A COGNITIVE COMPUTING ARCHITECTURE

A Dissertation Presented

by

John E. Lecky

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
Specializing in Electrical Engineering

May, 1999

Accepted by the Faculty of the Graduate College, The University of Vermont, in partial fulfillment of the requirements for the degree of Doctor of Philosophy specializing in Electrical Engineering.

Dissertation Examination Committee:

________________________________Advisor
Richard G. Absher, Ph.D.

____
Ronald W. Williams, Ph.D.

____
Rebecca McCauley, Ph.D.

________________________________Chairperson
Guo-Liang Xue, Ph.D.

________________________________Interim Dean
Andrew R. Bodman, Ph.D. Graduate College

Date: March 31, 1999

Abstract

This describes a new computer hardware architecture that complements existing computer architectures by making difficult decision problems, such as problems that are NP-complete, easy to solve.

A description of the kinds of problems that are difficult to solve with present-day electronic computers is contrasted and coupled with a broad overview of the anatomy of the human brain. A model for how the brain might solve these problems is proposed, and then a new computer hardware architecture, called *scouting*, is developed which operates based on this model.

Scouting, at its foundations, is a simultaneous forward simulation technique in which many simple independent processors search for solutions in parallel. In the biological analogy, each of these processors would be a group of 100 to 2000 neurons, and 10,000 or more processors would be assumed to exist. This biological model derives from the anatomy of cortical *macrocolumns*.

Scouting simulations are performed on a variety of problems that are challenging to solve using standard "artificial intelligence" computer programming techniques. Scouting performance compares very favorably with these traditional computer methods. Two simulated problems, Garage Door and Stream Crossing, are extracted from real-world daily living situations to illustrate the generality of the technique.

The theoretical statistics of scouting performance are derived and verified experimentally. Related mathematical results are derived and used to provide an accurate estimation of the accuracy and run time of scouting in the solution of any puzzle or problem requiring sequences of decisions.

By coupling scouting with efficient memory storage architectures, a complete cognitive computing system can be developed.

Acknowledgements

This project began in 1988 when Tony Grass, a fellow graduate student, gave me a copy of Susan Allport's *Explorers of the Black Box*. This book gave me my first hint that the gap between computers and the mind might be narrow enough to at least throw a stone across. Much thanks to Ms. Allport, and to Tony, for handing me the stone. This dissertation tries to throw it.

Many people have contributed to the project since then, and they deserve more thanks than I can give them here. A few deserve special mention.

First of all, thanks go to my advisor, Professor Richard Absher of the University of Vermont, who kept asking the difficult questions without dampening my enthusiasm or shaking my confidence in the principle of scouting and its value.

Thanks also go to a very patient and questioning committee team that had to suffer through disorganized drafts and lengthy rewrites. You were my sanity check, and some of the ideas didn't pass the test. Thanks for catching them.

Thanks go to the many authors whose works I read and tried to understand. This is a difficult, multi-disciplinary field, and would have been incomprehensible to me without all of your clear, lucid, insightful thoughts, words, and ideas. Please keep writing.

To my children, Ryan and Alex, special thanks are also due. You have contributed immeasurable insight into the processes of learning and thinking as they happen in minds that are still fresh and open to developing alternatives. You have also watched your father struggle, and spend much time away from you, to finish this project. Thanks for your help.

Last, and most of all, thanks go to my wife Phyllis for support, encouragement, patience, and constant motivation far beyond the call of duty throughout this trial of trying to understand so much that is not yet ready to be understood.

Table of Contents

List of Tables

List of Figures

1 Introduction

For as long as there have been intelligent humans, many of them have wondered about the nature of intelligence. And ever since some of them invented electronic computers, many others have wondered how to build one that had human intelligence. Modern computer technology seems as though it provides all of the components necessary for the construction of intelligent machines. Nevertheless, the realization of a machine with even remotely human intelligence seems far off.

The weaknesses of current computer technology compared to brain technology are most obvious in the following three areas.

1. Memory. Computer memory strategies start with an address. We must know where something is stored to retrieve it. Human memory strategies require only an *association*; some set of properties or hints to trigger lookup. Location of storage is abstract and immaterial.

2. Thinking. Thinking is the generation of new information from old. Computers require algorithms to do this, but the existence of an algorithm implies that all of the thinking has already been done. What is the strategy for generating new information when an algorithm is unknown, or when no algorithm exists?

3. Massive Parallelism. A fly has 1 million neurons. A human has 10 billion. The obvious disparity in intellect between flies and humans suggests that there is something very important about computing with billions, not millions, of processing elements. Most parallel computing systems available today have at most thousands of processors, and it is not clear what number of neurons corresponds to our standard definition of "processor."

This dissertation describes a practical, digital computing hardware architecture capable of "cognition," which can be described as the use of knowledge to solve problems.

The design relies heavily on a biological model for analytical thought that involves the use of many individual processors; these individual processors are analogous to clusters of hundreds of biological neurons. While this digital model may or may not be an accurate representation of the true operation of biological cognitive hardware, there is strong anatomical and physiological support for the architecture proposed herein. In any event, the correctness of the biological model is not a requirement for the usefulness of the digital architecture.

1.1 Overview

A major obstacle to understanding and organizing the material to follow is the lack of knowledge regarding what pieces of the biological design are critical and immutable. Much as Orville and Wilbur Wright were able to advance powered flight much more rapidly once they realized that feathers and flapping wings were not required, this research requires that many details be ignored in order to develop a workable design. The key is in choosing which biological elements to retain, and which to ignore. An early task in this dissertation is to develop this critical subset and to describe it as a practical computer architecture.

The architecture is intended to be general purpose; that is, it should be equally effective at solving toy puzzles, playing checkers, analyzing speech, or figuring out the best way to get across a stream by stepping on dry rocks, much like its biological counterpart.

This requires that carefully crafted "heuristics" or "knowledge-bases" *not* be used to solve a problem. The system must be capable of synthesizing solutions directly from nothing more than a description of the problem and a desired outcome.

One way to construct such a general-purpose machine that does not require special algorithms to solve a new problem is to build an exhaustive search machine. Exhaustive search involves simulating every possible outcome based on every possible sequence of actions. In this way, an unintelligent machine can generate intelligence, which is a very attractive property. Unfortunately, exhaustive search, or "brute-force" computing, has typically only been effective for the simplest of problems.

Two factors suggest that a re-examination of minimally intelligent exhaustive-search-based computing is timely. First, the continued explosion of digital computing power means that many problems unsolvable with brute-force techniques just a few years ago may now be directly solved through the application of more computing horsepower. This justification, however, only serves to move the capability horizon for brute-force computing a few orders of magnitude further out, which in itself is not enough of an advancement to suggest that it qualifies as cognitive computing.

The second, more compelling reason to reinvestigate exhaustive search is driven by revelations in the architectural details of the animal brain. Most animals, especially humans, are capable of solving a variety of problems that digital computers still have a very difficult time with. Further, the learning capability of animal brains is entirely unduplicated in electronic computer architecture, since electronic computers require explicit programming on a per-problem basis, making learning seem a daunting task.

The most interesting development in our understanding of brain architecture is that a large number of relatively small and rather similar processing circuits seem to be responsible for problem solving in most areas, including logic, reasoning, speech recognition, speech synthesis, vision, and motion control. In humans, for example, a network of about 10^6 *macrocolumns*, cylindrical processors made up of 2000 neurons or so, comprise the outer covering of the brain known as the *neocortex*. The neocortex is responsible for our most sophisticated processing.

It is in front of this backdrop of much-accelerated computing hardware and a higher-level model for neocortical operation that this dissertation will proceed. A new computing architecture will be proposed that is a combination of a massively parallel biological macrocolumn-based hardware architecture and an intelligent exhaustive-search software methodology with the goal of describing a digital computer with capabilities much closer to those of the brain.

1.2 Material Flow

The chapters to come are organized as follows.

Chapter 2 develops background in the computer sciences of algorithms and puzzles.

Chapter 3 examines the neurobiological basis of thought, as much as it is known for certain today.

Chapter 4 describes the new computing architecture, and is followed by a mathematical effectiveness analysis in Chapter 5.

Chapter 6 examines the application of the cognitive computing architecture to additional problems, and Chapter 7 summarizes and proposes future research topics.

2 Algorithms and Puzzles

This chapter motivates the development of alternate computer architectures specifically designed to solve problems that current computers find *intractable*, or extremely difficult to solve. Cognition is often an "aha" process; that is, one in which the answer is obvious once it has been proposed. This property bears a strong resemblance to similar properties in class of problems known as *NP-complete*, long studied and debated in the realm of computers and tractability.

This chapter will examine this similarity and its implications.

2.1 Algorithms

Most early texts on computer science define the term *algorithm* in a way similar to the following:

> An algorithm can be defined as a sequential procedure consisting of a finite number of unambiguous instructions or actions working on some initial data or information and producing some final result.[1]

Robert Sedgewick, a well-published authority on algorithms, has more recently put forth a quite different definition:

> The term algorithm is used in computer science to describe a problem-solving method suitable for implementation as computer programs.[2]

This second definition is probably much more accurate (although arguably less correct from a theoretical standpoint). The implication of this definition is that *the acceptability of a problem-solving strategy depends upon the computing equipment it is intended to run on*.

[1] Polivka (1975), p. 90

[2] Sedgewick (1990), p. 4

For example, typical computer CPUs perform the following basic sets of operations:

1) Arithmetic (add, subtract, multiply, divide, multiply-accumulate, etc.)

2) Logical (AND, OR, XOR, NOT, etc.)

3) Comparison (equal, greater, less, etc.)

4) Conditional branching (branch if greater, branch if less, etc.)

5) Architectural control (subroutine management, interrupt management, etc.)

6) Floating point (mathematical and trigonometric operations)

Many of these basic operations (multiply-accumulate, or floating-point addition, for example) actually require algorithms for their implementation. Once the algorithm is built into microcode or otherwise permanently captured in the CPU, however, what was formerly a high-level operation now becomes a single deterministic step.

Algorithms must be made up of deterministic steps. By *deterministic* is meant that the next operation of the algorithm is completely specified by its current state. In other words, no "guessing" or "miraculous" steps are required. An example of a *non-deterministic* step would be:

Step 1: Choose the largest prime number evenly divisible into an input value

The reason that this is a non-deterministic step is that the operation of factoring a number into primes is itself a difficult problem. Since we have no FIND-LARGEST-PRIME machine instruction, we need to implement prime factorization as an algorithm. For large enough numbers, this process could take years of processing time.

It seems that the definition of "non-determinism" depends on the computer one is imagining. If we could imagine a better machine, more operations would be deterministic.

2.2 Algorithmic Complexity

An enormous number of "problems without good algorithms" exist, and it is both frustrating and exciting that many of these problems have been shown to be equivalent, and require what we would call "intuition," "insight," or "intelligence" to solve. In no small sense, the definitions of these terms can be thought of in a new light: *intelligent behavior is nothing more than verifiably correct behavior, the origin of which we do not understand*. Once we understand intelligent behavior, it no longer "seems" intelligent. For this reason, a brief look at complex algorithms is in order.

To differentiate between a "good" algorithm and a "bad" one, it is customary in computer science circles to measure efficiency in terms of the increase of algorithm run time corresponding to a particular increase in the size of its input.

A *polynomial time* algorithm is an algorithm whose worst-case run time is bounded by a polynomial function of the number of bits used to encode the input efficiently. An example of a simple polynomial-time algorithm is matrix multiplication. Multiplying two $n \times n$ matrices takes n^3 multiplications and $n^2(n-1)$ additions. A more complex example is multiplication of two polynomials of degree n; this can be done with about $n^{1.58}$ multiplications, or better[3].

The fact that an algorithm is "polynomial" isn't necessarily a sign that no more study is needed; polynomials grow plenty fast, and many polynomial algorithms become intractable even for reasonably sized problems. Problems without polynomial or better algorithms, however, are usually the ones labeled *intractable* since their growth in complexity is staggering.

[3] Sedgewick (1990), p. 529

For example, consider finding the largest prime number evenly divisible into a given input. One algorithm involves successively dividing the input by smaller and smaller numbers (starting with one-half the input number) until no remainder is obtained. This divisor must then be inspected to see if it is a prime by dividing it by smaller and smaller numbers and verifying no remainder. For n input digits, the number of divisors to be considered is $O(10^n)$ ($O(x)$ signifies "on the order of x"), which is exponential in n, not polynomial. This step, which is the key step in prime factorization of a number, is considered completely intractable on today's computers for numbers more than a few hundred digits long. In fact, this intractability is presumed in much contemporary encryption technology.

Another example is the famous Traveling Salesperson problem. In this problem (which has many equally problematic variations), a hypothetical sales person must find the shortest tour of n points in a plane. Years of research suggest that there is no algorithm guaranteed to find the best solution except exhaustive search through $O(n!)$ possible tours. The occurrence of $n!$ (which is the product of all integers from 1 to n) is quite commonplace in the realm of intractable problems.

2.3 P and NP

It is customary to define the world of "reasonable" problems as *P*:

> P: The set of all problems that can be solved by deterministic algorithms in polynomial time[4]

Unfortunately, many problems do not seem to have polynomial time algorithms.

[4] Sedgewick (1990), p. 634

Some or all of these problems would be solvable in polynomial time, however, if we had a non-deterministic machine. The logic for developing these solutions is:

1) (Magically) guess the correct solution

2) Verify that this solution is correct and optimal

Problems for which the verification step can be performed in polynomial time are called *nondeterministic polynomial* (*NP*):

> NP: The set of all problems that can be solved by nondeterministic algorithms in polynomial time[5]

It is clear that any problem in P is also in NP.

