

# SIATEC and SIA: Efficient algorithms for translation-invariant pattern discovery in multi-dimensional datasets

David Meredith\*      Kjell Lemström\*<sup>‡</sup>      Geraint A. Wiggins\*

\*City University, School of Informatics, Department of Computing,  
Northampton Square, London, EC1V 0HB, United Kingdom.

<sup>‡</sup>Department of Computer Science, FIN-00014 University of Helsinki, Finland.

{dave,kjell,geraint}@soi.city.ac.uk

May 22, 2001

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Motivation behind research</b>	<b>2</b>
<b>3</b>	<b>Related Work</b>	<b>3</b>
3.1	Problem setting . . . . .	3
3.2	Solutions for P1 . . . . .	4
3.3	Solutions for P2 . . . . .	5
<b>4</b>	<b>The mathematical function that SIATEC computes</b>	<b>6</b>
<b>5</b>	<b>SIATEC: An overview</b>	<b>9</b>
<b>6</b>	<b>SIATEC: A closer look</b>	<b>13</b>
6.1	The data structures used . . . . .	13
6.2	Reading and preparing the dataset . . . . .	17
6.3	Computing all inter-datapoint vectors . . . . .	21
6.4	Computing $P'(D)$ . . . . .	23
6.5	Computing $P''(D)$ . . . . .	36
6.6	Computing $T'(D)$ . . . . .	47
6.7	Outputting the results . . . . .	52
<b>7</b>	<b>SIA</b>	<b>52</b>

## 1 Introduction

We present below **SIA** and **SIATEC**, two new algorithms for efficient and effective pattern-discovery in multidimensional datasets. (A *multidimensional dataset* is simply any set of points in an  $N$ -dimensional space.) These algorithms can be used as the basis of new applications for compression and indexing of databases, and data mining or structural analysis of data. The new algorithms are particularly appropriate for use with databases in which each item in the database is represented as a multidimensional dataset, as is the case, for example, in computer-based music libraries and databases of audio data, databases of 2- and 3-dimensional molecular structures, computer-based image and video libraries and collections of graphs representing scientific results or financial data.

When each element in the database to be processed is a multidimensional dataset and when the user wishes to discover sets of translationally-invariant patterns, **SIA** and **SIATEC** are more effective than previous string-based approaches.

Possible applications of **SIA** and **SIATEC** include but are not limited to:

- Prediction and analysis of financial data such as stock market performance.
- Storage and matching of image data in security systems based on recognition of images such as retinal scans, fingerprints, faces, hands etc.
- Compression and indexing of huge databases of MIDI or MPEG files.
- Transcription of digital audio to a symbolic representation (e.g. audio-to-MIDI).
- Compression and indexing of video data (e.g. MPEG-7 files).
- Analysis of 2- and 3-dimensional molecular structure as used, for example, in drug development, proteomics and genomics.

## 2 Motivation behind research

Our research was originally motivated by the desire for extraction (discovery) of patterns in music data. There are various reasons for automatic extraction of musical patterns. First of all, from a musicological point of view it is interesting to find out which patterns recur and what kind they are. It may be the case, for example, that such patterns can be used as stylistic ‘fingerprints’ for particular musical styles or composers. Moreover, a number of music analysts and music psychologists (Bent and Drabkin, 1987; Lerdahl and Jackendoff, 1983; Nattiez, 1975; Ruwet, 1966, 1972), have stressed that one of the fundamental steps in achieving an interpretation of a piece of music is identifying the significant instances of repetition within the piece. So, being able to extract the recurring patterns means that the search space for finding musically meaningful patterns has drastically been reduced, and some heuristics for deciding which of the recurring patterns really are meaningful can be conducted.

Secondly, the recurring patterns can be used for sparse indexing purposes in cases where the music databases used for content-based music retrieval are enormous. To enable interactive queries to such databases, some kind of indexing is required. However, even the most economical version of suffix trees (Kurtz, 1999), a very effective indexing structure, will be of excessive size (Lemström, 2000). For example, in the case of a database of 100,000 MIDI songs (Bainbridge et al., 1999) (which should still be considered only moderately sized), a suffix tree would require at least 5 GB of main memory. Therefore, we claim that the sensible strategy for such a case would involve an indexing structure storing only the musically most meaningful patterns. In a straightforward solution all the recurring patterns could be stored in such a structure, whereas a more complicated but more efficient solution would involve using analysis heuristics to store nothing but the most meaningful patterns. In this way we could maintain a reasonably

sized indexing structure and frequently enable interactive performance, because the musically meaningful patterns will frequently be used as query keys. If the query key cannot be located in the indexing structure, a fast online technique, for instance the techniques described by Lemström (2000), could be used.

We also believe that **SIA** and **SIATEC** could be used as the basis for a very effective and efficient compression algorithm for symbolically encoded polyphonic music. The idea behind compression techniques is to remove the recurrences of the data. However, the concept of a repetition in symbolically represented polyphonic music is quite different from the repetition of a word or phrase in text, which has been the application area for most of the available compression techniques. The difference is due to the fact that, no matter how the musical data has been ordered, there may always be some interleaved events that are actually not part of the meaningful pattern at all (such as ornamentations or part of an accompaniment—see Figure 1). Therefore, in effect, they would destroy the prominent lengthy pattern if text compression techniques were used.

### 3 Related Work

Because the previous work has been based on string matching techniques, we will now briefly talk about that framework in order to understand the difference between the problem that the previous methods have been able to solve, and the problem that we are facing.

#### 3.1 Problem setting

Let  $\Sigma$  be a finite set of symbols, called an *alphabet*. Then any  $A = (a_1, a_2, \dots, a_m)$  where each  $a_i$  is a symbol in  $\Sigma$ , is a *sequence* over  $\Sigma$ . The set of all sequences over  $\Sigma$  is denoted by  $\Sigma^*$ . If a sequence  $A$  is of form  $A = \alpha\beta\gamma$ , where  $\alpha, \beta, \gamma \in \Sigma^*$ , we say that  $\alpha$  is a *prefix*,  $\beta$  a *factor* (substring), and  $\gamma$  a *suffix* of  $A$ . A sequence  $A'$  is a *subsequence* of  $A$  if it can be obtained from  $A$  by deleting zero or more symbols, i.e.,  $A' = a_{i_1} a_{i_2} \dots a_{i_m}$ , where  $i_1 \dots i_m$  is an increasing sequence of indices in  $A$ .

To find repeated patterns within a string, one can consider the following problem setting as a starting point. Given a finite set  $W$  of words, the task is to find a string, called a *pattern*,  $p$  that is the longest *substructure* (i.e. factor or subsequence) of every word in  $W$ . Note, that  $W$  may have been obtained from a long string  $S$  that has been divided into  $|W|$  factors. It is known (Crochemore and Rytter, 1994, p. 25), that if  $|W|$  is not constant and the substructures to be considered are subsequences, the problem becomes NP- complete. However, if factors instead of subsequences are considered, the problem is solvable in polynomial time. Bearing this fact in mind, it is clear that finding repetitions with ‘gaps’ (the repetitions correspond to similar subsequences) is much more complex than finding repetitions without gaps (the repetitions correspond to similar factors).

Let us now define the problem under consideration and a related problem. The slightly easier problem, denoted here by P1, is to find every factor  $p_i$  such that each  $p_i$  occurs more than once in string  $S$  to be inspected. By assuming that the underlying alphabet is integer valued <sup>1</sup>, the string-matching problem that corresponds most closely to that solved by **SIATEC** and **SIA** which we denote by P2, can be formulated as follows. The task is to find all subsequences  $p'_j$  having more than one occurrence in  $S$  (note that, by definition, P2 contains P1 as a subcase). Naturally, by formulating the problems this way not only the transposition invariance but also approximate matching (or both of them combined) become available. For instance, in the transposition invariant case,  $p_i$  is similar to  $p_j$  if and only if all the elements of  $p_j$  are transposed from the corresponding element in  $p_i$  by the same integer  $c$ . Two strings are said to be  $k$ -approximately similar if the latter can be obtained from the former (or vice versa) by using  $k$  or fewer *editing operations* (see e.g. Crochemore and Rytter (1994)).

---

<sup>1</sup>We do not need this assumption in the multi-dimensional dataset definition of the problem given below.

### 3.2 Solutions for P1

Denoting the length of  $S$  by  $n$ , the straightforward solution for P1 is to generate all the  $O(n^2)$  substrings of  $S$ . Then, each substring is matched against  $S$  to find out how many times it occurs in  $S$ . For a substring of length  $m$ ,  $O(mn)$  comparisons have to be made. Thus, the overall complexity becomes  $O(n^4)$ .

By applying *suffix tries* an  $O(n^3)$  solution can be obtained. Formally, a trie is a rooted tree with two main properties, i.e.

1. each node, except the root, is associated with a symbol of an alphabet,
2. any two descendants of the same node cannot be associated with the same symbol.

A node corresponding to a last symbol of any suffix is called a *final state*. To solve P1, first a suffix trie for  $S$  is built, which can be done in  $O(n^2)$  time (because there are  $O(n^2)$  nodes in the structure). Then, for each node a link to the final states that are its descendants, is created. Finally, the repeated patterns can be found by traversing the trie; for each internal node corresponding to a substring, its occurrences can be located by going through the final states linked by the node. The time complexity follows from the number of the nodes in the trie, and the number of the final states (i.e.  $n$ ).

Hsu et al. (1998) used a dynamic programming technique to find the repeating patterns. Though having a  $O(n^4)$  worst case time complexity, it works much more efficiently in practice. In their study, Hsu et al. did not consider transposition invariance, but this can be obtained straightforwardly by using intervals between the consecutive pitches instead of the absolute pitch values. However, their approach can only be used for monophonic music or for discovering patterns wholly contained within a single voice of a polyphonic piece. First they use a correlative matrix, whose processing take  $O(n^2)$  time. The output of the phase is a candidate set including all the patterns comprised in  $S$  (even those that appear only once), together with their frequency. In a second phase, each candidate which is a subset of another candidate  $c$  (and whose frequency does not exceed the frequency of  $c$ ) is removed from the candidate set. In a pathological case the number of required operations for this phase can be  $O(n^4)$ . However, there are rather fewer candidates to be considered in practice.

Another approach to solving P1 was presented by Rolland (1999). However, his solution extends to the problem of allowing  $k$  proximity. The editing operations can be chosen from a set of predefined, musically relevant operations. In his method, the length of the considered patterns is bounded by setting a minimal and maximal length for them. (In SIA and SIATEC there are no such bounds—the patterns discovered may be of any size.) Then, all the patterns (substrings) falling in the allowed range are extracted from the musical string. Having created a node for a pattern, it is compared against other patterns (whose length is close enough to that of the considered pattern). If their similarity exceeds a given threshold value, an arc between the two nodes corresponding to these patterns is created and the weight of this arc is set according to their similarity. In this way, the related patterns form *stars* in a similarity graph. Finally, the patterns are sorted by their prominence: for every pattern in the similarity graph, the weights of the arcs by which they are connected to other nodes are accumulated. Then, the patterns are listed in a descending order according to the accumulations. Rolland claims an overall time complexity of  $O(n^2)$ . However, if the algorithm is required to find patterns of any size this time complexity rises to (at least)  $O(n^4)$ , making it less efficient in the worst case than SIATEC. Also, unlike SIATEC and SIA, Rolland’s algorithm is currently limited to the discovery of patterns in monophonic sources.

In music, the pattern in Figure 1(b) would be understood to be an ornamented repetition of the pattern in Figure 1(a). Notes 1, 2, 3 and 4 in Figure 1(a) correspond respectively to notes 1, 5, 9 and 13 in Figure 1(b). This kind of musical ornamentation is called *diminution* and the number of ornamental notes that can be inserted between the structurally more important notes of the theme can be arbitrarily high. A string-matching pattern-discovery algorithm such as Rolland’s considers two patterns to be ‘similar’ if and only if the number of edit operations required to transform one into the other is less than some threshold  $k$ . For the two patterns in Figure 1 to be considered ‘similar’ by Rolland’s algorithm, this threshold  $k$  would

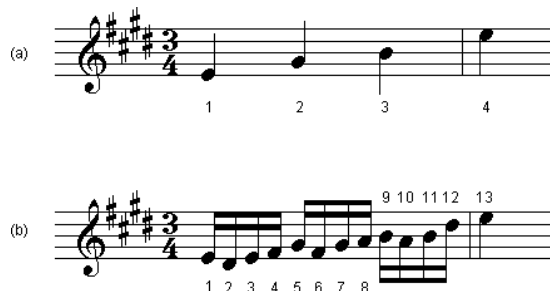


Figure 1: Two musically similar patterns separated by a large edit distance.

have to be set to at least 9 to allow for the 9 insertions required to transform Figure 1(a) into Figure 1(b). But this threshold is far too high in general, because, with such a high threshold, the algorithm would classify certain musically extremely dissimilar patterns to be ‘similar’.

This shortcoming applies to any pattern-discovery algorithm based on approximate string-matching techniques that employ the edit-distance approach.

Cambouropoulos’ (1998) GCTMS can be rather straightforwardly adapted to finding repeated patterns. However, the class of patterns found is highly restricted. First, the patterns cannot contain gaps; second, they are bounded by local boundaries that are found in a preprocessing phase using *gestalt*-like rules; and third, the system only works for monophonic music. Cambouropoulos’ theory categorizes the patterns found into “paradigms” in a manner similar to the “paradigmatic” music analysis of Ruwet (1972) and Nattiez (1975). Each of these paradigmatic classes corresponds approximately to one of the translational equivalence classes generated by SIATEC.

### 3.3 Solutions for P2

Very little effort has been made so far to solve the problem P2. However, musically this problem setting is more pertinent and useful than P1, and not least in those cases where the piece of music under consideration is polyphonic. The problem is somewhat related to the problem dealt with in molecular biology, where several DNA sequences are to be aligned in an optimal way (and therefore the common subsequences of the considered DNA sequences are to be found).

These problems are inherently different from each other and, in some respects, the discovery of patterns in DNA sequences and proteins is, in fact, somewhat simpler than the discovery of musically significant patterns in polyphonic music. This is because both proteins and nucleic acids can, at least at the primary level of structure, be appropriately modelled as 1-dimensional strings of symbols taken from a highly restricted alphabet. Whereas much polyphonic music cannot even be appropriately represented as a *set* of 1-dimensional strings and the musical “alphabets” used are (at least in principle if not generally in practice) infinite and multidimensional.

Nevertheless, let us roughly present, as an example, a string matching approach to 1-dimensional pattern discovery by Floratos and Rigoutsos (2000). The idea is to start with initial patterns of a given length (possibly containing also ‘don’t care’ characters), and proceed recursively to generate longer and longer patterns appearing in the data set. Floratos and Rigoutsos attempt to avoid the inherent NP-hardness of the problem by limiting the length of the considered patterns. With the aid of two suffix structures, they attempt to extend the considered patterns both backwards (by assigning a prefix to the

	Hsu et al.	Camb.	Roll.	Flor. & Rig.	SIATEC	SIA
Allows gaps	–	–	✓	✓	✓	✓
Approximate matching	–	✓	✓	✓	✓*	✓*
Transposition invariant	$-\sqrt{\ddagger}$	✓	✓	$-\sqrt{\ddagger}$	✓	✓
Equivalence classes	✓	$-\sqrt{\ddagger}$	–	–	✓	–
Considers polyphony	–	–	–	–	✓	✓
Time complexity	$O(n^4)?$	?	$O(n^2)^*$	NP	$O(n^3)$	$O(n^2 \log_2 n)$
Space complexity	?	?	$O(n^2)$	?	$o(n^3)$	$O(n^2)$

\* Finds certain classes of approximate matches (e.g. Figure 1).

‡ The property is achieved by using intervals instead of absolute pitches.

† Actually paradigmatic categories but similar to TECs.

\* But if size of patterns to be discovered is unlimited, this rises to at least  $O(n^4)$ .

Table 1: Table comparing features of a number of pattern-discovery algorithms.

pattern) and forwards (by assigning a suffix). Finally, the generated patterns representing exactly the same subsequences in a different way, are combined in a *maximal pattern*.

In Table 1 we have gathered the properties of the aforementioned, relevant methods so that they can be compared with our SIA and SIATEC algorithms that are presented below.

## 4 The mathematical function that SIATEC computes

In this section we develop an expression for the mathematical function that SIATEC computes. Some well-known mathematical concepts (e.g. *set* and *ordered set*) are not defined here. For definitions and explanations of such well-known concepts see, for example, Borowski and Borwein (1989) or Cormen et al. (1990, chap. 5).

We begin by defining some terms that we shall use frequently from now on.

**Definition 1 (Vector)** *An object may be called a **vector** if and only if it is a finite ordered set of numbers. An object may be called a **k-dimensional vector** if and only if it is a vector of cardinality  $k$ .*

We assume here that every number in a vector is integral, rational or real.

**Definition 2 (Vector set)** *An object may be called a **vector set** if and only if it is a set of vectors. An object may be called a **k-dimensional vector set** if and only if it is a vector set in which every vector has cardinality  $k$ .*

An object may be called a *pattern* or a *dataset* if and only if it is a  $k$ -dimensional vector set. An object may be called a *datapoint* if and only if it is a vector in a pattern or a dataset. We usually reserve the term *dataset* for a  $k$ -dimensional vector set that represents some complete set of data that we are interested in processing. We usually reserve the term *pattern* for either a  $k$ -dimensional vector set that is a subset of some specified dataset or a  $k$ -dimensional vector set that is a transformation of some subset of a dataset. Every pattern is a  $k$ -dimensional vector set and every dataset is a  $k$ -dimensional vector set but we only *call* a  $k$ -dimensional vector set a pattern or a dataset if the vectors it contains are intended to be interpreted as position vectors (i.e. datapoints).

We now define a number of basic concepts and notations relating to ordered sets and, in particular, vectors.

**Definition 3 (Concatenation of ordered sets)** If  $A$  and  $B$  are ordered sets such that

$$A = \langle a_1, a_2, \dots, a_m \rangle \text{ and } B = \langle b_1, b_2, \dots, b_n \rangle$$

then the **concatenation of  $B$  onto  $A$** , denoted by  $A \oplus B$ , is defined to be equal to

$$\langle a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_n \rangle$$

**Definition 4 (Concatenation of a collection of ordered sets)** If

$$S_1, S_2, \dots, S_k, \dots, S_n$$

is a collection of ordered sets then the expression

$$S_1 \oplus S_2 \oplus \dots \oplus S_k \oplus \dots \oplus S_n$$

is defined to be equivalent to

$$\bigoplus_{k=1}^n S_k$$

If  $S$  is a set or ordered set then we denote the cardinality of  $S$  by  $|S|$ .

**Definition 5 (Element of an ordered set)** If  $S$  is an ordered set,

$$S = \langle s_1, s_2, \dots, s_n, \dots \rangle$$

then

$$S[n] = s_n$$

for all integer  $n$  such that  $1 \leq n \leq |S|$ . That is, the expression  $S[n]$  evaluates to the  $n$ th element of  $S$ . If  $S[n]$  is itself an ordered set then  $S[n, m]$  returns the  $m$ th element of  $S[n]$ ,  $S[n, m, l]$  returns the  $l$ th element of  $S[n, m]$  and so on.

**Definition 6 (Addition and subtraction of vectors)** If  $u$  and  $v$  are vectors such that  $|u| = |v| = k$  then:

$$\begin{aligned} u - v &= \bigoplus_{i=1}^k \langle u[i] - v[i] \rangle \\ -v &= \bigoplus_{i=1}^k \langle -v[i] \rangle \\ u + v &= \bigoplus_{i=1}^k \langle u[i] + v[i] \rangle \end{aligned}$$

**Definition 7 (Vector inequality)** If  $u$  and  $v$  are vectors then  $u < v$  if and only if one of the following conditions is satisfied:

1.  $|u| < |v|$ ; or
2.  $|u| = |v| = k$  and there exists an integer  $i$  such that  $1 \leq i \leq k$  and  $u[i] < v[i]$  and  $u[j] = v[j]$  for  $1 \leq j < i$ .

We now define a number of concepts relating to the geometrical transformation of *translation*.

**Definition 8 (Translation of a pattern)** *If  $p$  is a pattern and  $v$  is a vector then the translation of  $p$  by  $v$ , denoted by  $\tau(p, v)$ , is given by the following equation:*

$$\tau(p, v) = \{d_2 \mid (\exists d_1 \mid (d_1 \in p) \wedge (d_1 + v = d_2))\} \quad (1)$$

Note that the expression  $d_1 + v$  on the right-hand side of Eq.1 is a *vector* addition as defined in Definition 6.

**Definition 9 (Translational equivalence)** *If  $p$  and  $q$  are patterns then  $q$  is translationally equivalent to  $p$ , denoted by  $q \equiv_{\tau} p$ , if and only if there exists a vector  $v$  such that  $q = \tau(p, v)$ . That is*

$$q \equiv_{\tau} p \iff \exists v \mid q = \tau(p, v) \quad (2)$$

**Definition 10 (Translational equivalence class of a pattern)** *If  $D$  is a dataset and  $p$  is a pattern such that  $p \subseteq D$  then the translational equivalence class (TEC) of  $p$  in  $D$ , denoted by  $E(p, D)$ , is given by the following equation:*

$$E(p, D) = \{q \mid q \equiv_{\tau} p \wedge q \subseteq D\} \quad (3)$$

**Definition 11 (Maximal translatable pattern (MTP) for a vector)** *If  $v$  is a vector and  $D$  is a dataset then the maximal translatable pattern (MTP) for  $v$  in  $D$ , denoted by  $p(v, D)$ , is given by the following equation:*

$$p(v, D) = \{d \mid d \in D \wedge d + v \in D\} \quad (4)$$

We say that  $p(v, D)$  is a **maximal translatable pattern (MTP) in  $D$**  if and only if  $p(v, D) \neq \emptyset$ .

