GRAPH-BASED HIERARCHICAL CONCEPTUAL CLUSTERING

ISTVAN JONYER, LAWRENCE B. HOLDER, and DIANE J. COOK
University of Texas at Arlington
Department of Computer Science and Engineering
Box 19015 (416 Yates St. Room 300)
Arlington, TX 76019-0015
{jonyer, holder, cook}@cse.uta.edu

Received _________
Revised _________

Hierarchical conceptual clustering has proven to be a useful, although greatly under-explored data mining technique. A graph-based representation of structural information combined with a substructure discovery technique has been shown to be successful in knowledge discovery. The SUBDUE substructure discovery system provides the advantages of both approaches. This work presents SUBDUE and the development of its clustering functionalities. Several examples are used to illustrate the validity of the approach both in structured and unstructured domains, as well as compare SUBDUE to earlier clustering algorithms. Results show that SUBDUE successfully discovers hierarchical clusterings in both structured and unstructured data.

Keywords: Clustering, Cluster Analysis, Data Mining, Concept Formation, Knowledge Discovery.

1. Introduction

Data mining has become a prominent research area in recent years. One of the major reasons is the ever-increasing amount of information collected in diverse areas of the industrial and scientific world. Much of this information contains valued knowledge that is not directly stored or accessible for retrieval. The increasing speed and capacity of computer technology has made feasible the utilization of various data mining techniques to automatically extract knowledge from this information. Such knowledge may take the form of predictive rules, clusters or hierarchies.

Structural databases provide a significant source of information for data mining. A well-publicized example is the human genome project, which set out to map the entire human DNA. DNA strands are structural in nature and therefore require a structured representation in a computer. One of the most prominent ways of representing structural data in computers is by the use of graphs. Graph-based data mining is therefore becoming more important. Substructure discovery is a data mining technique that—unlike many other algorithms—can process structural data, which contains not only descriptions of individual instances in a database, but relationships among these instances as well. The graph-based substructure discovery approach implemented in the SUBDUE system has been the subject of research for a number of years and has been shown to be effective for a wide range of applications.[1] Recent examples include the application of SUBDUE to earthquake activity, chemical toxicity domains, and human and other DNA sequences.[2,3,4]

Cluster analysis—or simply clustering—is a data mining technique often used to identify various groupings or taxonomies in real-world databases. Most existing methods for clustering apply only to unstructured data. This research focuses on hierarchical conceptual clustering in structured, discrete-valued databases.

This work is organized as follows. Section 2 discusses conceptual clustering in greater depth, giving examples and describing specific systems. Section 3 provides a discussion of structural knowledge discovery and an in-depth description of the SUBDUE knowledge discovery system. Section 4 describes the design and implementation of hierarchical conceptual clustering in SUBDUE. Section 5 describes the results of applying SUBDUE to examples from various domains and evaluates SUBDUE's success as a clustering tool. Conclusions and future work are discussed in section 6.

2. Conceptual Clustering

2.1. Introduction and definition

Conceptual clustering has been studied and developed in many areas for a wide variety of applications. Among these are model fitting, hypothesis generation, hypothesis testing, data exploration, prediction based on groups, data reduction and finding true topologies.[5] Clustering techniques have been applied in as diverse fields as analytical chemistry, image analysis, geology, biology, zoology and archeology. Many names have been given to this technique, including cluster analysis, Q-analysis, typology, grouping, clumping, classification, numerical taxonomy, mode separation and unsupervised pattern recognition, which further signifies the importance of clustering techniques.[6]

The purpose of applying clustering to a database is to gain a better understanding of the data, in many cases through revealing hierarchical topologies. An example is the classification of vehicles into groups such as cars, trucks, motorcycles, tricycles, and so on, which are then further subdivided into smaller groups based on observed traits.

Michalski defined conceptual clustering to be a machine learning task.[7] A clustering system takes a set of object descriptions as input and creates a classification scheme.[8] This classification scheme can be a set of disjoint clusters, or a set of clusters organized into a hierarchy. Each cluster is associated with a generalized conceptual description of the objects within the cluster. Hierarchical clusterings are often described as classification trees.

2.2. Overview and Related Work

Numerous clustering techniques have been devised, among which are statistical, syntactic, neural and hierarchical approaches. In all cases, clustering is inherently an unsupervised learning problem, since it consists of identifying valuable groupings of concepts, or facts, which hopefully reveal previously unknown information. Most techniques have some intrinsic disadvantages, however. Statistical and syntactic approaches have trouble expressing structural information, and neural approaches are greatly limited in representing semantic information.[9]

Nevertheless, many relatively successful clustering systems have been constructed. An example of an incremental approach is COBWEB, which successively considers a set of object descriptions, while constructing a classification tree.[8] This system was created with real-time data collection in mind, where a useful clustering might be needed at any moment. COBWEB's search algorithm is driven by the category utility heuristic which reflects intra-class similarity and inter-class dissimilarity using conditional probabilities. Instances are introduced into the classification tree at the top, and are moved down either by creating a new class or by merging it with an existing class. Other existing classes might also be merged or split to accommodate better definitions of classes.

Labyrinth, an extension to COBWEB, can represent structured objects using a probabilistic model.[10] COBWEB creates a knowledge structure based on some initial set of instances. Labyrinth is applied one step before COBWEB, resulting in a structure whose formal definition is exactly the same as that produced by COBWEB. Finally, COBWEB is used, employing both structures to refine the domain knowledge.

AutoClass is an example of a Bayesian classification system, which has a probabilistic class assignment scheme.[11] AutoClass can process real, discrete, or missing values. Another algorithm, called Snob, uses the Minimum Message Length (MML) principle to do mixture modeling—another synonym for clustering.[12]

There also exist hierarchical approaches that target databases containing data in Euclidean space. Among these are agglomerative approaches that merge clusters until an optimal separation of clusters is achieved based on intra- and inter-cluster distances. Divisive approaches split existing clusters until an optimal clustering is found. These approaches usually have the disadvantage of being applicable only to metric data, which excludes discrete-valued and structured databases. Examples of these are Chameleon[13] and Cure.[14]

Examining the major differences among the above mentioned systems, we can see that dichotomies exist between continuous and discrete databases and between structured and unstructured databases. COBWEB can handle discrete, unstructured databases. Labyrinth can work with discrete, structural databases. AutoClass can handle discrete or continuous unstructured databases. Chameleon and Cure work with continuous-valued, unstructured data.

Few existing systems address the problem of clustering in discrete-valued, structural databases. Labyrinth is one of them. SUBDUE is another approach, described in detail in subsequent sections. Our approach centers on discrete-valued, structural databases that are represented as graphs. Clustering is performed iteratively by looking for common patterns in the data. The search is driven by the minimum description length heuristic.

3. Structural Knowledge Discovery

3.1. Terminology

There are terms associated with structural knowledge discovery which are worth clarifying before proceeding. *Structured* data includes relationships among object descriptions in contrast to *unstructured* data that only includes unrelated object descriptions. Many databases currently exhibit structural properties.

Graphs provide a versatile representation of structural data. A *graph* consists of a set of *vertices* that may be connected by *edges*. Both vertices and edges are *labeled*. Edges may be *directed* or *undirected*, which may express different types of relationships. A *subgraph* is a subset of the graph, also referred to as a *substructure*.

Data mining tools may be *incremental*, which means that data is considered one element at a time. A given element is classified in the context of the currently known set of data. This is in contrast to *iterative* methods that require the entire data set before the algorithm can run. These techniques iteratively reconsider and reclassify data until the best result is achieved.

Knowledge discovery by *search* is a common concept. Most data mining algorithms use some type of search algorithm. Most of these use *computational constraints* to keep the search within tolerable time limits. The search progresses from one *search state* to another. Search states can be thought of as lists of partial hypotheses waiting for expansion. A common method of constraining the search is to order this list and only extend the most promising partial hypotheses.

Search algorithms are driven by *evaluation metrics* that assign numeric values to the usefulness of partial hypotheses. In knowledge discovery, these metrics often find their roots in statistics and information theory.

3.2. SUBDUE

This section describes SUBDUE, a structural knowledge discovery system that forms the basis of our research. First we discuss the data representation used by SUBDUE, and then describe the search algorithm in detail. The heuristic used to drive the search and the inexact graph matching used by SUBDUE are also presented. The SUBDUE source code is available at http://cygnus.uta.edu/subdue.

3.2.1. Data representation

SUBDUE is a knowledge discovery system that can deal with structured data—an important feature for many applications. SUBDUE expects a graph as its input, hence a

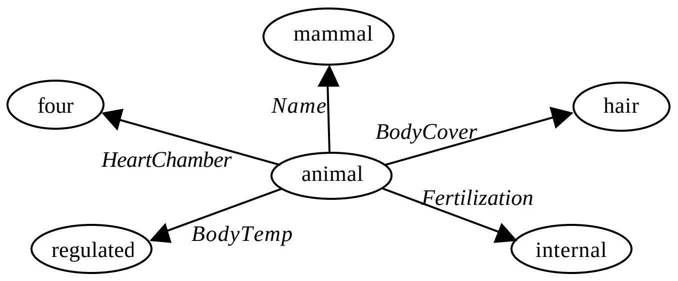

Fig. 1. Graph representation of an animal description.

database needs to be represented as a graph before passing it to SUBDUE. This graph representation includes vertex and edge labels, as well as directed and undirected edges, where objects and attribute values usually map to vertices, and attributes and relationships between objects map to edges (see Fig. 1 for an example).

