualization example, a pattern is said to be a *displayable entity* (or displayable) iff it is maximal and occurs more than once in S (in this case, all maximal patterns are displayable entities with the exception of S, which occurs once in itself) . All instances of the same displayable entity are highlighted in the same color. Instances of different displayable entities are highlighted in different colors (there is no relationship between colors representing different displayable entities). In the example string, S, abc and def are the only displayable entities. So, S would be displayed by highlighting abc in one color and def in another as shown in Figure 1.

Visualization conflicts that arise from the above technique are described in Section 2 and refinements to the display model that attempt to overcome these conflicts are provided in Section 3. In Section 4 some of the queries supported by a string visualization system are stated. Applications of string visualization are discussed in Section 5 and some of the issues that arise when one implements a string visualization system are addressed in Section 6. Some empirical results are also provided

in this section. Sections 7, 8, 9 discuss circular string, tree, and geometric series parallel graph visualization respectively. Section 10 demonstrates that many of our algorithms can be effectively coded using the object oriented paradigm. Finally, in Section 11, we provide an outline of how a visualization system based on our approach would function.

2 Visualization Conflicts

Consider the string $S = abcicdefcdegabchabcde$ and its displayable entities, abc and cde (both are maximal and occur thrice). So, they must be highlighted in different colors. Notice, however, that abc and cde both occur in the string $abcde$, which occurs as a suffix of S. Clearly, both displayable entities cannot be highlighted in different colors in $abcde$ as required by the model. This is a consequence of the fact that the letter c occurs in both displayable entities. This situation is known as a prefix-suffix conflict (because a prefix of one displayable entity is a suffix of the other).

Note, also, that c is a displayable entity in S. Consequently, all occurrences of c must be highlighted in a color different from those used for abc and cde. But this is impossible as c is a subword of both abc and cde. This situation is referred to as a subword conflict. Formally,
(i) A *subword conflict* between two displayable entities, D_1 and D_2, in S exists iff D_1 is a substring of D_2.
(ii) A *prefix-suffix conflict* between two displayable entities, D_1 and D_2, in S exists iff there exist substrings, S_p, S_m, S_s in S such that $S_pS_mS_s$ occurs in S and $S_pS_m = D_1$ and $S_mS_s = D_2$.

3 Refinements of Display Model

When subword and prefix-suffix conflicts occur, we need some criteria to determine which of the information previously required to be displayed actually gets displayed. For instance, in the example string $S = abcicdefcdegabchabcde$ from the previous section, three possible non-conflicting, displayable subsets are shown in Figure 2. In this section we present three refinements to the display model from Section 1 which attempt to overcome the display difficulties created by conflicts. They are

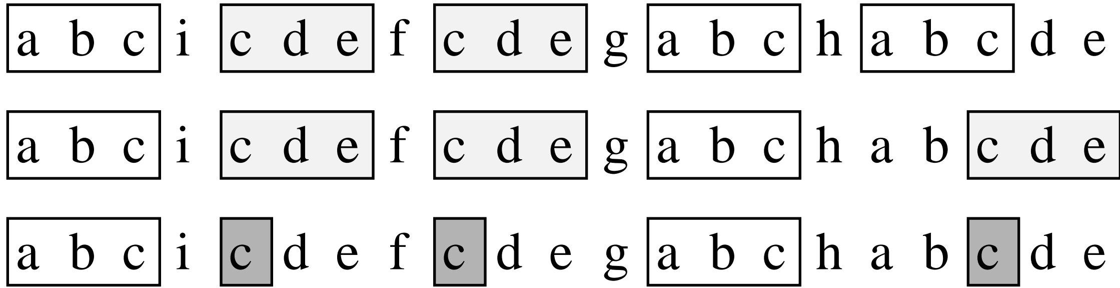

Figure 2: Possible Configurations

Figure 3: Optimal Configuration under Model 1

(1) One-Copy, Maximum-Content, No-Overlap: In this model, exactly one copy of the string is displayed. Occurrences of displayable entities are selected so that there are no mutually conflicting occurrences. Given this restriction, the model requires occurrences to be selected so that the amount of information conveyed by the display is maximized. This goal may be achieved in three ways.

Interactive : The user selects occurrences interactively by using his/her judgement. Typically, this would be done by examining the occurrences which are involved in a conflict and choosing one that is the most meaningful.

Automatic : A numeric weight is assigned to each occurrence. The higher the weight, the greater the desirability of displaying the corresponding occurrence. Criteria that could be used in assigning weights to occurrences include: length, position, number of occurrences of the pattern, semantic value of the displayable entity, information on conflicts, etc. The information is then fed to a routine which selects a set of occurrences so that the sum of their weights is maximized. For example, consider string $S = abcicdefcdegabchabcde$ of Section 2. If the weight assigned to each occurrence of *abc* is 4, *cde* is 2, *c* is 3, then Figure 3 shows the optimal display configuration. The total weight of the display is 18.

Semi-Automatic: In a practical environment, the most appropriate method would be a hybrid of the Interactive and Automatic approaches described above. The user could select some occurrences

a b c i c d e f c d e g a b c h a b c d e

a b c i c d e f c d e g a b c h a b c d e

Figure 4: Optimal Configuration under Model 2(a)

a b c i c d e f c d e g a b c h a b c d e

a b c i c d e f c d e g a b c h a b c d e

a b c i c d e f c d e g a b c h a b c d e

Figure 5: Configuration under Model 2(b)

that he/she wants included in the final display. The selection of the remaining occurrences can then be performed by a routine which maximizes the display information.

(2) Multiple-Copy, No-Overlap: Multiple copies of the string may be displayed. Mutually disjoint sets of occurrences are associated with the copies (one set per copy), so that the occurrences corresponding to each copy are mutually non-conflicting. There are two approaches:

(a) A constant number (max) of copies of the string may be displayed. The total content of the display, summed over all max copies, is to be maximized. For example, consider string S = *abcicdefcdegabchabcde* of Section 2. If the weights corresponding to *abc*, *cde*, and *c* are 4, 2, and 3, respectively, and $max = 2$, then Figure 4 shows the optimal display configuration. The total weight of the display is 31.

