An Empirical Study of Domain Knowledge and Its Benefits to Substructure Discovery

Surnjani Djoko, Diane J. Cook and Lawrence B. Holder
University of Texas at Arlington
Department of Computer Science and Engineering
Box 19015, Arlington, TX 76019
E-mail:{djoko,cook,holder}@cse.uta.edu

Abstract — Discovering repetitive, interesting, and functional substructures in a structural database improves the ability to interpret and compress the data. However, scientists working with a database in their area of expertise often search for predetermined types of structures, or for structures exhibiting characteristics specific to the domain. This paper presents a method for guiding the discovery process with domain-specific knowledge. In this paper, the SUBDUE discovery system is used to evaluate the benefits of using domain knowledge to guide the discovery process. Domain knowledge is incorporated into SUBDUE following a single general methodology to guide the discovery process. Results show that domain-specific knowledge improves the search for substructures which are useful to the domain, and leads to greater compression of the data. To illustrate these benefits, examples and experiments from the computer programming, computer aided design circuit, and artificially-generated domains are presented.

Keywords — data mining, minimum description length principle, data compression, inexact graph match, domain knowledge

I. INTRODUCTION

With the increasing amount and complexity of today's data, there is an urgent need to accelerate discovery of information in databases. In response to this need, numerous approaches have been developed for discovering concepts in databases using a linear, attribute-value representation [1], [2], [3], [4], [5]. These approaches address issues of data relevance, missing data, noise, and utilization of domain knowledge. However, much of the data that is collected is structural in nature, or is composed of parts and relations between the parts. Hence, there exists a need for methods to analyze and discover concepts in structural databases.

Recently, we introduced a method for discovering substructures in structural databases using the minimum description length (MDL) principle [6]. The system is called SUBDUE, and it discovers substructures that compress the original data and represent structural concepts in the data. Once a substructure is discovered, the substructure is used to simplify the data by replacing instances of the substructure with a pointer to the newly discovered substructure. The discovered substructures allow abstraction over detailed structures in the original data. Iteration of the substructure discovery and replacement process constructs a hierarchical description of the structural data in terms of the discovered substructures. This hierarchy provides varying levels of interpretation that can be accessed based on the specific goals of the data analysis.

Although the MDL principle is useful for discovering substructures that maximize compression of the data, scientists often employ knowledge or assumptions of a specific domain to guide the discovery process. A domain-independent discovery method is valuable in that the discovery of unexpected substructures is not blocked. However, the discovered substructures might not be useful to the user. On the other hand, using domain-specific knowledge can assist the discovery process by focusing search and can also help make the discovered substructures more meaningful to the user. Hence, in order to trade off between domain-independent and domain-dependent discovery method, we incorporate domain knowledge into SUBDUE system, and combine both the domain-independent and domain-dependent method to guide the search toward the more appropriate substructures.

A variety of approaches to discovery using structural data have been proposed [7], [8], [9], [10], [11]. Many approaches use a knowledge base of concepts to classify the structural data. The purposes of the knowledge base in these systems are 1) to improve the performance of graph comparisons and retrieval, where the individual graphs are maintained in a partial ordering defined by the subgraph-of relation [7], [8], [9], 2) to deepen the hierarchical description, and 3) to group objects into more general concepts [7], [10], [11]. These systems perform concept learning over examples and categorization of observed data. However, the purpose of the SUBDUE system is to discover knowledge, and allows the use of both domain-independent heuristics and domain-dependent knowledge. In addition, the hierarchical knowledge base is used to help compress the database. While the above methods process individual objects one at a time, our method is designed to process the entire structural database, which consists of many objects.

This paper focuses on a method of realizing the benefits of domain-dependent discovery approaches by adding domain-specific knowledge to a domain-independent discovery system. Secondly, this paper explicitly evaluates the benefits and costs of utilizing domain-specific information. In particular, the performance of the SUBDUE system is measured with and without domain-specific knowledge along the performance dimensions of compression, time needed to discover the substructures, and usefulness of the discovered substructures.

The following sections describe the approach in detail. Section II introduces needed definitions. Section III

Supported by NASA grant NAS5-32337.

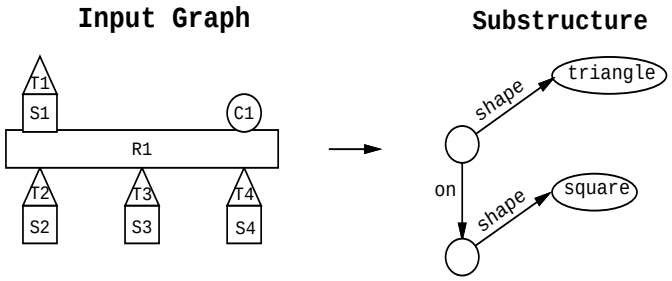
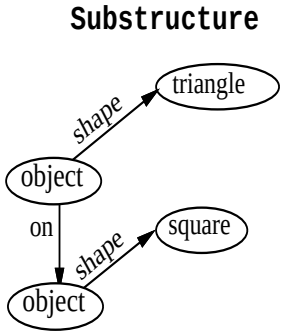


Fig. 1. Example substructure in graph form.



presents the inexact graph match algorithm employed by SUBDUE, and Section IV describes the minimum description length principle used by this approach, encoding scheme, and the discovery process. Section V describes methods of incorporating domain knowledge into the substructure discovery process. Section VI provides an analysis of the run-time complexity of SUBDUE. The evaluations detailed in Section VII, which demonstrate SUBDUE's ability to find substructures that compress the data and to re-discover known concepts in a variety of domains. We conclude with observations.

II. Structural Data Representation

The substructure discovery system represents structural data as a labeled graph. Objects in the data map to vertices or small subgraphs in the graph, and relationships between objects map to directed or undirected edges in the graph. A *substructure* is a connected subgraph within the graphical representation. This graphical representation serves as input to the substructure discovery system. Figure 1 shows a geometric example of such an input graph. The objects in the figure (e.g., `T1`, `S1`, `R1`) become labeled vertices in the graph, and the relationships (e.g., `on(T1,S1)`, `shape(C1,circle)`) become labeled edges in the graph. The graphical representation of the substructure discovered by SUBDUE from this data is also shown in Figure 1.

An *instance* of a substructure in an input graph is a set of vertices and edges from the input graph that match, graph theoretically, to the graphical representation of the substructure. For example, the instances of the substructure in Figure 1 are shown in Figure 2.

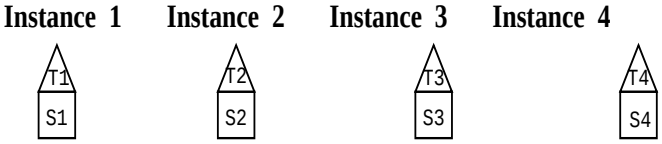


Fig. 2. Instances of the substructure.

