Eclectic Machine Learning

A Dissertation
Presented to the
Department of Computer Science
Brigham Young University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

by
Cory Barker
February 1994

This dissertation by Cory Barker is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the dissertation requirement for the degree Doctor of Philosophy.

Tony R. Martinez, Committee Chair

James K. Archibald, Committee Member

Dan R. Olsen, Committee Member

Date

David W. Embley, Graduate Coordinator

ABSTRACT

This dissertation presents a family of inductive learning systems that derive general rules from specific examples. These systems combine the benefits of neural networks, ASOCS, and symbolic learning algorithms. The systems presented here learn incrementally with good speed and generalization. They are based on a parallel architectural model that adapts to the problem being learned. Learning is done without requiring user adjustment of sensitive parameters, and noise is tolerated with graceful degradation in performance.

The systems described in this work are based on *features*. Features are subsets of the input space. One group of learning algorithms begins with general features and specializes those features to match the problem that is being learned. Another group creates specific features and then generalizes those features. The final group combines the approaches used in the first two groups to gain the benefits of both.

The algorithms are O(m log m), where m is the number of nodes in the network, and the number of inputs and output values are treated as constants. An enhanced network topology reduces time complexity to O(log m). Empirical results show that the algorithms give good generalization and that learning converges in a small number of training passes.

COMMITTEE APPROVAL:

Tony R. Martinez, Committee Chair

James K. Archibald, Committee Member

Dan R. Olsen, Committee Member

Date

David W. Embley, Graduate Coordinator

Acknowledgments

Thanks be to God for giving me the ideas, mental strength, and perseverance that have made this work possible. All that is good in this work is due to Him. I take responsibility for poor communication and errors in writing, but give all credit for the positive contributions of this work to God.

Thanks to my dear wife Colette and our children. They have given up many things (including time with husband and father) to allow me to return to school and complete this work. I also thank parents and others from both my family and Colette's family who have without exception supported my schooling.

Thanks to Helen Hepworth (my fourth grade schoolteacher) who convinced me that I had the ability to accomplish anything that I set my mind to do. Teachers who honestly care for their students are a huge blessing not only to the individual student but also to our society as a whole. Without her encouragement it is certainly possible that the shy boy of ten years that I was then would never have become confident enough to tackle this work.

Thanks to Mike Fitzgerald for putting the thought in my head that Cory Barker should get a Doctoral Degree and also for giving expert instruction on English language grammar and punctuation.

Thanks to my committee, Tony Martinez, Dan Olsen, and Jim Archibald, as well as other friends and colleagues, for taking time to read my work and give helpful constructive comments that have improved the result immensely. Thanks to Brigham Young University and the Computer Science Department as well as the Church of Jesus Christ of Latter-Day Saints for providing equipment and other support for this work. Thanks to the Air Force Office of Scientific Research and Wright Laboratory for providing an Air Force Laboratory Graduate Fellowship for funding this research.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1. Overview

This dissertation presents a family of inductive learning systems. Inductive learning systems extract general rules from specific examples [Kodratoff 1990, Michalski 1983b, Michalski 1986]. The system is given a set of examples that define a partial function from an input space to an output space. (The examples may be noisy, and thus the relation between input and output specified by the examples may be more general than a function.) The examples are called a training set or instance set. Each example defines the mapping from one input point to one output point. The system implements the mapping defined by the examples and extends the partial function to give good outputs in cases not covered by the examples. Most systems typically find a simple representation that agrees with the training set. The simple representation of the training set tends to predict the correct output for inputs that the system has not been trained for.

Inductive learning systems should provide:

- Good generalization.
- Good learning speed.
- Incremental learning.
- Graceful degradation in the presence of noise.
- Ease of use.
- A parallel architectural model.

Generalization is a basic requirement of inductive learning systems. As the number of inputs to a system grows linearly, the number of points in the input space of the system grows exponentially. Thus, a system cannot be given a training set that completely covers the input space, and the output for many points in the input space must be determined by the system. The system should give the correct output for most of the points not defined by the training set.

An inductive learning system should learn quickly. A training example should be incorporated by the system using the fewest possible steps. The correct output for a training example should be learned by the system in the fewest possible presentations.

Inductive learning systems should learn incrementally. Each training example should be given to the system separately. The example should be incorporated into the system without requiring the system to reprocess all previous training examples. The system should be capable of executing and producing an output between the presentation of training examples. A non-incremental system can be made to appear incremental by simply storing all training examples and processing them again as each new example is given. Fast learning is difficult to achieve with such an approach.

The training set may have incorrect values for either inputs or outputs. An inductive learning system should recognize noisy examples and filter them rather than learning them.

The system should not have sensitive parameters that the user must carefully adjust to give good learning.

An inductive learning system should be based on a parallel architectural model. Such a model allows the system to take advantage of the large numbers of processors available on current and future parallel machines.

The new systems presented in this work provide a balance of these six attributes. Simulation results for the new systems suggest that the systems give generalization performance comparable to that of other inductive learning systems.

A complexity analysis of the new algorithms shows that the systems incorporate a new example in O(m log m) time, where m is the number of nodes in the system. The simulation results suggest that a training example needs to be presented to the system only a small number of times before the example is learned.

The new learning algorithms accept training examples incrementally and incorporate the examples into the system. The system is then capable of either more learning or execution.

Simulation results on noisy data sets suggest that the new systems gracefully degrade in the presence of noise.

The new systems have a single parameter that may be adjusted by the user, but the learning performance of the systems is not significantly affected by relatively large changes to the parameter.

Two parallel architectures for implementation of the new algorithms are presented in this work. One of the architectures improves the time complexity for learning an example to O(log m), where m is the number of nodes in the system.

The remainder of Chapter 1 gives the motivation for building machines that learn followed by a description and comparison of current learning models. Section 1.2 gives several reasons why learning machines are needed. Section 1.3 describes ASOCS, neural networks, and AI symbolic inductive learning algorithms and gives strengths and weaknesses of each method. Section 1.4 lists techniques and ideas used in current models. Section 1.5 explains how the techniques are combined into the new systems developed in this work. Finally Section 1.6 gives an overview of the rest of the dissertation.

1.2. Motivation for Learning

Many problems that are desirable to solve, such as decision problems, vision, and speech, are very difficult for traditional procedural approaches. These problems are typically easy for humans to solve. A difference between humans and current computing approaches is learning. Use of inductive learning in portions of these problems could allow easier solutions.

Some problems are dynamic and require machines that can adapt in order to provide a robust solution. Traditional software is static and requires the programmer to anticipate all possible problem variations in advance. A learning machine can learn incrementally as the problem changes and thereby adapt to the problem.

Many problems are complex to the point that current approaches cannot compute the solution fast enough. Learning systems that make use of parallel hardware have potential to increase computing capability. An example from nature, the human brain, suggests that parallel techniques can improve speed for many of these interesting problems.

The rapid advance of semiconductor technology promises to provide a billion devices on a single die in the near future. Using traditional software techniques, it is difficult to effectively manage such a large amount of hardware. Self-organizing and adaptive machine learning techniques provide a means to harness this vast hardware resource.

The cost and complexity of developing software are very high and continue to rise while hardware costs decrease. Parts of a project may be specified in such a way that the desired functionality can be learned. Such portions of the system can be implemented without programming by using learning and thus lowering software development cost. Maintenance and enhancement costs for these portions of the product would be reduced. A learning system can automatically correct errors. Improvements or changing of requirements can be implemented by simply relearning.

Use of learning systems can reduce the cost and complexity of solving many significant computing problems while providing robustness and accuracy otherwise unachievable.

1.3. Machine Learning Algorithms

This work began with the study of existing approaches to machine learning. These existing approaches to machine learning each have strengths and weaknesses. The strength of an approach is usually the result of a particular feature of the approach. The plan for this work was to combine the good features of existing approaches into a new model. The hope was that by using this eclectic approach the new models would combine the benefits of many existing machine learning models. The simulation and complexity analysis results suggest that a balance of these benefits has been obtained.

Existing models are divided into three main categories, adaptive self-organizing concurrent systems (ASOCS), neural networks, and AI inductive learning algorithms. Each of the three categories will be described along with the strengths and weaknesses of the models in the category. ASOCS models have good speed and generality. Neural networks provide good noise filtering and generalization. AI inductive algorithms give good generalization and work at a symbolic level. The systems developed in this research provide all of these benefits.

1.3.1. ASOCS

An Adaptive Self-Organizing Concurrent System (ASOCS) [Martinez 1986] is an adaptive network composed of many simple computing elements operating in parallel. An ASOCS operates in one of two modes: learning and processing. In learning mode, rules are presented to the ASOCS and incorporated in a self-organizing fashion. Learning is done both by selecting node functions and by dynamically changing the network topology. In processing mode, the ASOCS acts as a parallel hardware circuit that performs the function defined by the learned rules.

Three models of ASOCS were originally developed; Adaptive Algorithms 1 [Martinez 1988], 2 [Martinez 1991a], and 3 [Martinez 1991b] (AA1, AA2, and AA3). Both AA2 and AA3 create and connect new nodes that perform simple two-input *and* functions. New nodes are connected to exactly match the conjunction specified in the input example.

The AA3 algorithm does not make this clear, but it has been shown to be true in [Barker 1994]. Thus, AA2 and AA3 do no generalization; they do not attempt to give good outputs for inputs they have not been trained on. The networks simply do rote learning of the instances that are given to them.

AA1 builds networks of nodes that perform two-input *and* and *or* functions. AA1 builds networks much differently than AA2 or AA3. AA1 attempts to find a small set of nodes that correctly discriminates between the examples that have different outputs. Subnetworks of nodes that correctly discriminate a portion of the total set of examples are combined by adding nodes with the correct function to create a network that discriminates all the examples. Thus, AA1 finds a simple representation that agrees with the training examples. AA1 generalization is not as good as current inductive learning systems [Martinez 1993].

ASOCS models have the following strengths. ASOCS models are guaranteed to learn any mapping. Learning a new instance and execution are both guaranteed to be done in log time (O(log m) where m is the number of nodes in the network). The topology of an ASOCS network dynamically changes to fit the problem to be solved. ASOCS models implement incremental learning, meaning that an existing network can learn and adapt to new training instances as they become available.

ASOCS models have the following weaknesses. ASOCS AA2 and AA3 models provide no generalization of untrained inputs. The AA1 model does generalization by discrimination but requires four bits of memory per instance in each node. Noisy inputs are a problem for ASOCS because errors caused by noise are learned perfectly. Extensions to basic ASOCS models support multistate inputs, but real valued inputs require additional input processing.

1.3.2. Neural Networks

Many neural network architectures have been developed including backpropagation [Rumelhart 1986], Boltzmann machines [Ackley 1985], and spontaneous learning systems [Rumelhart 1985]. These models are characterized by many simple computing elements (nodes) operating in parallel with a large number of interconnections. Most models use a static network topology and learn by changing node functions. Each node usually outputs either a threshold or logistic function of a weighted sum of its inputs. The network learns by making small adjustments to the values of the weights on the inputs to the nodes. The adjustments are ultimately based on the difference between the actual network output and the desired output.

The Perceptron uses only a single layer of nodes. Each node implements a linear function of its inputs. Thus, the Perceptron is limited to learning only linearly separable functions [Minsky 1969]. The backpropagation learning algorithm can train multiple layers of nodes. Backpropagation is theoretically capable of learning any function.

Neural network models have the following strengths. Neural network models such as backpropagation provide good generalization. Because training involves several presentations of the training set, noise is filtered statistically. Execution is fast because the nodes all compute their functions in parallel.

Neural network models have the following weaknesses. Neural network models lack any guarantee that the system will converge on a solution. The system may get stuck in a local minimum. Even if the system does converge it may do so very slowly and require a large number of training set presentations. Most current models use a static topology and therefore limit the set of problems that can be solved. Neural networks usually have parameters that must be carefully adjusted by the user to get good results.

1.3.3. AI Symbolic Induction Algorithms

AI Induction algorithms [Dietterich 1986] typically perform a search through a space of functions. A measure of goodness for a function is defined and the search attempts to find the best function. Some algorithms attempt to find an approximation to the best function. No architectural model is involved in these algorithms; the algorithms simply run on sequential machines.

ID3 [Quinlan 1986] is probably the most prominent algorithm for induction in the AI literature. ID3 attempts to find the smallest decision tree that fulfills the examples. The algorithm chooses the variable to use as the root of the tree by minimizing the amount of information or entropy contained in the subtrees. A root is then repeatedly selected for each subtree.

AQ [Michalski 1983a] and Version Space [Mitchell 1982] are similar methods that seek to find a disjunctive normal form generalization with the fewest disjuncts. General rules are constructed by applying generalization rules to examples. Rules that cover many positive instances but cover no negative instances are good rules. The best of these rules are combined into a disjunction.

AI Induction algorithms have the following strengths. AI Induction algorithms generalize about as well as neural network models [Mooney 1989, Weiss 1989]. The generalized result is expressed at a symbolic level allowing direct understanding of the result, as opposed to the neural network models where the result is expressed as a set of weight values.

AI Induction algorithms have the following weaknesses. Typical algorithms do not handle noise or inconsistent data. Most algorithms process the complete training set at one time and thus are not incremental learners. Usually, no architectural model is given for these algorithms. Most algorithms use a representation for the rules that result from induction that limits the set of problems that can be solved.

1.4. Learning Techniques

This section lists the techniques that were identified during the study of the learning algorithms described above. These techniques are combined to develop the new algorithms presented in this work. The techniques are listed below followed by a more detailed description of each technique and its origin.

- A distributed network topology with dynamic creation, deletion, and interconnection of nodes.
- Learning only when the system output differs from the example output.
- Generalization by dropping variables from conjunctions.
- Control of generalization by measuring contradictions.
- Filtering of noise by making only small changes to the system for each training example.

ASOCS algorithms use a distributed network topology with dynamic creation, deletion, and interconnection of nodes. This allows ASOCS to adapt the network structure to match the problem that is being learned.

The AA1 algorithm presents the input of an example to the network and allows the network to compute an output. If the output of the network is the same as the output of the training example, then the network is not modified.

The AQ and Version Space algorithms use generalization rules for making examples general. An important rule that is used in this work is dropping a variable from a conjunction. When any variable is removed from a conjunction, the resulting conjunction covers or matches more of the points in the input space than the original conjunction. Thus, the new conjunction is more general. A related rule used in this work for making examples more general is intersecting the sets of variables used in two conjunctions. The conjunction resulting from intersection is guaranteed to cover all input points covered by both of the original conjunctions.

The generalization rules can be reversed to give specialization rules. The specialization rules of adding a variable to a conjunction and taking the union of the sets of variables in two conjunctions are also used in this work.

AQ and Version Space algorithms control the amount of generalization that is done by stopping the generalization process when the general rules contradict one or more examples. An example is made more general until the resulting rule matches examples with different output. Thus when a general rule contradicts examples, generalization of that rule terminates.

Neural networks make small increments and decrements to weights on the inputs of nodes. Thus each example causes the activation of a node to change by a small amount until some threshold is finally reached. Since no single example causes major changes in the network, noisy examples are not learned as if they were correct. The models described in this work use a similar method to deal with noise and inconsistency. Each example causes counters to be incremented. The counters determine the strength of nodes in the network. Competition among nodes is according to strength. Typically no single example can cause a node to be the strongest and thus affect the network output.

1.5. Combination of Techniques

The techniques described above are combined to create the models developed in this work.

The models described in this work use a dynamic topology like ASOCS models where nodes are created and deleted as the system learns. Each node in the network stores a feature. A feature is a conjunction of input variables.

Each new example is given to the network and the network produces an output. When the network output is different from the example output, the network is modified to correctly handle the new example.

Each example causes counters in the nodes to be incremented. The counters determine the strength of a node for each possible output of the network. Noisy examples cause the counter for a different output to be incremented with a probability determined by the amount of noise. For most examples, the counter for the correct output is incremented, and the strongest output for a node is still the correct output.

One group of learning models described in this paper works from general to specific. The algorithms begin by creating general features and specialize the features. This technique is similar to ID3 where single input variables are selected as important features. The general features are specialized using AQ and Version Space specialization rules to create more specific features.

A second group of models developed in this work start with specific features and then generalize those features. These models are similar to AQ in that specific examples are generalized using the AQ generalization rules. Examples are allowed to generalize until they conflict with other examples as is done in AQ and Version Space.

The algorithms developed in this work do not suffer from the weaknesses described above for each of the categories of learning algorithms. The algorithms provide good generalization and noise handling unlike ASOCS models. Learning is accomplished with few passes over the training set and without requiring user adjustment of critical parameters unlike neural network models. Learning is incremental and based on dynamic parallel architectures unlike AI induction algorithms. Thus, combining techniques from all three categories of algorithms has resulted in new models that give a balance of the positive characteristics of previous algorithms while avoiding the problems with previous algorithms.

1.6. Overview of the Dissertation

This section gives a brief description of the contents of each chapter of the dissertation.

Chapter 2 defines the representation used for training sets. Input and output spaces are described along with what is meant by learning from examples. This chapter explains in detail the problem that is targeted by the systems described in this paper.

Chapter 3 defines features. Important properties of features such as generality, specificity, and strength are described. Important relationships between features and the input space are proven. The chapter explains how a set of features can provide a mapping from an input space to an output space. Finally, it is shown how a set of features can be implemented using a network of nodes.

Chapter 4 shows how a set of features can be learned using a general to specific approach. First general features are created followed by their specialization into more specific features. Five different methods for creating specific features are presented. Each method has increasing potential for better generalization performance. Details of each algorithm and examples of learning are presented.

Chapter 5 shows how a set of features can be learned using a specific to general approach. Examples are completely specific features. Features are first created that are equivalent to examples. The specific features are then generalized until they contradict other features. Two different methods for creating general features are presented. Details of each method along with examples of learning are given.

Chapter 6 shows how the general to specific and specific to general approaches can be combined into a single generalization and specialization approach. The two methods are combined either by doing both generalization and specialization or by doing the best of generalization or specialization. Details and examples of each algorithm are given.

Chapter 7 provides two different approaches to connecting nodes into a network structure. One method uses simple broadcast and gather operations that can be implemented on tree architectures. The second method reduces network traffic by connecting nodes that have general to specific relationships.

Chapter 8 gives a complexity analysis for each algorithm described in this dissertation. Time complexity bounds are given in terms of the number of inputs to the system, the number of nodes in the system, the number of output values, and the number of training examples.

Chapter 9 gives empirical test results from running each algorithm on ten data sets. The data sets cover a wide range of domains from medical diagnoses to plant disease classification. Accuracy results are given to provide a measure of generalization performance. Node counts and pass counts are given to provide a measure of learning speed.

Chapter 10 summarizes the contributions of this work and suggests possible directions for future work.

Chapter 2
Learning from Examples

This chapter describes what is meant by *learning from examples*. The concepts of *input* and *input space* are defined, as well as the concepts of *output* and *output space*. *Examples* and *training sets* are described and the issue of *generalization* is discussed. Examples of each of these concepts are presented.

2.1. Input and Output Spaces

The goal in learning from examples is to find a mapping from some input space I to some output space O. The inputs may be attributes of objects, and the outputs, classifications for those objects. Alternatively, the inputs may be measurements from an environment around the system, and the outputs, decisions about an action to take.

A system S that learns from examples has *inputs* $i_1, ..., i_n$, with domains $X_1, ..., X_n$, respectively, and *outputs* $o_1, ..., o_m$, with domains $Y_1, ..., Y_m$, respectively. The *input space* of S is $I = X_1 \times ... \times X_n$ (the cross product of the input domains) and the *output space* is $O = Y_1 \times ... \times Y_m$. The members of I are all possible combinations of the values of the input variables, while the members of O are all possible combinations of the values of the output variables.

In this work, a member of these sets is referred to as either a *point* in the input (output) space or a *state* of the input (output). For the purposes of this work, the input and output domains, $X_1, ..., X_n$ and $Y_1, ..., Y_m$, are restricted to be finite sets of discrete values. The system is also restricted to have only a single output. Continuous variables can be processed by discretizing them before they are presented to the system. Note that some states of the input space may never occur in practice. The single output restriction can be addressed in two ways; (1) a separate system can be implemented for each output or (2) each point in the output space can be treated as a different value of a single output.

We now give an example of input and output spaces. Suppose the problem to be learned is to decide whether to use manual or automatic flight controls when piloting an aircraft, based on the environmental conditions surrounding the aircraft. The inputs to the system are *Wind Speed, Wind Direction,* and *Visibility*. The output from the system is *Flight Control*. *Wind Speed* has domain {*Low, Medium, High*}. *Wind Direction* has domain {*Head, Tail*} and *Visibility* has domain {*Clear, Cloudy*}. *Flight Control* has domain {*Manual, Automatic*}.

The input space or input domain for the system is {(*Low, Head, Clear*), (*Low, Head, Cloudy*), (*Low, Tail, Clear*), (*Low, Tail, Cloudy*), (*Medium, Head, Clear*), (*Medium, Head, Cloudy*), (*Medium, Tail, Clear*), (*Medium, Tail, Cloudy*), (*High, Head, Clear*), (*High, Head, Cloudy*), (*High, Tail, Clear*), (*High, Tail, Cloudy*)}. This domain is shown in tabular form in « AircraftInputDomain». Since there is only one output, the output space or output domain is simply {*Manual, Automatic*}.

Table 2.1. Input Domain for the Aircraft Control Problem

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

The learning task of the system is to associate a member of the output domain with each of the 12 members of the input domain. For this example, the system must assign either *Manual* or *Automatic* to each space in « AircraftInputDomain». Thus, the system S implements a function f_S from I to O.

2.2. Examples and Training Sets

To enable the system to find a mapping that is better than a random mapping or some other arbitrary result, it is given a *training set* containing *examples*. A training *example* is a pair (i, o) from $I \times O$. Each example gives the input/output association for one point in the input domain. A training set is a subset of $I \times O$. Finding the output assignments for inputs that are defined by examples is simple: the system just uses the output given in the example. However, the training set is a subset of the complete mapping from input to output. Not all points in the input domain are defined by training examples. The system must *generalize* the examples to find good assignments for points in the input space that are not given in the training set.

Assume the aircraft control problem has the following training set:

(*High, Tail, Cloudy*) → *Automatic*

(*Low, Head, Cloudy*) → *Manual*

(*High, Head, Cloudy*) → *Manual*

(*Low, Tail, Clear*) → *Automatic*

The training set is shown in the tabular form of the input domain in « AircraftTrainSet».

Table 2.2. Training Set for the Aircraft Control Problem

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low		Manual	Automatic	
Medium				
High		Manual		Automatic

Note that eight spaces in the table are blank where the system must generalize on the training examples to determine the outputs. One possible way the system could generalize is to assign *Manual* to the six spaces on the left of the table and *Automatic* to the six spaces on the right of the table.

One may reason that learning would be trivial if the system were just given a training set containing the complete mapping from input to output. The difficulty with this approach is that for many applications, complete information is not available. For reasons of convenience, the system should be capable of learning from examples taken directly from its own experience, rather than being given a carefully prepared training set. The most important reason for not giving a complete set of examples is that the number of points in the input space—and thus the number of examples in the training set—grows exponentially in terms of the number of inputs to the system.

Chapter 3
Features

This chapter describes the concept of a *feature*. Properties of features such as *generality* and *strength* are discussed.

A feature is some subset of system inputs along with their associated values. A feature is *matched* when the values on the system inputs, that are part of the feature, are equal to the values for those inputs as given in the feature. Inputs that are not part of the feature can be any value. A feature matches some subset of the total input domain. It is assumed that the problem the system is learning has some interesting features (or combinations of inputs) that can be used for generalization, otherwise the problem is unlearnable.

More formally, a *feature* is a set of pairs (i_j, x), where i_j is an input and x is a member of the input domain X_j. A feature *matches* a point in the input space if for each pair (i_j, x), the value$(i_j) = x$. The set of points in the input space matched by a feature is called the *subdomain* of the feature.

A feature matches an example if the feature matches the input part of the example. Formally, a feature F matches an example $E = (i, o)$ if F matches i.

« FeatureSubdomains» shows examples of some features and their subdomains. The first feature, {(*Wind Speed, High*)}, matches any point in the input space that has the first input set to *High*. The other two inputs can be any value. The second feature in the table, {(*Wind Direction, Tail*), (*Visibility, Clear*)}, specifies the values for the last two inputs and so only matches points where those inputs are *Tail* and *Clear*.

Table 3.1. Features and their Subdomains

Feature	Subdomain
{(*Wind Speed, High*)}	{(*High, Head, Clear*), (*High, Head, Cloudy*), (*High, Tail, Clear*), (*High, Tail, Cloudy*)}
{(*Wind Direction, Tail*), (*Visibility, Clear*)}	{(*Low, Tail, Clear*), (*Medium, Tail, Clear*), (*High, Tail, Clear*)}
{(*Visibility, Cloudy*)}	{(*Low, Head, Cloudy*), (*Low, Tail, Cloudy*), (*Medium, Head, Cloudy*), (*Medium, Tail, Cloudy*), (*High, Head, Cloudy*), (*High, Tail, Cloudy*)}
{(*Wind Speed, Medium*), (*Wind Direction, Head*), (*Visibility, Clear*)}	{(*Medium, Head, Clear*)}

« SubdomainVisCdy» graphically shows the subdomain for the third feature in « FeatureSubdomains».

Table 3.2. Subdomain for feature {(*Visibility, Cloudy*)}

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

3.1. Feature Generality

The *generality* of a feature is a measure of how much of the input space is matched by the feature. A feature with few inputs is a *general* feature: it matches many points in the input space. For example, the feature {} with no inputs matches the complete input domain. Each pair (i_j, x) in a feature restricts the subdomain of the feature to only points that match that pair. If the feature contains few pairs, the subdomain is less restricted and therefore larger.

Formally, the generality of a feature F is $\frac{|S_F|}{|I|}$, where S_F is the subdomain of F and I is the complete input domain. Each input i_j *not* contained in a feature allows the feature to cover $|X_j|$ times the input space that it would cover with the input. So each input not contained in a feature increases the generality of the feature by the number of possible values for the input. The generality of the feature with no inputs, {}, is 1, which is also the maximum value for generality. The generality of a feature with all inputs specified is $\frac{1}{|I|}$, or near 0, since such a feature matches only a single point in the input space.

For example, the generality of feature {(*Wind Direction, Head*)} shown in « FeatureGenerality» is 6/12 = .5, since there are 6 points in the subdomain of the feature and 12 points in the complete input space.

Table 3.3. Feature Generality

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

For simplicity, a different measure of generality may be used for implementation. The *simple generality* of a feature F is defined as $\frac{n-|F|}{n}$, where n is the total number of inputs to the system. Simple generality is the number of inputs *not* contained in the feature divided by the total number of inputs. The simple generality of the feature {} is 1, and the simple generality of a feature containing all inputs is 0.

Simple generality is not as accurate as true generality, but it is easier to compute. When all inputs have domains of the same size, simple generality gives the same relative ordering of features as true generality. Simple generality differs more from true generality when the inputs have domain sizes that differ from each other.

3.2. Feature Relationships

3.2.1. Theorem 3.1. Feature subset (superset) ⇔ Subdomain superset (subset)

Given two feature input/value sets A and B and their subdomains D_A and D_B, A is a subset (superset) of B if-and-only-if D_A is a superset (subset) of D_B.

Proof

If A is a subset of B then the only difference between A and B is that B contains additional pairs not contained in A. The additional pairs in B serve to restrict points matched by B to be a subset of those matched by A. Therefore, D_A is a superset of D_B. Conversely if D_A is a superset of D_B then the points matched by A are a superset of those matched by B. Therefore A cannot contain any pair not contained in B. Such a pair would restrict A to not match some point matched by B. So A must be a subset of B.

The features in « FeatureSubSuper» show an example of this relationship. The feature {(*Wind Direction, Head*)} is a subset of the feature {(*Wind Direction, Head*), (*Wind Speed, Medium*)} while the subdomain of the feature {(*Wind Direction, Head*)} is a superset of the subdomain of the feature {(*Wind Direction, Head*), (*Wind Speed, Medium*)}.

Table 3.4. Feature Subset/Superset Relationships

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

This result is important from the standpoint of implementation since feature subdomains can be compared by comparing the feature sets rather than by comparing subdomains. The subdomain of a feature is potentially exponential in size relative to the size of the feature. Thus, comparing feature input/value sets is more efficient.

3.2.2. Definition 3.1. Feature Overlap

Feature A *overlaps* feature B if the intersection of the subdomain of feature A and the subdomain of feature B is not empty. In other words, there is a least one point in the domain that is matched by both feature A and feature B.

« FeatureOverlap» shows an example of feature overlap between the features {(*Wind Direction, Head*)} and {(*Wind Speed, Medium*), (*Visibility, Cloudy*)}.

Table 3.5. Feature Overlap

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

3.2.3. Definition 3.2. Feature Discrimination

Feature A is *discriminated* from feature B if the intersection of the subdomain of feature A and the subdomain of feature B is empty. In other words, there is no point matched by both A and B. Feature discrimination is the logical complement of feature overlap.

« FeatureDiscrim» shows two features that are discriminated, {(*Wind Direction, Head*), (*Visibility, Clear*)} and {(*Wind Speed, Medium*), (*Visibility, Cloudy*)}.

Table 3.6. Feature Discrimination

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

3.2.4. Theorem 3.2. Feature discrimination ⇔ Discriminant input

Two features A and B are discriminated if-and-only-if there is a pair (i_j, x) in A and a pair (i_j, y) in B such that $x \neq$ y. In other words, the features A and B both restrict the same input but each feature restricts the input to be a different value. The input i_j is called a *discriminant input* for the two features.

Proof

Let the two features A and B be discriminated and assume that there is no discriminant input for the two features. Then there is a feature $C = A \cup B$. The feature C only exists because there is no discriminant input for A and B. If a discriminant input i_j does exist for A and B, then feature C would contain both pairs (i_j, x) and (i_j, y) and C could not match any point in the input domain. If feature C does exist, then C is a superset of both features A and B because of the definition of set union. By « FsubsetSDsuperTh» the subdomain of C must be a subset of both the subdomain of A and the subdomain of B. Since both the subdomain of A and the subdomain of B must contain the subdomain of C, then A and B must both match at least one common point in the input domain. This is a contradiction of the statement that the features A and B are discriminated. Therefore, the assumption that there is no discriminant input must be false.

Conversely, if two features A and B have a discriminant input, then it is impossible for the two features to match the same point. There is a pair (i_j, x) in A that requires the input i_j to be equal to x and there is a pair (i_j, y) in B that requires the input i_j to be equal to y. There is no point in the input space where an input can be equal to two values at one time. Since the subdomains of the two features do not have any point in common the intersection of the subdomains is empty and the two features are discriminated by the definition of discrimination.

« FeatureDiscrim» shows the features {(*Wind Direction, Head*), (*Visibility, Clear*)} and {(*Wind Speed, Medium*), (*Visibility, Cloudy*)}. These two features are discriminated because they have a discriminant input. The first feature contains the input *Visibility* with a value of *Clear* while the second feature requires that the input *Visibility* must have the value of *Cloudy*.

3.2.5. Corollary 3.1. Feature overlap ⇔ No discriminant input

Feature A overlaps feature B if-and-only-if A and B do not have a discriminant input. The overlap relationship is the complement of the discriminated relationship. Therefore, the corollary is true since it is obtained by replacing both sides of the if-and-only-if in « FdiscrimDInputTh» by their complements.

Note in « FeatureOverlap» that the features {(*Wind Direction, Head*)} and {(*Wind Speed, Medium*), (*Visibility, Cloudy*)} do not have a discriminant input.

3.3. Feature Specialization

Let A be a feature. The feature B obtained by adding one or more input/value pairs to the feature A is a *specialization* of A. The feature B is a specialization of A because the subdomain of B is a subset of the subdomain of A. This follows directly from « FsubsetSDsuperTh» and that A is a subset of B since B is obtained by adding more pairs to those already contained in A.

Intuitively, the new input(s) added to the original feature only restrict the feature further. Any point outside the subdomain of the original feature cannot be in the subdomain of the specialization.

A feature that is already completely specific cannot be specialized; no more inputs can be added to the feature.

« FeatureSpecial» shows an example of feature specialization. The feature {(*Wind Speed, Low*)} initially covers the entire top row of the input domain. When the feature is specialized by adding the pair (*Wind Direction, Tail*), the new feature {(*Wind Speed, Low*), (*Wind Direction, Tail*)} only covers the two right points of the top row. The new feature is a specialization of the original feature.

Table 3.7. Feature Specialization

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

3.4. Feature Generalization

Let A be a feature. The feature B obtained by removing one or more input/value pairs from the feature A is a *generalization* of A. The feature B is a generalization of A because the subdomain of B is a superset of the subdomain of A. This follows directly from « FsubsetSDsuperTh» and that B is a subset of A since B is obtained by removing pairs from those contained in A.

Intuitively, inputs removed from the original feature only remove restrictions on the original subdomain. Any points in the subdomain of the original feature are also matched by the generalized feature. There can be no point in the original subdomain that is not matched by the generalized feature.

A feature that is completely general (has no inputs) cannot be generalized; there are no inputs to remove from the feature.

« FeatureGeneral» shows an example of feature generalization. The feature {(*Wind Speed, High*), (*Wind Direction, Tail*)} initially covers the two right points of the bottom row of the input domain. When the feature is generalized by removing the pair (*Wind Speed, High*), the new feature {(*Wind Direction, Tail*)} covers the entire right half of the input domain. The new feature is a generalization of the original feature.

Table 3.8. Feature Generalization

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

3.5. Feature Union

Union is a special case of feature specialization. Given two features A and B, the *union* of A and B is defined by the standard set union operation on the input/value sets of the features. Union is only defined for features that are not discriminated. Let $C = A \cup B$. The feature C exists only because there is no discriminant input for A and B. If a discriminant input i_j does exist for A and B, then feature C would contain both pairs (i_j, x) and (i_j, y) and C could not match any point in the input domain.

3.5.1. Theorem 3.3. Feature union $\Leftrightarrow$ Subdomain intersection

Let A, B, and C be features such that A and B have no discriminant input. Let D_A, D_B, and D_C be the subdomains of A, B, and C respectively. $C = A \cup B$ if-and-only-if $D_C = D_A \cap D_B$. In other words, the subdomain of the union of two features is the intersection of the original two subdomains.

Proof

The proof is by contradiction. There are two cases where the statement $C = A \cup B$ can be contradicted, either C is missing a pair that is contained in $A \cup B$, or C contains an additional pair not contained in $A \cup B$. In either case, $D_C \neq D_A \cap D_B$.

Case 1

Suppose C does not contain a pair $p = (i_j, x)$ that is contained in $A \cup B$. The subdomain D_C then contains points with input i_j set to all values in its domain, but any point with input i_j equal to a value other than x is outside of one of the original subdomains. Since p is in $A \cup B$, p must be contained in one of A or B. Suppose p is contained in A. Then D_A only contains points with input i_j equal to x and not equal to any other value, so the intersection of D_A and D_B cannot contain points with input i_j equal to a value other than x.

Case 2

Suppose C contains an extra pair $p = (i_j, x)$ that is not contained in $A \cup B$. Then D_C contains only points where input $i_j = x$ and no points with input $i_j \neq x$, but points with input $i_j \neq x$ are contained in both D_A and D_B and so should be contained in the intersection.

« FeatureUnion» shows an example of feature union. The feature {(*Wind Speed, Medium*), (*Wind Direction, Head*)} covers the first two points of the middle row. The feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)} covers the second column. The two features combine to form the feature {(*Wind Speed, Medium*), (*Wind Direction, Head*), (*Visibility, Cloudy*)}. The new feature covers the single point in the second row and second column which is the intersection of the original two subdomains.

Table 3.9. Feature Union

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

3.6. Feature Intersection

Intersection is a special case of feature generalization. Given two features A and B, the *intersection* of A and B is defined by the standard set intersection operation on the input/values sets of the features. Unlike union, intersection is defined for any two features.

3.6.1. Theorem 3.4. Feature intersection ⇒ Subdomain union containment

Let A, B, and C be features. Let D_A, D_B, and D_C be the subdomains of A, B, and C respectively. If $C = A \cap B$ then $D_C \supseteq D_A \cup D_B$. In other words, the subdomain of the intersection of two features is a superset of the union of the original two subdomains. The new subdomain completely encompasses the other two.

<u>Proof</u>

Feature C must be a subset of both feature A and feature B because of the definition of set intersection. By « FsubsetSDsuperTh», subdomain D_C is a superset of D_A since $C \subseteq A$. Likewise, D_C is a superset of D_B. Since there are no points in $D_A \cup D_B$ that are not in at least one of D_A or D_B, D_C must be a superset of $D_A \cup D_B$.

« FeatureIntersect» shows an example of feature intersection. The feature {(*Wind Speed, Low*), (*Wind Direction, Tail*), (*Visibility, Cloudy*)} covers the single point at the end of the first row. The feature {(*Wind Speed, High*), (*Wind Direction, Tail*), (*Visibility, Clear*)} covers the single point in the third column of the last row. The two features intersect to form the feature {(*Wind Direction, Tail*)}. The new feature covers the entire right half of the input domain and encompasses the union of the original two subdomains.

Table 3.10. Feature Intersection

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low				
Medium				
High				

3.7. Feature Strength

Let A be a feature. Let v be a point in the output domain. Let t be the total number of examples that match A. Let c_v be the number of examples that match A and have output equal to v. The *strength* of output value v for feature A is $\frac{c_v}{t}$. The strength of a feature is a measure of how well the feature can predict an output value. If a feature A matches no examples (t = 0), then the strength of all outputs for the feature is defined as 0.

For example the feature {(*Visibility, Cloudy*)} shown in « FeatureStrength» has strength 2/3 = .67 for output *Manual* and strength 1/3 = .33 for output *Automatic*. If the feature were changed to {(*Wind Direction, Head*), (*Visibility, Cloudy*)}, the strength would be 2/2 = 1.0 for output *Manual* and 0/2 = 0 for output *Automatic*.

Table 3.11. Feature Strength

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low		Manual	Automatic	
Medium				
High		Manual		Automatic

The *output* of a *feature* is its strongest output value. If the two strongest outputs of a feature have the same strength, the output of the feature is *Don't-Know*. The *strength* of a *feature* is the strength of its strongest output value. For example, the output of feature {(*Visibility, Cloudy*)} shown in « FeatureStrength» is *Manual* since *Manual* is stronger than *Automatic*. The strength of the feature is .67.

Let A and B be features. Let D_A and D_B be the subdomains of A and B, respectively. Let S_A and S_B be the strength of A and B, respectively. Let V_A and V_B be the output of A and B, respectively. Feature A *covers* feature B if $D_A \supseteq D_B$, $S_A \geq S_B$, and $V_A = V_B$. Feature A *contradicts* feature B if $D_A \supseteq D_B$, $S_A \geq S_B$, and $V_A \neq V_B$. In other words, a feature is *covered* if its subdomain is encompassed by the subdomain of a stronger feature with the same output. A feature is *contradicted* if its subdomain is encompassed by the subdomain of a stronger feature with a different output.

Two outputs V_A and V_B are said to be *concordant* if $V_A = V_B$. They are *discordant* if $V_A \neq V_B$. Two examples or features are *concordant* if their outputs are concordant. They are *discordant* if their outputs are discordant.

An example of feature covering is shown in « FeatureCover». The feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)} has output *Manual* with strength 2/3 = .67. The feature {(*Wind Direction, Head*)} covers the first feature since its subdomain is a superset of the subdomain of the first feature and the output *Manual* has strength 3/4 = .75.

Table 3.12. Feature Covering

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low		Manual	Automatic	
Medium	Manual	Automatic		
High		Manual		Automatic

3.8. Sets of Features

Recall that the goal of learning from examples is to find a mapping from an input space to an output space. This mapping can be done by a set of features. This section describes how a set of features can implement such a mapping. Later chapters describe how sets of features are selected or learned.

A set of features can implement a function from input to output. A feature matches a set of points in the input space and a feature has an output. So a feature defines the mapping for the subdomain that it matches. Each point in the subdomain of the feature is associated with the output point of the feature. A set of features defines the mapping for the union of the subdomains of the features. The set of features may cover the complete input space or only part of the input space and so may implement a total function or a partial function.

For example, « FeatureFunction» shows the features {(*Wind Direction, Head*), (*Visibility, Clear*)} with output *Manual*, {(*Wind Speed, Low*), (*Visibility, Cloudy*)} with output *Manual*, and {(*Wind Speed, High*), (*Wind Direction, Tail*)} with output *Automatic*. These features implement a partial function for the aircraft control problem as shown in « FunctionImpl».

Table 3.13. A Function Defined by Features

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low		Manual		
Medium	Manual			
High				Automatic

The function implemented by the features in « FeatureFunction» is partial. It defines 7 out of the 12 possible points in the input domain.

Table 3.14. Function implemented by features in « FeatureFunction»

Input	**Output**
(Low, Head, Clear)	Manual
(Medium, Head, Clear)	Manual
(High, Head, Clear)	Manual
(Low, Head, Cloudy)	Manual
(Low, Tail, Cloudy)	Manual
(High, Tail, Clear)	Automatic
(High, Tail, Cloudy)	Automatic

All the features in a set of features that defines a function may be discriminated. The output of a set of features for a given input point is then simply the output of the feature that matches the input point. The output for input points that are not matched by any feature is *Don't-Know* (undefined). The example above in « FeatureFunction» and « FunctionImpl» shows a set of features that are all discriminated from each other.

More likely is a set of features where some features in the set overlap each other. The output for a given input point is then the output of the strongest feature that matches the input point. If the two strongest features that match an input point have the same strength and different outputs, the output is *Don't-Know*. In this case the system could output one or both of the strongest features.

« FunctionOverlap» shows the features {(*Wind Direction, Head*)} with output *Manual* and strength 2/3, {(*Wind Speed, Low*), (*Visibility, Cloudy*)} with output *Manual* and strength 1, and {(*Wind Speed, High*)} with output *Automatic* and strength 1. The feature {(*Wind Direction, Head*)} matches 6 points in the input domain. However, three of the points are either covered or contradicted by other stronger features. Therefore, the feature only controls the function output for 3 points in the input domain. These features implement the function shown in « FunctionOlImpl».

Table 3.15. A Function Defined by Overlapping Features

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low		Manual		
Medium	Manual			
High		Automatic		Automatic

Table 3.16. Function implemented by features in « FunctionOverlap»

Input	**Output**
(Low, Head, Clear)	Manual
(Medium, Head, Clear)	Manual
(High, Head, Clear)	Automatic
(Low, Head, Cloudy)	Manual
(Medium, Head, Cloudy)	Manual
(High, Head, Cloudy)	Automatic
(Low, Tail, Cloudy)	Manual
(High, Tail, Clear)	Automatic
(High, Tail, Cloudy)	Automatic

A function can be implemented with many specific features or with few general features. A possible set of features is the set that exactly matches the input points defined in the set of training examples. This set of features provides no generalization although it outputs correctly for all training examples. For example, the training set shown in « FunctionOverlap» could be implemented with the features {(*Wind Speed, Low*), (*Wind Direction, Head*), (*Visibility, Cloudy*)}, {(*Wind Speed, Medium*), (*Wind Direction, Head*), (*Visibility, Clear*)}, {(*Wind Speed, High*), (*Wind Direction, Head*), (*Visibility, Cloudy*)}, {(*Wind Speed, High*), (*Wind Direction, Tail*), (*Visibility, Cloudy*)}.

General features with high strength match several training examples where the examples have the same output. Such features also match points in the input space that are not defined by any training example. High strength features should be good predictors of the correct output for such input points. The learning algorithms presented in later chapters try to find small sets of general features. The features shown in « FunctionOverlap» are quite general and have good strength. These features are a better choice from the standpoint of generalization than the very specific features above.

3.9. Networks of Features

A set of features can be implemented using a network of nodes. Each node is responsible for and represents a single feature. Each node contains a set of counters C for measuring feature strength. The set of counters C contains a counter c_o for each point o in the output space. Each node also contains a total counter T. The learning models presented in later chapters have several variations. In some variations the nodes contain additional items which will be discussed in connection with the variation.

All the learning models described in this work are incremental learners. This means that examples are presented one at a time. The system incorporates the example and is then able to execute the functional mapping that has been learned. Counters give an estimate of feature strength based on the examples that have been seen since the node was created.

A typical node is shown « NetworkNode». The node monitors the feature {(*Wind Speed, Low*)}. There is a counter for each of the possible output values and a total counter.

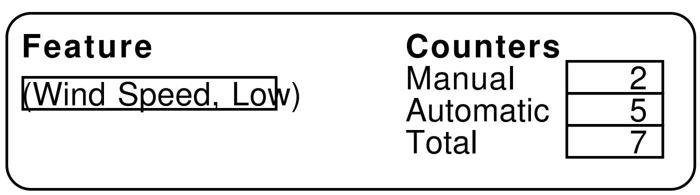

Figure 3.1. Typical network node

The counters in each node allow the node to calculate the strength of each output point based on the input examples that the node has seen. During training with example (i, o), all nodes that match i update their counters as follows:

- Increment the total counter T.
- Increment the counter c_o, associated with the output point o given in the training example.

The *strength* of output point o is $\frac{c_o}{T}$.

« BeforeCntUpdate» shows a node along with the strength of each output. When the example is given to the node the counters are updated and the strengths change as shown in « AfterCntUpdate».

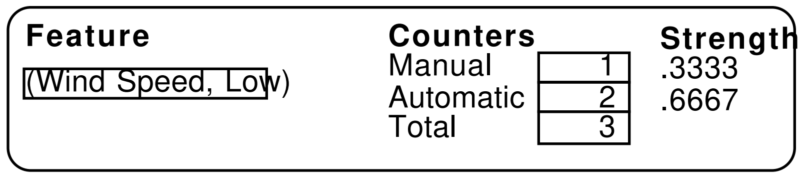

Figure 3.2. Node before counter update

Example: (*Low, Head, Clear*) → *Automatic*

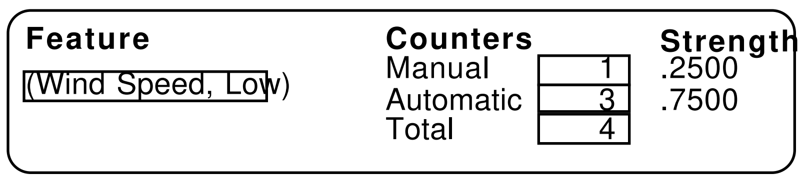

Figure 3.3. Node after counter update

When given input values, a network of nodes executes and produces an output as follows. Each node that matches the input state calculates the strength for each output value. The purpose of each node is to approximate the conditional probability of an output value given that the input values match a particular feature. The nodes compete according to strength and the strongest node wins the competition. The output value of the winning node is asserted as the output value of the network. Thus, the output of a network of nodes is the output of the highest strength node. If the two strongest nodes have the same strength and different outputs, the output of the network is *Don't-Know*.

« NetworkExecution» shows the network execution algorithm. *Broadcast* means that information is given to all nodes. *Gather* means that information is taken from all nodes. The implementation of broadcast and gather on tree architectures is described in detail in Chapter 7.

Network Execution

1. The input values presented to the network inputs are broadcast to all nodes in the network.
2. Each node compares the input values to the feature stored at the node.
3. Nodes where the feature matches the input values:
    4. Calculate the strength of each output value.
    5. Send the strongest output value along with its strength out to the network.
6. End.
7. The network gathers the outputs from each node.
8. The strongest output value is the network output.

End.

Figure 3.4. Network Execution

We now give an example of network execution. Given the network in « NetworkExecExample» and input point (*High, Head, Clear*), Node 1 outputs *Automatic* since *Automatic* has strength .67. Node 2 outputs *Don't-Know* since no output value has a higher strength than the others. Node 3 outputs *Manual* and Node 4 does not match the input values. Node 3 wins the competition and the network output is *Manual*.

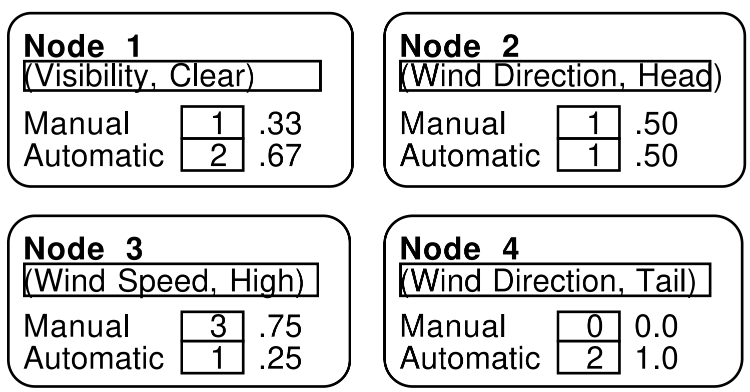

Figure 3.5. Example of network execution

A node is an implementation for the concept of a feature. Thus, the terms strength, output, generality, etc. that apply to a feature also apply to a node. When applied to nodes, these terms refer to the feature contained in the node. Since a network of nodes implements a set of features, the network implements a function from input to output.

Chapter 4
General to Specific Learning

This chapter presents one way our systems learn the features that are important to the problems they are solving. One possible approach to learning features is to create all possible features, then keep strong features and delete weak features. This is not a practical approach because the number of features is exponential in the number of inputs. For example, suppose the network has 16 Boolean input variables. If the network were to create all possible 16th-order features, the number of nodes would be 65,536 or 2^{16}.

Another problem with creating all features is that strong features may exist in the training set that are not needed to actually learn the problem. The system should approximate the minimum possible set of features if it is to provide the best generalization.

General to specific learning (GS) avoids these problems by first creating general features. The general features guide the creation of more specific features. A combination of generality and strength is used as a metric to measure the goodness of a feature.

A simplified example is given to illustrate the GS approach. Detailed explanation of general to specific learning is given in subsequent sections.