The input graph need not be connected, as is the case when representing unstructured databases. In those cases the instance descriptions can be represented as a collection of small, star-like, connected graphs. An example of the representation of an instance from the animal domain is shown in Fig. 1. Intuitively, one might map the "main" attribute *Name* in this case—to the center node and all other attributes would be connected to this central vertex with a single edge. This would follow from the semantics of most databases where objects and their attributes are listed. In our experience, however, a more general representation yields better results. In this representation the center node (*animal* in our example), becomes a very general description of the example. Note that the *Name* attribute becomes just a regular attribute. In the most general case, the center node could be named *entity*, or *object*, since the designation is quite irrelevant to the discovery process—the purpose is good structural representation.

3.2.2. Search algorithm

SUBDUE uses a variant of beam search for its main search algorithm (see Fig. 2). The goal of the search is to find the substructure that best compresses the input graph. A substructure in SUBDUE consists of a substructure definition and all its occurrences in the graph. The initial state of the search is the set of substructures representing one uniquely labeled vertex and its instances. The only search operator is the *Extend-Substructure* operator. As its name suggests, *Extend-Substructure* extends the instances of a substructure in all possible ways by a single edge and a vertex, or by a single edge if both vertices are already in the substructure. The Minimum Description Length (MDL) principle is used to evaluate the substructures.

The search progresses by applying the *Extend-Substructure* operator to each substructure in the current search frontier, which is an ordered list of previously discovered substructures. The resulting frontier, however, does not contain all the substructures generated by the *Extend-Substructure* operator. The substructures are stored

```
Subdue ( graph G, int Beam, int Limit )
   queue Q = { v | v has a unique label in G }
   bestSub = first substructure in Q
   repeat
      newQ = {}
      for each S in Q
         newSubs = S extended by an adjacent edge from G
                         in all possible ways
         newQ = newQ U newSubs
         Limit = Limit - 1
      evaluate substructures in newQ by compression of G
      Q = substructures in newQ with top Beam values
      if best substructure in Q better than bestSub
      then bestSub = best substructure in Q
   until Q is empty or Limit = 0
   return bestSub
```

Fig. 2. SUBDUE's discovery algorithm.

on a queue and are ordered based on their ability to compress the graph. The length of the queue is partially limited by the user. The user chooses how many substructures of different value—in terms of compression—are to be kept on the queue. Several substructures, however, might have the same ability to compress the graph, therefore the actual queue length can vary. The search terminates upon reaching a user specified limit on the number of substructures extended, or upon exhaustion of the search space.

Once the search terminates and returns the list of best substructures, the graph can be compressed using the best substructure. The compression procedure replaces all instances of the substructure in the input graph by a single vertex, which represents the substructure. Incoming and outgoing edges to and from the replaced substructure will point to, or originate from, the new vertex that represents the substructure. In our implementation, we do not maintain information on how vertices in each instance were connected to the rest of the graph. This means that we cannot accurately restore the information after compression. This type of compression is referred to as lossy compression, in contrast to lossless compression where the original data can be restored exactly. Since the goal of substructure discovery is interpretation of the database, maintaining information to reverse the compression is unnecessary.

The SUBDUE algorithm can be called again on this compressed graph. This procedure can be repeated a user-specified number of times, and is referred to as an iteration. The maximum number of iterations that can be performed on a graph cannot be predetermined; however, a graph that has been compressed into a single vertex cannot be compressed further.

3.2.3. Minimum description length principle

SUBDUE's search is guided by the Minimum Description Length (MDL) principle, originally developed by Rissanen.[15] In the next section we describe how to calculate the description length of a graph as the number of bits needed to represent the graph. According to the MDL heuristic, the best substructure is the one that minimizes the description length of the graph when compressed by the substructure.[16] This compression is calculated as

$$Compression = \frac{DL(S) + DL(G \mid S)}{DL(G)} \tag{1}$$

where *DL(G)* is the description length of the input graph, *DL(S)* is the description length of the substructure, and *DL(G|S)* is the description length of the input graph compressed by the substructure. The search algorithm is looking to maximize the *Value* of the substructure, which is simply the inverse of the *Compression*.

3.2.3.1 Calculation of description length

The description length of a graph is based on its adjacency matrix representation. A graph having v vertices, numbered from 0 to $v - 1$, has a $v \times v$ adjacency matrix. The adjacency matrix A can have only two types of entries, 0 or 1, where $A[i,j]=0$ represents no edges between vertices i and j, and $A[i,j]=1$ indicates at least one edge (possibly more) between vertices i and j. Undirected edges are represented by a single directed edge with the *directed* flag bit set to 0.

The vertex and edge labels are stored in two separate tables, which contain l_v unique vertex labels, and l_e unique edge labels. These tables might decrease in size as vertices and edges are compressed away, and the table of vertex labels might grow with new vertex labels that stand for substructures that are compressed away. The encoding of a graph is the sum of the *vbits*, *rbits* and *ebits*, which are calculated as follows.

vbits is the number of bits needed to encode the vertex labels of the graph. Each vertex in the graph has a label, and the vertices are assumed to be encoded in the order they appear in the adjacency matrix. First we specify the number of vertices in the graph, which can be done in ($\lg v$) bits. Then, the v labels can be represented in ($v \lg l_v$) bits, as expressed by Eq. 2.

$$vbits = \lg v + v \lg l_v \tag{2}$$

rbits is the number of bits needed to encode the rows of the adjacency matrix A. To do this, we apply a variant of the encoding scheme used by Quinlan and Rivest.[17] This scheme is based on the observation that most graph representations of real-world domains are sparse. In other words, most vertices in the graph are connected to only a small number of other vertices. Therefore, a typical row in the adjacency matrix will have much fewer 1s than 0s. We define k_i to be the number of 1s in row i of adjacency matrix A, and $b = max_i(k_i)$ (that is, the most 1s in any row). An entry of 0 in A means that there are no edges from vertex i to vertex j, and an entry of 1 means that there is at least one edge between vertices i and j. Undirected edges are recorded in only one direction (that is, just like directed edges). SUBDUE's heuristic is that if there is an undirected edge between nodes i and j such that $i < j$, then the edge is recorded in entry $A[i,j]$ and omitted in entry $A[j,i]$. A flag is used to signal if an edge is directed or undirected, which is accounted for in *ebits*. *rbits* is calculated as follows.

Given that k_i 1s occur in the i^{th} row's bit string of length v, only $C(v,k_i)$ strings of 0s and 1s are possible, where $C(n,k)$ is the number of combinations of n choose k. Since all of these strings have equal probability of occurrence, $\lg C(v,k_i)$ bits are needed to specify which combination is equivalent to row i. The value of v is known from the vertex encoding, but the value of k_i needs to be encoded for each row. This can be done in $\lg(b+1)$ bits.

To be able to read the correct number of bits for each row, we encode the maximum number of 1s any given row in A can have. This number is b, but since it is possible to have zero 1s, the number of different values is $b+1$. We need $\lg(b+1)$ bits to represent this value. Therefore,

$$\begin{aligned} rbits &= \sum_{i=1}^{v} \left[\lg(b+1) + \lg C(v,k_i)\right] + \lg(b+1) \\ &= (v+1)\lg(b+1) + \sum_{i=1}^{v} \lg C(v,k_i) \end{aligned} \tag{3}$$

ebits is the number of bits needed to encode the edges represented by $A[i\ j]=1$. The number of bits needed to encode a single entry $A[i\ j]$ is $(\lg m) + e(i,j)[1 + \lg l_e]$, where $e(i,j)$ is the number of edges between the vertices i and j in the graph and $m = \max_{i,j} e(i,j)$—the maximum number of edges between any two vertices i and j in the graph. The $(\lg m)$ bits are needed to encode the maximum number of edges between vertices i and j. For each edge we need $(\lg m)$ bits to specify the actual number of edges, and $[1 + \lg l_e]$ bits are needed per edge to encode the edge label and the *directed* flag (1 bit). Therefore,

$$\begin{aligned} ebits &= \lg m + \sum_{i=1}^{v}\sum_{j=1}^{v} \left[A[i,j](\lg m) + e(i,j)[1 + \lg l_e]\right] \\ &= \lg m + e(1 + \lg l_e) + \sum_{i=1}^{v}\sum_{j=1}^{v} A[i,j]\lg m \\ &= e(1 + \lg l_e) + (K+1)\lg m \end{aligned} \tag{4}$$

where e is the number of edges in the graph and K is the number of 1s in the adjacency matrix A.

