Chapter 8

CH1C: A Fast Concept Hierarchy Constructor for Discrete or Mixed Mode Databases

Paper appears with kind permission from the Knowledge Systems Institute.

Abstract

In this paper we propose an algorithm that automatically creates concept hierarchies or lattices for discrete databases and datasets. The reason for doing this is to accommodate later data mining operations on the same sets of data without having an expert create these hierarchies by hand.

Each step of the algorithm will be examined; We will show inputs and output for each step using a small example. The theoretical upper bound of the complexity for each part of the algorithm will be presented, as well as real time measurements for a number of databases. We will finally present a time model of the algorithm in terms of a number of attributes of the databases.

Keywords: Data Mining, Data Preprocessing, Hierarchy Generation, Lattice Generation
8.1 Introduction

Data mining in large sets of data is not a trivial occupation. Choosing one type of algorithm over another for a certain problem can mean the difference between getting good (for some definition of good) or no, or even inconclusive, results. Almost all of the methods used today in data mining require either structured data (so that clustering can easily be performed) or a concept hierarchy that envisions the inner structure of the data \[6, 4, 16, 5\].

Concept hierarchies are descriptors over data sets, in such a way that all records in the corresponding data sets can be described by the hierarchies. Concept hierarchies are typically used in retrieval systems \[12, 11, 2\] and for data mining \[5\]. Each facet or aspect of the records has a corresponding hierarchy, often in the form of a tree. An example of such a hierarchy that describes a C source code could be Imperative $\rightarrow$ C $\rightarrow$ ANSI for the language facet and Ordered $\rightarrow$ Tree $\rightarrow$ Binary for the data type facet. Constructing a concept hierarchy manually is very time consuming and relies on a thorough understanding of the actual data at hand. Construction of concept hierarchies can be automated (as in this work). However, there are still some doubts about the soundness of automatically generated facets/dimensions, especially since no exact rules for what is a “good” concept hierarchy exist today.

The remainder of the paper is organized as follows; The second section is the background to the work. The third section gives the definitions necessary for understanding the later parts of this paper. The fourth section is the algorithm, while the fifth contains an example of running the algorithm. The sixth section contains an analysis of the algorithm and the seventh shows relationships with previous work. The last two sections contains the discussion and the experiences that we have had with CHIC, respectively.

January 3, 2009 Ola Ågren
8.2 Background

A few years ago we were faced with a large data set consisting of two parts; One relational and one consisting of a set of keywords. Feeding only the relational part of the data set into data mining systems revealed no new information about it. We realized that the only way of getting more information from the data set was to use the given set of keywords as means of additional clustering. We were unable to find any already published algorithm for finding structures in discrete data, so we had to create one ourselves. The result of our work was a concept hierarchy constructor, called CHiC.

CHiC will automatically find concept hierarchies (or lattices, as the case might be) in sets of discrete data. Our definition of discrete data is either keyword based or otherwise enumerated data, e.g. the sample data in Figure 8.1 on the next page. Our algorithm knows nothing about differences between numbers, whether integer or floating point, so it cannot be used to cluster numeric or continuous data.

CHiC has been used by major divisions of at least two multinational companies to generate concept hierarchies from their mixed mode databases.
<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Makefile: make commands text</td>
</tr>
<tr>
<td>02</td>
<td>clean: Bourne shell script text</td>
</tr>
<tr>
<td>03</td>
<td>combinator: ELF 32-bit MSB executable, SPARC, version 1, dynamically linked, not stripped</td>
</tr>
<tr>
<td>04</td>
<td>db: directory</td>
</tr>
<tr>
<td>05</td>
<td>fest.txt: International language text</td>
</tr>
<tr>
<td>06</td>
<td>fil.aux: LaTeX auxiliary file</td>
</tr>
<tr>
<td>07</td>
<td>fil.dot: ASCII text</td>
</tr>
<tr>
<td>08</td>
<td>fil.eps: PostScript document text conforming at level 2.0</td>
</tr>
<tr>
<td>09</td>
<td>fil.log: TeX transcript text</td>
</tr>
<tr>
<td>10</td>
<td>fil.tex: LaTeX 2e document text</td>
</tr>
<tr>
<td>11</td>
<td>input: English text</td>
</tr>
<tr>
<td>12</td>
<td>lex.yy.c: C program text</td>
</tr>
<tr>
<td>13</td>
<td>main.c: C program text</td>
</tr>
<tr>
<td>14</td>
<td>main.o: ELF 32-bit MSB relocatable, SPARC, version 1, not stripped</td>
</tr>
<tr>
<td>15</td>
<td>words.l: lex description text</td>
</tr>
<tr>
<td>16</td>
<td>words.o: ELF 32-bit MSB relocatable, SPARC, version 1, not stripped</td>
</tr>
</tbody>
</table>