A network for the aircraft control problem is shown in « LandNetwork1». Node 1 monitors the input *Wind Speed* and matches when the value is *High*. Node 1 predicts output *Automatic* with higher strength than output *Manual*. Node 3 matches the feature {(*Visibility, Cloudy*)} which has the same strength for both outputs. The network in « LandNetwork1» is the result of training with several examples. Two additional examples are presented to the system.

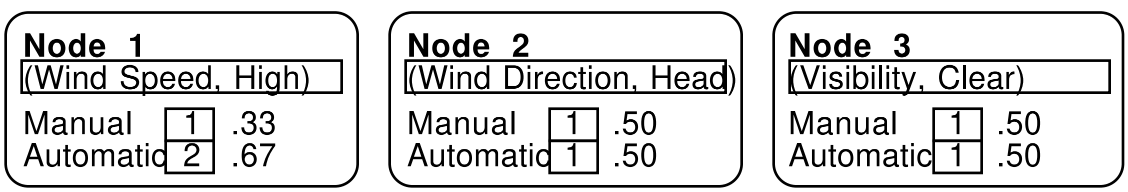

Figure 4.1. Network for the aircraft control problem

Example 1: (*High, Tail, Clear*) → *Automatic*

When a new example is presented, all possible first-order features that match the example are created. Since the example above is matched by the feature {(*Wind Direction, Tail*)} that does not have an existing node, a new node is created. The network nodes compete according to strength to assert their strongest output value. Node 1 wins the competition and outputs *Automatic*. The network output agrees with the example output, so no additional nodes are created. The nodes that match the input example update their counters giving the network in « LandNetwork2».

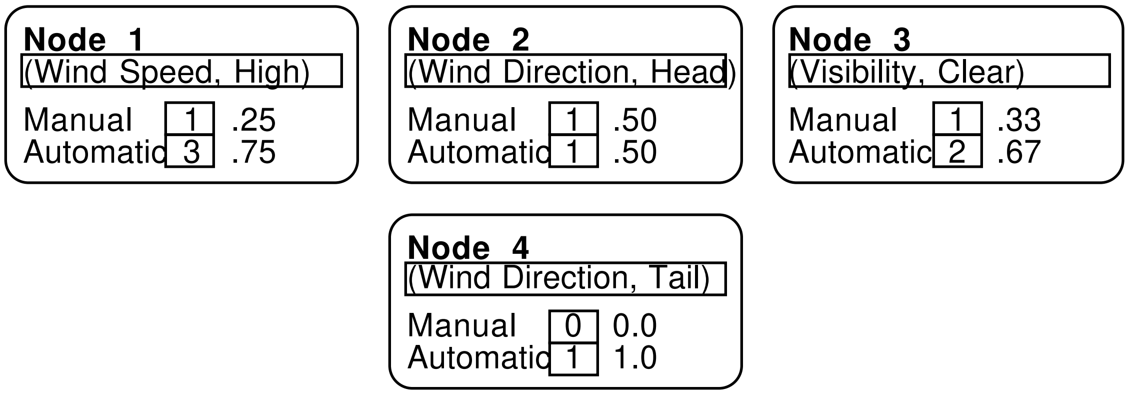

Figure 4.2. Network after the first example

Example 2: (*Low, Head, Clear*) → *Manual*

The next example is presented to the network and Node 3 wins the competition and outputs *Automatic*. The output is incorrect, so not only is the new first-order node with feature {(*Wind Speed, Low*)} created, but a new second-order node is created. The two nodes with highest strength for the output value *Manual* (Nodes 2 and 3) combine to create a new node with the feature {(*Wind Direction, Head*), (*Visibility, Clear*)}. Nodes matching the example update their counters giving the network in « LandNetwork3». In the examples given in this paper, connections are typically shown between nodes that have a general to specific relationship. Thus, Node 6 is connected to Nodes 2 and 3 since Node 6 is a specialization of the other two nodes.

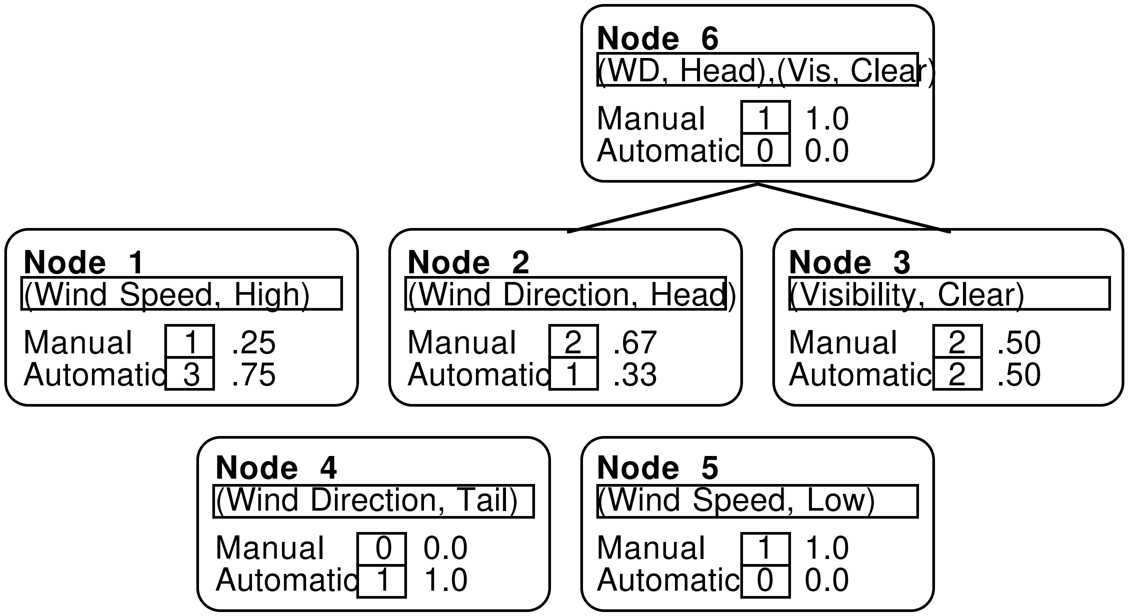

Figure 4.3. Network after the second example

4.1. General to Specific Learning Algorithms

Five different GS learning algorithms are presented in this chapter. The five algorithms differ only in the way they create new specific features. This section describes the main routine that is shared by all five algorithms. The main routine is executed for each example in the training set. Because nodes are created incrementally and nodes count outputs for matching examples, more than one presentation of a training example may be needed to accurately count outputs. Thus, typically a training set is presented to the system at least two or more times.

Before any examples are given, GS begins with the single general feature {} that covers the complete input space. The initial feature is contained in a node called the *root*. The counters of the root node are all set to 0. As examples are given to the system, inputs are added to the root feature to create more specific features. New specific features are selected based on a combination of generality and strength. The best new features are used to create new nodes.

A GS system repeatedly executes the learning algorithm shown in « GSLearnCode» for each example that is given to the system. When a training example is presented during learning, the network executes (Lines 1 to 2) to obtain the network output for the input given in the example. Later, a specific feature is created only if the output of the network is different from the output of the example.

After execution, the network can optionally create *single-input nodes* (Lines 3 to 6). Single-input nodes have only a single input/value pair in their feature. Single-input nodes are required for the *Feature-union* and *Feature-specialization* methods. The other methods can optionally create single-input features. Creation of single-input nodes effectively speeds up the specialization process by starting the process with one level of specialization already completed.

When single-input nodes are created a new feature is created for each input in the training example. For example, the example (*Low, Head, Clear*) $\rightarrow$ *Manual* causes the creation of features {(*Wind Speed, Low*)}, {(*Wind Direction, Head*)}, and {(*Visibility, Clear*)}. Each node is only created if an equivalent node does not already exist. The counters in new single-input nodes are initialized to zero.

If the network output is not equal to the example output, then a specific node is created using existing nodes. Otherwise, no additional nodes are added to the system. In other words, if an example is already handled by a simple node, a more complex node is not created to accomplish the same result. A variation on this test is that if the network output is *Don't-Know* then no additional nodes are added to the system. The reasoning for this variation is that if the output is *Don't-Know,* incrementing counters should be sufficient to correct the network output and additional nodes are not needed.

A specific node can be created using *feature-union, feature-specialization,* or *feature-specialization-with-strength-estimation* (Line 8). The three approaches are not mixed but actually give different learning algorithms. Feature-specialization-with-strength-estimation has three variations, giving a total of five different GS learning methods. Feature-union and feature-specialization use the strength of existing features to select good new features. Feature-specialization-with-strength-estimation uses an improved approach that estimates the strength of new features rather than using the strength of existing features.

After the training example is learned, nodes in the network that are not needed delete themselves. Deletion is described in the next section.

General to Specific Learning

1. Broadcast the training example (i, o) to all nodes in the network.
2. Gather the strongest node output giving the network output r.
3. If single-input nodes are to be created, then for each input value x_j in i.
    4. Create a new feature $F = \{(i_j, x_j)\}$.
    5. If a node with feature F does not already exist then create a new node with feature F. Initialize counters in the new node to zero.
6. End.
7. Each node that matches i increments its counters as described previously.
8. If the network output is not correct ($r \neq o$) then create a specific node by one of
    Feature-union
    Feature-specialization
    Feature-specialization-with-strength-estimation
9. Delete nodes that are not needed.

End.

Figure 4.4. General to Specific Learning

4.2. Deletion

This section describes how nodes that are not needed are deleted. All the learning algorithms described in this work use the same deletion methods. Thus, deletion is explained once in this section but is used in the algorithms that are described in other sections.

Because nodes are created incrementally, many times a node will be created that is not essential for learning the training set. If nonessential nodes can be identified, they can be deleted from the system. Measurement of node usage and comparison of node probabilities can both be used to identify nonessential nodes.

4.2.1. Measuring Node Usage

Nodes measure their usage by keeping a use counter. The use counter is zeroed at a regular interval (e.g., after each pass over the training set) and incremented whenever a node wins in execution. Nodes that have zero counters at the next interval for setting all counters to zero delete themselves.

The network execution algorithm is slightly enhanced to improve deletion based on usage. When two nodes tie in the execution strength competition, the node that is most general is made the winner. Thus the more general nodes are favored to survive while more specific nodes (that match input points handled properly by general nodes) delete themselves.

4.2.2. Comparing Node Strength

A node that is covered by another node can never change the output function. For every input point that the covered node can respond to, the covering node can also respond with at least as high a strength. A node can determine if it is covered by another node by comparing its feature and strength with the feature and strength of other nodes. The comparison can be done either as part of the learning algorithm or as a background process.

Comparison is done in the learning process immediately after the network produces an output. The feature and strength of the node that wins the execution are broadcast to the network. Nodes that are covered by the broadcast feature delete themselves.

More formally, let F_w and S_w be the feature and strength of the winning node W, respectively. Let F_n and S_n be the feature and strength of a node N receiving the broadcast, respectively. If $F_w \subseteq F_n$ and $S_w > S_n$, then node N deletes itself.

In the background process each node periodically broadcasts its feature and strength. The other nodes in the network then respond in the same way as when strength comparison is done in the learning algorithm. A complete round of background deletion is $O(m \log m)$, where m is the number of nodes. Each of the m nodes does a broadcast to the other nodes and each broadcast is $O(\log m)$ on a tree architecture.

4.3. Feature Union

One method for creating specific features is to combine existing features using the union operation. Recall from the last chapter that the union of two features covers a subdomain of the input space that is the intersection of the subdomains covered by the original features. Thus the feature created by such a union is more specific as desired for general to specific learning.

An important goal in learning is to converge on a solution quickly. Therefore, the system should not simply combine any two features at random. The feature combinations should be guided to new features that will have high strength. It is not practical to combine all existing features since the number of new nodes would grow exponentially. Therefore, existing features combine based on strength and generality.

Feature generality and strength are combined using a linear combination, $M = Gc + S(1-c)$, where G is generality, S is strength for the output given in the example, and c is a constant between 0 and 1. The result M is used as a metric to select the best existing features for combination. Empirical tests suggest that the constant c does not require careful adjustment. Values from .5 to .9 give consistently good results.

Existing nodes that match the example are sorted according to M and the two best nodes are combined. When two nodes A and B combine to create a new node C, the feature for C is the union of the features of A and B. Note that if the two original features contain the same input, that input will have the same value in both features because both features must match the training example. Thus the requirement that the two features not be discriminated for the union operation to be defined is always met.

A new candidate feature may already exist in a node in the network. If the feature already exists, the best node marks itself as not combinable and the next best two nodes attempt to combine. When a unique feature is found, a new node is created.

Single-input nodes must be created so that features exist to begin combining. Otherwise, the only node in the network would be the root which would have no other node to combine with. The feature-union algorithm is shown in « FeatureUnion».

Feature Union

1. For all nodes N.
    2. If node N matches the example and N is not the root then
        3. Node N marks itself as available for combination.
        4. Node N calculates the metric $M = Gc + S(1-c)$, where G is the generality of N, S is the strength of N for output o, and c is a constant between 0 and 1.
    5. Else
        6. Node N marks itself as unavailable.
    7. End.
8. End.
9. Gather M from the marked nodes returning the node A with highest M.
10. Node A marks itself as unavailable.
11. Repeat until either a node is created or all nodes are marked unavailable.
    12. Gather M from the marked nodes returning the node B with highest M.
    13. Node B marks itself as unavailable.
    14. Create a new feature $F = F_A \cup F_B$, where F_A and F_B are the features of node A and B, respectively.
    15. If a node with feature F does not already exist then
        16. Create a new node N with feature F.
        17. Initialize counters T and c_o in N to 1, where o is the output of the example.
        18. Initialize all other counters in N to 0.
    19. End.
    20. Set A to B.
21. End.

End.

Figure 4.5. Creating higher-order features through Feature Union

4.2.1. Training Example

We now give an example of general to specific learning using feature-union. The network starts with only the root node as shown in « UnionNetwork1». Connections are shown between nodes that have a general to specific relationship.

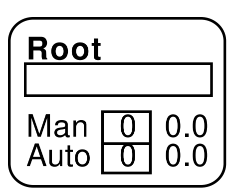

Figure 4.6. Initial Network

Example 1: (*Low, Tail, Cloudy*) → *Automatic*

Example 1 is given to the network. The network executes with the input (*Low, Tail, Cloudy*). The root node matches the example but the strength of both outputs for the root node is 0. Therefore, the output of the root node is *Don't-Know*. Since there are no other nodes in the network, the output of the network is the output of the root or *Don't-Know*.

Single-input nodes must be created for feature-union. The three features {(*Wind Speed, Low*)}, {(*Wind Direction, Tail*)}, and {(*Visibility, Cloudy*)} are created from the example. No existing nodes contain any of these features, so three new nodes are created. The root node and the new single-input nodes update their counters giving the result shown in « UnionNetwork2».

Empirical tests have shown that feature-union gives better results when a new specific node is not created when the network output is *Don't-Know*. Therefore, the feature-union algorithm is not executed and the network has completed incorporating the example.

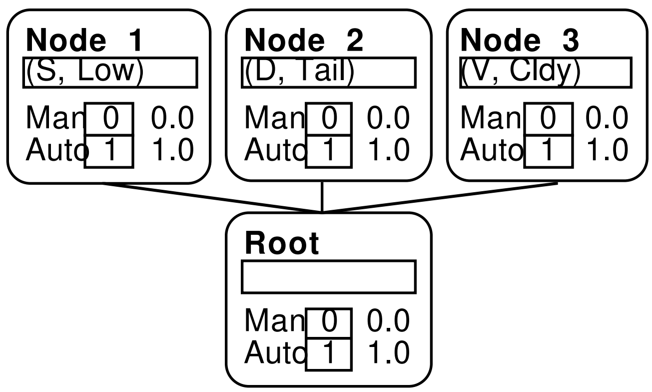

Figure 4.7. Network after first example

Example 2: (*High, Head, Clear*) → *Manual*

Example 2 is given to the network. The network executes with the input (*High, Head, Clear*). The root node is the only node that matches the example. The output of the root node is *Automatic*. Since there are no other matching nodes, the output of the network is the output of the root.

Three single-input features {(*Wind Speed, High*)}, {(*Wind Direction, Head*)}, and {(*Visibility, Clear*)} are created from the example. No existing nodes contain any of these features so three new nodes are created. The root node and the new single-input nodes update their counters giving the result shown in « UnionNetwork3».

The output of the network is different from the output of the example, so feature-union is executed. The three single-input nodes just created mark themselves as available for combination. The root node does not mark itself as available even though it matches because the union of the root and any other node is the same as the other node. The three new nodes are all equal in generality and strength. Two of the new nodes are arbitrarily selected for combination. Assume Nodes 4 and 5 in « UnionNetwork3» are selected. The union of the features in these nodes is {(*Wind Speed, High*), (*Wind Direction, Head*)}.

The new feature is not contained in any existing node, so Node 7 is created and its counters are set according to the training example.

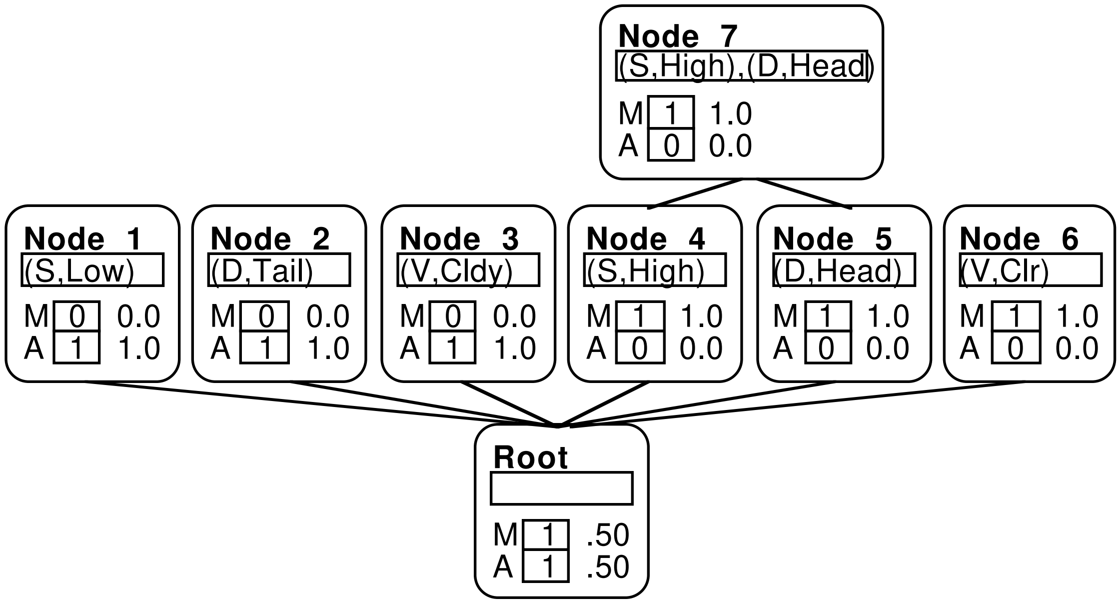

Figure 4.8. Network after second example

Example 3: (*High, Tail, Clear*) → *Automatic*

Example 3 is given to the network. The network executes with the input (*High, Tail, Clear*). The root and nodes 2, 4, and 6 match the example. Nodes 2, 4, and 6 are stronger than the root and of equal strength. Node 2 outputs different than Nodes 4 and 6 so the output of the network is *Don't-Know*.

The single-input features created from Example 3 already exist in Nodes 2, 4, and 6, so no new single-input nodes are created. The root and Nodes 2, 4, and 6 update their counters. Since the output of the network is *Don't-Know*, feature-union is not done.

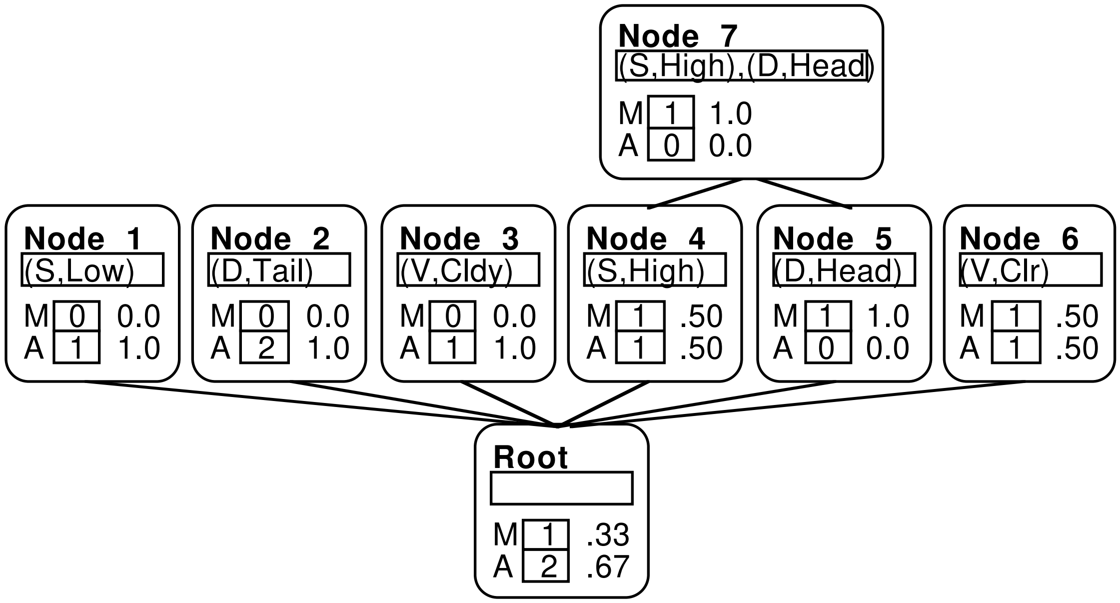

Figure 4.9. Network after third example

Example 4: (*High, Head, Cloudy*) → *Automatic*

Example 4 is given to the network. The network executes with the input (*High, Head, Cloudy*). The root and nodes 3, 4, 5, and 7 match the example. Nodes 3, 5, and 7 are stronger than the other nodes and of equal strength. Node 3 outputs different than Nodes 5 and 7 so the output of the network is *Don't-Know*.

The single-input features created from Example 4 already exist in Nodes 3, 4, and 5, so no new single-input nodes are created. The root and Nodes 3, 4, 5, and 7 update their counters. Since the output of the network is *Don't-Know*, feature-union is not done.

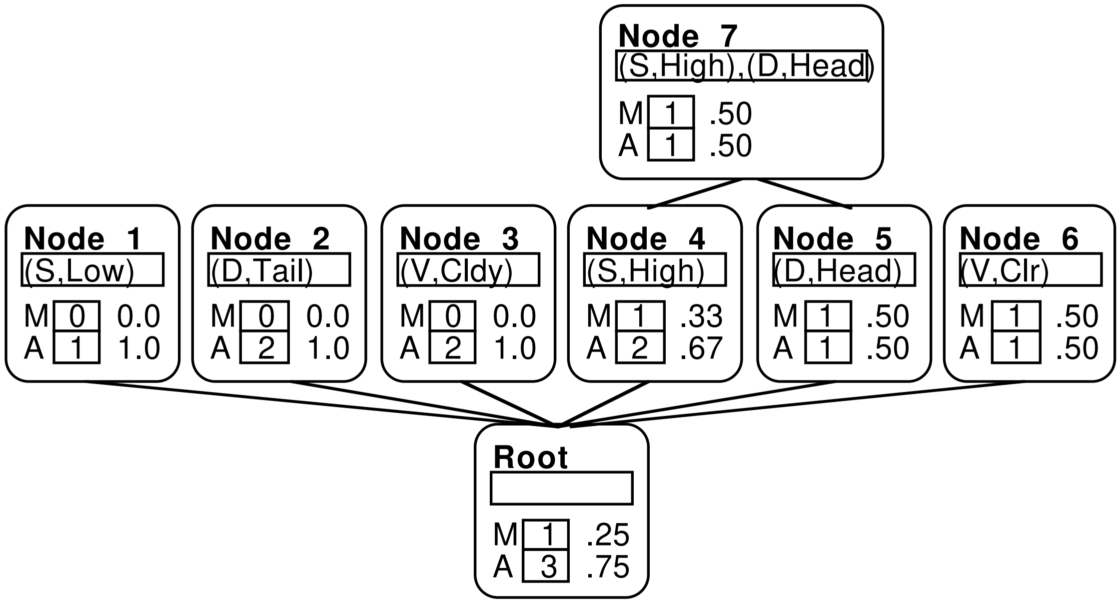

Figure 4.10. Network after fourth example

Example 5: (*Low, Head, Clear*) → *Manual*

Example 5 is given to the network. The network executes with the input (*Low, Head, Clear*). The root and nodes 1, 5, and 6 match the example. Node 1 is stronger than the other nodes so the network output is *Automatic*.

The single-input features created from Example 5 already exist in Nodes 1, 5, and 6, so no new single-input nodes are created. The root and Nodes 1, 5, and 6 update their counters to the values shown in « UnionNetwork6».

The output of the network is different from the output of the example, so feature-union is executed. Nodes 1, 5, and 6 mark themselves as available for combination. Assume that simple generality is used to measure generality and that the constant *c* is set to .5. The generality, strength, and metric for the three nodes (using counts in « UnionNetwork6») are as follows:

Node	**Generality**	**Strength**	**Metric**
1	2/3 = .67	1/2 = .50	.67 × .5 + .50 × .5 = .58
5	2/3 = .67	2/3 = .67	.67 × .5 + .67 × .5 = .67
6	2/3 = .67	2/3 = .67	.67 × .5 + .67 × .5 = .67

Nodes 5 and 6 have a higher metric so they are selected for combination. The union of the features in these nodes is {(*Wind Direction, Head*), (*Visibility, Clear*)}. The new feature is not contained in any existing node, so Node 8 is created and its counters are set according to the training example.

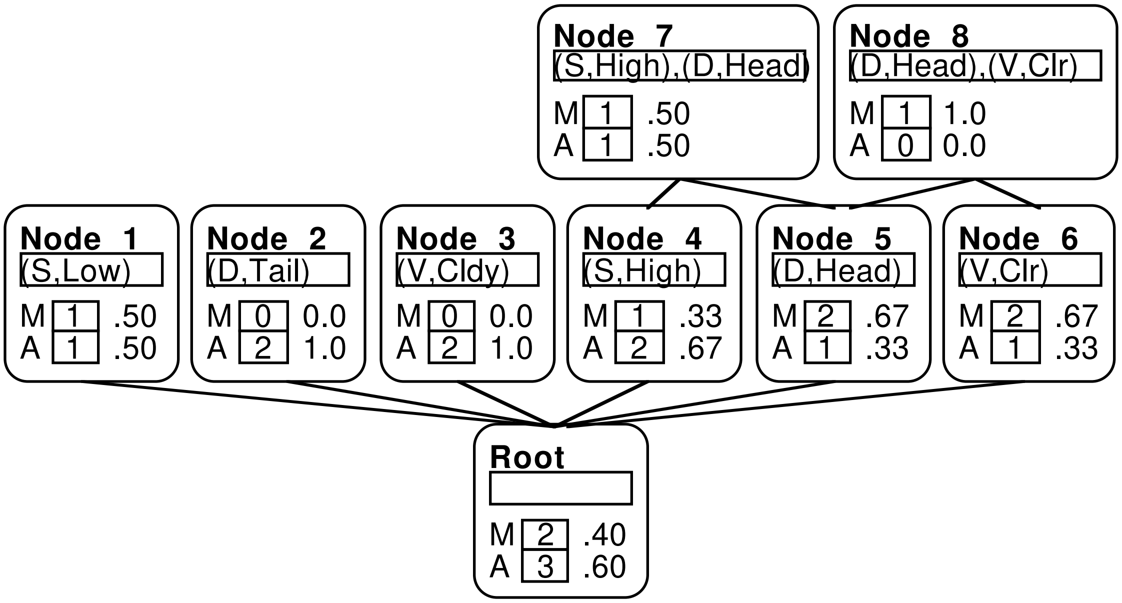

Figure 4.11. Network after last example

The function implemented by the network in « UnionNetwork6» is shown in « UnionFunction». Note that not all the nodes in the network are needed to implement the function. For example, Node 1 can never win a competition for output since it is a specialization of the root and the root is stronger. Node 7 is a specialization of Node 5 and Node 5 is stronger. These and other nodes can be removed by deletion.

Table 4.1. Function implemented by network in « UnionNetwork6»

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low	Manual	Automatic	Automatic	Automatic
Medium	Manual	Automatic	Automatic	Automatic
High	Manual	Automatic	Automatic	Automatic

The network shown in « NetworkDeletion» is the result of deleting unused nodes after a second pass over the training set with the network in « UnionNetwork6». The function implemented by this network is identical to that shown in « UnionFunction». Note that the root node is never deleted even though in this case it never wins in execution.

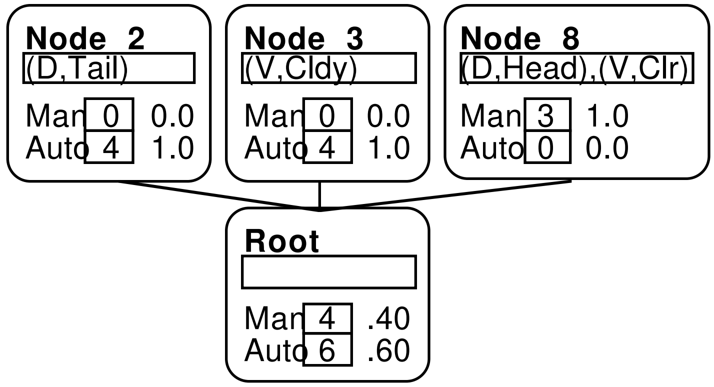

Figure 4.12. Network after deletion

Feature-union begins with the root node and single-input nodes which are very general. These general nodes gather statistics from the training examples. Strong general features combine to form more specific features. Not all combinations of general features are considered since the number of combinations is combinatoric. Combinations are restricted to adjacent nodes in an ordering by generality and strength. An alternative would be to consider combining the strongest node with all other nodes. Because nodes are selected for combination based on the strength of general features, feature-union works best when strong specific features are made up of strong general features. If a problem has strong specific features that are not composed of strong general features, feature-union may create many specific features in error before finding good specific features. Combining adjacent nodes in a strength ordering allows feature-union to eventually combine two weak general features. If the alternative method is used where the strongest general feature is combined with all other features, two weak features are never combined. This weakness of feature-union is overcome by feature-specialization-with-strength-estimation described in later sections.

4.4. Feature Specialization

Another method for creating specific features from existing nodes is for existing nodes individually to specialize themselves by adding an input/value pair to their feature. Thus nodes are able to operate more autonomously than in feature-union. Chapter 3 showed how adding a pair to a feature creates a new feature whose subdomain is a subset of the original feature's subdomain.

It is not practical for all existing nodes to specialize and create new features. If each existing node created a specialization for each training example, the number of nodes in the network would double for each example. Thus, the size of the network would grow at an exponential rate. On the other hand, the system should not allow random nodes to specialize. Therefore, just as in union, nodes are selected for specialization by using a goodness metric, $M = Gc + S(1-c)$.

Existing nodes that match the training example are sorted according to M and the best node is selected. The selected node then adds a pair to its current feature to create a new feature. The pairs are selected in order based on the strength of the corresponding single-input features as given by the single-input nodes. Single-input nodes must be created to provide this strength information.

If the new feature does not already exist, then a node with the new feature is created. Otherwise the process is repeated adding the next strongest pair to the node's original feature. If the selected node cannot create any unique new feature by adding a single pair, the next best node is selected.

The process repeats until either a unique feature is found or all nodes have been selected.

The feature-specialization algorithm is shown in « FeatureSpecial».

Feature Specialization

1. For all nodes N.
   2. If node N matches the example and N is not the root then
      3. Node N marks itself as available for specialization.
      4. Node N calculates the metric $M = Gc + S(1-c)$, where G is the generality of N, S is the strength of N for output o, and c is a constant between 0 and 1.
   5. Else
      6. Node N marks itself as unavailable.
   7. End.
8. End.
9. Repeat until either a node is created or all nodes are marked unavailable.
   10. Gather M from the marked nodes returning the node A with highest M.
   11. Node A marks itself as unavailable.
   12. Repeat until either a node is created or all pairs have been selected.
      13. Let $P_j = (i_j, x_j)$ be the pair defined by input value x_j in i. Let N_j be the node containing the single-input feature $F_j = \{(i_j, x_j)\}$. Let S_j be the strength of N_j for output o.
      14. Select the pair P_j that has not already been selected and whose feature F_j has highest S_j according to node N_j.
      15. Create a new feature F by adding the pair P_j to the feature F_A, where F_A is the feature of node A.
      16. If a node with feature F does not already exist then
         17. Create a new node N with feature F.
         18. Initialize counters T and c_o in N to 1, where o is the output of the example.
         19. Initialize all other counters in N to 0.
      20. End.
   21. End.
22. End.

End.

Figure 4.13. Creating higher-order features through Feature Specialization

A detailed example of feature-specialization is not given since for simple examples feature-specialization is essentially identical to feature-union. When existing features are all first-order, feature-union and specialization are identical. When existing features are higher-order, feature-union can combine two higher-order features and create a very specific feature quickly. Feature-specialization requires more steps to create a very specific feature. However, feature-specialization has the advantage of gathering strength information for the intermediate features and therefore feature-specialization may be guided to better specific features. Also, feature-specialization has the advantage of allowing nodes to work more independently, thus making it possible to add strength estimation techniques to specialization as described in the next section.

4.5. Specialization with Strength Estimation of New Features

This section presents the three remaining GS learning models. The three models share a common specialization routine that is called from the main GS routine when the network gives an output that is different from the output of the training example. The specialization routine creates new candidate features by adding inputs to nodes that match the example. The strength of the new features is estimated using one of three methods; next-input-counter strength estimation, limited-example-memory strength estimation, or example-node strength estimation. The new feature with highest generality and strength is selected to create a new node. The specialization routine is described in this section while the strength estimation methods are described in three subsections.

The feature-specialization routine is shown in « FeatureSpecSE». All nodes (in parallel) that match the training example create specializations of the current feature by adding a single additional input/value pair (Lines 1 to 3). For every input i_j *not* contained in the node's current feature, a new feature is created by adding the pair (i_j, x_j) to the current feature, where x_j is the value of input i_j from the training example. The new candidate features are stored in the node's local memory while strength estimation is done. Only a single new feature is selected to create a new node.

Each new candidate feature is compared with features already existing in the network. If a feature the same as a new candidate feature does not already exist, then the strength of the new feature is calculated using one of three strength estimation methods (Lines 4 to 6). The strength estimation methods are not mixed. All nodes over all training examples use a single method. Thus each estimation method results in a separate and distinct approach to GS learning.

If the strongest new feature is stronger than the original feature, the strongest new feature is saved in the node and the node optionally repeats the specialization process using the new feature (Lines 7 to 10). (If specialization cannot create a feature stronger than the original, there is no reason to create a new feature.) Specialization can be repeated as long as:

1. It is possible to specialize the current best new feature (the feature does not already contain all inputs).
2. A new feature is created that is stronger than the feature created on the previous iteration.

Repetition is not reasonable for next-input-counter estimation since this method only keeps information about a single additional input/value pair. No information is available about adding additional pairs. However, after a new specific node is created, the new node can gather statistics about features more specific than itself and such features can be created on subsequent training examples or on later iterations over the training set. Thus, feature-specialization can effectively be repeated but it requires more steps.

Since general features are preferred over specific features, a metric is calculated that combines generality and strength of the best new feature. As in feature-union and feature-specialization, the metric is a linear combination of generality and strength, given by $M = Gc + S(1-c)$. The important difference here is that S is the estimated strength of the new feature rather than the strength of the parent node.

The matching nodes compete according to the metric and the winning node is selected to create a new node using its best feature (Lines 13 to 15). The counters in the new node are initialized differently depending on the method of strength estimation that is used. Details are given in the subsections that follow.

Feature Specialization with Strength Estimation

1. For all nodes N that match the example
    2. Set F to the existing feature of the node N.
    3. For each input i_j not in F create a new feature F_j by adding the pair (i_j, x_j) to F, where x_j is the value of input i_j in the training example.
    4. For each new feature F_j.
        5. If a node with feature F_j does not already exist then calculate the strength of F_j using either *Next-Input Counters*, *Limited Example Memory*, or *Example Nodes*.
    6. End.
    7. If at least one new feature exists that is stronger than F.
        8. Set F to the new feature with highest strength and mark N.
        9. Optionally repeat lines 3 to 10 using the new feature F.
    10. End.
    11. Calculate the metric $M = Gc + S(1-c)$, where G is the generality of F, S is the strength of F for output o, and c is a constant between 0 and 1.
12. End.
13. Gather the new feature F with highest M from the marked nodes.
14. Create a new node NN with feature F.
15. Initialize counters in NN according to the strength estimation method.

End.

Figure 4.14. Feature Specialization with Strength Estimation

4.5.1. Next-Input Counter (NIC) Strength Estimation

The Next-Input Counter (NIC) method for strength estimation requires that each node have a set of counters for each possible input/value pair in addition to the standard set of counters in a node. These counters monitor the strength of features obtained by adding an additional pair to the current feature. Thus a node can estimate the strength of a new specialization of its current feature.

The number of possible input/value pairs in the system is $p = |X_1| + |X_2| + ... + |X_n|$, where X_j is the domain for input i_j and n is the number of inputs in the system. In other words, input 1 has $|X_1|$ possible input/value pairs, one for each value of the input. The possible pairs for each input are summed over all the inputs. The number of counters needed for each pair is $q = |O| + 1$, where O is the output domain as given in Chapter 2. There is one counter for each point in the output space and the total counter. This is the same as the number of standard counters in a node. The total number of counters for all pairs is thus pq.

« NICNode» shows a node that contains next-input counters. The *Wind Speed* input has three values so there are three sets of counters for that input. The other two inputs each have two values and thus two sets of counters. Each set (column) of counters consists of two counters for the two possible output points and the total counter.

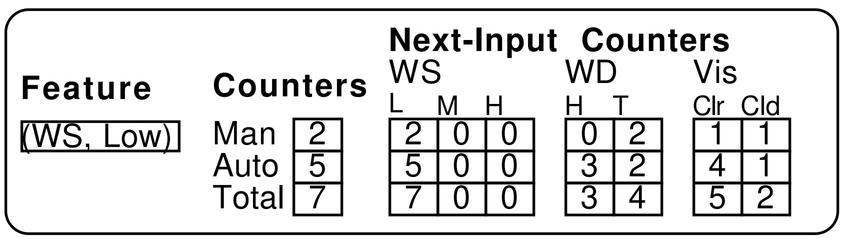

Figure 4.15. Network node with Next-Input Counters

Each pair has a set of counters just like the standard set of counters. The standard counters are updated if the node matches the training example. The counters associated with a pair (i, x) have this same condition for update as well as an additional condition; input i in the training example must be equal to x.

The additional counters simulate p additional nodes. Each simulated node effectively contains the feature obtained by adding the pair associated with the set of counters to the real feature of the node. The simulated nodes give the system the ability to look ahead into the specialization process.

The counters for a single input/value pair are updated just like the standard counters. The counter c_o corresponding to the output o given in the training example is incremented and the total counter is incremented.

« NICUpdate» shows how the counters in the node in « NICNode» are updated when given the training example (*Low, Head, Clear*) → *Automatic*. The counters for *Wind Speed* are updated exactly the same as the main counters since the node's feature contains the *Wind Speed* input. The *Wind Direction* counter for *Head* and *Automatic* is incremented since the example contains those two values for *Wind Direction* and the output, respectively. Similarly, the *Visibility* counter for *Clear* and *Automatic* is incremented. Note that the counters in each row for a single input must sum to the value given in the main counter for that row.

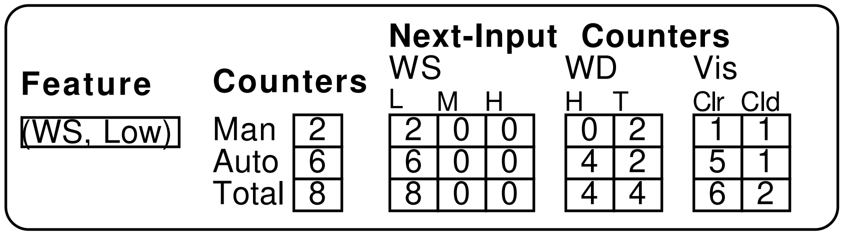

Figure 4.16. Next-Input Counter Update

Let F be a new feature created by adding the pair P to the node's feature. The estimated strength for F using this method is given by c_o/T, where c_o is the counter associated with pair P for the output o given in the training example and T is the total counter associated with pair P.

A new node created using this method is initialized as follows. Let P be the pair added to the parent node's feature to create the new node. The counters for pair P in the parent node are copied into the standard counters for the child node. The next-input counters in the child node are initialized to 0 and then set according to the training example.

« NICInit» shows how a new node created from the node in « NICUpdate» is initialized. The new node adds the input/value pair (*Wind Direction, Head*) to the feature in « NICUpdate». Therefore the counters for that pair are copied into the main counters for the new node. All other counters are set to 0.

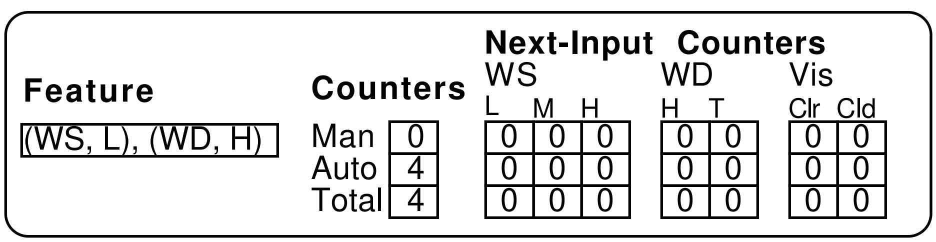

Figure 4.17. Node initialization with Next-Input Counters

4.5.1.1. Training Example

We now give an example of learning using NIC specialization. The network starts with only the root node as shown in « NICNetwork1». For simplicity, single-input nodes are not used in this example, although empirical results suggest that using single-input nodes typically gives better results for GS learning. Connections are shown between nodes that have a general to specific relationship. Strengths are shown after the main counters in each node. Due to space limitations, strengths are not shown for next-input counters.

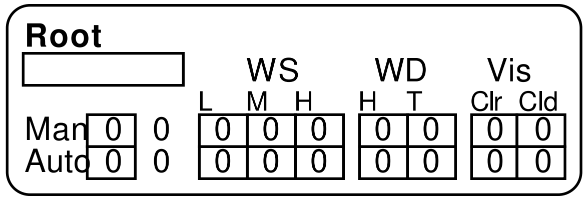

Figure 4.18. Initial Network

Example 1: (*Low, Tail, Cloudy*) → *Automatic*

Example 1 is given to the network. The network executes with the input (*Low, Tail, Cloudy*). The root node matches the example but the strength of both outputs for the root node is 0. Therefore, the output of the root node is *Don't-Know*. Since there are no other nodes in the network, the output of the network is the output of the root or *Don't-Know*.

The root node updates its counters giving the resulting root node shown in « NICNetwork2». Since the output of the network is *Don't-Know*, feature-specialization is not done and no additional nodes are created.

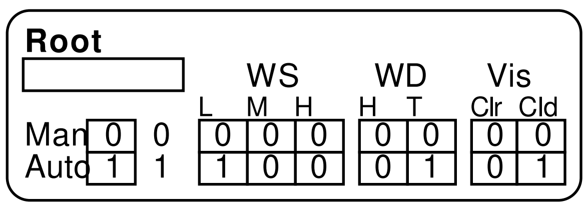

Figure 4.19. Network after first example

Example 2: (*High, Head, Clear*) → *Manual*

Example 2 is given to the network. The network executes with the input (*High, Head, Clear*). Again the root node matches the example. The output of the root node is *Automatic*. Since there are no other nodes, the output of the network is the output of the root.

The root node updates its counters giving the resulting root node shown in « NICNetwork3». The output of the network is different from the output of the example, so NIC specialization is executed.

The root node creates three new features by adding each of the three inputs in the training example to its current feature. The features and their strength for output *Manual* are as follows. (The strengths are calculated from the next-input counters of the root node in « NICNetwork3»).

{(*Wind Speed, High*)}	1/1 = 1.0
{(*Wind Direction, Head*)}	1/1 = 1.0
{(*Visibility, Clear*)}	1/1 = 1.0

Since all three features have equal strength, one is arbitrarily selected. Assume the feature {(*Wind Direction, Head*)} is selected and used to create Node 1 in « NICNetwork3». The counters in Node 1 are copied from the next-input counters in the root in the column under (*Wind Direction, Head*). The next-input counters in Node 1 are initialized to 0 and then incremented according to the training example.

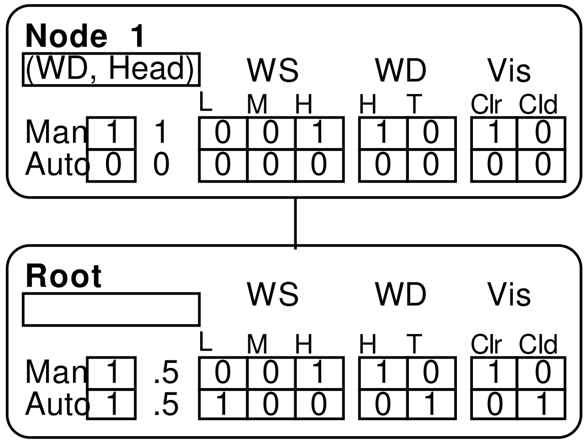

Figure 4.20. Network after second example

Example 3: (*High, Tail, Clear*) → *Automatic*

Example 3 is given to the network. The network executes with the input (*High, Tail, Clear*). The root is the only node that matches the example. The output of the root is *Don't-Know*. Since no other nodes match the example, the output of the network is the output of the root.

The root node updates its counters giving the resulting root node shown in « NICNetwork4». Since the output of the network is *Don't-Know*, feature-specialization is not done and no additional nodes are created.

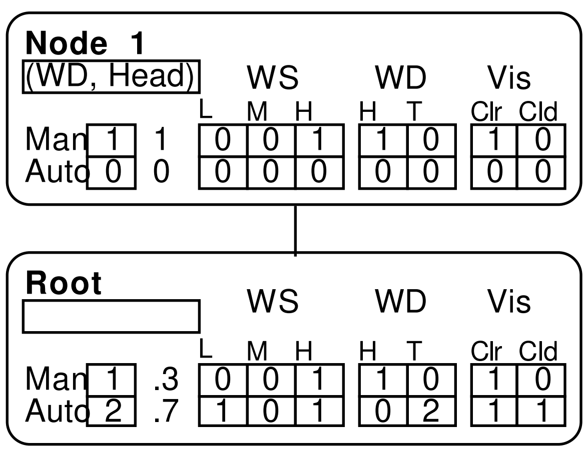

Figure 4.21. Network after third example

Example 4: (*Low, Head, Clear*) → *Manual*

Example 4 is given to the network. The network executes with the input (*Low, Head, Clear*). The root and Node 1 both match the example. The root outputs *Automatic* and Node 1 outputs *Manual*. Node 1 is stronger than the root, so the output of the network is *Manual*.

The root and Node 1 both update their counters giving the result shown in « NICNetwork5». Since the output of the network is the same as the example, feature-specialization is not done.

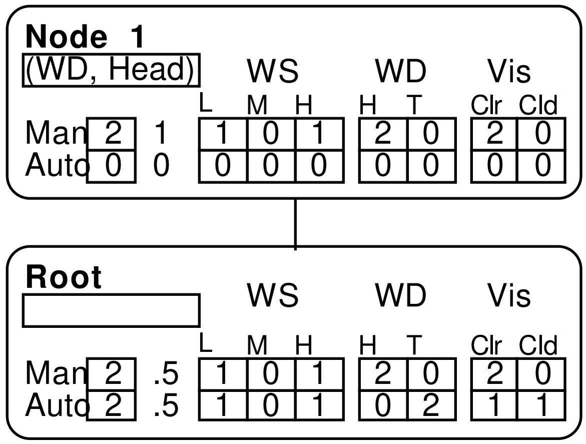

Figure 4.22. Network after fourth example

Example 5: (*High, Head, Cloudy*) → *Automatic*

Example 5 is given to the network. The network executes with the input (*High, Head, Cloudy*). The root and Node 1 both match the example. Node 1 is stronger than the root so the network output is *Manual*.

The root and Node 1 update their counters to the values shown in « NICNetwork6». The output of the network is different from the output of the example, so feature-specialization is executed.

The root node creates three candidate features with strengths for output *Automatic* as follows. (The strengths are given by the next-input-counters of the root node in « NICNetwork6»).

{(*Wind Speed, High*)}	2/3 = .67
{(*Wind Direction, Head*)}	1/3 = .33
{(*Visibility, Cloudy*)}	2/2 = 1.0

The feature {(*Visibility, Cloudy*)} is strongest, so it is the feature the root node will use to compete with Node 1 to obtain the right to create a new node.

Node 1 creates two new features with strengths for output *Automatic* as follows.

{(*Wind Speed, High*), (*Wind Direction, Head*)}	1/2 = .50
{(*Wind Direction, Head*), (*Visibility, Cloudy*)}	1/1 = 1.0

The feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)} is strongest, so it is the feature Node 1 will use to compete with the root.

Assume that simple generality is used to measure generality and that the constant *c* is set to .5. The generality, strength, and metric for the best new features of the two nodes are as follows.

Node	**Generality**	**Strength**	**Metric**
Root	2/3 = .67	1.0	$.67 \times .5 + 1 \times .5 = .83$
Node 1	1/3 = .33	1.0	$.33 \times .5 + 1 \times .5 = .67$

The root node has a higher metric, so it wins the competition and creates Node 2 with the feature {(*Visibility, Cloudy*)}. The counters in Node 2 are copied from the next-input counters in the root in the column under (*Visibility, Cloudy*). The next-input counters in Node 2 are initialized to 0 and then incremented according to the training example.

