

# METHOD OF PATTERN DISCOVERY

*PCT Patent application submitted on 23 May 2002*

*(Final draft)*

David Meredith

dave@titanmusic.com

Geraint A. Wiggins

geraint@soi.city.ac.uk

Kjell Lemström

klemstro@cs.helsinki.fi

23 May 2002

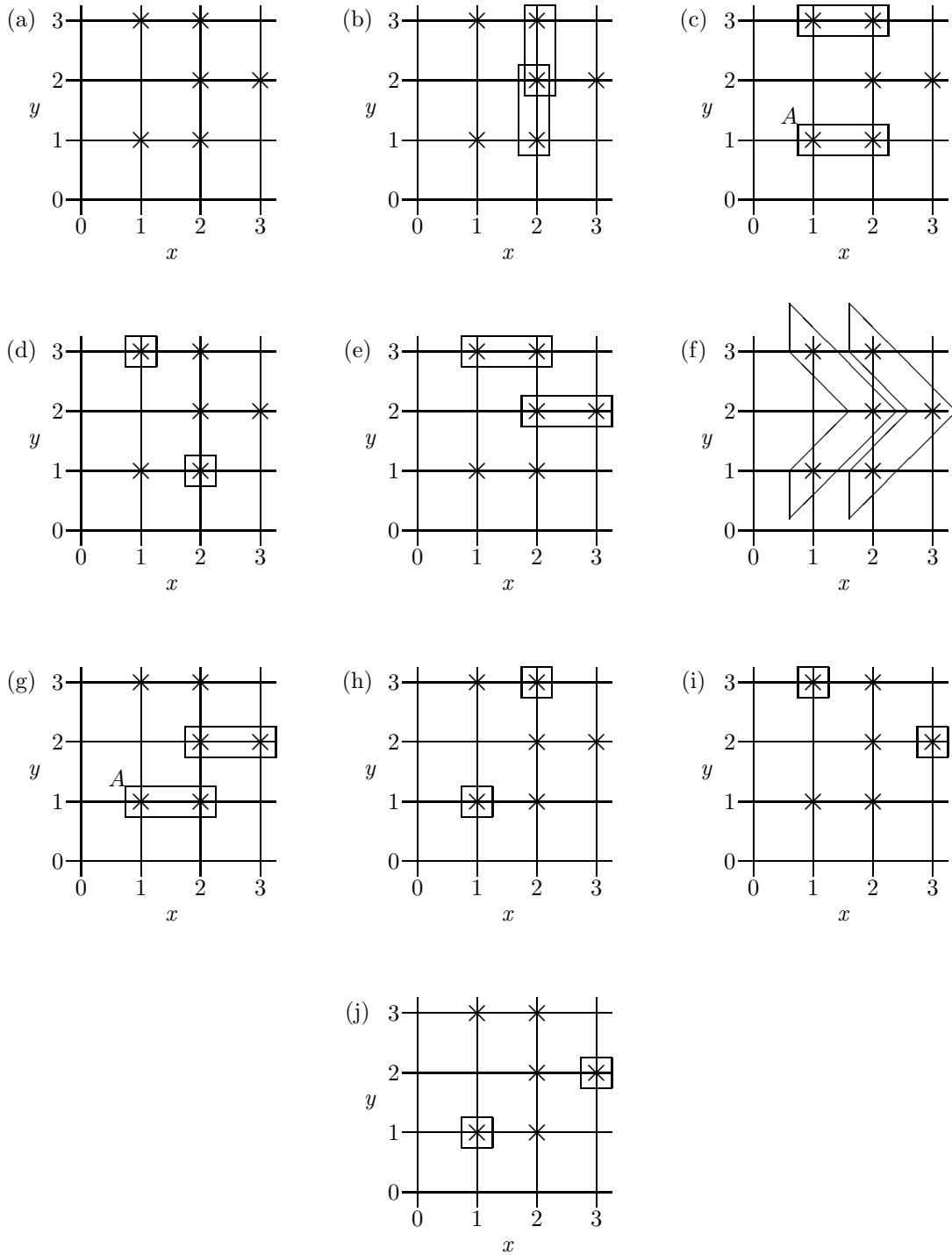


Figure 1: (a) shows a simple 2-dimensional dataset. (b)–(j) show the maximal repeated patterns found by SIA in the dataset in (a).

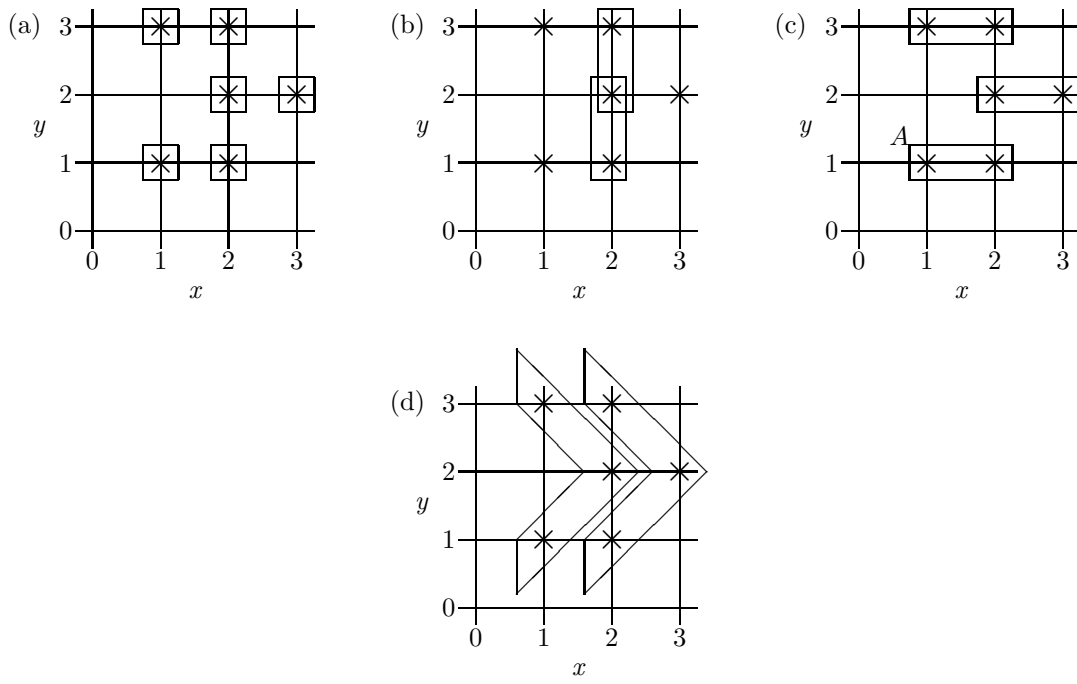


Figure 2: The sets of patterns discovered by SIATEC in the dataset in Figure 1(a).

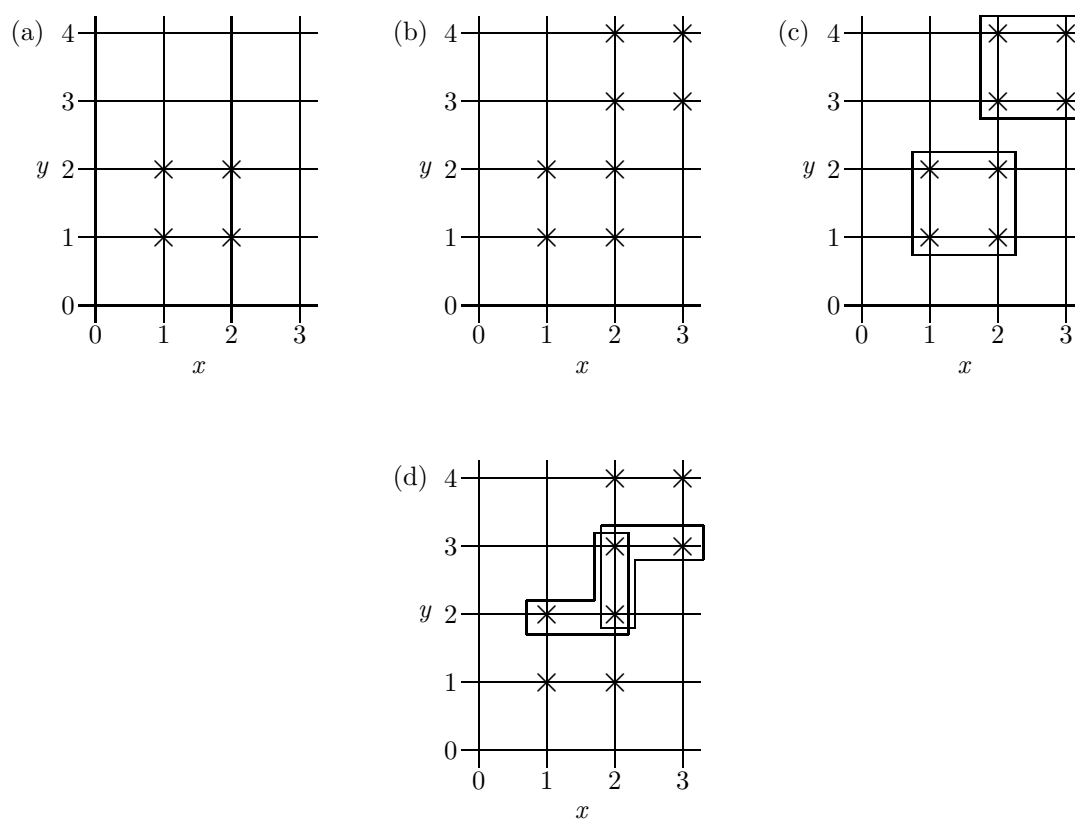


Figure 3: When **SIAME** searches for occurrences of the query pattern (a) in the dataset (b), it finds the exact matches shown in (c). It also finds the closest incomplete matches shown in (d).

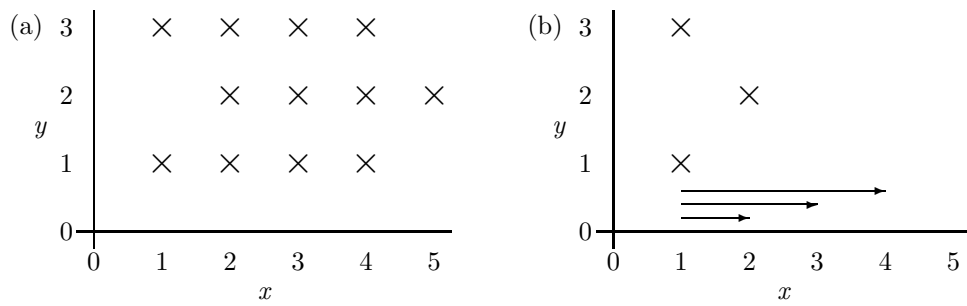


Figure 4: (b) shows the compressed representation generated by **COSIATEC** for the dataset (a). The dataset in (a) can be generated by translating the three-point pattern in (b) by the three vectors represented by arrows.

$$\begin{aligned}
& \{ \langle \langle 0, 1 \rangle, \{ \langle 2, 1 \rangle, \langle 2, 2 \rangle \} \rangle, \\
& \langle \langle 0, 2 \rangle, \{ \langle 1, 1 \rangle, \langle 2, 1 \rangle \} \rangle, \\
& \langle \langle 1, -2 \rangle, \{ \langle 1, 3 \rangle \} \rangle, \\
& \langle \langle 1, -1 \rangle, \{ \langle 1, 3 \rangle, \langle 2, 3 \rangle \} \rangle, \\
& \langle \langle 1, 0 \rangle, \{ \langle 1, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 2 \rangle \} \rangle, \\
& \langle \langle 1, 1 \rangle, \{ \langle 1, 1 \rangle, \langle 2, 1 \rangle \} \rangle, \\
& \langle \langle 1, 2 \rangle, \{ \langle 1, 1 \rangle \} \rangle, \\
& \langle \langle 2, -1 \rangle, \{ \langle 1, 3 \rangle \} \rangle, \\
& \langle \langle 2, 1 \rangle, \{ \langle 1, 1 \rangle \} \rangle \}
\end{aligned}$$

Figure 5: The set  $\mathcal{S}(D)$  for the dataset in Figure 1(a).

$$\begin{aligned}
\{ & \langle \{ \langle 1, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 2 \rangle \} \rangle, \langle \{ 1, 0 \} \rangle, \\
& \langle \{ \langle 1, 1 \rangle, \langle 2, 1 \rangle \} \rangle, \langle \{ 0, 2 \rangle, \langle 1, 1 \rangle \} \rangle, \\
& \langle \{ \langle 2, 1 \rangle, \langle 2, 2 \rangle \} \rangle, \langle \{ 0, 1 \} \rangle, \\
& \langle \{ \langle 1, 1 \rangle \} \rangle, \langle \{ 0, 2 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle \} \rangle \}
\end{aligned}$$

Figure 6: The set  $\mathcal{J}'(D)$  for the dataset in Figure 1(a).

```

1   $m \leftarrow |\mathbf{V}|$ 
2   $i \leftarrow 1$ 
3  PRINT_NEW_LINE
4  PRINT('{')
5  while  $i \leq m$ 
6      PRINT('<')
7      PRINT_VECTOR( $\mathbf{V}[i, 1]$ )
8      PRINT(', {')
9      PRINT_VECTOR( $\mathbf{D}[\mathbf{V}[i, 2]]$ )
10      $j \leftarrow i + 1$ 
11     while  $j \leq m$  and  $\mathbf{V}[j, 1] = \mathbf{V}[i, 1]$ 
12         PRINT(',')
13         PRINT_VECTOR( $\mathbf{D}[\mathbf{V}[j, 2]]$ )
14          $j \leftarrow j + 1$ 
15     PRINT('}')')
16     if  $j \leq m$ 
17         PRINT(',')
18         PRINT_NEW_LINE
19      $i \leftarrow j$ 
20 PRINT('}')')

```

Figure 7: An algorithm for printing out  $\mathcal{S}(D)$  using  $\mathbf{V}$  and  $\mathbf{D}$ .



$$\begin{aligned}
& \{\langle\langle 0, 1 \rangle, \{ \langle 2, 1 \rangle, \langle 2, 2 \rangle \} \rangle, \\
& \langle\langle 0, 2 \rangle, \{ \langle 1, 1 \rangle, \langle 2, 1 \rangle \} \rangle, \\
& \langle\langle 1, -2 \rangle, \{ \langle 1, 3 \rangle \} \rangle, \\
& \langle\langle 1, -1 \rangle, \{ \langle 1, 3 \rangle, \langle 2, 3 \rangle \} \rangle, \\
& \langle\langle 1, 0 \rangle, \{ \langle 1, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 2 \rangle \} \rangle, \\
& \langle\langle 1, 1 \rangle, \{ \langle 1, 1 \rangle, \langle 2, 1 \rangle \} \rangle, \\
& \langle\langle 1, 2 \rangle, \{ \langle 1, 1 \rangle \} \rangle, \\
& \langle\langle 2, -1 \rangle, \{ \langle 1, 3 \rangle \} \rangle, \\
& \langle\langle 2, 1 \rangle, \{ \langle 1, 1 \rangle \} \rangle
\end{aligned}$$

Figure 8: The output of the algorithm in Figure 7 for the dataset in Figure 1(a).

```

1   $m \leftarrow |\mathbf{V}|$ 
2   $i \leftarrow 1$ 
3   $\mathbf{X} \leftarrow \langle \rangle$ 
4  while  $i \leq m$ 
5       $\mathbf{Q} \leftarrow \langle \rangle$ 
6       $j \leftarrow i + 1$ 
7      while  $j \leq m$  and  $\mathbf{V}[j, 1] = \mathbf{V}[i, 1]$ 
8           $\mathbf{Q} \leftarrow \mathbf{Q} \oplus \langle \mathbf{D}[\mathbf{V}[j, 2]] - \mathbf{D}[\mathbf{V}[j - 1, 2]] \rangle$ 
9           $j \leftarrow j + 1$ 
10      $\mathbf{X} \leftarrow \mathbf{X} \oplus \langle \langle i, \mathbf{Q} \rangle \rangle$ 
11      $i \leftarrow j$ 

```

Figure 9: An algorithm for computing  $X$  using  $\mathbf{V}$  and  $\mathbf{D}$ .

$$\begin{aligned}
& \langle \langle 1, \langle \langle 0, 1 \rangle \rangle \rangle, \\
& \langle \langle 3, \langle \langle 1, 0 \rangle \rangle \rangle, \\
& \langle \langle 5, \langle \rangle \rangle, \\
& \langle \langle 6, \langle \langle 1, 0 \rangle \rangle \rangle, \\
& \langle \langle 8, \langle \langle 0, 2 \rangle, \langle 1, -1 \rangle \rangle \rangle, \\
& \langle \langle 11, \langle \langle 1, 0 \rangle \rangle \rangle, \\
& \langle \langle 13, \langle \rangle \rangle, \\
& \langle \langle 14, \langle \rangle \rangle, \\
& \langle \langle 15, \langle \rangle \rangle \rangle
\end{aligned}$$

Figure 10: The ordered set  $\mathbf{X}$  for the dataset in Figure 1(a).

$$\begin{aligned}
& \langle \langle 5, \langle \rangle \rangle \rangle, \\
& \langle \langle 13, \langle \rangle \rangle \rangle, \\
& \langle \langle 14, \langle \rangle \rangle \rangle, \\
& \langle \langle 15, \langle \rangle \rangle \rangle, \\
& \langle \langle 1, \langle \langle 0, 1 \rangle \rangle \rangle \rangle, \\
& \langle \langle 3, \langle \langle 1, 0 \rangle \rangle \rangle \rangle, \\
& \langle \langle 6, \langle \langle 1, 0 \rangle \rangle \rangle \rangle, \\
& \langle \langle 11, \langle \langle 1, 0 \rangle \rangle \rangle \rangle, \\
& \langle \langle 8, \langle \langle 0, 2 \rangle, \langle 1, -1 \rangle \rangle \rangle \rangle
\end{aligned}$$

Figure 11: The ordered set  $\mathbf{Y}$  for the dataset in Figure 1(a).

```

1   $r \leftarrow |\mathbf{Y}|$ 
2   $m \leftarrow |\mathbf{V}|$ 
3   $i \leftarrow 1$ 
4  PRINT_NEW_LINE
5  PRINT('{')
6  if  $r > 0$ 
7      repeat
8           $j \leftarrow \mathbf{Y}[i, 1]$ 
9           $\mathbf{I} \leftarrow \langle \rangle$ 
10         while  $j \leq m$  and  $\mathbf{V}[j, 1] = \mathbf{V}[\mathbf{Y}[i, 1], 1]$ 
11              $\mathbf{I} \leftarrow \mathbf{I} \oplus \langle \mathbf{V}[j, 2] \rangle$ 
12              $j \leftarrow j + 1$ 
13         PRINT('⟨')
14         PRINT_PATTERN( $\mathbf{I}$ )
15         PRINT(',')
16         PRINT_SET_OF_TRANSLATORS( $\mathbf{I}$ )
17         PRINT('⟩')
18         repeat
19              $i \leftarrow i + 1$ 
20         until  $i > r$  or  $\mathbf{Y}[i, 2] \neq \mathbf{Y}[i - 1, 2]$ 
21         if  $i \leq r$ 
22             PRINT(',')
23             PRINT_NEW_LINE
24     until  $i > r$ 
25 PRINT('}')

```

Figure 12: An algorithm for printing out  $\mathcal{J}'(D)$ .

```
PRINT_PATTERN(I)
1   $p \leftarrow |\mathbf{I}|$ 
2  PRINT('{')
3  PRINT_VECTOR(D[I[1]])
4  for  $k \leftarrow 2$  to  $p$ 
5      PRINT(',')
6      PRINT_VECTOR(D[I[ $k$ ]])
7  PRINT('}')
```

Figure 13: The PRINT\_PATTERN algorithm.

```

PRINT_SET_OF_TRANSLATORS(I)
1  p ← |I|
2  n ← |D|
3  if p = 1
4    PRINT('{')
5    for k ← 1 to n
6      if k ≠ I[1]
7        PRINT_VECTOR(W[I[1], k])
8        unless k = n ∨ (k = n - 1 ∧ I[1] = n)
9          PRINT(', ')
10   PRINT('}')
11  else
12   PRINT('{')
13   J ← ⟨⟩
14   for k ← 1 to p
15     J ← J ⊕ ⟨1⟩
16   if I[1] = 1 then J[1] ← 2
17   FINISHED ← FALSE
18   FIRST_VECTOR ← TRUE
19   k ← 2
20   while not FINISHED
21     if J[k] ≤ J[k - 1] then J[k] ← J[k - 1] + 1
22     while J[k] ≤ n - p + k and W[I[k], J[k]] < W[I[k - 1], J[k - 1]]
23       J[k] ← J[k] + 1
24     if J[k] > n - p + k then FINISHED ← TRUE
25     else if W[I[k], J[k]] > W[I[k - 1], J[k - 1]]
26       k ← 2
27       J[1] ← J[1] + 1
28       if J[1] = I[1] then J[1] ← J[1] + 1
29       if J[1] > n - p + 1 then FINISHED ← TRUE
30     else if k = p
31       if not FIRST_VECTOR then PRINT(', ')
32       else FIRST_VECTOR ← FALSE
33       PRINT_VECTOR(W[I[k], J[k]])
34       k ← 1
35       while k ≤ p
36         J[k] ← J[k] + 1
37         if J[k] = I[k] then J[k] ← J[k] + 1
38         if J[k] > n - p + k
39           FINISHED ← TRUE
40         k ← p
41         k ← k + 1
42       k ← 2
43     else k ← k + 1
44   PRINT('}')

```

Figure 14: The PRINT\_SET\_OF\_TRANSLATORS algorithm.

$$\begin{aligned}
& \langle \{ \langle 1, 3 \rangle \}, \{ \langle 0, -2 \rangle, \langle 1, -2 \rangle, \langle 1, -1 \rangle, \langle 1, 0 \rangle, \langle 2, -1 \rangle \} \rangle, \\
& \langle \{ \langle 2, 1 \rangle, \langle 2, 2 \rangle \}, \{ \langle 0, 1 \rangle \} \rangle, \\
& \langle \{ \langle 1, 1 \rangle, \langle 2, 1 \rangle \}, \{ \langle 0, 2 \rangle, \langle 1, 1 \rangle \} \rangle, \\
& \langle \{ \langle 1, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 2 \rangle \}, \{ \langle 1, 0 \rangle \} \rangle
\end{aligned}$$

Figure 15: The output of the algorithm in Figure 12 for the dataset in Figure 1(a).



$$\begin{aligned}
& \langle \langle -1, -1 \rangle, \langle 2, 2 \rangle \rangle, \\
& \langle \langle -1, 0 \rangle, \langle 2, 1 \rangle \rangle, \\
& \langle \langle -1, 0 \rangle, \langle 2, 2 \rangle \rangle, \\
& \langle \langle -1, 1 \rangle, \langle 2, 1 \rangle \rangle, \\
& \langle \langle 0, -1 \rangle, \langle 1, 2 \rangle \rangle, \\
& \langle \langle 0, -1 \rangle, \langle 2, 2 \rangle \rangle, \\
& \langle \langle 0, 0 \rangle, \langle 1, 1 \rangle \rangle, \\
& \langle \langle 0, 0 \rangle, \langle 1, 2 \rangle \rangle, \\
& \langle \langle 0, 0 \rangle, \langle 2, 1 \rangle \rangle, \\
& \langle \langle 0, 0 \rangle, \langle 2, 2 \rangle \rangle, \\
& \langle \langle 0, 1 \rangle, \langle 1, 1 \rangle \rangle, \\
& \langle \langle 0, 1 \rangle, \langle 2, 1 \rangle \rangle, \\
& \langle \langle 0, 1 \rangle, \langle 2, 2 \rangle \rangle, \\
& \langle \langle 0, 2 \rangle, \langle 2, 1 \rangle \rangle, \\
& \langle \langle 0, 2 \rangle, \langle 2, 2 \rangle \rangle, \\
& \langle \langle 0, 3 \rangle, \langle 2, 1 \rangle \rangle, \\
& \langle \langle 1, -1 \rangle, \langle 1, 2 \rangle \rangle, \\
& \langle \langle 1, 0 \rangle, \langle 1, 1 \rangle \rangle, \\
& \langle \langle 1, 0 \rangle, \langle 1, 2 \rangle \rangle, \\
& \langle \langle 1, 1 \rangle, \langle 1, 1 \rangle \rangle, \\
& \langle \langle 1, 1 \rangle, \langle 1, 2 \rangle \rangle, \\
& \langle \langle 1, 1 \rangle, \langle 2, 2 \rangle \rangle, \\
& \langle \langle 1, 2 \rangle, \langle 1, 1 \rangle \rangle, \\
& \langle \langle 1, 2 \rangle, \langle 1, 2 \rangle \rangle, \\
& \langle \langle 1, 2 \rangle, \langle 2, 1 \rangle \rangle, \\
& \langle \langle 1, 2 \rangle, \langle 2, 2 \rangle \rangle, \\
& \langle \langle 1, 3 \rangle, \langle 1, 1 \rangle \rangle, \\
& \langle \langle 1, 3 \rangle, \langle 2, 1 \rangle \rangle, \\
& \langle \langle 2, 1 \rangle, \langle 1, 2 \rangle \rangle, \\
& \langle \langle 2, 2 \rangle, \langle 1, 1 \rangle \rangle, \\
& \langle \langle 2, 2 \rangle, \langle 1, 2 \rangle \rangle, \\
& \langle \langle 2, 3 \rangle, \langle 1, 1 \rangle \rangle \rangle
\end{aligned}$$

Figure 16: The ordered set  $\mathbf{V}_{\text{SIAME}}$  computed by Step 2 of SIAME for the pattern in Figure 3(a) and the dataset in Figure 3(b).

```

 $r \leftarrow |\mathbf{V}_{\text{SIAME}}|$ 
 $\mathbf{N} \leftarrow \langle \rangle$ 
 $\ell \leftarrow 0$ 
 $i \leftarrow 1$ 
for  $k \leftarrow 2$  to  $r$ 
   $\ell \leftarrow \ell + 1$ 
  if  $\mathbf{V}_{\text{SIAME}}[k-1, 1] \neq \mathbf{V}_{\text{SIAME}}[k, 1]$ 
     $\mathbf{N} \leftarrow \mathbf{N} \oplus \langle \langle \ell, i \rangle \rangle$ 
     $i \leftarrow k$ 
     $\ell \leftarrow 0$ 
if  $r > 0$ 
   $\mathbf{N} \leftarrow \mathbf{N} \oplus \langle \langle \ell + 1, i \rangle \rangle$ 
return  $\mathbf{N}$ 