The total encoding of the graph is $DL(G) = vbits + rbits + ebits$. The following subsection gives a worked example on how the compression is calculated.

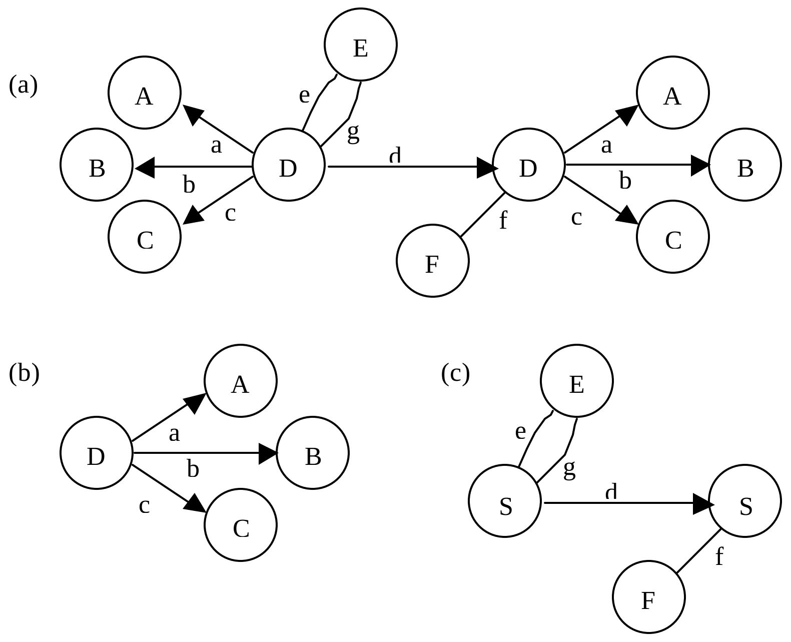

Fig. 3. Example input graph (a) along with the discovered substructure (b) and the resulting compressed graph (c).

3.2.3.2 An Illustrative Example

In this section an illustrative example of computing the description length of the input graph, a substructure, and the input graph compressed by the substructure is worked out. Finally, the computation of the compression and substructure value is shown.

Fig. 3a shows the input graph, Fig. 3b shows the best substructure *S* found after the first iteration, and Fig. 3c shows the input graph compressed with the two instances of substructure *S*. For the purposes of demonstration the input graph has directed edges, undirected edges and multiple edges between a pair of nodes. The undirected edges are represented as directed edges, as mentioned before, and in all three cases they originate in the vertices labeled *D*.

The table of vertex labels has the following entries: A, B, C, D, E, F and S. Therefore, $l_v = 7$. The table of edge labels consists of a, b, c, d, e, f and g, making $l_e = 7$. These two tables are considered global, therefore these values are used for the calculation of all three graphs. Calculating the description length of the input graph proceeds as follows:

vbits: Number of vertices $v = 10$.

$vbits = \lg v + v \lg l_v = \lg 10 + 10 * \lg 7 = 31.39$

rbits: $b = max_i(k_i) = 5$

$$rbits = (10+1)\lg(6+1) + \lg C(10,5) + \lg C(10,4)$$
$$= 30.88 + \lg 15120 + \lg 210$$
$$= 51.67$$

The value of b is 5, not 6 as one might think. Even though there are two edges between vertices D and E, there is only a single 1 standing for them in the adjacency matrix. The two separate edges will be accounted for in *ebits*.

ebits: $m = 2$

$K = 9$

$e = 10$

$$ebits = 10(1+\lg 7) + (9+1)\lg 2$$
$$= 48.07$$

Therefore $DL(G) = vbits + rbits + ebits = 31.39 + 51.67 + 48.07 = 131.13$.

The description length of the substructure is calculated as follows.

vbits: $vbits = \lg 4 + 4 * \lg 7 = 13.23$

rbits: $b = max_i(k_i) = 3$

$$rbits = (4+1)\lg(3+1) + \lg C(4,3)$$
$$= 10 + \lg 4 = 12.00$$

ebits: $m = 1$

$K = 3$

$e = 3$

$$ebits = 3(1+\lg 7) + (3+1)\lg 1$$
$$= 11.42$$

Therefore $DL(S) = vbits + rbits + ebits = 13.23 + 12.00 + 11.42 = 36.65$.

The description length of the input graph compressed by the best substructure is calculated as follows.

vbits: $vbits = \lg 4 + 4 * \lg 7 = 13.23$

rbits: $b = max_i(k_i) = 2$

$$rbits = (4+1)\lg(2+1) + \lg C(4,2) + \lg C(4,1)$$
$$= 7.92 + \lg 6 + \lg 4 = 12.51$$

ebits: $m = 2$

$K = 3$

$e = 4$

$$ebits = 4(1+\lg 7) + (3+1)\lg 2$$
$$= 19.23$$

Therefore *DL*(*G*|*S*) = *vbits* + *rbits* + *ebits* = 13.23 + 12.51 + 19.23 = 44.97. Following the compression formula given earlier, we get

$$Compression = \frac{DL(S) + DL(G \mid S)}{DL(G)} = \frac{36.65 + 44.97}{131.13} = 0.62$$

The *Value* is the inverse of the *Compression*, which is 1.61.

3.2.4. Variations of MDL

The encoding scheme described works well in most cases. It has been suggested, however, that it might not be minimal in all cases. The MDL used by SUBDUE uses a row-wise encoding. If the column-wise version of the same encoding scheme is used, the result might slightly differ, one offering a shorter description than the other. Also, it has been observed that this encoding offers false compression in some cases when only a single-instance substructure is compressed away in the input graph. When looking at the equation to calculate the compression, it can be seen that this should not be the case. When compressing the input graph with a single-instance substructure the sum of the description length of the substructure *DL*(*S*) and the description length of the graph compressed by the substructure *DL*(G|*S*) should not be less than the description length of the input graph *DL*(G) giving a compression of at least 1.0.

A variation of the MDL described here avoids this problem. In the original version the description length of the substructure and the input graph compressed by the substructure are calculated separately, based on their own adjacency matrix. If these two adjacency matrices are combined into one and the description length is calculated based on this adjacency matrix, the above-mentioned false compression will not happen. This variation is used for the results in Section 5.

3.3. Inexact graph matching

When applying the *Extend-Substructure* operator, SUBDUE finds all instances of the resulting substructure in the input graph. A feature in SUBDUE, called inexact graph matching, allows these instances to differ from each other. This feature is optional and the user must enable it as well as specify the degree of maximum dissimilarity allowed between substructures. The command line argument to be specified is *–threshold Number*, where *Number* is between 0 and 1 inclusive, 0 meaning no dissimilarities allowed, and 1 meaning all graphs are considered the same. Specifying 1 for *–threshold* is not particularly useful in practice. A value t between 0 and 1 means that one graph can differ from another by no more than t times the size of the larger graph.

The dissimilarity of two graphs is determined by the number of transformations needed to transform one graph to another. The transformations are to add or delete an edge, add or delete a vertex, change a label on either an edge or a vertex, and reverse the direction of an edge. All of these transformations are defined to have a cost of 1.

Inexact graph matching works by the method of Bunke and Allerman.[18] The algorithm constructs an optimal mapping of the vertices and edges between the two graphs by searching the space of all possible mappings employing a branch-and-bound search. This algorithm has an exponential running time. The implementation in SUBDUE, however, constrains the running time to polynomial by resorting to hill-climbing when the number of search nodes exceeds a polynomial function of the size of the substructures. This is a tradeoff between an acceptable running time and an optimal match cost, but in practice, the mapping found is at or near optimal (lowest cost).

3.4. Improving the search algorithm

In earlier versions of SUBDUE the length of the queue on which the best substructures are held was fixed. Consider the following. Suppose that a substructure *S* that best compresses the graph has 10 vertices, 9 edges, and 50 instances scattered throughout the input graph. This means that any substructure *s* of *S* will have at least 50 instances as well, and will offer the best compression thus far in the search space even when it has only a few vertices. There are 120 substructures having 3 vertices that are substructures of *S* if *S* has 10 vertices and is fully connected. Most substructures however are not fully connected, but if substructure *S* has 10 vertices arranged in a star-like manner having 9 edges connecting them, there can still be 72 distinct substructures having 3 vertices and 2 edges. All these 72 substructures will have the same compression, since they occur the same number of times in the graph and have the same size. Therefore, all these have equal right to be at the head of the queue. If the queue length is chosen to be 4, for example, then the queue will retain only 4 of these 72 substructures, arbitrarily keeping the 4 that happen to be at the top. In the next step only these four will be extended.

The solution is to use a *value-based queue*, which retains not a fixed number of substructures, but a number of *classes* of substructures, each substructure in the same class offering the same compression. In the above example all 72 substructures having the same compression might comprise the first class, leaving room for another three classes—assuming the queue length is four. The value-based queue therefore permits the exploration of a much larger search space.

