# SUPERVISED LEARNING FROM EMBEDDED SUBGRAPHS

by

JOSEPH TAYLOR POTTS

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2006

Dedicated to

Nancy Kay Holland,

without whom this could not have been done,

even though without her it was finished.

ACKNOWLEDGEMENTS

I would like to express both my deep appreciation of, and my high admiration for, my supervising professor Dr. Diane Cook. She has never failed to answer a question, has never stopped explaining until I said I understood, and has never given up on me, though my path to this degree was long and my progress, at times, painfully slow. I would also like to thank Dr. Larry Holder for his patience and tolerance as I worked with his Subdue code. To Dr. Lynn Peterson, with me in this pursuit from the very beginning, always smiling and always supportive, I say thank you for being there every time. Additionally, the two newest members of my committee, Sharma Chakravarthy and Alp Aslandogan, receive my thanks for their thoughts and helpful comments.

Finally, to all the members of my family, most of whom have, at one time or another, driven me to Arlington, read to me from what surely must have been difficult and boring material, and supported me in this endeavor, I say thank you so much. I truly could not have done it unless you had helped me.

This work was supported in part by NASA Headquarters under the Earth Systems Fellowship grant NGT5.

April 12, 2006

# ABSTRACT

## SUPERVISED LEARNING FROM EMBEDDED SUBGRAPHS

Publication No. _____

Joseph Taylor Potts, PhD.

The University of Texas at Arlington, 2006

Supervising Professor: Diane J. Cook

We develop a machine learning algorithm which learns rules for classification from training examples in a graph representation. However, unlike most other such algorithms which use one graph for each example, ours allows all of the training examples to be in a single, connected graph. We employ the Minimum Description Length principle to produce a novel performance metric for judging the value of a learned classification. We implement the algorithm by extending the Subdue graph-based learning system. Finally, we demonstrate the use of the new system in two different domains, earth science and homeland security.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

Learning systems capable of utilizing graph-based input have typically required representing the training examples as disjoint graphs. Input for these systems consists of training examples represented as individual graphs, each of which is an example of one of *n* classes. In the case of a concept learner, the goal is to learn a concept which allows the user to determine if a previously-unseen graph is an example of the concept. The goal for classification is to learn a set of rules which enable the user to place a novel graph into one of *n* classes.

In a domain where training examples are naturally contained in a single graph, it can be quite difficult to efficiently transform the data for input into systems such as those above. For example, imagine the graph traditionally called a "family tree" with five generations depicted. Suppose we have annotated the vertices of this graph with some attribute of the people represented, such as whether or not that person has suffered some kind of abuse. We would like to predict which members of the next generation of this family might also suffer some kind of abuse. We might speculate that children of abused people tend to be abused and / or to become abusers themselves. In order to look for

possible evidence of such a hypothesis in this family tree we would like to be able to classify novel vertices (new children) as "potential abuse victims" or not.

If a system requires individual graphs for each example, then it is necessary to excise each example along with some amount of surrounding graph structure to create a disconnected graph containing that example. In the case of the family tree, we might take a child, his siblings, his parents and his grandparents. If the examples are close enough to each other in the original graph, then this surrounding data may overlap with the surrounding data of another example as in the case of siblings. We would have to duplicate the parents and grandparents of the siblings in each of the individual sibling graphs created from the tree to serve as training examples. In fact, one might even have to include all or part of another example. This overlap will result in some data appearing in more than one example graph. It may be difficult to determine just exactly how much structure to include in the example. Should we include aunts and uncles? Should we include cousins? Taking too large a region around the example causes extra data to be handled. Taking too small a region may result in the loss of potentially vital information. Since processing graph-based data is very resource intensive, any redundant information can have a drastic effect on performance. Failure to include enough data may result in the inability of the system to learn.

We hypothesize that a compression-based graph mining algorithm can be used to learn class information embedded in a single, connected graph. We develop a learner that allows the input graph, containing all the training examples for all classes, to be input with a minimum of preprocessing and a minimum of added or redundant information.

In a highly complex relational domain, positive and negative examples of a concept are not easily separated into non-overlapping graphs. We call such a graph with embedded, possibly overlapping, examples a *supervised graph*, or a graph that contains embedded class information which may not be easily separated into individual labeled components.

For example, consider a social network in which we look for patterns distinguishing various income levels. Individuals of a particular income level can appear anywhere in the graph and may interact with or be related to individuals at other income levels, so we cannot easily partition the graph into separate training cases without potentially severing the target relationships. The high income CEO who raises orchids knows many of the same people at the nursery as his gardener does. The gardener buys his overalls at the same store as the CEO's daughter. And all of them go to the same church. But the person-who-is-the-daughter-of–the-CEO is in a different income class than the person-who-is-the-gardener-of-the-CEO.

To validate our hypothesis, we propose a representation requiring the addition to the input graph of only one vertex for each example. We also propose an algorithm which will construct substructures capable of identifying the examples of each class guided by a new performance metric called *classification compression*. Finally, we propose a representation for these learned substructures called a *classification sequence* which facilitates the determination of class membership for new observations.

We implement this algorithm by starting with some of the core routines of the Subdue graph-based relational learning system (Cook and Holder, 2000). We then

modify the search strategy, add new evaluation criteria, and enhance the algorithm with the capability to apply the learned concepts to novel graphs. The result is that the only preprocessing required on the graph is to add a vertex for each example identifying its class, and adding edges to connect those vertices to each vertex of the examples. The program then reads the augmented graph and produces a classification sequence or series of substructures which can be used to determine the class of marked examples in a novel graph.

In order to assess the effectiveness of the algorithm, we will compare results of learning using a supervised graph with results obtained from learning using a system that requires one graph per example. We will also compare the new learner with a linear feature-vector approach. These experiments will be conducted on both synthetic data and real-world data from two different domains.

The remainder of this dissertation is organized as follows. We describe related work and how our approach to the problem is different. We lay some foundation for the understanding of our algorithm including some background on the Subdue discovery system and one of its later extensions for concept learning. We will then describe the algorithm in detail and develop the ideas of classification compression and classification sequences. Finally, we will describe the data used to test the algorithm and provide results from experiments on this data.

# CHAPTER 2

# RELATED WORK

The most similar work to ours that addresses the stated problem is the Subdue graph-based learning project (Cook and Holder 2000). Subdue was developed as a discovery system to find interesting patterns in structural data. The initial set of heuristics (connectedness, completeness, etc.), drew from work in cognitive psychology but it was found to be an imperfect fit for the more general algorithm. Therefore, the developers turned to information theory which offered the Minimum Description Length theory (Rissanen 1989). The MDL theory supports the idea that the resulting discovery will most effectively describe the data,

This system was extended to perform concept learning, implemented as Subdue-CL (Gonzalez, Holder, Cook, 2001). This is the first manifestation of Subdue which is capable of supervised learning. Given a set of "positive" examples and "negative" examples each of which is represented by its own disjoint graph, Subdue-CL finds a subgraph which does a good job of compressing the positive examples and does a poor job of compressing the negative examples. This subgraph can be thought of as being the essence of the concept "positive". Subdue-CL also offers two other performance measures which will be discussed later.

Subdue-GL (Jonyer et al.), another Subdue variant, induces graph grammars. Graph grammars provide a more expressive concept capable of representing recursion and variables. However, this algorithm still requires disjoint graphs for each example used in learning. None of the prior versions of Subdue was capable of performing supervised learning from a single graph containing all the examples of each class.

Recently, other graph-based discovery systems have been introduced. Most are based on the Apriori algorithm (Agrawal and Srikant 1994) which finds frequently-occurring subgraphs. There are four algorithms currently able to find all frequently-occurring subgraphs with reasonable computational efficiency: 1) the AGM algorithm (Inokuchi et al. 2000), the FSG algorithm (Kuramochi and Karypis 2001), the chemical substructure discovery algorithm (Berthold and Borgelt 2002) and the gSpan algorithm (Yan and Han 2002). One of these, FSG, is used as part of a chemical classification system (Deshpande, Kuramochi, and Karypis 2005). This system keeps the substructure discovery separate from the classification process. It first finds frequently-occurring substructures, then uses them to build classification rules. This ensures that no important substructures will be unavailable for the generation of the rules. This system accepts disjoint graphs representing individual molecules as input.

Finally, we must mention inductive logic programming or ILP (Muggleton and De Raedt 1992) approaches to concept learning. ILP is not a graph-based system, but graphs can easily be transformed to the first order logic representation used by ILP. In fact, first order logic is a more powerful representation than a labeled graph. ILP produces rules made of Horn clauses that can be used for classification. Two current

systems that also use this concept are Progol (Srinivasan et al. 1997) and FOIL (Slattery and Craven 1998).

In a comparative study of graph-based and logic-based relational learning, Ketkar, et al. (2005) found that for large relational concepts a graph-based approach can outperform a logic-based approach. However, logic-based approaches are better suited for semantically complex concepts. In addition, logic-based learners are better able to incorporate background knowledge than graph-based ones. This richer representation typically results in increased computational complexity.

The distinguishing feature of our problem is the fact that all examples can be in a single graph. This is a crucial point for two reasons. First, it allows sharing of data between examples in a very efficient manner. Of the systems above, only an ILP-based one allows multiple examples to have a relation to a common piece of data which is presented only once. Of course, a large, single graph could be transformed into a set of disjoint example graphs, provided that one is able to select the proper size neighborhood around each example. Therein lies the second advantage of the single input graph: there is no need to excise each example. Virtually all information in the graph is available for the classification of each example.

# CHAPTER 3

# THEORETICAL FOUNDATION

In this chapter we discuss how data can be represented n the form of a graph, a concept familiar to most of us as the structure used to depict an organizational chart. We then discuss how machine learning can be accomplished using such data in Subdue.

## Representing Data as a Graph

The graph representation is a very general one. It can be used to represent social networks, organizations, family trees, communication networks, images, web pages and almost anything else where the relations between the data elements need to be part of the representation. A graph $g = (V, E, \alpha,\beta)$ is defined by its set of vertices, $V$, a set of directed or undirected edges, $E$, which is a subset of $VxV$, a function $\alpha: V \rightarrow L_V$ mapping vertices to labels, and function $\beta: E \rightarrow L_E$ mapping edges to labels. Edge $(x, y) \in E$ originates at node $x \in V$ and terminates at node $y \in V$. An undirected graph is allowed as a special case if there exists an edge $(y, x) \in E$ for every edge $(x, y) \in E$ with $\beta(x, y) = \beta(y, x)$.

In illustrations the vertices are commonly shown as circles or ovals. The edges that connect them are typically shown as arrows for directed edges and lines for undirected edges. Vertex labels are usually shown as words or numbers inside the

circles or ovals representing the vertices. Edge labels are usually on or along the lines or arrows representing the edges. The vertices usually represent the data elements while the edges typically represent the relations between data elements.

Let $g = (V, E, \alpha, \beta)$ and $g' = (V', E', \alpha', \beta')$ be graphs. We say $g'$ is a subgraph of $g$, or $g'$ is contained in $g$, if $V' \subseteq V$, $E' \subseteq E$ ,$\alpha(x) = \alpha'(x)$ for all $x \subseteq V'$, and $\beta(x, y) = \beta'(x, y)$ for all $(x, y) \subseteq E'$.

Figure 3.1 shows a graph G with one of its subgraphs, S. All of the edges in this graph have the same label, "e", but for clarity these edge labels are not depicted in the figure.

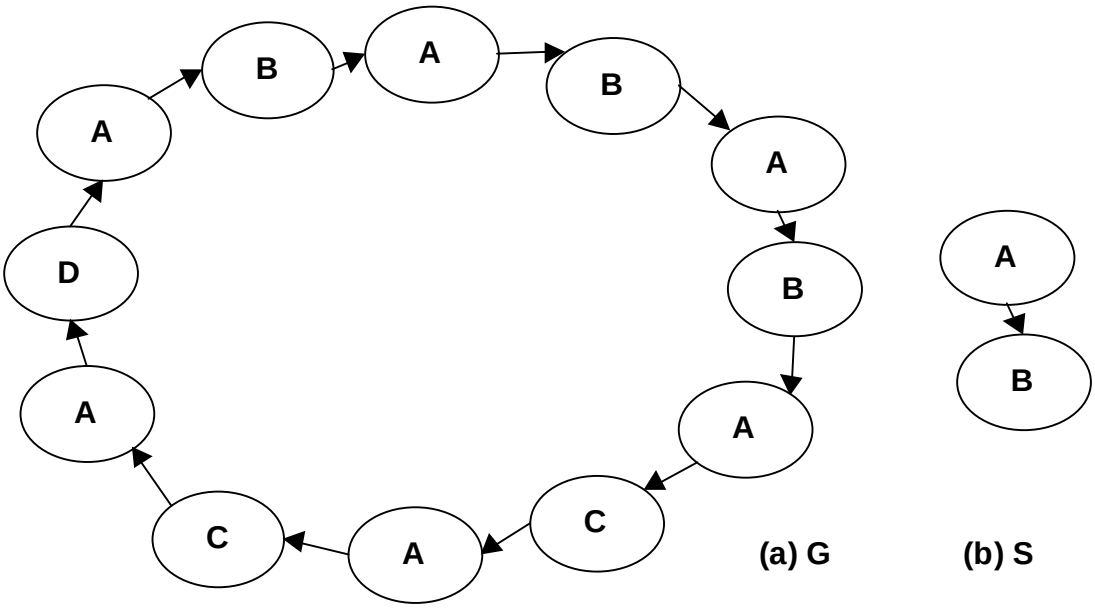



Figure 3.1 (a) Graph G with (b) subgraph S

## Subdue and the Minimum Description Length

Subdue is a graph-based learning system designed to find "interesting" substructures in relational data. It does this by finding "interesting" subgraphs in a graph representing the structural data. But what is "interesting"? Early versions of Subdue attempted to find subgraphs with properties that had been shown to be "interesting" to humans. These properties included things like compactness, coverage and connectivity (Holder et al, 1992). Our human visual processing system seems to be drawn to areas that are closely or tightly connected.

This concept of "interesting" was later supplanted by a concept relying on the Minimum Description Length (MDL) principle (Rissanen 1989). This principle had been applied to the induction of decision trees (Quinlan, 1989) and proved useful in Subdue also.

To understand how this principle can be used to find an "interesting" subgraph, we must first imagine that we are asked to send our graph to an agent by some electronic means. We must encode our graph into a bit stream. Ideally, we would use the optimal encoding so that we could send the absolute minimum number of bits and so that certain theoretical properties would hold. The encoding that Subdue uses is not optimal, but is smaller than the traditional adjacency matrix or list encoding and handles sparse matrices well (Holder, 1989). We can encode our graph *G* and count the bits in the resulting bit stream, calling the number the *description length* of *G* or *DL(G)*.

Next, suppose that some subgraph, *S*, occurs frequently in *G*. We might get a smaller graph encoding if we substituted a new vertex labeled "S1" for each instance of the

subgraph *S* in *G*. If we now encode this new graph and also encode the subgraph *S* we could send both the subgraph *S* and the new graph to our receiver. The receiver could restore the original graph from this information by replacing all the "S1" vertices in the graph we sent by the subgraph *S* which we also sent. If the description length of the new graph obtained from *G* using *S* and designated *DL(G|S)*, plus the description length of *S*, *DL(S)*, were less than the description length of *G*, *DL(G)*, we would have a shorter message to send. We would have *compressed* the graph and the length of the message required to send it. The MDL principle tells us that the "best" subgraph is the one that results in the *minimum description length* when it is used to compress the graph. We can calculate this value using the following formula:

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

We will call the reciprocal of the compression, *value.* Subdue uses value as a heuristic to guide it in its search for the "best" subgraph. If we use subgraph *S* in figure 3.1b to compress the graph *G* in figure 3.1a, we obtain the graph *G|S* shown in figure

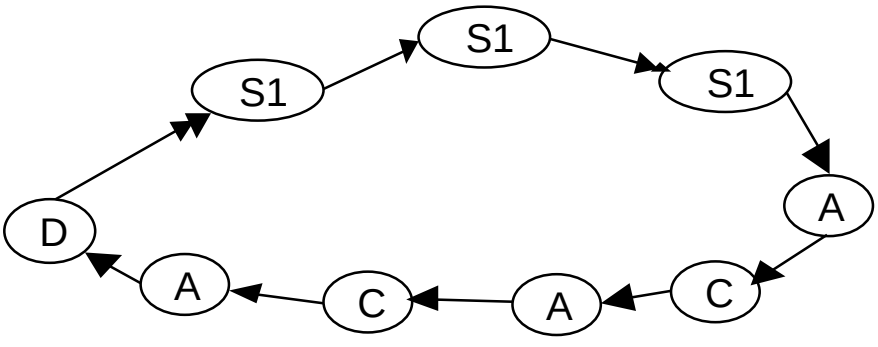



Figure 3.2 Graph G with vertex "S1" substituted for the subgraph in figure 3.1b

3.2. Utilizing the graph encoding method developed by Djoko (Djoko et al, 1992) and implemented in Subdue, the description length of *G* is 127.331 bits. The description length of *S* is 12.966 bits and the description length of G|S is *103.229* bits. Therefore the value of *S* in compressing *G* is 1.09584.

## The Subdue Discovery Process

A substructure in Subdue consists of a subgraph definition and all of its occurrences throughout the graph. Subdue searches for candidate substructures using a variant of beam search. The initial state of the search is the set of substructures consisting of single vertices. The only operator of the search is the *ExtendSubstructure* operator. As its name suggests, this operator extends a substructure in all possible ways by a single edge. If the edge is connected to a vertex that is not currently in the subgraph, then that vertex is also added to the subgraph definition. The search progresses by applying the *ExtendSubstructure* operator to each substructure in the current queue to create a new queue. The new search queue, however, does not contain all the substructures generated by the *ExtendSubstructure* operator. The substructures are kept on a queue of limited length and are ordered based on their value. Repeated execution of the *ExtendSubstructure* operator propels Subdue through the search space.

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 Subdue returns the list of best substructures found, Subdue compresses the graph using the best substructure. Subdue can perform as many iterations of this

search-and-compress cycle as it takes to completely compress the graph or as many as the user specifies.

## Learning from Disjoint Examples

As mentioned in Chapter 1, Subdue has been extended to allow it to learn a concept from example graphs. Suppose one had a set of examples of a concept, each of which could be represented by a graph (the positive graphs, for a binary concept). And suppose one also had a number of other graphs, perhaps similar, but NOT examples of the concept (the negative graphs). Now, if one supplied all of these graphs to Subdue as training examples, a question arises of whether the MDL principle can be used to "learn" the concept by selecting a subgraph that compresses the positive graphs and does not compress the negative ones? It turns out that it can.

An advantage of the MDL-based concept learning strategy is that the resulting concept is not only valuable for discriminating between classes, but the concept itself is also interesting from an information-theoretic perspective. On the other hand, sometimes the compression-guided strategy leads to unsatisfactory classification ability because the subgraph might occur many times in a few positive examples. This may produce great compression of the positive graphs as a whole, but result in poor classification performance, since most of the compression occurs in a few positive graphs. As a result, a second heuristic was developed called "set cover". An example is "covered" by a subgraph if that example's graph contains one or more instances of the subgraph. The set cover value of a candidate substructure is calculated as the number of positive examples covered by the subgraph plus the negative examples *not* covered.