(b) No limit is imposed on the number of copies of the string that may be displayed. However, each occurrence is highlighted in only one copy of the string. The number of copies of the string used should be minimized. It can be shown, that in the worst case, $O(n^2)$ copies may be required. Figure 5 shows a configuration for string S = *abcicdefcdegabchabcde*of Section 2.

(3) Single-Copy, Maximum-Content, Subword-Overlap: Exactly one copy of the string may be displayed. Occurrences are selected so that no pair of occurrences has a prefix-suffix conflict.

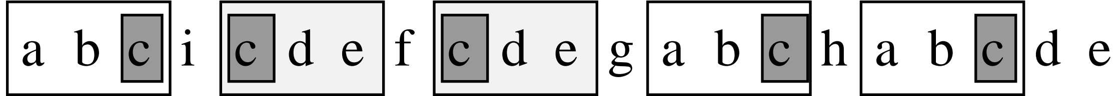

Figure 6: Optimal Configuration under Model 3

Subword conflicts are allowed. As in the Single-Copy, Maximum-Content, No-Overlap model, the goal is to maximize the information conveyed. Again, there are three approaches for selecting occurrences: Automatic, Interactive and Semi-Automatic. For example, consider string S = *abci-cdefcdegabchabcde* of Section 2. If the weights corresponding to each occurrence of *abc*, *cde*, and *c* are 4, 3, and 2, respectively, then Figure 6 shows the optimal display configuration. The total value of the display is 31.

Note that it is crucial to all methods to get information on prefix-suffix and subword conflicts.

4 String Visualization Queries

The following algorithmic problems arise as a result of the discussion in the previous section:

Given a string, S, of length n whose elements are chosen from an alphabet Σ of fixed size,

P1. Obtain a list of all displayable entities and their occurrences.

P2. Obtain a list of all prefix-suffix conflicts.

P3. Obtain a list of all subword conflicts.

Given a list of occurrences of displayable entities and a weight associated with each occurrence,

P4. Obtain a set of mutually non-conflicting occurrences so that the sum of the weights associated with them is maximum (Model 1).

P5. Obtain *max* mutually disjoint sets of mutually non-conflicting occurrences so that the sum of the weights associated with them is maximum (Model 2(a)).

P6. Obtain a minimum number of mutually disjoint sets of mutually non-conflicting occurrences required to partition the set of all occurrences (Model 2(b)).

P7. Obtain a set of occurrences , such that no two have a prefix-suffix conflict, so that the sum of the weights associated with them is maximized (Model 3).

In addition to the problems outlined above, restricted versions of these problems exist. These

problems represent typical queries that would be supported by an interactive visualization system based on our approach. We list some of these below.

(i) Restricted Queries: The *overlap* of a conflict is defined as the string common to the conflicting displayable entities. The overlap of a subword conflict is the subword displayable entity. The overlap of a prefix-suffix conflict is the substring common to the conflicting strings. The *size* of a conflict is the length of the overlap. It is useful to be able to list only those conflicts whose sizes are greater than some specified length. This simplifies the display and eliminates uninteresting displayable entities.

P8. Obtain all prefix-suffix conflicts of size greater than some integer *min*.

P9. Obtain all subword conflicts of size greater than some integer *min*.

(ii) Pattern-Oriented Queries: These queries are useful in applications where the fact that two patterns have a conflict is more important than the number of conflicts or where in the string the conflicts occur.

P10. List all pairs of *displayable entities* which have prefix-suffix or subword conflicts.

P11. List all pairs of *displayable entities* which have conflicts of size greater than some given constant.

P12. List all *displayable entities* that are superwords of a given displayable entity.

(iii) Statistical Queries: These queries are useful when conclusions are to be drawn from the data based on statistical facts.

P13. For each pair of displayable entities, D_1 and D_2, involved in a subword conflict (D_1 is the subword of D_2), obtain p = (*number of occurrences of* D_1 *which occur as subwords of* D_2)/(*number of occurrences of* D_1).

P14. For each pair of displayable entities, D_1 and D_2, involved in a prefix-suffix conflict, obtain q = (*number of occurrences of* D_1 (D_2) *which have prefix-suffix conflicts with* D_2 (D_1))/ (*number of occurrences of* D_1 (D_2)).

If p or q is greater than a statistically determined threshold, then the following could be be said with some confidence: *Presence of* D_1 *implies Presence of* D_2.

A detailed explanation of efficient algorithms for **P1**-**P3** and **P8**-**P14** is provided in [3]. Algorithms for **P1**-**P3** are briefly outlined in Section 6, while algorithms for **P4**, **P6**, and **P7** are presented in

Section 6.6.

5 Applications

An abstract strategy has been outlined for the visualization of strings. This section discusses some general methods for applying this strategy to specific areas. Applications to molecular biology, text and numerical sequences are also outlined.

5.1 General Methods

In order to apply this strategy successfully to actual data, it is important to first check that the data conforms to the definition of strings provided in Section 1. If this is not the case, then it may be possible to transform the data so that it satisfies the definition without losing vital information in the process.

(1) If a string consists of characters which are chosen from a fixed alphabet of a small size (≤ 50, say), then it is already in the correct format. For example, DNA sequences are made up from an alphabet of 4 elements. English text is made up of an alphabet of 26 characters and some special symbols.

(2) Otherwise, if a function, f, can be defined for each element in the alphabet such that:
a) The range of f can be determined and is of constant size and
b) $f(element_1) = f(element_2)$ iff $element_1$ and $element_2$ are similar,
then the given string may be converted to another string which is obtained by applying f to each element of the original string. The resulting string can now be input to the visualization routines.