```

Figure 17: An algorithm for computing  $N$  using  $\mathbf{V}_{\text{SIAME}}$ .

$$\langle \begin{array}{l} \langle 1, 1 \rangle, \\ \langle 2, 2 \rangle, \\ \langle 1, 4 \rangle, \\ \langle 2, 5 \rangle, \\ \langle 4, 7 \rangle, \\ \langle 3, 11 \rangle, \\ \langle 2, 14 \rangle, \\ \langle 1, 16 \rangle, \\ \langle 1, 17 \rangle, \\ \langle 2, 18 \rangle, \\ \langle 3, 20 \rangle, \\ \langle 4, 23 \rangle, \\ \langle 2, 27 \rangle, \\ \langle 1, 29 \rangle, \\ \langle 2, 30 \rangle, \\ \langle 1, 32 \rangle \end{array} \rangle$$

Figure 18:  $\mathbf{N}$  for the pattern in Figure 3(a) and the dataset in Figure 3(b).

$$\langle \begin{array}{l} \langle 4, 23 \rangle, \\ \langle 4, 7 \rangle, \\ \langle 3, 20 \rangle, \\ \langle 3, 11 \rangle \\ \langle 2, 30 \rangle, \\ \langle 2, 27 \rangle, \\ \langle 2, 18 \rangle, \\ \langle 2, 14 \rangle \\ \langle 2, 5 \rangle, \\ \langle 2, 2 \rangle, \\ \langle 1, 32 \rangle, \\ \langle 1, 29 \rangle \\ \langle 1, 17 \rangle, \\ \langle 1, 16 \rangle, \\ \langle 1, 4 \rangle, \\ \langle 1, 1 \rangle \end{array} \rangle$$

Figure 19:  $\mathbf{N}'$  for the pattern in Figure 3(a) and the dataset in Figure 3(b).

```

s ← |N'|
M ← ⟨⟩
for i ← 1 to s
  Q ← ⟨⟩
  for j ← 0 to N'[i, 1] - 1
    Q ← Q ⊕ ⟨VSIAME[N'[i, 2] + j, 2]⟩
  M ← M ⊕ ⟨⟨VSIAME[N'[i, 2], 1], Q⟩⟩
return M

```

Figure 20: An algorithm for computing  $\mathcal{M}'(P, D)$  from  $\mathbf{N}'$  and  $\mathbf{V}_{\text{SIAME}}$ .

$$\begin{aligned}
& \langle \langle \langle 1, 2 \rangle, \quad \langle \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 0, 0 \rangle, \quad \langle \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 1, 1 \rangle, \quad \langle \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 0, 1 \rangle, \quad \langle \langle 1, 1 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 2, 2 \rangle, \quad \langle \langle 1, 1 \rangle, \langle 1, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 1, 3 \rangle, \quad \langle \langle 1, 1 \rangle, \langle 2, 1 \rangle \rangle \rangle, \\
& \langle \langle \langle 1, 0 \rangle, \quad \langle \langle 1, 1 \rangle, \langle 1, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 0, 2 \rangle, \quad \langle \langle 2, 1 \rangle, \langle 2, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 0, -1 \rangle, \quad \langle \langle 1, 2 \rangle, \langle 2, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle -1, 0 \rangle, \quad \langle \langle 2, 1 \rangle, \langle 2, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 2, 3 \rangle, \quad \langle \langle 1, 1 \rangle \rangle \rangle, \\
& \langle \langle \langle 2, 1 \rangle, \quad \langle \langle 1, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 1, -1 \rangle, \quad \langle \langle 1, 2 \rangle \rangle \rangle, \\
& \langle \langle \langle 0, 3 \rangle, \quad \langle \langle 2, 1 \rangle \rangle \rangle, \\
& \langle \langle \langle -1, 1 \rangle, \quad \langle \langle 2, 1 \rangle \rangle \rangle, \\
& \langle \langle \langle -1, -1 \rangle, \quad \langle \langle 2, 2 \rangle \rangle \rangle \rangle
\end{aligned}$$

Figure 21:  $\mathbf{M}$  for the pattern in Figure 3(a) and the dataset in Figure 3(b).

```

COSIATEC( $D$ )
1   $\mathbf{C} \leftarrow \langle \rangle$ 
2   $D' \leftarrow D$ 
3  while  $D' \neq \emptyset$ 
4       $\mathbf{T} \leftarrow \mathbf{T}'(D')$ 
5       $x \leftarrow |\mathbf{T}|$ 
6       $E_{\text{best}} \leftarrow \mathbf{T}[1]$ 
7       $CR_{\text{best}} \leftarrow CR(\mathbf{T}[1])$ 
8       $COV_{\text{best}} \leftarrow COV(\mathbf{T}[1])$ 
9      for  $i \leftarrow 2$  to  $x$ 
10         if  $\langle CR(\mathbf{T}[i]), COV(\mathbf{T}[i]) \rangle > \langle CR_{\text{best}}, COV_{\text{best}} \rangle$ 
11              $E_{\text{best}} \leftarrow \mathbf{T}[i]$ 
12              $CR_{\text{best}} \leftarrow CR(\mathbf{T}[i])$ 
13              $COV_{\text{best}} \leftarrow COV(\mathbf{T}[i])$ 
14      $\mathbf{C} \leftarrow \mathbf{C} \oplus \langle E_{\text{best}} \rangle$ 
15      $D' \leftarrow D' \setminus \bigcup_{P \in E_{\text{best}}} P$ 
16 return  $\mathbf{C}$ 

```

Figure 22: The COSIATEC algorithm.

global types

```
NUMBER_NODE
  number : a real number
  next   : a NUMBER_NODE pointer

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

COV_NODE
  datapoint : a VECTOR_NODE pointer
  next      : a COV_NODE pointer

TEC_NODE
  pattern       : a VECTOR_NODE pointer
  translator_set : a VECTOR_NODE pointer
  pattern_size  : an integer
  translator_set_size : an integer
  covered_set   : a COV_NODE pointer
  coverage     : an integer
  compression_ratio : a rational number

X_NODE
  size      : an integer
  vec_seq   : a VECTOR_NODE pointer
  start_vec : a VECTOR_NODE pointer
  down     : an X_NODE pointer
  right    : an X_NODE pointer
```

Figure 23: Globally defined data types used in the algorithms.



```
SIA(DFN, OFN : filenames; SD : string of 0s and 1s representing selected dimensions)
  local variables
1     DF : files
2     D, V : VECTOR_NODE pointers

3   if (DF ← OPEN_FILE(DFN,READ)) = NULL
4     EXIT
5   D ← READ_VECTOR_SET(DF,DOWN,SD)
6   CLOSE_FILE(DF)
7   D ← SORT_DATASET(D)
8   D ← SETIFY_DATASET(D)
9   V ← SIA_COMPUTE_VECTORS(D)
10  V ← SIA_SORT_VECTORS(V)
11  PRINT_VECTOR_MTP_PAIRS(V,OFN)
```

Figure 24: The SIA algorithm.

```

READ_VECTOR_SET(F : file; DIR : direction; SD : string indicating chosen dimensions)
    local variables
1      S, L: VECTOR_NODE pointers
2      v : NUMBER_NODE pointer

3      S ← L ← NULL
4      v ← NULL
5      while not AT_END_OF_LINE(F)
6          v ← READ_VECTOR(F)
7          if SD ≠ NULL
8              v ← SELECT_DIMENSIONS_IN_VECTOR(v,SD)
9          if S = NULL
10             S ← MAKE_NEW_VECTOR_NODE
11             S↑vector ← v
12             v ← NULL
13             L ← S
14         else if DIR = DOWN
15             L↑down ← MAKE_NEW_VECTOR_NODE
16             L ← L↑down
17             L↑vector ← v
18             v ← NULL
19         else
20             L↑right ← MAKE_NEW_VECTOR_NODE
21             L ← L↑right
22             L↑vector ← v
23             v ← NULL
24     return S

```

Figure 25: The READ\_VECTOR\_SET algorithm.

```

SORT_DATASET(D : VECTOR_NODE pointer)
  local variables
1     ABOVE_A, A, B, BELOW_B, C : VECTOR_NODE pointers

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

```

Figure 26: The SORT\_DATASET algorithm.

```

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

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

```

Figure 27: The MERGE\_DATASET\_ROWS algorithm.

```
SETIFY_DATASET(D : VECTOR_NODE pointer)
  local variables
1   d1,d2 : VECTOR_NODE pointers

2   d1 ← D
3   d2 ← NULL
4   while d1 ≠ NULL and d1↑right ≠ NULL
5     if VECTOR_EQUAL(d1↑right↑vector,d1↑vector)
6       d2 ← d1↑right
7       d1↑right ← d2↑right
8       d2↑right ← NULL
9       d2 ← DISPOSE_OF_VECTOR_NODE(d2)
10    else
11      d1 ← d1↑right
12  return D
```

Figure 28: The SETIFY\_DATASET algorithm.

```

SIA_COMPUTE_VECTORS(D : VECTOR_NODE pointer)
  local variables
1     d1, d2, p, v, V : VECTOR_NODE pointers

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

```

Figure 29: The SIA\_COMPUTE\_VECTORS algorithm.

```

SIA_SORT_VECTORS(V : VECTOR_NODE pointer)
  local variables
1   BEFORE_A, A, B, AFTER_B, C : VECTOR_NODE pointers

2   while V ≠ NULL and V↑right ≠ NULL
3     BEFORE_A ← NULL
4     A ← V
5     V ← NULL
6     repeat
7       if V ≠ NULL
8         BEFORE_A↑right ← NULL
9         B ← A↑right
10        A↑right ← NULL
11        AFTER_B ← B↑right
12        B↑right ← NULL
13        C ← SIA_MERGE_VECTOR_COLUMNS(A,B)
14        if B ≠ NULL
15          B↑down ← NULL
16          B ← DISPOSE_OF_VECTOR_NODE(B)
17        if V = NULL
18          V ← C
19        else
20          BEFORE_A↑right ← C
21          C↑right ← AFTER_B
22          BEFORE_A ← C
23          A ← BEFORE_A↑right
24        until A = NULL or A↑right = NULL
25  BEFORE_A ← A ← B ← AFTER_B ← C ← NULL
26  if V ≠ NULL
27    A ← V↑down
28    V↑down ← NULL
29    V ← DISPOSE_OF_VECTOR_NODE(V)
30    V ← A
31    A ← NULL
32  return V

```

Figure 30: The SIA\_SORT\_VECTORS algorithm.

```

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

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

```

Figure 31: The SIA\_MERGE\_VECTOR\_COLUMNS algorithm.



```

PRINT_VECTOR_MTP_PAIRS(V : VECTOR_NODE pointer; FN : filename)
    local variables
1     v1, v2: VECTOR_NODE pointers
2     F : file

3     if (F ← OPEN_FILE(FN,WRITE)) = NULL
4         EXIT
5     if V ≠ NULL
6         v1 ← V
7         while v1 ≠ NULL
8             PRINT_VECTOR(v1↑vector,F)
9             PRINT_NEW_LINE(F)
10            PRINT_VECTOR(v1↑right↑vector,F)
11            v2 ← v1↑down
12            while v2 ≠ NULL and VECTOR_EQUAL(v2↑vector,v1↑vector)
13                PRINT_VECTOR(v2↑right↑vector,F)
14                v2 ← v2↑down
15            PRINT_NEW_LINE(F)
16            v1 ← v2
17 PRINT_NEW_LINE(F)
18 CLOSE_FILE(F)

```

Figure 32: The PRINT\_VECTOR\_MTP\_PAIRS algorithm.

```

SIATEC(DFN, OFN : filenames; SD : string representing selected dimensions)
  local variables
1     OF, DF : files
2     D, V : VECTOR.NODE pointers
3     X : X.NODE pointer
4     n : integer

5   if (DF ← OPEN_FILE(DFN,READ)) = NULL
6     EXIT
7   D ← READ_VECTOR_SET(DF,DOWN,SD)
8   CLOSE_FILE(DF)
9   if D = NULL
10    OF ← OPEN_FILE(OFN,WRITE)
11    CLOSE_FILE(OFN)
12  else
13    D ← SORT_DATASET(D)
14    D ← SETIFY_DATASET(D)
15    n ← SIZE_OF_DATASET(D)
16    if D↑right = NULL
17      OF ← OPEN_FILE(OFN,WRITE)
18      PRINT_VECTOR(D↑vector,OF)
19      PRINT_NEW_LINE(OF)
20      PRINT_NEW_LINE(OF)
21      PRINT_NEW_LINE(OF)
22      CLOSE_FILE(OF)
23    else
24      V ← COMPUTE_VECTORS(D)
25      V ← CONSTRUCT_VECTOR_TABLE(V)
26      V ← SORT_VECTORS(V)
27      X ← VECTORIZE_PATTERNS(V)
28      X ← SORT_PATTERN_VECTOR_SEQUENCES(X)
29      PRINT_TECS(X,OFN,n)

```

Figure 33: The SIATEC algorithm.

```

COMPUTE_VECTORS(D : VECTOR_NODE pointer)
  local variables
1   d1, d2, p, v, V: VECTOR_NODE pointers

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

```

Figure 34: The COMPUTE\_VECTORS algorithm.

```
CONSTRUCT_VECTOR_TABLE(V : VECTOR_NODE pointer)
  local variables
1     p, v, w : VECTOR_NODE pointers

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

Figure 35: The CONSTRUCT\_VECTOR\_TABLE algorithm.

```

SORT_VECTORS(V : VECTOR_NODE pointer)
  local variables
1   BEFORE_A, A, B, AFTER_B, C : VECTOR_NODE pointers

2   while V ≠ NULL and V↑right ≠ NULL
3     BEFORE_A ← NULL
4     A ← V
5     V ← NULL
6     repeat
7       if V ≠ NULL
8         BEFORE_A↑right ← NULL
9         B ← A↑right
10        A↑right ← NULL
11        AFTER_B ← B↑right
12        B↑right ← NULL
13        C ← MERGE_VECTOR_COLUMNS(A,B)
14        if B ≠ NULL
15          B↑down ← NULL
16          B ← DISPOSE_OF_VECTOR_NODE(B)
17        if V = NULL
18          V ← C
19        else
20          BEFORE_A↑right ← C
21          C↑right ← AFTER_B
22          BEFORE_A ← C
23          A ← BEFORE_A↑right
24        until A = NULL or A↑right = NULL
25  BEFORE_A ← A ← B ← AFTER_B ← C ← NULL
26  if V ≠ NULL
27    A ← V↑down
28    V↑down ← NULL
29    V ← DISPOSE_OF_VECTOR_NODE(V)
30    V ← A
31    A ← NULL
32  return V

```

Figure 36: The SORT\_VECTORS algorithm.

```

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

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

```

Figure 37: The MERGE\_VECTOR\_COLUMNS algorithm.

```

VECTORIZE_PATTERNS(V : VECTOR_NODE pointer)
  local variables
1     i, j, above_j, Q, q : VECTOR_NODE pointers
2     x, X : X_NODE pointers
3     size : integer

4     i ← j ← above_j ← Q ← q ← NULL
5     x ← X ← NULL
6     i ← V
7     while i ≠ NULL
8         size ← 1
9         j ← i↑down
10        above_j ← i
11        while j ≠ NULL and VECTOR_EQUAL(i↑right↑vector, j↑right↑vector)
12            if Q = NULL
13                Q ← MAKE_NEW_VECTOR_NODE
14                q ← Q
15            else
16                q↑down ← MAKE_NEW_VECTOR_NODE
17                q ← q↑down
18                size ← size + 1
19                q↑vector ← VECTOR_MINUS(j↑right↑right↑vector, above_j↑right↑right↑vector)
20                j ← j↑down
21                above_j ← above_j↑down
22        if X = NULL
23            X ← MAKE_NEW_X_NODE
24            x ← X
25        else
26            x↑down ← MAKE_NEW_X_NODE
27            x ← x↑down
28        x↑size ← size
29        x↑vec_seq ← Q
30        x↑start_vec ← i
31        Q ← q ← NULL
32        i ← j
33    i ← j ← above_j ← Q ← q ← NULL
34    x ← NULL
35    return X

```

Figure 38: The VECTORIZE\_PATTERNS algorithm.

```

SORT_PATTERN_VECTOR_SEQUENCES(X : X_NODE pointer)
  local variables
1   ABOVE_A, A, B, BELOW_B, C : X_NODE pointers

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

```

Figure 39: The SORT\_PATTERN\_VECTOR\_SEQUENCES algorithm.



```

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

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

```

Figure 40: The MERGE\_PATTERN\_ROWS algorithm.

```

PRINT_TECS(X : X_NODE pointer; FN : filename; n : integer)
  local variables
1     i, before_i : X_NODE pointer
2     Iptr, j, I : VECTOR_NODE pointer
3     F : file

4     F = OPEN_FILE(FN,WRITE)
5     i ← before_i ← NULL
6     I ← Iptr ← j ← NULL
7     i ← X
8     if X ≠ NULL
9       repeat
10        j ← i↑start_vec
11        if I ≠ NULL
12          Iptr ← I
13          while Iptr ≠ NULL
14            Iptr↑right ← NULL
15            Iptr ← Iptr↑down
16            I ← DISPOSE_OF_VECTOR_NODE(I)
17        while j ≠ NULL and VECTOR_EQUAL(j↑right↑vector,i↑start_vec↑right↑vector)
18          if I = NULL
19            I ← MAKE_NEW_VECTOR_NODE
20            Iptr ← I
21          else
22            Iptr↑down ← MAKE_NEW_VECTOR_NODE
23            Iptr ← Iptr↑down
24            Iptr↑right ← j↑right↑right
25            j ← j↑down
26        PRINT_PATTERN(I,F)
27        PRINT_SET_OF_TRANSLATORS(I,F,n,i↑size)
28        before_i ← i
29        i ← i↑right
30        while i ≠ NULL and PATTERN_VEC_SEQ_EQUAL(i,before_i)
31          i ← i↑right
32          before_i ← before_i↑right
33      until i = NULL
34      if I ≠ NULL
35        Iptr ← I
36        while Iptr ≠ NULL
37          Iptr↑right ← NULL
38          Iptr ← Iptr↑down
39        I ← DISPOSE_OF_VECTOR_NODE(I)
40      j ← NULL
41      i ← before_i ← NULL
42  PRINT_NEW_LINE(F)
43  CLOSE_FILE(F)

```

Figure 41: The PRINT\_TECS algorithm.

```
PRINT_PATTERN(I : VECTOR_NODE pointer; F : file)
    local variables
1     p : VECTOR_NODE pointer

2     p ← I
3     while p ≠ NULL
4         PRINT_VECTOR(p↑right↑vector,F)
5         p ← p↑down
6     PRINT_NEW_LINE(F)
```

Figure 42: The PRINT\_PATTERN algorithm.

```

PRINT_SET_OF_TRANSLATORS(I : VECTOR_NODE pointer; F : file; n, p : integers)
  local variables
1   v, J, j2, j1, i : VECTOR_NODE pointers
2   FINISHED : Boolean value
3   k : integer

4   v ← J ← j2 ← j1 ← i ← NULL
5   if I↑down = NULL
6     v ← I↑right↑down
7     while v ≠ NULL
8       if not IS_ZERO_VECTOR(v↑vector)
9         PRINT_VECTOR(v↑vector,F)
10      v ← v↑down
11    PRINT_NEWLINE(F)
12  else
13    J ← NULL
14    i ← I
15    while i ≠ NULL
16      if J = NULL
17        J ← MAKE_NEW_VECTOR_NODE
18        j2 ← J
19      else
20        j2↑down ← MAKE_NEW_VECTOR_NODE
21        j2 ← j2↑down
22        j2↑right ← i↑right↑down
23        j2↑vector ← MAKE_NEW_NUMBER_NODE
24        j2↑vector↑number ← 1
25        i ← i↑down
26    if J↑right ≠ NULL and IS_ZERO_VECTOR(J↑right↑vector)
27      J↑right ← J↑right↑down
28      J↑vector↑number ← J↑vector↑number + 1
29    FINISHED ← FALSE
30    j2 ← J↑down
31    k ← 2
32    j1 ← J
33    while not FINISHED
34      while j2↑right ≠ NULL and j2↑vector↑number ≤ j1↑vector↑number
35        j2↑right ← j2↑right↑down
36        j2↑vector↑number ← j2↑vector↑number + 1
37      while j2↑vector↑number ≤ n - p + k and VECTOR_LESS_THAN(j2↑right↑vector, j1↑right↑vector)
38        j2↑right ← j2↑right↑down
39        j2↑vector↑number ← j2↑vector↑number + 1
40      if j2↑vector↑number > n - p + k then FINISHED ← TRUE
41      else if VECTOR_LESS_THAN(j1↑right↑vector, j2↑right↑vector)
42        j1 ← J
43        j2 ← J↑down
44        k ← 2
45        J↑right ← J↑right↑down
46        J↑vector↑number ← J↑vector↑number + 1
47        if J↑right ≠ NULL and IS_ZERO_VECTOR(J↑right↑vector)
48          J↑right ← J↑right↑down
49          J↑vector↑number ← J↑vector↑number + 1
50        if J↑vector↑number > n - p + 1 then FINISHED ← TRUE
51      else if k = p
52        PRINT_VECTOR(j2↑right↑vector,F)
53        j2 ← J
54        k ← 1
55        while k ≤ p
56          j2↑right ← j2↑right↑down
57          j2↑vector↑number ← j2↑vector↑number + 1
58          if j2↑right ≠ NULL and IS_ZERO_VECTOR(j2↑right↑vector)
59            j2↑right ← j2↑right↑down
60            j2↑vector↑number ← j2↑vector↑number + 1
61          if j2↑vector↑number > n - p + k
62            FINISHED ← TRUE
63          k ← p
64        j1 ← j2
65        j2 ← j2↑down
66        k ← k + 1
67      j1 ← J
68      j2 ← J↑down
69      k ← 2
70    else
71      j1 ← j2
72      j2 ← j2↑down
73      k ← k + 1
74    PRINT_NEWLINE(F)
75    j2 ← NULL
76    j1 ← J
77    while j1 ≠ NULL
78      j1↑right ← NULL
79      j1 ← j1↑down
80    J ← DISPOSE_OF_VECTOR_NODE(J)

```

Figure 43: The PRINT\_SET\_OF\_TRANSLATORS algorithm.

```

COSIATEC(DFN, OFN : filenames; SD : string indicating chosen dimensions)
  local variables
1   TFN : filename
2   OF, TF, DF : files
3   D, V : VECTOR_NODE pointers
4   BT, T : TEC_NODE pointers
5   X : an X_NODE pointer
6   n : integer

7   TFN ← "TEMP_TEC_FILE"
8   if (DF ← OPEN_FILE(DFN,READ)) = NULL
9     EXIT
10  D ← READ_VECTOR_SET(DFN,DOWN,SD)
11  CLOSE_FILE(DF)
12  D ← SORT_DATASET(D)
13  D ← SETIFY_DATASET(D)
14  n ← SIZE_OF_DATASET(D)
15  OF ← OPEN_FILE(OFN,WRITE)
16  BT ← NULL
17  T ← NULL
18  while D ≠ NULL
19    if D↑right = NULL
20      TF ← OPEN_FILE(TFN,WRITE)
21      PRINT_VECTOR(D↑vector,TF)
22      PRINT_NEW_LINE(TF)
23      PRINT_NEW_LINE(TF)
24      PRINT_NEW_LINE(TF)
25      CLOSE_FILE(TF)
26    else
27      V ← COMPUTE_VECTORS(D)
28      V ← CONSTRUCT_VECTOR_TABLE(V)
29      V ← SORT_VECTORS(V)
30      X ← VECTORIZE_PATTERNS(V)
31      X ← SORT_PATTERN_VECTOR_SEQUENCES(X)
32      PRINT_TEC(X,TFN,n)
33      DISPOSE_OF_SIATEC_DATA_STRUCTURES(D,V,X)
34      TF ← OPEN_FILE(TFN,READ)
35      while not AT_END_OF_LINE(TF)
36        T ← READ_TEC(TF,D)
37        if IS_BETTER_TEC(T,BT)
38          BT ← DISPOSE_OF_TEC(BT)
39          BT ← T
40          T ← NULL
41        else
42          T ← DISPOSE_OF_TEC(T)
43      CLOSE_FILE(TF)
44      DELETE_FILE(TFN)
45      PRINT_TEC(BT,OF)
46      D ← DELETE_TEC_COVERED_SET(D,BT)
47      n ← n - BT↑coverage
48      BT ← DISPOSE_OF_TEC(BT)
49  PRINT_NEW_LINE(OF)
50  CLOSE_FILE(OF)

```

Figure 44: The COSIATEC algorithm.

```

DISPOSE_OF_SIATEC_DATA_STRUCTURES(D, V : VECTOR_NODE pointers; X : X_NODE pointer)
  local variables
1   p1, p2 : VECTOR_NODE pointers
2   x : X_NODE pointer

3   p1 ← D
4   while p1 ≠ NULL
5     p2 ← p1↑down
6     while p2 ≠ NULL
7       p2↑right ← NULL
8       p2 ← p2↑down
9     p1 ← p1↑right
10  p1 ← V
11  while p1 ≠ NULL
12    p1↑right ← NULL
13    p1 ← p1↑down
14  x ← X
15  while x ≠ NULL
16    x↑start_vec ← NULL
17    while x↑vec_seq ≠ NULL
18      p1 ← x↑vec_seq↑down
19      x↑vec_seq↑down ← NULL
20      FREE(x↑vec_seq)
21      x↑vec_seq ← p1
22    x ← x↑right
23  p1 ← D
24  while p1 ≠ NULL
25    p1↑down ← DISPOSE_OF_VECTOR_NODE(p1↑down)
26    p1 ← p1↑right
27  while V ≠ NULL
28    p1 ← V↑down
29    V↑down ← NULL
30    FREE(V)
31    V ← p1
32  while X ≠ NULL
33    x ← X↑right
34    X↑right ← NULL
35    FREE(X)
36    X ← x

```

Figure 45: The DISPOSE\_OF\_SIATEC\_DATA\_STRUCTURES algorithm.

```
READ_TEC(F : FILE; D : VECTOR_NODE pointer)
    local variables
1      T : TEC_NODE pointer

2      T ← MAKE_NEW_TEC_NODE
3      T↑pattern ← READ_VECTOR_SET(F,DOWN,NULL)
4      T↑translator_set ← READ_VECTOR_SET(F,DOWN,NULL)
5      SET_TEC_PATTERN_SIZE(T)
6      SET_TEC_TRANSLATOR_SET_SIZE(T)
7      SET_TEC_COVERED_SET(T,D)
8      SET_TEC_COVERAGE(T)
9      SET_TEC_COMPRESSION_RATIO(T)
10     return T
```

Figure 46: The READ\_TEC algorithm.

