To the Graduate Council:

I am submitting herewith a thesis written by Christopher Wayne Schumacher entitled "Black Box Search – Theory and Methods." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Michael D. Vose, Major Professor

We have read this dissertation
and recommend its acceptance:

Brad Vander Zanden

Accepted for the Council:

Interim Vice Provost and
Dean of The Graduate School

Black Box Search – Framework and Methods

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Chris Schumacher

December 2000

Abstract

A theoretical framework is constructed to analyze the behavior of all deterministic non-repeating search algorithms as they apply to all possible functions of a given finite domain and range. A *population table* data structure is introduced for this purpose, and many properties of the framework are discovered, including the number of deterministic non-repeating search algorithms. Canonical forms are presented for all elements of the framework, as well as methods for converting between the objects and their canonical numbers and back again. The theorems regarding population tables allow for a simple, alternate form of the No Free Lunch (NFL) theorem, an important theorem regarding search algorithm performance over all functions. Previously, this theorem has only been proven in overly-complicated, confusing fashion. Other statements of the NFL theorem are shown in the light of this framework and the theorem is extended to non-complete sets of functions and to a non-trivial definition of stochastic search. The framework allows for an extensive study of minimax distinctions between search algorithms. A change of representation is easily expressed in the framework with obvious performance implications.

The expected performance of random search with replacement, random search without replacement, and enumeration will be studied in some detail. Claims in the field regarding search algorithm robustness will be tested empirically.

Experiments were performed to determine how the compressibility of a function impacts its performance, with an emphasis on randomly selected functions. A genetic algorithm was run on two sets of functions: one set contained functions that were known to be compressible, and the other contained functions that had a high probability of being incompressible. Performance was found to be the same for both sets.

Contents

1 Introduction 1

1.1 Motivation . . . 1

1.2 Search Algorithms . . . 1

1.3 Black Box Search . . . 3

1.4 The No Free Lunch Theorem . . . 5

1.5 Analytical Methods in Black Box Search . . . 7

1.6 Compressibility of Functions . . . 9

2 Search Algorithm Framework 10

2.1 Definitions . . . 10

2.2 Population Tables . . . 14

2.2.1 Counting the Number of Search Algorithms . . . 19

2.3 Performance Tables . . . 22

2.3.1 An Example Performance Table . . . 26

2.3.2 The Number of Rows in a Performance Table . . . 29

2.3.3 Other Properties of Performance Tables . . . 31

2.4 Stochastic Search Algorithms . . . 32

2.5 Canonical Representations . . . 34

2.5.1 Functions . . . 35

2.5.2 Performance Vectors . . . 35

2.5.3 Search Paths . . . 37

2.5.4 Populations 40

2.5.5 Search Algorithms 42

2.6 The No Free Lunch Theorems 47

2.6.1 NFL Generalization 51

2.6.2 NFL Theorem for Stochastic Search 54

2.6.3 Nonuniform Function Distributions 55

2.7 Minimax Distinctions Between Algorithms 56

2.8 Other Possibilities 60

2.9 Summary of Population Table Properties 61

2.10 Summary of Performance Table Properties 61

3 Analytical Methods in Black Box Search **63**

3.1 Expected Performance of Random Search 64

3.1.1 Random Search with Replacement 65

3.1.2 Random Search without Replacement 68

3.2 Enumeration 70

3.3 An Empirical Investigation of Robustness 71

4 Representation and Black Box Search **77**

4.1 Changing a Representation 78

4.2 Change of Representation in a Population Table 79

4.3 Problem Difficulty and Representation 82

5 Compressibility of Functions and Black Box Search 84

6 Conclusions 89

6.1 Further Research 91

Bibliography 93

Appendix 96

A NFL Result with Unequally Weighted Functions 97

Vita 99

List of Tables

1 **Complete Population Table:** ($\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1\}$) For each search algorithm, the top row indicates the search paths and the bottom row indicates the corresponding performance vectors. 16

2 The number of unique non-repeating deterministic search algorithms for small values of N and M. 23

3 **Performance Table:** ($N = 3$ and $\mathcal{Y} = \{0,1\}$) This performance table corresponds to the population table in table 1. 27

4 An inconsistent "search algorithm" for the $N = 3$, $M = 2$ complete performance table. The row is inconsistent even though the row is a permutation of a true algorithm's row, and even though each column has the correct number of 1's and 0's. 28

5 Functions as strings and canonical numbers when $\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1\}$. The function $\{(x_0,a),(x_1,b),(x_2,c)\}$ is represented as string abc. 35

6 All performance vectors and their associated canonical numbers when $\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1\}$. 36

7 All non-repeating search paths and their associated canonical numbers when $\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1,2\}$. 38

8 Search path $\Longleftrightarrow$ canonical number. 38

9 All populations and their associated canonical numbers when $\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1\}$. Population $\langle(x,a),(y,b),(z,c)\rangle$ is represented as $\frac{xyz}{abc}$. . 41

10 A table listing of the search algorithm having canonical number 13,407,874 in the case where $\mathcal{X} = \mathcal{Y} = \{0,1,2,3\}$. 43

11 A breakdown of table 10. The columns are divided by population size. The second row is an ordered listing of all new x values from table 10. The third row is an ordered listing corresponding to the second row where each element indicates the index into a list of available values. Each element of the third row can be thought of as a single number of base 4, 3, and 2 respectively. The forth row lists the corresponding base 10 values of the elements of the third row. 44

12 All non-repeating search algorithms and their associated canonical numbers when $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1\}$. Compare against the population table in table 1. 46

13 The number of functions, performance vectors, non-repeating search paths, populations, and non-repeating search algorithms for various values of N and M. 48

14 Minimax table for the coin game. NDQ indicates an ordering of Nickel, Dime, Quarter. Each entry denotes the minimax advantage of the row's strategy over the column's strategy. Minimax tables are symmetric across the major diagonal with a sign change. 57

15 **Minimax Table:** ($\mathcal{X} = \{0, 1, 2\}$, $\mathcal{Y} = \{0, 1\}$) The objective function is to find the maximum value in the population. Note the non-symmetric relation $A_0 > A_8 > A_6 > A_0$. 58

16 **Minimax Table:** ($\mathcal{X} = \{0, 1, 2\}, \mathcal{Y} = \{0, 1\}$) The objective function is to find the *minimum* value in the population. Note that $A_0 > A_8 > A_5 > A_0$. 59

17 Three bit *linear sort*. 79

18 A table listing all functions and representations when $\mathcal{X} = \{x, y, z\}$ and $\mathcal{Y} = \{0, 1\}$. The row heading indicates how the domain is remapped for each representation, and the row shows the effect of the remapping on the original function space. Each element of row xyz shows $f(x)f(y)f(z)$ as a string. 81

19 An example of when the NFL result occurs when the functions are not weighted equally. 97

List of Figures

1 Graph representations of the functions in table 3. For example, $f_5 \equiv$ ((0,1), (1, 0), (2,1)). 27

2 Expected performance histograms for random search with replacement when the size of the search space is 1024 and there are n global optima. 66

3 Expected performance profiles for random search with replacement when the size of the search space is 1024 and there are n global optima. . . . 67

4 Expected performance histograms for random search without replacement when the size of the search space is 1024 and there are n global optima. 69

5 Expected performance profiles for random search without replacement when the size of the search space is 1024 and there are n global optima. 70

6 Performance profiles at various mutation rates for a randomly selected function. Profiles are shown at mutation rates of 0.05, 0.1, 0.2, 0.3, 0.4, and 0.5, with performance increasing with mutation. 72

7 **Black box histograms:** This figure shows how both a GA and hill climbing perform over 10,000 randomly selected functions. The expected distribution for enumeration is included for comparison. 74

8 Black box histograms for a GA at various mutation rates, and the black box histogram spike of random search with replacement. As the mutation increases, the GA's black box histogram more closely resembles the profile of random search with replacement. The profiles for 0.4 and 0.5 are almost indistinguishable. 75

9 Performance profiles for a GA on a randomly selected fitness function, and on the same function after a linear sort was performed as a change of representation. 80

1 Introduction

1.1 Motivation

Recently, an important theorem was introduced that outlines significant limitations to any search algorithm's performance. The importance of this so called "No Free Lunch" theorem is widely acknowledged, but its practical relevance has been frequently debated, and continues to be a source of contention. The theorem addresses search algorithm performance over *all* functions of a finite domain and range, making it relevant to the study of black box search, i.e. search over a completely arbitrary function.

Before this theorem was introduced, black box search had received only limited attention, and there is little accumulated theory on the subject. The lack of a basic theoretical framework for black box search contributes to the contention and confusion in the field, and explains in part why there has not been a clear and concise formulation of the theorem.

1.2 Search Algorithms

Consider functions from a finite domain $\mathcal{X}$ to a finite co-domain $\mathcal{Y}$. A search algorithm will be defined to be an algorithm that attempts to find a domain value that has a sufficiently high co-domain value. Simple examples include random search (with or without replacement) and enumeration.

Define a relation on the domain so that any point in the domain is related to a certain number of other points in the domain. Any two points related in this way will be called neighbors, and the set of neighbors for any point will be called its neighborhood. Define

a local search algorithm to be a search algorithm that employs a neighbor relation. A local search algorithm will typically proceed by selecting a set of points, examining the points in their neighborhoods, and based on their evaluations, determine a new set of points to continue the search. An example of this would be hill climbing, which proceeds by taking a point at random, evaluating all its neighbors, and selecting the point with the highest evaluation to continue the search; if a point has no neighbors of greater value, a new point is randomly selected from the domain.

Another example of local search is the genetic algorithm, or GA for short. Genetic algorithms are an attempt to mimic the success of biological evolution, where a population of organisms evolves to become more fit with respect to its environment. In a genetic algorithm, each element in $\mathcal{X}$ is represented as a string, and its fitness is its associated value from $\mathcal{Y}$. A random set of values from $\mathcal{X}$ is chosen as the initial population, and the elements with highest fitness are more likely to be used to create the next generation. This creation process typically involves crossover and mutation: Crossover is the process of combining two strings A and B into a new string C of the same size by picking an index in C at random and filling C with points from A before the crossover point and points from B at or after the crossover point.[1] Mutation is simply a matter of randomly altering each element in the string with (typically very small) probability. With the new population in place, the process continues, often for thousands or millions of generations. This process can be seen as a type of local search since the points in the new generation are related to the points in the previous generation. Introductions to GAs include Vose [12] and Whitley [15].

Hill climbing and genetic algorithms are two widely used probabilistic search algorithms, and many others can be seen as variations of these two (e.g. simulated annealing [9] can be seen as a variation of hill climbing where downhill moves are allowed

[1]This is called single point crossover. Multi-point crossover works similarly by alternating the copying of points from A and B at each crossover point.

with diminishing probability). One of the appeals of these two search algorithms is that they do not require sophisticated programming techniques. As long as a suitable representation and a suitable fitness function are used, a genetic algorithm or a hill climber can take it from there. Because of this, these search algorithms may seem to be a kind of silver bullet, finding optima with relatively little toil on the part of the practitioner. Unfortunately, it will be shown that these search algorithms perform well on only a small fraction of all functions.

Despite the countless variants and hybrids of hill climbing and GAs, the versions used in this dissertation are rather basic. In practice, these basic versions are often pulled off the shelf and used as a starting point for an optimization attempt, and they have a reputation as being rather all-purpose while many of the variants are considered to be more specialized. Even with a basic GA, there are a multitude of parameters that can be adjusted, allowing a certain amount of "tuning."

1.3 Black Box Search

Ideally, information about an optimization problem can be encoded into the search algorithm. For example, if the goal is to maximize airflow over a widget, one could employ the equations of fluid dynamics to direct the search towards promising areas of the search space. Another example is when a property of the function is known such as its convexity, smoothness, or modality. Black box search is commonly considered to be an attempt to optimize a function when there is no such prior knowledge about it. Another definition of black box search is search over an arbitrary function, where these functions are uniformly distributed. This is different from searching an "unknown function," since an unknown function class could be (for example) a limited and easy class of functions. Unfortunately, the two notions are often confused, as in [18]. This

dissertation will use the second definition.

Black box search is like a "black box" in the sense that the only information the function reveals is the answer to the question, "what is the value of point x?" Thus aside from its domain, the only thing that can be known about the function with certainty is the values of the points that have been evaluated. In order to perform black box optimization, a search algorithm must be able to proceed with only this evaluation history to guide it. As discussed in the previous section, both hill climbing and GAs can take on this task, as can enumeration, random search, and many of their variations.

Black box search theory received a tremendous flurry of activity with the introduction of the NFL theorem (discussed in the following section). In a black box scenario, the function to be solved could be any possible function, and thus there is a link to a search algorithm's performance across all functions and thus to the NFL theorem. This observation also allows a connection to algorithmic information theory (also known as the study of Kolmogorov complexity) with its attention towards randomness, compressibility, and "typical" input [6].

Black box optimization is an important subject of study because it presents a limiting case in terms of prior knowledge. It is also of interest because of its broad applicability: such a search algorithm can search *any* function. A large number of search algorithms can be thought of in terms of black box optimization, and the fact that they can be run "off the shelf," often with little specialization makes them very appealing. The popularity of these search algorithms is bolstered by their innumerable success stories across multiple fields. Typically, these applications are not strictly black box since domain knowledge can be coded into the representation, fitness function, and search algorithm parameters, but from that point on, the practitioner can only watch, wait, and tweak the parameters if things do not go as planned.

The next section will note that for any search algorithm, only a small fraction of all functions can be quickly optimized. Furthermore, it implies that the kind of generalization that these search algorithms aspire to must necessarily come at a performance price. In particular, the attempts to discover or accumulate regularities in the evaluation history that could be exploited to improve black box search are in vein. And yet, full of hope, people continue to apply these search algorithms to their optimization problems.

A framework will be developed in this dissertation to better understand how search algorithms behave over all functions having finite domain and co-domain. Basic data structures will be introduced and their properties will be explored. The framework will demonstrate its usefulness by allowing a straightforward proof of the No Free Lunch theorem and by facilitating an extensive study of the minimax relation which is a means of comparing search algorithm performance over all functions.

1.4 The No Free Lunch Theorem

The No Free Lunch (NFL) theorem is an attempt to address how search algorithms perform over all possible functions of a given finite domain and co-domain. Because the theorem is primarily concerned with a uniform distribution of functions, it addresses issues of black box search.

Roughly speaking, the NFL theorem states that when averaged over all functions, any two search algorithms will perform equally well, regardless of the performance measure. This implies, for example, that when averaged over all functions, a hill climber will do no better than a hill descender on a maximization problem, and that enumeration will perform at least as well as any other search algorithm. Needless to say, enumeration is very disappointing performance-wise, but in a black box scenario, this is as good as

one can expect. The implication is that one cannot simply choose a search algorithm off the shelf and expect it to do well on an unknown problem; instead one has to ensure that the search algorithm is well suited to the task. In other words, there's no "free lunch." Another way of looking at it is that there cannot exist a generalized search algorithm that will perform well on a sufficiently broad class of functions.

The NFL theorem was proven by Wolpert and Macready [18, 19] for search algorithms which sample all points in a finite search space. The original proof was for deterministic algorithms that do not revisit points already sampled, but their paper goes on to claim that the theorem also applies without these two conditions. Their proof was based largely on probability theory.

Radcliffe and Surry [7] proved the NFL Theorem in a very different way using representations and permutations, but they do not claim a proof for algorithms that revisit points (and criticize Wolpert and Macready's treatment on this subject), and their notion of stochastic search is rather weak.[2] While they claim to have provided a more accessible proof, their version has several leaps of faith and points of confusion.

Thomas English presents a proof of the NFL theorem [2] that is based largely on information theory. One of the basic assumptions of search algorithm practice is that the search algorithm "gains knowledge" of the search space as the search progresses. However, English states that without prior knowledge of the joint distribution of seen and unseen points, the optimizer cannot assume anything about the unseen points. Consequently, "an optimizer does not gain information about the objective function it optimizes, despite contrary claims in sources ranging from Goldberg's standard text to a recent survey of self-adaptive algorithms." The NFL theorem is thus shown to overturn a basic operating assumption of search algorithm practice.

[2] As will be discussed in section 2.4.

Section 2.6 will provide a much-needed straightforward proof of this important theorem from within the context of the framework developed. Instead of a complicated statistical, representational, or information theoretical approach, it will be shown how the theorem can be clearly understood almost upon inspection. The NFL statements of previous authors are shown in the light of the current framework, and an NFL generalization is given for non-complete sets of functions. A stochastic version of the NFL theorem is given for a non-trivial definition of stochastic search.

1.5 Analytical Methods in Black Box Search

Stochastic search algorithms such as hill climbers or genetic algorithms can produce dramatically different results depending on the seed of their pseudo-random number generator. In order to arrive at an estimate of expected performance for stochastic search, a large and representative sampling is needed. To arrive at an estimate of a search algorithm's expected performance on a function, the search algorithm may be run on the function many times, each time with a different seed. The results of each run can then be assembled into a measure of performance. It is assumed that the pseudo-random number generator will not introduce undue bias in this process. Similarly, in order to estimate expected performance over an ensemble of functions, one needs an unbiased sampling of functions from the set in question. Unfortunately, unbiased samplings are seldom used.

When discussing a search algorithm's performance, undue focus has been given to functions that are known to have low complexity (by various measures) [16, 17]. Even though functions from popular test suites are falling under increased criticism [10], they continue to be widely used as benchmarks. On another front, countless experiments, including very recent works [14, 11] continue to use function generators that result in

functions of great regularity. Claims are frequently made about a search algorithm's performance based solely on the performance over these relatively simple or highly concocted functions.

Define a *randomly selected function* to be a function that has been randomly and uniformly selected from the set of all functions of a given domain and co-domain. Studying randomly selected functions speaks to search in the broadest possible sense since the set of functions drawn on is maximal and unbiased. Since the NFL theorem implies that a search algorithm can only expect good performance on a small set of functions, randomly selected functions will be expected to be difficult for *any* search algorithm, and thus these functions serve as a practical and easily accessible sampling of difficult functions.

Relatively little attention has been given to randomly selected functions, but they will be used repeatedly in this dissertation. Because the performance of a search algorithm applied to a randomly selected function may vary from minimal to maximal, large numbers of them are needed before any reasonable generalizations can be made. In this dissertation, it will not be uncommon for thousands of randomly selected functions to be used in an experiment. The data can then be used to create various kinds of distribution curves.