The problem with the value-based queue is that the membership in each class explodes very quickly. For instance, if all the 72 substructures in the above example are in one class, and extended by applying the *Extend-Substructure* operator, the resulting number of substructures will be in the several hundreds. Since these are still substructures of *S*, they will offer the same compression, and will (most of them) be in the same class. This new class will have hundreds of members, which, when further extended, will result in thousands of substructures. In only a few steps, classes will have grown to an enormous size. Ironically, most of these substructures are headed towards the same substructure *S*.

Fortunately, there is a way to prevent the above phenomenon from happening. An important observation is that the operator *Extend-Substructure* is applied to a single substructure at a time, and that substructure is extended by only one vertex at a time.

These substructures can be kept on a local value-based queue by the operator. The substructures that offer the same compression can be suspected of being the substructure of the same larger substructure. To test this we check to see if either of these substructures can be extended with the vertex the other substructure was just extended by. If so, one of them can be eliminated from further consideration and be removed from the local queue. Finally the local queue is returned by the operator, as usual. This one step look-ahead procedure is referred to as *purging*, because it cleans the local queue by removing substructures that would introduce redundancy in the search process.

An example of purging is demonstrated in Fig. 4. Suppose that substructure S_a shown in Fig. 4a occurs in the input graph 20 times. After expanding vertex *A* in all possible ways, the substructures shown in Fig. 4b, 4c, and 4d result. Since these are substructures of substructure S_a, they occur at least 20 times. For the sake of argument, suppose that these three substructures do not occur anywhere else in the input graph, and therefore they too have 20 occurrences. Hence, these three substructures would offer the same amount of compression, since all three of them have two vertices and one edge, and all of them occur the same number of times. The purging algorithm would check if substructure S_b can be extended with vertex *C* of S_c, and that substructure S_c can be extended with vertex *B* of S_b. Since this is the case, substructure S_c would be eliminated from the queue. Next this check is performed on substructure S_b (which is still on the queue), and substructure S_d. The result is similar, and S_d is also eliminated from further expansion. This leaves the queue with one substructure instead of three. Since further extensions of substructure S_b results in substructures that would result from the extensions of substructures S_c and S_d, the same search space is explored using fewer extensions.

The value-based queue and purging together enable searching a wider search space while examining potentially fewer substructures when compared to the fixed length queue. The savings offered by purging has been observed to be substantial since the case described above arises almost every time a substructure is extended. The actual savings depend on particular graphs, the main factor being the connectivity of the graph. The more connected the graph, the more savings purging offers.

3.5. New features in SUBDUE

Throughout this research a number of improvements have been prompted either by the

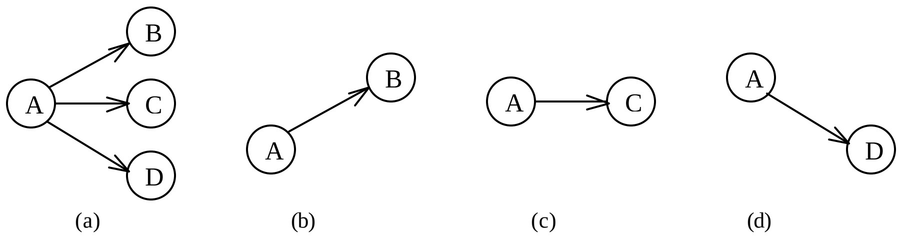

Fig. 4. Purging substructures from the queue; (a) best substructure S; (b) substructure of S; (c) substructure of S; (d) substructure of S.

research subject directly, or by ease of use or other reasons. This section describes some of these improvements.

New command line arguments that are related to cluster analysis are *–cluster*, *–truelabel* and *–exhaust*. Another option, *–savesub*, and an extra output level were also added to facilitate clustering, but those can also be used separately from clustering.

The option *–cluster* turns on cluster analysis in SUBDUE. Cluster analysis is described in detail in section 4. This option produces a classification lattice in the file "*inputFileName*.dot" that can be viewed with the GRAPHVIZ graph visualization package.[19] The option *–truelabel* will print the cluster definition into each node of the classification lattice when viewed with Dotty, part of the GRAPHVIZ package. The option *–exhaust* will prevent SUBDUE from stopping after discovering all substructures that can compress the graph, and have it continue until the input graph is compressed into a single vertex. To help evaluate the quality of clusterings the *–savesub* option was introduced. This option saves the definition and all the instances of the best substructure in all of the iterations. When clustering is enabled, it also saves the classification lattice hierarchy that can be used to reconstruct it. These files may be used with a tool specially written for evaluating clusterings. An extra output level was also added to display only the essential information concerned with clustering during the discovery process. The new output level is level 1, increasing the previous output levels by 1, making the default output level 2. All of these options are discussed in more detail in section 4.

There are also a few new command line arguments that are not concerned with clustering. The option *–plot fileName* saves information about the discovery process in the file called *fileName* that can later be plotted using any spreadsheet software, like Microsoft Excel. The information saved includes the iteration number, a number assigned to each substructure evaluated in each iteration, the number of vertices the particular substructure has, the description length of the input graph compressed with the substructure, and the compression offered by the substructure. In addition, if timing is enabled, it will also save various timings taken during the discovery process.

The option *–prune2 number* keeps track of local minima. The parameter *number* specifies how many more extensions are to be done after identifying a local minimum. This option is selected by default for clustering with the argument 2. Its benefits are described in more detail in section 4, in the context of clustering.

The option *–oldeval* enables the original MDL computation over the newer one. The original evaluation has the disadvantage of false compression, discussed earlier.

Another change is the removal of the command line argument *–nominencode*, which enabled the computation of the compression based solely on the size of the graph measured by adding up the number of edges and vertices. This option was observed to make SUBDUE run faster, but less successfully. With the introduction of the value-based queue and purging, however, this had to be removed since it was mostly incompatible with those features. Also, these features made SUBDUE run much faster, which, in turn, made this option less useful.

3.6. Other features

SUBDUE supports biasing the discovery process. Predefined substructures can be provided to SUBDUE, which will try to find and expand these substructures, this way "jump-starting" the discovery. The inclusion of background knowledge proved to be of great benefit.[20] SUBDUE also supports supervised learning, where positive and negative examples are provided to the system. Substructures found that are similar to positive examples are given a higher value, while substructures similar to the negative example graphs are penalized. This way of influencing the discovery process has proven successful, an example of which is the application of SUBDUE to the chemical toxicity domain.[3]

4. Hierarchical Conceptual Clustering of Structural Data

Section 3.2 describe the SUBDUE structural knowledge discovery system in detail. The main goal of the work described here was to extend the system to include cluster analysis functionalities. This section describes our approach to conceptual clustering of structural data and its implementation in SUBDUE.

Cluster analysis with SUBDUE uses the main SUBDUE algorithm to discover clusters.[21] These are then used to build a hierarchy of clusters to describe the input graph. The following subsections describe the theoretical background and inner workings of SUBDUE's clustering functionality.

4.1. Identifying clusters

The SUBDUE algorithm takes one iteration to find a substructure that compresses the input graph the best. The definition of the best substructure after a single iteration yields the definition of a *cluster*. The membership of the cluster is all the instances of the substructure in the input graph.

Within a single iteration SUBDUE has several ways to decide when to stop. SUBDUE always has a single best substructure at the head of the queue, so in effect it could stop at any point. SUBDUE has a *limit* which tells it how many substructures to consider at most in a single iteration. By default, the limit is set to the sum of the number of vertices and edges, divided by two. This number has been observed to be sufficiently large to allow the discovery of the best substructure. The trick, of course, is to stop the discovery process right after the best substructure is discovered during the iteration. A new feature, *–prune2*, attempts to do just that. This option keeps track of minima, and when one is found, it lets SUBDUE continue for a limited number of substructure extensions. If a new minimum is found during this time, the count is reset and SUBDUE is allowed to go a little further. This assures that each iteration of SUBDUE returns the substructure that is responsible for the first local minimum. As discussed later, this is just what the clustering algorithm needs. Since *prune2* will stop the discovery, setting a limit is not necessary when *prune2* is used. This is the default for cluster analysis.

4.2. *Creating hierarchies of clusters*

After each iteration SUBDUE can be instructed to physically replace each occurrence of the best substructure by a single vertex, this way compressing the graph. The resulting compressed graph can then be used as the new input graph and be given to SUBDUE to discover a substructure that compresses this graph the best.

This iterative approach to clustering imposes more and more hierarchy on the database with each successive iteration. Using the fact that each new substructure discovered—in successive iterations—may be defined in terms of previously discovered substructures, a hierarchy of clusters can be constructed. Since by default the number of iterations SUBDUE performs is one, when clustering is enabled the number of iterations is set to indefinite. This means that SUBDUE iterates until the best substructure in the last iteration does not compress the graph. If the *–exhaust* option is enabled, SUBDUE iterates until the input graph is compressed into a single vertex. This default behavior may be overridden by explicitly specifying the number of iterations to be done, in essence specifying the number of clusters to be discovered.

