Chapter 1

Introduction

Search is a very general problem-solving technique in artificial intelligence (AI), the archetypal "weak method" for attacking problems in the absence of strong domain constraints. However, generality is not without a price. Generic search strategies are inefficient compared to other, more specialized, problem-solving methods. In order to improve search, recent research leans toward domain-specific knowledge and the development of techniques to use it effectively. While this promises better and more practical algorithms, it suffers from the disadvantage that the resulting algorithms are special-purpose and do not apply to other domains. Of course, this may be inevitable, for in many cases acceptable performance is unattainable without the use of specialized domain knowledge. Even so, the development of more effective general search strategies provides a significant step towards further improvements of specialized strategies. That is the motivation behind this thesis research.

Because it is difficult to develop algorithms without a specific problem domain to work with, this thesis concentrates on solving the class of function induction problems. As these problems are frequently encountered in the field of inductive learning, a major research area in AI, any algorithm developed will have immediate use in many application domains. To maintain generality, domain-specific knowledge is not embedded within the search algorithm. The result is a general and efficient search strategy that potentially applies to other problems as well.

This chapter starts with a review of search in general and how it applies to the problem of function induction. It then states the main research goal and concludes with a brief preview of subsequent chapters.

1.1 Search as a general problem-solving technique

With the *problem space model* proposed by Newell and Simon [Newell 72], search has gained increasing acceptance over the years as a general purpose problem-solving technique. Under this model, the *problem space* is defined by two components, a set of problem states and a collection of state-mapping operators. An operator is usually a partial mapping since it may have preconditions which restrict the states to which it can be applied. Members of the space are tuples comprising a start state, an end state and a sequence of state-mapping operators. For any particular problem, the input data is the start state, the desired output is the end state, and the state-mapping operators are those that can manipulate the inputs. To solve a problem is to search its problem space exhaustively for a member that has the same start state, end state, or mapping sequence. Which item is sought depends on how the problem is posed. If the start and end states are unknown, the algorithm finds them by matching the mapping sequence. If the sequence is unknown, it uses the start and end states to find the sequence which transforms one into the other.

To illustrate the generality of the problem space model, consider the problem of sorting a list of items, say $\{c, a, b\}$. Normally, one does not think of sorting as a search problem, but it can nevertheless be formulated as such. In particular, the starting state is the input list $\{c, a, b\}$, the ending state is the desired output list $\{a, b, c\}$, and the mapping operators swap two items in the list, $swap_{1,2}$, $swap_{1,3}$ and $swap_{2,3}$. To solve this problem, one searches for a sequence of swap operations that maps the input list $\{c, a, b\}$ to the output list $\{a, b, c\}$. In this example, the solution is the sequence of operations

$$\{c, a, b\} \xrightarrow{swap_{1,3}} \{b, a, c\} \xrightarrow{swap_{1,2}} \{a, b, c\}.$$

1.1.1 A general abstract model

A general abstract model of problem space is a search graph. Each node in the graph represents a problem state, and each edge an operator. To solve the problem is to

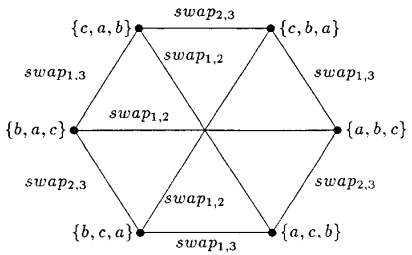


Figure 1.1: A search graph

find a path going from the start node to the end node. The solution is the sequence of edges on that path. As an example, the search graph for the above sorting problem is illustrated in Figure 1.1. Notice that this graph represents the problem of sorting any 3-element list, not just $\{c, a, b\}$. For instance, it reveals the sorting solution for the list $\{b, c, a\}$ to be

$$\{b, c, a\} \overset{swap_{1,3}}{\longrightarrow} \{a, c, b\} \overset{swap_{2,3}}{\longrightarrow} \{a, b, c\}$$

Under the problem space model, the complexity of a search algorithm is characterized by two parameters, the search graph's *average branching factor* and its *solution depth*. The average branching factor is the average number of choices in moving from one node to another. This corresponds to the average number of mappings one can potentially apply to a problem state. Obviously, a larger branching factor results in a slower search for the end state, for more choices are possible. And since there could be many paths leading to that state, the solution depth is defined as the length of the shortest path. The graph of Figure 1.1 has average branching factor 3 and solution depth 2 for the sorting problem of $\{c, a, b\}$.

1.1.2 Two classic search strategies

Consider the sorting problem of $\{c, a, b\}$ according to the search graph in Figure 1.1. Suppose that the algorithm applies the operator $swap_{1,3}$ to the start node $\{c, a, b\}$ and thus arrives at node $\{b, a, c\}$. Since this is not an end node, there are two possibilities, either the choice of $swap_{1,3}$ is correct or it is not. A choice is "correct" if it leads to the shortest path solution. In the former case, the algorithm should continue by applying another operator to the current node, as it is now closer to the end node. In the latter case, it should try to apply another operator to the previous node. Obviously, when an algorithm does not know whether the choice is correct, it can take either action but must allow the possibility of considering the other alternative. Algorithms that prefer to continue exploring the current choice are called depth-first searches, while ones that prefer to try many alternative choices simultaneously are called breadth-first searches.

Since breadth-first search investigates all possible alternatives (in the current depth) when the previous choice does not lead to an end state, it always find the solution with minimal depth. That is because it first examines paths of length 1, then those of length 2, and so on until one is found connecting the start and end nodes. Since extending paths of length k with one more operator produces paths of length $k + 1$, much work can be avoided by saving the former as they are constructed (from paths of length $k - 1$). In the worst case, all paths of length d must be examined, where d is the solution depth. The total number of paths generated and stored is $b + b^2 + \cdots + b^d$ which is $O(b^d)$, where b is the average branching factor. This means that breadth-first search in general requires exponential storage, although each path can be constructed and examined quickly.

Depth-first search on the other hand opts to continue on the assumption that the previous choice of operator was correct. If this turns out to be false, it must backtrack later. Consequently, if the problem has more than one solution, it does not guarantee to find the one with minimal depth. For example, the previous sorting

problem has another solution

$$\{c,a,b\} \stackrel{swap_{1,3}}{\longrightarrow} \{b,a,c\} \stackrel{swap_{2,3}}{\longrightarrow} \{b,c,a\} \stackrel{swap_{1,3}}{\longrightarrow} \{a,c,b\} \stackrel{swap_{2,3}}{\longrightarrow} \{a,b,c\}$$

This drawback is the price paid for reduced storage requirements. Since at each node the algorithm need only maintain records of alternate path choices, it takes at most $b \times d'$ units of storage, where d' is the maximum path length allowed. Ideally, d' should be equal to d, the solution depth. However, that is not always possible in practice, because one generally does not know the solution's depth beforehand. The difficulty is how to choose d' so that it is not less than d—otherwise the solution cannot be found. But if d' is greater than d, much search effort may be wasted. This explains why depth-first search cannot guarantee an optimal solution, and is potentially less efficient than breadth-first search. Optimality and efficiency are sacrificed in order to reduce storage requirements.

Both styles of search have their strengths and weaknesses. Breadth-first search requires exponential storage $O(b^d)$ but can examine each state quickly without redundant search. In contrast, depth-first search requires polynomial storage $O(b \times d')$, where $d' \geq d$, but takes longer because of possible redundant searches. Breadth-first search may exhaust memory before a solution is found, while depth-first search may never find a solution if d' is too small. Breadth-first search always find the optimal solution, while depth-first search may not find it if d' is too large. Clearly, the best choice of search style depends on the application. However, depth-first is generally preferable because storage is usually limited while time is relatively abundant. On the other hand, if optimality is essential, breadth-first is a better choice.

1.1.3 Other search strategies

To lessen the impact of an inappropriate choice of d', one could allow the possibility of increasing it when necessary. That is, search proceeds in depth-first style up to the current maximum limit, then if no solution is found, that limit is increased and the search repeats from the beginning. The process consists of successive depth-

first searches with increasing maximum limits. In search terminology, this is called *depth-first iterative-deepening* search. The storage requirement is the same as for depth-first search. Optimality is guaranteed because paths are examined in increasing length. The price paid is computational inefficiency, because each of the b^k paths with length k is examined $d-k+1$ times, where d is the solution depth. However, this inefficiency is asymptotically insignificant because, in search problems, the average branching factor b always exceeds 1 and so $\sum_{k=1}^{d-1}(d-k+1)\cdot b^k$ is always less than $\frac{b^2}{(b-1)^2}\cdot b^d$. Complexity analysis shows that depth-first iterative-deepening search is asymptotically optimal in both search time and storage [Korf 88a, Stickel 85].

Another variant is the *best-first* strategy. It is similar to breadth-first search, except that choices are ranked according to a pre-defined evaluation function. The ranking is usually based in terms of *distance-to-goal* estimation. In best-first search, those having the highest rank (the shortest distance) are expanded first. The problem is how to define the evaluation function [Gaschnig 77]. A good evaluation function is one that generally assigns highest ranks to choices that lead to a solution. If that is the case, best-first will find the solution quickly. Otherwise, time may be wasted searching non-solution paths. Furthermore, it can be shown that if the evaluation function consistently under-estimates the distance-to-goal, best-first search will find the optimal solution [Korf 88a]. Consequently, one needs to analyze the application domain carefully in order to select a good evaluation function. For this reason, best-first search is problem-dependent, unlike the strategies discussed previously.

1.2 Search in function induction

The problem of inducing functions from examples is frequently encountered in inductive learning systems, ranging from automated discovery of natural laws [Langley 87] to the inference of functional relations in robot programs [Andreae 84a]. The idea is to translate the information embedded in some "extensional" description of a func-

tion into a specific "intensional" description that can be effectively manipulated and reasoned about. The extensional description is given by providing the result of the function in a number of cases, for example, $f(2,3) = \frac{3}{5}$, $f(-1,7) = \frac{7}{6}$, $f(11,4) = \frac{4}{15}$ and $f(2,-11) = \frac{11}{9}$. The intensional description is a structural characterization of the function as a composition of given primitive ones, for example, $f(x,y) = y/(x+y)$.

Function induction can be viewed as the problem of abstracting a common relationship between the given examples. This relationship, if exists, is the solution of the induction problem. Obviously, such problems can be solved by generating all possible relationships and testing each against the examples. For example, the intensional description of $f(x,y)$ can be found by enumerating all possible arithmetic expressions with at most two variables

$$x+x,\ x+y,\ y+y,\ x-y,\ y-x,\ x^2,\ xy,\ y^2,\ x/y,\ y/x,\ \ldots$$
$$x+(x+x),\ x+(x+y),\ x+(y+y),\ x+(x-y),\ x-(y+y),\ x^3,\ x^2y,\ \ldots$$

and testing each against the given data $f(2,3) = \frac{3}{5}$, $f(-1,7) = \frac{7}{6}$, $f(11,4) = \frac{4}{15}$ and $f(2,-11) = \frac{11}{9}$. In this case, $y/(x+y)$ is an expression satisfying these examples, and thus is a solution of the problem.

1.2.1 Major tasks in function induction

Inducing functions in general consists of three major tasks: determining relevant arguments, selecting an applicable function formula, and fitting it to the data by adjusting its coefficients. Of these, the third is the easiest. Once the function's form is known, calculating its coefficients is straightforward using available techniques for solving system of equations.

Determining relevant arguments is a more difficult problem. Arguments are relevant only if they contribute to the function's formation. Obviously, the presence of irrelevant arguments makes the search more difficult as more incorrect choices are possible. This task, of course, can be solved by combinatorial selection. The induction algorithm chooses some of the arguments and attempts induction with only those. Upon failure, it chooses another argument set and tries again. Eventually, the correct argument set is found. This is certainly not an efficient selection technique, but is the only one not requiring any problem-specific knowledge. Others use problem-specific information to make the right choice quickly. For example, if the arguments have types, those that violate type restrictions are not considered. Or if there are known constraints on the solution, those that violate them are eliminated.

The remaining task, selecting an applicable function formula, has so far received less attention than the other two, because it seems uninteresting and is inevitably a mechanical and tedious procedure. In principle, it is trivial, for function formulas can be enumerated quite easily. What makes it difficult is the fact that there are too many potential formulas and it is hard to apply problem-specific information to reduce the possibilities. Previous researchers have avoided the problem by assuming, either implicitly or explicitly, that the function's form is given. A popular alternative is to assume that the function is either polynomial or can be approximated by a polynomial. For example, in BACON.6 [Langley 83], the function's form must be given; in COPER [Kokar 86], it is assumed to be polynomial; in both BACON.4 [Langley 87] and ABACUS [Falkenhainer 86], it is a ratio of polynomials.

This thesis addresses the problem of speeding up the task. Once a function's form has been selected, techniques developed previously can be applied to produce the final solution. Consequently, in this thesis, the function induction problem is restricted to that of finding the function form. That means that all necessary arguments are assumed to be present in the extensional descriptions. This effectively rules out induction problems that require unknown arguments or coefficients. Such problems are normally tackled by algorithms designed to solve the task of fitting a function form to given data.

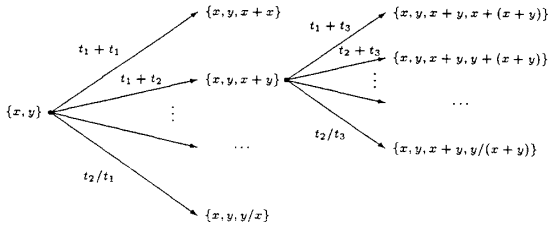


Figure 1.2: State transitions in function search

1.2.2 Selecting a function's form by search

Using the problem space model, the task of function form selection can be solved by exhaustive search as follows.

First, the task is represented by a search graph, whose nodes are labeled with a list of potential expressions, for example $\{x, y, x + y, x - y, xy\}$. The start node is labeled with a list of the function's arguments. The end node is labeled with a list containing an expression satisfying all given extensional descriptions. In this graph, the edges are labeled with any applicable operator, $+, -, \times, /$, *etc.*, whose operands are selected from the expression list representing the current node. With this labeling, the transition from one node to another has the property that

> The latter node's label is the former node's extended with one more expression formed by the connecting edge's label and its operands.

Then, the search proceeds from the start node, following the graph's edges, until it reaches any one of the end nodes. The solution is the sequence of operations that label the graph's edges. Figure 1.2 illustrates the first few transitions from the start state $\{x, y\}$, where t_i stands for the i^{th} item in the previous expression list.

1.3 Improving search in function induction

In order to maintain generality, induction algorithms should not rely on problem-dependent information. This condition makes search more attractive than other more specialized problem-solving methods. On the other hand, since function induction is a major component of learning systems, the ability to perform it quickly is an important asset. This tends to make search less attractive, because it is generally less efficient compared to other methods. Practical learning systems that employ function induction must strike a balance between generality and efficiency.

Previously, researchers have concentrated on methods of applying problem-specific knowledge effectively while maintaining as much generality as possible. This usually produces search algorithms that are general enough to handle many problems within some particular domain, but inapplicable to other domains without extensive modification. As a consequence of this emphasis on problem-specific knowledge, most research address the task of identifying relevant arguments, which is definitely domain-dependent, and the task of data fitting, which applies primarily to numeric functions. The task of selecting the function's form is either delegated to the user or trivialized by imposing a restricted choice of forms.

In contrast, the research described in this thesis addresses the problem head-on with a new search strategy called the *equal-value search*. The objective is to provide a problem-independent strategy that greatly increases the efficiency of selecting the function's form from examples. It rests on the idea of an equivalence transformation between different forms of a given function, but with a weaker definition of semantic equivalence than the usual one. Since no problem-specific knowledge is assumed, the strategy is general enough for non-numeric domains as well as numeric ones.

1.4 Thesis outline

Chapter 2 introduces the general function induction problem. It defines the applicable search spaces, and argues for the choice of the most problem-independent definition, the syntactic search space. It concludes with a brief overview of previous research in function induction and how other systems handle the problem of search space explosion.

Chapter 3 describes the equal-value search strategy, which is based on a non-standard equivalence relation, called the *equal-value relation*. It then discusses the strategy's characteristics and shows that the strategy is correct, complete and non-redundant. Combined with a high performance, this makes equal-value search an attractive strategy for general function induction. The chapter also explains how various programming techniques are utilized in the implementation of the search.

Chapter 4 presents empirical results to support the claim that the new search strategy is very efficient. There are seven sets of experiments, each addressing several related function induction problems. These experiments are designed to illustrate the use of the equal-value search and techniques to improve its performance. For comparison purposes, the last set of experiments solves several standard problems in function induction.

Chapter 5 concludes by discussing application problems that could benefit from the equal-value search and suggests extensions to solve related problems. These include inducing symbolic functions, finding equivalent forms of functions, optimizing compiler output, and classifying concepts using numeric descriptors.