III. Inexact Graph Match

The use of a graph as representation for data and concepts, requires methods for matching data to concepts. Methods of graph matching can be categorized into exact graph matching [11], and inexact matching based on graph distance or probability [12], transformation cost [13], [14], graph identity [15], and minimal representation criterion [9].

Although exact structure match can be used to find many interesting substructures, many of the substructures show up in a slightly different form throughout the data. These differences may be due to noise, distortion, or may just illustrate slight differences between instances of the same general class of structures.

Given an input graph and a set of defined substructures, we want to find those subgraphs of the input graph that most closely resemble the given substructures. To associate a measure between a pair of graphs consisting of a given substructure and a subgraph of the input graph, we adopt the approach of inexact graph match given by Bunke and Allermann [13]. In addition to that, we extend the Bunke's approach in order to speed up the search which will be described later in this section.

In this inexact match approach, each distortion of a graph is assigned a cost. A distortion is described in terms of basic transformations such as deletion, insertion, and substitution of vertices and edges. The distortion costs can be determined by the user to bias the match for or against particular types of distortions.

Given graphs g_1 with n vertices and g_2 with m vertices, $m \geq n$, the complexity of the full inexact graph match is $O(n^{m+1})$. Because this routine is used heavily throughout the discovery and evaluation process, the complexity of the algorithm can significantly degrade the performance of the system.

To improve the performance of the inexact graph match algorithm, we extend Bunke's approach by applying a branch-and-bound search to the tree. The cost from the root of the tree to a given vertex is computed as described above. Vertices are considered for pairings in order from the most heavily connected vertex to the least connected, as this constrains the remaining match. Because branch-and-bound search guarantees an optimal solution, the search ends as soon as the first complete mapping is found.

In addition, the user can place a limit on the number of search vertices considered by the branch-and-bound procedure (defined as a function of the size of the input graphs).

Once the number of vertices expanded in the search tree reaches the defined limit, the search resorts to hill climbing using the cost of the mapping so far as the measure for choosing the best vertex at a given level. By defining such a limit, significant speedup can be realized at the expense of accuracy for the computed match cost. A complete description of the inexact graph match procedure used by SUBDUE is provided in [16].

IV. SUBSTRUCTURE DISCOVERY USING MINIMUM DESCRIPTION LENGTH PRINCIPLE

The minimum description length (MDL) principle introduced by Rissanen [17] states that the best theory to describe a set of data is a theory which minimizes the description length of the entire data set. The MDL principle has been used for decision tree induction [5], image processing [18], [19], [20], concept learning from relational data [21], and learning models of non-homogeneous engineering domains [22].

We demonstrate how the minimum description length principle can be used to discover substructures in complex data. In particular, a substructure is evaluated based on how well it can compress the entire data set. We define the minimum description length of a graph to be the minimum number of bits necessary to completely describe the graph. SUBDUE searches for a substructure that minimizes $I(S) + I(G|S)$, where S is the discovered substructure, G is the input graph, $I(S)$ is the number of bits (description length) required to encode the discovered substructure, and $I(G|S)$ is the number of bits required to encode the input graph G with respect to S.

A. Graph Encoding Scheme

The graph connectivity can be represented by an adjacency matrix. Consider a graph that has n vertices, which are numbered $0, 1, \ldots, n-1$. An $n \times n$ adjacency matrix A can be formed with entry $A[i,j]$ set to 0 or 1. If $A[i,j] = 0$, then there is no connection from vertex i to vertex j. If $A[i,j] = 1$, then there is at least one connection from vertex i to vertex j. Undirected edges are recorded in only one entry of the matrix.

The encoding of the graph consists of the following steps. We assume that the decoder has a table of the l_u unique labels in the original graph G.

1. Determine the number of bits *vbits* needed to encode the vertex labels of the graph. First, we need $(\lg v)$ bits to encode the number of vertices v in the graph. Then, encoding the labels of all v vertices requires $(v \lg l_u)$ bits. We assume the vertices are specified in the same order they appear in the adjacency matrix. The total number of bits to encode the vertex labels is

$$vbits = \lg v + v \lg l_u.$$

2. Determine the number of bits *rbits* needed to encode the rows of the adjacency matrix A. Typically, in large graphs, a single vertex has edges to only a small percentage of the vertices in the entire graph. Therefore, a typical row in the adjacency matrix will have much fewer than v 1s, where v is the total number of vertices in the graph. We apply a variant of the coding scheme used by [5] to encode bit strings with length n consisting of k 1s and $(n-k)$ 0s, where $k \ll (n-k)$. In our case, row i $(1 \leq i \leq v)$ can be represented as a bit string of length v containing k_i 1s. If we let $b = \max_i k_i$, then the i^{th} row of the adjacency matrix can be encoded as follows:
   (a) Encoding the value of k_i requires $\lg(b+1)$ bits.
   (b) Given that only k_i 1s occur in the row bit string of length v, only $\binom{v}{k_i}$ strings of 0s and 1s are possible. Since all of these strings have equal probability of occurrence, $\lg \binom{v}{k_i}$ bits are needed to encode the positions of 1s in row i. The value of v is known from the vertex encoding.

   Finally, we need an additional $\lg(b+1)$ bits to encode the number of bits needed to specify the value of k_i for each row. The total encoding length in bits for the adjacency matrix is

$$\begin{aligned} rbits &= \lg(b+1) + \sum_{i=1}^{v} (\lg(b+1) + \lg \binom{v}{k_i}) \\ &= (v+1)\lg(b+1) + \sum_{i=1}^{v} \lg \binom{v}{k_i}. \end{aligned}$$

3. Determine the number of bits *ebits* needed to encode the edges represented by the entries $A[i,j] = 1$ of the adjacency matrix A. The number of bits needed to encode entry $A[i,j]$ is $(\lg m) + e(i,j)[1 + \lg l_u]$, where $e(i,j)$ is the actual number of edges between vertex i and j in the graph and $m = \max_{i,j} e(i,j)$. The $(\lg m)$ bits are needed to encode the number of edges between vertex i and j, and $[1+\lg l_u]$ bits are needed per edge to encode the edge label and whether the edge is directed or undirected. In addition to encoding the edges, we need to encode the number of bits $(\lg m)$ needed to specify the number of edges per entry. The total encoding of the edges is

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

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

B. Substructure Discovery without domain knowledge

The substructure discovery algorithm used by SUBDUE is a computationally-constrained beam search. The algorithm begins with an initial set of substructures matching every distinctly-labeled vertex in the graph. Each iteration

```
DiscoverSubstructure1(S, L)
 /* S = candidate substructures
    L = the user-defined limit on the number of substructures
        considered for expansion */

    D = {}
    n = 0
    S = sort candidate substructures by Compression
    while (n < L) and (S ≠ {}) do
        n = n + 1
        s = first(S)
        insert s into D sorted by Compression
        E = s extended in all possible ways
        for each e ∈ E do
            evaluate(e)     /* evaluate the Compression */
            if (Compression(e) < Compression(s)) /* pruning */
                insert e into S sorted by Compression
    return D
```

Fig. 3. The algorithm for the discovery without domain knowledge.