Hierarchies are usually pictured as various forms of a tree, as found in many previous works on hierarchical clustering. This research found, however, that in structured domains a strict tree representation is indeed inadequate. In those cases a lattice-like structure emerges instead of a tree. Therefore, newly discovered clusters are used to build a *classification lattice*.

The classification lattice is the consequence of the previously mentioned fact that any cluster definition—except for the very first one—may contain previously defined clusters. If a cluster definition does not contain any other clusters, it is inserted as the child of the root. If it contains one or more instances of another cluster it is inserted as the child of that cluster, the number of branches indicating the number of times the cluster is in the definition of the child cluster. If the cluster definition includes more than one other cluster, then it is inserted as the child for all of those clusters.

To provide an example of the explanation above, the generation of the hierarchical conceptual clustering for the artificial domain (shown in Fig. 11) is demonstrated here. SUBDUE in the first iteration discovers the substructure that describes the pentagon pattern in the input graph. This comprises the first cluster C_p. This cluster is inserted as a child for the root node. The resulting classification lattice is shown in Fig. 5a. In iterations 2 and 3 the square shape (cluster C_s) and the triangle shape (cluster C_t) are discovered, respectively. These are inserted at the root as well, since C_s does not contain C_p in its definition, and C_t does not contain either C_p or C_s. The resulting clustering is shown in Fig. 5b.

All of the basic shapes (pentagon, square and triangle) appear four times in the input graph. So why is it that they are discovered in the order described above? Since all of them have the same number of instances in the graph, the size of the substructure will decide how much they are capable of compressing the input graph. The substructure describing the pentagon has five vertices and five edges, that of the square has four

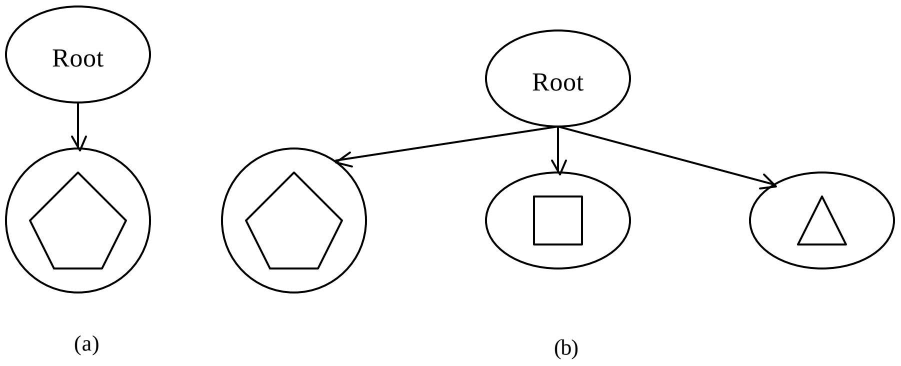

Fig. 5. Clustering of the artificial domain after one iteration (a) and after three iterations (b).

vertices and four edges, and that of the triangle has three vertices and three edges. Given the same number of instances, the bigger substructure will compress the input graph better.

In the fourth iteration SUBDUE deems the substructure the best that describes two pentagon shapes connected by a single edge. There are two of these formations in the graph, not four, as one might think, since no overlapping of instances are permitted. This cluster is inserted into the classification lattice as the child of the cluster describing the pentagon, since that cluster appears in its definition. The resulting classification lattice is shown in Fig. 6. There are two links connecting this new cluster to its parent, because the parent cluster definition appears twice.

In iteration 5 a substructure is discovered that contains a pair of squares connected by an edge, a pair of triangles connected by an edge, and these two pair are connected by a single edge. This substructure has two instances in the input graph. This cluster is

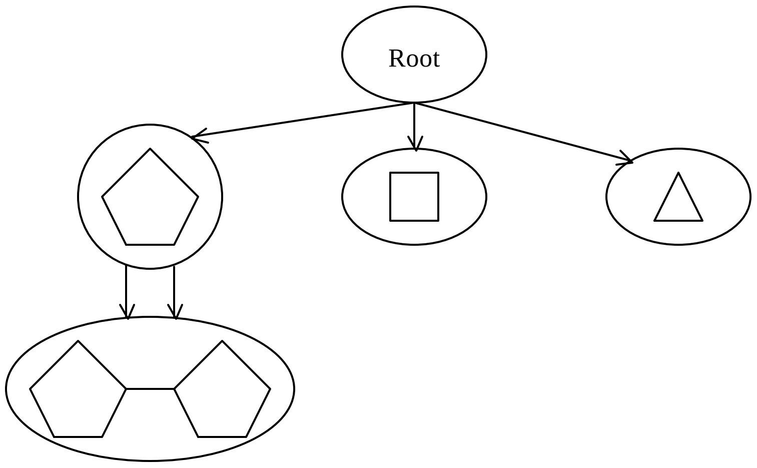

Fig. 6. Clustering of the artificial domain after four iterations.

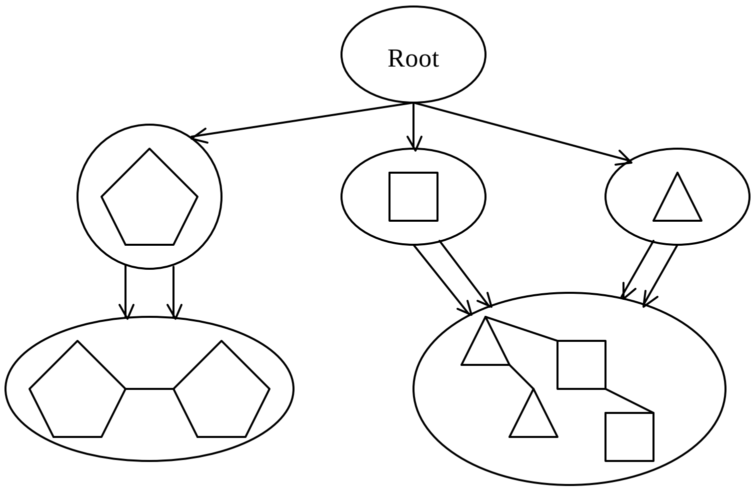

Fig. 7. Clustering of the artificial domain after five iterations.

inserted as a child of two clusters in the first level of the lattice, which appear in the definition of this new cluster. The new lattice is depicted in Fig. 7. Since both parent cluster definitions appear twice in the new cluster, there are two links from each of those parents to the new node.

4.3. First minimum heuristic

SUBDUE searches the hypothesis space of classification lattices. During each iteration of the search process (that is, while searching for each cluster), numerous local minima are encountered. The global minimum, however, tends to be one of the first few minima. For clustering purposes the first local minimum is used as the best cluster definition. The reason for this is easy to see. SUBDUE starts with all the single-vertex instances of all unique substructures, and iteratively expands the best ones by a single edge. The local minimum encountered first is therefore caused by a smaller substructure with more instances than the next local minimum, which must be larger, and have fewer instances. A smaller substructure is more general than a larger one, and should be a parent node in the classification lattice for any more specific clusters.

A good example is shown in Fig. 8. The horizontal axis of the plot shows the number of the substructure being evaluated, and the vertical axis shows the compression offered by the substructures. Fig. 8 has only one minimum, appearing at substructure number 37. The iteration appears to have several minima during the first portion of the iteration. Those minima, however, are caused by the dissimilarities in compression among the substructures on the queue in each search state. For instance, if the maximum queue length is set to be four, then there will be approximately four substructures in the queue after each extension. These four substructures will offer different compressions, the first

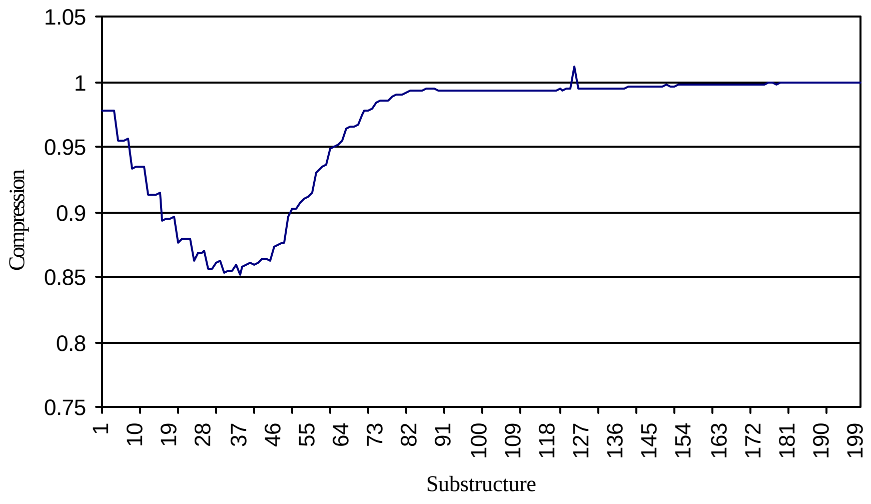

Fig. 8. Compression of substructures as considered during one iteration of SUBDUE.

in the queue offering the most, the last in the queue offering the least. This is reflected in Fig. 8. The staircase-like formation at the beginning of the iteration shows quite similar substructures in the queue. Each step of the staircase represents the compression values of the queue at each search state from left to right, the leftmost value representing the substructure at the head of the queue. As the iteration moves along we can see that the head of the queue offers more and more compression than the tail, resulting in local minima. The prune2 feature, however, does not consider fluctuations within each search state, but rather between states. In other words, minima are determined by looking at the best substructure in each search state in successive iterations. The first local minimum therefore occurs at substructure number 37. This minimum turns out to be the global minimum as well for this iteration.