Figure 8.1: A short example of discrete meta-data, in this case output of the "file" command.
8.3 Definitions

Given $k_i \in \text{keywords}$ (the keywords in the system) and $\sigma_k \in \mathcal{P}\text{records}$ (the set of records in the database that contain keyword $k_i$), we can define the concepts used in this work.

A **bucket** is a placeholder for a set of keywords.

Two or more keywords that always appear together are **keyword equivalent**, i.e.

$$k_i =_k k_j \iff \sigma_{k_i} = \sigma_{k_j}, \text{ where } k_i \neq k_j. \quad (8.1)$$

This implies that they may be **folded** into one keyword, i.e. the name of the latter is added as a synonym of the first and all references to the latter are removed or ignored. Such removed keywords are said to belong to $k_i$’s **family**, i.e.

$$k_i =_k k_j \implies k_i, k_j \in \text{fam}_i, \text{fam}_i \in \mathcal{P}\text{keywords}. \quad (8.2)$$

Formally, **subsumption** is defined as an implicit subset/superset relationship between the interpretations of the two concepts [1]. This means that if a keyword $k_i$ never appears in a record without $k_j$ appearing, but not vice versa, then $k_i$ is **subsumed** by $k_j$.

$$k_i <_k k_j \iff \sigma_{k_i} \subset \sigma_{k_j} \quad (8.3)$$

A keyword $k_i$ that is not subsumed by another keyword is said to be a **vector**. Vectors are used as the bases of the concept hierarchies.
8.4 The Algorithm

Figure 8.2: Each step of the algorithm and all intermediate results.
The algorithm (as outlined in Figure 8.2 on the facing page) given here is straightforward, although some of the implementation details can be anything but trivial to implement. The only requirements are that the keywords must have a consistent enumeration order (as given in Equation 8.4) and that the database has already been created. The different parts of the algorithm are otherwise independent of different implementations of sets, etc:

\[ \forall x, y \in \text{keywords} \bullet x \neq y \rightarrow \text{ord}_x \neq \text{ord}_y \]

Further information about each part of the algorithm are given in Section 8.4.7 (correctness of the algorithms) and Section 8.6 (cost estimations based on code complexity).

### 8.4.1 Read Index Data

All keywords should be given an enumeration order and the inverted file (see [8]) corresponding to each keyword is read from disk. The following must hold after the step in the algorithm has been executed:

\[ \forall x \in \text{keywords} \bullet \sigma_x = \text{inverted DB file}_x \]
\[ \forall x \in \text{keywords} \bullet |\sigma_x| = y \rightarrow x \in \text{bucket}_y \]

**Algorithm 8.1 (ReadInvertedFiles) This step of the algorithm reads the index/inverted data files from the disc, or must generate them (at some run time cost) if they do not exist. In this algorithm description (and in the cost estimation in Section 8.6.1) we expect them to be on disc before execution.**

```plaintext
proc ReadInvertedFiles() ≡
  r  Empty all buckets;
  foreach x ∈ keywords do
    \[ \sigma_x ← \text{inverted DB File}_x; \]
    \[ y ← |\sigma_x|; \]
    \[ \text{bucket}_y ← \text{bucket}_y \cup \{x\}; \]
  done
```

---

Finding, Extracting and Exploiting Structure in Text and Hypertext
8.4.2 Find and Fold Keyword Families

The second step of the algorithm finds keywords that are keyword equivalent, adds the latter to the formers family and removes the latter (according to the keyword enumeration order) from further computations.

The following must always hold true for the algorithm used:

$$\forall x, y \in \text{keywords} \cdot ord_x < ord_y \land \sigma_x = \sigma_y \rightarrow x =_{k} y$$ \hspace{1cm} (8.7)

where $y$ should be excluded from further computations.