This quantity is expressed as a proportion of total examples. This was a more effective measure in some domains than the MDL-based value.

$$Set\ Cover\ Value = \frac{|Positive\ Examples\ Covered + Negative\ Examples\ Not\ Covered|}{|Total\ Examples|}$$

In figure 3.3a, there are four configurations of geometric shapes which are

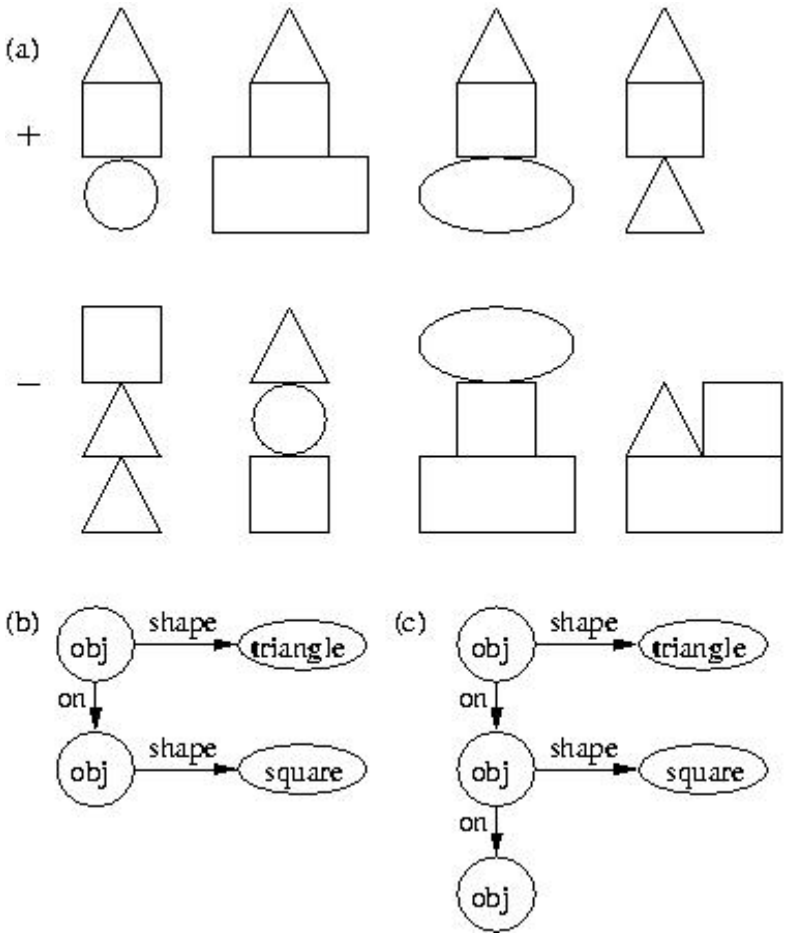



Figure 3.3 Given eight graphs (a), Subdue discovers two positive concepts, (b) and (c)

“positive” examples and there are four which are “negative” examples. The concept learned by Subdue from these examples is shown in figure 3.3b. This subgraph, representing the concept, is contained in all of the positive examples and none of the negative ones. Figure 3.3c is a second concept which is larger but equally effective in distinguishing the positive examples from the negative ones. To use the discovered subgraph to determine whether a new configuration of shapes is “positive” or “negative”, the configuration is first converted to a graph. Then, if the graph representing the concept is a subgraph of the new configuration’s graph, the figure is “positive”. If the configuration’s graph does not contain the subgraph, it is a “negative” configuration. The only test for “negative” is the failure of the “positive” test.

# CHAPTER 4

# LEARNING FROM EMBEDDED EXAMPLES

The problem that we are addressing is different from the problem described in the previous chapter in two fundamental ways. First, we want to allow learning from examples contained in a single, connected graph. Second, we want to allow learning of concepts that can be used to classify examples into *n* classes, not just the learning of a binary concept. Both of these important elements are included in the problem statement below. After defining the problem, we introduce two new terms, classification sequence and classification compression. Finally, we describe an algorithm for learning using these concepts.

## The Problem Statement

Given a labeled graph, $G = (V, E, \alpha, \beta)$; a set of examples, $X$, where each $x \in X$ is a set of one or more vertices, all of which are elements of $V$; and a set $C$ of class designations, one for each example from the set of $n$ classes; learn a set of subgraphs, $S$, which can be used to assign a class to designated sets of vertices in a new graph, $G'$. This is an $n$-class classification problem. For simplicity in the following discussion, we will consider a two-class example, but the algorithm, the performance measures and the classification concepts developed are applicable to problems with any number of

classes. Furthermore, the examples are sets of vertices. Again, for simplicity, we will use single vertex examples, but any number of vertices may be part of a single example.

Embedded Examples

Contrary to the situation in the preceding chapter, we do not have a set of disjoint positive and negative graphs. Instead, we can have one or more graphs that contain all of our examples, both positive and negative. Figure 4.1 below shows a graph containing some vertices that are members of one of two classes.

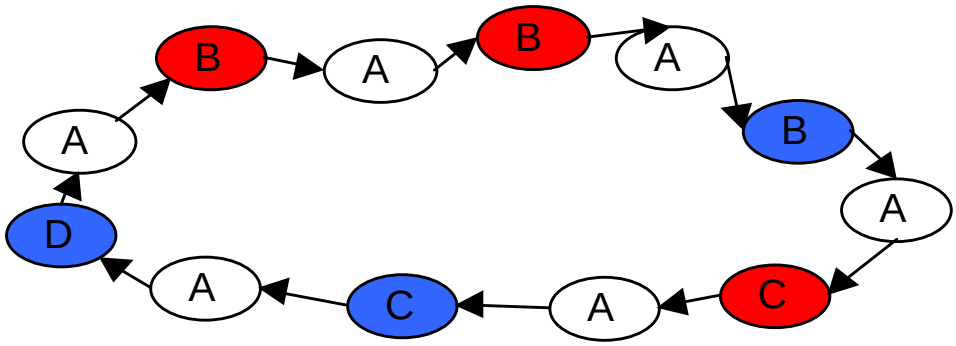



Figure 4.1 A single graph with both positive (red) and negative (blue) examples

Note that we could make this into a set of six disjoint graphs. We could then use them as input to Subdue-CL and seek a "positive" concept. To do so, we would have to guess at the size of the neighborhood to create around each example. If we decided a distance of two edges from each example was a good neighborhood size, then the six input graphs would be the ones shown in figure 4.2.

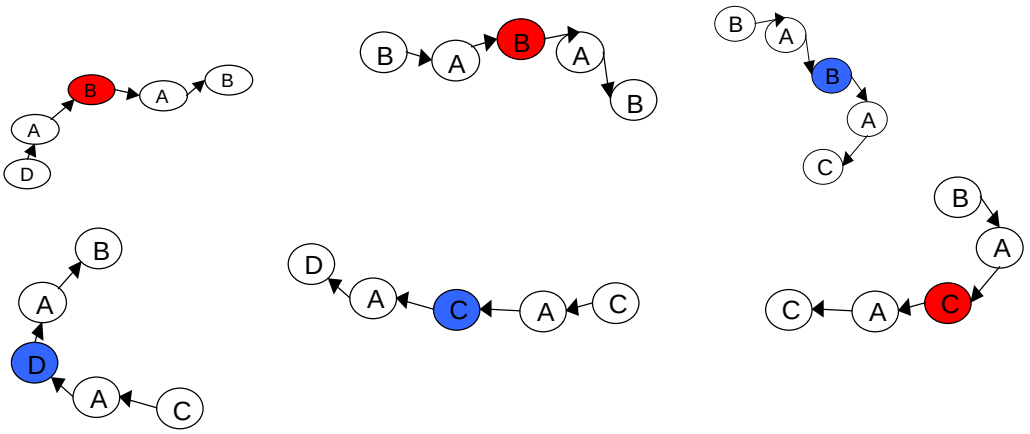



Figure 4.2 Six disjoint graphs containing one example each.

Now instead of twelve vertices and twelve edges, we suddenly have thirty vertices and twenty-four edges. In addition, we really do not know if we have a large enough neighborhood around each example vertex.

So, how can we utilize Subdue, without fracturing the graph? The first task is to determine a way to designate the examples. That is, we need to convey to Subdue exactly which of the vertices are included in each example. We choose to do this by augmenting the input graph with one more vertex for each example. We label this vertex with the class name and give it a special designation on input. We then connect this vertex to each vertex of the example with an edge. The label chosen for these connecting edges is arbitrary and could appear elsewhere in the graph, but it should be the same for all examples throughout the graph as should the directedness and direction. In figure 4.3 the examples are only one vertex in size and there are only two classes, but the method is not restricted to a specific example size or number of classes. Finally, the graph may contain any number of vertices that are not part of the examples such as the

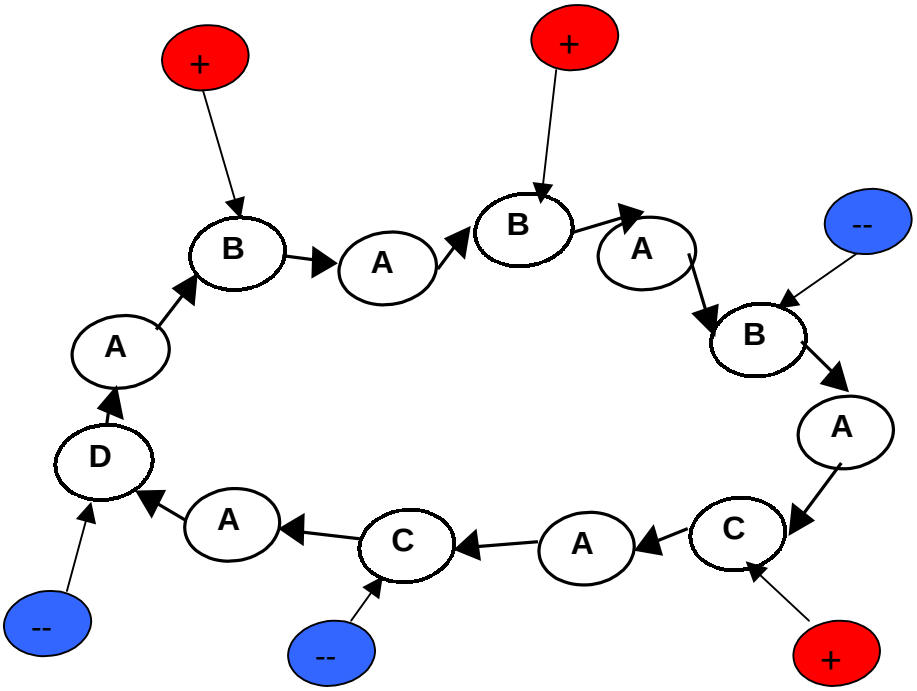



Figure 4.3 Graph augmented with example vertices.

"A" vertices in figure 4.3. Any vertices or edges from which there is a path to one or more vertices of an example may play a part in the classification of that example.

It would be much more efficient to use a single vertex to mark all of the examples. But it would still be necessary to show which vertices of a multi-vertex example were part of the same example. Unique edge-labels would accomplish this. With this scheme, a single vertex given a generic label such as "EXAMPLES" would be connected by edges labeled "EXAMPLE001" to all of the vertices of the first example. That same "EXAMPLES" vertex would also be connected to all the vertices of the second example vertex by edges labeled "EXAMPLE002". Although this scheme uses the same number of edges, it does reduce the number of added vertices to one regardless

of the number of examples. However, the number of labels required is much higher. Furthermore, special label-matching logic would be required to recognize that "EXAMPLE001" serves a similar function to "EXAMPLE002". Such a scheme would also require special matching logic during the application of the learned subgraphs to a novel graph. In addition, this still does not convey the class of each example to Subdue. So perhaps instead of a single vertex labeled "EXAMPLES", a vertex for each class could be used, connected by edges as above to all the vertices of all the examples of that class. Though we did not implement or test this scheme, we are confident that the program code for our chosen scheme is much simpler than that for the second scheme.

In should be obvious by now that we have avoided referring to the examples as subgraphs. They are sets of vertices but no edges are included in the examples. This does not preclude having examples that ARE subgraphs, but it is not necessary to mark these edges in any way. Any edges required for the purpose of distinguishing examples of one class from those of another, will be incorporated in the subgraph learned for that purpose. We do not require that each vertex of an example be in the learned subgraph and we would not require that each edge be included either. Therefore, we do not designate edges of examples that are subgraphs and thus we do not require that examples be subgraphs at all. Note also that the scheme does not preclude a vertex being part of more than one example, possibly even examples with different class labels.

## Classification Sequence

In concept learning, as implemented in Subdue-CL, there is always just one class, and that class is the "positive" class. The goal is to learn the concept exemplifying the "positive" class. The so-called negative examples are really just examples that do not exhibit the concept and would more accurately be called "not positive". They are referred to as negative, but they are not an independent, second class. Rather, these represent a default class.

If one executes a single iteration, Subdue-CL produces a "best subgraph" which can be used to identify new positive graphs. The mechanism used to identify positive graphs does not depend on the metric used when the subgraph was discovered. Even if one used MDL to discover the subgraph, a novel graph is considered positive if it contains an instance of the discovered subgraph. If the graph contains no instance of the discovered subgraph it is not positive.

It is also possible to run multiple iterations of Subdue-CL and obtain several subgraphs. If one is using compression as the metric, then at the end of each iteration the "best" subgraph is used to compress examples. If enough iterations are run, there will be only one vertex remaining after the last compression. One will have learned a hierarchical concept representing the positive examples (and not representing the negative ones). If one uses set covering as a metric, then at the end of each iteration the positive graphs covered by the best subgraph are removed from consideration in the subsequent iterations. The negative examples always remain and are input as part of the graphs to be processed in the subsequent iteration.

We are proposing an *n*-class classifier. In fact, one of our real-world domains on which we will demonstrate successful classification, contains three classes. For simplicity in discussing the new learner, and to allow comparison to the concept learner discussed above, we will use a two-class example, but the reader should keep in mind that problems with any number of classes can be solved by this method.

By simultaneously searching for subgraphs which identify examples of any class, rather than only ones which identify positive examples, subsequent iterations may have fewer examples to analyze and accuracy may be improved. In this approach, each iteration produces a subgraph that classifies examples as belonging to one of the *n* given classes – there is no default class that is being learned. All examples classified by the discovered subgraph are excluded from analysis in the next iteration. This would include any misclassified examples. In this regard the approach is similar to a decision tree (Winston, 1992) in that fewer and fewer examples are involved as the concept is refined. The techniques are also similar in that whenever we classify a new graph, we test for existence of the learned subgraphs in the order they were discovered, just as we progress down a decision tree in order from the root to a leaf, querying specific attribute values as we descend down the tree.

In Subdue's "one-class default" approach, the subgraphs can be applied in any order since only positive graphs are being selected. The order will not affect the classification (except perhaps for efficiency) since the concept is represented as a disjunction of subgraphs. Graphs that do not contain any of the discovered subgraphs are not positive. The discovered concept is simply a set of subgraphs. In our approach

the order of the subgraphs in the concept representation does matter. Because we eliminated all of the examples classified by each subgraph as we discovered it, the resulting set of classifying substructures is still a disjunction, but of a different form. When we apply subgraphs discovered in the order {A, B, C}, the resulting concept can be expressed logically as A ∨ (~A ∧ B) ∨ (~A ∧ ~B ∧ ~C). We therefore refer to the discovered subgraphs collectively as a *classification sequence,* represented by an *n*-tuple of subgraphs. Figure 4.4 represents the classification sequence learned from the graph in figure 4.3.

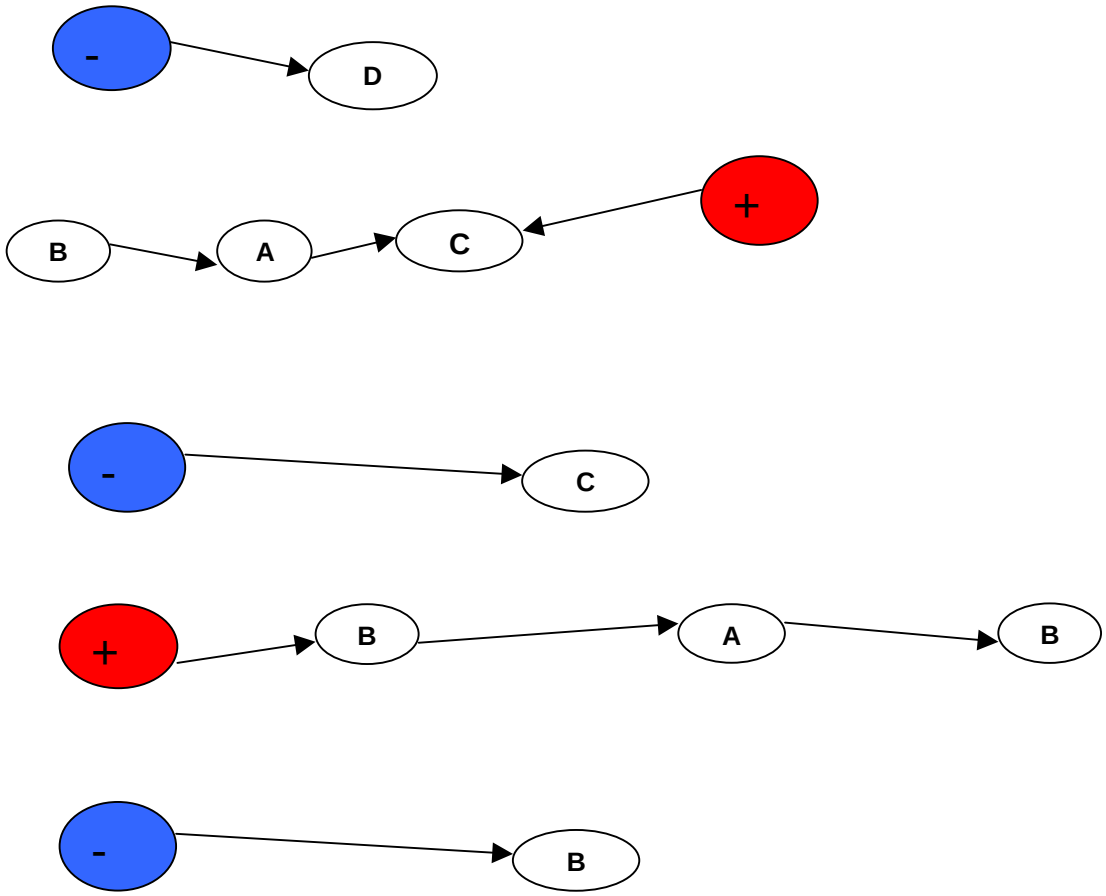



Figure 4.4 A classification sequence for the graph in figure 4.1, in order.