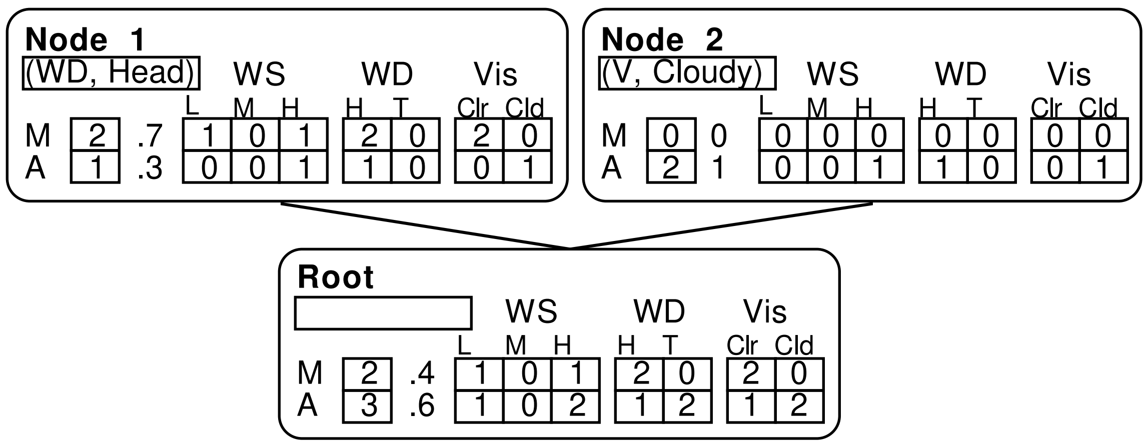

Figure 4.23. Network after last example

The function implemented by the network in « NICNetwork6» is shown in « NICFunction». The function is the same as that implemented by the network in « UnionNetwork6» using feature-union, although the networks that implement the functions are different. NIC specialization was able to learn the function with fewer nodes. In this case no deletion is needed. NIC specialization keeps more information about new features than feature-union, so it is able to learn a problem more efficiently.

Table 4.2. Function implemented by network in « NICNetwork6»

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low	Manual	Automatic	Automatic	Automatic
Medium	Manual	Automatic	Automatic	Automatic
High	Manual	Automatic	Automatic	Automatic

4.5.2. Limited Example Memory (LEM) Strength Estimation

The Limited Example Memory (LEM) method of strength estimation requires that each node have a table with space to store examples. Let *k* be the number of examples that can be stored in the table. The number *k* is always a constant and all nodes have the same value for *k*. Typical sizes for *k* that we have empirically tested are from 5 to 100.

« LEMNode» shows a node that contains example memory. The node has space to store a total of three examples. Two examples have been given that match the node.

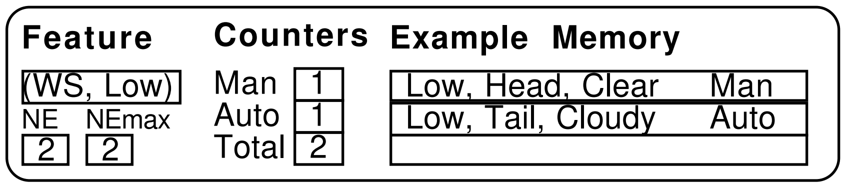

Figure 4.24. Network node with Example Memory

When an example is given during training, all nodes that match the example store the example in their example table. The example is stored in the next available space in the table. If no space is available, the new example replaces the oldest example in the table. The next available space in the table is kept in the variable *NE* (Next Example). The maximum value of the variable *NE* is kept in *NEmax*. The following lines for storing examples are inserted in the main GS learning algorithm just before Line 7.

Example Storage

1. If *N* matches *i*
    2. Store the example at location *NE* in the example table.
    3. Increment *NE* (the next example pointer).
    4. If *NE* > *NEmax* then set *NEmax* to *NE*.
    5. If *NE* ≥ *k* then set *NE* to 0. (the end of the table has been reached so start over at the beginning of the table)
6. End.

End.

« LEMUpdate» shows the result of giving the example (*Low, Tail, Clear*) → *Automatic* to the node in « LEMNode». If the node is given the example (*High, Head, Clear*) → *Manual*, there is no change to the node since the node does not match the example.

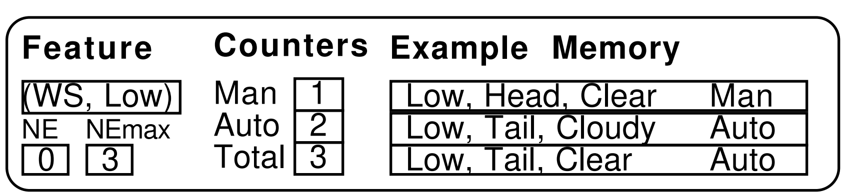

Figure 4.25. Updating a node with Example Memory

« LEMFull» shows the result of adding the example (*Low, Head, Cloudy*) $\rightarrow$ *Automatic* to the already full example table of « LEMUpdate». The new example overwrites the contents of the first location in the table since that location contained the oldest example.

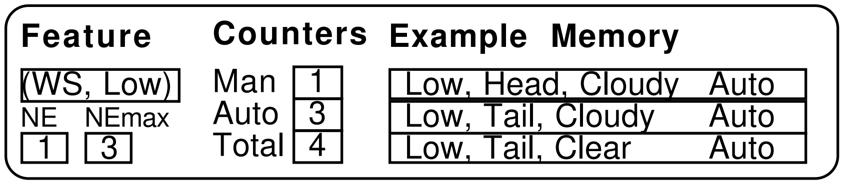

Figure 4.26. Adding an Example to a Full Table

Let F be a new feature created by specializing a node's feature. The estimated strength for F using the LEM method is calculated by counting the examples stored in the node that are matched by F. Let $e_n = (i_n, o_n)$ be an example stored in the node. Let c and t be temporary counters initialized to zero. If F matches e_n then the counter t is incremented. If F matches e_n and the output o_n is equal to the output o of the current training example, then the counter c is incremented. After counting all the stored examples in this manner, the strength of F is given by c/t.

LEM Strength Estimation (Input: *N* (parent node), *F* (new feature);
Output: *S* (estimated strength of *F*))

1. Set c and t to 0.
2. For each example $e_n = (i_n, o_n)$ stored in node N.
    3. If F matches i_n
        4. Increment t (total counter).
        5. If $o_n = o$ then
            6. Increment c (counter for output o).
        7. End.
    8. End.
9. End.
10. Return the strength of F as c/t.

End.

For example suppose the node in « LEMUpdate» creates the new feature {(*Wind Speed, Low*), (*Wind Direction, Tail*)}. Two of the examples stored in the node match the new feature, so the t counter is set to 2. Both examples that match the new feature have output *Automatic,* so the c counter is also set to 2 and the estimated strength of the new feature is 2/2 = 1.

A new node created using this method is initialized as follows. All counters in the new node are initialized to 0. Let $e_n = (i_n, o_n)$ be an example stored in the parent node. If the new node matches the example e_n, then the example is stored in the next available space in the new node. There are at most k examples stored in the parent node so no more than k examples will be stored in the new node. Thus there is no need to handle the case of examples replacing other examples in the new node. As each example is stored in the new node the total counter T is incremented. The counter c_{on} that is associated with the output o_n given in the stored example is also incremented.

LEM Node Initialization (Input: *P* (parent node), *N* (new node))

1. Set NE and $NEmax$ to 0.
2. Set all counters in the new node N to 0.
3. For each example $e_n = (i_n, o_n)$ stored in node P.
    4. If N matches i_n
        5. Increment T (total counter).
        6. Increment c_{on} (counter for output o_n).
        7. Store e_n at location NE in the example table.
        8. Increment NE (the next example pointer).
        9. If $NE > NEmax$ then set $NEmax$ to NE.
    10. End.
11. End.

End.

« LEMInit» shows the initialization of a new node created from the node in « LEMUpdate». The two examples from the original node that match the new node are copied into the new node. The counters are updated according to the examples that match the new node.

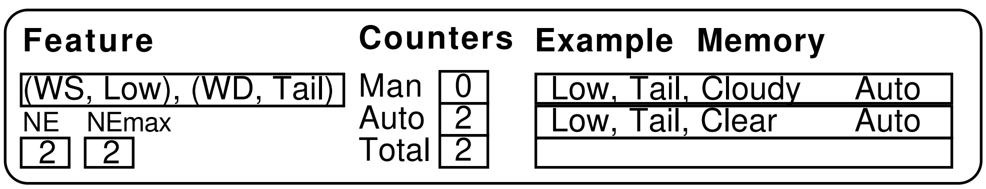

Figure 4.27. Node initialization with Example Memory

<u>4.5.2.1. Training Example</u>

We now give an example of learning using LEM specialization. The network starts with only the root node as shown in « LEMNetwork1». For simplicity, single-input nodes are not used in this example. Connections are shown between nodes that have a general to specific relationship. Strengths are shown after the counters in each node.

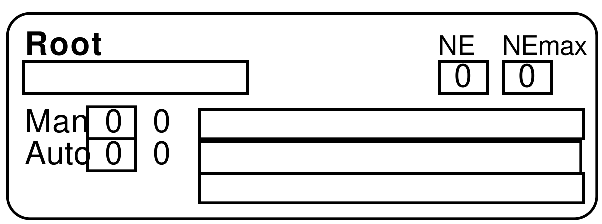

Figure 4.28. Initial Network

Example 1: (*Low, Tail, Cloudy*) → *Automatic*

Example 1 is given to the network. The network executes with the input (*Low, Tail, Cloudy*). The root node matches the example, but the strength of both outputs for the root node is 0. Therefore, the output of the root node is *Don't-Know*. Since there are no other nodes in the network, the output of the network is the output of the root or *Don't-Know*.

The root node updates its counters and stores the training example giving the resulting root node shown in « LEMNetwork2». Since the output of the network is *Don't-Know*, feature-specialization is not done and no additional nodes are created.

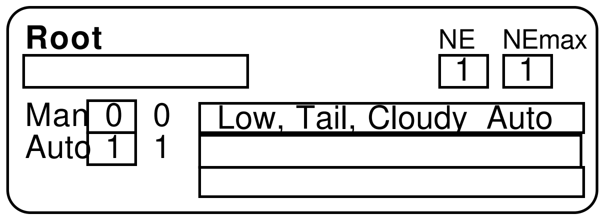

Figure 4.29. Network after first example

Example 2: (*High, Tail, Clear*) → *Automatic*

Example 2 is given to the network. The network executes with the input (*High, Tail, Clear*). Again the root node matches the example. The output of the root node is *Automatic*. Since there are no other nodes, the output of the network is the output of the root.

The root node updates its counters and stores the example giving the resulting root node shown in « LEMNetwork3». The output of the network is the same as the example, so feature-specialization is not done and no additional nodes are created.

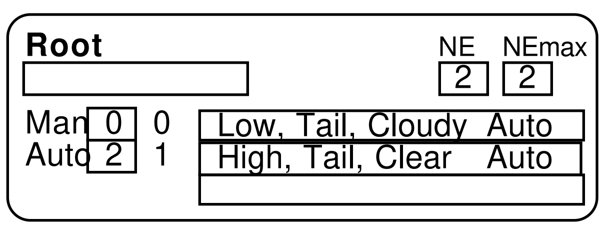

Figure 4.30. Network after second example

Example 3: (*High, Head, Clear*) → *Manual*

Example 3 is given to the network. The network executes with the input (*High, Head, Clear*). The root node matches the example. The output of the root and thus the output of the network is *Automatic*.

The root node updates its counters and stores the example giving the resulting root node shown in « LEMNetwork4». The output of the network is different from the output of the example, so LEM specialization is executed.

The root node creates three new features by adding each of the three inputs in the training example to its current feature. The features and their strength for output *Manual* are as follows. (The strengths are calculated from the examples stored in the root node in « LEMNetwork4»).

{(*Wind Speed, High*)}	1/2 = .50
{(*Wind Direction, Head*)}	1/1 = 1.0
{(*Visibility, Clear*)}	1/2 = .50

Since the feature {(*Wind Direction, Head*)} is strongest, it is selected and used to create Node 1 in « LEMNetwork4». The third example stored in the root node matches the new feature so it is copied into the new node. The counters in Node 1 are set according to the newly stored example.

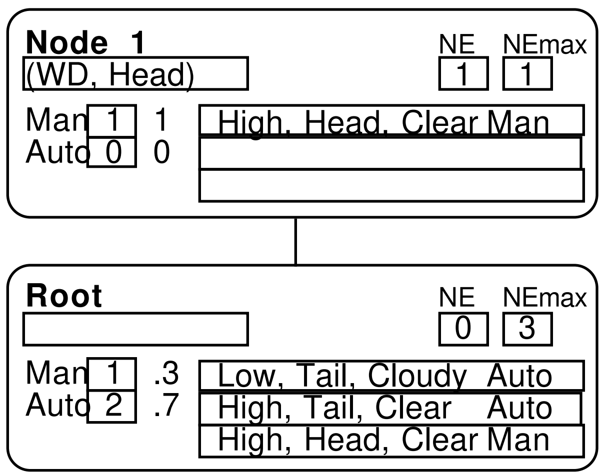

Figure 4.31. Network after third example

Example 4: (*Low, Head, Clear*) → *Manual*

Example 4 is given to the network. The network executes with the input (*Low, Head, Clear*). The root and Node 1 both match the example. The root outputs *Automatic* and Node 1 outputs *Manual*. The strength of Node 1 is 1/1 and the strength of the root is 2/3. Node 1 is stronger than the root, so the output of the network is *Manual*.

The root and Node 1 both update their counters and store the example giving the result shown in « LEMNetwork5». Note that the example is stored in the first position of the root node, overwriting the previous contents. Since the output of the network is the same as the example, feature-specialization is not done.

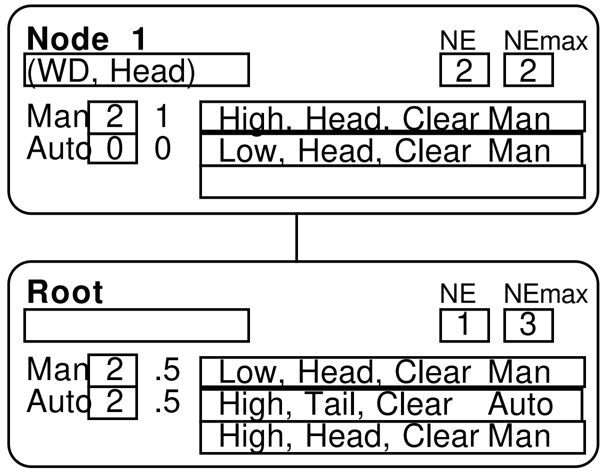

Figure 4.32. Network after fourth example

Example 5: (*High, Head, Cloudy*) → *Automatic*

Example 5 is given to the network. The network executes with the input (*High, Head, Cloudy*). The root and Node 1 both match the example. Node 1 is stronger than the root so the network output is *Manual*.

The root and Node 1 update their counters to the values shown in « LEMNetwork6». The example is stored in the last available position in Node 1 and overwrites the previous contents of the second position in the root node. The output of the network is different from the output of the example, so feature-specialization is executed.

The root node creates three candidate features with strengths for output *Automatic* as follows. (The strengths are calculated from the examples in the root node in « LEMNetwork6»).

{(*Wind Speed, High*)}	1/2 = .50
{(*Wind Direction, Head*)}	1/3 = .33
{(*Visibility, Cloudy*)}	1/1 = 1.0

The feature {(*Visibility, Cloudy*)} is strongest, so it is the feature the root node will use to compete with Node 1 to obtain the right to create a new node.

Node 1 creates two new features with strengths for output *Automatic* as follows. (The strengths are calculated from the examples in Node 1 in « LEMNetwork6»).

{(*Wind Speed, High*), (*Wind Direction, Head*)}	1/2 = .50
{(*Wind Direction, Head*), (*Visibility, Cloudy*)}	1/1 = 1.0

The feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)} is strongest, so it is the feature Node 1 will use to compete with the root.

Assume that simple generality is used to measure generality and that the constant *c* is set to .5. The generality, strength, and metric for the best new features of the two nodes are as follows.

Node	**Generality**	**Strength**	**Metric**
Root	2/3 = .67	1.0	$.67 \times .5 + 1 \times .5 = .83$
Node 1	1/3 = .33	1.0	$.33 \times .5 + 1 \times .5 = .67$

The root node has a higher metric, so it wins the competition and creates Node 2 with the feature {(*Visibility, Cloudy*)}. The example (*High, Head, Cloudy*) → *Automatic* is copied from the root into the new node. The counters in the new node are initialized to 0 and then the counter for *Automatic* is incremented since the only stored example has output *Automatic*.

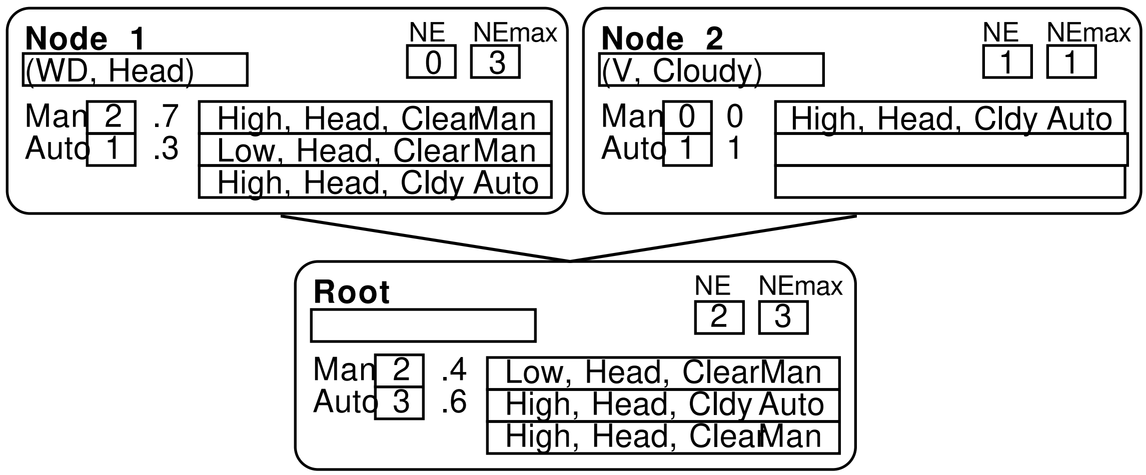

Figure 4.33. Network after last example

The final network created by LEM estimation is identical to that created by NIC estimation. The functions implemented by the two networks are also identical. As in the NIC estimation example, no deletion is needed. With more complex problems, LEM estimation can give different results than NIC estimation since the memory in LEM is fixed and some matching examples may be dropped. For example, the root node in the NIC example has complete strength information for all single-input features. Thus, NIC would correctly estimate the strength of the feature {(*Wind Speed, Low*)} to be .5 for both outputs. The root node in the LEM example had to drop two of its stored examples for lack of space. Thus, LEM would incorrectly estimate the strength of the feature {(*Wind Speed, Low*)} to be 1 for output *Manual* and 0 for output *Automatic*. Therefore, for problems that do not fit within the fixed limits of LEM, NIC is expected to be more accurate. On the other hand, when LEM can store enough examples, it is able to predict strength farther into the specialization process than NIC. LEM can predict the strength of a new feature created by adding two or even more pairs to a current feature. Thus, LEM should allow the system to find a solution more quickly than NIC. Also, the amount of memory used by LEM at each node can be adjusted to fit the resources available to the system. NIC, on the other hand, requires a specific amount of memory for its counters that is determined by the input and output domains of the problem.

4.5.3. Example Node (EN) Strength Estimation

Example Node (EN) strength estimation allows complete information to be stored about examples without the fixed limitation on the number of examples found in LEM estimation. The EN method of strength estimation requires no additional storage at a node. However, it does require that *example nodes* are created. The example nodes provide complete strength information about any new feature. LEM estimation could provide the same result, if the number of examples stored in a node is as large as the number of examples in the training set. However, using LEM in this way would typically require a large amount of memory in every node in the system. EN estimation does not suffer from this drawback.

An *example node* is a node whose feature is fully specific and defined directly by the example. For instance, the example (*Low, Head, Clear*) → *Manual* causes the creation of feature {(*Wind Speed, Low*), (*Wind Direction, Head*), (*Visibility, Clear*)}. The node is only created if an equivalent node does not already exist. Example nodes do not participate in network execution during training. They do participate in normal execution. The following lines for creating example nodes are inserted in the main GS algorithm just before Line 7.

Example Node Creation

1. Create a new feature $F = \{(i_1, x_1), ..., (i_n, x_n)\}$, where $x_1, ..., x_n$ are the values given in *i.*
2. If a node with feature *F* does not already exist, then create a new node with feature *F*. Initialize counters in the new node to zero.

End.

Let *F* be a new feature created by specializing a node's feature. The estimated strength for *F* using the EN method is calculated by summing the counters in the example nodes that are matched by *F*. Let *N* be an example node existing in the network. Let *c* and *t* be temporary counters initialized to zero. If *F* matches *N* then the *T* counter from *N* is added to the counter *t* and the c_o counter from *N* is added to *c*. After summing all the example node counters in this manner, the strength of *F* is given by *c*/*t*.

EN Strength Estimation (Input: *F* (new feature); Output: *S* (estimated strength of *F*))

1. Set *c* and *t* to 0.
2. For all nodes *N*.
    3. If *N* is an example node and *F* matches *N*
        4. Add *T* from node *N* to *t*.
        5. Add c_o from node *N* to *c*.
    6. End.
7. End.
8. Return the strength of *F* as *c*/*t*.

End.

For example, the root node in « ENCalc» could create a new feature {(*Wind Speed, High*)}. This feature matches all three example nodes shown in « ENCalc». Two of the example nodes have a count of 1 for output *Automatic,* so the strength of the new feature would be 2/3 for output *Automatic*. The feature has strength 1/3 for output *Manual*. Similarly, the feature {(*Wind Direction, Head*)} has strength 1/2 for either output. The feature {(*Visibility, Cloudy*)} has strength 2/2 for output *Automatic* and 0/2 for output *Manual*.

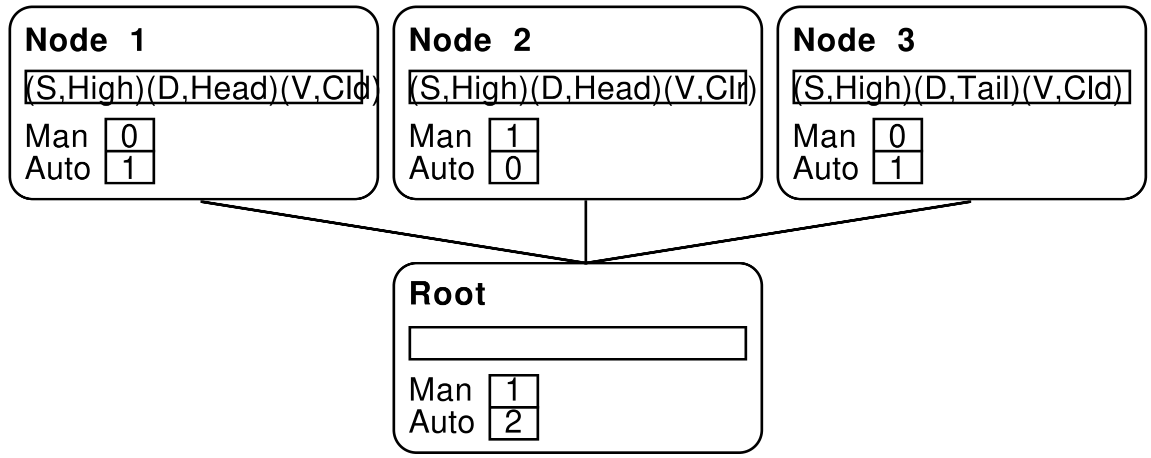

Figure 4.34. Strength Calculation using Example Nodes

The EN strength estimation algorithm can be implemented using either broadcast and gather or a general to specific interconnect topology. The general to specific topology reduces network traffic compared to broadcast and gather. Both implementations are described in Chapter 7.

A new node created using this method is initialized as follows. All counters in the new node are initialized to 0. Let N be an example node existing in the network. If the new node matches the node N, then the total counter T from node N is added to the total counter T for the new node. Also, for each output point o, the counter c_o from node N is added to counter c_o for the new node.

EN Node Initialization (*NN* (new node)).

1. Set all counters in the new node *NN* to 0.
2. For all nodes *N*.
    3. If *N* is an example node and *NN* matches *N*
        4. Add T from node N to T for node *NN*.
        5. For all output points o.
            6. Add c_o from node N to c_o for node *NN*.
        7. End.
    8. End.
9. End.

End.

An example of new node initialization is shown in « ENInit». Node 4 is added to the network shown in « ENCalc». Nodes 1 and 3 match the new feature in Node 4 so the counters for nodes 1 and 3 are summed into the new node's counters.

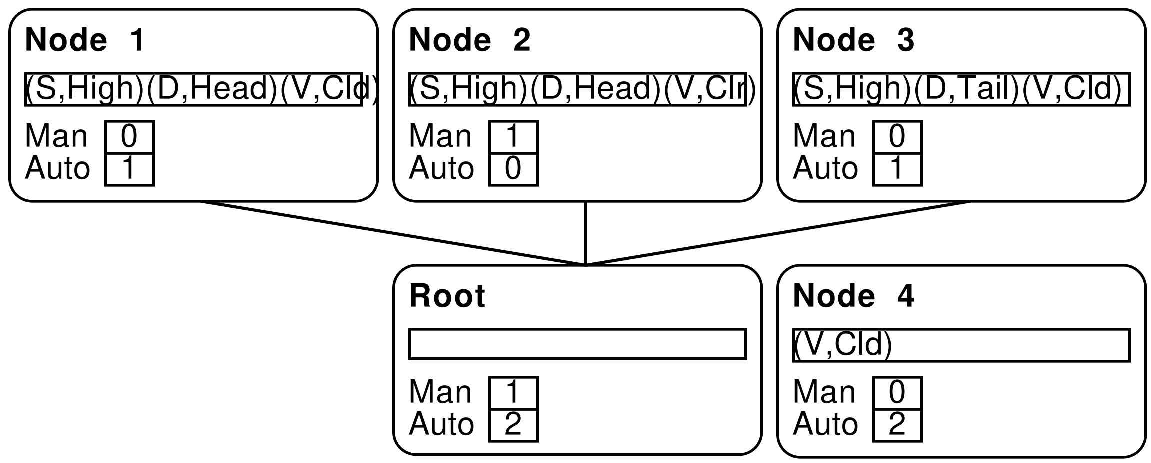

Figure 4.35. Node Initialization using Example Nodes

4.5.3.1. Training Example

We now give an example of learning using EN specialization. The network starts with only the root node as shown in « ENNetwork1». For simplicity, single-input nodes are not used in this example. Connections are shown between nodes that have a general to specific relationship. Strengths are shown after the counters in each node.

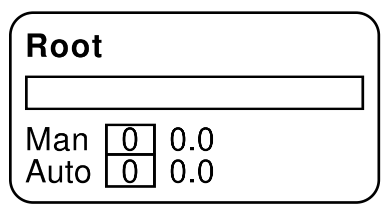

Figure 4.36. Initial Network

Example 1: (*Low, Tail, Cloudy*) → *Automatic*

Example 1 is given to the network. The network executes with the input (*Low, Tail, Cloudy*). The root node matches the example, but the strength of both outputs for the root node is 0. Therefore, the output of the root node is *Don't-Know*. Since there are no other nodes in the network, the output of the network is the output of the root or *Don't-Know*.

Node 1 is created as an example node with a feature that matches only the single input point given in the example. The root node and Node 1 update their counters giving the result shown in « ENNetwork2». Since the output of the network is *Don't-Know*, feature-specialization is not done and no additional nodes are created.

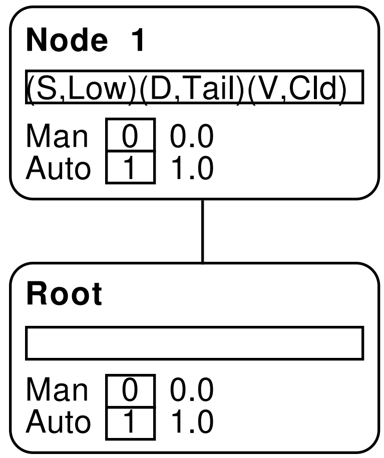

Figure 4.37. Network after first example

Example 2: (*High, Tail, Clear*) → *Automatic*

Example 2 is given to the network. The network executes with the input (*High, Tail, Clear*). The root node matches the example while Node 1 does not match. The output of the root node is *Automatic*. Since there are no other matching nodes, the output of the network is the output of the root.

An example node does not already exist for the training example, so Node 2 is created. The root node and Node 2 update their counters giving the result shown in « ENNetwork3». The output of the network is the same as the example, so feature-specialization is not done and no additional nodes are created.

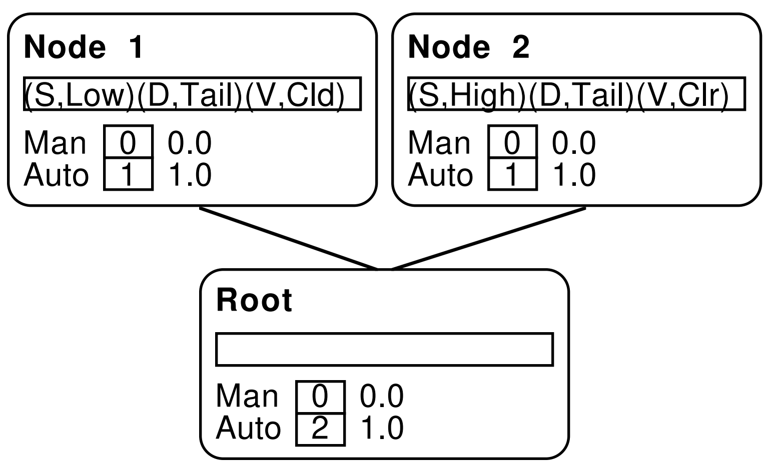

Figure 4.38. Network after second example

Example 3: (*High, Head, Clear*) → *Manual*

Example 3 is given to the network. The network executes with the input (*High, Head, Clear*). Again only the root node matches the example. The output of the root and thus the output of the network is *Automatic*.

Node 3 is created for the example. The root node and Node 3 update their counters giving the result shown in « ENNetwork4». The output of the network is different from the output of the example, so EN specialization is executed.

The three example nodes are already fully specific, so no additional inputs can be added to their features. They do not participate in the specialization process.

The root node creates three new features by adding each of the three inputs in the training example to its current feature. The features and their strength for output *Manual* are as follows. (The strengths are calculated from the example nodes in « ENNetwork4»).

{(*Wind Speed, High*)}	1/2 = .50
{(*Wind Direction, Head*)}	1/1 = 1.0
{(*Visibility, Clear*)}	1/2 = .50

Since the feature {(*Wind Direction, Head*)} is strongest, it is selected and used to create Node 4 in « ENNetwork4». The counters in Node 4 are set according to the single matching example node (Node 3).

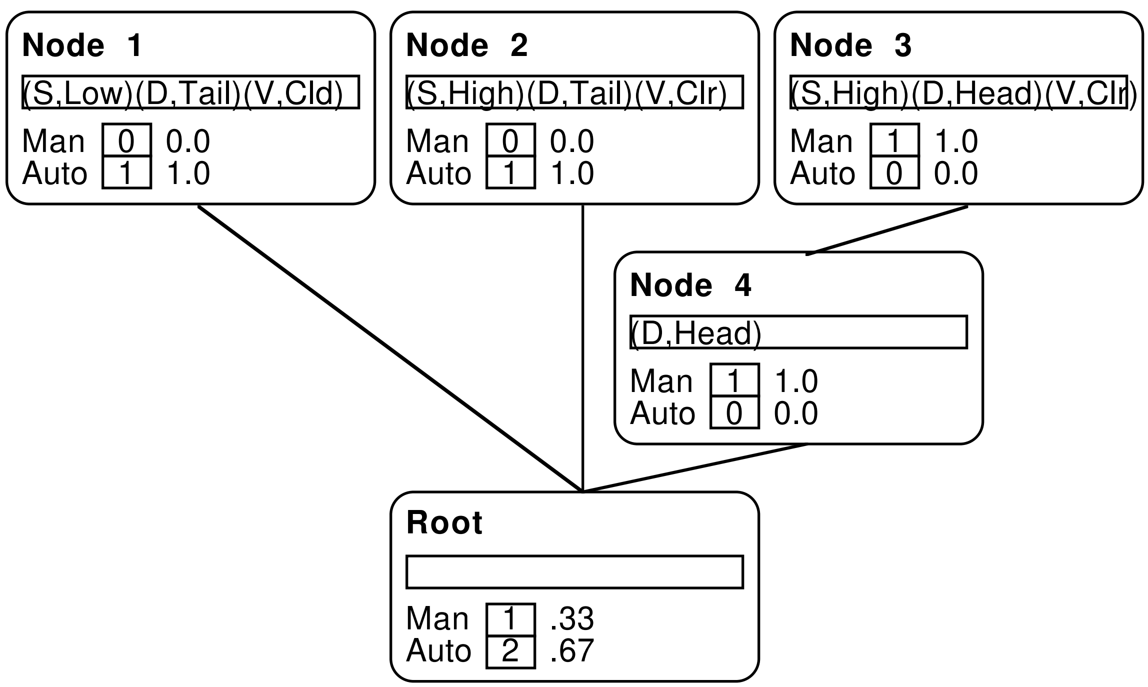

Figure 4.39. Network after third example

Example 4: (*Low, Head, Clear*) → *Manual*

Example 4 is given to the network. The network executes with the input (*Low, Head, Clear*). The root and Node 4 both match the example. The root outputs *Automatic* and Node 4 outputs *Manual*. The strength of Node 4 is 1/1 and the strength of the root is 2/3. Node 4 is stronger than the root, so the output of the network is *Manual*.

A new example node (Node 5) is created. The root, Node 4, and Node 5 update their counters giving the result shown in « ENNetwork5». Since the output of the network is the same as the example, feature-specialization is not done.

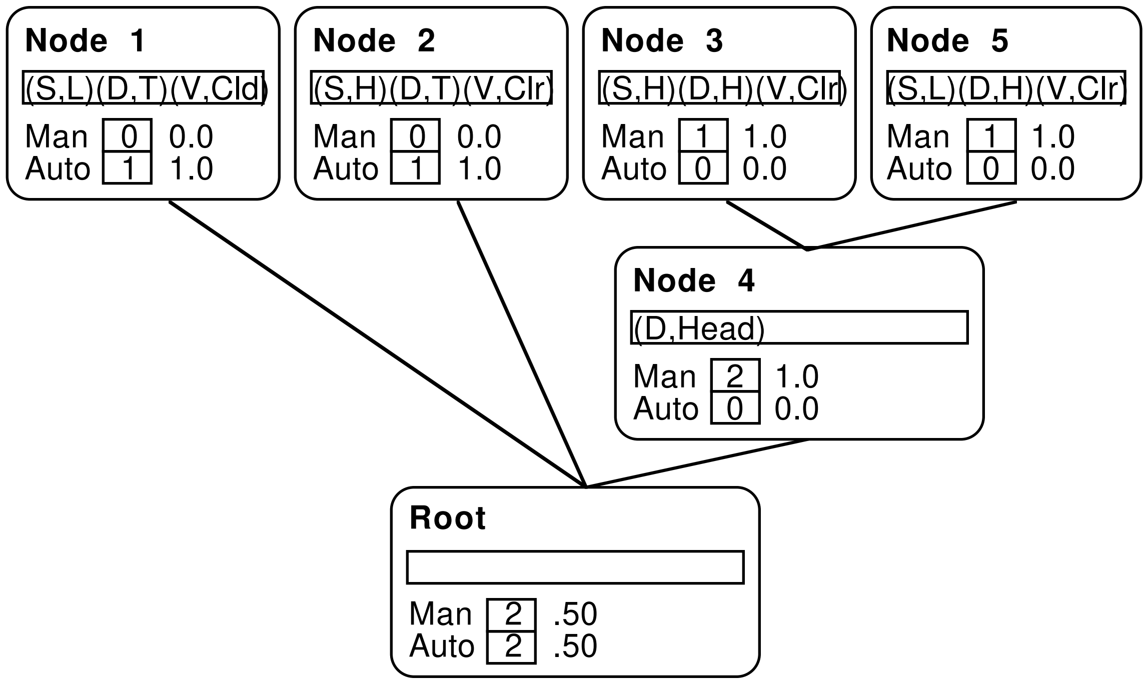

Figure 4.40. Network after fourth example

Example 5: (*High, Head, Cloudy*) → *Automatic*

Example 5 is given to the network. The network executes with the input (*High, Head, Cloudy*). The root and Node 4 both match the example. Node 4 is stronger than the root, so the network output is *Manual*.

An example node is created (Node 6). The root, Node 4, and Node 6 update their counters to the values shown in « ENNetwork6». The output of the network is different from the output of the example, so feature-specialization is executed.

The root node creates three candidate features with strengths for output *Automatic* as follows. (The strengths are calculated from the example nodes in « ENNetwork6»).

{(*Wind Speed, High*)}	2/3 = .67
{(*Wind Direction, Head*)}	1/3 = .33
{(*Visibility, Cloudy*)}	2/2 = 1.0

The feature {(*Visibility, Cloudy*)} is strongest, so it is the feature the root node will use to compete with Node 4 to obtain the right to create a new node.

Node 4 creates two new features with strengths for output *Automatic* as follows.

{(*Wind Speed, High*), (*Wind Direction, Head*)}	1/2 = .50
{(*Wind Direction, Head*), (*Visibility, Cloudy*)}	1/1 = 1.0

The feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)} is strongest, so it is the feature Node 4 will use to compete with the root.

Assume that simple generality is used to measure generality and that the constant c is set to .5. The generality, strength, and metric for the best new features of the two nodes are as follows.

Node	**Generality**	**Strength**	**Metric**
Root	2/3 = .67	1.0	$.67 \times .5 + 1 \times .5 = .83$
Node 4	1/3 = .33	1.0	$.33 \times .5 + 1 \times .5 = .67$

The root node has a higher metric, so it wins the competition and creates Node 7 with the feature {(*Visibility, Cloudy*)}. The counters in the new node are initialized to 0 and then the counter for *Automatic* is incremented for Node 1 and Node 6 giving a total of 2.

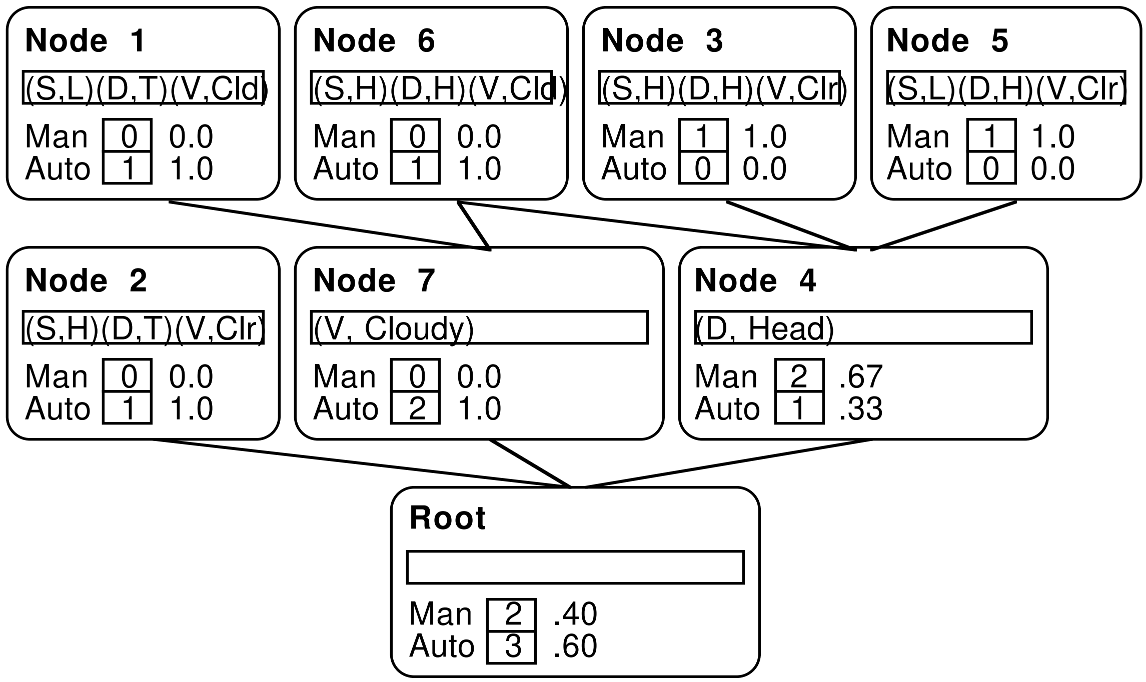

Figure 4.41. Network after last example

The final network created by EN estimation is identical to that created by both NIC and LEM estimation, except that the network contains example nodes. The functions implemented by the networks are also identical. As in the other strength-estimation examples, no deletion is needed.

EN estimation provides complete strength information about any new feature. NIC estimation only gives information about features that are one level beyond current nodes in terms of their specificity. LEM estimation may lose information about some examples due to limited memory resources. Thus, it is expected that EN estimation should give better accuracy and speed of learning results than the other two methods. Simulation results in Chapter 9 confirm this expectation.

EN estimation conserves memory resources: information about an example is stored in only one location. LEM estimation may store several copies of the same example in many different nodes. NIC estimation may have redundant counters keeping statistics about the same new feature.

EN estimation requires communication between nodes to estimate the strength of a new feature. Both NIC and LEM estimation can estimate the strength of a new feature using only local information. The general to specific network topology described in Chapter 7 reduces the network traffic required for EN estimation by localizing the communications.

Five algorithms for learning a set of features have been presented in this chapter. All the algorithms begin with general features. The general nodes gather statistics from the training examples. General features combine and specialize to form more specific features. Feature-union and feature-specialization algorithms create specific features based on the strength of general features. If a problem has strong specific features that are not composed of strong general features, the algorithms may create many specific features in error before finding good specific features. This weakness is overcome by three approaches to feature-specialization-with-strength-estimation. NIC estimation allows complete information to be kept locally about one level of additional specialization. EN estimation keeps complete information about multiple levels of specialization but requires internode communication. LEM estimation accuracy depends on the problem being learned, but LEM allows flexibility in the use of system resources.

Chapter 5
Specific to General Learning

This chapter presents another method of learning features that avoids the exponential nature of the number of features. Specific to general learning (SG) takes an approach essentially opposite to that of GS learning. The SG approach to creating features is to start with specific nodes and expand around those nodes creating general nodes. The system first creates nodes with features that match the training example exactly. Inputs are removed from the feature to create more general nodes until the new nodes conflict with discordant nodes. Expansion of a feature terminates when it encounters another feature with contradicting outputs.

One way SG generalizes is by combining similar specific features. If two features are similar, they are close to each other in the input space. Combining the two features by dropping inputs that are not common between the features creates a new feature that encompasses both the original features. The new feature is general: it matches points in the input space that have not been defined by any example.

5.1. Specific to General Learning Algorithms

Two different SG learning algorithms are presented in this chapter. The two algorithms differ only in the way they create new general features. This section describes the main routine that is shared by both algorithms.

Before any examples are given, SG begins with the single general feature {} that covers the complete input space. The initial feature is contained in a node called the *root*. The counters of the root node are all set to 0.

An SG system repeatedly executes the learning algorithm shown in « SGLearning» for each example that is given to the system. When a training example is presented during learning, the network executes (Lines 1 to 2) to obtain the network output for the input given in the example. Later, a general feature is created only if the output of the network is different from the output of the example. (The network executes using only general nodes. Example nodes do not participate since the network output would always be correct for the training examples after the first pass.)

After execution, the system creates an *example node* for the training example, if an equivalent node does not already exist (Lines 3 to 4). An example node is a node whose feature is fully specific and defined directly by the example. For instance, the example (*Low, Head, Clear*) → *Manual* causes the creation of feature {(*Wind Speed, Low*), (*Wind Direction, Head*), (*Visibility, Clear*)}. Example nodes participate in normal network execution, but they do not participate in execution during training. Example nodes are always created for SG learning since SG learning starts with specific features. The example nodes are the starting point for the process.

If the network output is not equal to the example output, then a general node is created using existing nodes. Otherwise, no additional nodes are added to the system. Because SG learning works from specific to general, it creates a general node even when the network output is *Don't-Know*. A *Don't-Know* output during learning suggests that the network does not cover the input point given in the example. Thus, a general node is created to cover the point and other points around it.

A general node can be created using *feature-intersection* or *feature-generalization* (Line 6). The two approaches are not mixed but actually give different learning algorithms. Feature-intersection and feature-generalization are described in the sections that follow.

After the training example is learned, nodes in the network that are not needed delete themselves. Deletion is described in Chapter 4.

Specific to General Learning

1. Broadcast the training example (i, o) to all nodes in the network.
2. Gather the strongest non-example node output giving the network output r.
3. Create a new feature $F = \{(i_1, x_1), ..., (i_n, x_n)\}$,
   where $x_1, ..., x_n$ are the values given in i.
4. If a node with feature F does not already exist, then create a new node with feature F. Initialize counters in the new node to zero.
5. Each node that matches i increments its counters as described previously.
6. If the network output is not correct ($r \neq o$) then create a general node by one of
   Feature-intersection
   Feature-generalization
7. Delete nodes that are not needed.

End.

Figure 5.1. Specific to General Learning

5.2. Feature Intersection

One method for creating general features is for existing nodes that are concordant with the training example to combine their current feature with the training example. The new feature is the intersection of the node's current feature and the feature defined by the training example. Theorem 3.4 shows that the subdomain of the intersection of two features completely encompasses the two original subdomains. Thus, the new feature is a generalization of both original features.

The new candidate feature is compared with features already existing in the network. If a feature the same as the new feature does not already exist, then the strength of the new feature is calculated using example nodes. The counters in example nodes that are matched by the new feature are summed to give a count for output o and a total count. The counts are divided to give the estimated strength. If the newly created feature already exists in another node in the network, the feature is not processed further.

If the new feature with the strength as estimated is not covered or contradicted by any other node, then the feature competes with all the other new features. If the new feature is covered or contradicted, there is no need to create a node with the feature since the node would not affect the network output. A new node is created using the strongest and most general new feature. The example nodes that are matched by the new node are used to initialize the counters in the new node.

Recall from Section 3.7 that a feature is *covered* if its subdomain is encompassed by the subdomain of a stronger feature with the same output. A feature is *contradicted* if its subdomain is encompassed by the subdomain of a stronger feature with different output. New features that are equal to, covered by, or contradicted by existing features are found using either a broadcast and gather or using a GS network topology. These techniques are described in Chapter 7.

Feature Intersection

1. For all nodes N with the same output as the training example.
    2. Let $F_e = \{(i_1, x_1), ..., (i_n, x_n)\}$, where $x_1, ..., x_n$ are the values given in i.
    3. Create a new feature $F = F_e \cap F_n$, where F_n is the feature of node N.
    4. If a node with feature F does not already exist then
        5. Calculate the strength S of F using example nodes.
        6. If F with strength S is not covered or contradicted then mark N.
    7. End.
    8. Calculate the metric $M = Gc + S(1–c)$, where G is the generality of F, S is the strength of F for output o, and c is a constant between 0 and 1.
9. End.
10. Gather the new feature F with highest M from the marked nodes.
11. Create a new node NN with feature F.
12. Initialize counters in NN by counting example nodes that match NN.

End.

Figure 5.2. Feature Intersection

5.2.1. Training Example

We now give an example of specific to general learning using feature-intersection. The network starts with only the root node as shown in « IntersectNet1». Connections are shown between nodes that have a general to specific relationship. Strengths are shown after the counters in each node.

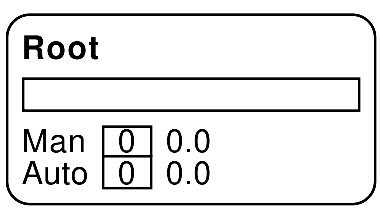

Figure 5.3. Initial Network

Example 1: (*Low, Tail, Cloudy*) $\rightarrow$ *Automatic*

Example 1 is given to the network. The network executes with the input (*Low, Tail, Cloudy*). The root node matches the example, but the strength of both outputs for the root node is 0. Therefore, the output of the root node is *Don't-Know*. Since there are no other nodes in the network, the output of the network is the output of the root or *Don't-Know*.

Node 1 is created as an example node with a feature that matches only the single input point given in the example. The root node and Node 1 update their counters giving the result shown in « IntersectNet2». Typically in SG learning general nodes are created even when the output of the network is *Don't-Know.* The root node is already fully general so no new features can be created from it by intersection. The intersection of the root feature and any other feature is the same as the root feature since the root feature is the empty set. The root effectively does not participate in the generalization process. Therefore, no additional nodes are created.

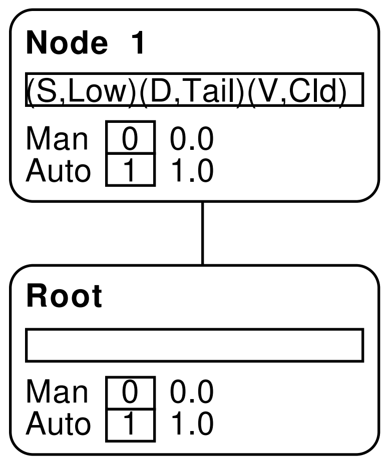

Figure 5.4. Network after first example

Example 2: (*High, Tail, Clear*) → *Automatic*

Example 2 is given to the network. The network executes with the input (*High, Tail, Clear*). The root node matches the example. Since Node 1 is an example node, it does not participate in network execution during training. The output of the root node is *Automatic*. Since there are no other matching nodes, the output of the network is the output of the root.