The classic example of a problem in NP is satisfiability. This problem starts with a Boolean equation of the form:

$$(x_1 + x_3 + x_5)(x_1 + \overline{x}_5 + x_4)(x_2 + x_3 + x_5)$$

Satisfiability asks if there is an assignment of logical values to the set of $n \; x_i's$ such that the equation is true. The standard solution is to proceed with generation of all 2^n input combinations until either a solution is found, or all possibilities have been exhausted. The number of inputs to apply is thus exponential, far worse than any polynomial.

It is interesting to note that no one has ever been able to prove that any problem is in NP but not in P; this suggests that there may be a polynomial time algorithm lurking out there that solves satisfiability. The theory of NP-completeness makes this extremely unlikely, however.

[5] Sedgewick (1990), p. 635

2.4 NP-completeness

The likelihood of finding a "good" deterministic algorithm for a problem in NP is extremely small. S.A. Cook proved that satisfiability is NP-complete in 1971. This means that if a polynomial time algorithm can be found to solve satisfiability, then all problems in NP can be solved in polynomial time. This very strong result forms the core motivation for the abandonment of the search for better algorithms and the development instead of different computing methodologies. The following problems, along with literally thousands of others, have been shown to be NP-complete:[6]

PARTITION: Given a set of integers, can they be divided into two sets each of whose sum is equal?

INTEGER LINEAR PROGRAMMING: Given a linear program, is there a solution in integers?

MULTIPROCESSOR SCHEDULING: Given a deadline and a set of tasks of varying length to be performed on two identical processors, can the tasks be arranged so that the deadline is met?

VERTEX COVER: Given a graph and an integer N, is there a set of fewer than N vertices which touches all the edges?

2.5 Solving NP-complete Problems

Attempts at "artificial intelligence" run into NP-complete problems almost continuously. Often, a clever attack can allow solutions to at least moderate instances of NP-complete problems. These approaches fall into three categories:

[6] Sedgewick (1990), p. 639

1) Approximation Algorithms. Polynomial algorithms can often be developed which can be shown to generate optimal solutions to a simplified problem. The discovery of such algorithms is very time-consuming, their performance is inconsistent and data dependent, and their applicability is typically limited to very specific problems.

2) Average Case Treatment. By ignoring worst-case situations, polynomial *heuristics* can sometimes be found which *usually* generate optimal solutions. Again, this technique tends to be extremely application-specific, and the possibility of occasionally finding no solution at all is quite troubling.

3) Efficient Exponential Algorithms. This requires the development of "branch-and-bound" algorithms which attempt to traverse the exponential tree as efficiently as possible. While such algorithms are still exponential, not polynomial, they can be effective in allowing traditional computers to attack much larger instances of the exponential problems.

This third area has given rise to forward checking, backtracking, iterative deepening, A* search, IDA* search, and a very ric h array of techniques, tricks, and variations on themes which vastly increase the size of problems that can be approached with traditional sequential computers. However, these techniques merely postpone the inevitable, since a large enough instance of a exponential problem cannot be solved in reasonable time.

Unfortunately, the kinds of problems a cognitive computer would be expected to solve are almost universally exponential, and often of large size. This is explored in the examples below.

2.6 An Illustration; 8- and 15-Puzzles

Let's consider the problem of solving an 8-puzzle. This is a toy puzzle in which eight tiles are set in a 3x3 grid in such a way that tiles adjacent to the one blank may be slid into the position of the blank. The goal is to get the eight numbered tiles into some desired state. Consider the following goal:

1	2	3
8		4
7	6	5

With the following initial state:

5	4	7
1	2	3
6	8	

The first move must be to move either tile 3 down (D) or tile 8 to the right (R). Instead of calling this 3-D or 8-R, we can focus on the motion of the empty space at each turn and just call the move Up (U) or Left (L). This considers the move to be the movement of the space either into the square occupied by tile 3 or tile 8, respectively. After this, three moves are possible from either new location. A move tree for the first three moves can be built up as follows. (Note that the "complete" tree has infinite depth and breadth!)

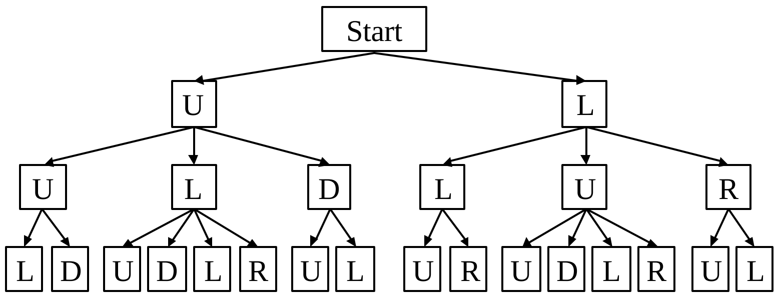

Figure 1: 8-puzzle Search Tree for Three Moves

Even in a simple game like the 8-puzzle, the variety of possible moves is enormous. Since two moves are possible when the blank is in any of the four corners, and three moves are possible from each of the side midpoints, and four moves from the center position, the average number of possible moves is $\frac{4 \cdot 2 + 4 \cdot 3 + 4}{9} = 2.67$. Thus, a puzzle that takes 20 moves to solve has a tree with $2.67^{20} = 3.39 \times 10^{8}$ paths. The actual number of starting situations is easy to compute. The first position can be occupied by any of 9 tiles, including the blank. The second position can be occupied by any of the 8 remaining tiles. Continuing in like fashion, the total number of 8-puzzle situations is seen to be $\frac{9!}{2} = 1.81 \times 10^{5}$. The division by 2 is required to preserve parity; since each move really is an interchange of the blank with a numbered tile, it is impossible to move an odd number of tiles! Thus, a desired final state may only be reached from ½ of the number of possible starting configurations.

A nice simplification arises from realizing that it is useless to move U followed by D, or L followed by R, since these move combinations simply return the puzzle to its most recent configuration. Such operations can be called *unmoves*, since they undo the effect of a preceding move. More sophisticated search algorithms attempt to extend this notion further by cataloging *all* previously visited states, and insuring that no state is ever visited twice. The relative expense of this operation makes its usefulness extremely situation-dependent.

At any rate, eliminating unmoves in an 8-puzzle solution space cuts the tree branching factor from 2.67 to 1.67, since moving back in the previous direction is always eliminated. This means that a 20-move solution can only proceed down $1.67^{20} = 28466$ paths.

The situation rapidly deteriorates with scaling, however. A more complex version of the 8-puzzle is the 15-puzzle, in which 15 numbered tiles are set in a 4x4 grid. The rules are the same, although the complexity of the puzzle is much greater than is that of the 8-puzzle.

Since two moves are possible when the blank is in any of the four corners, and three moves are possible from each of two side midpoints, and four moves are possible from each of four the central positions, the average number of possible moves is $\frac{4 \cdot 2 + 8 \cdot 3 + 4 \cdot 4}{16} = 3$. Also, 15-puzzles take more moves to solve than 8-puzzles. A 45-move solution, which is not atypical, would have a tree with $3^{45} = 2.95 \times 10^{21}$ paths. The actual number of starting situations is $\frac{16!}{2} = 1.05 \times 10^{13}$. The division by 2 is again required to preserve parity.

2.7 *Another Illustration: The Garage Door Problem*

The "garage door problem" is a more typical real-world problem. This problem requires fewer moves to solve than an 8-puzzle, but is confounded by an enormous branching factor. This is more common in real-world human situations, where the solution may only require two or three steps, but the number of possible operations to choose from at each step is large.

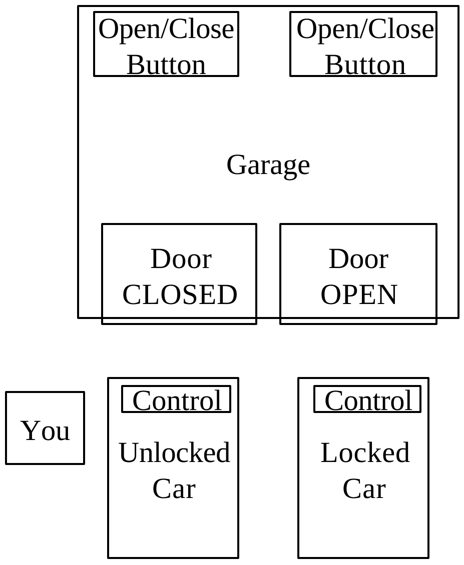

Figure 2: The Garage Door Problem

The problem, which comes from the author's own personal experience, is described as follows. Two cars are in the driveway, each in front of its own garage door leading into a single large room. Each car contains its own garage door controller clipped to the sun-visor. There are two pushbuttons inside the garage allowing either garage door to be opened or closed from inside.

You are standing next to the car on the left. The garage door on the right is open. The car on the right is locked, and you do not have the key. The goal is to close both garage doors before you back out of the driveway in the unlocked car and go to work.

A simple solution, once the problem is understood, is obvious.

1) Walk into the garage through the open right door.

2) Press the right open/close button to close the right garage door.

3) Walk over to and press the left open/close button to open the left garage door.

4) Walk out through the left garage door and get in the left car.

5) Press the control button on the sun-visor to close the left garage door

6) Drive away.

Categorizing the allowable actions at each step in terms of the following may approximate the complexity of this solution:

A) Walk to: left car, right car, left open/close button, right open/close button

B) Push nearest button

This isn't fair, however, because we have made the list of allowable operations using knowledge of what operations are required to solve the problem! Other operations that must be considered are:

C) lock/unlock nearest car

D) Get in/out of nearest car

E) Adjust mirror

F) Sweep garage floor

G) Etc., etc., etc.

In fact, there may be 25 or more potential operations that can be taken at each step. Knowing that the solution requires approximately 6 steps, the total number of sequences of operations that might yield a solution is $25^6 = 2.4 \times 10^8$.

As is common in cognitive problems, there is also plenty of room for innovation and unexpected solutions. For example, solutions involving closing the garage door from inside and then quickly running out under the door as it closes can save some steps. The challenge in the exhaustive search-based solution model to be proposed will be to select a reasonably efficient solution from this vast sea of potential solutions quickly and reliably.

2.8 Generalization to Games

The problems considered above are puzzles. The difference between *puzzles* and *games* is that in a puzzle there is only one brain involved, while in a game there are two or more. Games imply the notion of adversaries.

Games can be roughly partitioned into two major categories:

1. Games in which chance plays no role

2. Games in which chance plays a role

Games in which chance plays a role typically require dice, shuffled cards, or some sort of randomizing device to inject the uncontrolled element of "chance" into the game.

Game play introduces a new element into thinking, since successful game play requires prediction of the actions of an opponent. The most likely way this is handled is to "put one in the place of" the other player, and try to select their countering moves based on an expectation of their style of play. This activity in itself may be cast as a puzzle, as will be shown shortly.

2.9 Situation, Rules, and Goal

Many problems in computer science are simplified, or even trivialized, by the appropriate choice of data structures for their representation. This benefit comes from optimizing the data representation for the hardware on which it will be analyzed.

Since the brain is an efficient computing machine, it is likely that the representation of data in the brain is particularly well suited to the manner in which it computes. An analysis of how one learns to play a game, then, should provide valuable insight into how the brain thinks.

When a child is taught to play a game, she will automatically ask the same three questions in one way or another. These questions cover the following three areas.

1) Situation. What the game looks like.

2) Rules. Allowable moves or operations.

3) Goal. What the object is.

Once all three are known, we are ready to play the game. Perhaps our level of play will not be very advanced, but that is improved over time by developing more sophisticated ways of analyzing the situation, or by adding rules of thumb or experience, or by adding subgoals or intermediate states that we try to attain.

The situation, rules, and goal (SRG) description of a problem is explored in more detail below.

2.9.1 Situation

The situation of a game is the board, pieces, or other symbology arranged to represent a game situation. In chess, this involves understanding the 8x8 grid square of the board, being able to recognize each piece, and being able to differentiate between white pieces and black pieces. In an 8-puzzle, this involves being able to read the symbols on the tiles, and seeing their respective positioning relative to one another.

2.9.2 Rules

Every game has rules, which may also be thought of as obstacles or impediments. In chess, the rules consist primarily of declarations governing how each piece may move, how play alternates, and a couple of special case rules, like castling and queening. In 8-puzzles, the rules include just the restriction of motion to tiles adjacent to the blank.

2.9.3 Goal

The goal is the solution of the puzzle or victory in the game. In chess, this is checkmate, with a strong warning about the dangers of stalemate. In the 8-puzzle, it is simply a picture of the desired location for the 8 tiles in the final position.

It seems that armed with an understanding of these three areas, humans can begin solving a puzzle or playing a game immediately. That is:

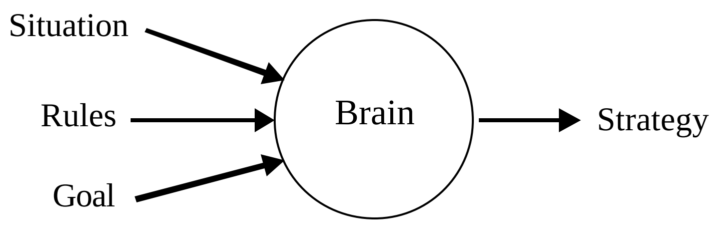

Figure 3: Synthesizing Strategy

It is highly likely that the brain has a problem-solving machine that synthesizes strategy from the SRG representation of a problem. If it did not, one would require more information to be able to play a game.

2.10 Example: Casting a Game as a Puzzle

Consider the simple game of tic-tac-toe. The problem of making a move in this game can be described as a puzzle by using the SRG description below:

1) Situation. A 3x3 grid containing Xs and Os
2) Rules. For each step, place an X and an O in two unoccupied squares. If only one unoccupied square is available, place an X in it.
3) Goal. Get three Xs in a row across, up-and-down, or diagonally, before getting three Os in a row.