Chapter 2

Background

In this chapter, the general function induction problem is formally introduced, along with its parameters. The problem is general in the sense that it can be applied in almost any domain, not just the realm of mathematical functions. The term "function" denotes any mapping from one set of values to another. The only constraint is that the mapping must be single-valued—that is, identical input arguments always map to identical output values. Furthermore, the formal problem statement does not make any particular assumption about the application domain or about the nature of the function to be induced. Instead, all domain-specific information are considered to be parameters of the function induction problem.

The search space is defined in Section 2.2 using two conventional definitions of function equivalence, semantic and syntactic. It is argued that syntactic equivalence, while simple to check, generates an astronomically large search space, which is practically impossible to search within reasonable resource limits. On the other hand, semantic equivalence is very difficult and time-consuming to check, although the resulting search space is much smaller.

Previous researchers resolved the dilemma by compromising between these two definitions of function equivalence. Either semantic equivalence is relaxed so that testing becomes easier, or syntactic equivalence is augmented with more constraints to reduce the search space size. The result is an approximation to the semantic space in the form of a reduced syntactic search space whose members satisfy various constraints. In Section 2.3, these constraints are discussed and grouped by their characteristics and effect on the expected solution of the induction problem. For illustration, Section 2.4 briefly reviews four function induction systems, NODDY [Andreae 84a], BACON [Langley 83], ABACUS [Falkenhainer 86] and COPER [Kokar 86], and discusses

Input examples			
x_1	1	2	-3
x_2	2	3	0
$f(x_1, x_2)$	5	13	9

Problem: Find $f(x_1, x_2)$ in terms of $+, -, \times, /$

Solution: $f(x_1, x_2) = x_1 \times x_1 + x_2 \times x_2$

Figure 2.1: Example of a function induction problem

their use of search constraints.

2.1 Inducing functions from examples

2.1.1 Problem statement

The function induction problem is to induce functional relationships from examples of input-output mappings, or in other words to find a function satisfying a given set of input-output examples. Although this simple description captures the essence of the problem, it must be augmented to prevent induction from becoming trivial. For example, consider the problem of finding a function f satisfying the examples given in Figure 2.1. Obviously, a trivial solution is the extensional definition $f(x_1, x_2)=g((x_1, x_2))$, where (x_1, x_2) represents a member of the Cartesian product of the argument sets, and g is a partial mapping defined as $g : (1, 2) \mapsto 5$, $(2, 3) \mapsto 13$ and $(-3, 0) \mapsto 9$. Although this solution may be acceptable when f cannot be described any other way,[1] it is usually not considered to be a satisfactory solution of a function induction problem. Instead a structural or intensional characterization of the function is sought by imposing the extra condition that it must be expressed in terms of some given primitive functions, in this example $+, -, \times$ and $/$.

[1] In machine learning, this is categorized as *rote learning* [Michalski 83].

Thus the general problem of function induction can be stated as:

Given: A list of examples of the form $f(x_1, x_2, \ldots, x_k) = y$

A list of primitive functions $\{f_1, f_2, \ldots, f_m\}$

Find: The function f expressed as a composition of the f_i

For convenience, the list of examples is denoted by $\mathcal{F}$. Thus the problem quoted earlier has $\mathcal{F} = \{f(1,2)=5, f(2,3)=13, f(-3,0)=9\}$.

The primitive function list defines a search space comprising all functions constructible from these primitives. If the expected function resides in this space, it can be found by systematically constructing and checking each candidate against all given examples (the *generate-and-test* approach).

As an example, solving the induction problem in Figure 2.1 using any suitable exhaustive search returns the function $f(x_1, x_2)=x_1^2 + x_2^2$ as a potential solution. Of course, there are infinitely many other functions satisfying those examples—for example, polynomials with degree 3 or more passing through the points (1,2,5), (2,3,13), (-3,0,9) in 3-dimensional space—but, under a natural definition of "complexity", they are more complex than $x_1^2 + x_2^2$, a polynomial of degree 2. By convention, the "simplest" functions that satisfy all examples are taken to be the preferred solution set of the given induction problem.

As another illustration, suppose a manufacturer wants to predict the maximum stopping distances for cars equipped with various braking systems. For this purpose, four tests are conducted using four different braking systems installed on two different cars running at four different speeds. The results of these (hypothetical) tests are recorded in Figure 2.2, where the input arguments are values reported by various sensors and the corresponding output values are distances measured after each test. Using different lists of primitive functions, the search finds different (but functionally equivalent) forms of the stopping-distance function.

This illustrates two important characteristics of function induction problems.

Input examples				
Initial speed (v)	10	20	30	40
Mass (m)	1000	1200	1200	1000
Braking force (f)	1600	2000	2800	3200
Braking duration (t)	6.25	12.00	12.86	12.50
Stopping distance (d)	31.25	120.00	192.86	250.00

Problem: (a) Find distance d in terms of $+, -, \times, /$
(b) Find distance d in terms of $+, \times$, *negate*, *reciprocal*

Solution: (a) $d = v \times v \times m/(f + f)$
(b) $d = v \times v \times m \times (f + f)^{-1}$

Figure 2.2: Another example: Stopping distance for cars

First, they can be solved by exhaustive search, and this is particularly appropriate when there is not enough information available to support specialized problem-solving methods. Second, the search space—and thus the solution—is influenced by the information given in the statement of the problem. Different primitive functions generate different search spaces and thus give different solutions.

2.1.2 Parameters of the function induction problem

The first parameter is the list of *primitive functions*. Although both the above examples have been formulated in terms of $+, -, \times$ and $/$, this is by no means the only choice. In the first example, the two operators $+$ and $\times$ would suffice. This speeds up the search considerably, but at the cost of reduced generality, for it is not possible to express all arithmetic operators in terms of these two. However, arithmetic completeness is regained by including the operators *negate* ($-x$) and *reciprocal* ($1/x$). This changes the search space, of course, and thus may change the form of the solution as in Example 2.2b. In general, if the function to be induced requires a particular

component which is not derivable from the given primitives, then it will be impossible to induce regardless of the complexity limit.[2] Therefore, this thesis assumes that the given list contains all primitive functions needed.

The next parameter is the definition of *function complexity*. Although not explicitly stated, it is an important component of the problem. Specifically, it defines a partial ordering on the function space, which is invariably unbounded since an infinite number of functions can be composed just from one unary function ($g(x)$, $g(g(x))$, $g(g(g(x)))$, ...). A search algorithm starts from the least complex part of the space,[3] and searches until it either finds a solution or reaches a pre-determined complexity limit. In the latter case, the given induction problem is too complicated to be solved by unaided search. As computing resources (and the user's patience) are limited, this thesis assumes that the induction problem is simple enough that the function can be found by exhaustive search algorithms.

The third parameter is the *arity* of each primitive function. This determines the branching factor of the search space. In Figure 2.2b, replacing $-$ and $/$ by *negate* and *reciprocal* results in much faster induction. This improvement in search time can be attributed to the fact that the average branching factor of the search graph in example (b) is less than in (a).

The final parameter is the *number of variables* in each example. Obviously, the function space grows larger when more variables are included, for there are more choices for each function's arguments. Ideally, the given examples should not contain redundant variables, but in practice it is usually impossible to identify redundancy without prior domain knowledge. As an example, inducing the stopping-distance function (Figure 2.2) with only three variables v, m and f is much faster because t is redundant. But inducing the same function using any two of these is impossible, for v, m, and f are independent variables, all of which are needed to compute the distance d. Consequently, in this thesis, the search space size is measured using the

[2]For instance, it is impossible to induce $f(x, y) = x/y$ given the primitive function list $\{+, \times\}$.
[3]In the problem space model, that is the node labeled with the list of function's arguments.

number of variables given, regardless of any redundancy.

2.2 Defining the search space

With the four parameters identified, the search space can now be defined using either one of the following two function equivalence relations, *syntactic equivalence* and *semantic equivalence*. Under the former, two functions are equivalent if and only if they have identical expressed forms. Thus the functions $g(x_1, h(x_2, x_3))$ and $h(g(x_1, x_2), g(x_1, x_3))$ are considered to be distinct, and will be counted twice in the function space. Under the latter, functions are equivalent if and only if they have equal values under all possible parameter instantiations. So the above functions are semantically equivalent if g distributes over h, as in the case of $\times$ and $+$.[4] If so, they are counted as one in the function space, and either one can represent the other.

The function space defined by syntactic equivalence is called the *syntactic space*, while that defined by semantic equivalence is called the *semantic space*. The term *search space* is used to refer to the one currently being considered, or to either one when the distinction is not relevant. To distinguish members of the syntactic space from those of the semantic space, the terms *expressions* and *functions* are used respectively. In addition, different forms of a function are referred to as *expressed forms* or in some cases simply *expressions*. This should not cause any confusion as it reflects the usual notion that a (semantic) function may be represented by many different (syntactic) expressions.

Either equivalence relation divides the search space into partitions, each of which contains only equivalent expressions. In the syntactic space, there is exactly one expression per partition, while in the semantic space, there may be one or more syntactically different but semantically equivalent expressions. Thus, as Figure 2.3 illustrates, each partition in the semantic space is a set of one or more partitions in the syntactic space. In other words, the size of the semantic space is bounded above

[4] That is. $\times(x_1, +(x_2, x_3)) = +(\times(x_1, x_2), \times(x_1, x_3))$

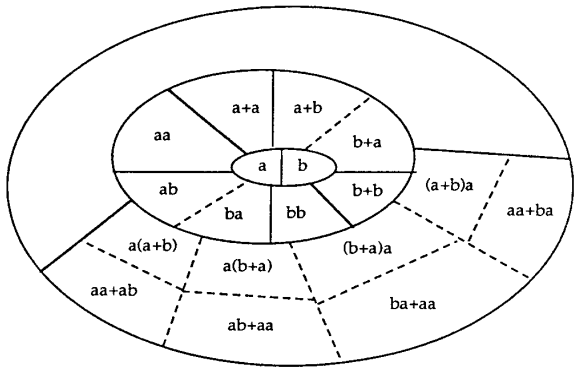


Figure 2.3: Partitioning the search space

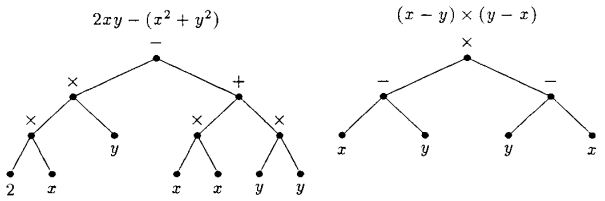


Figure 2.4: Examples of evaluation trees for expressions

by that of the syntactic space.

2.2.1 Syntactic search space

The syntactic search space is the set of all possible syntactic combinations, called *expressions*, of the given primitive functions. Two expressions are syntactically equivalent if they are identical in form. Expressions are represented by their *evaluation trees*, with each internal node representing the corresponding subexpression. The *complexity* of an expression is the depth of its evaluation tree, and its *arguments* are the leaves of this tree. Figure 2.4 illustrates the evaluation trees for $2xy - (x^2 + y^2)$ and $(x - y) \times (y - x)$ over the primitives $+, -, \times, /$. They have complexities 3 and 2 respectively, even though they represent the same function.

The size of the syntactic search space is estimated in terms of the number of primitive functions, and each function's arity. Specifically, let C_n be the accumulated number of expressions with complexity n or less, F be the number of primitive functions, and suppose each primitive f_i has arity A_i. The search space size C_n is determined as follows.

Consider any expression with complexity n and top-level function f_i. It has A_i subexpressions, each of which has complexity $n-1$ or less. Since there are F choices of primitive function, and C_{n-1} choices of expression with complexity $n-1$ or less,

the number of expressions with complexity n is at most $\sum_{i=1}^{F} C_{n-1}^{A_i}$.

However, as the value for C_{n-1} also includes expressions with complexity $n-2$ or less, not all choices will produce expressions with complexity n. Specifically, there are exactly $C_{n-2}^{A_i}$ choices that will produce expressions with complexity $n-1$ or less. Correcting for this, the exact number of possibilities for the argument set is $C_{n-1}^{A_i} - C_{n-2}^{A_i}$, and consequently the number of new expressions is $\sum_{i=1}^{F}(C_{n-1}^{A_i} - C_{n-2}^{A_i})$. Therefore, the size of the search space of expressions with complexity n or less is

$$\begin{aligned} C_n &= C_{n-1} + \sum_{i=1}^{F} C_{n-1}^{A_i} - \sum_{i=1}^{F} C_{n-2}^{A_i} \\ &= C_{n-2} + \sum_{i=1}^{F} C_{n-2}^{A_i} - \sum_{i=1}^{F} C_{n-3}^{A_i} + \sum_{i=1}^{F} C_{n-1}^{A_i} - \sum_{i=1}^{F} C_{n-2}^{A_i} \\ &= C_{n-2} + \sum_{i=1}^{F} C_{n-1}^{A_i} - \sum_{i=1}^{F} C_{n-3}^{A_i} \\ &\vdots \\ &= C_0 + \sum_{i=1}^{F} C_{n-1}^{A_i} \end{aligned}$$

for $n > 0$. Letting $A = \max(A_1, A_2, \cdots, A_F)$, this can be simplified to

$$C_n \leq C_0 + F \cdot C_{n-1}^{A}, \qquad \text{for } n > 0$$

with equality occurring when all primitive functions have the same arity A.

By definition, C_0 is the number of expressions with complexity 0. This is the number of variables (and constants) that may be included in expressions. Not surprisingly, the number of expressions grows exponentially with this quantity. For example, when all primitive functions are binary, the number of expressions with complexity n or less is $C_n = C_0 + FC_{n-1}^2$, which is shown in Table 2.1.

For simplicity, assume that all primitive functions have the same arity A. If $A = 1$, the relation becomes $C_n = C_0 + FC_{n-1}$, which has the solution $C_n = C_0 \sum_{i=0}^{n} F^i$. Otherwise, if $A > 1$, then

$$C_n = C_0 + FC_{n-1}^{A}$$

Number of primitive functions

Level $n = 1$		F=1	F=2	F=4	F=8	F=16
Number of variables and constants	C_0=1	2	3	5	9	17
	=2	6	10	18	34	66
	=4	20	36	68	132	260
	=8	72	136	264	520	1032
	=16	272	528	1040	2064	4112

Level $n = 2$		F=1	F=2	F=4	F=8	F=16
Number of variables and constants	C_0=1	5	1.90×10^{1}	1.01×10^{2}	6.49×10^{2}	4.62×10^{3}
	=2	38	2.02×10^{2}	1.30×10^{3}	9.25×10^{3}	6.97×10^{4}
	=4	404	2.60×10^{3}	1.85×10^{4}	1.39×10^{5}	1.08×10^{6}
	=8	5192	3.70×10^{4}	2.79×10^{5}	2.16×10^{6}	1.70×10^{7}
	=16	74000	5.58×10^{5}	4.33×10^{6}	3.41×10^{7}	2.71×10^{8}

Level $n = 3$		F=1	F=2	F=4	F=8	F=16
Number of variables and constants	C_0=1	2.60×10^{1}	7.23×10^{2}	4.08×10^{4}	3.37×10^{6}	3.42×10^{8}
	=2	1.45×10^{3}	8.16×10^{4}	6.74×10^{6}	6.85×10^{8}	7.77×10^{10}
	=4	1.63×10^{5}	1.35×10^{7}	1.37×10^{9}	1.55×10^{11}	1.87×10^{13}
	=8	2.70×10^{7}	2.74×10^{9}	3.11×10^{11}	3.74×10^{13}	4.65×10^{15}
	=16	5.48×10^{9}	6.22×10^{11}	7.49×10^{13}	9.29×10^{15}	1.17×10^{18}

Table 2.1: Number of expressions composed from F binary functions

$$
\begin{aligned}
&= C_{n-1}^{A}\left(F+\frac{C_0}{C_{n-1}^{A}}\right)\\
&= C_{n-2}^{A^2}\left(F+\frac{C_0}{C_{n-2}^{A}}\right)^{A}\left(F+\frac{C_0}{C_{n-1}^{A}}\right)\\
&\vdots\\
&= C_0^{A^n}\prod_{i=1}^{n}\left(F+\frac{C_0}{C_{i-1}^{A}}\right)^{A^{n-i}}\\
&= C_0^{A^n}F^{1+A+\cdots+A^{n-1}}\prod_{i=1}^{n}\left(1+\frac{C_0}{FC_{i-1}^{A}}\right)^{A^{n-i}}
\end{aligned}
$$

Since $\frac{C_0}{FC_{i-1}^{A}} > 0$ for all $1 \le i \le n$, a lower bound for C_n is simply

$$
\begin{aligned}
C_n &> C_0^{A^n}F^{1+A+\cdots+A^{n-1}}\prod_{i=1}^{n}(1+0)^{A^{n-i}}\\
&= C_0^{A^n}F^{1+A+\cdots+A^{n-1}}
\end{aligned}
$$

From this, it follows that $C_{n-1} \ge C_0^{A^{n-1}}F^{1+A+\cdots+A^{n-2}}$, with equality occurring when $n = 1$ and $F = 1$. Multiplying both sides by F gives $FC_{n-1}^{A} \ge C_0^{A^n}F^{1+A+\cdots+A^{n-1}}$. Substituting this in the formula for C_n yields

$$
C_n \le C_0^{A^n}F^{1+A+\cdots+A^{n-1}}\prod_{i=1}^{n}\left(1+\frac{C_0}{C_0^{A^i}F^{1+A+\cdots+A^{i-1}}}\right)^{A^{n-i}}
$$

Simplifying right hand side gives the upper bound for C_n as

$$
C_n \le C_0^{A^n}F^{1+A+\cdots+A^{n-1}}\prod_{i=1}^{n}\left(1+(C_0F^{1/(A-1)})^{1-A^i}\right)^{A^{n-i}}
$$

Note that the derived bounds are for any values of $A > 1$,[5] C_0, and F. In the case of $FC_0 > 1$, the term $(C_0F^{1/(A-1)})^{1-A^i}$ decreases rapidly to 0 as i increases towards n. This means that the product $\prod_{i=1}^{n}\left(1+(C_0F^{1/(A-1)})^{1-A^i}\right)^{A^{n-i}}$ can be accurately approximated using the first few terms. When $FC_0 = 1$, the actual value of C_n can be calculated directly from the relation $C_n = 1 + C_{n-1}^{A}$, which is the number of A-ary trees with height n or less. For example, when $A = 2$, $C_n = 1 + C_{n-1}^{2}$ is shown

[5] When $A = 1$, C_n can be computed directly as shown before.

in [Purdom 85] to have the value of K^{2^n}, where K is approximately 1.5028368015. When $A = 3$, it has the value of K'^{3^n}, where $K' \approx 1.2765828181$.