As discussed above, commonly used function generators often produce highly regular functions of low complexity. When these generators are then used to study search algorithm performance, the results can be unclear since it can be difficult to know whether the performance effects can be generalized to larger sets of functions. In contrast, results from randomly selected functions naturally generalize to the set of all functions since the sampling is not biased. Their use even suggests a method: Run the search algorithm on a large number of randomly selected functions, and for any

property of interest, decompose the results according to that property. One can discover how that property is distributed across functions in general, and how changing it affects performance. If one is only interested in performance over a certain range of a property, one can either "filter" randomly selected functions for this property, or a function generator can make an effort to draw functions uniformly from all functions having the given property. This method will be used to study the effects of compressibility on search algorithm performance.

1.6 Compressibility of Functions

A string is said to be *compressible* if there exists an encoding that can express it in fewer bits than the string itself requires. Since functions can be expressed as a series of co-domain values, they too can be either compressible or incompressible. There is speculation that compressible functions may in general be easier for search algorithms, and as with modality, this notion has been used to try to diminish the importance of the NFL theorem. It has been argued that the functions that occur in practice are functions of high compressibility [5], as opposed to the set of all functions, which is what the NFL theorem directly addresses. Thus NFL type results may not apply to compressible functions. This assertion is put to the test by comparing search algorithm performance on a diverse set of compressible functions against search algorithm performance on a diverse set of functions where each function has a high probability of being incompressible. The performance on both sets is shown to be nearly identical, once again upholding the reach of the NFL theorem.

2 Search Algorithm Framework

This section sets forth a framework for the analysis of deterministic non-repeating search algorithms. By design, this framework is able to address issues regarding all such algorithms over all functions of a given finite domain and co-domain. The main idea involves the creation of a table where columns label all possible functions, rows label all possible deterministic non-repeating search algorithms, and each element of the table outlines the search path of the given search algorithm on the given function. Many properties of this table will be proven. It will be possible to perform analysis over all functions or over a class of functions, over all search algorithms or over a class of search algorithms. It allows analysis over a broad range of performance measures and is thus quite versatile. Canonical versions for all associated objects are provided, allowing for exhaustive experiments. Section 2.6 outlines a straightforward proof of the NFL theorem and extends the result to a strong definition of stochastic search.

2.1 Definitions

Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets, let $f : \mathcal{X} \to \mathcal{Y}$ be a function, and define y_i as $f(x_i)$. Throughout this dissertation, let $N = |\mathcal{X}|$ and $M = |\mathcal{Y}|$. Define a *population* of size m ($m \geq 0$) to be a sequence of pairs:

$$P_m \equiv \langle (x_0, y_0), (x_1, y_1), \ldots, (x_{m-1}, y_{m-1}) \rangle$$

At times the size subscript of a population will be omitted to refer to populations of arbitrary size. Let $\mathcal{P}_m$ be the set of all populations of size m, and let $\mathcal{P}$ be the set of

all populations. Adopt the following notation:

$$\begin{aligned} P_0 &= \langle\rangle \\ P_m^x &\equiv \langle x_0, x_1, \ldots, x_{m-1}\rangle \\ P_m^y &\equiv \langle y_0, y_1, \ldots, y_{m-1}\rangle \\ P_m[i] &\equiv (x_i, y_i) \\ P_m^x[i] &\equiv x_i \\ P_m^y[i] &\equiv y_i \end{aligned}$$

A concatenation operator $\|$ will be used to extend the size of a population in the following way:

$$P_m \parallel (x, y) \equiv \langle P_m[0], P_m[1], \ldots, P_m[m-1], (x, y)\rangle$$

The *prefix* $\pi_i P$ of population P is defined as the first i elements of P, i.e. $\pi_i P = \langle (x_0, y_0), \ldots, (x_{i-1}, y_{i-1})\rangle$.

Define a *non-repeating population* P to be a population with unique x components, i.e. $P^x[i] = P^x[j] \Rightarrow i = j$.[3] A *complete population* P is defined to be a population that covers the domain, i.e. for all $x \in \mathcal{X}$ there exists an i such that $P^x[i] = x$. Because a population is a sequence of ordered pairs, it corresponds to a function; if the population is complete, then the corresponding function is f. Define a permutation of a population to be any rearrangement of its pairs, and thus a permutation of a population is itself a population.

Consider a selection operator $g : \mathcal{P} \to \mathcal{X}$ which when given a population as an argument

[3]This paper will follow the convention that free variables are universally quantified.

returns a point in the search space. A *deterministic search algorithm* A corresponds to a selection operator g, and takes as arguments a population P_m and a function $f \in \mathcal{Y}^{\mathcal{X}}$ and returns the population

$$P_{m+1} = P_m \parallel (g(P_m), f \circ g(P_m)).$$

For example, the first two steps of a deterministic search algorithm A would proceed as follows:

$$\begin{aligned} A_f(P_0) &= P_0 \parallel (g(P_0), f \circ g(P_0)) = P_1 \\ A_f(P_1) &= P_1 \parallel (g(P_1), f \circ g(P_1)) = P_2 \end{aligned}$$

Deterministic search algorithms therefore operate in discrete steps where each step generates a new pair that is concatenated into the previous population to form the next population. Note that selection operator g is used to generate the x components of the population, and that function f is used to evaluate the utility of those points; this reflects the separation between "selection" (choosing a new point in the search space) and "fitness evaluation" (evaluating the utility of that new point). Multiple applications of a deterministic search algorithm will be abbreviated in the natural way, i.e. $P_m = A_f^m(P_0)$, and in particular, $A_f^0(P_0) = P_0$.

A *non-repeating search algorithm* is defined to be a search algorithm whose co-domain contains only non-repeating populations. The largest population such a search algorithm could generate is clearly a complete population which has size N, and this implies that such a search algorithm has domain elements of size less than N.

After m steps, algorithm A and function f will generate population P_m from initial population P_0. In this dissertation, search algorithms always start from the empty population P_0, which may seem to be a limitation. However, algorithms with an

arbitrary initial population size are actually special cases of algorithms that start from the empty population, as the following illustrates: Consider algorithm A and initial population P_m. A corresponds to another algorithm A' that given initial population P_0 will generate P_m after m steps, and will behave exactly as A afterwards. Designating an initial population is thus simulated by using a slightly modified algorithm that starts at P_0. In other words, algorithms that can set all points in their populations are powerful enough to encompass algorithms that cannot.

A complete population fully summarizes the behavior of a non-repeating search algorithm on a given function. Accordingly, non-repeating search algorithms A and B will be considered identical if and only if they both generate the same complete population for all $f \in \mathcal{Y}^{\mathcal{X}}$, i.e. $A_f^N(P_0) = B_f^N(P_0)$ for all $f \in \mathcal{Y}^{\mathcal{X}}$.

Define a *search path* of length m to be a sequence of m values from $\mathcal{X}$, and define a *non-repeating search path* to be a search path where no element may occur more than once. The search path associated with population P_m is P_m^x. The number of unique non-repeating search paths of length m is $\prod_{i=0}^{m-1}(N-i) = (N)_m$ since there are N ways to choose the first x value, $N-1$ ways to choose the second value, and so on.

Define a *performance vector* of length m to be a sequence of m values from $\mathcal{Y}$. The performance vector associated with population P_m is P_m^y. A performance vector and a search path can thus be said to be *derived* from a population, and a function and a search algorithm together can be said to generate a performance vector or a search path from P_0. A *complete performance vector* is a performance vector derived from a complete population, and a *complete non-repeating performance vector* is derived from a complete non-repeating population. There are $|\mathcal{Y}|^m$ unique performance vectors of length m.

Random search with replacement is defined to be a search algorithm that stochastically

chooses points in the search space based on a probability distribution, allowing for the possibility that points in the search space may be revisited. *Random search without replacement* is a search algorithm that stochastically chooses points in the search space with the restriction that points seen may not be revisited.

2.2 Population Tables

The previous section defined the selection operator of a deterministic search algorithm to be a mapping from populations to points in the search space. One could therefore represent such a selection operator as a listing of all possible populations along with the domain value that would be chosen for each of those populations. However, this turns out to be quite inefficient since a given deterministic non-repeating search algorithm will only be exposed to a tiny fraction of possible populations. On the other hand, recall from the previous section that a complete population fully captures the behavior of a deterministic non-repeating search algorithm on a given function. Because of this, a deterministic non-repeating search algorithm could be fully described by a list of M^N complete populations, one for each function. This section introduces the population table data structure, which lists the populations that are produced for each algorithm acting on each function. This is a far more efficient way of completely representing a search algorithm since it only records relevant behavior.[4]

Define a *population table of size m* to be a table where rows are labeled by all deterministic non-repeating search algorithms, and where columns are labeled by all M^N possible functions; each element of the table will contain the population generated by the corresponding algorithm (row) and function (column) after m steps. Define a *complete population table* to be a population table consisting of complete populations, i.e.

[4]Likewise, in section 2.5, a canonical form for a search algorithm is given that includes only populations that the search algorithm is exposed to.

a population table of size N. A complete population table contains complete descriptions of all deterministic non-repeating search algorithms. Table 1 shows the complete population table when $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1\}$. Population $\langle (x, a), (y, b), (z, c) \rangle$ is represented as $\begin{smallmatrix} xyz \\ abc \end{smallmatrix}$.

Let T_m represent a population table of size m. Note that each element in T_0 will be the empty population. A row in a population table will be referred to by the name of the algorithm to which it corresponds, and similarly, a column will be referred to by its corresponding function. Since populations in T_m are non-repeating, any two (x, y) pairs within a population must have different x values. Accordingly, two (x, y) pairs will be called *mutually exclusive* if they both have the same x value.

Note how the population in row A and column f of T_{i+1} corresponds to the population in the same row and column of T_i: the populations are exactly the same except for the addition of the last (x, y) pair. In other words, each population in T_m becomes a population in T_{m+1} by the concatenation of a new (x, y) pair.

Lemma 1 : In any arbitrary row of T_i $(0 \leq i < N)$, if two populations are identical, those two populations will have the same new x value in T_{i+1}.

Proof: The new value of x is determined by the search algorithm's selection operator g which accepts a population as its sole argument. Since the selection operator and the input are the same, the new value of x will be the same as well. □

Theorem 1 : No two populations within any single row of T_m may be permutations of one another unless they are equal.

Proof: by induction on m. The base case is clearly true since all elements of T_0 are

Table 1: **Complete Population Table:** ($\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1\}$) For each search algorithm, the top row indicates the search paths and the bottom row indicates the corresponding performance vectors.

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
A_0	012	012	012	012	012	012	012	012
	000	001	010	011	100	101	110	111
A_1	012	012	012	012	021	021	021	021
	000	001	010	011	100	110	101	111
A_2	021	021	021	021	012	012	012	012
	000	010	001	011	100	101	110	111
A_3	021	021	021	021	021	021	021	021
	000	010	001	011	100	110	101	111
A_4	102	102	102	102	102	102	102	102
	000	001	100	101	010	011	110	111
A_5	102	102	120	120	102	102	120	120
	000	001	100	110	010	011	101	111
A_6	120	120	102	102	120	120	102	102
	000	010	100	101	001	011	110	111
A_7	120	120	120	120	120	120	120	120
	000	010	100	110	001	011	101	111
A_8	201	201	201	201	201	201	201	201
	000	100	001	101	010	110	011	111
A_9	201	210	201	210	201	210	201	210
	000	100	001	110	010	101	011	111
A_{10}	210	201	210	201	210	201	210	201
	000	100	010	101	001	110	011	111
A_{11}	210	210	210	210	210	210	210	210
	000	100	010	110	001	101	011	111

identical. For the inductive step, assume that no two different populations in any single row of T_m are permutations of each another.

case 1: Any two populations that are the same in any row of T_m will have the same new x value in T_{m+1} (lemma 1). If they also have the same new y value, the populations are also the same in T_{m+1}. Otherwise the populations differ only in their new y value and each contains one of a mutually exclusive pair (since each x value may occur only once in a population). Thus, the two populations cannot be permutations of each other, and this also ensures that the populations cannot be extended to permutations of each other. To summarize: if any two populations in a row of T_m are the same, they will either be the same in T_{m+1} or they will be non-permutations of each other in T_{m+j} for all $0 < j \leq N - m$.

case 2: If two populations in a row of T_m are different, there must have been a T_i $(i < m)$ for which the two populations were the same and a T_{i+1} where they became different. But as shown above, from that point on, the populations contain mutually exclusive elements, and cannot be permutations in T_{i+j} for any $j > 0$. □

Consider population $P_m = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_{m-1}, y_{m-1}) \rangle$ in row A of T_m. Let F be the set of functions which label columns of T_m containing P_m in row A. In other words, F contains those functions which, together with algorithm A, generate P_m. Note how the definition of F involves the algorithm A. Let F' consist of all functions f from $\mathcal{X}$ to $\mathcal{Y}$ subject to the condition that for all $0 \leq i < m$, $f(x_i) = y_i$, where (x_i, y_i) is the ith component of P_m as defined above. Note that the definition of F' depends on the population being generated[5] (i.e. P_m) but is independent of which algorithm generates it.

[5]Permutations of the population will also result in the same F'.

Theorem 2: $F = F'$

Proof: by double containment. Assume $f \in F$. Thus A and f generate P and so $f(x_i) = y_i$ for $0 \leq i < m$, and therefore $f \in F'$. Next assume $f \in F'$. The following proof by induction on i will show that A and f generate $\pi_i P$ in T_i. For $i = m$ this becomes $f \in F$. For the base case of $i = 0$, each element of table T_0 is P_0, and thus A and f generate P_0. For the inductive step, assume that prefix $\pi_i P$ is generated by A and f in table T_i. Since P exists in row A of T_m, $\pi_i P$ exists in row A of T_i. By lemma 1, all entries in row A of T_i that equal $\pi_i P$ will have the same new x value in T_{i+1}, and since $\pi_{i+1} P$ exists in row A of T_{i+1}, the new x value in row A and column f of T_{i+1} must be x_{i+1}. By the constraints of F' it is known that $f(x_{i+1}) = y_{i+1}$, and thus A and f generate P_{i+1} in T_{i+1}. □

Theorem 2 is useful because it demonstrates that for a population P in a particular row of T_m, the columns that are labeled by P (in the sense that they contain P) are precisely those that correspond to functions that satisfy the restrictions imposed by the elements of P. Put another way, a population enumerates constraints that define the set of functions. This result would not be possible if non-trivial permutations of a population could exist in a row of a population table (theorem 1).

Theorem 3: For any arbitrary population P_m, let S be the set of rows in T_m which contain P_m. Every row in S has P_m in exactly the same columns.

Proof: Consider any two rows A and B which both contain P_m. Let F_x denote the columns containing P_m in row x. Theorem 2 demands that $F_A = F' = F_B$. □

Because the set F' is the same whether it was defined from P_m or from a permutation of P_m, the following corollary arises:

Corollary 1: Permutations of P_m can only occur in columns containing P_m.

These theorems explain how rows are related with respect to a given population and its permutations. In particular, if population P_A occupies columns F in row A, then a non-trivial permutation of P_A can only occupy the exact same columns, and only in a row different from A.

Theorem 4: All possible non-repeating populations of size m exist in T_m. More precisely, if X is an arbitrary non-repeating search path and Y is an arbitrary performance vector, there exists population P in T_m such that $P^x = X$ and $P^y = Y$.

Proof: To show that any arbitrary non-repeating population P_m exists in T_m, induction will be used to show that $\pi_i P$ exists in T_i. For the base case, $P_0 = \pi_0 P$ exists in T_0. For the inductive step, assume that $\pi_i P$ exists in columns F of table T_i. A search algorithm in this table is restricted only in that x values must be non-repeating, and thus there exists a search algorithm A that chooses x_{i+1} for the new value of x in T_{i+1}. By theorem 2, F consists of *all* functions f from $\mathcal{X}$ to $\mathcal{Y}$ subject to the condition that for all $0 \leq j < i$, $f(x_j) = y_j$. F therefore contains functions with all possible y values for x_{i+1}, and so there exists an $f \in F$ for which $f(x_{i+1}) = y_{i+1}$. Therefore A and f generate P_{i+1}. □

2.2.1 Counting the Number of Search Algorithms

To determine the number of unique rows in a population table, consider first the number of unique populations in a row. As will be shown, this number is independent of the algorithm (row). Let R_m be the set of populations in a row of T_m. Recall that each element of T_0 is the empty population, and thus $R_0 = \{\langle\rangle\}$.

Theorem 5: $|R_m| = M^m$

Proof: Restrict attention to a given row A. Once again, consider the transition from T_m to T_{m+1}. Let $P \in R_m$ determine the set of columns F, i.e. F contains those functions which together with A generate P. In row A of T_{m+1}, all columns which are labeled by F will have the same new x value (lemma 1). By theorem 2 there are no constraints on what the new y value may be and so all M values will occur across the functions of F. It is clear that two different populations in R_m could not correspond to the same population in R_{m+1}, and thus for every element P in R_m there are $|\mathcal{Y}|$ unique elements in R_{m+1}, i.e. $|R_{m+1}| = |R_m| \cdot |\mathcal{Y}|$. Combining this with the fact that $|R_0| = 1$ yields the desired result. □

Note that $|R_m|$ is independent of the row. At this point a number of population table properties can be established.

Corollary 2:

1. The rows in a population table will contain distinct elements if and only if the table is complete.
2. Any particular population may only exist in a single column of a complete population table.
3. There are $N!$ different populations within a column of a complete population table.
4. All possible search paths exist in each column of a complete population table.

Proof: The first item follows from the fact that in a complete population table $m = N$ and thus $|R_m| = M^N$, which is also the number of columns in the table. The second

item is a consequence of the first item and theorem 3. The third item is shown to be true as follows: a complete population completely and uniquely defines the function labeling its column, and since by theorem 4 all possible populations exist in the table, all $N!$ possible permutations of the population must exist in the column. The forth item is also a consequence of theorem 4. □

Let S_m represent the set of rows in the table T_m. Each row in S_m will correspond to several rows in S_{m+1}, in that there will be rows in S_{m+1} that are the same as a row in S_m except for the addition of a final (x, y) pair to each population. It is clear that two different rows in S_m could not correspond to the same row in S_{m+1}, and thus determining how many rows in S_{m+1} correspond to each row in S_m provides a way of counting the number of unique rows in T_m.