If this puzzle is solved many times and statistics are kept on the probability of the goal being achieved given each possible X placement, then the first X placement with the highest probability of success is the best move.

Since a game is just a sequence of puzzles, any computer system that is adept at solving puzzles will also be useful for developing game strategy. Furthermore, nearly all real-world cognitive tasks can be reduced to puzzle or game descriptions, as we have seen from examination of cognitive problems like the garage door problem.

The next chapter will examine directions in neural architecture and computer-based neural networks, and develop a more sophisticated, modular model for neural computing systems. This model will be particularly well suited for implementation in digital hardware.

3 The Biological Side of Computing

A machine is explained in terms of physics and chemistry plus (at least) an engineer. An organism is described in terms of physics and chemistry plus (at least) evolution through natural selection.

A.G. Cairns-Smith[7]

In engineering, it has been long known that successive enhancement and augmentation to a well-planned original design eventually lead to inefficiency, "legacy" anachronisms, and archaic structures. Great engineering designs require a periodic redesign "from scratch."

Since evolution began, there has not been an opportunity for a from-scratch redesign of the human brain. There simply is no biological mechanism for such an undertaking. The same biological devices selected millions of years ago to solve the problems of that age are present in our cells today.

If our cellular architecture is as it is primarily due to historical accident, our behavioral history is even more problematic. Modern humans are "the dazed survivors of a continuous, 5-million-year habit of lethal aggression."[8] This is not social commentary; it is just our history. On the positive side, realizing this provides a way of exploring and analyzing the mind that may be more in line with the actual design principles that were in play during its evolution (at least, up to modern times).

Most of the evolutionary change in the human mind was selected to insure survival during millions of years of upheaval, chaos, and savage competition. Our present behavior and

[7] Cairns-Smith (1996), p. 55

[8] R. Wrangham and D. Peterson, *Demonic Males*, as quoted by DeBecker (1997), p.43

"intellectual" pursuits (like the development of this dissertation, or even just reading the Sunday paper) are so far removed from 99.99%[9] of our developmental history that it would seem naïve to use our present behavior as a model for describing the operation of the brain.

That we can engage at all in more "intellectual" pursuits with the hardware at our disposal is a tribute to the generality and extensibility of the system. This fact, however, likely indicates that research on how *modern* humans think, from a behavioral standpoint, cannot reveal much about how the brain really works.

The neuron is not so much different from the other cells in our bodies. The only really new idea is "long distance telegraphy,"[10] the use of action potentials to rapidly transmit information from one place to another. This form of communication is only in addition to, and not in place of, the standard chemical communication that constantly occurs between cells.

How is it that with not much more than this idea and a massive interconnection of these neurons, we have evolved an intellectual machine of such formidable capacity? The remainder of this chapter will look at the brain from the biological side and build up a model that makes sense from both the presumed complexity of human intellectual capacity and the certain simplemindedness of our evolutionary past.

---

[9] Humans have been engaged (sporadically) in intellectual pursuits since around the time the ancient Sumerians began trading in their farming villages, ca. 5000 BC. Human evolution didn't really start until the mammalian explosion post-dinosauria, 65 million years ago. Thus, we've only gotten intellectual in the last 0.01% of our developmental history.

[10] As noted by Cairns-Smith (1996), p. 122

3.1 Measuring Intelligence Using Puzzles and Games

Human intelligence is quantified through one's ability to visualize and to solve puzzles. Visualization and puzzle solving form the bulk of any IQ test; even word associations like apple : tree :: grape : vine are only the lingual equivalent of visualization as well as a form of puzzle. As will later be examined, much of the human visualization capability is driven by complex image processing filter banks that exist in the lateral geniculate nucleus of the thalamus. These filter banks take the raw visual data and generate many alternate forms in which edges, motion, and color variation are alternately emphasized and suppressed. Taking these various forms of image data and recognizing them as a particular object, say, is really just a formulation of a type of puzzle in which one simultaneously tries to fit all of the scene information with various known models to find the best fit.

For this reason, focusing on puzzle solution is a reasonable way to explore the deepest core of cognitive behavior. If one could build a general-purpose machine that was exceptionally good at solving arbitrary puzzles, then it could be used to solve other puzzles, play games, and perform real-world cognitive tasks.

But there is also another reason to study human puzzles and games. They provide a window into the strengths and weakness of the human mind, since the puzzles and games that we play are invented in other human brains.

For example, adult humans quickly tire of a child's games like tic-tac-toe, since it doesn't take long for one to become adept at playing to a draw every time. Not many adults would enjoy playing tic-tac-toe for recreation.

The game of chess is closer to the other end of the spectrum. Chess is "hard," and millions of players play it only passably well. A large number of people do play it extremely well, but a special-purpose computer currently plays it best. It is doubtful that any human will ever again regain the top seat in chess play, since it has been shown that a great way to play chess is to simply consider the outcome of as many different move combinations as possible, selecting the one that is most promising.

If there are “easy” games, like tic-tac-toe, and “hard” games, like chess, are there “impossible” games? Not too many exist, since humans don’t invent games that are too hard; we would be frustrated by our inability to play. But an example can be easily inverted. Imagine a game called “morph chess,” in which each piece has an identical shape, but has rules of movement that are dependent upon the last three game squares visited by the piece. Furthermore, all pieces have the same color, and it is the job of each opponent to remember which pieces belong to which player.

Human brains are just not *smart* enough to play this game. “Smartness,” in this sense, seems more to be a hardware limitation, not an architectural or software limitation. In fact, a larger brain with a little more temporary or scratch-pad storage would probably be able to play morph chess quite competently.

The justification for our lack of this scratchpad memory is that solving problems like the garage door problem is a much more crucial skill to our survival than would be solving problems like the 8-puzzle or playing morph chess. But since the brain is so well designed (or well-selected, perhaps) for solving the garage door problem, this puzzle seems boringly easy and is no challenge. This boredom indicates, however, that the garage door puzzle is exactly the kind of thing that brain architecture is most suited for.

The implications of this realization on the construction and behavior of a synthetic *neural network* are great. Unfortunately, most neural network research so far has looked in different directions.

3.2 Traditional Neural Network Research

Since we know that the brain is made up of cells called *neurons*, it has long been tempting to create intelligent machines by assembling or simulating small networks of synthetic neurons. This motivation has influenced a large part of the history of artificial intelligence research.

Unfortunately, most such research uses a homogeneous neural model that is rather trivialized. It has been discovered that the brain is made up of tens or hundreds of different types of neurons with various "personalities." Though made from the same cellular components, these neurons have different personalities in much the same way that various digital logic gates have different personalities in spite of their all being made from transistors. In addition, the *synapses*, or interconnections between neurons, are equally varied and demonstrate hundreds of behaviors that also have their own logic-gate-like personalities.

The synthetic neuron of the literature is almost always a much simpler construct that simply sums a set of *n* inputs multiplied by a set of *n* weights, and which generates an output that is a non-linear function of this sum. This model is incapable of memory, oscillation, hysteresis, or logical combination of synaptic inputs, all of which are fairly commonplace in real neuronal behavior.

The traditional neural network literature is full of ideas and variations on a basic theme; start with an arbitrary neural network, apply known inputs, compare to expected outputs, and try to "nudge" the synapse weights to decrease the error between expected and received output. These techniques are grouped under many categories, but can be generically called *back propagation* techniques.

The problem with any such tuning procedure is that the outcome must be known before the learning can begin. This is an insurmountable paradox for a cognitive machine capable of learning on its own.

Nevertheless, there has been considerable success in the area of tuning simulated neural network weights to perform various pattern classification functions. Using *supervised learning*, a sequence of input patterns is applied to a proposed network in conjunction with a sequence of desired outputs. The difference between the actual network output and the desired network output for each input/output combination is propagated backward through the network in such a way as to suggest modifications to the network weights.

In practice, many factors influence how quickly, and how well, the network develops the ability to generate all of the expected responses. Many scenarios set up oscillations that prevent the network weights from ever reaching useable values.

These experiments also generally rely on small, largely feed forward networks. Extensive networks, with many simulated neurons or many interconnections, or extensive feedback paths quickly overwhelm standard simulation hardware. Networks with more than a hundred or so neurons become very unwieldy, and are exceptionally slow to train.

If humans learned this way, it seems that it could take a lifetime to learn how to stand upright. Gerald Edelman, Nobel laureate in physiology and medicine, has said

> Whatever the mind is, it is an ensemble property. That is to say it is a property of a collection of things, a very large collection of things, just as, for example, temperature is an ensemble property of molecules.[11]

Imagine trying to understand the property of temperature by the study of a single molecule. It would be as futile as trying to deduce computer architecture by studying an AND gate, or worse, a single transistor. When we think in terms of individual neurons, or even small clusters of neurons, we make the task of deducing overall brain operation extremely difficult.

---

[11] As quoted by Allport (1986), p. 47

The key to understanding the operation of the brain, the most important real neural network, will come from a top-down analysis that results in an architectural model. The remainder of this chapter will first take a better look at neurons and the brain, and then develop that model.

3.3 The Neuron

It is estimated that there are about 10^{11} neurons in the human brain. Since many have interconnects that are tens of centimeters long, and since their interconnectedness is so prolific, it is likely that no neuron is more than six or seven network hops away from any other.[12]

The whole brain is composed of more like 10^{12} cells, with the 90% that are not neurons consisting of insulating cells, blood and neurotransmitter transport networks, and perhaps a communications and control network. The exact behavior of this "other 90%" of the brain seems relatively simple compared to neural activity, but it might not be. Specific answers are not yet available. The existence of some independent control network that coordinates groups of neurons without signaling across the synapses is quite likely, however.

An individual neuron might have several thousand *axons* from other cells inputting signals into it; a general rule is that the receipt of just a few messages from other neurons is unlikely to make a neuron fire. However, different signals get different weights. Certainly, axons connecting closer to the cell body have a better chance of making the neuron fire, much like the loud voices at the front of the room count for more "votes" in a voice vote. Cairns-Smith, for this reason, introduces the (still oversimplified) idea of the "Union Neuron," in which the neuron takes a voice vote to decide whether to fire.[13]

[12] Baars, B.J. *A Cognitive Theory of Consciousness*, quoted by Cairns-Smith (1996), p. 99

[13] Cairns-Smith (1996), p. 113

It is tempting to think of the neuron as an inquiry/response kind of system: we apply inputs, wait, and then receive (possibly) an output. But there are many kinds of neurons, and each behaves differently. Many spontaneously active neurons have been found in both vertebrates and invertebrates, prompting physiologists to give them names like bursters and oscillators.[14]

Much of our understanding of individual neuronal activity is due to the Nobel Prize-winning work of Alan Hodgekin and Andrew Huxley, two English neurophysiologists who in 1939 developed a wire electrode fine enough to be threaded into a squid's largest axon, opening up the possibility for study of neuron response to synthetic stimuli. Their work, and the research their invention permitted others to perform, triggered development of a view of the neuron as a summing circuit, which somehow prepared a weighted sum of all of its inputs, and then fired, or did not fire, based upon the total sum. This early model of the neuron was the one that propagated over to synthetic neural network research without receiving much subsequent enhancement.

And this view is woefully oversimplified. The electrical behavior of the neuron is based upon its ability to build up, and discharge, electric potential, a capability provided by the selective, controlled passage of sodium and chloride ions through various parts of the neuron membrane. Hodgekin and Huxley only hypothesized the existence of one trans-membrane channel for each ion; a *charging* channel, and a *discharging* channel.

Since 1939, however, researchers have discovered at least ten additional ionic channel types, and most agree that many more await discovery. The electrochemistry of these individual channels is not important to this dissertation. Their effect, however, is extraordinary. Says Allport:

[14] Allport (1986), p. 57

> They are indeed responsible for the fact that one neuron is a spontaneously bursting cell that produces rhythmic bursts of action potentials without any synaptic input at all, while another is a silent cell that requires a long, slow stimulus before it will fire. They are also the reason that cells react to an injection of the same amount of positive current in quite different ways: one neuron might respond with a long train of action potentials, while a second shows no response at all; a third cell might bark – or fire - once, then remain silent.[15]

Up through the 1960s, understanding the brain was thought to be a wiring game; if you could map out which neurons were connected to which, you could deduce the function of the brain. But the continued discovery of new, complex, and ever-more clever ionic channels in neurons began to make the emphasis seem like it should be more on the neuron, and less on the interconnects.

Gerald Fischbach, former president of the Society for Neuroscience, says, "One of the great embarrassments in neurobiology is that we don't know whether there are thousands, millions, or billions of electrophysiologically distinct neurons."[16]

From a digital electronic standpoint, it is easy to think of the neuron as a generic logic gate. We have many different kinds of specific gates, with different properties, AND, OR, INVERT, One-shots, oscillators, flip-flops, and so forth. We can have edge-sensitive inputs, as well as open-collector or high-impedance outputs. It appears that by building the right ionic channels into a neuron, we could construct a biological version of almost any digital logic gate in common practical use. In fact, the biological neuron is much like a complex digital logic macrocell in which internal behavior is extremely varied and programmable, while its inputs and outputs can

[15] Allport (1986), p. 164

[16] As quoted by Allport (1986), p. 168

be gated together logically with or without feedback to perform a rich set of core “library” functions.

There are some interesting developments in the area of neural interconnection, as well. Neural interconnections occur at synapses where individual neurons come into close physical contact, but do not really touch. The transfer of information is carried out through the release of chemical neurotransmitters from one side of the synapse, and their attachment to receptors on the receiving neuron.