For example, consider a sequence of objects which are chosen from a large (possibly infinite) alphabet. Assume that a set of properties, $P = \{P_1, P_2, \ldots, P_m\}$, is associated with each object. Suppose that patterns of property, P_i, of the sequence of objects are interesting (where P_i may take on one of a constant, fixed number of values). Then, for the purposes of visualization, each object in the sequence is replaced by the corresponding P_i value. This approach can be used with the other properties as well. Some examples where this approach is useful are:

(1) *Protein Sequences* [2]: A protein sequence consists of a sequence of amino acids. While the number of amino acids that could form a sequence is large , it is possible to place amino acids in groups on the basis of physical properties such as hydrophobicity, acidity, polarity etc. Amino acids in the sequence may then be replaced by a symbol representing the groups to which they belong.
(2) Chronological sequences of Multidimensional Data [1]: Here, a number of measurements relating to a particular scientific phenomenon are taken at regular intervals of time. The measurements for each variable are classified as LOW, MEDIUM, or HIGH. Consequently, the sequence of multidimensional data may be replaced by a sequence of symbols: L,M,H which represent the values of a particular variable.

Many applications would benefit by comparisons between two or more *different strings* (as opposed to comparisons within the same string). This can also be supported by a simple extension of our techniques.

5.2 Numerical Data

An important category of data is numerical data. These arise whenever properties of objects are described by measurements. Numerical information, in general, is chosen from large alphabet sizes which are determined by the accuracy of measurement required. Clearly, numerical sequences cannot be directly input to our visualization system. This is remedied by determining the range of values that a variable can take on. This range is then subdivided into a constant number of subranges (this is essentially the same strategy used in [1]). Each value in the sequence is then replaced by a symbol representing the subrange to which it belongs. The resulting sequence may then be input to our visualization system. Consider a sequence of numbers which lie in the range 1-200. Assume that subranges have been defined as 1-20, 21-40, ..., 181-200 which are respectively represented by the symbols: a,b,......,j. So, the sequence: 7 142 63 94 6 148 69 becomes: $\underline{ahd}$ e $\underline{ahd}$.

Sequences of numbers, such as financial data, are usually studied by using graphs. We expect that the techniques outlined here if used in conjunction with graphs could reduce the possibility of overlooking important patterns in the data. This can be done by either appropriately coloring pieces of the graph or by coloring a string which is aligned with the graph.

Often, comparisons are made not between the values of numbers in a sequence, but between the

increase/decrease in consecutive values. For example, in [5 20 15 75 90 85], 5 20 15 is not obviously related to 75 90 85. However, the increases/decreases in values are identical. An increase of 15 followed by a decrease of 5. Information of this type may be obtained by transforming the string by taking the difference between successive values before inputting it to the visualization system. I.e., 15 -5 60 15 -5. Similar transformations may be used for percentage increases/decreases, second order differences, etc.

5.3 Molecular Biology

In molecular biology, RNA,DNA, and protein sequences are studied. Sequence comparison helps to answer questions about evolution, structure and function in organisms, and the structural configuration of individual RNA molecules [4]. Of particular importance are repeating patterns and their relative positions [5]. [5] uses the scdawg data structure, which is described in the next section, to analyze sequences. Our work improves upon [5] by suggesting more effective display methods as well as by introducing more sophisticated analysis techniques involving prefix-suffix and subword conflicts.

For example, let D_1 and D_2 be displayable entities and D_1 be a subword of D_2. If the fraction (*number of occurrences of* D_1 *contained in* D_2) /(*number of occurrences of* D_1) ≈ 1, then we can infer that D_1 usually occurs as a subword of D_2 which could mean that D_1 does not perform any significant functions except as a subword of D_2.

Suppose patterns P_1 and P_2 perform functions F_1 and F_2 in an organism. Then, if

(*number of prefix-suffix conflicts between* P_1 *and* P_2)/ (*min*{number of occurrences of P_1,number of occurrences of P_2}) $\approx$ *1*, then we can infer that F_1 and F_2 are generally performed by the same region and are therefore related in some way.

5.4 Textual Data

Structural information about text may be obtained by studying prefix-suffix and subword conflicts. Information about the *contexts* in which certain phrases are used is provided by subword conflicts. Information on the *combination of phrases* is provided by prefix-suffix conflicts. This information

can be used to identify anomalies in sentence structure and possibly identify the author of a text by the structure. It can also be used to decipher text coded using sophisticated substitution ciphers where patterns are substituted by other patterns.

6 Implementation Considerations

A string visualization system along the lines described above requires efficient algorithms for problems P1-P14. These problems can be solved in optimal or near optimal time [3] by using the Symmetric Compact Directed Acyclic Word Graph (scdawg) data structure [6, 7].

6.1 Directed Acyclic Word Graphs

An scdawg, $SCD(S)$, corresponding to a string S is a directed acyclic graph defined by a set of vertices, $V(S)$, a set, $R(S)$, of labeled directed edges called right extension (re) edges, and a set, $L(S)$, of labeled directed edges called left extension (le) edges . Each vertex of $V(S)$ represents a substring of S. Specifically, $V(S)$ consists of a source (which represents the empty word, λ), a sink (which represents S), and a vertex corresponding to each displayable entity of S.

Let $de(v)$ denote the string represented by vertex, v ($v \in V(S)$). Define the *implication*, $imp(S, \alpha)$, of a string α in S to be the smallest superword of α in $\{de(v)$: $v \in V(S)\}$, if such a superword exists. Otherwise, $imp(S, \alpha)$ does not exist.

Re edges from a vertex, v_1, are obtained as follows: for each letter, x, in Σ, if $imp(S, de(v_1)x)$ exists and is equal to $de(v_2) = \beta de(v_1)x\gamma$, then there exists an re edge from v_1 to v_2 with label $x\gamma$. If β is the empty string, then the edge is known as a *prefix extension edge*. Le edges from a vertex, v_1, are obtained as follows: for each letter, x, in Σ, if $imp(S, xde(v_1))$ exists and is equal to $de(v_2)$ $= \gamma x de(v_1)\beta$, then there exists an le edge from v_1 to v_2 with label γx. If β is the empty string, then the edge is known as a *suffix extension edge*.