Although this seems to make exhaustive search practically impossible for non-trivial functions, in practice the primitives' arities seldom exceed 2, and the complexity is fixed at some very small number, say 3 or 4. Therefore the search space size is effectively polynomial with respect to the number of primitive functions F and the number of argument choices C_0. However, the discussion in this thesis is theoretically motivated, and does not assume limited complexity or restrictions on arity.

2.2.2 Semantic search space

The semantic search space is the set of all semantically different functions that can be composed from the given primitives. Two functions are semantically different if there exists a set of input arguments for which they have different values. Since one function may be expressible in many different ways, its complexity is defined to be the complexity of the simplest expressed form, and its arguments are the arguments of this form. Using the previous example, the function $2xy - x^2 - y^2$ has complexity 2 with respect to the primitive list $\{+, -, \times, /\}$, since its simplest expressed form $(x - y) \times (y - x)$ has complexity 2.

It is very difficult to estimate the size of the semantic search space, because semantic equivalence is not syntactically determined. For example, given an arbitrary function $g(x_1, h(x_2, x_3))$, it is impossible to determine whether $h(g(x_1, x_2), g(x_1, x_3))$ is one of its expressed forms without some information about g and h—in particular, whether g distributes over h. Consequently, the semantic search space cannot be characterized unless the primitive functions are predefined and their properties known. This makes the size of the search space problem-dependent, as the primitive functions belong to the induction problem, not to the search algorithm.

Even when such information is given, it is still difficult to determine semantic

equivalence. The standard approach is to translate expressed forms into some canonical representation so that they can be compared lexically. This requires procedures to manipulate functions syntactically without changing their meaning. For illustration, consider the case of functions composed from + and ×. Suppose that complex functions are generated using previously generated functions as arguments. Every time a new function is generated, the search algorithm must check if any of its equivalent forms has been generated before. Using some canonical form like sum-of-products or product-of-sums, such checking is trivial. However, there is no guarantee that newly generated functions will appear in canonical form,[6] and so they must be explicitly converted for later comparison.

In general, transforming an arbitrary function from a given expressed form to a canonical form requires knowledge of the algebraic properties of the primitive functions. In the case of + and ×, these include the commutativity and associativity of both functions, and the distributivity of × over +. Furthermore, the transformation between different expressed forms of the same function, while not difficult conceptually, still requires additional processing time. Even when the construction is organized so that new functions are automatically expressed in canonical form, which is possible in the case of + and ×, comparison time is still proportional to the number of symbols in the expressed forms.

To make matters worse, when trigonometric functions such as *sin* and *cos* are included, the equivalence of two syntactically different forms, such as $\sin(a + b)$ (complexity 2) and $\sin a \cos b + \cos a \sin b$ (complexity 3) is not immediately obvious from the definitions of *sin* and *cos*. Transforming these functions requires knowledge of trigonometry properties. Another example is the inclusion of operations like differentiation or definite integration in the function space, so that expressions such as $f'(x) + \int_a^x f(t)dt$ can be generated. It is impossible to canonize such functions without explicit transformation using the rules of symbolic differentiation and integration.

[6] For example, both $x - y$ and $-x + y$ are in sum-of-products form but the new function $(x - y) \times (-x + y)$ is not.

To summarize, it is impractical to search the semantic space unless the problem is specific enough to incorporate function properties directly in the generate phase of a generate-and-test search algorithm. When the primitive function list contains functions other than the usual arithmetic operators, it becomes increasingly difficult to ensure that expressions generated are semantically distinct. To avoid this problem, the conventional approach has been to define the syntactic search space with as little semantic duplication as possible, and then to search for the required function syntactically. Using a few known mathematical properties to eliminate most of the equivalent expressions, the remaining ones are treated as though they represent different functions. The resulting search space is greatly reduced even though it is still a large superset of the semantic search space.

2.2.3 Different orderings of the search space

Throughout this thesis, the complexity of a function is taken to be the nesting depth of function calls. This is by no means the only possible definition. Complexity could be measured by the number of primitive functions needed, or the polynomial degree, or the number of product terms. These definitions result in different orderings of the search space. For example, the function $(x_1^2 + x_2^2) + (x_3 + x_4)$, which has complexity 3 in terms of nesting depth, has complexity 5 by function count, complexity 2 by maximum polynomial degree, and complexity 4 by number of product terms. But regardless of how complexity is defined, the size of the search space remains the same. Thus any definition is acceptable as long as it is possible to search for functions in order of increasing complexity. This will ensure that the solution is the simplest function satisfying all given examples.

2.3 Constraining the search space

Because of the difficulty in determining the exact semantic space, function induction systems often opt for an approximated search space, using various constraints

to restrict the formation of undesirable or redundant functions. In these systems, constraints are expressed as conditions on the syntax or semantics of the function to be induced. The intention is to eliminate as many expressions as possible that are semantically equivalent or irrelevant in the current problem domain.

Such constraints are called *unconditionally reliable* if all functions in the semantic space lie in the constrained space. They are called *conditionally reliable* if all problem-relevant[7] functions lie in the constrained space. If, on the other hand, the constrained space omits some members of the semantic space, the constraints are called *unreliable*. The standard practice is first to define the problem domain; then to determine applicable constraints which are at least conditionally reliable and embed them in the search algorithm; finally, if the search space is still unacceptably large, to impose further, unreliable, constraints.

Although search constraints vary in form, they can be grouped into four general classes: problem-specific, semantic, syntactic and heuristic.

2.3.1 Problem-specific constraints

The most useful problem-specific constraints are *type compatibility* conditions on function arguments. In strongly-typed domains, these eliminate many ill-formed expressions. For example, placing a type compatibility constraint on the function + prevents expressions such as $12kg + 2in^2$ (kilogram and square-inch) from being considered at subsequent complexity levels. Since type checking is relatively simple, the overhead of ensuring compatibility is negligible compared to the total search time. For this reason, almost all function induction systems incorporate these constraints directly into their search algorithms.

An additional benefit of type compatibility conditions is that constraint propagation techniques can be used to eliminate redundant variables in an induction problem. As variables are typed, an initial check can determine if any is not needed in deriv-

[7] Functions are said to be *problem-relevant* if they are meaningful within the context of the given problem domain.

ing the appropriate function type. With a smaller number of choices of argument, the search space is reduced significantly (Section 2.2.1). An example of the effective use of type compatibility constraints is the COPER system [Kokar 86], in which an argument's type is its physical dimensionality.[8]

Other examples of problem-specific constraints include *limiting argument values,* such as "an object's mass must be positive ($m>0$)", *checking physical constraints,* such as "no speed can exceed the speed of light ($v \leq 3\times10^{10}$ cm/s)", and *restricting functions' applicability* on arguments, such as "do not take the square root of time".

Problem-specific constraints are always conditionally reliable, as they only eliminate functions that are meaningless in the given domain. They are not unconditionally reliable because many eliminated functions are mathematically valid although inapplicable to the current problem.

2.3.2 Semantic constraints

Semantic constraints are similar to but more general than problem-specific ones. There are two types: *semantically meaningful* constraints, which ensure that only mathematical meaningful expressions are composed, and *semantically nonredundant* constraints, which prohibit the composition of redundant expressions. The former resemble problem-specific constraints, and so are easy to specify; while the latter are quite different and generally more difficult to specify.

Examples of semantically meaningful constraints are the no-division-by-zero rule for /, the no-identical-arguments rule for / and −, and the positive-argument-only condition for *ln*. Examples of semantically nonredundant constraints are rules of equivalent compositions, such as the commutative property for both + and × and the distributive property of × over +. Another is the constraint that arguments of / cannot be expressions having / as their top-level function.[9] Another is the constraint

[8]According to [Kokar 86], COPER can do more than just induce functions. Using type compatibility constraints and the notion of ***syntactic linear independence***, it is also able to detect the given arguments' relevance, infer a missing argument and deduce its dimensionality.

[9]That is because $\frac{a}{b/c} = \frac{ac}{b}$, $\frac{a/b}{c} = \frac{a}{bc}$, and $\frac{a/b}{c/d} = \frac{ad}{bc}$.

that expressions must be generated in a predefined canonical form, such as sum-of-products, so that duplicates can be detected easily by lexical comparison. Further semantically nonredundant constraints are those that eliminate irrelevant expressions such as $a-(a+b)$ or a^2/a.

Both types of semantic constraint assume that the search algorithm has prior knowledge of the primitive functions and their properties. In other words, they are domain dependent, although to a lesser degree than problem-specific constraints. If all possible semantic constraints were specified, the resulting search space would cover the semantic space exactly. However, this is not generally the case, and in practice the constrained space is normally a large superset of the semantic space.

Since semantic constraints are used to avoid mathematically equivalent forms of functions, the resulting search space includes all possible functions in the semantic space. Consequently these constraints are unconditionally reliable, as all functions appear in the constrained search space.

2.3.3 Syntactic constraints

Analogous to the semantic constraints are the syntactic constraints, which restrict the expressed forms of function instead of their meaning. Being syntactic, they are easily compiled into the generate phase of a generate-and-test search algorithm. They can vastly reduce the search space but unfortunately result in a biased search. Their effect is to transform the induction problem for arbitrary functions into one for a restricted class of functions.

For example, the BACON.6 induction system [Langley 83] insists that the syntactic form of the function be given in advance. It uses this information to infer unknown coefficients, which may or may not be constants. However, since constant coefficients can be determined by solving systems of equations, BACON.6 concentrates on finding non-constant coefficients. Different forms are proposed for these coefficients[10] and

[10]Coefficients' forms are restricted to $y = ax+b$, where x, y are any two variables, and a, b are any unknown coefficients. This recursive restriction implicitly restricts the coefficients to polynomials.

tested against the examples supplied. The final solution is the set of coefficients that best fits the examples. Combining these coefficients with the pre-specified function form yields a specific formula for the function.

Although syntactic constraints are easily checked and offer a potentially large reduction in the search space, they may not be reliable. Well-chosen constraints can accelerate search time significantly, for the space is now much smaller. But if they are poorly chosen, the expected function may never be found, for all its expressed forms may have been eliminated. This contrasts with problem-specific and semantic constraints, which both guarantee to find the solution so long as it appears in the function space. Thus syntactic constraints should only be used when other constraints fail to reduce the search space to a reasonable size. Properly used, they do offer a chance of solving problems that are otherwise intractable.

2.3.4 Heuristic constraints

Heuristic constraints are ones that selectively discard parts of the function space that are deemed uninteresting or known not to contain the solution. Whereas the search space resulting from satisfying syntactic constraints is predefined and independent of the given set of examples, the space resulting from satisfying heuristic constraints usually depends on the examples. This is because heuristic constraints are not evaluated until after the search space has been (partially) constructed. Since different example sets may result in different organizations of the search space, it is not known *a priori* which parts of the search space will be discarded.

Heuristic constraints are usually expressed in terms of an evaluation function, which is used to assess the likelihood that a particular part of the function space contains the solution. Since they are not infallible—or they would not be called heuristic—there is a chance that the simplest expressed form of the solution lies in one of the discarded parts. In this case, the search algorithm must find another more complex expressed form, which invariably takes longer. In the extreme case, all

expressed forms may lie in the discarded parts. In other words, heuristic constraints may be misleading or even incorrect. For this reason, they are unreliable.

Nevertheless, heuristic constraints are useful in problems that cannot be searched exhaustively within practical limits. While their validity cannot be proved rigorously, good heuristics based on insight and experience can solve problems that are otherwise unsolvable within given resource limits. Examples of heuristic constraints include *preferred selection* [Michalski 83], in which search choices are ranked according to some user-defined preference criteria and only those exceeding a given threshold are considered; *search-tree pruning* [Hart 68], in which once the search reaches a certain depth a static evaluation function is used to avoid further search, and *user-directed search*, in which the user indicates which choice is most likely to succeed when the induction algorithm cannot make the decision itself.

A different form of heuristic constraint is a limit on computing resources. The implied heuristic is that the expected function is simple enough to be induced within some predefined resource bounds. When the limit is reached, the current search path is abandoned and another is chosen. Appropriate limits include *function complexity*, *search time*, and *maximum storage* limits. All of these implicitly restrict the search space to a small subset of the original space, ensuring a faster search but allowing the possibility of failure by biasing the search space towards simple functions.

2.4 Example systems for function induction

2.4.1 Noddy: Bidirectional search

NODDY is a system for learning robot procedures from traces of example executions [Andreae 84a, 84b]. Given several example executions of a particular task, it tries to construct a procedure to accomplish that task. The procedure is a sequence of actions, each represented by a given primitive function. NODDY's learning algorithm comprises three stages: skeleton matching, propagation and event generalization, and function

induction. The first two attempt to generalize the current hypothesis to satisfy new examples while the last tries to express the hypothesis in terms of executable procedures. When a procedure has been found, NODDY performs a consistency check to ensure that it is deterministic. The final solution is a step-by-step procedure to accomplish the given task.

In the function induction stage, NODDY assumes that the primitive functions f_i also include the corresponding inverses $f_j = f_i^{-1}$, so that it can perform the search in two directions—from the input towards the output and from the output towards the input. For example, suppose the function to be induced is

NODDY searches for the sequence of primitive functions

Using the given functions' inverses, NODDY then re-expresses the latter sequence to match the former sequence.

However as inverses of n-ary functions are usually non-deterministic, NODDY imposes the constraint that all n-ary functions must have $n-1$ constants so that their inverses are readily determinable. For example, + does not have a deterministic inverse when both arguments are variable, but if either is a constant then its inverse is trivial. In other words, NODDY uses syntactic constraints to reduce the search space, in addition to the usual problem-specific and semantic constraints.

2.4.2 Bacon: Trend analysis

BACON is a series of discovery systems, of which the most notable are BACON.4 and BACON.6 [Langley 83]. The systems attempt to find physical and chemical laws by analyzing the relationship between variables in the given examples. The search algorithm in BACON forms new expressions, called *theoretical terms*, by comparing

previously constructed expressions. The solution is the theoretical term having the same value in all examples.

In both BACON.4 and BACON.6, the search is performed recursively on related theoretical terms. Two terms x, y are related if they are monotonically increasing or decreasing. Depending on their relationships, either $x \cdot y$ or x/y is proposed, or if possible $y = ax + b$ where a, b are constants. For example, the ideal gas law (PV/NT=8.32) is discovered by first proposing the terms y_1=PV, y_2=y_1/T, and then finally y_3=y_2/N−8.32, which is identically zero.

BACON.4 limits the formation of new theoretical terms to products, ratios and linear equations. This effectively restricts the search space to polynomials and ratio of polynomials. BACON.6 relaxes this restriction by allowing the user to specify the expected function's form. With the given form, BACON.6 tries to solve for the coefficients using the same search technique as BACON.4.

Thus like NODDY's function induction, the BACON systems use syntactic constraints to restrict the search space.

2.4.3 Abacus: Functional clustering

ABACUS is another discovery system using function induction as its main problem-solving method [Falkenhainer 86]. It discovers functional relations by searching the space of possible terms which relate the user-supplied variables. These terms are similar to the theoretical terms in BACON. They are formed by noticing proportional relations between previously proposed terms. However, ABACUS allows the user to specify the forms of these new terms—in addition to the conventional product and ratio forms.