In the 1970s, it was thought there might be three or four such chemical messengers. Today, researchers suspect the existence of hundreds, each of which has special properties. Some remain in effect for as little as 10 mS, while others can linger for minutes, hours, or even days. Some are inhibitory, some are excitatory, and others seem to have a regional effect over any number of cells, as long as they have the right receptors.[17]

There is no question that the neuron and its myriad capabilities and behaviors is an incredible model of efficiency and power. But the final analysis is that the neuron, while spectacularly flexible and reconfigurable, contains no magic wand, and exhibits no miraculous properties. There is no neuronal or synaptic behavior that cannot be duplicated with a fairly simple configuration of digital logic gates.

Since neither neuron nor synapse provides the answer to how cognitive computing works, blindly assembling groups of them and measuring their behavior will not tell us any more than would the analysis of a random interconnection of digital logic. The next section will examine in detail the higher level design of the brain, for it is here that the answers must lie.

[17] Paraphrase of Allport (1986), p. 169

3.4 The Brain

The following evolutionary discourse is largely due to Dr. Graham Cairns-Smith in *Evolving the Mind* (1996). In this book, Dr. Cairns-Smith provides an illuminating, insightful, readable and altogether enjoyable look at the physics and biology of human brain evolution.

The human brain is a highly evolved organ. This is to say that the brain is built from successively newer structures, and includes ancient places, and modified ancient places, as well as very new places.

Neurons were first evolved to handle reflexive actions in simple animals. They were arranged in small groups called *ganglia,* and interposed directly between an input sensor and an output motor neuron. A few neurons could easily be wired up to, for example, automatically retreat a snail into its shell when something touched it.

Over time, evolution created more complex ganglia that expanded into a rudimentary spinal cord. The spinal cord contained an ever-growing library of autonomous reactions that helped the evolutionary survivors stay alive. The brain, a coalescence of many ganglia, started as a swelling at the top of this simple cord, eventually dividing into a *hindbrain*, a *midbrain*, and a *forebrain*. The forebrain later subdivided into two districts, the *telencephalon* and the *diencephalon*, , to give a set of five regions:

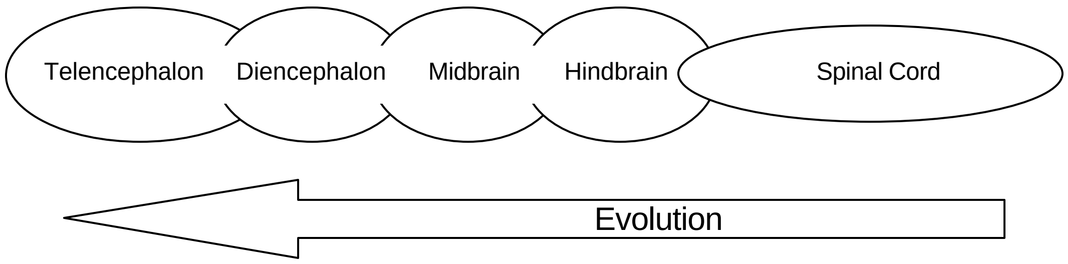

Figure 4: A Brain In Five Parts

Finally, a more complete structure evolved:

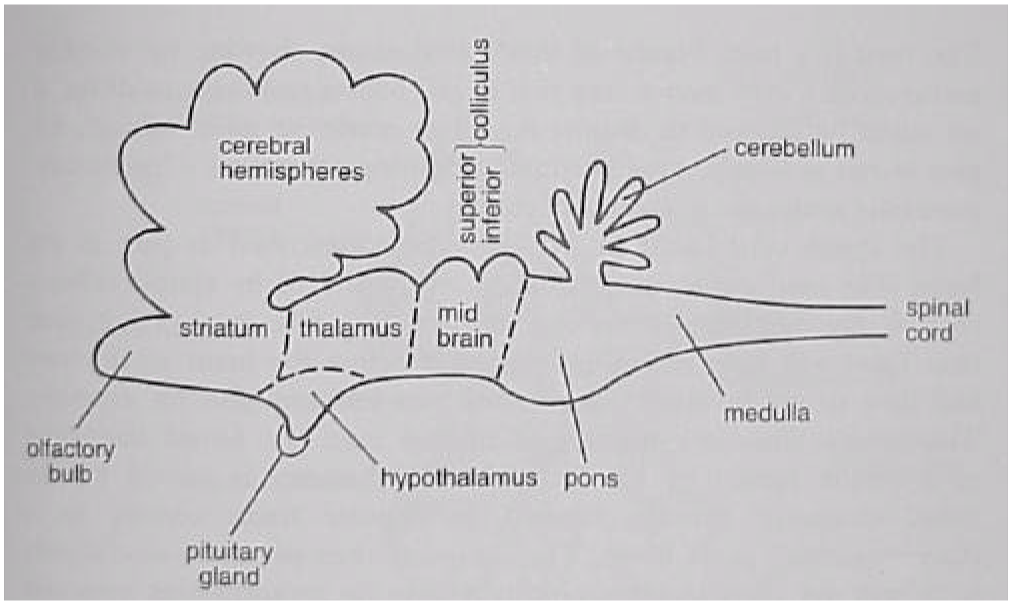

Figure 5: A More Complete Brain Schematic

from Cairns-Smith (1996), p. 129

This diagram is worth describing in more detail, working from the spinal cord to the most recent structures.

Spinal Cord. This is bilaterally symmetrical, and segmented, in that bundles of nerve connections repeat along its length. Here, the white matter (long connections) is on the outside, and the gray matter (processors) is on the inside. Most processing here is restricted to tight closed-loop reflexes, although the more intricate parts of balance necessary for walking are automatically handled here as well. (We couldn't walk if our motor updates were only 3 or 4 per second.) Also, the spinal chord ganglia processors insure that our finger is pulled away from the candle long before we register any pain.

Medulla. Remnants of the hindbrain are here. This is where most upper sensory and control connections are made, including hearing and control of face and eye muscles. More importantly, control of the major organs, including heart and lungs, takes place here. This is a background process that is controlled and monitored perfectly well, even after massive damage to the cerebral hemispheres. The medulla allows someone who is cognitively dead to remain alive in a vegetative state, hopefully while the remainder of the brain heals.

Pons. This is a bridge-like swelling that may be thought of as a giant junction box, like the main distribution frame in a telephone office where thousands of telephone wire pairs are organized and connected to a massive switch.

Cerebellum. This is the first area of truly higher order processing. It may be thought of as the first draft of the cerebral hemispheres, and is where "clever," non-reflexive motor functions like riding a bike or juggling are controlled. It is the place where unconscious activity or expertise is controlled.

Inferior colliculus. This is the earliest part of the midbrain and is concerned mostly with the processing of auditory information.

Superior colliculus. This contains some basic vision functions, such as lower order image processing and coordination of eye control.

Thalamus. This is the earliest part of the diencephalon, the lower half of the forebrain. The thalamus is a critical, fist-like structure at the heart of the brain, and is surrounded by the higher cerebral hemispheres, and heavily connected with them. This may be the central controller, or just a junction box, or something in between. The much smaller, but vital *hypothalamus* may play a key role here as well, but that role is still uncertain.

Telencephalon. This is the end of the evolutionary line, at least for a few million more years. This region is dominated by the enormous cerebral hemispheres of the *cerebral cortex,* often called just the *cortex*. It is here, where the gray matter is on the outside, and the wiring is on the inside, that the most frenzied processing activity of the brain takes place.

The cortex is organized as a continuous sheet of cells 2–3 mm thick, and generally made up of six sub-layers of a large variety of different kinds of neurons. It is traditionally divided up into several regions, called the *frontal*, *parietal*, *occipital*, and *temporal lobes*.

The original cortex is now small and obscured in the human brain, having been overwhelmed by the explosion in size of the *neocortex*, or the new extension to the cortex. In fact, the most apparent difference in successive evolutionary generations of the mammalian brain is the size of the neocortex.

The layers of the neocortex contain a variety of neuron shapes known as *chandeliers*, *basket cells*, *pyramidal cells*, and many others. The reason for the different cell shapes is unclear except for its tendency to naturally route some cell connections locally, and others far, far away. For example, pyramidal cells in the lower layers of the cortex tend to route downward toward the thalamus for long distance communication, rather than to other cells in the cortex.

A more recent discovery is that the neurons of the neocortex are further organized into *macrocolumns*, cylindrical groups of 1000-10,000 neurons which are 0.5 mm or so in diameter and extend through all of the neocortical layers. A human has perhaps one million of these macrocolumns spread out over the entire neocortex[18]. Some are known to be associated with visual processing, and some to be associated with hearing or speech. Some are associated with complex movements and hand-eye coordination. Of particular note is the frontal lobe of the neocortex, known to be associated with planning and strategy.

It isn't clear what the visual appearance of macrocolumns in the neocortex has to do with their operation. It is known, however, that appearance varies with function. An early effort to

---

[18] Squire (1987), p. 70

understand neuroanatomy involved mapping the regions that looked different and then attempting to associate them with specific functions.

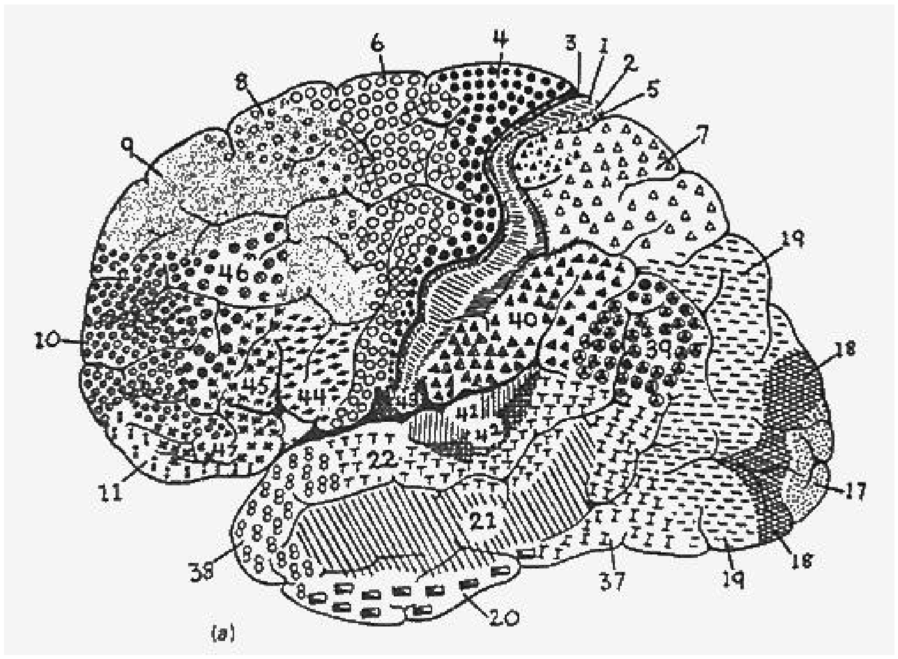

Figure 6: Neocortical Map by Physical Appearance

From Cairns-Smith (1996), p. 139

The map above, first developed by K. Brodman in 1914, gives evidence that even though the neocortex is built from the same components, it exhibits different structure and interconnection patterns in different regions of the brain. Later work localized these different regions to specific functions, as shown below.

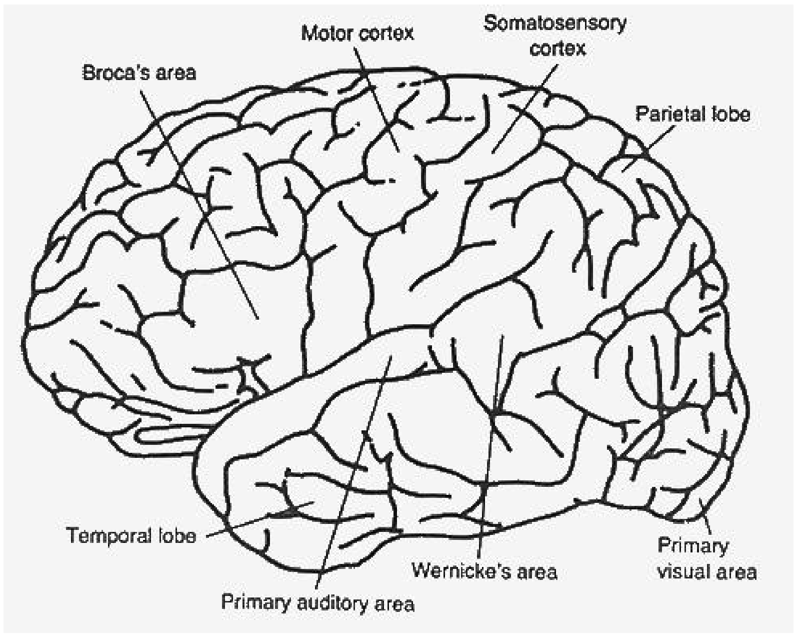

Figure 7: Neocortical Map by Function

Figure from Cairns-Smith (1996), p. 140

The visual and auditory areas are clearly shown. The *somatosensory cortex* is a region onto which are mapped the control and sense connections to individual body parts. Next forward is the motor cortex, a region onto which are mapped everything from our toes to lips. Especially dexterous regions, like the hands and the facial muscles, have a much larger region of motor cortex dedicated to them.

The sensory and motor portions of the neocortex consume about 25% of the device. Forward of the motor cortex is the premotor cortex, which shows signs of intense activity just prior to the initiation of complex actions. It is thought that planning of complex motions is carried out here.

The neocortex also includes a region underneath the temporal lobe specifically involved in face recognition. From an evolutionary perspective, face recognition is crucially important to survival and selective procreation. It therefore makes sense that the brain has a specialized region for it.