Figure 7 shows $V(S)$ and $R(S)$ corresponding to $S = cdefabcgabcde$. abc, cde, and c are the displayable entities of S. There are two outgoing re edges from the vertex representing abc. These edges correspond to $x = d$ and $x = g$. $imp(S, abcd) = imp(S, abcg) = S$. Consequently, both edges

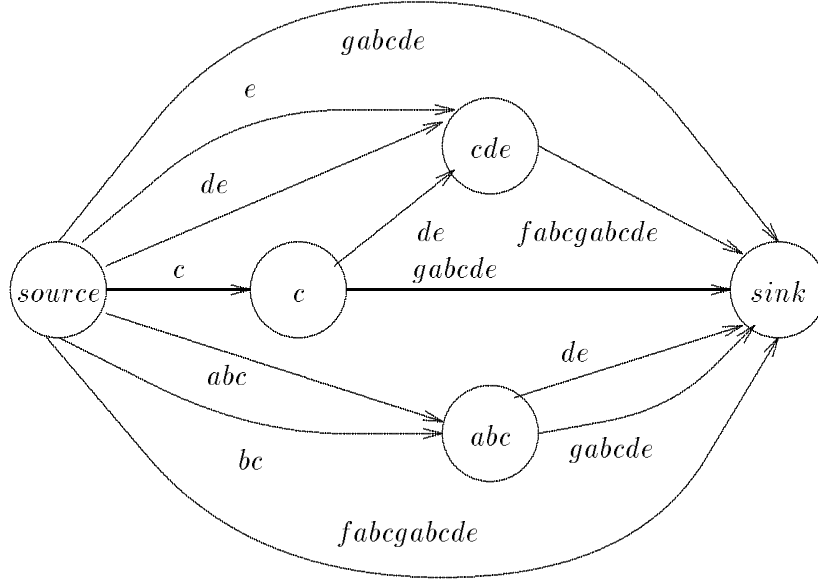

Figure 7: Scdawg for $S = cdefabcgabcde$ ($L(S)$ not shown)

are incident on the sink. There are no edges corresponding to the other letters of the alphabet as $imp(S, abcx)$ does not exist for $x \in \{a, b, c, e, f\}$.

The space required for $SCD(S)$ is $O(n)$ and the time needed to construct it is $O(n)$ [6, 7]. While we have defined the scdawg data structure for a single string, S, it can be extended to represent a *set* of strings.

6.2 Computing Subword Conflicts Efficiently

6.2.1 Representing Subword Conflicts

Consider, **P2**, the problem of finding all subword conflicts in string S. Let k_s be the number of subword conflicts in S. Any algorithm to solve this problem requires (i) $O(n)$ time to read in the input string and (ii) $O(k_s)$ time to output all subword conflicts. So, $O(n + k_s)$ is a lower bound on the time complexity for this problem. For the string $S = a^n$, $k_s = n^4/24 + n^3/4 - 13n^2/24 - 3n/4 + 1$ $= O(n^4)$. This is an upper bound on the number of conflicts as the maximum number of substring occurrences is $O(n^2)$ and in the worst case, all occurrences conflict with each other. In this section, a compact method for representing conflicts is presented. Let k_{sc} be the size of this representation.

k_{sc} is $n^3/6 + n^2/2 - 5n/3$ or $O(n^3)$, for a^n. Compaction never increases the size of the output and may yield up to a factor of n reduction, as in the example. The compaction method is described below.

Consider S= *abcdbcgabcdbchbc*. The displayable entities are D_1 = *abcdbc* and D_2 = *bc*. The ending positions of D_1 are 6 and 13 while those of D_2 are 3, 6, 10, 13, and 16. A list of the subword conflicts between D_1 and D_2 can be written as: {(6,3), (6,6), (13,10), (13,13)}. The first element of each ordered pair is the last position of the instance of the superstring (here, D_1) involved in the conflict; the second element of each ordered pair is the last position of the instance of the substring (here, D_2) involved in the conflict.

The cardinality of the set is the number of subword conflicts between D_1 and D_2. This is given by: *frequency*(D_1)∗*number of occurrences of* D_2 *in* D_1, where *frequency*(D_1) is the number of occurrences of D_1 in S. Since each conflict is represented by an ordered pair, the *size* of the output is 2(*frequency*(D_1)∗*number of occurrences of* D_2 *in* D_1).

Observe that the occurrences of D_2 in D_1 are in the same relative positions in all instances of D_1. It is therefore possible to write the list of subword conflicts between D_1 and D_2 as: (6,13):(0,-3). The first list gives all the occurrences in S of the superstring (D_1), and the second gives the relative positions of all the occurrences of the substring (D_2) in the superstring (D_1) from the right end of D_1. The size of the output is now: *frequency*(D_1)+*number of occurrences of* D_2 *in* D_1. This is more economical than our earlier representation.

In general, a substring, D_i, of S will have conflicts with many instances of a number of displayable entities (say, $D_j, D_k, \ldots, D_z$) of which it (D_i) is the superword. We would then write the conflicts of D_i as:

$$(l_i^1, l_i^2, \ldots, l_i^{m_i}) : (l_j^1, l_j^2, \ldots, l_j^{m_j}), (l_k^1, l_k^2, \ldots, l_k^{m_k}), \ldots, (l_z^1, l_z^2, \ldots, l_z^{m_z}).$$

Here, the l_i's represent all the occurrences of D_i in S; the $l_j's, l_k's, \ldots, l_z's$ represent the relative positions of all the occurrences of $D_j, D_k, \ldots, D_z$ in D_i. One such list will be required for each displayable entity that contains other displayable entities as subwords. The following equalities are easily obtained:

Size of Compact Representation = $\sum_{D_i \epsilon D} (f_i + \sum_{D_j \epsilon D_i^s} (r_{ij}))$.

Size of Original Representation = $2 \sum_{D_i \epsilon D} (f_i * \sum_{D_j \epsilon D_i^s} (r_{ij}))$.
f_i is the frequency of D_i (only D_i's that have conflicts are considered). r_{ij} is the frequency of D_j in one instance of D_i. D represents the set of all displayable entities of S. D_i^s represents the set of all displayable entities that are subwords of D_i.