**Algorithm 8.2 (FindAndFoldFamilies)**: This step of the algorithm compares each keyword in a bucket with all other keywords with a later enumeration order in the same bucket, removing the latter if the two are keyword equivalent.

```
proc FindAndFoldFamilies() ≡
  foreach bucket ∈ {buckets} do
    while $\exists x, y \in \text{bucket} \cdot ord_x < ord_y \land \sigma_x = \sigma_y$ do
      fam_x ← fam_x ∪ fam_y;
      bucket ← bucket \ fam_y;
      keywords ← keywords \ fam_y;
    done
  done
```

---

January 3, 2009  
Ola Ågren
8.4.3 Find All Subsets and Subsumptions

This step of the algorithm finds where one keyword is subsumed by another, while ignoring folded keywords altogether. The effect of this step can be summarized as:

\[
\forall x, y \in \text{keywords} \bullet x \neq y \land \sigma_x \subset \sigma_y \rightarrow x <_k y
\]  
(8.8)

**Algorithm 8.3 (FindSubsets)** This step of the algorithm will find all subsumptions in the database. CHiC works in decreasing bucket size order to further optimize the algorithm.

```
proc FindSubsets() ≡
  foreach x ∈ keywords do
    foreach y ∈ keywords • |σ_y| < |σ_x| do
      if σ_x ⊂ σ_y then x <_k y;
    done
  done
```

The reason for performing family folding before finding subsets is to prune the solution space as quickly as possible, since all keywords in a family will subsume/be subsumed by exactly the same keywords.

8.4.4 Find All Vector Nodes

The goal of the next step is to find all vectors (keywords that are not subsumed by any other keyword) in the given dataset, i.e.:

\[
\text{vectors} = \{ x \in \text{keywords} \mid \nexists y \in \text{keywords} \bullet x <_k y \}
\]  
(8.9)

**Algorithm 8.4 (FindVectorNodes)** This step of the algorithm will add all found vectors in an array, sorted in decreasing cardinality order. The order is important for the heuristics of the next step.

```
proc FindVectorNodes() ≡
  vectors ← empty array;
  foreach x ∈ keywords do
    if \nexists y ∈ keywords \land x <_k y
      then insert k in vectors;
  done
```
### 8.4.5 Group Vectors into Different Dimensions/Facets

Partition the set of vectors into as few partitions \((\text{vectors}_i)\) as possible, while maintaining:

\[
\forall x, y \in \text{vectors}_i \cdot x \neq y \land \sigma_x \cap \sigma_y = \emptyset \tag{8.10}
\]

**Algorithm 8.5 (ColourVectorNodes)** The algorithm is a greedy implementation based on heuristics that maintains Equation 8.10. CHiC contains two different versions of keyword selection: The first version chooses the keyword with the highest cardinality that fits and the second selects the keyword that has the highest number of conflicts with other keywords first. The latter version is much more costly in terms of CPU time as can be seen in Section 8.6.1.

```
proc ColourVectorNodes() ≡
  \!
  \!
  \!
dim ← 0;
  while vectors ≠ empty array do
    \!
    \!
    \!
dim ← dim + 1;
    \!
    \!
    \!
    while \exists k \in \text{vectors} \cdot \sigma_k \cap \bigcup_{y \in \text{vectors}_\text{dim}} \sigma_y = \emptyset do
      Remove k from vectors;
      \!
      \!
      \!
      Add k to vectors_{\text{dim}};
    \!
    \!
    \!
done
  \!
  \!
  \!
done

maxDimension ← dim;
```

There exists yet another option to CHiC that changes the behavior of this step slightly. It is possible to ask the program to fill the dimensions as much as possible by reusing vectors from one dimension in later dimensions if there is no conflict with vectors already added in that dimension. The easiest way to do this is to mark vectors with no conflict with any remaining vector in \(\text{vectors}\) to be used in current dimension and all that follows it. It is also very easy to go back and check already used vectors for conflicts when setting up latter dimensions.

The reason for using a heuristics based approach rather than trying all possible combinations is the extreme processing cost that it would incur. Finding the optimal colouring for even a small data base of, e.g., 36 vectors and 6 dimensions would yield \(6^{35}\) combinations. Given a computer system that would test 50,000 combinations per second that would take in the order of \(3.44 \times 10^{22}\) seconds, or roughly \(10^{15}\) years. Our system yields an answer in a fraction of a second for the same data base, but we can not conclusively say that it is an optimal solution (neither in the number of dimensions, nor in the spanning of each dimension).