As a different example, Fig. 9 shows the compression of substructures as discovered by SUBDUE in an aircraft safety database. The details of the database are not important here. The search depicted in Fig. 9 features numerous local minima, the first one occurring at substructure number 46. This is not the global minimum, but for clustering purposes this one will be used as the best substructure—for reasons described earlier.

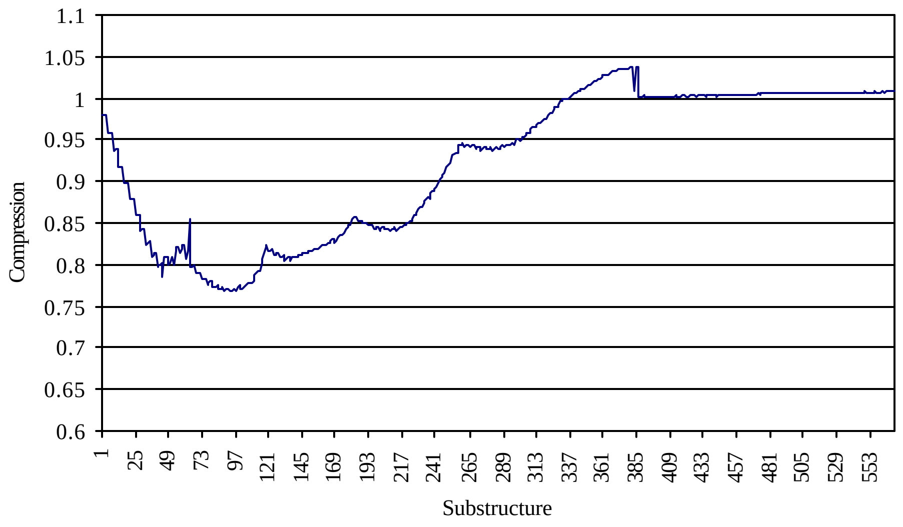

Fig. 9. Compression of substructures as considered by SUBDUE during one iteration on an aircraft safety database.

Even though it is possible to use the global minimum as the best substructure, we found that if the global minimum is not the first local minimum, it may produce *overlapping clusters*. Overlapping clusters are those that include the same information. For example, in a particular clustering of the vehicles domain two clusters may include the information “number of wheels: 4”. This suggests that perhaps a better clustering may be constructed in which this information is part of a cluster at a higher level.

4.4. Implementation

This section discusses the implementation details for cluster analysis in SUBDUE. Most of the clustering functionalities center around building, printing and finally destroying the classification lattice. We will also describe the Dotty visualization package with emphasis on interpreting the classification lattice displayed by Dotty.

A classification lattice describes a hierarchical conceptual clustering of a database. Each node in the lattice represents a cluster. The classification lattice is a tree-like data structure that has the special property that one node may have several parents. One obvious exception is the root node that does not have any parents. Information stored in a node includes pointers to children, number of children, number of parents, the substructure label, a descriptive label and a shape flag. Some of these need explanation.

The substructure label is the vertex label assigned to the substructure that represents the cluster. This label is assigned to the substructure when compressing the input graph, and replacing each occurrence of the substructure with a single vertex. This information is useful for identifying the parents of a cluster.

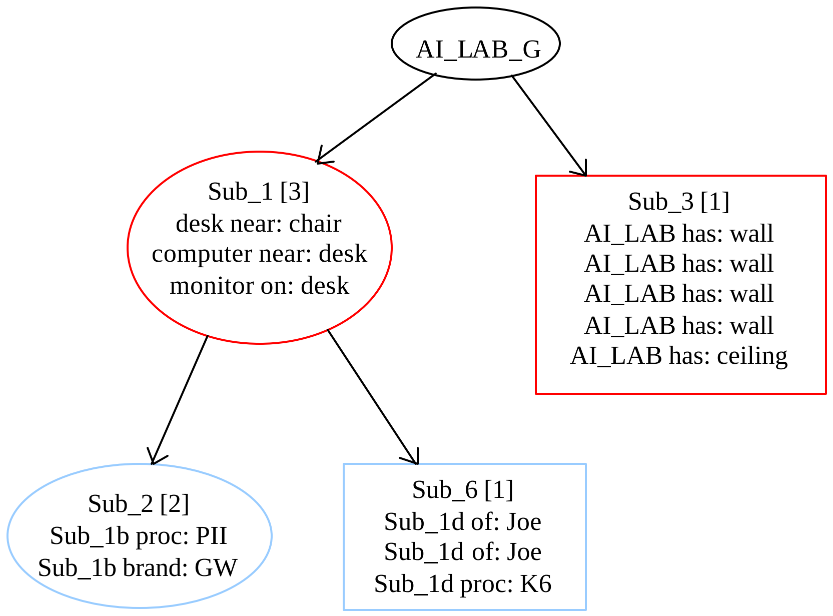

Fig. 10. Example of a classification lattice produced by SUBDUE and visualized by dotty.

The descriptive label holds information about the cluster definition in an easy-to-read format. This has significance when displaying the lattice with Dotty. The label is generated when the –*truelabel* option is set by taking all pairs of vertices connected by an edge, and printing in the format `*sourceVertex edge: targetVertex*.'

For example, if a certain substructure contains the two vertices labeled *car* and *red*, connected by an edge labeled *color*, the descriptive label would read *car color: red*. The information in the descriptive label reads well for many domains.

The shape flag determines the shape of the cluster when displayed by Dotty. The shape of the cluster is just another visual aid in interpreting the clustering. By default, all clusters are displayed having an oval shape. When the –*exhaust* option is set, however, SUBDUE is instructed to form clusters out of substructures that do not compress the input graph further, and these clusters are given a rectangular shape. In this case it is nice to be able to distinguish the compressing clusters from the non-compressing ones.

4.5. Visualization

For a more sophisticated appearance the GRAPHVIZ graph visualization package is used.[19] When clustering is enabled in SUBDUE, a file with the *.dot* extension is created. This file can be used by the program *dot* to create a PostScript file, or by *dotty* to view it interactively. From dotty one can directly print the lattice to a printer or a file. Dotty also allows the rearrangement of clusters, and changing cluster parameters. Fig. 10 shows a portion of a classification lattice that is suitable to use as an example.

The root node contains the file name of the input graph, slightly modified. Characters like comma, period, colon, semicolon, slash and backslash are replaced by the underline character, since dotty cannot handle those characters. The root node does not contain any other information.

Nodes other than the root node contain the sub-label of the substructure that defines the cluster, the number of instances the substructure has in the input graph (shown in brackets), and a series of descriptive labels. Each line, except for the first one has a descriptive label.

Clusters on the same level have the same color. In some cases the lattice can become highly interconnected, and loses the shape of a tree. The colors help to identify levels in those cases.

5. Results

This section describes the results of cluster analysis using SUBDUE. First the algorithm's proper behavior is established using an artificially generated database as the test domain. Next the algorithm is compared to existing systems. Other applications of the algorithm are also discussed.

5.1. Validation in an artificial domain

An artificial domain will serve as an example to demonstrate SUBDUE's ability to generate valid clusterings in structured databases. This artificial domain is depicted in its graph form in Fig. 11, where only edges are shown. Vertices are at the meeting points of the edges. This graph demonstrates regular and irregular patterns found in structured databases. Smaller, clearly recognizable shapes—triangles, squares and pentagons—are embedded in the graph. They are organized into rings, and some edges are added between some of the triangles and squares to somewhat disturb the regularity. The vertices in the

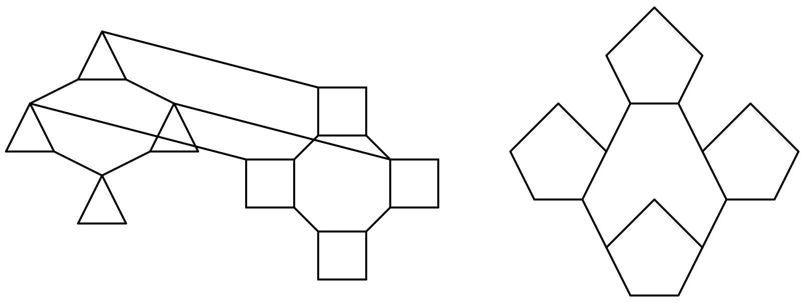

Fig. 11. Artificial domain.