**Definition 12 (Complete set of maximal translatable patterns)** *If  $D$  is a dataset then the complete set of maximal translatable patterns in  $D$ , denoted by  $P(D)$ , is given by the following equation:*

$$P(D) = \{p(d_1 - d_2, D) \mid d_1, d_2 \in D\} \quad (5)$$

**Definition 13 (Complete set of MTP TECs)** *If  $D$  is a dataset then the complete set of MTP TECs for  $D$ , denoted by  $T(D)$ , is given by the following equation:*

$$T(D) = \{E(p, D) \mid p \in P(D)\} \quad (6)$$

In other words, the complete set of MTP TECs for a dataset  $D$  is the set that only contains every TEC which is the TEC of a maximal translatable pattern in  $D$ . When given a dataset  $D$  as input, SIATEC computes  $T(D)$  as defined in Definition 13.

**Lemma 1** *If  $D$  is a dataset then*

$$T(D) = \{E(p(d_1 - d_2, D), D) \mid d_1, d_2 \in D\} \quad (7)$$

*Proof*

If we substitute Eq.5 into Eq.6 then we find that

$$\begin{aligned} T(D) &= \{E(p, D) \mid p \in \{p(d_1 - d_2, D) \mid d_1, d_2 \in D\}\} \\ &= \{E(p(d_1 - d_2, D), D) \mid d_1, d_2 \in D\} \end{aligned}$$

■

Lemma 1 tells us that if  $d_1$  and  $d_2$  are any two datapoints in a dataset  $D$  then the TEC of the maximal translatable pattern for the vector from  $d_2$  to  $d_1$  will be contained in  $T(D)$ . Moreover, this lemma tells us that if  $E$  is a TEC in  $T(D)$  for some specified dataset  $D$ , then there will exist at least one pair of points  $d_1, d_2 \in D$  such that the maximal translatable pattern  $p(d_1 - d_2, D)$  is in  $E$ .

```

SIATEC(FN : dataset file-name, SD : bit-vector indicating selected dimensions)
1  READ_DATASET(FN,SD)
2  SORT_DATASET
3  SETIFY_DATASET
4  COMPUTE_VECTORS
5  CONSTRUCT_VECTOR_TABLE
6  SORT_VECTORS
7  CONSTRUCT_PATTERN_LIST
8  VECTORIZE_PATTERNS
9  COMPUTE_PATTERN_SIZES
10 SORT_PATTERN_VECTOR_SEQUENCES
11 SETIFY_PATTERN_VECTOR_SEQUENCES
12 COMPUTE_TECS
13 OUTPUT_TECS

```

Figure 2: SIATEC algorithm.

## 5 SIATEC: An overview

We now present SIATEC, an efficient algorithm for computing  $T(D)$  for any dataset  $D$ . The worst-case running time of SIATEC is  $O(kn^3)$  for a  $k$ -dimensional dataset containing  $n$  datapoints. A loose upper bound on the worst-case space complexity of SIATEC is  $O(kn^3)$ . We are currently trying to find a tight upper bound on this space complexity.

We assume that a  $K$ -dimensional dataset  $\mathcal{D}$  of cardinality  $N$  has been saved in a file whose name is given to SIATEC in the parameter FN (see Figure 2). We further assume that we wish to process a  $k$ -dimensional orthogonal projection of  $\mathcal{D}$  defined by the parameter SD.

For example,  $\mathcal{D}$  may be a collection of 3-dimensional datapoints representing points in a 3-d Cartesian co-ordinate system. However, we may only be interested in discovering patterns in a particular two-dimensional orthogonal projection of  $\mathcal{D}$ . If we were only interested in the projection of  $\mathcal{D}$  in the ‘X–Y’ plane, then we would set SD to 110. If we were interested in the projection of  $\mathcal{D}$  in the ‘X–Z’ plane, then we would set SD to 101. We denote this projection of  $\mathcal{D}$  by  $D$  and we denote the cardinality of  $D$  by  $n$ . In general,  $n \leq N$  and  $k \leq K$ .

In lines 1 to 3 of SIATEC (see Figure 2), the dataset is first read into memory, then it is sorted and then duplicate datapoints are removed from it. Sorting the dataset allows us to implement later stages of the algorithm much more efficiently. Setifying the dataset is necessary because, in general, it is possible for more than one of the  $K$ -dimensional datapoints in  $\mathcal{D}$  to be projected onto the same  $k$ -dimensional datapoint in  $D$ . READ\_DATASET has a worst-case running time of  $O(KN)$ . SORT\_DATASET has a worst-case running time of  $O(kN \log_2 N)$ . SETIFY\_DATASET has a worst-case running time of  $O(kN)$ . The total space used by lines 1–3 of SIATEC is  $O(kN)$ .

In line 4, the vector subtraction  $d_1 - d_2$  is computed for all pairs of datapoints  $d_1, d_2$  in the dataset. The worst-case running time and space complexity of this step are both  $O(kn^2)$ .

If  $D$  is a dataset and  $p_1$  and  $p_2$  are patterns, then it follows directly from Definition 10 that

$$p_1 \subseteq D \wedge p_2 \subseteq D \wedge p_1 \equiv_{\tau} p_2 \Rightarrow E(p_1, D) = E(p_2, D) \quad (8)$$

**Lemma 2** *If  $D$  is a dataset and  $v$  is a vector then*

$$\tau(p(v, D), v) = p(-v, D) \quad (9)$$

*Proof*

Definition 11 implies

$$p(-v, D) = \{d \mid d \in D \wedge d - v \in D\} \quad (10)$$

Definition 11 and Definition 8 together imply

$$\begin{aligned} \tau(p(v, D), v) &= \{d_2 \mid (\exists d_1 \mid d_1 \in p(v, D) \wedge d_1 + v = d_2)\} \\ &= \{d_2 \mid (\exists d_1 \mid d_1 \in D \wedge d_1 + v \in D \wedge d_1 + v = d_2)\} \\ &= \{d_2 \mid (\exists d_1 \mid d_1 \in D \wedge d_2 \in D \wedge d_1 = d_2 - v)\} \\ &= \{d_2 \mid (\exists d_1 \mid d_2 \in D \wedge d_2 - v \in D)\} \\ &= \{d_2 \mid d_2 \in D \wedge d_2 - v \in D\} \end{aligned} \quad (11)$$

Eq.11 and Eq.10 together imply

$$\tau(p(v, D), v) = p(-v, D)$$

■

Lemma 2 tells us that if we translate by  $v$  the maximal translatable pattern for  $v$  then we get the maximal translatable pattern for the vector  $-v$ .

**Lemma 3** *If  $D$  is a dataset and  $d_1, d_2 \in D$  then*

$$E(p(d_1 - d_2, D), D) = E(p(d_2 - d_1, D), D) \quad (12)$$

*Proof*

If  $D$  is a dataset and  $d_1, d_2 \in D$  then Definition 12 implies that  $p(d_1 - d_2, D) \in P(D)$  and  $p(d_2 - d_1, D) \in P(D)$ . Lemma 2 tells us that

$$\tau(p(d_1 - d_2, D), d_1 - d_2) = p(d_2 - d_1, D)$$

Therefore, by Definition 9,

$$p(d_1 - d_2, D) \equiv_{\tau} p(d_2 - d_1, D)$$

and consequently, by Eq.8,

$$E(p(d_1 - d_2, D), D) = E(p(d_2 - d_1, D), D)$$

■

Since the goal of SIATEC is to compute the complete set of MTP TECs,  $T(D)$ , there is no point in us computing both  $E(p(d_1 - d_2, D), D)$  and  $E(p(d_2 - d_1, D), D)$  for each pair of datapoints  $d_1, d_2 \in D$  since we know from Eq.12 that for any given pair of datapoints these two TECs are the same. Consequently, there is no need for us to compute both  $p(d_1 - d_2, D)$  and  $p(d_2 - d_1, D)$  for every pair of datapoints  $d_1, d_2 \in D$ —we only need to compute one of these MTPs for each pair of datapoints.

Therefore, in lines 5–7 of SIATEC (Figure 2), instead of computing  $P(D)$ , the complete set of maximal translatable patterns, we compute the set  $P'(D)$  which is defined as follows.

**Definition 14** *If  $D$  is a dataset then*

$$P'(D) = \{p(d_1 - d_2, D) \mid d_1, d_2 \in D \wedge d_1 > d_2\} \quad (13)$$

The naïve algorithm for computing  $P(D)$  (see Definition 12) for a dataset containing  $n$  datapoints would involve computing  $p(d_1 - d_2, D)$  for  $n^2$  vectors. However, to compute  $P'(D)$  we only have to do this for the  $\frac{n(n-1)}{2}$  datapoint pairs that satisfy the condition  $d_1 > d_2$ . The worst-case running time for computing  $P'(D)$  is therefore less than half that of computing  $P(D)$ . Since SIATEC computes  $P'(D)$  instead of  $P(D)$ , the set of TECs generated by SIATEC for a dataset  $D$  is, in fact, not  $T(D)$  but  $T'(D)$  which is defined as follows.

**Definition 15** *If  $D$  is a dataset then*

$$T'(D) = \{E(p, D) \mid p \in P'(D)\} \quad (14)$$

It can be shown (see Lemma 6) that  $P'(D)$  does not contain the maximal translatable pattern in  $D$  for the zero vector,  $\mathbf{0}$ . Clearly, every point in a dataset can be translated by the zero vector to give another point in the dataset. Therefore the MTP for the zero vector is equal to the complete dataset, that is,

$$p(\mathbf{0}, D) = D \quad (15)$$

Moreover, it can be shown that if  $D$  is a non-empty dataset and  $d_1$  and  $d_2$  are two distinct datapoints in  $D$  then the maximal translatable pattern in  $D$  for the vector  $d_1 - d_2$  is never equal to the whole dataset  $D$ . We will now prove this.

**Lemma 4** *If  $D$  is a dataset then*

$$D \neq \emptyset \wedge d_1 \neq d_2 \Rightarrow p(d_1 - d_2, D) \neq D \quad (16)$$

*Proof*

Let us denote by

$$\Delta = \langle \delta_1, \delta_2, \dots, \delta_n \rangle$$

the ordered set that results from sorting the dataset

$$D = \{d_1, d_2, \dots, d_n\}$$

so that all the datapoints are in increasing order. Let us now assume that  $p(d_1 - d_2, D) = D$  for some pair of datapoints  $d_1, d_2 \in D, d_1 \neq d_2$ . We know that there is no datapoint greater than  $\delta_n$  in  $D$ . However, if  $d_1 > d_2$ , then  $\delta_n + d_1 - d_2 > \delta_n$  therefore

$$d_1 > d_2 \Rightarrow \delta_n \notin p(d_1 - d_2, D) \Rightarrow p(d_1 - d_2, D) \neq D \quad (17)$$

Similarly, we know that there is no datapoint less than  $\delta_1$  in  $D$ . However, if  $d_1 < d_2$ , then  $\delta_1 + d_1 - d_2 < \delta_1$  therefore

$$d_1 < d_2 \Rightarrow \delta_1 \notin p(d_1 - d_2, D) \Rightarrow p(d_1 - d_2, D) \neq D \quad (18)$$

Eq.17 and Eq.18 together imply that

$$D \neq \emptyset \wedge d_1 \neq d_2 \Rightarrow p(d_1 - d_2, D) \neq D$$

■

For a given dataset  $D$ , the set  $T'(D)$  has the following property:

$$T'(D) = T(D) \setminus \{\{D\}\}$$

In other words,  $T'(D)$  is the relative complement of  $\{\{D\}\}$  in  $T(D)$ , or, to put it yet another way,  $T'(D)$  only contains all the elements of  $T(D)$  except  $\{D\} = E(D, D) = E(p(\mathbf{0}, D), D)$ . The following two lemmas prove this result.

**Lemma 5** *If  $D$  is a dataset then*

$$T'(D) = \{E(p(d_1 - d_2, D), D) \mid d_1, d_2 \in D \wedge d_1 \neq d_2\} \quad (19)$$

*Proof*

Eq.14 and Eq.13 together imply

$$T'(D) = \{E(p(d_1 - d_2, D), D) \mid d_1, d_2 \in D \wedge d_1 > d_2\} \quad (20)$$

Eq.12 and Eq.20 together imply

$$\begin{aligned} T'(D) &= \{E(p(d_2 - d_1, D), D) \mid d_1, d_2 \in D \wedge d_1 > d_2\} \\ &= \{E(p(d_1 - d_2, D), D) \mid d_1, d_2 \in D \wedge d_1 < d_2\} \end{aligned} \quad (21)$$

Eq.20 and Eq.21 imply

$$T'(D) = \{E(p(d_1 - d_2, D), D) \mid d_1, d_2 \in D \wedge d_1 \neq d_2\}$$

■

**Lemma 6** *If  $D$  is a non-empty dataset then*

$$T'(D) = T(D) \setminus \{\{D\}\} \quad (22)$$

*Proof*

From Eq.15 and Lemma 4 it follows that if  $D \neq \emptyset$  then

$$p(d_1 - d_2, D) = D \iff d_1 = d_2 \quad (23)$$

Lemma 5 and Lemma 1 imply

$$T(D) = T'(D) \cup \{E(p(\mathbf{0}, D), D)\}$$

and from Eq.15 we know that  $p(\mathbf{0}, D) = D$ . Therefore

$$\begin{aligned} T(D) &= T'(D) \cup \{E(D, D)\} \\ &= T'(D) \cup \{\{D\}\} \end{aligned} \quad (24)$$

Eq.23 and Lemma 5 imply  $E(D, D) \notin T'(D)$  which in turn implies

$$\{\{D\}\} \notin T'(D) \quad (25)$$

Eq.24 and Eq.25 imply

$$T'(D) = T(D) \setminus \{\{D\}\}$$

■

Since  $T'(D)$  contains all and only the TECs in  $T(D)$  except for  $E(D, D)$  and since in general we are not interested in  $E(D, D)$ , it is more efficient (and perfectly sufficient) for SIATEC to compute the set  $P'(D)$  instead of the set  $P(D)$ .

Each time a vector  $v = d_1 - d_2$  is computed by COMPUTE\_VECTORS (line 4 of SIATEC (Figure 2)), the vector  $v$  is stored in a linked list node that has a pointer to the vector's 'source' datapoint  $d_2$ . This feature, together with the fact that the dataset is sorted before COMPUTE\_VECTORS is called, allows  $P'(D)$  to be computed simply by sorting the vectors computed in line 4. This can be achieved in a worst-case running time of  $O(kn^2 \log_2 n)$  and a worst-case space complexity of  $O(kn^2)$ .

Eq.8 tells us that if two patterns are translationally equivalent then their TECs will be identical. As already explained, SIATEC computes the set  $T'(D) = \{E(p, D) \mid p \in P'(D)\}$  (Eq.14). However, it is possible in general for  $P'(D)$  to contain distinct but translationally equivalent patterns. Clearly, if  $p_1, p_2 \in P'(D)$

and  $p_1 \equiv_\tau p_2$  then there is no point in computing both  $E(p_1, D)$  and  $E(p_2, D)$ . Therefore, before computing  $T'(D)$ , we first compute the partition

$$\mathcal{P}(D) = \{\mathcal{E}(p, P'(D)) \mid p \in P'(D)\} \quad (26)$$

where

$$\mathcal{E}(p, P'(D)) = \{q \mid q \in P'(D) \wedge q \equiv_\tau p\} \quad (27)$$

We then construct a set  $P''(D)$  which contains exactly one pattern from each  $\mathcal{E} \in \mathcal{P}(D)$ . This guarantees that there are no two patterns in  $P''(D)$  that are translationally equivalent.

This process of generating a set  $P''(D)$  from  $P'(D)$  is implemented in lines 8–11 of SIATEC. The worst-case running time and worst-case space complexity of lines 8–11 of SIATEC are  $O(kn^2 \log_2 n)$  and  $O(kn^2)$  respectively.

$P''(D)$  can be computed from  $P'(D)$  by simply examining each pattern  $p$  in  $P'(D)$  and removing  $p$  if and only if one of the remaining patterns in  $P'(D)$  is translationally equivalent to  $p$ . From Eq.8 we know that if two patterns are translationally equivalent then their TECs will also be the same. Let's say that we initialize  $A$  to be equal to  $P'(D)$  and then we examine each pattern  $p$  in  $A$  in turn and remove  $p$  if and only if there is an as yet unscanned pattern remaining in  $A$  that is translationally equivalent to  $p$ . We know that whenever a pattern  $p$  is removed from  $A$  during this process of computing  $P''(D)$  there is always a pattern remaining in  $A$  whose TEC is the same as that of  $p$ . Therefore we know that by the time the process has completed and  $A = P''(D)$ ,

$$\{E(p, D) \mid p \in P'(D)\} = \{E(p, D) \mid p \in P''(D)\}$$

which, together with Definition 15 implies that

$$T'(D) = \{E(p, D) \mid p \in P''(D)\} \quad (28)$$

The function COMPUTE\_TECS called in line 12 of SIATEC computes  $T'(D)$  by computing the TEC of each pattern in  $P''(D)$ . COMPUTE\_TECS achieves this in a worst-case running time of  $O(kn^3)$ . A loose upper bound on the worst-case space complexity of COMPUTE\_TECS is  $O(kn^3)$ . But in practice it seems to be much better than this and the tight upper bound may be as low as  $O(kn^2)$ .

## 6 SIATEC: A closer look

### 6.1 The data structures used

The implementation of SIATEC described here uses linked list data structures. Three different types of node are used to construct the data structures used in SIATEC: NUMBER\_NODES, VECTOR\_NODES and PATTERN\_NODES. These types are defined in Figure 3.

NUMBER\_NODES are used to construct linked lists that represent vectors. Each NUMBER\_NODE has two fields, one called **number** and the other called **next**. The **number** field of a NUMBER\_NODE is used to hold a numerical value. The **next** field is a NUMBER\_NODE pointer used to point to the node that holds the next element in the vector. A NUMBER\_NODE is represented diagrammatically as a rectangular box divided into two cells (see Figure 4). The left-hand cell represents the **number** field and the right-hand cell represents the **next** field. A cell with a diagonal line drawn across it represents a pointer whose value is NULL. The pointer  $v$  in Figure 4 heads a linked list of NUMBER\_NODES that represents the vector  $\langle 3, 4 \rangle$ .

VECTOR\_NODES are used to construct linked lists that represent vector sets, such as patterns and datasets. Each VECTOR\_NODE has three fields: a NUMBER\_NODE pointer called **vector** and two VECTOR\_NODE pointers, one called **down** and the other called **right** (see definition in Figure 3). A VECTOR\_NODE is represented diagrammatically as a rectangular box divided into three cells (see Figure 5). The left-hand cell represents

```

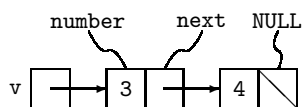
types
  NUMBER_NODE
    number : a numerical value
    next   : a NUMBER_NODE pointer

  VECTOR_NODE
    vector : a NUMBER_NODE pointer
    down, right : VECTOR_NODE pointers

  PATTERN_NODE
    vec_seq, pattern, vectors : VECTOR_NODE pointers
    down, right : PATTERN_NODE pointers
    size : an integer

global variables
  D : a VECTOR_NODE pointer used to head the list representing the
    dataset
  V : a VECTOR_NODE pointer used to head the table of vectors used
    in finding patterns
  P : a PATTERN_NODE pointer used to head the list of patterns

```

Figure 3: *Types and global variables.*Figure 4: *Using NUMBER\_NODES to represent vectors.*

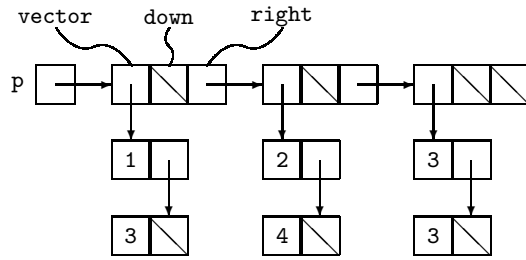


Figure 5: A right-directed list of VECTOR\_NODES.

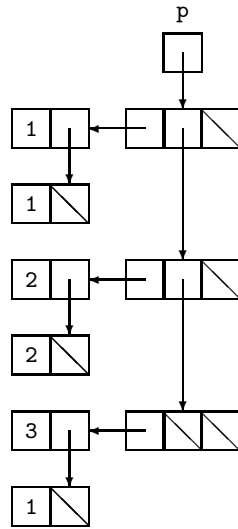


Figure 6: A down-directed list of VECTOR\_NODES.

the **vector** field, the middle cell represents the **down** field and the right-hand cell represents the **right** field. The field called **vector** is always used to head a linked list of **NUMBER\_NODES** representing a vector. The **right** field is used to point to the next **VECTOR\_NODE** in a *right-directed list* such as the one shown in Figure 5. The **down** field is used to point to the next **VECTOR\_NODE** in a *down-directed list* such as the one shown in Figure 6. The linked list in Figure 5 could be used to represent the ordered set of vectors  $\langle\langle 1, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 3 \rangle\rangle$  or, of course, the vector set  $\{\langle 1, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 3 \rangle\}$ . The linked list in Figure 6 could be used to represent the ordered vector set  $\langle\langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 1 \rangle\rangle$  or the vector set  $\{\langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 1 \rangle\}$ . The fact that each **VECTOR\_NODE** has both a **down** and a **right** field allows for a linked list of **VECTOR\_NODES** to be efficiently sorted using an implementation of merge sort that converts an unsorted down-directed list into a sorted right-directed list (see the algorithms **SORT\_DATASET** (Figure 12) and **SORT\_VECTORS** (Figure 26)).