---

\(^1\)One vector locked in the first dimension and with many permutations of combinations occurring more than once but in different dimensions.
8.4.6 Fill Out the Concept Hierarchies/Lattices According to the Subsumptions Found in Step 8.4.3 on page 85

This step generates a number of graphs $G_i = (N_i, E_i)$ where the nodes and the edges are given by equations 8.11 and 8.12.

\[
N_i = \{ x \in \text{keywords} | x \in \text{vectors} \lor \exists y \in \text{vectors} \bullet x <_k y \} \quad (8.11)
\]

\[
E_i = \{ (y, x) | x, y \in \text{keywords} \land x <_k y \land \exists z (x <_k z \land z <_k y) \} \quad (8.12)
\]

Algorithm 8.6 (GenerateDimensions, Hierarchy Version) This algorithm generates concept hierarchies for each dimension (see Figure 8.4 on page 93 for an example), i.e. the desired output for later data mining on the data.

\begin{verbatim}
proc GenerateDimensions() ≡
  for i ← 1 to maxDimension do
    N_i ← vectors_i;
    foreach x ∈ vectors_i do
      N_i ← N_i ∪ { y ∈ keywords | y <_k x };
    done
    E_i ← { (y, x) | \forall x, y ∈ N_i \bullet x <_k y };
    Perform topological sort on E_i;
  done
\end{verbatim}

Finding, Extracting and Exploiting Structure in Text and Hypertext
Algorithm 8.7 (generateDimensions, Lattice Version) This step of the algorithm generates concept lattices for each dimension. This means that there might be more than one path between a vector and a given node further down in the graph.

\[
\text{proc } \text{GenerateDimensions() } \equiv \\
\quad \text{for } i \leftarrow 1 \text{ to } \text{maxDimension} \text{ do} \\
\quad \quad N_i \leftarrow \text{vectors}_i; \\
\quad \quad \text{foreach } x \in \text{vectors}_i \text{ do} \\
\quad \quad \quad N_i \leftarrow N_i \cup \{y \in \text{keywords} \mid y \leq_k x\}; \\
\quad \text{done} \\
\quad \text{done} \\
\quad \text{foreach } x \in \text{keywords} \text{ do} \\
\quad \quad \text{foreach } y \in \text{keywords} \bullet y \leq_k x \text{ do} \\
\quad \quad \quad \text{foreach } z \in \text{keywords} \bullet z \leq_k y \text{ do} \\
\quad \quad \quad \quad \text{Remove } z \leq_k x \\
\quad \quad \text{done} \\
\quad \text{done} \\
\quad \text{for } i \leftarrow 1 \text{ to } \text{maxDimension} \text{ do} \\
\quad \quad E_i \leftarrow \{(y,x) \mid \forall x,y \in N_i \bullet x \leq_k y\}; \\
\quad \text{done}
\]

8.4.7 Correctness of the Algorithm

Equation 8.4 on page 83 implies that each keyword should have an unique order number. This number is used when selecting which keyword to fold and which to keep in Algorithm 8.2 on page 84 and it is also used by CHIC to prune the solution space in Algorithm 8.3 on page 85. Our solution to the ordering number is to give each keyword a number from 1024 and up in the order that they were seen when generating the database, which implies that a keyword cannot be subsumed by a keyword with a higher number unless they appeared for the first time in the same record.

Equation 8.5 on page 83 is trivial. The implication of this equation is that the values have to be available at later stages, but how these are made available is not important for the algorithm.

Equation 8.6 on page 83 shows the bucket sorting stage of the algorithm. The algorithm will work even if the sorting stage is removed, but at a much higher computational cost (see [17]).
The slightly rewritten form of Algorithm 8.1 on page 83 seen below shows that it does indeed fulfill Equations 8.5 and 8.6 on page 83.

\[
\text{proc} \ \text{ReadInvertedFiles}() \equiv \\
\quad \forall x \in \text{keywords} \bullet \sigma_x \leftarrow \text{inverted DB File}_x; \\
\quad \forall x \in \text{keywords} \bullet \text{bucket}_{|\sigma_x|} \leftarrow \text{bucket}_{|\sigma_x|} \cup \{x\}; \\
\]

Equation 8.7 on page 84 is a combination of Definitions 8.2 and 8.3 on page 81. The implication of the equation is that only one of each pair of keyword equivalent keywords should be used in later computation; The latter keyword is used as yet another name on the syntactic rather than semantic level of the keyword still remaining. Proving that Algorithm 8.2 on page 84 fulfills Equation 8.7 on page 84 is trivial, since it follows directly from Definitions 8.2 and 8.3 on page 81 together with Equation 8.4 on page 83.