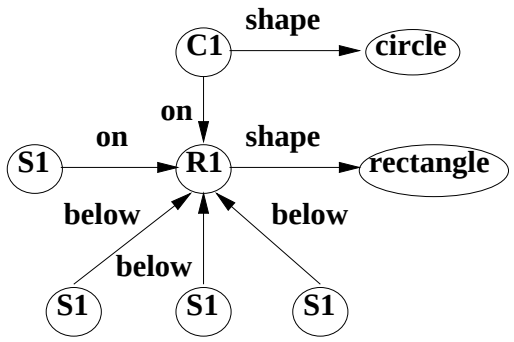


Fig. 4. The graph after compression using the discovered substructure S1.

through the algorithm selects the best substructure according to its ability to minimize the description length of the entire graph, and expands the instances of the best substructure by one neighboring edge in all possible ways. The new unique generated substructures become candidates for further expansion. The algorithm searches for the best substructure until all possible substructures have been considered or the total amount of computation exceeds a given limit. The evaluation of each substructure is guided by the MDL principle.

Once the description length (DL) of an expanding substructure begins to increase, further expansion of the substructure may not yield a smaller description length. As a result, SUBDUE makes use of an optional pruning mechanism that eliminates substructure expansions from consideration when the description lengths for these expansions increases (see Figure 3).

To represent an input graph using a discovered substructure involves additional overhead to replace the substructure's instances with a pointer to the newly-discovered substructure. Therefore, the number of bits needed to represent G, given the discovered substructure S, is

$$\begin{aligned} I(G|S) &= I(G) - \sum_{i=1}^{n} I(S) + \sum_{i=1}^{n} I(pointer) \\ &= I(G) - nI(S) + nI(pointer), \end{aligned}$$

where n represents the number of instances found for the discovered substructure. The second term represents the sum of bits saved over the discovered substructure, and the last term represents the sum of bits needed for the overhead.

We define a compression measure to evaluate a substructure's ability to compress an input graph as the following:

$$Compression = 1 - \left(\frac{DL\ of\ compressed\ graph}{DL\ of\ original\ graph}\right),$$

where $DL\ of\ compressed\ graph$ is $I(G|S) + I(S)$, and $DL\ of\ original\ graph$ is $I(G)$. If $Compression$ is greater than zero, the representation of G using S is used instead of the original representation, since it required less bits.

Both the input graph and the discovered substructure can be encoded using the above encoding scheme. After a substructure is discovered, each instance of the substructure in the input graph is replaced by a single vertex representing the entire substructure. Figure 4 showed compression of the input graph in Figure 1 using the discovered substructure.

V. Adding domain knowledge to the SUBDUE system

The SUBDUE discovery system was initially developed using only domain independent heuristics to evaluate potential substructures. As a result, some of the discovered substructures may not be useful and relevant to specific domains of interest. For instance, in a programming domain, the BEGIN and END statements may appear repetitively within a program; however, they do not perform any meaningful function on their own; hence they exhibit limited usefulness. Similarly, in the CAD circuit domain, some subcircuits or substructures may appear repetitively within the data; however, they may not perform meaningful functions within the domain of usage. To make SUBDUE's discovered substructures more interesting and useful across a wide variety of domains, domain knowledge is added to guide the discovery process. Furthermore, compressing the graph using the domain knowledge can increase the chance of realizing greater compression than without using the domain knowledge.

In this section we present two types of domain knowledge that are used in the discovery process and explain how they bias discovery toward certain types of substructures.

A. Model/Structure knowledge

Model/Structure knowledge provides to the discovery system specific types of structures that are likely to exist in a database and that are of particular interest to a scientist using the system. The model knowledge is organized in a hierarchy that specifies the connection between individual structures. Nodes of the hierarchical graph can be classified as either primitive (nondecomposable) or nonprimitive. The primitive nodes reside in the lowest level,

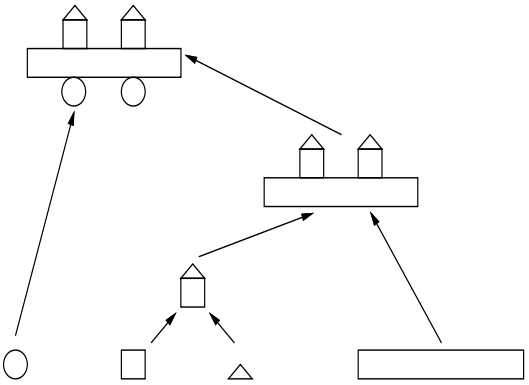


Fig. 5. A hierarchical graph.

```
DiscoverSubstructure2(S)        /* S = initial substructure */
  D = {}
  S = sort initial substructures by Compression
  while (S ≠ {}) do
    candidate = first(S)
    M = find candidate's models
    while (size(candidate) ≤ size(M)) do
      E = candidate extended in all possible ways
      for each substructure e ∈ E do
        for each model m ∈ M do
          if (gmatch(e, m) ≤ Threshold) then
          /* inexact whole graph match */
            evaluate(e)
            insert e into L sorted by Compression
            return D
          else if (subgmatch(e, m) ≤ Threshold) then
          /* inexact subgraph match */
            evaluate(e)
            insert e into S sorted by Compression
          else discard e
  return D
```

Fig. 6. The algorithm for discovery using the domain knowledge.

i.e., the leaves, and all nonprimitive nodes reside in the higher levels of the hierarchy. The primitive nodes represent basic elements of the domain, whereas the nonprimitive nodes represent models or structures which consist of a conglomeration of primitive nodes and/or lower-level nonprimitive . The higher the node's level, the more complex is the structure it represents. The hierarchy for a particular domain is supplied by a domain expert. The structures in the hierarchy and their functionalities are well known in the context of that domain. This knowledge is formed in a bottom-up fashion. Users can extend the hierarchy by adding new models.

To illustrate the structure knowledge, a simple example is shown in Figure 5, representing a hierarchical graph based on the shape structure. The primitive nodes are triangle, square, circle and rectangle. The nonprimitive nodes are built upon the primitive nodes and/or nonprimitive nodes. While Figure 5 represents a hierarchy built using commonalities between individuals' shape, in the programming and computer aided design (CAD) circuit domain, the hierarchical graphs are built based on commonalities between individuals' functional structure. For example, in the programming domain, special symbols and reserved words are represented by primitive nodes, and functional subroutines (e.g., swap, sort, increment) are represented by nonprimitive nodes. In the CAD circuit domain, basic components of a circuit (e.g., resistor, transistor) are represented by primitive nodes, and functional subcircuits such as operational amplifier, filter, etc. are represented by non primitive nodes. This hierarchical representation allows examining of the structure knowledge at various level of abstraction, focusing the search and reducing the search space.