Without this enforced sequence some examples might be classified as members of more than one class. This can most easily be seen by looking at the last classifying substructure which contains only the vertex labeled "B". The positive "B" vertices were classified by the more complex classifying substructure B→A→B. As a result, the only remaining "B" vertices are all negative. This fifth substructure is a "catch-all" substructure that is minimally sized and sufficient to bring the number of unclassified substructures to zero in the last iteration of the learning process. Applying this last substructure first would no more produce correct classifications than would using the last branch of a decision tree without having traversed the tree from the root.

Another issue that arises in using the learned subgraphs for classification is related to the fact that instead of being given a set of disjoint graphs which we wish to classify with the learned concept, we may be given a single, large graph in which we must classify designated vertices. We do not have the luxury of being asked to classify several individual graphs as positive or not. One might be tempted to just find all instances of each classifying subgraph in the novel graph and declare them to be members of the corresponding class. However, that is not what we "learned" to do. What was learned was to classify designated sets of vertices in the context of a graph. So, we require that "areas of interest" in the novel graph be indicated. We use an added vertex as before, connected to vertices we wish to classify.

By virtue of the way the classifying subgraphs are grown, each subgraph that we discover will contain exactly one vertex labeled with the class name. To use a

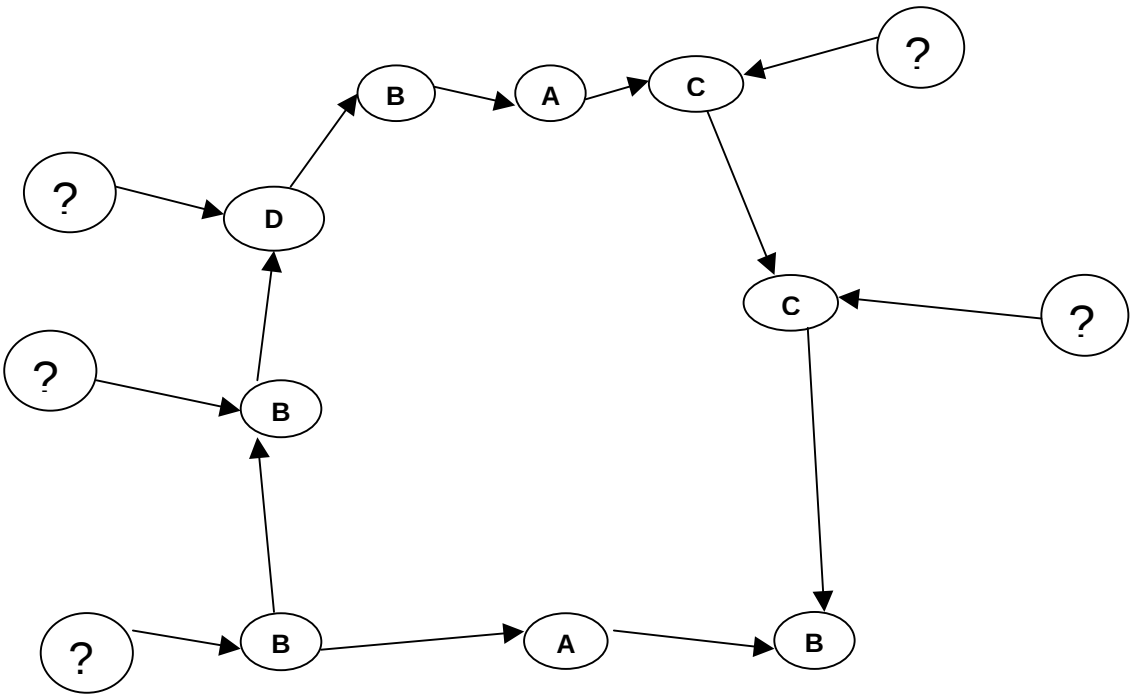



Figure 4.5 A novel graph with areas of interest marked by "?" vertices

discovered subgraph to classify an area of interest we align this class vertex with the vertex marking the area of interest. We then determine if there is an instance of the subgraph present at that place in the graph. That is, we try to find an instance of the subgraph "anchored" in the area of interest. If such an instance exists, then we classify that area of interest as the class represented by the subgraph. 4.6 shows the result of applying the classification sequence of figure 4.4 to the novel graph from figure 4.5. Each subgraph of the classifying sequence was tried, in turn, at each vertex containing a question mark until a matching subgraph was found. Only the first matching subgraph is shown in the figure. When the match is found, the class label on the classifying subgraph's class marker is the classification of the area of interest.

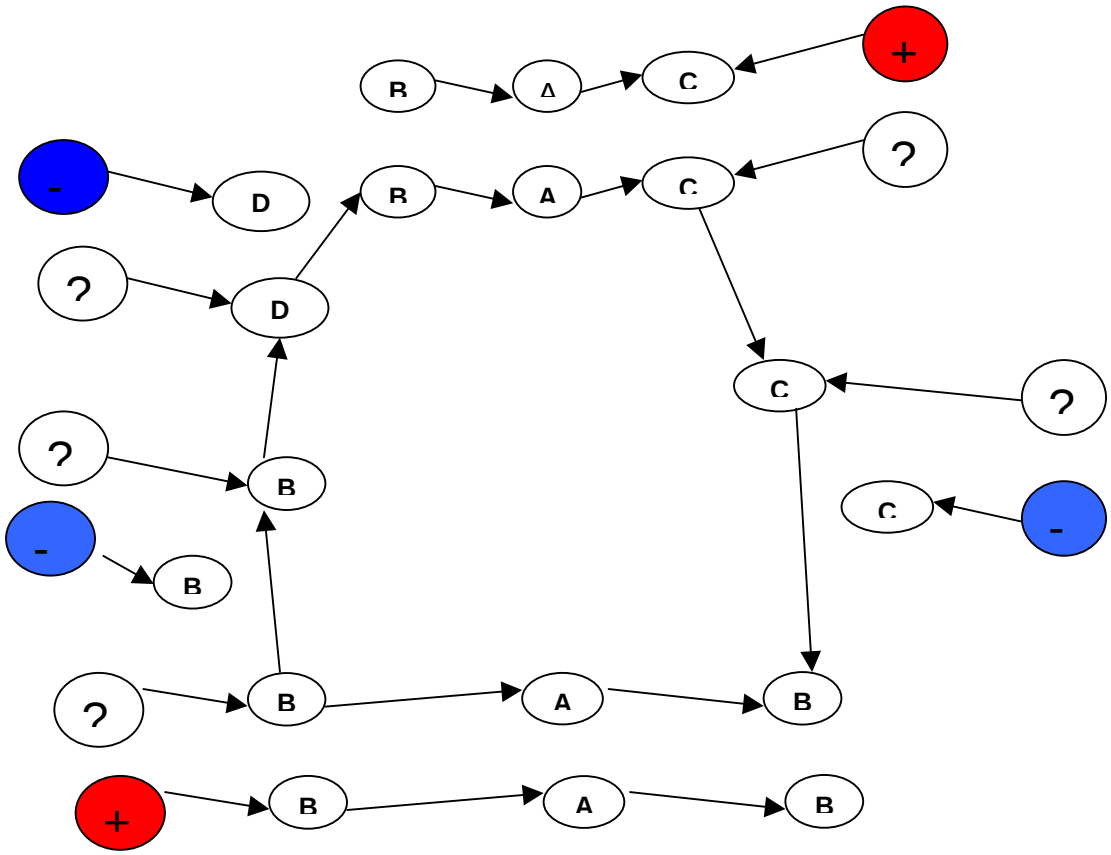



Figure 4.6 Novel graph classified with subgraphs from a classifying sequence

Evaluating Classification Sequences - Classification Compression

We have described the use of MDL by Subdue to find interesting substructures. We have also described the use of both MDL and set coverage by Subdue-CL to learn concepts from disjoint examples. Now we address the question of how we can guide our search for effective classification sequences.

We would like to use Subdue to learn classification sequences, but the MDL principle which drives it, does not look too attractive at first blush for discovering the "best" subgraph for embedded classification. Since all of the examples are contained in

a single graph, the subgraph that produces the greatest compression of that graph does not necessarily provide any information about classifying the examples. Likewise, we could use the set covering metric of Subdue-CL, but we do not even require our examples to be subgraphs. They are only collections of one or more vertices, so requiring examples to contain the classifying subgraphs makes little sense. Furthermore, the set covering measure does not constrain the size of the substructures in any way.

A discussion on classification and MDL in (Mitchell 1997) inspired an approach for using MDL in our algorithm. What we are seeking is a description of the classification of the examples we are presented. What we want to convey to the receiver, in the information theoretic sense, is neither the entire graph containing the examples nor the names of the classes, but merely the class to which each example belongs. So let us assume that the receiver has the graph, including the designation of the vertices which are examples and the class names. What the receiver does not have is the class associated with each example. It is the classification which we need to encode into a bit stream and "send".

Since the receiving agent can order the examples and the class names with the information it already has (for example using vertex id to order examples and class name to order the classes), all we really need to send is a class number for each example. The agent can then associate the appropriate class name with each example.

The description length of this naïve classification, $C_{naive,}$ is simply the number of bits it takes to provide a class number for each of the examples. Thus the $DL(C_{naive})$ is $N*log_2 K$, where $N$ is the number of examples and $K$ is the number of classes. Suppose

we discovered a classification sequence, $CS = (s_1, s_2, s_3, \ldots, s_n)$. We could encode this classification sequence and transmit it instead of the naïve classification. The agent could use the classification sequence as described above on the graph which it already has, and classify all of the examples. The description length of the classification sequence would be the sum of the individual description lengths of all subgraphs, $s_i$, in the sequence using our same graph encoding as before. So $DL(CS) = \sum DL(s_i)$.

We know there could be noise in class labels and there could be errors in the graph that would cause misclassifications. Since we do not have a default class to assign to unclassified examples, we must also address any examples that are not classified by any of the subgraphs in the sequence. Therefore, in addition to sending the classification sequence, we will also transmit an exception list to correct any mistakes made by the classification sequence and to classify any unclassified examples. The message to be sent consists of all of the example numbers that should have been labeled as class one, followed by all the example numbers that should have been labeled as class two, and so forth through the example numbers that should have been labeled as class K. We can separate the list for each class with a non-existent example number, say, zero. The description length of this exception list, *DL(EL)* will require $(K+M+U)*\log_2(N+1)$ bits. In this calculation *K* and *N* are as before, *M* is the number of examples misclassified by the subgraphs of *CS*, and *U* is the number of examples left unclassified by the subgraphs of *CS*. We use the log of *(N+1)* to account for an extra sentinel value which marks the end of the list for each class. A more efficient method could be used, such as Huffman

Coding (Storer, 1988), but the list should be short and the sentinel approach seems adequate.

If the description length of *CS* together with the description length of *EL* is smaller than the description length of the naïve classification, $DL(CS) + DL(EL) < DL(C_{naïve})$, then we will have reduced the message size required to convey the classification to our receiving agent. We will have compressed the classification using our classification sequence, *CS*. Now we can utilize the MDL principle once again. In the same way that Subdue seeks a subgraph that compresses a graph, we will seek the classification sequence which provides the best compression of the naïve classification. Likewise, in a fashion similar to the calculation of the compression of a graph by a subgraph, we can calculate:

$$Compression = \frac{DL(CS)+DL(EL)}{DL(C_{naïve})}$$

$$= \frac{\sum DL(s_1) + (K+M+U)*log_2(N+1)}{N*log_2 K}$$

As before, we take the reciprocal of the compression and use the resulting value as the evaluation measure for potential concepts (classification sequences). The classification sequence that yields the largest value is the "best" classification sequence.

Reasonable alternatives to this classification metric include set covering, as employed by Subdue-CL, as well as an accuracy measure such as classification error rate calculated as follows:

$$Classification\ Error\ Rate = \frac{True\ Positives + True\ Negatives}{Number\ of\ examples}$$

Classification error rate is the standard evaluation metric in machine learning research (Weiss and Provost, 2001). However, neither of these measures includes a penalty for the size of the subgraphs. In keeping with the Occam's Razor rationale (Blummer, 1987), a small subgraph should be preferred over a large subgraph with equal classification power. The classification compression measure will select a classification sequence with a smaller description length over one with a longer length if they both have the same total number of misclassified and unclassified examples. The classification compression measure is a means to an end. We are trying to find classifying substructures. We are not really interested in compressing the description of the classification per se. After all, the compressed classification structure is really only relative to the training data. The exception list for examples is not useful in classifying a novel graph. Remember that the goal is to find an ordered list of classifying substructures that we will use to classify the novel graph. We hope that classification compression will guide us to a sequence that does not make very many mistakes, is not too big and does not leave very much data unclassified.

Another possible metric sometimes used for measuring classification performance is Receiver Operating Characteristic (ROC) analysis (Provost and Fawcett, 1998). The primary advantage of ROC analysis is that it measures the performance of a classifier without regard to class distribution or error cost (Swets et al, 2000).

Unfortunately, it also requires varying the threshold value of the classifier, which is not a concept used in Subdue.

A weakness of the classification compression metric is that there is no weight assigned so that the user can express a preference for, or assign a cost to, either misclassification or non-classification of examples. It would be a minor extension of the metric to assign a weight to one or both. This could most easily be done in the computation of the description length of the error list.

Another observation about the underlying "message" concept of classification compression is that the receiver may spend a great deal of time looking for matches to the classifying substructure. Doing so constitutes an *NP*-complete problem. The worst case is never encountered since the receiver is always starting from example nodes and does not have to search the entire graph. Though the cost of decoding the message is not part of the MDL principle and is not normally considered, one could include a "match cost" in the compression classification calculation value computation which would provide a measure of how "bad" the search process might be for the substructures contained in a particular classification sequence.

Since the classification compression does not currently penalize unclassified examples differently than misclassified examples, it was found that some intuitively bad classification subsequences of very short length were sometimes chosen as best. In the extreme case, the single vertex subgraph containing the "_EXAMPLE" vertex matches all examples. It is very small and leaves nothing unclassified. The number of misclassifications is dependent on the ratio of majority class examples to the remaining

examples. Reducing the number of misclassifications made by this subgraph requires a larger subgraph. However, this will probably increase the number of unclassified examples. If growing the subgraph only results in a longer description length for the subgraph and trades misclassified examples for unclassified examples, there may be no gain in classification compression and the search process may halt with the initial, one-vertex subgraph. The simple fix is to require a minimum subgraph size of two vertices. Of course, a similar situation can develop with larger subgraphs.

To overcome this potential problem, we also evaluated a second metric. Our goal was to identify an evaluation measure which would guide us toward a high classification compression value. To achieve a high classification compression value, one should correctly classify a large number of examples with a small classification sequence. Also, at any single iteration, we will prefer unclassified over misclassified examples since we may be able to classify an example properly later, but if we misclassify it now, it is misclassified forever.

To this end we developed a measure of *hits per bit,* or HPB. HPB is the number of correctly classified examples divided by the sum of the description length of the subgraph and the description length of the error list for misclassifications only. It can be thought of as the efficiency with which a subgraph classifies examples expressed in terms of correct classifications per bit of length. By selecting the subgraph with the largest HPB in each iteration, we may learn a classification sequence with a large classification compression value.

## The Learning Algorithm

The search for a classification sequence was implemented by extending the Subdue system. The discovery of substructures is performed as described in Chapter 3, but the initial state for the search is different. The evaluation process for the candidate substructures has also been changed. We call this implementation Subdue-EC for Embedded Classification.

Immediately upon input of a graph to Subdue-EC the vertices containing the class designation of the examples are relabeled with the label "_EXAMPLE". The class name is entered in a list along with the vertex id of each relabeled vertex connected to an example of that class.

The initial search state for Subdue consists of all single vertex substructures. For Subdue-EC we start with the initial state containing only the set of all "_EXAMPLE" vertices. This may be a very large reduction in the size of the initial state. We can justify this reduction based on the way the completed substructure will be used. Recall that when we later classify with the substructure, we will anchor the "_EXAMPLE" vertex on the marker vertex for the vertices we are trying to classify. If we grew a substructure that did not contain an "_EXAMPLE" vertex, it could not be used for classification. So it is sufficient to start growing from the "_EXAMPLE" vertex. We also must ensure that the classifying substructure has only one "_EXAMPLE" vertex. We therefore have altered the expansion routine to preclude the addition of any vertex labeled "_EXAMPLE" during the expansion.

The evaluation step of Subdue-EC is slightly more elaborate than traditional Subdue since we do not assume we are looking only for substructures that classify examples of just one class. Indeed, we are searching for a classifier that works well on any one of our classes. So after building a substructure, we have to decide which class will be associated with it. That is, we must decide what class label to attach to this subgraph based on all the instances we have found in the training graph. Ideally, the candidate substructure would only have instances of a single class. But due to errors in the data, noise, or simply the lack of discriminatory ability of the substructure, instances of a subgraph may belong to different classes. Generally we choose the class which results in the most correct classifications. We break ties by choosing the class which classifies the largest proportion of remaining examples in the classes. Once we decide which class the substructure classifies, the calculation of value takes place according to the user-specified evaluation method, either classification compression or hits per bit.

As with Subdue 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 Subdue-EC returns the list of best substructures found, the "_EXAMPLE" vertices for examples classified by the best substructure are removed from the graph and a new iteration begins. Note that the classified examples themselves are NOT removed, just the "_EXAMPLE" vertices attached to them. All the vertices of the example are still in the graph, available for building classifying substructures in subsequent iterations. Unlike traditional Subdue, the graph does not have to be

compressed, so it does not have to be copied and rebuilt, a resource-intensive operation for large graphs.

It is also important to point out that ALL examples classified by the "best" substructure are removed, not just the correctly classified ones. The reason is that they must not be allowed to influence the selection in the next iteration Since the classification sequence is used in order of discovery, any areas of interest in a novel graph that are classified will not be available to subsequent classifying subgraphs even if they are classified wrong. Therefore, misclassified training examples should not be available in the next iteration after they have been misclassified by the "best" substructure.

Improving Multiple Iteration Learning

During implementation it was observed that taking only the best substructure from each iteration may not result in the best classification performance of the entire classification sequence. This problem is not unique to Subdue-EC but rather is common in greedy algorithms in general. Because of the minimal resources involved in maintaining and applying multiple sequences, an additional beam search was implemented for the classifying sequence itself. The user specifies a "sequence beam" width defining the number of classification sequences to maintain at any time. At the end of the first iteration, the sequence list is filled to a limit of "seqbeam width" with the single-substructure classification sequences composed of the best substructures. To begin the next iteration, the best classifying sequence is applied to the initial graph. In other words, all examples classified by that sequence are removed from the graph,

including misclassifications. Then the best substructures are found as before. These best substructures are added to the classifying sequence to create several new classifying sequences. The new sequences are placed in a child queue based on their sequence evaluation method. The "_EXAMPLE" vertices are then restored to their original condition and the process is repeated using the next classifying sequence in the list. This is completely analogous to the extension and evaluation of substructures. Like the substructures, we calculate a value for the classification sequence after it is extended by another subgraph. This value can be calculated using either classification compression or hits per bit calculated for the entire classification sequence. At termination the best classification sequence can be output to a file for classifying novel graphs.