A **PATTERN\_NODE** is used to hold information about an individual pattern. It has six fields (see definition in Figure 3) and is thus represented diagrammatically as a rectangle divided into six cells as shown in Figure 7. The left-most cell represents the **vec\_seq** field, the next cell to the right represents the **vectors** field and the subsequent cells represent, in order, the **size**, **down**, **right** and **pattern** fields.

By the time that **OUTPUT\_TPCS** is called in line 13 of **SIATEC** (Figure 2), the global **PATTERN\_NODE** pointer variable **P** (see Figure 3) heads a right-directed linked list of **PATTERN\_NODES** in which each node

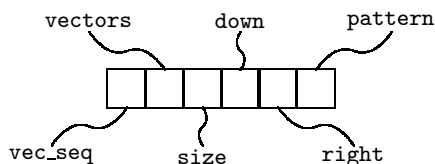


Figure 7: The structure of a PATTERN\_NODE.



Figure 8: A simple two-dimensional dataset.

represents one complete MTP TEC. Figure 9 shows the data structure generated by SIATEC for the simple two-dimensional dataset shown in Figure 8. The data structure in Figure 9 consists of a right-directed list of three PATTERN\_NODES headed by P. This list as a whole represents  $T'(D)$  for the dataset represented by the right-directed list headed by the global variable D. Each node in the PATTERN\_NODE list headed by P represents the MTP TEC  $E(p, D)$  for a single pattern in  $P''(D)$ . For example, the first node in this list (the node pointed to by P) represents the TEC

$$\{\langle 1, 1 \rangle, \langle 3, 1 \rangle\}, \{\langle 1, 3 \rangle, \langle 3, 3 \rangle\} \quad (29)$$

In this implementation, a TEC  $E(p, D)$  is represented in a compact form as an ordered pair  $\langle p, V(p, D) \rangle$  where

$$V(p, D) = \{v \mid \tau(p, v) \subseteq D\} \quad (30)$$

In a PATTERN\_NODE, the pattern  $p$  is stored as a linked list headed by the `pattern` field and the set of vectors  $V(p, D)$  is stored as a list headed by the `vectors` field. Thus, the node pointed to by P in Figure 9 represents the TEC in Eq.29 by means of the ordered pair

$$\langle \{\langle 1, 1 \rangle, \langle 3, 1 \rangle\}, \{\langle 0, 0 \rangle, \langle 0, 2 \rangle\} \rangle$$

The `pattern` field points to a linked list of VECTOR\_NODES that represents a pattern, each VECTOR\_NODE storing a datapoint in the pattern. However, the datapoint associated with a VECTOR\_NODE in such a list is not stored explicitly as a linked list of NUMBER\_NODES headed by the `vector` field of the node. Instead, to save space, the `down` field of each VECTOR\_NODE in a pattern list is used to point to the appropriate node in the dataset list headed by the global variable D (see Figure 9).

A PATTERN\_NODE has both a `down` and a `right` field for the same reason that a VECTOR\_NODE has these fields: it allows a list of patterns to be sorted efficiently using an implementation of merge sort that converts an unsorted down-directed list into a sorted right-directed list (see the SORT\_PATTERN\_VECTOR\_SEQUENCES algorithm in Figure 35).

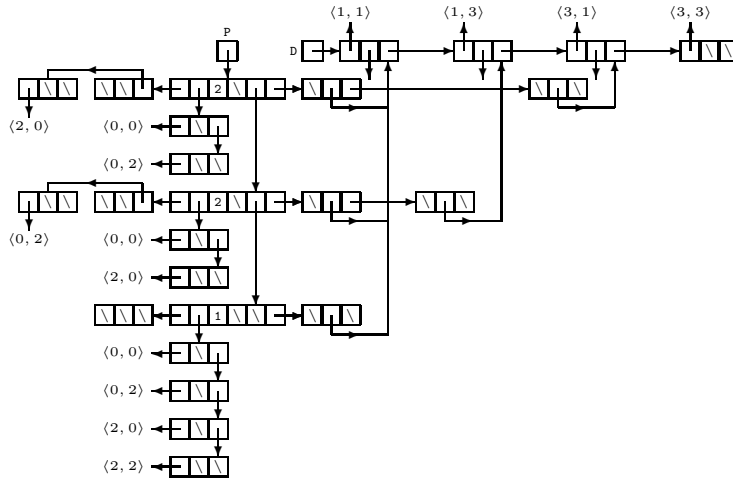


Figure 9: An example of a data structure generated by COMPUTE\_TCS.

The `vec_seq` field in a `PATTERN_NODE` is used to store an intervallic representation of the pattern stored in the `pattern` field. This intervallic representation of the pattern is used together with the cardinality of the pattern which is stored in the `size` field to enable a set  $P''(D)$  to be derived more efficiently from  $P(D)$ .

### 6.2 Reading and preparing the dataset

Line 1 of `SIATEC` calls the procedure `READ_DATASET` which is given in Figure 10. The pseudo-code used here should be easy to read for anyone who has written programs in C or Pascal using linked list data structures. If  $x$  is a pointer variable then the expression  $x \uparrow y$  denotes the field called  $y$  in the node pointed to by  $x$ . The expression  $x \leftarrow y$  should be read “ $x$  becomes equal to  $y$ ”. Block structure is indicated by indentation.

$\mathcal{D}$ ,  $N$ ,  $K$ ,  $D$ ,  $k$  and  $n$  are as defined on page 9 above. For each of the  $N$   $K$ -dimensional datapoints  $p$  in  $F$  (the file whose name is `FN`—see lines 2 and 5 of `READ_DATASET`), `READ_DATASET` reads  $p$ , computes the required  $k$ -dimensional projection of  $p$  and stores the resulting  $k$ -dimensional datapoint in a down-directed linked list of `VECTOR_NODES`. This list of `VECTOR_NODES` is headed by the global variable `D` (see Figure 3). For example, if

$$\langle \langle 3, 3, 3 \rangle, \langle 3, 3, 2 \rangle, \langle 3, 1, 3 \rangle, \langle 1, 1, 3 \rangle, \langle 1, 1, 4 \rangle, \langle 1, 3, 5 \rangle \rangle$$

is the sequence of 3-d datapoints stored in a file called “filename.dat” then the procedure call

```
READ_DATASET("filename.dat", 110)
```

would result in the linked list shown in Figure 11. Note that each vector in Figure 11 (and all the other data structure diagrams that follow) is actually a linked list of `NUMBER_NODES` but to draw these in full would clutter the diagrams.

It is assumed that the function `OPEN_FILE` (Figure 10, line 5) attempts to open the file whose name is `FN` returning a pointer to the beginning of the file if it succeeds and `NULL` if it does not. The function `READ_DATAPOINT` (lines 8 and 12) reads the next  $K$ -dimensional datapoint from the file `F` returning either the required orthogonal projection of this datapoint or `NULL` if the end of the file has been reached. The function `MAKE_NEW_VECTOR_NODE` (lines 9 and 13) simply allocates a new `VECTOR_NODE`, initializes all its fields to `NULL` and returns a pointer to the new node.

```

READ_DATASET(FN : dataset filename, SD : bit-vector indicating selected dimensions)
1   local variables
2     F : a file containing a dataset
3     d : a pointer to a NUMBER_NODE
4     p : a VECTOR_NODE pointer

5   if (F ← OPEN_FILE(FN)) = NULL
6     EXIT
7   D ← NULL
8   if (d ← READ_DATAPOINT(F,SD)) ≠ NULL
9     D ← MAKE_NEW_VECTOR_NODE
10    p ← D
11    p↑vector ← d
12    while (d ← READ_DATAPOINT(F,SD)) ≠ NULL
13      p↑down ← MAKE_NEW_VECTOR_NODE
14      p ← p↑down
15      p↑vector ← d
16  CLOSE_FILE(F)

```

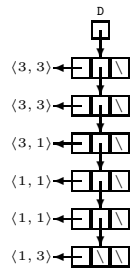
Figure 10: READ\_DATASET *algorithm*.

Figure 11: An example linked list of VECTOR\_NODES constructed by READ\_DATASET.

```

SORT_DATASET
1   local variables
2   ABOVE_A, A, B, BELOW_B, C : VECTOR_NODE pointers

3   while D ≠ NULL and D↑down ≠ NULL
4   ABOVE_A ← NULL
5   A ← D
6   D ← NULL
7   repeat
8   if D ≠ NULL
9   ABOVE_A↑down ← NULL
10  B ← A↑down
11  A↑down ← NULL
12  BELOW_B ← B↑down
13  B↑down ← NULL
14  C ← MERGE_DATASET_ROWS(A,B)
15  if D = NULL
16  D ← C
17  else
18  ABOVE_A↑down ← C
19  C↑down ← BELOW_B
20  ABOVE_A ← C
21  A ← ABOVE_A↑down
22  until A = NULL or A↑down = NULL

```

Figure 12: SORT\_DATASET algorithm.

The worst-case time complexity of READ\_DATASET is clearly  $O(KN)$  since it involves reading  $N$  vectors each containing  $K$  numbers. Its worst-case space complexity is  $O(kN)$  since only  $k$  of the  $K$  numbers read from the file for each datapoint are actually stored in memory.

The efficiency of this implementation of SIATEC depends upon the dataset being sorted and this sorting is done in line 2 of SIATEC using the procedure SORT\_DATASET shown in Figure 12. This procedure is an implementation of merge sort that converts the unsorted down-directed list generated by READ\_DATASET into a sorted right-directed list. On the first iteration of the outer while loop (lines 3–22), SORT\_DATASET scans the down-directed list of unsorted datapoints, merging each pair of consecutive datapoints into a single, sorted, right-directed list. For example, Figure 13 shows the state of the linked list  $D$  after one iteration of the outer while loop has been completed on the dataset list shown in Figure 11. On subsequent iterations, each pair of adjacent right-directed lists is merged into a single list and the process continues until the whole list has been merged into a single, sorted, right-directed list. The merging process is carried out by the algorithm MERGE\_DATASET\_ROWS shown in Figure 14. Figure 15 shows the right-directed list produced by SORT\_DATASET from the down-directed list shown in Figure 11.

The algorithm MERGE\_DATASET\_ROWS is an implementation of the standard merge technique used in merge sort. The function VL called in lines 5 and 14 of MERGE\_DATASET\_ROWS takes two NUMBER\_NODE pointer arguments, each representing a vector. The procedure call  $VL(v_1, v_2)$  returns TRUE if and only if  $v_1 < v_2$  (vector inequality is defined in Definition 7). It is well-known that the worst-case running time for merge sort to sort a list of  $n$  items is  $O(n \log_2 n)$ . The worst-case running time of SORT\_DATASET is  $O(kN \log_2 N)$  where  $k$  is the dimensionality of the required orthogonal projection  $D$  of the input dataset  $\mathcal{D}$  (see page 9) and  $N$  is the cardinality of  $\mathcal{D}$ . This follows directly from two facts: 1) there are  $N$  items in the dataset list headed by  $D$ ; and 2) each comparison carried out by VL takes  $O(k)$  time. SORT\_DATASET

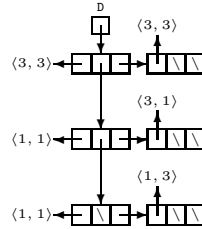


Figure 13: The state of the linked list D after one iteration of the outer while loop of SORT\_DATASET on the dataset list in Figure 11.

```

MERGE_DATASET_ROWS(A, B : VECTOR_NODE pointers)
1   local variables
2   a, b, C, c : VECTOR_NODE pointers

3   a ← A
4   b ← B
5   if VL(a↑vector, b↑vector)
6   C ← a
7   a ← a↑right
8   else
9   C ← b
10  b ← b↑right
11  C↑right ← NULL
12  c ← C
13  while a ≠ NULL and b ≠ NULL
14  if VL(a↑vector, b↑vector)
15  c↑right ← a
16  a ← a↑right
17  else
18  c↑right ← b
19  b ← b↑right
20  c ← c↑right
21  c↑right ← NULL
22  if a = NULL
23  c↑right ← b
24  else
25  c↑right ← a
26  return C
    
```

Figure 14: MERGE\_DATASET\_ROWS algorithm.

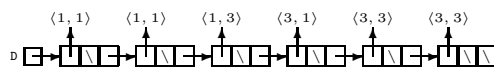


Figure 15: The sorted, right-directed linked list produced by SORT\_DATASET from the unsorted, down-directed dataset list in Figure 11.

```

SETIFY_DATASET
1   local variables
2   d1,d2 : VECTOR_NODE pointers

3   d1 ← D
4   while d1 ≠ NULL and d1↑right ≠ NULL
5     if VE(d1↑right↑vector,d1↑vector)
6       ▷ Delete d1↑right.
7       d2 ← d1↑right
8       d1↑right ← d2↑right
9       d2↑right ← NULL
10      d2 ← DISPOSE_OF_VECTOR_NODE(d2)
11    else
12      d1 ← d1↑right

```

Figure 16: SETIFY\_DATASET algorithm.

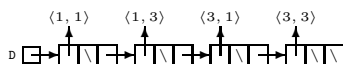


Figure 17: The linked list that results when SETIFY\_DATASET has been executed on the linked list in Figure 15.

simply rearranges the nodes in the linked list created by READ\_DATASET and therefore uses no extra space.

As explained on page 9, it is possible for more than one of the  $K$ -dimensional datapoints in  $\mathcal{D}$  to be projected onto the same  $k$ -dimensional datapoint in  $D$ . This means that the right-directed list headed by  $D$  that results after the execution of SORT\_DATASET may contain duplicate datapoints. If this is so, then these datapoints will clearly be adjacent to each-other in the list and therefore can be removed using the simple procedure SETIFY\_DATASET shown in Figure 16. This procedure determines for each datapoint  $d_i$  whether or not it is equal to the one that precedes it in the list ( $d_{i-1}$ ) (line 5). If this is the case, then  $d_i$  is deleted from the list. Otherwise we proceed to comparing  $d_{i+1}$  with  $d_i$ . Each datapoint in the list is therefore compared with one other preceding datapoint using the function VE which returns TRUE if and only if the two datapoints are equal. VE runs in  $O(k)$  time therefore the worst-case running time of SETIFY\_DATASET is  $O(kN)$ . SETIFY\_DATASET reduces the amount of space used by the linked list  $D$  from  $O(kN)$  to  $O(kn)$ . Figure 17 shows the linked list that results after SETIFY\_DATASET has been executed on the sorted right-directed dataset list shown in Figure 15.

### 6.3 Computing all inter-datapoint vectors

Definition 14 suggests that computing  $P'(D)$  would require computing the set

$$\mathcal{V}'(D) = \{d_1 - d_2 \mid d_1, d_2 \in D \wedge d_1 > d_2\} \quad (31)$$

which is less than half the size of the set

$$\mathcal{V}(D) = \{d_1 - d_2 \mid d_1, d_2 \in D\} \quad (32)$$

In a previous version of SIATEC, we did indeed only compute  $\mathcal{V}'(D)$ . However, we then discovered that computing the full set  $\mathcal{V}(D)$  allowed us to use a significantly more efficient implementation of the procedure COMPUTE\_TECS which is called in line 12 of SIATEC (see section 6.6 below). Indeed, by using  $\mathcal{V}(D)$  instead

```

COMPUTE_VECTORS
1   local variables
2    $d_1, d_2, p, v$  : VECTOR_NODE pointers

3    $V \leftarrow \text{NULL}$ 
4   if  $D \neq \text{NULL}$  and  $D \uparrow \text{right} \neq \text{NULL}$ 
5      $d_1 \leftarrow D$ 
6     while  $d_1 \neq \text{NULL}$ 
7        $p \leftarrow d_1$ 
8        $d_2 \leftarrow D$ 
9       while  $d_2 \neq \text{NULL}$ 
10         $\triangleright$  Make new VECTOR_NODE under  $d_1$ .
11         $p \uparrow \text{down} \leftarrow \text{MAKE\_NEW\_VECTOR\_NODE}$ 
12         $p \leftarrow p \uparrow \text{down}$ 
13         $\triangleright$  Connect  $p$  to  $d_1$ .
14         $p \uparrow \text{right} \leftarrow d_1$ 
15         $p \uparrow \text{vector} \leftarrow \text{VM}(d_2 \uparrow \text{vector}, d_1 \uparrow \text{vector})$ 
16        if  $\text{VE}(d_1 \uparrow \text{vector}, d_2 \uparrow \text{vector})$  and  $d_1 \uparrow \text{right} \neq \text{NULL}$ 
17          if  $V = \text{NULL}$ 
18             $V \leftarrow \text{MAKE\_NEW\_VECTOR\_NODE}$ 
19             $v \leftarrow V$ 
20          else
21             $v \uparrow \text{right} \leftarrow \text{MAKE\_NEW\_VECTOR\_NODE}$ 
22             $v \leftarrow v \uparrow \text{right}$ 
23             $v \uparrow \text{down} \leftarrow p$ 
24             $d_2 \leftarrow d_2 \uparrow \text{right}$ 
25             $d_1 \leftarrow d_1 \uparrow \text{right}$ 

```

Figure 18: COMPUTE\_VECTORS algorithm.

of  $\mathcal{V}(D)$  we were able to reduce the worst-case running time of COMPUTE\_TECS from  $O(kn^3 \log_2 n)$  in our previous version to  $O(kn^3)$  in the version described here. In our implementation,  $\mathcal{V}(D)$  is computed using the procedure COMPUTE\_VECTORS shown in Figure 18.

Recall that we denote by  $\Delta = \langle \delta_1, \delta_2, \dots, \delta_n \rangle$  the ordered set that results from sorting the dataset  $D = \{d_1, d_2, \dots, d_n\}$  so that all the datapoints are in increasing order. COMPUTE\_VECTORS effectively computes the table of vectors shown in Table 2. Because the dataset is sorted each row of inter-datapoint vectors in this table is sorted in increasing order from left to right and each column is sorted in increasing order from top to bottom. These two features are used to compute  $P'(D)$  and  $T'(D)$  more efficiently.

In this implementation,  $\Delta$  is represented by the right-directed dataset list headed by  $D$  that results after SETIFY\_DATASET has been executed in line 3 of SIATEC (see Figure 19). Figure 20 shows the structure computed by COMPUTE\_VECTORS for the dataset list shown in Figure 19. This data structure is essentially a linked list representation of Table 2. Let us denote by  $\Phi_v$  the VECTOR\_NODE in the structure in Figure 20

Figure 19: The linked list representation of the sorted dataset  $\Delta$ .

		From				
		$\delta_1$	$\delta_2$	$\dots$	$\delta_{n-1}$	$\delta_n$
To	$\delta_1$	$\delta_1 - \delta_1$	$\delta_1 - \delta_2$	$\dots$	$\delta_1 - \delta_{n-1}$	$\delta_1 - \delta_n$
	$\delta_2$	$\delta_2 - \delta_1$	$\delta_2 - \delta_2$	$\dots$	$\delta_2 - \delta_{n-1}$	$\delta_2 - \delta_n$
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
	$\delta_{n-1}$	$\delta_{n-1} - \delta_1$	$\delta_{n-1} - \delta_2$	$\dots$	$\delta_{n-1} - \delta_{n-1}$	$\delta_{n-1} - \delta_n$
	$\delta_n$	$\delta_n - \delta_1$	$\delta_n - \delta_2$	$\dots$	$\delta_n - \delta_{n-1}$	$\delta_n - \delta_n$

Table 2: The table of vectors represented by the data structure generated by COMPUTE\_VECTORS.

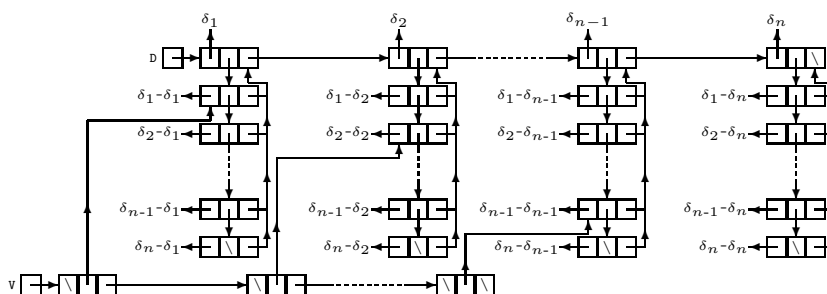


Figure 20: The data structure generated by COMPUTE\_VECTORS.

whose `vector` field stores the result of evaluating a sentence denoted by  $v$ . `COMPUTE_VECTORS` constructs for each datapoint node  $\Phi_{\delta_i}$  a down-directed list of `VECTOR_NODES`,  $\Phi_{\delta_1 - \delta_i}$  down to  $\Phi_{\delta_n - \delta_i}$ , headed by  $\Phi_{\delta_i} \uparrow \text{down}$  (see Figure 20).