An example node does not already exist for the training example, so Node 2 is created. The root node and Node 2 update their counters giving the result shown in « IntersectNet3». The output of the network is the same as the example, so feature-intersection is not done and no additional nodes are created.

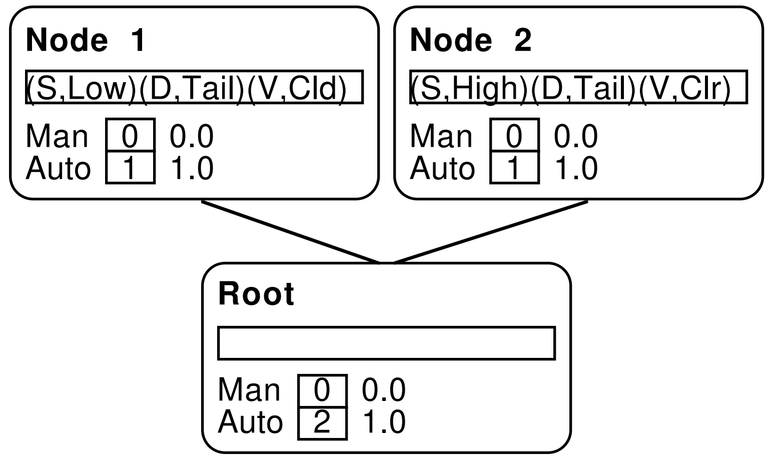

Figure 5.5. Network after second example

Example 3: (*High, Head, Clear*) → *Manual*

Example 3 is given to the network. The network executes with the input (*High, Head, Clear*). Again only the root node matches the example. The output of the root and thus the output of the network is *Automatic*.

Node 3 is created for the example. The root node and Node 3 update their counters giving the result shown in « IntersectNet4». The output of the network is different from the output of the example, so feature-intersection is executed.

The root node cannot generalize. Additionally, the root node and both Nodes 1 and 2 have outputs that are different from the training example. Intersection is only done with nodes that have the same output as the training example. Therefore, no nodes intersect with the example and no general features are created.

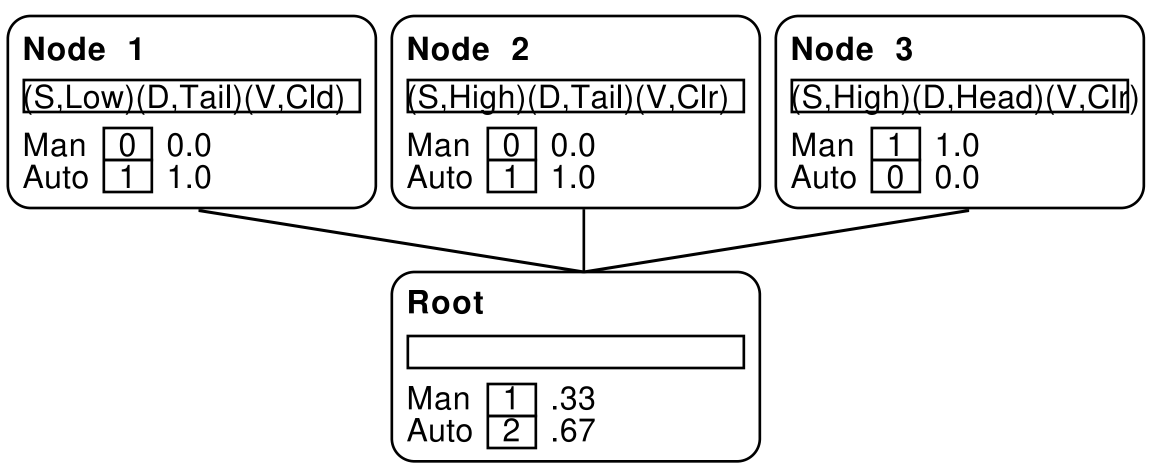

Figure 5.6. Network after third example

Example 4: (*Low, Head, Clear*) → *Manual*

Example 4 is given to the network. The network executes with the input (*Low, Head, Clear*). The root node matches the example and the root is the only node that is not an example node. The output of the root node as well as the network is *Automatic*.

A new example node (Node 4) is created. The root node and Node 4 update their counters giving the result shown in « IntersectNet5». Since the output of the network is different from the example, feature-intersection is executed.

The only node with output the same as the training example is Node 3. Node 3 intersects its feature, {(*Wind Speed, High*), (*Wind Direction, Head*), (*Visibility, Clear*)}, with the feature given by the training example, {(*Wind Speed, Low*), (*Wind Direction, Head*), (*Visibility, Clear*)}.

The resulting feature, {(*Wind Direction, Head*), (*Visibility, Clear*)}, is given an estimated strength of 2/2 = 1 by summing the counters of Nodes 3 and 4. The generality of the new feature is 1/3 = .33. Assuming the constant *c* is set to .5, the metric for the new feature is $.33 \times .5 + 1 \times .5 = .67$.

No other nodes create new features so Node 3 wins the competition and creates Node 5 with its new feature. The counters in Node 5 are first set to 0 and then the counters from Nodes 3 and 4 are summed into Node 5.

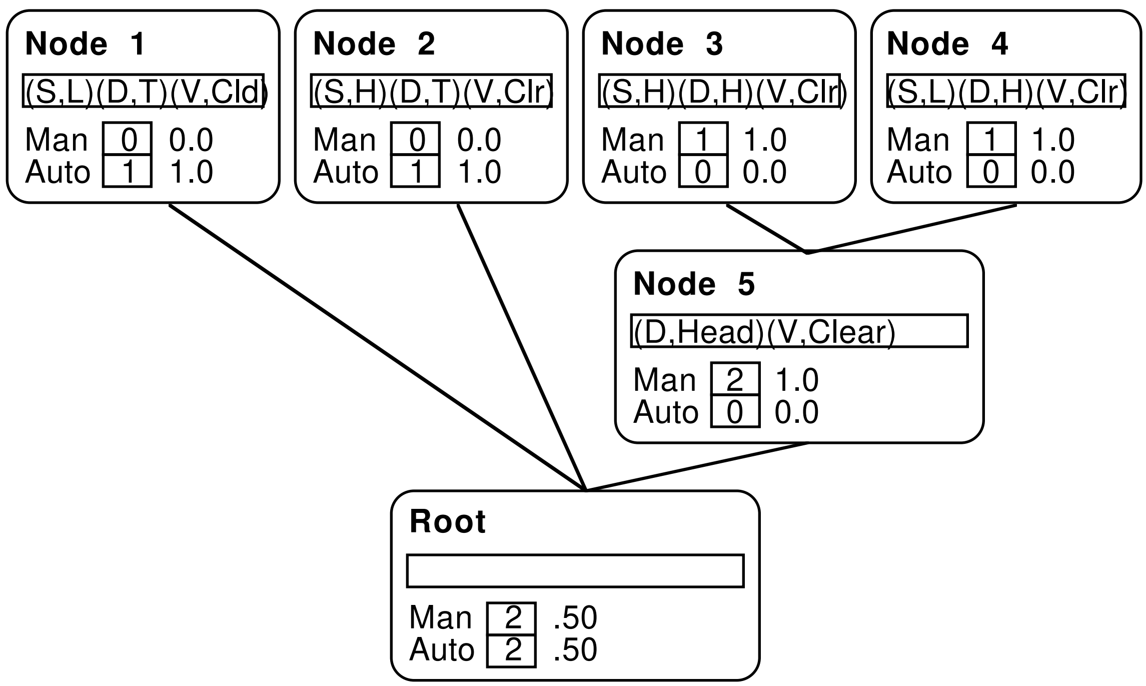

Figure 5.7. Network after fourth example

Example 5: (*High, Head, Cloudy*) → *Automatic*

Example 5 is given to the network. The network executes with the input (*High, Head, Cloudy*). The root is the only node that matches the example. The output of the network is the output of the root which is *Don't-Know*.

An example node is created (Node 6). The root and Node 6 update their counters to the values shown in « IntersectNet6». The output of the network is different from the output of the example, so feature-intersection is executed.

Nodes 1 and 2 have the same output as the example, so they both intersect their own feature with that defined by the example. Node 2 creates the new feature {(*Wind Speed, High*)}. The new feature is not equal to any existing feature. The feature has an estimated strength of 2/3 = .67. The strength of the root is 3/5 = .60 so the new feature is not covered or contradicted by any other node (the root is the only node that is a generalization of the new feature and thus the only node that has potential to cover or contradict the new feature).

Node 1 creates the new feature {(*Visibility, Cloudy*)}. The generality, strength, and metric for the new features of the two nodes are as follows. (Assume that simple generality is used to measure generality and that the constant *c* is set to .5).

Node	**Generality**	**Strength**	**Metric**
Node 1	2/3 = .67	2/2 = 1.0	$.67 \times .5 + 1 \times .5 = .83$
Node 2	2/3 = .67	2/3 = .67	$.67 \times .5 + .67 \times .5 = .67$

Nodes 1 and 2 compete using the metrics for their new features. Node 1 wins the competition and creates Node 7. The counters in the new node are initialized to 0 and then the counters from Nodes 1 and 6 are summed into Node 7.

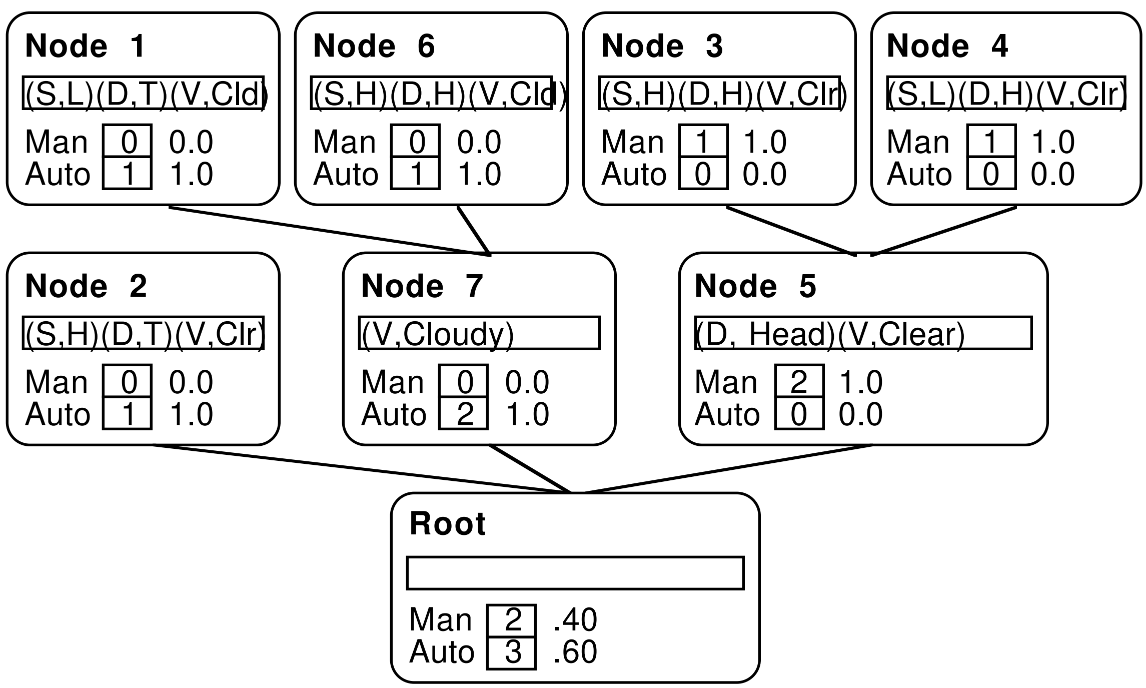

Figure 5.8. Network after last example

The function implemented by the network in « IntersectNet6» is shown in « IntersectFunction». The function is the same as that implemented by the GS learning models although the networks that implement the functions are different.

Table 5.1. Function implemented by network in « IntersectNet6»

	Head		Tail	
	Clear	Cloudy	Clear	Cloudy
Low	Manual	Automatic	Automatic	Automatic
Medium	Manual	Automatic	Automatic	Automatic
High	Manual	Automatic	Automatic	Automatic

5.3. Feature Generalization

Rather than combining nodes to create new nodes, individual existing nodes can generalize themselves by removing inputs from their feature. All nodes that match the current example and are concordant with the example generalize their feature until it is covered or contradicted.

The feature-generalization routine is shown in « FeatureGeneral». All nodes (in parallel) that match the training example and are concordant with the example create generalizations of the current feature by removing an input/value pair (Lines 1 to 3). For every input i_j contained in the node's current feature, a new feature is created by removing the pair (i_j, x_j) from the current feature. The new candidate features are stored in the node's local memory while strength estimation is done. Only a single new feature is selected to create a new node.

If a newly created feature does not exist in another node in the network, the strength of the new feature is calculated using example-node strength estimation (Lines 4 to 6). The candidate feature with highest strength for the target output that is not covered or contradicted is selected for further processing. It is possible that all new features are either covered or contradicted. In this case the generalization process terminates for the node.

If a new feature exists that is not covered or contradicted, then the strongest new feature is saved and the node optionally repeats the generalization process using the new feature (Lines 7 to 10). Generalization can be repeated as long as:

1. It is possible to generalize the current best new feature (the feature still contains inputs to be removed).
2. A new feature is created that is not covered or contradicted.

Since general features are preferred over specific features, a metric is calculated that combines generality and strength of the best new feature. As in GS learning and feature-intersection, the metric is a linear combination of generality and strength, given by $M = Gc + S(1-c)$.

The nodes compete according to the metric and the winning node is selected to create a new node using its best feature (Lines 13 to 15). The counters in the new node are initialized by counting the example nodes that match the new feature.

Feature Generalization

1. For all nodes N that match i and have the same output as o.
    2. Set F to the existing feature of the node N.
    3. For each input i_j in F create a new feature F_j by removing the pair (i_j, x_j) from F.
    4. For each new feature F_j.
        5. If a node with feature F_j does not already exist then calculate the strength of F_j using *Example Nodes*.
    6. End.
    7. If at least one new feature exists that is not covered or contradicted.
        8. Set F to the new feature with highest strength and mark N.
        9. Optionally repeat lines 3 to 10 using the new feature F.
    10. End.
    11. Calculate the metric $M = Gc + S(1–c)$, where G is the generality of F, S is the strength of F for output o, and c is a constant between 0 and 1.
12. End.
13. Gather the new feature F with highest M from the marked nodes.
14. Create a new node *NN* with feature F.
15. Initialize counters in *NN* by counting example nodes that match *NN*.

End.

Figure 5.9. Feature Generalization

5.3.1. Training Example

We now give an example of specific to general learning using feature-generalization. The network starts with only the root node as shown in « GenNet1». Connections are shown between nodes that have a general to specific relationship. Strengths are shown after the counters in each node.

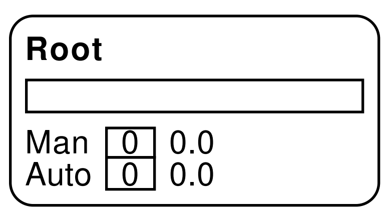

Figure 5.10. Initial Network

Example 1: (*High, Head, Cloudy*) → *Automatic*

Example 1 is given to the network. The network executes with the input (*High, Head, Cloudy*). The root node matches the example, but the strength of both outputs for the root node is 0. Therefore, the output of the root node is *Don't-Know*. Since there are no other nodes in the network, the output of the network is the output of the root or *Don't-Know*.

Node 1 is created as an example node with a feature that matches only the single input point given in the example. The root node and Node 1 update their counters giving the result shown in « GenNet2». Since the output of the network is *Don't-Know,* feature-generalization is executed.

Both Node 1 and the root node match the example, so they both participate in generalization. The root is already completely general (it contains no input/value pairs that can be removed), so the root cannot create any new nodes more general than itself.

Node 1 must decide which of the three input/value pairs to remove from its current feature. The three features resulting from removing each of the possible input/value pairs all match the single example node in the network (Node 1). All three features therefore have the same estimated strength of 1/1 = 1. One of the features is arbitrarily chosen. Suppose that the feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)}, resulting from dropping the first input/value pair (*Wind Speed, High*), is chosen.

The feature must be tested against current network nodes to determine if a new node can be created. The feature is not equal to that contained in any node in the network. The feature is not *contradicted* by any node in the network since the output of all current nodes is *Automatic* which is the same as the new feature. However, the feature is *covered* by the root node since the root node has the same output and strength as the new feature. Therefore, a new node is not created for the feature. The resulting network is shown in « GenNet2».

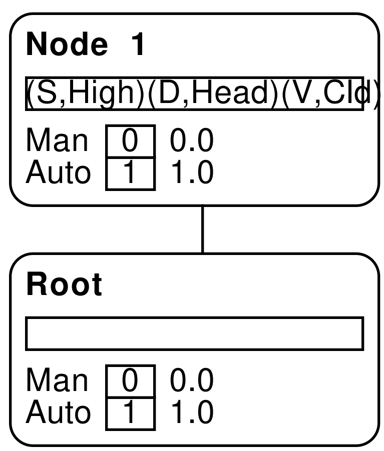

Figure 5.11. Network after first example

Example 2: (*High, Tail, Clear*) → *Automatic*

Example 2 is given to the network. The network executes with the input (*High, Tail, Clear*). The root node matches the example. Since Node 1 is an example node, it does not participate in network execution during training. Even if example nodes did take part, Node 1 does not match the example. The output of the root node is *Automatic*. Since there are no other matching nodes, the output of the network is the output of the root.

An example node does not already exist for the training example, so Node 2 is created. The root node and Node 2 update their counters giving the result shown in « GenNet3». The output of the network is the same as the example, so feature-generalization is not done and no additional nodes are created.

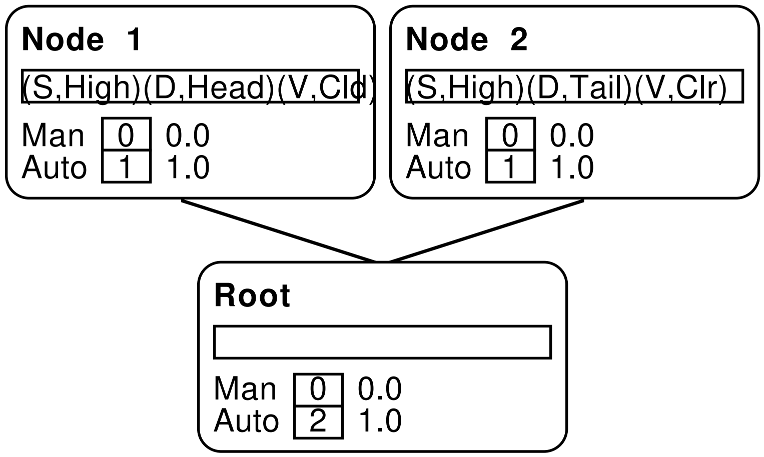

Figure 5.12. Network after second example

Example 3: (*High, Head, Clear*) $\rightarrow$ *Manual*

Example 3 is given to the network. The network executes with the input (*High, Head, Clear*). Again only the root node matches the example. The output of the root and thus the output of the network is *Automatic*.

Node 3 is created for the example. The root node and Node 3 update their counters giving the result shown in « GenNet4». The output of the network is different from the output of the example, so feature-generalization is executed.

The root node cannot generalize. Additionally, the root node and both Nodes 1 and 2 have outputs that are different from the training example. Generalization is only done with nodes that match the training example and have the same output as the example. Node 3 is the only node that meets this criteria.

Node 3 calculates the strength of the three features resulting from removing each input/value pair from its current feature. The feature {(*Wind Speed, High*), (*Wind Direction, Head*)} matches Example Nodes 1 and 3 so the strength of the feature for *Manual* is 1/2. The strength of feature {(*Wind Speed, High*), (*Visibility, Clear*)} is also 1/2 since it matches Nodes 2 and 3. The strength of feature {(*Wind Direction, Head*), (*Visibility, Clear*)} is 1/1 because it only matches Node 3.

The feature {(*Wind Direction, Head*), (*Visibility, Clear*)} is the strongest feature for Node 3. The generality of the new feature is 1/3 = .33. Assuming the constant *c* is set to .5, the metric for the new feature is $.33 \times .5 + 1 \times .5 = .67$. There are no other nodes to compete with Node 3 for the creation of a new node.

Node 3's candidate feature is not equal to the feature of any node, nor is it covered or contradicted by any node. The subdomain of the root node completely contains the subdomain of the new feature, so the root could potentially cover or contradict the new feature. However, the strength of the root is lower than that of the new feature. Therefore, Node 4 is created using the new feature. The counters in Node 4 are initialized using the only example node that is matched by Node 4, which is Node 3.

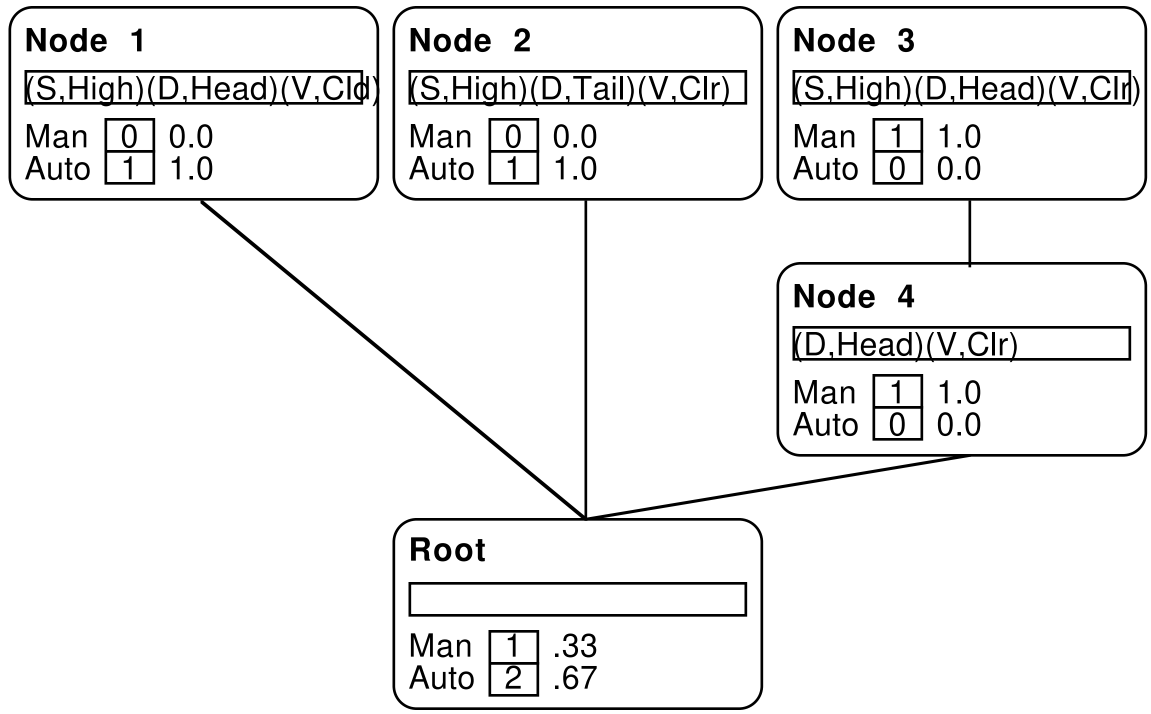

Figure 5.13. Network after third example

Example 4: (*Low, Head, Clear*) → *Manual*

Example 4 is given to the network. The network executes with the input (*Low, Head, Clear*). The root node and Node 4 both match the example. The strength of the root is 2/3 = .67 and the output is *Automatic*. The strength of Node 4 is 1/1 = 1 and the output is *Manual*. Since Node 4 is stronger, the output of the network is *Manual*.

A new example node (Node 5) is created. The root node, Node 4, and Node 5 update their counters giving the result shown in « GenNet5». Since the output of the network is the same as the example, feature-generalization is not done. The resulting network is shown in « GenNet5».

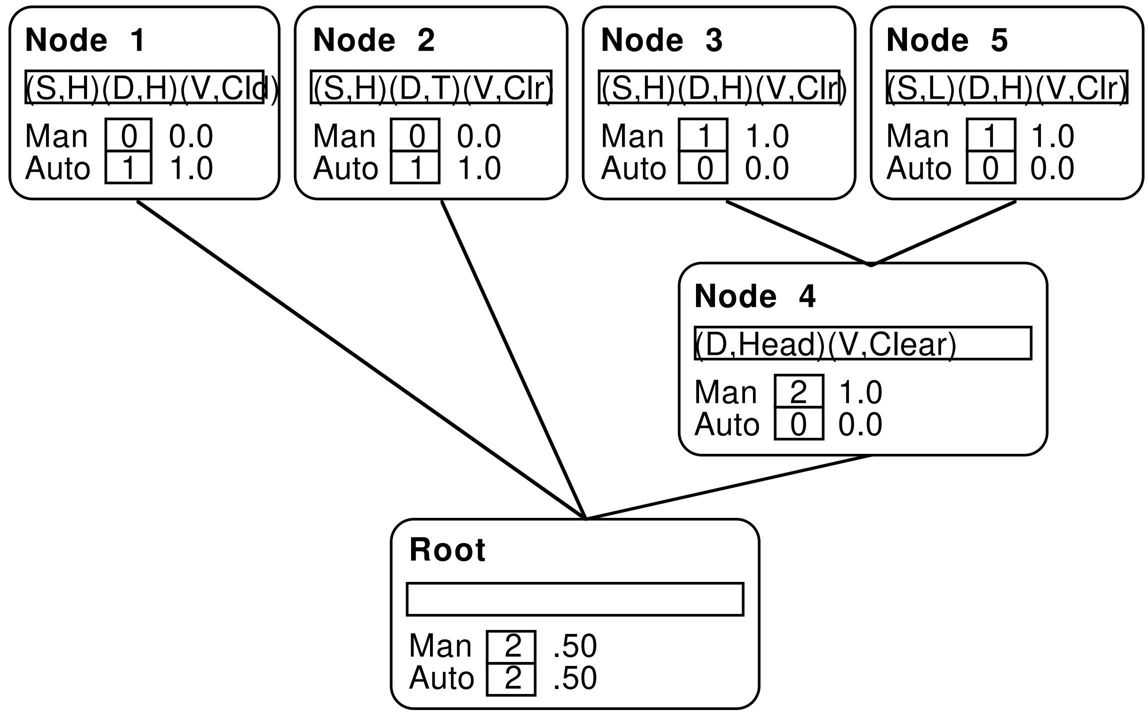

Figure 5.14. Network after fourth example

Example 5: (*Low, Head, Cloudy*) → *Automatic*

Example 5 is given to the network. The network executes with the input (*Low, Head, Cloudy*). The root is the only node that matches the example. The output of the network is the output of the root which is *Don't-Know*.

An example node is created (Node 6). The root and Node 6 update their counters to the values shown in « GenNet6». The output of the network is different from the output of the example, so feature-generalization is executed.

Node 6 is the only node that matches the example and has the same output as the example. Node 6 creates three new candidate features with strengths for output *Automatic* as follows.

{(*Wind Speed, Low*), (*Wind Direction, Head*)}	1/2 = .50
{(*Wind Speed, Low*), (*Visibility, Cloudy*)}	1/1 = 1.0
{(*Wind Direction, Head*), (*Visibility, Cloudy*)}	2/2 = 1.0

None of the features is equal to any existing feature. Two of the features have equal highest strength, so one of the two is arbitrarily chosen. Suppose that the last feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)} is chosen. The new feature is not covered or contradicted by any other node.

No other nodes match the example, so Node 6 has no competition for creation of a new node. Node 7 is created with counters initialized from Nodes 1 and 6.

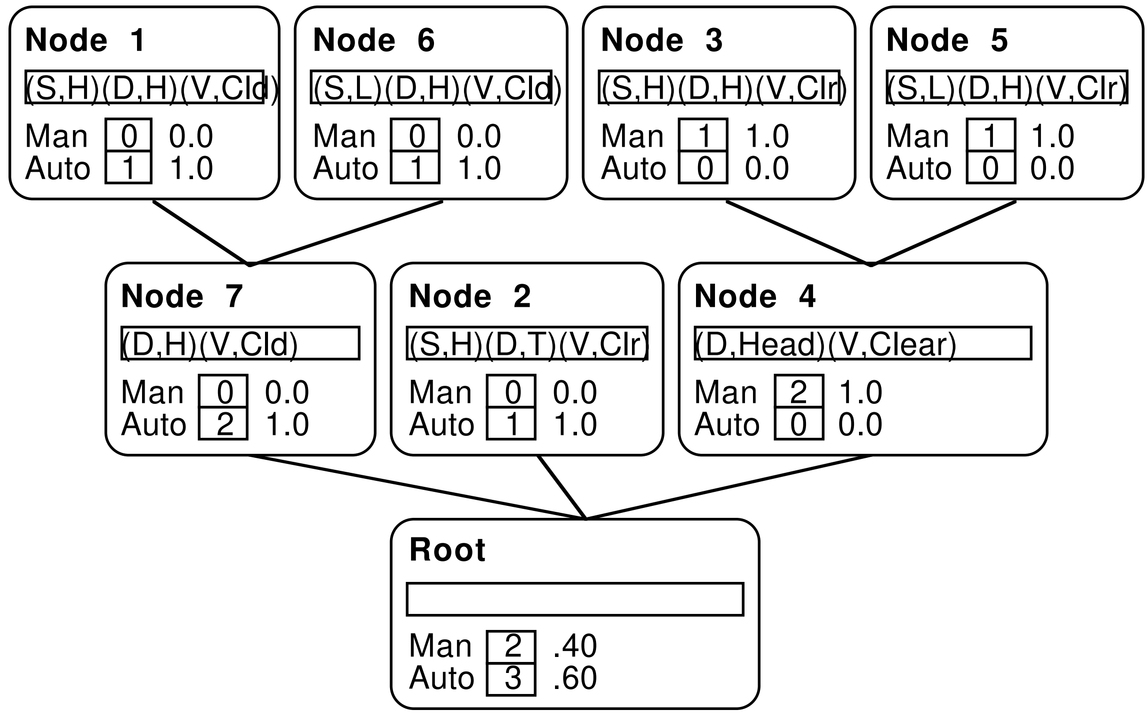

Figure 5.15. Network after last example

The function implemented by the network in « GenNet6» is the same as that implemented by feature-intersection although the networks that implement the functions are different.

Two algorithms for learning a set of features have been presented in this chapter. Both algorithms begin with specific features. The specific features are combined and generalized to create more general features. Feature-intersection can combine two specific features and create a very general feature quickly. Feature-generalization requires more steps to create a very general feature since it can only remove one input at a time. Therefore, feature-intersection is expected to converge to a solution more quickly than feature-generalization. On the other hand, feature-intersection can only create new features that are an intersection between an existing feature and the training example. Feature-generalization has more options for the new features it can create. Thus, feature-generalization may find better solutions for the problem being learned.

SG learning begins with example nodes and expands around them to find general features that match the problem. GS learning begins with general features and specializes them. Thus, SG is expected to give better results when the problem has a large number of specific features in its solution. GS should perform better when a small number of general features give a good problem solution.

Chapter 6
Combining GS Learning and SG Learning

GS learning begins with general features and specializes those features to learn a simple set of features that implements a function. SG learning begins with specific features and generalizes those features also to learn a set of features that implements a function. GS learning performs better on some training sets than SG learning and SG learning performs better than GS learning on other sets of data. In this chapter the two methods are combined so that both specialization and generalization are done in the same model. The resulting model provides good performance over a wider range of applications.

GS and SG models are combined in two ways. One method executes an SG algorithm followed by a GS algorithm. Each algorithm creates a new node so two new nodes are created each time the network does not output correctly for a training example. Another way to combine GS and SG models is to create only a single new node with the feature that is the best of the features generated from the SG algorithm and the GS algorithm.

When creating a single best feature, an individual node considers both generalizations and specializations of its current feature. The node generates the generalizations of its feature that would be generated by feature-generalization in SG learning. The node also generates the specializations of its feature that would be generated by feature-specialization in GS learning. The best new feature is then selected for further consideration.

It may appear that union and intersection operations also could be combined. However, union combines two relatively general features into a more specific feature. Intersection combines two relatively specific features into a more general feature. Union and intersection are therefore difficult to combine into a single model. A model combining union and intersection would require complex interactions between multiple nodes.

Since SG models exclusively use EN strength estimation, the most sensible GS models to combine with SG are those that use EN strength estimation. If new features are to be comparable between the GS and SG approaches then the same method of strength estimation should be used for both generalization and specialization.

One combined algorithm runs feature-specialization with EN strength estimation and creates a new node. Then feature-generalization with EN strength estimation is executed and a second new node is created.

In the second combined algorithm, individual nodes create both generalizations and specializations of their current feature as described above. The strength of both the generalizations and specializations is estimated using EN strength estimation. The strongest new feature is used in competition with other nodes. The winning node creates a single new node using the strongest feature.

6.1. Generalization And Specialization Learning Algorithm

This section describes the main routine that is shared by both types of combined generalization and specialization (GAS) algorithms. A GAS system repeatedly executes the learning algorithm shown in « GASLearn» for each example that is given to the system. Before any examples are given, GAS begins with the single general feature {} that covers the complete input space. The initial feature is contained in a node called the *root*. The counters of the root node are all set to 0.

When a training example is presented during learning, the network executes (Lines 1 to 2) to obtain the network output for the input given in the example. Later, generalization-and-specialization is only executed if the output of the network is different from the output of the example.

After execution, the system creates an *example node* for the training example, if an equivalent node does not already exist (Lines 3 to 4). Example nodes are always needed for GAS Learning for two reasons. First, EN strength estimation is used in GAS Learning. Second, since generalization is done, example nodes are needed as a starting point for the process.

The network can optionally create *single-input nodes* (Lines 5 to 8) just as in GS learning. The system compares the output of the network with the output of the example to determine if generalization-and-specialization should be done just as in both GS and SG.

Lines 11 and 12 list the two alternatives for GAS Learning. Either two nodes are created, one each for generalization and specialization, or one node is created that is the best feature created by both of the methods.

Generalization And Specialization (GAS) Learning

1. Broadcast the training example (i, o) to all nodes in the network.
2. Gather the strongest non-example node output giving the network output r.
3. Create a new feature $F = \{(i_1, x_1), ..., (i_n, x_n)\}$,
   where $x_1, ..., x_n$ are the values given in i.
4. If a node with feature F does not already exist, then create a new node with feature F. Initialize counters in the new node to zero.
5. If single-input nodes are to be created, then for each input value x_j in i.
    6. Create a new feature $F = \{(i_j, x_j)\}$.
    7. If a node with feature F does not already exist, then create a new node with feature F. Initialize counters in the new node to zero.
8. End.
9. Each node that matches i increments its counters as described previously.
10. If the network output is not correct ($r \neq o$) then do one of the following.
    11. Create two new nodes, one for *feature-specialization* and one for *feature-generalization*.
    12. Create one new node, the best of *feature-specialization* and *feature-generalization*.
13. End.
14. Delete nodes that are not needed.

End.

Figure 6.1. Generalization and Specialization Learning

After the training example is learned, nodes in the network that are not needed delete themselves. Deletion is described in Chapter 4.

6.2. Both Generalization and Specialization

When creating two new nodes, the previously described feature-generalization and feature-specialization procedures can be used without modification. Feature-generalization with EN strength estimation is executed and then feature-specialization with EN strength estimation is executed.

An example is given to illustrate. The network starts with only the root node as shown in « GSSGNet1». Connections are shown between nodes that have a general to specific relationship. Strengths are shown after the counters in each node. For simplicity, single-input nodes are not created.

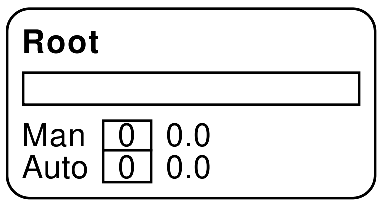

Figure 6.2. Initial Network

Example 1: (*High, Tail, Clear*) → *Automatic*

Example 1 is given to the network. The network executes with the input (*High, Tail, Clear*). The root node matches the example but the strength of both outputs for the root node is 0. Therefore, the output of the root node is *Don't-Know*. Since there are no other nodes in the network, the output of the network is the output of the root or *Don't-Know*. For the purpose of simplifying this example, when the output of the network is *Don't-Know,* feature-generalization and feature-specialization will not be executed.

Node 1 is created as an example node with a feature that matches only the single input point given in the example. The root node and Node 1 update their counters giving the result shown in « GSSGNet2». Since the output of the network is *Don't-Know,* no other nodes are created.

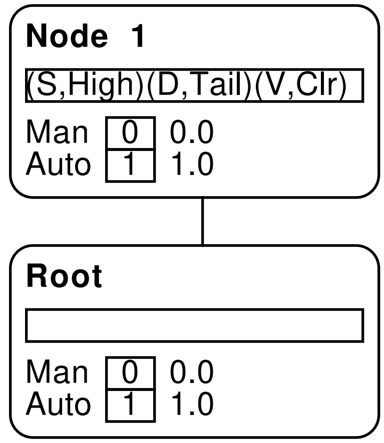

Figure 6.3. Network after first example

Example 2: (*High, Head, Cloudy*) → *Automatic*

Example 2 is given to the network. The network executes with the input (*High, Head, Cloudy*). The root node matches the example. Since Node 1 is an example node, it does not participate in network execution during training. Even if example nodes did take part, Node 1 does not match the example. The output of the root node is *Automatic*. Since there are no other matching nodes, the output of the network is the output of the root.

An example node does not already exist for the training example so Node 2 is created. The root node and Node 2 update their counters giving the result shown in « GSSGNet3». The output of the network is the same as the example, so feature-generalization and feature-specialization are not done and no additional nodes are created.

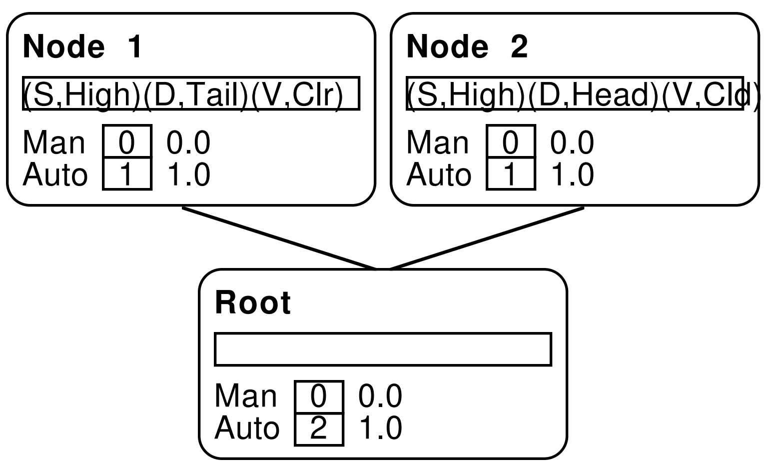

Figure 6.4. Network after second example

Example 3: (*High, Head, Clear*) → *Manual*

Example 3 is given to the network. The network executes with the input (*High, Head, Clear*). Only the root node matches the example. The output of the root and thus the output of the network is *Automatic*.

Node 3 is created for the example. The root node and Node 3 update their counters giving the result shown in « GSSGNet4». The output of the network is different from the output of the example, so feature-generalization is executed followed by feature-specialization.

The root node is already completely general (it contains no input/value pairs that can be removed) so the root cannot create any new nodes more general than itself. Additionally, the root node and both Nodes 1 and 2 have outputs that are different from the training example. Generalization is only done with nodes that match the training example and have the same output as the example. Node 3 is the only node that meets this criteria.

Node 3 calculates the strength of the three features resulting from removing each input/value pair from its current feature. The feature {(*Wind Speed, High*), (*Wind Direction, Head*)} matches Example Nodes 2 and 3, so the strength of the feature for *Manual* is 1/2. The strength of feature {(*Wind Speed, High*), (*Visibility, Clear*)} is also 1/2 since it matches Nodes 1 and 3. The strength of feature {(*Wind Direction, Head*), (*Visibility, Clear*)} is 1/1 because it only matches Node 3.

The feature {(*Wind Direction, Head*), (*Visibility, Clear*)} is the strongest feature for Node 3. Node 3's candidate feature is not equal to that in any node, nor is it covered or contradicted by any node. There are no other nodes to compete with Node 3 for the creation of a new node. Therefore, Node 4 is created using the new feature. The counters in Node 4 are initialized using the only example node that is matched by Node 4 which is Node 3.

The root node is the only node able to attempt feature-specialization since Node 3 (the only other node that matches the example) is already completely specific. The root node creates three features from the training example with strengths for output *Manual* as follows. (The strengths are calculated from the example nodes in « GSSGNet4»).

{(*Wind Speed, High*)}	1/3 = .33
{(*Wind Direction, Head*)}	1/2 = .50
{(*Visibility, Clear*)}	1/2 = .50

Since the last two features have equal strength, assume that the feature {(*Wind Direction, Head*)} is arbitrarily selected. No other nodes compete with the root node so the root creates Node 5. The counters in Node 5 are initialized by summing the counters in Nodes 2 and 3. The resulting network is shown in « GSSGNet4».

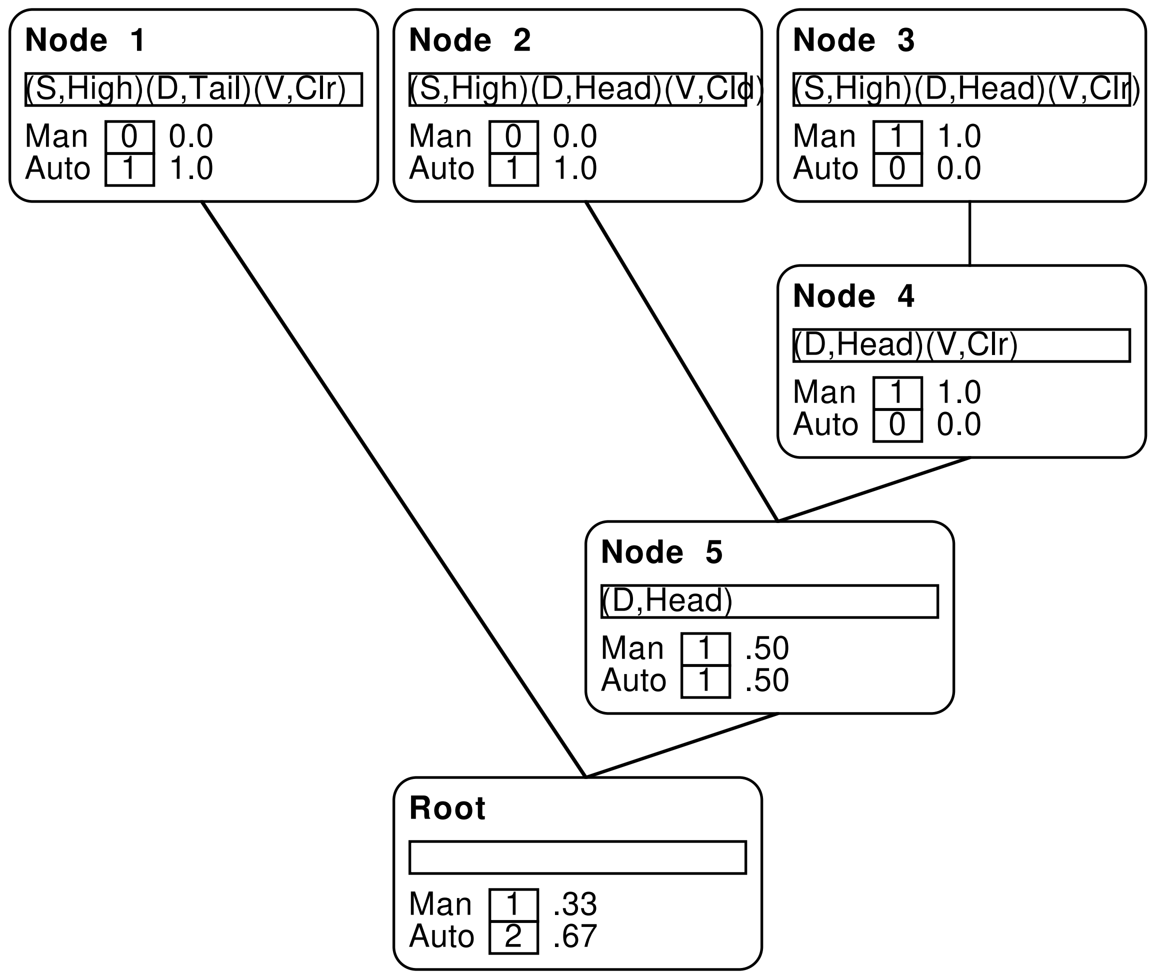

Figure 6.5. Network after third example

Example 4: (*Low, Head, Clear*) → *Manual*

Example 4 is given to the network. The network executes with the input (*Low, Head, Clear*). The root node, Node 4, and Node 5 all match the example. The strength of the root is 2/3 = .67 and the output is *Automatic*. The strength of Node 4 is 1/1 = 1 and the output is *Manual*. The strength of Node 5 is 1/2 = .5 and the output is *Don't-Know*. Since Node 4 is strongest, the output of the network is *Manual*.

A new example node (Node 6) is created. The root node, Node 4, Node 5, and Node 6 update their counters giving the result shown in « GSSGNet5». Since the output of the network is the same as the example, feature-generalization and feature-specialization are not done. The resulting network is shown in « GSSGNet5».

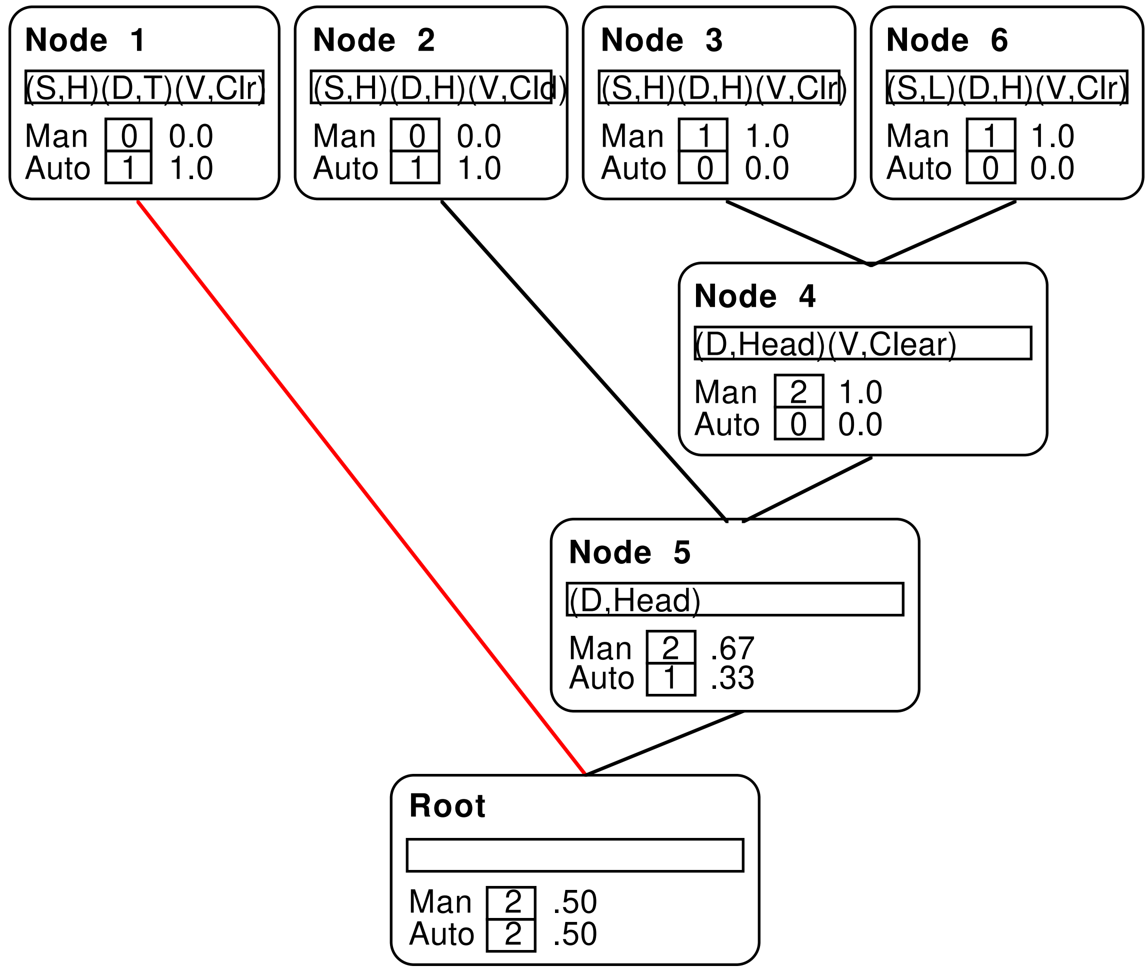

Figure 6.6. Network after fourth example

Example 5: (*Low, Head, Cloudy*) → *Automatic*

Example 5 is given to the network. The network executes with the input (*Low, Head, Cloudy*). The root and Node 5 match the example. The strength of the root is 2/4 = .5 and the output is *Don't-Know*. The strength of Node 5 is 2/3 = .67 and the output is *Manual*. Therefore, the output of the network is *Manual* since Node 5 is the strongest.

An example node is created (Node 7). The root, Node 5, and Node 7 update their counters to the values shown in « GSSGNet6». The output of the network is different from the output of the example, so feature-generalization is executed followed by feature-specialization.

Node 7 is the only node that matches the example and has the same output as the example. (The root node meets the criteria but as before cannot generalize. Node 5 matches the example but the output is *Don't-Know*. Even if Node 5 could participate its only generalization is equal to the root.)

Node 7 creates three new candidate features with strengths for output *Automatic* as follows.