Theorem 6: $|S_{m+1}| = |S_m|(N-m)^{M^m}$

Proof: An element of S_m will contain $|R_m|$ unique populations, and for each of them there are $(N-m)$ possibilities for a new value of x. There are thus $(N-m)^{|R_m|}$ different ways to add that last point to each population in a row, i.e. $|S_{m+1}| = |S_m|(N-m)^{|R_m|}$. An application of theorem 5 completes the proof. □

To determine $|S_N|$, i.e. the number of unique algorithms in a complete population table, note that all elements in T_0 are P_0, and so $|S_0| = 1$. Combining this fact with theorem 6 provides one of the major results of this dissertation.

Algorithm Count Result 1: The number of unique deterministic non-repeating search algorithms for a given $\mathcal{X}$ and $\mathcal{Y}$ is

$$\prod_{i=0}^{N-1} (N-i)^{M^i}$$

Table 2 displays the number of unique deterministic non-repeating search algorithms for small values of N and M, showing how quickly this value skyrockets. Notice that when $N = 2$, the number of algorithms is always 2 since the only two possible algorithms are to either pick x_0 first or to pick x_1 first. If the population size is not limited (i.e. if repeating search algorithms are allowed) it is easy to see that the number of search algorithms would be infinite. Section 2.5 outlines a different way of counting search algorithms for the purpose of easily assigning a number to each search algorithm, and vice-versa.

2.3 Performance Tables

A performance vector can be used in various ways to measure the performance of search algorithm A on function f. The term *performance vector measure* will be used to denote a measure that can be derived from a performance vector. Example performance vector measures include:

- the number of times the optimum is obtained by evaluation n
- the position i in the performance vector having highest value by evaluation n

Define a *complete performance vector measure* to be a performance vector measure that only applies to complete performance vectors. This type of measure takes full account of the performance of A on f and is able to more directly address issues of convergence. Examples include:

- the number of evaluations before the optimum is reached
- the number of evaluations before the best n elements are found

Table 2: The number of unique non-repeating deterministic search algorithms for small values of N and M.

N	M	number of unique search algorithms
2	2	2
2	3	2
2	4	2
2	5	2
2	6	2
2	7	2
3	2	12
3	3	24
3	4	48
3	5	96
3	6	192
3	7	384
4	2	576
4	3	55296
4	4	$\approx$ 2.1 e7
4	5	$\approx$ 3.3 e10
4	6	$\approx$ 2.0 e14
4	7	$\approx$ 4.9 e18
5	2	1658880
5	3	$\approx$ 8.4 e14
5	4	$\approx$ 1.0 e30
5	5	$\approx$ 1.8 e50
5	6	$\approx$ 3.2 e86
5	7	$\approx$ 3.5 e131
6	2	$\approx$ 1.7 e13
6	3	$\approx$ 3.6 e45
6	4	$\approx$ 6.4 e120
6	5	$\approx$ 1.3 e267

- the number of evaluations until error tolerance is achieved

Note that *the number of local optima in a search space* could not be a performance vector measure because it requires more information than a performance vector can provide. Generally speaking, a property that can be derived from a function without reference to a performance vector will not be considered to be a suitable performance vector measure. Furthermore, such a measure would be algorithm invariant for any single function, and therefore could not be used to compare the performance of search algorithms (although it could be used as a measure of function difficulty). Note that *the number of times the global optima occurs* is a suitable performance vector measure but that it is not a suitable complete performance vector measure.

In the original proof of the NFL theorem [18, 19], Wolpert and Macready propose using a histogram as the basis for performance rather than a performance vector. This histogram would chart the number of times each co-domain element occurs in a performance vector, and a performance measure would then be derived from that histogram. Such a measure tosses out all time information, e.g. it is not possible to compute the number of evaluations before the optimum is reached from such a histogram. Furthermore, this type of histogram is algorithm independent as a complete performance vector measure, which is curious since the goal of their NFL proof was to show algorithm independence. Fortunately, their proof employs (strong) performance vectors in the service of these (weak) histograms.

Define a *performance table* of size m to be a table where rows and columns are labeled just as with a population table, but where each element of the table is the corresponding performance vector of length m. A complete performance table contains performance vectors of length N. A population table of size m is said to *correspond* to a performance table of size m, since for every element P_m in a population table, there is P_m^y in the

corresponding row and column of the performance table.

Theorem 7 No row of a complete performance table contains an element more than once.

Proof: Assume that functions f and f' generate the same complete performance vector for a given search algorithm A. It will be shown that $f = f'$. The assumption allows a proof by induction that the populations generated would be the same, i.e. for $0 \leq i \leq N$, $A_f^i(P_0) = A_{f'}^i(P_0)$. The base case $i = 0$ is trivial to verify: $A_f^0(P_0) = P_0 = A_{f'}^0(P_0)$. The following two equations are used for the inductive step:

$$\begin{aligned} A_f(A_f^i(P_0)) &= A_f^i(P_0) \parallel (x, f(x)) \\ A_{f'}(A_{f'}^i(P_0)) &= A_{f'}^i(P_0) \parallel (x', f'(x')) \end{aligned}$$

where $x = g(A_f^i(P_0))$ and $x' = g(A_{f'}^i(P_0))$. By the inductive hypothesis, $A_f^i(P_0) = A_{f'}^i(P_0)$, and so by lemma 1, $x = x'$. Since by assumption the two performance vectors are the same, $f(x) = f'(x')$, thus completing the inductive proof. Since the complete populations are the same (i.e. $A_f^N(P_0) = A_{f'}^N(P_0)$), their corresponding functions are the same, i.e. $f = f'$. □

Note that theorem 7 implies that for a given algorithm (row in the table), a complete performance vector determines the function (column of the table) to which the algorithm was applied. As will be shown below, the converse is not true since a single column can have duplicate entries. Theorem 7 demands that for any search algorithm in the table, performance must be different for each function.[6]

[6]Some performance vector measures may not notice that each performance vector is different.

Theorem 8 Any two rows in a complete performance table are either the same or are permutations of each other.

Proof: The number of distinct complete performance vectors is M^N (since there are M ways to choose each of the N elements of a performance vector), and this is also the number of entries in a row of the table. By theorem 7, all of the performance vectors in a row are unique, and thus each possible performance vector occurs exactly once in each row. □

2.3.1 An Example Performance Table

When $N = 3$ and $M = 2$ there are eight possible functions, and (as was shown in section 2.2.1) 12 possible deterministic search algorithms. For simplicity, the two possible y values are represented as zero and one, and the performance vectors are represented by the ordered string of their values.

Several things become clear on first inspection. Table 3 obeys theorems 7 and 8. Two of the columns are constant, and each element in a column is a permutation of other elements in the same column. Each row is unique (as will be proven in section 2.3.2).

One way to understand the occurrence of these patterns is to observe a graphical representation of each function as is done in figure 1. Without loss of generality, each function has been assigned so as to correspond to its standard binary representation, e.g. the graph of f_5 can be seen to be a representation of the binary number 5 (101). By definition, a performance vector for a given function is the ordered y values obtained during the search, and thus a complete non-repeating performance vector must be a permutation of that function's representation.

Table 3: **Performance Table:** ($N = 3$ and $\mathcal{Y} = \{0,1\}$) This performance table corresponds to the population table in table 1.

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
A_0	000	001	010	011	100	101	110	111
A_1	000	001	010	011	100	110	101	111
A_2	000	010	001	011	100	101	110	111
A_3	000	010	001	011	100	110	101	111
A_4	000	001	100	101	010	011	110	111
A_5	000	001	100	110	010	011	101	111
A_6	000	010	100	101	001	011	110	111
A_7	000	010	100	110	001	011	101	111
A_8	000	100	001	101	010	110	011	111
A_9	000	100	001	110	010	101	011	111
A_{10}	000	100	010	101	001	110	011	111
A_{11}	000	100	010	110	001	101	011	111

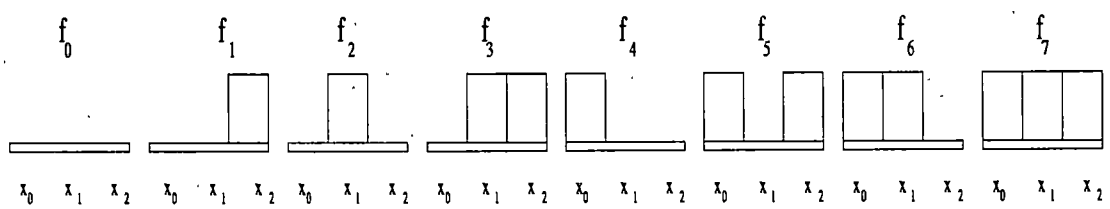


Figure 1: Graph representations of the functions in table 3. For example, $f_5 \equiv$ ((0,1), (1, 0), (2,1)).

The proof of theorem 8 spoke of how the number of distinct complete performance vectors is the same as the number of possible functions. This correspondence might have seemed odd at first glance, but by viewing the function as a graph or as a string of numbers, it becomes clear how a performance vector is a permutation of a function's representation. A list of all possible functions and a list of all possible performance vectors would not only have the same size, they would in fact be the same list in a (possibly) different order. Throughout the rest of this dissertation, a performance vector or a fitness function may be referred to as a list of numbers or even as a single base M number with N digits.

Because each function (column) of a performance table contains performance vectors that are permutations of the same "number," it is clear that not all permutations of a row correspond to valid algorithms. It is the construction process outlined in section 2.3 that imposes such constraints on resulting algorithms. Another restriction on the number of ways a row may be permuted is the fact that a search algorithm always picks the same initial x value. For example, the "search algorithm" A_{false} in table 4 cannot be a deterministic non-repeating search algorithm because it does not always choose the same initial x value, as demonstrated below.

To see the inconsistency of A_{false}, notice that the third performance vector (100) finds the value of 1 on the first trial. This performance vector is based on f_2, so A_{false} must have chosen x_1 first to get that performance vector (refer to figure 1). But this is

Table 4: An inconsistent "search algorithm" for the $N = 3$, $M = 2$ complete performance table. The row is inconsistent even though the row is a permutation of a true algorithm's row, and even though each column has the correct number of 1's and 0's.

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
A_{false}	000	001	100	011	010	101	110	111

inconsistent with the performance vector under f_3, since $f_3(x_1) = 1$ whereas the first value in its performance vector is 0. A_{false} therefore could not have chosen x_1 first in all columns.

2.3.2 The Number of Rows in a Performance Table

This section will parallel section 2.2.1 by proving the corresponding theorems for performance tables. Let T_m be a population table of size m and let Q_m be the corresponding performance table of size m. Let R_m be the set of populations in row A of T_m, and let V_m be the set of performance vectors for the corresponding row A of Q_m.

Theorem 9: $|V_m| = M^m$

Proof: This proof proceeds by first demonstrating by induction that $|V_m| = |R_m|$. The base case is proven by noting that $|V_0| = |R_0| = 1$. Two different populations in R_n could not correspond to the same population in R_{n+1} since those populations would have different prefixes, and similarly for the transition from V_n to V_{n+1}. Therefore, focus shifts to the populations in row A that are the same. As explained in the proof of theorem 5, these populations will have the same new x value in T_{n+1} and will take on the full range of y values, and thus $|R_{n+1}| = |R_n|M$ and similarly $|V_{n+1}| = |V_n|M$. Because $|R_n| = |V_n|$ (induction hypothesis), the induction is complete. Theorem 5 can be applied to achieve the desired result. □

As before, let S_m represent the set of rows in the table T_m, and let W_m represent the set of rows in the corresponding table Q_m. As with S_m, each row in W_m will correspond to several rows in W_{m+1}, in that there will be rows in W_{m+1} that are the same as a row in W_m except for the addition of a final y value to each performance vector. Determining

how many rows in W_{m+1} correspond to each row in W_m provides a way of counting the number of unique rows in Q_m.

Theorem 10: $|W_{m+1}| = |W_m|(N-m)^{M^m}$

Proof: It will be shown that for every unique row $A_s \in S_m$, there is exactly one corresponding row $A_w \in W_m$. The base case is clearly true since both T_0 and Q_0 contain the empty set at all entries. It is also clear that $|W_n| \leq |S_n|$ since removing the x values in T_n cannot result in a larger number of unique rows in Q_n. Thus as T_n and Q_n transition to T_{n+1} and Q_{n+1} respectively, the idea is to determine if *fewer* rows are possible in Q_{n+1}. It must be shown that removing the x values in the populations of T_{m+1} will not result in duplicate corresponding rows in Q_{m+1}.

From the inductive hypothesis, there is a one-to-one correspondence between the rows of T_n and Q_n. Consider population P in row A of T_n. Let F be the set of functions which together with A generate P in T_n. As discussed in section 2.2.1, row A in T_n will correspond to several rows in T_{n+1}. Let two distinct rows A_T and B_T in T_{n+1} both correspond to row A in T_n.

In order for two distinct rows A_T and B_T in T_{n+1} to correspond to two identical rows A_Q and B_Q in Q_{n+1}, the y values in A_T and B_T would all have to agree with the y values in A_Q and B_Q, and thus the only way for A_T and B_T to be distinct is if they differ in only the last x value. This is shown to be impossible as follows: As explained in the proof of theorem 6, a search algorithm must make an x value choice for $P \in R_n$. Assume that search algorithm A_T chooses x_a for the population P and that algorithm B_T chooses x_b ($x_b \neq x_a$) for the same population. For rows A_Q and B_Q to be equal, the new y value in each population of row A_T must be the same as each new y value in row B_T. By theorem 2, F contains all functions with only the restrictions of P, and thus F

contains functions for which $f(x_a) \neq f(x_b)$, creating a contradiction. This completes the induction and so shows a one-to-one correspondence between the number of rows in a population table and the number of rows in the corresponding performance table. Theorems 6 and 9 can be applied to complete the proof. □

Algorithm Count Result 2: The number of unique rows in a complete performance table, i.e. $|W_N|$, is the same as the number of unique rows in a population table:

$$\prod_{i=0}^{N-1} (N-i)^{M^i}$$

A search algorithm can therefore be completely and uniquely expressed by the complete performance vectors it generates over all functions. Because of this result, theorem 8 can be strengthened as follows:

Theorem 11: Any two rows in a complete performance table are permutations of one another.

Proof: Since a complete population table has unique rows, and since the number of unique rows is the same in both a population table and a performance table, a complete performance table will also have unique rows. □

2.3.3 Other Properties of Performance Tables

Each column in a complete performance table is unique in its ordering, which follows from the fact that every element in a row is unique (theorem 7) and thus any two columns will be different at each position. Section 2.2.1 showed how all permutations of a population exist in a column of a population table, and thus all permutations of a

performance vector must occur in a column of a performance table. Each performance vector in a column is a permutation of the others (a consequence of corollary 2). This implies that for any performance measure based solely on the position of the first optimum, the expected performance for an unknown search algorithm on a function would depend only on the number of maxima in the co-domain of that function.

There are $N!$ different enumerations of the search space which in many cases will be a smaller number than the number of performance vectors in a row of a complete performance table (M^N). Using Stirling's formula to approximate factorial, the second number is found to be $O((eM/N)^N/\sqrt{N})$ times larger than the first number. This implies that a search algorithm will often use the same enumeration of the state space for different functions. Indeed, some algorithms will use the same enumeration of state space for *all* functions. Even with a static enumeration, the resulting complete performance vectors will be distinct because of the different evaluation functions used.

Section 2.3.1 demonstrated that not all permutations of a row exist in a performance table. Algorithm Count Result 2 makes it possible to determine the percentage of permutations that do exist in a row. There are $M^N!$ permutations of a row for a complete population table, a number which vastly exceeds Algorithm Count Result 2. Therefore only a very small percentage of row permutations exist in a complete performance table.[7]

2.4 Stochastic Search Algorithms

A population table contains all deterministic non-repeating search algorithms, but many search algorithms such as GAs or hill climbers have a significant stochastic component. This section will discuss how the population table framework can be related

[7]Even 999! has over two thousand digits. Compare to row $N = M = 5$ in table 13.

to stochastic algorithms.

Randomness should not simply be swept under the rug (as is so often the case) by claiming that any algorithm implemented on a computer must be deterministic; simulated randomness can be improved arbitrarily, and true random input sources are available that make it possible to have true random algorithms on computers.

In the NFL proof given by Radcliffe and Surry [7], A *stochastic search algorithm* is defined to be a search algorithm that uses a deterministic "random" number generator to simulate random choices. Similarly, Whitley et. al. [17, 8] discuss "stochastic" algorithms as deterministic algorithms coupled with a *specific* corresponding random seed. Because they assume the seed remains constant, they are still referring to deterministic algorithms, and thus the above theorems hold for this weak notion of stochastic search. To make their notion more straightforward, consider how it relates to a population table. A population table captures the behavior of all possible deterministic search algorithms. By using a fixed seed with a pseudo-random number generator, the "stochastic" algorithm will behave exactly as a row of a population table, with the seed determining the row (as described above).

In discussing the stochastic case, Wolpert and Macready [19] introduce "stochastic" algorithm σ, and claim "One can now reproduce the derivation of the NFL result for deterministic algorithms, only with a replaced by σ throughout. In so doing, all steps in the proof remain valid." However, the situation is not nearly so transparent as to warrant the omission. Furthermore, the description of stochastic search is not complete (do they mean stochastic in the same sense as Radcliffe and Surry? some other way?), and the interpretation of the theorems in terms of probabilities can be confusing. For example, in summing probabilities in their theorem statement, they arrive at a number that is not itself a probability (indeed, it can exceed the value of one). Is it possible that

two stochastic search algorithms may perform differently but are not expected to do so? If their notion of stochastic is weak, then their use of probability is inappropriate, and if their notion of stochastic is stronger, they fail to elucidate what it is, their conclusions in the stochastic case are unclear, and their lack of discussion is suspect.

For the remainder of this dissertation, a *stochastic search algorithm* will be defined as a vector λ having an element for each deterministic search algorithm in a population table, where each element $\lambda(A_i)$ indicates the probability that the stochastic search algorithm will behave as the deterministic search algorithm A_i. It will be shown in section 2.6.2 how this definition allows a version of the No Free Lunch theorem for stochastic search.