Note that

$$\Phi_{\delta_j - \delta_i} \uparrow \text{right} = \Phi_{\delta_i}$$

for each node  $\Phi_{\delta_j - \delta_i}$ . Note also that `COMPUTE_VECTORS` constructs the right-directed linked list `V` which stores the node  $\Phi_{\delta_i - \delta_i}$  for each datapoint  $\delta_i$ . These two features are used to compute the set  $P'(D)$  more efficiently (see section 6.4 below).

`COMPUTE_VECTORS` calculates the vector  $d_1 - d_2$  for all pairs of datapoints  $d_1, d_2 \in D$ , making a total of  $n^2$  vectors to be computed. Each of these vector subtractions is carried out by the function `VM` called in line 15. This function takes two `NUMBER_NODE` pointer arguments, each representing a vector. The call `VM(v_1, v_2)` returns a pointer to a `NUMBER_NODE` list representing the vector  $v_1 - v_2$ . Each of these  $n^2$  vector subtractions takes  $O(k)$  time (recall that  $k$  is the dimensionality of  $D$ , the required orthogonal projection of  $\mathcal{D}$ ). This implies that the worst-case running time of `COMPUTE_VECTORS` is  $O(kn^2)$ . The resulting data structure, as can be seen in Figure 20, occupies  $O(kn^2)$  space.

## 6.4 Computing $P'(D)$

Lines 5 to 7 of `SIATEC` compute the set  $P'(D)$  defined in Eq.13 above. The fact that

$$\Phi_{\delta_j - \delta_i} \uparrow \text{right} = \Phi_{\delta_i}$$

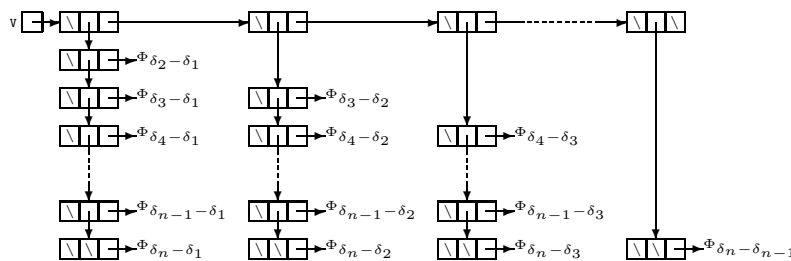
```

CONSTRUCT_VECTOR_TABLE
1   local variables
2   p, v, w : VECTOR_NODE pointers

3   p ← V
4   while p ≠ NULL
5     v ← p↑down↑down
6     w ← p
7     while v ≠ NULL
8       w↑down ← MAKE_NEW_VECTOR_NODE
9       w ← w↑down
10    w↑right ← v
11    v ← v↑down
12    p ← p↑right

```

Figure 21: CONSTRUCT\_VECTOR\_TABLE algorithm.

Figure 22: The data structure  $V$  constructed by CONSTRUCT\_VECTOR\_TABLE.

in the data structure shown in Figure 20 means that  $P'(D)$  can be computed simply by sorting the nodes corresponding to the vectors below the leading diagonal of Table 2.<sup>2</sup> However, the efficient computation of  $T'(D)$  carried out by COMPUTE\_TECs in line 12 of SIATEC relies upon the fact that the down-directed VECTOR\_NODE lists ( $\Phi_{\delta_1-\delta_i}$  to  $\Phi_{\delta_n-\delta_i}$  in Figure 20) remain sorted in increasing order from top to bottom. Therefore we cannot actually change the order of the nodes in Figure 20. To get around this problem, we use the linked list  $V$  in Figure 20 and the procedure CONSTRUCT\_VECTOR\_TABLE (Figure 21) to construct the data structure shown in Figure 22. This data structure is a representation of the region of Table 2 below the leading diagonal and the nodes in this data structure can safely be sorted without disturbing the arrangement in Figure 20.

The data structure in Figure 22 consists of  $n - 1$  down-directed lists. The list headed by the **down** node of the  $j$ th node in the right-directed list headed by  $V$  contains  $n - j$  nodes. There are therefore  $\frac{n(n-1)}{2}$  nodes constructed by CONSTRUCT\_VECTOR\_TABLE. The worst-case running time of CONSTRUCT\_VECTOR\_TABLE is therefore  $O(n^2)$  (note that this is independent of  $k$ , the dimensionality of  $D$ ). The total space used by SIATEC up to the completion of CONSTRUCT\_VECTOR\_TABLE remains  $O(kn^2)$ .

We shall now present an example of how simply sorting the data structure in Figure 22 yields  $P'(D)$ . Let us consider the dataset in Figure 23. Figure 24 shows the table of vectors represented by the data structure  $V$  computed by CONSTRUCT\_VECTOR\_TABLE for this dataset. This table corresponds to the data

<sup>2</sup>If we denote by  $v_{i,j}$  the location in the  $i$ th column and  $j$ th row of a matrix or table (counting left-to-right and top-to-bottom respectively), then the leading diagonal is the set of locations  $\{v_{i,j} \mid i = j\}$ .

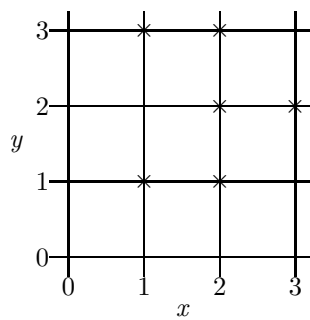
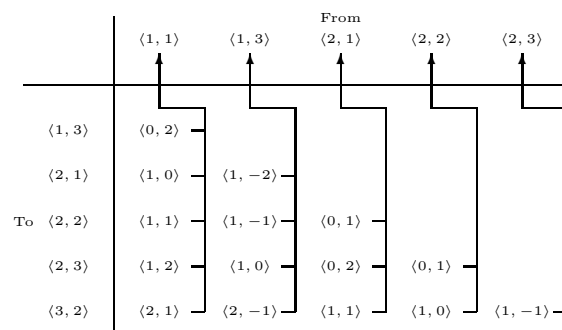


Figure 23: A simple two-dimensional dataset.

Figure 24: The vector table constructed by `CONSTRUCT_VECTOR_TABLE` for the dataset in Figure 23.

Vector	Datapoint
$\langle 0, 1 \rangle$	$\langle 2, 1 \rangle$
$\langle 0, 1 \rangle$	$\langle 2, 2 \rangle$
$\langle 0, 2 \rangle$	$\langle 1, 1 \rangle$
$\langle 0, 2 \rangle$	$\langle 2, 1 \rangle$
$\langle 1, -2 \rangle$	$\langle 1, 3 \rangle$
$\langle 1, -1 \rangle$	$\langle 1, 3 \rangle$
$\langle 1, -1 \rangle$	$\langle 2, 3 \rangle$
$\langle 1, 0 \rangle$	$\langle 1, 1 \rangle$
$\langle 1, 0 \rangle$	$\langle 1, 3 \rangle$
$\langle 1, 0 \rangle$	$\langle 2, 2 \rangle$
$\langle 1, 1 \rangle$	$\langle 1, 1 \rangle$
$\langle 1, 1 \rangle$	$\langle 2, 1 \rangle$
$\langle 1, 2 \rangle$	$\langle 1, 1 \rangle$
$\langle 2, -1 \rangle$	$\langle 1, 3 \rangle$
$\langle 2, 1 \rangle$	$\langle 1, 1 \rangle$

Figure 25: The list that results from sorting the vectors in the table in Figure 24.

structure shown in Figure 22. Each entry in the table in Figure 24 is linked to the datapoint at the top of its column to represent the fact that

$$\Phi_{\delta_j - \delta_i} \uparrow \text{right} = \Phi_{\delta_i}$$

in Figure 20. We now sort the entries in the table in Figure 24 to obtain the list of vectors shown in Figure 25. Note that each vector  $v$  in this list is still linked to the datapoint at the head of the column in Figure 24 that contained  $v$ . Simply reading off all the datapoints attached to the adjacent occurrences of a given vector  $v$  in this list yields the maximal translatable pattern for  $v$ .  $P'(D)$  can be obtained simply by scanning the list once, reading off the attached datapoints and starting a new pattern each time the vector changes. Each box in the right-hand column of Figure 25 corresponds to a maximal translatable pattern.

From the foregoing discussion, it should be clear that the next step after executing `CONSTRUCT_VECTOR_TABLE` is to sort the nodes in the resulting data structure (Figure 22). This is done using `SORT_VECTORS` (Figure 26), an implementation of merge sort that converts the structure in Figure 22 into a single, sorted, down-directed list representing a list of vectors like the one in Figure 25.

`SORT_VECTORS` works in a way that is essentially identical to `SORT_DATASET` (see Figure 12). Each pair of adjacent down-directed lists in Figure 22 is merged into a single list using the procedure `MERGE_VECTOR_COLUMNS` (Figure 27) which is called in line 14 of `SORT_VECTORS`. Each iteration of the main `while` loop (Figure 26, lines 3 to 22) corresponds to one pass along the right-directed list of lists headed by  $V$ . This `while` loop terminates when the complete structure has been converted into a single down-directed list—that is, when  $V \uparrow \text{right} = \text{NULL}$ . This leaves a single sentinel node at the top of the list that is not associated with any vector. This sentinel is removed in lines 23 to 28 of `SORT_VECTORS` producing a sorted, down-directed list like the one in Figure 28. The procedure `DISPOSE_OF_VECTOR_NODE` called in line 27 of `SORT_VECTORS` takes a single `VECTOR_NODE` pointer argument  $v$ . It deallocates the `VECTOR_NODE` pointed to by  $v$  and any other nodes connected to  $v$ . Therefore, if one wishes to remove only the node

```

SORT_VECTORS
1   local variables
2   BEFORE_A, A, B, AFTER_B, C : VECTOR_NODE pointers

3   while V ≠ NULL and V↑right ≠ NULL
4     BEFORE_A ← NULL
5     A ← V
6     V ← NULL
7     repeat
8       if V ≠ NULL
9         BEFORE_A↑right ← NULL
10        B ← A↑right
11        A↑right ← NULL
12        AFTER_B ← B↑right
13        B↑right ← NULL
14        C ← MERGE_VECTOR_COLUMNS(A,B)
15        if V = NULL
16          V ← C
17        else
18          BEFORE_A↑right ← C
19          C↑right ← AFTER_B
20          BEFORE_A ← C
21          A ← BEFORE_A↑right
22        until A = NULL or A↑right = NULL
23  ▷ Finally we delete the sentinel node pointed to by V.
24  if V ≠ NULL
25    A ← V↑down
26    V↑down ← NULL
27    DISPOSE_OF_VECTOR_NODE(V)
28    V ← A

```

Figure 26: SORT\_VECTORS algorithm.

```

MERGE_VECTOR_COLUMNS(A, B : VECTOR_NODE pointers)
1   local variables
2   a, b, C, c : VECTOR_NODE pointers

3   a ← A↑down
4   b ← B↑down
5   C ← A
6   C↑down ← NULL
7   c ← C
8   while a ≠ NULL and b ≠ NULL
9     if VL(b↑right↑vector, a↑right↑vector)
10      c↑down ← b
11      b ← b↑down
12    else
13      c↑down ← a
14      a ← a↑down
15      c ← c↑down
16      c↑down ← NULL
17    if a = NULL
18      c↑down ← b
19    else
20      c↑down ← a
21  return C

```

Figure 27: MERGE\_VECTOR\_COLUMNS algorithm.

pointed to by  $v$ , both the `down` and `right` fields of  $v$  should be `NULL` before `DISPOSE_OF_VECTOR_NODE` is called.

The number of items to be sorted by `SORT_VECTORS` is equal to  $\frac{n(n-1)}{2}$ , the number of entries below the leading diagonal in Table 2. Each comparison involves a call to `VL` in line 9 of `MERGE_VECTOR_COLUMNS` which takes  $O(k)$  time. The worst-case running time of `SORT_VECTORS` is therefore  $O(kn^2 \log_2(n^2)) = O(kn^2 \log_2 n)$ . No extra space is used in the process. In fact, the total amount of space used is reduced by  $n-1$  `VECTOR_NODES` because each sentinel node that heads a down-directed list in Figure 22 is removed. The total worst-case space used by `SIATEC` up to the completion of `SORT_VECTORS` therefore remains  $O(kn^2)$ .

We now present a formal proof that `SORT_VECTORS` yields  $P'(D)$ . The data structure generated by `CONSTRUCT_VECTOR_TABLE` (see Figures 22 and 24) is essentially a representation of the set of ordered pairs  $Z(D)$ , defined as follows.

**Definition 16** *If  $D$  is a dataset then*

$$Z(D) = \{\langle d_2 - d_1, d_1 \rangle \mid d_1, d_2 \in D \wedge d_2 > d_1\} \quad (33)$$

We now prove two lemmas concerning  $Z(D)$ .

**Lemma 7** *If  $D$  is a dataset and  $d_1, d_2, d_3, d_4 \in D$  then*

$$d_2 \neq d_4 \vee d_1 \neq d_3 \Rightarrow \langle d_1 - d_2, d_2 \rangle \neq \langle d_3 - d_4, d_4 \rangle \quad (34)$$

*Proof*

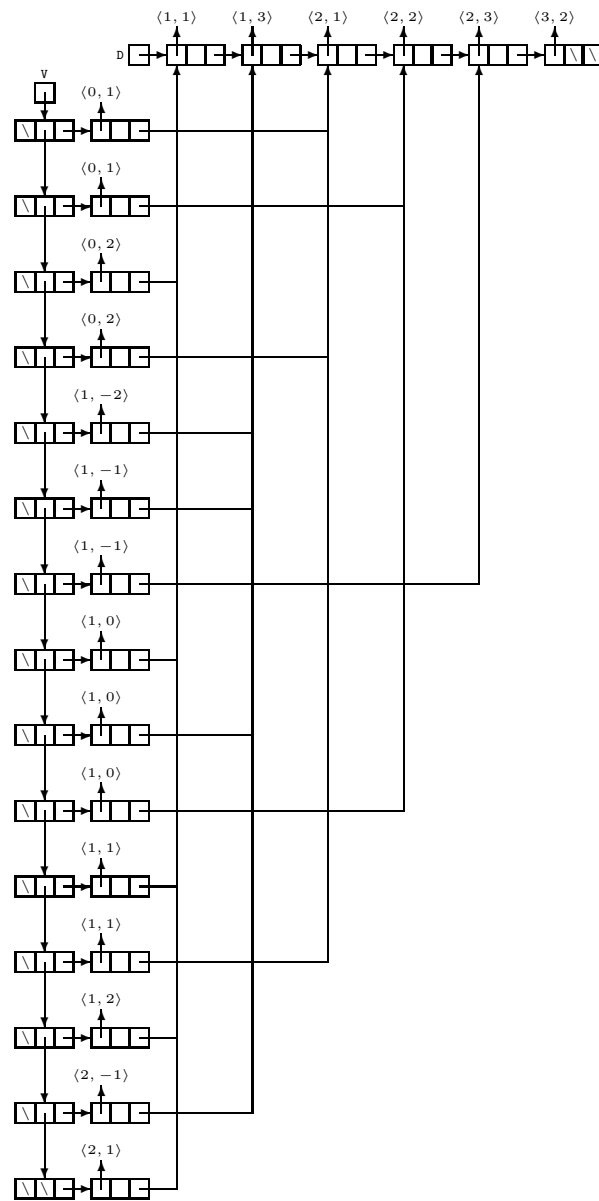


Figure 28: The data structure headed by V at the conclusion of SORT\_VECTORS for the dataset in Figure 23.

$$d_2 \neq d_4 \Rightarrow \langle d_1 - d_2, d_2 \rangle \neq \langle d_3 - d_4, d_4 \rangle \quad (35)$$

$$\begin{aligned} d_1 \neq d_3 \wedge d_2 = d_4 &\Rightarrow d_1 - d_2 \neq d_3 - d_4 \\ &\Rightarrow \langle d_1 - d_2, d_2 \rangle \neq \langle d_3 - d_4, d_4 \rangle \end{aligned} \quad (36)$$

Eq.35 and Eq.36 together imply that

$$d_2 \neq d_4 \vee d_1 \neq d_3 \Rightarrow \langle d_1 - d_2, d_2 \rangle \neq \langle d_3 - d_4, d_4 \rangle$$

■

**Lemma 8** *If  $D$  is a dataset of order  $n$  then*

$$|Z(D)| = \frac{n(n-1)}{2} \quad (37)$$

*Proof*

Definition 16 and Lemma 7 together imply that  $|Z(D)|$  is equal to the number of elements in the set

$$\{\langle d_1, d_2 \rangle \mid d_1, d_2 \in D \wedge d_2 > d_1\} \quad (38)$$

which has the same cardinality as the set

$$\{\{d_1, d_2\} \mid d_1, d_2 \in D\} \quad (39)$$

The cardinality of the set in Eq.39 is equal to the number of distinct combinations possible when 2 distinct objects are drawn from a set of  $n$  distinct objects. This implies that

$$\begin{aligned} |Z(D)| &= \binom{n}{2} \\ &= \frac{n!}{(n-2)!2!} \\ &= \frac{n(n-1)}{2} \end{aligned}$$

■

**Definition 17** *If  $D$  is a dataset and  $z$  is an ordered pair of the form  $\langle e - d, d \rangle$  where  $e, d \in D$  then*

$$F(z, Z(D)) = \{y \mid y \in Z(D) \wedge y[1] = z[1]\} \quad (40)$$

Figure 25 shows the complete set  $Z(D)$  for the dataset in Figure 23. Each row in this list corresponds to an ordered pair  $z = \langle d_2 - d_1, d_1 \rangle$  in  $Z(D)$ . By sorting  $Z(D)$  so that the vectors are in increasing order, the list has effectively been partitioned into classes of adjacent rows such that each class contains all the ordered pairs  $z$  for a particular vector. Each of these classes of ordered pairs corresponds to a set  $F(z, Z(D))$ . For example, if we denote the dataset in Figure 23 by  $D_1$ , then the first class of ordered pairs in the list in Figure 25 corresponds to the set

$$F(\langle\langle 0, 1 \rangle, \langle 2, 1 \rangle\rangle, Z(D_1)) = \{\langle\langle 0, 1 \rangle, \langle 2, 1 \rangle\rangle, \langle\langle 0, 1 \rangle, \langle 2, 2 \rangle\rangle\}$$

**Definition 18** *If  $D$  is a dataset then*

$$\mathcal{Z}(D) = \{F(z, Z(D)) \mid z \in Z(D)\} \quad (41)$$

The set  $\mathcal{Z}(D)$  corresponds to the partition of  $Z(D)$  represented by the sorted list that results after `SORT_VECTORS` has been executed (see Figures 25 and 28). We now prove formally that  $\mathcal{Z}(D)$  is indeed a partition of  $Z(D)$ .

**Lemma 9** *If  $D$  is a dataset then  $\mathcal{Z}(D)$  is a partition of  $Z(D)$ .*

*Proof*

To prove this lemma, we need to prove all three of the following:

$$z \in Z(D) \Rightarrow (\exists F \mid F \in \mathcal{Z}(D) \wedge z \in F) \quad (42)$$

$$F \in \mathcal{Z}(D) \wedge z \in F \Rightarrow z \in Z(D) \quad (43)$$

$$F_1, F_2 \in \mathcal{Z}(D) \wedge F_1 \neq F_2 \wedge z \in F_1 \Rightarrow z \notin F_2 \quad (44)$$

From Definition 17 we know that

$$z \in Z(D) \wedge z[1] = z[1] \Rightarrow z \in F(z, Z(D))$$

Clearly  $z[1] = z[1]$  therefore

$$z \in Z(D) \Rightarrow z \in F(z, Z(D)) \quad (45)$$

From Definition 18 we know that

$$z \in Z(D) \Rightarrow F(z, Z(D)) \in \mathcal{Z}(D) \quad (46)$$

Eq.45 and Eq.46 together prove Eq.42.

From Definition 18 we can deduce that

$$F \in \mathcal{Z}(D) \Rightarrow \exists y \mid F = F(y, Z(D)) \wedge y \in Z(D) \quad (47)$$

Let  $y \in Z(D)$  and let  $F = F(y, Z(D))$ . If

$$F = F(y, Z(D)) \wedge y \in Z(D) \wedge z \in F$$

then it follows that

$$z \in F(y, Z(D)) \wedge y \in Z(D) \quad (48)$$

But from Definition 17 we know that if Eq.48 holds then  $z \in Z(D)$  which proves Eq.43.