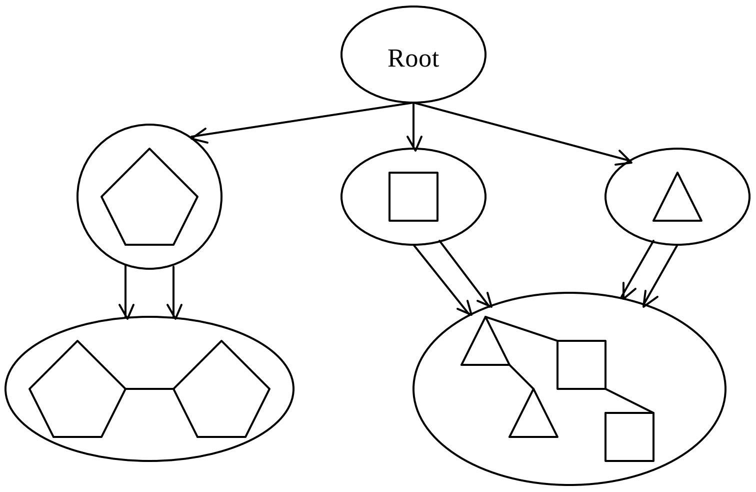

Fig. 12. Hierarchical clustering of the artificial domain.

graph are labeled as *a*, *b*, *c*, and so on, for each primitive shape. Edge labels are as follows: for each primitive object the sides are labeled as T_side, S_side and P_side, for triangle, square and pentagon, respectively. The edges connecting these primitive objects are labeled as T_link, S_link and P_link. Edges connecting different shapes are labeled TS (for triangle-square link).

SUBDUE was invoked using the command

```
Subdue -cluster -truelabel -prune2 1 artif-tsp2.g
```

where *-cluster* enables clustering, *-truelabel* enables the descriptive labels, and *-prune2* 1 overrides the default option for clustering, *-prune2* 2. This results in increased sensitivity to local minima, which is more desirable in smaller databases like this one. We have observed that in general the larger and more complex the database, the more clearly defined the local minima.

The classification lattice generated by SUBDUE is shown in Fig. 12. For clarity, the substructures are shown that define the clusters rather than the textual description extracted from the graph representation (as in Fig. 10).

The lattice closely resembles a tree, with the exception that a node (bottom-right) has two parents. As the figure shows, smaller, more commonly occurring structures are found first that compose the first level of the lattice. These account for most of the graph, therefore, they are the most general clusters. Subsequently identified clusters are based on these smaller clusters that are either combined with each other, or with other vertices or edges to form new, more specific clusters. This can clearly be seen in the second level of the lattice where two pentagons and a connecting edge comprise a new cluster (bottom-left), and a pair of triangles and a pair of squares comprise another cluster along with three additional connecting edges. Both of the clusters in the second level have two instances.

The second level nodes in the classification lattice are connected with two branches from their parents. This means that there are two pentagons used in the bottom-left cluster, and two triangles and two squares are used in the bottom right cluster. This is simply a visual aid that helps the researcher.

SUBDUE performs as expected on this artificial domain. It was able to find the most commonly embedded structures, and construct the expected classification lattice. To further support the algorithm's validity, the following section compares SUBDUE to an existing hierarchical clustering system.

5.2. Comparison to other systems

A small experiment devised by Fisher[8] can serve as a basis for comparison of SUBDUE and COBWEB. This example will also demonstrate SUBDUE's performance on unstructured data.

The database used for the experiment is given in Table 1. The animal domain is represented in SUBDUE as a graph, where attribute names (like *Name* and *BodyCover*) were mapped to labeled edges, and attribute values (like *mammal* and *hair*) were mapped to labeled vertices, as suggested in section 3.1. An example of the representation was given in Fig. 1, where the *mammal* instance is depicted in graph form.

Table 1. Animal Descriptions.

Name	***Body Cover***	***Heart Chamber***	***Body Temp.***	***Fertilization***
mammal	hair	four	regulated	internal
bird	feathers	four	regulated	internal
reptile	cornified-skin	imperfect-four	unregulated	internal
amphibian	moist-skin	three	unregulated	external
fish	scales	two	unregulated	external

COBWEB produces the classification tree shown in Fig. 13, as reported by Fisher.[8] SUBDUE generated the hierarchical clustering shown in Fig. 14.

SUBDUE's result is similar to that of COBWEB's. The "*mammal/bird*" branch is clearly the same. Amphibians and fish are grouped in the same cluster based on their external fertilization, which is done the same way by COBWEB. SUBDUE, however, incorporates reptiles with amphibians and fish, based on their commonality in unregulated body temperature. This clustering of the animal domain seems better, since SUBDUE eliminated the overlap between the two clusters (*reptile* and *amphibian/fish*) by

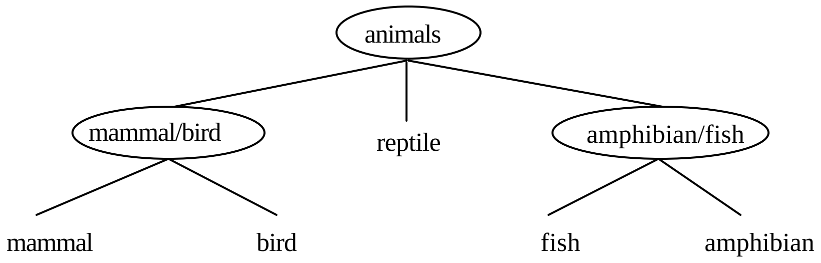

Fig. 13. Hierarchical clustering over animal descriptions by COBWEB.

creating a common parent for them that describes this common trait. This example

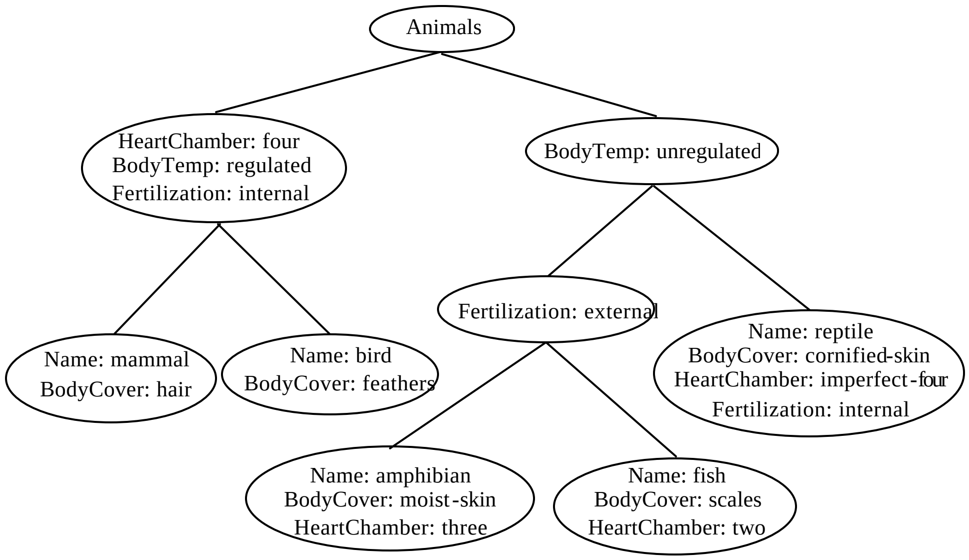

Fig. 14. Hierarchical clustering over animal descriptions by SUBDUE.

demonstrates that SUBDUE is capable of dealing with unstructured domains successfully.

5.3. Applications

Another array of practical applications can be found in the field of chemistry. SUBDUE (not using the clustering functionality) has been used to find potential gene regulatory sequences in DNA,[4] to identify structural regularities in proteins,[22] and to predict the carcinogenecity of various chemical compounds.[3]

Clustering with SUBDUE might also be useful in chemistry. In the following example a portion of a DNA sequence is described by clustering. This portion of the DNA is shown in Fig. 15. To represent the DNA as a graph, atoms and small molecules (like CH_2) are mapped to vertices, and bonds are represented by undirected edges. The edges

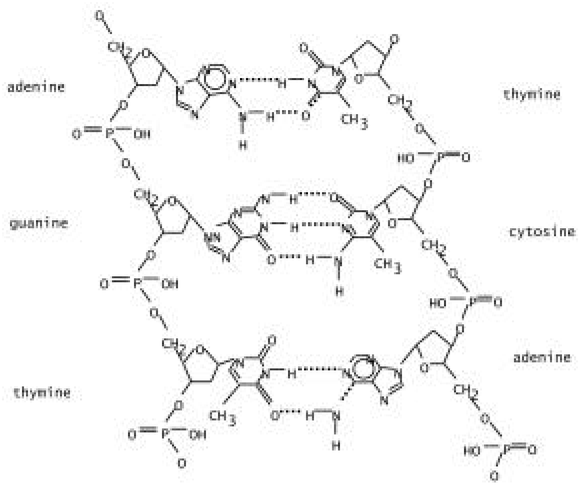

Fig. 15. Portion of a DNA sequence.

are labeled according to the type of bond, single or double. A portion of the classification lattice generated by SUBDUE is shown in Fig. 16. As with the artificial domain, the chemical compounds defined by the clusters are shown, rather than the textual description extracted from the graph representation of the DNA.