Other neocortical regions of special interest are Broca's place and Wernicke's place. These regions are associated with speech generation. Broca's place is concerned with grammar; with a damaged Broca's place, you might know what you want to say, but have trouble assembling words into a coherent and correctly constructed sentence.

Wernicke's area is concerned more with global meaning; if you have a damaged Wernicke's area, you might speak grammatically correct, but nonsensical, sentences.

It is possible that a large reason for the different appearances of the regions is simply the fact that different inputs and outputs are connected there. That is, there is nothing particularly "visual" about the visual cortex, *per se*, nor is anything particularly "motor" about the motor cortex. In fact, the macrocolumn design used for processing in the neocortex seems to be quite capable of performing a variety of cognitive tasks.

It is easy to over-generalize; on close enough examination, the neocortex looks chaotic and the perception of reasonably uniform organization into macrocolumns vanishes. But a too-close look at a digital microprocessor also only reveals a sea of millions of transistors and their interconnections, obscuring the logical and regular design that would only be evident at lower resolution. The key to understanding any complicated machine involves thinking about it at the right resolution.

A drawing of the brain that may be at just the right resolution is shown below. This drawing shows thalamic routing of input sensory data to associated macrocolumn clusters in the neocortex.

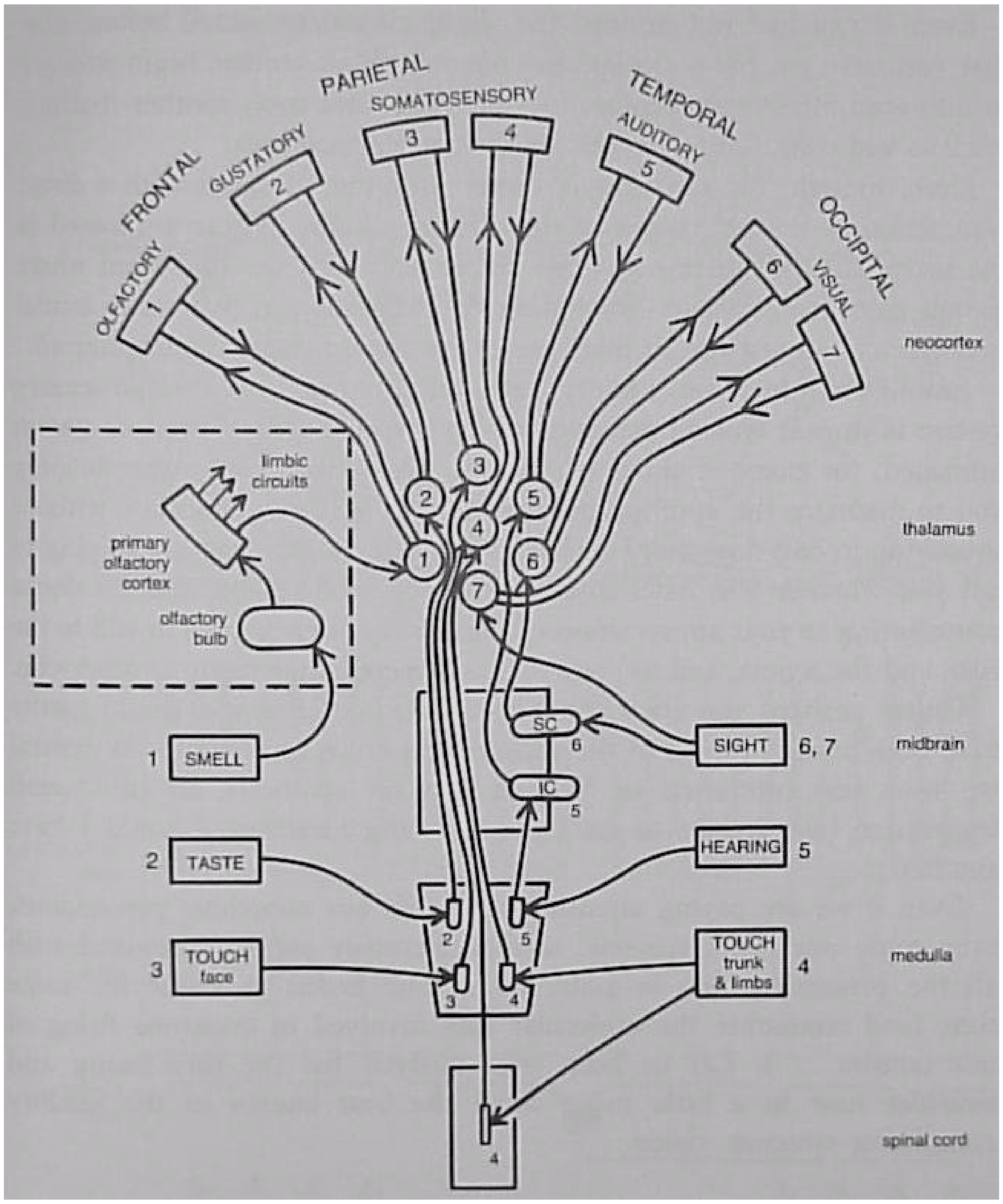

Figure 8: Thalamic Input Preprocessing

Figure from Cairns-Smith (1996), p. 159

From an engineering perspective, it is easy to view this as a computation system with similar regions in it, each designed specifically for a similar subtask. Furthermore, there is substantial evidence that input signal conditioning plays an enormous role in simplifying processor design.

As a way of introducing a theory on cognitive computing, it is interesting to take a brief look at the most complex input conditioning subsystem, the human visual processing architecture.

3.5 Brain Signal Input Conditioning

It was noted earlier that IQ tests focus on visualization as part of puzzle solving. In this section, it should become clear that visualization skills are different from pure cognitive skills because they require a sophisticated image processing preprocessing system to reformat the data into a reasonable form that the cognitive system can use. It is a less well-studied fact that the human auditory system relies on a data preprocessor much like that of the visual system.

The human retina, located at the rear of each eye, is composed of about 180 million rods and cones, cells which respond to light in three different color bands.

We commonly think of these bands as red, green, and blue, but they are actually unevenly spaced and centered around 420, 530, and 560 nm, and overlap significantly. Furthermore, the response bands are not very sharp; many colors significantly stimulate all three receptors. To perceive a color, we must compute and integrate signals from all different bands.

The 180 million rods and cones of each eye are reduced to about 1 million nerve signals on the way to the brain. This reduction is effected through averaging, differencing, and several other direct data reduction strategies. Here, the summing (and differencing, using inhibitory inputs) capacity of the neuron is used to great effect.

The 1 million input signals first reach the optic chiasm, where they are redistributed. Then, they pass to the left and right lateral geniculate nuclei (LGN) of the thalamus. This wiring structure is shown below.

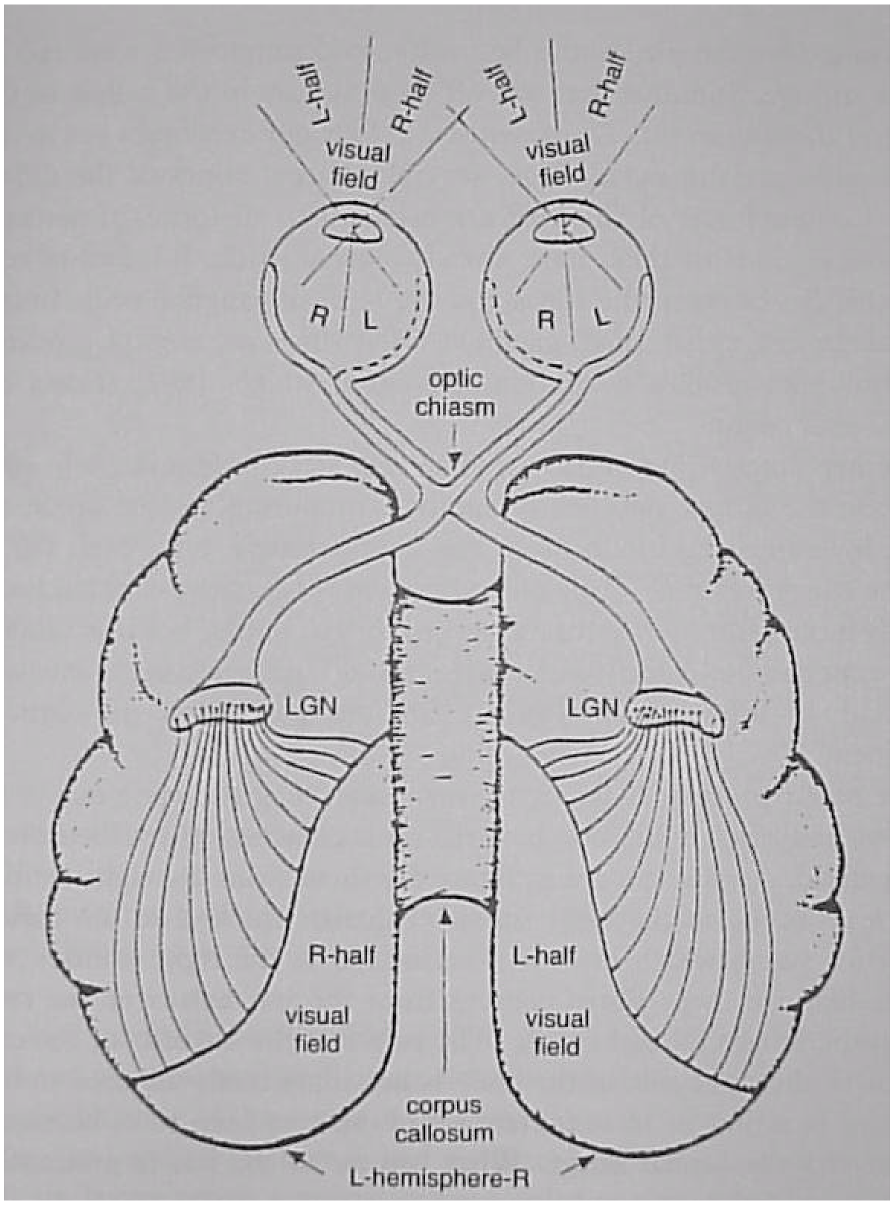

Figure 9: Optical Routing via the LGN

Figure from Cairns-Smith (1996), p. 163

The LGN is a powerful filter bank, performing detailed processing and reprocessing of the vision signals. A structural diagram of the LGN is shown below.

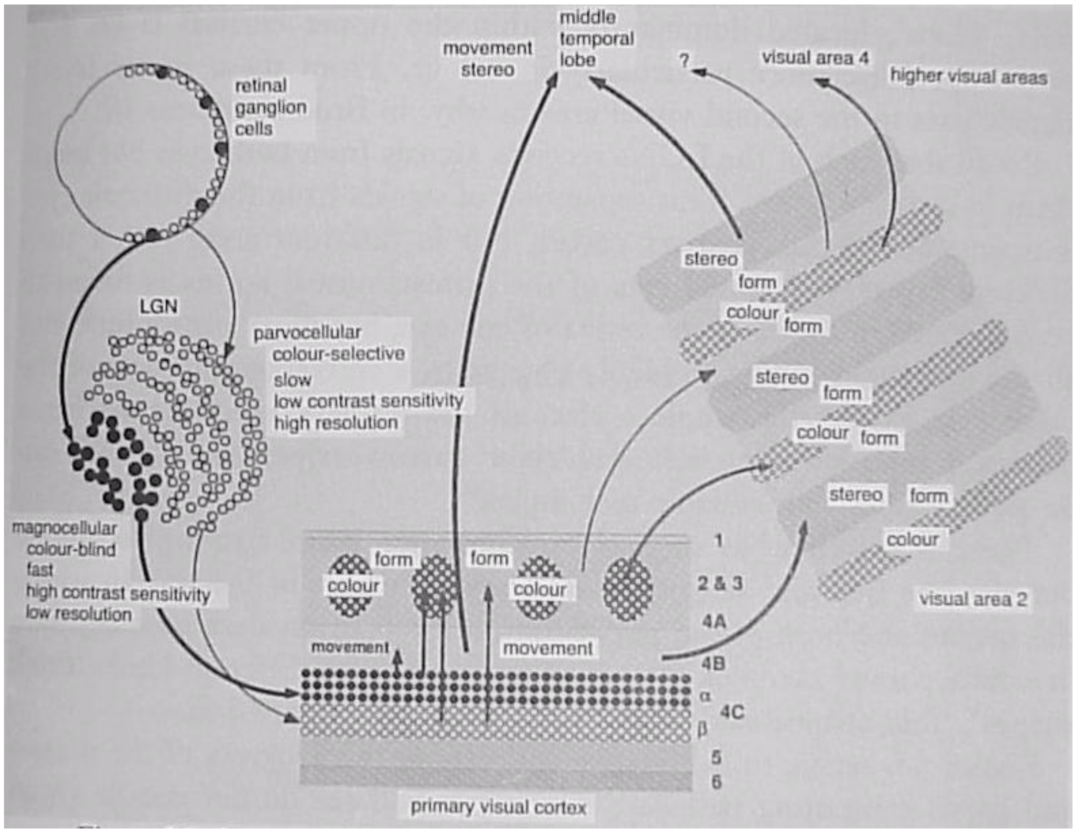

Figure 10: Detailed Map of LGN Interconnection

Figure from Cairns-Smith (1996), p. 165

This processing highlights motion, edges, angles, colors, and a variety of other features of interest before passing these multiple data representations on to the visual cortex. This arrangement of data makes the information particularly well suited to associative processing; that is, processing in which the features of interest are not necessarily known *a priori*, and which only become recognized as "features of interest" after analysis. A block diagram of this processing is presented below.