{(*Wind Speed, Low*), (*Wind Direction, Head*)} 1/2 = .50

{(*Wind Speed, Low*), (*Visibility, Cloudy*)}	1/1 = 1.0
{(*Wind Direction, Head*), (*Visibility, Cloudy*)}	2/2 = 1.0

None of the features is equal to any existing feature. Two of the features have equal highest strength, so one of the two is arbitrarily chosen. Suppose that the last feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)} is chosen.

The strength of the root is 3/5 and the strength of Node 5 is 2/4 = .5, so the new feature is not covered or contradicted by any other node. (The root node and Node 5 are the only nodes with subdomains that encompass the new feature and thus the only nodes that have potential to cover or contradict the new feature.)

Node 7 has no competition for creation of a new node. Node 8 is created with counters initialized from Nodes 2 and 7.

The root node and Node 5 both perform feature-specialization since they both match the example. The root node creates three features from the training example with strengths for output *Automatic* as follows.

{(*Wind Speed, Low*)}	1/2 = .50
{(*Wind Direction, Head*)}	2/4 = .50
{(*Visibility, Cloudy*)}	2/2 = 1.0

The feature {(*Wind Direction, Head*)} is already contained in Node 5, so the feature is not considered further. The feature {(*Visibility, Cloudy*)} has highest strength and is selected.

Node 5 creates two new features with strengths for output *Automatic* as follows.

{(*Wind Speed, Low*), (*Wind Direction, Head*)}	1/2 = .50
{(*Wind Direction, Head*), (*Visibility, Cloudy*)}	2/2 = 1.0

The feature {(*Wind Direction, Head*), (*Visibility, Cloudy*)} is equal to the feature in Node 8 and is not processed further.

Assume that simple generality is used to measure generality and that the constant *c* is set to .5. The generality, strength, and metric for the competing features are as follows.

Node	**Generality**	**Strength**	**Metric**
Root	2/3 = .67	1.0	$.67 \times .5 + 1 \times .5 = .83$
Node 5	1/3 = .33	.50	$.33 \times .5 + .5 \times .5 = .42$

The root node wins the competition and creates Node 9 using the new feature. The counters in Node 9 are initialized by summing the counters in Nodes 2 and 7. The resulting network is shown in « GSSGNet6».

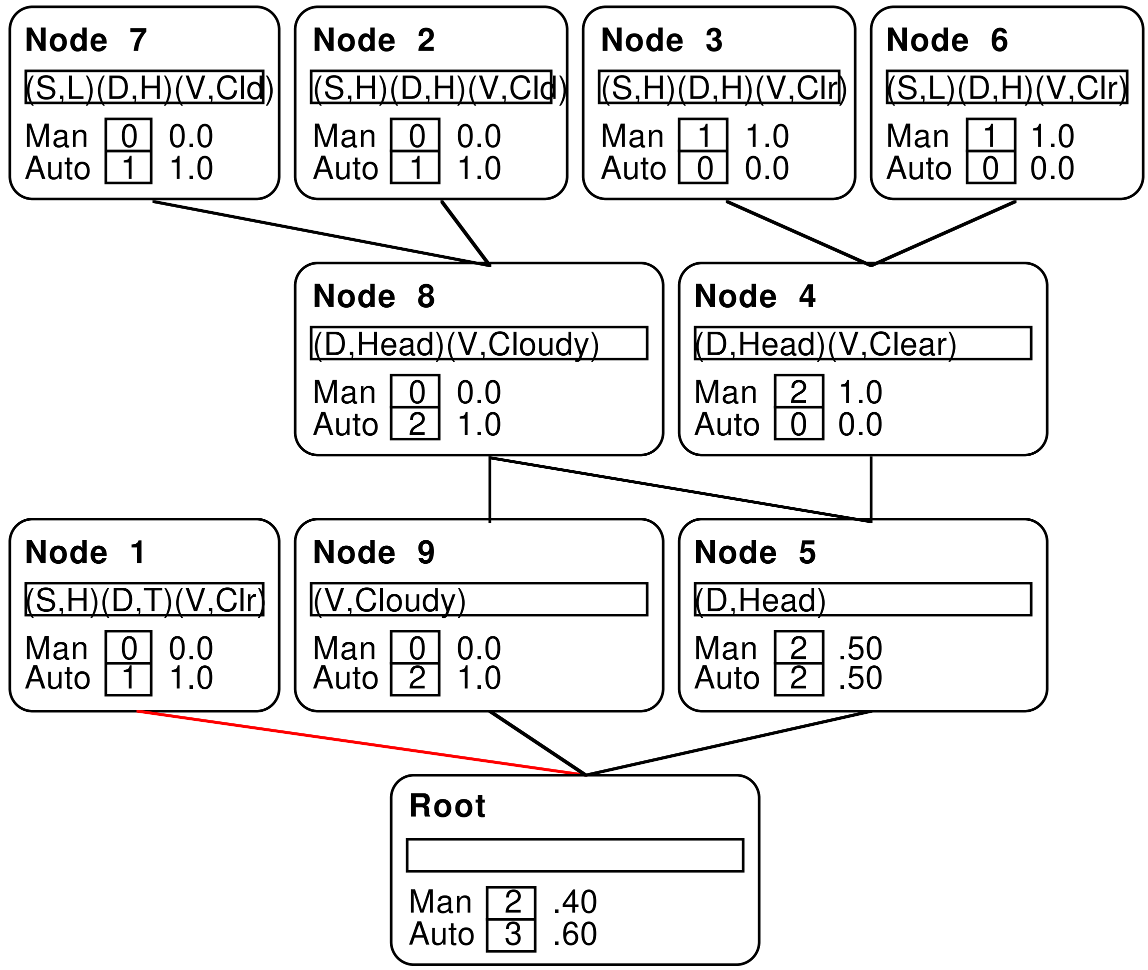

Figure 6.7. Network after last example

After a second pass over the training examples, Nodes 5 and 8 delete themselves, since neither node ever wins the competition during execution. Any example matched by Node 5 is also matched by the root node. The root node has higher strength than Node 5, so the root will always win. Any example matched by Node 8 is also matched by Node 9. Node 9 has the same strength as Node 8. Since Node 9 has higher generality, it will always win the competition.

The function implemented by the resulting network is the same as that implemented by both the GS learning models and the SG learning models, although the networks that implement the functions are different. When learning more complex problems, GAS should give better results than either GS or SG since GAS combines both models. GAS has potential to learn (1) problems that have a small number of general features and (2) problems that have a larger number of more specific features.

6.3. Best of Generalization or Specialization

When doing generalization-and-specialization nodes may be created that are not of high strength such as Node 5 in the last example. In that example, if only the best of the two possible new nodes were created, then Node 4 would be selected over Node 5 and Node 5 would not have been created. This section shows how generalization and specialization are combined by creating the best new feature of all those that could be created by either feature-generalization or feature-specialization.

The generalization-or-specialization routine is shown in « GoSLearn». All nodes (in parallel) that match the training example create generalizations of the current feature by removing an input/value pair and specializations by adding an input/value pair (Lines 1 to 6). For every input i_j contained in the node's current feature, a new more general feature is created by removing the pair (i_j, x_j) from the current feature. For every input i_j *not* contained in the node's current feature, a new more specific feature is created by adding the pair (i_j, x_j) to the current feature, where x_j is the value of input i_j from the training example. The new candidate features are stored in the node's local memory while strength estimation is done. Only a single new feature is selected to create a new node.

If a newly created feature does not exist in another node in the network, the strength of the new feature is calculated using example nodes (Lines 7 to 9). Candidate features that are specializations and are weaker than the current feature of the node are eliminated from competition. Candidate features that are covered or contradicted by another node are also eliminated from competition. The remaining candidate features compete and the candidate feature with highest strength for the target output is selected for further processing. It is possible that all new features are either covered or contradicted. In this case the process terminates for the node.

If a new feature exists that is not covered or contradicted, then the strongest new feature is saved and the node optionally repeats the process using the new feature (Lines 10 to 13).

Since general features are to be preferred over specific features, a metric is calculated that combines generality and strength of the best new feature. As in both GS learning and SG learning, the metric is a linear combination of generality and strength, given by $M = Gc + S(1–c)$.

The marked nodes compete according to the metric and the winning node is selected to create a new node using its best feature (Lines 16 to 18). The counters in the new node are initialized by summing the counters in all the example nodes that are matched by the new node.

Generalization or Specialization

1. For all nodes N that match the example
2. Set F to the existing feature of the node N.
3. For each input i_j create a new feature F_j as follows.
4. If i_j is not contained in F then $F_j = F \cup \{(i_j, x_j)\}$, where x_j is the value of input i_j in the training example.
5. If i_j is contained in F then $F_j = F - \{(i_j, x_j)\}$.
6. End.
7. For each new feature F_j.
8. If a node with feature F_j does not already exist then calculate the strength of F_j using *Example Nodes*.
9. End.
10. If at least one new feature exists that is not covered or contradicted.
11. Set F to the new feature with highest strength and mark N.
12. Optionally repeat lines 3 to 13 using the new feature F.
13. End.
14. Calculate the metric $M = Gc + S(1-c)$, where G is the generality of F, S is the strength of F for output o, and c is a constant between 0 and 1.
15. End.
16. Gather the new feature F with highest M from the marked nodes.
17. Create a new node NN with feature F.
18. Initialize counters in NN by counting example nodes that match NN.

End.

Figure 6.8. Feature Generalization or Specialization

6.3.1. Training Example

We now give an example of GAS learning selecting the best of generalization and specialization. The network starts with only the root node as shown in « GoSNet1». Connections are shown between nodes that have a general to specific relationship. Strengths are shown after the counters in each node. For simplicity, single-input nodes are not created.

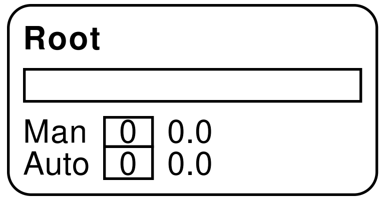

Figure 6.9. Initial Network

Example 1: (*High, Tail, Clear*) → *Automatic*

Example 1 is given to the network. The network executes with the input (*High, Tail, Clear*). The root node matches the example but the strength of both outputs for the root node is 0. Therefore, the output of the root node is *Don't-Know*. Since there are no other nodes in the network, the output of the network is the output of the root or *Don't-Know*.

Node 1 is created as an example node with a feature that matches only the single input point given in the example. The root node and Node 1 update their counters giving the result shown in « GoSNet2». Since the output of the network is *Don't-Know,* no other nodes are created.

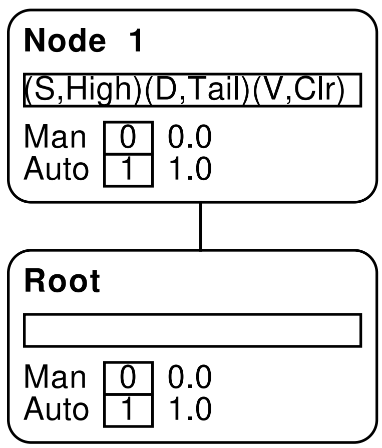

Figure 6.10. Network after first example

Example 2: (*High, Head, Cloudy*) → *Automatic*

Example 2 is given to the network. The network executes with the input (*High, Head, Cloudy*). The root node matches the example. Since Node 1 is an example node, it does not participate in network execution during training. Even if example nodes did take part, Node 1 does not match the example. The output of the root node is *Automatic*. Since there are no other matching nodes, the output of the network is the output of the root.

An example node does not already exist for the training example so Node 2 is created. The root node and Node 2 update their counters giving the result shown in « GoSNet3». The output of the network is the same as the example, so generalization-or-specialization is not done and no additional nodes are created.

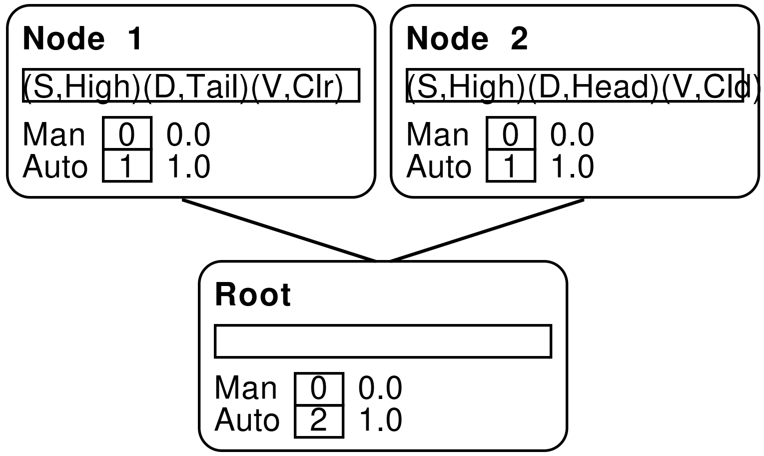

Figure 6.11. Network after second example

Example 3: (*High, Head, Clear*) → *Manual*

Example 3 is given to the network. The network executes with the input (*High, Head, Clear*). Only the root node matches the example. The output of the root and thus the output of the network is *Automatic*.

Node 3 is created for the example. The root node and Node 3 update their counters giving the result shown in « GoSNet4». The output of the network is different from the output of the example, so generalization-or-specialization is executed.

The root node and Node 3 match the training example, so both nodes execute generalization-or-specialization. Nodes 1 and 2 do not match the example so they are idle during the generalization-or-specialization process.

The root node has no input/value pairs that can be removed from its feature, so the root cannot create any features more general than itself. The root node creates three features, all more specific than its current feature, by adding input/value pairs from the training example. The features and their strengths for output *Manual* are as follows.

{(*Wind Speed, High*)}	1/3 = .33
{(*Wind Direction, Head*)}	1/2 = .50
{(*Visibility, Clear*)}	1/2 = .50

All three features are contradicted by the root node which has strength of 2/3 = .67 for output *Automatic*. Therefore, the root node has no viable new features to create and does not compete with other nodes.

Node 3 is completely specific. No additional input/value pairs can be added to its feature. Therefore, the new features created by Node 3 are all generalizations of its current feature created by removing input/value pairs. The features and their strengths for output *Manual* are as follows.

{(*Wind Speed, High*), (*Wind Direction, Head*)}	1/2 = .50
{(*Wind Speed, High*), (*Visibility, Clear*)}	1/2 = .50
{(*Wind Direction, Head*), (*Visibility, Clear*)}	1/1 = 1.0

The feature {(*Wind Direction, Head*), (*Visibility, Clear*)} is the strongest feature for Node 3. Node 3's candidate feature is not equal to that in any node, nor is it covered or contradicted by any node. There are no other nodes to compete with Node 3 for creation of a new node. Therefore, Node 4 is created using the new feature. The counters in Node 4 are initialized using the only example node that is matched by Node 4 which is Node 3. The resulting network is shown in « GoSNet4».

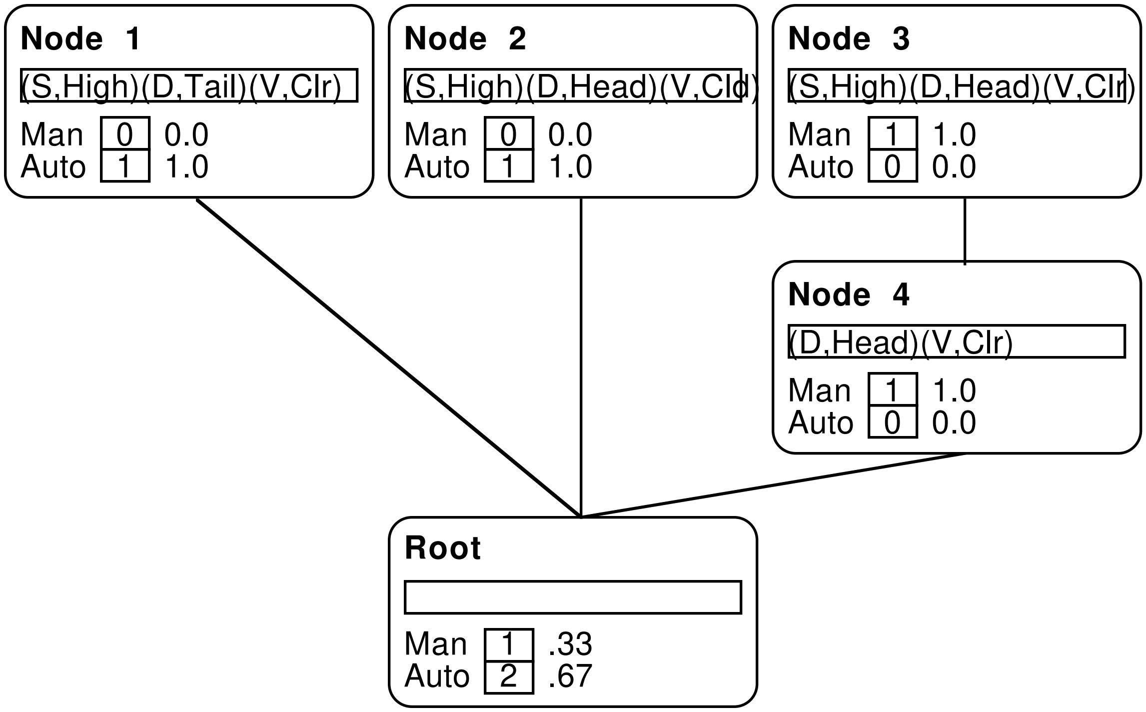

Figure 6.12. Network after third example

Example 4: (*Low, Head, Clear*) → *Manual*

Example 4 is given to the network. The network executes with the input (*Low, Head, Clear*). The root node and Node 4 both match the example. The strength of the root is 2/3 = .67 and the output is *Automatic*. The strength of Node 4 is 1/1 = 1 and the output is *Manual*. Since Node 4 is stronger, the output of the network is *Manual*.

A new example node (Node 5) is created. The root node, Node 4, and Node 5 update their counters giving the result shown in « GoSNet5». Since the output of the network is the same as the example, generalization-or-specialization is not done. The resulting network is shown in « GoSNet5».

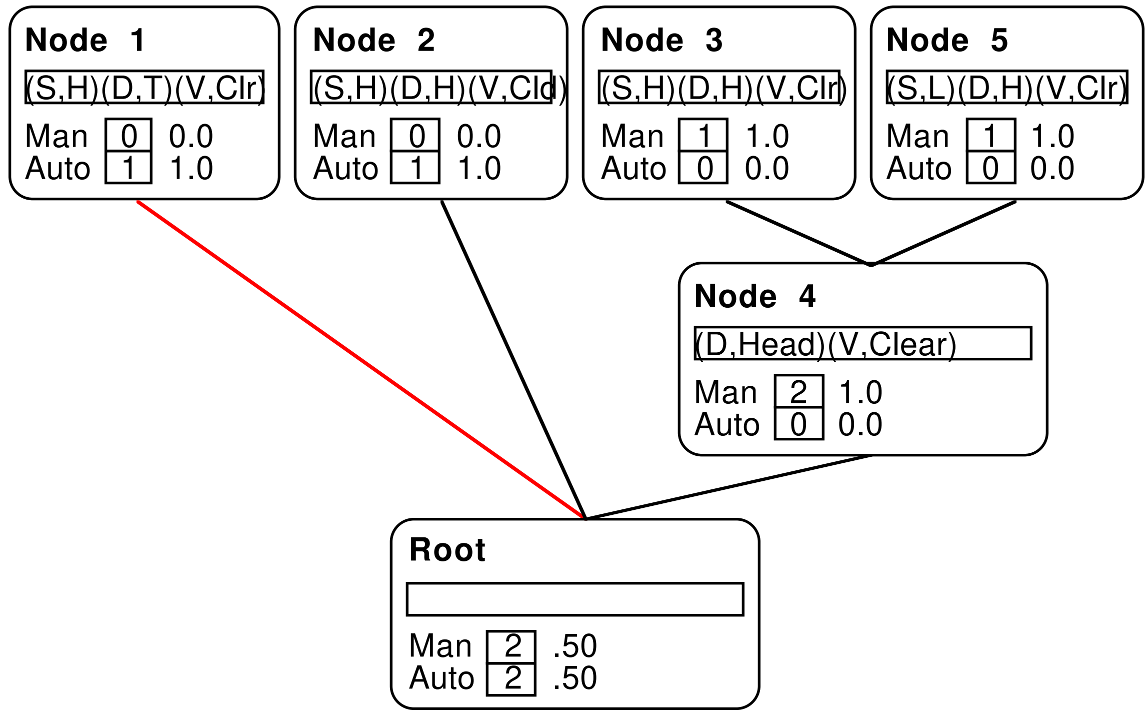

Figure 6.13. Network after fourth example

Example 5: (*Low, Head, Cloudy*) → *Automatic*

Example 5 is given to the network. The network executes with the input (*Low, Head, Cloudy*). The root is the only node that matches the example. The output of the network is the output of the root which is *Don't-Know*.

An example node is created (Node 6). The root and Node 6 update their counters to the values shown in « GoSNet6». Since the output of the network is *Don't-Know*, no additional nodes are created.

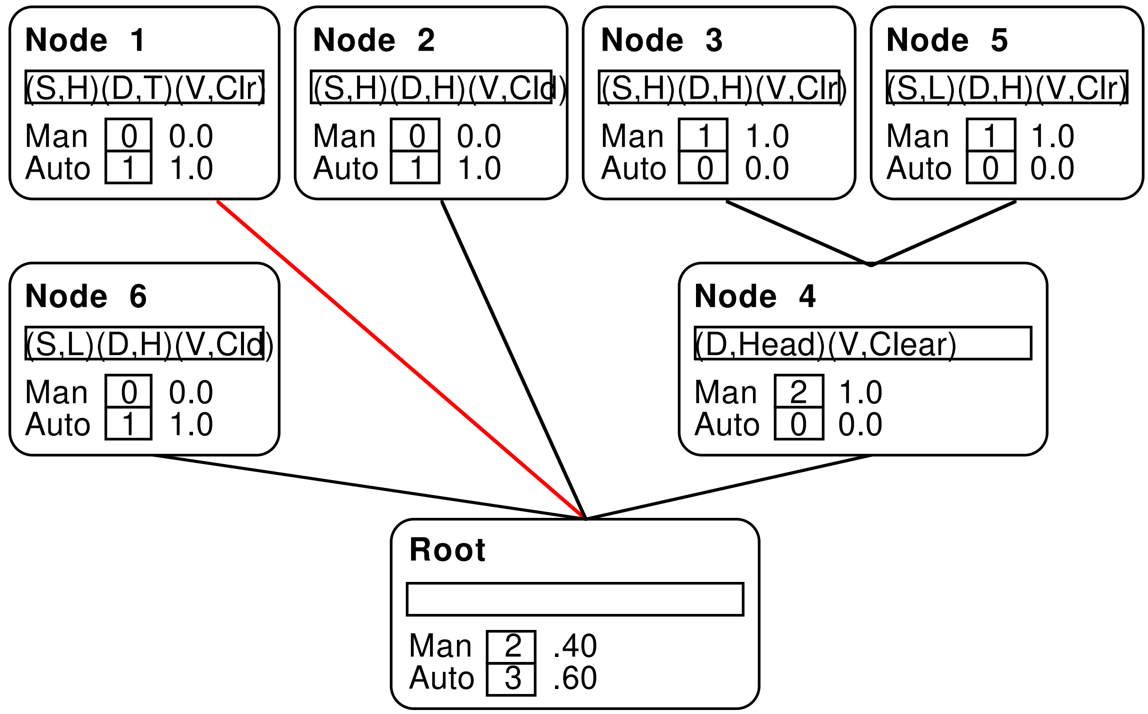

Figure 6.14. Network after last example

Note that the final network in this example implements exactly the same function as that in the last example. However, the network here is simpler with fewer nodes. In fact, for this training set, the generalization-or-specialization method requires no deletion of nodes.

Because GAS learning using generalization-or-specialization creates a node using the best new feature, it should learn with fewer mistakes in new feature creation than GAS learning using feature-generalization and feature-specialization. Generalization-or-specialization is expected to learn (1) problems that have few general features and (2) problems that have many specific features since, like the first GAS model, generalization-or-specialization combines both GS and SG learning.

Chapter 7
Interconnect Architectures

This chapter explains how a number of nodes are connected together into a network that can carry out the algorithms of network execution and learning described in previous chapters. The first architecture, *Broadcast/Gather*, is a simple approach that can be implemented easily on existing parallel systems with static topologies. The second architecture, *General/Specific*, is a more complicated approach that requires dynamic interconnect in a full hardware implementation. However, the second approach provides increased performance in learning.

7.1. Broadcast/Gather Architecture

Broadcast/Gather is an approach that uses only two communication operations between nodes, broadcast and gather. *Broadcast* is the operation of sending data to all nodes in the system. *Gather* is the operation of getting data from all nodes in the system and combining that data together. Broadcast and gather can be implemented on several different network topologies including trees and hypercubes.

7.1.1. Broadcast

Broadcast is the operation of communicating the same data object to every node in the system. Some models may use a restricted form of broadcast where an object is only given to nodes that meet some test. There are many examples of the use of broadcast:

1. When the network must execute and produce an output, the input pattern is broadcast to all nodes in the network.
2. When an example is given during learning, that example is broadcast to all nodes in the network.
3. When a new candidate feature is created, the feature is broadcast to the network to determine if a node already exists with the feature.

Broadcast can be implemented in several ways. Here we give only the binary tree implementation. The method can be generalized so that broadcast can be done on any type of tree.

A *tree* consists of *nodes* and *connections* between nodes. Each node in a tree has a single *parent*. Each node in a binary tree has 0, 1, or 2 *children*. A connection is made between each pair of nodes that has a parent/child relationship. One node in the tree is the *root*. The root has no parent. Nodes with no children are called *leaves*.

An example of a tree is shown in « Tree». The root is typically shown at the top of the tree. The children of nodes at level i in the tree are drawn at level $i + 1$ or one level lower in the tree. The parents are drawn at level $i - 1$ or one level higher. The parent of Node 1 is the Root. Node 1 is a child of the Root. The children of Node 1 are Nodes 3 and 4. Node 1 is the parent of Nodes 3 and 4. Nodes 3 through 6 in the tree are leaf nodes.

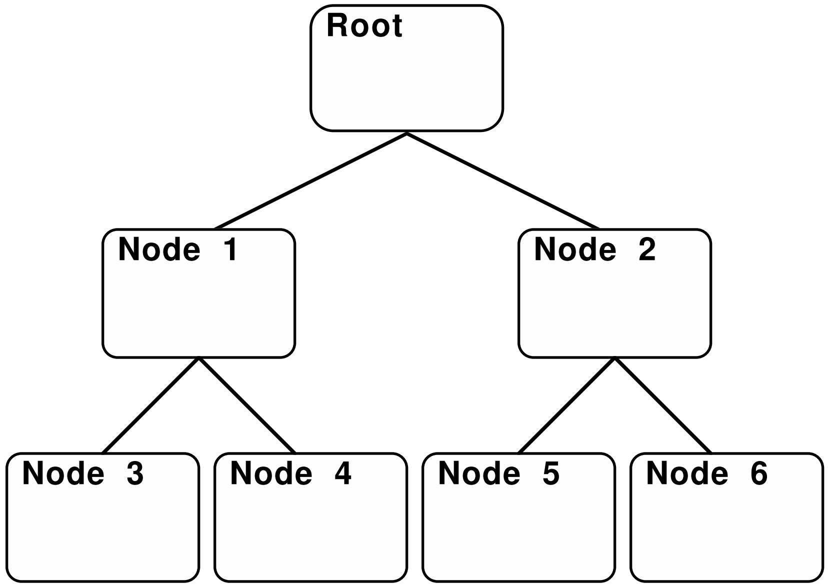

Figure 7.1. Binary Tree

Broadcast on a tree begins when the root node receives data from the environment. The root node stores the data locally. The root then sends the data to both of its children. The children of the root store the data locally. The children then send the data to their children. The process repeats until the data reaches the leaf nodes.

Note that the number of steps required to broadcast on a tree is proportional to the number of levels in the tree. The broadcast operation is O(d), where d is the number of levels in the tree. It can be shown that d is O(log n) for a balanced tree, where n is the number of nodes in the tree.

An example of broadcast on a tree is shown in « TreeBC». The training example (*Low, Tail, Clear*) → *Automatic* is to be broadcast to all the nodes in the network. The training example is first given to the root node as shown in « TreeBC».

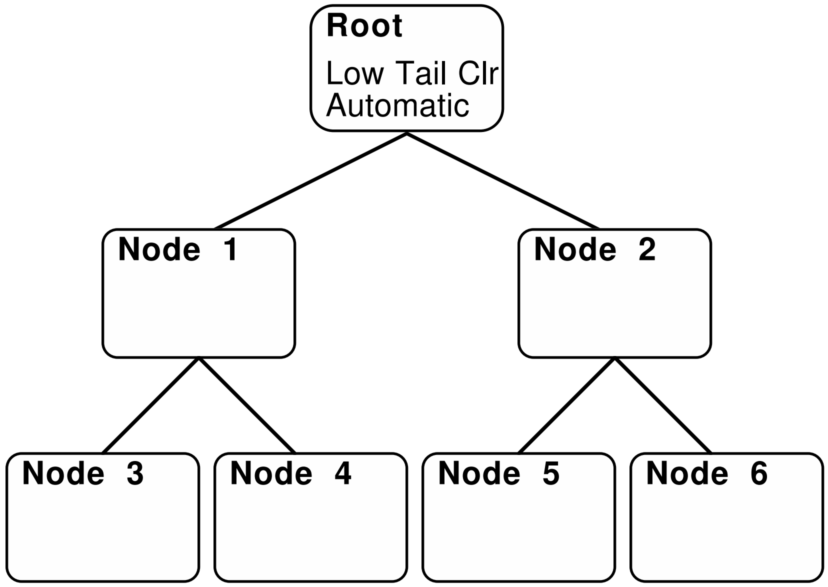

Figure 7.2. Broadcast on a Binary Tree

The root node stores the training example and then sends the training example to its children, Nodes 1 and 2, as shown in « TreeBC2». Nodes 1 and 2 store the training example and send the example to their children, Nodes 3 through 6, which are leaf nodes.

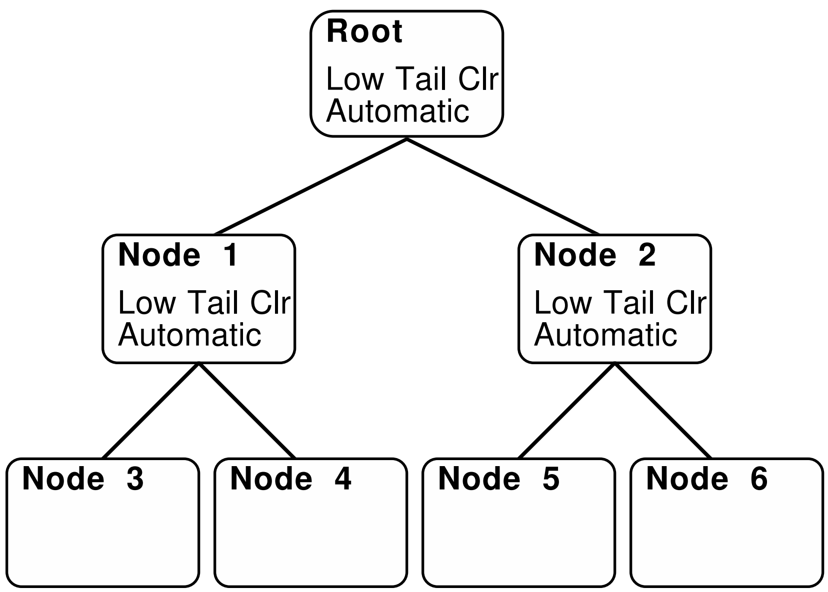

Figure 7.3. Broadcast to the children of the root

Broadcast on a tree can be broken into two parts, (1) the part done by the system, and (2) the part done by each node. These two parts are shown in the following routines.

System Broadcast(Input: *Data*)

```
        Send Data to the root node.
End.
Node Broadcast()
        Take Data from parent node.
        Store Data into self.
        If children exist then
                Send Data to child nodes.
        End.
End.
```

7.1.2. Gather

Gather is the operation of taking data from all nodes in the system and combining that data into a single datum. The combining function used in a gather operation will usually combine any number of pieces of data without regard to order. For example when obtaining a sum of several numbers, the individual numbers can be added together in any order and the same result is obtained. Some models may use a restricted form of gather where data is only taken from nodes that meet some test. There are many examples of the use of gather:

1. When the network must execute and produce an output, the output of each node is gathered and combined using a maximum function on the strength of the output. The final output of the gather operation is thus the strongest output of all the nodes.
2. When several nodes create new candidate features, the features are gathered and combined using a maximum function on the goodness measure for the feature. The best new candidate feature is the output of the gather operation.

Gather can be implemented in several ways. Here we give only the binary tree implementation. The method can be generalized so that gather can be done on any type of tree.

Gather on a tree is done starting at the leaves. The leaf nodes send their data to their parent nodes. The parent nodes combine the data received from the children and their own local data using the combining function. Note that for a binary tree the combining function takes three inputs (two children and the local data) and produces one output. The parent nodes then send the result of the combining function to their parent nodes. The parent nodes at the next level perform the combining function and send the result to the next level. The process repeats until the data reaches the root node.

Note that the number of steps required to gather on a tree is proportional to the number of levels in the tree. The gather operation is $O(d)$, where d is the number of levels in the tree. It can be shown that d is $O(\log n)$ for a balanced tree, where n is the number of nodes in the tree.

An example of gather on a tree is shown in « TreeGather». Each node contains its output and the strength of the output. This data is gathered using a maximum function to obtain an output for the network.

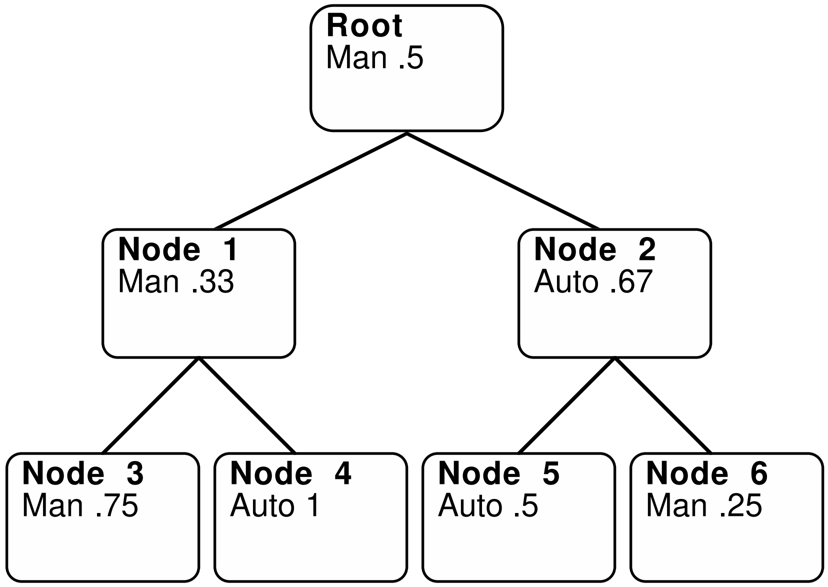

Figure 7.4. Gather on a Binary Tree

The first step is for the leaf nodes, Nodes 3 through 6, to send their data to their parents, Nodes 1 and 2. Node 1 performs the maximum function on the values .75, 1, and .33 from Nodes 3, 4, and 1, respectively. The maximum value of 1 along with the output *Auto* is temporarily stored in Node 1 as shown in « TreeGather2». Node 2 also performs the maximum operation. The maximum strength in this case is the strength from Node 2 itself. The maximum is temporarily stored in Node 2 as shown in « TreeGather2».

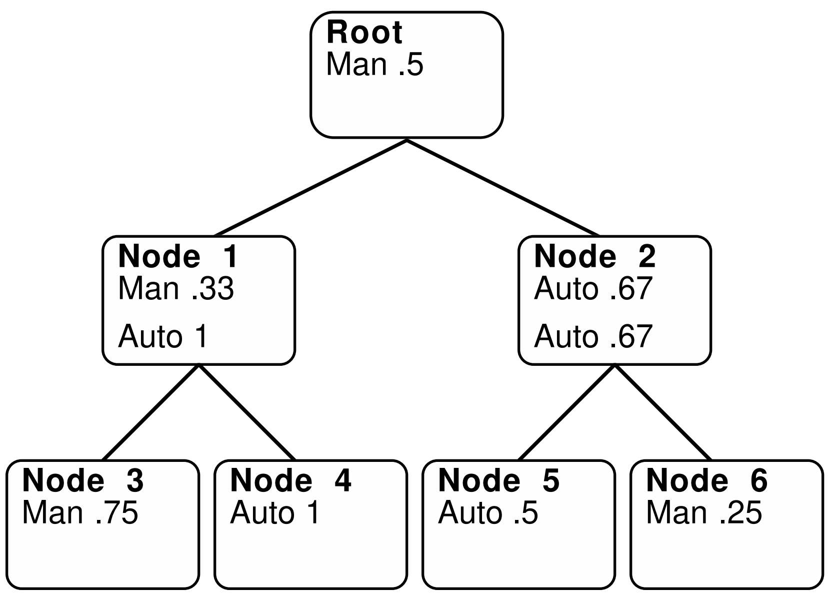

Figure 7.5. Gather from the leaves

The maximums stored in Nodes 1 and 2 are passed up the tree to the root node. The root node performs the maximum function on the two values passed up from the children and the value stored in the root. The maximum of the three strengths is 1 with output *Auto*. The maximum is temporarily stored in the root node and sent out as the output of the network as shown in « TreeGather3».

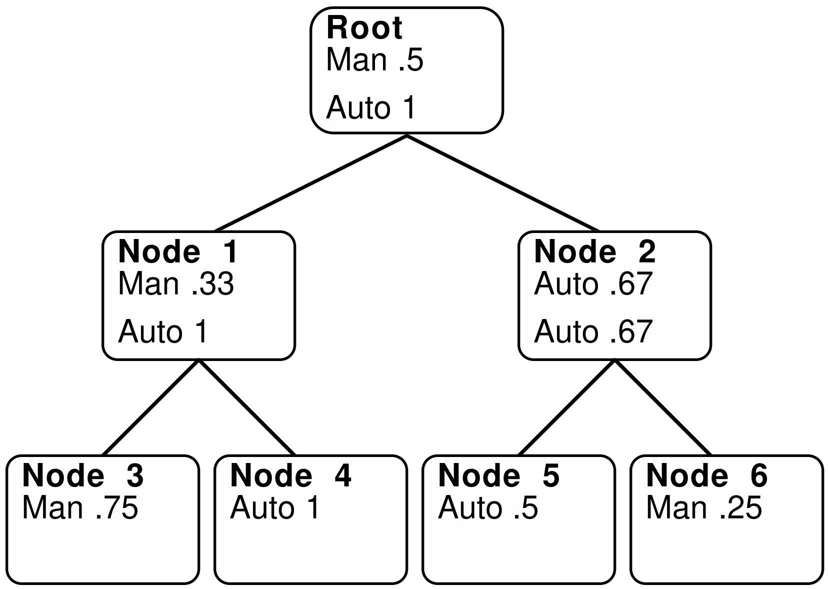

Figure 7.6. Gather to the root

After the gather operation on the tree is complete, the system takes the resulting data from the root node. The system operation is given in the routine below followed by the routine executed by all nodes in the tree.

System Gather(Output: *Data*)
    Get *Data* from the root node.
End.
Node Gather()
    If children exist then
        Get *Data* from child nodes.
        Compute combining function on data from child nodes and data from self.
        Send *Result* to parent node.
    Else
        Send *Data* from self to parent node.
    End.
End.

7.1.3. EN Strength Estimation using Broadcast/Gather

EN Strength Estimation requires communication between nodes to produce a result since the strength information is distributed throughout the example nodes. One method for providing the required communication is to use broadcast/gather.

First, the new candidate feature is broadcast to all nodes in the network. The example nodes that are matched by the new candidate feature respond by providing two counts, the count for the output that is given in the training example and the total count. The gather operation performs two separate sums, one for the output count and one for the total count. Note that only example nodes that are matched by the new feature provide data for the gather operation. The result of the gather operation is the two sums. The sum of the count for the output is divided by the sum of the total count to give the estimated strength of the new feature.

7.2. General/Specific (GS) architecture

This section describes a network topology where nodes are connected when they have a general to specific or a specific to general relationship. The topology allows learning to be done more efficiently. Learning is O(log m), where m is the number of network nodes, when using the GS topology compared to O(m log m) with broadcast/gather. (The details of the complexity results are given in Chapter 8.) This section describes the GS interconnection structure. The algorithms for creation of connections are described. Algorithms for finding equal features using the network are given.

During learning suppose n nodes are attempting to create a new feature. Each time a node considers creating a new feature it must determine if the feature is already contained in another node in the network. Using a broadcast/gather architecture there can be n nodes broadcasting on the network at one time. Each node may consider the creation of several new features for each training example. The process of testing for the existence of features in the network can become very costly.

The test for feature existence should be localized so that not all nodes are involved in the broadcast that searches for an equal feature. When a node creates a new feature, the new feature is either a specialization or a generalization of the current feature at the node. Suppose that each node A is connected to all nodes that contain specializations of the feature stored in A. If node A is doing GS learning and creating specializations of its current feature, then node A can find any existing feature that is equal to a new candidate feature by searching only the nodes that are connected to node A. Similarly if node A is connected to all nodes that contain generalizations of the feature contained in A and if node A is doing SG learning then node A need only search connected nodes to find existing features that are equal to new candidate features.

7.2.1. Topology

A network using the GS topology is a set of nodes N and a set of edges E. Let A, B, and C be nodes from N. Let F_a, F_b, and F_c be the features for nodes A, B, and C, respectively. Suppose the network is defined such that there is an edge from A to B if-and-only-if F_a is a generalization of F_b.

An example of such a network is shown in « GS». The root node is connected to all other nodes since the root is a generalization of every possible feature. Node 1 is a generalization of both Nodes 2 and 3. Note that the connections between the root and Nodes 2 and 3 are essentially redundant. These connections can be obtained by computing the transitive closure of other connections. The number of connections required in the network can be reduced by removing these redundant connections.

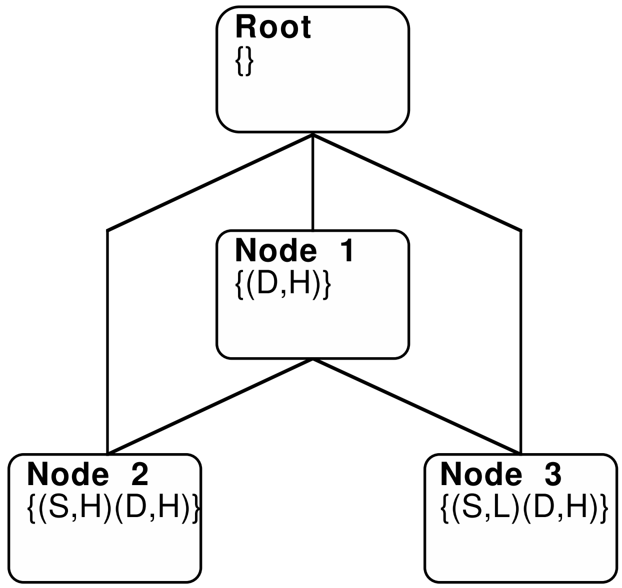

Figure 7.7. General/Specific Architecture

The definition for edges is modified to remove edges that are redundant. An edge exists from A to B if-and-only-if F_a is a generalization of F_b and there is not some other node C, such that F_c is a generalization of F_b and F_c is a specialization of F_a. In other words, two nodes are connected if the feature in one node is either the most specific generalization or the most general specialization of the feature in the other node.

An example of a network without redundant connections is shown in « GS2». Note that a node can still have multiple generalizations. Node 3 is connected to Nodes 1 and 2 since both nodes are generalizations of Node 3 but Nodes 1 and 2 are not generalizations or specializations of each other. The resulting network is thus a directed acyclic graph (DAG) rather than a tree.

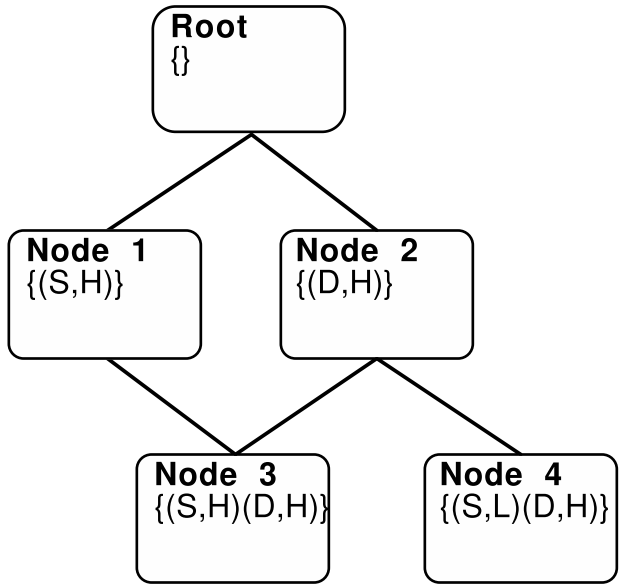

Figure 7.8. General/Specific Architecture

7.2.2. Adding Nodes

When adding a new node to a GS network the new node must be connected to existing nodes that are the most specific generalizations of the new node. The new node must also be connected to existing nodes that are the most general specializations of the new node. It is desirable to reduce the number of existing nodes that must be searched in order to connect a new node. By using information that is already known about a new node the new node can be connected in a more efficient manner.

When a new node is known to be a specialization of an existing network node *N*, specializations of the new node can be found efficiently by starting at node *N* in the network. This process is described in Section 7.2.2.1. When a node is known to be a generalization of an existing network node *N*, generalizations of the new node can be found efficiently by starting at node *N* in the network. This process is described in Section 7.2.2.2. Once a node is linked into the network, other nodes previously existing in the network may have become redundant. Section 7.2.2.3 shows how these links are removed. Some new nodes may have no specialization or a specialization of the node may not be known. Generalization links for such a node can be found using the procedure described in Section 7.2.2.4.

The remaining four sections (7.2.2.5 to 7.2.2.8) describe how the four building block routines are combined to completely link four different types of nodes into the network.

1. Single-Input Nodes.
2. Example Nodes.
3. Specializations of existing nodes.
4. Generalizations of existing nodes.

7.2.2.1. Specialization Links

The *specialization DAG* of a node *N* is the set of nodes and connections starting at *N* and going in the direction of specialization.

The routine *LinkSpec* shown in *« LinkSpec»* finds the most general specializations of a new node *N* given an existing node *G* in the network that is a generalization of *N*. By using node *G* that is already connected in the network the search is narrowed to only those nodes in the specialization DAG starting at node *G*. Since *G* is a generalization of *N*, *N* is a specialization of *G*, and all specializations of *N* are guaranteed to be contained in the specialization DAG of *G*.

The routine searches the specialization DAG of *G* starting at *G* and proceeding in a general to specific direction. Let *T* be a node in the specialization DAG of *G* that is also found to be a specialization of *N*. Since the search is general to specific, *T* is guaranteed to be a most general specialization. Therefore, Node *T* is linked to node *N*.

Node *T* is not guaranteed to be a most general specialization if the network allows redundant links. If redundant links are allowed, *G* can be linked to two specializations, S_1 and S_2, where S_1 is a specialization of S_2. The algorithm would link both nodes to *N* but the link to S_1 would be redundant.

All nodes in the specialization DAG beyond *T* are specializations of *T* so any links between those nodes and node *N* are redundant links. Therefore, the search of the DAG does not continue beyond node *T*.

Let *F* be a specialization of *G* that is not a specialization of *N*. If node *F* overlaps node *N* it is possible that some specialization of *F* is a specialization of *N*. Therefore, the search of the DAG is continued to the next level by calling the routine recursively starting with node *F*. If node *F* does not overlap (is discriminated with) node *N* then there is no specialization of *F* that can ever be a specialization of *N*, so the search of the DAG does not continue.

LinkSpec(*G*, *N*).

1. For all nodes *S* that are specializations connected to node *G*.
   2. If *S* is a specialization of *N* then
      3. If a link between *S* and *N* is not a redundant link then
         4. Link *S* and *N*.
      5. End.
   6. else If *S* overlaps *N* then
      7. LinkSpec(*S*, *N*).
   8. End.
9. End.

End.

Figure 7.9. Link a Node to its Specializations

« LinkSpecExample» shows an example of linking a new node to its specializations. Suppose Node 2 has just created Node 8 which is a new specialization of Node 2. Since Node 2 is creating a specialization it is known that Node 2 is a generalization of Node 8 and the LinkSpec routine can be used.