Equation 8.8 on page 85 follows directly from Definition 8.3 on page 81. Algorithm 8.3 on page 85 is a rewriting of the equation to ignore some of the impossible solutions (i.e., checking if the keyword $x$ subsumes the keyword $y$ iff $|\sigma_x| > |\sigma_y|$), thereby making it more efficient.

Equation 8.9 on page 85 is the definition of vector and Algorithm 8.4 on page 85 is just an extension of that equation.

Equation 8.10 on page 86 is the minimal requirements for step 8.5 on page 86 but it does not specify how it achieved. Algorithm 8.5 on page 86 is written is such a way that it complies with Equation 8.10 on page 86 but the critical part in this step is the selection algorithm in the inner while loop. CHiC contains two different selection algorithms, as mentioned in Section 8.4.5.

Equation 8.11 on page 87 makes sure that only those keywords that can be reached from the vectors in a given dimension is added to that dimension.

Equation 8.12 on page 87 yields the edges (subsumptions) that are between keywords belonging to that dimension, as per Equation 8.11 on page 87.

Algorithm 8.7 on the preceding page follows Equations 8.11 and 8.12 on page 87 while Algorithm 8.6 on page 87 has the added rule that there must exist at most one path between a vector and any given keyword of that dimension. This is the difference between generating a hierarchy and a lattice.
8.5 Example of Execution

If we take the dataset in Figure 8.1 on page 80 as input to our algorithm we will get the following results:

1. Reading inverted files yields:

   \[
   \begin{array}{ccc}
   k_i & \sigma[k_i] \\
   \hline
   \text{make} & 01 & \text{\LaTeX} \ 06, 10 \\
   \text{commands} & 01 & \text{auxiliary} \ 06 \\
   \text{text} & 01, 02, 05, 07 – 13, 15 & \text{file} \ 06 \\
   \text{Bourne} & 02 & \text{ASCII} \ 07 \\
   \text{shell} & 02 & \text{PostScript} \ 08 \\
   \text{script} & 02 & \text{document} \ 08, 10 \\
   \text{ELF} & 03, 14, 16 & \text{conforming} \ 08 \\
   \text{32-bit} & 03, 14, 16 & \text{at} \ 08 \\
   \text{MSB} & 03, 14, 16 & \text{level} \ 08 \\
   \text{executable} & 03 & 2.0 \ 08 \\
   \text{SPARC} & 03, 14, 16 & \text{\LaTeX} \ 09 \\
   \text{version} & 03, 14, 16 & \text{transcript} \ 09 \\
   \text{I} & 03, 14, 16 & 2e \ 10 \\
   \text{dynamically} & 03 & \text{English} \ 11 \\
   \text{linked} & 03 & \text{C} \ 12, 13 \\
   \text{not} & 03, 14, 16 & \text{program} \ 12, 13 \\
   \text{stripped} & 03, 14, 16 & \text{relocatable} \ 14, 16 \\
   \text{directory} & 04 & \text{lex} \ 15 \\
   \text{International} & 05 & \text{description} \ 15 \\
   \text{language} & 05 & \\
   \end{array}
   \]

   The buckets are also filled in with the following contents:

   \[\text{bucket} \ # \quad \text{Content}\]
   \[1 \quad \text{make, commands, Bourne, shell, script, executable, dynamically, linked, directory, International, language, auxiliary, file, ASCII, PostScript, conforming, at, level, 2.0, \LaTeX, transcript, 2e, English, lex, description}\]
   \[2 \quad \text{\LaTeX, document, C, program, relocatable}\]
   \[3 \quad \text{ELF, 32-bit, MSB, SPARC, version, I, not, stripped}\]
   \[11 \quad \text{text}\]
Figure 8.3: Subsumption and family graph of the example data in Figure 8.1.
2. The following non-trivial keyword families are found in the data:

<table>
<thead>
<tr>
<th>family</th>
<th>folded keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>commands</td>
</tr>
<tr>
<td>Bourne</td>
<td>shell, script</td>
</tr>
<tr>
<td>ELF</td>
<td>32-bit, MSB, SPARC, version, 1, not, stripped</td>
</tr>
<tr>
<td>executable</td>
<td>dynamically, linked</td>
</tr>
<tr>
<td>International</td>
<td>language</td>
</tr>
<tr>
<td>auxiliary</td>
<td>file</td>
</tr>
<tr>
<td>PostScript</td>
<td>conforming, at, level, 2.0</td>
</tr>
<tr>
<td>TeX</td>
<td>transcript</td>
</tr>
<tr>
<td>C</td>
<td>program</td>
</tr>
<tr>
<td>lex</td>
<td>description</td>
</tr>
</tbody>
</table>

3. Subsumptions found are:

\[
\begin{align*}
2e & <_k document & \text{English} & <_k \text{text} & \text{document} & <_k \text{text} \\
2e & <_k \text{text} & \text{International} & <_k \text{text} & \text{executable} & <_k \text{ELF} \\
2e & <_k \text{TeX} & \text{PostScript} & <_k \text{document} & \text{lex} & <_k \text{text} \\
\text{ASCII} & <_k \text{text} & \text{PostScript} & <_k \text{text} & \text{make} & <_k \text{text} \\
\text{Bourne} & <_k \text{text} & \text{TeX} & <_k \text{text} & \text{relocatable} & <_k \text{ELF} \\
\text{C} & <_k \text{text} & \text{auxiliary} & <_k \text{TeX} \\
\end{align*}
\]

The results of steps 2 and 3 can be seen in Figure 8.3 on the previous page.

4. The vector families in the sample are text, ELF, \text{TeX} and directory with 11, 3, 2 and 1 instances respectively.

5. The first dimension consists of the vectors text, ELF and directory, while the second dimension has only one vector, \text{TeX}.

6. Only 2 edges are removed in the topological sorting of this small dataset, (text,PostScript) and (text,2e). The end result of running the algorithm is seen in Figure 8.4 on the facing page.
Figure 8.4: The concept hierarchy generated from the data in Figure 8.1.
8.6 Algorithm Analysis

The algorithm given in Section 8.4 is a step by step description of what should be done in order to get a consistent concept hierarchy for later data mining. The steps are to be performed one after another and some of the steps can easily be broken down in parts to be executed in parallel (e.g. all dimensions can be handled simultaneously in steps 8.6 on page 87 and 8.7 on page 88).

Two things have to be considered though, and that is the maximal cost of running the algorithm and the actual running times on real data. The complexity of the algorithm is calculated in Section 8.6.1 and the running time of the prototype is given in Section 8.6.2.

8.6.1 Cost Estimation

Given \( m \) (number of records) and \( n \) (number of keywords), we can give the following upper bounds of the algorithmic complexities:

\[
\begin{align*}
\text{proc } \text{ReadInvertedFiles()} & \equiv O(mn) \\
\text{proc } \text{FindAndFoldFamilies()} & \equiv O(mn^2) \\
\text{proc } \text{FindSubsets()} & \equiv O(mn^2) \\
\text{proc } \text{FindVectorNodes()} & \equiv O(mn^2) \\
\text{proc } \text{ColourVectorNodes()} & \equiv O(mn^2) \text{ or } O(mn^{2.5}) \\
\text{proc } \text{GenerateDimensions()} & \equiv O(mn + n^2) \text{ or } O(mn^2) \\
\text{total} & \equiv O(mn^2) \text{ or } O(mn^{2.5}).
\end{align*}
\]

This includes the cost of set operations that we estimate to be linear to the number of possible elements in a set, e.g. \( O(m) \) or \( O(n) \).
Table 8.1: Descriptions of the test data bases.