A.1 Using model/structure knowledge to guide the discovery

Although the minimum description length principle still drives the discovery process, domain knowledge is used to glean certain types of known structures from the input graph. First, the modified version of SUBDUE can be biased to look specifically for structures of the type specified in the model hierarchy. The discovery process begins by matching a single vertex in the input graph to primitive nodes of the model knowledge hierarchy. If the primitive nodes do not match the input vertices, the higher level nodes of the hierarchy are pursued. The models in the hierarchy pointed to by the matched vertex in the input graph are selected as candidate models and are matched with the input substructure. Each iteration through the process, SUBDUE selects a substructure from the input graph which provides the best match to the one of selected models, and which can be used to compress the input graph. The match can either be a subgraph match or a whole graph match. If the match is a subgraph match, SUBDUE expands the instances of the best substructure by one neighboring edge in all possible ways. The newly generated substructure becomes a candidate for the next iteration. However, if the match is a whole graph match, the process has found the desired substructure, and the chosen substructure is used to compress the entire input graph. The process continues to expand the substructure until either a substructure has been found or all possible substructures have been considered (see Figure 6).

To represent an input graph using a discovered substructure from the model hierarchy, the representation incurs additional overhead to replace the substructure's instances with a pointer to the model hierarchy. In addition to this overhead, some domains involve extra parameters. Consider an example in the programming domain where a substructure of the model hierarchy (e.g., $Sort(a, b)$, where a and b are dummy variables) is discovered in a program. SUBDUE replaces each of the discovered substructure's instances with $Sort(a_i, b_i)$, where Sort is a pointer to the model hierarchy, and a_i and b_i are parameters of the ith instance.

Therefore, the number of bits needed to represent G, given substructure S which matches model M, is

$$I(G|M) = I(G) - \sum_{i=1}^{n} I(S) + \sum_{i=1}^{n} I(pointer) + \sum_{i=1}^{n} I(parameters_i)$$

$$= \quad I(G) - nI(S) + nI(pointer) + \sum_{i=1}^{n} I(parameters_i),$$

where n represents the number of instances found for the discovered substructures. The second term represents the sum of the bits saved over the discovered substructure, and the last two terms represent the sum of bits needed for the overhead.

When the substructure only matches part of a model graph (subgraph match), then representing the model includes an overhead associated with specifying the path to the model in the hierarchy ($I(path)$), and the mapping of all substructure's vertices and edges to part of the model's vertices and edges ($I(mapping_v)$ and $I(mapping_e)$). The mapping describes the number of vertices of the model, the number of edges of the model, and which vertices and edges of the model are matched to the substructure. However, when the substructure matches all parts of model graph (whole graph match), there is no need to indicate the mapping, because we assume the same order of vertex and edge labels in each graph.

Hence, the number of bits needed to represent M is

$$I(M) = I(path) + I(mapping_v) + I(mapping_e).$$

$I(path)$ is encoded as a path in the hierarchy of model knowledge, where a path is initiated at the matched node and terminated at the found model.

$$I(path) \quad = \quad Level \times \lg l_h,$$

where *Level* is the depth of model in the hierarchy and l_h is the number of unique models in the hierarchy.

$I(mapping_v)$ is encoded as the following:

$$I(mapping_v) \quad = \quad \lg nv_s + \lg \binom{nv_m}{nv_s},$$

where nv_s is the number of substructure vertices and nv_m is the number of model vertices. The first term describes how many vertices are mapped, and the second term describes which vertices are mapped.

Similarly, $I(mapping_e)$ is encoded as the following:

$$I(mapping_e) \quad = \quad \lg ne_s + \lg \binom{ne_m}{ne_s},$$

where ne_s is the number of substructure edges and ne_m is the number of model edges. The first term describes how many edges are mapped, and the second term describes which edges are mapped.

The description length of the compressed graph is

$$I(G|M) + I(M).$$

Therefore, if the portion of the substructure represented by the model is too small, the savings may not cover the overhead cost. If *Compression* is greater than zero, the representation of G using S which matches the model M is used instead of the original representation.

After a substructure is discovered, each instance of the substructure in the input graph is replaced by a pointer to a predefined model in the model hierarchy represents the substructure S. *DiscoverSubstructure*2 is repeated on the newly compressed input graph until no more substructures can be found. The newly compressed graph is input into *DiscoverSubstructure*1 to discover new substructures. *DiscoverSubstructure*1 is repeated until no more substructures can be discovered.

B. Graph match rules

At the heart of the SUBDUE system lies an inexact graph match algorithm that finds instances of a substructure definition. The graph match is used to identify isomorphic substructures in the input graph. Since many of those substructures could show up in a slightly different form throughout the data, and each of these differences is described in terms of basic transformations performed by the graph match, we can use graph match rules to assign each transformation a cost based on the domain of usage. This type of domain-specific information is represented using if-then rules such as the following:

IF (domain = x) and (perform graph match transformation y)
THEN (graph match cost = z)

To illustrate this rule, consider an example in the programming domain. We allow a vertex representing a variable to be substituted by another variable vertex, and do not allow a vertex representing an operator which is a special symbol, a reserved word, or a function call, to be substituted by another vertex. These rules can then be represented as the following:

IF (domain = programming) and (substitute variable vertex)
THEN graph match cost = 0.0;

IF (domain = programming) and (substitute operator vertex)
THEN graph match cost = 2.0

The graph match rules allow a specification of the amount of acceptable generality between a substructure definition and its instances, or between a model definition and its instances in the domain graph. Given $g1$, $g2$, and a set of distortion costs, the actual computation of $matchcost(g1, g2)$ can be performed using a tree search procedure. As long as $matchcost(g1, g2)$ does not exceed the threshold set by the user, the two graphs $g1$ and $g2$ are considered to be isomorphic.

VI. COMPUTATIONAL COMPLEXITY ANALYSIS

Since knowledge discovery algorithms should scale for use on large databases, the issue of computational complexity is very significant. The algorithms employed by SUBDUE are computationally expensive. For example, an

unconstrained graph match is exponential in the number of graph vertices. In practice, SUBDUE employs constraints that makes the program more scalable. Since the algorithm spends most of its time perform graph matches, the total running time of the algorithm can be expressed as the number of search vertices expanded during graph matches throughout the entire discovery process. In this section, the computational complexity of algorithms employed by SUBDUE is analyzed. We show how the algorithm can avoid exponential behavior, and we generate an upper bound on the complexity of SUBDUE as a function of the number of vertices in the input graph. Additionally, the algorithm without using domain knowledge and the algorithm using domain knowledge are compared.