The main difference between BACON and ABACUS is that the latter does not assume that the given examples represent just one functional relationship. This allows it to induce several functions characterizing subsets of the given example set. To avoid spurious solutions, such as one function for each example, ABACUS adopts the notion

of nominal subsets, which are identified by means other than function induction. In [Falkenhainer 86], it calls this the *qualitative pre-condition* of the function to be induced.

2.4.4 Coper: Dimensional analysis

COPER represents a different approach to function induction. Instead of directly searching for functions satisfying the given examples, it first determines if relevant arguments are missing or if any irrelevant arguments are presented. This is done by the use of *dimensional analysis* [Kokar 86]. Briefly, each variable is assumed to have a physical dimension, which is a product of primitive descriptors. As dimensions can be manipulated algebraically, COPER selects only those that can be combined to form the dimension of the expected function. Irrelevant variables are ones that do not contain any of the selected dimensions. If COPER fails to determine the necessary dimensions, it concludes that some relevant variables are missing, and then suggests their expected dimensions.

The main advantage of COPER over both BACON and ABACUS is its ability to pre-process the variables before embarking on an exhaustive search. This allows it to eliminate most irrelevant variables and to check if all necessary variables are present. However, as this ability is a consequence of dimensional analysis techniques, COPER requires all variables to have dimensions, and it must know how to manipulate them effectively. To avoid the search space explosion, COPER assumes that all continuous functions can be approximated by polynomials (the Weiertrass theorem [Rudin 76]). In other words, it uses both problem-specific constraints and syntactic constraints to reduce the search space.

Chapter 3

The Equal-Value Search

As discussed previously, the syntactic space is very large but easy to search, while the semantic space is much smaller but more difficult to search. In practice, function induction systems compromise by searching a modified syntactic space from which easily-determined semantic equivalences have been eliminated. In cases where the modified space is actually smaller than the semantic space, efficiency is gained by restricting the application domain.

In this chapter, a new search strategy called the "equal-value search" is presented. It allows the search space to be examined efficiently without the need for prior domain knowledge. In fact, although it is unconditionally reliable and therefore runs no risk of missing solutions, it generates a search space which is often smaller than the semantic one. Since the cost of generating expressions is the same as when generating syntactically distinct expressions, this results in a general but efficient search strategy.

The new method hinges on a weakened definition of equivalence. Section 3.1 shows that, with the help of this new definition, the search space approaches the semantic space asymptotically as the number of examples increases. Next, Section 3.2 describes the equal-value search and proves its correctness, completeness and non-redundancy. Since the completeness proof shows that arbitrary functions can be induced using the new search space, it follows that the equal-value search is unconditionally reliable. Finally, Section 3.3 discusses features of an implementation of the method.

3.1 Partitioning the search space

Both syntactic and semantic spaces emerge from partitioning the search space by equivalence relations—syntactic and semantic equivalence respectively. The former

relation is stronger than the latter, because identical expressions always produce the same value for a given argument set, whereas the same function may be expressed in different syntactic forms. In other words, the semantic partitioning is coarser than the syntactic one.

Using the same idea, this section introduces a new relation that gives an even coarser partitioning than the semantic equivalence relation. The new relation has three important characteristics. First, it defines a new search space that approximates the semantic one. Second, there is a correcting procedure to obtain the exact semantic space when desired. Third, it is computationally easy to check. With this new relation, the induction algorithm searches the approximate semantic space, obtains a potentially correct solution, and then adjusts it if necessary to satisfy all examples given in the induction problem.

3.1.1 Approximating the semantic equivalence relation

Clearly, two semantically equivalent expressions have the same output value for every possible set of input arguments. In particular, they produce the same value for any sample set of arguments. This suggests a weaker definition of equivalence, called the *equal-value* relation:

> Two expressions are *equal-valued with respect to a set $\mathcal{F}$ of input-output examples* if they have the same values for those examples.

For example, the expressions $x_1^2+x_2^2$ and $x_1^2+x_1x_2+x_2$ are equal-valued with respect to the set $\mathcal{F} = \{f(1,2)=5,\ f(2,3)=13,\ f(-3,0)=9\}$. The inputs in this case are (1,2), (2,3) and (-3,0), while the respective outputs are 5, 13 and 9. In this example, the value of the expressions is not only the same for each input pair, but also equals the output as specified in $\mathcal{F}$. However, this need not be the case. For example, $(x_1-x_2)^2$ and $(x_1x_2-x_1-x_2)^2$ are also equal-valued with respect to $\mathcal{F}$, since both produce the outputs 1, 1, 9 respectively.

The equal-value relation can be viewed as a coarser version of semantic equivalence in the same way that the latter is a coarser version of syntactic equivalence. Equal-valued expressions (with respect to $\mathcal{F}$) are not necessarily semantically equivalent, but semantically equivalent expressions are always equal-valued. Consequently, the resulting search space, called the *equal-value space*, may be smaller than the semantic space, and thus can be searched faster.

However, as more input-output examples are included in the set $\mathcal{F}$, the chance of two semantically different expressions having equal values diminishes. Moreover, given two expressions that are equal-valued with respect to $\mathcal{F}$ but not semantically equivalent, one can always find a larger example set $\mathcal{F}' \supset \mathcal{F}$ that distinguishes them. Thus the equal-value space will approach the semantic space as $\mathcal{F}$ grows to include more examples of the expected function. So a possible correcting procedure is to enlarge the set $\mathcal{F}$ with randomly-generated examples.

Using the previous example, $\mathcal{F}=\{f(1,2)=5,\ f(2,3)=13,\ f(-3,0)=9\}$ can be extended to distinguish between $x_1^2+x_2^2$ and $x_1^2+x_1x_2+x_2$. The expression $x_1^2+x_2^2$ satisfies the set $\mathcal{F}' = \mathcal{F} \cup \{f(3,2)=13\}$, while the other does not. On the other hand, the latter satisfies another set $\mathcal{F}'' = \mathcal{F} \cup \{f(3,2)=17\}$, while the former does not.

3.1.2 The equal-value partitioning

Since $\mathcal{F}$ contains only a finite number of examples of the function to be induced, it is likely that the equal-value space will not coincide with the semantic space. For example, there are an infinite number of polynomial functions whose graphs pass through a given set of k distinct points. Thus the equal-value space with respect to any finite $\mathcal{F}$ may not be identical to the semantic space.

For the purpose of inducing a function f, $\mathcal{F}$ is said to be *sufficiently representative* if it has the property that the simplest expression satisfying $\mathcal{F}$ also satisfies any $\mathcal{F} \cup \mathcal{F}'$, where $\mathcal{F}'$ is a set containing more examples of f. In other words, including more examples does not invalidate the solution found using $\mathcal{F}$. Consequently, if $\mathcal{F}$ is

	Number of different		Number of equal-value partitions		
	Expressions	Functions	$f(3,3)=a$	$f(7,17)=b$	$f(2,4)=c$
$n=1$	8	6	2	6	4
$n=2$	192	52	8	52	17
$n=3$	81408	2344	55	2058	155

Table 3.1: Number of equal-value partitions for $f(x,y)$

sufficiently representative, the first expression found is the solution.

As an example, consider the search space formed by $\times, +$ and two variables x_1, x_2. From the recurrence formula in Section 2.2.1, there are 8 syntactic partitions at level one (expressions with complexity one), 192 at level two and 81408 at level three. In comparison, there are 6 semantically equivalent partitions at level one (functions with complexity one), 52 at level two, and 2344 at level three. Moreover, while the number of equal-value partitions varies depending on $\mathcal{F}$, it can be even smaller as illustrated in Table 3.1.

When $\mathcal{F}$ contains only one example $f(3,3)=a$,[1] there are only 2 equal-value partitions at level one, 8 at level two and 55 at level three—much less than the number of partitions in the semantic space. In the worst case, there are as many equal-value partitions as there are semantic ones; for instance, when $\mathcal{F}$ contains the example $f(7,17)=b$. This is because all semantically equivalent expressions are equal-valued, which implies that equal-value partitions cannot out-number semantic-equivalent partitions. In other cases, as when $\mathcal{F} = \{f(2,4)=c\}$, the number of equal-value partitions lies between these extremes. In all cases, any reduction in the number of expressions at one level will propagate exponentially to deeper levels. This follows from the formula for C_n in Section 2.2.1—a small reduction factor R at level i will produce a reduction factor $F \cdot R^A$ at level $i+1$. So sometimes there are far fewer equal-value partitions than semantic-equivalent ones.

[1]The actual value of $f(3,3)$ is irrelevant in this illustration, as expressions in different partitions have different values for $f(3,3)$.

In practice, the set $\mathcal{F}$ usually contains several members, and the equal-value relation with respect to $\mathcal{F}$ is the same as semantic equivalence—otherwise the induction problem is under-specified. In other words, $\mathcal{F}$ is sufficiently representative. Thus searching the space of expressions that have equal values with respect to $\mathcal{F}$ is tantamount to searching the space of semantically equivalent expressions. However, it is easier to perform the former search since determining whether expressions are equal-valued is much easier than determining whether they are semantically equivalent.

3.1.3 Insufficiently representative example sets

What if $\mathcal{F}$ is not sufficiently representative? In this situation, the partition induced by $\mathcal{F}$ which contains the expression found also contains other semantically different expressions. The expression found is only one of many possible expressions which are equal-valued with respect to $\mathcal{F}$ but not with respect to some larger set of examples $\mathcal{F}' \supset \mathcal{F}$. However, all is not lost since the eventual solution must satisfy $\mathcal{F}$ and so must lie in the same partition. Hence, instead of restarting from the beginning, one can examine this partition more carefully by reconstructing its members and testing each against the additional examples. Any that pass the test are solutions of the original induction problem, assuming $\mathcal{F}'$ is sufficiently representative. If none passes, the problem has no solution within the given complexity limit.

Thus the problem of insufficient examples can be resolved by reconstructing and testing expressions in the equal-value partitions defined by $\mathcal{F}$. Consequently, it is not necessary to insist that $\mathcal{F}$ is sufficiently representative. Any expression that satisfies the examples seen so far can be modified if necessary whenever new examples become available. This suggests a faster way of inducing a function from examples.

3.1.4 Inducing a function from one example

Recall that the function induction problem is to find a function satisfying a set $\mathcal{F}$ of examples. Let $\mathcal{E}$ be the set containing any one of these examples. The search is

first performed under the equal-value relation determined by $\mathcal{E}$, which is particularly speedy to check since expressions need only be evaluated on one set of input arguments and compared with one output value. The expression found is then tested against the remaining examples in $\mathcal{F}$. If it satisfies them, it is the solution of the given induction problem. Otherwise, other expressions in the same equal-value partition with respect to $\mathcal{E}$ are reconstructed and tested. This process is repeated until an expression is found satisfying all examples in $\mathcal{F}$. If none is found, the induction problem as given does not have any solution.

The advantage in performing search this way is a potential reduction in storage requirements. The partitioning under $\mathcal{E}$ is much coarser than under $\mathcal{F}$, as there is a better chance of semantically different expressions being equal-valued. Since each equal-value partition must be retained for future reference, storage is directly related to the partition count. As in all breadth-first-style searches, it grows exponentially with complexity level. Therefore searching the expression space with respect to $\mathcal{E}$ generally requires much less storage than searching with respect to $\mathcal{F}$.

The only remaining problem is how to reconstruct all expressions in the particular partition determined by $\mathcal{E}$. Ideally, only semantically different expressions should be reconstructed, but that would require prior knowledge of the properties of the primitive functions. Hence, to maintain generality, syntactically different but semantically equivalent expressions are allowed to be reconstructed. This results in some redundancy, but that can be eliminated with appropriate semantic checks if desired. The next section applies this idea to an actual function search algorithm, using a simple reconstruction technique based solely on syntactic manipulation.

3.2 The equal-value search algorithm

This section presents a complete function induction algorithm utilizing the equal-value relation. Subsection 3.2.1 describes the equal-value transformation and its application in the equal-value search. Then subsection 3.2.2 presents the four steps

TRACING(EXPRESSION) returns ALTERNATE_EXPRESSION
  Let $F_{expr} \stackrel{def}{=} F(f_1, \ldots, f_k)$, be the input expression,
    where $f_1, \ldots, f_k$ are its immediate subexpressions
  For i=1, ..., k
    Select any expression with same value and complexity as f_i
    Call TRACING recursively to find its alternate form
    Denote the resulting alternate form as g_i
  Return the new expression $F(g_1, \ldots, g_k)$

Figure 3.1: Algorithm for tracing alternate expressions

of the equal-value search. Briefly, the first step selects one example $\mathcal{E}$ from the given set $\mathcal{F}$. The next finds an equal-value group satisfying $\mathcal{E}$, and the third tests all expressions in that group with the remaining examples in $\mathcal{F}$. If this step fails, the algorithm enters the final step, which extends the search space to include more complex expressions and restarts from step two. Finally, subsection 3.2.3 proves that the equal-value search algorithm is correct, complete and non-redundant.

3.2.1 The equal-value transformation

Consider any equal-value partition in the search space. Expressions with same complexity in this partition are said to be *alternate forms* of one another. The algorithm groups them according to their top-level functions. Two expressions are in the *same group* if they are equal-valued (with respect to $\mathcal{E}$) and have the same top-level function. Since replacing any subexpression by its alternate form does not change the expression's value, expressions in the same group can be obtained from each other by replacing some subexpressions with their corresponding alternate forms. Thus to reconstruct all expressions in the same group, the algorithm need only substitute equal-valued subexpressions. This task is done by the recursive tracing algorithm shown in Figure 3.1.

Since expressions in a group can be obtained from each other using the tracing al-

TRANSFORMATION(EXPRESSION) returns EXPRESSION
  Evaluate the given expression to determine its partition
  For each expression F_{expr} representing a group in that partition
    Recursively call TRACING(F_{expr}) (Figure 3.3)
  Return the expression returned by TRACING

Figure 3.2: Algorithm for reconstructing equal-value expression.

[illegible]

[illegible]

This reconstruction is called the equal-value transformation because it transforms a given expression to another with equal value. Since the only criterion for subexpression selection is identical value, the transformation is problem independent. It does not require prior knowledge about the primitive functions or their composition rules. Applying it to a breadth-first search gives a very efficient search strategy for arbitrary functions, called the equal-value search.

3.2.2 Major search components

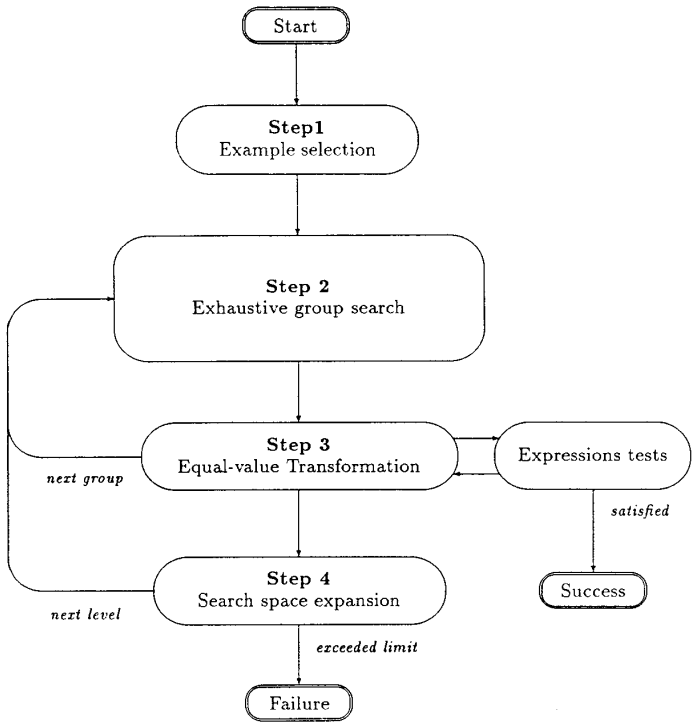
There are four steps in the equal-value search: example selection, exhaustive group search, expression transformation and testing, and search space expansion.

The first step is to select a member f of the given set F of examples. Since different choices result in different numbers of equal-value partitions, it may be worthwhile to [illegible]

[illegible]

[illegible]

Once an expression is found, the third step is to test all expressions in its group against other examples in F. The expression obtained from the previous step represents a group of many same-top-level-function expressions satisfying f. Since these may not be semantically equivalent, each must be reconstructed and tested against all remaining examples in F. The equal-value transformation (Figure 3.2, Section 3.2.1) is used in this step to reconstruct same-group expressions. The process stops when an expression is found satisfying all examples.



Step 1	Select one example $\mathcal{E}$ in the given set $\mathcal{F}$
Step 2	Search the equal-value space with respect to $\mathcal{E}$
Step 3	Test all expressions in the equal-value group found
Step 4	On failure, repeat for more complex expressions

Figure 3.3: Algorithm for the equal-value search

Assuming $\mathcal{F}$ is sufficiently representative, the first expression found will be the solution of the original induction problem. Otherwise, as new examples become available the algorithm can reconstruct more expressions in the same group and test them against the new examples.