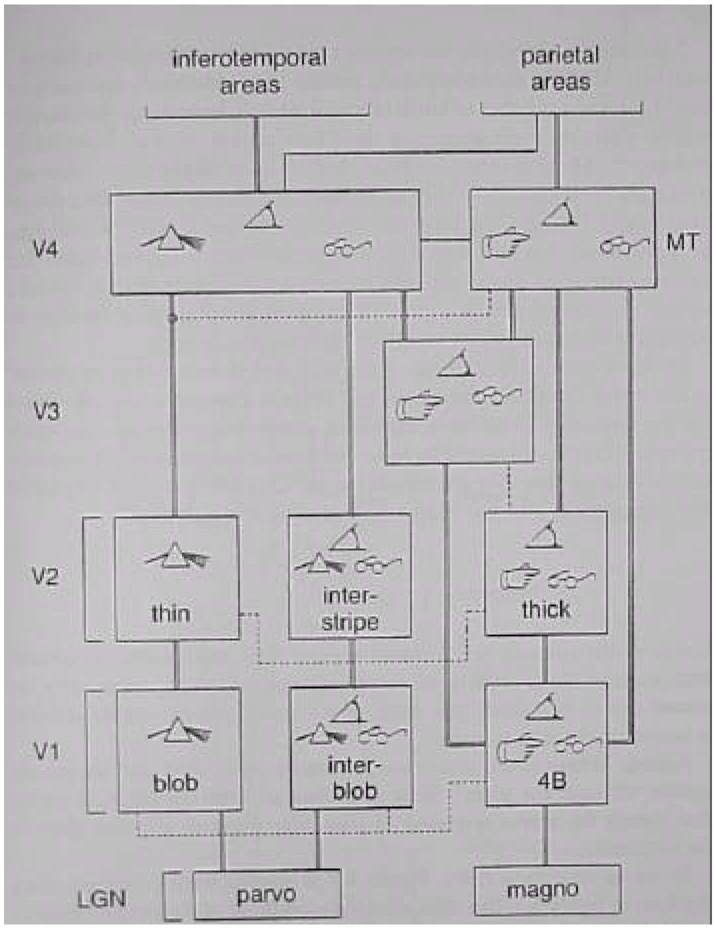

Figure 11: Approximate Visual Filter Bank

Figure from Cairns-Smith (1996), p. 167

This type of feed-forward signal conditioning and repackaging is easy to implement with neurons. It insures that a marvelously rich and information-packed array of data is available to the cortex for processing.

It is perhaps here in the visual processing system that an application of neural networks more similar to those built in traditional neural network research can be seen. This architecture is well suited to the production of multiple alternative data representations which simplify visualization activities by providing information that is more relevant to a given problem. The trick that remains involves selecting the relevant information from the multiple formats without being confused or overburdened by the unnecessary formats. This activity can be described as a puzzle, and can therefore be attacked with the same type of cognitive hardware that already exists in the neocortex.

This leaves us with the impression that the overwhelming power of the human visual processing system derives from two components: expansion of the input into redundant forms, and analysis. It is the analytical part of neocortical processing that constitutes intelligence, and a direct examination of this neocortical processing is in order.

3.6 How Does Neocortical Processing Work?

It has long been believed that the convolutions and folds of the neocortex have something to do with intelligence, since they are most pronounced in the human brain, and are less and less pronounced in brains of successively less evolved mammals. This strongly suggests that the surface area of the neocortex is somehow correlated with intelligence. Increasing the area of the neocortex hasn't really changed its structure; the only difference has been the increase in the number of macrocolumns available for processing. It must be that the size of the machine is very important.

For example, a fly has about 10^6 neurons, while a human has more like 10^{11}. Though it is true that flies have very impressive 3-D flight control capabilities, and can sometimes frustrate the most determined human attempting to catch them, it is clear that the first million neurons don't provide a brain with very much intelligence.

The human neocortex, if flattened out, is the size of four sheets of paper. A chimpanzee neocortex is the size of a single sheet of paper, while a rat's cortex is the size of a postage stamp.[19] It appears that rat intelligence is based on the same principles as human intelligence, and that the only material difference is that of size. The fact that the design is scalable from postage stamp size to four-sheet size indicates that it could probably be scaled even larger.

The example of the observed behavior of the premotor cortex gives a good hint at just how neocortical processing probably works. This area of the brain shows intense activity just prior to initiating movement. If one is told to pick up a cup of coffee, for example, the premotor cortex exhibits immense activity for ¼ second or so, right up until movement begins.

This activity suggests that a motion planning procedure is performed by a fairly large set of macrocolumns working in concert. Once this processing cluster has a plan selected, its job is done. The plan is somehow transferred to the cerebellum for actual execution.

Computerized motion planning is best accomplished by developing dynamic models of the mechanical system and then solving for their inverse. In this way, a desired motion trajectory can be supplied to the inverse model, which will then generate the actuator inputs needed to drive the desired motion. When the inverse model cannot be found, it can be estimated through numerical integration of the differential equations of motion of the system.

The complex dynamics of the human body, however, exhibit hundreds of degrees of freedom. Any attempt to solve such equations for their explicit inverse is completely unfathomable, and a unique solution never exists for human motion problems since many degrees of freedom ore non-orthogonal or even redundant. At the same time, some type of parallel biological integration of so many simultaneous multidimensional differential equations would have to be too computationally intensive, even for a significant fraction of the neocortical macrocolumns operating at their comparatively slow 1KHz cycle rate.

[19] Greenfield (1997), p. 14-15

A simple thought experiment helps to clarify the situation. If you imagine contracting your right biceps, you can mentally picture how far and fast the contraction would make your forearm move. This makes sense, since you have daily experience with contracting this muscle and observing its effect starting at birth.

But if you imagine raising your hand above your head, it is much harder to picture which muscles must contract to make the motion possible. Shoulder, back, neck, biceps, and triceps are all involved, as well as leg and waist muscle re-tensioning to compensate for dynamic and static load shifting. This suggests that we might have a good mental model predicting motion as a function of muscle contraction, but not have a good inverse model predicting required muscle contraction as a function of desired motion.

The forward model, however, is all that is required if an undirected exhaustive search is used. Specifically, one could repeatedly simulate the application of random sequences of inputs to the muscles involved until a solution was found. If random or typical input sequences were simulated many times in parallel and with great speed, the likelihood of a good solution being found quickly might be great, and the computational complexity of the solution of the inverse motion problem could be totally avoided. In addition, it would be equally likely that an unusual, unexpected, or “clever,” solution might be found.

The notion of an independent, random search in advance of the actual execution of the solution is analogous to the function of the *scout* in a military operation. One or more scouts explore the space in front of the main army, developing and returning intelligence to the commanders. Running many scouts in an efficient, parallel manner in hopes of finding an adequate solution is a search technique that will be called *scouting*.

Scouting could be easily applied to the garage door problem. We understand the parameters and the effect of each of our possible actions. To determine a solution to the problem, many sequences of actions could be simulated in parallel until one with a desirable outcome was found. Continued “thinking” could potentially generate a shorter solution, although the best solution found so far could be selected for execution when it was time to commit to a plan.

From a purist’s standpoint, scouting may seem crude and unsatisfactory. But scouting is a simple, robust, and easy-to-implement method that would make sense as an evolutionary choice.

3.7 Neocortical Reconfiguration

One potential weakness in scouting is that it would require a large number of macrocolumn simulators (scouts) to solve any problem. This would imply that there had to be a mechanism for “programming” each macrocolumn with the model for the problem at hand.

There does not appear to be “software” or “microcode” in the brain; that is, there is no central data store of program instructions that are uploaded into general-purpose hardware on demand to reprogram pieces of the system. This is a great architecture, but it does not appear to be part of brain design.

Instead, the extremely large fan-in of inputs to each neuron may be the reconfiguration mechanism. Though a typical neuron might have over ten thousand inputs, many different subsets of these inputs have the capacity to cause it to fire. Furthermore, the structure of a particular input subset can alter the way in which the neuron fires, controlling its function. That is, the neuron may be made to fire by any one of a number of different input conditions, each of which may be stimulated by totally unrelated groups of inputs.

The implication is that the neurons in a given macrocolumn could be reprogrammed to simulate different problem models by stimulating certain input axons and not stimulating others. The pyramidal cells in the lower neocortical layers, in particular, are good candidates for this personalization since so many of their inputs come from the thalamus, where macrocolumn allocation and configuration could occur.

This idea is reinforced by the comparative dearth of neural interconnect volume in the cerebellum. The “programs” executed in the cerebellum have been developed and selected over millions of years of evolution; they won’t be changing any time soon. Processing at this level is far more dedicated, so that the neurons have fewer modes of reconfiguration.

The idea that some neural inputs sensitize the neuron to other inputs while suppressing others is an excellent model for future biological research. This particular implementation of logic reconfiguration is not critical to a digital cognitive system, however. Very high-speed rewiring technologies like floating-gate transistors which can be programmed to act like on/off switches allow digital logic rewiring in array logic, gate arrays, and Complex Programmable Logic Devices (CPLDs). Using these digital building blocks, synthetic macrocolumns could be reprogrammed on the fly to solve successive problems. For this reason, it is probable that the extremely high fan-in of the biological neuron is thankfully not a necessary component of a cognitive computing system.

Using this model for neocortical processing and the notion of scouting, it is now possible to propose a digital processing architecture that can duplicate cognitive capability.

4 A Practical Scouting Machine

Scouting may be at least a partial explanation of how biological computing works. But scouting is not only attractive for its biological motivations. There is a strong computer science motivation for it as well. Consider the following definition of intelligence, taken from cognitive psychology:

> Intelligence is the spontaneous generation of a new piece of information from old information.

This definition implies that *intelligence requires a non-deterministic step.* The term "spontaneous" indicates that we have no algorithmic reason to give for how we came to a solution. Also, the prevalence of terms like "flash of inspiration," "leap of faith," and even "Eureka!" as associated with intellectual brilliance only underscores the notion of a non-deterministic step as being crucial to intelligence.

This non-deterministic definition of intelligence is comparable to more formal definitions of computational problems in NP and those that are NP-complete. As noted earlier, the only approaches to these problems are to either develop approximations, heuristics, use directed exhaustive search, or to take non-deterministic steps. The first three approaches have been investigated in the artificial intelligence field for decades. For example, an expert system is nothing more than a collection of heuristic rules that are examined repeatedly in an attempt to deduce an intelligent response. The limited success of such systems makes it tempting to think about creating a non-deterministic machine.

The only known way to take a non-deterministic step is to guess. We propose a solution by guessing, and then verify whether it actually would work. If many solutions can be considered rapidly, the likelihood of guessing the correct solution greatly increases. If our guess is "lucky," we look brilliant. The difference between luck and brilliance is probably very slight in practice.

An extension of this idea involves the use of many parallel digital hardware scouts to search for solutions simultaneously. Scouts can be undirected, or directed, based on how much attention they pay to their prior actions. A data-flow diagram of the operation of a single scout is shown below:

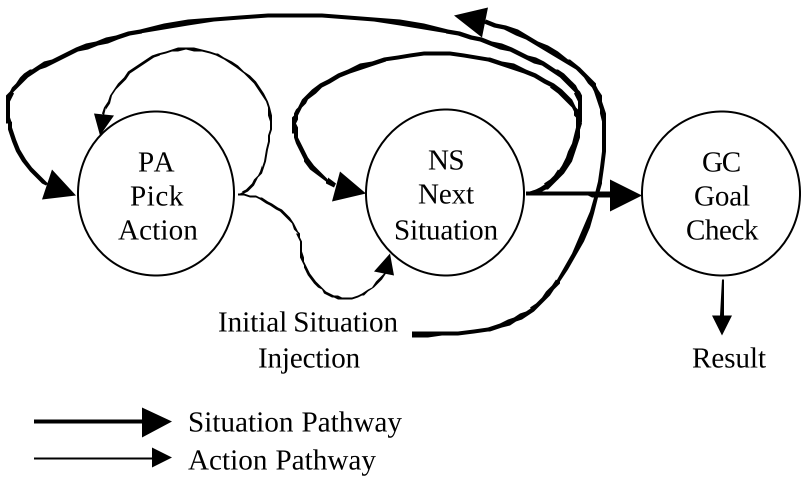

Figure 12: Individual Scout Dataflow

The figure shows the basic operation of a scouting macrocolumn. A *situation pathway* represents the current problem situation in a forward simulation. The *action pathway* represents the current action that has been selected. A *Pick Action* (PA) block uses the current situation to generate a list of possible actions, one of which it selects purely at random. It filters out some of the actions to avoid unmoves; that is, to eliminate consideration of moves which serve no purpose other than to undo the effects of previous moves. A scout which looks back *n* moves to detect unmoves is referred to as an scout-*n*.

The *Next Situation* (NS) block simply updates the current situation to the new situation given the chosen action. The resulting new situation is fed back into the PA block to automatically begin generation of another move. The *Goal Check* (GC) block is continually monitoring the situation to detect whether the scout has stumbled onto the solution.

4.1 Hardware Implementation

This data-flow diagram is easily converted into a block diagram for a digital hardware representation of a scout.