Other Possible Ways to Learn Classifying Substructures

We start at each of the vertices of each example and grow outward. We hope that we are growing these subgraphs "toward" any classifying substructures that might exist in the graph. It is possible that we could grow a large subgraph that is of no use in discriminating between classes. Suppose that we had a graph containing two classes of vertices. Vertices that are labeled A and are connected by a chain of ten vertices labeled B to a vertex labeled X are in class 1. Vertices that are labeled A and are connected by a chain of ten vertices labeled B to a vertex labeled Y are in class 2. Thus our training graph has several instances of the chain ABBBBBBBBBBX and several of the chain ABBBBBBBBBBY. This is an example where the concise discriminating subgraph (the single vertex substructure X or Y) is located "far" from the vertex "A" that we are learning to classify.

We acknowledge that the concise subgraph that allows discrimination between examples of different classes might be far from the example. If we allow the substructure to grow far enough, we will reach that concise subgraph, and incorporate it into the larger classifying subgraph we are growing. Each instance of the subgraph will contain at least one vertex of the example it classifies because there is a path back to the example vertex. However, it is frustrating to visually examine this training graph and immediately intuit that examples of class 1 are connected to X and examples of class 2 are connected to Y. It raises the question, should we start elsewhere and grow toward the examples rather than starting at the example vertices and growing away?

As mentioned above, one good reason for the approach we take is that we almost surely have fewer vertices in the initial search state. In this example we would be expanding all of the "B", "X", and "Y" vertices in addition to the "A" vertices. Ordering the single-vertex substructures by frequency of occurrence might be a rewarding strategy for some graphs. Imagine if all of our class 1 examples were connected to a single "X" vertex, for example. Then expanding that uniquely-labeled vertex first would be quite productive. These are all interesting situations to contemplate, but we believe our current strategy is a good one because 1) every instance of a substructure that we expand has one and only one "_EXAMPLE" vertex; 2) we will never have more initial substructures to expand than if we did them all; and 3) we are very likely to have far fewer initial substructures to expand.

An entirely different approach to finding classifying substructures could be based on frequent subgraph discovery (Kuramochi and Karypis, 2002). A very efficient

algorithm exists for discovering subgraphs that occur frequently in a graph (Kuramochi and Karypis, 2004). This could be used to find subgraphs from which to start growing classifying subgraphs that could then be evaluated using the same metrics as above. Of course, it is not clear at all that the frequently occurring subgraphs will be the ones that are able to classify examples.

A final thought along these lines also involves frequently-occurring subgraphs but a different evaluation mechanism. Suppose that it is the proximity of particular subgraphs that discriminates between classes. In this situation it is conceivable that there might be no "path" connecting the classifying substructure to a vertex in the example that is common among all examples of a class, even though the same substructure is commonly located "close" to each example. This would mean no matter where one started growing a substructure, either from the example end or the substructure end, one might never discover a substructure which contained both the examples' vertices and the initial subgraph. In this case a distance-based evaluation measure could be developed and the "best" classifying substructure would be the one which discriminates between the classes of the examples it is "close" to. Such a distance measure might be based on edge labels or solely on shortest path distance measured in number of edges.

None of these alternate strategies was implemented or tested. They are discussed here as alternatives that were considered and provide ideas for future work.

# CHAPTER 5

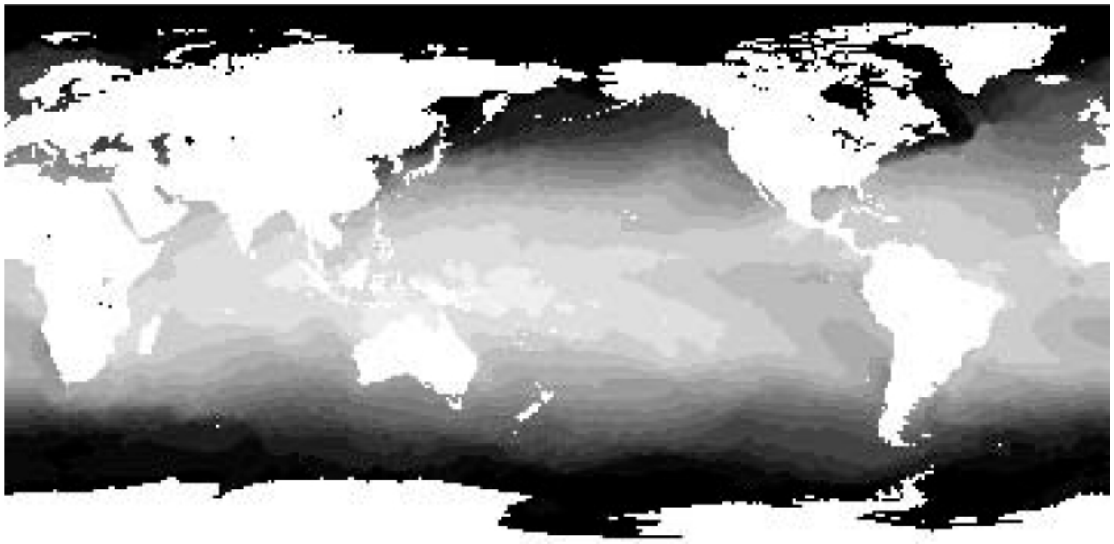
# EXPERIMENTAL RESULTS

Subdue-EC is designed to solve a particular problem; namely, learning concepts from examples embedded in a single graph. It is quite capable of learning from examples which are each represented by a disjoint graph. Subdue-EC can also learn from multiple disjoint graphs, each containing any number of examples. The algorithm is not unique in this aspect, but its unique classification sequence concept and evaluation criteria might offer an advantage over other classifiers when examples are each represented by a disjoint graph. A comparison of learning from examples in disjoint graphs to traditional learners appears in Gonzales (Gonzalez, Holder, Cook, 2002). Therefore, we did not run Subdue-EC on data traditionally used for testing classifiers. Rather we sought data for which a single graph was an appropriate representation. We created one synthetic set of data and used two real world sets of data for our remaining experiments.

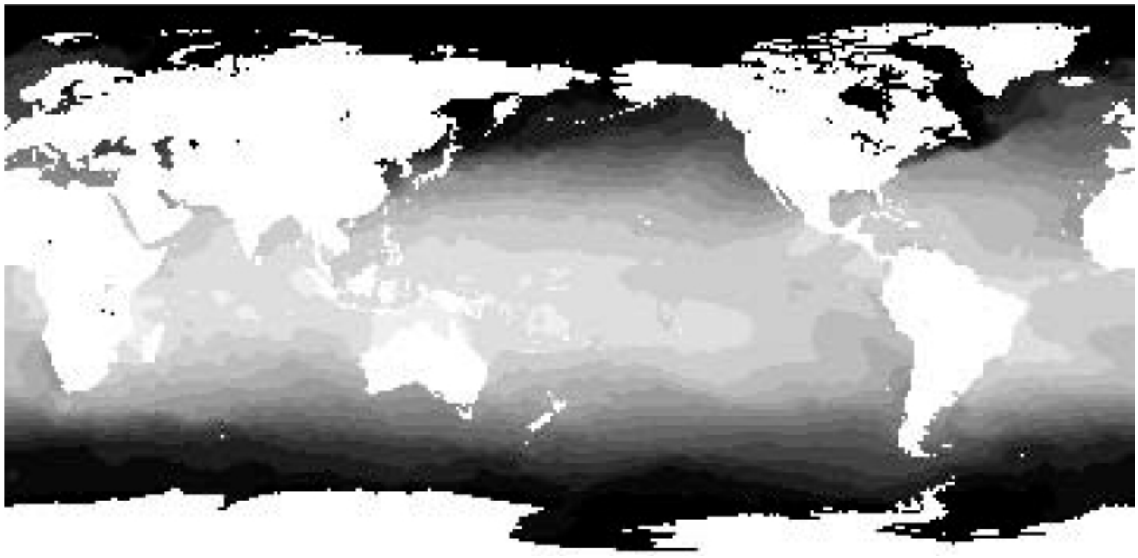
## Data Used for Experiments

We constructed a synthetic set of data based on the graph in figure 4.1. This twelve-vertex graph was replicated ten times to provide an input graph with 120 vertices and 120 edges. It contains thirty positive examples and thirty negative examples. We refer to this graph as the “testclsx” graph.

For the second set of test data we obtained sea surface temperature (SST) data from NASA (JPL 2000). This data is averaged over a five day period and placed on a one degree global grid. The data contains a fill value for grid points for which the SST is not available, such as those on land or due to missing information.



(a)



(b)

Figure 5.1 Sea surface temperature on (a) January 8, 1990, and (b) February 7, 1991. Lighter areas are warmer

We first determined for each grid point whether the temperature INCREASED, DECREASED or stayed the SAME from January 8, 1990 (figure 5.1a), to February 7, 1990 (figure 5.1b). Figure 5.2 shows whether the temperature increased, decreased or stayed the same at each grid point.

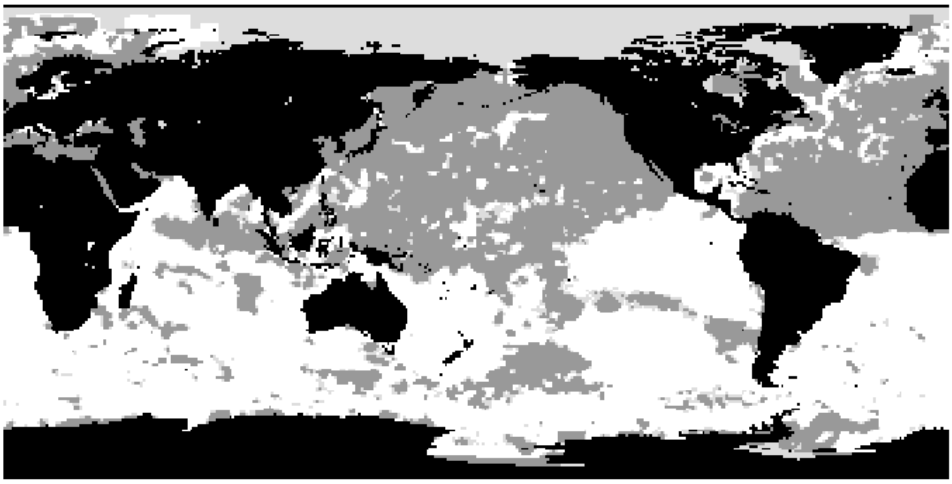


Figure 5.2 Sea surface temperature change over one month. Dark gray indicates decrease. Light gray indicates no change. White indicates increase.

We then placed the non-fill temperature values into one of 9 equal-width bins. We created a graph composed of one vertex for each grid point (labeled "JAN"). Each JAN vertex was connected to its neighbor on the west by a directed edge labeled "W" and its neighbor on the north by one labeled "N". The westerly edges continued in a circle around the entire globe. The northerly edges ended at latitude 89.5 N. The grid would thus look like a mesh cylinder. Each grid point also was connected to a unique vertex containing its temperature bin or the fill value by a directed edge labeled TEMP

and to another unique vertex labeled N or S by an edge labeled HEMI. This value was based on the node's latitude.

From this structural data, we then created a training graph by randomly selecting 90% of the nodes to which we attached each node's class vertex. This vertex was labeled , DECREASE or SAME depending on whether the temperature at that node was higher, lower or unchanged after one month. A single JAN vertex with its HEMI, TEMP, and class vertices and its edges that would connect to neighboring JAN vertices is shown in figure 5.3.

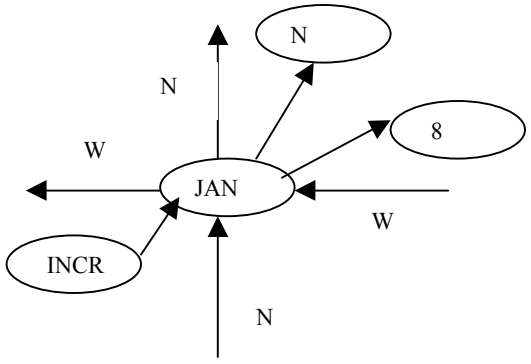



Figure 5.3 Graph Representation of a single SST grid point

This formed a graph containing 259,200 vertices and 323,640 edges. We will refer to this data as the SST data. We also created a subset of this graph that contained a ten degree wide slice from pole to pole. This smaller graph is a rectangle, 10 X 180 degrees and is referred to as "diff180".

The second set of real-world data is from the Homeland Security Domain (Holder et al, 2005). As part of a government-sponsored program, a domain has been

built to simulate the evidence available about terrorist groups and their plans prior to execution. This domain is motivated from an understanding of the real problem of intelligence data analysis. The domain consists of a number of concepts, including threat and non-threat actors, threat and non-threat groups, targets, exploitation modes (vulnerability modes are exploited by threat groups, productivity modes are exploited by threat and non-threat groups), capabilities, resources, communications, visits to targets, and transfer of resources between actors, groups, and targets.

The domain follows a general plan of starting a group, recruiting members with needed capabilities, acquiring needed resources, visiting a target, and then exploiting the target. The data we use for our experiments represents the activities of terrorist organizations as they attempt to exploit vulnerable targets, represented by the execution of five different event types. They are:

- Two-way-communication: Involves one initiating person and one responding person.
- N-way-communication: Involves one initiating person and multiple respondents.
- Generalized-transfer: One person transfers a resource.
- Applying-capability: One person applies a capability to a target.
- Applying-resource: One person applies a resource to a target.

All data is generalized so that no specific names are used. The simulator generates evidence related to all of these events, and this evidence is passed through filters varying the degree of observability and noise in the final evidence.

For our experiments, a graph was created in which vertices are used to represent member agents from Threat and Non-threat groups. Anyone with whom these agents communicate is also added to the graph and connected to the agent with an undirected "association" edge. Communication events between associates are similarly represented with "association" edges.

In addition, each person may be described using attribute and capability vertices. In the simulated data, every individual is assigned at least two strong "trust-link" attributes (e.g., school, place of worship, former military unit, extended family) and at least two weaker "culture-link" attributes (e.g., nationality, language, religion) that are commonly applied in social network development. Capabilities refer to unique abilities exhibited by the individual. Figure 5.4 shows a portion of the graph generated for this dataset. We will refer to this data as the Eagle data. We initially generated two graphs from this domain. Graph1 contains 496,534 vertices and 824,609 edges. The graph has 1732 examples of known terrorist threats and 59,373 examples of non-threats. There are 5,608 unique weak link attributes, 6,855 unique strong link attributes, and 150 unique capabilities.

Graph 2, the second graph constructed, is somewhat smaller. It contains 248,616 vertices and 345,508 edges. The graph has 1225 examples of threats and 25,490 examples of non-threats. There are 5,562 unique weak link attributes, 6,792 unique strong link attributes, and 150 unique capabilities.

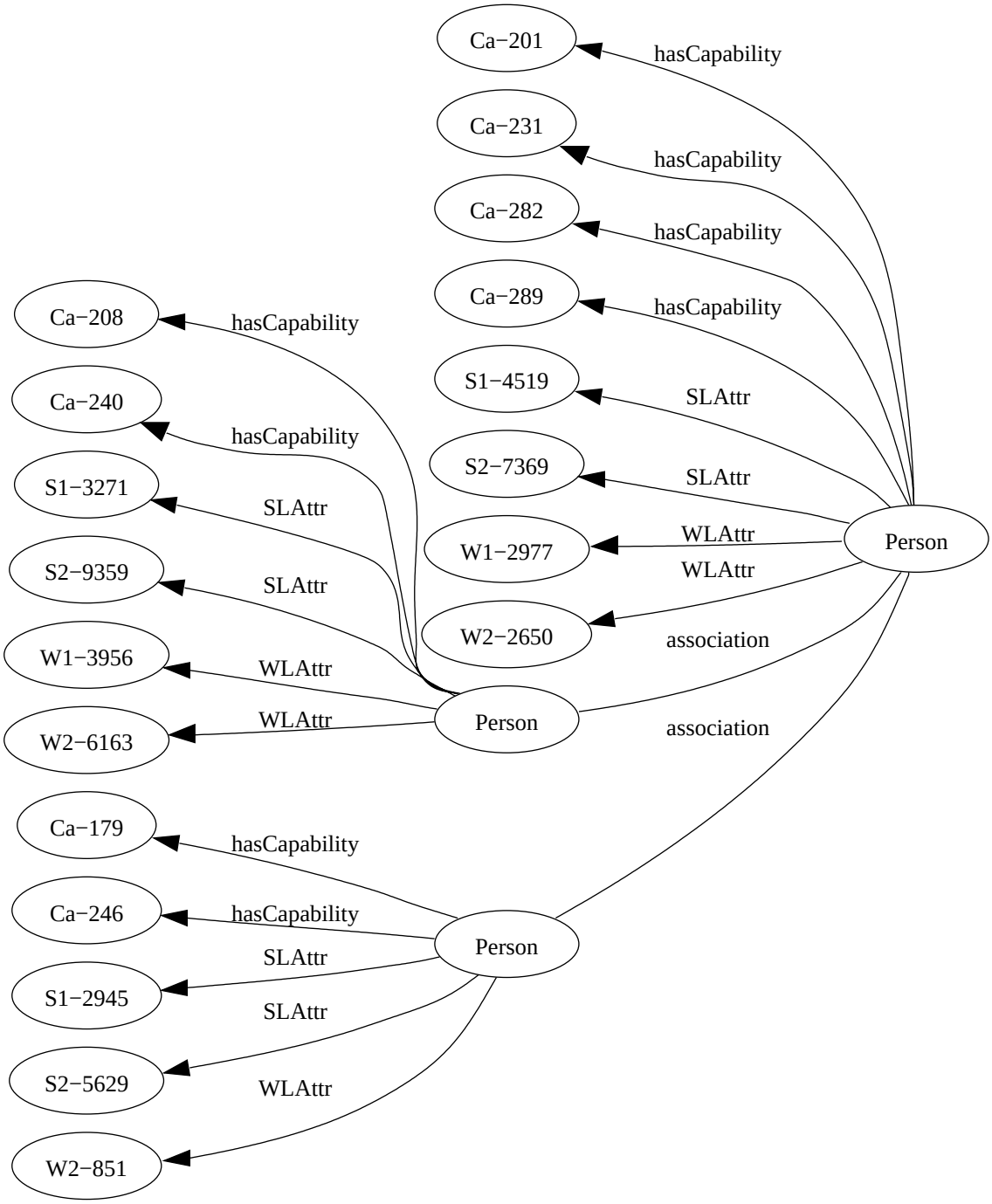



Figure 5.4 A portion of a graph from the Homeland Security Domain

## Execution of Experiments

There are many command line parameters which affect the performance of the Subdue family of learning systems. In the following descriptions of experiments, the parameters are generally set as follows: limit, the number of substructures to be evaluated is usually set to ten; beam width varies from experiment to experiment and will be given in the description, though generally it is three or four; for Subdue-EC runs, the minimum size of a substructure to be considered is generally set to two vertices, since the first vertex is always the "_EXAMPLE" vertex. Most of the SST and Homeland Security Domain problems were run one of two systems in the UTA AI Lab. Each of these machines has two hyperthreaded, 2.40GHz Intel Xeon processors and 2,064,528 kb of memory. The reported run times were obtained during the normal operation of the machines in a lab environment. That is to say, the experiment in question was almost surely not the only process running on the machine at that time. To that extent, the run times might be different if the very same job were run a second time. Finally, we include a note about Subdue itself. Subdue-EC is based on version 5.06 of Subdue. The current version of Subdue, available on the Subdue web site (http://cygnus.uta.edu/subdue/), is version 5.10. However, version 5.06 was unable to complete problems using the real-world data sets that we had constructed. We therefore wrote a new extendSubstructure routine based on a different approach. This approach substantially reduced the number of calls to a routine that matches one graph to another, an extremely expensive operation. These changes are detailed in Appendix A. This allowed complete runs on the SST data and partial runs on the Eagle data.