The final step, invoked on the failure of the third step, is to continue the search for other groups and ultimately for more complex expressions. To do this, the algorithm backtracks to the second step and searches for another group of expressions at the same complexity level. If no expression at the current complexity level satisfies the examples, it begins searching the next level.

3.2.3 Proof of correctness, completeness, and non-redundancy

Obviously, the algorithm is correct if it can produce a solution that satisfies the problem statement. The equal-value search's correctness follows from the correctness of its two components, the exhaustive search and the transformation. This is because it finds a solution either by directly constructing it from original subexpressions or by transforming from another expression. In the former case, the algorithm's correctness follows from that of the exhaustive search component. In the latter case, it follows from that of the transformation component. Therefore, correctness can be established as follows.

Proposition 1 (correctness) *Provided that the set of examples $\mathcal{F}$ is sufficiently representative, the first solution found satisfying $\mathcal{F}$ is the function to be induced.*

PROOF: First, because the exhaustive search component is a conventional breadth-first search, the first expression f found is the simplest expression satisfying the first example $\mathcal{E}$. Therefore, if it satisfies all remaining examples in $\mathcal{F}$ and $\mathcal{F}$ is sufficiently representative, it must be the function to be induced.

Otherwise, expressions in the same group with f are reconstructed by the tracing algorithm (Section 3.2.1). Since expressions returned by the tracing algorithm are

equal-valued to the supplied expression, all of them satisfy the first example $\mathcal{E}$. In other words, the transformation algorithm always returns equal-valued expressions. And since the solution is the first expression found satisfying all remaining examples, it must be the function to be induced.

Therefore, the expression induced by the equal-value search always satisfies the examples, regardless of it being constructed directly or traced back from another. □

The completeness property of the equal-value search is its ability to induce any expression in the search space, either by explicitly constructing it or by tracing back from a non-solution expression. In other words, the search algorithm completely covers the whole semantic space.

Proposition 2 (completeness) *Any expression F_{expr} in a given group can be found either by explicit construction or by tracing back from an alternate expression F'_{expr} in its group.*

PROOF: By induction on n, the complexity of F_{expr}.

When $n = 1$, the result is trivial since F_{expr} has no subexpressions. The algorithm either constructs F_{expr} directly or an expression F'_{expr} in its group which has some (or all) arguments replaced by other variables with equal value.

Suppose the current complexity level is N, where $N>n$, so that all expressions of complexity n or less have been generated. If not, expand the database by generating more complex expressions until $N>n$.

Recall that an original expression is the first one constructed in an equal-value partition, and that all expressions in a partition are alternate forms of one another. Let F_i be any subexpression of F_{expr}. Since F_i has complexity $n-1$ or less, either it or an alternate form has been constructed. In the latter case, F_i is assumed to be constructible by tracing back from that alternate form (inductive assumption).

If any F_i is not original, replace it by the corresponding alternate form that is original. Let F'_{expr} be the resulting expression. By the definition of same-group

expressions (Section 3.2.1), F'_{expr} is in the same group as F_{expr}. Since F'_{expr} has complexity n or less and all of its subexpressions are original, it must already have been constructed. Applying TRACING to F'_{expr} will give F_{expr}. □

As each expression in the search space can be accessed in two different ways—direct construction or tracing back from another expression in its group—it is possible that some may be examined more than once. However, it can be proven that there is no redundant search by showing that each constructed expression belongs to a different group. From this, it follows immediately that the equal-value search does not trace any expression more than once. Therefore, by ensuring that only syntactically different expressions are generated, there will be no duplication in the search.

Proposition 3 (non-redundancy) *Each of the constructed expressions is in a different group.*

PROOF: Let F_{expr} and F'_{expr} be two constructed expressions. If they have different values, then they are in different groups. Otherwise, there are two cases:

1. If the top-level functions of F_{expr} and F'_{expr} are different, then F and F' must be in different groups, because they are formed using different functions.

2. Otherwise, compare corresponding arguments of F_{expr} and F'_{expr}. Since each argument is an *original* expression and original expressions have different values, there is no way to transform F_{expr} to F'_{expr} by replacing subexpressions by their alternate forms. Therefore F_{expr} and F'_{expr} cannot be in the same group, unless they are identical.

Consequently, constructed expressions are in different groups. □

Furthermore, by using only one example in the first stage of searching, the algorithm can avoid many expressions without explicitly considering them. To illustrate, suppose an expression F_{expr} fails to satisfy the first example. Since all expressions in the same group as F_{expr} are equal-valued to F_{expr}, they cannot be the solution, and thus can safely be ignored. And since each group may contain semantically different

expressions, it follows that the algorithm need only search a subset of the semantic space without compromising completeness. In other words, the equal-value search is at best very efficient, and at worst non-redundant.

3.3 Implementing the equal-value search

As in any algorithm, the performance of the equal-value search can be enhanced by appropriate choices of data structure and algorithm coding. In the current implementation, these decisions are influenced by the fact that Prolog is the implementation language and by the desire to maintain problem independence. In decreasing order of language dependency, the important implementation choices are as follows.

3.3.1 Bottom-up composition procedure

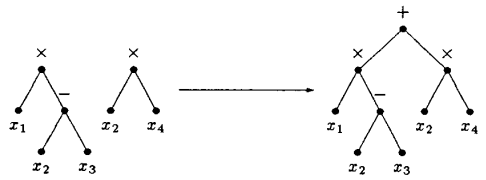
New expressions are constructed in a bottom-up fashion by selecting a top-level function and instantiating its arguments with previously constructed expressions (Figure 3.4a). Since the new expression is to be at the next complexity level, at least one of its subexpressions must be at the current complexity level. With Prolog's backtracking facility, successive choices of arguments can be selected quickly via enumeration, and thus construction time is proportional to the top-level function's arity—which is at most A (Section 2.2.1).

An alternative composition procedure is to extend previously constructed expressions by replacing their arguments with function calls (Figure 3.4b). That is, expressions are constructed in top-down fashion. Again, at least one argument chosen for replacement must be at the deepest level so that the new expression will belong to the next complexity level. With this procedure, argument-replacement time is proportional to the number of leaves in the corresponding evaluation tree—which may be as large as A^n, where n is its complexity (Section 2.2.1).

In principle, either approach is acceptable, as long as the composition procedure ensures that all and only syntactically different expressions are constructed.

(a) Bottom-up composition for $x_1(x_2 - x_3) + x_2x_4$

Selecting sub-expressions *Instantiating function arguments*



(b) Top-top composition for $x_1(x_2 - x_3) + x_2x_4$

Selecting main expression *Replacing variable by function*

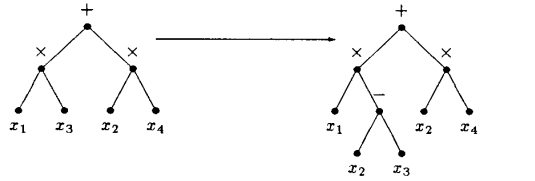


Figure 3.4: Composing new expressions

The bottom-up version was chosen because of its simplicity, which is an immediate consequence of Prolog's back-tracking facility. New expressions are generated by enumerating different choices for the function's arguments. In contrast, the top-down version requires more effort to avoid duplicate expressions, and so is less efficient.

Furthermore, as expressions are constructed from the bottom up, constraints of the induction problem can be checked immediately after each subexpression has been constructed. This incremental checking ensures that only constraint-satisfying subexpressions are considered. Expressions that violate the constraints are eliminated as soon as they are detected. This in turns leads to the elimination of many more unsatisfiable expressions at subsequent levels.

3.3.2 Indexed and indirect addressing

In the equal-value search, the two most frequent operations are *selecting* original expressions and *checking* duplicate values when constructing them. To minimize search time, expressions are stored in two different sets, ORIGINAL and ALTERNATE. This separation results in faster selection of subexpressions, which must be original, and faster transformation, which requires alternate form expressions. Members in both sets are indexed by their corresponding values so that they can be accessed quickly. The values of expressions in the ALTERNATE set are used as equal-value pointers—the links between equal-valued expressions—for tracing alternate form expressions.

This organization has two immediate benefits: first, no extra storage is needed for equal-value pointers, and second, the composition procedure does not have to consider duplicate expressions. Moreover, using values as addresses, duplication checks can take constant time if associative memory is used. Many implementations of Prolog provide a limited form of associative memory by hashing the first argument of each predicate. Hence duplication can be determined in (almost) constant time by making the value the first element of each expression's definition, which is then stored internally as a Prolog predicate (Figure 3.5).


```
original(value(1.2),level(1),expr(17)).
expression(17,/,[var(1),var(3)]).

original(value(12),level(0),var(1)).
original(value(10),level(0),var(3)).
```

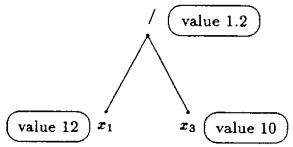


Figure 3.5: Index addressing for expressions

Since new expressions are constructed from previous ones, duplicate definitions will occur if each expression's definition also contains the definitions of its subexpressions. Therefore, instead of storing the full definition of each expression, only the top level function and the pointers to its arguments are stored (Figure 3.6). Thus each definition requires only a fixed amount of storage, regardless of its complexity. Although this appears to increase the access time for expressions, it turns out that there is no need to access the full definition at all, except when printing the final solution. This saving in storage is another reason for selecting bottom-up composition. The top-down version must store all expression definitions, even when they differ only at the deepest level.

Moreover, this indirect definition scheme offers advantages unrelated to storage reduction. First, constructing alternate form expressions is now even quicker, since replacing a pointer is faster than replacing a full definition. Second, when expression values are used as pointers (as discussed above), there is no need to replace subexpressions. Selecting the next alternate expression will be sufficient. And finally, since pointers are the subexpression values, expressions can be evaluated without accessing the full definition, by applying the top-level function to the pointers themselves.

3.3.3 Temporary definition buffer

Once an expression is constructed, its definition is not used until expressions at the next level are being constructed. If definitions were stored immediately in the internal

```
original(value(7.2),level(2),expr(215)).
expression(215,×,[var(2),expr(17)]).

original(value(1.2),level(1),expr(17)).
expression(17,/,[var(1),var(3)]).

original(value(12),level(0),var(1)).
original(value(6),level(0),var(2)).
original(value(10),level(0),var(3)).
```

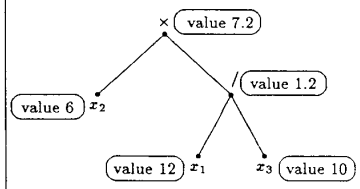


Figure 3.6: Indirect expression definitions

database, efficiency would be reduced since access time increases with the size of the database. This is further compounded by potential swap time in systems with virtual memory, when the database outgrows real memory. In order to avoid an unnecessary increase in access time and memory requirements, newly-constructed expressions are stored in an external file (or a null file if they are not needed[2]). This file is then read before embarking on the next complexity level. Note that if dynamic compilation is possible, it can be compiled before being loaded for constructing expressions in the next complexity level. This will speed up the search considerably.

3.3.4 Problem independent search algorithm

The equal-value search is problem independent in the sense that it contains no domain knowledge. It does not rely on any property of the primitive functions to avoid redundant search, nor does it make any assumption about the function being induced. Normally, the primitive functions are specified by the user as part of the induction problem and are thus different for different problems. For this reason, the method can be used for many induction problems.

On the other hand, it can take advantage of any special properties of the current

[2]The reason for this is simple. If the current complexity level has too many expressions, then it is practically impossible to induce expressions in the next level. Consequently, expressions in this level do not have to be stored.


```
function(+,(X+Y),Z) :-   pre-condition(+,X,Y),
                         evaluate((X+Y),Z).

pre-condition(+,X,Y) :-  X @=< Y.

evaluate((X+Y),Z) :-     value-of(X,A),
                         value-of(Y,B),
                         Z is A + B.
```

Figure 3.7: Example definition of +

induction problem. This is done by pre-compiling them into the specification of the primitive functions. For example, suppose + (addition) is a primitive in the current induction problem. To utilize its commutativity, + should be defined as shown in Figure 3.7. The `pre-condition` predicate encodes commutativity as a syntactic condition "`X @=< Y`", where `@=<` is a lexical ordering relation supplied by Prolog. It states that the name of argument `X` should alphabetically precede or be identical to the name of argument `Y`, and thus allows either `X + Y` or `Y + X` but not both. Similarly, type constraints or other special conditions can be encoded and included in the `pre-condition` predicate to avoid generating irrelevant expressions.

3.3.5 Scoop-and-filter approach

As described in Section 3.2, the equal-value search is performed in two stages, first seeking any expression in an equal-value group satisfying the first example, and then finding an expression in that group which satisfies all examples. This is essentially the *scoop-and-filter* approach in searching [Preparata 85]. With some easily evaluated criteria (in this case the equal-value relation), the search space is divided into many partitions. During the search, the algorithm looks for any member of the satisfying partition, which possibly contains non-solution members. Once a partition has been identified, its members are examined more carefully to filter out non-solutions.

In addition to the reduction in storage (Section 3.1.4), splitting the search process

offers the possibility of individual improvements on each stage. For example, the current implementation uses a conventional breadth-first search in the first stage, and a syntactic substitution procedure in the second. Although it is argued that these two simple algorithms are sufficient to ensure a tremendous performance increase, both of them could be replaced by more efficient algorithms to take advantage of any available domain knowledge.

In particular, a more efficient exhaustive search in the first stage would be preferable as the algorithm spends most of its time in this stage. The only condition is that the new search must be able to record traces of equal-valued expressions, which are required by the transformation in the second stage. Also, as noted in Section 3.1.4, the transformation is based solely on syntactic modification of expressions. This has the drawback that expressions being reconstructed are syntactically different but possibly semantically equivalent. A potential improvement is to replace it with a more efficient transformation that does not require expensive semantic checks but could reduce the amount of duplication.

3.4 Summary

The equal-value search described in this chapter takes advantage of a weak form of semantic equivalence. While the resulting search space does not coincide with the semantic space, it can nevertheless be used to approximate the semantic space for induction purposes. The result is a very small search space that can be used to locate all possible expressions.

Moreover, as the new strategy does not assume a sufficiently representative set of examples, it can search for functions using only one example. However, because one example is rarely representative, a transformation algorithm is necessary to ensure that all expressions are accessible. Therefore, the equal-value search algorithm is a two-stage strategy: first searching for a group of expressions with a particular value, and then for individual expressions within that group.

This multi-stage approach has several advantages. First, there are fewer expression-groups than individual expressions, giving substantial savings in both storage cost and search time. Second, if an expression does not satisfy the first example, none of its same-group expressions will, so they need not be reconstructed. In consequence, only a subset of the search space is examined. And third, by carefully selecting the first example, the user can control the coarseness of the search space division, thereby minimizing both storage requirements and search time. As demonstrated in the next chapter, the selection process is simple enough to be performed automatically by the induction algorithm.

On the implementation side, the equal-value search adheres to the principle of generality. Problem-specific information is not known *a priori*. This allows it to be applied to different problem domains with little modification. Moreover, the segregation of the two search stages enables the user to select more efficient algorithms as they become available. At present, the first stage is a breadth-first search, while the second is a syntactic substitution procedure. They can be replaced by other algorithms provided that the condition stated in Section 3.3.5 is satisfied.

Chapter 4

Experiments

This chapter presents empirical results to illustrate the equal-value search's performance. The induction problems are specifically chosen to test its ability to induce general functions—ones that are not amenable to domain-dependent short-cuts. In particular, the search algorithm does not know which primitives are unnecessary nor which variables are redundant in each problem. Neither does it require a knowledgeable user to provide "easy" induction examples. And with only a few examples to work from, it cannot rely on statistical analysis to extract additional information. Moreover, the function's arguments in all examples have no type nor constraint, so that techniques such as type-checking or constraint propagation are ineffective.

There are seven sets of experiments in this chapter, each describing several related induction problems. The first two demonstrate the search's performance in both worst-case and best-case situations. The third illustrates how to take advantage of the equal-value grouping effect to increase performance dramatically. The next three show the effect of different ways of formulating an induction problem. In all cases, the equal-value search consistently shows a high level of performance. For comparison purposes, the last set of experiments solves several standard problems in function induction. The results support the claim that the equal-value search is very efficient despite its generality.

4.1 Basis of comparison

Since the objective is to provide a better strategy for general function search, specialized strategies that assume prior knowledge or only apply to specific domains are inappropriate for comparison. Only generic search strategies are potential candidates

for a fair comparison.

Among those strategies, breadth-first search is chosen for two reasons. First, there is no difference in principle between various exhaustive search strategies. Breadth-first search consumes more storage but is generally more efficient than depth-first. Best-first is a variant of breadth-first that relies on some pre-defined evaluation functions. Iterative-deepening is another variant that combines successive depth-first searches. Second, equal-value search is basically a breadth-first strategy. Both search in order of complexity and store results for later use. Search time and storage cost are thus comparable, and so differences in performance are measurable in these terms. The choice of breadth-first search makes comparison more meaningful.