```

SET_TEC_COVERED_SET(T : TEC_NODE pointer; D : VECTOR_NODE pointer)
  local variables
1   p, d, t : VECTOR_NODE pointers
2   s : NUMBER_NODE pointers
3   c : COV_NODE pointers

4   if T ≠ NULL and D ≠ NULL
5     p ← T↑pattern
6     d ← D
7     while p ≠ NULL
8       while d ≠ NULL and not VECTOR_EQUAL(d↑vector, p↑vector)
9         d ← d↑right
10      if d = NULL
11        EXIT
12      d↓down ← MAKE_NEW_VECTOR_NODE
13      p ← p↓down
14    p ← T↑pattern
15    while p ≠ NULL
16      t ← T↑translator_set
17      d ← D
18      while t ≠ NULL
19        s ← VECTOR_PLUS(p↑vector, t↑vector)
20        while d ≠ NULL and not VECTOR_EQUAL(d↑vector, s)
21          d ← d↑right
22        if d = NULL
23          EXIT
24        d↓down ← MAKE_NEW_VECTOR_NODE
25        s ← DISPOSE_OF_NUMBER_NODE(s)
26        t ← t↓down
27      p ← p↓down
28    d ← D
29    c ← NULL
30    while d ≠ NULL
31      if d↓down ≠ NULL
32        if c = NULL
33          T↑covered_set ← MAKE_NEW_COV_NODE
34          c ← T↑covered_set
35        else
36          c↑next ← MAKE_NEW_COV_NODE
37          c ← c↑next
38          c↑datapoint ← d
39          d↓down ← DISPOSE_OF_VECTOR_NODE(d↓down)
40      d ← d↑right
41    c ← NULL

```

Figure 47: The SET\_TEC\_COVERED\_SET algorithm.



```
IS_BETTER_TEC(T1, T2: TEC_NODE pointers)
1   if T1 = NULL
2       PRINT_ERROR_MESSAGE('IS_BETTER_TEC: T1 is NULL.')
```

```
3       EXIT
4   if T2 = NULL
5       return TRUE
6   if T1↑compression_ratio > T2↑compression_ratio
7       return TRUE
8   if T1↑compression_ratio < T2↑compression_ratio
9       return FALSE
10  if T1↑coverage > T2↑coverage
11      return TRUE
12  return FALSE
```

Figure 48: The IS\_BETTER\_TEC algorithm.

```
PRINT_TEC(T : TEC_NODE pointer; F : file)
1   if T ≠ NULL
2       PRINT_VECTOR_SET(T↑pattern,DOWN,F)
3       PRINT_VECTOR_SET(T↑translator_set,DOWN,F)
```

Figure 49: The PRINT\_TEC algorithm.

```
PRINT_VECTOR_SET(V : VECTOR_NODE pointer; DIRECTION : either DOWN or RIGHT; F : file)
    local variables
1     v : VECTOR_NODE pointer

2     v ← V
3     while v ≠ NULL
4         PRINT_VECTOR(v↑vector,F)
5         if DIRECTION = DOWN
6             v ← v↑down
7         else
8             v ← v↑right
9     PRINT_NEW_LINE(F)
```

Figure 50: The PRINT\_VECTOR\_SET algorithm.

```

DELETE_TEC_COVERED_SET(D : VECTOR_NODE pointer; T : TEC_NODE pointer)
  local variables
1   c : COV_NODE pointer
2   d : VECTOR_NODE pointer

3   if T ≠ NULL
4     c ← T↑covered_set
5     while c ≠ NULL and D = c↑datapoint
6       D ← D↑right
7       c↑datapoint↑right ← NULL
8       c↑datapoint ← DISPOSE_OF_VECTOR_NODE(c↑datapoint)
9       c ← c↑next
10    d ← D
11    while c ≠ NULL
12      while d↑right ≠ c↑datapoint
13        d ← d↑right
14      d↑right ← c↑datapoint↑right
15      c↑datapoint↑right ← NULL
16      c↑datapoint ← DISPOSE_OF_VECTOR_NODE(c↑datapoint)
17      c ← c↑next
18  return D

```

Figure 51: The DELETE\_TEC\_COVERED\_SET algorithm.

```
1 1 1 』  
  └─┘ └─┘ └─┘  
1 3 2 』  
  └─┘ └─┘ └─┘  
2 1 2 』  
  └─┘ └─┘ └─┘  
2 2 2 』  
  └─┘ └─┘ └─┘  
2 3 3 』  
  └─┘ └─┘ └─┘  
3 2 2 』  
  └─┘ └─┘ └─┘  
』
```

Figure 52: Example of format used as input to `READ_VECTOR_SET` algorithm.

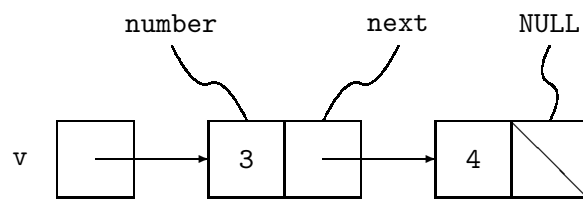
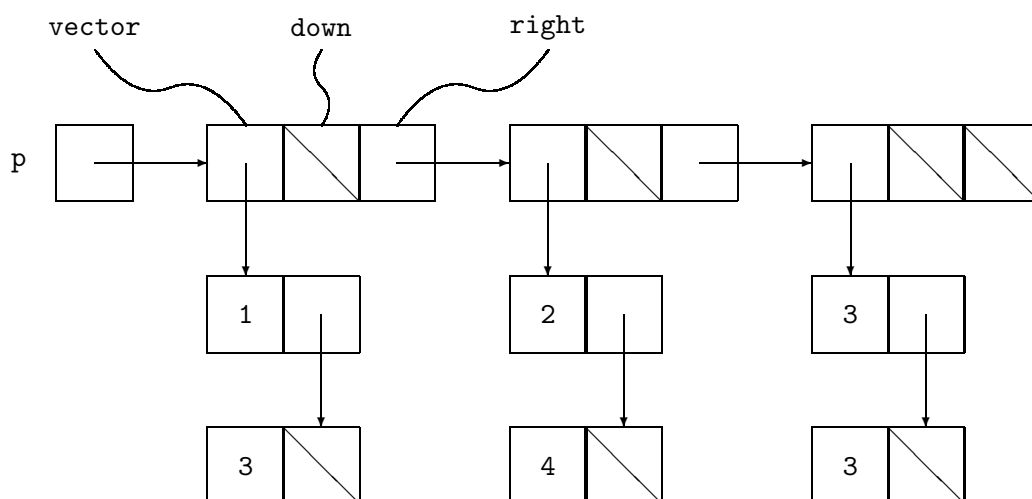


Figure 53: Using NUMBER\_NODES to represent vectors.

Figure 54: A right-directed list of `VECTOR_NODES`.

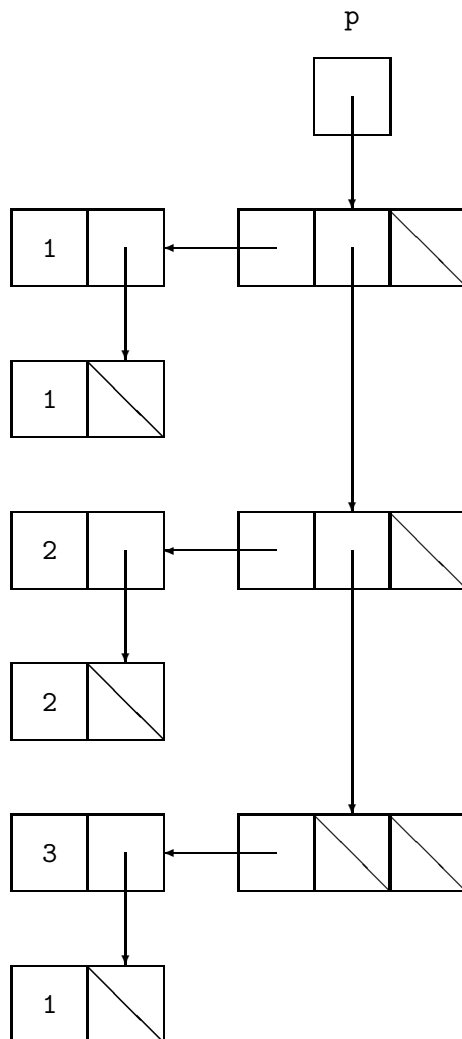


Figure 55: A down-directed list of VECTOR\_NODES.



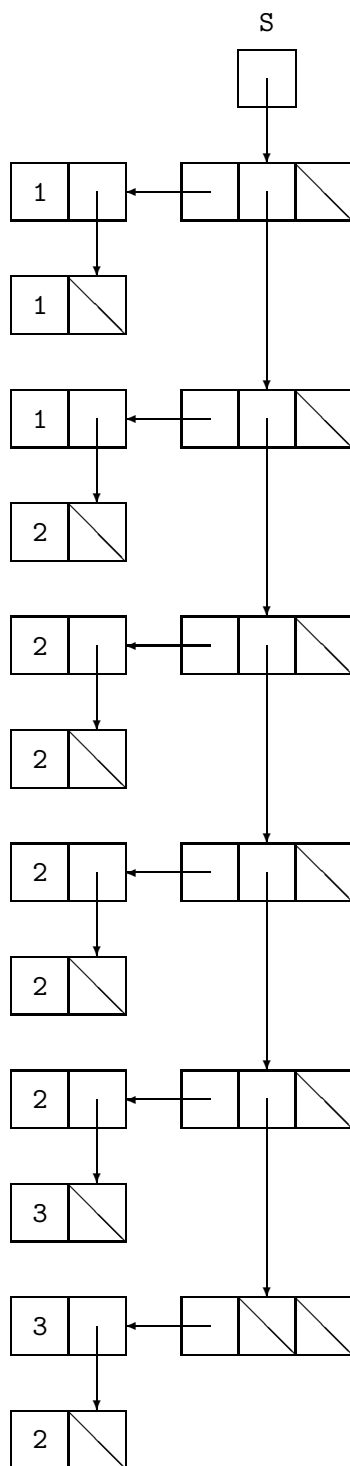


Figure 56: The linked list constructed by `READ_VECTOR_SET` when `F` is the data in Figure 52, `DIR = DOWN` and `SD = "101"`.

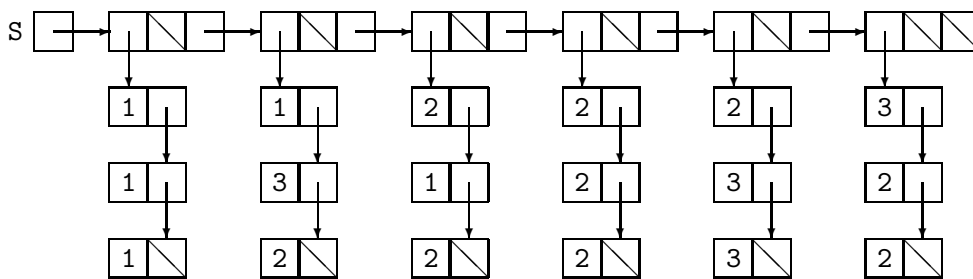


Figure 57: The linked list constructed by `READ_VECTOR_SET` when `F` is the data in Figure 52, `DIR = RIGHT` and `SD = NULL`.

3 3 ┘  
┘ ┘  
3 3 ┘  
┘ ┘  
3 1 ┘  
┘ ┘  
1 1 ┘  
┘ ┘  
1 1 ┘  
┘ ┘  
1 3 ┘  
┘ ┘  
┘

Figure 58: Example input data.

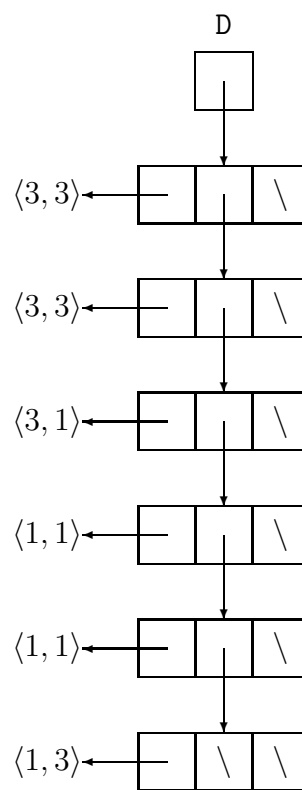


Figure 59: The linked list generated by line 5 of SIA (Figure 24) for the data in Figure 58.

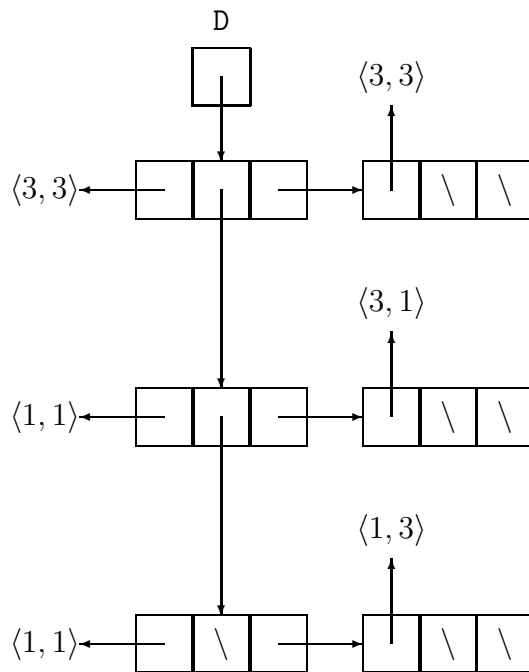


Figure 60: The state of the linked list  $D$  after one iteration of the outer `while` loop of `SORT_DATASET` on the dataset list in Figure 59.

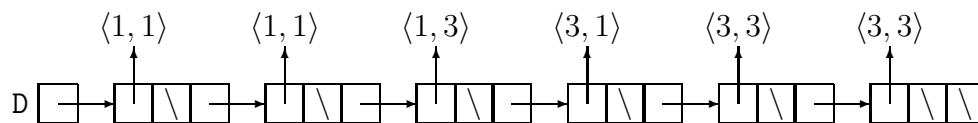


Figure 61: The sorted, right-directed linked list produced by `SORT_DATASET` from the unsorted, down-directed dataset list in Figure 59.

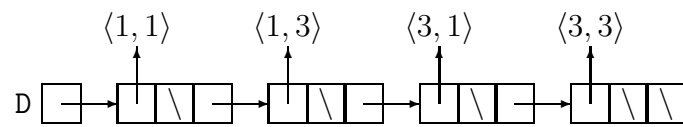


Figure 62: The linked list that results when `SETIFY_DATASET` has been executed on the linked list in Figure 61.

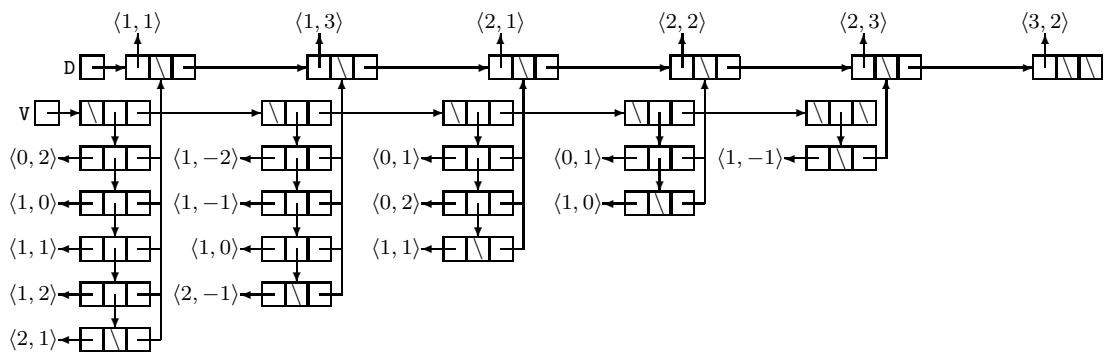


Figure 63: The data structure that results after `SIA_COMPUTE_VECTORS` has executed when the `SIA` algorithm in Figure 24 is carried out on the dataset shown in Figure 1(a).



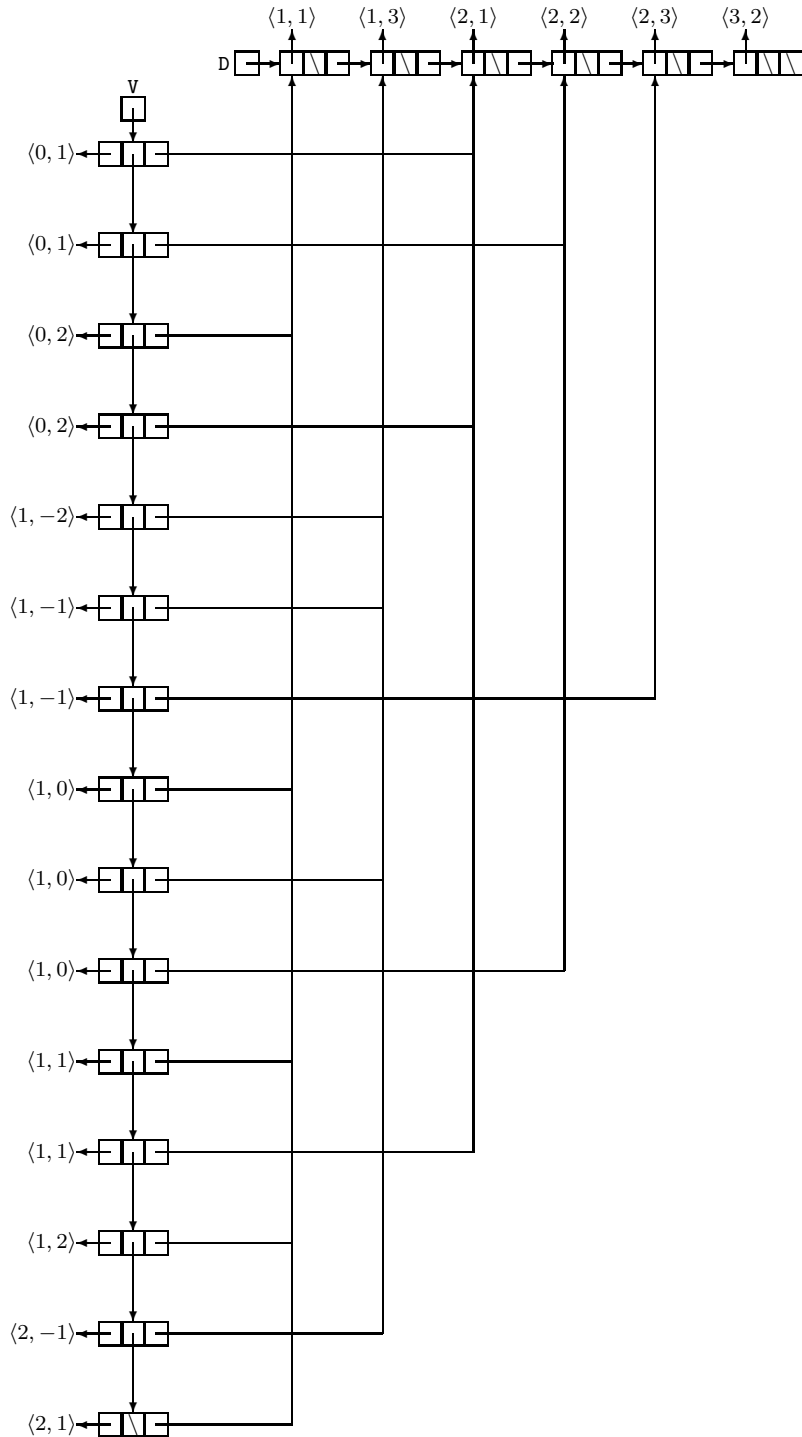


Figure 64: The data structure headed by `V` after `SIA_SORT_VECTORS` has executed when `SIA` is carried out on the dataset in Figure 1(a).

```

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 65: The output generated by `PRINT_VECTOR_MTP_PAIRS` (Figure 32) for the dataset in Figure 1(a).

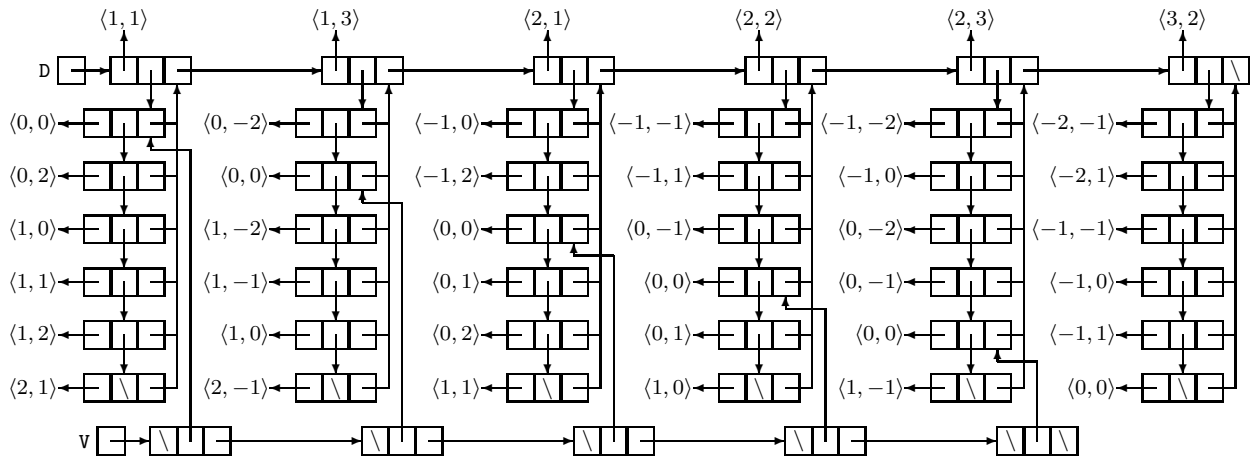


Figure 66: The data structure generated by `COMPUTE_VECTORS` for the dataset in Figure 1(a).

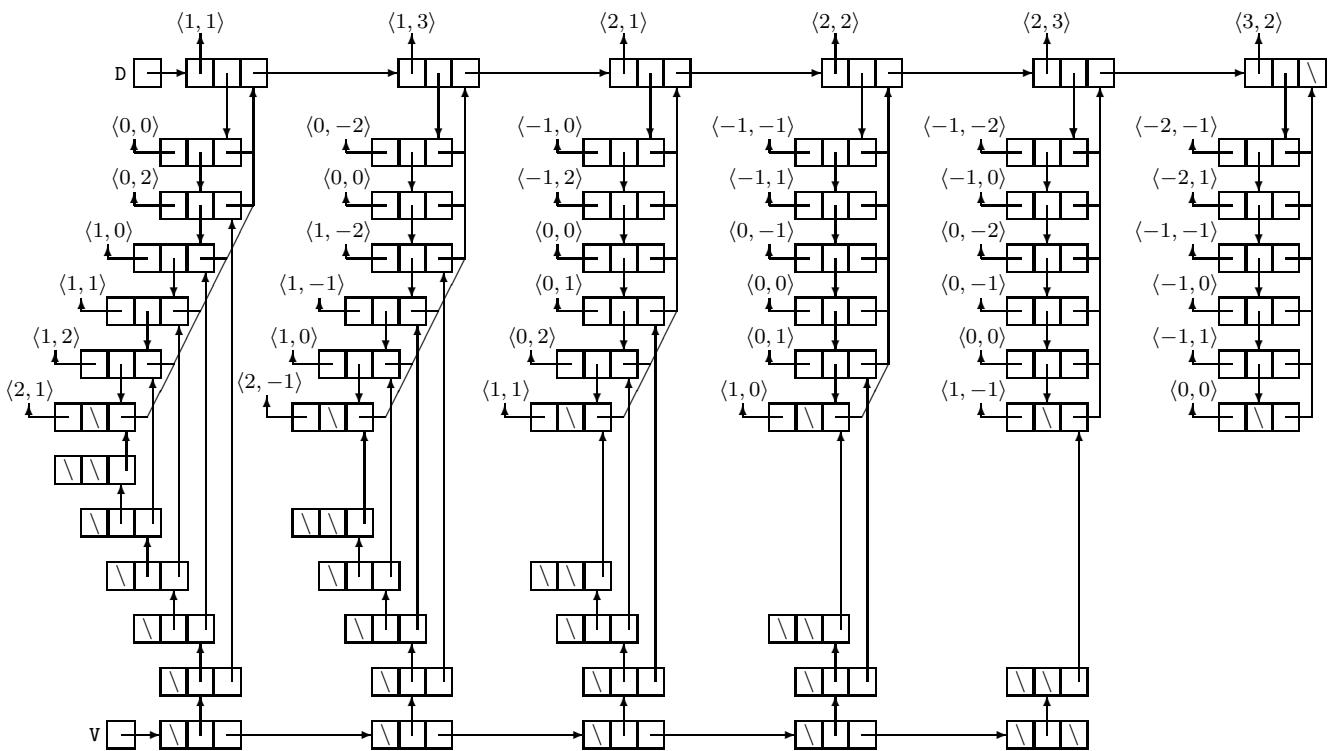


Figure 67: The data structures that result after `CONSTRUCT_VECTOR_TABLE` has executed when the SIATEC implementation in Figure 33 is run on the dataset in Figure 1(a).