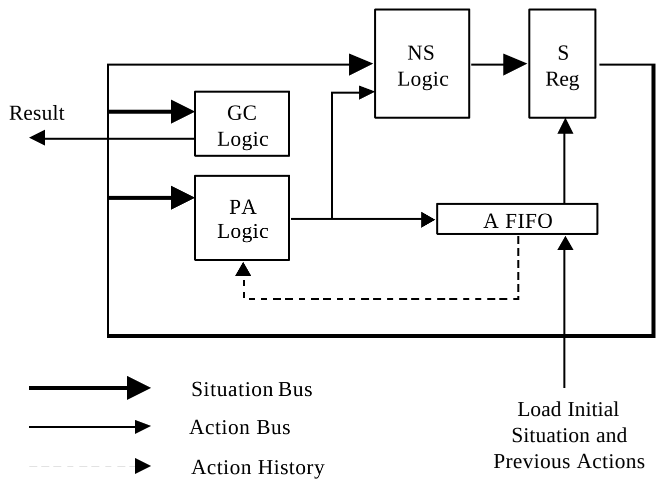

Figure 13: Scout Hardware Block Diagram

The GC, PA, and NS logic blocks are simple combinational logic for most basic problems. The situation and action pathways are buses into which registers are inserted to build a generic synchronous logic state machine. The *S Register* is simply a storage register for the current situation, while the *A FIFO* is a shift register for remembering previous actions to allow the PA block to rule out unmoves. The FIFO for a scout-*n* will contain *n* action stages. Increases in *n* generally impact the complexity of the PA block since avoiding unmoves becomes more complex as a large number of prior moves is considered. In a scout-0, the FIFO may be completely eliminated since there is no looking back required.

As a piece of traditional computing hardware, the scout is quite useless. If the first random action chosen, for example, is a pathological mistake that makes solution impossible, the circuit will run forever without finding a solution. Instead, as with macrocolumn-based processing of the neocortex, it is critical that many scouts run in parallel to increase the likelihood that at least one scout will find a solution.

This does not imply that thousands of physical scout circuits are required for the system to be capable of generating good solutions. In fact, a cluster of N scouts is capable of making many more than N scout attempts. This is a consequence of the fact that in most problems an expected worst-case solution length can be estimated. If a scout runs deeper than that depth, it has probably wandered off course. At this point, it may be reset to the initial situation and rerun from move one.

Furthermore, once one scout finds a solution, this establishes a new upper bound on solution length. Now it makes sense to reset all scouts and run them only to the depth of the best solution found so far. This process may be repeated several times, until little progress seems to be forthcoming in reducing solution length, or until a satisfactory solution is found, or until we are out of processing time and must execute the best solution found so far.

Scouting requires this procedural, iterative control scheme for the scout cluster. The overall scouting process is described in the two flowcharts below, using the following definitions:

s	a situation
a	an action
d	the number of moves generated so far
d0	the maximum number of moves to consider
A	the set of all actions taken so far by a particular scout in attempting to solve the problem
N	the total number of scouts available for processing
Si	initial situation
Sg	goal situation
pa(s,A)	a random legal action when in situation s that is not an unmove given the A moves so far
ns(s,a)	the new situation resulting from taking action a in situation s

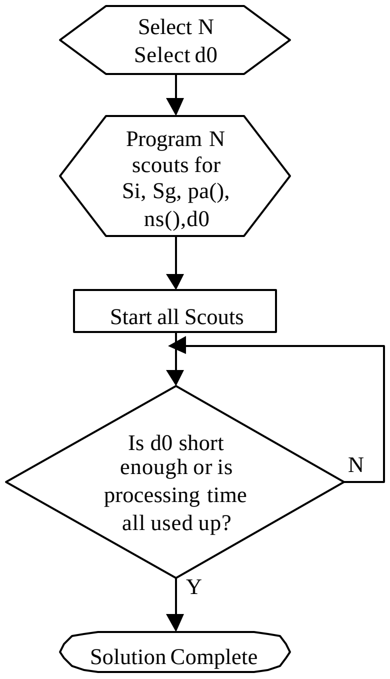
Select N
Select d0
Program N scouts for Si, Sg, pa(), ns(),d0
Start all Scouts
Is d0 short enough or is processing time all used up?
N
Y
Solution Complete

Figure 14: Scout Cluster Control Flowchart

This flowchart may be summarized by saying that first we must setup *N* scouts for the problem at hand. Then they are started and we wait until the global shortest solution *d0* is short enough for our purposes, or until we are out of processing time and must execute whatever solution has been found so far.

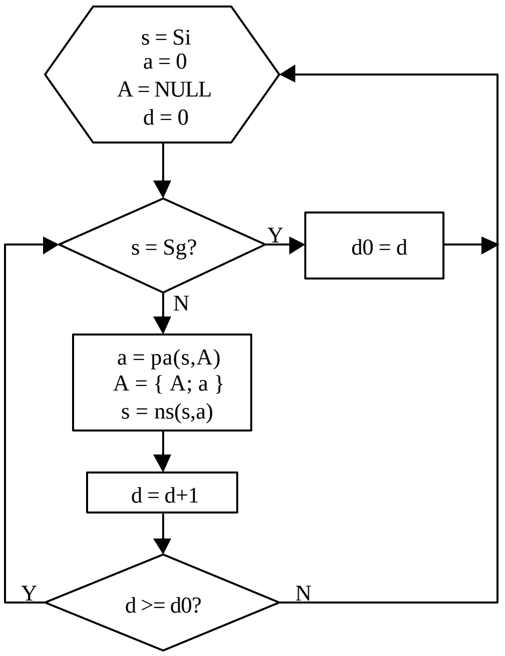

Figure 15: Individual Scout Operation Flowchart

This flowchart is a bit more complicated, but still straightforward. A scout maintains a local depth *d* that it initializes to 0 and increments as it picks and executes successive random moves. If *d* reaches *d0*, then the scouting search has failed and the scout should reset to the initial situation and try again. If at any time a better solution is found, the global *d0* is updated to shorten the execution time of all scouts.

To further explore this architecture, it is instructive to examine a simulation. This is undertaken in the next section.

4.2 Scouting Simulation for the 8-puzzle

While the speed advantage of running thousands of scouts in parallel is not possible with a direct software simulation, a simulation is very useful in revealing the properties of scouting as long as a simple enough problem is chosen. The 8-puzzle is appropriate in that it has non-trivial solutions requiring 20 steps or more, but is computationally simple enough to allow the cluster to be effectively simulated one-scout-at-a-time on a sequential machine.

For simulation purposes, the puzzle below is considered. This puzzle was generated by starting with the goal situation and then making 1,000,001 random scout-1 moves.

1	8	3
	4	2
5	7	6

Figure 16: A Test 8-puzzle

For a computer implementation, the *situation pathway* is considered to carry a vector of ten integers. The first nine represent the tiles in each of the 3x3 grid elements top to bottom, left to right, with the following array indices:

s[0]	s[1]	s[2]
s[3]	s[4]	s[5]
s[6]	s[7]	s[8]

Tile 0 represents the blank. The tenth element is a redundant data element specifying the location of the blank. This redundant data item allows the move generator to find the blank without iterating through all of the puzzle tiles, and accelerates the simulator. Such an augmentation might not be particularly useful in an actual hardware implementation.

Using this representation, the initial situation for the 8-puzzle above is {1,8,3,0,4,2,5,7,6,3}.

The *action pathway* contains integer pairs specifying the source and destination positions for the blank. For example, moving tile 5 up in the example above is represented by the action {3,6}.

The code in C for the simulation is shown in Appendix A. This simulation includes the necessary PA code for both a scout-0 and a scout-1, and the main(…) routine runs both on the test puzzle above as well as on several randomly generated puzzles. A complete output run from the program is included in Appendix B. All runs executed 100,000 scouts.

One note about the coding of these simulation programs is in order. The C rand() function returns a pseudo-random sequence of all integers from 0 to 32,767 inclusive. Since the rand() function drives the scout simulation, the seed at the start of the scouting operation determines which action sequence will ultimately be explored. Due to the rand() limitations, only 32,768 distinct action sequences can normally be generated. This would mean that simulating more than 32,768 scouts would have no effect, since all additional scouts would repeat the pattern of previous scouts.

To partially relax this constraint, the VRand() function shown in the appendix A listing was developed. This function calls rand() three times and concatenates its 15-bit value into a 31-bit positive integer (only one bit is used from the third call). This technique extends the length of the pseudo-random sequence since the three calls do not evenly divide into the normal rand() sequence length of 32,768. However, this technique is not optimal, and may at least slightly degrade the simulation results for situations where more than 32K scouts are used.

The table below shows some example results from the Appendix B listing.

Table 1: Scout-0 Performance Examples

Scout-0 Performance Examples					
run 1		**run 2**		**run 3**	
Scout	Solution Length	Scout	Solution Length	Scout	Solution Length
7125	81	7414	61	7449	83
13068	79	38348	59	19100	61
20133	49	42699	47		
26410	39	88603	45		

The interpretation of these results is as follows. In run 1, the first 7124 scouts found no solution before resetting at the initial 100-move depth limit.

Scout 7125 found an 81-move solution. Scouts 7126 through 13067 found no better solution, while scout 13068 found a 79-move solution. Later, scout 20133 found a 49-move solution, and scout 26410 found a 39-move solution. Scouts 26411 through 100,000 found no better solution.

Run 2 ultimately found only a 45-move solution, while run 3 found only a 61-move solution.

These results are unacceptably poor for two reasons:

1. 8-puzzles rarely take more than 20 or so moves to solve, making these solutions look rather unintelligent.

2. The results are highly inconsistent, yielding solutions of length 39 to 67, after even 100,000 scouts have run. This indicates that many more scouts would need to be run to achieve consistent performance.

Fortunately, scout-1 performance is markedly different. Eight out of the ten test runs found the same 21-move solution. The table below shows the specifics for the first three runs in the scout-1 section of Appendix B.

Table 2: Scout-1 Performance Examples

Scout-1 Performance Examples							
run 1			run 2			run 5	
Scout	Solution Length		Scout	Solution Length		Scout	Solution Length
4925	69		4442	21		7	81
11692	57					1529	53
15129	29					7324	33
30450	23					7558	31
56727	21					27527	25
						32459	21

Performance is much more consistent, and the solution length is much shorter than that achieved with the scout-0 approach. A more complete look at this initial simulation is shown in the table below. The scout-0 solutions are much less consistent; in fact, the last scout-0 run of the Rand3 puzzle never found any solution under the initial 100-move limit.

Table 3: Solution Length Summary

		Solution Length Per Run										**Solution**	
Puzzle	**Scout-n**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Avg**	**StDev**
Test	0	39	45	61	67	45	47	51	47	49	63	51.4	8.66
Rand1	0	81	85	49	89	91	61	57	75	77	83	74.8	13.61
Rand2	0	53	53	45	77	51	71	61	49	49	57	56.6	9.75
Rand3	0	93	77	77	71	77	55	75	87	69		75.7	10.15
Test	1	21	21	21	21	21	25	21	23	21	21	21.7	1.28
Rand1	1	27	27	33	27	29	27	27	31	31	27	28.8	2.15
Rand2	1	27	29	27	29	27	25	27	25	27	23	27.0	1.74
Rand3	1	25	29	25	25	29	31	29	27	27	27	27.4	1.96

The table also shows average and standard deviation of the solution lengths developed. The scout-1 behavior generates solutions less than half the length of the scout-0 solutions, and their consistency is many times better.

Experimentally, it appears that while 100,000 scout-1s do not always find the optimal solution, they generally find quite good 8-puzzle solutions. The solution speeds implied by this are quite impressive. For example, imagine a system of 1,000 scout-1s setup for 8-puzzle solution. Since the worst-case 8-puzzle takes less than 30 moves to solve, the initial d0 setting for each scout would be 30; that is, no scout will ever probe past a 30-move sequence, instead opting to reset and try again. In this way, the complete 1,000 scout cluster would execute 100,000 scout-1 searches in the amount of time it would take a single scout to generate 100 30-move sequences.

If the scouts were crafted from present digital logic technology, their simple combinational logic and register architecture could circulate moves at a rate of at least 50 million moves per second. At this rate, the total execution time for 100 30-move sequences per scout would be only 60 μS.

Thus, the complete 8-puzzle solution would be generated in 60 μS, allowing the scout cluster to solve 16,667 8-puzzles per second.

A more modest scout cluster containing only 16 scouts would still solve the puzzles in 3.75 mS, while even a single-scout scout cluster would solve arbitrary 8-puzzles in 60 mS.

This initial experiment proves that scout clusters can solve simple problems at an incredible speed. It remains to be seen, however, how well the architecture scales to more difficult problems. Furthermore, in these few example puzzles, 100,000 scouts were determined only experimentally to be a good number of scouts. There is not yet any good way to quantify how well 100,000 scouts generally solve 8-puzzles, or what the expected benefit would be in increasing to, say, 200,000 scouts. Determining the limits of scouting will require a more theoretical examination of its efficiency, and this is the focus of the next chapter.

5 A Theoretical Analysis of Scouting Efficiency

In scouting, each scout is randomly and independently searching in parallel. Maintaining this independence is important, since it makes the architecture scalable; additional scouts can be added without increasing the control overhead.

The drawback to scouting is the fact that the uncoordinated searches will occasionally duplicate their efforts; there is no mechanism to prevent one (or all) of the scouts from generating the same action sequences, negating the usefulness of the parallelism.

The following theoretical treatment will develop the relations necessary to measure this inefficiency and to estimate how many scouts are needed to solve various problems to desired accuracy.

5.1 The k-sided Die

The analysis begins by considering n tosses of a k-sided die. Later on, we will consider the n tosses to be n scout runs, and the k sides of the die to be the number of possible action sequences for each scout run, but this generalization is not required at the moment.