<table>
<thead>
<tr>
<th>Set</th>
<th>records ((m))</th>
<th>keywords ((n))</th>
<th>vectors ((n'))</th>
<th>subsets ((m'))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12492</td>
<td>634</td>
<td>36</td>
<td>2409</td>
</tr>
<tr>
<td>2</td>
<td>12492</td>
<td>780</td>
<td>63</td>
<td>3850</td>
</tr>
<tr>
<td>3</td>
<td>256388</td>
<td>7182</td>
<td>114</td>
<td>49631</td>
</tr>
<tr>
<td>4</td>
<td>9480</td>
<td>5042</td>
<td>2458</td>
<td>6781</td>
</tr>
<tr>
<td>5</td>
<td>7496</td>
<td>3204</td>
<td>1400</td>
<td>4357</td>
</tr>
</tbody>
</table>

8.6.2 Actual Execution Times

Some of the data bases used for testing the algorithm are summarized in Table 8.1. The first three are prototypical for the discrete databases that we have encountered elsewhere. The last two are more academic in their nature, since they have unusually high number of keywords and vectors for their size.

The execution times of the prototype as given in Table 8.2 on the following page are from a Sun Blade-1000 workstation with a 750 MHz UltraSPARC-III processor, 8 MB level 2 cache and 512 MB of memory. CHiC was compiled using gcc version 2.95.2.

Modeling the execution time in terms of the values given in Table 8.1 and the time values in Table 8.2 on the following page (together with multiple other data sets) yielded a simple model for each step. We have not modeled the extra time added in step 5a/5b if reusing of vectors in later dimensions is used (as per Section 8.4.5), but since it is in the order of 0.2% of the execution time (for all but the smallest of our data bases, where it increases the time 0.7%) it is almost negligible.
Table 8.2: Average execution times per step in seconds (rounded to three decimals).

<table>
<thead>
<tr>
<th>Set</th>
<th>1</th>
<th>2</th>
<th>3 and 4</th>
<th>5a</th>
<th>5b</th>
<th>6a</th>
<th>6b</th>
<th>fill+6a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.287</td>
<td>0.363</td>
<td>0.869</td>
<td>0.012</td>
<td>0.081</td>
<td>0.329</td>
<td>0.116</td>
<td>0.437</td>
</tr>
<tr>
<td>2</td>
<td>0.442</td>
<td>0.382</td>
<td>1.574</td>
<td>0.041</td>
<td>0.397</td>
<td>0.516</td>
<td>0.230</td>
<td>0.654</td>
</tr>
<tr>
<td>3</td>
<td>34.931</td>
<td>1732.080</td>
<td>3047.250</td>
<td>5.060</td>
<td>6.678</td>
<td>64.088</td>
<td>23.861</td>
<td>67.413</td>
</tr>
<tr>
<td>4</td>
<td>2.274</td>
<td>8.436</td>
<td>42.454</td>
<td>36.407</td>
<td>1012.340</td>
<td>5.241</td>
<td>5.952</td>
<td>48.582</td>
</tr>
</tbody>
</table>

\[
t_1 \approx 1.77 \times 10^{-8} mn + 3.19 \times 10^{-4} n \\
t_2 \approx 1.31 \times 10^{-10} mn^2 \\
t_{3\&4} \approx 2.30 \times 10^{-10} mn^2 \\
t_{5a} \approx 6.40 \times 10^{-10} mn^2 \\
t_{5b} \approx 1.36 \times 10^{-6} \ln(m)n^{2.333} \\
t_6 \approx 1.77 \times 10^{-7} m'n \\
t_{\text{fill}+6} \approx 2.44 \times 10^{-11} m'n^2 + 3.51 \times 10^{-6} m'n' \\
t_7 \approx 4.04 \times 10^{-8} m'n + 1.79 \times 10^{-7} n^2 \quad (8.13) \]

The number of vectors and subsets can often be hard to guess before calculations have been done, but we have found that \( n' \approx n^2/m \) and \( m' \approx \sqrt{mn} \) are useful approximations. They will usually yield results within 25% from the correct value for almost all data bases.

\[\text{Set three in Table 8.2 is one such exception, since it overestimates } n' \text{ by 76%}.\]
8.7 Related Work

The work reported in this article is closely related to semantic and knowledge indexing. It is a widely spread research area that includes such diverse topics as finding keywords for hierarchical summarization [9], knowledge acquisition tools [3] and automatic indexing of system commands based on their manual pages on a UNIX system [14].

Conceptual clustering systems [15] are also quite closely related to this work, but from the viewpoint of the data mining system rather than at the preprocessing stage.