On the other hand, a standard breadth-first search, or any generic search for that matter, is too inefficient to provide a realistic comparison. Of course, such comparison is still useful because two different algorithms can be compared indirectly via another algorithm. For example, if algorithm A is twice as fast as algorithm B and four times as fast as algorithm C, then one can conclude that algorithm B is two times faster than algorithm C. However, it would make more sense if one could provide a more realistic comparison by improving the generic search somewhat within the current domain. This can be done by incorporating some properties of the function induction domain, strictly for the purpose of speeding up the generic search.

In particular, it is trivial to detect certain equivalent forms of functions, such as $a \odot b$ and $b \odot a$ when $\odot$ is commutative. There is no reason why one should not take advantage of simple equivalence checks like this providing they are not expensive. As it turns out, only the commutative check has negligible cost in implementation complexity and execution speed. For this reason, the breadth-first search is implemented with builtin commutative checks.

In all experiments, results are given separately for the equal-value search with and without commutative check. The comparison between breadth-first search *with* commutative check and equal-value search *without* commutative check illustrates the

Algorithm	Best case		Worst case		Average	
	Storage	Time	Storage	Time	Storage	Time
Breadth-first search with commutative check	752	4.95	1.64×10^6	4152	8.20×10^5	2078
Equal-value search w/o commutative check	564	1.83	1.81×10^5	634	9.08×10^4	318
Equal-value search with commutative check	432	1.78	1.36×10^5	507	6.82×10^4	254

Table 4.1: Inducing a binary function $f(x, y)$

improvement in performance even when comparison is not on an equal footing. When both incorporate the commutative check, it demonstrates the actual performance increase obtained from the new search strategy.

4.2 Experiment 1: Worst-case performance

4.2.1 Experiment 1a: Inducing any binary function

Consider the problem of finding a binary function $f(x, y)$ with complexity 3. Using the primitives $+, -, \times, /$, one could construct as many as 1.64×10^6 binary functions with complexity 3 but only 738 with complexity 2. Hence the best- and worst-case induction times could differ by several orders of magnitude. Assuming that all expressions have the same probability of being the solution, the expected search time is the average of the best- and worst-case times.

In this experiment, both algorithms seek all functions with complexity 3 or less, and at each complexity level, record the search time in CPU seconds and the number of expressions stored (Table 4.1). The best case is when the unknown function is the first one generated with complexity 3, while the worst case is when it is the last. Storage costs are represented by the number of expressions examined at each level, since each expression needs the same storage regardless of its complexity.

The worst-case and average-case figures in Table 4.1 show that the equal-value

search is 6.5 times faster than the breadth-first search. With the commutative check, it is 8.2 times faster. In terms of storage cost, the reduction ratios are 9.0 and 12.0 respectively. Ideally, the reductions in search time and storage cost should be the same since, since in breadth-first-style searches, both reflect the number of expressions generated and examined. But in practice, they usually differ for two reasons. First, it normally takes more time to access an item in a larger database. Second, depending on the induction problem, each partition contains a different number of expressions, and that changes the time required to examine partitions satisfying the first example. Both reduction figures are useful in comparing performance, although those for time vary depending on the actual implementation.

It is worth mentioning that including the commutative check does not offer significant additional improvement in either search time or storage cost (reduction ratios of 1.25 and 1.33 respectively). This is an immediate consequence of the effect of grouping equal-value expressions. Recall that the commutative check is one of many semantic constraints used to reduce the number of semantically equivalent expressions. But since semantically equivalent expressions are always equal-valued, the equal-value search uses only one of them to construct more complex expressions. That results in many semantically equivalent expressions not being constructed at subsequent complexity levels.

4.2.2 Experiment 1b: Inducing any ternary function

In Experiment 1a, arguments of the binary function are chosen from two given variables x, y. When more variables are included, the choice for arguments increases accordingly. However, the search space now contains not only functions with arity 2 or less but also functions with arity C_0 or less as well, where C_0 is the number of available variables. For example, when a new variable z is included, ternary functions can also be found in addition to binary functions. In general, the problem of finding n-ary functions with k redundant variables is a sub-problem of finding $(n+k)$-ary

Algorithm	Best case		Worst case		Average	
	Storage	Time	Storage	Time	Storage	Time
Breadth-first search with commutative check	3201	28.61	3.00×10^7	81329	1.50×10^7	40679
Equal-value search w/o commutative check	2862	10.92	5.01×10^6	12245	2.51×10^6	6128
Equal-value search with commutative check	2160	10.27	3.76×10^6	9343	1.88×10^6	4677

Table 4.2: Inducing a ternary function $f(x, y, z)$

functions. Thus this experiment could be viewed as an extension of the previous experiment having one redundant variable z. Table 4.2 summarizes the experiment's results.

As expected, the equal-value search offers large reductions in both search time and storage cost, even without the commutative check. The average search time is 6.6 times faster without and 8.7 times faster with the commutative check. The corresponding storage costs are 6.0 and 8.0 times less. As in Experiment 1a, the small additional reduction of 1.3 for both search time and storage cost emphasizes the advantage of the equal-value search when domain knowledge is unavailable.

4.3 Experiment 2: Two specific induction problems

For a more typical comparison, in this set of experiments the problem is to induce specific functions satisfying the examples given in Figure 4.1. With the primitives $+, -, \times, /$, the expected solutions are $f(x_1, x_2, x_3, x_4) = x_1x_3 + x_2(x_1 - x_2)$ and $g(x_1, x_2, x_3, x_4) = x_3/x_2 - x_1x_2/x_3$. Both have complexity 3 and require only three primitives from the list of four. Thus, as in typical function induction problems, both unnecessary primitives ($/$ for f and $+$ for g) and redundant variables (x_4 for both functions) are included.

Input examples				
x_1	1	3	3	3
x_2	11	10	7	6
x_3	91	15	19	18
x_4	10	37	−13	71
$f(\cdots)$	83	25	29	36
$g(\cdots)$	8.03	−0.50	1.61	2.00

Figure 4.1: Two induction problems

4.3.1 Experiment 2a: With no redundant variable

First, consider the case where there is no redundant variable. That is, the problem is to find $f(x_1, x_2, x_3)$ and $g(x_1, x_2, x_3)$ satisfying the examples in Figure 4.1. Theoretically, there are as many as 8.77×10^7 expressions with 3 variables or less in the search space defined by $+, -, \times, /$. Assuming an average time of 2.8 ms per expression as in Experiment 1b, it would take approximately 68.2 hours to examine them all. However, since this time f and g are not last in the function space, search times are much faster. In fact, it takes only 242 seconds and 458 seconds to find f and g respectively, using a plain breadth-first search. Adding the commutative check reduces these figures by almost half to 133 seconds and 231 seconds.

This illustrates that even a trivial semantic check can improve search time and reduce storage cost. Unfortunately, as more complicated checks are performed, the rate of reduction diminishes sharply. In practice, only the simplest semantic equivalence checks are worthwhile. That is why only commutativity is used in the breadth-first search.

In Table 4.3, search times are in CPU seconds while storage costs are represented by the number of expressions stored. The performance varies with different choices of first example, and the figures in the *Average* columns are the average of four different choices. The best-case figures are obtained with the first example $f(3, 6, 18)$=36 for f and $g(3, 6, 18)$=2 for g. The reduction ratios as compared to breadth-first search

(a) Inducing $f(x_1, x_2, x_3) = x_1x_3 + x_2(x_1 - x_2)$

Equal-value search algorithm	Best case		Average		Reduction ratio	
	Storage	Time	Storage	Time	Storage	Time
w/o commutative check	9794	35.6	15702	48.1	3.07	2.77
with commutative check	9144	32.8	14985	47.2	3.21	2.82

(b) Inducing $g(x_1, x_2, x_3) = x_3/x_2 - x_1x_2/x_3$

Equal-value search algorithm	Best case		Average		Reduction ratio	
	Storage	Time	Storage	Time	Storage	Time
w/o commutative check	16268	56.5	27369	82.9	3.17	2.79
with commutative check	15618	54.7	26652	79.5	3.26	2.91

Table 4.3: Inducing with no redundant variable

are reported in the final columns. They are obtained by dividing the corresponding figures of the breadth-first search by those of the equal-value search.

As in previous experiments, the reductions in storage and times show the higher performance of the equal-value search. They are not as large as in Experiments 1a-b because this is not worst-case induction; the solutions for f and g were found well before the last function with complexity 3 is constructed. Also, as expected, the inclusion of the commutative check in the equal-value search does not offer much increase in performance.

4.3.2 Experiment 2b: With one redundant variable

Now, suppose that there is one redundant variable x_4, so that the problem is to find $f(x_1, x_2, x_3, x_4)$ and $g(x_1, x_2, x_3, x_4)$ satisfying the examples in Figure 4.1. From the relation in Section 2.2.1, the maximum number of expressions with 4 variables or less is 1.37×10^9. Assuming an average access time of 2.8 ms per expression, it would take 44.4 days to examine them all. But because both functions are not last in the search space, it needs only 1147 seconds to find f and 2243 seconds to find g using a breadth-first search without any semantic check. As in Experiment 2a, the

(a) Inducing $f(x_1, x_2, x_3, x_4) = x_1x_3 + x_2(x_1 - x_2)$

Equal-value search algorithm	Best case		Average		Reduction ratio	
	Storage	Time	Storage	Time	Storage	Time
w/o commutative check	63613	201	71096	223	2.99	2.64
with commutative check	61451	196	68910	213	3.09	2.77

(b) Inducing $g(x_1, x_2, x_3, x_4) = x_3/x_2 - x_1x_2/x_3$

Equal-value search algorithm	Best case		Average		Reduction ratio	
	Storage	Time	Storage	Time	Storage	Time
w/o commutative check	112580	336	128352	361	2.96	2.88
with commutative check	110418	326	126165	352	3.01	2.96

Table 4.4: Inducing with one redundant variable

commutative check reduces search times by half to 589 and 1041 seconds respectively.

While these new times represent considerable improvement, those from the equal-value method are even better (Table 4.4). On average, it takes only 223 seconds to find f and 361 seconds to find g even when no semantic check is available. With the commutative check, the average search times are 213 seconds and 352 seconds respectively. Compared with the breadth-first search without the commutative check, the same search with commutative check is only 2.0 times faster for f and 2.2 times faster for g, while equal-value search without the commutative check is, on average, 5.1 times faster for f and 6.2 times faster for g. This presents a compelling argument for using the equal-value search when little domain knowledge is available.

4.4 Observations

4.4.1 General performance increase

The equal-value search achieves significant reductions in both search time and storage cost compared to a standard breadth-first search. Although these reductions vary with the induction problem, they are generally large enough to make equal-

value search attractive. When domain knowledge is unavailable, replacing a standard breadth-first search by the equal-value search results in much better performance. As demonstrated by Experiment 1, the improvement at complexity level 3 can be up to a factor of 12.

This gives the main reason for employing the equal-value relation in place of any syntactically defined equivalence relation. Instead of resorting to complicated semantic checks, which may be costly to perform, one simply assumes that equal value implies equivalence. If additional examples invalidate this assumption, the equal-value transformation provides a simple but fast correction. The net result is a very fast search strategy for inducing arbitrary functions.

4.4.2 Effect of equal-value grouping

Recall that the equal-value relation divides the search space into partitions containing equal-value expressions. In effect, it groups expressions according to their values. This reduces storage cost, because each group needs the same amount of storage regardless of how many expressions it contains. And since most of the search is spent seeking the right group, the total time is reduced accordingly. This grouping is similar to the grouping resulting from semantic checks—such as the commutative check, which groups expressions according to their top-level function's arguments—except that expressions in the same semantic group are semantically equivalent, unlike those in the same equal-value group.

Although both reduce the number of partitions—and thus search time—considerably, the equal-value grouping offers a much larger reduction than does commutative grouping. But when both are included, search performance is about the same as for equal-value grouping alone. This indicates an overlap in the techniques' effectiveness, with equal-value grouping offering most of the reduction.

4.4.3 Effect of different choices of examples

As illustrated in Experiments 2a and 2b, different choices of first example result in different storage costs and search times. This behavior is a consequence of the equal-value relation's definition. Recall that in Section 3.1.1 the equal-value relation is defined *with respect to a set of input-output examples.* Hence different examples result in different divisions of the search space. And since each partition contains at most F expressions (where F is the number of primitive functions) at any given complexity level, this gives different storage costs and search times for different sets of examples. Taking advantage of this behavior, one can further improve search performance at runtime by providing appropriate examples. This possibility will be discussed in Section 4.5.

4.4.4 Problem-independent reduction

As described in Chapter 3, the equal-value search is simply a breadth-first search augmented with the equal-value relation and its corresponding transformation. With the search component based on the standard breadth-first algorithm, its generality is governed by that of the transformation. And since the latter is based on syntactic substitution, it is independent of the induction problem. Consequently, the equal-value search strategy can be used in specialized algorithms, in which case the combined algorithm is even more efficient than equal-value search or domain-specialized search alone.

The added efficiency is possible because the reduction is independent of both the actual search algorithm and the input data. If an algorithm can offer even a small reduction in search space, the equal-value grouping multiplies that reduction at subsequent complexity levels. It is irrelevant whether the reduction results from a better search or from a particular choice of examples and primitive functions.

4.5 Experiment 3: Synthesizing appropriate examples

With the observations from Experiments 1 and 2, it is clear that the equal-value search's runtime performance can be improved with changes in the input data. Of course, these changes could also affect the breadth-first search's performance as well. Hence, in this and subsequent experiments, any change in the problem is evaluated using both breadth-first and equal-value searches. Thus any gain in performance undoubtedly stems from the equal-value search.

In these experiments, the search algorithms are presented with the problem of inducing the relationship between values obtained in a hypothetical testing of braking systems' effectiveness (Figure 2.2, Section 2.1). Solving this problem using a breadth-first search with commutative check requires 66.2 hours. However, it takes on average 9.42 hours using the equal-value search without the commutative check and 7.54 hours with it.

As expected from previous experiments, the equal-value search offers a large reduction in search time, whether or not the commutative check is included. In fact, the inclusion of commutative checking only increases the performance a little. For this reason, in Experiments 3 to 6, the equal-value search does not use the commutative check while breadth-first search does.

Moreover, search times vary depending on which example is used to define the equal-value relation. With the example $f(40, 1000, 3200.12.5)$=250. search time is only 6.94 hours. But with the example $f(30, 1200, 2800, 12.86)$=192.86. it increases to 14.74 hours. This wide variation indicates the importance of selecting appropriate examples for induction.

4.5.1 Defining appropriate examples

Examples are said to be *appropriate* if they result in faster induction. In general, such examples are not easy to determine except by trial and error. However, it turns out that in the equal-value search, they are easily obtainable by setting all variables to the same value. Specifically, the system synthesizes its own examples by presenting a set of variables $x_1, \cdots, x_k$, all with the same value, to the user who will supply the corresponding function value $f(x_1, \cdots, x_k)$.

Recall that in the formula for the search space size C_n (Section 2.2.1), C_0 is the number of variables that can be used in constructing expressions. In this experiment, C_0 is 4 because all four arguments v, m, f, t are different variables. But when any two are equal in value, they are treated as one, due to the effect of equal-value grouping. This reduces C_0 on subsequent levels, and thus reduces the search space. In cases where it is not possible to equate arguments' values, the algorithm could select examples in which one value is expressible in terms of others, eg: $x_3 = x_2 - x_1$. The search would still be faster, as x_3 will eventually be found to have the same value as $x_2 - x_1$ and then both are treated as one.

This selection method eliminates any possibility of choosing an inappropriate example. If there is more than one independent variable, they are set equal to one another. If there is only one, it is set to 1 or 0 (the identity element for $\times$ and $+$ respectively). In this way, the equal-value search can choose its own examples to further speed up the induction process.

Figure 4.2 illustrates the difference in search times when using different first examples. Search times are compared against those obtained with the example $f(30, 1200, 2800, 12.86)$=192.86, which is the worst case of the four examples given in Figure 2.2. Because of large differences, the horizontal scale in Figure 4.2 is logarithmic. Each unit of length corresponds to a doubling of search time. Times for the breadth-first search are not reported because they are within 5% of the previous figure of 66.2 hours. This should be evident from the fact that breadth-first search constructs expressions using variable names rather than values, and so its search time is unaffected by different sets of examples.

Note that the comparisons in this experiment are against the equal-value search using the example $f(30, 1200, 2800, 12.86)$=192.86. The reduction ratios against the

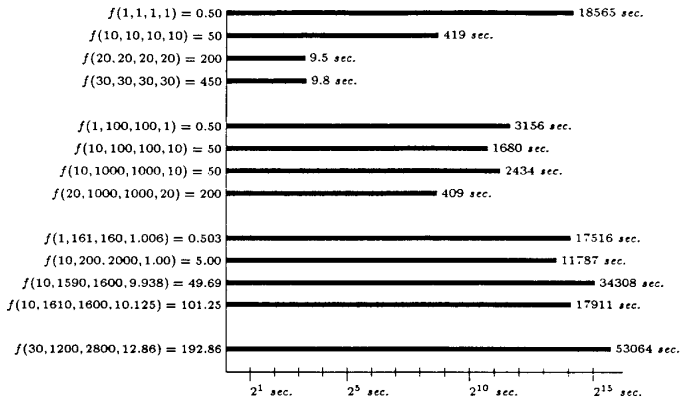


Figure 4.2: Search times using various first examples

breadth-first search are 4.5 times greater. This factor is the search time ratio between the breadth-first search (with commutative check) and the equal-value search using $f(30, 1200, 2800, 12.86)$=192.86 (without commutative check).