6.2.2 Computing Subword Conflicts

Algorithm A3 of Figure 8 computes the subword conflicts of S. These are represented using the scheme described in Section 6.2.1.

$SG(S, v)$, $v \ \epsilon \ V(S)$, is defined as the subgraph of $SCD(S)$ which consists of the set of vertices, $SV(S, v) \subset V(S)$ which represent displayable entities that are subwords of $de(v)$ and the set $SE(S, v)$ of all re and suffix extension edges that connect any pair of vertices in $SV(S, v)$. Define $SG_R(S, v)$ as $SG(S, v)$ with the directions of all the edges in $SE(S, v)$ reversed.

The subword conflicts are computed for precisely those displayable entities which have subword displayable entities. Lines 4 to 6 of *Algorithm A3* determine whether $de(v)$ has subword displayable entities. Procedure *Getsubwords*(v), which computes the subword conflicts of $de(v)$ is invoked if $v.subword$ is *true*.

Procedure *Occurrences*$(S, v, 0)$ (line 2 of *GetSubwords*) computes the occurrences of $de(v)$ in S and places them in *v.list*. Procedure *SetUp* in line 5 traverses $SG_R(S, v)$ and initializes fields in each vertex of $SG_R(S, v)$ so that a reverse topological traversal of $SG(S, v)$ may be subsequently performed. Procedure *SetSuffixes* in line 6 marks vertices whose displayable entities are suffixes of $de(v)$. A list of relative occurrences, *sublist*, is associated with each vertex, x, in $SG(S, v)$. $x.sublist$ represents the relative positions of $de(x)$ in an occurrence of $de(v)$. If $de(x)$ is a suffix of $de(v)$ then $x.sublist$ is initialized with the element, 0. The remaining elements of $x.sublist$ are computed from the *sublist* fields of vertices, w, in $SG(S, v)$ such that a right extension edge goes from x to w. Consequently, $w.sublist$ must be computed before $x.sublist$. This is achieved by traversing $SG(S, v)$ in reverse topological order [8].

Theorem 1 *Algorithm A3 takes* $O(n + k_{sc})$ *time and space and is therefore optimal [3].*

Algorithm A3

```
1       begin
2           for each vertex, v, in SCD(S) do
3               begin
4                   v.subword = false;
5                   for all vertices, u, such that a right or suffix extension edge, < u, v >, is incident on v do
6                       if u ≠ source then v.subword = true;
7               end
8           for each vertex, v, in SCD(S) such that v ≠ sink and v.subword is true do
9               GetSubwords(v);
10      end
```

Procedure *GetSubwords*(v)

```
1       begin
2           Occurrences(S,v,0);
3           output(v.list);
4           v.sublist = {0};
5           SetUp(v);
6           SetSuffixes(v);
7           for each vertex, x (≠ source), in reverse topologicalorder of SG(S, v) do
8               begin
9                   if de(x) is a suffix of de(v) then x.sublist = {0} else x.sublist = {};
10                  for each vertex, w, in SG(S, v) on which an re edge, e from x is incident do
11                      begin
12                          for each element, l, in w.sublist do
13                              x.sublist = x.sublist ∪ {l − |label(e)|};
14                      end;
15                  output(x.sublist);
16              end;
17      end
```

Figure 8: Optimal algorithm to compute all subword conflicts

Algorithm A2A3
Step 1: Obtain a list of *all* occurrences of *all* displayable entities in the string. This list is obtained by first computing the lists of occurrences corresponding to each vertex of the scdawg (except the source and the sink) and then concatenating these lists.
Step 2: Sort the list of occurrences using the start positions of the occurrences as the primary key (increasing order) and the end position as the secondary key (decreasing order). This is done using radix sort.
Step3:

```
for i:= 1 to (number of occurrences) do
    begin
        j:= i + 1;
        while(lastpos(occ_i) ≥ firstpos(occ_j) do
            begin
                if (lastpos(occ_i) ≥ lastpos(occ_j))
                then occ_i is a superword of occ_j
                else (occ_i, occ_j) have a prefix-suffix conflict;
                j:= j + 1;
            end;
    end;
```

Figure 9: A simple algorithm for computing conflicts

6.3 Computing Prefix Suffix Conflicts Efficiently

As with subword conflicts, the lower bound for the problem of computing prefix-suffix conflicts is $O(n + k_p)$, where k_p is the number of prefix-suffix conflicts in S. For $S = a^n$, k_p is $n^4/24 - n^3/12 - 25n^2/24 - 21n/12 + 1 = O(n^4)$, which is also the upper bound on k_p. Unlike subword conflicts, it is not possible to compact the output representation.

Theorem 2 *All prefix-suffix conflicts in S can be computed in $O(n + k_p)$ space and time, which is optimal [3].*

6.4 Alternative Algorithms

In this section, an alternative solution for computing subword and prefix-suffix conflicts is presented. The solution is relatively simple and has competitive running times. However, it lacks the flexibility required to solve many of the problems listed in Section 4. The algorithm is presented in Figure 9.