In what follows, we will be using the following definitions:

- L = the user-defined limit on the number of substructures considered for expansion
- nv = the number of vertices in the input graph
- $nsub$ = the total number of substructures that can be generated
- gm = the user-defined maximum number of partial mappings that are considered during each graph match
- n_{inst} = the total number of instances of a given substructure
- m = the maximum number of model vertices in the model knowledge
- M = the average model branching factor in the model knowledge
- MC = the average number of models that are parents of other models in the model kowledge
- $N1$ = the total number of vertices expanded in Subdue without using domain knowledge
- $N2$ = the total number of vertices expanded in Subdue using model knowledge and graph match rules

A. Complexity without domain knowledge

This section provides an expression for the run-time requirement of the algorithm without using domain knowledge, showing that it depends on the number of vertices in the input graph and the limitations set by the user.

Since the algorithm spends most of its time perform graph match, the total running time of the algorithm can be expressed as

$$N1 \quad = \quad nsub \times n_{inst} \times gm.$$

Considering an upper bound time complexity, assume the input graph is a fully connected graph, where the number of neighbors for a given vertex is $(nv - 1)$, the maximum size of a substructure generated in iteration i of the algorithm is i vertices, and the number of vertices which have already been considered in previous iterations is $(i - 1)$. Hence, the total number of vertices that can be expanded is $((nv - 1) - (i - 1))$. Therefore, the total number of substructures that can be generated is

$$nsub \quad = \quad \sum_{i=1}^{L} (\ i \times ((nv - 1) - (i - 1))).$$

The total number of instances needed to be compared for a given substructure is affected by the instances of the substructure itself and the instances of the substructure's parent. For a substructure with i vertices, the maximum number of nonoverlapping instances is $\frac{nv}{i}$. Since we consider an upper bound case, the maximum number of nonoverlapping instances is nv. Hence, the total number of instances needed to be compared for a given substructure is

$$n_{inst} \quad = \quad nv \times (L - 1).$$

We have shown that by placing a limit on gm and L, the time complexity for the graph match is polynomial in nv. If either of the two limits L or gm is removed, the complexity of the discovery algorithm becomes exponential in nv. A parallel implementation of SUBDUE that is underway may further improve the scalability of the algorithm.

B. Complexity using domain knowledge

This section provides an expression for the run-time requirement of the algorithm using domain knowledge, showing that it depends on the number of vertices in the input graph, the limitations set by the user, and the model knowledge used. We will point out that for the upper bound case, the number of vertices expanded for discovery using domain knowledge can be less than the number of vertices expanded for discovery without using domain knowledge under certain circumstances.

Since the algorithm not only searches for the instances of a substructure, it also searches for a model in the model hierarchy which matches the substructure, the total running time of the algorithm can be expressed as

$$N2 \quad = \quad (nsub \times n_{inst} \times gm) + (nsub \times M \times MC \times gm),$$

where the first term represents the number of vertices expanded for the search of substructures' instances, and the second term represents the number of vertices expanded for the search of a model in the model hierarchy.

The maximum number of expanded vertices for a substructure is limited to the maximum number of vertices of a model in the model hierarchy (m). Hence, the number of iterations is limited to m. Therefore, $nsub$ can be expressed as

$$nsub \quad = \quad \sum_{i=1}^{m} \ i \times ((nv - 1) - (i - 1)).$$

The total number of instances needed to be compared for a given substructure is

$$n_{inst} \quad = \quad nv \times (m - 1).$$

We have shown that by placing a limit on gm, the time complexity for the graph match algorithm is polynomial in nv. If the gm limitation is removed, the complexity of the discovery algorithm becomes exponential in nv.

$(M \times MC)$ is dependent upon the size of the model knowledge. In general, L is set to half of the input graph,

gm is set to the forth power of the size of a substructure or model, whichever is bigger. Therefore, L is much larger than m. When the size of a substructure is big, which means that $(M \times MC)$ is small compared to gm, and $(M \times MC)$ is negligible, the number of vertices expanded for discovery using domain knowledge is less than number of vertices expanded for discovery without domain knowledge.

In conclusion, the number of vertices expanded for discovery using domain knowledge and without domain knowledge depends on the size of the input graph and model knowledge (m, M, MC), the size of the discovered substructures, and the limitations set by the user.

VII. Evaluation of SUBDUE's Domain-Independent versus Domain-Dependent Discovery

In this section, we evaluate the benefits and costs of utilizing the domain-specific information to perform substructure discovery. We will measure the performance of SUBDUE with and without domain-specific information when applied to databases in the programming, circuit, and artificial domains. The goals of our substructure discovery system are to efficiently find substructures that can reduce the description length needed to describe the data, and to discover substructures that are considered useful for the given domain.

To evaluate SUBDUE, we apply human ratings to each of SUBDUE's discovered substructures. If the approach demonstrates some validity, SUBDUE should prefer substructures which were rated highly by humans. Two types of discovered substructures are evaluated: 1) substructures discovered without using the domain knowledge, and 2) substructures discovered using domain knowledge. The performance of the system is measured along three dimensions: 1) compression, which shows a substructure's ability to compress an input graph, 2) number of search vertices expanded by SUBDUE, which indicates the time to discover a substructure, and 3) average evaluation value and standard deviation of human rating, which measure the interestingness of a substructure according to human experts. The interestingness of SUBDUE's discovered substructures are rated by a group of 8 domain experts on a scale of 1 to 5, where 1 means not useful in the domain and 5 means very useful. The number of instances of the discovered substructure that exist in the input database is also listed.

The discovered substructures are plotted, and grouped into figures. Substructures inside the boxes indicate substructures discovered in earlier iterations. Therefore, if the newly discovered substructures are defined in terms of previously discovered substructure concepts, the substructure definitions form a hierarchy of substructure concepts. Numbers inside the circles indicate the iteration in which the substructures are discovered.

A. Evaluation of substructures in programming domain

The discovery of familiar structures in a program can help a programmer to understand the function and modularity of the code. The recognition of substructures from the domain knowledge helps in understanding the codes, and the discovery of repetitive and functional substructures helps in modularizing the codes. Hence, SUBDUE helps describe a program which in turn helps facilitate many tasks that require program understanding, e.g., maintenance and translation.

```
sorted = 0;        /* bubble sort */
while (sorted == 0)
    sorted = 1;
    for (j = 0; j < listsize - 1; j++)
        if (list[j] > list[j + 1])
            temp = list[j];
            list[j] = list[j + 1];
            list[j + 1] = temp;
            sorted = 0;
for (gap = n/2; gap > 0; gap = gap/2)        /* shell sort */
    for (i = gap; i < n; i++)
        for (j = i - gap; j >= 0 && v[j] > v[j + gap]; j = j - gap)
            temp = v[j];
            v[j] = v[j + gap];
            v[j + gap] = temp;
/* bubble sort operates as a type of selection sort */
for (i = n; i > 0; i--)
    for (j = 2; j >= i; j++)
        if (a[j - 1] > a[j])
            t = a[j - 1];
            a[j - 1] = a[j];
            a[j] = t;
```

Fig. 7. Part of a sample program concatenating three different sort procedures.