## A Simple Two Class Experiment

A graph constructed from the twelve vertex circular graph shown in figure 4.1 should provide a simple classification challenge for Subdue-EC. We would expect to learn the five subgraphs shown in figure 4.4. The purpose of this experiment is to compare the classification performance of Subdue-EC to a learning approach that relies on learning from disjoint graphs. To compare this to learning from disjoint graphs, we choose a fixed-size neighborhood around each example vertex that includes all vertices that are within two edges. We will then have six disjoint graphs, one for each example for input into Subdue 5.10. This neighborhood size is an arbitrary decision but since we know what the classifying substructures should be, we know it should be an adequate size. These are the disjoint graphs shown in figure 4.2.

Surprisingly, only Subdue 5.10 using MDL evaluation and the six disjoint graphs successfully learned substructures capable of correctly classifying all of the training examples. Subdue-EC learned classifying sequences of three single vertex substructures using both HPB and Classification Compression, but both of these sequences resulted in two false positives. Using HPB Subdue-EC learned the sequence {B +, D -, C +}. Using classification compression it learned {D -, B +, C +}. Recall that Subdue 5.10 only learns to classify the positive examples and declares anything unclassified to be negative. Subdue 5.10 learned the disjunct {A→B→A→B ∨ B→A→C→A→C}. These two substructures compress all three positive examples, but none of the negative ones. Using the alternative evaluation measures available in Subdue 5.10 was not successful. Evaluation "size" found the disjunct {A→B→A→B ∨

B $\vee$ A $\vee$ C} which compresses all examples, thus resulting in three false positives. Evaluation "Set Cover" found the single subgraph B$\rightarrow$A which actually covers all of the positives and one of the negatives, resulting in one false positive.

This simple example serves as a grim reminder that despite what we might wish, sometimes the arithmetic involved in the calculation of the "value" of a substructure simply does not cause the selection of what seems obvious to us. The penalty terms in this case are just not large enough to support the growth of larger structures. We can merely replicate the input graphs nine times and there will be ten times as many examples. Then Subdue-EC will behave as expected, producing the five subgraphs shown in the classifying sequence in figure 4.4. This also causes the "size" evaluation measure of version 5.10 to find the same disjunct as the MDL measure. It has no effect on the substructures found for the "set cover" evaluation.These results illustrate a shortcoming of the classification compression measure. If there are not very many examples, it is cheaper in terms of bits of description length to encode the exceptions than it is to encode the substructures. The substructure description length (DL) is related to its structure (and to the number of labels), but the size of the exception list is related not only to the number of exceptions but also to the total number of examples. One solution for this would be an explicit weighting for misclassifications and unclassified examples.

## A Three Class Experiment

Subdue-EC is capable of leaning from data containing examples of any number of classes. The SST data was used to conduct a test of this ability (Potts et al. 2005a). This data contains three classes, grid points where the sea surface temperature increased, those where it decreased, and those where it stayed the same.

We made a training graph and a test graph by starting with two identical copies of the grid graph as described at the beginning of this chapter, but without the class vertices attached. We randomly selected ninety percent of the grid nodes to be used for training and attached their class vertices in the training graph. The remainder of the nodes had their class vertices attached in the other copy of the graph to become the test graph. Ten sets of training/test graphs were constructed and tested with the results shown in table 5.1.

Table 5.1 Ten Fold Cross Validation

| run # | subs | secs. | % correct | |
|---|---|---|---|---|
| 0 | 106 | 52822 | 86.07% | 85.31% |
| 1 | 104 | 49669 | 85.81% | 85.26% |
| 2 | 109 | 76336 | 85.57% | 85.25% |
| 3 | 100 | 71679 | 85.81% | 85.32% |
| 4 | 104 | 78874 | 85.66% | 85.69% |
| 5 | 111 | 80388 | 85.84% | 86.22% |
| 6 | 108 | 73174 | 85.84% | 85.00% |
| 7 | 112 | 77236 | 85.81% | 85.39% |
| 8 | 99 | 75392 | 85.64% | 84.57% |
| 9 | 108 | 80497 | 85.76% | 84.10% |
| Min | 99 | 49669 | 85.57% | 84.10% |
| Max | 112 | 80497 | 86.07% | 86.22% |
| Avg. | 106.1 | 71607 | 85.78% | 85.21% |

The first column of "% correct" reflects accuracy on the training data. The second column is based on applying the learned sequence of classifying substructures to the corresponding test data held back (6,480 grid points). The accuracy on the test data was consistent with the accuracy of the training data. The reported accuracy of 85% seems unexpectedly low and raises two questions: How much better is it than just guessing? and Why is it so low? If we made guesses in proportion to the known class distribution, we would expect to get about 35.8% of the grid points correct. We would do better always guessing the majority class of "SAME". This would result in an accuracy of 44.5%. So we did substantially better than chance. But why are the results so low? Simply put, there is not enough information present in the data to make the correct determination all of the time. We have no information at all on longitude and only the coarsest information on latitude. See below for a comparison of these results to other methods.

## A Large Scale Test of a Learned Classifying Sequence

After showing through cross-validation that Subdue-EC was capable of learning on the SST data, we sought to determine if the classifying sequence was general enough to use in the following year. We trained Subdue-EC on 100% of the January 1990 data and then applied that learned sequence of classifying substructures to the 1991 data. That is, we created a similar graph for the same dates in the following year January 8, 1991 to February 7, 1991, determined the correct classification for all of those grid points, and then applied the classifying substructures learned from the 1990 graph to this new,

previously unseen 1991 graph. As shown in Table 2, using the classifying substructures learned from the January/February 1990 graph, we were 81.98 % accurate in predicting the SST direction of change for January/February 1991, one year later. Table 5.2 shows how we classified each of the three classes of grid points in the 1991 graph.

Table 5.2 Classification of the Entire Training Set

| Actual | CalledSame | CalledDecr. | Called Total | Incr. |
|---|---|---|---|---|
| Same | 25473 | 1076 | 1079 | 27628 |
| Decr | 1474 | 9737 | 2487 | 13698 |
| Incr | 1633 | 3925 | 17916 | 23474 |
| Total | 28580 | 14738 | 21482 | 64800 |
| | Called | Right: | 53126 | 81.98% |

As a final confirmation that we had learned something we constructed a graph for July/August of 1990, a time period six months away from the training period. We would not expect sea surface temperatures to behave the same from July to August as they did from January to February. We would therefore expect our learned sequence to do poorly on this test graph. Table 5.3 shows the results of applying the classifying sequence learned for the temperature change from January to February to the period from July to August. The accuracy on the "Same" class showed only a small decrease. Since sea surface temperatures are undefined for land masses, all grid points over land are classed as "SAME". That is one reason we continue to do well on the SAME class even in the summer time. Another reason is that the extreme polar regions also stay the same year round. But note the poor accuracy on both the INCREASE class and the

DECREASE class. This is due largely to classifying INCREASE as DECREASE and vice versa. It is exactly what you would expect if you learned a winter weather pattern and applied it to a summer globe.

Table 5.3 Applying the 1990 Winter Classifying Sequence

| | ----------Classified as--------- | | | Total |
|---|---|---|---|---|
| for next yr | Same | Decr | Incr | Correct |
| Same | 25473 | 1076 | 1079 | 27628 |
| Decr | 1474 | 9737 | 2487 | 13698 |
| Inc | 1633 | 3925 | 17916 | 23474 |
| | 28580 | 14738 | 21482 | 53126 |
| for next summer | | | | 81.98% |
| Same | 22759 | 547 | 2730 | 26036 |
| Decr | 1753 | 3007 | 17532 | 22292 |
| Incr | 3316 | 10611 | 2545 | 16472 |
| | 27828 | 14165 | 22807 | 28311 |
| | | | | 43.69% |

A Large Two Class Experiment

As previously mentioned, manipulating graph data involves many extremely resource intensive operations. The original Eagle data graphs were simply too large to be processed to completion by Subdue-EC in a reasonable amount of time. Since we had far more non-threat examples than threat ones, our first attempt to reduce the problem size was to randomly select non-threat examples so that we had an equal number of non-threats and threats. The new subset of graph1 contains 438,893 vertices and 766,968 edges. The new subset of graph2 contains 220,351 vertices and 317,243

edges. This did allow us to start classifying in both graph1 and graph2 of the Eagle data (Potts et al. 2005b). However, after identifying 16 to 20 classifying substructures, Subdue-EC encountered the substructure containing three "person" vertices. In graph2 there are 379,380 instances of this subgraph. Extending all of these instances by one edge is what halts progress of any attempt at fully classifying this data with Subdue-EC. The point at which this particular substructure is encountered in the expansion process depends on many factors but we have always encountered it no later than the twentieth iteration. Table 5.4 shows the results of twenty substructures learned for graph1. The accuracy on the training data is 72.98%.

Table 5.4 Classification Results on Graph1.

| | Total | Correct | Incorrect | Unclassified |
|---|---|---|---|---|
| Threats | 1732 | 765 | 35 | 932 |
| Non-threats | 1732 | 70 | 290 | 1372 |

Due to the computational limit described above, 2,304 individuals remained unclassified. The greatest number of misclassifications were false positives (classified as a threat when the true classification is non-threat), which is a preferred type of

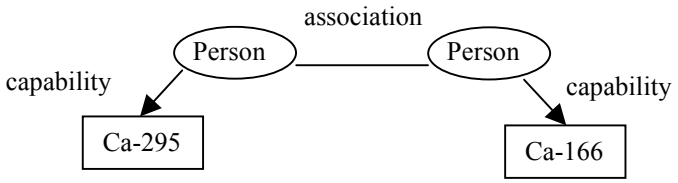



Figure 5.5 A sample discovered substructure showing an association between two individuals, each with certain capabilities. The individual on the left is a known terrorist threat.

mistake for this problem. Recall and precision were 0.956 and 0.725 respectively. Sensitivity and specificity were 0.956 and 0.194 respectively.

Of the substructures that were discovered, many consisted of an individual exhibiting a particular capability. However, a few of the substructures, such as the one shown in figure 5.4, highlight an association between two individuals in addition to attributes and capabilities of the individuals. Twenty iterations on this data took 70.2 hours utilizing the RB-tree option and a seqBeam of 0.

In a separate experiment, we evaluated the generalizability of Subdue-EC's results by using the substructures discovered in the first experiment to classify individuals from a separate dataset, graph2.

Table 5.5 Classification Results on Graph2

| | | correct | incorrect | uncl. |
|---|---|---|---|---|
| Threat | 1225 | 463 | 28 | 734 |
| Non-Threat | 29490 | 876 | 5596 | 23018 |

Recall and precision were 0.943 and 0.076 respectively. Sensitivity and specificity were 0.943 and 0.135 respectively. We must ask if the learned classification is of any value given its apparent poor accuracy and poor coverage. We assert that it is by the following argument: With the few substructures we discovered, we can only classify 23% of the people of interest. On the other hand, we know which ones we can and cannot classify. That is to say, given a person of interest either we can say we cannot provide any additional information or we can provide a classification.

Furthermore, for the ones we can classify, the probability of a person of interest being a threat when we say he is a threat based on the classification sequence is .0764. The a priori probability of a person of interest in the novel graph being a threat is .0398. We therefore have twice the likelihood of properly identifying a threat. The a priori probability of a person of interest in the novel graph being a non-threat is .9601. The probability of a person of interest being a non-threat when we say he is a non-threat based on the classification increases to .969.

## Comparing Subdue-EC to Disjoint Example Classification

We wanted to compare Subdue and Subdue-EC on both the Eagle data and the SST data. However, Subdue was unable to complete the SST data and Subdue-EC was unable to complete the Eagle data. In order to give Subdue a fair chance we decided to use the Eagle data and compare it to a partial classification sequence discovered by Subdue-EC. Since Subdue requires one graph per example, we excised a neighborhood of people around each example and output that as a separate graph. We limited the neighborhood to people who knew the example person and people who knew one or more individuals who knew the example person. The neighborhood thus had a "radius" of two people around the example person. This resulted in a total of 2,595,621 vertices and 2,808,128 edges for 1225 positive and 1225 negative examples. The Subdue-EC input consisted of a single graph containing 220,351 vertices and 317,243 edges with all of the examples. Subdue-EC was unable to get past the 17th iteration so was terminated

after 16 iterations and 61.5 hours. Subdue ran to completion and found 224 substructures using the set cover evaluation measure in 91.9 hours.

The reason Subdue completed but Subdue-EC did not is that Subdue never attempted an expansion containing three person nodes. This expansion step is what caused Subdue-EC to run out of space. To test the quality of these substructures they were used to classify the examples in graph1. The results of this classification are shown in Table 5.7.

Table 5.6 Disjoint Example Results

| | Subdue-EC | Subdue |
|---|---|---|
| TP | 670 | 1726 |
| TN | 28 | 75 |
| FP | 333 | 1657 |
| FN | 23 | 6 |
| Recall | 0.967 | 0.997 |
| Precisio | 0.668 | 0.510 |
| Sensitivity | 0.967 | 0.997 |
| Specificity | 0.078 | 0.043 |
| Error | 0.338 | 0.480 |

The error rate for Subdue-EC is lower, but the total number of examples classified is also lower. We believe the reason Subdue completed but Subdue-EC did

not is related to the substructures explored by Subdue. Subdue never attempted to expand to a three-person substructure, the point at which Subdue-EC floundered. All of the substructures discovered by Subdue were simple, one-person substructures with a single attribute. If the limit parameter had been higher, Subdue might also have encoutered the same problem as Subdue-EC.

## Comparing Subdue-EC to Feature-Vector Classification

We want to compare Subdue-EC to a feature-based classification method also. We can convert the SST data to a feature vector representation by once again excising a neighborhood around each example and then taking all of the attributes of each grid point and placing them in the vector. If we do this in a consistent order (such as column-wise from left to right), the results will make some sense with regard to the original grid. We utilize WEKA (Waikato Environment for Knowledge Acquisition) (Witten and Frank, 2005) to perform the analysis on the feature vector file. We chose a neighborhood containing one grid node in every direction for a total of eight nodes in

Table 5.7 Next Winter Classified by WEKA Decision Tree

| Classified as-< | Same | Decrease | Increase | Total |
|---|---|---|---|---|
| Same | 25447 | 1122 | 1059 | 27628 |
| Decrease | 1387 | 9810 | 2501 | 13698 |
| Increase | 1455 | 3899 | 18120 | 23474 |
| Total | 28289 | 14831 | 21680 | 53377 |
| | | | | 82.37% |

addition to the example node. This gives us eighteen features. We chose the decision tree option in WEKA since that learner supports missing data and nominal values.

Table 5.8 Next Winter Classified by Subdue-EC

| Classified as-< | Same | Decrease | Increase | Total |
|---|---|---|---|---|
| Same | 25473 | 1076 | 1079 | 27628 |
| Decrease | 1474 | 9737 | 2487 | 13698 |
| Increase | 1633 | 3925 | 17916 | 23474 |
| Total | 28580 | 14738 | 21482 | 53126 |
| | | | | 81.98% |

The accuracy of the two resulting classifiers was quite similar. Subdue-EC was slightly worse when tested on the SST data from the following winter. The first several structures are equivalent although WEKA often used the hemisphere from the node west of the target node. This is probably just a question of order in the feature vector since the hemisphere of the target node (used by Subdue-EC) is always the same as that of the nodes east and west of the target. Subdue-EC had no reason to expand east or west adding two additional vertices and edges to the classifying substructure to obtain the hemisphere. An advantage of the decision tree representation is that a decision node might be based on a node that is not adjacent to the target, whereas Subdue-EC must grow to that node. A disadvantage of the decision tree representation is the size of the classifying "device". The decision tree contains many leaves that have no instances. If these "empty" leaves are removed, then the 18 variable model had 150 useful leaves. We also constructed a smaller data set using only the nodes vertically and horizontally

adjacent to the target node. This model contained 78 useful leaves and finally a 3 variable model (based on the WEKA feature selector) that contained 31 on-zero leaves. None of these smaller models was as accurate as the Subdue-EC classification sequence with 26 substructures. The classifiers learned by both systems are listed Appendix C.

A subsequent run of Subdue-EC with different parameters produced a more accurate classification sequence and a larger number of substructures. By decreasing the beam size to one, the limit could be increased and still complete the run with the available resources. With a limit of sixty Subdue-EC produced 170 (rather than 26) classifying substructures which classified the winter 1991 SST data with an accuracy of 82.83%, slightly better than the WEKA decision tree.

To test our choice of neighborhood size, we used a smaller neighborhood to construct a feature vector. By taking only the four nodes horizontally and vertically adjacent and not using the northwest, northeast, southwest and southeast nodes, we constructed a feature vector with 10 attributes. The decision tree from this data, applied to the 1991 winter SST data, produced the results shown in Table 5.10. This smaller

Table 5.9 Next Winter Classified by WEKA with 10 Features

| Classified as-< | Same | Decrease | Increase | Total |
|---|---|---|---|---|
| Same | 25466 | 1116 | 1046 | 27628 |
| Decrease | 1422 | 9876 | 2400 | 13698 |
| Increase | 1522 | 3954 | 17998 | 23474 |
| Total | 28410 | 14946 | 21444 | 53340 |
| | | | | 82.31% |

vector had little effect on the decision tree but the smaller limit of 10 reduced both the accuracy and the size and number of classifying substructures learned by Subdue-EC.

The Limit Parameter

In the process of performing experiments with Subdue-EC, it became apparent that the limit parameter is the single most important factor in determining both run time and the quantity of classifying substructures. It does not, however, have a large effect on the quality of the substructures discovered. We performed a series of experiments to see how this parameter changed results on the SST data. Figure 5.6 shows run time, number of substructures, number of training examples classified and classification compression as the limit parameter was increased. Note that accuracy gradually increases as limit increases, though not by a lot.