Automatic generation of concept hierarchies as defined in [13] is related but works on a probabilistic, rather than a discrete, view of the data. The rule used to find subsumptions in that work is that \( x \) would subsume \( y \) if \( P(x | y) \geq 0.8, P(y | x) < 1 \). This means that transitivity will not work for subsumptions; Given three keywords (\( x, y \) and \( z \)) such that \( P(x | y) \geq 0.8, P(y | x) < 1, P(y | z) \geq 0.8, P(z | y) < 1 \) and \( P(x | z) \geq 0.8, P(z | x) < 1 \) would yield a very problematic state since \( z \) is subsumed by \( y \) which in turn is subsumed by \( x \) but \( z \) is not subsumed by \( x \). Whether this is acceptable or not is up to the user of the system.

The concept hierarchies generated by our algorithm have strong resemblances to the feature-oriented classification trees used by Salton [12], Prieto-Díaz [11] and Börstler [2]. We believe that this is not a coincident, and that such structures evolve naturally when working with closely related pieces of data like software assets.
8.8 Discussion

One method of speeding up the average case while keeping the worst case the same has been used in a newer version of CHIC. The keywords are sorted (using bucket sort) in decreasing cardinality order in step 8.1. This speeds up the algorithm considerably for the average case, since keyword equality implies the same cardinality and subset a lower cardinality. This implies that the number of keywords to be checked in steps 8.2 & 8.3 on page 85 was decreased significantly (with factors of approximately $1/\log(n)$ and at least 1/2, respectively).

The grouping of vectors into dimensions uses a heuristic algorithm rather than a best fit or even backtrack based graph colouring algorithm [10, 7]. We have experimented with multiple versions of backtracking algorithms, but were not able to find any that were both fast enough and gave sufficiently better results than our heuristic algorithm.

Some of the non-vector keywords are subsumed by more than one vector, thereby generating a lattice structure with rather complex properties. This is both a weakness and a strength; A weakness since it will generate the same set of vertices more than once and a strength since such keywords are excellent candidates for finding correlations and interesting data points in later Data Mining operations. It would be rather simple to update CHIC so that it excludes all keywords after they have been used for the first time in a dimension if that would fit a certain problem.

Previous version of CHIC generated a conceptual hierarchy for each dimension even though the data indicated that a lattice would be more fitting (see Figure 8.5 on the next page for examples of both). The reason for this was the topological sorting done in the last step of GenerateDimensions (Algorithm 8.6 on page 87). Removing the correct vertices from the set to generate a lattice was fortunately not that hard (see Algorithm 8.7 on page 88); Subsumption is a transitive function, e.g. if keyword $a$ subsumes both $b$ and $c$ while $b$ subsumes $c$ then the vertex $(a,c)$ can safely be removed from the resulting set. This technique proved to be very useful (especially when the keywords are already in decreasing cardinality order after Algorithm 8.1 on page 85 is done), but requires that the chosen Data Mining system can handle lattices rather than hierarchies.

The time required for generating the concept hierarchy for our biggest dataset seems rather high (just over 80 minutes of computation, see Section 8.6.2., but since this is done only once for each dataset before doing data mining we believe it to be satisfactory anyway. One way of speeding this up is to create an extra dataset that contains no duplicate records and use this dataset when constructing the concept hierarchy. The time for concept hierarchy generation on our large dataset (with 38187 unique records) falls to around 11 minutes execution time. Data mining is often done on static databases, e.g. data warehouses, so the cost
Figure 8.5: Hierarchical (a) and lattice (b) structures of attributes.

of concept hierarchy generation should generally be amortized over the number of times that the resulting data is later used. Our approach should probably not be used in a constantly evolving database with major upgrades going on simultaneously.

8.9 Experiences

All work so far on the prototype have been fruitful in the form of data relations both directly from concept hierarchy generation and also from later data mining. The concept hierarchies generated have generally been of good quality (i.e. logically connected keywords tend to be in the same dimension or even subsumed, etc.) with some minor glitches and we hope to see future use of our algorithm in the works of others.

The divisions of two multinational companies that have used CHIC have been very pleased with the results obtained from the program, and have found other, more novel, uses for it as well. One of the comments that we have received is that “The only other option [available to us] would have been to hire a very expensive expert in the field, and she would probably have come up with something remarkably similar to what we get from CHIC.”
8.10 References