For any given run, a stochastic search algorithm will behave *exactly* like a deterministic search algorithm, and thus the vector λ provides a connection between stochastic and deterministic search. Furthermore, a population table fully captures the behavior of a stochastic search algorithm for any given run.

2.5 Canonical Representations

Objects such as populations and search algorithms can be immense and unwieldy. A canonical representation allows for a short but complete naming system that also has the benefit of a natural ordering. Such a strict ordering makes it easy to create tables (such as performance tables), step through all objects for exhaustive experiments, and to randomly select objects. This section will describe canonical representations for many of the objects that have been previously discussed. When rules are given for an ordering, they are presented in order of precedence.

Table 5: Functions as strings and canonical numbers when $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1\}$. The function $\{(x_0, a), (x_1, b), (x_2, c)\}$ is represented as string abc.

canonical	function string
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

2.5.1 Functions

To put functions in canonical form, first assume that sets $\mathcal{X}$ and $\mathcal{Y}$ are ordered. Since the domain has an order, a function can be represented as a string of length N where each position is a base M number, i.e. the function $\{(x_0, a), (x_1, b), (x_2, c)\}$ will be represented as string abc. To effect an ordering on functions, leftmost places in the string are more significant. In this way, a function can be described in at most $O(\log M^N)$ bits. Table 5 shows the canonical and string representations for all functions when $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1\}$.

2.5.2 Performance Vectors

A performance vector can have length from 1 to N. The number of performance vectors of length n is M^n, and the total number of performance vectors is $\sum_{n=1}^{N} M^n$. To achieve an ordering, the following two rules suffice: 1) performance vectors of length n will occur before those of length $n+1$, 2) within a given length, leftmost places of the performance vector will be most significant. Table 6 shows all performance vectors and

Table 6: All performance vectors and their associated canonical numbers when $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1\}$.

canonical	performance vector
0	0
1	1
2	00
3	01
4	10
5	11
6	000
7	001
8	010
9	011
10	100
11	101
12	110
13	111

their corresponding canonical numbers for the case when $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1\}$.

When $N = M = 4$, the canonical number associated with performance vector $\langle 2, 3, 2, 2 \rangle$ can be computed as follows: the performance vector has length 4, and the index of the first performance vector with length 4 is $\sum_{n=1}^{3} M^n = 84$. Now one only needs to compute the offset of this performance vector into performance vectors of size 4. This is simply

$$2 \cdot 4^3 + 3 \cdot 4^2 + 2 \cdot 4^1 + 2 = 186$$

Adding this offset to the offset for length gives the canonical number of 270. To go from the canonical number of 270 to the corresponding performance vector, first determine the length of the performance vector. Because 270 is greater than $\sum_{n=1}^{3} M^n = 84$, the length must be four, and thus $270 - 84 = 186$ will be the offset into length 4

performance vectors. Modulus arithmetic can be performed as follows:

$$\begin{aligned} 186/(4^3) &= 2 \text{ Remainder } 58 \\ 58/(4^2) &= 3 \text{ Remainder } 10 \\ 10/(4^1) &= 2 \text{ Remainder } 2 \end{aligned}$$

Which yields the numbers 2, 3, 2, 2 for performance vector $\langle 2,3,2,2 \rangle$.

2.5.3 Search Paths

A non-repeating search path can have length from 1 to N. There are $(N)_n$ non-repeating search paths of length n, and thus $\sum_{n=1}^{N} (N)_n$ non-repeating search paths. The number of search paths is thus independent of M. The ordering rules are basically the same as for performance vectors: 1) search paths of length n will come before search paths of length $n+1$, 2) leftmost places in the search path are considered more significant. Table 7 shows all non-repeating search paths and their canonical numbers in the case when $\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1,2\}$.

When $N = 5$, search path $\langle 1,3,2,0 \rangle$ will have canonical number 123, as will now be shown. The length of this search path is 4, and the index of the first search path with length 4 is $\sum_{n=1}^{3} (N)_n = 85$. Search paths are non-repeating, so there will be N choices for the first position, $N-1$ choices for the second, and so on. Therefore, the value at position i can be represented as a base $N-i$ digit, allowing for a maximally compact encoding. This can be done by converting each search path value to an index into a list of unused values. Refer to table 8. The list of unused values starts off as $[0,1,2,3,4]$. The first position value in the search path is 1, and its position index in

Table 7: All non-repeating search paths and their associated canonical numbers when $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1, 2\}$.

canonical	search path
0	0
1	1
2	2
3	01
4	02
5	10
6	12
7	20
8	21
9	012
10	021
11	102
12	120
13	201
14	210

Table 8: Search path $\Longleftrightarrow$ canonical number.

position value	1	3	2	0
position index	1	2	1	0
choices at position	5	4	3	2

the list of unused values is also 1 (since indexing starts at zero). Once 1 is removed, the list of unused values is $[0, 2, 3, 4]$. The next search path position value is 3, which occupies position index 2 in the list of unused values. Once 3 is removed, that list will be $[0, 2, 4]$. The next position value is 2 which occupies position index 1, and when the list is $[0, 4]$, 0 occupies position index 0, and this completes the second row of table 8.

The position indices can be combined into a single number that provides the offset of this search path into length 4 search paths. Note from the third row of table 8 that there are 2 possible values for the last position, $3 \cdot 2$ possible values for the last two

positions, and so on, leading to the following equation for this offset:

$$1 \cdot (4 \cdot 3 \cdot 2) + 2 \cdot (3 \cdot 2) + 1 \cdot 2 + 0 = 38$$

The canonical number for this search path is thus $85 + 38 = 123$.

To derive a search path given $N = 5$ and canonical number 123, the first step is to determine the length of the search path. Because $\sum_{n=1}^{3} (N)_n \leq 123 < \sum_{n=1}^{4} (N)_n$, the search path length must be 4. Accordingly one can subtract $\sum_{n=1}^{3} (N)_n$ in order to get the offset into search paths of length 4, i.e. $123 - 85 = 38$. One can now do modulus arithmetic as follows:

$$\begin{aligned} 38/(4 \cdot 3 \cdot 2) &= 1 \text{ Remainder } 14 \\ 14/(3 \cdot 2) &= 2 \text{ Remainder } 2 \\ 2/2 &= 1 \text{ Remainder } 0 \end{aligned}$$

This results in numbers 1, 2, 1, 0, which are the position index values from the second row of table 8. A list of unused values will again be used, but this time one will go from index to value. The list of unused values is initialized to $[0, 1, 2, 3, 4]$. The first index is 1, which has value 1, and thus the first position value of the search path is 1. The list is then $[0, 2, 3, 4]$, and the next index value is 2, which corresponds to value 3 in the list. The list then becomes $[0, 2, 4]$, and the next index value of 1 will correspond to value 2 in the list. When the list is finally $[0, 4]$, and the last index of 0 results in the final position of the search path being 0. The search path is thus $\langle 1, 3, 2, 0 \rangle$.

2.5.4 Populations

A population may have length from 1 to N, and consists of a search path and a performance vector. Theorem 4 implies that a population can be created from any non-repeating search path together with any possible performance vector, and thus the number of populations of length n is $(N)_n M^n$ and the total number of populations is $\sum_{n=1}^{N} (N)_n M^n$. The ordering relies on the following rules: 1) populations of length n will come before populations of length $n+1$, 2) populations will be ordered by their search paths, and 3) populations will be ordered by their performance vectors. Table 9 shows all populations for $\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1\}$.

The canonical number of a population can be computed by combining the canonical numbers of the search path and performance vector that comprise the population. For example, when $N = M = 4$, the population $\langle (2,1),(1,3),(0,1),(3,2) \rangle$ can be expressed as canonical value 5446 as follows: The length of the population is 4, its search path is $\langle 2,1,0,3 \rangle$ (which has canonical value 54) and its performance vector is $\langle 1,3,1,2 \rangle$ (which has canonical value 202). The first population of length 4 has the canonical value of $\sum_{i=1}^{3} M^n \ (N)_n = 1744$. The first search path of length 4 has canonical value of 14, and the first performance vector of length 4 has canonical value of 118 (see previous sections). There are 4^4 possible performance vectors of length 4, and thus the canonical number is the result of $14 \cdot 4^4 + 118 + 1744 = 5446$.

The conversion back to a population involves first determining that the length must be 4 since the canonical value is greater than $\sum_{i=1}^{3} M^n \ (N)_n = 1744$. The offset into populations of size 4 would therefore be $5446 - 1744 = 3702$. Integer division can then be performed to give the indices into a length 4 search path and a length 4 performance vector:

$$3702/(4^3) = 14 \text{ Remainder } 118$$

Table 9: All populations and their associated canonical numbers when $\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1\}$. Population $\langle (x,a),(y,b),(z,c)\rangle$ is represented as $\frac{xyz}{abc}$.

	pop		pop		pop		pop		pop
0	0 0	16	10 10	32	012 010	48	102 010	64	201 010
1	0 1	17	10 11	33	012 011	49	102 011	65	201 011
2	1 0	18	12 00	34	012 100	50	102 100	66	201 100
3	1 1	19	12 01	35	012 101	51	102 101	67	201 101
4	2 0	20	12 10	36	012 110	52	102 110	68	201 110
5	2 1	21	12 11	37	012 111	53	102 111	69	201 111
6	01 00	22	20 00	38	021 000	54	120 000	70	210 000
7	01 01	23	20 01	39	021 001	55	120 001	71	210 001
8	01 10	24	20 10	40	021 010	56	120 010	72	210 010
9	01 11	25	20 11	41	021 011	57	120 011	73	210 011
10	02 00	26	21 00	42	021 100	58	120 100	74	210 100
11	02 01	27	21 01	43	021 101	59	120 101	75	210 101
12	02 10	28	21 10	44	021 110	60	120 110	76	210 110
13	02 11	29	21 11	45	021 111	61	120 111	77	210 111
14	10 00	30	012 000	46	102 000	62	201 000		
15	10 01	31	012 001	47	102 001	63	201 001		

This yields 14 for the search path and 118 for the performance vector. From there one may create the search path and performance vector as described in previous sections.

2.5.5 Search Algorithms

As described in section 2.1, a search algorithm consists of a selection operator $g : \mathcal{P} \rightarrow \mathcal{X}$, i.e. a mapping from populations to points in the domain. What follows is a means of representing this mapping as a table that will allow a maximally compact representation. Table 10 shows a search algorithm represented in this way, and can serve as an example for the following discussion. A deterministic non-repeating search algorithm is limited in the populations it can generate, and a maximally compact representation should only encode those populations. A table's first entry is the empty population and the search algorithm's first choice from $\mathcal{X}$. From that first choice, populations are created with all M possible values from $\mathcal{Y}$, and these populations comprise the next M populations in the table (following the ordering of set $\mathcal{Y}$). The table proceeds as follows: take each population of length n as it occurs in the table, and use it and its associated new choice from $\mathcal{X}$ to create M new populations of length $n+1$. These new populations are entered into the table along with their associated new domain values. The table has M^n populations of length n, for a total of $\sum_{n=0}^{N-2} M^n$ entries.[8]

This table representation allows a means of computing the number of non-repeating search algorithms: since there are M^n populations of length n in the table, and since there are $N-n$ choices for a new x value at each such entry, there must be $\prod_{n=0}^{N-2}(N-n)^{M^n}$ non-repeating search algorithms, which is in accord with Search Algorithm Count 1.

[8]The table need not include populations of size $N-1$ since the search algorithm is non-repeating and thus has no choice in the final entry.

Table 10: A table listing of the search algorithm having canonical number 13,407,874 in the case where $\mathcal{X} = \mathcal{Y} = \{0, 1, 2, 3\}$.

population	new x	population	new x
$\langle\rangle$	2	21 12	3
2 0	1	21 13	0
2 1	1	23 20	1
2 2	3	23 21	0
2 3	0	23 22	0
21 00	3	23 23	0
21 01	0	20 30	1
21 02	0	20 31	1
21 03	3	20 32	3
21 10	0	20 33	1
21 11	3		

Table 11: A breakdown of table 10. The columns are divided by population size. The second row is an ordered listing of all new x values from table 10. The third row is an ordered listing corresponding to the second row where each element indicates the index into a list of available values. Each element of the third row can be thought of as a single number of base 4, 3, and 2 respectively. The forth row lists the corresponding base 10 values of the elements of the third row.

n = pop size	0	1	2
new x values	2	1130	3003033010001131
new x indices	2	1120	1001011010000010
base 10 values	2	42	38530

Because size n populations can be built up from size $n-1$ populations already in the table, and since the null population always starts the table, a non-repeating search algorithm can be represented by a listing of new values from $\mathcal{X}$ alone. Table 11 shows such a listing. Furthermore, since this is a non-repeating search algorithm, this listing can be transformed into another listing where each entry is an index into a list of available positions. For example, in table 10, note the last entry on the left side, where population $\begin{smallmatrix}2\\1\end{smallmatrix}\ \begin{smallmatrix}1\\1\end{smallmatrix}$ is being mapped to new x value 3. Because the population contains x values 2 and 1, the new x value could only be a 0 or a 3. These two values 0 and 3 comprise a listing of available positions. For new x value 3, the corresponding index into the list of available positions is 1. The third row of table 11 lists each index value for each new x value from the second row.

In this listing of indices, an element that corresponds to population size n can be thought of as a base $N-n$ digit since there are $N-n$ choices for a new x value at that point. For example, in table 11 when the population size is 1, the index values can be seen as base 3 digits, and when the population size is 2, the index values can be seen as base 2 digits.

All digits having the same population size can be combined into a *single* base $N-n$

number number having M^n digits. This can be done for all values of n from zero to $N-2$. In the example of table 11, $1001011010000010_2 = 38530_{10}$, $1120_3 = 42_{10}$, and $2_4 = 2_{10}$.

To combine these numbers into a single canonical number, note that there are 2^{16} possible values when the population size is 2, and 3^4 possible values when the population size is 1. Generally, there will be $(N-n)^{M^n}$ possible values when the population size is n. The three numbers in the example can therefore be combined as follows:

$$2 \cdot 3^4 \cdot 2^{16} + 42 \cdot 2^{16} + 38530 = 13,407,874$$

To arrive at the table form of a search algorithm given its canonical number, start with modulus arithmetic to decompose the single number into a number for each population size. If the canonical number is 13,407,874 and $N = M = 4$, the modulus arithmetic proceeds as follows:

$$\begin{aligned} 13,407,874/(3^4 \cdot 2^{16}) &= 2 \text{ Remainder } 2,791,042 \\ 2,791,042/2^{16} &= 42 \text{ Remainder } 38,530 \end{aligned}$$

This yields the numbers 2, 42, and 38530. These can be converted to base 4, 3, and 2 numbers respectively, creating the list of new x indices as shown in row 3 of table 11. From here, the process of creating the table proceeds as described above, with the following added step: as the populations are constructed, a list of available values is created so that the new x index can be converted into a new x value.

Table 12 lists all search algorithms and their canonical numbers for the case when $\mathcal{X} = \{0,1,2\}$ and $\mathcal{Y} = \{0,1\}$. This table can be compared against the population table in table 1 which contains the same information in a different form.

Table 12: All non-repeating search algorithms and their associated canonical numbers when $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1\}$. Compare against the population table in table 1.

num	pop	new x	num	pop	new x
0	$\langle\rangle$	0	6	$\langle\rangle$	1
	$\begin{smallmatrix}0\\0\end{smallmatrix}$	1		$\begin{smallmatrix}1\\0\end{smallmatrix}$	2
	$\begin{smallmatrix}0\\1\end{smallmatrix}$	1		$\begin{smallmatrix}1\\1\end{smallmatrix}$	0
1	$\langle\rangle$	0	7	$\langle\rangle$	1
	$\begin{smallmatrix}0\\0\end{smallmatrix}$	1		$\begin{smallmatrix}1\\0\end{smallmatrix}$	2
	$\begin{smallmatrix}0\\1\end{smallmatrix}$	2		$\begin{smallmatrix}1\\1\end{smallmatrix}$	2
2	$\langle\rangle$	0	8	$\langle\rangle$	2
	$\begin{smallmatrix}0\\0\end{smallmatrix}$	2		$\begin{smallmatrix}2\\0\end{smallmatrix}$	0
	$\begin{smallmatrix}0\\1\end{smallmatrix}$	1		$\begin{smallmatrix}2\\1\end{smallmatrix}$	0
3	$\langle\rangle$	0	9	$\langle\rangle$	2
	$\begin{smallmatrix}0\\0\end{smallmatrix}$	2		$\begin{smallmatrix}2\\0\end{smallmatrix}$	0
	$\begin{smallmatrix}0\\1\end{smallmatrix}$	2		$\begin{smallmatrix}2\\1\end{smallmatrix}$	1
4	$\langle\rangle$	1	10	$\langle\rangle$	2
	$\begin{smallmatrix}1\\0\end{smallmatrix}$	0		$\begin{smallmatrix}2\\0\end{smallmatrix}$	1
	$\begin{smallmatrix}1\\1\end{smallmatrix}$	0		$\begin{smallmatrix}2\\1\end{smallmatrix}$	0
5	$\langle\rangle$	1	11	$\langle\rangle$	2
	$\begin{smallmatrix}1\\0\end{smallmatrix}$	0		$\begin{smallmatrix}2\\0\end{smallmatrix}$	1
	$\begin{smallmatrix}1\\1\end{smallmatrix}$	2		$\begin{smallmatrix}2\\1\end{smallmatrix}$	1

A search algorithm will be said to be compressible if it can be expressed in fewer bits than its canonical number. It is well known [1] that the percentage of strings that are compressible is rather small, and thus the percentage search algorithms that are compressible must also be small.

For sufficiently large domain and co-domain, search algorithms such as enumeration, hill climbing, and GAs are highly compressible. One might think that a search algorithm's complexity might contribute to its optimization power, but that need not be the case. For example, a static enumeration is highly compressible, but the NFL theorem guarantees that it has optimal expected performance in black box scenarios. On the other hand, if there are known correlations between seen and unseen points, memory and processing of those seen points may lead to improved performance.

Table 13 shows how the size of the domain and co-domain affects the number of functions, performance vectors, non-repeating search paths, populations, and non-repeating search algorithms. Code was written to generate any of these objects given their canonical number, and this was used to generate the population tables in this dissertation as well as the canonical tables in this section. The code was also used extensively in the Minimax section (2.7) below to test the performance of all search algorithms over all functions.