From Definition 18 we know that

$$F_1 \in \mathcal{Z}(D) \Rightarrow \exists z_1 \mid F_1 = F(z_1, Z(D)) \wedge z_1 \in Z(D) \quad (49)$$

$$F_2 \in \mathcal{Z}(D) \Rightarrow \exists z_2 \mid F_2 = F(z_2, Z(D)) \wedge z_2 \in Z(D) \quad (50)$$

Let  $z_1, z_2 \in Z(D)$  and let  $F_1 = F(z_1, Z(D))$  and  $F_2 = F(z_2, Z(D))$ . It follows from Definition 17 that

$$\begin{aligned} F_1 \neq F_2 &\Rightarrow F(z_1, Z(D)) \neq F(z_2, Z(D)) \\ &\Rightarrow \{y \mid y \in Z(D) \wedge y[1] = z_1[1]\} \neq \{y' \mid y' \in Z(D) \wedge y'[1] = z_2[1]\} \\ &\Rightarrow (\exists y' \mid y'[1] = z_2[1] \wedge y'[1] \neq z_1[1]) \vee (\exists y \mid y[1] = z_1[1] \wedge y[1] \neq z_2[1]) \\ &\Rightarrow z_2[1] \neq z_1[1] \end{aligned} \quad (51)$$

From Definition 17 it follows directly that

$$z \in F_1 \Rightarrow z \in Z(D) \wedge z[1] = z_1[1] \quad (52)$$

From Eq.51 and Eq.52 we deduce that

$$z \in F_1 \Rightarrow z[1] \neq z_2[1]$$

which, taken with Definition 17, implies

$$z \notin F(z_2, Z(D)) \Rightarrow z \notin F_2$$

thus proving Eq.44. ■

We now prove that the number of nodes in the down-directed list generated by `SORT_VECTORS` is always  $\frac{n(n-1)}{2}$ .

**Lemma 10** *If  $D$  is a dataset then*

$$\sum_{F \in \mathcal{Z}(D)} |F| = \frac{n(n-1)}{2} \quad (53)$$

*Proof*

From Lemma 9 we know that  $\mathcal{Z}(D)$  is a partition of  $Z(D)$ . Therefore

$$\sum_{F \in \mathcal{Z}(D)} |F| = |Z(D)|$$

But from Lemma 8 we know that

$$|Z(D)| = \frac{n(n-1)}{2}$$

therefore

$$\sum_{F \in \mathcal{Z}(D)} |F| = \frac{n(n-1)}{2} \quad \blacksquare$$

**Definition 19** *If  $F \in \mathcal{Z}(D)$  then*

$$\pi(F) = \{y[2] \mid y \in F\}$$

Each boxed set of datapoints in the right-hand column of Figure 25 corresponds to  $\pi(F(z, Z(D)))$  for one of the classes  $F(z, Z(D)) \in \mathcal{Z}(D)$ . (Recall that the sorted list in Figure 25 represents the partition  $\mathcal{Z}(D)$  for the dataset in Figure 23).

**Lemma 11** *If  $D$  is a dataset and  $z \in Z(D)$  then*

$$|\pi(F(z, Z(D)))| = |F(z, Z(D))| \quad (54)$$

*Proof*

From Definition 17 it follows that if  $z_1, z_2 \in F(z, Z(D))$  then  $z_1[1] = z_2[1]$ . This, in turn, implies that if  $z_1, z_2 \in F(z, Z(D))$  and  $z_1 \neq z_2$  then  $z_1[2] \neq z_2[2]$ . This, together with Definition 19 implies that  $|\pi(F(z, Z(D)))| = |F(z, Z(D))|$ . ■

We now prove that each of the boxed sets of datapoints in Figure 25 is the maximal translatable pattern for the vector associated with it.

**Theorem 1** *If  $D$  is a dataset and  $z \in Z(D)$  then*

$$\pi(F(z, Z(D))) = p(z[1], D) \quad (55)$$

*Proof*

Definition 11 implies that

$$p(z[1], D) = \{d \mid d \in D \wedge d + z[1] \in D\} \quad (56)$$

Definition 19 implies that

$$\pi(F(z, Z(D))) = \{y[2] \mid y \in F(z, Z(D))\} \quad (57)$$

Definition 17 and Eq.57 imply that

$$\begin{aligned} \pi(F(z, Z(D))) &= \{y[2] \mid y \in \{x \mid x \in Z(D) \wedge x[1] = z[1]\}\} \\ &= \{y[2] \mid y \in Z(D) \wedge y[1] = z[1]\} \end{aligned} \quad (58)$$

Eq.58 and Definition 16 imply

$$\begin{aligned} \pi(F(z, Z(D))) &= \{y[2] \mid y \in \{(d_2 - d_1, d_1) \mid d_1, d_2 \in D \wedge d_2 > d_1\} \wedge y[1] = z[1]\} \\ &= \{d_1 \mid d_1, d_2 \in D \wedge d_2 > d_1 \wedge z[1] = d_2 - d_1\} \\ &= \{d_1 \mid d_1, d_2 \in D \wedge d_2 > d_1 \wedge d_2 = d_1 + z[1]\} \end{aligned} \quad (59)$$

It is clear that

$$d_2 > d_1 \wedge d_2 = d_1 + z[1] \iff z[1] > \mathbf{0} \wedge d_2 = d_1 + z[1] \quad (60)$$

Eq.59 and Eq.60 together imply

$$\pi(F(z, Z(D))) = \{d_1 \mid d_1, d_2 \in D \wedge z[1] > \mathbf{0} \wedge d_2 = d_1 + z[1]\} \quad (61)$$

Definition 16 implies that

$$z \in Z(D) \Rightarrow z[1] > \mathbf{0} \quad (62)$$

Eq.61 and Eq.62 imply

$$\begin{aligned} \pi(F(z, Z(D))) &= \{d_1 \mid d_1, d_2 \in D \wedge d_2 = d_1 + z[1]\} \\ &= \{d_1 \mid d_1 \in D \wedge d_1 + z[1] \in D\} \end{aligned} \quad (63)$$

Eq.56 and Eq.63 imply

$$\pi(F(z, Z(D))) = p(z[1], D)$$

■

**Definition 20** *If  $D$  is a dataset then*

$$\Pi(D) = \{\pi(F(z, Z(D))) \mid z \in Z(D)\} \quad (64)$$

If  $D_1$  is the dataset in Figure 23 then  $\Pi(D_1)$  is the set that only contains every boxed set of datapoints in the right-hand column of Figure 25.

We now prove that  $\Pi(D) = P'(D)$  and therefore that the sorted list of nodes generated by SORT\_VECTORS does indeed represent  $P'(D)$ .

**Theorem 2** *If  $D$  is a dataset then*

$$P'(D) = \Pi(D) \quad (65)$$

*Proof*

Theorem 1 and Definition 20 imply

$$\Pi(D) = \{p(z[1], D) \mid z \in Z(D)\} \quad (66)$$

Eq.66 and Definition 16 imply

$$\begin{aligned}\Pi(D) &= \{p(z[1], D) \mid z \in \{(d_2 - d_1, d_1) \mid d_1, d_2 \in D \wedge d_2 > d_1\}\} \\ &= \{p(d_2 - d_1, D) \mid d_1, d_2 \in D \wedge d_2 > d_1\}\end{aligned}\tag{67}$$

Eq.67 and Definition 14 imply

$$P'(D) = \Pi(D)$$

■

We now prove that the sum of the sizes of all the maximal translatable patterns in  $P'(D)$  is less than or equal to  $\frac{n(n-1)}{2}$ .

**Theorem 3** *If  $D$  is a dataset of cardinality  $n$  then*

$$\sum_{p \in P'(D)} |p| \leq \frac{n(n-1)}{2}\tag{68}$$

*Proof*

Theorem 2 implies that

$$\sum_{p \in P'(D)} |p| = \sum_{\pi \in \Pi(D)} |\pi|\tag{69}$$

Defs.18, 19 and 20 imply

$$|\Pi(D)| \leq |Z(D)|\tag{70}$$

Lemma 11, Eq.70, Definition 18 and Definition 20 taken together imply

$$\sum_{\pi \in \Pi(D)} |\pi| \leq \sum_{F \in Z(D)} |F|\tag{71}$$

Eq.71 and Lemma 10 imply that

$$\sum_{\pi \in \Pi(D)} |\pi| \leq \frac{n(n-1)}{2}\tag{72}$$

Eq.72 and Eq.69 imply

$$\sum_{p \in P'(D)} |p| \leq \frac{n(n-1)}{2}$$

■

Theorem 3 follows from the fact that  $\Pi(D)$  is derived from  $Z(D)$  simply by partitioning  $Z(D)$ . Theorem 3 is critical for computing the worst-case running time of the procedures called in lines 7 to 13 of SIATEC.

As explained above, the data structure generated by SORT\_VECTORS can be used directly to output  $P'(D)$ . However, the purpose of SIATEC is to compute  $T'(D)$  and this process can be simplified by converting the data structure produced by SORT\_VECTORS (Figures 25 and 28) into a more explicit representation of  $P'(D)$ . This conversion is carried out using the procedure CONSTRUCT\_PATTERN\_LIST defined in Figure 29. CONSTRUCT\_PATTERN\_LIST is called in line 7 of SIATEC. Figure 30 shows the data structure that results when CONSTRUCT\_PATTERN\_LIST is carried out on the data structure shown in Figure 28.

As can be seen in Figure 30, CONSTRUCT\_PATTERN\_LIST builds a down-directed list of PATTERN\_NODES. The `pattern` field of each of these nodes is made to point to a right-directed list of VECTOR\_NODES that indirectly represents one of the maximal translatable patterns in  $P'(D)$ . The function MAKE\_NEW\_PATTERN\_NODE called in line 7 of CONSTRUCT\_PATTERN\_LIST allocates a new PATTERN\_NODE, setting its `size` field to 0 and all its pointer fields to NULL.

```

CONSTRUCT_PATTERN_LIST
1   local variables
2   p : a PATTERN_NODE pointer
3   v1, v2, q : VECTOR_NODE pointers

4   P ← NULL
5   if V ≠ NULL
6     v1 ← V
7     P ← MAKE_NEW_PATTERN_NODE
8     p ← P
9     while v1 ≠ NULL
10      p↑pattern ← MAKE_NEW_VECTOR_NODE
11      q ← p↑pattern
12      q↑down ← v1↑right↑right
13      v2 ← v1↑down
14      while v2 ≠ NULL and VE(v2↑right↑vector, v1↑right↑vector)
15        q↑right ← MAKE_NEW_VECTOR_NODE
16        q ← q↑right
17        q↑down ← v2↑right↑right
18        v2 ← v2↑down
19      v1 ← v2
20      if v1 ≠ NULL
21        p↑down ← MAKE_NEW_PATTERN_NODE
22        p ← p↑down
    
```

Figure 29: CONSTRUCT\_PATTERN\_LIST algorithm.

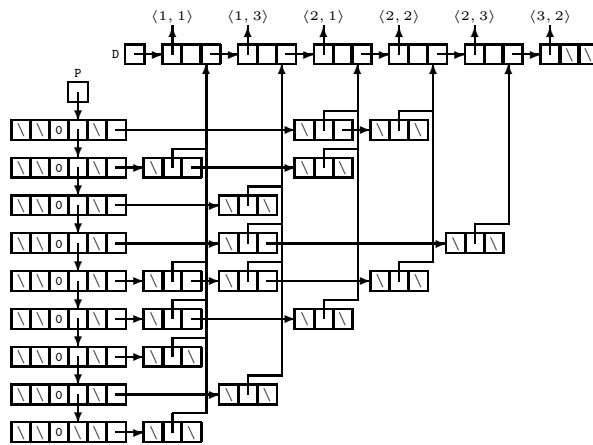


Figure 30: The data structure that results after CONSTRUCT\_PATTERN\_LIST has executed for the dataset in Figure 23.

```

VECTORIZE_PATTERNS
1   local variables
2   p : a PATTERN_NODE pointer
3   v, q : VECTOR_NODE pointers

4   p ← P
5   while p ≠ NULL
6     p↑vec_seq ← MAKE_NEW_VECTOR_NODE
7     v ← p↑vec_seq
8     q ← p↑pattern
9     while q↑right ≠ NULL
10      v↑right ← MAKE_NEW_VECTOR_NODE
11      v ← v↑right
12      v↑vector ← VM(q↑right↑down↑vector, q↑down↑vector)
13      q ← q↑right
14      p ← p↑down

```

Figure 31: VECTORIZE\_PATTERNS algorithm.

CONSTRUCT\_PATTERN\_LIST simply scans once the down-directed list headed by  $V$  (see Figure 28). For each node, if the vector is different from the last one a new `PATTERN_NODE` is created and if the vector is the same as the last one a new `VECTOR_NODE` is added to the `pattern` list for the current `PATTERN_NODE`. The worst-case running time of `CONSTRUCT_PATTERN_LIST` is proportional to the length of the list headed by  $V$  which was shown above to be  $\frac{n(n-1)}{2}$  (Lemma 8). The running time is therefore  $O(kn^2)$  because for each node in the list, the function `VE` (whose running time is  $O(k)$ ) must be executed (Figure 29, line 14). After execution of `CONSTRUCT_PATTERN_LIST` the overall space complexity of `SIATEC` remains  $O(kn^2)$ .

## 6.5 Computing $P''(D)$

An examination of Figure 30 reveals that the pattern  $\{\langle 1, 1 \rangle, \langle 2, 1 \rangle\}$  occurs twice—it is the maximal translatable pattern for both the vector  $\langle 0, 2 \rangle$  and the vector  $\langle 1, 1 \rangle$  (see Figure 25). It can also be seen that another pattern in the list is  $\{\langle 1, 3 \rangle, \langle 2, 3 \rangle\}$  which is translationally equivalent to  $\{\langle 1, 1 \rangle, \langle 2, 1 \rangle\}$ . Clearly, all these patterns are members of the same TEC so we only need to compute the TEC of one of these patterns. Therefore, as described on page 13, before computing  $T'(D)$  we compute the partition  $\mathcal{P}(D)$  (defined in Eqs.26 and 27) and then construct a set  $P''(D)$  that contains exactly one pattern from each  $\mathcal{E} \in \mathcal{P}(D)$ . This guarantees that there are no patterns in  $P''(D)$  that are translationally equivalent, thus ensuring that the procedure `COMPUTE_TECS` (Figure 2, line 12) does no more work than necessary. The procedures called in lines 8 to 11 of `SIATEC` construct such a set  $P''(D)$  from the list of patterns generated by `CONSTRUCT_PATTERN_LIST`.

The first step in this process is carried out by the procedure `VECTORIZE_PATTERNS` which is called in line 8 of `SIATEC`. This procedure is defined in Figure 31. For each pattern  $p$  in the list headed by the global variable `P` that results after the execution of `CONSTRUCT_PATTERN_LIST` (see Figure 30), `VECTORIZE_PATTERNS` computes an intervallic representation of  $p$  which is stored in the `vec_seq` field of the `PATTERN_NODE` representing  $p$ . Figure 32 shows the data structure that results when `VECTORIZE_PATTERNS` has been executed on the data structure shown in Figure 30.

Each pattern is represented in the data structure as a right-directed list headed by the `pattern` field of a `PATTERN_NODE`. This right-directed list actually represents an *ordered* set of datapoints in which the datapoints are in increasing order. This ordering is a result of the way that `SORT_VECTORS` operates and the

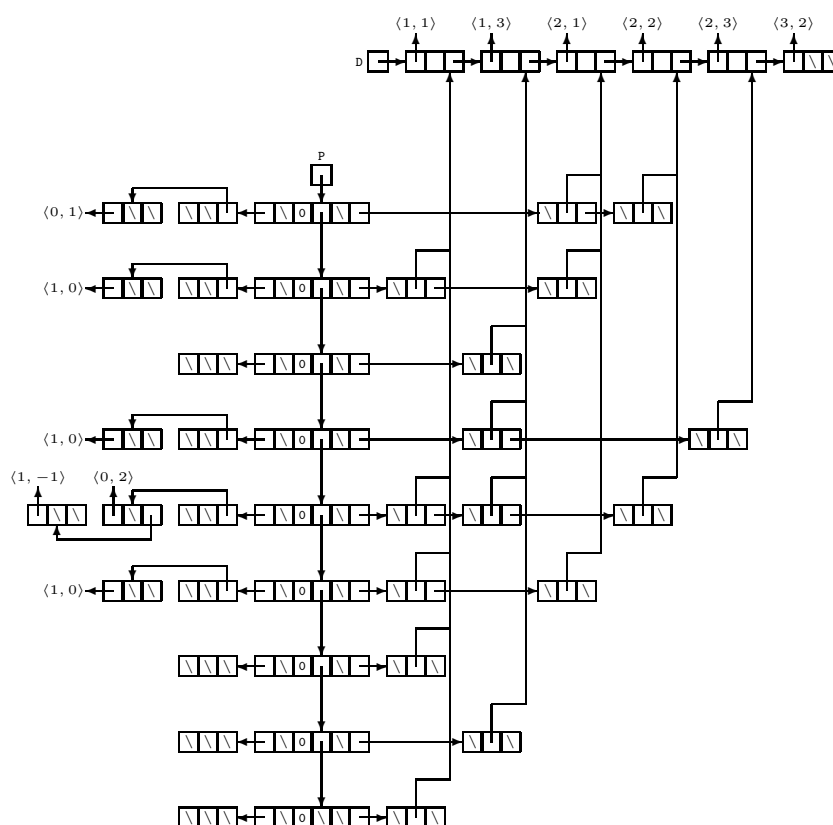


Figure 32: The data structure that results after `VECTORIZE_PATTERNS` has executed for the dataset in Figure 23.

fact that the dataset list is sorted. The fact that each pattern representation is ordered in this way means that two patterns will have an identical intervallic representation as computed by `VECTORIZE_PATTERNS` if and only if they are translationally equivalent (see, for example, the second, fourth and sixth patterns in the list shown in Figure 32).

We now formalize this concept of “intervallic representation”.

**Definition 21** *An object  $\beta$  is a **vector sequence** if and only if it is an ordered set of vectors.  $\beta$  is a  **$k$ -dimensional vector sequence** if and only if every vector in  $\beta$  has cardinality  $k$ .*

Let

$$p_i = \{d_{i,1}, d_{i,2}, \dots, d_{i,|p_i|}\}$$

be one of the patterns in the list of patterns generated by `CONSTRUCT_PATTERN_LIST` and let

$$\gamma_i = \{\delta_{i,1}, \delta_{i,2}, \dots, \delta_{i,|p_i|}\}$$

be the ordered set of datapoints that results from sorting  $p_i$  so that the datapoints are in increasing order. The procedure `VECTORIZE_PATTERNS` takes each pattern  $p_i$  and computes the vector sequence

$$\beta_i = \bigoplus_{j=1}^{|p_i|-1} \langle \delta_{i,j+1} - \delta_{i,j} \rangle \quad (73)$$

The worst-case running time of `VECTORIZE_PATTERNS` is linear in the sum of the cardinalities of all the patterns  $p_i$  which was shown in Lemma 10 to be equal to

$$\frac{n(n-1)}{2}$$

For each datapoint but one in each pattern, the vector subtraction in line 12 of `VECTORIZE_PATTERNS` must be computed. Each of these vector subtractions takes  $O(k)$  time therefore the overall worst-case running time of `VECTORIZE_PATTERNS` is  $O(kn^2)$ . The overall space used remains  $O(kn^2)$ .

If the `PATTERN_NODES` headed by `P` are sorted by their `vec_seq` fields then all the patterns with a given intervallic representation will be adjacent to each-other in the list and we will effectively have computed the partition  $\mathcal{P}(D)$ .

Clearly, if two patterns have different cardinalities then they are not translationally equivalent. We can therefore make this sorting process more efficient by first computing the cardinality of each pattern and storing this integer in the `size` field of the `PATTERN_NODE` for the pattern. This is more efficient than computing the pattern cardinalities “on the fly” during the sorting process itself. Using the latter strategy, each time a pair of patterns is compared one must compute the cardinalities of both patterns. If there are  $m$  patterns, then using merge sort this would involve  $O(m \log_2 m)$  pattern-cardinality computations whereas if we simply compute the cardinalities of the patterns in advance, only  $O(m)$  pattern-cardinality computations are necessary.

This

one-time computation of pattern cardinalities is carried out by the procedure `COMPUTE_PATTERN_SIZES` which is defined in Figure 33. The worst-case running time for `COMPUTE_PATTERN_SIZES` is proportional to the sum of the cardinalities of the patterns in the list headed by `P`. From Lemma 10 we can deduce that this running time is therefore  $O(n^2)$ . This running time is independent of the dimensionality of the dataset since we are simply counting the number of datapoints in each pattern.

Figure 34 shows the data structure that results after `COMPUTE_PATTERN_SIZES` has been executed for the dataset in Figure 23. The only difference between Figure 34 and Figure 32 is that each pattern’s cardinality has been computed and stored in the `size` field of the pattern’s `PATTERN_NODE`.

```

COMPUTE_PATTERN_SIZES
1   local variables
2   p : a PATTERN_NODE pointer
3   q : a VECTOR_NODE pointer

4   p ← P
5   while p ≠ NULL
6     q ← p↑pattern
7     p↑size ← 0
8     while q ≠ NULL
9       p↑size ← p↑size + 1
10      q ← q↑right
11     p ← p↑down
    
```

Figure 33: COMPUTE\_PATTERN\_SIZES algorithm.