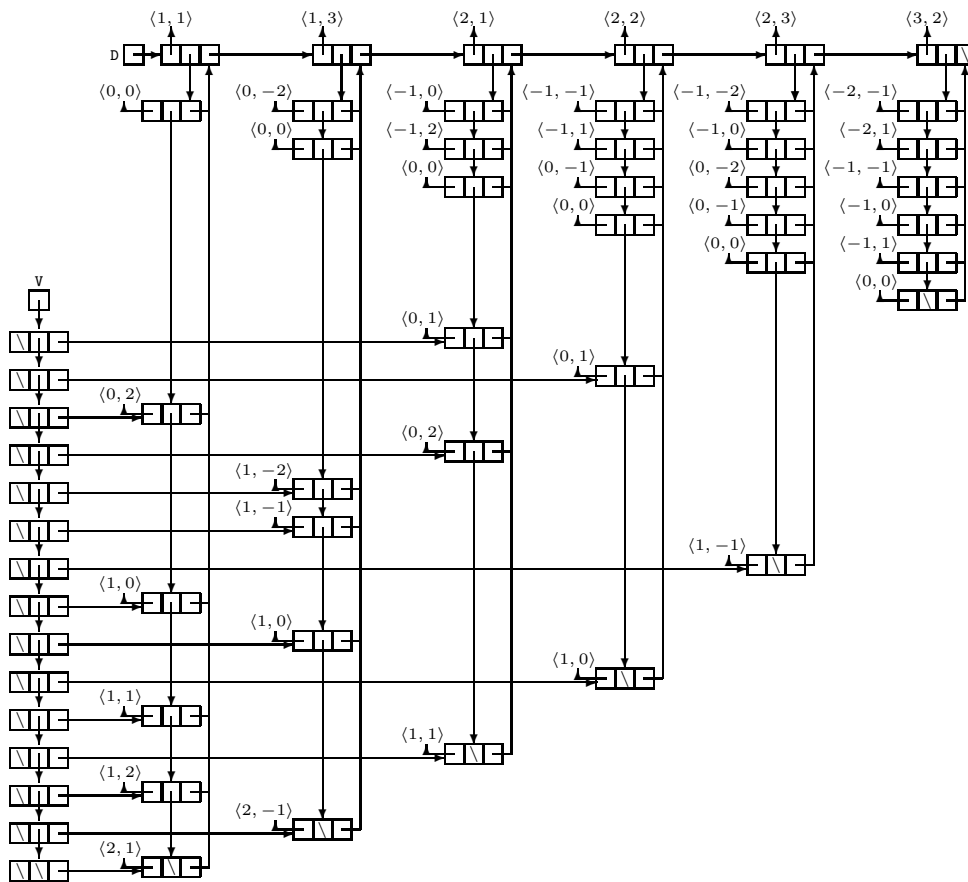


Figure 68: The data structures that result after SORT\_VECTORS has executed when the SIATEC implementation in Figure 33 is run on the dataset in Figure 1(a).

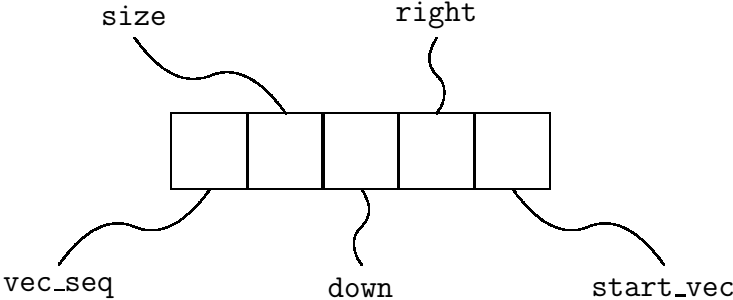


Figure 69: Diagrammatic representation of an X\_NODE.

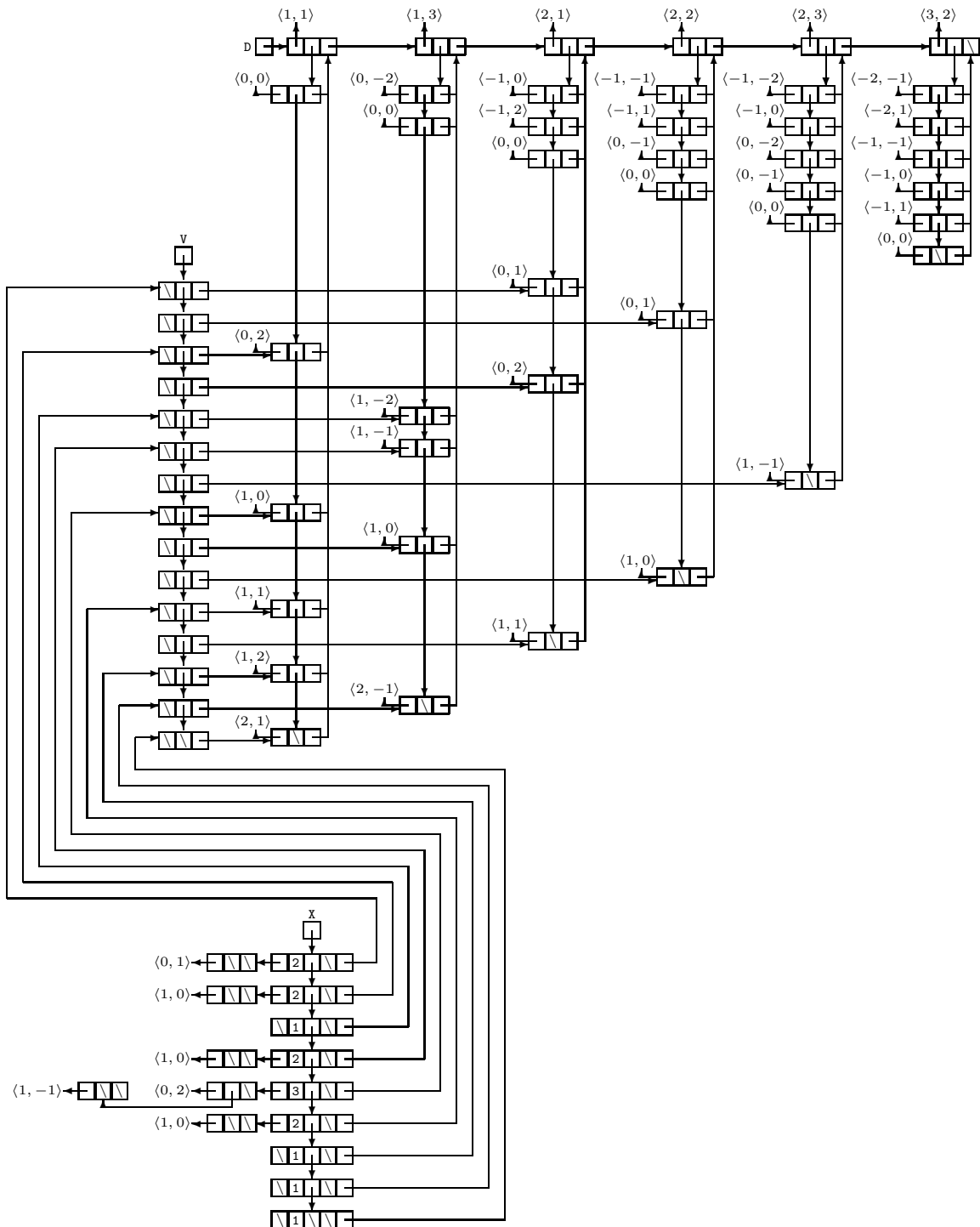


Figure 70: The state of the data structures headed by D, V and X in the SIATEC implementation in Figure 33 after line 27 has been executed when this implementation is run on the dataset in Figure 1(a).

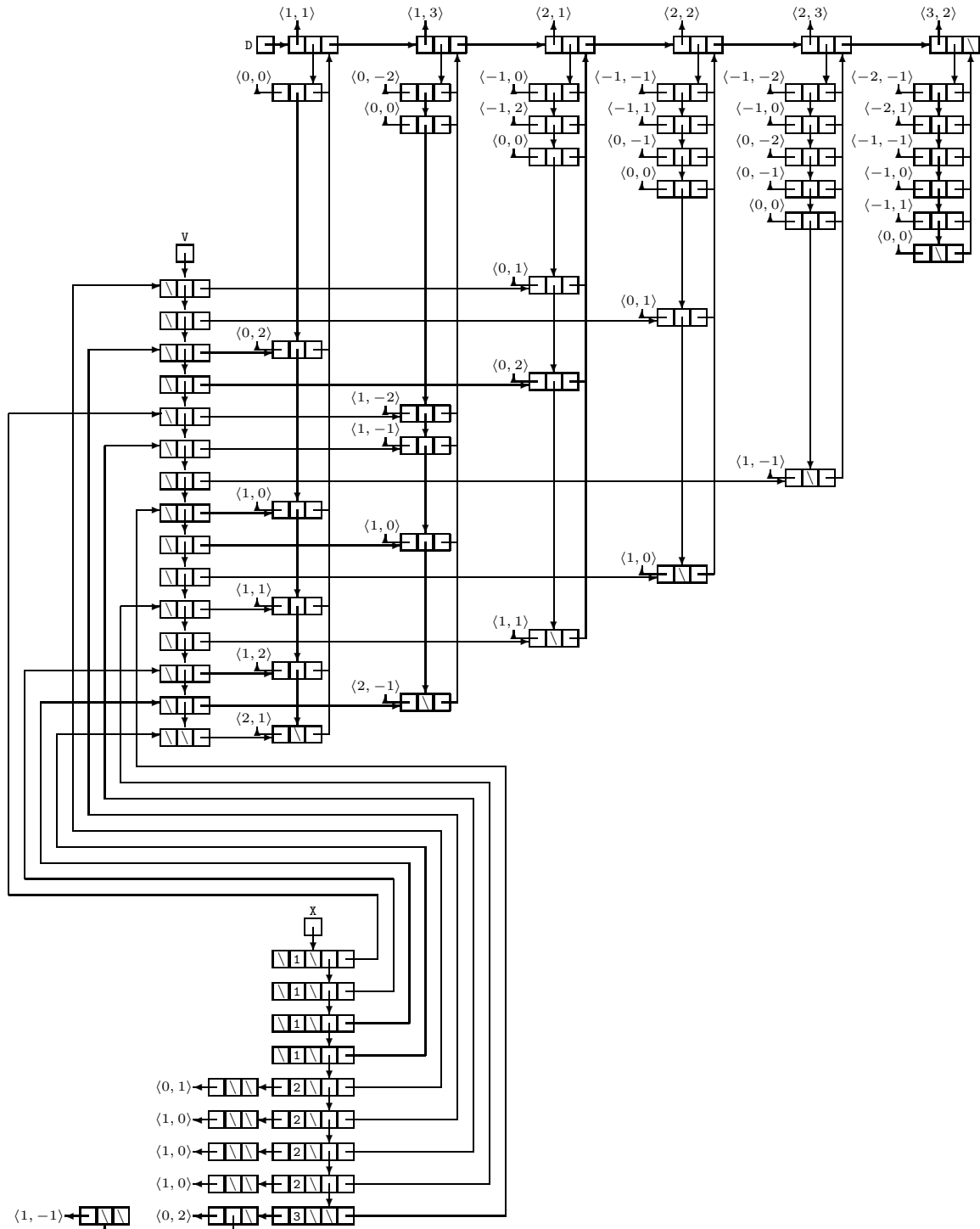


Figure 71: The state of the data structures headed by D, V and X in the SIATEC implementation in Figure 33 after line 28 has been executed when this implementation is run on the dataset in Figure 1(a).



```

1 3
└─┘
0 -2
1 -2
1 -1
1 0
2 -1
└─┘
2 1
2 2
└─┘
0 1
└─┘
1 1
2 1
└─┘
0 2
1 1
└─┘
1 1
1 3
2 2
└─┘
1 0
└─┘
└─┘

```

Figure 72: The output generated by PRINT\_TECS (Figure 41) for the dataset in Figure 1(a).

```
1 1 』
1 3 』
2 2 』
』
1 0 』
2 0 』
3 0 』
』
』
```

Figure 73: The output generated by COSIATEC (Figure 44) for the dataset in Figure 4.

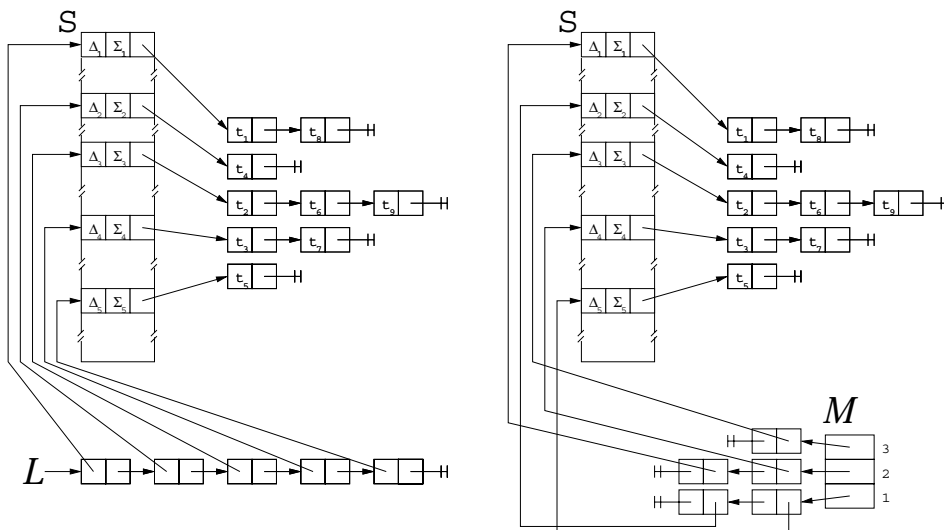


Figure 74: An illustration of the data structures used in SIAME.

**Function** NEWLINK(*data*, *next*)

1.  $p \leftarrow \mathbf{new}(next)$ ;
2.  $p \uparrow \text{next} \leftarrow next$ ;
3.  $p \uparrow \text{data} \leftarrow data$ ;
4. **return**  $p$ ;

Figure 75: The NEWLINK algorithm.

**Algorithm SIAME**( $T, D, S$ )

1.  $p \leftarrow \mathbf{nil}; \mathcal{L} \leftarrow \mathbf{nil};$
2. **for each**  $t \in T$  **do**
3.   **for each**  $d \in D$  **do**
4.      $\bar{v} \leftarrow d - t;$
5.      $p \leftarrow S[\mathbf{F}(\bar{v})];$
6.      $p \uparrow \text{ptr} \leftarrow \text{NEWLINK}(t, p \uparrow \text{ptr});$
7.      $p \uparrow \Delta \leftarrow \bar{v};$
8.      $p \uparrow \Sigma \leftarrow p \uparrow \Sigma + 1;$
9.     **if**  $p \uparrow \Sigma = 1$  **then**  $\mathcal{L} \leftarrow \text{NEWLINK}(p, \mathcal{L});$
10.  $p \leftarrow \mathcal{L};$
11. **while**  $p \neq \mathbf{nil}$  **do**
12.    $tmp \leftarrow p \uparrow \text{data} \uparrow \Sigma;$
13.    $\mathcal{M}[tmp] \leftarrow \text{NEWLINK}(p \uparrow \text{data}, \mathcal{M}[tmp]);$
14.    $p \leftarrow p \uparrow \text{next};$
15. **return**  $\mathcal{M};$

Figure 76: First implementation of SIAME algorithm.

**Algorithm** SIAME<sub>2</sub>( $T, D, S$ )

1.  $p \leftarrow \mathbf{nil}; \mathcal{L} \leftarrow \mathbf{nil};$
2.  $i \leftarrow 0;$
3. **for each**  $t \in T$  **do**
4.   **for each**  $d \in D$  **do**
5.      $S[i].\Delta \leftarrow d - t;$
6.      $S[i].\text{ptr} \leftarrow \text{NEWLINK}(t, S[i]. \text{ptr});$
7.      $i \leftarrow i + 1;$
8.  $S \leftarrow \text{MERGESORT}(S);$
9.  $\mathcal{M} \leftarrow \text{MERGEDUPLICATES}(S);$
10. **return**  $\mathcal{M};$

Figure 77: Second implementation of SIAME.

**Function** MERGEDUPLICATES( $S$ )

1.  $j \leftarrow 0$ ;
2. **while**  $j < (m \times n)$  **do**
3.    $k \leftarrow 1$ ;
4.   **while**  $S[j].\Delta = S[j+k].\Delta$  **do**
5.      $S[j+k].\text{ptr} \uparrow \text{next} \leftarrow S[j].\text{ptr}$ ;
6.      $S[j].\text{ptr} \leftarrow S[j+k].\text{ptr}$ ;
7.      $k \leftarrow k + 1$ ;
8.    $S[j].\Sigma \leftarrow k$ ;
9.    $\mathcal{M}[k] \leftarrow \text{NEWLINK}(S[j], \mathcal{M}[k])$ ;
10.  $j \leftarrow j + k$ ;
11. **return**  $\mathcal{M}$ ;

Figure 78: The MERGEDUPLICATES algorithm.

		From					
		$\langle 1, 1 \rangle$	$\langle 1, 3 \rangle$	$\langle 2, 1 \rangle$	$\langle 2, 2 \rangle$	$\langle 2, 3 \rangle$	$\langle 3, 2 \rangle$
	$\langle 1, 1 \rangle$						
	$\langle 1, 3 \rangle$	$\langle \langle 0, 2 \rangle, 1 \rangle$					
	$\langle 2, 1 \rangle$	$\langle \langle 1, 0 \rangle, 1 \rangle$	$\langle \langle 1, -2 \rangle, 2 \rangle$				
To	$\langle 2, 2 \rangle$	$\langle \langle 1, 1 \rangle, 1 \rangle$	$\langle \langle 1, -1 \rangle, 2 \rangle$	$\langle \langle 0, 1 \rangle, 3 \rangle$			
	$\langle 2, 3 \rangle$	$\langle \langle 1, 2 \rangle, 1 \rangle$	$\langle \langle 1, 0 \rangle, 2 \rangle$	$\langle \langle 0, 2 \rangle, 3 \rangle$	$\langle \langle 0, 1 \rangle, 4 \rangle$		
	$\langle 3, 2 \rangle$	$\langle \langle 2, 1 \rangle, 1 \rangle$	$\langle \langle 2, -1 \rangle, 2 \rangle$	$\langle \langle 1, 1 \rangle, 3 \rangle$	$\langle \langle 1, 0 \rangle, 4 \rangle$	$\langle \langle 1, -1 \rangle, 5 \rangle$	

Table 1: A vector table showing the set  $V$  for the dataset shown in Figure 1(a).



$i$	$\mathbf{V}[i]$	$\mathbf{D}[\mathbf{V}[i, 2]]$	
1	$\langle\langle 0, 1 \rangle, 3\rangle$	$\langle 2, 1 \rangle$	= MTP for $\langle 0, 1 \rangle$
2	$\langle\langle 0, 1 \rangle, 4\rangle$	$\langle 2, 2 \rangle$	
3	$\langle\langle 0, 2 \rangle, 1\rangle$	$\langle 1, 1 \rangle$	= MTP for $\langle 0, 2 \rangle$
4	$\langle\langle 0, 2 \rangle, 3\rangle$	$\langle 2, 1 \rangle$	
5	$\langle\langle 1, -2 \rangle, 2\rangle$	$\langle 1, 3 \rangle$	= MTP for $\langle 1, -2 \rangle$
6	$\langle\langle 1, -1 \rangle, 2\rangle$	$\langle 1, 3 \rangle$	= MTP for $\langle 1, -1 \rangle$
7	$\langle\langle 1, -1 \rangle, 5\rangle$	$\langle 2, 3 \rangle$	
8	$\langle\langle 1, 0 \rangle, 1\rangle$	$\langle 1, 1 \rangle$	= MTP for $\langle 1, 0 \rangle$
9	$\langle\langle 1, 0 \rangle, 2\rangle$	$\langle 1, 3 \rangle$	
10	$\langle\langle 1, 0 \rangle, 4\rangle$	$\langle 2, 2 \rangle$	
11	$\langle\langle 1, 1 \rangle, 1\rangle$	$\langle 1, 1 \rangle$	= MTP for $\langle 1, 1 \rangle$
12	$\langle\langle 1, 1 \rangle, 3\rangle$	$\langle 2, 1 \rangle$	
13	$\langle\langle 1, 2 \rangle, 1\rangle$	$\langle 1, 1 \rangle$	= MTP for $\langle 1, 2 \rangle$
14	$\langle\langle 2, -1 \rangle, 2\rangle$	$\langle 1, 3 \rangle$	= MTP for $\langle 2, -1 \rangle$
15	$\langle\langle 2, 1 \rangle, 1\rangle$	$\langle 1, 1 \rangle$	= MTP for $\langle 2, 1 \rangle$

Table 2: Reading the second column from top to bottom gives  $\mathbf{V}$  for the dataset shown in Figure 1(a). The third column gives  $\mathbf{D}[\mathbf{V}[i, 2]]$  for each element  $\mathbf{V}[i]$  in the second column. The right-hand side of the third column shows how the non-empty MTPs may be derived directly from  $\mathbf{V}$ .

		From					
		$\langle 1, 1 \rangle$	$\langle 1, 3 \rangle$	$\langle 2, 1 \rangle$	$\langle 2, 2 \rangle$	$\langle 2, 3 \rangle$	$\langle 3, 2 \rangle$
To	$\langle 1, 1 \rangle$	$\langle 0, 0 \rangle$	$\langle 0, -2 \rangle$	$\langle -1, 0 \rangle$	$\langle -1, -1 \rangle$	$\langle -1, -2 \rangle$	$\langle -2, -1 \rangle$
	$\langle 1, 3 \rangle$	$\langle 0, 2 \rangle$	$\langle 0, 0 \rangle$	$\langle -1, 2 \rangle$	$\langle -1, 1 \rangle$	$\langle -1, 0 \rangle$	$\langle -2, 1 \rangle$
	$\langle 2, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 1, -2 \rangle$	$\langle 0, 0 \rangle$	$\langle 0, -1 \rangle$	$\langle 0, -2 \rangle$	$\langle -1, -1 \rangle$
	$\langle 2, 2 \rangle$	$\langle 1, 1 \rangle$	$\langle 1, -1 \rangle$	$\langle 0, 1 \rangle$	$\langle 0, 0 \rangle$	$\langle 0, -1 \rangle$	$\langle -1, 0 \rangle$
	$\langle 2, 3 \rangle$	$\langle 1, 2 \rangle$	$\langle 1, 0 \rangle$	$\langle 0, 2 \rangle$	$\langle 0, 1 \rangle$	$\langle 0, 0 \rangle$	$\langle -1, 1 \rangle$
	$\langle 3, 2 \rangle$	$\langle 2, 1 \rangle$	$\langle 2, -1 \rangle$	$\langle 1, 1 \rangle$	$\langle 1, 0 \rangle$	$\langle 1, -1 \rangle$	$\langle 0, 0 \rangle$

Table 3: A vector table showing  $\mathbf{W}$  for the dataset shown in Figure 1(a).

		From			
		$\langle 1, 1 \rangle$	$\langle 1, 2 \rangle$	$\langle 2, 1 \rangle$	$\langle 2, 2 \rangle$
To	$\langle 1, 1 \rangle$	$\langle \langle 0, 0 \rangle, \langle 1, 1 \rangle \rangle$	$\langle \langle 0, -1 \rangle, \langle 1, 2 \rangle \rangle$	$\langle \langle -1, 0 \rangle, \langle 2, 1 \rangle \rangle$	$\langle \langle -1, -1 \rangle, \langle 2, 2 \rangle \rangle$
	$\langle 1, 2 \rangle$	$\langle \langle 0, 1 \rangle, \langle 1, 1 \rangle \rangle$	$\langle \langle 0, 0 \rangle, \langle 1, 2 \rangle \rangle$	$\langle \langle -1, 1 \rangle, \langle 2, 1 \rangle \rangle$	$\langle \langle -1, 0 \rangle, \langle 2, 2 \rangle \rangle$
	$\langle 2, 1 \rangle$	$\langle \langle 1, 0 \rangle, \langle 1, 1 \rangle \rangle$	$\langle \langle 1, -1 \rangle, \langle 1, 2 \rangle \rangle$	$\langle \langle 0, 0 \rangle, \langle 2, 1 \rangle \rangle$	$\langle \langle 0, -1 \rangle, \langle 2, 2 \rangle \rangle$
	$\langle 2, 2 \rangle$	$\langle \langle 1, 1 \rangle, \langle 1, 1 \rangle \rangle$	$\langle \langle 1, 0 \rangle, \langle 1, 2 \rangle \rangle$	$\langle \langle 0, 1 \rangle, \langle 2, 1 \rangle \rangle$	$\langle \langle 0, 0 \rangle, \langle 2, 2 \rangle \rangle$
	$\langle 2, 3 \rangle$	$\langle \langle 1, 2 \rangle, \langle 1, 1 \rangle \rangle$	$\langle \langle 1, 1 \rangle, \langle 1, 2 \rangle \rangle$	$\langle \langle 0, 2 \rangle, \langle 2, 1 \rangle \rangle$	$\langle \langle 0, 1 \rangle, \langle 2, 2 \rangle \rangle$
	$\langle 2, 4 \rangle$	$\langle \langle 1, 3 \rangle, \langle 1, 1 \rangle \rangle$	$\langle \langle 1, 2 \rangle, \langle 1, 2 \rangle \rangle$	$\langle \langle 0, 3 \rangle, \langle 2, 1 \rangle \rangle$	$\langle \langle 0, 2 \rangle, \langle 2, 2 \rangle \rangle$
	$\langle 3, 3 \rangle$	$\langle \langle 2, 2 \rangle, \langle 1, 1 \rangle \rangle$	$\langle \langle 2, 1 \rangle, \langle 1, 2 \rangle \rangle$	$\langle \langle 1, 2 \rangle, \langle 2, 1 \rangle \rangle$	$\langle \langle 1, 1 \rangle, \langle 2, 2 \rangle \rangle$
	$\langle 3, 4 \rangle$	$\langle \langle 2, 3 \rangle, \langle 1, 1 \rangle \rangle$	$\langle \langle 2, 2 \rangle, \langle 1, 2 \rangle \rangle$	$\langle \langle 1, 3 \rangle, \langle 2, 1 \rangle \rangle$	$\langle \langle 1, 2 \rangle, \langle 2, 2 \rangle \rangle$

Table 4: A vector table showing the set  $V_{\text{SIAME}}$  generated by Step 1 of SIAME for the query pattern in Figure 3(a) and the dataset in Figure 3(b).

# METHOD OF PATTERN DISCOVERY

## Field of the invention

This invention relates to the fields of pattern matching, pattern discovery and data compression. In particular, it relates to pattern matching, pattern discovery and data compression in multidimensional numerical data.

Pattern discovery, pattern matching and data compression in multidimensional numerical datasets can be used in many areas such as audio and video compression, data indexing and drug design.

## Related art

Algorithms already exist for data compression, information retrieval and structural analysis of data. However, most existing approaches are based on string matching techniques that require the datasets to be represented as strings of characters before they are processed. In other words, most existing approaches attempt to process multidimensional numerical data using techniques originally designed for processing one-dimensional textual data. String-based approaches to processing multidimensional datasets are artificially limited as to the types of patterns that can be discovered and searched for; and certain information-retrieval tasks (such as, for example, searching for patterns with gaps in multidimensional data) are unnecessarily awkward to accomplish using these techniques. For an overview of string-matching techniques in general, see Crochemore and Rytter (1994). For an introduction to pattern-matching techniques in bioinformatics, see Gusfield (1997).