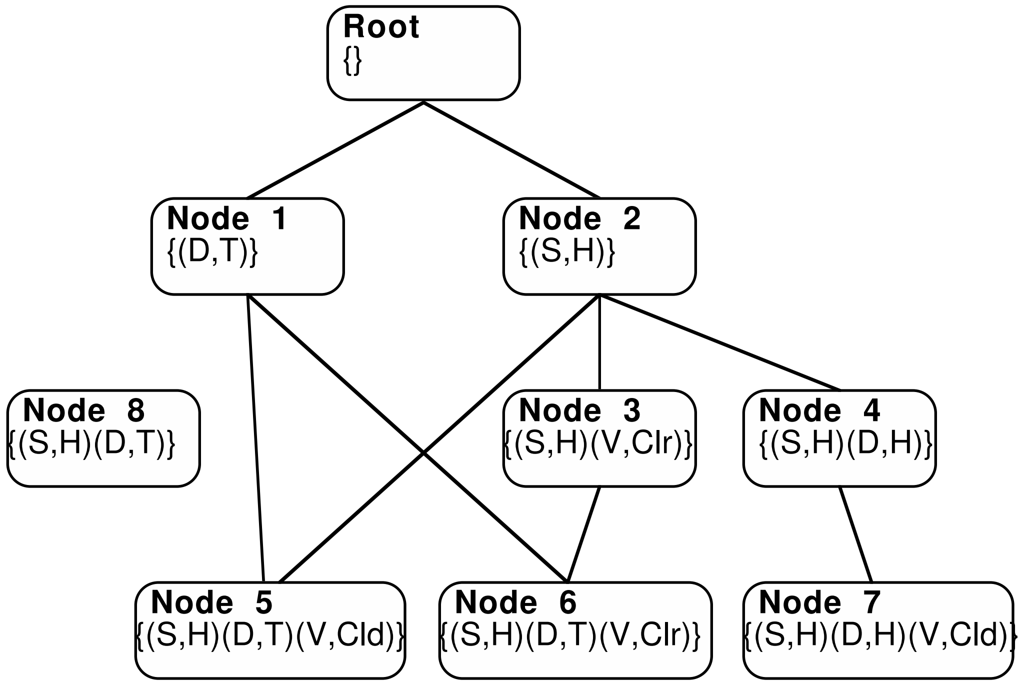

Figure 7.10. LinkSpec Example

The routine starts at Node 2. Nodes 3, 4, and 5 are compared to Node 8 since they are specializations connected to Node 2. Node 5 is found to be a specialization of Node 8, so Node 5 is linked to Node 8 as a specialization. Node 3 is not a specialization of Node 8. However, Node 3 does overlap with Node 8. Therefore, LinkSpec is called recursively starting at Node 3. Specializations of Node 3 are compared with Node 8. Node 6 is found to be a specialization of Node 8, so Node 6 is linked to Node 8. No other specializations of Node 3 exist, so the recursion terminates and processing continues back with specializations of Node 2. The remaining specialization of Node 2, Node 4, is discriminated with Node 8 by the *Wind Direction* input. Therefore LinkSpec is not called recursively on Node 4 and the linking process is complete. The resulting network is shown in « LinkSpecExample2». Note that Node 8 must still be linked to its generalizations.

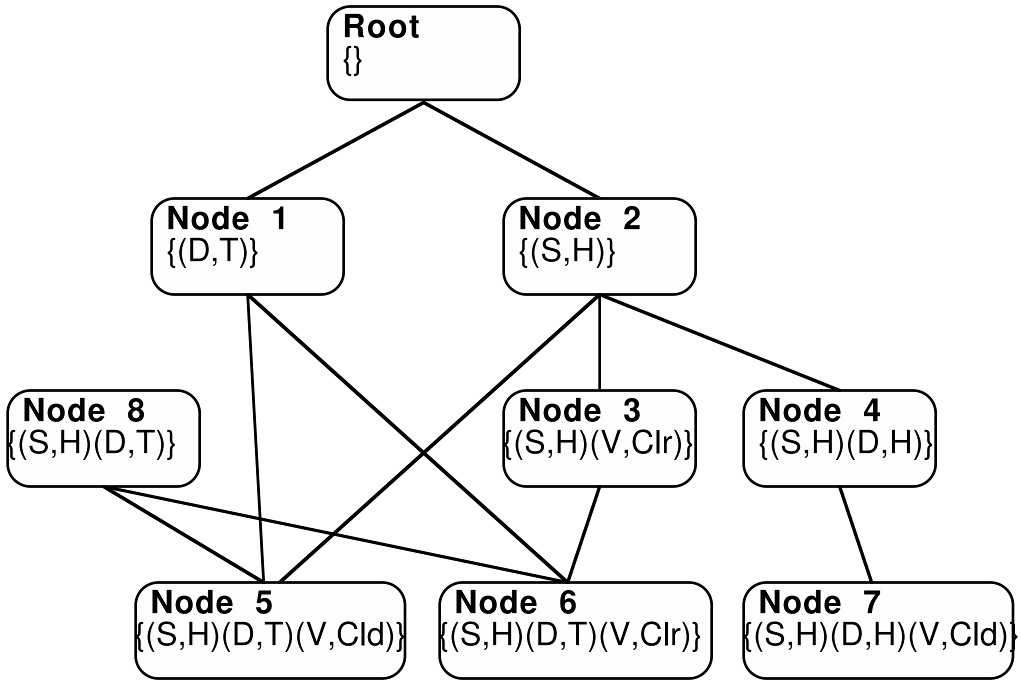

Figure 7.11. Completed Specialization Links

7.2.2.2. Generalization Links

The *generalization DAG* of a node *N* is the set of nodes and connections starting at *N* and going in the direction of generalization.

The routine *LinkGen* shown in « LinkGen» finds the most specific generalizations of a new node *N* given an existing node *S* in the network that is a specialization of *N*. By using node *S* that is already connected in the network the search is narrowed to only those nodes in the generalization DAG starting at node *S*. Since *S* is a specialization of *N*, *N* is a generalization of *S*, and all generalizations of *N* are guaranteed to be contained in the generalization DAG of *S*.

The routine searches the generalization DAG of *S* starting at *S* and proceeding in a specific to general direction. Let *T* be a node in the generalization DAG of *S* that is also found to be a generalization of *N*. Since the search is specific to general, *T* is guaranteed to be a most specific generalization. Therefore, Node *T* is linked to node *N*.

Node *T* is not guaranteed to be a most specific generalization if the network allows redundant links. If redundant links are allowed, *S* can be linked to two generalizations G_1 and G_2, where G_1 is a generalization of G_2. The algorithm would link both nodes to *N* but the link to G_1 would be redundant.

All nodes in the generalization DAG beyond *T* are generalizations of *T* so any links between those nodes and node *N* are redundant links. Therefore, the search of the DAG does not continue beyond node *T*.

Let *F* be a generalization of *S* that is not a generalization of *N*. There may be some generalization of *F* that is a generalization of *N*. If *F* and *N* have no input/value pair in common (the intersection of the features contained in *F* and *N* is the empty set), then the only generalization of *F* that is a generalization of *N* is the root node. Therefore, when *F* and N have no input/value pair in common the search for generalizations is not continued beyond node F. If F and N have a common input/value pair, the search of the DAG is continued to the next level by calling LinkGen recursively starting with node F. If, after the entire search is completed, no generalizations of N are found, then N must be linked to the root node.

LinkGen(*S*, *N*).

1. For all nodes G that are generalizations connected to node S.
    2. If G is a generalization of N then
        3. If a link between G and N is not a redundant link then
            4. Link G and N.
        5. End.
    6. else If $G \cap N \neq \emptyset$ then
        7. LinkGen(G, N).
    8. End.
9. End.

End.

Figure 7.12. Link a Node to its Generalizations

« LinkGenExample» shows an example of linking a new node to its generalizations. Suppose Node 6 has just created Node 8 which is a new generalization of Node 6. Since Node 6 is creating a generalization, it is known that Node 6 is a specialization of Node 8 and the LinkGen routine can be used.

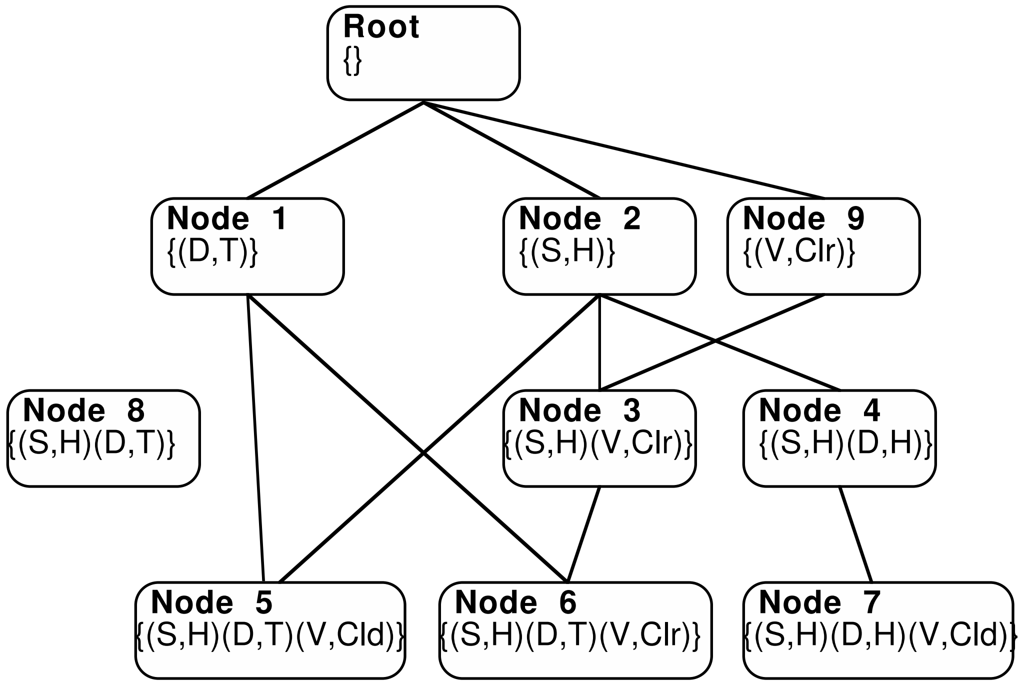

Figure 7.13. LinkGen Example

The routine starts at Node 6. Nodes 1 and 3 are compared to Node 8 since they are generalizations connected to Node 6. Node 1 is found to be a generalization of Node 8, so Node 1 is linked to Node 8 as a generalization. Node 3 is not a generalization of Node 8. However, Node 3 and Node 8 have a common input/value pair, (*Wind Speed, High*). Therefore, LinkGen is called recursively starting at Node 3. Generalizations of Node 3 are compared with Node 8. Node 2 is found to be a generalization of Node 8, so Node 2 is linked to Node 8. Node 9 is not a generalization of Node 8 and the two nodes have no input/value pairs in common. Therefore, LinkGen is not called recursively starting at Node 9. No other generalizations of Node 3 exist, so the recursion terminates and processing continues back with generalizations of Node 6. No other generalizations of Node 6 exist, so the linking process is complete. The resulting network is shown in « LinkGenExample2». Note that Node 8 must still be linked to its specializations.

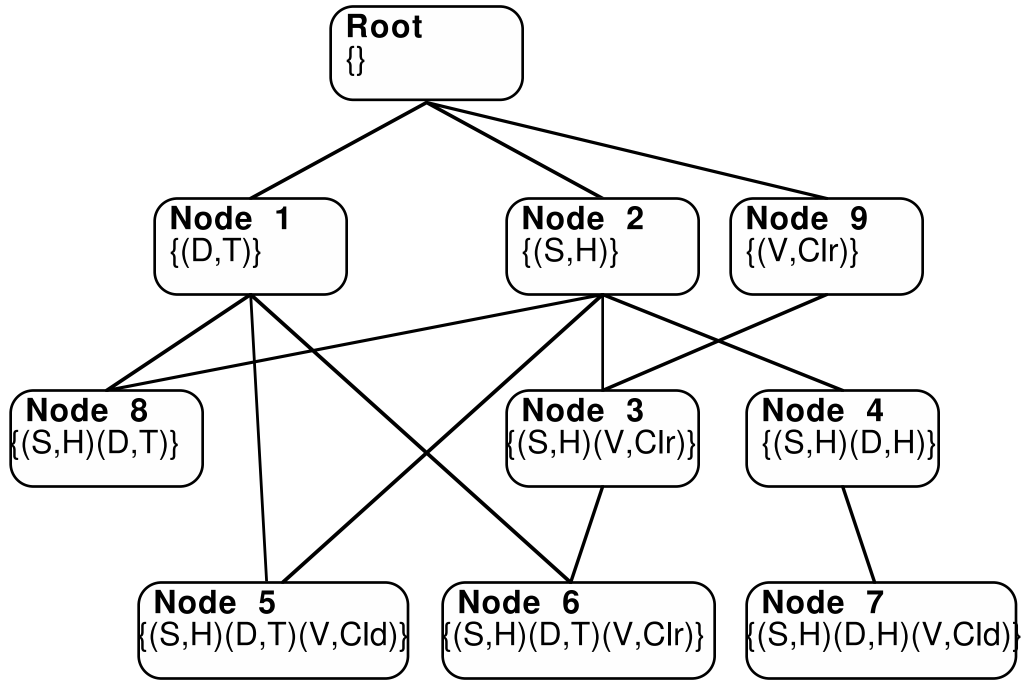

Figure 7.14. Completed Generalization Links

7.2.2.3. Removing Redundant Links

« RedunLinks» shows Node 8 after it has been linked to both specializations and generalizations as described in the previous two sections. Note that linking Node 8 into the network has created three redundant links (Node 1 to Node 5, Node 1 to Node 6, and Node 2 to Node 5). The routine *RemoveRedundant* must be run on Node 8 to remove the extra links.

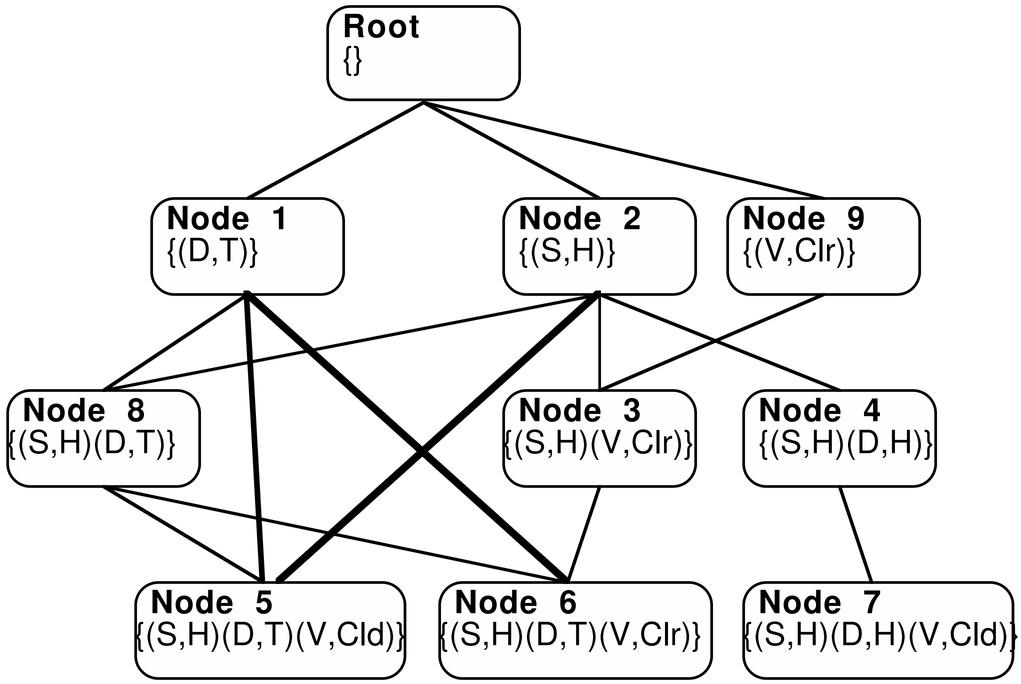

Figure 7.15. Redundant Links After Adding a New Node

RemoveRedundant considers all pairs of nodes (*G, S*), where *G* is a generalization directly connected to *N* and *S* is a specialization directly connected to *N*. If a link exists between *G* and *S,* the link is redundant and is removed. This simple routine keeps the network free from redundant links because it is run whenever a new node is added to the network. Thus, when the routine is run, the only redundant links in the network are those that were created by adding the new node. The routine is not intended to remove arbitrary redundant links.

RemoveRedundant(*N*).

1. For all nodes *G* that are generalizations connected to node *N*.
    2. For all nodes *S* that are specializations connected to node *N*.
        3. If *G* and *S* are linked then
            4. Unlink *G* and *S*.
        5. End.
    6. End.
7. End.

End.

Figure 7.16. Remove Redundant Links After Linking a New Node

The example shown in « RedunLinks» has four general/specific pairs for Node 8, (Node 1, Node 5), (Node 1, Node 6), (Node 2, Node 5), and (Node 2, Node 6). The first three pairs of nodes are linked so the links are removed giving the network shown in « RemoveRedunExample».

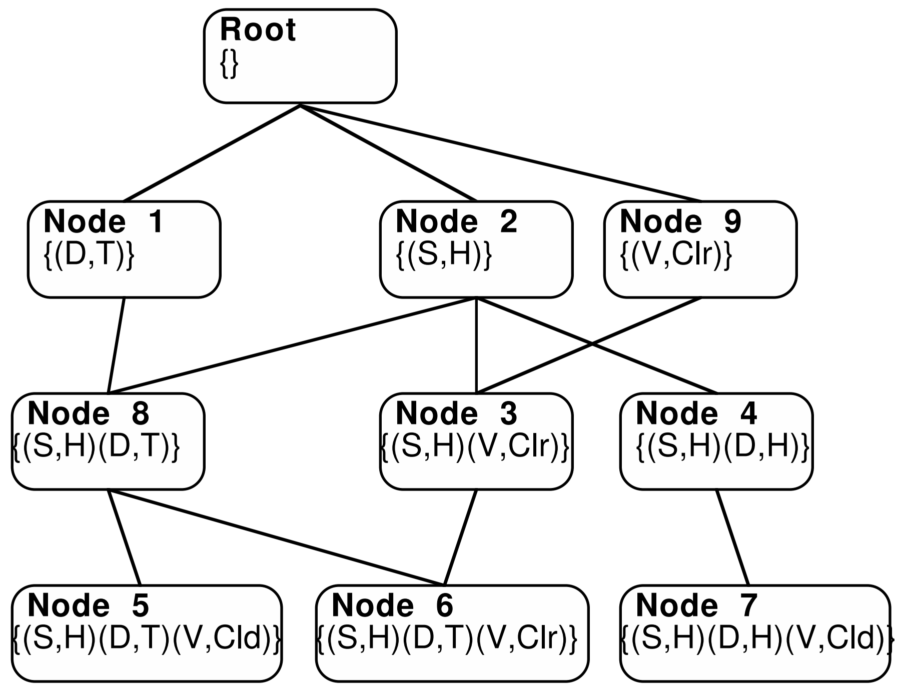

Figure 7.17. Network After Removing Redundant Links

7.2.2.4. Finding Generalization Links without a Specialization Node

The routine *LinkRoot* in « LinkRoot» finds the most specific generalizations of a new node without knowing any relationship between the new node and any existing nodes. The routine is called starting at the root node. The root node is known to be a generalization of all possible other nodes. However, there may be some specialization of the root *S* that is also a generalization of the new node. In such a case the new node should not be linked to the root because the root is not the most specific generalization.

The routine checks all specializations of the root node. If no specialization is found to be a generalization of the new node then the root is known to be the most specific generalization and the root is linked to the new node.

If any specialization *S* of the root node is found to be a generalization of the new node then LinkRoot is called recursively starting at *S*. A flag is set to indicate that the root node should not be linked to the new node. LinkRoot must be called recursively because node *S* may have a specialization that is a generalization of the new node. In such a case the new node should not be linked to node *S* because *S* is not the most specific generalization.

The specializations of node *S* are compared to the new node. If no specializations are found to be generalizations of the new node then node *S* is linked to the new node. Otherwise the process is repeated with the specialization by again calling LinkRoot recursively and node *S* is not linked.

Note that the search is limited to nodes that are generalizations of the new node. Thus the number of nodes visited is relatively small.

LinkRoot(*G*(search node), *N*(new node)).

1. *Link* = TRUE.
2. For all nodes *S* that are specializations connected to node *G*.
    3. If *S* is a generalization of *N* then
        4. LinkRoot(*S, N*).
        5. *Link* = FALSE.
    6. End.
7. End.
8. If *Link* and a link between *G* and *N* is not a redundant link then
    9. Link *G* and *N*.
10. End.

End.

Figure 7.18. Link Generalizations

« LinkRootExample» shows an example of adding a new node (Node 10) when no specialization of the new node exists in the network. The generalization links must be found by using the LinkRoot procedure. LinkRoot begins at the root node. Specializations connected to the root node, Nodes 1, 2, and 9, are compared to the new node. Node 1 is not a generalization of the new node, so it is not processed further. Node 2 is a generalization of the new node, so (1) a flag is set indicating that the root node should not be linked to the new node and (2) LinkRoot is called recursively starting at Node 2.

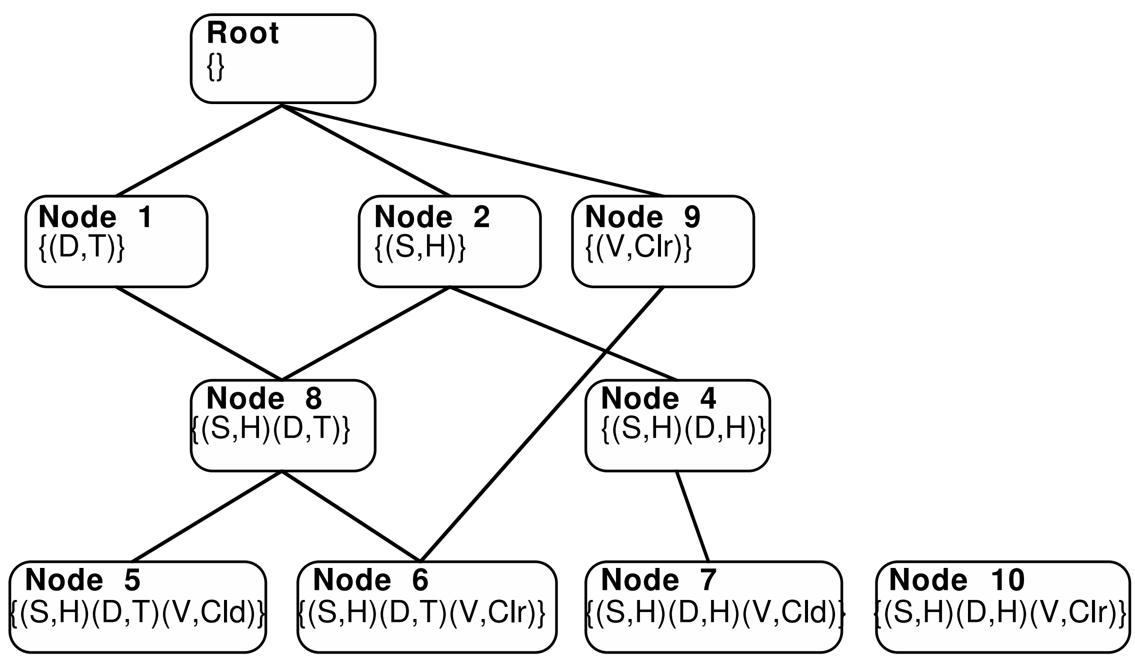

Figure 7.19. Finding Generalizations without a Specialization Node

Specializations of Node 2, Nodes 8 and 4, are compared with the new node. Node 8 is not a generalization of the new node, so it is not processed further. Node 4 is a generalization of the new node, so (1) a flag is set indicating that Node 2 should not be linked to the new node and (2) LinkRoot is called starting at Node 4.

Node 7, the only specialization of Node 4, is compared to the new node. Node 7 is not a generalization of the new node, so it is not processed further. The *Link* flag remains set to true since no specializations of Node 4 were found to be generalizations of the new node. Therefore, Node 4 is linked to the new node as a generalization. The recursion terminates and returns to the LinkRoot procedure at Node 2.

Node 2 has no more specializations to check, so that recursion terminates and returns to the LinkRoot procedure at the root node. A generalization was found beyond Node 2, so Node 2 is not linked to the new node.

Node 9, the remaining specialization of the root node, is a generalization of the new node, so LinkRoot is called recursively starting at Node 9. Node 6, the only specialization of Node 9, is compared to the new node. Node 6 is not a generalization of the new node, so it is not processed further. The *Link* flag remains set to true since no specializations of Node 9 were found to be generalizations of the new node. Therefore, Node 9 is linked to the new node as a generalization. The recursion terminates and returns to the LinkRoot procedure at the root node. No specializations of the root node remain to be processed, so the linking procedure has completed. The resulting network is shown in « LinkRootExample2».

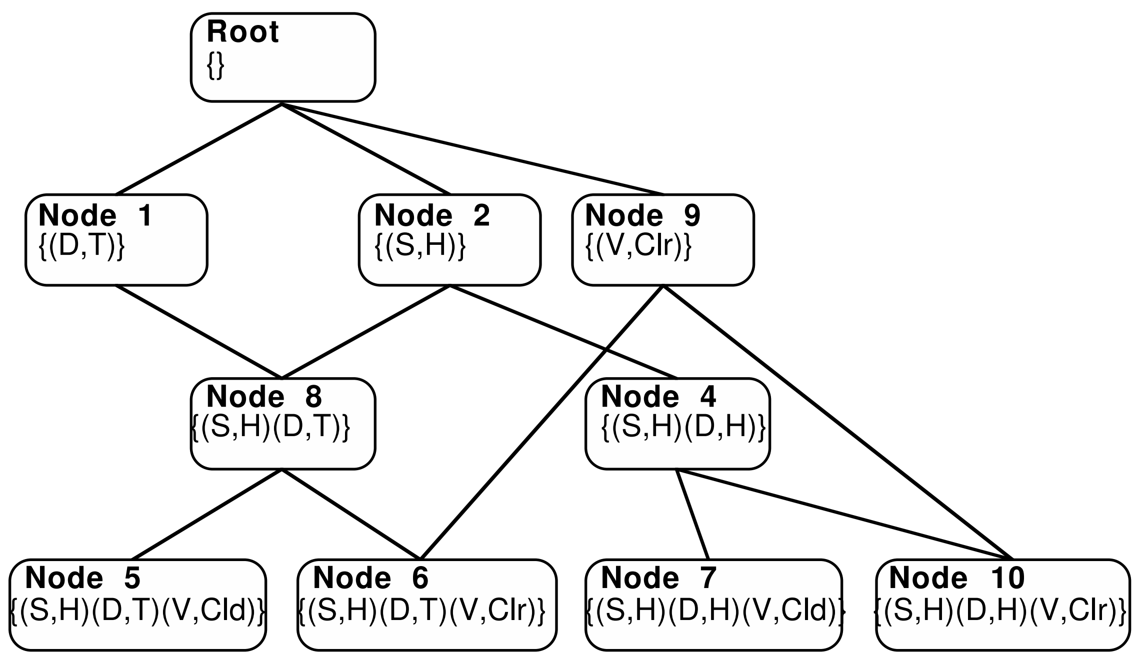

Figure 7.20. Network After Linking Generalizations

7.2.2.5. Single-Input Nodes

Since single-input nodes contain only a single input/value pair, it is known that single-input nodes have only a single generalization link to the root node. The only way a node with a single input/value pair can be generalized is by removing that input/value pair. This will always give the empty set as the result for the generalized feature. This feature is the feature that is always found in the root node. Therefore, a new single-input node is always linked directly to the root.

Specializations of the new node can then by found by calling LinkSpec starting at the root node. Finally, any redundant links must be removed by calling RemoveRedundant.

LinkSingleInputNode(*N*).

Link the Root Node and Node *N*.
LinkSpec(*Root, N*).
RemoveRedundant(*N*).

End.

Figure 7.21. Linking a Single-Input Node

7.2.2.6. Example Nodes

Example nodes are completely specific. No node can be a specialization of an example node. Therefore, no specialization links are possible to an example node. An example node does need to be linked to its generalizations. Since no specialization exists for an example node, the LinkGen routine cannot be used to find generalizations for the new node. When adding an example node the LinkRoot routine must be used.

Since example nodes have no specialization links, it is not possible for the addition of an example node to cause a previous link in the network to become redundant. Therefore, after adding an example node the RemoveRedundant routine does not need to be called.

LinkExampleNode(*N*).

LinkRoot(*Root, N*).

End.

Figure 7.22. Linking an Example Node

7.2.2.7. Specializations of Existing Nodes

When a new specific node is created in GS Learning, an existing node is always known to be a generalization of the new node. Therefore, specialization links can be created for the new node by calling the LinkSpec routine.

After specializations are linked to the new node, one of those specializations can be used to find generalizations of the new node using the LinkGen routine. If the new node does not have any specializations then the LinkRoot routine must be used to find generalizations of the new node. A specialization of a new node may not exist when doing GS learning without creating example nodes.

Finally, after the node is linked, redundant links must be removed by calling RemoveRedundant.

LinkSpecializationNode(*G*, *N*).

1. LinkSpec(*G, N*).
2. If node *N* is linked to a specialization *S* then
    3. LinkGen(*S, N*).
4. Else
    5. LinkRoot(*Root, N*).
6. End.
7. RemoveRedundant(*N*).

End.

Figure 7.23. Linking a Specialization of an Existing Node

7.2.2.8. Generalizations of Existing Nodes

When a new general node is created in SG Learning, an existing node is always known to be a specialization of the new node. Therefore, generalization links can be created for the new node by calling the LinkGen routine.

After generalizations are linked to the new node, one of those generalizations can be used to find specializations of the new node using the LinkSpec routine. A generalization always exists for every possible new general node since the root is a generalization of every other node.

Finally, after the node is linked, redundant links must be removed by calling RemoveRedundant.

LinkGeneralizationNode(*S*, *N*).

1. LinkGen(*S, N*).
2. Let *G* be one of the generalizations just linked to *N*.
3. LinkSpec(*G, N*).
4. RemoveRedundant(*N*).

End.

Figure 7.24. Linking a Generalization of an Existing Node

7.2.3. Deleting Nodes

When a node *N* is deleted, all pairs of nodes (*G, S*), where *G* is a generalization directly connected to *N* and *S* is a specialization directly connected to *N,* must be considered for linking. The process is essentially the reverse of removing redundant links when a node is added. If a link between *G* and *S* is not redundant (treating the node *N* as though it has already been removed from the network) then *G* and *S* are linked. If the node *N* is not treated as though is has already been removed from the network, then *N* will cause all links between all pairs (*G, S*) to appear redundant.

After the nodes around the node being deleted are linked together, then the node being deleted is unlinked from the network.

RemoveLinks(*N*).

1. For all nodes *G* that are generalizations connected to node *N*.
    2. For all nodes *S* that are specializations connected to node *N*.
        3. If a link between *G* and *S* is not a redundant link (assuming *N* is deleted) then
            4. Link *G* and *S*.
        5. End.
    6. End.
7. End.
8. For all nodes *G* that are generalizations connected to node *N*.
    9. Unlink *G* and *N*.
10. End.
11. For all nodes *S* that are specializations connected to node *N*.
    12. Unlink *S* and *N*.
13. End.

End.

Figure 7.25. Removing Links when Deleting a Node

An example of removing a node from a network is shown in « DeleteExample». Assume that Node 8 is deleted. Four new links must be considered for creation (Node 1, Node 5), (Node 1, Node 6), (Node 2, Node 5), and (Node 2, Node 6).

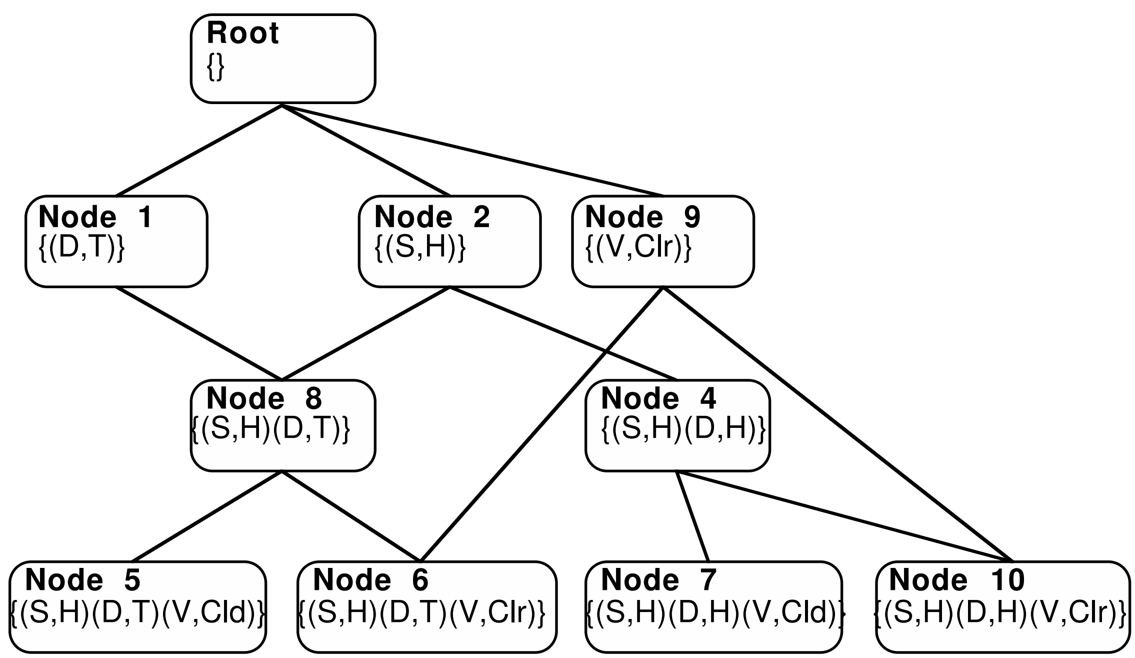

Figure 7.26. Network Before Deleting Node

None of the four new links are redundant so all four are created. Then Node 8 is removed from the network giving the result shown in « DeleteExample2».

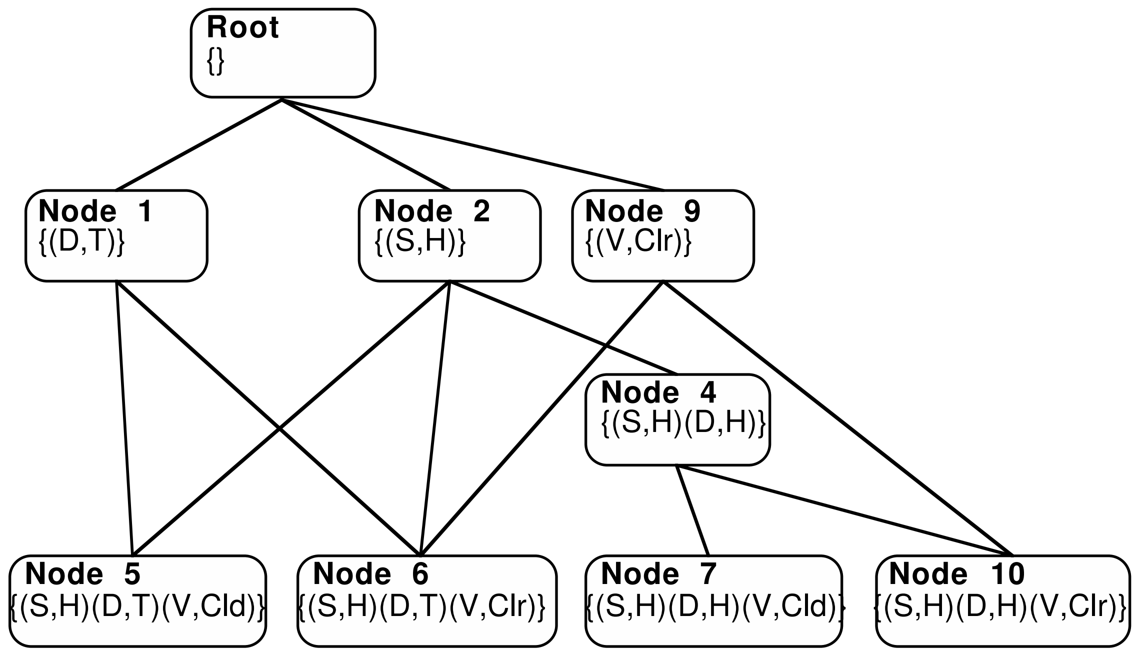

Figure 7.27. Network After Deleting Node

7.2.4. Using the Network

The structure of the GS network has been shown. Algorithms for creation and deletion of the network have been given. This section describes how the network is used for learning and execution.

The GS architecture must still be capable of broadcast to give the input pattern to the nodes during execution and the training example to the nodes during learning. The network must be capable of gather to collect the strongest output during execution and to determine the best new feature during learning. The first two sections (7.2.4.1 and 7.2.4.2) explain how broadcast and gather are done on the GS network topology.

The next three sections (7.2.4.3 to 7.2.4.5) describe how the GS topology is used to increase efficiency of the learning algorithms. The GS architecture improves efficiency in learning by:

1. Localizing the search for equal nodes.
2. Restricting the search for covering and contradicting nodes.
3. Localizing the estimation of strength using example nodes.

Finally, the last section (7.2.4.6) presents an alternative to example node strength estimation. This new strength estimation method is made possible by the GS topology.

7.2.4.1. Broadcast on a GS Architecture (DAG)

The GS interconnect creates a Directed Acyclic Graph (DAG) rather than a tree. The broadcast algorithm must be modified to work with a DAG. A DAG is similar to a tree except that in a DAG a node can have multiple parents. The GS DAG does have a single root node where broadcast operations begin and gather operations terminate.

The central issue concerning broadcast on a DAG is that the data or command that is being broadcast must only be given to each node one time. Since each node can have multiple parents the standard tree broadcast algorithm could give the data to a node as many times as the node has parents. The information broadcast to the network may be a command such as increment. If such a command is executed multiple times at some nodes the results will not be correct.

The sending of data is controlled by using an increasing message number in each message. A register in each node stores the message number from the last message that was accepted. When a new message is received the message number in the new message is compared with that stored in the node. If the number in the new message is greater than the number stored in the node then it is known that the node has not yet seen the new message. Therefore, the message is accepted and processed. If the number in the new message is less than or equal to the number stored in the node then it is known that the new message has already been seen by the node.

Because of the structure of the GS network, the broadcast of messages can be restricted to only those nodes that match the input pattern. For many operations requiring broadcast and gather, the only nodes in the network that need to participate are nodes that match the input pattern. The BroadcastMatching routine only broadcasts to nodes that match the input pattern. Since the network is organized from general to specific, once a node *N* is found to not match the input pattern, none of the nodes in the specialization DAG beyond *N* need to be visited. Thus the broadcast is efficient in terms of visiting only those nodes that must be visited.

A full broadcast routine can be obtained by removing the matching test on line 6 of BroadcastMatching.

SystemBroadcastMatching(*D*).

1. Increment *Message Number*.
2. Store *Message Number* in *D*.
3. BroadcastMatching(*Root, D*).

End.

BroadcastMatching(*N, D*)

1. If *Message Number* from *D* > *Message Number* from the last *D* then
    2. Store *Message Number* from *D* into *N*.
    3. Store *D* into *N*.
    4. Execute any commands contained in *D*.
    5. For all nodes *S* that are specializations connected to node *N*.
        6. If *S* matches the input pattern then
            7. BroadcastMatching(*S, D*).
        8. End.
    9. End.
10. End.

End.

Figure 7.28. Broadcast on a GS Network

7.2.4.2. Gather on a GS Architecture (DAG)

Gather on a GS topology is not very different from gather on a tree. The main consideration is that the data being gathered must only be sent to one parent in the GS network so that the data does not participate multiple times in the result. Most times a gather operation is associated with and follows a broadcast operation. Therefore, one approach is for each node involved in the gather to send its data to the parent from which it received the associated broadcast.

System Gather(Output: *Data*)

1. Get *Data* from the root node.

End.

Node Gather(*N, D*)

1. If node *N* has specializations then
    2. For all nodes *S* that are specializations connected to node *N*.
        3. If node *S* has data to send then
            4. Get *Data* from node *S*.
        5. End.
    6. End.
    7. Compute combining function on data from child nodes and data from self.
    8. Send *Result* to a single parent node.
9. Else
    10. Send *Data* from self to a single parent node.
11. End.

End.

Figure 7.29. Gather on a GS Network

7.2.4.3. Finding Equal Nodes on a GS Architecture

When an existing node *G* creates a new specialization *N* of itself, *G* can determine if a node equal to *N* already exists in the network by searching the specialization DAG starting at *G*. The search need not proceed beyond nodes in the specialization DAG of *G* that are not generalizations of *N*. Such nodes can never be specialized further such that the result is equal to *N*.

The routine *EqualSpec* first compares *N* to nodes that are connected to *G* as specializations. If such a node *S* is equal to *N* then an equal node has been found and the search terminates. Otherwise, if *S* is a generalization of *N* then EqualSpec is called recursively to check the next level of specialization nodes. The search terminates when either an equal node is found or none of the nodes on the frontier of the search are generalizations of the new node.

```
EqualSpec(G, N).
        1.  For all nodes S that are specializations connected to node G.
                2.  If S = N then
                        3.  Return TRUE.
                4.  End.
                5.  If N is a specialization of S then
                        6.  E = EqualSpec(S, N).
                        7.  If E = TRUE then
                                8.  Return TRUE.
                        9.  End.
                10.     End.
        11.     End.
        12.     Return FALSE.
End.
```

Figure 7.30. Finding Equal Specializations

For example, suppose the Root Node in « EqSpecExample» creates the new feature {(*Wind Direction, Tail*)}. Node 1 is found to be equal to the new feature by searching only direct specializations. Suppose the Root creates the feature {(*Wind Speed, Low*)}. None of the direct specializations are equal to the new feature. None of the direct specializations are generalizations of the new feature, so specializations at the next level do not need to be considered. Whenever a node creates a new feature by adding only a single input/value pair to its current feature, the node can determine if the new feature is redundant by only searching direct connections.

Suppose the Root Node creates the new feature {(*Wind Speed, High*), (*Visibility, Cloudy*)}. Node 1 is found to be neither equal to the new feature nor a generalization of the new feature. Node 2 is found to be a generalization of the new feature, so the search continues with specializations of Node 2. Node 5 is found not equal to the new feature and Node 5 is a leaf node, so the search cannot continue beyond Node 5. Nodes 3 and 4 are not equal to the new feature. Nodes 3 and 4 are not generalizations of the new feature, so the search terminates having not found the new feature redundant.

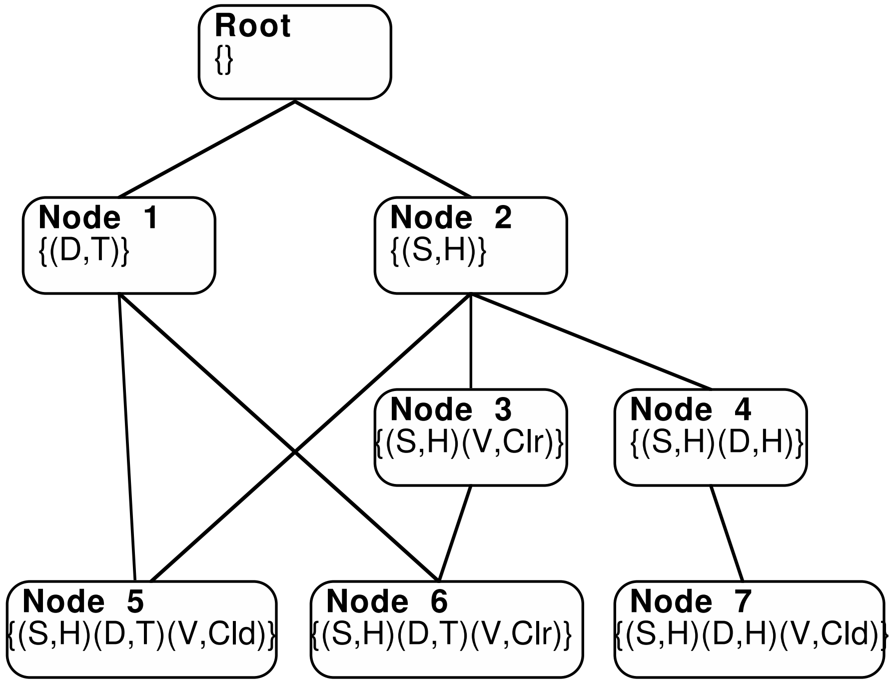

Figure 7.31. Finding Equal Nodes Example

When an existing node *S* creates a new generalization *N* of itself, *S* can determine if a node equal to *N* already exists in the network by searching the generalization DAG starting at *S*. The search need not proceed beyond nodes in the generalization DAG of *S* that are not specializations of *N*. Such nodes can never be generalized further such that the result is equal to *N*.

The routine *EqualGen* first compares *N* to nodes that are connected to *S* as generalizations. If such a node *G* is equal to *N* then an equal node has been found and the search terminates. Otherwise, if *G* is a specialization of *N* then EqualGen is called recursively to check the next level of generalization nodes. The search terminates when either an equal node is found or none of the nodes on the frontier of the search are specializations of the new node.

EqualGen(*S, N*).

```
1. For all nodes G that are generalizations connected to node S.
    2. If G = N then
        3. Return TRUE.
    4. End.
    5. If N is a generalization of G then
        6. E = EqualGen(G, N).
        7. If E = TRUE then
            8. Return TRUE.
        9. End.
    10. End.
11. End.
12. Return FALSE.
```

End.

Figure 7.32. Finding Equal Generalizations

For example, suppose Node 6 in « EqSpecExample» creates the new feature {(*Wind Speed, High*), (*Wind Direction, Tail*)}. Neither Nodes 1 or 3 are equal to the new feature. Neither Node 1 or 3 are specializations of the new feature, so generalizations at the next level do not need to be considered. Whenever a node creates a new feature by removing only a single input/value pair from its current feature, the node can determine if the new feature is redundant by only searching direct connections.

Suppose Node 6 creates the new feature {(*Visibility, Clear*)}. Node 1 is found to be neither equal to the new feature nor a specialization of the new feature. Node 3 is found to be a specialization of the new feature, so the search continues with generalizations of Node 3. Node 2 is found not equal to the new feature and Node 2 is not a specialization of the new feature, so the search terminates having not found the new feature redundant.

7.2.4.4. Finding Covering Nodes on a GS Architecture

It can be determined if a node is covered or contradicted by another node by comparing a node with its directly connected generalizations. Let *N* be a node in the network. Let *G* be a generalization connected to *N*. If the strength of *G* is greater than the strength of *N* then node *N* is covered. The node should either not be created or should be deleted from the network.

7.2.4.5. EN Strength Estimation on a GS Architecture

EN Strength Estimation can be done more efficiently on a GS architecture because the broadcast and gather to sum the counts of matching example nodes can be restricted to only visit nodes that match the new candidate feature. The new candidate feature is only broadcast to the specialization DAG of the new feature. All the example nodes at the leaves of the specialization DAG match the new feature. These example nodes participate in a gather operation to sum their counts into a strength measure.

7.2.4.6. Direct Specialization (DS) Strength Estimation

The GS network structure allows a new type of strength estimation for new candidate nodes. The new method is a variation on EN strength estimation but uses only the direct specializations of a new node rather than example nodes matched by a new node. Direct

Specialization (DS) strength estimation uses information that is local to the new node and thus reduces traffic in the network. However, because the GS network is a DAG rather than a tree, the resulting strength from DS strength estimation is not guaranteed to be the same as that obtained from EN strength estimation.

Let *N* be a new node created by specializing an existing node. The estimated strength for *N* using the DS method is calculated by summing the counters in the nodes that are directly connected to *N* as specializations. Let *S* be a specialization connected to *N*. Let *c* and *t* be temporary counters initialized to zero. The *T* counter from *S* is added to the counter t and the c_o counter from *S* is added to c. After summing all the direct specialization counters in this manner, the strength of *N* is given by *c*/*t*. Note that use of DS strength estimation requires the creation and linking of a new node rather than just the creation of a new feature as in EN strength estimation.

DS Strength Estimation (Input: *N* (new node)**; Output: *S*** (estimated strength of N)**).**

1. Set *c* and *t* to 0.
2. For all nodes *S* that are specializations connected to node *N*.
    4. Add *T* from node *S* to *t*.
    5. Add c_o from node *S* to *c*.
7. End.
8. Return the strength of *N* as *c*/*t*.

End.

Figure 7.33. Direct Specialization Strength Estimation

For example, suppose Node 6 in « DSExample» is a new candidate node being considered for creation. Nodes 4 and 5 are directly connected to Node 6 as specializations so the counters in these two nodes are added to the sum. The total for output *Automatic* is 1, for *Manual* is 3, and for the total counter is 4. The estimated strength of output *Automatic* for the new feature is 1/4 and the estimated strength of output *Manual* is 3/4.

Note that, using EN strength estimation, Nodes 1, 2, and 3 would be summed. The total for *Automatic* is still 1 but the total for *Manual* is 2. Thus, the two methods give different results. Node 2 contributes twice when using DS strength estimation because it is a specialization of both Nodes 4 and 5.

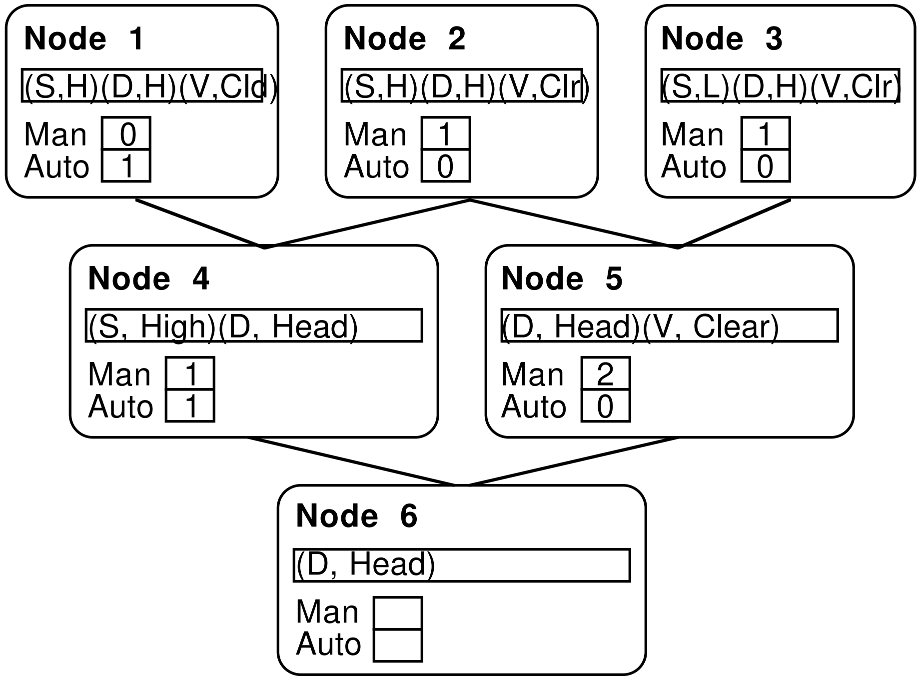

Figure 7.34. Strength Calculation using Direct Specializations

A new node created using this method is initialized as follows. All counters in the new node are initialized to 0. Let *S* be a specialization connected to the new node. The total counter *T* from node *S* is added to the total counter *T* for the new node. Also, for each output point *o,* the counter c_o from node *S* is added to counter c_o for the new node.