The lattice property of the classification lattice is apparent in Fig. 16, where the

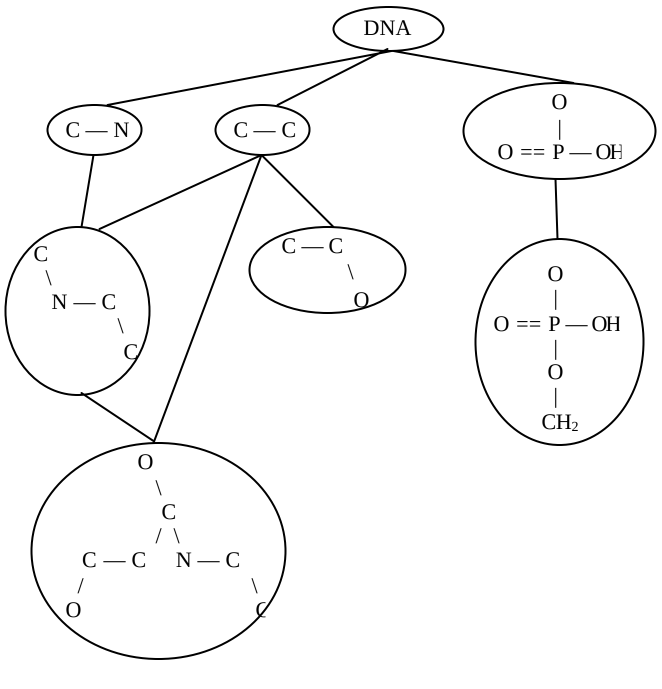

Fig. 16. Partial hierarchical clustering of the DNA sequence.

bottom-left nodes have multiple parents. This lattice describes 71% of the DNA sequence shown in Fig. 15. As the figure shows, smaller, more commonly occurring compounds are found first that compose the first level of the lattice. These account for more than 61% of the DNA. Subsequently identified clusters are based on these smaller clusters that are either combined with each other, or with other atoms or molecules to form a new cluster. The second level of the lattice extends the conceptual clustering description such that an additional 7% of the DNA is covered.

5.4. Evaluation

The previous sections have shown that SUBDUE's clustering functionality is appealing in many respects. SUBDUE has performed according to expectations in an artificial structured domain, has paralleled an existing system in an unstructured domain, and has discovered clusterings in real-world domains. The arguments made towards SUBDUE's success, however, have been based purely on human observers' opinion.

Evaluation of clustering systems has always been anecdotal, lacking the existence of an objective evaluation measure. We are developing such an objective measure which will be the subject of a subsequent publication. Until then, however, let us review some of the properties of good clusterings.

The best clustering is usually the one that has the minimum number of clusters, with minimum overlap between clusters, such that the entire data set is described. Too many clusters can arise if the clustering algorithm fails to generalize enough in the upper levels of the hierarchy, in which case the classification lattice may become shallow with a high branching factor from the root, and a greater amount of overlap. On the other extreme, if the algorithm fails to account for the most specific cases, the classification lattice may not describe the data entirely. Experimental results indicate that SUBDUE finds clusterings that effectively trade off these extremes.

6. Discussion and Conclusions

This work set out to explore the mostly uncharted territory of hierarchical conceptual clustering in discrete-valued structural databases. There have been numerous attempts at clustering. Most of these, however, were applicable only in unstructured domains that simply enlist object descriptions. SUBDUE overcomes this restriction by representing databases using graphs, which allows for the representation of a large number of relationships between objects.

The technique of cluster analysis is of unquestionable importance. This is demonstrated by the wide variety of fields in which this technique is used, and the different names by which it has been referred to. Many real world domains are unstructured, like a listing of animals and their traits, but many are structured, like a DNA strand. Cluster analysis is equally applicable to both types of databases. A modern data mining system must be able to handle these different types of data, and operate on them successfully. In fact, many unstructured data sets may be made structured by a simple preprocessing algorithm. An example of this might be the establishment of relationships among books with the same author in the domain of book listings, or the creation of *near* and *far* relationships, both spatial and temporal, between events in a log of earthquakes. In doing so a data set can be made more valuable from a data mining point of view.

SUBDUE has been demonstrated to be a successful multipurpose data mining tool in the most diverse of domains. Since clustering can be applied to any data set that SUBDUE can handle, clustering is a very important addition in functionality to SUBDUE as has been demonstrated using various examples.

One of the major contributions of this work is the synthesis of the classification lattice. Previous work in clustering suggested classification trees, which are inadequate in structured domains. On the other hand, a classification lattice in unstructured domains reduces to a tree, which suggests that classification trees are a proper subset of classification lattices.

Future work on SUBDUE includes defining hierarchical clusterings of other real-world domains, and comparisons to other clustering systems. As mentioned earlier, we are developing an objective quality measure for hierarchical, conceptual, structural clusterings. Such a measure will allow a better technique for comparison of results between systems and may eventually serve as a heuristic to guide SUBDUE's search for the best clustering.

Acknowledgements

This research was supported by National Science Foundation grant IRI-9615272 and the State of Texas Higher Education Coordinating Board Advanced Technology Program grant 003656-45.

References

[1] L. B. Holder and D. J. Cook, *Discovery of Inexact Concepts from Structural Data*, IEEE Transactions on Knowledge and Data Engineering **5:6** (1993) 992-994.

[2] J. A. Gonzalez, L. B. Holder, and D. J. Cook, *Structural Knowledge Discovery Used to Analyze Earthquake Activity*, Proceedings of the Thirteenth Annual Florida AI Research Symposium (2000).

[3] R. Chittimoori, L. B. Holder, and D. J. Cook, *Applying the Subdue Substructure Discovery System to the Chemical Toxicity Domain*, Proceedings of the Twelfth International Florida AI Research Society Conference (1999) 90-94.

[4] R. Maglothin, *Data Mining In DNA: Using The Subdue Knowledge Discovery System To Find Potential Gene Regulatory Sequences*, Masters Thesis, Department of Computer Science and Engineering, UTA (1999).

[5] G. H. Ball, *Classification Analysis*, Stanford Research Institute SRI Project 5533 (1971).

[6] B. S. Everitt, *Cluster Analysis*, Wiley & Sons, New York (1980).

[7] R. S. Michalski, *Knowledge acquisition through conceptual clustering: A theoretical framework and algorithm for partitioning data into conjunctive concepts*, International Journal of Policy Analysis and Information Systems **4** (1980) 219-243.

[8] D. H. Fisher, *Knowledge Acquisition Via Incremental Conceptual Clustering*, Machine Learning **2** (1987) 139-172.

[9] R. Schalkoff, *Pattern Recognition*. Wiley & Sons, New York (1992).

[10] K. Thompson and P. Langley, *Concept formation in structured domains*, In D. H. Fisher and M. Pazzani (Eds.), Concept Formation: Knowledge and Experience in Unsupervised Learning, Chap. 5. Morgan Kaufmann Publishers, Inc. (1991) 127-161.

[11] P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor, and D. Freeman, *AutoClass: A Bayesian classification system*, Proceedings of the Fifth International Workshop on Machine Learning (1988) 54–64.

[12] C. S. Wallace, and D. M. Boulton, *An Information Measure for Classification*, Computer Journal, **11 :2** (1968) 185-194.

[13] G. Karypis, E. Han and V. Kumar, *Chameleon: Hierarchical Clustering Using Dynamic Modeling*, Computer, (1999) 68-75.

[14] S. Guha, R. Rastogi and K. Shim, *CURE: An Efficient Clustering Algorithm for Large Databases*, ACM SIGMOD International Conference on Management of Data (1998).

[15] J. Rissanen, *Stochastic Complexity in Statistical Inquiry*, World Scientific Company (1989).

[16] D. J. Cook and L. B. Holder, *Substructure Discovery Using Minimum Description Length and Background Knowledge*, Journal of Artificial Intelligence Research **1** (1994) 231-255.

[17] J. R. Quinlan and R. L. Rivest, *Inferring decision trees using the minimum decription length principle*, Information and Computation **80** (1980) 227–248.

[18] H. Bunke and G. Allerman, *Inexact graph matching for structural pattern recognition*, Pattern Recognition Letters **1(4)** (1983) 245-253.

[19] E. Koutsofios and S. C. North, *Graphviz - graph drawing software*. www.research.att.com/sw/tools/graphviz (1999).

[20] S. Djoko, D. J. Cook and L. B. Holder, *An Empirical Study of Domain Knowledge and Its Benefits to Substructure Discovery*, IEEE Transactions on Knowledge and Data Engineering, **9:4** (1997) 575-586.

[21] I. Jonyer, L. B. Holder and D. J. Cook, *Graph-Based Hierarchical Conceptual Clustering,* Proceedings of the Thirteenth Annual Florida AI Research Symposium (2000).

[22] S. Su, *Applications Of Knowledge Discovery To Molecular Biology: Identifying Structural Regularities In Proteins*. Masters Thesis, University of Texas at Arlington (1998).