Although previous approaches to pattern matching, pattern discovery and data compression are based on the assumption that the data to be processed is represented in the form of a string of symbols or as a set of such symbol strings, there are many domains in which data cannot be appropriately represented using strings. In such domains, existing

methods for pattern matching, pattern discovery and data compression are not effective. In many domains in which information cannot appropriately be represented using strings, multidimensional numerical datasets can be used instead.

## Summary of the invention

In a first aspect of the present invention, there is a method of pattern discovery in a dataset, in which the dataset is represented as a set of datapoints in an  $n$ -dimensional space, comprising the step of computing inter-datapoint vectors.

The present invention is based on the insight that the properties of multidimensional datasets can be expressed naturally in geometrical terms (using concepts such as vectors, points and geometrical transformations like translation) and that pattern discovery can be based on computing inter-datapoint vectors. Multidimensional datasets can therefore be directly analysed using the mathematical concepts and theory that were originally developed for manipulating this kind of data. More specifically, in an implementation designed to identify translation invariant sets of datapoints within the dataset, the method comprises the further steps of:

- (a) computing the largest set of datapoints that can be translated by a given inter-datapoint vector to another set of datapoints in the dataset; and
- (b) computing all sets of datapoints which are translationally equivalent to the largest set identified in step (a).

This method of finding internal recurring structures within a multi-dimensional dataset can be used (without limitation) for any of the following purposes:

- (a) lossless data-compression;
- (b) predicting the future price of a tradable commodity;
- (c) locating repeating elements in a molecule; and

(d) indexing.

A pattern matching implementation of the present invention further differs over the prior art as follows: most existing approaches to pattern-discovery and pattern-matching employ techniques based on the idea of trying to align a query pattern (e.g. a user-supplied regular expression) against the dataset at each possible position. Implementations of the present invention eschew alignment-based techniques in favour of a data driven approach based on the fact that if there exists a pattern  $P$  in a dataset that is translationally invariant to a query pattern  $Q$ , then there will exist at least one query pattern datapoint  $q$  and one dataset point  $p$  such that the vector that maps  $q$  onto  $p$  is equal to the vector that maps  $Q$  onto  $P$ . Hence, in an implementation adapted to identify the occurrence of a user supplied set of datapoints in a dataset, the method comprises the further steps of:

- (a) computing inter-datapoint vectors from each datapoint in the user supplied set of datapoints to each datapoint in the dataset;
- (b) computing the largest set of datapoints in the user supplied set of datapoints that can be translated by a given inter-datapoint vector to another set of datapoints in the dataset.

This implementation can be used (without limitation) for any of the following purposes:

- (a) locating specific elements in a molecule;
- (b) visual pattern comparison;
- (c) speech or music recognition.

The present invention finds broad application whenever multi-dimensional datasets need to be analysed for internal patterns or for matches against external queries. Typically, datapoints in an  $n$ -dimensional space can therefore represent any of the following:

- (a) audio data;

- (b) 2D image data;
- (c) 3D representations of virtual spaces;
- (d) video data;
- (e) molecular structure;
- (f) chemical spectra;
- (g) financial data;
- (h) seismic data:
- (i) meteorological data;
- (j) symbolic music representations;
- (k) CAD circuit data.

In another aspect of the invention, there is provided computer software adapted to perform the method described above.

## List of figures and tables

The present invention will be described with reference to the accompanying drawings and tables, a brief description of which follows.

**Figure 1** (a) shows a simple 2-dimensional dataset. (b)–(j) show the maximal repeated patterns found by SIA in the dataset in (a).

**Figure 2** The sets of patterns discovered by SIATEC in the dataset in Figure 1(a).

**Figure 3** When SIAME searches for occurrences of the query pattern (a) in the dataset (b), it finds the exact matches shown in (c). It also finds the closest incomplete matches shown in (d).

**Figure 4** (b) shows the compressed representation generated by COSIATEC for the dataset (a). The dataset in (a) can be generated by translating the three-point pattern in (b) by the three vectors represented by arrows.

**Figure 5** The set  $\mathcal{S}(D)$  for the dataset in Figure 1(a).

**Figure 6** The set  $\mathcal{J}'(D)$  for the dataset in Figure 1(a).

**Figure 7** An algorithm for printing out  $\mathcal{S}(D)$  using  $\mathbf{V}$  and  $\mathbf{D}$ .

**Figure 8** The output of the algorithm in Figure 7 for the dataset in Figure 1(a).

**Figure 9** An algorithm for computing  $X$  using  $\mathbf{V}$  and  $\mathbf{D}$ .

**Figure 10** The ordered set  $\mathbf{X}$  for the dataset in Figure 1(a).

**Figure 11** The ordered set  $\mathbf{Y}$  for the dataset in Figure 1(a).

**Figure 12** An algorithm for printing out  $\mathcal{J}'(D)$ .

**Figure 13** The PRINT\_PATTERN algorithm.

**Figure 14** The PRINT\_SET\_OF\_TRANSLATORS algorithm.

**Figure 15** The output of the algorithm in Figure 12 for the dataset in Figure 1(a).

**Figure 16** The ordered set  $\mathbf{V}_{\text{SIAME}}$  computed by Step 2 of SIAME for the pattern in Figure 3(a) and the dataset in Figure 3(b).

**Figure 17** An algorithm for computing  $N$  using  $\mathbf{V}_{\text{SIAME}}$ .

**Figure 18**  $\mathbf{N}$  for the pattern in Figure 3(a) and the dataset in Figure 3(b).

**Figure 19**  $\mathbf{N}'$  for the pattern in Figure 3(a) and the dataset in Figure 3(b).

**Figure 20** An algorithm for computing  $\mathcal{M}'(P, D)$  from  $\mathbf{N}'$  and  $\mathbf{V}_{\text{SIAME}}$ .

**Figure 21**  $\mathbf{M}$  for the pattern in Figure 3(a) and the dataset in Figure 3(b).



**Figure 22** The COSIATEC algorithm.

**Figure 23** Globally defined data types used in the algorithms.

**Figure 24** The SIA algorithm.

**Figure 25** The READ\_VECTOR\_SET algorithm.

**Figure 26** The SORT\_DATASET algorithm.

**Figure 27** The MERGE\_DATASET\_ROWS algorithm.

**Figure 28** The SETIFY\_DATASET algorithm.

**Figure 29** The SIA\_COMPUTE\_VECTORS algorithm.

**Figure 30** The SIA\_SORT\_VECTORS algorithm.

**Figure 31** The SIA\_MERGE\_VECTOR\_COLUMNS algorithm.

**Figure 32** The PRINT\_VECTOR\_MTP\_PAIRS algorithm.

**Figure 33** The SIATEC algorithm.

**Figure 34** The COMPUTE\_VECTORS algorithm.

**Figure 35** The CONSTRUCT\_VECTOR\_TABLE algorithm.

**Figure 36** The SORT\_VECTORS algorithm.

**Figure 37** The MERGE\_VECTOR\_COLUMNS algorithm.

**Figure 38** The VECTORIZE\_PATTERNS algorithm.

**Figure 39** The SORT\_PATTERN\_VECTOR\_SEQUENCES algorithm.

**Figure 40** The MERGE\_PATTERN\_ROWS algorithm.

**Figure 41** The PRINT\_TECS algorithm.

**Figure 42** The PRINT\_PATTERN algorithm.

**Figure 43** The PRINT\_SET\_OF\_TRANSLATORS algorithm.

**Figure 44** The COSIATEC algorithm.

**Figure 45** The DISPOSE\_OF\_SIATEC\_DATA\_STRUCTURES algorithm.

**Figure 46** The READ\_TEC algorithm.

**Figure 47** The SET\_TEC\_COVERED\_SET algorithm.

**Figure 48** The IS\_BETTER\_TEC algorithm.

**Figure 49** The PRINT\_TEC algorithm.

**Figure 50** The PRINT\_VECTOR\_SET algorithm.

**Figure 51** The DELETE\_TEC\_COVERED\_SET algorithm.

**Figure 52** Example of format used as input to READ\_VECTOR\_SET algorithm.

**Figure 53** Using NUMBER\_NODES to represent vectors.

**Figure 54** A right-directed list of VECTOR\_NODES.

**Figure 55** A down-directed list of VECTOR\_NODES.

**Figure 56** The linked list constructed by READ\_VECTOR\_SET when F is the data in Figure 52, DIR = DOWN and SD = "101".

**Figure 57** The linked list constructed by READ\_VECTOR\_SET when F is the data in Figure 52, DIR = RIGHT and SD = NULL.

**Figure 58** Example input data.

**Figure 59** The linked list generated by line 5 of SIA (Figure 24) for the data in Figure 58.

**Figure 60** The state of the linked list D after one iteration of the outer `while` loop of `SORT_DATASET` on the dataset list in Figure 59.

**Figure 61** The sorted, right-directed linked list produced by `SORT_DATASET` from the unsorted, down-directed dataset list in Figure 59.

**Figure 62** The linked list that results when `SETIFY_DATASET` has been executed on the linked list in Figure 61.

**Figure 63** The data structure that results after `SIA_COMPUTE_VECTORS` has executed when the `SIA` algorithm in Figure 24 is carried out on the dataset shown in Figure 1(a).

**Figure 64** The data structure headed by `V` after `SIA_SORT_VECTORS` has executed when `SIA` is carried out on the dataset in Figure 1(a).

**Figure 65** The output generated by `PRINT_VECTOR_MTP_PAIRS` (Figure 32) for the dataset in Figure 1(a).

**Figure 66** The data structure generated by `COMPUTE_VECTORS` for the dataset in Figure 1(a).

**Figure 67** The data structures that result after `CONSTRUCT_VECTOR_TABLE` has executed when the `SIATEC` implementation in Figure 33 is run on the dataset in Figure 1(a).

**Figure 68** The data structures that result after `SORT_VECTORS` has executed when the `SIATEC` implementation in Figure 33 is run on the dataset in Figure 1(a).

**Figure 69** Diagrammatic representation of an `X_NODE`.

**Figure 70** The state of the data structures headed by `D`, `V` and `X` in the `SIATEC` implementation in Figure 33 after line 27 has been executed when this implementation is run on the dataset in Figure 1(a).

**Figure 71** The state of the data structures headed by  $D$ ,  $V$  and  $X$  in the SIATEC implementation in Figure 33 after line 28 has been executed when this implementation is run on the dataset in Figure 1(a).

**Figure 72** The output generated by PRINT\_TECS (Figure 41) for the dataset in Figure 1(a).

**Figure 73** The output generated by COSIATEC (Figure 44) for the dataset in Figure 4.

**Figure 74** An illustration of the data structures used in SIAME.

**Figure 75** The NEWLINK algorithm.

**Figure 76** First implementation of SIAME algorithm.

**Figure 77** Second implementation of SIAME.

**Figure 78** The MERGEDUPLICATES algorithm.

**Table 1** A vector table showing the set  $V$  for the dataset shown in Figure 1(a).

**Table 2** Reading the second column from top to bottom gives  $\mathbf{V}$  for the dataset shown in Figure 1(a). The third column gives  $\mathbf{D}[\mathbf{V}[i, 2]]$  for each element  $\mathbf{V}[i]$  in the second column. The right-hand side of the third column shows how the non-empty MTPs may be derived directly from  $\mathbf{V}$ .

**Table 3** A vector table showing  $\mathbf{W}$  for the dataset shown in Figure 1(a).

**Table 4** A vector table showing the set  $V_{\text{SIAME}}$  generated by Step 1 of SIAME for the query pattern in Figure 3(a) and the dataset in Figure 3(b).

## Detailed Description of Preferred Implementations

The aim of the present invention is to provide methods for pattern matching, pattern discovery and data compression in multidimensional datasets. More specifically, the following four related algorithms are described:

1. an algorithm called **SIA** that takes a multidimensional dataset as input and computes all the largest repeated patterns in the dataset;
2. an algorithm called **SIATEC** that takes a multidimensional dataset as input and computes all the occurrences of all the largest repeated patterns in the dataset;
3. an algorithm called **SIAME** that takes a multidimensional query pattern and a multidimensional dataset as input and finds all partial and complete occurrences of the query pattern in the dataset; and
4. an algorithm called **COSIATEC** that takes a multidimensional dataset as input and computes a compressed (i.e. space-efficient) representation of the dataset (i.e., it losslessly compresses the dataset).

**SIA** discovers the largest (or ‘maximal’) repeated patterns in a multidimensional dataset. For example, if the 2-dimensional dataset shown in Figure 1(a) is given to **SIA** as input, **SIA** discovers the pairs of patterns shown in Figure 1(b)-(j).

**SIATEC** first uses **SIA** to find all the maximal repeated patterns and then it finds all the occurrences of these patterns in the dataset. Figure 2(a)-(d) shows the output of **SIATEC** for the dataset in Figure 1(a).

**SIA** and **SIATEC** are pattern discovery algorithms: they autonomously discover repeated structures in data. **SIAME**, on the other hand, is an information-retrieval or pattern *matching* algorithm: the user supplies a query pattern and a dataset and **SIAME** searches the dataset for occurrences of the query pattern. For example, if a molecular biologist wanted to find all the occurrences of the purine base adenine in a DNA molecule, he/she could give **SIAME** two items of input:

1. a multidimensional representation of adenine as the query pattern; and
2. a multidimensional representation of the DNA molecule as the dataset.

**SIAME** would then output a list indicating, first, all the exact occurrences of adenine in the DNA molecule; then, all the closest incomplete matches (i.e., one atom different); then all the incomplete matches with two atoms different; and so on. **SIAME** can also be used to compare datasets: the two datasets to be compared are given to **SIAME** as input and **SIAME** computes all the ways in which the two datasets may be matched, returning the best matches first. Figure 3(c) shows the exact matches found by **SIAME** for the query pattern in Figure 3(a) in the dataset in Figure 3(b). Figure 3(d) shows the closest incomplete matches found by **SIAME** for the same query pattern in the same dataset.

**COSIATEC** generates a compressed representation of a dataset by repeatedly applying **SIATEC**. For example, Figure 4(a) shows the dataset

$$\{\langle 1, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle, \langle 4, 1 \rangle, \langle 4, 2 \rangle, \langle 4, 3 \rangle, \langle 5, 2 \rangle\}.$$

Note that to store this dataset explicitly, 12 vectors need to be specified, one for each datapoint in the dataset. When this dataset is given as input to **COSIATEC**, the algorithm generates the following ordered pair of sets

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

The first set of vectors in this ordered pair,  $\{\langle 1, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 2 \rangle\}$ , represents the three-point pattern shown in Figure 4(b). The second set of vectors,  $\{\langle 1, 0 \rangle, \langle 2, 0 \rangle, \langle 3, 0 \rangle\}$ , represents the three translation vectors indicated by arrows in Figure 4(b). The dataset in Figure 4(a) can be generated by translating the three-point pattern in Figure 4(b) by the vectors indicated by the arrows in the diagram. Note that to store this compressed representation, only 6 vectors need to be specified. In this particular case, therefore, **COSIATEC** generates a compressed representation that uses only half the space used to

store the original dataset. The degree of compression achievable using COSIATEC depends on the amount of repetition in the dataset to be compressed.

# 1 The mathematical functions computed by the algorithms

## 1.1 Preliminary mathematical concepts

Before specifying the mathematical functions computed by the SIA, SIATEC, COSIATEC and SIAME algorithms, it is necessary to define some preliminary mathematical concepts.

A *vector* is a  $k$ -tuple of real numbers viewed as a member of a  $k$ -dimensional Euclidean space (Borowski and Borwein, 1989, p. 624, s.v. **vector**, sense 2). A vector in a  $k$ -dimensional Euclidean space will be represented here as an ordered set of  $k$  real numbers.

If  $\mathbf{A}$  is an ordered set or a vector then we denote the cardinality of  $\mathbf{A}$  by  $|\mathbf{A}|$  and the  $i$ th element of  $\mathbf{A}$  by  $\mathbf{A}[i]$ . If  $\mathbf{u}$  and  $\mathbf{v}$  are two vectors such that  $|\mathbf{u}| = |\mathbf{v}| = k$  then we say that  $\mathbf{u}$  is *less than*  $\mathbf{v}$ , denoted by  $\mathbf{u} < \mathbf{v}$ , if and only if there exists an integer  $i$  such that  $1 \leq i \leq k$  and  $\mathbf{u}[i] < \mathbf{v}[i]$  and  $\mathbf{u}[j] = \mathbf{v}[j]$  for  $1 \leq j < i$ . For example,  $\langle 1, 1 \rangle < \langle 1, 2 \rangle < \langle 2, 1 \rangle$ .

If  $A$  and  $B$  are ordered sets such that  $A = \langle a_1, a_2, \dots, a_m \rangle$  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

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

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

In set theory, recall that  $\emptyset$  denotes the empty set and that  $A \setminus B$  denotes the set

that contains all elements of  $A$  except those that are also elements of  $B$ . Otherwise, a knowledge of basic set theory and notation will be assumed.

An object is a *vector set* if and only if it is a set of vectors. An object is 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 a  $k$ -dimensional vector set that is a subset of some specified dataset or a transformation of some subset of a dataset. Also, if we have two  $k$ -dimensional vector sets  $P$  and  $D$  and we wish to search for occurrences of  $P$  in  $D$  then we would usually refer to  $P$  as a pattern and  $D$  as a dataset.

Let  $D$  be a dataset and let  $\mathbf{d}_1$  and  $\mathbf{d}_2$  be any two datapoints in  $D$ . The vector from  $\mathbf{d}_1$  to  $\mathbf{d}_2$  is given by  $\mathbf{d}_2 - \mathbf{d}_1$  where the minus sign denotes *vector* subtraction. If  $\mathbf{v} = \mathbf{d}_2 - \mathbf{d}_1$  then  $\mathbf{d}_2 = \mathbf{v} + \mathbf{d}_1$  ('+' here denotes *vector* addition) which expresses the fact that the datapoint  $\mathbf{d}_1$  can be *translated* by the vector  $\mathbf{v}$  to give the datapoint  $\mathbf{d}_2$ .

We denote by  $\tau(P, \mathbf{v})$  the pattern that results when the pattern  $P$  is translated by the vector  $\mathbf{v}$ . Formally,

$$\tau(P, \mathbf{v}) = \{\mathbf{d} + \mathbf{v} \mid \mathbf{d} \in P\}. \quad (1)$$

We say that two patterns  $P_1$  and  $P_2$  are *translationally equivalent*, denoted by  $P_1 \equiv_{\tau} P_2$ , if and only if there exists a vector  $\mathbf{v}$  such that  $\tau(P_1, \mathbf{v}) = P_2$ . We say that a pattern  $P$  is *translatable* by a vector  $\mathbf{v}$  in a dataset  $D$  if and only if  $\tau(P, \mathbf{v}) \subseteq D$ .

The *maximal translatable pattern* (MTP) for a vector  $\mathbf{v}$  in a dataset  $D$ , denoted by  $MTP(\mathbf{v}, D)$ , is the largest pattern translatable by  $\mathbf{v}$  in  $D$ . Formally,

$$MTP(\mathbf{v}, D) = \{\mathbf{d} \mid \mathbf{d} \in D \wedge \mathbf{d} + \mathbf{v} \in D\}. \quad (2)$$

The MTP for a vector  $\mathbf{v}$  in a dataset  $D$  is non-empty if and only if there exist at least



two datapoints  $\mathbf{d}_1$  and  $\mathbf{d}_2$  in  $D$  such that  $\mathbf{v} = \mathbf{d}_2 - \mathbf{d}_1$ . This implies that the complete set of non-empty MTPs for a dataset  $D$  is given by

$$\mathcal{P}(D) = \{MTP(\mathbf{d}_2 - \mathbf{d}_1, D) \mid \mathbf{d}_1, \mathbf{d}_2 \in D\}. \quad (3)$$

## 1.2 The function computed by SIA

SIA computes all the non-empty MTPs in a dataset. However, it is not necessary for SIA to compute explicitly all the elements of  $\mathcal{P}(D)$  in Eq.3, because, in general, if the MTP for  $\mathbf{v}$  is translated by  $\mathbf{v}$ , the resulting pattern is the MTP for the vector  $-\mathbf{v}$ . This will now be proved.

**Lemma 1** *If  $D$  is a dataset and  $\mathbf{v}$  is a vector then*

$$\tau(MTP(\mathbf{v}, D), \mathbf{v}) = MTP(-\mathbf{v}, D). \quad (4)$$

*Proof*

From Eq.1 we deduce that

$$\tau(MTP(\mathbf{v}, D), \mathbf{v}) = \{\mathbf{d}_1 + \mathbf{v} \mid \mathbf{d}_1 \in MTP(\mathbf{v}, D)\}. \quad (5)$$

Substituting Eq.2 into Eq.5, we find that

$$\begin{aligned} \tau(MTP(\mathbf{v}, D), \mathbf{v}) &= \{\mathbf{d}_1 + \mathbf{v} \mid \mathbf{d}_1 \in \{\mathbf{d}_2 \mid \mathbf{d}_2 \in D \wedge \mathbf{d}_2 + \mathbf{v} \in D\}\} \\ &= \{\mathbf{d}_2 + \mathbf{v} \mid \mathbf{d}_2 \in D \wedge \mathbf{d}_2 + \mathbf{v} \in D\}. \end{aligned} \quad (6)$$

If we let  $\mathbf{d}_3 = \mathbf{d}_2 + \mathbf{v}$  and substitute this into Eq.6, we deduce that

$$\tau(MTP(\mathbf{v}, D), \mathbf{v}) = \{\mathbf{d}_3 \mid \mathbf{d}_3 - \mathbf{v} \in D \wedge \mathbf{d}_3 \in D\}. \quad (7)$$

Eqs.7 and 2 together imply

$$\tau(MTP(\mathbf{v}, D), \mathbf{v}) = MTP(-\mathbf{v}, D).$$

■

Lemma 1 tells us that if we compute  $MTP(\mathbf{d}_2 - \mathbf{d}_1, D)$  then we can find  $MTP(\mathbf{d}_1 - \mathbf{d}_2, D)$  simply by translating  $MTP(\mathbf{d}_2 - \mathbf{d}_1, D)$  by  $\mathbf{d}_2 - \mathbf{d}_1$ . It is also clear that  $MTP(\mathbf{0}, D) = D$  where  $\mathbf{0}$  is the zero vector. These two facts imply that if our goal is only to compute all the non-empty MTPs in a dataset then we only really need to compute the set

$$\mathcal{P}'(D) = \{MTP(\mathbf{d}_2 - \mathbf{d}_1, D) \mid \mathbf{d}_1, \mathbf{d}_2 \in D \wedge \mathbf{d}_1 < \mathbf{d}_2\}. \quad (8)$$

However, if **SIA** simply generated the set  $\mathcal{P}'(D)$ , then it would not be possible to determine the vector for which any given element of  $\mathcal{P}'(D)$  was the MTP. Therefore, **SIA** actually computes the set

$$\mathcal{S}(D) = \{\langle \mathbf{d}_2 - \mathbf{d}_1, MTP(\mathbf{d}_2 - \mathbf{d}_1, D) \rangle \mid \mathbf{d}_1, \mathbf{d}_2 \in D \wedge \mathbf{d}_1 < \mathbf{d}_2\}. \quad (9)$$

Each member of  $\mathcal{S}(D)$  is an ordered pair in which the first element is a vector  $\mathbf{v}$  and the second element is the MTP for  $\mathbf{v}$  in  $D$ . Figure 5 shows  $\mathcal{S}(D)$  for the dataset in Figure 1(a).

### 1.3 The function computed by SIATEC

SIATEC computes all the occurrences of all the non-empty MTPs in a dataset. If  $D$  is a dataset and  $P \subseteq D$  is a pattern in  $D$  then we define the *translational equivalence class* (TEC) of  $P$  in  $D$  to be the set

$$TEC(P, D) = \{Q \mid Q \equiv_{\tau} P \wedge Q \subseteq D\}. \quad (10)$$

The four graphs in Figure 2(a)-(d) show the four TECs computed by **SIATEC** for the dataset in Figure 1(a). The aim of **SIATEC** is to compute efficiently all the TECs of all the non-empty MTPs for a dataset  $D$ , that is,

$$\mathcal{T}(D) = \{TEC(MTP(\mathbf{d}_2 - \mathbf{d}_1, D), D) \mid \mathbf{d}_1, \mathbf{d}_2 \in D\}. \quad (11)$$

The translational equivalence relation is reflexive, transitive and symmetric and partitions the power set of a dataset into translational equivalence classes. This means that every pattern in a dataset is a member of exactly one TEC. However, from Lemma 1 we know that

$$\tau(MTP(\mathbf{d}_2 - \mathbf{d}_1, D), \mathbf{d}_2 - \mathbf{d}_1) = MTP(\mathbf{d}_1 - \mathbf{d}_2, D).$$

Therefore

$$TEC(MTP(\mathbf{d}_2 - \mathbf{d}_1, D), D) = TEC(MTP(\mathbf{d}_1 - \mathbf{d}_2, D), D).$$

Moreover, we know that  $MTP(\mathbf{0}, D) = D$  and therefore  $TEC(MTP(\mathbf{0}, D), D) = \{D\}$  which is a trivial translational equivalence class. Therefore, instead of computing  $\mathcal{T}(D)$  as defined in Eq.11, **SIATEC** actually computes the set

$$\mathcal{T}'(D) = \{TEC(MTP(\mathbf{d}_2 - \mathbf{d}_1, D), D) \mid \mathbf{d}_1, \mathbf{d}_2 \in D \wedge \mathbf{d}_1 < \mathbf{d}_2\}. \quad (12)$$

It can easily be seen that  $\mathcal{T}(D) = \mathcal{T}'(D) \cup \{\{D\}\}$ .