2.6 The No Free Lunch Theorems

In section 1.4 there was discussion regarding the lack of clarity of previous proofs of the No Free Lunch theorem. On the other hand, the performance table data structure provides a natural foundation for comparing search algorithms. This section will prove the No Free Lunch theorem as a direct and natural consequence of the basic properties of performance tables. Previous statements of the NFL theorem will be discussed from

Table 13: The number of functions, performance vectors, non-repeating search paths, populations, and non-repeating search algorithms for various values of N and M.

N	M	Fs	PVs	SPs	Ps	SAs
		M^N	$\sum_{n=1}^{N} M^n$	$\sum_{n=1}^{N} (N)_n$	$\sum_{n=1}^{N} (N)_n M^n$	$\prod_{n=0}^{N-2} (N-n)^{M^n}$
2	2	4	6	4	12	2
2	3	9	12	4	24	2
2	4	16	20	4	40	2
2	5	25	30	4	60	2
2	6	36	42	4	84	2
2	7	49	56	4	112	2
3	2	8	14	15	78	12
3	3	27	39	15	225	24
3	4	64	84	15	492	48
3	5	125	155	15	915	96
3	6	216	258	15	1530	192
3	7	343	399	15	2373	384
4	2	16	30	64	632	576
4	3	81	120	64	2712	55296
4	4	256	340	64	7888	$\approx$ 2.1 e7
4	5	625	780	64	18320	$\approx$ 3.3 e10
4	6	1296	1554	64	36744	$\approx$ 2.0 e14
4	7	2401	2800	64	66472	$\approx$ 4.9 e18
5	2	32	62	325	6330	1658880
5	3	243	363	325	40695	$\approx$ 8.4 e14
5	4	1024	1364	325	157780	$\approx$ 1.0 e30
5	5	3125	3905	325	458025	$\approx$ 1.8 e50
5	6	7776	9330	325	1102350	$\approx$ 3.2 e86
5	7	16807	19607	325	2326555	$\approx$ 3.5 e131
6	2	64	126	1956	75972	$\approx$ 1.7 e13
6	3	729	1092	1956	732528	$\approx$ 3.6 e45
6	4	4096	5460	1956	3786744	$\approx$ 6.4 e120
6	5	15625	19530	1956	13740780	$\approx$ 1.3 e267

within the current framework, and will all be shown to be equivalent. A proof will follow that demonstrates how the NFL theorem can be generalized to other sets of functions. Finally, an NFL proof will be given for stochastic search (as defined in section 2.4).

Let $P_m(A, f)$ denote the length m population generated by search algorithm A and function f. Similarly let $V_m(A, f)$ denote the length m performance vector generated by A and f. The size subscript may be omitted when it is not needed. Let $M(V(A, f))$ be the value of the performance vector measure M when applied to performance vector $V(A, f)$. Define an *overall measure* of search algorithm A and set of functions F to be a function that maps the set of performance vectors generated by A and F to a real number. An overall measure can be used to compare the overall performance of two search algorithms on a set of functions, and if the two search algorithms have identical overall measures, it can be said that they perform equally well over F. An example of an overall measure would be to take a performance vector measure and apply it to every element in F and then combine the results in some way such as an average, i.e. $\sum_{f \in F} M(V(A, f))/|F|$. A *complete measure* is an overall measure where the set of functions F is the set of all functions from $\mathcal{Y}^{\mathcal{X}}$. Three statements of the NFL theorem will now be proven.

Theorem 12 (NFL1) For any complete measure, each non-repeating deterministic search algorithm performs equally well.

Proof: From theorem 11, each row of a complete performance table contains the same set of performance vectors, and thus any two rows will be using the exact same data for computing the complete measure. □

The following statement of the theorem is a pivotal component of the proof by Radcliffe

and Surry [7], phrased more directly in the language of the current framework. In their language, isomorphic search algorithms generate the same sequences.

Theorem 13 (NFL2) For any two deterministic non-repeating search algorithms A and B, and for any function f, there exists a function g such that $V(A, f) = V(B, g)$.

Proof: Theorem 11 states that all rows in a complete performance table are permutations of each other, and thus any performance vector in row A will necessarily occur in row B. □

As defined above, an overall measure is a function of a set of performance vectors. Consider instead a *weighted overall measure* which is defined with respect to a search algorithm's *row* in a complete performance table. This allows the performance measure of each performance vector to be weighted according to the function that generates it, i.e. $W(f)M(V(A, f))$. A weighted overall measure is *not* generally subject to the NFL theorem except in the case where the functions are equally weighted. An *equally weighted overall measure* is a weighted overall measure where each of the performance vectors in the algorithm's row is weighted equally, i.e. certain functions are not deemed more important than others. The NFL theorem statement below is essentially that given in Wolpert and Macready [18].

Theorem 14 (NFL3) For any equally weighted overall measure, each deterministic non-repeating search algorithm will perform equally well.

Proof: When performance vectors are weighted equally, $W(f)M(V(A, f))$ becomes $cM(V(A, f))$ for some constant c. Let performance vector measure $M'(V) = cM(V)$.

Thus an equally weighted overall measure is in fact a non-weighted overall measure M', allowing the result to follow as in NFL1. □

Note that all three versions of the NFL theorem are direct consequences of theorem 11, demonstrating the significance of that theorem.

A corollary to theorem 11 is that if a search algorithm performs better than average on one set of functions, it must perform worse on the complementary set. This is essentially an argument for specialization: a search algorithm will perform well on a small set of functions at the expense of poor performance on the complementary set.

An even stronger consequence follows from theorem 11: *all* deterministic non-repeating search algorithms are equally specialized. This contradicts commonly stated beliefs (e.g. [2, 3]) about how there can be robust general purpose search algorithms that perform reasonably well on a broad class of functions at the expense of not performing extremely well on any set of functions. Since every search algorithm has precisely the same collection of performance vectors when all functions are considered, it follows that *if any search algorithm is robust, then every search algorithm is, and if some search algorithm is not robust, than no search algorithm can be.*

2.6.1 NFL Generalization

Let $f : \mathcal{X} \rightarrow \mathcal{Y}$ be a function and let $\sigma : \mathcal{X} \rightarrow \mathcal{X}$ be a permutation (i.e. σ is one-to-one and onto). The permutation σf of f is the function $\sigma f : \mathcal{X} \rightarrow \mathcal{Y}$ defined by $\sigma f(x) = f(\sigma^{-1}(x))$.

Define a set F of functions to be *closed under permutation* if for every $f \in F$, every permutation of f is also in F. Define an *NFL result over* F to be a situation where any

two deterministic non-repeating search algorithms will have equal overall performance with respect to the set of functions F.

Let A be a search algorithm with selection operator g and let σ be a permutation (of $\mathcal{X}$). The permutation σA of A is the search algorithm with selection operator σg defined by $\sigma g(\phi) = \sigma^{-1}(g(\sigma_x(\phi)))$ where $\sigma_x(\phi)$ operates on the x values of population ϕ by applying σ to each of them, while leaving the y values untouched.

Theorem 15: If $P_n(A, \sigma f) = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}) \rangle$ then

$$P_n(\sigma A, f) = \langle (\sigma^{-1}(x_0), y_0), (\sigma^{-1}(x_1), y_1), \ldots, (\sigma^{-1}(x_{n-1}), y_{n-1}) \rangle$$

Proof: By induction on the length of the populations. The base case is true since all populations of length 0 are the same, i.e. $P_0(\sigma A, f) = P_0(A, \sigma f) = \langle\rangle$.

Assume the inductive hypothesis:

$$\begin{aligned} P_n(A, \sigma f) &= \langle (x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}) \rangle \\ P_n(\sigma A, f) &= \langle (\sigma^{-1}(x_0), y_0), (\sigma^{-1}(x_1), y_1), \ldots, (\sigma^{-1}(x_{n-1}), y_{n-1}) \rangle \end{aligned}$$

By definition,

$$\sigma g(P_n(\sigma A, f)) = \sigma^{-1} \circ g(\sigma_x(P_n(\sigma A, f)) = \sigma^{-1} \circ g(P_n(A, \sigma f)) = \sigma^{-1}(x_n)$$

Moreover, $f(\sigma^{-1}(x_n)) = \sigma f(x_n) = y_n$. Accordingly:

$$\begin{aligned} P_{n+1}(A, \sigma f) &= P_n(A, \sigma f) \parallel (x_n, y_n) \\ P_{n+1}(\sigma A, f) &= P_n(\sigma A, f) \parallel (\sigma^{-1}(x_n), y_n) \end{aligned}$$

Which completes the proof. □

Corollary 3: $V(\sigma A, f) = V(A, \sigma f)$

Corollary 3 is true since the y values are the same in both populations. This corollary is striking in the way that it shows a correspondence between a permutation of a search algorithm and a permutation of a function.

Lemma 2 If the set of functions F is closed under permutation, then there is an NFL result over F.

Proof: Let A and B be arbitrary deterministic non-repeating search algorithms. If one can show that set $S_1 = \{V(A, f) : f \in F\}$ is equal to set $S_2 = \{V(B, g) : g \in F\}$, the result will follow naturally. This will be shown by double containment. By theorem 13, for any A, B, and f, there exists a function h such that $V(A, f) = V(B, h)$. Because these two complete performance vectors are equal, f and h must be permutations, and thus $h \in F$. It is therefore true that for any element $V(A, f) \in S_1$, there exists an equal element $V(B, g) \in S_2$. The case in the other direction can be demonstrated in the same way, and so $S_1 = S_2$ by double containment. Thus any two deterministic non-repeating search algorithms will be using the same data to compute their combined performance measures and will therefore arrive at the same result. □

The above lemma was an intermediate result in Radcliffe and Surry's proof of the NFL theorem [7]. The converse of this lemma is also true:

Lemma 3 If an NFL result occurs over the set of functions F, then F is closed under permutation.

Proof: Assume by way of contradiction that an NFL result occurs over the set F, but that F is not closed under permutation. Consequently, there is some function f in F which has a permutation g which is not in F. Consider an arbitrary search algorithm A. Let $M(V(A,f)) = 1$, and let M equal zero for all other performance vectors in A's row. Since rows in a performance table are permutations of one another, each row will have a single performance vector having measure 1, and all other performance vectors will have measure 0. Let the overall measure be a sum of performance vector measures, i.e. $\sum_{f \in F} M(V(A,f))$. Note that this sum is 1 for search algorithm A, and since an NFL result is assumed over F, the value of this sum should be 1 for every search algorithm. As f and g are permutations, let $f = \sigma g$. By corollary 3, $V(A,f) = V(A,\sigma g) = V(\sigma A, g)$, and thus $M(V(\sigma A, g)) = 1$. Accordingly, $\sum_{h \in F} M(V(\sigma A, h)) = 0$, a contradiction. □

Combining lemmas 2 and 3 yields the following theorem:

Theorem 16 (NFL Generalization) An NFL result occurs over the set of functions F if and only if F is closed under permutation.

When an NFL result occurs, no search algorithm can outperform enumeration. More generally the phrase "NFL curse" will be used to describe situations where enumeration provides the best possible expected performance.

2.6.2 NFL Theorem for Stochastic Search

The definition of stochastic search in section 2.4 allows for the following proof of the NFL theorem for stochastic search. Let $\lambda(A)$ be the probability that stochastic search algorithm λ behaves as deterministic search algorithm A.

Theorem 17 (NFL for Stochastic Search) For any equally weighted measure, each non-repeating stochastic search algorithm has equal overall expected performance.

Proof: Let $\mathcal{E}_\lambda$ be the overall expected performance of stochastic search algorithm λ. The following shows how the overall expected performance is constant and independent of λ. A is a search algorithm and f is a function.

$$\begin{aligned}
\mathcal{E}_\lambda &= \sum_f \sum_A \lambda(A) M(V(A,f)) \\
&= \sum_A \lambda(A) \sum_f M(V(A,f)) \\
&= c \sum_A \lambda(A) && \text{(by NFL)} \\
&= c && \square
\end{aligned}$$

2.6.3 Nonuniform Function Distributions

The NFL curse is not necessarily escaped with non-uniform distributions of functions, as pointed out by Wolpert and Macready [18]. Appendix A presents two search algorithms having identical performance with respect to a non-uniform function distribution. English proposes [2] that the NFL result occurs in an "uncountable infinitude" of function distributions, and he states that NFL consequences may occur whenever the function set to be optimized is "large and diffuse," which is the very signature of black box search.

2.7 Minimax Distinctions Between Algorithms

Wolpert and Macready [18, 19] discuss a sense in which one search algorithm can outperform another search algorithm, even when all functions are considered. This *minimax distinction* between search algorithms A and B is described as follows: For each function $f \in \mathcal{F}$ (where $\mathcal{F}$ is the set of all functions from the domain to the co-domain), compare the performance measure of A on f to the performance measure of B on f. Keep a tally of the number of times each algorithm has the better performance measure when compared in this pair-wise way. If after all the comparisons have been made A has a larger tally, then A is said to have a *minimax advantage* over B, and the value of the minimax advantage would be the value of A's tally minus the value of B's tally. Let $A > B$ denote the relation that algorithm A has a minimax advantage over algorithm B.

While it may at first seem that the No Free Lunch Theorem would preclude one algorithm from having a minimax advantage over another, that is not the case. The minimax distinction "loses information" in the sense that it does not take into account how much better one algorithm did on a function, but only that it did better. When that information is accounted for, the equal performance guaranteed by the NFL theorem will be evident. Furthermore, it is conjectured that there is still a type equivalence between all search algorithms even when minimax distinctions are present, as will be shown below.

As a simple example of minimax, consider the following two-player game: each player is given a nickel, a dime, and a quarter, and each player then orders them without the other player's knowledge. When both players are satisfied with their orderings, the values at each position are compared, and the winner is the one who has more positions at higher value. The minimax table for this game is presented in table 14. Each entry

Table 14: Minimax table for the coin game. NDQ indicates an ordering of Nickel, Dime, Quarter. Each entry denotes the minimax advantage of the row's strategy over the column's strategy. Minimax tables are symmetric across the major diagonal with a sign change.

	NDQ	NQD	DNQ	DQN	QND	QDN
NDQ	·	0	0	-1	1	0
NQD	0	·	-1	0	0	1
DNQ	0	1	·	0	0	-1
DQN	1	0	0	·	-1	0
QND	-1	0	0	1	·	0
QDN	0	-1	1	0	0	·

denotes the minimax advantage of the row's strategy over the column's strategy, with a negative value indicating a disadvantage.

For any strategic ordering, there is one way to win, and one way to lose. For example, NDQ has minimax advantage over QND, QND has minimax advantage over DQN, but DQN has minimax advantage over NDQ, showing that the relation is non-transitive. Radcliffe and Surry [7] observed this circular relation under a different set of circumstances.

As a more general example, table 15 lists the minimax distinctions for all search algorithms having $\mathcal{X} = \{0, 1, 2\}$ and $\mathcal{Y} = \{0, 1\}$ (as in table 3). The objective in this case is to find the first occurrence of the maximum value in each performance vector. An assumption is made that the maximum will be recognized when it is seen, and thus the search may stop at the first such occurrence. The performance vector measure is the length of the performance vector minus the number of evaluations performed in finding that maximum value. This has the effect that finding the maximum value quickly will yield a high score. Each entry (A_a, A_b) denotes the value of the minimax advantage of A_a over A_b, with a negative number indicating that A_b has a minimax advantage over

Table 15: **Minimax Table:** ($\mathcal{X} = \{0,1,2\}$, $\mathcal{Y} = \{0,1\}$) The objective function is to find the maximum value in the population. Note the non-symmetric relation $A_0 > A_8 > A_6 > A_0$.

	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}
A_0	·	0	0	0	0	0	-1	-1	1	1	0	0
A_1	0	·	0	0	0	0	-1	-1	1	1	0	0
A_2	0	0	·	0	1	1	0	0	0	0	-1	-1
A_3	0	0	0	·	1	1	0	0	0	0	-1	-1
A_4	0	0	-1	-1	·	0	0	0	0	0	1	1
A_5	0	0	-1	-1	0	·	0	0	0	0	1	1
A_6	1	1	0	0	0	0	·	0	-1	-1	0	0
A_7	1	1	0	0	0	0	0	·	-1	-1	0	0
A_8	-1	-1	0	0	0	0	1	1	·	0	0	0
A_9	-1	-1	0	0	0	0	1	1	0	·	0	0
A_{10}	0	0	1	1	-1	-1	0	0	0	0	·	0
A_{11}	0	0	1	1	-1	-1	0	0	0	0	0	·

A_a. For example, it can be seen that A_0 has a minimax advantage over A_8 and A_9, and that algorithms A_6 and A_7 have minimax advantage over A_0. Consider the sum across a row of a minimax table.

Table 16 also lists the minimax distinctions for all algorithms in table 3, but in this table the objective function involves finding the *minimum* value rather than the maximum. The performance vector measure is the length of the performance vector minus the number of evaluations needed to find the minimum value. In this table too, all rows sum to zero.

Just as in table 14, tables 15 and 16 exhibit circular non-transitive minimax advantages. For example, in table 15 $A_0 > A_8 > A_6 > A_0$. This suggests a notion of equivalence. Search algorithms A and B are said to be *minimax equivalent* if there exists a circular relation involving A and B or if neither A nor B has a minimax advantage over the other.

Table 16: **Minimax Table:** ($\mathcal{X} = \{0,1,2\}, \mathcal{Y} = \{0,1\}$) The objective function is to find the *minimum* value in the population. Note that $A_0 > A_8 > A_5 > A_0$.