DS Node Initialization (*N* (new node)).

1. Set all counters in the new node *N* to 0.
2. For all nodes *S* that are specializations connected to node *N*.
    4. Add *T* from node *S* to *T* for node *N*.
    5. For all output points *o*.
        6. Add c_o from node *S* to c_o for node *N*.
    7. End.
9. End.

End.

« DSInitExample» shows how Node 6 from « DSExample» is initialized using Direct Specializations. Nodes 4 and 5 are direct specializations of Node 6 so the counters for nodes 4 and 5 are summed into the new nodes counters.

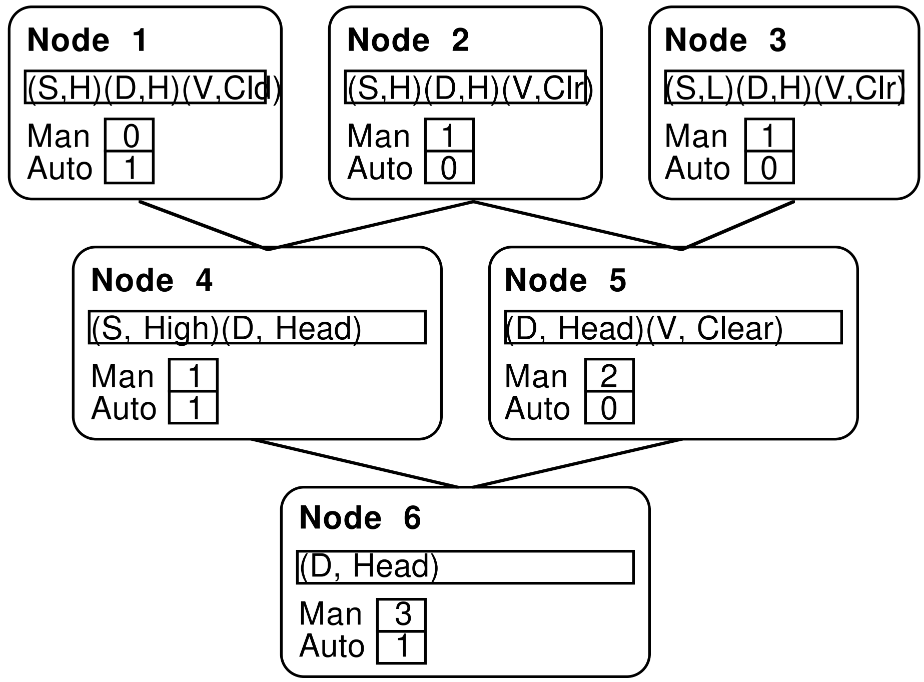

Figure 7.35. Node Initialization using Direct Specializations

GS, SG, and GAS learning algorithms can be implemented using either a broadcast/gather architecture or a more complex GS architecture. Broadcast/gather is simple and can be implemented directly using existing systems. The GS architecture is complex but provides better learning performance. Allocating new nodes with broadcast/gather is not difficult. No attempt is made to localize related nodes. The only concern is maintaining balance in the network structure. Adding new nodes with GS requires more overhead since related nodes are connected to one another. However, because of the connections between related nodes, GS allows increased performance for many learning and execution operations. Many operations are localized resulting in reduced network traffic.

Chapter 8
Complexity Analysis

This chapter gives a complexity analysis for the algorithms described in this paper. The complexity of algorithms in this paper is given in terms of the following variables:

Number of inputs to the system	n
Number of nodes in the system	m
Number of output values	k
Sum of number of input values over all inputs	p
Number of training examples	t

The number of nodes m is linear in terms of t since at most 4 nodes are created for each training example (1 single-input node, 1 example node, 1 generalization, and 1 specialization). Thus, whenever m is used in the complexity results in this chapter, m can be replaced by t. Since m and t are related linearly, $m = vt$, where v is a constant. Simulation results (Chapter 9) suggest that typical values for v are between .5 and 1.5. (The analysis here assumes one pass over t training examples. However, the simulations in Chapter 9 iterate over the examples from 2 to 10 times, depending on the choice of learning algorithm and training set.)

8.1. Space Complexity

The space complexity of all algorithms in this paper is the number of nodes times the number of elements in each node. All nodes require $O(n)$ space to hold the node's feature. All nodes also require $O(k)$ space for output counters. Nodes with next-input counters require additional space of $O(kp)$ for the next-input counters. All other information contained in nodes is constant size or $O(1)$. The total space complexity for algorithms without next-input counters is $O(m) \times (O(n) + O(k) + O(1)) = O(mn + mk)$. Algorithms using next-input counters require space of $O(m) \times (O(n) + O(k) + O(kp) + O(1)) = O(mn + mkp)$.

8.2. Common Operations

Several operations are found in many of the learning algorithms presented in this paper. The complexity analysis of these operations is given in this section.

8.2.1. Feature/Example/Input Operations

Several operations can be done on features, on examples, or on the system inputs. Features contain n input/value pairs, examples contain n input values, and the system has n inputs. Therefore, if it is assumed that the operations done on individual elements are $O(1)$, then the operations done on the complete object are $O(n)$ since the object contains n elements. Typical feature/example/input operations are:

Copy or transmission
Test for match

Test for equality
Creation
Compute generality
Compute union
Compute intersection

8.2.2. Constant Time Operations

Operations that are normally done in parallel on most processors such as addition, subtraction, multiplication, division, comparison, increment, and array indexing are assumed to be O(1).

8.2.3. Network Execution

This section gives the time complexity of the network execution algorithm given in Figure 3.4 in Section 3.9. The first step in network execution (Line 1) is to broadcast the network input values to all nodes in the network. Broadcast is O(log m) assuming a balanced tree network topology. When the input values are broadcast, each node must copy and transmit the values to the node's children. The copy of a list of inputs is O(n) so the total time is O(n log m).

Each node must determine if it matches the input by comparing its feature with the broadcast input values. The compare (Line 2) requires O(n) time.

Lines 3 to 6 compute the strongest output value for nodes that match the input. The strength of each output value is computed by dividing the counter for the output by the total count. Divide is assumed to require constant time. Computing the strength of all output values is $O(1) \times O(k) = O(k)$ since there are k output values. Finding the strongest of the k output values is $O(k) \times O(1) = O(k)$ assuming scalar comparison is O(1).

The network output is gathered on Line 7 of the algorithm. Gather is O(log m) assuming a balanced tree network topology. During the gather each node must compare the values given by the children to the node's own value. The comparison is assumed to be O(1) giving a total time for Line 7 of $O(\log m) \times O(1) = O(\log m)$.

The total time for network execution is $O(n \log m) + O(n) + O(k) + O(\log m) = O(n \log m + k)$.

8.2.4. Counter Update

When a new example is given to a network, the counters in all nodes that match the example must be updated. The test for match is O(n) as discussed above. The correct output counter is accessed using an array index giving time O(1). The increment of the counters is assumed to be O(1). The total time for counter update is $O(n) + O(1) + O(1) = O(n)$.

If the node contains next-input counters then O(n) is added to the total time since n next-input counters are incremented. The total time does not change since $O(n) + O(n) = O(n)$. If examples are stored in the node for LEM strength estimation then O(n) is added to the total time to account for copying the training example into the node's storage. Again, the total time does not change since $O(n) + O(n) = O(n)$.

8.2.5. Computing the New Feature Metric

The algorithms described in this paper use a metric that combines generality and strength to compare new features. Typically the new feature with the highest metric is used to create a new node. This section gives the time complexity for computation of the new feature metric. The metric is $M = Gc + S(1 - c)$, where G is the generality of the new feature, S is the strength of the new feature, and c is a constant between 0 and 1. Computing the generality of the feature is $O(n)$. The strength of the new feature is always previously computed. The multiplies, addition, and subtraction are all $O(1)$. The total time to compute the metric is $O(n) + O(1) + O(1) + O(1) + O(1) = O(n)$.

8.2.6. Finding an Equal/Covering/Contradicting Feature

Whenever a new feature is considered for creation it must be determined if a node with the same feature already exists in the network. This requires that the new feature is broadcast to all nodes in the network. The broadcast is $O(n \log m)$ as described in the discussion for network execution. All nodes in parallel then test the new feature for equality with the node's feature. This operation is $O(n)$ as discussed in Section 8.2.1. The result of the comparisons is gathered using time $O(\log m)$. The total time to determine if a feature already exists in the network is $O(n \log m) + O(n) + O(\log m) = O(n \log m)$.

If all m nodes are simultaneously considering the creation of new features, then the time complexity for finding an equal feature must be multiplied by m giving $O(mn \log m)$.

The time complexity of finding nodes with covering or contradicting features is the same as that for finding equal features. The only difference in the process is that the test done at each node between features is a test for covering or contradicting rather than a test for equality. All feature operations have the same complexity of $O(n)$ so the total complexity is the same for all three types of searches.

8.2.7. Creating a New Node

This section gives the time complexity for creating a new node. The first step in creating a new node is to verify that a node does not already exist in the network with the same feature. This operation is shown to be $O(n \log m)$ in Section 8.2.6. The time required for creating a new node is $O(n + k)$. This is the time to copy the new feature into the node plus the time to initialize the output counters. Creating a new node with next-input counters adds $O(kp)$ time to initialize the next-input counters. The total time for node creation is $O(n \log m) + O(n + k) = O(n \log m + k)$. If the node has next-input counters the total time is $O(n \log m + k) + O(kp) = O(n \log m + kp)$.

Creation of an example node is $O(n \log m + k)$ since example nodes and next-input counters are never used together.

When creating single-input nodes at most n new nodes are created. Each node creation requires time $O(n \log m + k)$ giving a total time of $O(n) \times O(n \log m + k) = O(n^2 \log m + nk)$. If the nodes have next-input counters the total time is $O(n) \times O(n \log m + kp) = O(n^2 \log m + nkp)$.

8.2.8. Strength Estimation

This section analyzes the complexity of NIC, LEM, and EN methods for estimating the strength of new features. NIC strength estimation is $O(1)$ since the counters for the pair that was added to create the new feature are simply divided to give the strength estimate. LEM strength estimation is $O(n)$ since a fixed number of examples stored in the node must be tested for matching with the new feature. EN strength estimation is $O(n \log m)$ for each node since the new feature must be broadcast to all the nodes followed by gathering the sum of the output counters for the current example output from example nodes that match the new feature. Since m nodes can do EN strength estimation at once, the total time for EN strength estimation is $O(mn \log m)$.

8.2.9. Node Initialization

This section gives the time complexity for node initialization for each of the three methods of strength estimation. NIC strength estimation requires time $O(k) + O(kp) = O(kp)$ to initialize the counters in the new node since k next-input counters from the parent node are copied into the output counters in the new node and the next-input counters in the new node are initialized to zero. LEM strength estimation requires $O(n)$ time since a fixed number of examples stored in the parent node are tested for matching with the new node. EN strength estimation is $O(n \log m) + O(k \log m)$ since the new node's feature must be broadcast to all nodes in the network followed by gathering the sums of the counters for the k different output values from the example nodes that match the new node.

8.3. General to Specific Learning

This section analyzes the time complexity of the GS learning algorithm given in Figure 4.4 in Section 4.1. The algorithm will be first analyzed in parts and then the parts will be summed to give the overall time complexity of the algorithm.

Network execution (Lines 1 to 2) is shown to be $O(n \log m + k)$ in Section 8.2.3.

Creation of single-input nodes (Lines 3 to 6) is shown to be $O(n^2 \log m + nk)$ in Section 8.2.7. The operation is $O(n^2 \log m + nkp)$ if the nodes contain next-input counters.

Counter update (Line 7) is shown to be $O(n)$ in Section 8.2.4.

Line 8 creates a new specific node if the network output is different from the example. The complexity of each specialization routine is described in a separate section below. The complexity of line 8 is $O(1) + O(s) = O(s)$, where $O(s)$ is the complexity of the specialization routine.

The total time complexity for the GS learning algorithm is the sum of the complexities listed above for each section or $O(n \log m + k) + O(n^2 \log m + nk) + O(n) + O(s) = O(n^2 \log m + nk) + O(s)$. If the nodes have next-input counters the total time is $O(n^2 \log m + nkp) + O(s)$.

If example nodes are created then $O(n \log m + k)$ is added to the total time giving the same result as without example node creation. If single-input nodes are not created then the second term is dropped giving $O(n \log m + k) + O(n) + O(s) = O(n \log m + k) + O(s)$.

For many cases n, k, and p are less than 30 and can be treated as constants. When n, k, and p can be treated as constants, the time complexity simplifies to $O(\log m) + O(s)$.

8.3.1. Feature Union

This section analyzes the complexity of feature-union and then combines that result with the result from the last section giving a total time complexity for GS learning using feature-union. The feature-union algorithm is given in Figure 4.5 in Section 4.3.

Lines 1 to 8 mark nodes that match the training example as available for union and compute a metric for such nodes. These lines are executed by all nodes in parallel. The most complex path is for nodes that match the example. The match operation (Line 2) is $O(n)$. The marking (Line 3) is $O(1)$. Computing the metric (Line 4) is $O(n)$. The total time for Lines 1 to 8 is $O(n) + O(1) + O(n) = O(n)$.

The metrics computed on Line 4 are gathered on Line 9. Gather is $O(\log m)$. The winning node from the gather is marked (Line 10) in $O(1)$ time. The total time for Lines 9 and 10 is $O(\log m) + O(1) = O(\log m)$.

Lines 11 to 21 form a loop that is repeated at most $m - 1$ times since when the loop starts at most $m - 1$ nodes are marked available and an additional available node is marked unavailable on each pass through the loop. The gather (Line 12) followed by the mark (Line 13) is $O(\log m)$ as shown above. Lines 14 to 19 create a new feature and a new node for the feature, if a node with the feature does not already exist. This operation is shown to be $O(n \log m + k)$ in Section 8.2.7. Finally Line 20 is $O(1)$. The total time for the body of the loop is $O(\log m) + O(n \log m + k) + O(1) = O(n \log m + k)$. The total time for Lines 11 to 21 is $O(m) \times O(n \log m + k) = O(mn \log m + mk)$.

The total time for feature-union is $O(n) + O(\log m) + O(mn \log m + mk) = O(mn \log m + mk)$.

The total time for GS learning using feature-union is given by $O(n^2 \log m + nk) + O(s)$ from the last section with $O(mn \log m + nk)$ substituted for $O(s)$, giving $O(n^2 \log m + nk) + O(mn \log m + mk) = O(mn \log m + mk)$ since in practical terms $m > n$.

For many cases n and k are less than 30 and can be treated as constants. When n and k can be treated as constants, the time complexity simplifies to $O(m \log m)$.

8.3.2. Feature Specialization

This section analyzes the complexity of feature-specialization and then combines that result with the time complexity for GS learning giving the total time complexity for GS learning using feature-specialization. The feature-specialization algorithm is given in Figure 4.13 in Section 4.4.

Lines 1 to 8 of feature-specialization are identical to Lines 1 to 8 of feature-union. Thus, the time for Lines 1 to 8 is $O(n)$.

Lines 9 to 22 form a loop that is repeated at most $m - 1$ times similar to Lines 11 to 21 in feature-union. The gather (Line 10) followed by the mark (Line 11) is $O(\log m)$.

Lines 12 to 21 form a loop that is repeated at most $n - 1$ times since node A must already contain at least 1 pair and the total number of pairs defined by the training example is n. The selection of a pair (Line 14) is $O(n)$ since the maximum strength pair is selected. Lines 15 to 20 create a new feature and a new node for the feature. This operation is shown to be $O(n \log m + k)$ in Section 8.2.7. The total time for the body of the loop is $O(n) + O(n \log m + k) = O(n \log m + k)$. The total time for Lines 12 to 21 is $O(n) \times O(n \log m + k) = O(n^2 \log m + nk)$

The total time for the body of the loop from Line 9 to Line 22 is O(log m) + O(n^2 log m + nk) = O(n^2 log m + nk). The total time for Lines 9 to 22 is O(m) × O(n^2 log m + nk) = O(mn^2 log m + mnk).

The total time for feature-specialization is O(n) + O(mn^2 log m + mnk) = O(mn^2 log m + mnk).

The total time for GS learning using feature-specialization is given by O(n^2 log m + nk) + O(s) with O(mn^2 log m + mnk) substituted for O(s), giving O(n^2 log m + nk) + O(mn^2 log m + mnk) = O(mn^2 log m + mnk).

If n and k can be treated as constants, the time complexity simplifies to O(m log m).

8.3.3. Feature Specialization with Strength Estimation

This section analyzes the complexity of feature-specialization-with-strength-estimation and then combines that result with the time complexity for GS learning giving the total time complexity for GS learning using feature-specialization-with-strength-estimation. The feature-specialization-with-strength-estimation algorithm is given in Figure 4.14 in Section 4.5.

All nodes in parallel do Lines 1 to 12. Each node matching the training example selects the best new feature that can be made by specializing the node's current feature. The match test (Line 1) is O(n).

Line 3 creates all possible features obtained from adding single input/value pairs to the current feature of the node. At most n new features are created. Each new feature creation is O(n). Thus the creation of all the new features is O(n^2).

Lines 4 to 6 are executed for each of the n new features. The search for a node with an equal feature (Line 5) is shown to be O(mn log m) in Section 8.2.6. The strength estimation calculation (Lines 5) is O(1) for NIC strength estimation, O(n) for LEM strength estimation, and O(mn log m) for EN strength estimation as shown in Section 8.2.8. The total for Lines 4 to 6 is O(mn log m) + O(1) = O(mn log m) for NIC strength estimation, O(mn log m) + O(n) = O(mn log m) for LEM strength estimation, and O(mn log m) + O(mn log m) = O(mn log m) for EN strength estimation. Thus, regardless of the strength estimation method used, the total for Lines 4 to 6 is O(mn log m). This total must then be multiplied by O(n) since the lines are executed for each of the n new features, giving O(mn^2 log m).

Line 7 is O(n) and Line 8 is O(1). The total time for Lines 3 to 8 is O(n^2) + O(mn^2 log m) + O(n) + O(1) = O(mn^2 log m). Line 9 allows these lines to be optionally repeated up to n – 1 times giving a total time for Lines 3 to 10 of O(mn^3 log m) when using repetition.

Calculation of the metric (Line 11) is O(n) as shown in Section 8.2.5. The total time for Lines 1 to 12 is O(n) + O(mn^2 log m) + O(n) = O(mn^2 log m). If repetition is used the total time is O(mn^3 log m).

The gather (Line 13) is O(log m). Creating a new node (Line 14) is shown to be O(n log m + k) in Section 8.2.7. The time for initializing the new node (Line 15) is O(kp) for NIC strength estimation, O(n) for LEM strength estimation, and O(n log m) + O(k log m) for EN strength estimation. The total time for Lines 13 to 15 is O(log m) + O(n log m + k) + O(kp) = O(n log m + kp) for NIC strength estimation, O(log m) + O(n log m + k) + O(n) = O(n log m + k) for LEM strength estimation, and O(log m) + O(n log m + k) + O(n log m) + O(k log m) = O(n log m + k log m) for EN strength estimation.

The total time for specialization using strength estimation is $O(mn^2 \log m) + O(n \log m + kp) = O(mn^2 \log m + kp)$ for NIC strength estimation, $O(mn^2 \log m) + O(n \log m + k) = O(mn^2 \log m + k)$ for LEM strength estimation, and $O(mn^2 \log m) + O(n \log m + k \log m) = O(mn^2 \log m) + O(k \log m)$ for EN strength estimation. Since typically $m > k$ and $m > p$, all three complexities simplify to $O(mn^2 \log m)$.

The total time for GS learning using feature-specialization-with-strength-estimation is given by $O(n^2 \log m + nk) + O(s)$, where $O(s)$ is replaced by the time required for feature-specialization-with-strength-estimation. For NIC strength estimation the total time is $O(n^2 \log m + nkp) + O(mn^2 \log m + kp) = O(mn^2 \log m + nkp)$. For LEM strength estimation the total time is $O(n^2 \log m + nk) + O(mn^2 \log m + k) = O(mn^2 \log m + nk)$. For EN strength estimation the total time is $O(n^2 \log m + nk) + O(mn^2 \log m + k \log m) = O(mn^2 \log m + k \log m + nk)$.

If n, k, and p can be treated as constants, the time complexities simplify to $O(m \log m)$.

8.4. Specific to General Learning

This section analyzes the time complexity of the SG learning algorithm given in Figure 5.1 in Section 5.1. SG learning is almost identical to GS learning except that 1) example nodes are always created, 2) single-input nodes are never created, and 3) when the network output is different from the example, generalization routines are executed rather than specialization routines. Therefore, the time complexity for SG learning is $O(n \log m + k) + O(g)$. $O(n \log m + k)$ is the complexity of GS learning without single-input nodes as shown in Section 8.3. $O(g)$ is the complexity of the generalization routine. The complexity of each generalization routine is described in a separate section below. As with GS learning, if n and k are treated as constants, the time complexity simplifies to $O(\log m) + O(g)$.

8.4.1. Feature Intersection

This section analyzes the complexity of feature-intersection and then combines that result with the result from the last section giving a total time complexity for SG learning using feature-intersection. The feature-intersection algorithm is given in Figure 5.2 in Section 5.2.

All nodes in parallel do Lines 1 to 9. Each node with the same output as the training example creates a new feature that is the intersection of the node's feature and the training example. Comparing outputs (Line 1) is an $O(1)$ operation. The creation of the intersection feature (Lines 2 to 3) is $O(n)$. The search for a node with an equal feature (Line 4) is shown to be $O(mn \log m)$ in Section 8.2.6. The EN strength estimation (Line 5) is shown to be $O(mn \log m)$ in Section 8.2.8. Line 6 determines if the new feature is covered or contradicted in $O(mn \log m)$ time as shown in Section 8.2.6. Marking the node (Line 6) is $O(1)$. Computing the metric (Line 8) is shown to be $O(n)$ in Section 8.2.5. The total time for Lines 1 to 9 is $O(1) + O(n) + O(mn \log m) + O(mn \log m) + O(mn \log m) + O(1) + O(n) = O(mn \log m)$.

The time required for the gather (Line 10) is $O(\log m)$. Creation of a new node (Line 11) is shown to be $O(n \log m + k)$ in Section 8.2.7. Initialization of the new node (Line

12) is $O(n \log m) + O(k \log m)$ as shown in Section 8.2.9. The total time for Lines 10 to 12 is $O(\log m) + O(n \log m + k) + O(n \log m) + O(k \log m) = O(n \log m) + O(k \log m)$.

The total time for feature-intersection is $O(mn \log m) + O(n \log m) + O(k \log m) = O(mn \log m)$ since m is typically greater than k.

The total time for SG learning using feature-intersection is given by $O(n \log m + k) + O(g)$, where $O(g)$ is replaced by the time required for feature-intersection. The total time is $O(n \log m + k) + O(mn \log m) = O(mn \log m + k)$. If it is assumed that $m > k$ then $O(mn \log m + k) = O(mn \log m)$.

If n and k can be treated as constants, the time complexity simplifies to $O(m \log m)$.

8.4.2. Feature Generalization

This section analyzes the complexity of feature-generalization and then combines that result with the time complexity for SG learning giving a total time complexity for SG learning using feature-generalization. The feature-generalization algorithm is given in Figure 5.9 in Section 5.3.

All nodes in parallel do Lines 1 to 12. Each node with the same output as the training example selects the best new feature that can be created by generalizing the node's current feature. Checking for feature match and comparing outputs (Line 1) is an $O(n)$ operation.

Line 3 creates all possible features obtained from removing single input/value pairs from the current feature of the node. At most n new features are created. Each new feature creation is $O(n)$. Thus the creation of all the new features is $O(n^2)$.

Lines 4 to 6 are executed for each of the n new features. The search for a node with an equal feature (Line 5) is shown to be $O(mn \log m)$ in Section 8.2.6. The EN strength estimation calculation (Line 5) is $O(mn \log m)$ as shown in Section 8.2.8. The total time for Lines 4 to 6 is $O(mn \log m) + O(mn \log m) = O(mn \log m)$. This total must then be multiplied by $O(n)$ since the lines are executed for each of the n new features, giving $O(mn^2 \log m)$.

Finding the strongest new feature (Lines 7 to 8) requires that as many as n new features must be checked for covering or contradiction. Each check requires $O(mn \log m)$ giving a total time of $O(mn^2 \log m)$.

The total time for Lines 3 to 8 is $O(n^2) + O(mn^2 \log m) + O(mn^2 \log m) = O(mn^2 \log m)$. Line 9 allows these lines to be optionally repeated up to $n - 1$ times giving a total time for Lines 3 to 10 of $O(mn^3 \log m)$ when using repetition.

Calculation of the metric (Line 11) is $O(n)$ as shown in Section 8.2.5. The total time for Lines 1 to 12 is $O(n) + O(mn^2 \log m) + O(n) = O(mn^2 \log m)$. If repetition is used the total time is $O(mn^3 \log m)$.

The gather (Line 13) is $O(\log m)$. Creating a new node (Line 14) is $O(n \log m + k)$ as shown in Section 8.2.7. The time for initializing counters in the new node (Line 15) is $O(n \log m) + O(k \log m)$ since EN strength estimation is used.

The total time for feature-generalization is $O(mn^2 \log m) + O(\log m) + O(n \log m + k) + O(n \log m) + O(k \log m) = O(mn^2 \log m) + O(k \log m)$. Since typically $m > k$, $O(mn^2 \log m) + O(k \log m) = O(mn^2 \log m)$.

The total time for SG learning using feature-generalization is given by $O(n \log m + k) + O(g)$, where $O(g)$ is replaced by the time required for feature-generalization. The total time is $O(n \log m + k) + O(mn^2 \log m) = O(mn^2 \log m + k)$. If it is assumed that $m > k$ then $O(mn^2 \log m + k) = O(mn^2 \log m)$.

If n and k can be treated as constants, the time complexity simplifies to $O(m \log m)$.

8.5. Generalization and Specialization Learning

This section analyzes the time complexity of the GAS learning algorithm given in Figure 6.1 in Section 6.1. GAS learning is almost identical to GS learning except that 1) example nodes are always created and 2) when the network output is different from the example, generalization and specialization routines are executed rather than just specialization routines. Therefore, the time complexity for GAS learning is $O(n^2 \log m + nk) + O(gs)$ with single-input nodes and $O(n \log m + k) + O(gs)$ without single-input nodes. $O(gs)$ is the complexity of the generalization and specialization routine. The complexity of each generalization and specialization routine is described in a separate section below. As with GS learning, if n and k are treated as constants, the time complexity simplifies to $O(\log m) + O(gs)$.

8.5.1. Both Generalization and Specialization

When both generalization and specialization are done, feature-specialization with EN strength estimation is executed followed by feature-generalization. Feature-specialization with EN strength estimation is $O(mn^2 \log m)$ as shown in Section 8.3.3. Feature-generalization is $O(mn^2 \log m)$ as shown in Section 8.4.2. The time complexity for both generalization and specialization is the sum of the two, or $O(mn^2 \log m) + O(mn^2 \log m) = O(mn^2 \log m)$.

Combining this result with the result for GAS learning from Section 8.5 gives $O(n^2 \log m + nk) + O(mn^2 \log m) = O(mn^2 \log m + nk)$. Assuming n and k can be treated as constants, the result simplifies to $O(m \log m)$.

8.5.2. Best of Generalization and Specialization

This section analyzes the complexity of the generalization-or-specialization algorithm and then combines that result with the time complexity for GAS learning giving the total time complexity for GAS learning using generalization-or-specialization. The generalization-or-specialization algorithm is given in Figure 6.8 in Section 6.3.

All nodes in parallel do Lines 1 to 15. Each node matching the training example selects the best new feature that can be made by either generalizing or specializing the node's current feature. The match test (Line 1) is $O(n)$.

Lines 3 to 6 create all possible features obtained from either adding or removing single input/value pairs from the current feature of the node. Exactly n new features are created. Each new feature creation is $O(n)$. Thus the creation of all the new features is $O(n^2)$.

Lines 7 to 9 are executed for each of the n new features. The search for a node with an equal feature (Line 8) is shown to be $O(mn \log m)$ in Section 8.2.6. The EN strength estimation calculation (Line 8) is $O(mn \log m)$ as shown in Section 8.2.8. The total for Lines 7 to 9 is $O(mn \log m) + O(mn \log m) = O(mn \log m)$. This total must then be multiplied by $O(n)$ since the lines are executed for each of the n new features, giving $O(mn^2 \log m)$.

Finding the strongest new feature (Lines 10 to 11) requires that as many as n new features must be checked for covering or contradiction. Each check requires $O(mn \log m)$ giving a total time of $O(mn^2 \log m)$.

The total time for Lines 3 to 11 is $O(n^2) + O(mn^2 \log m) + O(mn^2 \log m) = O(mn^2 \log m)$. Line 12 allows these lines to be optionally repeated up to $n - 1$ times giving a total time for Lines 3 to 13 of $O(mn^3 \log m)$ when using repetition.

Calculation of the metric (Line 14) is $O(n)$ as shown in Section 8.2.5. The total time for Lines 1 to 15 is $O(n) + O(mn^2 \log m) + O(n) = O(mn^2 \log m)$. If repetition is used the total time is $O(mn^3 \log m)$.

The gather (Line 16) is $O(\log m)$. Creating a new node (Line 17) is shown to be $O(n \log m + k)$ in Section 8.2.7. Initialization of the new node (Line 18) is $O(n \log m) + O(k \log m)$.

The total time for generalization-or-specialization is $O(mn^2 \log m) + O(\log m) + O(n \log m + k) + O(n \log m) + O(k \log m) = O(mn^2 \log m) + O(k \log m)$. Since typically $m > k$, $O(mn^2 \log m) + O(k \log m) = O(mn^2 \log m)$.

The total time for GAS learning using generalization-or-specialization is given by $O(n^2 \log m + nk) + O(gs)$, where $O(gs)$ is replaced by the time required for generalization-or-specialization. The total time is $O(n^2 \log m + nk) + O(mn^2 \log m) = O(mn^2 \log m + nk)$. If it is assumed that $m > k$ then $O(mn^2 \log m + k) = O(mn^2 \log m)$. If n and k are treated as constants, the time complexity simplifies to $O(m \log m)$.

8.6. General/Specific Architecture Complexity

8.6.1. Theorem 8.1. Depth of a GS architecture network

The maximum depth of a GS architecture network is n. There can be at most n edges from the root to any leaf node.

Proof

Let the network be divided into $n + 1$ levels. Let node A be assigned to level r_a where r_a is the number of input/value pairs in node A. The root node is at level 0 and example nodes are at level n, since example nodes contain n input/value pairs. In the GS architecture, a general to specific link from node A to B only exists when node B is a specialization of node A. Node B is a specialization of A only if $r_b > r_a$. Therefore, a link exists from A to B only if $r_b > r_a$. Since $r_b > r_a$, node B must be on a higher level than node A.

Suppose the network is traversed and let i be the current level of the traversal. Whenever an edge in the network is traversed, i must be increased since an edge only exists if the destination node is at a higher level. A traversal from root to leaves begins at level 0 and ends at level n. The traversal requires at most n steps since at each step i is increased at least by 1.

8.6.2. Theorem 8.2. Number of specialization connections.

Let E be an edge connecting node A to node B in a GS network. Let r_a and r_b be the number of input/value pairs in the features of nodes A and B, respectively. Let the network be restricted such that sufficient intermediate nodes are present so that for any edge E, $|r_a - r_b| = 1$. In other words, each edge connects two nodes that only differ in generality by exactly 1 input/value pair. In such a restricted network, the number of edges connected to a node in the specialization direction is bounded by p, where p is the sum of the number of input values over all the inputs.

Proof

Consider all edges connecting node A to a more specific node B_i. Let r_a be the number of input/value pairs in node A. Let r_i be the number of input/value pairs in node B_i. For each node B_i connected to node A, $r_i = r_a + 1$ (i.e. each node B_i has exactly one more input/value pair than node A). Since B_i is a specialization of node A, the pairs in node B_i must be a superset of the input/value pairs in node A. In other words, each node B_i is made up of the feature contained in node A with a single additional input/value pair. There are exactly p unique input/value pairs since each pair is composed of an input and a possible value for that input. Therefore, there can be at most p nodes B_i that are unique. So there are at most p edges connecting node A to more specific nodes.

8.6.3. Theorem 8.3. Number of generalization connections.

Let a network be restricted exactly as described in Theorem 8.2. In such a restricted network, the number of edges connected to a node in the generalization direction is bounded by n, where n is the number of inputs to the network.

Proof

Consider all edges connecting node A to a more general node B_i. Let r_a be the number of input/value pairs in node A. Let r_i be the number of input/value pairs in node B_i. For each node B_i connected to node A, $r_i = r_a - 1$ (i.e. each node B_i has exactly one less input/value pair than node A). Since B_i is a generalization of node A, the pairs in node B_i must be a subset of the input/value pairs in node A. In other words, each node B_i is made up of the feature contained in node A with a single input/value pair removed. There are at most n input/value pairs in node A that can be removed. Therefore, there can be at most n nodes B_i that are unique. So there are at most n edges connecting node A to more general nodes.

Note that no bounds are given for the number of edges for unrestricted networks. However, simulation results have shown that in practice the bounds for restricted networks typically hold for unrestricted networks. Thus, the bounds for restricted networks are used in the following analysis of complexity for network operations.

8.6.4. Creating Specialization Links

This section analyzes the complexity of the LinkSpec algorithm found in Figure 7.9 in Section 7.2.2.1. LinkSpec recursively traverses through the levels of the GS network architecture. Since the network has at most $n + 1$ levels, LinkSpec is called recursively at most n times. Nodes at each level operate in parallel.

Lines 2 to 8 are executed for each node that is connected to G in the specialization direction. As shown in Section 8.6.2 above, at most p nodes can be connected to any node in the specialization direction. If the test on Line 2 fails and the test on Line 6 succeeds then LinkSpec is called recursively. The specialization test on Line 2 is $O(n)$. The overlap test on Line 6 is $O(n)$. Thus, the time at each recursive call is $p \times (O(n) + O(n)) = O(np)$. The total time for recursive calls is $n \times O(np) = O(n^2p)$ since LinkSpec is called recursively at most n times.

Lines 3 to 5 are executed at the terminal points of the recursion. The test for a redundant link requires two parts. Let N be the new node and S be the possible specialization. Node N must be tested against all specializations of S. If N is a specialization of any specialization of S, then the new link is redundant. Node S must be tested against all generalizations of N. If S is a generalization of any generalization of N, then the new link is redundant.

The first part of the redundant test is $O(p) \times O(n)$ since S has at most p specializations and each test is $O(n)$. The second part of the test is $O(n) \times O(n)$ since N has at most n generalizations. The creation of a link is O(1). The total time for Lines 3 to 5 is $O(np) + O(n^2) + O(1) = O(np + n^2)$. Since $p > n$ the total time simplifies to $O(np)$.

The total time for LinkSpec is the sum of the recursive time and the time at the terminal points of the recursion. Recall that the nodes at the terminal points all operate in parallel. Thus, the total time for LinkSpec is $O(n^2p) + O(np) = O(n^2p)$.

8.6.5. Creating Generalization Links

This section analyzes the complexity of the LinkGen algorithm found in Figure 7.12 in Section 7.2.2.2. LinkGen recursively traverses through the levels of the GS network architecture. Since the network has at most $n + 1$ levels, LinkGen is called recursively at most n times. Nodes at each level operate in parallel.

Lines 2 to 8 are executed for each node that is connected to S in the generalization direction. As shown in Section 8.6.3 above, at most n nodes can be connected to any node in the generalization direction. If the test on Line 2 fails and the test on Line 6 succeeds then LinkGen is called recursively. The generalization test on Line 2 is $O(n)$. The intersect test on Line 6 is $O(n)$. Thus, the time at each recursive call is $n \times (O(n) + O(n)) = O(n^2)$. The total time for recursive calls is $n \times O(n^2) = O(n^3)$ since LinkGen is called recursively at most n times.

Lines 3 to 5 are executed at the terminal points of the recursion. Since these lines are identical to the same lines in LinkSpec described in the last section, the total time for Lines 3 to 5 is $O(np)$.

The total time for LinkGen is the sum of the recursive time and the time at the terminal points of the recursion. Recall that the nodes at the terminal points all operate in parallel. Thus, the total time for LinkGen is $O(n^3) + O(np) = O(n^3 + np)$.

8.6.6. Removing Redundant Links

This section analyzes the complexity of the RemoveRedundant algorithm found in Figure 7.16 in Section 7.2.2.3.

Line 1 loops for all nodes connected to N that are generalizations of N. Line 2 loops for all nodes connected to N that are specializations of N. Lines 3 to 5 are O(1) since two nodes can be tested for connection in constant time and two nodes can be disconnected in constant time. The total time for RemoveRedundant is $O(np)$ since the number of generalizations connected to a node is bounded by n and the number of specializations connected to a node is bounded by p.

8.6.7. Creating Generalization Links without a Specialization Node

This section analyzes the complexity of the LinkRoot algorithm found in Figure 7.18 in Section 7.2.2.4. LinkRoot recursively traverses through the levels of the GS network architecture. Since the network has at most $n + 1$ levels, LinkRoot is called recursively at most n times. Nodes at each level operate in parallel.

Lines 3 to 6 are executed for each node that is connected to G in the specialization direction. As shown in Section 8.6.2 above, at most p nodes can be connected to any node in the specialization direction. If a specialization of G is a generalization of the new node then LinkRoot is called recursively. The generalization test on Line 3 is $O(n)$. Thus, the time at each recursive call is $p \times O(n) = O(np)$. The total time for recursive calls is $n \times O(np) = O(n^2p)$ since LinkRoot is called recursively at most n times.

Lines 8 to 10 are executed at the terminal points of the recursion. Since these lines are identical to Lines 3 to 5 in LinkSpec described in Section 8.6.4, the total time for Lines 8 to 10 is $O(np)$.

The total time for LinkRoot is the sum of the recursive time and the time at the terminal points of the recursion. Recall that the nodes at the terminal points all operate in parallel. Thus, the total time for LinkRoot is $O(n^2p) + O(np) = O(n^2p)$.

8.6.8. Linking Single-Input Nodes

This section analyzes the complexity of the LinkSingleInputNode algorithm found in Figure 7.21 in Section 7.2.2.5. The first line of the algorithm links the new node directly to the root node. This operation is $O(1)$. The LinkSpec routine is shown to be $O(n^2p)$ in Section 8.6.4. The RemoveRedundant routine is shown to be $O(np)$ in Section 8.6.6. The total time for LinkSingleInputNode is $O(1) + O(n^2p) + O(np) = O(n^2p)$.

8.6.9. Linking Example Nodes

The LinkExampleNode routine found in Figure 7.22 in Section 7.2.2.6 simply calls the LinkRoot routine analyzed in Section 8.6.7. Thus the LinkExampleNode routine is $O(n^2p)$ as shown in Section 8.6.7.

8.6.10. Linking Specializations of Existing Nodes

This section analyzes the complexity of the LinkSpecializationNode algorithm found in Figure 7.23 in Section 7.2.2.7. The LinkSpec routine (Line 1) is shown to be $O(n^2p)$ in Section 8.6.4. One of either Line 3 or Line 5 will be executed. The LinkGen routine (Line 3) is $O(n^3 + np)$. The LinkRoot routine (Line 5) is $O(n^2p)$. The RemoveRedundant routine (Line 7) is shown to be $O(np)$ in Section 8.6.6. If Line 5 is executed the total time for LinkSpecializationNode is $O(n^2p) + O(n^2p) + O(np) = O(n^2p)$. If Line 3 is executed the total time for LinkSpecializationNode is $O(n^2p) + O(n^3 + np) + O(np) = O(n^3 + n^2p)$. Since $p > n$, $O(n^3 + n^2p) = O(n^2p)$. Thus, the time for LinkSpecializationNode is $O(n^2p)$ regardless of which of Lines 3 and 5 are executed.

8.6.11. Linking Generalizations of Existing Nodes

This section analyzes the complexity of the LinkGeneralizationNode algorithm found in Figure 7.24 in Section 7.2.2.8. The LinkGen routine (Line 1) is shown to be $O(n^3 + np)$ in Section 8.6.5. The LinkSpec routine (Line 3) is $O(n^2p)$ as shown in Section 8.6.4. The RemoveRedundant routine (Line 4) is shown to be $O(np)$ in Section 8.6.6. The total time for LinkGeneralizationNode is $O(n^3 + np) + O(n^2p) + O(np) = O(n^3 + n^2p)$. Since $p > n$ the time for LinkGeneralizationNode simplifies to $O(n^2p)$.

8.6.12. Removing Nodes

This section analyzes the complexity of the RemoveLinks algorithm found in Figure 7.25 in Section 7.2.3.

Line 1 loops for all nodes connected to N that are generalizations of N. Line 2 loops for all nodes connected to N that are specializations of N. Since Lines 3 to 5 are identical to the same lines in LinkSpec described in Section 8.6.4, the total time for Lines 3 to 5 is $O(np)$. Since the number of generalizations connected to a node is bounded by n and the number of specializations connected to a node is bounded by p, the total time for Lines 1 to 7 is $np \times O(np) = O(n^2p^2)$.

The loop on Lines 8 to 10 executes at most n times since at most n generalizations are connected to N. The removal of a link (Line 9) is O(1). The total time for Lines 8 to 10 is $O(n)$. The loop on Lines 11 to 13 is similar to the loop on Lines 8 to 10 except that it executes at most p times since at most p specializations are connect to N. The total time for Lines 11 to 13 is $O(p)$.

The total time for RemoveLinks is $O(n^2p^2) + O(n) + O(p) = O(n^2p^2)$.

8.6.13. Broadcast/Gather on a GS Architecture

The complexity analysis of algorithms in this paper assumes broadcast and gather are done on a tree architecture. The depth of a balanced tree with m nodes is $O(\log m)$. The complexity of broadcast and gather is dependent on the depth of the network. Thus the complexity of most algorithms contains $O(\log m)$.

Section 8.6.1 shows that the depth of a GS network is at most n. Thus the complexity results of algorithms using broadcast and gather should have $O(\log m)$ replaced with $O(n)$ if broadcast and gather are done using the GS network.

Note that a set of nodes can be connected using both a tree architecture and a GS architecture at the same time. Thus, the system can obtain the best complexity characteristics depending on the values of n and m.

8.6.14. Finding Equal Nodes

This section analyzes the complexity of the EqualSpec and EqualGen algorithms found in Figures 7.30 and 7.32 in Section 7.2.4.3.

Lines 2 to 10 of EqualSpec are executed for each node that is connected to G in the specialization direction. As shown in Section 8.6.2 above, at most p nodes can be connected to any node in the specialization direction. If a node equal to the new node is not found among the specializations of G then EqualSpec is called recursively. The equal test (Line 2) is $O(n)$. The specialization test (Line 5) is also $O(n)$. Thus, the time at each recursive call is $p \times (O(n) + O(n)) = O(np)$. The total time for EqualSpec is $n \times O(np) = O(n^2p)$ since EqualSpec is called recursively at most n times.

Algorithms that use feature-specialization create new features that have one input/value pair more than the parent node. Therefore, the EqualSpec routine is not called recursively for these algorithms. One level of testing for equal nodes is sufficient to determine if equal nodes exist. Thus, the complexity of EqualSpec is $O(np)$ for algorithms using feature-specialization.

The EqualGen algorithm is similar to the EqualSpec algorithm except that the main loop is executed at most n times rather than p times since at most n generalizations of S can be connected to S. Thus the total time for EqualGen is $O(n^3)$.

Algorithms that use feature-generalization create new features by removing a single input/value pair from the feature in the parent node. Therefore, the EqualGen routine is not called recursively for these algorithms. Thus, the complexity of EqualGen is O(n^2) for algorithms using feature-generalization.

Note that using EqualSpec or EqualGen in learning algorithms typically causes the factor m in the complexity result to be replaced with either p or n. Typically p and n are both much smaller than m so using the GS architecture can speed up learning.

8.6.15. Strength Estimation

This section analyzes the complexity of the DS strength estimation algorithm given in Figure 7.33 in Section 7.2.4.6. DS strength estimation totals the counters of nodes that are specializations of the new node and are directly connected to the new node. Since at most p specializations of a node can be connected to the node, DS strength estimation is O(p). This result compares favorably with the O(mn log m) complexity of EN strength estimation given in Section 8.2.8.

Node initialization for DS strength estimation computes $k + 1$ totals by summing the counters in nodes that are specializations of the new node and are connected to the new node. Again there are at most p such nodes connected to the new node. Thus, the total time for node initialization is O(kp).

8.7. Conclusion

Table 8.1 summarizes the results of this chapter. All algorithms have polynomial learning times using either broadcast/gather or GS architectures. When n, k, and p are treated as constants, all algorithms are O(m log m). Using the GS architecture improves the time of many algorithms to O(log m). Thus, the GS architecture can provide a significant increase in learning speed.

Adding any type of node to the GS architecture requires time O(n^2p). Deleting a node in the GS architecture is O(n^2p^2). Adding a node to a tree is O(log m). Thus, connecting nodes in the more complex GS architecture does not incur a significant overhead compared to the simpler tree structure.

Table 8.1. Complexity Summary

Algorithm	**Broadcast/Gather**	***n*, *k*, *p* constant**	**GS architecture**
Union	O(mn log $m + mk$)	O(m log m)	O(m log m)
Specialization	O(mn^2 log $m + mnk$)	O(m log m)	O(m log m)
NIC	O(mn^2 log $m + nkp$)	O(m log m)	O(log m)
LEM	O(mn^2 log $m + nk$)	O(m log m)	O(log m)
EN	O(mn^2 log $m + nk$)	O(m log m)	O(log m) (DS)
Intersection	O(mn log m)	O(m log m)	O(log m) (DS)
Generalization	O(mn^2 log m)	O(m log m)	O(log m) (DS)
GAS	O(mn^2 log $m + nk$)	O(m log m)	O(log m) (DS)

Chapter 9
Simulation Results

This chapter gives the results of testing each learning algorithm on 10 data sets. This section describes the data sets, the procedure used for testing, and the results of the testing. Variations on the algorithms are described along with the test results for the variations.

9.1. Data Sets

The 10 data sets used for testing are listed in « DataSets». The data was obtained from UC Irvine [Murphy & Aha 1992]. For each data set the table shows (1) the number of inputs for the data set, (2) the number of values for the inputs, (3) the number of values for the output, (4) the number of examples in the complete data set, and (5) the range of test accuracy obtained from other algorithms on the data set. Following the table, detailed information is given for each data set.

Table 9.1. Data Sets

Data Set	Inputs	In Values	Out Val	Examples	Accuracy	
Breast Cancer	9	10	2	699	92.2	95.9
Chess End-Game	36	2	2	3196	67.6	99.2
Hepatitis	19	2–10	2	155	38.7	85.8
Iris	4	continuous	3	150	84	98
LED	7	2	10	1000	71	73.3
Mushroom	22	2–12	2	8124	91.2	100
Soybean	35	2–7	4	47	87.0	97.5
Tic Tac Toe End-Game	9	3	2	958	76.7	99.1
House Voting 1984	16	3	2	435	84	95.6
Zoo	16	2–6	7	90	NA	

9.1.1. Breast Cancer

The breast cancer data comes from the University of Wisconsin Hospitals, Madison [Magnasarian & Wolberg 1990]. The examples in the breast cancer data set have nine inputs describing a tumor. Each input is given a value from 1 to 10. The inputs are as follows:

Clump Thickness
Uniformity of Cell Size
Uniformity of Cell Shape

Marginal Adhesion
Single Epithelial Cell Size
Bare Nuclei
Bland Chromatin
Normal Nucleoli
Mitoses

The output for each example is either Benign or Malignant. There are 458 (65.5%) Benign examples and 241 (34.5%) Malignant examples.

9.1.2. Chess End-Game

The chess end-game examples are created from a game with a King and a Rook against a King and a Pawn. The Pawn is on square a7 which means that it is one square away from queening. The King and Rook are White and it is White's turn to move. Each example has 36 inputs that describe the board position. Each input is binary except for number 15 which has 3 states.

The output of each example is either White-can-win or White-cannot-win. There are 1669 (52.2%) examples with output White-can-win and 1527 (47.8%) examples with output White-cannot-win.

9.1.3. Hepatitis

The hepatitis data comes from Yugoslavia. The examples have 19 inputs. Several inputs are binary. Other inputs appear to be continuous but are actually discrete when split at the boundary values given in the following table:

Age	10, 20, 30, 40, 50, 60, 70, 80
Sex	male, female
Steroid	no, yes
Antivirals	no, yes
Fatigue	no, yes
Malaise	no, yes
Anorexia	no, yes
Liver Big	no, yes
Liver Firm	no, yes
Spleen Palpable	no, yes
Spiders	no, yes
Ascites	no, yes
Varices	no, yes
Biliruben	0.39, 0.80, 1.20, 2.00, 3.00, 4.00
Alk Phosphate	33, 80, 120, 160, 200, 250
Sgot	13, 100, 200, 300, 400, 500
Albumin	2.1, 3.0, 3.8, 4.5, 5.0, 6.0
Protime	10, 20, 30, 40, 50, 60, 70, 80, 90
Histology	no, yes

The output of each example is either Die or Live. There are 32 (20.6%) Die examples and 123 (79.4%) Live examples.