In this domain, the model graphs are built based on commonalities between subroutines' functional structure. For example, special symbols and reserved words are represented by primitive nodes, and functional subroutines (e.g., swap, sort, increment) are represented by nonprimitive nodes. Furthermore, the graph match rule is used to allow two variables to be matched as long as their binding is consistent.

In order to determine the value of substructures discovered by SUBDUE, we concatenate three different sort routines (written in C) into one program (see Figure 7), and transform it into a graph representation which is independent of the source language.

The description length of the sample program shown in Figure 7 is 2598.99 (in bits). Figure 8 shows discovered substructures without domain knowledge from the sample program. Figure 9 demonstrates discovered substructures using domain kowledge.

The substructures discovered without domain knowledge yield low human ratings. The overall compression achieved is 0.11, and the total number of search vertices considered is 118,950. On the other hand, the discovered substructure using domain knowledge receive very high human rating, because the substructure represent a conditional swap function, which is useful to programming experts. The overall compression achieved is 0.2 and the total number of search vertices considered is 21,648. The results demonstrate that the discovery using domain knowledge achieve better human rating and compression than the discovery without

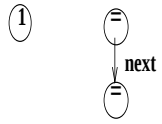
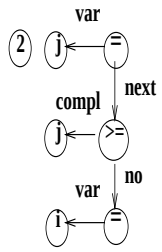
Discovered Substructures Without Domain Knowledge	Compression	Number of Vertices Expanded	Average Human Rating [std dev]	Number of Instances
	0.07	68,386	1.33[1.2]	9
	0.04	50,564	2.0[1.4]	2

Fig. 8. Program–Discovered substructures without domain knowledge.

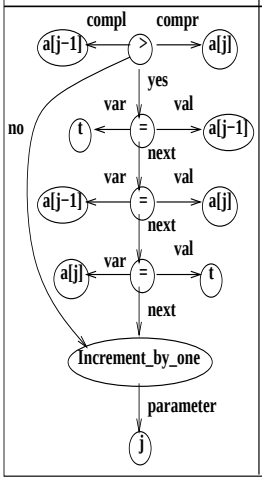
Discovered Substructures Without Domain Knowledge	Compression	Number of Vertices Expanded	Average Human Rating [std dev]	Number of Instances
	0.07	17,974	4.8[0.41]	2

Fig. 9. Program–Discovered substructures using domain knowledge.

domain knowledge. Futhermore, the number of search vertices considered for discovery using domain knowledge is significantly less than discovery without domain knowledge.

B. Evaluation of substructures in CAD circuit domain

As a result of increased complexity of design and changes in the implementation technologies of integrated electronic circuitry, the discovery of familiar structures in circuitry can help a designer to understand the design, and to identify common reusable parts in circuitry.

We evaluate SUBDUE by using CAD circuit data representing a sixth-order bandpass "leapfrog" ladder [23]. The circuit is made up of a chain of somewhat similar structures

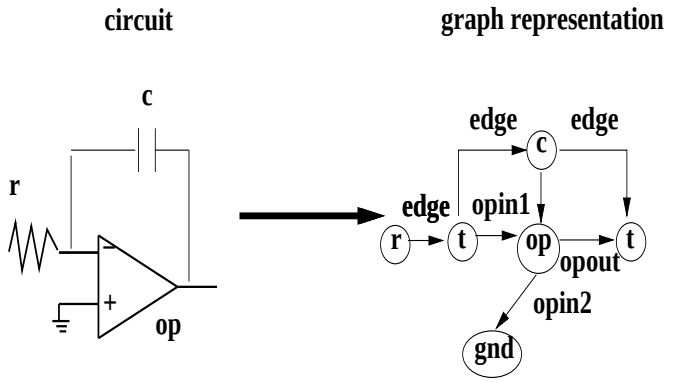


Fig. 10. The transformation from a sample circuit into a graph representation.

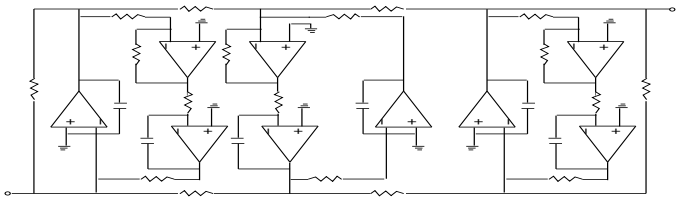


Fig. 11. Bandpass "leapfrog": sixth-order.

(see Figure 11). We transform the circuit into a graph representation in which the component units and interconnection between several component units appear as vertices and the current flows appear as edges (see Figure 10).

In this domain, the hierarchical graphs are built based on commonalities between circuits' functional structure. For example, basic components of a circuit (e.g., resistor, transistor) are represented by primitive nodes, and functional subcircuits such as operational amplifier, filter, etc. are represented by nonprimitive nodes. Furthermore, a graph match rule is used to allow two similar components with different labels to be matched.

The description length of the circuit shown in Figure 11 is 3139.05 (in bits). Figure 12 shows discovered substructures of the circuit without domain knowledge and Figure 13 shows discovered substructures of circuit using domain knowledge.

When the domain knowledge is used, all of the discovered substructures receive very high human ratings, because the substructures represent functional circuits. The overall compression achieved is 0.79, and the total number of vertices expanded is 161,515. When the domain knowledge is not used, the discovered substructures receive lower human ratings. The first substructure obtains a high human rating, because the substructure represents an inverter and appears many times in the input graph. The overall compression achieved is 0.72, and the total number of search vertices considered is 677,678. The results again reveal that discovery using domain knowledge offers better human ratings and greater compression than discovery without domain knowledge. Additionally, the number of search vertices considered using domain knowledge is

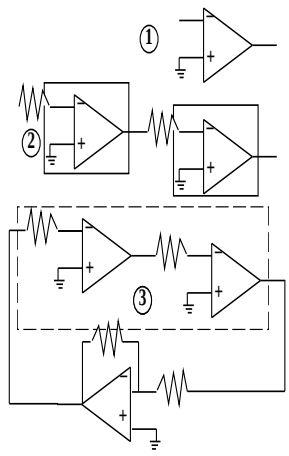
Discovered Substructures without domain knowledge	Compression	Nodes Expanded	Human Rating [std dev]	Instances
	0.63	571,370	4.2 [1.2]	9
	0.68	46,422	2.7 [1.2]	3
	0.72	59.886	2.7 [1.0]	2

Fig. 12. CAD circuit–Discovered substructures without domain knowledge.