Increasing limit allows larger, more precise substructures to be discovered, but since these structures are larger, they generally classify fewer examples. Figure 5.8 shows the cumulative number of examples classified by the 71 substructures learned by a limit 20 run and the 137 substructures learned by a limit 80 run. Although both of these runs achieve nearly the same accuracy, that is not always the case. If a particular substrstructure misclassifies any examples, it is possible that just one more vertex or edge might eliminate some or all of those misclassifications. So a larger substructure may misclassify fewer examples, but it will never classify more examples. Thus growing a classifying sequence composed of larger substructures may provide more accurate classification, but the sequence is likely to be longer in terms of both number and size of subsstructures

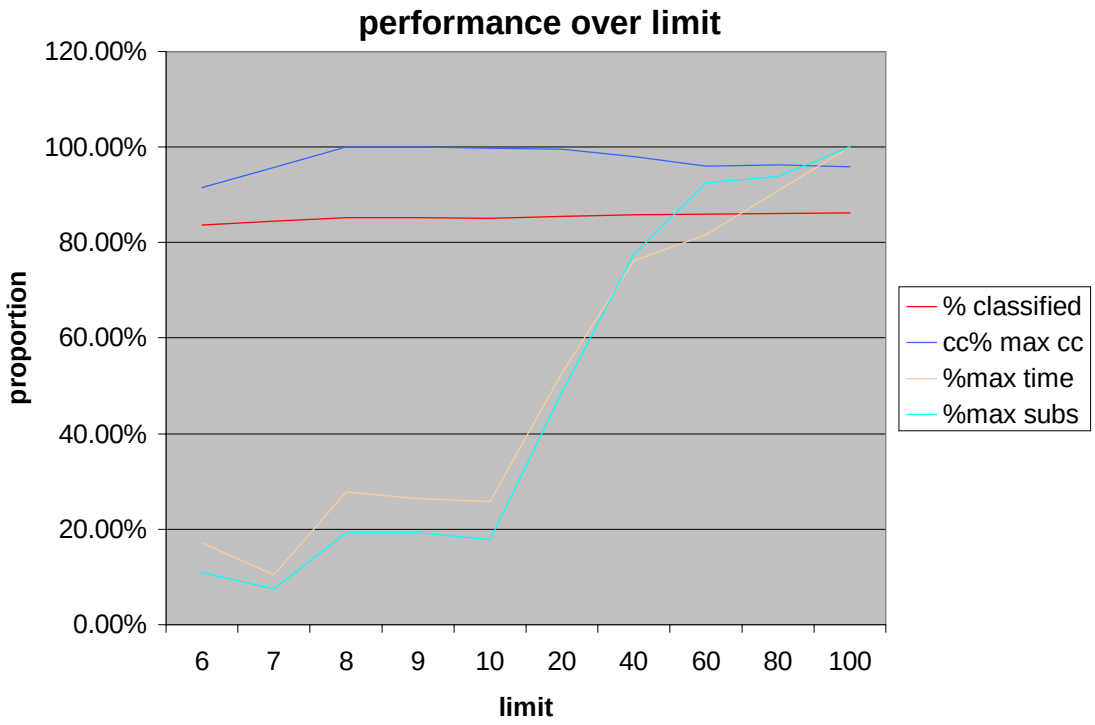



Figure 5.6 Performance as limit increases

## The Beam Parameter

Since Subdue expands substructures incrementally it is always possible that the substructure with the best value will not remain the best value in the next round of expansion. The beam width parameter specifies the number of substructures to keep for the next round of expansion. This same concept was used in Subdue-EC and is also used to allow more than one classification sequence to be evaluated in each iteration (the "seqBeam" parameter). However, using this type of search can increase run time substantially. Encouraged by the greedy approach used in learning decision trees, we have run Subdue-EC with a beam of width one and found that accuracy does not suffer

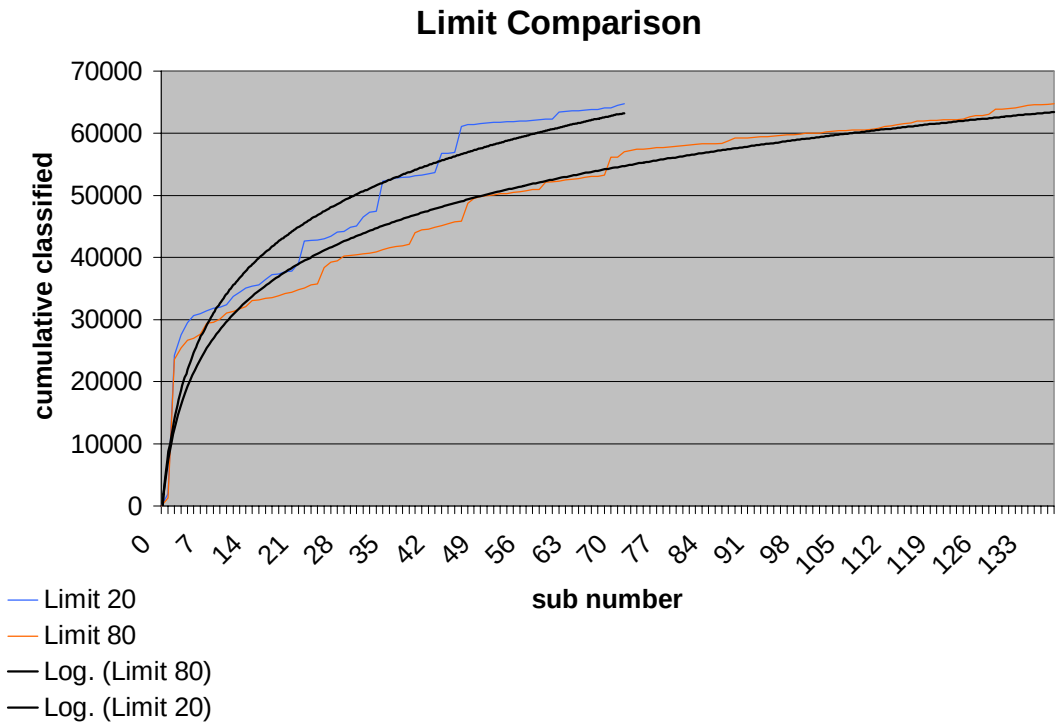



Figure 5.7 Cumulative examples classified for limit 20 and 80

on the SST data. Table 5.11 shows runs with beam widths of one and two. A beam width of one is essentially hill climbing and can prune strong candidate substructures. Some of these, however, can be recaptured in subsequent iterations. The limit parameter also has to be considered along with the beam parameter since a wider beam with the same limit will probably result in less expansion of each substructure being considered. Or, put another way, if one doubles the beam size, one must approximately double the limit in order to expand substructures to the same point.

Table 5.11 Effect of Beam Width

| beam width 1 limit | subs | secs | accuracy on training data |
|---|---|---|---|
| 10 | 26 | 27 | 84.93% |
| 20 | 141 | 185 | 86.37% |
| 40 | 169 | 568 | 86.59% |
| 60 | 170 | 1658 | 86.68% |
| 80 | 170 | 4960 | 86.68% |
| 100 | 170 | 12736 | 86.68% |
| beam width 2 | | | |
| 80 | 160 | 1077 | 86.35% |
| 100 | 162 | 1967 | 86.46% |
| 140 | 164 | 6896 | 86.40% |
| 160 | 172 | 11369 | 86.53% |
| 200 | 166 | 31237 | 86.54% |

## Noise Sensitivity

To test the sensitivity of the algorithm to noise we randomly substituted lower case letters for a certain percentage of the vertex labels “A”, “B”, “C” and “D” in the testclsx graph. We did not alter any edge labels or class vertices. We started out with 10% of the vertex labels corrupted. We kept these same corrupt vertex labels for the second test set but corrupted another 10%. This ensures that changes from one set to the next are due to the new corruption. This test was run once with HPB for the evaluation method and once with classification compression as the evaluation method. Each learned sequence was applied to the uncorrupted graph to determine the effect of the noise on the learned classifying sequence. The HPB evaluation criteria performed much better than the Classification Compression criteria. The CC criteria was extremely

sensitive to the noise. At a ten percent noise level, less than twenty percent of the examples were classified correctly. At noise levels of 30% and higher, there were no examples correctly classified. This result was quite unexpected and additional experiments with other data might lead to a better understanding of this rapidly deteriorating performance.

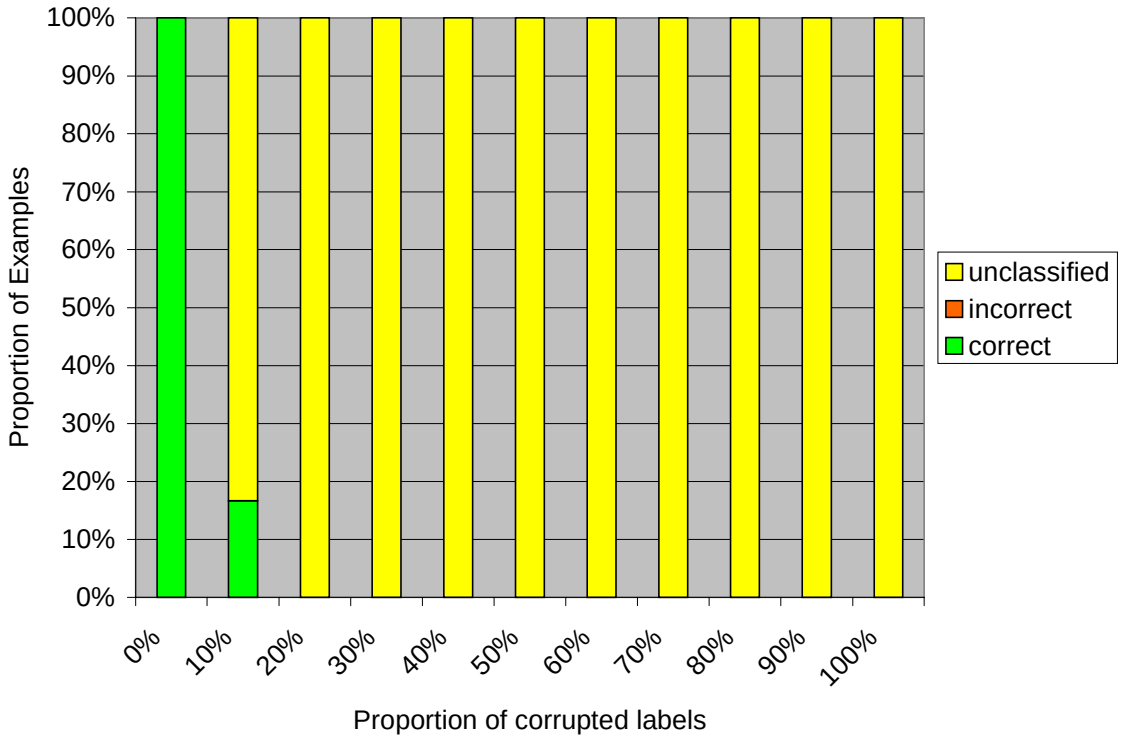



Figure 5.8 Noise sensitivity with CC

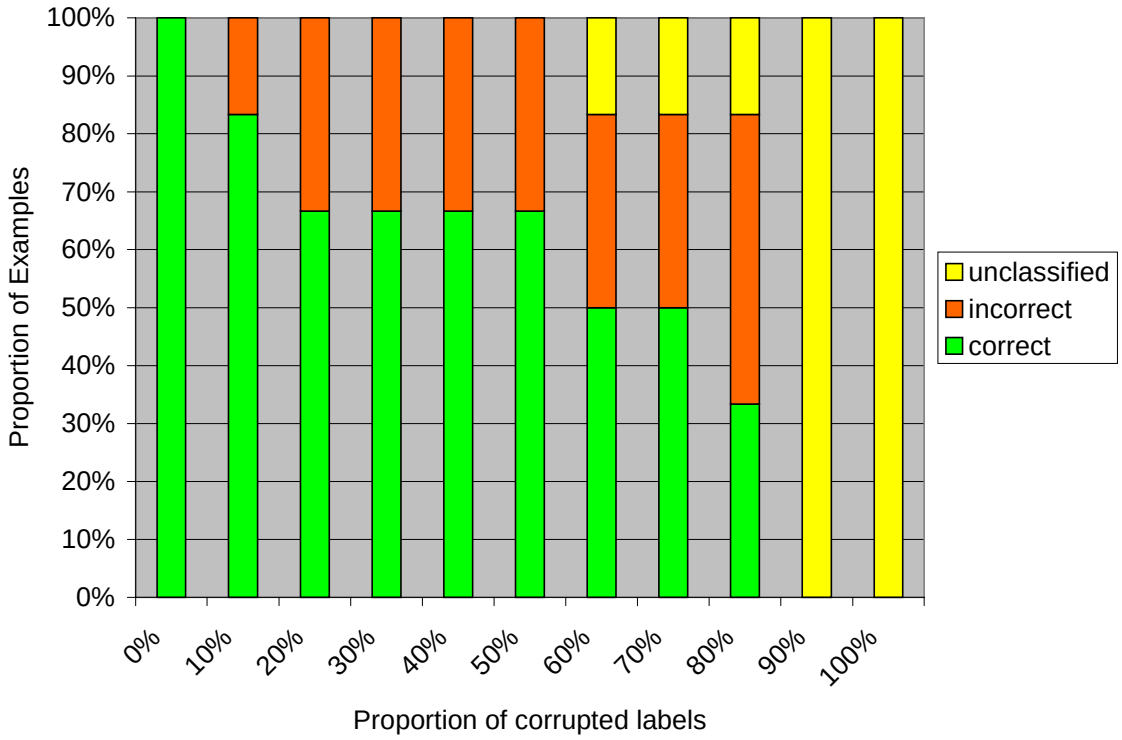



Figure 5.9 Noise sensitivity with HPB

## Error Sensitivity

We used the same graph to test for sensitivity to errors in the examples. This time we randomly selected an example marker on which to invert the class. Since this data contains duplicate examples, this caused conflicting class information for one or more identical examples. Again, the HPB measure tolerated the errors better than the Classification Compression measure. With HPB the performance fell off gradually as it did with added noise, but with this data having class errors on the training data the performance degradation was evident in misclassified examples rather than unclassified

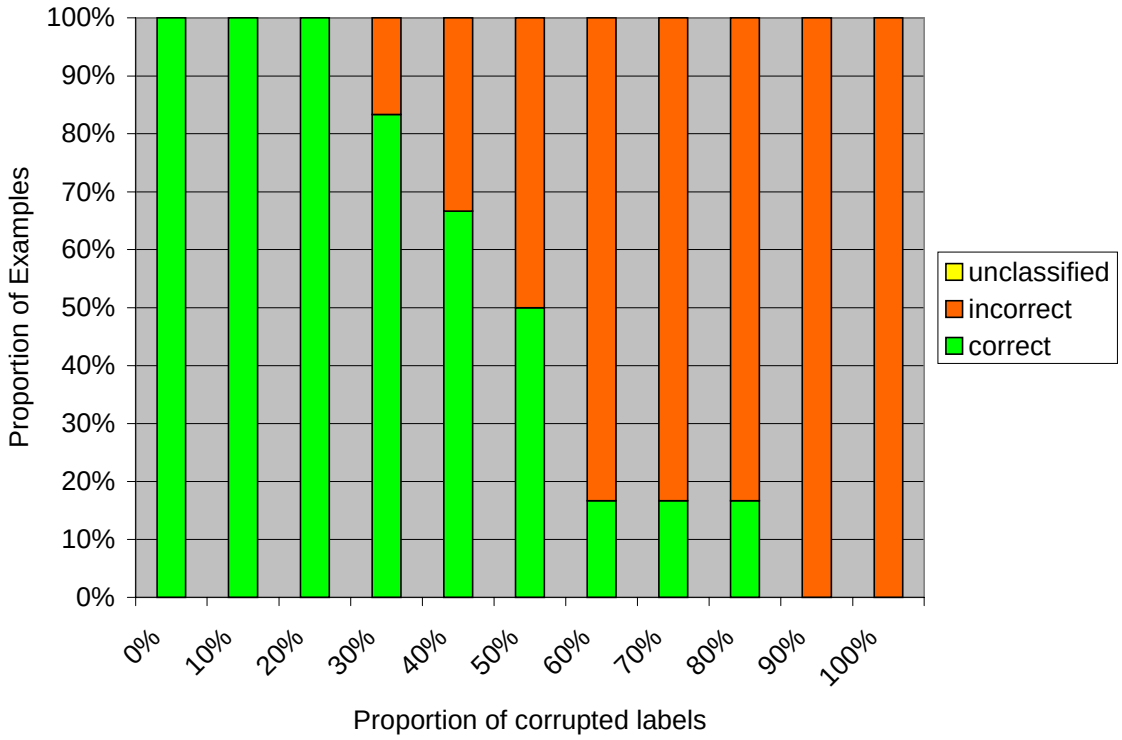



Figure 5.10 Error sensitivity with CC

ones. The Classification Compression performed slightly less well with error. It also showed increasing misclassifications as the error level increased.

The classification performance of the algorithm with both noise and errors in the data was encouraging, but further study is needed to draw any conclusions about the general robustness of the algorithm under these circumstances. Although the flexibility of the representation lends itself to these types of experiments, the run time is prohibitive for large graphs.

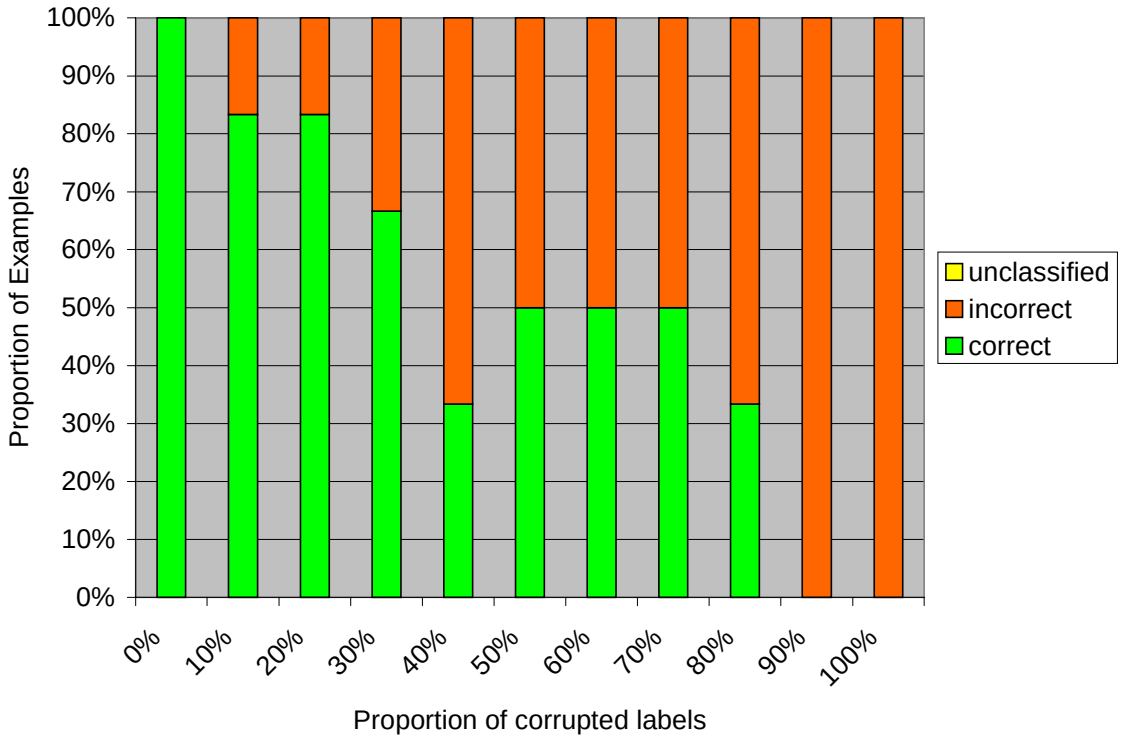



Figure 5.11 Error sensitivity with HPB

# CHAPTER 6

# CONCLUSIONS

Our representation for training examples embedded in a single connected graph is a useful one for learning. In addition, the representation is easily manipulated to create test data for *n*-fold testing and other purposes. A classification sequence is a concise and effective representation for classification of examples embedded in a graph. Our approach can be used for concept learning and for *n*-class classification.