4.5.2 Case 1: All arguments have the same value

In Figure 4.2, the first four bars represent examples in which all arguments have the same identical value. This shows how to maximize the equal-value grouping effect with synthesized examples. Although the change—setting all arguments to the same value—is simple, it has a dramatic effect on search time. At best, it reduces the search time to 9.5 seconds from 14.74 hours, achieving a reduction ratio of almost 5586. At worst, it takes only 5.16 hours, giving a reduction ratio of 2.8.

What causes the large variation? At first sight, the value of the arguments should be immaterial, since $C_0 = 1$ in all four examples. Not surprisingly, the answer comes from the fact that search reduction relies on the equal-value grouping effect. As more and more expressions have the same value, it takes more time to examine the selected

partition to find the solution. Using the example $f(1, 1, 1, 1)$=0.5 takes longer because there are more expressions with value 0.5 than with value 50, 200 or 450.

From this observation, it becomes clear that the algorithm should try to synthesize examples with "uncommon" output values, and reject those whose value is somehow "special". In the domain of arithmetic functions, examples of such values include small integers such as 0 (which results from $x-x$), 1 (from x/x) and 2 (from $(x+x)/x$), or simple fractions such as $\frac{1}{2}$ (from $x/(x+x)$).

4.5.3 Case 2: Some arguments have the same value

In cases where all arguments cannot have the same value, one should try to equate the arguments as much as possible. While this is not as satisfactory as before, it is still better than selecting random values for arguments. The next four examples in Figure 4.2 gives the search times when two pairs of arguments have identical values. As expected, the worst time results from a choice of example which has the output 0.5—namely, $f(1, 100, 100, 1)$—this gives a reduction ratio of only 16.8. An example with a more general output, $f(20, 1000, 1000, 20)$=200, results in a reduction ratio of 130.

4.5.4 Case 3: Arguments expressed in terms of others

When it is not possible to have identical values for the arguments, it is beneficial to make some argument easily expressible in terms of other arguments' values. For example, in $f(1, 161, 160, 1.006)$=0.503, the third argument has value $161-1$=160. In another example, $f(10, 200, 2000, 1)$=5.00, the third argument has value 10×200= 2000. The last four problems in Figure 4.2 shows the search times when examples are synthesized this way. The reduction in search times are comparable since none of the output values is special. Even though all arguments have different values, the average reduction ratio 2.6 demonstrates the benefit of synthesizing appropriate examples instead of relying on the user to provide them.

4.6 Experiments with different formulations of a problem

Search time is also affected by the way an induction problem is formulated. While both breadth-first and equal-value search result in similar behavior under these changes, the latter consistently outperforms the former. The next four sets of experiments use the stopping-distance problem in Figure 2.2 to illustrate this characteristic.

4.6.1 Experiment 4: Changing the list of primitive functions

If it is known or suspected that the function to be induced does not require all of the available primitives, the search algorithm could proceed with a partial list of primitives, expanding the list later as necessary.

Using the primitive list $\{+, \times, negate, reciprocal\}$ (Figure 2.2(b)), the average search time reduces to 121 seconds, achieving a reduction ratio of 281. Using only three primitives $+, \times, /$, it reduces to a mere 16.2 seconds. giving a reduction ratio of 2093. But when the primitive list contains the five operators $+, -, \times, /, negate$, search time increases to 11.3 hours—1.2 times worse than before.

Obviously, the reduced performance is inevitable because there are now five choices of primitive instead of four as previously. However. this is a relatively small effect compared to what could have happened if the choice of primitive list was appropriate (1.2 times worse compared to 281 and 2093 times better). This illustrates the high pay-off of correctly choosing an appropriate primitive list but the relatively low risk of exceeding resource limits with an inappropriate list.

In comparison, the breadth-first search with commutative check takes 235 seconds (1.9 times slower than the equal-value search) with the primitive list $\{+, \times, negate, reciprocal\}$, 39.6 seconds (2.4 times slower) with the list $\{+. \times, /\}$, and 94.2 hours (8.3 times slower) with the list $\{+, -, \times, /, negate\}$.

4.6.2 Experiment 5: Incorporating function preferences

Suppose it is known or suspected that the top level function is $\times$, then the function list could be organized so that $\times$ is the preferred primitive function choice. This will enable the algorithm to examine expressions containing $\times$ first in each complexity level, thus avoiding unnecessary search. Preferences could be assigned by the user or by statistically analyzing the input examples.

When inducing the stopping-distance function, a preference for $\times$ reduces the search time almost 500 times from 9.42 hours (in Experiment 3) to 68 seconds. But when the preference is for $+$, the search time increases to 19.25 hours. Of course, the breadth-first search also exhibits the same behavior, but with longer search times. With a preference for $\times$, it takes more than 157 seconds (2.3 times slower than the equal-value search). When $+$ is preferred, it requires 98.5 hours (5.1 times slower).

4.6.3 Experiment 6: Utilizing monotonicity

Suppose it is given that $\times$ is a monotonic function of each of its arguments. With this information, an induction system can propose a hypothesis that the distance function $d(v, m, f, t)$ is monotonic with respect to one of its arguments, for example the car's mass m. It can then ask the user for more examples to confirm or reject this hypothesis. Repeating this for all variables, it can conclude that the function d is monotonic with respect to both v and m. This information can be used by proposing three new *pseudo-variables* $v \times v$, $v \times m$ and $m \times m$. These are called pseudo-variables since they are used as if they were variables given in the induction problem.

The benefit depends on whether the induced expression contains any of these pseudo-variables. If so, there is a tremendous advantage in their inclusion, as the expressions containing them are in effect examined before all other expressions with the same complexity. Otherwise, the penalty is minimal since they will be introduced anyway at the first complexity level, and thus will not be considered as separate variables subsequently.

In this particular induction problem, the penalty is that C_0 momentarily increases from 4 to 7 in the first iteration, but decreases to 4 subsequently (when inducing expressions with complexity 1 or more). As it turns out, $v \times m$ is one subexpression in the solution, and the search time is reduced from 9.42 hours (33913 seconds) to 11.23 minutes (674 seconds)—a reduction of more than 50 times. In the case of breadth-first search, it is not clear how to take advantage of the monotonicity property without extensive modifications to the search algorithm. Consequently, there is no comparison in this experiment.

4.6.4 Experiment 7: Combining the above

Since each of the above is independent of the others, the equal-value search could employ all of them to achieve a larger reduction factor. In this set of experiments, the comparison is between the equal-value search and the breadth-first search, both with the same input data. In each experiment, there are two different sets of input data. One is the *standard set* which contains four randomly generated examples and the four primitives $+, -, \times, /$. The other is the *minimal set* which contains four synthesized examples (as discussed in Experiment 3) and only necessary primitives (including their commutativity property). The two sets are designed to simulate the effect of different input data on the two search algorithms. The standard set represents cases where there is no prior knowledge about the problem. The minimal set represents cases where such knowledge is maximally utilized.

Table 4.5 summarizes the results for some of the standard problems in function induction [Falkenhainer 86] and the previous stopping-distance function. These problems are listed in order of increasing difficulty.

1. *The ideal gas law* $\frac{PV}{NT} = 8.32$, with pressure P, volume V, number of molecules N, and temperature T. There is no redundant variable in this problem. Using breadth-first search, it takes 38.3 seconds with the standard set of inputs.

2. *Falling bodies experiment* $v = 2h/t$, with height h, falling time t, and terminal velocity v. In this problem, there are two redundant variables, the object's mass m and its radius r. Using the standard set, breadth-first search requires 85 seconds to solve this problem.

3. *The stopping-distance function* $d = v^2 m/(2f)$, with mass m, starting velocity v, brake force f, and stopping distance d. In this problem, there is one redundant variable, the brake duration time t. This problem takes 75.9 hours to solve using breadth-first search and the standard set of inputs.

4. *The kinetic energy conservation law* $(v_1^2 - v_1'^2)/(v_2'^2 - v_2^2) = m_2/m_1$, with starting velocities v_1, v_2, ending velocities v_1', v_2', and corresponding objects' masses m_1, m_2. There is no redundant variable in this problem. Solving this problem by breadth-first search requires 113.6 hours using the standard set.

Search time for	Breadth-first		Equal-value	
	Standard	Minimal	Standard	Minimal
Ideal gas law	3.83×10^{1}	7.84×10^{-1}	2.36×10^{1}	2.83×10^{-1}
Falling bodies	8.50×10^{1}	4.40×10^{0}	3.84×10^{1}	8.17×10^{-1}
Stopping distance	2.73×10^{5}	9.49×10^{1}	3.52×10^{4}	2.02×10^{0}
Kinetic energy	4.09×10^{5}	4.42×10^{4}	1.39×10^{4}	6.33×10^{2}

Storage cost for	Breadth-first		Equal-value	
	Standard	Minimal	Standard	Minimal
Ideal gas law	8.84×10^{3}	3.62×10^{2}	6.48×10^{3}	1.35×10^{2}
Falling bodies	1.50×10^{4}	1.65×10^{3}	8.49×10^{3}	4.07×10^{2}
Stopping distance	8.54×10^{7}	3.89×10^{4}	1.41×10^{7}	7.68×10^{2}
Kinetic energy	1.17×10^{8}	1.55×10^{7}	3.40×10^{6}	1.35×10^{5}

Table 4.5: Results of combining several improvements

In Table 4.6, the columns *Standard* give the reduction ratios between the breadth-first search with commutative check and the basic equal-value search using the standard set of input data. This is the normal comparison as in previous experiments.

Induction problems	Storage cost		Search time	
	Standard	Minimal	Standard	Minimal
Ideal gas law	1.36	2.68	1.42	2.71
Falling bodies	1.77	4.05	2.47	10.12
Stopping distance	6.07	50.65	7.76	46.98
Kinetic energy	34.41	114.82	29.42	69.83

Table 4.6: Combined reduction ratios

Here it illustrates the advantage of the equal-value search on different induction problems. To illustrate the combined performance increase, the figures in columns *Minimal* show the reduction ratios as compared to breadth-first search.

For accurate performance comparisons, ratios for both storage cost and search time must be used, although the latter will depend on the implementation language and the efficiency of the implementation. Since both breadth-first and equal-value search use the same input data, these figures reflect the net performance increase offered by the equal-value search as opposed to that offered by the minimal set of inputs alone.

4.7 Summary

The results in this chapter clearly demonstrate the equal-value search's superior performance. In general, the absence of problem-specific knowledge has less impact on it than on standard breadth-first search. But if such knowledge is supplied—for example, in the form of conditions for meaningful values of arguments—they can be encoded as constraints on the composition of new expressions. This way, invalid and irrelevant compositions can be avoided earlier in the generate phase instead of at the test phase. Other constraints which cannot be encoded can still be checked later on.

Moreover, as the search's performance depends on the actual input data (examples and primitives), one can request appropriate data that speed up search time and decrease storage cost. As demonstrated in Experiments 3–5, such data can be

supplied by the user or requested by the system using a pre-processing module. In general, this module consists of routines to statistically assign function preferences, to experimentally reduce the list of primitives, to estimate the search space size and most importantly to suggest alternative examples if those given result in a large search space.

Chapter 5

Applying the Equal-Value Search

With its generality and high performance, the equal-value search can be used to reduce solution time in situations where syntactic search was previously the only viable problem-solving technique. This chapter explores several applications, showing how the method can be applied and discussing potential extensions that take advantage of available domain knowledge.

First, Section 5.1 explains how the new search is used to induce functions in both numeric and symbolic domains. In the former, the presence of round-off errors does not adversely affect performance as it does in some other function induction methods. This leads to a solution of another problem, *function approximation*, which seeks a function that approximates a given set of data. In the latter, the equal-value search is less effective and in some cases does not offer any improvement. Fortunately, this can easily be remedied on a case by case basis.

Next, Section 5.2 discusses the related problem of finding equivalent forms of a given function. It outlines a procedure which uses the new strategy as a filtering module to accelerate the search for equivalent forms. Also discussed is how to use the same procedure in another application domain—synthesizing straight-line programs from input-output examples. Synthesized programs are sequences of code sections, and there is no provision for handling looping or conditional branching, for these require extensive knowledge about programming constructs.

Section 5.3 then addresses the problem of unknown constants in function induction. As it stands, the equal-value search cannot induce functions with unknown constants. With some extensions, however, it could induce functions with one unknown constant. One application for this extended method is to solve the problem of clustering examples using functional descriptors. In the case of many unknown

constants, the equal-value search does not offer any advantage over simply restricting the function forms as in BACON.6 [Langley 83] and COPER [Kokar 86].

Finally, Section 5.4 summarizes and suggests future research directions.

5.1 Conventional function induction

5.1.1 Numeric functions

In practice, it is usually acceptable to induce a reasonable approximation instead of insisting on the absolutely correct solution. There are two reasons for this. First, the approximation may be much simpler than the correct solution, so that induction time is much faster. Second, there is the question of whether the input-output examples are completely accurate. Beside the usual round-off errors, the given data could be subject to experimental or measurement errors. Thus a function that satisfies all examples may not capture the relation underlying them.

For example, consider the robot learning problem. Suppose one wishes to teach a robot how to move around by giving it a few examples. The robot's sensors record electrical voltage, current, elapsed time, starting and final positions with respect to a coordinate system. Internally, these values could be represented as a cause-effect sequence such as

position(12.4,30.9) —[*apply(port-A,12,4.5)*]→ position(15.1,30.9)

where "*apply(port-A,12,4.5)*" is the command for applying 12 volts to port A for 4.5 seconds. The effect of this command is to move the robot from position (12.4,30.9) to position (15.1,30.9).

Since sensors will not be absolutely accurate, recorded values are subject to both measurement and digitizing errors. The learning routine will have to take these errors into account when inducing the mapping between the old and new positions. Such a function will be acceptable if it leads the robot from the old position to within

a small radius of the target. Minor position adjustments would then be trivial to perform.

The equal-value search can easily accommodate this situation. By considering values that are very close to each other as being identical, the equal-value search can tolerate a certain degree of perturbation in the input values.[1] The actual magnitude of tolerance depends on the sensitivity of the function to fluctuations in the inputs. Measurement errors within this tolerance will have no effect on the final solution—they will give the same result.

In the case when there is no measurement error, the grouping of almost identical values could be reversed by the equal-value transformation. Recall that the transformation is used to distinguish equal-value but syntactically different functions (Section 3.2.1). Applying it to the solution will bring back all expressions previously deemed "equal-value". Hence, by choosing the expression with output value closest to the expected function value, one can avoid the erroneous assumption that all previously-deemed-equal values are actually identical.

5.1.2 Symbolic functions

The problem of symbolic function induction is a special case where the function values are not numeric. One example of this problem is to learn structural descriptions using a hierarchical classification of objects according to their properties. The solution is a boolean expression that correctly describes the given set of examples. This problem can be solved by first translating the input examples to tuples of boolean values using the membership hierarchy, then inducing a boolean expression satisfying all converted tuples. Since boolean arithmetic is a special case of ordinary arithmetic, one could apply the equal-value transformation with appropriate boolean operators as primitive functions.

However, experiments have shown that searching for boolean functions is disappointedly inefficient, because the search space can only be divided into two large equal-value partitions, instead of many small ones. Little acceleration is achieved because, on average, half of the search space must be examined in order to find the correct expression. In all experiments, equal-value search using one example performs no better on average than a breadth-first search. But when 3 examples are used to define the equal-value relation, performance is better as the space is now implicitly divided into 2^3=8 partitions instead of 2^1=2 partitions. This is because with three examples, there is less chance of semantically different functions being equal-valued.

[1] In fact, all previously reported experiments were performed with the error tolerance of 1% to avoid floating-point errors.

Consequently, the equal-value search is best applied when the range of functions has many different values. If the range is limited, as in the case of boolean functions, more than one example must be used to increase the number of equal-value partitions. This will result in many small divisions of the search space, offsetting the overhead in tracing alternate expressions. The break-even point is, of course, dependent on the particular implementation. In general, if tracing an alternate expression takes k times as long as constructing it from scratch, then the range of functions should have at least k values to offset the overhead of transformations. Failing that, the equal-value relation must be defined using multiple examples. To handle this possibility, one could extend the algorithm with a pre-processing module capable of determining how many examples are needed to induce the given function.