	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}
A_0	·	0	0	0	0	-1	0	-1	1	0	1	0
A_1	0	·	0	0	1	0	1	0	0	-1	0	-1
A_2	0	0	·	0	0	-1	0	-1	1	0	1	0
A_3	0	0	0	·	1	0	1	0	0	-1	0	-1
A_4	0	-1	0	-1	·	0	0	0	0	1	0	1
A_5	1	0	1	0	0	·	0	0	-1	0	-1	0
A_6	0	-1	0	-1	0	0	·	0	0	1	0	1
A_7	1	0	1	0	0	0	0	·	-1	0	-1	0
A_8	-1	0	-1	0	0	1	0	1	·	0	0	0
A_9	0	1	0	1	-1	0	-1	0	0	·	0	0
A_{10}	-1	0	-1	0	0	1	0	1	0	0	·	0
A_{11}	0	1	0	1	-1	0	-1	0	0	0	0	·

Complete performance tables were created for many values of N and M, and from them minimax tables were produced.[9] Let $[x, y]$ be shorthand for the minimax table where $N = x$ and $M = y$. For [2,2], [2,3], [2,4], [2,5], [2,6] and [2,7], there are no minimax distinctions between search algorithms. For [3,2], [3,3], [3,4], [3,5], [3,6], [3,7] and [4,2], there are minimax distinctions between search algorithms, although in each table, every search algorithm is in the same minimax equivalence class.

In these tables, the performance measure was the number of evaluations required to find the maximum value in the population. Tables were also created where the performance measure was the number of evaluations required to find the minimum value in the population, and once again all search algorithms were found to be in the same minimax equivalence class. These results suggest to the following conjecture:

Minimax Conjecture: All deterministic non-repeating search algorithms with a given

[9]Large values of N and M require prohibitive amounts of memory (see table 2) and thus only modestly sized tables were created.

finite domain and co-domain will be in the same minimax equivalence class.

2.8 Other Possibilities

The overriding value of something as abstract as a population table is in how it facilitates conceptualization. Population tables make clear how each search algorithm behaves under all possible scenarios, and they provide a foundation for proofs that can lead to greater insight into how search algorithms behave.

There are many conjectures as to what makes a function hard for a particular type of search algorithm. Modality, epistasis, compressibility, and other factors have been implicated. It is possible to consider a search algorithm class such as a GA, and to then determine which functions in the population table perform best on that class. These functions could be dissected, and their properties could be stated definitively, at least for the given domain and co-domain. Factors such as modality could be exhaustively examined. Such analysis may take some of the guesswork out of knowing which type of search algorithm to apply to which type of problem. Part of such an analysis would involve the translation from what a search algorithm *gets*, i.e. the populations in the table, to what the search algorithm *does*, i.e. how it makes its selections.

These techniques can require vast amounts of computational time and space, but that does not necessarily make them impractical. For starters, it may be sufficient in certain situations to randomly sample functions rather than testing all functions. In addition, direct simulation may be avoided by mathematical analysis applied to the objects formalized in the framework developed in this dissertation. Algorithmic information theory (commonly known as the study of Kolmogorov complexity) may well contribute to this understanding. The fact that the table is full of functions and search algorithms that are incompressible makes it ripe for this type of analysis. Furthermore, the *incom-*

pressibility method [6] studies the effect of *typical input*, which will be used extensively in the following sections with the discussion of uniformly selected functions.

2.9 Summary of Population Table Properties

- **All Population Tables**
  - Within a row, no two populations may be permutations of each other unless they are equal
  - All possible populations of size m exist in T_m
  - All rows containing P_m will have that population in exactly the same columns
  - Each row has the same number of unique populations
  - Permutations of P_m will occur in the same columns as P_m
  - All permutations of a population (and thus all possible search paths) exist in each column
- **Complete Population Tables**
  - Rows are unique
  - Each population in a row is unique
  - No two populations in a row are permutations of each other
  - Rows are not permutations of each other
  - No population exists in more than one column

2.10 Summary of Performance Table Properties

- **All Performance Tables**

  - Rows are permutations of each other
  - Each row has the same number of unique performance vectors
  - Within a column, all performance vectors are permutations of one another
  - Within a column, all performance vectors exist

- **Complete Performance Tables**
  - Rows are unique
  - Each performance vector in a row is unique
  - Same number of rows as a complete population table
  - For any row, few of its permutations are also in the table
  - All search algorithms (rows) will have the same complete measure (NFL)
  - Two different columns will differ at each position
  - The number of unique elements in a column will depend on the number of unique co-domain values of the function defining the column

3 Analytical Methods in Black Box Search

Because search algorithms such as genetic algorithms and hill climbing are stochastic in nature, performance results will vary according to the seed of the pseudo-random number generator. To overcome this variation, statistical methods can be used to allow reasonable generalizations regarding performance. One of the main goals of this section is to determine a means of finding the expected behavior of a given search algorithm on a given function. This will allow performance comparisons over both algorithms and functions, as well as performance comparisons of classes of algorithms over classes of functions.

Random search (with and without replacement) and enumeration will be analyzed in detail. Since sets $\mathcal{X}$ and $\mathcal{Y}$ are finite, they can be enumerated, and therefore identified with sets of non-negative integers. And so without loss of generality, the analysis in this section will pertain to functions that map from $\mathcal{X} = \{0, 1, \ldots, N-1\}$ to $\mathcal{Y} = \{0, 1, \ldots, M-1\}$. This section will also introduce a means of graphing the expected performance of an algorithm over a class of functions.

For most experiments, each function will be run to convergence many times, each time with a new seed.[10] The number of evaluations required to converge is recorded for each run. Note that it is the number of function evaluations being counted and not the number of generations the GA is being run.[11] This allows hill climbers and genetic algorithms to be compared on equal footing. All runs were required to achieve acceptable performance.

The data from running a stochastic search algorithm on a function multiple times can

[10] Unless otherwise specified, convergence is defined as the first occurrence of a value in a given subset of the co-domain.

[11] In certain situations an evaluation will not be counted, e.g. when using elitism, the best strings need not be reevaluated in the next generation.

be used to construct a histogram that shows the percentage of runs that converged at each trial number. Such a histogram will be called a *performance histogram.* With an adequate sample size, it can be thought of as a way to visualize the expected probability of first success at a given evaluation number.

Now consider a graph that charts the *cumulative* percentage of success at each trial number, i.e. $f(x) = \sum_{i=1}^{x} h(i)$ where $h(i)$ is the percentage of runs that converged at trial i. This type of graph will be called a *performance profile.* Notice that since the percentages are summed, a performance profile graphs an approximation of the probability of success by trial x.

3.1 Expected Performance of Random Search

Random search is an important benchmark to compare against, especially when considering the ramifications of the No Free Lunch theorem. Therefore its expected behavior will be studied in some detail. Let N be the size of the search space, let M be the size of the co-domain, and let n be the size of the set that indicates convergence.[12] Convergence is recognized at the first evaluation of an element from the convergence set.

The expected optimum value of a uniformly selected function can be found with the following analysis. $P(S)$ will denote the probability of event S. Let y be the event that the optimum co-domain value is y. Let A be the event that all co-domain values are less than or equal to y, i.e. $A = \{f \ : \ f(x) \leq y \ \forall x \in \mathcal{X}\}$. Let B be the event that at least one co-domain value is equal to y, i.e. $B = \{f \ : \ \exists x \in \mathcal{X} : f(x) = y\}$. Let C be the event that no co-domain value is equal to y, i.e. $C = \{f \ : \ f(x) \neq y \ \forall x \in \mathcal{X}\}$.

[12]This definition allows for two usages: either the maximum value is repeated n times, or more generally, the top n values in the space indicate adequate convergence.

$$
\begin{aligned}
P(y) &= P(A \cap B) \\
&= P(A)P(B|A) \\
&= P(A)(1 - P(C|A)) \\
&= \left(\frac{y}{M}\right)^N \left(1 - \left(\frac{y-1}{y}\right)^N\right) \\
&= \frac{y^N - (y-1)^N}{M^N}
\end{aligned}
$$

The expected optimum value can be computed as follows:

$$
\begin{aligned}
\mathcal{E}(y) &= \sum_{y=1}^{M} y \left(\frac{y^N - (y-1)^N}{M^N}\right) \\
&= \frac{1}{M^N}\left(\sum_{y=1}^{M} y^{N+1} - \sum_{y=0}^{M-1}(y+1)y^N\right) \\
&= \frac{1}{M^N}\left(\sum_{y=1}^{M-1} y^{N+1} + M^{N+1} - \sum_{y=1}^{M-1} y^{N+1} - 0^{N+1} - \sum_{y=0}^{M-1} y^N\right) \\
&= M - \sum_{y=1}^{M-1}\left(\frac{y}{M}\right)^N
\end{aligned}
$$

In the following experiments, the domain size and co-domain size will often be 10 bit and 16 bit respectively. In such a case, the expected optimum value is approximately 65,472, or about 64 less than the optimum.

3.1.1 Random Search with Replacement

Consider a random search algorithm that chooses points with replacement according to a uniform distribution, and consider the application of this search algorithm to an

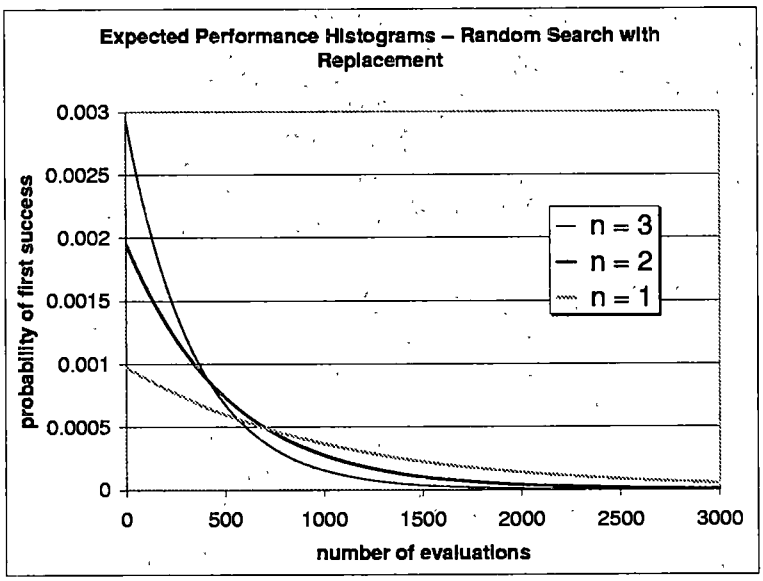


Figure 2: Expected performance histograms for random search with replacement when the size of the search space is 1024 and there are n global optima.

arbitrary function. Let p be the probability of generating an optimum value for a single trial, let q be the probability of not generating an optimum value for a single trial, and let P_i be the probability that the *first* optimum occurs at trial i. First success at the ith trial is a matter of $i-1$ failures and the single final success, all of which are independent because of replacement:

$$\begin{aligned} P_i &= p \cdot q^{i-1} \\ &= \frac{n}{N} \cdot \left(\frac{N-n}{N}\right)^{i-1} \end{aligned}$$

This value allows the *expected* performance histogram and performance profile to be created, i.e. a profile that outlines the expected performance on black box search. The expected performance histograms for various values of n are shown in figure 2. The associated performance profiles are shown in figure 3.

Consider now the expected number of evaluations before convergence. For random

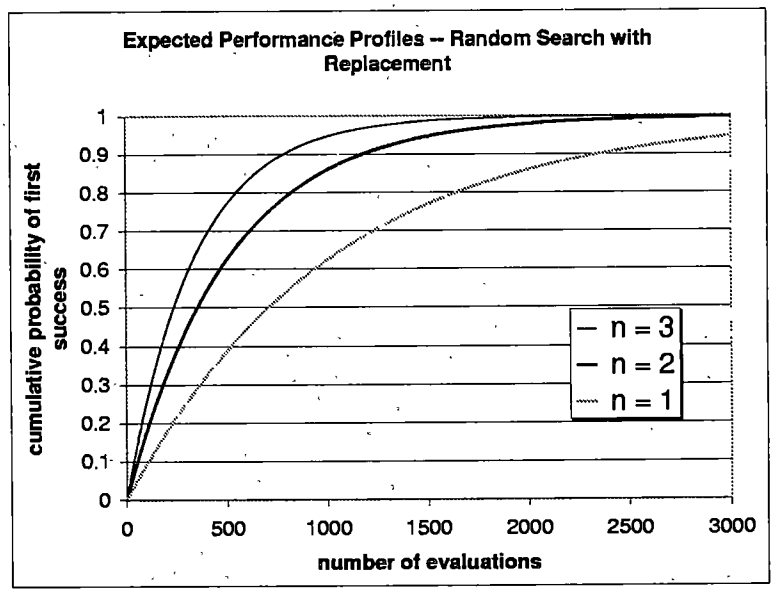


Figure 3: Expected performance profiles for random search with replacement when the size of the search space is 1024 and there are n global optima.

search with replacement, the derivation for expected time to convergence is as follows:

$$\begin{aligned}
\mathcal{E}(x) &= \sum_{x=1}^{\infty} x \cdot P_x \\
&= \frac{n}{N} \sum_{x=1}^{\infty} x \cdot q^{x-1} \qquad\qquad q = (N-n)/N \\
&= \frac{n}{N} \frac{d}{dq} \sum_{x=1}^{\infty} q^x \\
&= \frac{n}{N} \frac{d}{dq} \frac{1}{1-q} \\
&= \frac{n}{N(1-q)^2} \\
&= \frac{n}{N(n/N)^2} \\
&= \frac{N}{n}
\end{aligned}$$

In the case where there is a single global maximum, random search with replacement will thus be expected to perform as many evaluations as there are points in the search

space.

3.1.2 Random Search without Replacement

Just as with random search with replacement, this search algorithm will have equal expected performance for any two functions having the same values for N and n. Let p_i be the event of finding a global optimum at step i, and let P_i be the event of finding the first optimum value at step i. Let q_i be the event of not finding a global optimum at step i, and let Q_i be the event of not finding a global optimum at any step less than or equal to i. As before, let N be the size of the search space and let n be the size of the set that indicates convergence. The last step in the following derivation for the probability of Q_i utilizes a telescoping product simplification.

$$
\begin{aligned}
P(Q_i) &= P(q_i \cap Q_{i-1}) \\
&= P(q_i \mid Q_{i-1}) \cdot P(Q_{i-1}) \\
&= \frac{N-n-i+1}{N-i+1} \cdot P(Q_{i-1}) \\
&= \prod_{j=1}^{i} \frac{N-n-j+1}{N-j+1} \qquad \left(P(Q_1) = P(q_1) = \frac{N-n}{N}\right) \\
&= \prod_{j=0}^{i-1} \frac{N-n-j}{N-j} \\
&= \prod_{k=0}^{n-1} \frac{N-i-k}{N-k}
\end{aligned}
$$

Note that in the final result there is no danger of dividing by zero since $n-1$ will always be less than N. $P(P_i)$, the probability of first success at step i, is the probability of $i-1$ failures and the single final success:

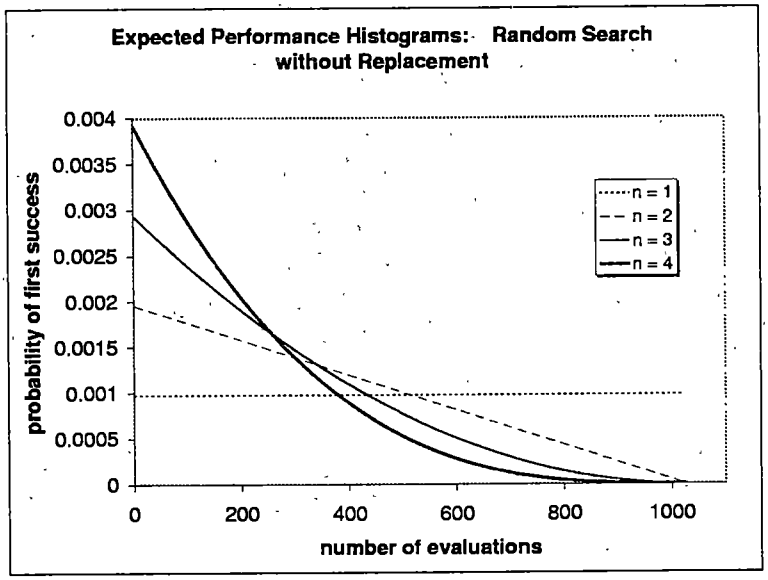


Figure 4: Expected performance histograms for random search without replacement when the size of the search space is 1024 and there are n global optima.

$$\begin{aligned} P(P_i) &= P(p_i \cap Q_{i-1}) \\ &= P(p_i \mid Q_{i-1}) \cdot P(Q_{i-1}) \\ &= \frac{n}{N-i+1} \prod_{k=0}^{n-1} \frac{N-i+1-k}{N-k} \end{aligned}$$

This formula was used to graph expected performance histograms and expected performance profiles for random search without replacement, as shown in figures 4 and 5.

For random search without replacement, the expected number of evaluations would be

$$\begin{aligned} \mathcal{E}(x) &= \sum_{x=1}^{N} x \cdot P(P_x) \\ &= \sum_{x=1}^{N} \frac{x \cdot n}{N-x+1} \prod_{k=0}^{n-1} \frac{N-x+1-k}{N-k} \end{aligned}$$

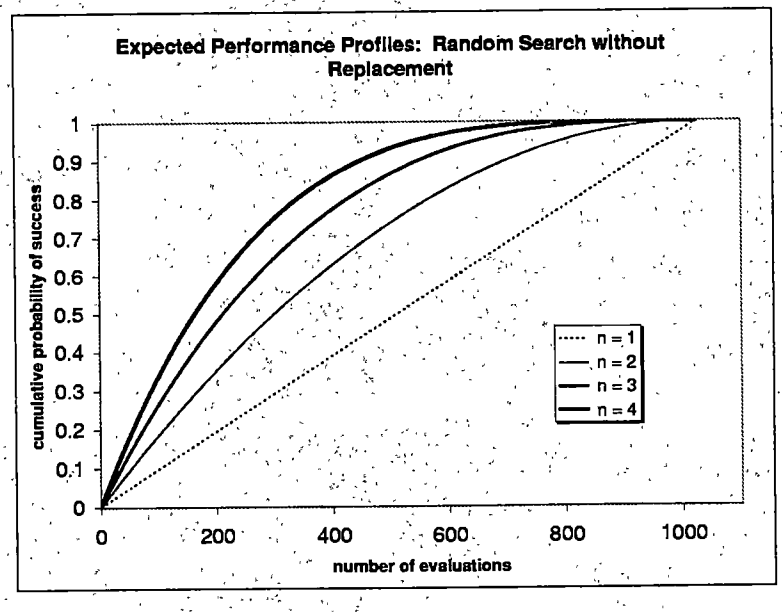


Figure 5: Expected performance profiles for random search without replacement when the size of the search space is 1024 and there are n global optima.

$$= \frac{N+1}{n+1}$$

Recall that the expected convergence time of random search with replacement is N/n, and thus as n increases, the advantage of non-repeating search diminishes.