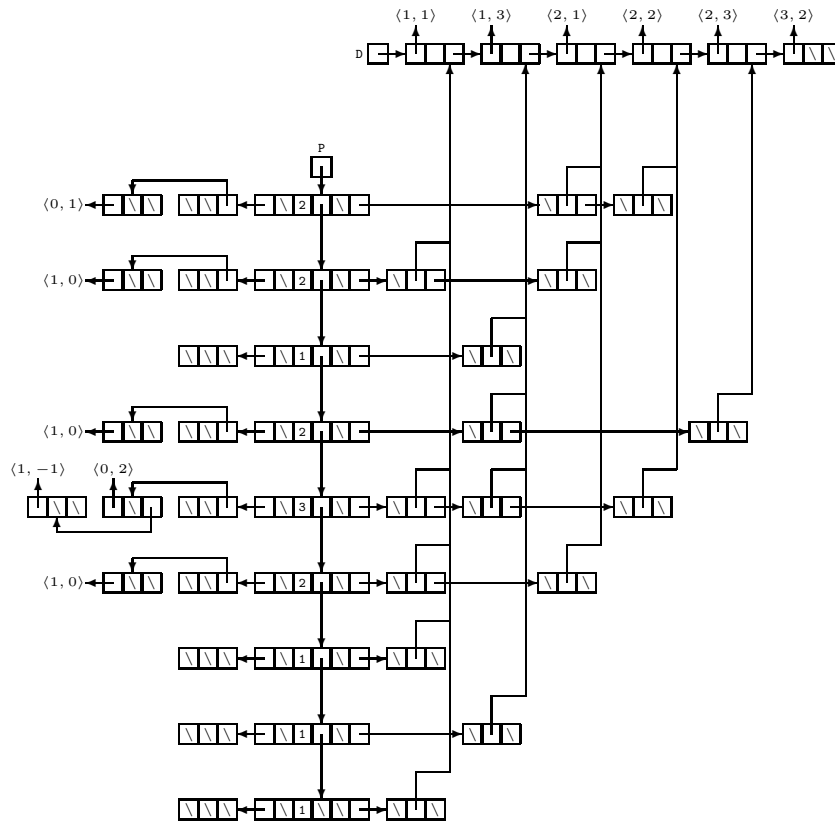


Figure 34: The data structure that results after COMPUTE\_PATTERN\_SIZES has executed for the dataset in Figure 23.

```

SORT_PATTERN_VECTOR_SEQUENCES
1   local variables
2   ABOVE_A, A, B, BELOW_B, C : PATTERN_NODE pointers

3   while P ≠ NULL and P↑down ≠ NULL
4     ABOVE_A ← NULL
5     A ← P
6     P ← NULL
7     repeat
8       if P ≠ NULL
9         ABOVE_A↑down ← NULL
10      B ← A↑down
11      A↑down ← NULL
12      BELOW_B ← B↑down
13      B↑down ← NULL
14      C ← MERGE_PATTERN_ROWS(A,B)
15      if P = NULL
16        P ← C
17      else
18        ABOVE_A↑down ← C
19      C↑down ← BELOW_B
20      ABOVE_A ← C
21      A ← ABOVE_A↑down
22  until A = NULL or A↑down = NULL

```

Figure 35: SORT\_PATTERN\_VECTOR\_SEQUENCES algorithm.

The partition  $\mathcal{P}(D)$  can now be computed efficiently by simply sorting the down-directed list of PATTERN\_NODES headed by P. This is done using the procedure SORT\_PATTERN\_VECTOR\_SEQUENCES (Figure 35), an implementation of merge sort that converts the unsorted down-directed list headed by P into a sorted right-directed list. SORT\_PATTERN\_VECTOR\_SEQUENCES works in essentially the same way as SORT\_DATASET (Figure 12). The function MERGE\_PATTERN\_ROWS called in line 14 of SORT\_PATTERN\_VECTOR\_SEQUENCES merges two sorted right-directed lists into a single list using the comparison function PVSL. MERGE\_PATTERN\_ROWS is defined in Figure 36 and PVSL is defined in Figure 37.

We now formalize the concept of vector sequence inequality that is used in SORT\_PATTERN\_VECTOR\_SEQUENCES.

**Definition 22** *If  $\beta_1$  and  $\beta_2$  are two vector sequences then*

$$\beta_1 < \beta_2$$

*if and only if one of the following conditions is satisfied:*

1.  $|\beta_1| < |\beta_2|$ ;
2.  $|\beta_1| = |\beta_2|$  and  $\beta_1[1] < \beta_2[1]$ ;
3.  $|\beta_1| = |\beta_2|$  and there exists an integer  $j$  such that  $j > 1$  and  $\beta_1[j] < \beta_2[j]$  and  $\beta_1[i] = \beta_2[i]$  for all  $i$  such that  $1 \leq i < j$ .

```

MERGE_PATTERN_ROWS(A, B : PATTERN_NODE pointers)
1   local variables
2   a, b, C, c : PATTERN_NODE pointers

3   a ← A
4   b ← B
5   if PVSL(b,a)
6     C ← a
7     a ← a↑right
8   else
9     C ← b
10    b ← b↑right
11   C↑right ← NULL
12   c ← C
13   while a ≠ NULL and b ≠ NULL
14     if PVSL(b,a)
15       c↑right ← a
16       a ← a↑right
17     else
18       c↑right ← b
19       b ← b↑right
20     c ← c↑right
21     c↑right ← NULL
22   if a = NULL
23     c↑right ← b
24   else
25     c↑right ← a
26   return C

```

Figure 36: MERGE\_PATTERN\_ROWS algorithm.

```

PVSL(p1, p2 : PATTERN_NODE pointers)
1   if p1↑size < p2↑size
2     return TRUE
3   if p1↑size > p2↑size
4     return FALSE
5   return VEC_LIST_LESS_THAN(p1↑vec_seq↑right, p2↑vec_seq↑right)

```

Figure 37: PVSL algorithm.

```

      VEC_LIST_LESS_THAN(p1, p2 : VECTOR_NODE pointers)
1     if p1 = NULL and p2 = NULL
2         return FALSE
3     if VL(p1↑vector, p2↑vector)
4         return TRUE
5     if VL(p2↑vector, p1↑vector)
6         return FALSE
7     return VEC_LIST_LESS_THAN(p1↑right, p2↑right)

```

Figure 38: VEC\_LIST\_LESS\_THAN algorithm.

The function PVSL implements the binary relation ‘<’ as this applies to vector sequences according to Definition 22.

If the ordering of the two patterns to be compared cannot be determined from their cardinalities (lines 1–4 of PVSL) then the recursive function VEC\_LIST\_LESS\_THAN is called. This function is defined in Figure 38. VEC\_LIST\_LESS\_THAN effectively implements the second and third conditions in Definition 22.

The effect of SORT\_PATTERN\_VECTOR\_SEQUENCES is to produce a right-directed list of PATTERN\_NODES in which the patterns are sorted so that their associated vector sequences are in non-*increasing* order. The larger patterns therefore end up nearer the head of the list.

We now demonstrate that the worst-case running time of SORT\_PATTERN\_VECTOR\_SEQUENCES is  $O(kn^2 \log_2 n)$  where  $k$  and  $n$  are as defined on pages 9 and 9 respectively.

SORT\_PATTERN\_VECTOR\_SEQUENCES effectively uses merge sort to sort an ordered set of vector sequences. Let us denote by  $\mathcal{B}$  the ordered set of vector sequences to be sorted and let us assume for convenience (and with no loss of generality) that

$$|\mathcal{B}| = m = 2^x \quad (74)$$

where  $x$  is an integer greater than zero. We define

$$\begin{aligned} \mathcal{B}_0 &= \langle \langle \beta_{0,1} \rangle, \langle \beta_{0,2} \rangle, \dots, \langle \beta_{0,m} \rangle \rangle \\ &= \bigoplus_{i=1}^m \langle \langle \mathcal{B}[i] \rangle \rangle \end{aligned} \quad (75)$$

After one iteration of the outer **while** loop (lines 3–22) of SORT\_PATTERN\_VECTOR\_SEQUENCES we have effectively converted  $\mathcal{B}_0$  into the ordered set

$$\mathcal{B}_1 = \langle \langle \beta_{1,1}, \beta_{1,2} \rangle, \langle \beta_{1,3}, \beta_{1,4} \rangle, \dots, \langle \beta_{1,m-1}, \beta_{1,m} \rangle \rangle \quad (76)$$

where each ordered pair  $\langle \beta_{1,i}, \beta_{1,i+1} \rangle$  in  $\mathcal{B}_1$  is the result of merging the two ordered sets  $\langle \beta_{0,i} \rangle$  and  $\langle \beta_{0,i+1} \rangle$  in  $\mathcal{B}_0$  so that

$$\beta_{1,i} \geq \beta_{1,i+1}$$

By the end of the second iteration of the outer **while** loop of SORT\_PATTERN\_VECTOR\_SEQUENCES,  $\mathcal{B}_1$  has been converted into the ordered set

$$\mathcal{B}_2 = \langle \langle \beta_{2,1}, \beta_{2,2}, \beta_{2,3}, \beta_{2,4} \rangle, \langle \beta_{2,5}, \beta_{2,6}, \beta_{2,7}, \beta_{2,8} \rangle, \dots, \langle \beta_{2,m-3}, \beta_{2,m-2}, \beta_{2,m-1}, \beta_{2,m} \rangle \rangle \quad (77)$$

Where each ordered set of vector sequences

$$\langle \beta_{2,i}, \beta_{2,i+1}, \beta_{2,i+2}, \beta_{2,i+3} \rangle$$

in  $\mathcal{B}_2$  is the result of merging the two ordered sets  $\langle \beta_{1,i}, \beta_{1,i+1} \rangle$  and  $\langle \beta_{1,i+2}, \beta_{1,i+3} \rangle$  in  $\mathcal{B}_1$  so that

$$\beta_{2,i} \geq \beta_{2,i+1} \geq \beta_{2,i+2} \geq \beta_{2,i+3}$$

This process continues until, after the  $x$ th iteration, we have computed the ordered set

$$\mathcal{B}_x = \mathcal{B}_{\log_2 m} = \langle \langle \beta_{x,1}, \beta_{x,2}, \dots, \beta_{x,m} \rangle \rangle \quad (78)$$

where the single element of  $\mathcal{B}_x$  is the ordered set that results from sorting  $\mathcal{B}$  so that all the vector sequences are in non-increasing order according to Definition 22.

To derive  $\mathcal{B}_1$  from  $\mathcal{B}_0$  we have to first merge  $\langle \beta_{0,1} \rangle$  with  $\langle \beta_{0,2} \rangle$  to get  $\langle \beta_{1,1}, \beta_{1,2} \rangle$ . Then we merge  $\langle \beta_{0,3} \rangle$  with  $\langle \beta_{0,4} \rangle$  to get  $\langle \beta_{1,3}, \beta_{1,4} \rangle$  and so on until we have merged  $\langle \beta_{0,m-1} \rangle$  with  $\langle \beta_{0,m} \rangle$  to get  $\langle \beta_{1,m-1}, \beta_{1,m} \rangle$ .

Each of these merges involves comparing two vector sequences and determining which of the two is the lesser according to Definition 22. The total time taken to derive  $\mathcal{B}_1$  from  $\mathcal{B}_0$  is therefore

$$\sum_{i=1}^{m/2} t(\langle \beta_{0,2i-1} \rangle, \langle \beta_{0,2i} \rangle)$$

where  $t(\mathcal{C}_1, \mathcal{C}_2)$  is the time taken to merge the two (sorted) ordered sets of vector sequences  $\mathcal{C}_1$  and  $\mathcal{C}_2$ .

To derive  $\mathcal{B}_2$  from  $\mathcal{B}_1$  we first have to merge the two ordered sets  $\langle \beta_{1,1}, \beta_{1,2} \rangle$  and  $\langle \beta_{1,3}, \beta_{1,4} \rangle$ . Then we merge the sets  $\langle \beta_{1,5}, \beta_{1,6} \rangle$  and  $\langle \beta_{1,7}, \beta_{1,8} \rangle$  and so on until we have merged the sets  $\langle \beta_{1,m-3}, \beta_{1,m-2} \rangle$  and  $\langle \beta_{1,m-1}, \beta_{1,m} \rangle$ . The time taken to derive  $\mathcal{B}_2$  from  $\mathcal{B}_1$  is therefore

$$\sum_{i=1}^{m/4} t(\langle \beta_{1,4i-3}, \beta_{1,4i-2} \rangle, \langle \beta_{1,4i-1}, \beta_{1,4i} \rangle)$$

Similarly, the time taken to derive  $\mathcal{B}_3$  from  $\mathcal{B}_2$  is

$$\sum_{i=1}^{m/8} t(\langle \beta_{2,8i-7}, \beta_{2,8i-6}, \dots, \beta_{2,8i-4} \rangle, \langle \beta_{2,8i-3}, \beta_{2,8i-2}, \dots, \beta_{2,8i} \rangle)$$

and, in general, the time taken to derive  $\mathcal{B}_j$  from  $\mathcal{B}_{j-1}$  is

$$\sum_{i=1}^{m/2^j} t(\langle \beta_{j-1,2^j i-2^j+1}, \dots, \beta_{j-1,2^j i-2^j-1} \rangle, \langle \beta_{j-1,2^j i-2^j-1+1}, \dots, \beta_{j-1,2^j i} \rangle) \quad (79)$$

This implies that the total time taken to derive  $\mathcal{B}_x$  from  $\mathcal{B}_0$ , which we denote by  $\mathcal{T}_1$ , is given by

$$\mathcal{T}_1 = \sum_{j=1}^x \left( \sum_{i=1}^{m/2^j} t(\langle \beta_{j-1,2^j i-2^j+1}, \dots, \beta_{j-1,2^j i-2^j-1} \rangle, \langle \beta_{j-1,2^j i-2^j-1+1}, \dots, \beta_{j-1,2^j i} \rangle) \right) \quad (80)$$

Now let's assume that we have two sorted ordered sets of vector sequences,

$$\begin{aligned} A &= \langle \beta_1, \beta_2, \dots, \beta_y \rangle \\ B &= \langle \beta_{y+1}, \beta_{y+2}, \dots, \beta_{2y} \rangle \end{aligned}$$

that we wish to merge into a single sorted ordered set,  $C$ . We begin by comparing  $\beta_1$  with  $\beta_{y+1}$  and appending the greater of the two to  $C$ . Then if, say,  $\beta_1 > \beta_{y+1}$  we compare  $\beta_2$  with  $\beta_{y+1}$  and append the greater of the two to  $C$ . This continues until either all the elements of  $A$  or all the elements of  $B$  have

been appended to  $C$  at which point all the remaining unmerged elements are appended to  $C$  in a single step to give the desired result. As can be seen, this process involves carrying out a series of comparisons in which an element of  $A$  is compared with an element of  $B$  to determine which of the two is the greater. Each of these comparisons results in a new element being appended to  $C$ . In the worst case, the number of comparisons to be carried out is  $|A| + |B| - 1$ . It is clear from Definition 22 that the worst-case running time for comparing two  $k$ -dimensional vector sequences,  $\beta_1$  and  $\beta_2$ , is  $ck|\beta_{\min}|$  where  $c$  is a constant and  $\beta_{\min}$  is the lesser of  $\beta_1$  and  $\beta_2$ . For each element  $\beta_i$  in  $A$  or  $B$ , we know that  $\beta_i$  is the greater vector sequence in at most one comparison because if  $\beta_i$  is the greater sequence in a comparison it is immediately appended to  $C$  and is not compared with any other member of  $A$  or  $B$  after that. We therefore know that the time taken to carry out the comparison that results in  $\beta_i$  being appended to  $C$  is less than or equal to  $ck|\beta_i|$ . All this implies that

$$t(A, B) < \sum_{i=1}^{2y} ck|\beta_i| \quad (81)$$

Eqs.80 and 81 together imply that

$$\mathcal{T}_1 < \sum_{j=1}^x \left( \sum_{i=1}^{m/2^j} \sum_{r=2^{j-1}i+1}^{2^j i} ck|\beta_{j-1,r}| \right) \quad (82)$$

We now wish to show that in Eq.82 the greatest value of  $r$  for a given value of  $i$  is one less than the least value of  $r$  for  $i + 1$ . That is

$$2^j(i+1) - 2^j + 1 = 2^j i + 1$$

This can be shown straightforwardly as follows:

$$\begin{aligned} 2^j(i+1) - 2^j + 1 &= 2^j(i+1-1) + 1 \\ &= 2^j i + 1 \end{aligned} \quad (83)$$

We also know that the least value of  $r$  for the least value of  $i$  for a given value of  $j$  in Eq.82 is

$$2^j - 2^j + 1 = 1 \quad (84)$$

and the greatest value of  $r$  for the greatest value of  $i$  for a given value of  $j$  is

$$2^j \times m/2^j = m \quad (85)$$

Eqs.82, 83, 84 and 85 taken together imply

$$\mathcal{T}_1 < \sum_{j=1}^x \sum_{r=1}^m ck|\beta_{j-1,r}| \quad (86)$$

But from Lemma 10 we know that the sum of the cardinalities of all the patterns in the list of patterns generated by `CONSTRUCT_PATTERN_LIST` is  $n(n-1)/2$  and we know that the cardinality of the vector sequence computed by `VECTORIZE_PATTERNS` for a given pattern is one less than the cardinality of the pattern. Therefore

$$\sum_{r=1}^m ck|\beta_{j-1,r}| < \frac{ckn(n-1)}{2} \quad (87)$$

and therefore (from Eq.86)

$$\mathcal{T}_1 < \sum_{j=1}^x \frac{ckn(n-1)}{2} \quad (88)$$

We know from Eq.74 that  $x = \log_2 m$  therefore

$$\begin{aligned} \mathcal{T}_1 &< \sum_{j=1}^{\log_2 m} \frac{ckn(n-1)}{2} \\ \Rightarrow \mathcal{T}_1 &< \frac{ckn(n-1)}{2} \log_2 m \end{aligned} \quad (89)$$

From Lemma 10 we know that the number of vector sequences (which is equal to the number of patterns generated by `CONSTRUCT_PATTERN_LIST`) is certainly less than or equal to  $n(n-1)/2$  which is less than  $n^2$ . Therefore

$$\begin{aligned} \mathcal{T}_1 &< \frac{ckn(n-1)}{2} \log_2(n^2) \\ \Rightarrow \mathcal{T}_1 &< ckn(n-1) \log_2 n \end{aligned} \quad (90)$$

which implies that  $\mathcal{T}_1$ , the worst-case running time of `SORT_PATTERN_VECTOR_SEQUENCES` for a  $k$ -dimensional dataset of size  $n$ , is  $O(kn^2 \log_2 n)$ . The space used by `SIATEC` up to the completion of `SORT_PATTERN_VECTOR_SEQUENCES` is  $O(kn^2)$ . Figure 39 shows the data structure that results after `SORT_PATTERN_VECTOR_SEQUENCES` has been executed for the dataset in Figure 23. The data structure that results after execution of `SORT_PATTERN_VECTOR_SEQUENCES` represents  $\mathcal{P}(D)$ .

Having generated the partition  $\mathcal{P}(D)$  by sorting the pattern vector sequences we can now compute a set  $\mathcal{P}''(D)$  which contains exactly one pattern from each element of  $\mathcal{P}(D)$ . This computation is carried out by the procedure `SETIFY_PATTERN_VECTOR_SEQUENCES` defined in Figure 40 which employs a strategy that is essentially identical to that used in the procedure `SETIFY_DATASET` (see Figure 16). `SETIFY_PATTERN_VECTOR_SEQUENCES` simply scans the sorted right-directed list of `PATTERN_NODES` generated by `SORT_PATTERN_VECTOR_SEQUENCES`, deleting each node whose vector sequence (`vec_seq` field) is equal to that of the previous node. The function `PVSE` (Figure 40, line 5) takes two `PATTERN_NODE` pointer variables and returns `TRUE` if and only if the vector sequences stored in the `vec_seq` fields of the nodes pointed to by the argument pointers are equal. If the vector sequence stored in the `vec_seq` field of a `PATTERN_NODE`  $p$  is the same as the one stored in the `vec_seq` field of the previous node then  $p$  is deleted using the function `DISPOSE_OF_PATTERN_NODE` called in line 10 of `SETIFY_PATTERN_VECTOR_SEQUENCES`. `DISPOSE_OF_PATTERN_NODE` recursively disposes of all the other nodes attached to the node pointed to by its argument which is why the `right` field of the node to be deleted is set to `NULL` in line 9 of `SETIFY_PATTERN_VECTOR_SEQUENCES`.

In `SETIFY_PATTERN_VECTOR_SEQUENCES`, each pattern vector sequence is compared with the preceding one and deleted if it is the same. This means that each pattern in the list (except the first) is compared with exactly one other pattern. If we denote the number of patterns by  $m$  then the running time of `SETIFY_PATTERN_VECTOR_SEQUENCES` is therefore

$$\mathcal{T}_2 < \sum_{i=1}^m t(\beta_i) \quad (91)$$

where  $t(\beta_i)$  is the time taken to determine if  $\beta_i$  is equal to the previous pattern vector sequence in the list.

If  $\beta_1$  and  $\beta_2$  are two  $k$ -dimensional vector sequences then the time taken to determine if  $\beta_1 = \beta_2$  is  $ck|\beta_{\min}|$  where  $\beta_{\min}$  is the lesser of the two vector sequences being compared and  $c$  is a constant. The worst case occurs when  $\beta_1 = \beta_2$  in which case the time taken is  $ck|\beta_2|$ .