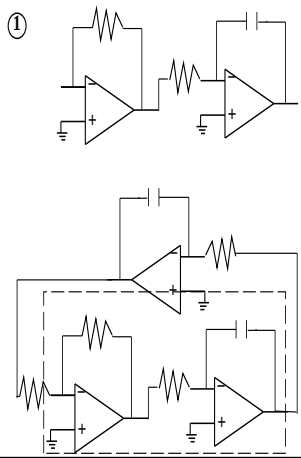
Discovered Substructures using domain knowledge	Compression	Nodes Expanded	Human Rating [std dev]	Instances
	0.18	8,107	3.7[1.0]	3
	0.33	119,047	4.5[0.8]	2

Fig. 13. CAD circuit–Discovered substructures using domain knowledge.

smaller than without domain knowledge.

C. Evaluation of substructures in the artificial domain

While we have evaluated the result of discovery using domain knowledge in two domains, we also examine whether such domain knowledge is useful in general. We would like to evaluate whether the use of domain knowledge can improve SUBDUE's average case performance in an artificially-controlled graph.

To test this performance, we create two tests. Firstly, an artificial substructure is created and is embedded in larger graphs of varying sizes. The graphs vary in terms of graph size and the amount of deviation in the substructure's instances, but are constant with respect to the percentage of the graph that is covered by the substructure's instances. For each deviation value, we run SUBDUE on the graphs until no more compression can be achieved with cases: a) without domain knowledge, b) with domain knowledge. The effects of the varying deviation values are measured against the average compression value (Figure 14), the average number of vertices expanded (Figure 15) , and the average number of embedded instances discovered (Figure 16). As the amount of deviation increases, the compression in all cases decreases as expected, except at the deviation of 1.5. A slight increase in compression at 1.5 is due to the randomness of the data. Case (b) yields better compression and expand lesser vertices than case (a). The number of vertices expanded by case (b) remains about the same for all deviations, because the same instances (of the same size) are discovered consistently. Furthermore, as the deviation is increased, case (b) is capable of finding embedded instances, and case (a) is not capable of finding embedded instances for a slight increase in deviation.

Secondly, we again embed an artificial substructure into

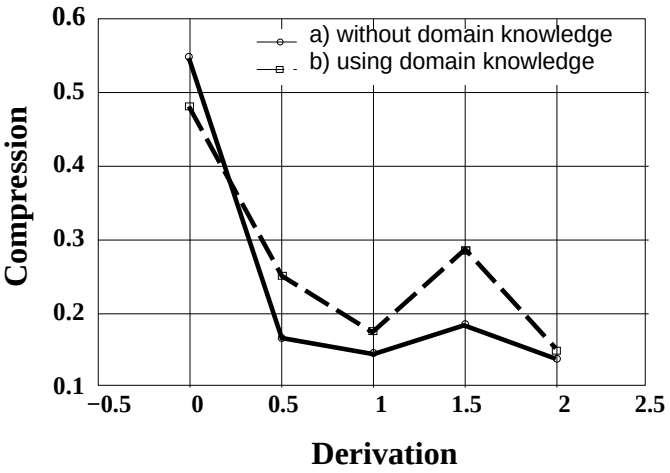


Fig. 14. Deviation versus compression.

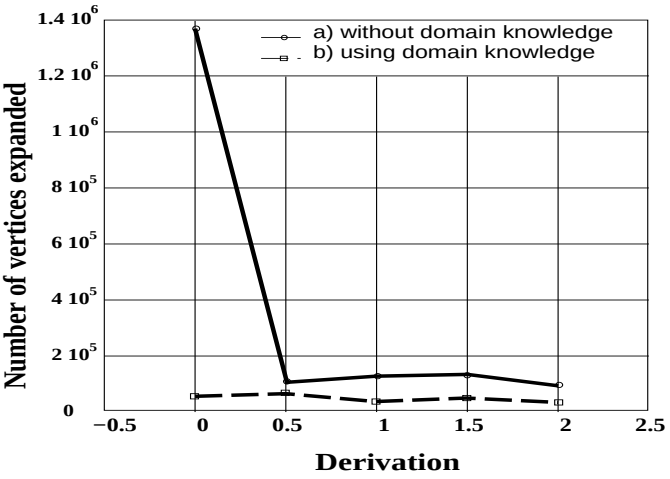


Fig. 15. Deviation versus number of vertices expanded.

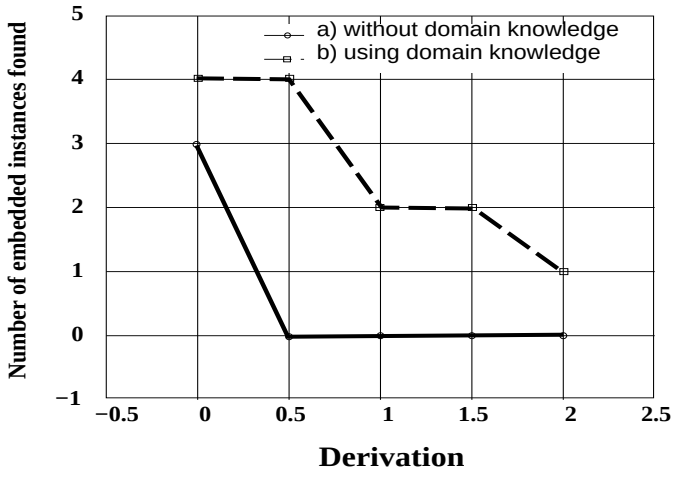


Fig. 16. Deviation versus number of instances found.

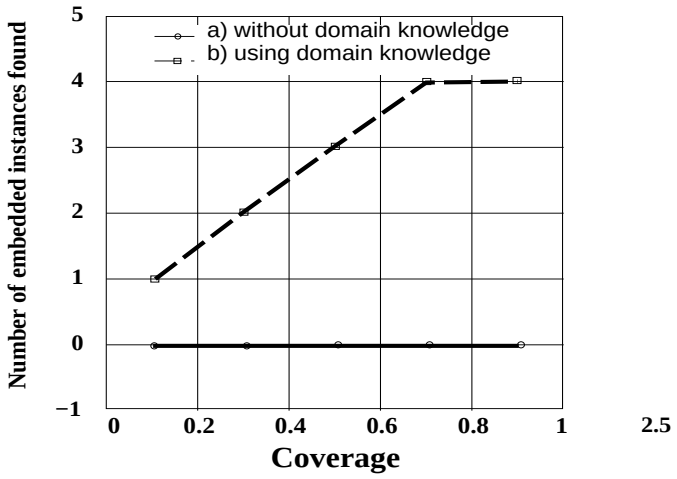


Fig. 17. Coverage versus number of instances found.

a larger graphs of varying sizes. Each of the graphs varies in the size, as well as the amount of the input graph covered by the embedded substructure. For each coverage value, we evaluate the same two cases. The effect of the varying coverage values are measured against the average number of embedded instances discovered (Figure 17). As the coverage is increased, case (b) finds an increasing number of embedded instances. Case (a) does not find any instance.