3.2 Enumeration

Search without replacement is an enumeration, and by the NFL theorem all enumeration strategies have equal performance over the ensemble of all functions. The NFL theorem also makes clear that no other search algorithm can have better performance over all functions, and thus enumeration's performance profile can serve as a useful baseline for judging the performance of other search algorithms. If the search algorithms are allowed to differentiate themselves by the points they have already seen (as is the case in population tables), then the number of enumeration search algorithms is

the Algorithm Count Result 1 from section 2.2.1.

While random search without replacement will have the same expected performance on all functions having the same N and n values, enumeration strategies can perform very differently depending on the function.[13]

3.3 An Empirical Investigation of Robustness

Genetic algorithms and hill climbers are widely believed to be robust search algorithms, i.e. they are expected to perform reasonably well on a very large set of functions. This section will investigate this assertion by exploring the performance of these two search algorithms on randomly selected functions and by comparing their performance against random search.

A randomly selected function is generated by randomly selecting a co-domain element (with uniform probability) for each value in the domain. It is clear that such a function generator selects uniformly over *all* possible functions having the given domain and co-domain. Even though these functions have been randomly generated, they should not be called random functions since they are completely deterministic; rather, they are *randomly selected* functions. Unless otherwise mentioned, a randomly selected function will come from a uniform distribution.

A function having 10 bit domain and 16 bit co-domain was randomly selected for this experiment. This function turns out to be extremely difficult for the GA, more difficult than many of the toughest commonly used test functions for GAs. The performance profiles of this function at different mutation rates are shown in figure 6. Performance improves as the mutation rate is increased to 0.5, at which point the profile become

[13]For example, if the optimum value of the function occurs at the first value that the enumeration tests, optimal performance is achieved.

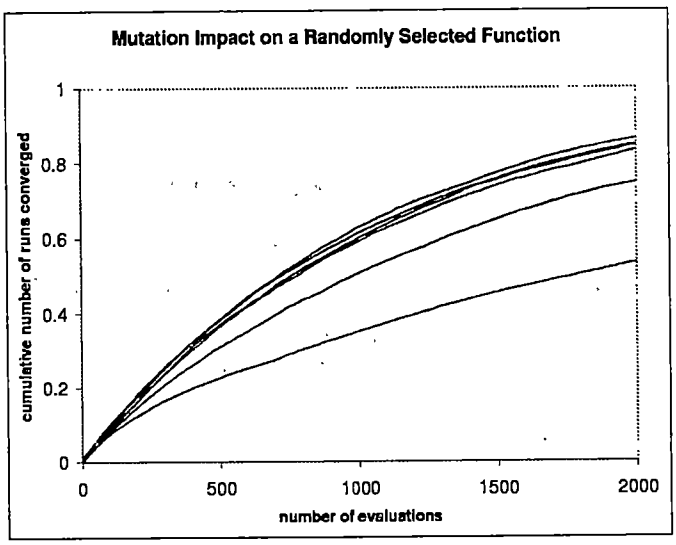


Figure 6: Performance profiles at various mutation rates for a randomly selected function. Profiles are shown at mutation rates of 0.05, 0.1, 0.2, 0.3, 0.4, and 0.5, with performance increasing with mutation.

indistinguishable from that of random search with replacement.

For this randomly selected function the genetic algorithm does not even beat random search with replacement, no matter what the mutation rate. Furthermore, it performs far worse than random search without replacement. Although this is only one randomly selected function, its performance is typical, as will be shown by looking at large numbers of randomly selected functions.

The poor performance of the GA begs the question: what percentage of functions are well suited to solving with a GA? A performance summary can be produced over a set of functions by running each function to convergence many times and accumulating the results for all functions in the set. If the set of functions is very large, a further approximation may involve using a reasonable number of randomly selected functions from that set.

For this section, histograms were generated by randomly selecting 10,000 functions (via a uniform distribution), where each function is run 100 times to convergence. The average convergence time for each function is noted, and these averages are graphed as a histogram. Such a histogram will be called a *black box histogram* because of its relevance to black box search. In this section, the functions used will have $N = 1024$ (10 bit) and $M = 65{,}536$ (16 bit).

Because random search with replacement has the same expected performance for each function, a black box histogram for it would be an impulse at the expected value of N/n. For the same reason, a black box histogram of random search without replacement would be an impulse at the expected value of $(N+1)/(n+1)$.

The situation is different for enumeration since it *can* have different expected performances on different functions. In order to determine the expected percentage of functions that converge at each evaluation number, note that in the case when $n = 1$, the number of functions that have the single optimum at point a is the same as the number of functions that have the single optimum at any other point b, leading to a uniform distribution. This is the same distribution that exists in enumeration's expected performance histogram for $n = 1$. In general, note that the formula for $P(P_i)$ in section 3.1.2 is the probability of first success at evaluation i, and that for enumeration, this can be seen as the percentage of functions that converge at the given evaluation number. The expected performance histogram of enumeration (figure 4) is thus the black box histogram for enumeration.

Figure 7 shows a frequency distribution indicating how both a GA and a hill climber perform over 10,000 functions. The width of each bin is 50. The expected distribution of enumeration ($n = 1$) is included for comparison, adjusted for bin width and the number of functions run. Recall that each function was run 100 times to convergence,

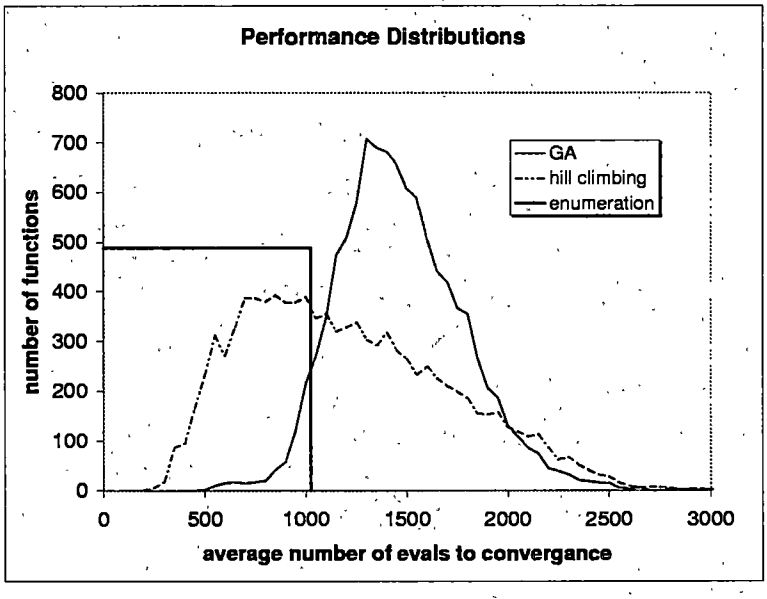


Figure 7: **Black box histograms:** This figure shows how both a GA and hill climbing perform over 10,000 randomly selected functions. The expected distribution for enumeration is included for comparison.

and the average number of evaluations were noted. For the GA, the average of these averages was 1447 evaluations, and for the hill climber, the average was 1185. Given $n = 1$, enumeration's average run time would be 512.5, less than half of either of its competitors. These graphs can be scaled to show the expected percentage of functions that converge at each evaluation number.

The same experiment was performed at mutation rates of 0.2, 0.4, and 0.5, with the resulting black box histograms shown in figure 8. Once again it is clear that a low mutation rate leads to poor GA performance for black box search. As the mutation rate increases, the GA approaches the performance of random search with replacement, and at 0.5, the GA *becomes* random search with replacement since the elements in one generation are not expected to bear any resemblance to the elements in the next generation.

At high mutation, increasing the number of runs per function will have the effect of

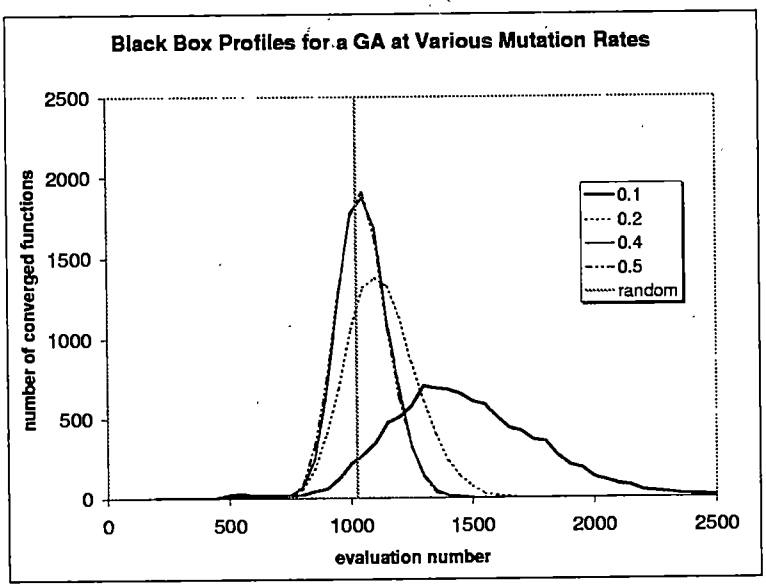


Figure 8: Black box histograms for a GA at various mutation rates, and the black box histogram spike of random search with replacement. As the mutation increases, the GA's black box histogram more closely resembles the profile of random search with replacement. The profiles for 0.4 and 0.5 are almost indistinguishable.

narrowing and heightening the curve, making it even more like the expected black box histogram spike for random search with replacement. The variation at high mutation in figure 8 is attributable to the modest sample size when computing the average for each function.

Just as a GA becomes more like random search with replacement as the mutation rate increases, hill climbing can become more like random search without replacement by changing a parameter, namely the neighborhood size. The hill climbing experiment above shows performance with neighborhood size $s = 10$. When $s = N$, hill climbing *is* random search without replacement, and thus hill climbing performance is expected to improve on a black box function as neighborhood size increases. Note that the performance limit with hill climbing (random search without replacement) is significantly better than the performance limit with GAs (random search with replacement).

In these experiments, only a small fraction of functions were solved faster than the expected search time of enumeration. The NFL theorem makes clear that in black box situations, enumeration can't be beat, but these experiments show that enumeration is *vastly* superior to genetic algorithms and hill climbing for black box search. By the above experiments and because of the pathological way that a GA can retest points, the following conjecture is made, which runs in stark contrast to hype in the field:

Black Box Futility Conjecture: In a black box search scenario, the following four search algorithms will be expected to exhibit the following performance ranking, from best to worst: random search without replacement, random search with replacement, hill climbing, genetic algorithms.

In [4], it is mentioned that Rik Belew "once jokingly remarked to [David Goldberg] that it requires an evil mind to dream up deceptive, blocked, or otherwise 'GA-yucky' functions." On the contrary, *most* functions are "GA-yucky" and finding one is as easy as rolling the dice.

4 Representation and Black Box Search

Section 3 identified $\mathcal{X}$ and $\mathcal{Y}$ with sets of integers. Consider $\mathcal{X} = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ and $\mathcal{Y} = \{\triangle, \square, \bigcirc\}$. These may be identified with sets $\{0,1,2,3\}$ and $\{0,1,2\}$ respectively in the following manner:

$$\clubsuit \Rightarrow 0 \quad \diamondsuit \Rightarrow 1 \quad \heartsuit \Rightarrow 2 \quad \spadesuit \Rightarrow 3$$

$$\triangle \Rightarrow 0 \quad \square \Rightarrow 1 \quad \bigcirc \Rightarrow 2$$

In this representation, the function

$$f = \{(\clubsuit, \square), (\diamondsuit, \triangle), (\heartsuit, \square), (\spadesuit, \bigcirc)\}$$

is represented by the function $f' = |x-1|$, i.e. $\{(0,1),(1,0),(2,1),(3,2)\}$. If, however, set $\mathcal{Y}$ was identified as above but set $\mathcal{X}$ was identified with $\{0,1,2,3\}$ in some other way, for instance

$$\clubsuit \Rightarrow 3 \quad \diamondsuit \Rightarrow 2 \quad \heartsuit \Rightarrow 1 \quad \spadesuit \Rightarrow 0$$

Then f is represented by $f'' = |x-2|$, i.e. $\{(3,1),(2,0),(1,1),(0,2)\}$.

It is thus clear that changing the representation as above changes one function into another. Section 2.6.1 defined a permutation of a function as follows: $\sigma f(x) = f \circ \sigma^{-1}(x)$. The permutation σ that allows $\sigma f' = f''$ is $\{(0,3),(3,0),(1,2),(2,1)\}$. This can be verified by computing $\sigma f'$ for all elements in the domain and comparing the results with f'' as follows:

$$\sigma f'(0) = f'\sigma^{-1}(0) = f'(3) = 2 = f''(0)$$

$$\sigma f'(1) = f'\sigma^{-1}(1) = f'(2) = 1 = f''(1)$$

$$\sigma f'(2) = f'\sigma^{-1}(2) = f'(1) = 0 = f''(2)$$

$$\sigma f'(3) = f'\sigma^{-1}(3) = f'(0) = 1 = f''(3)$$

Representations therefore can be identified with permutations. The application of a permutation to a function will be called a change of representation (for that funtion). In this section, it will be assumed that the co-domain $\mathcal{Y}$ has a fixed representation, and thus it is a change of representation of the *domain* that will be discussed.

The following section will demonstrate that a change of representation can dramatically affect the performance of a search algorithm. It will then be shown how changing a representation impacts the ensemble of all functions by examining its effect on a performance table.

4.1 Changing a Representation

As was seen in section 3.3, a GA will generally perform miserably on a randomly selected function. Vose and Liepins[13] have shown how a function can be made easier for a GA by changing its representation. This is illustrated in the following example.

Consider a function f which maps binary strings of length n to a real number, i.e. $f : \{0,1\}^n \rightarrow \Re$. Create an ordering on the domain with the following two rules: binary strings with fewer 1's will come before strings having more 1's; if two strings have the same number of ones, leftmost bits will be deemed more significant. Create a list X containing all strings from the domain of f and sort it based on this ordering of the domain, resulting in list X'. Create another list Y where each element $Y_i = f(X_i)$ (note that Y is a list, not a set, and may contain duplicates). Sort Y by numeric

Table 17: Three bit *linear sort.*

string	fitness
000	*lowest fitness*
001	
010	
100	
011	
101	
110	
111	*highest fitness*

value resulting in list Y'. Create a new function $f' = \{(X'_i, Y'_i) : 0 \leq i \leq N-1\}$. The function f' is thus a change of representation of the original function f where low ordered domain elements are paired with low valued range elements, and high ordered domain elements are paired with high valued range elements. The function f' will be called a *linear sort* of function f, with an example given in table 17.

Such a change of representation effects a unimodal landscape in Hamming space. Figure 9 shows a performance profile for a GA working on a randomly selected function and a performance profile where the same function is given a linear sort and thus the GA is working with a new representation. The change of representation dramatically improves performance. Of course, sorting the search space is more difficult than searching it and thus this is by no means practical, but it shows that a difficult problem can be made easy with a change of representation.

4.2 Change of Representation in a Population Table

Population tables deal directly and completely with all functions and all deterministic non-repeating search algorithms of a finite domain and co-domain, and thus incorporate

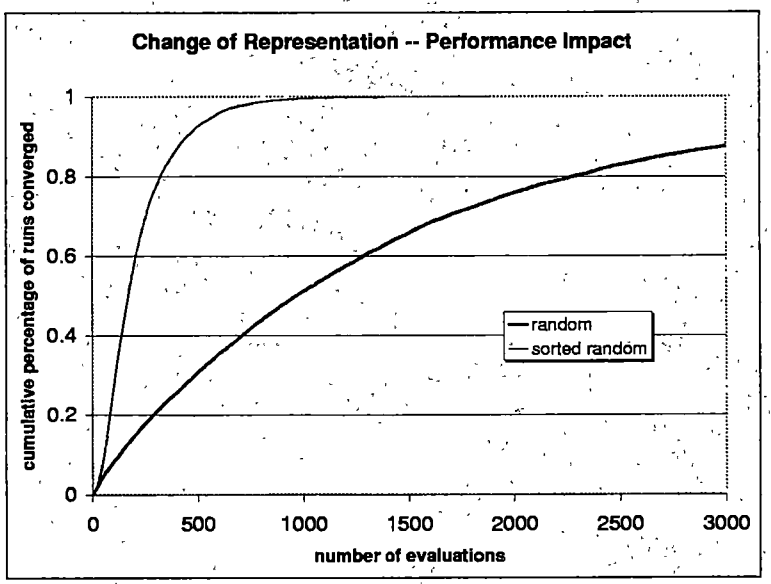


Figure 9: Performance profiles for a GA on a randomly selected fitness function, and on the same function after a linear sort was performed as a change of representation.

all representations.

Consider the situation with $\mathcal{X} = \{x, y, z\}$ and $\mathcal{Y} = \{0, 1\}$. There are eight possible functions and six possible representations of the search space. An ordering of $\{0, 1, 2\}$ will show how the original input space is mapped to a new representation. Table 18 shows how each representation alters the effect of the function in the original domain. For example, in row 021 and column f_1, $f_1(0) = 0$, $f_1(1) = 0$, and $f_1(2) = 1$. The effect is that f_1 with the change of representation looks like f_2 without the change.

Each row is a permutation of the others, and thus a change of representation shuffles functions when seen as the effect it has on the original problem space. From an abstract perspective this corresponds to the fact that the map $f \mapsto \sigma f$ is one-to-one and onto (since $f \mapsto \sigma^{-1} f$ is the inverse). Because of this, a change of representation can be seen as rearranging the columns (functions) of a population table or a performance

Table 18: A table listing all functions and representations when $\mathcal{X} = \{x, y, z\}$ and $\mathcal{Y} = \{0, 1\}$. The row heading indicates how the domain is remapped for each representation, and the row shows the effect of the remapping on the original function space. Each element of row xyz shows $f(x)f(y)f(z)$ as a string.

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
xyz	000	001	010	011	100	101	110	111
012	000	001	010	011	100	101	110	111
021	000	010	001	011	100	110	101	111
102	000	001	100	101	010	011	110	111
120	000	010	100	110	001	011	101	111
201	000	100	001	101	010	110	011	111
210	000	100	010	110	001	101	011	111

table. Seen this way, it is clear that a change of representation can result in very different performance for a given algorithm working on a given function, but that the overall measure of performance will be the same for any representation. This is one way to conclude that all representations provide equal performance over all functions, a foundation of the NFL proof of Radcliffe and Surry [7].