For example, consider the problem of inducing the function *xor* with the set of examples { *xor*(0,0)=0, *xor*(0,1)=1, *xor*(1,0)=1, *xor*(1,1)=0 } and the primitives *and, or, not*. Suppose the equal-value relation is defined using only the first example *xor*(0,0)=0. Then all boolean expressions must be either in the partition with value 0 or the one with value 1, under the input (0,0). Those in the 0-partition, for example *and*(0,0), are equal-valued to the solution *xor*(0,0) and so must be examined carefully. Those in the 1-partition, for example *or*(0,*not*(0)), are not equal-valued to the solution and so can be ignored safely. Clearly, such a partitioning does not offer any search improvement, for on average half of the search space must still be

examined.

However, suppose the equal-value relation is defined using the last three examples $xor(0,1)=1$, $xor(1,0)=1$, $xor(1,1)=0$. Now, all boolean expressions are in one of the 8 partitions (0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0) and (1,1,1), but only those in the (1,1,0)-partition are equal-valued to the solution. Undoubtedly, searching one-eighth of the search space is quicker than searching one-half of it. Of course, one could use all four examples to divide the search space into 16 partitions instead of 8.

5.2 Accelerating equivalence search

5.2.1 Semantic equivalence

A variant on function induction is the problem of finding all semantic equivalents of a given function. Since there are invariably an infinite number of equivalent forms, finding semantic equivalents is constrained by a maximum complexity limit and by a list of available primitive functions. As the syntactic space is astronomically large even for functions with complexity 3 or 4, solving this problem by exhaustive search is very inefficient. However, using the equal-value search, it becomes relatively easy to exhaust the syntactic space, using a two-stage search method.

The first stage is the elimination of easily-detectable semantically different expressions, namely those with values different from the given function's. This is done with the equal-value search as follows. First, a large set of input-output examples is generated by evaluating the function with random arguments. Then the equal-value space is searched using only one example from that set. Any expression found is tested against the remaining examples. The algorithm repeats and collects all expressions (up to a given complexity limit) that satisfy these examples. After testing against a large set of randomly generated examples, it is highly probable that the remaining expressions are semantically equivalent to the original function. Of course,

there is a small chance that they are not, but that possibility can always be reduced by generating additional examples.

The second stage translates each expression collected into some predefined canonical form. For this, one needs equivalence transformation rules, such as $y+x = x+y$ or $x(y+z) = xy+xz$, if the domain includes the arithmetic operators. The canonical forms are compared with the canonical form of the original function. Any one which is the same must have been derived from an expression that is semantically equivalent to the original function.

The advantage of using the equal-value search as a filtering stage is that instead of applying the equivalence transformation rules directly to all expressions in the syntactic space, one need only apply them to a much smaller number of equal-valued expressions. This improves solution time significantly, because equivalence transformation is not easy in general. The increase in performance is directly proportional to the ratio of syntactic space size to the number of equal-valued expressions, which grows very quickly for non-trivial functions.

5.2.2 Operational equivalence

Of course, the function to be induced is not necessarily restricted to a mathematical formula. It could be any form of mapping, even a partial mapping in a non-numeric domain. When the primitive operations are symbolic, every composition of them can be thought of as a program, with the mapping being the program statement. the parameters the input data and the mappings' values the output data (or intermediate results). In a sense, instead of inducing a function, one synthesizes a program from examples.

For example, suppose the problem is to find a program which produces certain outputs when receiving some particular inputs. If the program is too complex, it is broken down into smaller blocks with intermediate results between any two blocks. This splitting can be done either by the user or by the system automatically. The

function induction problem is then rephrased as inducing a sequence of code sections using a predefined list of program statements and example sequences of the form

$$\textit{Input} \longrightarrow \textit{result-1} \longrightarrow \cdots \longrightarrow \textit{result-k} \longrightarrow \textit{Output}.$$

One practical application of this type of induction is program optimization, in which, given a program, the problem is to find an operationally equivalent program with better performance.

Using the original program as a model, the intermediate results *result-1*, $\cdots$, *result-k* are generated by repeatedly executing that program with different input data. These intermediate values and the corresponding input-output values are then used as examples for induction. The solving algorithm will induce one block at a time until the final output value is obtained. The sequence of code sections induced is the problem's solution.

As in all function induction problems, this can be solved by any exhaustive search algorithm. However, it will be very inefficient due to the enormous size of the search space. Even so, in some circumstances the cost will be justifiable, since it is performed only once while the improved program may be used repeatedly as in the case of optimizing compiler output. However, with the equal-value search, induction time is reduced significantly, and hence more complex code sections can be optimized directly, instead of being broken down into smaller pieces.

Two code sections are said to be equal-valued if they produce identical outputs using identical inputs. Using only one input-output example, searching the program space yields a set of equal-value code sections. Each section is then tested against the remaining examples to determine its exact behavior. The first one satisfying all examples is the code section to be induced in the current step. Repeating this for subsequent program steps gives the complete program.

5.3 Inducing unknown constants

The equal-value search as proposed has one major deficiency: it cannot determine unknown constants. Any necessary constant must be included in the input arguments. While this is an obvious drawback, it is not surprising as the equal-value search is only concerned with formulating functions, not with solving for unknown values. While both are needed in practical induction systems, the latter is a different problem. For example, searching for a polynomial with known coefficients is tedious but simple, while finding unknown coefficients requires matrix algebra to solve systems of equations.

In principle, (rational) constants can be synthesized by repeated application of the four operators $+, -, \times, /$. For example, the constant 1 results from dividing any (non-zero) variable by itself. Adding 1 to itself gives 2. Rational numbers result from dividing any two integers. Negative numbers result from subtraction. However, this is not a practical way to induce unknown constants because there are an infinite number of them and one would never know which is needed in the current induction problem. Fortunately, with the help of the equal-value search, there is a better way.

5.3.1 One unknown constant

Suppose the function $f(x, y)$ requires a constant c in its formulation. In other words, f is a special case of a ternary function $g(x, y, c)$ with the third argument always constant. Let z be the value of $g(x, y, c)$, and rewrite the equation $g(x, y, c) = z$ as $g'(x, y, z) = c$, where g' is a partial inverse of g.[2] In effect, the constant c is assumed to be expressible in terms of the input arguments x, y and the output value z. Henceforth, the algorithm does not distinguish between input and output variables, but simply calls them *variables*. The problem now is to find a *functional relationship* between these variables. The solution is the function g' that has constant value for different argument sets.

[2]For example, if $g(x, y, c) = x + y/c$ then $g'(x, y, z) = y/(z - x)$.

Unfortunately, this always leads to functions that are mathematical identities, such as $x/x - y/y = 0$. Strictly speaking, $x/x - y/y$ does describe a functional relation between the variables, and thus qualifies as a solution. However, it is not a solution in the normal sense, because it satisfies infinitely many examples in addition to the given set. However, this situation can easily be detected by changing any one variable, say x, and checking the resulting value of c. If c remains constant for different values of x, then its value does not depend on the value of x. Obviously, if the function's value does not depend on that of any variable, it must be a mathematical identity and thus is not a solution to the induction problem.

Augmented with this check, the equal-value search can induce functions with one unknown constant as follows. First, it finds a function as before but treats the function output as a variable. Then, upon constructing an expression, it checks whether that expression has the same value for different choices of first example. If so, it applies the equal-value transformation and checks all resulting expressions as described above. To conserve storage and avoid unproductive search, mathematical identities are discarded and deleted from the internal database as soon as they are found. Those that remain are solutions of the induction problem. The user then chooses one that minimally describes the given examples.

In addition to inducing functions containing one unknown constant, the procedure can be used to solve another problem in machine learning, the concept clustering problem. Here, the goal is to cluster the given example set into subsets identifiable using some relations, also called *descriptors*. When the descriptors are numeric functions, the solution is a function that gives different values for different subsets of the example set. To avoid the degenerate case where each subset has only one example, the induction algorithm either sets a minimum number of examples per subset or insists on additional confirmation of the clustering, based perhaps on symbolic attributes. As an example, the latter criterion was used by ABACUS to ensure meaningful clustering [Falkenhainer 86].

5.3.2 Many unknown constants

Now consider the problem of inducing a function with several unknown constant coefficients, such as $f(x, y) = c_1x + c_2y + c_3$. This is a much more difficult problem, unless the form of $f(x, y)$ is given. Presently, there is no efficient algorithm to induce arbitrary functions with unknown constant coefficients. In BACON.6 [Langley 83], the whole problem is sidestepped by requiring that the function's form must be given as part of the input data. In COPER [Kokar 86], the Weierstrass theorem is used to justify the restriction that the expected function must be a polynomial.[3] On the other hand, ABACUS is capable of inducing functions with one unknown constant only [Falkenhainer 86].

In principle, it is not impossible to induce arbitrary functions with several unknown constants. With the use of matrix algebra, once the function's form is known, it is relatively simple to find its coefficients. Briefly, the induction algorithm in this case must first search for a function form and then try to solve for unknown coefficients in each sub-expression. Unfortunately, it is impractical as there are many possible function forms and solving systems of equations is a time-consuming process. Currently, the equal-value search does not provide an efficient way of inducing functions with multiple unknown constant coefficients. Further research is necessary as this is an important problem in function induction.

5.4 Conclusions

5.4.1 Summary

The equal-value search is a general strategy that provides a tremendous performance increase over the standard breadth-first search. The gain in performance does not come from restricting the search space, as it does in conventional approaches. Rather, it results from replacing the usual semantic equivalence relation by a weaker version—

[3]The Weierstrass theorem says that any continuous function can be approximated with any accuracy by a polynomial [Rudin 76].

the equal-value relation. In itself, this provides only a modest gain in performance. But because of the explosive nature of the search space in function induction, small gains multiply repeatedly as the search proceeds from one level of complexity to another. That explains why the equal-value search is very efficient despite its generality.

In one case, a several-thousand-fold improvement in performance was obtained. This was achieved by choosing arguments to be equal, which implies that they are treated as one by the equal-value search. While not all problems can take advantage of this trick, it nevertheless demonstrates that very impressive results can be obtained under the right circumstances.

The research presented in this thesis has thus accomplished its stated objective by developing the equal-value search, a general but fast strategy for function induction. Nevertheless, this does not mean that it cannot be improved further. On the contrary, there are several areas that have not been fully considered yet. They provide excellent opportunities for further research to refine the basic equal-value search and apply it to practical induction systems.

5.4.2 Future research

One immediate avenue for further work is to enhance the capability of the equal-value search within the domain of function induction. So far, functions have been assumed to be total in that they can be applied to any set of argument values. Although this assumption was made to simplify induction, it is rather restrictive as practical problems often have boolean expressions as the function's pre-conditions. ABACUS [Falkenhainer 86] is one recent induction system that recognizes the importance of such integration. The reason is simple: by splitting the examples into subsets with identifiable characteristics, induction becomes much easier. At present, one has to assume that such splitting is done explicitly using other means, prior to the actual task of function induction. This is unsatisfactory since it neglects the possibility that the pre-conditions themselves could be (numeric) functions. For example, the income

tax rate varies according to one's net income bracket. Consequently, to make the equal-value search more versatile, it must include the capability to induce functions with numeric pre-conditions.

Another area to explore is the possibility of inducing functions with several unknown constants. As discussed previously, this is a difficult task in general. To make it easier, one could assume that problems are numeric. This allows the use of many mathematical tools, and may encourage the development of similar tools in other domains. Even if they are difficult to develop, the research will provide a better understanding of function induction in numeric domains and so lead to an even more effective equal-value search strategy.

On the implementation side, another possibility is to improve search performance by taking advantage of its dependence on the input example. This requires developing a way of estimating the search space size and the cost of transformation, to strike the best compromise and to generate appropriate examples for induction. For non-trivial problems, it is expected that the cost of this pre-processing will be amply repaid by the resulting increase in performance. Once this is accomplished, a full-blown inductive learning system that uses the extended equal-value search could be constructed. Combined with the previously-discussed enhancements, the resulting system would be a major contribution to the field of machine learning.

Alternately, one could study the theoretical ramifications of the equal-value search strategy. In principle, exponential storage reduction can be achieved by equating all variable arguments. As the search space size of $C_n = C_0 + FC^A_{n-1}$ becomes $C_n = 1 + FC^A_{n-1}$, this results in a reduction factor of at least $C_0^{A^n}$ in addition to that obtained from accidentally equal-value expressions. Undoubtedly, a reduction of this magnitude deserves careful investigation. If such reduction also results when the equal-value search is applied to other domains, exhaustive search will enjoy a renaissance as a general problem-solving technique. Clearly, a rigorous theoretical analysis of the equal-value search is a very important contribution.

References

[Andreae 84a] P.M. Andreae. "Constraint limited generalization: Acquiring procedures from examples." *Proc American Association of Artificial Intelligence*, Austin, TX, 1984.

[Andreae 84b] P.M. Andreae. "Justified generalization: Acquiring procedures from examples." PhD dissertation, Dept. Electrical Engineering and Computer Science, MIT, 1984.

[Cohen 82] P.R. Cohen and E.A. Feigenbaum. *The handbook of artificial intelligence, Volume 3*, Morgan Kaufmann, California, 1981.

[Dietterich 81] T. Dietterich and R.S. Michalski. "Inductive learning of structural description: Evaluation criteria and comparative review of selected methods." *Artificial Intelligence*, **16**(3): 257–294, 1981.

[Falkenhainer 86] B.C. Falkenhainer and R.S. Michalski. "Integrating quantitative and qualitative discovery: The ABACUS system." *Machine Learning*, 1:367–401, 1986.

[Gaschnig 77] J. Gaschnig. "Exactly how good are heuristics?: Toward a realistic predictive theory of best-first search." *Proc Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA, 1977.

[Hart 68] T.P. Hart, N.J. Nilsson and B. Raphael. "A formal basis for the heuristic determination of minimum cost paths." *IEEE Trans. on Systems Science and Cybernetics* SSC-4, **2**:100–107.

[Knuth 73] D.E. Knuth. *The Art of computer programming, Vol 1: Fundamental algorithms*, Addison-Wesley Publishing Company, Massachusetts, 1973.

[Kokar 86] M.M. Kokar. "Determining arguments of invariant functional descriptions." *Machine Learning*, **1**:403–22, 1986.

[Korf 85] R.E. Korf. "A weak method for learning." *Artificial Intelligence*, **26**(1):35–77, 1985.

[Korf 88a] R.E. Korf. "Optimal path-finding algorithms." In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pp. 224-267, Springer-Verlag, New York, 1988.

[Korf 88b] R.E. Korf. "Search: A survey of recent results." In Howard E. Shrobe, editor, *Exploring Artificial Intelligence*, pp. 197-237, Morgan Kaufmann Publishers, California, 1988.

[Langley 83] P. Langley, G.L. Bradshaw and H.A. Simon. "Rediscovering chemistry with the BACON system." In J.G. Carbonell R.S. Michalski and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pp. 307–29, Morgan Kaufmann Publishers, California, 1983.

[Langley 86] P. Langley, J.M. Zytkow, H.A. Simon and G.L. Bradshaw. "The search for regularity: Four aspects of scientific discovery." In J.G. Carbonell, R.S. Michalski and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach, Volume II*, pp. 425–69, Morgan Kaufmann Publishers, California, 1986.

[Langley 87] P. Langley, H.A. Simon, G.L. Bradshaw, and J.M. Zytkow. *Scientific Discovery: Computational explorations of the creative processes*, MIT Press, Cambridge, Massachusetts, 1987.

[Lenat 82] D.B. Lenat. "AM: Discovery in mathematics as heuristic search." In Randal L. Davis and Douglas B. Lenat, *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill International, New York, 1982.

[Michalski 83] R.S. Michalski. "A theory and methodology of inductive learning." *Machine Learning: An Artificial Intelligence Approach*, pp. 83–134, Morgan Kaufmann Publishers, 1986.

[Mitchell 82] T.M. Mitchell. "Generalization as search." *Artificial Intelligence*, **18**:203–226, 1982.

[Newell 72] A. Newell and H. Simon. *Human problem solving*, Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[Pearl 87] J. Pearl and R.E. Korf. "Search techniques." *Annual Review of Computer Since*, **2**, California, Annual Reviews Inc, 1987.

[Preparata 85] F.P. Preparata and M.I. Shamos. *Computational geometry*, Springer-Verlag, New York, 1985.

[Purdom 83] P.W. Purdom. "Search rearrangement backtracking and polynomial average time." *Artificial Intelligence*, **21**(1):117–133, 1983.

[Purdom 85] P.W. Purdom and C.A. Brown. *The analysis of algorithms*, Holt, Rinehart and Winston, New York, 1985.

[Rudin 76] W. Rudin. *Principles of Mathematical Analysis*, McGraw-Hill, Inc., New York, 1976.

[Stickel 85] M.E. Stickel and W.M. Tyson. "An analysis of consecutively bounded depth-first search with applications in automated deduction." *Proceedings of the International Joint Conference on Artificial Intelligence*, Los Angeles, California, 1985.

[Winston 84] P.H. Winston. *Artificial Intelligence*, Addison-Wesley, Reading, Massachusetts, 1984.