Subdue-EC successfully learns rules for classification from training examples in a single graph. The algorithm requires less run time than previous versions of Subdue because it makes fewer calls to the graph match routines. The algorithm utilizes classification compression to guide its search for classification sequences. This measure, based on the Minimum Description Length Principle, is useful in finding an ordered sequence of substructures which can be used to classify previously unseen areas of interest in novel graphs.

## Future Work

Subdue-EC, with its efficient data representations and improved run-times over previous versions of Subdue, provides a good starting point for further exploration of learning classification sequences from embedded examples. Additional evaluation criteria as well as additional improvements in run-time could be fruitful areas for investigation. We believe that "hits per bit" should be explored further as a measure that can guide the search for a classification sequence. It could easily accommodate a

weight for each class for domains where classification errors of some classes might be more or less costly than other classes. It could also support a weight factor (or cost) for unclassified examples for domains where it might be better to leave examples unclassified than to misclassify them. Using information gain might be another useful evaluation measure.

Another avenue to explore would be the use of frequently-occurring subgraph algorithms to provide seeds of subgraphs to expand for classification purposes. This might save valuable time by quickly finding candidate subgraphs for further expansion to "specialize" them into classifying substructures.

Finally, consideration should be given to a different search strategy. Perhaps reconstructing instances as needed when it is time for expansion rather than storing them would reduce the heavy memory usage for large data sets.

We have developed a machine learning algorithm which learns rules for classification from training examples in a graph representation. However, unlike most other such algorithms which use one graph for each example, ours allows all of the training examples to be in a single, connected graph. We applied the Minimum Description Length principle to produce a novel performance metric for judging the value of a learned classification. We implemented the algorithm by extending the Subdue graph-based learning system. Finally, we demonstrated the use of the new system in two different domains, earth science and homeland security.

# APPENDIX A

# SUBDUE-EC SUBSTRUCTURE EXPANSION ALGORITHM

Subdue-EC uses a different expansion algorithm than previous versions of Subdue. The new algorithm substantially reduces the number of times that one must check for subgraph isomorphism. The disadvantage of the algorithm is that it does not currently support inexact matching of graphs. The key feature of the new algorithm is that the instances are expanded in such a way that all of the instances of a substructure are always isomorphic. With the previous algorithm, both subgraph definitions and instances generated had to be checked for isomorphism. The new algorithm produces equivalent results in general, but there may be a slight difference in output due to the order in which the instances are stored. This order effects the instance chosen to discard when instances are duplicated or overlapped. The pseudo-code for expanding the instances of a substructure is:

```
For each vertex of the substructure definition graph
  For each classification class
    For each instance of the substructure
      For each unused edge of this instance
        If the edge connects to a vertex already included
          Construct new instance by adding this edge
        Else
          Construct new instance by adding edge and vertex
      Add this instance to the substructure that was created by the addition of this
      same edge or edge-vertex.
      If none exists create a new Substructure with this instance
```

Else

Add this new instance to the Substructure

When the ith vertex of all instances for all classes has been expanded by all of its unused edges, then the values of all substructures are calculated and they are inserted into the childlist subject to the beam width. This is the first time it may be necessary to perform subgraph isomorphism. That is done in order to determine if a suitably valued substructure is already present, in which case the one with the higher value is used in the child list.

# APPENDIX B

# COMMAND LINE PARAMETERS FOR SUBDUE-EC

Subdue-EC is based on version 5.06 of Subdue. It supports all of the command line parameters supported by that version. It does not use the new, faster expansion algorithm if inexact graph match parameters are specified and it does not support inexact graph match for embedded classification. It still is capable of performing concept learning with "XP" and "XN" indicators in the input graph and in addition can perform this type of classification (one example per graph) with multiple classes (k>2). The following parameters, used by version 5.06 have been extended:

-eval added 5 additional options all related to embedded classification

4 use hits per bit

5 use classification compression without counting unclassified

6 use classification compression counting unclassified

7 use classification compression with number of hits for tiebreaker

8 use hits per bit selecting based on sub value

The following parameters are new to Subdue-EC:

-byList and –nullNode These two parameter now have no effect. They were used during development in the old extend routine

--treeType 0 use list for extending substructures, 1 use binary tree, 2 use red/black tree; For small graphs 0 is ok. RB tree requires more space but is faster if there are a large number of instances for each substructure.

-labelList print all labels after reading input graph

-cs use this filename to read a classification sequence and classify a novel graph

-seqEval 1 use HPB or 2 use classification compression for evaluating classification sequence

-preferShorter used in old style classification to force acceptance of shorter classifying substructure if tie

-classify specify classification method 0 all classes, or n only class number n

-compress specify compression 0 remove any classified class. n remove only members classified as n

-neverAdd specify label of class marker (also set by the presence of a Class vertex in the graph input file)

-growFrom specify label of class marker (for initial classifying substructure) (also set by the presence of a Class vertex in the graph input file)

-seqBest number of best sequences to keep

-egNeighborhoodType special value to excise a neighborhood around an example. Hardcoded for SST, EAGLE or circabcd

-removeEdges special to force removal of extra edges in example neighborhood

# APPENDIX C

## SUBDUE-EC CLASSIFYING SEQUENCE AND WEKA DECISION TREE FOR SST DATA

Subdue-EC learned the following classifying sequence from the SST data.

```
% classifying subGraph 1
c 1 SAME
v 2 JAN
v 3 32766
d 1 2 DeltaNextMonth
d 2 3 TEMP

% classifying subGraph 2
c 1 INCREASE
v 2 JAN
v 3 Southern
v 4 1
d 1 2 DeltaNextMonth
d 2 3 HEMI
d 2 4 TEMP

% classifying subGraph 3
c 1 INCREASE
v 2 JAN
v 3 Southern
v 4 6
d 1 2 DeltaNextMonth
d 2 3 HEMI
d 2 4 TEMP

% classifying subGraph 4
c 1 INCREASE
v 2 JAN
v 3 Southern
v 4 5
d 1 2 DeltaNextMonth
d 2 3 HEMI
d 2 4 TEMP

% classifying subGraph 5
c 1 DECREASE
v 2 JAN
v 3 5
d 1 2 DeltaNextMonth
d 2 3 TEMP
```

```
% classifying subGraph 6
c 1 DECREASE
v 2 JAN
v 3 8
v 4 Northern
d 1 2 DeltaNextMonth
d 2 3 TEMP
d 2 4 HEMI

% classifying subGraph 7
c 1 DECREASE
v 2 JAN
v 3 3
v 4 Northern
d 1 2 DeltaNextMonth
d 2 3 TEMP
d 2 4 HEMI

% classifying subGraph 8
c 1 INCREASE
v 2 JAN
v 3 3
d 1 2 DeltaNextMonth
d 2 3 TEMP

% classifying subGraph 9
c 1 DECREASE
v 2 JAN
v 3 4
v 4 Northern
d 1 2 DeltaNextMonth
d 2 3 TEMP
d 2 4 HEMI

% classifying subGraph 10
c 1 INCREASE
v 2 JAN
v 3 4
d 1 2 DeltaNextMonth
d 2 3 TEMP

% classifying subGraph 11
c 1 DECREASE
```

```
v 2 JAN
v 3 1
d 1 2 DeltaNextMonth
d 2 3 TEMP

% classifying subGraph 12
c 1 DECREASE
v 2 JAN
v 3 2
v 4 Northern
d 1 2 DeltaNextMonth
d 2 3 TEMP
d 2 4 HEMI

% classifying subGraph 13
c 1 INCREASE
v 2 JAN
v 3 Southern
v 4 0
d 1 2 DeltaNextMonth
d 2 3 HEMI
d 2 4 TEMP

% classifying subGraph 14
c 1 DECREASE
v 2 JAN
v 3 6
v 4 JAN
v 5 Northern
d 1 2 DeltaNextMonth
d 2 3 TEMP
d 4 2 N
d 4 5 HEMI

% classifying subGraph 15
c 1 INCREASE
v 2 JAN
v 3 Southern
v 4 7
d 1 2 DeltaNextMonth
d 2 3 HEMI
d 2 4 TEMP

% classifying subGraph 16
```

```
c 1 INCREASE
v 2 JAN
v 3 2
d 1 2 DeltaNextMonth
d 2 3 TEMP

% classifying subGraph 17
c 1 SAME
v 2 JAN
v 3 0
d 1 2 DeltaNextMonth
d 2 3 TEMP

% classifying subGraph 18
c 1 INCREASE
v 2 JAN
v 3 6
d 1 2 DeltaNextMonth
d 2 3 TEMP

% classifying subGraph 19
c 1 DECREASE
v 2 JAN
v 3 JAN
v 4 6
d 1 2 DeltaNextMonth
d 2 3 N
d 3 4 TEMP

% classifying subGraph 20
c 1 DECREASE
v 2 JAN
v 3 JAN
v 4 Northern
v 5 8
d 1 2 DeltaNextMonth
d 2 3 N
d 3 4 HEMI
d 2 5 TEMP

% classifying subGraph 21
c 1 INCREASE
v 2 JAN
v 3 JAN
```

```
v 4 6
d 1 2 DeltaNextMonth
d 3 2 N
d 3 4 TEMP

% classifying subGraph 22
c 1 DECREASE
v 2 JAN
v 3 JAN
v 4 Northern
d 1 2 DeltaNextMonth
d 3 2 N
d 3 4 HEMI

% classifying subGraph 23
c 1 INCREASE
v 2 JAN
v 3 Northern
d 1 2 DeltaNextMonth
d 2 3 HEMI

% classifying subGraph 24
c 1 INCREASE
v 2 JAN
v 3 JAN
v 4 7
d 1 2 DeltaNextMonth
d 3 2 N
d 3 4 TEMP

% classifying subGraph 25
c 1 DECREASE
v 2 JAN
v 3 JAN
v 4 8
d 1 2 DeltaNextMonth
d 2 3 N
d 3 4 TEMP

% classifying subGraph 26
c 1 INCREASE
v 2 JAN
d 1 2 DeltaNextMonth
```