If  $P$  is a pattern in a dataset  $D$  then we say that  $\mathbf{v}$  is a *translator* of  $P$  in  $D$  if and only if  $P$  is translatable by  $\mathbf{v}$  in  $D$ . The *set of translators* for  $P$  in  $D$ , which we denote by  $T(P, D)$ , is the set that only contains all vectors by which  $P$  is translatable in  $D$ . Formally,

$$T(P, D) = \{\mathbf{v} \mid \tau(P, \mathbf{v}) \subseteq D\}. \quad (13)$$

For example, the set of translators for the three-point pattern in Figure 4(b) is the set  $\{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 2, 0 \rangle, \langle 3, 0 \rangle\}$ . Any pattern  $P$  in a dataset  $D$  is translatable in  $D$  by the

zero vector,  $\mathbf{0}$ .  $\mathbf{0}$  is therefore considered a *trivial translator*. Any non-zero translator of a pattern  $P$  in a dataset  $D$  is a *non-trivial translator* of  $P$  in  $D$ . The *set of non-trivial translators* for a pattern  $P$  in a dataset  $D$  is therefore given by

$$T(P, D) \setminus \{\mathbf{0}\}. \quad (14)$$

The TEC of a pattern  $P$  in a dataset  $D$  can therefore be represented efficiently by the ordered pair  $\langle P, T(P, D) \setminus \{\mathbf{0}\} \rangle$ . That is,  $\langle P, T(P, D) \setminus \{\mathbf{0}\} \rangle$  denotes the set of patterns

$$\bigcup_{\mathbf{v} \in T(P, D)} \{\tau(P, \mathbf{v})\}. \quad (15)$$

For any given TEC,  $E$ , there are  $|E|$  such representations, one for each pattern in  $E$ . In general, this ordered-pair representation for a TEC can be much more space-efficient than explicitly writing out every member pattern of the TEC in full. For example, if there are 20 patterns in a dataset that are translationally equivalent to a pattern  $P$  containing 10 datapoints, then printing out the TEC for  $P$  in full would involve printing 200 datapoints. However, if this TEC were represented as the ordered pair  $\langle P, T(P, D) \setminus \{\mathbf{0}\} \rangle$  then only  $10 + 19 = 29$  vectors would need to be printed. This provides the basis for the compression algorithm, COSIATEC, described below.

In the output of SIATEC, each distinct TEC,  $E$ , in  $\mathcal{T}(D)$  is therefore represented as an ordered pair  $\langle P, T(P, D) \setminus \{\mathbf{0}\} \rangle$  where  $P$  is a member of  $E$  and  $T(P, D)$  is the set of translators for  $P$  in  $D$ . Figure 6 shows  $\mathcal{T}(D)$  for the dataset shown in Figure 1(a).

## 1.4 The function computed by SIAME

SIAME takes a query pattern  $P$  and a dataset  $D$  and finds all the partial and complete translation-invariant occurrences of  $P$  in  $D$ . The *maximal match* (MM) for a query pattern  $P$  and a vector  $\mathbf{v}$  in a dataset  $D$ , denoted by  $MM(P, \mathbf{v}, D)$  is the set of datapoints in  $P$

that can be translated by  $\mathbf{v}$  to give datapoints in  $D$ . Formally,

$$MM(P, \mathbf{v}, D) = \{\mathbf{p} \mid \mathbf{p} \in P \wedge \mathbf{p} + \mathbf{v} \in D\}. \quad (16)$$

Note that for any dataset  $D$ ,  $MM(D, \mathbf{v}, D) = MTP(\mathbf{v}, D)$  (see Eq.2). The concept of a maximal match is therefore a generalization of the concept of a maximal translatable pattern. A maximal match  $MM(P, \mathbf{v}, D)$  will be non-empty if and only if there exist two datapoints,  $\mathbf{p} \in P, \mathbf{d} \in D$ , such that  $\mathbf{v} = \mathbf{d} - \mathbf{p}$ . The complete set of maximal matches for a pattern  $P$  and a dataset  $D$  is therefore given by

$$\mathcal{M}(P, D) = \{MM(P, \mathbf{d} - \mathbf{p}, D) \mid \mathbf{d} \in D \wedge \mathbf{p} \in P\}. \quad (17)$$

Note that  $\mathcal{M}(D, D) = \mathcal{P}(D)$  (see Eq.3). The aim of **SIAME** is to compute all the non-empty maximal matches for a given pattern and dataset. However, if **SIAME** simply generated the set  $\mathcal{M}(P, D)$ , it would be impossible to determine the vector for which each pattern in  $\mathcal{M}(P, D)$  was a maximal match. **SIAME** therefore computes the set

$$\mathcal{M}'(P, D) = \{\langle \mathbf{d} - \mathbf{p}, MM(P, \mathbf{d} - \mathbf{p}, D) \rangle \mid \mathbf{d} \in D \wedge \mathbf{p} \in P\}. \quad (18)$$

## 1.5 The mapping computed by COSIATEC

**COSIATEC** uses **SIATEC** to generate a compressed representation of a dataset. As explained above, each TEC,  $E$ , in the output of **SIATEC** is represented as an ordered pair  $\langle P, T(P, D) \setminus \mathbf{0} \rangle$  such that

$$E = \bigcup_{\mathbf{v} \in T(P, D)} \{\tau(P, \mathbf{v})\}.$$

If  $E = \langle P, T(P, D) \setminus \mathbf{0} \rangle$  is a TEC in a dataset  $D$ , then the *coverage* of  $E$ , denoted by  $COV(E)$  is given by

$$COV(E) = \left| \bigcup_{Q \in E} Q \right| \quad (19)$$

and the *compression ratio* of  $E$ , denoted by  $CR(E)$  is defined to be

$$CR(E) = \frac{COV(E)}{|P| + |T(P, D) \setminus \mathbf{0}|} \quad (20)$$

We can now define  $\mathcal{E}_{\text{best}}(D)$  to be the set of TECs,  $E \in \mathcal{T}(D)$ , for which the vector  $\langle CR(E), COV(E) \rangle$  is a maximum (recall definition of vector inequality on page 12 above). That is,  $E \in \mathcal{E}_{\text{best}}(D)$  if and only if  $E \in \mathcal{T}(D)$  and there exists no  $E' \in \mathcal{T}(D)$  such that  $\langle CR(E), COV(E) \rangle < \langle CR(E'), COV(E') \rangle$ .

COSIATEC takes a dataset  $D$  as input and computes an ordered set of TECs

$$\langle E_1, E_2, \dots, E_r \rangle$$

satisfying the following conditions:

1. For all  $1 \leq k \leq r$ ,  $E_k \in \mathcal{E}_{\text{best}}(D_k)$  where

$$D_k = \begin{cases} D, & \text{when } k = 1; \\ D_{k-1} \setminus \bigcup_{P \in E_{k-1}} P, & \text{when } 1 < k \leq r. \end{cases}$$

2.  $D_r \neq \emptyset$  and  $D_{r+1} = \emptyset$ .

## 2 The algorithms

The SIA, SIATEC, SIAME and COSIATEC algorithms will now be described. Detailed example implementations will then be presented in section 3.

### 2.1 The SIA algorithm

When given a multidimensional dataset,  $D$ , as input, SIA computes  $\mathcal{S}(D)$  as defined in Eq.9 above. For a  $k$ -dimensional dataset containing  $n$  datapoints, the worst-case running

time of **SIA** is  $O(kn^2 \log_2 n)$  and its worst-case space complexity is  $O(kn^2)$ . The algorithm consists of the following four steps.

### 2.1.1 **SIA: Step 1 – Sorting the dataset**

The first step in **SIA** is to sort the dataset  $D$  to give an ordered set  $\mathbf{D}$  that contains all and only the datapoints in  $D$  in increasing order. For the dataset in Figure 1(a), the result of this first step would be the ordered set

$$\mathbf{D} = \langle \langle 1, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle \rangle. \quad (21)$$

For a  $k$ -dimensional dataset of size  $n$ , this can be done using merge sort (Cormen *et al.*, 1990, pp. 12–15) in a worst-case running time of  $O(kn \log_2 n)$ . When merge sort is implemented using arrays, it requires linear extra memory and the additional work spent copying to and from the temporary array throughout the algorithm has the effect of slowing down the sort considerably. However, in the example implementation described in section 3.1 below, we use a special implementation of merge sort that employs linked lists and in this implementation no extra memory is required and no copying of data is performed.

### 2.1.2 **SIA: Step 2 – Computing inter-datapoint vectors**

The second step in **SIA** is to compute the set

$$V = \{ \langle \mathbf{D}[j] - \mathbf{D}[i], i \rangle \mid 1 \leq i < j \leq |\mathbf{D}| \}. \quad (22)$$

Note that each member of  $V$  is an ordered pair in which the first element is the vector from datapoint  $\mathbf{D}[i]$  to datapoint  $\mathbf{D}[j]$  and the second element is the index of the ‘origin’ datapoint,  $\mathbf{D}[i]$ , in  $\mathbf{D}$ . For the dataset in Figure 1(a),  $V$  contains all the elements below the leading diagonal in Table 1.

We call a table like the one in Table 1 a *vector table*. Each element in this table is an ordered pair  $\langle \mathbf{v}, i \rangle$  where  $i$  gives the number of the column in which the element occurs and  $\mathbf{v}$  is the vector *from* the datapoint at the head of the column in which the element occurs *to* the datapoint at the head of the row in which the element occurs. For a  $k$ -dimensional dataset of size  $n$ , this second step of SIA involves computing  $\frac{n(n-1)}{2}$  vector subtractions. It can be accomplished in a worst-case running time of  $O(kn^2)$ .

### 2.1.3 SIA: Step 3 – Sorting the vectors in the vector table

If  $\langle \mathbf{u}, i \rangle$  and  $\langle \mathbf{v}, j \rangle$  are any two elements in the set  $V$  computed in the second step SIA (Eq.22) then we define that  $\langle \mathbf{u}, i \rangle$  is *less than*  $\langle \mathbf{v}, j \rangle$ , denoted by  $\langle \mathbf{u}, i \rangle < \langle \mathbf{v}, j \rangle$ , if and only if  $\mathbf{u} < \mathbf{v}$  or  $\mathbf{u} = \mathbf{v}$  and  $i < j$ .

The third step in SIA is to sort  $V$  to give an ordered set  $\mathbf{V}$  that contains the elements of  $V$  in increasing order. For example, the column headed  $\mathbf{V}[i]$  in Table 2 gives  $\mathbf{V}$  for the dataset in Figure 1(a). An examination of Table 1 reveals that the vectors increase as one descends a column and decrease as one goes from left to right along a row. In the implementation of SIA that we describe in section 3.1 below we use a two-dimensional linked list to represent  $V$  as a vector table like the one in Table 1 (see Figure 63). We then use a modified version of merge sort, that exploits the fact that the columns and rows in this vector table are already sorted, to accomplish this third step of the algorithm more rapidly than would be achievable using plain merge sort on the completely unsorted set  $V$ . The worst-case running time of this step of the algorithm is  $O(kn^2 \log_2 n)$ .

### 2.1.4 SIA: Step 4 – Printing out $\mathcal{S}(D)$

If  $\mathbf{A}$  is an ordered set of ordered sets then  $\mathbf{A}[i, j]$  denotes the  $j$ th element of the  $i$ th element of  $\mathbf{A}$ . For example, if  $\mathbf{A} = \langle \langle a, b, c \rangle, \langle d, e \rangle, \langle f \rangle \rangle$  then  $\mathbf{A}[1, 3] = c$ ,  $\mathbf{A}[2, 1] = d$  and  $\mathbf{A}[3, 1] = f$ . As pointed out above, the column headed  $\mathbf{V}[i]$  in Table 2 gives  $\mathbf{V}$  for the dataset in Figure 1(a). For each of these ordered pairs,  $\mathbf{V}[i]$ , the datapoint  $\mathbf{D}[\mathbf{V}[i, 2]]$  is printed next to it in the third column in Table 2. For example,  $\mathbf{V}[1] = \langle \langle 0, 1 \rangle, 3 \rangle$  in



Table 2, so  $\mathbf{V}[1, 2] = 3$  and  $\mathbf{D}[\mathbf{V}[1, 2]] = \langle 2, 1 \rangle$ , the third datapoint in the ordered set  $\mathbf{D}$  for the dataset shown in Figure 1(a).

As indicated on the right-hand side of the third column in Table 2, the MTP for a vector  $\mathbf{v}$  is the set of consecutive datapoints  $\mathbf{D}[\mathbf{V}[i, 2]]$  in the third column that corresponds to the set of consecutive ordered pairs  $\mathbf{V}[i]$  in the second column for which  $\mathbf{V}[i, 1] = \mathbf{v}$ . The complete set  $\mathcal{S}(D)$  as defined in Eq.9 can be printed out using the algorithm in Figure 7. In our pseudocode, block structure is indicated by indentation and the symbol ‘ $\leftarrow$ ’ indicates assignment. Figure 8 shows the output generated by this algorithm for the dataset in Figure 1(a).

SIA discovers the set  $\mathcal{P}'(D)$  of non-empty MTPs defined in Eq.8 and from Table 2 it can easily be seen that SIA accomplishes this simply by sorting the set  $V$  defined in Eq.22. It is clear from Table 1 that, for a dataset of size  $n$ , the number of elements in  $V$  is  $\frac{n(n-1)}{2}$ . Therefore, if we use  $P$  to denote an MTP in  $\mathcal{P}'(D)$ ,

$$\sum_{P \in \mathcal{P}'(D)} |P| = \frac{n(n-1)}{2}.$$

Therefore the total number of vectors that have to be printed when  $\mathcal{S}(D)$  is printed is  $\frac{n(n-1)}{2}$  plus one vector for each MTP in  $\mathcal{P}'(D)$ . Since  $|\mathcal{P}'(D)| \leq \frac{n(n-1)}{2}$ , the total number of vectors to be printed out is certainly less than or equal to  $n(n-1)$ . Therefore, for a  $k$ -dimensional dataset containing  $n$  datapoints,  $\mathcal{S}(D)$  can be printed out in a worst-case running time of  $O(kn^2)$ .

## 2.2 The SIATEC algorithm

When given a multidimensional dataset,  $D$ , as input, SIATEC computes  $\mathcal{J}'(D)$  as defined in Eq.12 above. For a  $k$ -dimensional dataset containing  $n$  datapoints, the worst-case running time of SIATEC is  $O(kn^3)$  and its worst-case space complexity is  $O(kn^2)$ . The algorithm consists of the following seven steps.

### 2.2.1 SIATEC: Step 1 – Sorting the dataset

This is exactly the same as Step 1 of SIA as described in section 2.1.1 above.

### 2.2.2 SIATEC: Step 2 – Computing $\mathbf{W}$

The second step in SIATEC is to compute the ordered set of ordered sets

$$\mathbf{W} = \langle \langle \mathbf{W}[1, 1], \dots, \mathbf{W}[1, |\mathbf{D}|] \rangle, \dots, \langle \mathbf{W}[|\mathbf{D}|, 1], \dots, \mathbf{W}[|\mathbf{D}|, |\mathbf{D}|] \rangle \rangle$$

where

$$\mathbf{W}[i, j] = \mathbf{D}[j] - \mathbf{D}[i]. \quad (23)$$

$\mathbf{W}$  can be visualized as a vector table like Table 3 (which shows  $\mathbf{W}$  for the dataset in Figure 1(a)). Note that each element in  $\mathbf{W}$  is simply a vector whereas each element in the vector table computed in Step 2 of SIA is an ordered pair (see Table 1).  $\mathbf{W}$  is used in Step 7 of SIATEC to compute the set of translators for each MTP.

Computing  $\mathbf{W}$  for a  $k$ -dimensional dataset of size  $n$  involves computing  $n^2$  vector subtractions. Each of these vector subtractions involves carrying out  $k$  scalar subtractions so the overall worst-case running time of this step is  $O(kn^2)$ .

### 2.2.3 SIATEC: Step 3 – Computing $V$

The third step of SIATEC is to compute the set  $V$  as defined in Eq.22. This is the same set as that computed in Step 2 of SIA. In the example implementation of SIATEC described in section 3.2 below,  $V$  is constructed from  $\mathbf{W}$  so that the inter-datapoint vectors are only computed once. This step can therefore be carried out in a worst-case time complexity of  $O(n^2)$  and not  $O(kn^2)$ . Table 1 shows  $V$  for the dataset in Figure 1(a).

#### 2.2.4 SIATEC: Step 4 – Sorting $V$ to produce $\mathbf{V}$

This step is exactly the same as Step 3 of SIA. The second column of Table 2 shows  $\mathbf{V}$  for the dataset in Figure 1(a).

#### 2.2.5 SIATEC: Step 5 – ‘Vectorizing’ the MTPs

$\mathbf{V}$  is effectively a sorted representation of  $\mathcal{S}(D)$  (Eq.9) (see Step 4 of SIA and Table 2). The purpose of SIATEC is to compute  $\mathcal{J}'(D)$  (Eq.12) which is the set that only contains every TEC that is the TEC of an MTP in  $\mathcal{P}'(D)$  (Eq.8).  $\mathcal{P}'(D)$  can be obtained from  $\mathbf{V}$  but it is possible for two or more MTPs in  $\mathcal{P}'(D)$  to be translationally equivalent. For example, the MTPs in the dataset in Figure 1(a) for the vectors  $\langle 0, 2 \rangle$ ,  $\langle 1, -1 \rangle$  and  $\langle 1, 1 \rangle$  are translationally equivalent (see Table 2 and Figure 1(c), (e) and (g)). If two patterns are translationally equivalent then they are members of the same TEC. Therefore, if we naïvely compute the TEC of each MTP in  $\mathcal{P}'(D)$ , we run the risk of computing the same TEC more than once which is inefficient. We therefore partition  $\mathcal{P}'(D)$  into translational equivalence classes and then select just one MTP from each of these classes, discarding the others.

If  $P$  is a pattern then let  $SORT(P)$  be the function that returns the ordered set that only contains all the datapoints in  $P$  sorted into increasing order. If  $\mathbf{P}$  is an ordered set of datapoints then let  $VEC(\mathbf{P})$  be the function that returns the ordered set of vectors

$$VEC(\mathbf{P}) = \langle \mathbf{P}[2] - \mathbf{P}[1], \mathbf{P}[3] - \mathbf{P}[2], \dots, \mathbf{P}[|P|] - \mathbf{P}[|P| - 1] \rangle. \quad (24)$$

If  $P_1$  and  $P_2$  are two patterns in a dataset, then

$$VEC(SORT(P_1)) = VEC(SORT(P_2)) \iff P_1 \equiv_{\tau} P_2. \quad (25)$$

We say that  $VEC(SORT(P))$  is the *vectorized representation* of the pattern  $P$ . In the ordered set  $\mathbf{V}$  computed in Step 4 of SIATEC, each MTP,  $P$ , is represented in its sorted

form as  $SORT(P) = \mathbf{P}$  (see Table 2). Therefore, if we want to use Eq.25 to partition  $\mathcal{P}'(D)$  we first have to compute  $VEC(\mathbf{P})$  for each of the sorted MTPs,  $\mathbf{P}$ , in  $\mathbf{V}$ . Step 5 of SIATEC is therefore to compute

$$X = \{\langle i, VEC(SORT(P)) \rangle \mid \langle \mathbf{v}, P \rangle \in \mathcal{S}(D) \wedge \mathbf{V}[i, 1] = \mathbf{v} \wedge (i = 1 \vee \mathbf{V}[i-1, 1] \neq \mathbf{v})\}. \quad (26)$$

If  $\mathbf{V}[i]$  and  $\mathbf{V}[j]$  are two distinct elements of  $\mathbf{V}$  and  $\mathbf{V}[i] < \mathbf{V}[j]$  but  $\mathbf{V}[i, 1] = \mathbf{V}[j, 1]$  (i.e., the vectors in  $\mathbf{V}[i]$  and  $\mathbf{V}[j]$  are the same) then  $\mathbf{V}[i, 2] < \mathbf{V}[j, 2]$  which implies that  $\mathbf{D}[\mathbf{V}[i, 2]] < \mathbf{D}[\mathbf{V}[j, 2]]$ . This means that the datapoints within each MTP in the  $\mathbf{V}$  representation of  $\mathcal{S}(D)$  are sorted in increasing order, as can be seen in the output of SIA (Figure 8) generated by the algorithm in Figure 7.

$X$  can be efficiently computed directly from  $\mathbf{V}$  and  $\mathbf{D}$  using the algorithm in Figure 9 which exploits the fact that the MTPs in  $\mathbf{V}$  are already sorted. In Figure 9, the set  $X$  is actually represented as an ordered set  $\mathbf{X}$ . When the algorithm in Figure 9 has terminated, the ordered set  $\mathbf{X}$  only contains all the elements of  $X$  (with no duplicates). In Figure 9,  $\langle \rangle$  denotes the empty ordered set.

Figure 10 shows the state of  $\mathbf{X}$  for the dataset in Figure 1(a) at the termination of Step 5 of SIATEC. For a  $k$ -dimensional dataset of size  $n$ , the worst-case running time of the algorithm in Figure 9 is  $O(kn^2)$ .

### 2.2.6 SIATEC: Step 6 – Sorting $\mathbf{X}$

Let  $\mathbf{Q}_1$  and  $\mathbf{Q}_2$  be any two ordered sets in which each element is a  $k$ -dimensional vector. We define that  $\mathbf{Q}_1$  is *less than*  $\mathbf{Q}_2$ , denoted by  $\mathbf{Q}_1 < \mathbf{Q}_2$  if and only if one of the following two conditions is satisfied:

1.  $|\mathbf{Q}_1| < |\mathbf{Q}_2|$ .
2.  $|\mathbf{Q}_1| = |\mathbf{Q}_2|$  and there exists an integer  $1 \leq i \leq |\mathbf{Q}_1|$  such that  $\mathbf{Q}_1[i] < \mathbf{Q}_2[i]$  and  $\mathbf{Q}_1[j] = \mathbf{Q}_2[j]$  for all  $1 \leq j < i$ .

(See page 12 for a definition of the expression  $\mathbf{u} < \mathbf{v}$  when  $\mathbf{u}$  and  $\mathbf{v}$  are vectors.) In Step 6 of SIATEC, the ordered set  $\mathbf{X}$  generated by the algorithm in Figure 9 is sorted to produce the ordered set  $\mathbf{Y}$  which satisfies the following two conditions:

1.  $\mathbf{Y}$  only contains all the elements of  $\mathbf{X}$ .
2. If  $\mathbf{Y}[i]$  and  $\mathbf{Y}[j]$  are any two distinct elements of  $\mathbf{Y}$  then  $i < j$  if and only if

$$\mathbf{Y}[i, 2] < \mathbf{Y}[j, 2] \vee (\mathbf{Y}[i, 2] = \mathbf{Y}[j, 2] \wedge \mathbf{Y}[i, 1] < \mathbf{Y}[j, 1]).$$

Figure 11 shows  $\mathbf{Y}$  for the dataset in Figure 1(a). For a  $k$ -dimensional dataset of size  $n$ , this step of the algorithm can be accomplished in a worst-case running time of  $O(kn^2 \log_2 n)$  using merge sort.

We know that

$$MTP(\mathbf{V}[\mathbf{Y}[i, 1], 1], D) \equiv_{\tau} MTP(\mathbf{V}[\mathbf{Y}[j, 1], 1], D) \iff \mathbf{Y}[i, 2] = \mathbf{Y}[j, 2].$$

So Figure 11 tells us, for example, that the MTPs for the vectors  $\mathbf{V}[3, 1] = \langle 0, 2 \rangle$ ,  $\mathbf{V}[6, 1] = \langle 1, -1 \rangle$  and  $\mathbf{V}[11, 1] = \langle 1, 1 \rangle$  are translationally equivalent since the vectorized representation of each of these patterns is  $\langle \langle 1, 0 \rangle \rangle$ . This implies that we only have to compute the TEC of one of these patterns and the other two can be disregarded.

### 2.2.7 SIATEC: Step 7 – Printing out $\mathcal{J}'(D)$

The final step of SIATEC is to print out  $\mathcal{J}'(D)$ . This can be done using the algorithm in Figure 12. Recall that each TEC in  $\mathcal{J}'(D)$  is represented as an ordered pair  $\langle P, T(P, D) \setminus \mathbf{0} \rangle$  where  $P$  is an MTP and  $T(P, D)$  is the set of translators for  $P$  in the dataset  $D$  (see Eq.13 and discussion on page 16 above). In Figure 12, each MTP is printed out using the algorithm PRINT\_PATTERN called in line 14. This algorithm is given in Figure 13.

The set of translators for each TEC is printed out using the algorithm PRINT\_SET\_OF\_TRANSLATORS called in line 16 of Figure 12. This algorithm, which is given

in Figure 14, exploits the fact that

$$T(\{\mathbf{D}[i]\}, D) = \bigcup_{j=1}^{|D|} \{\mathbf{W}[i, j]\}.$$

That is, the set of translators for a datapoint  $\mathbf{D}[i]$  is the set that only contains every vector that occurs in the  $i$ th column in the vector table computed in Step 2 of SIATEC (see Table 3). In Figure 12, each MTP is represented as a set of indices,  $\mathbf{I}$  such that the pattern represented by  $\mathbf{I}$  is simply  $\{\mathbf{D}[i] \mid i \in \mathbf{I}\}$ . The set of translators for the pattern represented by  $\mathbf{I}$  is therefore

$$\bigcap_{i \in \mathbf{I}} T(\{\mathbf{D}[i]\}, D) = \bigcap_{i \in \mathbf{I}} \left( \bigcup_{j=1}^{|D|} \{\mathbf{W}[i, j]\} \right). \quad (27)$$

In other words, the set of translators for a pattern is the set that only contains those vectors that occur in *all* the columns in the vector table corresponding to the datapoints in the pattern. For example, if  $D$  is the dataset in Figure 1(a), the set of translators for the pattern  $\{\mathbf{a}, \mathbf{c}\} = \{\langle 1, 1 \rangle, \langle 2, 1 \rangle\}$  is the set that only contains all the vectors that occur in *both* the first and third columns in Table 3:

$$\begin{aligned} T(\{\langle 1, 1 \rangle, \langle 2, 1 \rangle\}, D) &= \{\langle 0, 0 \rangle, \langle 0, 2 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle\} \\ &\quad \cap \{\langle -1, 0 \rangle, \langle -1, 2 \rangle, \langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 1, 1 \rangle\} \\ &= \{\langle 0, 0 \rangle, \langle 0, 2 \rangle, \langle 1, 1 \rangle\} \end{aligned}$$

The algorithm PRINT\_SET\_OF\_TRANSLATORS is an efficient algorithm for computing the expression on the right-hand side of Eq.27.

Using the algorithms in Figures 12, 13 and 14, Step 7 can be accomplished in a worst-case running time of  $O(kn^3)$  for a  $k$ -dimensional dataset of size  $n$ . Figure 15 shows the output generated by the algorithm in Figure 12 for the dataset in Figure 1(a).

## 2.3 The SIAME algorithm

When given a  $k$ -dimensional query pattern,  $P$ , and a  $k$ -dimensional dataset,  $D$ , as input, SIAME computes  $\mathcal{M}'(P, D)$  as defined in Eq.18 above. For a  $k$ -dimensional query pattern containing  $m$  datapoints and a  $k$ -dimensional dataset containing  $n$  datapoints, the worst-case running time of SIAME is  $O(kmn \log_2(mn))$  and its worst-case space complexity is  $O(kmn)$ . The algorithm consists of the following 5 steps.