The effects confirm the results demonstrated by the application domains. Therefore, we conclude that SUBDUE using domain knowledge is capable of discovering useful substructures, achieving better compression, and focusing the search for concepts.

VIII. CONCLUSIONS

SUBDUE is a system devised for experimenting with general-purpose automated discovery using domain knowledge, allows the domain knowledge to be generic, and can be reused over a class of similar applications. Hence, the method can be applied to many structural domains.

This paper describes the process by which a scientist reduces the complexity of a problem by applying what is known and abstracting detail in the form of regular structure. For the domains of CAD circuit, programming, SUBDUE has shown success in compressing data and discovering useful substructures. SUBDUE can aid the scientist in reducing the complexity of the data and may uncover new concepts of importance to the domain. Results indicate that discovery using domain-specific knowledge has better chance of discovering substructures which are useful to domain experts, leads to greater compression of the data, has better performance than the results of discovery without using domain knowledge.

A parallel implementation of SUBDUE is underway that may further improve the scalability of the algorithm. Parallelization on a MIMD machine by distributing the search space will allow SUBDUE to scale up to much larger databases without significant increase in processing time.

REFERENCES

[1] P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor, and D. Freeman, "Autoclass: A bayesian classification system", in *Proceedings of the Fifth International Workshop on Machine Learning*, 1988, pp. 54–64.

[2] D. H. Fisher, "Knowledge acquisition via incremental conceptual clustering", *Machine Learning*, vol. 2, pp. 139–172, 1987.

[3] W. J. Frawley, G. Piatetsky-Shapiro, and editors C. J. Matheus, *Knowledge Discovery in Databases*, AAAI Press / The MIT Press, 1991.

[4] J. R. Quinlan, "Induction of decision trees", *Machine Learning*, vol. 1, pp. 81–106, 1986.

[5] J. R. Quinlan and R. L. Rivest, "Inferring decision trees using the minimum description length principle", *Information and Computation*, vol. 80, pp. 227–248, 1989.

[6] D. J. Cook, L. B. Holder, and S. Djoko, "Knowledge discovery from structural data", *Journal of Intelligent Information Systems*, vol. 5, no. 3, pp. 229–245, 1995.

[7] D. Conklin, S. Fortier, J. Glasgow, and F. Allen, "Discovery of spatial concepts in crystallographic databases", in *Proceedings of the Ninth International Machine Learning Workshop*, 1992, pp. 111–116.

[8] R. Levinson, "A self-organizing retrieval system for graphs", in *Proceedings of the Second National Conference on Artificial Intelligence*, 1984, pp. 203–206.

[9] J. Segen, "Learning graph models of shape", in *Proceedings of the fifth International Conference on Machine Learning*, 1988, pp. 29–35.

[10] K. Thompson and P. Langley, "Concept formation in structured domains", in *Concept Formation: Knowledge and Experience in Unsupervised Learning*, D. H. Fisher and M. Pazzani, Eds. Morgan Kaufmann Publishers, Inc., 1991.

[11] P. H. Winston, "Learning structural descriptions from examples", in *The Psychology of Computer Vision*, P. H. Winston, Ed., pp. 157–210. McGraw-Hill, 1975.

[12] A. K. C. Wong and M. You, "Entropy and distance of random graphs with application to structural pattern recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, no. 5, pp. 599–609, 1985.

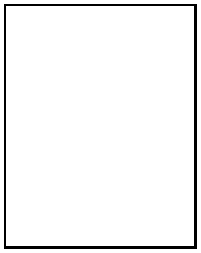
[13] H. Bunke and G. Allermann, "Inexact graph matching for structural pattern recognition", *Pattern Recognition Letters*, vol. 1, no. 4, pp. 245–253, 1983.

[14] A. Sanfeliu and K. S. Fu, "A distance measure between attributed relational graphs for pattern recognition", *IEEE Transactions on Systems, Man and Cybernetic*, vol. 13, pp. 353–362, 1983.

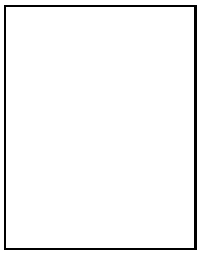
[15] K. Yoshida, H. Motoda, and N. Indurkhya, "Unifying learning methods by colored digraphs", in *Proceedings of the Learning and Knowledge Acquisition Workshop at IJCAI-93*, 1993.

[16] D. J. Cook and L. B. Holder, "Substructure discovery using minimum description length and background knowledge", *Journal of Artificial Intelligence Research*, vol. 1, pp. 231–255, 1994.

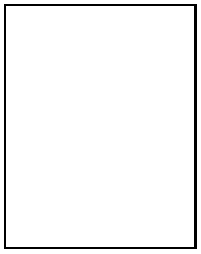
[17] J. Rissanen, *Stochastic Complexity in Statistical Inquiry*, World Scientific Publishing Company, 1989.
[18] Y. G. Leclerc, "Constructing simple stable descriptions for image partitioning", *International Journal of Computer Vision*, vol. 3, no. 1, pp. 73–102, 1989.
[19] E. P. D. Pednault, "Some experiments in applying inductive inference principles to surface reconstruction", in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1989, pp. 1603–1609.
[20] A. Pentland, "Part segmentation for object recognition", *Neural Computation*, vol. 1, pp. 82–91, 1989.
[21] M. Derthick, "A minimal encoding approach to feature discovery", in *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991, pp. 565–571.
[22] R. B. Rao and S. C. Lu, "Learning engineering models with the minimum description length principle", in *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992, pp. 717–722.
[23] L. T. Bruton, *RC-Active Circuits Theory and Design*, Prentice Hall, 1980.



Surnjani Djoko received the B.S.E.E. degree from Tamkang University, Taiwan, R.O.C. in 1986, the M.S.E.E. degree and the Ph.D. degree in Computer Science and Engineering from the University of Texas at Arlington, TX, in 1989 and 1995, respectively. Her research interests have been in the areas of knowledge discovery in databases, machine learning, statistical methods for inducing models from data, and parallel algorithms. She is currently a Member of Scientific Staff at Bell Northern Research, Richardson, TX.



Diane Cook is an Assistant Professor in the Computer Science and Engineering Department at the University of Texas at Arlington. Dr. Cook received her B.S. from Wheaton College in 1985, and her M.S. and Ph.D. from the University of Illinois in 1987 and 1990, respectively. Dr. Cook's research interests include artificial intelligence, machine planning, machine learning, robotics, and parallel algorithms for artificial intelligence.



Lawrence Holder is currently an Assistant Professor in the Department of Computer Science and Engineering at the University of Texas at Arlington. He received his M.S. and Ph.D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 1988 and 1991. He received his B.S. degree in Computer Engineering also from the University of Illinois at Urbana-Champaign in 1986. Dr. Holder's research interests include artificial intelligence and machine learning.