Therefore, in the worst case,  $t(\beta_i) = ck|\beta_i|$  which, substituting into Eq.91, gives

$$\mathcal{T}_2 < \sum_{i=1}^m ck|\beta_i|$$

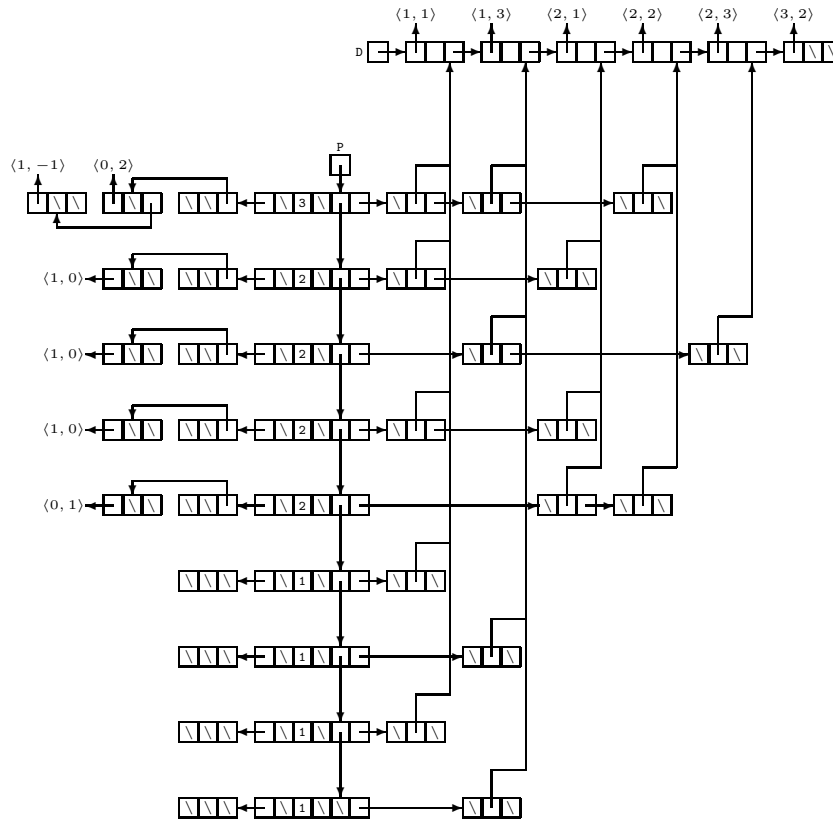


Figure 39: The data structure that results after `SORT_PATTERN_VECTOR_SEQUENCES` has executed for the dataset in Figure 23.

```

SETIFY_PATTERN_VECTOR_SEQUENCES
1   local variables
2    $p_1, p_2$  : PATTERN_NODE pointers

3    $p_1 \leftarrow P$ 
4   while  $p_1 \neq \text{NULL}$  and  $p_1 \uparrow \text{right} \neq \text{NULL}$ 
5     if PVSE( $p_1 \uparrow \text{right}, p_1$ )
6        $\triangleright$  Delete  $p_1 \uparrow \text{right}$ .
7        $p_2 \leftarrow p_1 \uparrow \text{right}$ 
8        $p_1 \uparrow \text{right} \leftarrow p_2 \uparrow \text{right}$ 
9        $p_2 \uparrow \text{right} \leftarrow \text{NULL}$ 
10       $p_2 \leftarrow \text{DISPOSE\_OF\_PATTERN\_NODE}(p_2)$ 
11    else
12       $p_1 \leftarrow p_1 \uparrow \text{right}$ 

```

Figure 40: `SETIFY_PATTERN_VECTOR_SEQUENCES` algorithm.

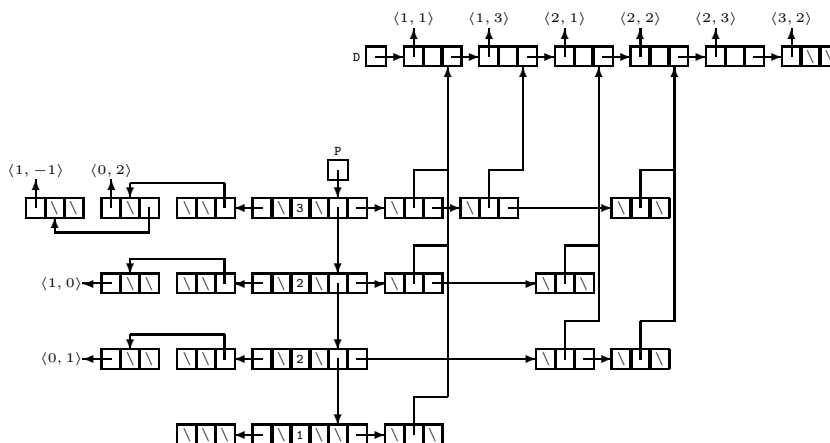


Figure 41: The data structure that results after `SETIFY_PATTERN_VECTOR_SEQUENCES` has executed for the dataset in Figure 23.

But from Lemma 10 we know that

$$\sum_{i=1}^m |\beta_i| < \frac{n(n-1)}{2}$$

therefore

$$\mathcal{J}_2 < ck \frac{n(n-1)}{2}$$

which implies that the worst-case running time of `SETIFY_PATTERN_VECTOR_SEQUENCES` is  $O(kn^2)$ .

The total space used by SIATEC up to the completion of `SETIFY_PATTERN_VECTOR_SEQUENCES` is  $O(kn^2)$ .

Figure 41 shows the data structure that results after `SETIFY_PATTERN_VECTOR_SEQUENCES` has been executed for the dataset in Figure 23.

## 6.6 Computing $T'(D)$

Having computed  $P''(D)$ ,  $T'(D)$  can now be found by computing the TEC of each pattern in  $P''(D)$  in accordance with Eq.28. Recall that the TEC of a pattern  $p$  is represented in SIATEC as an ordered pair  $\langle p, V(p, D) \rangle$  (see page 16 and Eq.30). The computation of  $T'(D)$  therefore consists of computing  $V(p, D)$  for each of the patterns in the right-directed list headed by the global variable `P` after `SETIFY_PATTERN_VECTOR_SEQUENCES` has been executed (see Figure 41).

Recall that we denote by  $\Delta = \langle \delta_1, \delta_2, \dots, \delta_n \rangle$  the ordered set that results from sorting the dataset  $D = \{d_1, d_2, \dots, d_n\}$  so that all the datapoints are in increasing order. Recall also that `COMPUTE_VECTORS` (called in line 4 of SIATEC) computes a linked-list representation of Table 2.

The  $i$ th column in Table 2 contains all and only those vectors that map  $\delta_i$  onto datapoints in  $D$ . We define that for a datapoint  $d$  in a dataset  $D$ ,

$$V(d, D) = \{v \mid v + d \in D\} \quad (92)$$

So we can say that the  $i$ th column of Table 2 contains all and only those vectors in  $V(\delta_i, D)$ .

Clearly, a pattern  $p$  that is a subset of a dataset  $D$  can be translated by a vector  $v$  to give another pattern in  $D$  if and only if every datapoint in  $p$  is mapped by  $v$  onto another datapoint in  $D$ . Therefore, from Eqs.92 and 30,

$$V(p, D) = \bigcap_{d \in p} V(d, D) \quad (93)$$

In other words,  $V(p, D)$  is the set that only contains every vector which is an element of  $V(d, D)$  for *all* the datapoints  $d$  in  $p$ . Thus, if

$$p = \{ \delta_{i_1}, \delta_{i_2}, \delta_{i_3}, \dots, \delta_{i_{|p|}} \}$$

then  $v \in V(p, D)$  if and only if  $v$  appears in the  $i_1$ th column of Table 2 *and* the  $i_2$ th column *and* the  $i_3$ th column *and*... *and* the  $i_{|p|}$ th column.

Let's denote by  $v_{i,j}$  the vector in the  $i$ th column and the  $j$ th row of Table 2 and let's say that we are trying to find  $V(p, D)$  where

$$\phi = \langle \delta_{i_1}, \delta_{i_2}, \dots, \delta_{i_{|p|}} \rangle$$

is the ordered set of datapoints that results when  $p$  is sorted so that the datapoints are in increasing order.

Let us denote by  $A$  a set which will be used to store the vectors in  $V(p, D)$  as they are computed.  $A$  is initialized to  $\emptyset$  but by the end of the computation it will be equal to  $V(p, D)$ . We first determine if  $v_{i_1,1}$  is a member of  $V(\delta_{i_2}, D)$  by scanning down the  $i_2$ th column in Table 2 until we find a vector  $v_{i_2,j}$  that is either greater than or equal to  $v_{i_1,1}$ . We now initialize a pointer—let's call it  $t_{i_2}$ —and set it to point to  $v_{i_2,j}$ . Recall that in Table 2 the vectors increase as one descends a column and decrease as one scans a row from left to right. Let's assume that  $v_{i_2,j} > v_{i_1,1}$  which implies that there is no vector in the  $i_2$ th column that is equal to  $v_{i_1,1}$  so we can proceed to checking  $v_{i_1,2}$ . Now we wish to determine if  $v_{i_1,2}$  is in the  $i_2$ th column.  $v_{i_1,2}$  is greater than  $v_{i_1,1}$  and we know that there is no vector above  $v_{i_2,j}$  that is greater than  $v_{i_1,1}$ . Therefore, our search for an occurrence of  $v_{i_1,2}$  in the  $i_2$ th column can begin with  $v_{i_2,j}$ , the vector currently pointed to by  $t_{i_2}$ . Let's assume that  $v_{i_1,2} = v_{i_2,j}$ . We now have to determine if  $v_{i_1,2}$  occurs in the  $i_3$ th column so we scan down this column until we arrive at a vector—let's denote it by  $v_{i_3,k}$ —that is either equal to or greater than  $v_{i_1,2}$ . We set a pointer  $t_{i_3}$  to point to  $v_{i_3,k}$ . If  $v_{i_3,k} = v_{i_1,2}$  then we go on to check for an occurrence of  $v_{i_1,2}$  in the  $i_4$ th column, tagging the first vector in this column greater than or equal to  $v_{i_1,2}$  with a pointer  $t_{i_4}$ , and so on until one of the following three events occurs:

1. the first vector greater than or equal to  $v_{i_1,2}$  in the  $i_l$ th column is greater than  $v_{i_1,2}$ ;
2. we find an occurrence of  $v_{i_1,2}$  in the  $i_{|p|}$ th column;
3. we reach the bottom of the  $i_l$ th column without finding a vector equal to or greater than  $v_{i_1,2}$ .

In the first case, we know that  $v_{i_1,2} \notin V(\delta_{i_l}, D)$  and therefore that  $v_{i_1,2} \notin V(p, D)$  so we repeat the process with  $v_{i_1,3}$ , starting our search for  $v_{i_1,3}$  in the  $i_2$ th column with the vector pointed to by  $t_{i_2}$ , starting our search in the  $i_3$ th column with the vector pointed to by  $t_{i_3}$  and so on.

In the second case we have shown that  $v_{i_1,2} \in V(p, D)$  so  $A$  becomes equal to  $A \cup \{v_{i_1,2}\}$  and we proceed to examining  $v_{i_1,3}$ , again starting our searches in the  $i_2$ th,  $i_3$ th,... columns with the vectors pointed to by  $t_{i_2}, t_{i_3}, \dots$  respectively.

In the third case we have shown not only that  $v_{i_1,2} \notin V(p, D)$  but also that no vectors below  $v_{i_1,2}$  in the  $i_1$ th column are in  $V(p, D)$  because any such vector would be greater than the vector at the bottom of the  $i_l$ th column. We can therefore stop the process and return the set  $A$  which is equal to  $V(p, D)$ .

This process of computing  $V(p, D)$  is repeated for each pattern in the list pointed to by  $P$  after the execution of `SETIFY_PATTERN_VECTOR_SEQUENCES`.

The foregoing procedure is implemented in the algorithm `COMPUTE_TECS` which is shown in Figure 42.

On each iteration of the outer `while` loop in `COMPUTE_TECS` (Figure 42, lines 7–47) the vector set  $V(p, D)$  is computed for one pattern in  $P''(D)$ . The pointer  $p$  is used to access the `PATTERN_NODE` of the pattern  $p$  being processed on a given iteration.

The pointers  $t_{i_1} \dots t_{i_{|p|}}$  are implemented using a right-directed linked list of `VECTOR_NODES` headed by `T`. The `down` field of the  $l$ th node in this list implements  $t_{i_l}$ —that is, it points to the `VECTOR_NODE` in the down-directed list headed by  $\Phi_{\delta_i} \uparrow \text{down}$  whose vector is the one to be pointed to by  $t_{i_l}$  (see Figure 20).

The outer `while` loop in lines 7 to 47 of `SIATEC` is divided into two main parts. In lines 8–23, the linked list headed by `T` (implementing the pointers  $t_{i_1}, t_{i_2}, \dots, t_{i_{|p|}}$ ) for the previous pattern is first deallocated

```

COMPUTE_TECS
1   local variables
2   p : a PATTERN_NODE pointer
3   q, T, t, v : VECTOR_NODE pointers
4   FINISHED, VECTOR_EQUAL : Booleans

5   p ← P
6   T = NULL
7   while p ≠ NULL
8     q ← p↑pattern
9     if T ≠ NULL
10      t = T
11      while t ≠ NULL
12        t↓down ← NULL
13        t ← t↑right
14      T ← DISPOSE_OF_VECTOR_NODE(T)
15    T ← MAKE_NEW_VECTOR_NODE
16    t ← T
17    t↓down ← q↓down↓down
18    q ← q↑right
19    while q ≠ NULL
20      t↑right ← MAKE_NEW_VECTOR_NODE
21      t ← t↑right
22      t↓down ← q↓down↓down
23      q ← q↑right
24    FINISHED ← FALSE
25    while not FINISHED
26      t ← T↑right
27      VECTOR_EQUAL ← TRUE
28      while t ≠ NULL and VECTOR_EQUAL
29        while t↓down ≠ NULL and VL(t↓down↑vector, T↓down↑vector)
30          t↓down ← t↓down↓down
31        if t↓down = NULL or VL(T↓down↑vector, t↓down↑vector)
32          VECTOR_EQUAL ← FALSE
33        if t↓down = NULL
34          FINISHED ← TRUE
35        t ← t↑right
36      if VECTOR_EQUAL
37        if p↑vectors = NULL
38          p↑vectors ← MAKE_NEW_VECTOR_NODE
39          v ← p↑vectors
40        else
41          v↑right ← MAKE_NEW_VECTOR_NODE
42          v ← v↑right
43          v↑vector ← T↓down↑vector
44          T↓down ← T↓down↓down
45          if T↓down = NULL
46            FINISHED ← TRUE
47    p ← p↑right

```

Figure 42: COMPUTE\_TECS algorithm.

(lines 8–14) and then a new list is initialized for the current pattern begin processed (lines 15–23). To start with, each pointer  $t_{i_k}$  is set to point to  $v_{i_k,1}$ .

The second main part of the outer `while` loop is concerned with actually computing and storing  $V(p, D)$  for the pattern stored in the `PATTERN_NODE` pointed to by `p`. In line 26, the pointer `t` is first made to point to the node in the list headed by `T` that implements  $t_{i_2}$ . `t↑down` now points to  $\Phi_{v_{i_2,1}}$ . In lines 29–30, the pointer `t↑down` steps down the list headed by  $\Phi_{\delta_{i_2}}$  `↑down` (that is, the implementation of the  $i_2$ th column in Table 2) until one of the following two events occurs:

1. a vector is found which is not less than `T↑down↑vector` (the implementation of  $t_{i_1}$ );
2. `t↑down` becomes `NULL`—that is, it gets to the bottom of the list without finding a vector that is not less than `T↑down↑vector`.

In lines 31–32, the boolean variable `VECTOR_EQUAL` which was initialized to `TRUE` in line 24 is made equal to `FALSE` if we fail to find a vector in the  $i_2$ th column that is equal to `T↑down↑vector` (the vector pointed to by  $t_{i_1}$ ). In lines 33–34, the boolean variable `FINISHED` is set to `TRUE` if `t↑down` becomes `NULL`. The `while` loop in lines 28–35 iterates until one of the three terminating conditions listed above (see page 48) is satisfied.

Lines 36–43 of `COMPUTE_TECS` corresponds to the vector pointed to by  $t_{i_1}$  becoming included in the set  $A$  in the event of this vector being a member of  $V(p, D)$ . By the time line 46 is reached, the set  $V(p, D)$  has been computed for another pattern and stored in the `vectors` field of the `PATTERN_NODE` for that pattern.

The worst-case time taken to compute  $V(p, D)$  for a single pattern using the above procedure is  $O(|p|kn)$ . This is because each column in Table 2 contains  $n$ ,  $k$ -dimensional vectors and, in the worst case, for each datapoint  $\delta_{i_l} \in \phi$  we have to scan down the whole of the  $i_l$ th column in Table 2. The total worst-case running time of `COMPUTE_TECS` is therefore  $O(\sum_{p \in P''(D)} |p|kn)$ . But we know from Theorem 3 that

$$\sum_{p \in P'(D)} |p| \leq \frac{n(n-1)}{2}$$

and that therefore

$$\sum_{p \in P''(D)} |p| \leq \frac{n(n-1)}{2}$$

since  $P''(D) \subseteq P'(D)$ . Therefore

$$\sum_{p \in P''(D)} |p|kn \leq kn \times \frac{n(n-1)}{2}$$

which implies that the worst-case running time of `COMPUTE_TECS` is  $O(kn^3)$ .

Figure 43 shows the data structure that results after `COMPUTE_TECS` has been executed for the dataset in Figure 23. Note that each set  $V(p, D)$  is stored as a right-directed linked list headed by the `vectors` field of the `PATTERN_NODE` for  $p$ .

It is straightforward to show that  $O(kn^3)$  is a loose upper bound on the space occupied by the data structure generated by `COMPUTE_TECS`. However, we believe that a significantly better tight upper bound exists even though as yet we have not succeeded in finding one. Our belief stems from the fact that so far we have not been able to dream up a dataset that would cause the data structure generated by `COMPUTE_TECS` to occupy more than  $O(kn^2)$  space.

The loose upper bound of  $O(kn^3)$  can be derived as follows. We first need to prove the following lemma.

**Lemma 12** *If  $D$  is a dataset and  $p$  is a pattern such that  $p \subseteq D$  and  $|D| = n$  and  $V(p, D)$  is as defined in Eq.30 then*

$$|p| + |V(p, D)| \leq n + 1$$

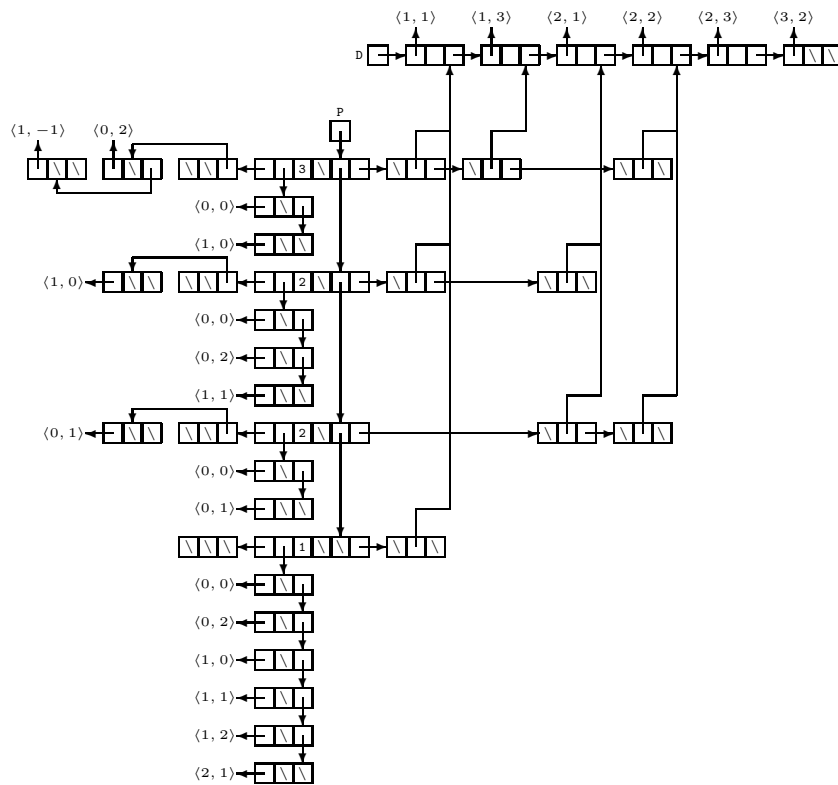


Figure 43: The data structure that results after COMPUTE\_TCS has executed for the dataset in Figure 23.

*Proof*

Let  $\Delta = \langle \delta_1, \delta_2, \dots, \delta_n \rangle$  be the ordered set that results from sorting the datapoints in  $D$  into increasing order and let  $\phi = \langle \delta_{i_1}, \delta_{i_2}, \dots, \delta_{i_{|p|}} \rangle$  be the ordered set that results from sorting the datapoints in  $p$  into increasing order.

If  $v \in V(p, D)$  and we translate  $p$  by  $v$  then the datapoint  $\delta_{i_1}$  will be mapped onto  $\delta_{i_1} + v$  and  $\delta_{i_1} + v \in \Delta$  because  $v \in V(p, D)$ . Therefore there exists a datapoint  $\delta_k \in \Delta$  such that  $\delta_k = \delta_{i_1} + v$ .