### 2.3.1 SIAME: Step 1 – Computing the set of inter-datapoint vectors

The first step in SIAME is very similar to Step 2 of SIA (see section 2.1.2): given a query pattern  $P$  and a dataset  $D$ , the set

$$V_{\text{SIAME}} = \{\langle \mathbf{d} - \mathbf{p}, \mathbf{p} \rangle \mid \mathbf{d} \in D \wedge \mathbf{p} \in P\} \quad (28)$$

is computed. For example, for the query pattern in Figure 3(a) and the dataset in Figure 3(b),  $V_{\text{SIAME}}$  would contain all and only the elements in Table 4. Note that each element in  $V_{\text{SIAME}}$  is an ordered pair of vectors. In an implementation (such as the one described in section 3.4 below) the second vector in each of these ordered pairs would probably be represented by a pointer to the datapoint in the representation of  $P$  or by an index to an element of an array storing  $P$ .

For a  $k$ -dimensional pattern of size  $m$  and a  $k$ -dimensional dataset of size  $n$ , this step can be accomplished in a worst-case running time of  $O(kmn)$  using  $O(kmn)$  space.

### 2.3.2 SIAME: Step 2 – Sorting the inter-datapoint vectors

In our description of Step 6 of SIATEC in section 2.2.6 above we defined the concept of ‘less than’ when applied to ordered sets of vectors. The second step in SIAME is similar to Step 3 of SIA (see section 2.1.3): the set  $V_{\text{SIAME}}$  computed in Step 1 of SIAME is sorted to give an ordered set  $\mathbf{V}_{\text{SIAME}}$  that contains the elements of  $V_{\text{SIAME}}$  sorted into increasing order. Again, as can be seen in Table 4, each column in the table is already sorted. This

fact can be used to advantage if  $V_{\text{SIAME}}$  is represented as a two-dimensional linked list and merge sort is used to perform the sort (see section 3.4 below). This step of the algorithm can be accomplished in a worst-case running time of  $O(kmn \log_2(mn))$ . Alternatively, if hashing is used, the step can be accomplished in an expected time of  $O(kmn)$ . Figure 16 shows  $V_{\text{SIAME}}$  for the query pattern in Figure 3(a) and the dataset in Figure 3(b).

### 2.3.3 SIAME: Step 3 – Computing the size of each set in $\mathcal{M}(P, D)$

It is very useful if the matches found by SIAME are listed so that the best matches occur first. To achieve this, it is necessary to compute the size of each element of  $\mathcal{M}(P, D)$ . Therefore, in this third step of SIAME, the set

$$N = \{ \langle |M|, i \rangle \mid \langle \mathbf{v}, M \rangle \in \mathcal{M}'(P, D) \wedge \mathbf{V}_{\text{SIAME}}[i, 1] = \mathbf{v} \wedge (i = 1 \vee \mathbf{V}_{\text{SIAME}}[i - 1, 1] \neq \mathbf{v}) \} \quad (29)$$

is computed. This can be done directly from  $V_{\text{SIAME}}$  using the algorithm in Figure 17 which returns an ordered set,  $\mathbf{N}$ , that only contains every element of  $N$  exactly once. Figure 18 shows  $\mathbf{N}$  for the pattern in Figure 3(a) and the dataset in Figure 3(b). The worst-case running time of the algorithm in Figure 17 is  $O(kmn)$ .

### 2.3.4 SIAME: Step 4 – Sorting $\mathbf{N}$

The fourth step of SIAME is to sort the vectors in  $\mathbf{N}$  to produce a new ordered set,  $\mathbf{N}'$  that only contains all the vectors in  $\mathbf{N}$  sorted into *decreasing* order. This can be achieved in a worst-case running time of  $O(mn \log_2(mn))$ . Note that this step is not dependent on the cardinality of the datapoints in the pattern and dataset. Figure 19 shows  $\mathbf{N}'$  for the pattern in Figure 3(a) and the dataset in Figure 3(b).

### 2.3.5 SIAME: Step 5 – Computing $\mathcal{M}'(P, D)$

Finally,  $\mathcal{M}'(P, D)$ , expressed as an ordered set,  $\mathbf{M}$ , in which the best matches occur first, can be computed directly from  $\mathbf{N}'$  and  $V_{\text{SIAME}}$  using the algorithm shown in Figure 20.



The worst-case running time of this algorithm is  $O(kmn)$ . Figure 21 shows  $\mathbf{M}$  for the pattern in Figure 3(a) and the dataset in Figure 3(b).

## 2.4 The COSIATEC algorithm

When given a multidimensional dataset  $D$  as input, COSIATEC uses SIATEC to compute a compressed representation of  $D$  in the form of an ordered set of TECs satisfying the conditions described on page 19 above.

Figure 22 shows a simple (but inefficient) version of the COSIATEC algorithm. The ordered set variable  $\mathbf{C}$  is used to store the compressed representation and it is initialised to equal the empty ordered set in line 1. The variable  $D'$  is used to hold the current value of  $D_k$  as defined on page 19 above. This variable is initialised to equal  $D$  in line 2.

On each iteration of the ‘while’ loop (lines 3–15), SIATEC is first used to compute  $\mathcal{T}'(D')$  (line 4). Then, in lines 5–13, an element  $E_{\text{best}}$  of  $\mathcal{E}_{\text{best}}(D')$  (see page 19) is computed which is appended to  $\mathbf{C}$  (line 14). In line 15,  $D'$  has all datapoints removed from it that are elements of patterns in  $E_{\text{best}}$ . The while loop terminates when  $D'$  is empty (line 3).

In line 4, the function  $\mathbf{T}'(D')$  uses SIATEC to compute an ordered set containing the elements of  $\mathcal{T}'(D')$  arranged in some arbitrary order. The functions  $COV(E)$  and  $CR(E)$  are as defined in Eqs.19 and 20 above.

## 3 Example implementations of the algorithms

In this section, efficient implementations of the SIA, SIATEC, SIAME and COSIATEC algorithms will be described.

### 3.1 Example implementation of SIA

In this section we describe an efficient implementation of the SIA algorithm described in section 2.1 above.

### 3.1.1 The SIA procedure

Figure 24 gives pseudocode for an efficient implementation of SIA. In this algorithm, the dataset to be analysed is stored in a file whose name is given in the parameter `DFN`. The output of the algorithm is written to a file whose name is given in the parameter `OFN`.

The third parameter to the algorithm, `SD`, is either `NULL` or a string of 0s and 1s indicating the orthogonal projection of the dataset to be analysed. For example, if the dataset stored in the file whose name is `DFN` is a 5-dimensional dataset but the user only wishes to analyse the 2-dimensional projection of this dataset onto the plane defined by the first and third dimensions, then `SD` would be set to "10100". If `SD` is `NULL`, all the dimensions are considered.

In line 3 of the SIA implementation in Figure 24, an attempt is made to open the file whose name is `DFN`. The function `OPEN_FILE` returns `NULL` and the program exits (line 4) if this attempt is unsuccessful.

If the file `DFN` exists, then the dataset is read into memory in line 5 using the `READ_VECTOR_SET` function which is defined in Figure 25 and discussed further in section 3.1.2 below. The file containing the input dataset is then closed in line 6.

In line 7, the dataset is sorted using the `SORT_DATASET` algorithm which is defined in Figure 26 and discussed further in section 3.1.3 below.

If the `SD` parameter is used to select an orthogonal projection of the dataset, then it is possible for two or more datapoints in the dataset stored in `DF` to be projected onto the same datapoint in the chosen projection of this dataset. If this happens, then `D` may contain duplicate datapoints. These are removed in line 8 of the SIA implementation (see Figure 24) using the `SETIFY_DATASET` algorithm which is defined in Figure 28 and discussed further in section 3.1.4 below.

This accomplishes Step 1 of the SIA algorithm as described in section 2.1.1 above.

The function `SIA_COMPUTE_VECTORS`, defined in Figure 29 and called in line 9 of the SIA implementation in Figure 24, accomplishes Step 2 of the SIA algorithm as described

in section 2.1.2 above. `SIA_COMPUTE_VECTORS` is discussed further in section 3.1.5 below.

The function `SIA_SORT_VECTORS`, defined in Figure 30 and called in line 10 of the `SIA` implementation in Figure 24, accomplishes Step 3 of the `SIA` algorithm as described in section 2.1.3 above. `SIA_SORT_VECTORS` is discussed further in section 3.1.6 below.

Finally, Step 4 of the `SIA` algorithm, described in section 2.1.4 above, is carried out using the `PRINT_VECTOR_MTP_PAIRS` procedure which is defined in Figure 32 and called in line 11 of the `SIA` implementation in Figure 24. `PRINT_VECTOR_MTP_PAIRS` is an implementation of the algorithm in Figure 7. It is discussed further in section 3.1.7 below.

For a  $k$ -dimensional dataset containing  $n$  datapoints, the worst-case running time of this implementation of the `SIA` algorithm is  $O(kn^2 \log_2 n)$  (this is the running time of `SIA_SORT_VECTORS` called in line 10 of the implementation). The worst-case space complexity is  $O(kn^2)$ .

### 3.1.2 The `READ_VECTOR_SET` function

Figure 25 gives pseudocode for the `READ_VECTOR_SET` function which is called in line 5 of the `SIA` implementation given in Figure 24. This algorithm reads a list of vectors from a file and stores the list in memory as a linked list, returning a pointer (`S` in Figure 25) to the head of this list.

`READ_VECTOR_SET` takes three parameters: `F` is a text file containing the list of vectors to be read; `DIR` determines the type of linked list used to store the vectors (see below); and `SD` is either `NULL` or a string of 0s and 1s indicating a specific orthogonal projection of the vector set to be read (see section 3.1.1 above).

It is assumed that the collection of vectors to be read from the file `F` is represented as a list with one vector per line, the list being terminated by an empty line. Each vector is represented as a list of numerical values, each one followed by a single space character and terminated by an end-of-line character. For example, Figure 52 shows how the ordered vector set

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

would be represented in the input file F. In Figure 52, ‘ ’ represents a space character and ‘\_’ represents an end-of-line character.

The linked list constructed by `READ_VECTOR_SET` uses two types of node: `NUMBER_NODES` and `VECTOR_NODES`.

`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` (see definition in Figure 23). 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` can be represented diagrammatically as a rectangular box divided into two cells (see Figure 53). 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 53 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 23). A `VECTOR_NODE` can be represented diagrammatically as a rectangular box divided into three cells (see Figure 54). The left-hand cell represents 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 54. The `down` field is used to point to the next `VECTOR_NODE` in a *down-directed list* such as the one shown in Figure 55. The linked list in Figure 54 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 the vector set  $\{ \langle 1, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 3 \rangle \}$ . The linked list in Figure 55 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` (defined in Figure 26 and discussed in section 3.1.3) and `SIA_SORT_VECTORS` (defined in Figure 30 and discussed in section 3.1.6)).

If the `DIR` parameter of the `READ_VECTOR_SET` function (Figure 25) has the value `DOWN`, the vector set read by the algorithm is stored as a down-directed list of `VECTOR_NODES`, otherwise the vector set is stored as a right-directed list. If `F` contains the data in Figure 52, then Figure 56 shows the linked list returned by the call

```
READ_VECTOR_SET(F,DOWN,"101")
```

and Figure 57 shows the linked list returned by

```
READ_VECTOR_SET(F,RIGHT,NULL)
```

In our pseudocode, the symbol ‘ $\uparrow$ ’ denotes pointer dereferencing: that is, the expression ‘ $x\uparrow y$ ’ denotes the field called `y` in the data structure pointed to by `x`.

The function `AT_END_OF_LINE(F)` used in line 5 of `READ_VECTOR_SET` (see Figure 25) returns `TRUE` if the next character to be read from `F` is an end-of-line character or an end-of-file character. The function is used to determine whether or not all the vectors in a list have been read.

The function `READ_VECTOR` called in line 6 of `READ_VECTOR_SET` reads a vector from a file and returns a linked list of `NUMBER_NODES` representing the vector (as in Figure 53).

The function `SELECT_DIMENSIONS_IN_VECTOR(v,SD)` called in line 8 of `READ_VECTOR_SET` uses `SD` to remove those elements of `v` that are not required in the chosen orthogonal projection of the vector set.

The function `MAKE_NEW_VECTOR_NODE` called in lines 10, 15 and 20 of `READ_VECTOR_SET` creates a new `VECTOR_NODE` and sets all its fields to `NULL`.

### 3.1.3 The `SORT_DATASET` function

Figure 26 gives pseudocode for the `SORT_DATASET` algorithm called in line 7 of the `SIA` algorithm implementation given in Figure 24. In Figure 24, the call to `READ_VECTOR_SET`

in line 5 stores the orthogonal projection of the dataset to be analysed as an unsorted, down-directed list of `VECTOR_NODES`. For example, in Figure 24, if `DFN` is the name of a file containing the data in Figure 58 then the call to `READ_VECTOR_SET` in line 5 would return the linked list in Figure 59.

`SORT_DATASET` is a version of merge sort that converts the unsorted down-directed list of `VECTOR_NODES` generated by the call to `READ_VECTOR_SET` in line 5 of `SIA` into a sorted, right-directed list. On the first iteration of the outer `while` loop (lines 2–21 in Figure 26), `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 59 shows the unsorted, down-directed list generated by line 5 of `SIA` (see Figure 24) for the data in Figure 58 and Figure 60 shows the state of the linked list `D` after one iteration of the outer `while` loop of `SORT_DATASET` has been completed on the dataset list shown in Figure 59. 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. Figure 61 shows the right-directed list produced by `SORT_DATASET` from the down-directed list shown in Figure 59.

The merging process is carried out by the `MERGE_DATASET_ROWS` algorithm which is called in line 13 of `SORT_DATASET` and defined in Figure 27.

In lines 4 and 13 of the `MERGE_DATASET_ROWS` algorithm in Figure 27, the function `VECTOR_LESS_THAN( $v_1, v_2$ )` is used to compare two vectors represented as `NUMBER_NODE` lists headed by the pointers  $v_1$  and  $v_2$ . The function `VECTOR_LESS_THAN` returns `TRUE` if and only if the vector represented by the `NUMBER_NODE` list headed by  $v_1$  is less than that represented by the list headed by  $v_2$ .

### 3.1.4 The `SETIFY_DATASET` function

Figure 28 gives pseudocode for the `SETIFY_DATASET` algorithm called in line 8 of the `SIA` implementation in Figure 24. `SETIFY_DATASET` removes duplicate datapoints from the sorted right-directed list generated by `SORT_DATASET`. For example, if `SETIFY_DATASET`

is given the linked list shown in Figure 61 as input, it returns the linked list shown in Figure 62. The call to `SORT_DATASET` in line 7 of the `SIA` implementation and the call to `SETIFY_DATASET` in line 8 together accomplish Step 1 of the `SIA` algorithm described in section 2.1 above.

The `VECTOR_EQUAL` function used in line 5 of `SETIFY_DATASET` in Figure 28 takes two `NUMBER_NODE` pointer arguments, each heading a list of `NUMBER_NODES` representing a vector, and returns `TRUE` if and only if the two vectors are equal.

The `DISPOSE_OF_VECTOR_NODE` function used in line 9 of `SETIFY_DATASET` destroys the linked multi-list of `VECTOR_NODES` headed by its argument and deallocates the memory used by this list.

### 3.1.5 The `SIA_COMPUTE_VECTORS` function

The function `SIA_COMPUTE_VECTORS`, defined in Figure 29 and called in line 9 of `SIA` (see Figure 24), accomplishes Step 2 of the `SIA` algorithm as described in section 2.1.2 above.

Figure 63 shows the data structure that results after `SIA_COMPUTE_VECTORS` has executed when the `SIA` implementation in Figure 24 is carried out on the dataset shown in Figure 1(a). The resulting data structure is a representation of the vector table shown in Table 1.

The `VECTOR_MINUS( $v_1, v_2$ )` function called in line 14 of `SIA_COMPUTE_VECTORS` (see Figure 29) takes two `NUMBER_NODE` pointer arguments, each pointing to a linked-list representing a vector, and subtracts the vector pointed to by  $v_2$  from the vector pointed to by  $v_1$ , returning a pointer to the linked list representing the result.

### 3.1.6 The `SIA_SORT_VECTORS` function

The function `SIA_SORT_VECTORS`, defined in Figure 30 and called in line 10 of the `SIA` implementation in Figure 24, accomplishes Step 3 of the `SIA` algorithm as described in section 2.1.3 above.

The call to `SIA_SORT_VECTORS` in line 10 of the `SIA` implementation is the most expensive step in the program, requiring  $O(kn^2 \log_2 n)$  time in the worst case.

`SIA_SORT_VECTORS` takes the data structure headed by `V` returned by `SIA_COMPUTE_VECTORS` (see Figure 63) and uses a modified version of merge sort to generate a single down-directed list representing the ordered set `V` defined in section 2.1.3 above.

As can be seen in Figure 63, the structure headed by `V` consists of a right-directed list of `VECTOR_NODES` from each of which ‘hangs’ a down-directed list of nodes. Each of these ‘hanging’ down-directed lists represents a column in Table 1. Within each of these down-directed lists the vectors are already sorted into increasing order. `SIA_SORT_VECTORS` exploits this fact to accomplish its task more efficiently.

In `SIA_SORT_VECTORS`, the merging process is carried out using the `SIA_MERGE_VECTOR_COLUMNS` function which is called in line 13 and defined in Figure 31.

Figure 64 shows the data structure that results after the call to `SIA_SORT_VECTORS` in line 10 of the implementation of `SIA` in Figure 24 has executed when this implementation is run on the dataset in Figure 1(a). This data structure represents the second column in Table 2.

### 3.1.7 The `PRINT_VECTOR_MTP_PAIRS` function

Step 4 of the `SIA` algorithm, described in section 2.1.4 above, is carried out in this implementation using the `PRINT_VECTOR_MTP_PAIRS` algorithm which is defined in Figure 32 and called in line 11 of the `SIA` procedure in Figure 24.

`PRINT_VECTOR_MTP_PAIRS` is an implementation of the algorithm in Figure 7 except that the format of the output is simpler than that produced by the algorithm in Figure 7.

In the output of `PRINT_VECTOR_MTP_PAIRS`, each  $\langle \text{vector}, \text{MTP} \rangle$  pair is represented as a pair of consecutive vector lists in the same format as that used for input to `SIA` (see Figure 52). That is, for each  $\langle \text{vector}, \text{MTP} \rangle$  pair, the vector is first printed out on a single



line, then there is an empty line, then the MTP is printed out as a list of vectors, each vector being printed on a separate line, and the MTP being terminated by an empty line. The end of the file is also signalled by an empty line. This means that every odd-numbered vector list in the output file represents the vector of a  $\langle \text{vector}, \text{MTP} \rangle$  pair and every even-numbered vector list represents the MTP in such a pair.

Figure 65 shows the output generated by the `PRINT_VECTOR_MTP_PAIRS` algorithm for the dataset in Figure 1(a). This provides the same information as Figure 8 except that it is presented in a different (and less complicated) format.

In lines 8, 10 and 13 of the `PRINT_VECTOR_MTP_PAIRS` procedure in Figure 32, `PRINT_VECTOR` is used to print the vectors. `PRINT_VECTOR` takes two arguments: the first is a pointer to a `NUMBER_NODE` list representing a vector and the second is the file to which the vector is to be written.

`PRINT_VECTOR_MTP_PAIRS` also uses the procedure `PRINT_NEW_LINE(F)` (lines 9, 15 and 17) to print an end-of-line character to the file stream `F`.

## 3.2 Example implementation of SIATEC

In this section we describe an efficient implementation of the SIATEC algorithm described in section 2.2 above.

### 3.2.1 The SIATEC procedure

Figure 33 gives pseudocode for an efficient implementation of SIATEC.

Like the SIA implementation in Figure 24, the SIATEC procedure in Figure 33 takes three arguments: `DFN` is the name of the file containing the dataset to be analysed; `OFN` is the name of the file to which the output is written; and `SD` is a string of 1s and 0s indicating the orthogonal projection of the dataset to be analysed (see discussion in section 3.1.1 above).

If the file whose name is `DFN` exists, then the call to `READ_VECTOR_SET` in line 7 of

Figure 33 reads the dataset into memory and stores it in an unsorted, down-directed list of `VECTOR_NODES`. This is exactly the same as the task carried out in line 5 of the `SIA` implementation in Figure 24 (see discussion of `READ_VECTOR_SET` in section 3.1.2 above).

If the dataset is empty (line 9, Figure 33), then an empty output file is created and the algorithm terminates.

If the dataset is not empty, then it is sorted in line 13 using the `SORT_DATASET` function and ‘setified’ in line 14 using the `SETIFY_DATASET` function. These functions are defined in Figures 26 and 28 and were described above in sections 3.1.3 and 3.1.4.

This accomplishes Step 1 of the `SIATEC` algorithm as described in section 2.2.1 above.

The `PRINT_SET_OF_TRANSLATORS` algorithm defined in Figure 14 and used in Step 7 of the `SIATEC` algorithm described in section 2.2.7 above, uses a knowledge of the size of the dataset (stored in the variable  $n$ ) to increase efficiency (see line 2 in Figure 14). Therefore, in line 15 of the implementation of `SIATEC` given in Figure 33, the size of the dataset is computed using a function `SIZE_OF_DATASET` which simply scans the sorted, right-directed list of `VECTOR_NODES` generated by `SETIFY_DATASET` in line 14 and counts the number of datapoints in the list.

If a dataset  $D$  contains only one point,  $D = \{d\}$ , then the only TEC in  $D$  is  $\{\{d\}\}$ . If the dataset given as input to the procedure in Figure 33 contains only one datapoint, then `D↑right = NULL` in line 16 and an output file is generated containing the single datapoint in the dataset.

If the dataset contains more than one datapoint, lines 24–29 in Figure 33 are executed.

The function `COMPUTE_VECTORS` called in line 24 of Figure 33 and defined in Figure 34 accomplishes Step 2 of the `SIATEC` algorithm described in section 2.2.2 above. The `COMPUTE_VECTORS` function is discussed further in section 3.2.2 below.

The function `CONSTRUCT_VECTOR_TABLE` called in line 25 of Figure 33 and defined in Figure 35 accomplishes Step 3 of the `SIATEC` algorithm described in section 2.2.3 above. It is discussed further in section 3.2.3 below.

The function `SORT_VECTORS` called in line 26 of Figure 33 and defined in Figure 36 ac-

completes Step 4 of the SIATEC algorithm described in section 2.2.4 above. `VECTORIZE_PATTERNS` is discussed further in section 3.2.4 below.

The function `VECTORIZE_PATTERNS` called in line 27 of Figure 33 and defined in Figure 38 accomplishes Step 5 of the SIATEC algorithm described in section 2.2.5 above. `VECTORIZE_PATTERNS` is an implementation of the algorithm in Figure 9. It is discussed further in section 3.2.5 below.

The function `SORT_PATTERN_VECTOR_SEQUENCES` called in line 28 of Figure 33 and defined in Figure 39 accomplishes Step 6 of the SIATEC algorithm described in section 2.2.6 above. It is discussed further in section 3.2.6 below.

Finally, the `PRINT_TECS` algorithm called in line 29 of Figure 33 and defined in Figure 41 accomplishes Step 7 of the SIATEC algorithm described in section 2.2.7 above. `PRINT_TECS` is an implementation of the algorithm in Figure 12. It is discussed further in section 3.2.7 below.

For a  $k$ -dimensional dataset containing  $n$  datapoints, the worst-case running time of this implementation of the SIATEC algorithm is  $O(kn^3)$ . This is the running time of `PRINT_TECS` which is the most expensive step in the implementation. The worst-case space complexity is  $O(kn^2)$ . This is kept to a minimum by avoiding the need for storing the TECs in memory at any point—`PRINT_TECS` computes the TECs as it prints them out.

### 3.2.2 The `COMPUTE_VECTORS` algorithm

The function `COMPUTE_VECTORS` called in line 24 of Figure 33 and defined in Figure 34 accomplishes Step 2 of the SIATEC algorithm described in section 2.2.2 above.

`COMPUTE_VECTORS` constructs a two-dimensional linked-list structure that represents the ordered set of ordered sets,  $\mathbf{W}$ , defined in Eq.23. Figure 66 shows the data structure that results after `COMPUTE_VECTORS` has executed when the SIATEC algorithm in Figure 33 is run on the dataset in Figure 1(a). The data structure in Figure 66 is a representation of Table 3.

### 3.2.3 The CONSTRUCT\_VECTOR\_TABLE function

The function `CONSTRUCT_VECTOR_TABLE` called in line 25 of Figure 33 and defined in Figure 35 accomplishes Step 3 of the `SIATEC` algorithm described in section 2.2.3 above.

Figure 67 shows the data structures that result after `CONSTRUCT_VECTOR_TABLE` has executed when the `SIATEC` implementation in Figure 33 is run on the dataset in Figure 1(a). That is, `CONSTRUCT_VECTOR_TABLE` converts the data structure in Figure 66 into the data structure in Figure 67. The two-dimensional list headed by `V` in Figure 67 is a representation of Table 1 while the pointer `D` is used to access the multi-list that represents Table 3.

### 3.2.4 The SORT\_VECTORS algorithm

The function `SORT_VECTORS` called in line 26 of Figure 33 is defined in Figure 36 and accomplishes Step 4 of the `SIATEC` algorithm described in section 2.2.4 above.

Like `SIA_SORT_VECTORS` in Figure 30, `SORT_VECTORS` is a version of merge sort. In fact, the only difference between `SORT_VECTORS` and `SIA_SORT_VECTORS` is that in line 13 of `SORT_VECTORS`, the merging process is performed by the `MERGE_VECTOR_COLUMNS` function defined in Figure 37 whereas in line 13 of `SIA_SORT_VECTORS`, this process is performed using the function `SIA_MERGE_VECTOR_COLUMNS` defined in Figure 31.

Similarly, the only difference between `SIA_MERGE_VECTOR_COLUMNS` (Figure 31) and `MERGE_VECTOR_COLUMNS` (Figure 37) occurs in line 8 where the arguments to the `VECTOR_LESS_THAN` function are `b↑right↑vector` and `a↑right↑vector` in `MERGE_VECTOR_COLUMNS` and `b↑vector` and `a↑vector` in `SIA_MERGE_VECTOR_COLUMNS`.