Note that there are more rows in the population table than there are representations, which makes it clear that search algorithm A cannot always be transformed into search algorithm B through a change of representation. On the other hand, for any search algorithm and any change of representation, there exists another search algorithm in the original representation with exactly the same performance for every function. Expressed formally, $\forall A\ \forall \sigma\ \exists B\ \forall f\ V(B, f) = V(A, \sigma f)$. This may be viewed as saying that a change of representation amounts to a change of search algorithm. In fact the search algorithm B that exists is none other than σA.

4.3 Problem Difficulty and Representation

The previous sections made clear that an optimization problem can have varying degrees of difficulty depending on the representation used. However, there is often a desire to think of an optimization problem as being somehow independent of representation. This section explores how a function's difficulty can be thought of as being independent of representation by averaging over representations. The surprising result is that a function's difficulty is the same whether averaged over representations or search algorithms.

Function difficulty across all representations for a fixed search algorithm A can be based on the performance vectors generated from A and G where G is a listing of functions that result from applying all representations to function f (this listing may contain duplicates). Because G is closed under permutation, the value of a complete measure over G is independent of the search algorithm (by theorem 16). In order to determine a function's difficulty across all search algorithms, one can refer to that function's column in a performance table. For example, the collection of performance vector measures from a column could be combined in a normalized sum.

The following theorem shows that average difficulty of a function over all representations coincides with average difficulty over all search algorithms. Let $|A|$ indicate the number of rows in a performance table, and let $|\sigma|$ indicate the number of representations (permutations), i.e. $N!$.

Theorem 18:

$$\frac{1}{|\sigma|}\sum_{\sigma} M(V(A,\sigma f)) = \frac{1}{|A|}\sum_{A} M(V(A,f))$$

where the left hand side of the equation is over all permutations σ and the right hand side is over all search algorithms A.

Proof: In the first step of the following derivation, σ is applied to A without affecting the sum since σA ranges over all search algorithms as A does. Because the sum is independent of σ, one may sum over all σ and divide by the number of permutations as is done in line two. In line three, the summation over σ is moved inside, and corollary 3 is applied, moving the σ from the A to the f. Because $\sum_\sigma M(V(A, \sigma f))$ is search algorithm independent (theorem 16), it may be factored out of the summation over search algorithms as is done in line four.

$$\begin{aligned}
\frac{1}{|A|}\sum_A M(V(A,f)) &= \frac{1}{|A|}\sum_A M(V(\sigma A,f)) && (1)\\
&= \frac{1}{|\sigma|}\sum_\sigma \frac{1}{|A|}\sum_A M(V(\sigma A,f)) && (2)\\
&= \frac{1}{|\sigma|}\frac{1}{|A|}\sum_A\sum_\sigma M(V(A,\sigma f)) && (3)\\
&= \frac{1}{|\sigma|}\sum_\sigma M(V(A,\sigma f))\frac{1}{|A|}\sum_A 1 && (4)\\
&= \frac{1}{|\sigma|}\sum_\sigma M(V(A,\sigma f)) && (5)
\end{aligned}$$

□

In theorem 18, the measure of overall function difficulty is the same whether one averages over search algorithms or representations. Recall that in section 2.6, a version of the NFL theorem (theorem 14) was given that required equally weighted functions. The results from this section indicate that this restriction can be lifted *provided* that an overall difficulty measure (over all representations) is used for each function rather than a performance vector measure: Let $D(f)$ indicate an overall difficulty measure for function f as described in theorem 18, and let $W(f)$ indicate the weight given to function f. Because both $D(f)$ and $W(f)$ are independent of search algorithm, the combined measure $\sum_f W(f)D(f)$ must be as well.

5 Compressibility of Functions and Black Box Search

As discussed in section 2.6, the NFL theorem is primarily concerned with a uniform distribution of functions; search algorithms *can* perform better than enumeration, but only on a small subset of functions. Holland [5] has argued that compressible functions could be such a fruitful subset for local search algorithms such as hill climbing and GAs.[14] He argues that the NFL theorem is not particularly relevant in practice for just this reason, i.e. since most functions in practice are compressible, the NFL theorem will not impact such work. This section will test the underlying premise, namely that non-compressible functions are more difficult for a search algorithm than compressible functions.

There are many examples of functions that are easy to describe, but difficult for a GA or a hill climber. For example, a needle in a haystack function is one where there is a single global maximum with relatively high fitness, with all other points in the search space having relatively low fitness. It is clear that no search algorithm can outperform enumeration on such a function class, and thus the NFL theorem can be applicable in situations involving highly compressible functions. Furthermore, section 3.3 indicated how enumeration performs better than a GA or a hill climber for the majority of randomly selected functions. Consequently, enumeration will be expected to perform better than a GA or a hill climber on *any* size set of randomly selected functions. The original NFL statement involves performance over all functions, but even on a single randomly selected function, enumeration is expected to have better performance than a GA or a hill climber.

In a test of how function compressibility impacts search algorithm performance, one might wish to compare GA performance on a compressible set of functions against GA

[14] Compressible functions are known to be a small fraction of all functions [1].

performance on an incompressible set of functions. While compressible functions can be generated easily enough, obtaining a set of incompressible functions is problematic since determining compressibility is in general uncomputable [6]. Instead of comparing against a set of incompressible functions, this section will compare against a set of functions whose members have a high probability of being incompressible.

One conceivable source of compressible functions takes functions from common testing and benchmark functions for search algorithms. However, this is a problematic source since so many of these functions are poor measures of performance (Whitley et. al. [16]), commonly exhibiting low modality (Whitley et. al. [17]) and low epistasis (Salomon [10]). Furthermore, in order to have the greatest relevance to black box search, both compressible and incompressible function classes should be as large and as diverse as possible.

For the compressible set of functions, a pseudo-random number generator will be used. Since it can be implemented in a relatively small number of bits, it will generate highly compressible functions when the functions are sufficiently sized. Furthermore, since the pseudo-random number generator will attempt to select numbers in a uniform fashion, it will draw from a large and diverse class of functions. For the incompressible function set, a source of truly random numbers will be used to build the functions. This provides a function set that is maximal in that it includes all possible functions, and diverse in that it will be expected to sample all functions uniformly. Because most functions are incompressible, this set is expected to contain mostly incompressible functions. Pseudo-random numbers were generated with the `Random.nextInt(int n)` function of Java 1.2.[15] The pseudo-random number generator in Java is extremely compact.

[15] http://java.sun.com/products/jdk/1.2/docs/api/java/util/Random.html. Java uses a linear congruential pseudo-random number generator as described by Donald E. Knuth in *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, section 3.2.1.

True random functions were generated from streams of random bits produced by **www.random.org**, an organization which uses atmospheric data to generate a random bit stream. Their numbers have been tested in various ways to ensure a uniform distribution at various resolutions. The details of their methodology can be found on their web site.

A GA with 10 bit strings was run on 100 different pseudo-random functions with 16 bit co-domain. Such a function can be expressed as a table in 2 kilobytes (1024 entries at 2 bytes each), and there are roughly $10^{4,932}$ functions having this domain and co-domain. The pseudo-random number generator is easily expressible in less than 200 bytes, and is therefore generating compressible functions. For each function, a performance profile was generated based on 1000 runs to convergence. The average of the average run lengths was 1438 evaluations, with a standard deviation of 276. When the same experiment was performed with 100 true random functions, the results were almost identical: the average of the average run lengths was 1405, with a standard deviation of 274. With both true random and pseudo-random, the performance was worse than the expected performance of random search with replacement, and more than twice as bad as the expected performance of enumeration (512.5 evaluations for functions with a single global optimum).

These results can be compared with the results from section 3.3 where 10,000 randomly selected functions (also having 10 bit domain and 16 bit co-domain) were converged with a GA and a hill climber. In that experiment, enumeration performed better than the GA and the hill climber on the vast majority of those 10,000 functions. Because that experiment used a pseudo-random number generator, the resulting functions were all compressible, and yet enumeration was still the clear winner. Furthermore, this sampling of 10,000 was only a tiny fraction of all functions, amounting to a one in $10^{4,928}$ sampling. By contrast, there is estimated to be about 10^{79} atoms in the universe, and

thus even with extremely small samplings of functions (less than an atom per universe), the NFL result can be evident. The above experiment with true random numbers had a sample size of only 100, and yet the results were quantitatively the same. Indeed, in this dissertation, roughly a million pseudo-randomly selected functions were converged, and even though they were in actuality highly compressible, almost all of them resulted in miserable performance for both GA and hill climber.

Because no significant performance difference was found in the previous experiment, another experiment was performed where the function domains were considerably larger. In this case, the GA searched over 15 bit strings, seeking the optimum 16 bit number. There are roughly $10^{157,826}$ functions having this domain and co-domain, and such a function can be described as a table in 64 kilobytes. The pseudo-random number generator will therefore be producing functions at a much higher rate of compression.[16] The GA optimized 100 pseudo-random functions, and 100 true random functions, with each function being run to convergence 100 times. For each function, the average time to convergence was noted, and from those averages, an average over the 100 functions was found. For pseudo-random functions, the average of the averages was 32,190 with a standard deviation of 9121, while for true random functions, the value was 32,040 with a standard deviation of 9361.[17] Once again, the pseudo-random number generator creates functions that are just as difficult for a GA as a true random number generator, even though the functions from the pseudo-random number generator are highly compressible. As in the previous experiment, the GA's performance is far worse than the expected performance of enumeration (16,384 evaluations for functions with a single global optimum).

Because the GA's performance in both of these experiments was so much worse than

[16]The pseudo-random number generator need not be any larger than in the previous experiment – it simply generates more numbers.

[17]The range of averages was from 11,427 to 47,923 for pseudo-random functions and from 8415 to 45,894 for true random functions.

enumeration, it is clear that the NFL curse is not being circumvented, even when using small sets of highly compressible randomly selected functions. As discussed in section 2.6.3, English claimed [2] that NFL consequences may occur whenever the function set to be optimized is "large and diffuse," but the results from this section underscore the fact that the set of functions does not need to be large.

6 Conclusions

A framework was introduced for the study of deterministic non-repeating search algorithms. Population tables were developed to capture the dynamics of all such search algorithms on all functions of a finite domain and co-domain. Many properties of this framework were proven, and the number of deterministic non-repeating search algorithms was established.

All objects associated with the framework were put into canonical form, allowing for tables to be produced and experiments to be performed over all search algorithms of a given size. The canonical form of each object was designed to be maximally compact. Methods were provided for translating an object to its canonical number and back again.

The framework demonstrated its usefulness with a much-needed straightforward proof of the NFL theorem. The framework also proved useful in describing the works of others on the subject, and in extending the NFL result to non-complete function sets and to a meaningful definition of stochastic search. It facilitates an understanding of how a change of representation affects search.

Section 2.7 used performance tables to explore in detail the minimax relation between search algorithms with the unique benefit of being able to compare *all* such search algorithms for a given domain and co-domain. This relation was studied for two performance measures and for many values of N and M. The notion of a minimax equivalence class was introduced, and at each domain and co-domain size studied, all search algorithms were found to be in the same minimax equivalence class.

The expected performance of random search (with and without replacement) and enu-

meration were explored. Commonly stated claims regarding the robustness of hill climbing and genetic algorithms were tested with randomly selected functions, and both search algorithms were shown to perform poorly when compared against random search without replacement. The genetic algorithm even performed worse than random search *with* replacement. The expected performance of random search without replacement serves as an important baseline since no search algorithm is expected to beat it in a black box scenario.

There has been increased criticism of the test suite functions used to determine search algorithm performance. The use of randomly selected functions can provide a good alternative, providing an unbiased sampling of function space, and thereby allowing results to be generalized to the set of all functions. Results from randomly selected functions of a certain property can generalize to all functions having the given property. For these reasons, randomly selected functions were used at several points in this dissertation.

The analytical methods described in section 3 were put to use to determine the impact of compressibility on search algorithm performance. A set of compressible functions was pitted against another set of functions where most elements were expected to be incompressible. A simple genetic algorithm ran each function in both sets to convergence 1000 times, and the overall performance was judged to be the same for both sets, and thus compressibility was not a factor in performance. In accordance with the results from section 3.3, enumeration is shown to outperform a GA even with small sets of randomly selected functions. In short, neither compressibility nor small set size can be expected to save randomly selected functions from the NFL curse.

6.1 Further Research

The framework introduced in section 2 can serve as a foundation for further theoretical work, allowing more properties of population tables and performance tables to be discovered. Because search algorithms are related by row permutations in a population table, minimax relations could possibly be formalized using group theory, which may also be useful in exploring relations other than minimax. Section 2.8 discussed several possible applications of the framework. The field of algorithmic information theory (also known as the study of Kolmogorov complexity) should be able to make contributions to the study of black box search, since it concerns itself with incompressible objects.

The compressibility results could be extended to include randomly selected deterministic non-repeating search algorithms. Such a search algorithm could be generated by randomly selecting a permutation for each new function encountered. It is speculated that the results would mirror the results of section 5.

References

References

[1] Chaitin, G. J., *On the Number of N-bit Strings with Maximum Complexity, Applied Mathematics and Computation*, 59, 1993, pp. 97-100.

[2] English, Thomas M., *Information is Conserved in Optimization*, Computer Science Department, Texas Tech University, 1997.

[3] Goldberg, D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning*; Addison-Wesley: Reading, MA, 1989.

[4] Goldberg, D.E., Deb, K., Horn, J., *Massive Multimodality, Deception, and Genetic Algorithms.* Illinois Genetic Algorithms Laboratory Tech Report 92005, 1992.

[5] Holland, John, from a workshop presentation on evolutionary algorithms at *The Institute for Mathematics and its Applications*, Minneapolis, Minnesota, October, 1996.

[6] M. Li and P.M.B. Vitanyi, *Algorithmic Complexity*, in *International Encyclopedia of the Social and Behavioral Sciences*, N.J. Smelser and P.B. Baltes, Eds., Pergamon, 2001.

[7] Radcliffe, Surry, *Fundamental Limitations on Search Algorithms: Evolutionary Computing in Perspective*, in Jan van Leeuwen, editor, *Lecture Notes in Computer Science*, volume 1000. Springer-Verlag, 1995.

[8] Rana, S., Whitley, D., *Search, Binary Representations and Counting Optima*, Colorado State University Technical Report, 1997.

[9] Rumelhart, D.E., J.L. McClelland, *Parallel Distributed Processing, Explorations in the Microstructure of Cognition*, Volume 1: Foundations, M.I.T. Press, Cambridge Mass., 1986.

[10] Salomon, R., *Some Comments on Evolutionary Algorithm Theory*, in *Evolutionary Computation*, 4(4):405-415, 1996, MIT Press, Cambridge, MA.

[11] Spears, W., *The Role of Mutation and Recombination in Evolutionary Algorithms.* Dissertation, George Mason University, Computer Science Department, Summer 1998.

[12] Vose, M., *The Simple Genetic Algorithm: Foundations and Theory*, MIT Press, 1999.

[13] Vose, M., Liepins, G., *Schema Disruption*, in *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 237-243. Morgan Kaufmann, 1991.

[14] Whitley, D. *A Free Lunch Proof for Gray versus Binary Encodings*, in *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference.* Morgan Kaufmann, 1999.

[15] Whitley, D. *A Genetic Algorithm Tutorial*, in *Statistics and Computing* Volume 4, pp. 65-85, 1994.

[16] Whitley, D., Mathias, K., Rana, S., Dzubera, J., *Building Better Test Functions*, In *International Conference on Genetic Algorithms*, L. Eshelman, ed., Morgan Kaufmann, 1995.

[17] Whitley, D., Rana, S., Heckendorn, R., *Representation Issues in Neighborhood Search and Evolutionary Algorithms*; In Quagliarella, D. et. al. editors, *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pp: 39-57, John Wiley & Sons Ltd, 1998.

[18] Wolpert, D., Macready, W., *No Free Lunch Theorems for Search.* Santa Fe Institute Technical Report SFI-TR-9502-010, 1995.

[19] Wolpert, D., Macready, W., *No Free Lunch Theorems for Optimization*, in *IEEE Transactions on Evolutionary Computation*, Vol. 1, No. 1, April 1997.

Appendix

A NFL Result with Unequally Weighted Functions

One version of the NFL theorem (theorem 14) applies when the overall measure uses equally weighted performance vector measures. It is easy to see how placing greater importance on a set of functions could lead to results contrary to the conclusion of the NFL theorem since search algorithms that did best on the favored functions could have a large advantage over others. However, under certain circumstances, the NFL result occurs even when overall measures are not equally weighted, as the following example illustrates:

When $N = M = 2$, there are only two possible non-repeating search algorithms: either x_0 is chosen first (search algorithm A) or x_1 is chosen first (search algorithm B). There are four possible functions, and without loss of generality we will assume that $y_0 < y_1$. Table 19 shows how each function maps the two points x_0 and x_1, and also shows the number of evaluations required to find the optimum value for each search algorithm. That number of evaluations will be used as the performance vector measure.

Table 19: An example of when the NFL result occurs when the functions are not weighted equally.

	x_0	x_1	Algorithm A	Algorithm B
f_0	y_0	y_0	1	1
f_1	y_0	y_1	2	1
f_2	y_1	y_0	1	2
f_3	y_1	y_1	1	1

The overall measure will be weighted, and the desire is to know where in weight space the NFL result occurs, i.e. where the overall measures for the two search algorithms are equal. The following two equations follow from table 19:

$$\text{overall measure for Algorithm A} = w_0 + 2w_1 + w_2 + w_3$$

$$\text{overall measure for Algorithm B} = w_0 + w_1 + 2w_2 + w_3$$

Setting these two equations equal shows that the NFL result occurs when $w_1 = w_2$, independent of the values of w_0 and w_3, showing that all the weights need not be equal for the NFL result to hold.

Vita

The author is currently working on a project to disentangle corporate influence from the Internet experience through a standardization of abstract objects and by bringing visual programming to the masses. In his spare time, he works on a screenplay. He received a B.A. in Philosophy and Religion at the University of Tennessee in Chattanooga.