Moreover, we know that  $\delta_{i_2} + v \in \Delta$  and that  $\delta_{i_2} + v > \delta_k$  because  $\delta_{i_2} > \delta_{i_1}$ . Similarly, we know that all the points

$$\delta_{i_3} + v, \delta_{i_4} + v, \dots, \delta_{i_{|p|}} + v$$

are elements of  $\Delta$  greater than  $\delta_k$ . Therefore there must be at least  $|p| - 1$  datapoints in  $\Delta$  that are greater than  $\delta_k$ . Therefore

$$n - k \geq |p| - 1 \Rightarrow n + 1 - |p| \geq k$$

There are therefore at most  $n - |p| + 1$  distinct points in  $D$  onto which  $\delta_{i_1}$  could be mapped by a vector in  $V(p, D)$ . Therefore  $|V(p, D)| \leq n - |p| + 1$  which implies that  $|p| + |V(p, D)| \leq n + 1$ . ■

Clearly, the space occupied by the data structure generated by `COMPUTE_TECS` is

$$O \left( \sum_{p \in P''(D)} k(|p| + |V(p, D)|) \right) \quad (94)$$

(see Figure 43). Lemma 12 tells us that  $|p| + |V(p, D)| \leq n + 1$  therefore the space complexity is at most

$$O \left( \sum_{p \in P''(D)} k(n + 1) \right) = O(kn |P''(D)|) \quad (95)$$

Recall that  $P''(D)$  is derived from  $P'(D)$  by picking just one pattern from each subclass in a partition of  $P'(D)$ . Therefore the cardinality of  $P''(D)$  is less than or equal to that of  $P'(D)$  and from Theorem 3 we know that

$$|P'(D)| \leq \frac{n(n-1)}{2}$$

Therefore  $|P''(D)| < n^2$  which, substituting into Eq.95, gives an upper bound on the space used of  $O(kn^3)$ .

## 6.7 Outputting the results

The final step in `SIATEC` is to output the set of TECs  $T'(D)$  computed by `COMPUTE_TECS`. This task is carried out by the procedure `OUTPUT_TECS` called in line 13 of `SIATEC` and defined Figure 44. Figure 45 shows the output produced by `OUTPUT_TECS` for the dataset shown in Figure 23.

Each line in the output represents a TEC as an ordered pair  $\langle p, V(p, D) \rangle$  with the pattern  $p$  printed before the colon and the vector set  $V(p, D)$  printed after the colon.

The worst-case running time of `OUTPUT_TECS` is clearly the same as the worst-case space complexity of the data structure produced by `COMPUTE_TECS`—that is, it has a loose upper bound of  $O(kn^3)$ .

## 7 SIA

As described above, `SIATEC` can be used to compute efficiently the complete set of MTP TECs,  $T(D)$ , for a dataset  $D$ . (`SIATEC` actually computes  $T'(D)$  but from Lemma 6 (p.12) we know that  $T(D) = T'(D) \cup \{\{D\}\}$ .) However, for some applications it may be sufficient to know the maximal translatable pattern  $p(v, D)$  for each vector in  $\mathcal{V}(D)$  (see Eq.32, p.21). We know from Lemma 2 (p.9) that the maximal

```

OUTPUT_TECS
1   local variables
2   p : a PATTERN_NODE pointer
3   q, v : VECTOR_NODE pointers

4   p ← P
5   while p ≠ NULL
6     q ← p↓pattern
7     while q ≠ NULL
8       PRINT_VECTOR(q↑down↑vector)
9       PRINT(' ')
10      q ← q↑right
11      PRINT(': ')
12      v ← p↓vectors
13      while v ≠ NULL
14        PRINT_VECTOR(v↑vector)
15        PRINT(' ')
16        v ← v↑right
17      PRINT_NEWLINE
18      p ← p↓right

```

Figure 44: OUTPUT\_TECS algorithm.

```

<1,1> <1,3> <2,2> : <0,0> <1,0>
<1,1> <2,1> : <0,0> <0,2> <1,1>
<2,1> <2,2> : <0,0> <0,1>
<1,1> : <0,0> <0,2> <1,0> <1,1> <1,2> <2,1>

```

Figure 45: The output produced by OUTPUT\_TECS for the dataset shown in Figure 23.

```

SIA(FN : dataset file-name, SD : bit-vector indicating selected dimensions)
1   READ_DATASET(FN,SD)
2   SORT_DATASET
3   SETIFY_DATASET
4   SIA_COMPUTE_VECTORS
6   SIA_SORT_VECTORS
7   OUTPUT_VECTOR_PATTERN_PAIRS

```

Figure 46: SIA algorithm.

translatable pattern for  $-v$  can be found by translating the maximal translatable pattern for  $v$  by the vector  $v$  itself. We also know from Eq.15 (p.11) that the maximal translatable pattern for the zero vector is the complete dataset. Therefore we do not actually have to compute  $p(v, D)$  explicitly for every vector in  $\mathcal{V}(D)$ . It is sufficient to compute  $p(v, D)$  for every vector in the set

$$\mathcal{V}'(D) = \{d_1 - d_2 \mid d_1, d_2 \in D \wedge d_1 > d_2\}$$

(see Eq.31, p.21). That is, it is sufficient to compute

$$\begin{aligned} \{p(v, D) \mid v \in \mathcal{V}'(D)\} &= \{p(v, D) \mid v \in \{d_1 - d_2 \mid d_1, d_2 \in D \wedge d_1 > d_2\}\} \\ &= \{p(d_1 - d_2, D) \mid d_1, d_2 \in D \wedge d_1 > d_2\} \\ &= P'(D) \quad (\text{from Definition 14, p.10}) \end{aligned}$$

Lines 1 to 7 of SIATEC compute  $P'(D)$ . However, for reasons discussed above (see p.21) the procedure COMPUTE\_VECTORS called in line 4 of SIATEC computes  $\mathcal{V}(D)$  whereas if our final goal is only to compute  $P'(D)$  then we only have to compute the much smaller set  $\mathcal{V}'(D)$ . Also, the procedures CONSTRUCT\_VECTOR\_TABLE and CONSTRUCT\_PATTERN\_LIST, called in lines 5 and 7 of SIATEC respectively, are not necessary if one's goal is only to compute  $P'(D)$  and not  $T'(D)$ .

In this section we therefore present SIA, an algorithm that efficiently computes  $P'(D)$  for a dataset  $D$ . Strictly speaking, SIA computes the set of ordered pairs

$$\mathcal{S}(D) = \{\langle v, p(v, D) \rangle \mid v \in \mathcal{V}'(D)\} \quad (96)$$

For a  $k$ -dimensional dataset of size  $n$ , the worst-case running time of SIA is  $O(kn^2 \log_2 n)$  and the worst-case space complexity is  $O(kn^2)$ .

Figure 46 shows the SIA algorithm. Like SIATEC, SIA uses linked-list data structures constructed from NUMBER\_NODES and VECTOR\_NODES as defined in Figure 3 (see p.14). PATTERN\_NODES are not used in SIA. SIA uses two global variables:

1. a VECTOR\_NODE pointer called D which is used (as in SIATEC) to head the list of VECTOR\_NODES that stores the dataset;
2. a VECTOR\_NODE pointer called V which is used to access the linked list structure used to store  $\mathcal{V}'(D)$ .

The two arguments to SIA, FN and SD, are as for SIATEC (see p.9). The procedures READ\_DATASET, SORT\_DATASET and SETIFY\_DATASET called in the first three lines of SIA are identical to those called in lines 1 to 3 of SIATEC (see section 6.2).

The procedure COMPUTE\_VECTORS, called in line 4 of SIATEC and defined in Figure 18 on page 22, is replaced in SIA with the procedure SIA\_COMPUTE\_VECTORS which is defined in Figure 47.

Figure 48 shows the data structure that results after SIA\_COMPUTE\_VECTORS has executed when SIA is carried out on the dataset shown in Figure 23 on page 25. The resulting data structure is a representation

```

SIA_COMPUTE_VECTORS
1   local variables
2   d1,d2,p,v : VECTOR_NODE pointers

3   V ← NULL
4   if D ≠ NULL and D↑right ≠ NULL
5       d1 ← D
6       d2 ← d1↑right
7       V ← MAKE_NEW_VECTOR_NODE
8       v ← V
9       repeat
10          p ← v
11          repeat
12              p↓down ← MAKE_NEW_VECTOR_NODE
13              p ← p↓down
14              p↑right ← d1
15              p↑vector ← VM(d2↑vector,d1↑vector)
16              d2 ← d2↑right
17          until d2 = NULL
18          d1 ← d1↑right
19          if d1↑right ≠ NULL
20              v↑right ← MAKE_NEW_VECTOR_NODE
21              v ← v↑right
22              d2 ← d1↑right
23          until d1↑right = NULL

```

Figure 47: SIA\_COMPUTE\_VECTORS algorithm.

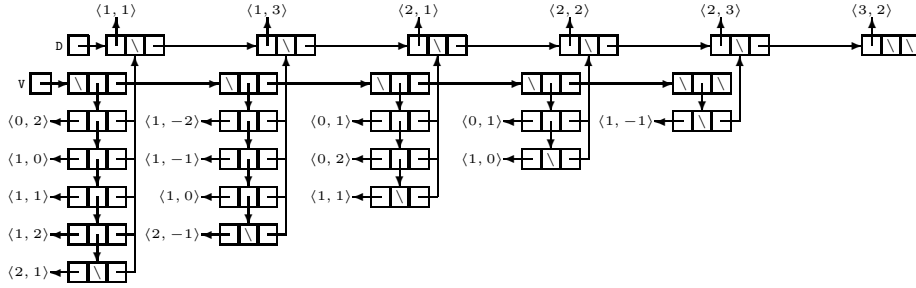


Figure 48: The data structure that results after `SIA_COMPUTE_VECTORS` has executed when `SIA` is carried out on the dataset shown in Figure 23 on page 25.

of the vector table shown in Figure 24 on page 25. The data structure headed by `V` after execution of `SIA_COMPUTE_VECTORS` therefore represents the same vector table as the data structure headed by `V` after `CONSTRUCT_VECTOR_TABLE` (Figure 21, p.24) has executed in line 5 of `SIATEC`. However, in `SIA` this vector table is represented more directly (compare Figure 22 (p.24) with Figure 48.)

For a  $k$ -dimensional dataset of size  $n$ , line 15 of `SIA_COMPUTE_VECTORS` executes  $\frac{n(n-1)}{2}$  times as compared to the  $n^2$  times that line 15 of `COMPUTE_VECTORS` executes in `SIATEC`. For a given dataset, `SIA_COMPUTE_VECTORS` therefore makes less than half the number of computations made by `COMPUTE_VECTORS`. The worst-case running time of `SIA_COMPUTE_VECTORS` is therefore  $O(kn^2)$  and the worst-case space complexity is also, clearly,  $O(kn^2)$ .

The procedure `SORT_VECTORS` (Figure 26, p.27) called in line 6 of `SIATEC` is replaced in `SIA` with the procedure `SIA_SORT_VECTORS` which is defined in Figure 49. The only difference between `SIA_SORT_VECTORS` and `SORT_VECTORS` is that in line 14 of `SIA_SORT_VECTORS` the function `SIA_MERGE_VECTOR_COLUMNS` is called, whereas in line 14 of `SORT_VECTORS` the function `MERGE_COLUMN_VECTORS` (Figure 27, p.28) is called. The function `SIA_MERGE_VECTOR_COLUMNS` is defined in Figure 50.

The only difference between `SIA_MERGE_VECTOR_COLUMNS` and `MERGE_VECTOR_COLUMNS` is that in line 9 of `SIA_MERGE_VECTOR_COLUMNS` the arguments are `b↑vector` and `a↑vector` rather than `b↑right↑vector` and `a↑right↑vector`. This reflects the difference between the data structure headed by `V` in `SIATEC` and the data structure headed by `V` in `SIA`. The extra layer of indirection in `SIATEC` is necessary in order to preserve the ordering of the data structure that stores  $\mathcal{V}(D)$  so that the ‘column-scanning’ technique for finding `TECs` can be employed in `COMPUTE_TECs`.

Like `SORT_VECTORS`, `SIA_SORT_VECTORS` is an implementation of merge sort. The worst-case running time for `SIA_SORT_VECTORS` is therefore  $O(kn^2 \log_2 n)$  for a  $k$ -dimensional dataset of size  $n$ .

Figure 51 shows the data structure that results after `SIA_SORT_VECTORS` has executed when `SIA` is carried out on the dataset in Figure 23. This data structure represents the list shown in Figure 25.

Finally, in line 7 of `SIA`, the procedure `OUTPUT_VECTOR_PATTERN_PAIRS` is called. This algorithm is defined in Figure 52 and it is based on the `CONSTRUCT_PATTERN_LIST` algorithm (see Figure 29, p.35). `OUTPUT_VECTOR_PATTERN_PAIRS` simply outputs the set  $\mathcal{S}(D)$  defined in Eq.96.

Figure 53 shows the output generated by `OUTPUT_VECTOR_PATTERN_PAIRS` when `SIA` is carried out on the dataset in Figure 23. Each line of the output represents an ordered pair  $\langle v, p(v, D) \rangle$  in  $\mathcal{S}(D)$ , with the vector  $v$  being printed before the colon and the datapoints in the set  $p(v, D)$  being printed in increasing order after the colon.

For a  $k$ -dimensional dataset of size  $n$ , the worst-case running time of `OUTPUT_VECTOR_PATTERN_PAIRS` is clearly  $O(kn^2)$  since the number of vectors to be printed is  $O(|Z(D)|)$  (see Definition 16) and  $|Z(D)| = O(n^2)$  as proved in Lemma 8 on page 30.

For a  $k$ -dimensional dataset of size  $n$ , the overall worst-case running time of `SIA` is therefore

```

SIA_SORT_VECTORS
1   local variables
2   BEFORE_A, A, B, AFTER_B, C : VECTOR_NODE pointers

3   while V ≠ NULL and V↑right ≠ NULL
4     BEFORE_A ← NULL
5     A ← V
6     V ← NULL
7     repeat
8       if V ≠ NULL
9         BEFORE_A↑right ← NULL
10        B ← A↑right
11        A↑right ← NULL
12        AFTER_B ← B↑right
13        B↑right ← NULL
14        C ← SIA_MERGE_VECTOR_COLUMNS(A,B)
15        if V = NULL
16          V ← C
17        else
18          BEFORE_A↑right ← C
19          C↑right ← AFTER_B
20          BEFORE_A ← C
21          A ← BEFORE_A↑right
22        until A = NULL or A↑right = NULL
23    ▷ Finally we delete the sentinel node pointed to by V.
24    if V ≠ NULL
25      A ← V↑down
26      V↑down ← NULL
27      DISPOSE_OF_VECTOR_NODE(V)
28      V ← A

```

Figure 49: SIA\_SORT\_VECTORS algorithm.

```

SIA_MERGE_VECTOR_COLUMNS(A, B : VECTOR_NODE pointers)
1   local variables
2   a, b, C, c : VECTOR_NODE pointers

3   a ← A↑down
4   b ← B↑down
5   C ← A
6   C↑down ← NULL
7   c ← C
8   while a ≠ NULL and b ≠ NULL
9     if VL(b↑vector, a↑vector)
10      c↑down ← b
11      b ← b↑down
12    else
13      c↑down ← a
14      a ← a↑down
15      c ← c↑down
16      c↑down ← NULL
17  if a = NULL
18      c↑down ← b
19  else
20      c↑down ← a
21  return C

```

Figure 50: SIA\_MERGE\_VECTOR\_COLUMNS algorithm.

$O(kn^2 \log_2 n)$  and the worst-case space complexity is  $O(kn^2)$ .

## References

- D. Bainbridge, C. Nevill-Manning, I. Witten, L. Smith, and R. McNab. Towards a digital library of popular music. In *Proceedings of the fourth ACM conference on digital libraries*, pages 161–169, Berkeley, CA., 1999. Association of Computing Machinery, ACM.
- I. Bent and W. Drabkin. *Analysis*. New Grove Handbooks in Music. Macmillan, 1987.
- E. J. Borowski and J. M. Borwein. *Dictionary of Mathematics*. Collins, 1989.
- E. Cambouropoulos. *Towards a General Computational Theory of Musical Structure*. PhD thesis, University of Edinburgh, Feb. 1998.
- T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. M.I.T. Press, Cambridge, Mass., 1990. ISBN 0-262-53091-0.
- M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
- A. Floratos and I. Rigoutsos. Method and apparatus for pattern discovery in 1-dimensional event streams, Aug. 2000. U.S. Patent no. 6,108,666.
- J.-L. Hsu, C.-C. Liu, and A. L. Chen. Efficient repeating pattern finding in music databases. In *Proceedings of the 1998 ACM 7th International Conference on Information and Knowledge Management*, pages 281–288. Association of Computing Machinery, 1998.

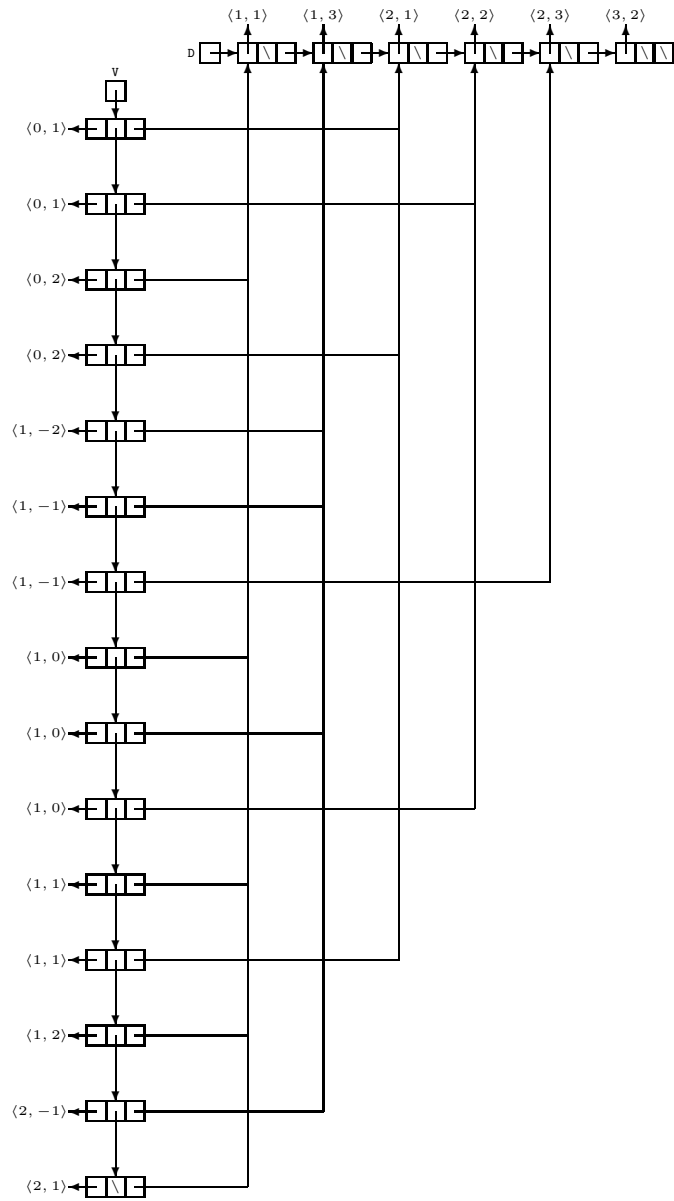


Figure 51: The data structure headed by  $v$  after `SIA_SORT_VECTORS` has executed when `SIA` is carried out on the dataset in Figure 23.

```

OUTPUT_VECTOR_PATTERN_PAIRS
1   local variables
2   v1, v2: VECTOR_NODE pointers

3   if V ≠ NULL
4   v1 ← V
5   while v1 ≠ NULL
6   PRINT_VECTOR(v1↑vector)
7   PRINT(' : ')
8   PRINT_VECTOR(v1↑right↑vector)
9   PRINT(' ')
10  v2 ← v1↑down
11  while v2 ≠ NULL and VE(v2↑vector,v1↑vector)
12  PRINT_VECTOR(v1↑right↑vector)
13  PRINT(' ')
14  v2 ← v2↑down
15  PRINT_NEW_LINE
16  v1 ← v2

```

Figure 52: OUTPUT\_VECTOR\_PATTERN\_PAIRS algorithm.

```

<0,1> : <2,1> <2,2>
<0,2> : <1,1> <2,1>
<1,-2> : <1,3>
<1,-1> : <1,3> <2,3>
<1,0> : <1,1> <1,3> <2,2>
<1,1> : <1,1> <2,1>
<1,2> : <1,1>
<2,-1> : <1,3>
<2,1> : <1,1>

```

Figure 53: The output generated by OUTPUT\_VECTOR\_PATTERN\_PAIRS for the dataset in Figure 23 on page 25.

- S. Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience*, 29(13): 1149–1171, 1999.
- K. Lemström. *String Matching Techniques for Music Retrieval*. PhD thesis, University of Helsinki, Faculty of Science, Department of Computer Science, 2000. Report A-2000-4.
- F. Lerdahl and R. Jackendoff. *A Generative Theory of Tonal Music*. M.I.T. Press, Cambridge, Mass., 1983.
- J.-J. Nattiez. *Fondements d'une sémiologie de la musique*. Union Générale d'Éditions, Paris, 1975.
- P.-Y. Rolland. Discovering patterns in musical sequences. *Journal of New Music Research*, 28(4):334–350, 1999.
- N. Ruwet. Méthodes d'analyse en musicologie. *Revue belge de musicologie*, 20:65ff., 1966. Republished in (Ruwet, 1972, 100–134); English translation in *Music Analysis*, Vol.5, 1986.
- N. Ruwet. *Langage, Musique, Poésie*. Éditions du seuil, 27, rue Jacob, Paris VI., 1972.