The reason for this difference can be seen by comparing the multi-list headed by `V` in Figure 67 with that headed by `V` in Figure 63. In both cases, the multi-list data structure accessed via `V` represents Table 1. In both cases, each down-directed list of nodes that ‘hangs’ off the `down` field of a node in the right-directed list headed by `V` represents a column in Table 1, that is, the set of inter-datapoint vectors originating on a particular

datapoint. In Figure 63, the **vector** field of each node in these down-directed ‘column’ lists points directly at an inter-datapoint vector. However, in Figure 67, the **vector** field of each of these nodes is empty and instead the **right** field is used to point to the node in the multi-list headed by **D** that holds the required inter-datapoint vector.

This extra level of indirection is necessary in **SIATEC** because the structure of the multi-list representing Table 3 must be preserved as it is used to compute TECs by the **PRINT\_TECs** function (defined in Figure 41 and called in line 29 of the **SIATEC** implementation in Figure 33).

Figure 68 shows the state of the data structures headed by **D** and **V** after **SORT\_VECTORS** has executed when the implementation of **SIATEC** in Figure 33 is run on the dataset in Figure 1(a).

### 3.2.5 The **VECTORIZE\_PATTERNS** algorithm

The function **VECTORIZE\_PATTERNS** called in line 27 of Figure 33 and defined in Figure 38 accomplishes Step 5 of the **SIATEC** algorithm described in section 2.2.5 above. **VECTORIZE\_PATTERNS** is an implementation of the algorithm in Figure 9.

**VECTORIZE\_PATTERNS** uses the data structure accessed by **V** in the **SIATEC** procedure (see Figure 33) to compute a linked-list representation of the ordered set **X** in Figure 9 which is itself an ordered set representation of the set  $X$  defined in Eq.26.

The representation of **X** generated by **VECTORIZE\_PATTERNS** is a linked list of **X\_NODES** headed by the variable **X** in Figure 38. The **X\_NODE** data type is defined in Figure 23. Each **X\_NODE** in the list headed by **X** computed by **VECTORIZE\_PATTERNS** represents one of the ordered pairs  $\langle i, \mathbf{Q} \rangle$  in **X** (see line 10 in Figure 9). **Q** in Figure 9 is modelled in **VECTORIZE\_PATTERNS** as a linked list of **VECTOR\_NODES** which is first headed by the variable **Q** (see, e.g., line 12 in Figure 38) but then stored in the **vec\_seq** field of its **X\_NODE** (line 29, Figure 38). The first element of each  $\langle i, \mathbf{Q} \rangle$  ordered pair in **X** in Figure 9 is represented in an **X\_NODE** by the field **start\_vec** which is used to point to the appropriate **VECTOR\_NODE** in the list headed by **V** (see line 30 in Figure 38). The **size** field of an **X\_NODE** representing an

ordered pair  $\langle i, \mathbf{Q} \rangle$  in  $\mathbf{X}$  is used to store the size of the pattern for which  $\mathbf{Q}$  is the vectorized representation (see line 28 in Figure 38). The `down` and `right` fields of an `X_NODE` are used to construct two different types of linked list. The unsorted down-directed list of `X_NODES` generated by `VECTORIZE_PATTERNS` is converted into a sorted right-directed list by the function `SORT_PATTERN_VECTOR_SEQUENCES` which is called in line 28 of Figure 33 and defined in Figure 39.

An `X_NODE` can be represented diagrammatically as a rectangular box divided into 5 cells as shown in Figure 69. As shown in this figure, the cells represent, from left to right, the `vec_seq`, `size`, `down`, `right` and `start_vec` fields.

The `MAKE_NEW_X_NODE` function called in lines 23 and 26 of `VECTORIZE_PATTERNS` simply creates a new `X_NODE`, sets its `size` field to zero and all the other fields to `NULL`.

Figure 70 shows the state of the data structures headed by `D`, `V` and `X` in the implementation of `SIATEC` in Figure 33 after line 27 has been executed when this implementation is run on the dataset in Figure 1(a).

### 3.2.6 The `SORT_PATTERN_VECTOR_SEQUENCES` algorithm

The function `SORT_PATTERN_VECTOR_SEQUENCES` called in line 28 of the `SIATEC` implementation in Figure 33 and defined in Figure 39 accomplishes Step 6 of the `SIATEC` algorithm described in section 2.2.6 above.

Like `SORT_DATASET` (Figure 26) and `SORT_VECTORS` (Figure 36), `SORT_PATTERN_VECTOR_SEQUENCES` is an implementation of merge sort. The function `VECTORIZE_PATTERNS` called in line 27 of the `SIATEC` implementation in Figure 33 returns an unsorted, down-directed list of `X_NODES` that represents the ordered set  $\mathbf{X}$  computed by the algorithm in Figure 9 (see, for example, Figure 70). The call to `SORT_PATTERN_VECTOR_SEQUENCES` in line 28 of the `SIATEC` implementation (Figure 33) converts this unsorted down-directed list into a sorted, right-directed list of `X_NODES` that represents the ordered set  $\mathbf{Y}$  computed in Step 6 of the `SIATEC` algorithm described in section 2.2.6 above.

In `SORT_PATTERN_VECTOR_SEQUENCES` (Figure 39), the merging process is performed by the function `MERGE_PATTERN_ROWS` called in line 13 and defined in Figure 40. The function `PATTERN_VEC_SEQ_LESS_THAN` called in line 13 of `MERGE_PATTERN_ROWS`, implements the definition of ‘less than’ when applied to ordered sets of vectors defined in section 2.2.6 above.

Figure 71 shows the state of the data structures headed by `D`, `V` and `X` in the `SIATEC` implementation in Figure 33 after line 28 has been executed when this implementation is run on the dataset in Figure 1(a).

### 3.2.7 The `PRINT_TECS` algorithm

The `PRINT_TECS` algorithm called in line 29 of the `SIATEC` implementation in Figure 33 and defined in Figure 41, accomplishes Step 7 of the `SIATEC` algorithm described in section 2.2.7 above.

`PRINT_TECS` is an implementation of the algorithm in Figure 12. In `PRINT_TECS`, the variable `X` heads the right-directed list of `X_NODES` representing the ordered set `Y` computed in Step 6 of the `SIATEC` algorithm described in section 2.2.6 above.

The `PRINT_PATTERN` procedure called in line 26 of `PRINT_TECS` and defined in Figure 42 is an implementation of the algorithm in Figure 13.

The `PRINT_SET_OF_TRANSLATORS` procedure called in line 27 of `PRINT_TECS` and defined in Figure 43 is an implementation of the algorithm in Figure 14.

The `IS_ZERO_VECTOR` function called in lines 8, 26, 47 and 58 of the `PRINT_SET_OF_TRANSLATORS` procedure in Figure 43 returns `TRUE` if and only if its argument is equal to the zero vector (i.e., a linked list of `NUMBER_NODES` in which every number is 0).

The `PATTERN_VEC_SEQ_EQUAL` function called in line 30 of `PRINT_TECS` (see Figure 41) takes two `X_NODE` pointer arguments and returns `TRUE` if and only if the ordered vector sets represented by the `vec_seq` fields of the two `X_NODES` are equal.

Figure 72 shows the output generated by `PRINT_TECS` for the dataset in Figure 1(a).

This represents the set of TECs shown in Figure 15. Recall that each TEC in the output of **SIATEC** is represented as an ordered pair  $\langle P, T(P, D) \setminus \mathbf{0} \rangle$  where  $P$  is a non-empty MTP and  $T(P, D)$  is the set of translators for  $P$ . For each of the  $\langle \text{pattern, translator set} \rangle$  pairs generated by **SIATEC**, the **PRINT\_TECS** procedure in Figure 41 first prints out the pattern as a list of vectors, each vector on its own line and the whole list terminated by an empty line (see Figure 72). It then prints an empty line before printing out the translator set, also as a list of vectors each vector on its own line and the set terminated by an empty line. Thus, in the output shown in Figure 72, the odd-numbered vector lists represent patterns and each even-numbered vector list represents the set of translators for the pattern that precedes it.

### 3.3 Example implementation of COSIATEC

Figure 44 shows an efficient implementation of the **COSIATEC** algorithm in Figure 22.

Like the **SIA** and **SIATEC** implementations described above, the **COSIATEC** implementation in Figure 44 takes three arguments: **DFN** is the name of the file containing the dataset to be analysed; **OFN** is the name of the file to which the output will be written; and **SD** is a string of 1s and 0s representing the orthogonal projection of the dataset to be analysed (see section 3.1.1 above).

If the file called **DFN** exists then it is opened (line 8, Figure 44) and the dataset is read (line 10) using **READ\_VECTOR\_SET** (defined in Figure 25). The dataset is then sorted (line 12) and setified (line 13) using the **SORT\_DATASET** (Figure 26) and **SETIFY\_DATASET** (Figure 28) functions already described. The size of the dataset is then computed (line 14) using the **SIZE\_OF\_DATASET** function described in section 3.2.1 above.

The while loop that begins at line 18 in Figure 44 implements the while loop beginning at line 3 in Figure 22. Lines 19–32 in Figure 44 are essentially the same as lines 16–29 of the **SIATEC** implementation in Figure 33. On each iteration of the while loop, this code from **SIATEC** is used to compute  $\mathcal{J}'(D)$  for the dataset stored in the right-directed list of



VECTOR\_NODES headed by the variable `D`. This set of TECs is then stored in a temporary file whose name is kept in `TFN` (line 32, Figure 44).

To prevent memory leakage, the data structures headed by `V` and `X` are deallocated in line 33 of Figure 44 using the function `DISPOSE_OF_SIATEC_DATA_STRUCTURES` defined in Figure 45.

The temporary TEC file `TF` is then opened (line 34, Figure 44) and each TEC in this file is read into memory in turn using the `READ_TEC` function called in line 36 of Figure 44 and defined in Figure 46. This function will be discussed further in section 3.3.1 below.

The function `IS_BETTER_TEC` called in line 37 of the `COSIATEC` implementation in Figure 44 is an implementation of line 10 in Figure 22. It is defined in Figure 48 and discussed further in section 3.3.3 below.

If `IS_BETTER_TEC` returns `TRUE` then the newly read TEC is stored as the best TEC so far and the previously best TEC is deleted using the function `DISPOSE_OF_TEC` called in line 38 of Figure 44.

Once all the TECs have been read from the temporary TEC file, `TF`, the while loop beginning at line 35 terminates. and the best TEC is stored in the variable `BT`. The file `TF` is then closed and deleted (lines 43 and 44 of Figure 44). The best TEC is then written to the output file `OF` in line 45 using the `PRINT_TEC` procedure defined in Figure 49 and described further in section 3.3.4 below. Line 45 in Figure 44 is an implementation of line 14 in Figure 22.

Finally, line 15 of the `COSIATEC` algorithm in Figure 22 is implemented in line 46 of the implementation in Figure 44 using the `DELETE_TEC_COVERED_SET` function defined in Figure 51.

In line 47 of Figure 44, the variable `n` is recalculated so that it once more stores the number of remaining datapoints in the list headed by `D`. The `coverage` field of a `TEC_NODE` stores the coverage of the TEC as defined in Eq.19 above.

### 3.3.1 The READ\_TEC function

In line 36 of Figure 44, the function `READ_TEC`, defined in Figure 46, is used to read each TEC from the temporary TEC file. Each TEC is stored in a `TEC_NODE` data structure as defined in Figure 23.

In line 2 of `READ_TEC`, a new `TEC_NODE` is created, the numerical fields are set to zero and the pointer fields are set to `NULL`. The pointer `T` is set to point to the new node. If  $\langle P, T(P, D) \setminus \mathbf{0} \rangle$  is the TEC that is to be read, then in line 3 of `READ_TEC`, the pattern  $P$  is represented as a down-directed list of `VECTOR_NODES` pointed to by the `pattern` field of `T`. The set of non-trivial translators,  $T(P, D) \setminus \mathbf{0}$ , is then, in line 4 of `READ_TEC`, represented as a down-directed list of `VECTOR_NODES` pointed to by the `translator_set` field of `T`. The size of  $P$  (that is `T↑pattern`) is then computed in line 5 and stored in the field `T↑pattern_size`. In line 6, the size of  $T(P, D) \setminus \mathbf{0}$  is computed and stored in the field `T↑translator_set_size`. In line 7 of `READ_TEC`, the set

$$\bigcup_{v \in T(P, D)} \tau(P, v)$$

is computed and stored in the `covered_set` field of `T`. This is done using the `SET_TEC_COVERED_SET` function defined in Figure 47 and described further in section 3.3.2 below. This allows the coverage of the TEC (see Eq.19) to be computed in line 8 of `READ_TEC` and stored in the `coverage` field of `T`.

Finally the compression ratio of the TEC as defined in Eq.20 is computed in line 9 of `READ_TEC` and stored in the `compression_ratio` field of `T`.

### 3.3.2 The SET\_TEC\_COVERED\_SET function

If the `TEC_NODE` pointer `T` represents the TEC  $\langle P, T(P, D) \setminus \mathbf{0} \rangle$  then the function `SET_TEC_COVERED_SET(T)`, called in line 7 of the `READ_TEC` function and defined in Fig-

ure 47, computes the set

$$\bigcup_{v \in T(P,D)} \tau(P, v)$$

and stores this set as a linked list of `COV_NODES`s, headed by the pointer `T↑covered_set`.

Each `COV_NODE` has two fields as defined in Figure 23: the `datapoint` field is a `VECTOR_NODE` pointer used to point at a `VECTOR_NODE` representing a datapoint in the list headed by `D`; the `next` field simply points at the next `COV_NODE` in the linked list. In this way, a linked list of `COV_NODES` can be used to represent a subset of the dataset.

The function `VECTOR_PLUS` called in line 19 of `SET_TEC_COVERED_SET` simply returns a `NUMBER_NODE` list representing the vector that results from adding the two vectors represented by its arguments.

The `DISPOSE_OF_NUMBER_NODE` function called in line 25 of the `SET_TEC_COVERED_SET` function in Figure 47 destroys and deallocates the list of `NUMBER_NODES` headed by its argument.

The `MAKE_NEW_COV_NODE` function called in lines 33 and 36 of `SET_TEC_COVERED_SET` makes a new `COV_NODE` and sets both of its fields to `NULL`.

### 3.3.3 The `IS_BETTER_TEC` function

The function `IS_BETTER_TEC` called in line 37 of the `COSIATEC` implementation in Figure 44 is an implementation of line 10 in Figure 22. It is defined in Figure 48.

The `PRINT_ERROR_MESSAGE` procedure called in line 2 of `IS_BETTER_TEC` simply prints out its argument to the standard output.

As can be seen in Figure 48, the `IS_BETTER_TEC` function uses the `compression_ratio` and `coverage` fields of its argument `TEC_NODES`,  $T_1$  and  $T_2$ , to determine whether or not  $T_1$  would be a preferable choice to  $T_2$  for use in the compressed representation generated by `COSIATEC`.

### 3.3.4 The PRINT\_TEC function

The PRINT\_TEC function called in line 45 of the COSIATEC implementation in Figure 44 is used to output the ‘best TEC’ for the current state of the dataset to the output file.

PRINT\_TEC, which is defined in Figure 49, uses the procedure PRINT\_VECTOR\_SET defined in Figure 50 to print out first the pattern and then the set of translators for the TEC.

Figure 73 shows the output generated by the COSIATEC implementation in Figure 44 for the dataset in Figure 4. The format of the output for the COSIATEC function in Figure 44 is the same as that generated by the SIATEC implementation in Figure 33.

## 3.4 Example implementations of SIAME

Two versions of the SIAME algorithm will now be described: for a pattern of size  $m$  and a dataset of size  $n$ , the first version has an average running time of  $O(nm)$ ; the second has a worst-case running time of  $O(nm \log(nm))$ .

In Figure 74, we illustrate the working of SIAME. Given the points  $t_i$  of the pattern  $T$  and  $d_j$  of dataset  $D$ , the aim is to generate the structure  $\mathcal{M}$  in the bottom right-hand corner. The first version does this with the aid of an array,  $S$ , and a linked list,  $\mathcal{L}$ ; the second version needs only the former.  $\mathcal{M}$  stores the ⟨vector, point-set⟩ pairs in decreasing order of point-set size.

Let us briefly describe the structures before introducing the pseudo-codes. Each element of the array  $S$  contains three fields: ptr,  $\Delta$ , and  $\Sigma$ . Field “ptr” is a pointer to a linked list of  $t_i$ s that are translatable by a vector  $\bar{v}$  which, itself, is stored in field  $\Delta$ .  $\Sigma$  stores the number of  $t_i$ s translatable by  $\bar{v}$ , that is, the size of the subset of  $T$  represented by this list.

For the first version of SIAME, it is crucial that the (used) nodes in the array  $S$  are reachable in constant time. Hence it maintains a temporary linked list  $\mathcal{L}$ , in which each element contains two pointer fields. Field “ptr” points to a used element in  $S$ , while “next” points to the next element in the list.  $\mathcal{M}$  is an array of pointers, each of which is

pointing to a linked list of the same form as that of  $\mathcal{L}$ .

Let us first introduce a function that shall be called by both versions of **SIAME**. We denote by square brackets ( $[]$ ) and an upwards-arrow ( $\uparrow$ ) array indexing and element pointing, respectively. The function **NEWLINK** (Figure 75) takes two parameters: the first is either a datapoint or a pointer; the second is a pointer to a linked list. **NEWLINK** allocates a new node of the element type pointed to by the latter parameter, and adds this created node as the first element of the linked list. The value of the first parameter is stored in the “data” field of the created node. Note that because the newly created node is put at the very beginning of the list, **NEWLINK** is executed in constant time.

### 3.4.1 Finding Patterns in $O(mn)$ Time on Average.

In order to execute **SIAME** in  $O(mn)$  time, we need to choose the right element of  $S$  in constant time. A simple solution allocates space for the whole possible value range along each dimension and uses array indirection based on the translation vectors,  $\bar{v} = d - t$ , which select members of the **SIAME** output set. This works in constant time, and so is efficient in this respect. The input dataset  $D$  for **SIAME**, however, may be very large in quite ordinary applications. Furthermore, the data may be quite sparse. Therefore, not only is there a potential for the data structures to be generated to become of excessive size, but it is very likely that a large proportion of the space that the program attempts to allocate for them is never actually needed. So we have to balance the strictures of space against the time required to access the data.

In this first version we do so by using a hash function **F** that hashes the translation vectors into an array of size  $O(nmk)$  where  $m$  and  $n$  are, respectively, the size of the pattern to be searched for and the size of the dataset being searched, and  $k$  is the number of dimensions represented in the input data. We use *closed hashing* (Weiss, 1993), in other words, only identical values are hashed to the same location of the array. To make the hashing work in an *expected* constant time, the frequency of *collisions* should be kept low. A collision occurs when two different input values  $p_1$  and  $p_2$ ,  $p_1 \neq p_2$ , have an identical

hashed value,  $F(p_1) = F(p_2)$ . This is possible with a hashing array of size approximately twice the number of the items to be hashed (Weiss, 1993). Moreover, a *secondary hashing procedure* (or a *resolution function*) is needed. For more details on this, see Weiss (1993).

Given  $T$ ,  $D$ , and  $S$  as input, the first version of **SIAME** is as shown in Figure 76. In the nested loops at lines 2–9, **SIAME** operates by comparing each point  $t$  in the query pattern with each point  $d$  in the dataset and uses the main structure  $S$  to store the  $\langle$ vector, point-set $\rangle$  pairs. The hashing function  $F$  (including also the resolution function) is used at line 5 to find the index in  $S$  corresponding to  $\bar{v}$ . After a new node storing the value  $t$  is added to the linked list associated with the vector, then the fields of  $S$ , at the element  $F(\bar{v})$ , are updated. If the current vector,  $\bar{v}$ , has not been met before, a new node is added to the head of the linked list  $\mathcal{L}$  (line 9) and the “data” field of this new node is set to point to  $S[F(\bar{v})]$ .

Having executed these nested loops, the main structure  $S$  contains the  $\langle$ vector, point-set $\rangle$  pair information, and the list elements of  $\mathcal{L}$  point to the nodes of  $S$  corresponding to the vectors that were found to be present in the input data. The length of the list  $\mathcal{L}$  is  $O(mn)$ .

The next phase is to go through the  $\langle$ vector point-set $\rangle$  pairs (lines 11–14) and sort them according to their size counts. The pairs are stored in the structure  $\mathcal{M}$  of size  $O(mn)$ . To give an example, see Figure 74, where  $\Sigma_3 = 3$ ;  $\Sigma_1 = \Sigma_4 = 2$ ; and  $\Sigma_2 = \Sigma_5 = 1$ ).

The total expected time complexity of this first version of **SIAME** is  $O(mn)$ . This is because the execution of line 5 takes a constant time on average. In the worst case, however, it takes  $O(mn)$  time and, therefore, the worst case time complexity for this version is  $O((mn)^2)$ . The remaining lines within the nested **for** loops are executable in constant time. Thus, the execution of lines 2–9 takes  $O(mn)$  on average, while the loop at lines 11–14 is clearly executable in  $O(mn)$  time, even in the worst case.

### 3.4.2 Finding Patterns in $O(mn \log(mn))$ Time in the Worst Case.

In the former implementation,  $S$  comprised an array of size  $2nm$  for each dimension of the vectors. It is in our interest to reduce that still further for our databases may be very large. Our second version needs an array of size  $nm$ . On average it may be slower than the former version, but in the worst case it needs  $O(mn \log(mn))$  time, where  $m$  is usually very small. The second version of **SIAME** is as shown in Figure 77.

This version of **SIAME** first stores all the vectors with the associated  $t_i$  in  $S$ . Then  $S$  is sorted with respect to the vectors by the conventional merge sort. Although Quicksort is faster on average than merge sort, the worst-case time-complexity of Quicksort is  $O(n^2)$  which is worse than the worst-case running time of merge sort. Another reason for preferring merge sort here is because the implementation could be based on linked lists, which would make merge sort an appropriate choice. Finally, the function **MERGEDUPLICATES** in Figure 78 is executed. If the vectors at the consecutive indices in  $S$  are identical, **MERGEDUPLICATES** merges them; all these query pattern datapoints are collected at the location, say  $j$ , where the vector first occurred in  $S$ . Then the  $\Sigma$  field is updated, and an element at the corresponding index of  $\mathcal{M}$  is created to point to  $S[j]$ .

The worst case time complexity for this second version of **SIAME** is  $O(mn \log(mn))$ . The nested loops at lines 3–7 take time  $O(mn)$ , and it is well-known that merge sort has a worst case time complexity of  $N \log N$  for sorting  $N$  objects. The function **MERGEDUPLICATES** runs in time  $O(nm)$ , since every location of  $S$  is visited exactly once (note that the inner loop is executed  $k$  times, after which the outer loop variable  $j$  is updated to  $j + k$ ).

Instead of using merge sort and **MERGEDUPLICATES**, one possibility would have been to sort  $S$  “on-the-fly” within the nested loops of **SIAME**<sub>2</sub> by using, e.g., insertion sort (Weiss, 1993). This would, however, lead to a worst-case time-complexity of  $O((nm)^2)$  (the case where the vectors are given in reversed order).

## References

- Borowski, E. J. and Borwein, J. M. (1989). *Dictionary of Mathematics*. Collins.
- Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. M.I.T. Press, Cambridge, Mass.
- Crochemore, M. and Rytter, W. (1994). *Text Algorithms*. Oxford University Press, Oxford.
- Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge.
- Weiss, M. A. (1993). *Data Structures and Algorithm Analysis in C*. Benjamin Cummings, Redwood City, CA.



## CLAIMS

1. A method of pattern discovery in a dataset, in which the dataset is represented as a set of datapoints in an  $n$ -dimensional space, comprising the step of computing inter-datapoint vectors.
2. The method of Claim 1, adapted to identify translation invariant sets of datapoints within the dataset, comprising the further steps of:
  - (a) computing the largest set of datapoints that can be translated by a given inter-datapoint vector to another set of datapoints in the dataset; and
  - (b) computing all sets of datapoints which are translationally equivalent to the largest set identified in step (a).
3. The method of Claim 2 used for any of the following purposes:
  - (a) lossless data-compression;
  - (b) predicting the future price of a tradable commodity;
  - (c) locating repeating elements in a molecule
  - (d) indexing.
4. The method of Claim 1, adapted to identify the occurrence of a user supplied set of datapoints in a dataset, comprising the further steps of:
  - (a) computing inter-datapoint vectors from each datapoint in the user supplied set of datapoints to each datapoint in the dataset;
  - (b) computing the largest set of datapoints in the user supplied set of datapoints that can be translated by a given inter-datapoint vector to another set of datapoints in the dataset.
5. The method of Claim 4 used for any of the following purposes:

- (a) locating specific elements in a molecule;
  - (b) visual pattern comparison;
  - (c) speech or music recognition.
6. The method of any preceding claim in which the datapoints in an  $n$ -dimensional space represent any of the following:
- (a) audio data;
  - (b) 2D image data;
  - (c) 3D representations of virtual spaces;
  - (d) video data;
  - (e) molecular structure;
  - (f) chemical spectra;
  - (g) financial data;
  - (h) seismic data;
  - (i) meteorological data;
  - (j) symbolic music representations;
  - (k) CAD circuit data.
7. Computer software adapted to perform the method of any preceding Claim 1–6.

**Abstract****METHOD OF PATTERN DISCOVERY**

This invention provides methods for pattern discovery, pattern matching and data compression in multidimensional numerical datasets. The invention can usefully be applied in any domain in which information represented in the form of multidimensional datasets needs to be retrieved, compared, analysed or compressed. Such domains include 2D images, audio and video data, biomolecular data, seismic, meteorological and financial data.

There already exist methods for pattern discovery, pattern matching and data compression but these methods have been designed for processing data represented as strings and there are many domains in which data cannot be appropriately represented using strings. In such domains, existing data-processing methods are not effective.

In many of the domains in which strings cannot be effectively used to represent information (e.g., audio and video data), the data can be represented using multidimensional numerical datasets. The present invention provides methods for processing such datasets.

The method allows maximal matches for a query pattern to be found in a dataset by computing the inter-datapoint vectors between datapoints in the pattern and datapoints in the dataset. The method allows maximal recurring patterns in a dataset to be found by computing inter-datapoint vectors between datapoints in the dataset. An extension of the method allows all occurrences of all maximal recurring patterns in a dataset to be found. This extension to the method can be used to compute a compressed (i.e. space-efficient) representation of a dataset from which the dataset can be reconstructed by multiple translations of an optimal set of generating patterns.