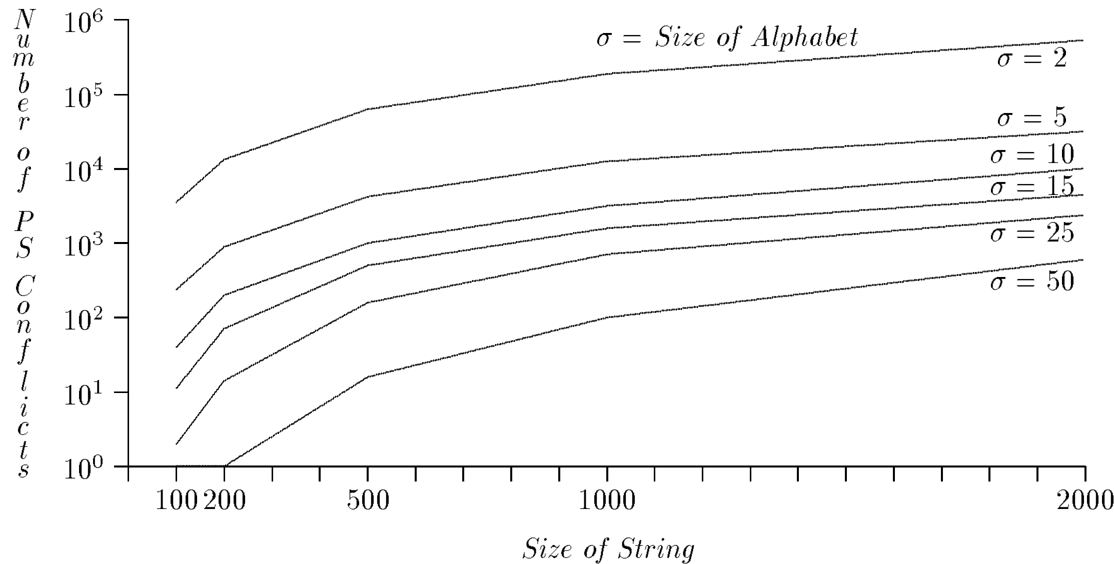

Figure 10: Graph of Number of Prefix Suffix Conflicts vs String Size

6.5 Implementation

The algorithms to compute all prefix-suffix and subword conflicts (i.e., algorithms from Sections 6.2, 6.3, and 6.4) were implemented on a SUN SPARCstation 1 in GNU C++. 50 randomly generated strings of lengths ranging from 100 to 2000 from alphabets whose size ranged from 2 to 50 were input to our algorithms. Statistical information such as the number of vertices in the scdawg, the number of prefix-suffix and subword conflicts etc was obtained. The run times of the algorithms were also recorded.

(i) Figure 10 shows Number of prefix-suffix conflicts vs String size. There is one curve for each alphabet size. The plot illustrates that the number of prefix-suffix conflicts increases with string size and decreases with alphabet size. The graph for subword conflicts is similar.

(ii) Figure 11 shows Time per prefix-suffix conflict (μs) vs String size. There is one curve for each alphabet size. It illustrates that the time per conflict generally decreases with increasing string size and increases with alphabet size. The graph for subword conflicts is similar.

(iii) The factor by which the compact representation of subword conflicts is smaller than the fully expanded representation varies from 2 to 9. It increases with string size and decreases with alphabet size.

(iv) Figure 12 shows the size of the largest displayable entity for each combination of alphabet size

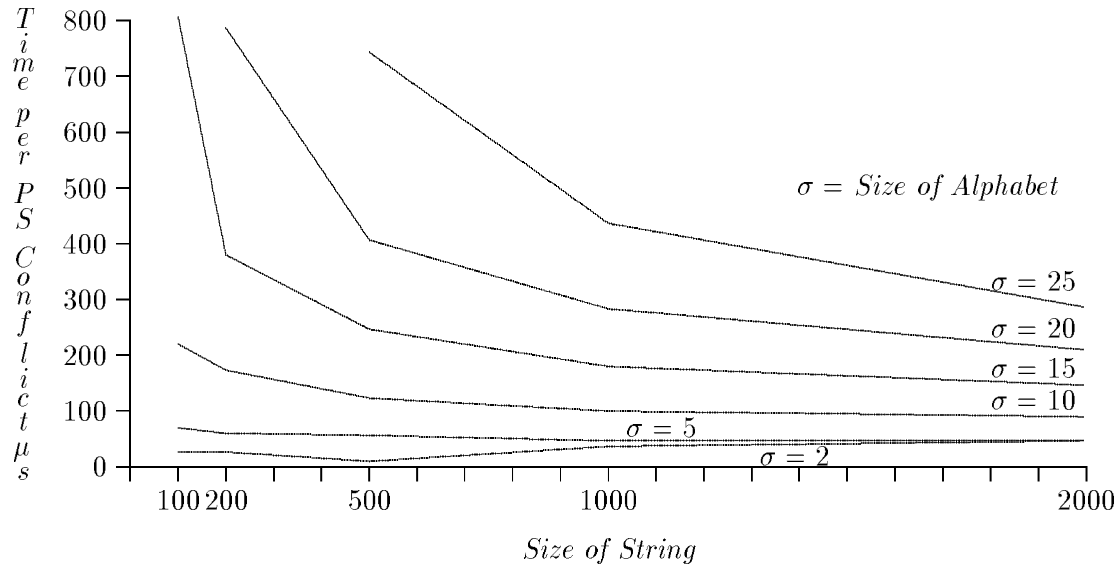

Figure 11: Graph of Time per Prefix Suffix Conflict vs String Size

Size of Alphabet	*Size of String*				
	100	200	500	1000	2000
2	11	12	15	18	19
5	5	6	7	8	9
10	3	4	5	6	6
15	3	3	4	5	5
20	3	3	4	4	5
25	2	3	3	4	4
50	2	2	3	3	4

Figure 12: Lengths of Largest Displayable Entity for Random Strings

and string size. It shows that only displayable entities of small lengths occur in random strings (in practice we would like to be able to distinguish between displayable entities that occur randomly and those that do not in a given string. This can be done by selecting those displayable entities whose frequency is large compared to other displayable entities of the same length in a random string).

6.6 Display Algorithms

A list of all occurrences of a displayable entity may be obtained from the scdawg data structure described earlier. A list of all conflicts between displayable entities may be obtained by operations

on the scdawg. This information is then used to assign numeric weights to each occurrence of each displayable entity.

In this section we shall discuss algorithms to implement some of the refined display models of Section 3. Specifically, we shall consider models 1, 2(b), and 3 under the Automatic mode (i.e., problems **P4**, **P6**, and **P7** of Section 4). Our algorithms also apply to the Semi Automatic mode.