9.1.4. Iris

The iris data set is a classic from the pattern recognition literature [Fisher 1936]. The examples have 4 inputs that are all continuous. Each of the 4 continuous inputs were split into 10 discrete values using a K-means clustering algorithm.

Sepal Length	4.3 cm – 7.9 cm
Sepal Width	2.0 cm – 4.4 cm
Petal Length	1.0 cm – 6.9 cm
Petal Width	0.1 cm – 2.5 cm

The output of each example is either Iris-Setosa, Iris-Versicolour, or Iris-Virginica. There are 50 (33.3%) examples with each type of output. One output class is linearly separable from the other two. The remaining two classes are not linearly separable from each other.

9.1.5. LED

The LED examples have 7 inputs that represent the 7 segments of an LED display [Brieman, et al. 1984]. Each input is binary indicating whether the segment is on or off. The inputs are intentionally made noisy. Each input has a 10% chance of being inverted. The inputs are as follows:

Top
Top Left
Top Right
Middle
Bottom Left
Bottom Right
Bottom

The output of each example is the digit represented by the original 7 segments before noise is introduced. There are approximately 100 (10%) examples with each output since the output is randomly chosen. The optimal Bayes classification accuracy is 74%.

9.1.6. Mushroom

The mushroom examples are drawn from *The Audubon Society Field Guide to North American Mushrooms*. The examples have 22 inputs with possible values as shown in the following table:

Cap Shape	Bell, Conical, Convex, Flat, Knobbed, Sunken
Cap Surface	Fibrous, Grooves, Scaly, Smooth
Cap Color	Brown, Buff, Cinnamon, Gray, Green, Pink, Purple, Red, White, Yellow
Bruises	True, False

Odor	Almond, Anise, Creosote, Fishy, Foul, Musty, None, Pungent, Spicy
Gill Attachment	Attached, Descending, Free, Notched
Gill Spacing	Close, Crowded, Distant
Gill Size	Broad, Narrow
Gill Color	Black, Brown, Buff, Chocolate, Gray, Green, Orange, Pink, Purple, Red, White, Yellow
Stalk Shape	Enlarging, Tapering
Stalk Root	Bulbous, Club, Cup, Equal, Rhizomorphs, Rooted, Missing
Stalk Surface Above Ring	Fibrous, Scaly, Silky, Smooth
Stalk Surface Below Ring	Fibrous, Scaly, Silky, Smooth
Stalk Color Above Ring	Brown, Buff, Cinnamon, Gray, Orange, Pink, Red, White, Yellow
Stalk Color Below Ring	Brown, Buff, Cinnamon, Gray, Orange, Pink, Red, White, Yellow
Veil Type	Partial, Universal
Veil Color	Brown, Orange, White, Yellow
Ring Number	None, One, Two
Ring Type	Cobwebby, Evanescent, Flaring, Large, None, Pendant, Sheathing, Zone
Spore Print Color	Black, Brown, Buff, Chocolate, Green, Orange, Purple, White, Yellow
Population	Abundant, Clustered, Numerous, Scattered, Several, Solitary
Habitat	Grasses, Leaves, Meadows, Paths, Urban, Waste, Woods

The output of each example is either Edible or Poisonous. There are 4208 (51.8%) examples with output Edible and 3916 (48.2%) examples with output Poisonous.

9.1.7. Soybean

The soybean examples have 35 inputs with values as shown in the following table:

Date	April, May, June, July, August, September, October
Plant Stand	Normal, lt Normal, Unknown
Precipitation	lt Normal, Normal, gt Normal, Unknown
Temperature	lt Normal, Normal, gt Normal, Unknown
Hail	Yes, No, Unknown
Crop History	Diff Last Year, Same Last Year, Same Last Two Years, Same Last Several Years, Unknown
Area Damaged	Scattered, Low Areas, Upper Areas, Whole Field, Unknown
Severity	Minor, pot Severe, Severe, Unknown
Seed Treatment	None, Fungicide, Other, Unknown
Germination	90–100%, 80–89%, lt 80%, Unknown

Plant Growth	Normal, Abnormal, Unknown
Leaves	Normal, Abnormal
Leafspots Halo	Absent, Yellow Halos, No Yellow Halos
Leafspots Marg	ws Marg, No ws Marg, dna, Unknown
Leafspot Size	lt 1/8, gt 1/8, dna, Unknown
Leaf Shred	Absent, Present, Unknown
Leaf Malf	Absent, Present, Unknown
Leaf Mild	Absent, Upper Surface, Lower Surface, Unknown
Stem	Normal, Abnormal, Unknown
Lodging	Yes, No, Unknown
Stem Cankers	Absent, Below Soil, Above Soil, Above sec nde, Unknown
Canker Lesion	dna, Brown, Dark Brown, Tan, Unknown
Fruiting Bodies	Absent, Present, Unknown
External Decay	Absent, Firm and Dry, Watery, Unknown
Mycelium	Absent, Present, Unknown
int Discolor	None, Brown, Black, Unknown
Sclerotia	Absent, Present, Unknown
Fruit Pods	Normal, Diseased, Few Present, dna, Unknown
Fruit Spots	Absent, Colored, Brown w/Black Specs, Distort, dna, Unknown
Seed	Normal, Abnormal, Unknown
Mold Growth	Absent, Present, Unknown
Seed Discolor	Absent, Present, Unknown
Seed Size	Normal, lt Normal, Unknown
Shriveling	Absent, Present, Unknown
Roots	Normal, Rotted, Galls Cysts, Unknown

The output of each example is one of four soybean diseases (D1, D2, D3, D4). There are 10 (21.3%) D1 examples, 10 (21.3%) D2 examples, 10 (21.3%) D3 examples, and 17 (36.2%) D4 examples.

9.1.8. Tic Tac Toe End Game

The tic-tac-toe end-game data encodes the complete set of possible board positions at the end of tic-tac-toe games. It is assumed that 'x' played first. Each example has 9 inputs representing the 9 tic-tac-toe squares. Each input is either 'x', 'o', or 'blank'. The inputs are as follows:

Top Left
Top Middle
Top Right
Middle Left
Middle Middle
Middle Right
Bottom Left
Bottom Middle
Bottom Right

The output of each example is either Win-for-X or Not-win-for-X. There are 626 (65.3%) examples with output Win-for-X and 332 (34.7%) examples with output Not-win-for-X.

9.1.9. Voting

The voting data is taken from 1984 United States congressional voting records [CQA 1985]. Each example has 16 inputs corresponding to the 16 key votes identified by the *Congressional Quarterly Almanac*. Each input is either For, Against, or No Vote. Three kinds of votes, voted for, paired for, and announced for, are combined into For. Three kinds of votes, voted against, paired against, and announced against, are combined into Against. Three kinds of votes, voted present, voted present to avoid conflict of interest, and did not vote, are combined into No Vote. The 16 votes are as follows:

Handicapped Infants
Water Project Cost Sharing
Adoption of the Budget Resolution
Physician Fee Freeze
El Salvador Aid
Religious Groups in Schools
Anti Satellite Test Ban
Aid to Nicaraguan Contras
MX Missile
Immigration
Synfuels Corporation Cutback
Education Spending
Superfund Right to Sue
Crime
Duty Free Exports
Export Administration Act South Africa

The output of each example is either Democrat or Republican. There are 267 (61.4%) examples with output Democrat and 168 (38.6%) examples with output Republican.

9.1.10. Zoo

The zoo data set contains examples of animals. Each example has 16 inputs with values as shown in the following table.

Hair	Yes, No
Feathers	Yes, No
Eggs	Yes, No
Milk	Yes, No
Airborne	Yes, No
Aquatic	Yes, No
Predator	Yes, No
Toothed	Yes, No
Backbone	Yes, No
Breathes	Yes, No

Venomous	Yes, No
Fins	Yes, No
Legs	0, 2, 4, 5, 6, 8
Tail	Yes, No
Domestic	Yes, No
Catsize	Yes, No

The output of each example is Mammal, Bird, Reptile, Fish, Amphibian, Insect, or Other. There are 41 examples with output Mammal, 20 examples with output Bird, 5 examples with output Reptile, 13 examples with output Fish, 4 examples with output Amphibian, 8 examples with output Insect, and 10 examples with output Other. Previous results on this data set are not known.

9.2. Test Process

This section describes the methods used for testing the learning algorithms in this paper. The test methods are intended to insure that the results are not skewed by a fortunate selection of training and testing sets.

Each algorithm is run 100 times, ten times on each data set. The results for each data set are averaged over the ten runs. Each data set is split into a training set and a test set. One half of the examples in a data set are randomly selected as the training set for each run. The remaining examples are used as the test set. Thus, the test set does not contain examples from the training set. New training and test sets are selected for each run.

On each run the training set is used to train the system. The system iterates over the training set. Training stops when the system completes a full pass over the training set without creating or deleting any nodes. (In addition, the number of passes over the training data is limited to a maximum of 10.) When training is complete the test set is used to test the system. During testing the system does not learn. The system only operates in execution mode.

9.3. Results Gathered

This section explains what variables are measured and reported as results. The results are intended to measure generalization performance, speed, and useability of the algorithms. The six variables that are reported are as follows:

Accuracy on the training set
Accuracy on the test set
Number of nodes created
Number of nodes deleted
Number of nodes in the final network
Number of passes over the training set

Accuracy on the training set is the number of training examples for which the network gives the correct answer divided by the total number of training examples. This result gives an indication of how accurate the system is for inputs that the system has been trained on. Training set accuracy is measured on the last pass over the training data.

Accuracy on the test set is the number of test examples for which the system gives the correct answer divided by the total number of test examples. This result indicates how well the system generalizes or how well the system can predict the correct output for inputs that the system has not been trained on. Test set accuracy is measured after training is complete.

The number of nodes in the system as well as the number of passes over the training set required for convergence both give an indication of how easily the system is learning the problem. A high number of nodes or passes indicates that the system has difficulty learning the problem. Note that models that use example nodes have at least as many nodes as there are examples in the training set. The interesting number for these models is the number of nodes created in addition to the example nodes.

Deleted nodes indicate poor choices were made for creation of new features. No incremental learning model can make perfect choices about creating new features since all information needed for a correct choice is not available. Thus, some deletion of nodes is needed, but it is desirable to minimize the number of nodes created in error.

9.4. GS Results

This section gives results for GS learning on the 10 data sets described above.

The three strength estimation models have more information available to them than union and specialization. Therefore, they are expected to be more accurate than the first two algorithms that do not use strength estimation. The strength estimation models are expected to reach a solution for the training set with fewer wrong choices in creating and deleting nodes. The strength estimation methods should converge in fewer passes than the other two methods since they choose better new features. The results confirm these expectations. (There are two exceptions that are discussed later.)

The three strength estimation methods have differing amounts of information available. NIC strength estimation has the least information and EN strength estimation has the most. Therefore, LEM is expected to perform better than NIC and EN is expected to do better than LEM. The results confirm these expectations.

Results for the five basic GS learning models on the 10 data sets are shown in « GSResults». All models have similar good results for Breast Cancer, Iris, Mushroom, Soybean, and Voting.

Note that create and node counts are high for EN strength estimation because EN strength estimation creates a node for every example in the training set. A fair comparison of EN strength estimation requires subtracting the number of example nodes created.

On the Chess data, union and specialization have high create, delete, and node counts compared to the strength estimation models. Even though the counts are lower for the strength estimation models the counts are still higher than is desired. The strength estimation models give about 2% better accuracy for the Chess data. Thus, the Chess data shows better results for strength estimation as expected.

Test accuracy is low on the Hepatitis data for all models. This result is comparable to previous results on Hepatitis. Note that the strength estimation models do better in training accuracy than the other models. Contrary to expectations, specialization performs almost 2% higher than any of the strength estimation methods in test accuracy.

Both train and test accuracy appear low for the LED data, but the Bayes optimal result is 74% because of the noise in the data set. Thus, accuracy is good for union, LEM strength estimation, and EN strength estimation. Accuracy is not good for specialization or NIC strength estimation even considering the Bayes optimal result of 74%. The create, delete, node, and pass counts are high for all models. Pass counts are particularly high for union and specialization suggesting that these models have difficulty learning the data set.

The Tic-Tac-Toe data set strongly shows the advantage of the strength estimation methods. This suggests that this data set has important higher-order features. Union and specialization create and delete about 3 times as many nodes as the strength estimation models. The pass counts are also very high for union and specialization. The two better strength estimation models have 2% better test accuracy than specialization. Test accuracy for union is very poor.

Table 9.2. GS Results

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Breast Cancer	Union	99.74%	95.03%	105.1	63.9	41.2	4.8
	Special	99.66%	95.20%	113.1	64.5	48.6	7.3
	NIC	100.00%	95.06%	96.4	58.6	37.8	3.6
	LEM	99.86%	95.26%	97.4	59.3	38.1	3.9
	EN	100.00%	95.03%	345.1	59.1	286.0	3.7
Chess	Union	95.08%	95.85%	746.8	472.1	274.7	9.0
	Special	94.39%	96.02%	636.1	379.7	256.4	7.8
	NIC	97.55%	97.83%	370.0	305.7	64.3	7.3
	LEM	97.94%	97.58%	418.4	356.5	61.9	7.8
	EN	98.60%	97.73%	1874.1	221.0	1653.1	5.6
Hepatitis	Union	94.03%	81.15%	92.9	63.2	29.7	6.7
	Special	94.16%	82.69%	96.1	63.2	32.9	6.3
	NIC	99.09%	80.00%	78.4	55.3	23.1	4.2
	LEM	99.61%	80.26%	76.8	53.1	23.7	3.8
	EN	96.36%	81.03%	168.7	69.7	99.0	5.7
Iris	Union	97.33%	92.13%	44.1	23.5	20.6	4.2
	Special	99.47%	92.80%	48.6	29.1	19.5	6.3
	NIC	98.40%	93.60%	38.3	19.2	19.1	3.6
	LEM	98.80%	92.93%	37.5	18.9	18.6	3.4
	EN	98.80%	92.80%	103.5	18.5	85.0	3.2

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
LED	Union	74.24%	72.10%	1031.3	636.8	394.5	9.9
	Special	68.48%	69.34%	1313.3	73.1	1240.2	9.3
	NIC	56.22%	62.60%	1166.1	965.5	200.6	6.1

	LEM	73.18%	71.80%	980.7	820.0	160.7	6.5
	EN	72.72%	73.00%	984.0	751.5	232.5	6.2
Mushroom	Union	99.96%	99.92%	120.3	92.4	27.9	5.2
	Special	99.94%	99.94%	121.0	93.9	27.1	4.2
	NIC	100.00%	99.98%	109.5	84.6	24.9	2.4
	LEM	99.98%	99.93%	114.8	89.0	25.8	3.5
	EN	99.96%	99.94%	4177.0	87.7	4089.3	3.3
Soybean	Union	100.00%	97.08%	66.2	61.5	4.7	2.7
	Special	100.00%	97.08%	66.2	61.5	4.7	2.7
	NIC	100.00%	97.08%	65.2	60.5	4.7	2.7
	LEM	100.00%	97.08%	65.2	60.5	4.7	2.7
	EN	100.00%	97.08%	88.2	60.5	27.7	2.7
Tic Tac Toe	Union	91.40%	85.26%	646.7	469.7	177.0	9.6
	Special	96.74%	95.28%	630.1	364.5	265.6	9.3
	NIC	95.82%	94.41%	200.3	150.0	50.3	5.0
	LEM	98.43%	97.29%	148.9	111.3	37.6	3.8
	EN	98.46%	97.37%	617.8	100.7	517.1	3.2

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Voting	Union	95.90%	94.38%	116.4	67.6	48.8	8.6
	Special	96.04%	94.65%	82.3	54.2	28.1	5.0
	NIC	96.77%	94.10%	68.2	50.3	17.9	4.6
	LEM	98.34%	95.53%	61.1	43.8	17.3	4.0
	EN	98.20%	95.48%	246.2	47.3	198.9	4.4
Zoo	Union	96.89%	88.00%	48.1	37.9	10.2	4.1
	Special	95.56%	90.44%	52.8	38.2	14.6	4.8
	NIC	100.00%	92.67%	44.5	35.5	9.0	3.0
	LEM	100.00%	92.44%	44.3	35.4	8.9	3.0
	EN	100.00%	92.89%	77.5	35.6	41.9	3.1

Average	Union	94.46%	90.09%	301.8	198.9	102.9	6.5
	Special	94.44%	91.35%	316.0	122.2	193.8	6.3
	NIC	94.39%	90.73%	223.7	178.5	45.2	4.3
	LEM	96.61%	92.01%	204.5	164.8	39.7	4.2
	EN	96.31%	92.23%	868.2	145.2	723.1	4.1

The Zoo data has similar counts for all models but the train and test accuracy is 2–4% better for the strength estimation models.

The averages over all the data sets show the expected improvement in performance for the strength estimation algorithms. An exception is accuracy for NIC strength estimation. The average for NIC strength estimation is low because of the poor result on the LED data set. NIC specialization may have problems handling noise. Performance improves over the three strength estimation algorithms as is expected since EN has more information available to it than LEM which has more information than NIC.

The averages of create, node, and pass counts also show the advantage of the strength estimation methods. EN strength estimation has more nodes created as expected. Note the steady decline in deleted nodes over the three strength estimation methods. This result agrees with the amount of information available to each method for the purpose of estimating the strength of new features. The number of passes required for Union and Specialization is about 1.5 times the number needed for the strength estimation methods.

In summary, all models perform similarly on Breast Cancer, Hepatitis, Iris, Mushroom, Soybean, and Voting. Strength estimation models do better than other models on Chess, LED, Tic-Tac-Toe, and Zoo. Overall, the strength estimation models appear to perform better than union and specialization as expected.

9.5. GS Variations

The GS learning algorithm can be run with two variations. Single-input nodes can either be created or not created and specialization-with-strength-estimation can either be done one time or repeated. The base case for GS learning is to create single-input nodes and to do specialization one time. The following two sections give results for the algorithm with the two variations.

9.5.1. Strength Estimation without Single-Input Nodes

The specialization methods that use strength estimation can operate without creating single-input nodes. This section compares the results for these methods without single-input nodes to the base results reported above. « GSnoSI» shows the test accuracy percentages for the three strength estimation methods both with and without single-input node creation.

Strength estimation without single-input nodes was expected to perform as well as strength estimation with single-input nodes because strength estimation does not require single-input nodes to choose good new features. However, with few exceptions, the table shows that each strength estimation method performs better when single-input nodes are created than when single-input nodes are not created. As expected, though not shown in the table, more nodes are created and deleted when single-input nodes are used. Pass counts typically are lower when single-input nodes are used.

Table 9.3. Test Accuracy for GS without Single-Input Nodes

Data Set	NIC	NIC w/o SI	LEM	LEM w/o SI	EN	EN w/o SI
Breast Cancer	95.06	94.26	95.26	94.80	95.03	95.26
Chess	97.83	97.51	97.58	97.45	97.73	97.57
Hepatitis	80.00	79.36	80.26	79.87	81.03	81.15
Iris	93.60	85.87	92.93	85.47	92.80	86.13
LED	62.60	60.36	71.80	71.70	73.00	73.06

Mushroom	99.98	99.97	99.93	99.92	99.94	99.93
Soybean	97.08	91.25	97.08	90.00	97.08	90.00
Tic Tac Toe	94.41	95.14	97.29	97.31	97.37	97.08
Voting	94.10	94.75	95.53	95.62	95.48	95.30
Zoo	92.67	91.11	92.44	90.00	92.89	90.00
Average	90.73	88.96	92.01	90.21	92.23	90.55

9.5.2. Repeat Specialization

LEM and EN specialization can repeatedly specialize a feature by adding more than one input/value pair to a current feature. All other GS learning methods store too little information about examples that have been presented to support such repeat specialization. This section compares the results for LEM and EN specialization using repetition to the base results reported above. « GSRep» shows the test accuracy percentages for LEM and EN specialization both with and without repetition.

Table 9.4. Test Accuracy for GS with Repetition

Data Set	LEM	LEM w/Rep	EN	EN w/Rep
Breast Cancer	95.26	95.17	95.03	95.03
Chess	97.58	97.07	97.73	98.27
Hepatitis	80.26	80.64	81.03	81.41
Iris	92.93	92.40	92.80	92.40
LED	71.80	71.64	73.00	73.00
Mushroom	99.93	99.93	99.94	99.95
Soybean	97.08	97.08	97.08	97.08
Tic Tac Toe	97.29	97.89	97.37	98.39
Voting	95.53	94.52	95.48	95.02
Zoo	92.44	92.22	92.89	92.67
Average	92.01	91.86	92.23	92.32

Specialization with repetition was expected to perform better than specialization without repetition because repetition should allow the system to find the correct specific feature without creating intermediate features. However, the table shows that each specialization method performs about the same with or without repetition.

9.6. SG Results

Results for the two base SG learning models on the 10 data sets are shown in « SGResults». The table also gives results for feature-generalization without repetition and GS learning using EN strength estimation. Base SG models have similar good results for Breast Cancer, Chess, LED, and Voting.

Hepatitis test accuracy results for both base models are as expected from previous results. Training accuracy is somewhat low for feature-generalization.

Feature-intersection has poor accuracy on Iris and Soybean while feature-generalization does well on the same data. Feature-generalization performs poorly on Mushroom and Tic-Tac-Toe while feature-intersection does well. Both algorithms have low test accuracy on the Zoo data.

Feature-intersection has better create, delete, node, and pass counts than feature-generalization on LED and Mushroom.

The averages over all data sets are very close for the two base SG models. Test accuracy for SG models is 3–4% lower than the GS learning algorithm. This suggests that starting with general features provides a better generalization result than starting with specific features. However, the SG model using feature-generalization performs better than GS on Iris and Soybean. Thus, SG can outperform GS on specific types of data.

The create, delete, node, and pass counts are higher for the GS algorithm than for the base SG models. GS creates about twice as many nodes on the Hepatitis, Soybean, and Zoo data. For the LED data, GS creates almost 10 times as many nodes. The GS algorithm typically deletes an order of magnitude more nodes than the SG models. For the LED data, GS deletes almost 100 times as many nodes as feature-intersection. So, although GS may do better than SG in terms of generalization performance, SG does have the strength of reducing the number of nodes created and deleted as well as the pass count.

Table 9.5. SG Results

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Breast Cancer	Intersect	99.86%	94.71%	279.6	4.9	274.7	2.9
	General	99.77%	92.83%	293.1	9.8	283.3	3.3
	Gen no Rep	100.00%	66.60%	375.4	0.0	375.4	2.3
	GS EN	100.00%	95.03%	345.1	59.1	286.0	3.7
Chess	Intersect	99.98%	97.94%	1797.4	136.7	1660.7	3.4
	General	100.00%	96.65%	1800.3	122.9	1677.4	3.4
	Gen no Rep	100.00%	48.98%	3188.6	4.8	3183.8	3.0
	GS EN	98.60%	97.73%	1874.1	221.0	1653.1	5.6
Hepatitis	Intersect	98.44%	79.87%	106.6	16.8	89.8	5.1
	General	89.87%	81.15%	85.1	2.4	82.7	2.7
	Gen no Rep	100.00%	78.59%	94.3	0.0	94.3	2.1
	GS EN	96.36%	81.03%	168.7	69.7	99.0	5.7
Iris	Intersect	98.67%	90.93%	84.7	4.0	80.7	2.5
	General	100.00%	94.53%	81.8	1.5	80.3	3.1
	Gen no Rep	100.00%	66.67%	113.0	1.9	111.1	2.8
	GS EN	98.80%	92.80%	103.5	18.5	85.0	3.2
LED	Intersect	65.08%	71.58%	117.4	8.3	109.1	3.3
	General	74.36%	71.98%	157.9	42.2	115.7	3.7
	Gen no Rep	75.68%	70.92%	175.3	40.3	135.0	5.0

	GS EN	72.72%	73.00%	984.0	751.5	232.5	6.2

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Mushroom	Intersect	100.00%	99.99%	4102.3	15.3	4087.0	2.2
	General	94.10%	93.86%	4202.1	94.7	4107.4	4.8
	Gen no Rep	100.00%	76.95%	6901.9	0.0	6901.9	3.0
	GS EN	99.96%	99.94%	4177.0	87.7	4089.3	3.3
Soybean	Intersect	100.00%	79.58%	33.4	6.5	26.9	2.1
	General	100.00%	97.92%	29.8	2.3	27.5	2.4
	Gen no Rep	100.00%	24.17%	43.7	0.0	43.7	2.8
	GS EN	100.00%	97.08%	88.2	60.5	27.7	2.7
Tic Tac Toe	Intersect	100.00%	97.10%	583.0	69.3	513.7	2.9
	General	100.00%	81.02%	650.0	67.7	582.3	3.6
	Gen no Rep	100.00%	64.99%	645.9	0.0	645.9	2.1
	GS EN	98.46%	97.37%	617.8	100.7	517.1	3.2
Voting	Intersect	99.22%	95.16%	217.8	17.2	200.6	5.2
	General	96.96%	92.12%	212.5	13.2	199.3	3.8
	Gen no Rep	100.00%	77.56%	259.2	0.0	259.2	2.5
	GS EN	98.20%	95.48%	246.2	47.3	198.9	4.4
Zoo	Intersect	98.22%	81.33%	46.0	6.8	39.2	2.1
	General	100.00%	83.78%	53.5	8.3	45.2	2.8
	Gen no Rep	100.00%	66.22%	57.9	0.0	57.9	2.1
	GS EN	100.00%	92.89%	77.5	35.6	41.9	3.1

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Average	Intersect	95.95%	88.82%	736.8	28.6	708.2	3.2
	General	95.51%	88.58%	756.6	36.5	720.1	3.4
	Gen no Rep	97.57%	64.16%	1185.5	4.7	1180.8	2.8
	GS EN	96.31%	92.23%	868.2	145.2	723.1	4.1

The only variation for SG algorithms is repetition for feature-generalization. The base configuration for feature-generalization is to use repetition. Without repetition the average test accuracy for the 10 data sets drops to 64%. Thus, repetition is important for feature-generalization. It appears that without repetition, features do not expand around the training examples sufficiently to cover the complete input space.

In summary, GS has better accuracy than SG, but SG creates and deletes fewer nodes and requires fewer training passes than GS.

9.7. GAS Results

Results for GAS learning on the 10 data sets are shown in « GASResults». The table also gives results for GS learning using EN specialization and SG learning using feature-generalization. The base results for GAS do not use single-input nodes so the GS results are also those without single-input nodes.

GAS learning is expected to perform better than either GS or SG learning since GAS combines GS and SG. GAS learning that selects the best feature of specialization or generalization is expected to do better than GAS learning that creates both features since fewer poor features should be created. It is expected that the GAS model that creates both generalization and specialization nodes should create more nodes.

All four models have similar results for Breast Cancer and Chess. SG is somewhat lower than the others on test accuracy for Breast Cancer. GS is worse than the others on pass count for Chess. GAS avoids the SG and GS problems on these data sets and obtains close to the GS accuracy result on Breast Cancer and the SG pass count result on Chess. On these data sets GAS is able to get the benefits of the individual methods without suffering the drawbacks.

GAS has close to the GS training accuracy on Hepatitis, LED, and Voting. However, GAS also suffers from the higher node count and pass count problems of GS on these data sets. GAS is not able to benefit from the lower counts that SG is able to achieve.

SG has better performance than GS in every category on the Iris data. GAS creating the best node gets exactly the same as the SG result and avoids the GS problems.

GAS does better than both GS and SG in every category on the Mushroom data.

GAS obtains close to the accuracy of SG on the Soybean data.

The GAS model that creates the best node gets results almost identical to SG on the Tic-Tac-Toe data. Unfortunately, SG results on Tic-Tac-Toe are poor. GAS creating both nodes obtains the better results of the GS model.

On the Zoo data, GAS is able to get the better SG result on training accuracy, the better GS result on test accuracy, and the better SG result on pass count.

Table 9.6. GAS Results

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Breast Cancer	Both	99.89%	94.49%	314.6	31.2	283.4	3.5
	Best	99.86%	94.23%	291.5	11.2	280.3	3.2
	GS	99.37%	95.26%	290.1	14.3	275.8	3.9
	SG	99.77%	92.83%	293.1	9.8	283.3	3.3
Chess	Both	100.00%	97.60%	1893.5	224.8	1668.7	3.3
	Best	100.00%	96.75%	1793.7	116.5	1677.2	3.3
	GS	98.49%	97.57%	1808.0	155.2	1652.8	5.1
	SG	100.00%	96.65%	1800.3	122.9	1677.4	3.4
Hepatitis	Both	96.10%	79.49%	123.6	29.4	94.2	5.7
	Best	96.49%	80.51%	116.6	22.9	93.7	5.5
	GS	97.40%	81.15%	113.4	21.9	91.5	5.8
	SG	89.87%	81.15%	85.1	2.4	82.7	2.7
Iris	Both	100.00%	93.73%	91.5	8.9	82.6	3.0
	Best	100.00%	94.53%	81.8	1.5	80.3	3.1
	GS	93.20%	86.13%	88.6	7.1	81.5	3.8
	SG	100.00%	94.53%	81.8	1.5	80.3	3.1
LED	Both	76.30%	72.66%	965.3	727.6	237.7	6.3
	Best	76.60%	72.24%	855.9	618.9	237.0	6.0
	GS	72.02%	73.06%	954.7	723.3	231.4	6.1
	SG	74.36%	71.98%	157.9	42.2	115.7	3.7
Mushroom	Both	100.00%	100.00%	4107.8	23.1	4084.7	2.0
	Best	100.00%	100.00%	4095.5	11.2	4084.3	2.0
	GS	99.98%	99.93%	4114.7	26.2	4088.5	4.0
	SG	94.10%	93.86%	4202.1	94.7	4107.4	4.8
Soybean	Both	100.00%	97.08%	36.6	8.6	28.0	2.4
	Best	100.00%	93.75%	32.2	4.6	27.6	2.4
	GS	94.35%	90.00%	31.2	3.6	27.6	2.4
	SG	100.00%	97.92%	29.8	2.3	27.5	2.4
Tic Tac Toe	Both	100.00%	96.12%	719.5	178.8	540.7	3.3
	Best	100.00%	80.92%	649.0	66.2	582.8	3.6
	GS	98.41%	97.08%	602.5	87.6	514.9	3.1
	SG	100.00%	81.02%	650.0	67.7	582.3	3.6
Voting	Both	98.80%	93.64%	236.5	35.2	201.3	4.0
	Best	98.85%	93.87%	229.9	30.2	199.7	5.9
	GS	98.11%	95.30%	228.3	30.4	197.9	5.6

	SG	96.96%	92.12%	212.5	13.2	199.3	3.8
Zoo	Both	100.00%	88.67%	61.3	17.7	43.6	2.9
	Best	100.00%	90.00%	50.9	8.1	42.8	2.8
	GS	99.56%	90.00%	47.2	5.6	41.6	3.8
	SG	100.00%	83.78%	53.5	8.3	45.2	2.8

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Average	Both	97.11%	91.35%	855.0	128.5	726.5	3.6
	Best	97.18%	89.68%	819.7	89.1	730.6	3.8
	GS	95.09%	90.55%	827.9	107.5	720.4	4.4
	SG	95.51%	88.58%	756.6	36.5	720.1	3.4

The average over all data sets is higher for GAS learning creating both features; the opposite of what is expected. It is suspected that this is because the SG part of GAS learning dominates the GS part when only the best feature is created.

On average, the best GAS model is about 1% more accurate than the GS model. At the same time the GAS model is 3% better than the SG model. On average, GAS is closer to SG in pass count.

Overall, GAS is able to obtain the better characteristics of either GS or SG on many data sets. On other data sets, GAS gets results similar to either GS or SG alone, without being able to combine the best of the two approaches.

9.8. Direct Specialization Strength Estimation

Direct specialization strength estimation is an approximation to EN strength estimation. DS strength estimation is potentially more efficient than EN strength estimation, but is not guaranteed to give the same result. This section compares simulation results for DS strength estimation with results for EN strength estimation to determine if DS strength estimation gives good strength estimates.

Results for DS strength estimation are given for SG learning using intersection and for GAS learning. GAS learning uses both feature-specialization and feature-generalization using EN strength estimation. Thus, good results for DS strength estimation on GAS learning should indicate good results for both feature-specialization and feature-generalization alone.

Results for feature-intersection using both EN and DS strength estimation and GAS learning using both EN and DS strength estimation are shown in « DSResults». On many data sets (Chess, Iris, LED, Mushroom, Soybean, Tic-Tac-Toe, Zoo) DS strength estimation gives results exactly the same as EN strength estimation. On other data sets DS results are slightly lower than EN results. On average DS strength estimation is very close to EN strength estimation in every category.

Table 9.7. DS Strength Estimation Results

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Breast Cancer	Intersect	99.86%	94.71%	279.6	4.9	274.7	2.9
	Inter DS	99.89%	94.69%	279.7	5.0	274.7	2.8
	GAS	99.86%	94.23%	291.5	11.2	280.3	3.2
	GAS DS	100.00%	93.86%	291.5	11.1	280.4	3.1
Chess	Intersect	99.98%	97.94%	1797.4	136.7	1660.7	3.4
	Inter DS	99.98%	97.94%	1797.4	136.7	1660.7	3.4
	GAS	100.00%	96.75%	1793.7	116.5	1677.2	3.3
	GAS DS	100.00%	96.86%	1794.3	117.3	1677.0	3.3
Hepatitis	Intersect	98.44%	79.87%	106.6	16.8	89.8	5.1
	Inter DS	99.48%	79.23%	93.9	5.6	88.3	2.6
	GAS	96.49%	80.51%	116.6	22.9	93.7	5.5
	GAS DS	99.61%	78.59%	102.9	8.7	94.2	3.1
Iris	Intersect	98.67%	90.93%	84.7	4.0	80.7	2.5
	Inter DS	98.67%	90.93%	84.7	4.0	80.7	2.5
	GAS	100.00%	94.53%	81.8	1.5	80.3	3.1
	GAS DS	100.00%	94.53%	81.8	1.5	80.3	3.1
LED	Intersect	65.08%	71.58%	117.4	8.3	109.1	3.3
	Inter DS	65.08%	71.58%	117.4	8.3	109.1	3.3
	GAS	76.60%	72.24%	855.9	618.9	237.0	6.0
	GAS DS	76.52%	72.36%	851.6	613.7	237.9	5.9

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Mushroom	Intersect	100.00%	99.99%	4102.3	15.3	4087.0	2.2
	Inter DS	100.00%	99.99%	4102.3	15.3	4087.0	2.2
	GAS	100.00%	100.00%	4095.5	11.2	4084.3	2.0
	GAS DS	100.00%	100.00%	4095.5	11.2	4084.3	2.0
Soybean	Intersect	100.00%	79.58%	33.4	6.5	26.9	2.1
	Inter DS	100.00%	79.58%	33.4	6.5	26.9	2.1
	GAS	100.00%	93.75%	32.2	4.6	27.6	2.4
	GAS DS	100.00%	93.75%	32.2	4.6	27.6	2.4
Tic Tac Toe	Intersect	100.00%	97.10%	583.0	69.3	513.7	2.9
	Inter DS	100.00%	97.10%	583.0	69.3	513.7	2.9
	GAS	100.00%	80.92%	649.0	66.2	582.8	3.6
	GAS DS	100.00%	80.92%	649.0	66.2	582.8	3.6
Voting	Intersect	99.22%	95.16%	217.8	17.2	200.6	5.2
	Inter DS	99.68%	94.56%	209.7	10.1	199.6	3.0
	GAS	98.85%	93.87%	229.9	30.2	199.7	5.9

	GAS DS	99.95%	92.67%	219.4	17.1	202.3	3.7
Zoo	Intersect	98.22%	81.33%	46.0	6.8	39.2	2.1
	Inter DS	98.22%	81.33%	46.0	6.8	39.2	2.1
	GAS	100.00%	90.00%	50.9	8.1	42.8	2.8
	GAS DS	100.00%	90.00%	50.9	8.1	42.8	2.8

Data Set	Algorithm	Train Accuracy	Test Accuracy	Create Count	Delete Count	Node Count	Pass Count
Average	Intersect	95.95%	88.82%	736.8	28.6	708.2	3.2
	Inter DS	96.10%	88.69%	734.8	26.8	708.0	2.7
	GAS	97.18%	89.68%	819.7	89.1	730.6	3.8
	GAS DS	97.61%	89.35%	816.9	86.0	731.0	3.3

9.9. Conclusion

Results of testing the learning algorithms in this paper suggest that GS algorithms have better accuracy than SG algorithms. However, SG algorithms create and delete fewer nodes than GS algorithms. SG algorithms also require fewer passes over the training examples. On many data sets, when GS and SG have different results, GAS algorithms are able to obtain the better of the two results. On other data sets, GAS algorithms get results at least as good as either GS or SG alone.

Chapter 10
Conclusion

10.1. Summary

This paper has presented a family of algorithms for inductive machine learning. The algorithms combine the benefits of neural networks, ASOCS, and symbolic learning algorithms. The algorithms learn incrementally with good speed and generalization. The algorithms are based on a parallel architectural model that adapts to the problem being learned. Learning is done without requiring user adjustment of sensitive parameters and noise is tolerated with graceful degradation in performance.

The approach to development of the systems in this work was to study current approaches to machine learning. The strengths of current systems were identified along with the characteristic of the system that provided the strength. Neural networks can tolerate noise because each training example causes only incremental changes in the state of the system. ASOCS models learn quickly and adapt to a large class of problems because of a dynamic parallel architectural model. Symbolic AI induction algorithms generalize at a symbolic level by using generalization rules.

The techniques of (1) using incremental changes to the state of the system, (2) dynamically creating and deleting nodes in a parallel network, and (3) symbolic representations modified with generalization rules were combined to create the new models for inductive learning in this work. The new models were thus able to gain the benefits of the various current approaches.

An explanation of the meaning of learning from examples was given in Chapter 2. This provided the needed foundation to explain the idea of a feature in Chapter 3. The concept of a feature was described as an important set of states in the input space. The feature then became the basic building block for construction of networks to learn general mappings from input to output. Relationships between features and the input space were proven. This provided the basis for creating generalizations or specializations of existing features.

Chapter 4 showed that it is possible to begin with general features and specialize those features to match a problem that is being learned. Five different methods were given for creating more specific features. New specific features were created by either computing the union of existing features or by adding inputs to existing features. The goodness of new features was measured either based on the strength and generality of the existing features or by estimating the strength of the new feature. The strength estimation methods required remembering additional information about training examples.

Chapter 5 showed that an approach almost opposite to that in Chapter 4 can learn features to match a problem. Specific features were first created. These features were generalized by either intersection or removing inputs. The generalization of features was controlled by contradiction with other features.

GS learning and SG learning were then combined in Chapter 6 resulting in a model that gives the benefits of both GS and SG learning.

Two possible architectural models were presented in Chapter 7. The models allow a large number of nodes to operate in parallel. The broadcast/gather model is suitable for implementation using static interconnects. The general/specific model requires dynamic interconnection but provides better learning performance. Dynamic interconnects can be emulated using a system with static connections.

The algorithms were shown to have good complexity characteristics in Chapter 8 both in terms of space and time. All algorithms were shown to be $O(m \log m)$, where m is the number of nodes, when the number of inputs and output values can be treated as constants. When the general/specific network topology is used, the time complexity becomes $O(\log m)$ because m nodes are no longer broadcasting new features to the entire network.

Chapter 9 gave results from testing the algorithms on ten data sets. The results indicate that the algorithms give good generalization and that learning converges in a small number of training passes. GS models were shown to have better accuracy than SG models, but SG models were shown to converge in fewer passes. SG models were also shown to create and delete fewer nodes. GAS models were shown to have potential to gain the benefits of both GS and SG models.

In summary, several new algorithms for inductive learning have been presented. The new algorithms compare well with other learning models. Learning time for the new algorithms is $O(\log m)$, where m is the number of network nodes, when the newly developed GS network topology is used. The accuracy of the new models compares well with previous results on several real world data sets. The new models are easy to use and can be applied to many learning problems.

10.2. Future Research

Several possibilities exists for future extensions and improvements to the algorithms given here. Though the models presented here have a good combination of benefits, they do still have some weaknesses.

The current algorithms have a limited representation for features. Each element of a feature restricts an input to be equal to a value. Features should be extended so that inputs can be specified to cover a range of values. Features could also specify more complex relationships between inputs. For example, a feature could specify that two inputs must be equal or that the sum of several inputs must exceed some threshold.

Such extensions of feature representation would require the feature generality and specificity measures to be enhanced to handle the new complexity of the representation. Feature generalization and specialization operations would need modification. These feature representation improvements would potentially allow the algorithms to learn a larger class of functions. The more complex feature representations would require new methods for learning good features.

The current models are designed to only accept nominal inputs. Continuous data must be discretized before the data can be use for training the system. Using a more general feature representation as described above, feature creation and measurement could be enhanced to learn good ranges of continuous inputs.

The learning models presented in this work are composed of several building blocks. New features can be created from existing features using Union, Intersection, Specialization, and Generalization. The strength of new features can be estimated using Next-Input Counters, Limited Example Memory, and Example Nodes. Features can be deleted either by measuring node usage or by comparing node strength. These various building blocks can be combined in many new and different ways. Analysis and testing of these new combinations may provide fruitful areas for future research.

Applications for the models developed here should be identified. The models are well suited to decision problems as shown by several of the data sets given in the empirical results. Medical diagnoses, credit screening, and character recognition are problems that can easily be presented to the systems given here.

Problems requiring adaptive response are suitable for these systems. The systems learn incrementally such that learning and execution can be interleaved. Applications that need fast adaptation can be done by the systems presented here because of the ability of these systems to learn new information quickly. Examples of such applications are adaptive network routing, adaptive computer vision, and adaptive knowledge based systems.

Bibliography

Ackley, D. H., Hinton, D. E., & Sejnowski, T. J. (1985). A Learning Algorithm for Boltzmann Machines. *Cognitive Science, 9*, 147-169.

Aha, D. W. (1991). Incremental constructive induction: An instance-based approach. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp 117-121. Evanston, Il: Morgan Kaufmann.

Anderson, J. R. (1989). A Theory of the Origins of Human Knowledge. *Artificial Intelligence, 40,* 313-352.

Barker, J. C. and Martinez, T. R. (1993a). GS: A Network that Learns Important Features. *Proceedings of The World Congress on Neural Networks*, Portland, Oregon, July 11-15, 1993.

Barker, J. C. and Martinez, T. R. (1993b). Learning and Generalization Controlled by Contradiction. *Proceedings of The International Conference on Artificial Neural Networks*, Amsterdam, Netherlands, September 13-16, 1993.

Barker, J. C. and Martinez, T. R. (1993c). Generalization by Controlled Intersection of Examples. *Proceedings of The Sixth Australian Joint Conference on Artificial Intelligence*, Melbourne, November, 1993.

Barker, J. C. and Martinez, T. R. (1994). Proof of Correctness for ASOCS AA3 Networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 24, 3, Mar. 1994.

Booker, L. B., Goldberg, D. E., & Holland, J. H. (1989). Classifier Systems and Genetic Algorithms. *Artificial Intelligence, 40*, 235-282.

Brieman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group: Belmont, California. pp 43-49.

Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., & Freeman, D. (1988). AUTOCLASS: A Bayesian Classification System. *Proceedings of the Fifth International Conference on Machine Learning*, (pp. 54-64). Ann Arbor, MI: Morgan Kaufmann.

Congressional Quarterly Almanac, 98th Congress, 2nd session 1984, Volume XL: Congressional Quarterly Inc. Washington, DC., 1985.

DeJong, G. F., & Mooney, R. J. (1986). Explanation-based Learning: An Alternative View. *Machine Learning, 1*, 145-176.

Dietterich, T. G. (1986). Learning at the Knowledge Level. *Machine Learning*, 1, 287-316.

Fisher, D. H. (1987a). *Knowledge Acquisition via incremental conceptual clustering*. Doctoral Dissertation, Department of Information and Computer Science, University of California, Irvine, CA.

Fisher, D. H. (1987b). Knowledge Acquisition via Incremental Conceptual Clustering. *Machine Learning, 2*, 139-172.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annual Eugenics, 7,* Part II, 179-188.

Gentner, D. (1989). Mechanisms of Analogical Learning. In S. Vosniadou & A. Ortony (Eds.), *Similarity and Analogical Reasoning*. London: Cambridge University Press.

Grefenstette, J. J.(1988). Credit Assignment in Rule Discovery Systems Based on Genetic Algorithms. *Machine Learning, 3*, 225-245.

Hopfield, J. J. & Tank, D. W. (1985). Neural Computation of Decisions in Optimization Problems. *Biological Cybernetics, 52*, 141-152.

Judd, J. S. (1988). On the Complexity of Loading Shallow Neural Networks. *Journal of Complexity, 4*, 177-192.

Kibler, D. & Aha, D. W. (1987). Learning Representative Exemplars of Concepts: An Initial Case Study. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 24-30). Irvine, CA: Morgan Kaufmann.

Kibler, D. & Langley, P. (1988). Machine Learning as an Experimental Science. *Proceedings of the Third European Working Session on Learning*. Glasgow, Scotland: Pitman.

Kodratoff, Y. and Michalski, R. S. (Eds.) (1990). *Machine Learning: An Artificial Intelligence Approach, Volume III,* Morgan Kaufmann Publishers, San Mateo, CA.

Lang, K. J., Waibel, A. H., & Hinton, G. E. (1990). A Time-delay Neural Network Architecture for Isolated Word Recognition. *Neural Networks, 3*, 23-43.

Langley, P., Simon, H. A., & Bradshaw, G. L. (1987). Heuristics for Empirical Discovery. In L. Bolc (Ed.), *Computational Models of Learning*, Springer Verlag.

Lenat, D. B. (1977). The Ubiquity of Discovery. *Artificial Intelligence, 9*, 257-285.

Mangasarian, O. L. and Wolberg W. H. (1990). Cancer diagnosis via linear programming. *SIAM News*, 23, 5, 1 - 18.

Martinez, T. R. (1986). Adaptive Self-Organizing Logic Networks, Ph.D. Dissertation, UCLA Technical Report - CSD 860093, (274 pp.), June 1986.

Martinez, T. R. & Vidal, J. J. (1988). Adaptive Parallel Logic Networks. *Journal of Parallel and Distributed Computing*, 5, 26-58.

Martinez, T. R., & Campbell, D. M. (1991a). A Self-Adjusting Dynamic Logic Module. *Journal of Parallel and Distributed Computing, 11*, 303-313.

Martinez, T. R. & Campbell, D. M. (1991b). A Self-Organizing Binary Decision Tree for Incrementally Defined Rule Based Systems. *IEEE Transactions on System, Man, and Cybernetics, 21*.

Martinez, T. R., Barker, J. C., and Giraud-Carrier, C. (1993). A Generalizing Adaptive Discriminant Network. *Proceedings of The World Congress on Neural Networks*, Portland, Oregon, July 11-15, 1993.

Michalski, R. S. (1983a). A Theory and Methodology of Inductive Learning. *Artificial Intelligence, 20,* 111-161.

Michalski, R. S., Carbonell, J., and Mitchell, T. (Eds.) (1983b). *Machine Learning: An Artificial Intelligence Approach,* Morgan Kaufmann Publishers, San Mateo, CA.

Michalski, R. S., Carbonell, J., and Mitchell, T. (Eds.) (1986). *Machine Learning: An Artificial Intelligence Approach, Volume II,* Morgan Kaufmann Publishers, San Mateo, CA.

Minsky, M. L. & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry.* Cambridge MA: MIT Press.

Mitchell, T. M. (1980). *The need for biases in learning generalizations* (Technical Report No. CBM-TR-117). New Brunswick, NJ: Rutgers University, Department of Computer Science.

Mitchell, T. M. (1982). Generalization as Search. *Artificial Intelligence, 18,* 203-226.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-Based Generalization: A Unifying View. *Machine Learning, 1*, 47-80.

Mooney, R. J., Shavlik, J. W., Towell, G. G., & Gove, A. (1989). An Experimental Comparison of Symbolic and Connectionist Learning Algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 775-780). Detroit: Morgan Kaufmann.

Murphy, P. M. & Aha, D. W. (1992). *UCI Repository of machine learning databases*. Irvine, CA: University of California, Department of Information and Computer Science.

Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning, 1,* 81-106.

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review, 65,* 386-408.

Rumelhart, D. E. and McClelland, J. (1986a). *Parallel Distributed Processing*, Vol. 1, Cambridge, MA. MIT Press.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986b). Learning Internal Representations by Error Propagation. In D. E. Rumelhart and J. L. McClelland (Eds.) *Parallel Distributed Processing* (Vol. 1). Cambridge, MA: MIT Press.

Rumelhart, D. E. & Zipser D. (1985). Feature Discovery by Competitive Learning. *Cognitive Science, 9*, 75-112.

Schlimmer, J. S. (1987). *Concept Acquisition Through Representational Adjustment*. Technical Report 87-19, Doctoral Dissertation, Department of Information and Computer Science, University of California, Irvine.

Shavlik, J. W. & Towell, G. G. (1989). An approach to combining explanation-based and neural learning algorithms. *Connection Science, 1*, 233-255.

Shavlik, J. W. (1990). Acquiring Recursive and Iterative Concepts with Explanation-Based Learning. *Machine Learning, 5*, 39-70.

Stanfill, G. & Waltz, D. (1986). Toward Memory-Based Reasoning. *Communications of the ACM, 29:12*, 1213-1228.

Valiant, L. G. (1984). A Theory of the Learnable. *Communications of the ACM, 27*, 1134-1142.

Weiss, S. & Kapouleas, I. (1989). An Empirical Comparison of Pattern Recognition, Neural Nets, and Machine Learning Classification Methods. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 781-787). Detroit: Morgan Kaufmann.