WEKA test output from 1991 SST data.

```
=== Model information ===

Filename:     neig90w5.model
Scheme:       trees.J48 -C 0.25 -M 2
Relation:     neighd.ARFF-
weka.filters.unsupervised.attribute.Remove-R1-2,5-6,13-
14,17-18
Attributes:   11
              wHemisphere
              wTempBin
              nHemisphere
              nTempBin
              targetHemisphere
              targetTempBin
              sHemisphere
              sTempBin
              eHemisphere
              eTempBin
              class

=== Classifier model ===

J48 pruned tree
------------------

targetTempBin = 0
|   wHemisphere = Southern
|   |   sTempBin = 0: INCREASE (3845.0/715.0)
|   |   sTempBin = 1: INCREASE (23.0)
|   |   sTempBin = 2: INCREASE (0.0)
|   |   sTempBin = 3: INCREASE (0.0)
|   |   sTempBin = 4: INCREASE (0.0)
|   |   sTempBin = 5: INCREASE (0.0)
|   |   sTempBin = 6: INCREASE (0.0)
|   |   sTempBin = 7: INCREASE (0.0)
|   |   sTempBin = 8: INCREASE (0.0)
|   |   sTempBin = 32766
|   |   |   wTempBin = 0: DECREASE (408.0/217.0)
|   |   |   wTempBin = 1: DECREASE (0.0)
```

```
|   |   |   wTempBin = 2: DECREASE (0.0)
|   |   |   wTempBin = 3: DECREASE (0.0)
|   |   |   wTempBin = 4: DECREASE (0.0)
|   |   |   wTempBin = 5: DECREASE (0.0)
|   |   |   wTempBin = 6: DECREASE (0.0)
|   |   |   wTempBin = 7: DECREASE (0.0)
|   |   |   wTempBin = 8: DECREASE (0.0)
|   |   |   wTempBin = 32766
|   |   |   |   eTempBin = 0: INCREASE (51.0/24.0)
|   |   |   |   eTempBin = 1: INCREASE (0.0)
|   |   |   |   eTempBin = 2: INCREASE (0.0)
|   |   |   |   eTempBin = 3: INCREASE (0.0)
|   |   |   |   eTempBin = 4: INCREASE (0.0)
|   |   |   |   eTempBin = 5: INCREASE (0.0)
|   |   |   |   eTempBin = 6: INCREASE (0.0)
|   |   |   |   eTempBin = 7: INCREASE (0.0)
|   |   |   |   eTempBin = 8: INCREASE (0.0)
|   |   |   |   eTempBin = 32766: DECREASE (10.0/5.0)
|   wHemisphere = Northern
|   |   sTempBin = 0
|   |   |   wTempBin = 0
|   |   |   |   eTempBin = 0
|   |   |   |   |   nTempBin = 0: SAME (4697.0/1209.0)
|   |   |   |   |   nTempBin = 1: DECREASE (9.0/1.0)
|   |   |   |   |   nTempBin = 2: SAME (0.0)
|   |   |   |   |   nTempBin = 3: SAME (0.0)
|   |   |   |   |   nTempBin = 4: SAME (0.0)
|   |   |   |   |   nTempBin = 5: SAME (0.0)
|   |   |   |   |   nTempBin = 6: SAME (0.0)
|   |   |   |   |   nTempBin = 7: SAME (0.0)
|   |   |   |   |   nTempBin = 8: SAME (0.0)
|   |   |   |   |   nTempBin = 32766: SAME (514.0/82.0)
|   |   |   |   eTempBin = 1: DECREASE (10.0/1.0)
|   |   |   |   eTempBin = 2: SAME (0.0)
|   |   |   |   eTempBin = 3: SAME (0.0)
|   |   |   |   eTempBin = 4: SAME (0.0)
|   |   |   |   eTempBin = 5: SAME (0.0)
|   |   |   |   eTempBin = 6: SAME (0.0)
|   |   |   |   eTempBin = 7: SAME (0.0)
|   |   |   |   eTempBin = 8: SAME (0.0)
|   |   |   |   eTempBin = 32766: SAME (101.0/44.0)
|   |   |   wTempBin = 1
|   |   |   |   eTempBin = 0: DECREASE (19.0/4.0)
|   |   |   |   eTempBin = 1: DECREASE (0.0)
```

```
|   |   |   |   eTempBin = 2: DECREASE (0.0)
|   |   |   |   eTempBin = 3: DECREASE (0.0)
|   |   |   |   eTempBin = 4: DECREASE (0.0)
|   |   |   |   eTempBin = 5: DECREASE (0.0)
|   |   |   |   eTempBin = 6: DECREASE (0.0)
|   |   |   |   eTempBin = 7: DECREASE (0.0)
|   |   |   |   eTempBin = 8: DECREASE (0.0)
|   |   |   |   eTempBin = 32766: INCREASE (2.0)
|   |   |   wTempBin = 2: INCREASE (1.0)
|   |   |   wTempBin = 3: SAME (0.0)
|   |   |   wTempBin = 4: SAME (0.0)
|   |   |   wTempBin = 5: SAME (0.0)
|   |   |   wTempBin = 6: SAME (0.0)
|   |   |   wTempBin = 7: SAME (0.0)
|   |   |   wTempBin = 8: SAME (0.0)
|   |   |   wTempBin = 32766: SAME (119.0/51.0)
|   |   sTempBin = 1
|   |   |   wTempBin = 0
|   |   |   |   eTempBin = 0: INCREASE (89.0/35.0)
|   |   |   |   eTempBin = 1: DECREASE (15.0/6.0)
|   |   |   |   eTempBin = 2: INCREASE (0.0)
|   |   |   |   eTempBin = 3: INCREASE (0.0)
|   |   |   |   eTempBin = 4: INCREASE (0.0)
|   |   |   |   eTempBin = 5: INCREASE (0.0)
|   |   |   |   eTempBin = 6: INCREASE (0.0)
|   |   |   |   eTempBin = 7: INCREASE (0.0)
|   |   |   |   eTempBin = 8: INCREASE (0.0)
|   |   |   |   eTempBin = 32766
|   |   |   |   |   nTempBin = 0: DECREASE (4.0/1.0)
|   |   |   |   |   nTempBin = 1: DECREASE (0.0)
|   |   |   |   |   nTempBin = 2: DECREASE (0.0)
|   |   |   |   |   nTempBin = 3: DECREASE (0.0)
|   |   |   |   |   nTempBin = 4: DECREASE (0.0)
|   |   |   |   |   nTempBin = 5: DECREASE (0.0)
|   |   |   |   |   nTempBin = 6: DECREASE (0.0)
|   |   |   |   |   nTempBin = 7: DECREASE (0.0)
|   |   |   |   |   nTempBin = 8: DECREASE (0.0)
|   |   |   |   |   nTempBin = 32766: INCREASE (3.0/1.0)
|   |   |   wTempBin = 1
|   |   |   |   nTempBin = 0: DECREASE (21.0/10.0)
|   |   |   |   nTempBin = 1: INCREASE (0.0)
|   |   |   |   nTempBin = 2: INCREASE (0.0)
|   |   |   |   nTempBin = 3: INCREASE (0.0)
|   |   |   |   nTempBin = 4: INCREASE (0.0)
```

```
|   |   |   |   nTempBin = 5: INCREASE (0.0)
|   |   |   |   nTempBin = 6: INCREASE (0.0)
|   |   |   |   nTempBin = 7: INCREASE (0.0)
|   |   |   |   nTempBin = 8: INCREASE (0.0)
|   |   |   |   nTempBin = 32766: INCREASE (2.0)
|   |   |   wTempBin = 2: INCREASE (0.0)
|   |   |   wTempBin = 3: INCREASE (1.0)
|   |   |   wTempBin = 4: INCREASE (0.0)
|   |   |   wTempBin = 5: INCREASE (0.0)
|   |   |   wTempBin = 6: INCREASE (0.0)
|   |   |   wTempBin = 7: INCREASE (0.0)
|   |   |   wTempBin = 8: INCREASE (0.0)
|   |   |   wTempBin = 32766: DECREASE (4.0)
|   |   sTempBin = 2
|   |   |   eTempBin = 0: DECREASE (3.0/1.0)
|   |   |   eTempBin = 1: INCREASE (2.0)
|   |   |   eTempBin = 2: INCREASE (0.0)
|   |   |   eTempBin = 3: INCREASE (0.0)
|   |   |   eTempBin = 4: INCREASE (0.0)
|   |   |   eTempBin = 5: INCREASE (0.0)
|   |   |   eTempBin = 6: INCREASE (0.0)
|   |   |   eTempBin = 7: INCREASE (0.0)
|   |   |   eTempBin = 8: INCREASE (0.0)
|   |   |   eTempBin = 32766: INCREASE (0.0)
|   |   sTempBin = 3: SAME (0.0)
|   |   sTempBin = 4: SAME (0.0)
|   |   sTempBin = 5: SAME (0.0)
|   |   sTempBin = 6: SAME (0.0)
|   |   sTempBin = 7: SAME (0.0)
|   |   sTempBin = 8: SAME (0.0)
|   |   sTempBin = 32766: SAME (677.0/155.0)
targetTempBin = 1
|   wHemisphere = Southern: INCREASE (2363.0/216.0)
|   wHemisphere = Northern
|   |   nTempBin = 0
|   |   |   sTempBin = 0: DECREASE (0.0)
|   |   |   sTempBin = 1: DECREASE (114.0/45.0)
|   |   |   sTempBin = 2: INCREASE (18.0/7.0)
|   |   |   sTempBin = 3: DECREASE (0.0)
|   |   |   sTempBin = 4: DECREASE (0.0)
|   |   |   sTempBin = 5: DECREASE (0.0)
|   |   |   sTempBin = 6: DECREASE (0.0)
|   |   |   sTempBin = 7: DECREASE (0.0)
|   |   |   sTempBin = 8: DECREASE (0.0)
```

```
|   |   |   sTempBin = 32766: INCREASE (7.0/1.0)
|   |   nTempBin = 1: DECREASE (917.0/176.0)
|   |   nTempBin = 2
|   |   |   eTempBin = 0: SAME (1.0)
|   |   |   eTempBin = 1: DECREASE (6.0/1.0)
|   |   |   eTempBin = 2: INCREASE (2.0)
|   |   |   eTempBin = 3: DECREASE (0.0)
|   |   |   eTempBin = 4: DECREASE (0.0)
|   |   |   eTempBin = 5: DECREASE (0.0)
|   |   |   eTempBin = 6: DECREASE (0.0)
|   |   |   eTempBin = 7: DECREASE (0.0)
|   |   |   eTempBin = 8: DECREASE (0.0)
|   |   |   eTempBin = 32766: DECREASE (0.0)
|   |   nTempBin = 3: DECREASE (0.0)
|   |   nTempBin = 4: DECREASE (0.0)
|   |   nTempBin = 5: DECREASE (0.0)
|   |   nTempBin = 6: DECREASE (0.0)
|   |   nTempBin = 7: DECREASE (0.0)
|   |   nTempBin = 8: DECREASE (0.0)
|   |   nTempBin = 32766: DECREASE (120.0/22.0)
targetTempBin = 2
|   wHemisphere = Southern: INCREASE (1925.0/523.0)
|   wHemisphere = Northern: DECREASE (1181.0/234.0)
targetTempBin = 3
|   wHemisphere = Southern: INCREASE (1670.0/320.0)
|   wHemisphere = Northern
|   |   sTempBin = 0: DECREASE (0.0)
|   |   sTempBin = 1: DECREASE (0.0)
|   |   sTempBin = 2: DECREASE (3.0)
|   |   sTempBin = 3: DECREASE (674.0/58.0)
|   |   sTempBin = 4: DECREASE (163.0/32.0)
|   |   sTempBin = 5: INCREASE (8.0/1.0)
|   |   sTempBin = 6: DECREASE (0.0)
|   |   sTempBin = 7: DECREASE (0.0)
|   |   sTempBin = 8: DECREASE (0.0)
|   |   sTempBin = 32766: DECREASE (41.0/2.0)
targetTempBin = 4
|   wHemisphere = Southern: INCREASE (1599.0/306.0)
|   wHemisphere = Northern: DECREASE (979.0/143.0)
targetTempBin = 5
|   wHemisphere = Southern: INCREASE (2089.0/284.0)
|   wHemisphere = Northern: DECREASE (1467.0/151.0)
targetTempBin = 6
|   sHemisphere = Southern: INCREASE (3603.0/348.0)
```

```
|   sHemisphere = Northern
|   |   nTempBin = 0: DECREASE (0.0)
|   |   nTempBin = 1: DECREASE (0.0)
|   |   nTempBin = 2: DECREASE (0.0)
|   |   nTempBin = 3: DECREASE (0.0)
|   |   nTempBin = 4: DECREASE (0.0)
|   |   nTempBin = 5: DECREASE (203.0/43.0)
|   |   nTempBin = 6: DECREASE (1642.0/352.0)
|   |   nTempBin = 7: INCREASE (44.0)
|   |   nTempBin = 8: DECREASE (0.0)
|   |   nTempBin = 32766: DECREASE (65.0/18.0)
targetTempBin = 7
|   sHemisphere = Southern: INCREASE (4015.0/1075.0)
|   sHemisphere = Northern
|   |   sTempBin = 0: DECREASE (0.0)
|   |   sTempBin = 1: DECREASE (0.0)
|   |   sTempBin = 2: DECREASE (0.0)
|   |   sTempBin = 3: DECREASE (0.0)
|   |   sTempBin = 4: DECREASE (0.0)
|   |   sTempBin = 5: DECREASE (0.0)
|   |   sTempBin = 6: INCREASE (45.0)
|   |   sTempBin = 7
|   |   |   nTempBin = 0: DECREASE (0.0)
|   |   |   nTempBin = 1: DECREASE (0.0)
|   |   |   nTempBin = 2: DECREASE (0.0)
|   |   |   nTempBin = 3: DECREASE (0.0)
|   |   |   nTempBin = 4: DECREASE (0.0)
|   |   |   nTempBin = 5: DECREASE (0.0)
|   |   |   nTempBin = 6: DECREASE (223.0/48.0)
|   |   |   nTempBin = 7: DECREASE (3973.0/1432.0)
|   |   |   nTempBin = 8
|   |   |   |   eTempBin = 0: INCREASE (0.0)
|   |   |   |   eTempBin = 1: INCREASE (0.0)
|   |   |   |   eTempBin = 2: INCREASE (0.0)
|   |   |   |   eTempBin = 3: INCREASE (0.0)
|   |   |   |   eTempBin = 4: INCREASE (0.0)
|   |   |   |   eTempBin = 5: INCREASE (0.0)
|   |   |   |   eTempBin = 6: INCREASE (0.0)
|   |   |   |   eTempBin = 7: INCREASE (22.0/11.0)
|   |   |   |   eTempBin = 8: SAME (8.0/4.0)
|   |   |   |   eTempBin = 32766: INCREASE (0.0)
|   |   |   nTempBin = 32766: INCREASE (148.0/67.0)
|   |   sTempBin = 8: DECREASE (107.0/33.0)
|   |   sTempBin = 32766: DECREASE (122.0/45.0)
```

```
targetTempBin = 8
|   nHemisphere = Southern
|   |   sTempBin = 0: DECREASE (0.0)
|   |   sTempBin = 1: DECREASE (0.0)
|   |   sTempBin = 2: DECREASE (0.0)
|   |   sTempBin = 3: DECREASE (0.0)
|   |   sTempBin = 4: DECREASE (0.0)
|   |   sTempBin = 5: DECREASE (0.0)
|   |   sTempBin = 6: DECREASE (0.0)
|   |   sTempBin = 7: INCREASE (161.0/65.0)
|   |   sTempBin = 8
|   |   |   eTempBin = 0: DECREASE (0.0)
|   |   |   eTempBin = 1: DECREASE (0.0)
|   |   |   eTempBin = 2: DECREASE (0.0)
|   |   |   eTempBin = 3: DECREASE (0.0)
|   |   |   eTempBin = 4: DECREASE (0.0)
|   |   |   eTempBin = 5: DECREASE (0.0)
|   |   |   eTempBin = 6: DECREASE (0.0)
|   |   |   eTempBin = 7: INCREASE (25.0/7.0)
|   |   |   eTempBin = 8
|   |   |   |   wTempBin = 0: DECREASE (0.0)
|   |   |   |   wTempBin = 1: DECREASE (0.0)
|   |   |   |   wTempBin = 2: DECREASE (0.0)
|   |   |   |   wTempBin = 3: DECREASE (0.0)
|   |   |   |   wTempBin = 4: DECREASE (0.0)
|   |   |   |   wTempBin = 5: DECREASE (0.0)
|   |   |   |   wTempBin = 6: DECREASE (0.0)
|   |   |   |   wTempBin = 7
|   |   |   |   |   nTempBin = 0: DECREASE (0.0)
|   |   |   |   |   nTempBin = 1: DECREASE (0.0)
|   |   |   |   |   nTempBin = 2: DECREASE (0.0)
|   |   |   |   |   nTempBin = 3: DECREASE (0.0)
|   |   |   |   |   nTempBin = 4: DECREASE (0.0)
|   |   |   |   |   nTempBin = 5: DECREASE (0.0)
|   |   |   |   |   nTempBin = 6: DECREASE (0.0)
|   |   |   |   |   nTempBin = 7: DECREASE (16.0/6.0)
|   |   |   |   |   nTempBin = 8: INCREASE (6.0/2.0)
|   |   |   |   |   nTempBin = 32766: INCREASE (1.0)
|   |   |   |   wTempBin = 8
|   |   |   |   |   nTempBin = 0: DECREASE (0.0)
|   |   |   |   |   nTempBin = 1: DECREASE (0.0)
|   |   |   |   |   nTempBin = 2: DECREASE (0.0)
|   |   |   |   |   nTempBin = 3: DECREASE (0.0)
|   |   |   |   |   nTempBin = 4: DECREASE (0.0)
```

```
|   |   |   |   |   nTempBin = 5: DECREASE (0.0)
|   |   |   |   |   nTempBin = 6: DECREASE (0.0)
|   |   |   |   |   nTempBin = 7: INCREASE (40.0/19.0)
|   |   |   |   |   nTempBin = 8: DECREASE (1093.0/458.0)
|   |   |   |   |   nTempBin = 32766: DECREASE (48.0/27.0)
|   |   |   |   wTempBin = 32766: INCREASE (54.0/23.0)
|   |   |   eTempBin = 32766: DECREASE (51.0/27.0)
|   |   sTempBin = 32766
|   |   |   eTempBin = 0: INCREASE (0.0)
|   |   |   eTempBin = 1: INCREASE (0.0)
|   |   |   eTempBin = 2: INCREASE (0.0)
|   |   |   eTempBin = 3: INCREASE (0.0)
|   |   |   eTempBin = 4: INCREASE (0.0)
|   |   |   eTempBin = 5: INCREASE (0.0)
|   |   |   eTempBin = 6: INCREASE (0.0)
|   |   |   eTempBin = 7: INCREASE (2.0)
|   |   |   eTempBin = 8: INCREASE (68.0/33.0)
|   |   |   eTempBin = 32766
|   |   |   |   wTempBin = 0: DECREASE (0.0)
|   |   |   |   wTempBin = 1: DECREASE (0.0)
|   |   |   |   wTempBin = 2: DECREASE (0.0)
|   |   |   |   wTempBin = 3: DECREASE (0.0)
|   |   |   |   wTempBin = 4: DECREASE (0.0)
|   |   |   |   wTempBin = 5: DECREASE (0.0)
|   |   |   |   wTempBin = 6: DECREASE (0.0)
|   |   |   |   wTempBin = 7: INCREASE (1.0)
|   |   |   |   wTempBin = 8
|   |   |   |   |   nTempBin = 0: DECREASE (0.0)
|   |   |   |   |   nTempBin = 1: DECREASE (0.0)
|   |   |   |   |   nTempBin = 2: DECREASE (0.0)
|   |   |   |   |   nTempBin = 3: DECREASE (0.0)
|   |   |   |   |   nTempBin = 4: DECREASE (0.0)
|   |   |   |   |   nTempBin = 5: DECREASE (0.0)
|   |   |   |   |   nTempBin = 6: DECREASE (0.0)
|   |   |   |   |   nTempBin = 7: DECREASE (0.0)
|   |   |   |   |   nTempBin = 8: DECREASE (15.0/6.0)
|   |   |   |   |   nTempBin = 32766: INCREASE (2.0)
|   |   |   |   wTempBin = 32766: DECREASE (8.0/2.0)
|   nHemisphere = Northern: DECREASE (693.0/115.0)
targetTempBin = 32766: SAME (22319.0)

Number of Leaves  :      280

Size of the tree :  319
```

```
=== Re-evaluation on test set ===

User supplied test set
Relation:     neigh91.ARFF-
weka.filters.unsupervised.attribute.Remove-R1-2,5-6,13-
14,17-18
Instances:    64800
Attributes:   11

=== Summary ===

Correctly Classified Instances       53340
82.3148 %
Incorrectly Classified Instances     11460
17.6852 %
Kappa statistic                          0.7256
Mean absolute error                      0.1633
Root mean squared error                  0.296
Relative absolute error                 38.1328 %
Root relative squared error             63.9711 %
Total Number of Instances            64800

=== Detailed Accuracy By Class ===

TP Rate   FP Rate   Precision   Recall  F-Measure   Class
  0.767     0.083      0.839     0.767     0.801
INCREASE
  0.721     0.099      0.661     0.721     0.69
DECREASE
  0.922     0.079      0.896     0.922     0.909    SAME

=== Confusion Matrix ===

     a     b     c   <-- classified as
 17998  3954  1522 |     a = INCREASE
  2400  9876  1422 |     b = DECREASE
  1046  1116 25466 |     c = SAME
```

# REFERENCES


Agrawal, R. and R. Srikant. 1994. *Fast algorithms for mining association rules*. In Proceedings of the International Conference on Very Large Databases, pages 487-499, Santiago, Chile.

Berthold, M. R., and Borgelt, C. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *Proceedings of IEEE International Conference on Data Mining,* 2002.

Blummer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M., 1987. Occam's Razor. Information Processing Letters. 24:377-380



Blummer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M., 1987. Occam's Razor. Information Processing Letters. 24:377-380



Cook, D. and Holder L., 2000. Graph-Based Data Mining. *IEEE Intelligent Systems 15(2): 32-41.*

Deshpande, M., Kuramochi, M. and .Karypis, G. Chapter 9, Mining Chemical Compounds. In *Data Mining in Bioinfomatics*, Springer-Verlag, London 2005.

Gonzalez, J., Holder, L. B., and Cook, D. J. 2001. Graph-Based Concept Learning. Florida Artificial Intelligence Research Symposium*,*377-381.

Gonzalez, J., Holder, L. B., and Cook, D. J. 2002. Graph-Based Relational Concept Learning, *Proceedings of the International Machine Learning Conference 2002.*

Djoko, S., Cook, D. J. and Holder, L. B., and 1992. An Empirical Study of Domain Knowledge and its Benefits to Substructure Discovery, *IEEE Transactions on Knowledge and Data Engineering.,* 9(4): 575-586

Holder, L., Cook, D., and Bunke, H., 1992 Fuzzy Substructure Discovery. In *Proceedings of international Conference on Machine Learning,* 218-233.

Holder, L., Cook, D., and Djoko, S., 1994 Substructure Discovery in the Subdue Sytem. In *Proceedings of Knowledge Discovery in Databases Workshop,* (KDD-94), 169-180.

Holder, L., Cook, D., Coble, J. and Mukherjee, M., 2005 Graph-Based Relational Learning with Application to Security, 2005 Fundamenta Informaticae Special Issue on Mining Graphs, Trees, and Sequences, 6(1-2):83-101, March 2005

Inokuchi, A., Washio, T., and Motoda, H. An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. Ih *Proceedings of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases,* pages 13-23,Lyon, France 2000.

Jonyer, Holder, L. B., and Cook, D. J. 2000. Graph-Based Hierarchical Conceptual Clustering. In reply to: *Proceedings of the Florida AI Research Symposium*,91—95.

JPL, Physical Oceanography DACC, WOCE Global Data, V2.0,Satellite Data, Sea Surface Temperature, July 2000

Ketkar, N., Holder, L., and Cook, D., A Comparative Study of Graph-based and Logic-based Relational Learning. Preliminary work. Under review by the International Conference on Machine Learning, 2005.

Kuramochi, M and Karypis. G., Frequent Subgraph Discovery. Ih *Proceedings of IEEE Innternational Conference on Data Mining*, pages 313-320, San Jose, CA 2001

Kuramochi, M and Karypis. G., Discovering Frequent Geometric Subgraphs. Ih *Proceedings of IEEE Innternational Conference on Data Mining*, pages 258-265, 2002

Kuramochi, M and Karypis. G., 2004 An efficient algorithm for discovering frequent subgraphs. In *IEEE transactions on Knowledge and Data Engineering, 16(6)2004*

Mitchell, Tom M. 1997. *Machine Learning.* McGraw-Hill, New York, New York.

Muggleton, S., and De Raedt, L. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming,* 19:629-679,1994.

Potts, J., Cook, D., Holder, L., Learning From Examples in a Single Graph. Florida Artificial Intelligence Research Symposium, 2005

Potts, J., Holder, L., Cook, D., Coble, J., Learning Concepts from Intelligence Data Embedded in a Supervised Graph. International Conference on Intelligence Analysis, 2005

Provost, F. and Fawcett, T., 1998 Robust classification systems for imprecise environments. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence.* 706-713,1998

Quinlan, J. and Rivest, R., 1989. Inferring Decision Trees Using the Minimum Descriptive Length Principle. *Information and Computation* 80:227-248

Rissanen, J. , 1989. Stochastic Complexity in Statistical Inquiry. World Scientific Publishing Company.

Storer, J.,1989 Stjochastic Complexity in Statistical Inquiry. World Scientific Publishing Company.

Swets, J., Dawes, R. and Monahan, J., Better decisions through Science. Scientific American, October 2000:82-87

Winston, Patrick Henry: *Artificial Intelligence, Third Edition.* page 403, Addison-Wesley, Reading, Massachusetts, 1992

Witten, I.. and Frank, E: *Data Mining, Practical Machine Learning and Techniques, Second Edition.* page 365, Morgan Kaufmann, Palo Alto, California, 2005

Yan, X. and Han, J. gSpan: Graph-based Substructure Pattern Mining. In *Proceedings of International Conference on Data Mining*, pages 721-724, Maebashi, Japan, 2002

# BIOGRAPHICAL INFORMATION

Joseph T. Potts graduated from Oklahoma State University in Stillwater, Oklahoma, in 1970. He worked in the petroleum industry and the data processing industry or a number of years as a programmer and system designer. He also managed system development teams, data centers, and customer service departments. He returned to school at the University of Texas at Arlington in 1992 to study artificial intelligence. He received a Master of Science in Computer Science in December of 1996. He received his Doctor of Philosophy in Computer Science in 2006.