Problem **P4** can be reduced to the single pair, longest path problem in a directed acyclic graph as follows: let vertex V_i, $1 \leq i \leq n$, correspond to the position between the i'th and $i+1$'th characters in the string. V_0 corresponds to the position preceding the first character in the string. For each occurrence $S_{i,j}$ of each displayable entity, we create an edge from V_{i-1} to V_j. The weight associated with this edge is exactly the weight of the occurrence it represents. Finally, for each pair (V_i, V_{i+1}), $0 \leq i \leq n$, of vertices such that an edge from V_i to V_{i+1} does not already exist, we create an edge from V_i to V_{i+1} of weight 0.

Figure 13 shows the directed acyclic graph corresponding to the string, $S = abcicdefcdegabchancde$ of Figure 2, assuming that the weights corresponding to each occurrence of *abc*, *cde*, and *c* are 4, 3, and 2 respectively. The longest path from V_0 to V_n in the dag is $V_0 \rightarrow V_3 \rightarrow V_4 \rightarrow V_7 \rightarrow V_8 \rightarrow V_{11} \rightarrow V_{12} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{19} \rightarrow V_{20} \rightarrow V_{21}$. All edges on this path with non-zero weight represent occurrences of displayable entities that are to be highlighted. Here $V_0 \rightarrow V_3$, $V_4 \rightarrow V_7$, $V_8 \rightarrow V_{11}$, $V_{12} \rightarrow V_{15}$, $V_{16} \rightarrow V_{19}$ correspond to occurrences $< 1,3 >$, $< 13,15 >$, and $< 17,19 >$ of *abc*, and $< 5,7 >$ and $< 9,11 >$ of *cde*. The length of the longest path (here, 18) represents the total weight of the display.

Algorithm A4 of Figure 14 solves problem **P4**. $A[0..n]$ is an array of integers, where $A[i]$ represents the longest path from V_0 to V_i detected upto that point of time. All elements of A are initialized to 0. The auxiliary array, $T[1..n]$ stores the occurrences that have been chosen for display. The array *delist* contains all the occurrences of all the displayable entities in the string. Each element of *delist* contains three fields: *start*, *end*, and *weight* which respectively represent the start position, the end position, and the numeric weight of the occurrence. It is assumed that *delist* is sorted in increasing order of *end*. The vertices are processed in topological order (i.e., V_0, V_1, ..., V_n). When a vertex, V_j is being processed, each vertex, V_i $(i < j)$ preceding it has associated with it the cost of the longest path from V_0 to V_i. The cost of the longest path from V_0 to V_j is then

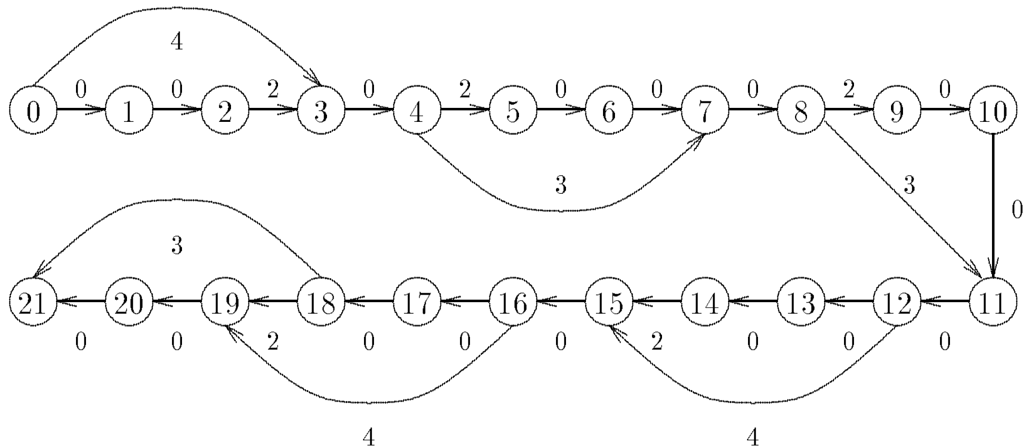

Figure 13: Dag corresponding to *abcicdefcdegabchabcde*

determined by examining each of the incoming edges to V_j.

The complexity of **Algorithm** *A4* is $O(n+e)$, where e represents the size of *delist*. So, **Algorithm** *A4* is optimal. Note that sorting *delist* on end position can also be accomplished in $O(n+e)$ time using radix sort.

Problem **P6** of Section 4 is solved by *Algorithm A6* of Figure 15 using the greedy method. *endpoints[1..2e]* is a list of both, the start positions and end positions of all occurrences of all displayable entities in the string. Each element of *endpoints* contains three fields: *position*, *type*, and *id*. *position* contains the position of the particular endpoint in the string; *type* determines whether the endpoint is a "start" position or an "end" position. *id* uniquely identifies the occurrence corresponding to the endpoint. It is assumed that *endpoints* is sorted in increasing order on primary key, *position*, and secondary key, *type* ("start" < "end"). *LineStack* is a stack that contains line numbers (or partition numbers) which are currently available. On completion of the algorithm, *line[i]* contains the line or the particular copy of the string in which the i'th occurrence is to be highlighted for $1 \leq i \leq e$.

Proof of Correctness: From the algorithm the following assertions can be made:
(1) The final value of *max* (*fmax*) represents the number of partitions of S.
(2) At least one position in the string is covered by *fmax* occurrences. Statement (2) indicates that *fmax* is the smallest possible number of partitions that satisfy the problem. Statement (1) indicates

```
Algorithm A4
begin
    A[0]:= 0; j:= 1;
    for i := 1 to n do
        begin
            A[i]:= 0;
            while (delist[j].end = i) do
                begin
                    if A[i] < A[delist[j].start-1] + delist[j].weight
                    then
                        begin
                            A[i] := A[delist[j].start-1] + delist[j].weight;
                            T[i] := j;
                        end;
                    j:= j + 1;
                end;
        end;
end.
```

Figure 14: Algorithm for **P4**

that a partition of that size is in fact obtained.

Algorithm A6 consumes $O(n+e)$ time, which is optimal. Note that sorting *endpoints* can also be accomplished in $O(n+e)$ time, if radix sort is used.

We outline two solutions to problem **P7**. In the first, we assume that all occurrences of the same displayable entity are assigned the same numeric weight. In the second, we do not make this assumption.

Algorithm A7(a) of Figure 16 solves the first version of **P7**. The second version of **P7** may be solved by executing steps a-c for each occurrence of each displayable entity as shown in *Algorithm A7(b)* of Figure 17. Both solutions involve a traversal of $SCD(S)$ in topological order. For each occurrence, an optimal selection of its subwords are chosen. The weight of the occurrence is then obtained by adding the sum of the weights of the chosen subwords to its weight. This is done because an occurrence is highlighted along with the chosen subwords. *Algorithm A7(a)* consumes $O(n^3)$ time, while *Algorithm A7(b)* consumes $O(n^4)$ time.

```
Algorithm A6
begin
    max := 0; current := 0;
    for i := 1 to 2e do
        begin
            x := endpoints[i];
            if (x.type = "start") then
                begin
                    current := current + 1;
                    if (current ≤ max) then
                        begin
                            CurrentLine := top(LineStack);
                            pop(LineStack);
                        end
                    else
                        begin
                            max := max + 1;
                            CurrentLine := max;
                        end;
                    line[x.id] := CurrentLine;
                end
            else {x.type = "end"}
                begin
                    current := current - 1;
                    return line[x.id] to LineStack;
                end;
        end;
    fmax := max;
end.
```

Figure 15: Algorithm for **P6**

Algorithm *A7(a)*
begin
  for each vertex, v, of $SCD(S)$, in topological order **do**
    begin
      Step a. Compute the relative positions of all subword displayable entities in a single instance of $de(v)$ using Algorithm A of Section 6.2.
      Step b. Choose a mutually non overlapping subset from the set of subwords of $de(v)$ (obtained in step (a) above) so that the sum of their weights is maximized. This is achieved by an algorithm similar to A4.
      Step c. Reset the numeric weight of $de(v)$ by adding to it the total weight of the configuration obtained in step (b).
    end;
end.

Figure 16: Algorithm for **P7**, same weights

Algorithm *A7(b)*
begin
  for each vertex, v, of $SCD(S)$, in topological order **do**
    begin
      for each occurrence $< i, j >$ of $de(v)$ **do**
        begin
          Step a. Compute the occurrences of all subword displayable entities in $< i, j >$.
          Step b. Choose a mutually non overlapping subset from the set of occurrences (obtained in step (a) above) so that the sum of their weights is maximized. This is achieved by an algorithm similar to A4.
          Step c Reset the numeric weight of $< i, j >$ by adding to it the total weight of the configuration obtained in step (b).
        end;
    end;
end.

Figure 17: Algorithm for **P7**, different weights

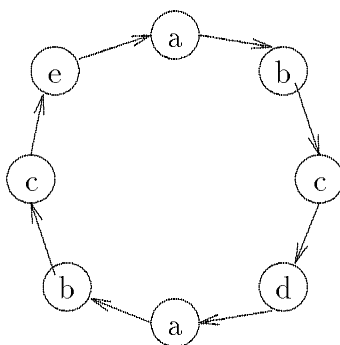

Figure 18: Circular string

7 Circular String Visualization

In this section, we consider the problem of circular string visualization. Section 7.2 mentions some of its applications while Section 7.3 outlines the algorithms.

7.1 Problem Definition

As with the linear string, we provide a specification for the circular string visualization problem:

1. Structure of Data to be Visualized: A circular string of length n whose characters are chosen from a fixed alphabet Σ of constant size. Figure 18 shows an example circular string of size 8.

2. Structure of Patterns: A linear substring of the circular string of length less than n. For example, *abc* and *ceab* are patterns in Figure 18.

3. Maximality of Patterns: The definition of maximality of a pattern in a circular string is similar to that of a linear string. I.e., a pattern is said to be maximal iff its occurrences in the circular string are not all preceded by the same character nor all followed by the same character.

4. Measure of Similarity (**MS**): If two patterns are identical, then **MS** = 1. Otherwise, **MS** = 0.

5. Display Model: A maximal pattern in a circular string is called a displayable entity if it occurs at least twice in the circular string. In the example string of Figure 18, *abc* is a displayable entity. All instances of the same displayable entity are highlighted in the same color. As with linear strings, we encounter the problem of prefix-suffix and subword conflicts. Similar techniques are used to overcome these.

7.2 Applications

Circular strings may be used to represent circular genomes [5] such as $G4$ and $\phi X174$. The detection and analysis of patterns in genomes helps to provide insights into the evolution, structure, and function of organisms. [5] analyzes $G4$ and $\phi X174$ by linearizing them and then constructing their scdawg. Our work improves upon [5] by :
(i) analyzing circular strings without risking the "loss" of patterns.
(ii) extending the analysis and visualization techniques presented for linear strings to circular strings.

Circular strings in the form of chain codes are also used to represent closed curves in computer vision [9]. The objects of Figure 19(a) are represented in chain code as follows:
(1) Arbitrarily choose a pixel through which the curve passes. In the diagram, the starting pixels for the chain code representation of objects 1 and 2 are marked by arrows.
(2) Traverse the curve in the clockwise direction. At each move from one pixel to the next, the direction of the move is recorded according to the convention shown in Figure 19(b).

Objects 1 and 2 are represented by *112210224324466666666* and *666666661122002242242446*, respectively. The alphabet is {0, 1, 2, 3, 4, 5, 6, 7} which is fixed and of constant size (8) and therefore satisfies the definitions of Section 7.1. We may now use our visualization techniques to compare the two objects. For example, our methods would show that objects 1 and 2 share the segments S1 and S2 (Figure 19(c)) corresponding to *0224* and *2446666666661122*, respectively. Information on other common segments would also be available. The techniques of this paper make it possible to detect all patterns irrespective of the starting pixels chosen for the two objects.

Circular strings may also be used to represent polygons in computer graphics and computational geometry [10]. Figure 20 shows a polygon which is represented by the following alternating sequence