Let's assume that one side of the die is the goal side, and our hope is to roll this side at least once during our n tosses. The probability that this single side will be seen is the same as the probability that *all* sides will be seen. For this reason, it is useful to introduce the term *coverage*, which will refer to the fraction of the die sides that have been seen after n tosses, and which will be referred to as $c(n)$:

$c(n)$	The coverage function is the fraction of the k-sides that have been seen at least once after n tosses, $0 \le c(n) \le 1$

The first toss always results in one side being seen. Therefore:

$$c(1) = \frac{1}{k}$$

The second toss will reveal a second side unless the same side happens to show up as in toss one. Similarly, the third roll will reveal a new side as long as that side was not seen in roll one or two.

In general, if we let r be the number of distinct sides seen in all tosses so far with $0 \le r \le k$, we have:

$$c(n) = \frac{r}{k}; \quad 0 \le r \le k$$

From one set of n tosses to the next, $c(n)$ will vary, since the randomness of the die will make r vary from experiment to experiment. When scouting cognitive problems, however, both n and k will be very large, so an expression for $c_{avg}(n)$, the average coverage, can be developed which should be useful in making accurate predictions.

To develop this expression, first consider three tosses of a four-sided die in which we are hoping to roll a one (that is $k = 4$ and $n = 3$). The one may be rolled in the following ways:

```
Success on first roll:
    1 X X
Success on second roll:
    2 1 X, 3 1 X, 4 1 X
Success on third roll:
    2 2 1, 2 3 1, 2 4 1
    3 2 1, 3 3 1, 3 4 1
    4 2 1, 4 3 1, 4 4 1
```

It is easy to see that the first roll generates the one in k^2 different three-roll sequences. The second roll generates the one in $(k-1)k$ sequences, while the third role generates the one in $(k-1)^2$ sequences. Since there are a total of k^n sequences generated by the tosses, an equation for average coverage is easily generalized from tabulation:

$$c_{avg}(n) = \frac{\sum_{i=0}^{n-1} k^{n-1-i} (k-1)^i}{k^n}$$

This can be manipulated algebraically to give:

$$c_{avg}(n) = \frac{1}{k} \sum_{i=0}^{n-1} \left(\frac{k-1}{k} \right)^i = \frac{1}{k} \sum_{i=0}^{n-1} \left(1 - \frac{1}{k} \right)^i$$

We can now use the well-known series identity:

$$\sum_{n=0}^{N-1} a^n = \frac{1-a^N}{1-a}; \quad a \neq 1$$

to arrive at the exact form for the average coverage expression:

$$c_{avg}(n) = 1 - \left(1 - \frac{1}{k} \right)^n$$

Equation 1: Exact Equation for $c_{avg}(n)$

It will be useful to try to solve this equation for n to determine how many tosses are required to achieve a certain probability of finding the optimal solution. Performing the solution is made easier by recalling the identity:

$$\lim_{n \to \infty} \left(1 + \frac{x}{n} \right)^n = e^x$$

Since n is always large, the following equation holds:

$$c_{avg}(n) \cong 1 - e^{-\frac{n}{k}}$$

Equation 2: Approximate Equation for $c_{avg}(n)$

5.2 Generalization to Puzzles

The result of the last section can be generalized for puzzles. In a puzzle, a series of decisions need to be made to arrive at a solution. These decisions define a decision tree which can often be thought of as having an average branching factor b and a typical solution depth d. In this way, the solution space of the puzzle becomes a k-sided die with $k = b^d$.

Two problems (which are really inverses of one another) are of interest:

1) Given a puzzle, if one runs n scouts, what is the probability P_d of discovering a good solution?

2) For a given desired probability of finding a good solution, how many scouts should be run?

The probability of discovering a solution to a puzzle is also influenced by how many different acceptable solutions exist. If there are *m* equivalent but distinct solutions, the effect is to reduce the number of "sides on the die" by a factor of *m*. Since the probability of finding an acceptable solution is the same as the probability of finding any particular solution, the coverage expressions from the previous section may be used. Substituting $k = \frac{b^d}{m}$ into Equation 1 yields:

$$P_d(n) = c_{avg}(n) = 1 - \left(1 - \frac{m}{b^d}\right)^n$$

Equation 3: Probability of Discovering a Solution

The inverse problem of solving for the number of scouts needed to achieve a desired probability of discovery involves solving Equation 3 for n, which is cumbersome. Instead, it is advantageous to use the approximate form given in Equation 2. Substituting $k = \frac{b^d}{m}$ into Equation 2 yields:

$$P_d(n) = c_{avg}(n) \cong 1 - e^{-\frac{nm}{b^d}}$$

Solving this relation for n yields the desired relation:

$$n \cong \frac{-b^d}{m} \ln(1 - P_d)$$

Equation 4: Necessary n to Achieve P_d

Consider the 8-puzzle as an example. The branching factor for scout-1 processing is 1.67, and typical solution depth is around 20. If we assume in the worst case that there is only one way to solve the puzzle in 20 moves, and we want a 90% likelihood of finding the solution, we have:

$$n = \frac{-1.67^{20}}{1} \ln(1 - .9) = 65{,}546$$

That is, running 65,546 scout-1s gives a 90% probability of finding the optimal solution. Requesting a probability of 99% increases the number of scout-1s needed to 131,092. This is still a very small number of scout cycles, given that the circuits execute so rapidly.

This agrees well with the experimental consistency of the 100,000 scout-1 simulation data summarized in Table 3 on page 56.

5.3 Experimental Verification

Equations 1-4 form the basis for the theoretical justification of scouting, and as such deserve some experimental verification. It is an easy matter to write a simple computer program that simulates n tosses of k-sided dice, and to tabulate real results with those predicted by the equations. This allows verification of Equations 1-3, and a computer program designed to do just that is presented in Appendix C.

Appendix D shows the output of this program, which simulates various combinations of n, k, and m. All data show extremely good correlation with theoretical results. In particular, the approximate form of coverage given in Equation 2 is demonstrated to be very accurate for all but the most trivially small problems.

A program for verifying Equation 4 is shown in Appendix E. This is a variation of the program of Appendix A, which solves 8-puzzles using scouting. The program has been modified to generate a database of 10 puzzles and run 200,000 scouts on each to find an optimal or at least near-optimal solution. The program restricts consideration to puzzles with solutions between 18 and 20 moves long. Shorter puzzles may become trivial to solve, while longer puzzles require excessive simulation time and do not seem to alter these conclusions in any way.

Once this database is built, its average solution depth is used in Equation 4 to predict how many scouts are required to achieve a given $P_d(n)$. This number of scouts is run 20 times on each of the 10 puzzles, and a tally is made of what percentage of the time the optimal solution is found.

The summary below is from the run data from this program as shown in Appendix F.

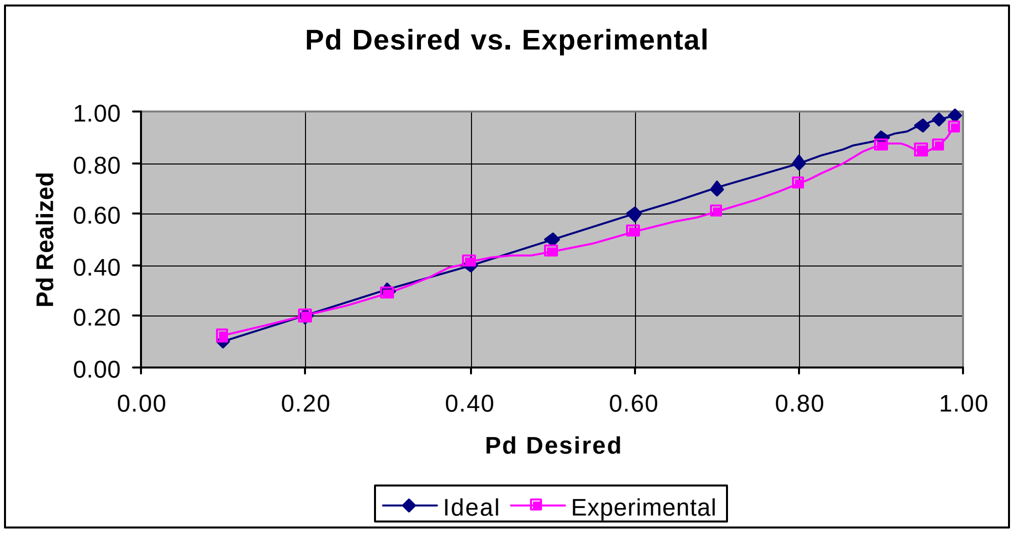

Figure 17: Desired vs. Experimental $P_d(n)$

A very close correlation is observed over the complete range of $P_d(n)$. Over the course of many tests of several different puzzle databases, similar results were observed.

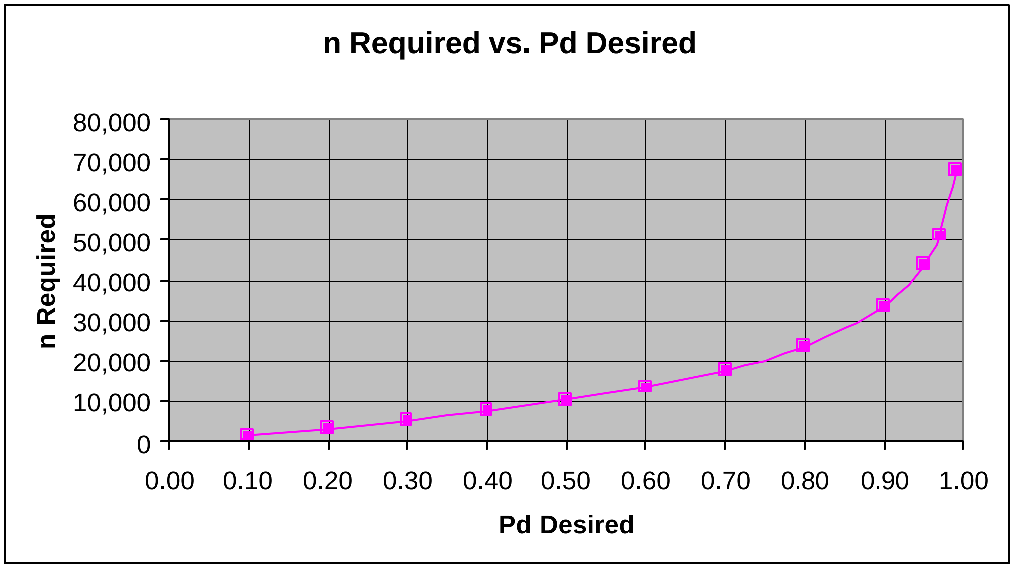

Figure 18: n Required vs.Desired $P_d(n)$

The actual number of scouts predicted by Equation 4 to achieve each level of probability is shown in the figure above. It is likely, but not proved, that the slight divergence of theoretical vs. experimental $P_d(n)$ beyond $P_d = 0.40$ is related to random number generation errors. At this level, $n = 7465$. Since each scout run proceeds a bit deeper than the 19-move average solution (indeed, the first scout may proceed to 100 moves), and since each move necessitates a call to Vrand(), this level requires more than $7465 \cdot 19 = 141{,}835$ calls to Vrand(). At this length, even Vrand() will loop back to the beginning and start to regenerate previously seen sequences, reducing the effectiveness of the scouting.

This hypothesis is reinforced by the fact that the divergence between expected and experimental $P_d(n)$ is amplified when the Vrand() function is replaced with a simple call to the rand() function that has a pseudo-random period of only 32,768.

This problem is not further explored in this dissertation, since other techniques will be used for random choice generation in real scout implementations. Future research involving simulation, especially involving problems with deep and broad solution spaces, would benefit from a random number generation routine with more significant bits.

6 Using Scouting to Solve Other Problems

Armed with the theoretical results of the previous section, we can now look at other problems and develop expressions for the extent of scouting required for their solution. A small set of other problems is introduced below.

6.1 The 15-Puzzle

The 15-puzzle is an extension of the 8-puzzle, except that it is played with 15 tiles in a 4x4 grid. The average branching factor of 3 can be cut to 2 using the scout-1 approach, but since the puzzle has an average solution depth of 40, we still have $k = 2^{40} \cong 10^{12}$ different solution paths to explore. Over such a deep tree, scouting seems likely to fail.

No human would attempt to plan all the way through to the solution before making the first move in such a puzzle, however. Problems with this much complexity are best handled using a divide and conquer approach in which intermediate goals, or *subgoals*, are defined.

For example, the same strategy that is used to solve an 8-puzzle could be used to solve a 15-puzzle, if only the other 7 tiles were in place first. Consider using the following goal situation for the 15-puzzle:

9	10	11	12
13	1	2	3
14	8		4
15	7	6	5

Figure 19: 15-puzzle Goal Situation

One can start by only attempting to place tiles 9-15 first, without regard to what happens in the lower right 3x3 quadrant. Once this is accomplished, a standard 8-puzzle is all that is left to solve.

It usually only takes around 20 moves to place tiles 9-15, since the 15-puzzle has a scout-1 non-backtracking branching factor of 2. Then, the final 8-puzzle solution takes 20 additional moves with a branching factor of 1.67. This combination problem is much more tractable than would be a direct assault on the 15-puzzle.

6.2 Missionaries and Cannibals

This is a time-honored problem involving 3 missionaries and 3 cannibals crossing a river in a canoe that can only hold two people at a time. They must schedule their trips across the river with the caveat that if at any time a group containing more cannibals than missionaries is left alone on either side, the cannibals' rude eating habits will end the trip.

There are two well-known optimal solutions for this problem requiring 11 steps. At any time, the canoe must be moved from one side to the other containing some combination of 1 or 2 missionaries and/or cannibals. All combinations are not usually possible due to lack of available passengers, so the average branching factor for this problem is around 3.

6.3 The Stream Crossing Problem

Stream(20,100,36) is defined as the "stream crossing" problem, in which 36 rocks are randomly placed in a 20 unit wide channel across a 100 unit wide stream. Given a maximum stride length of 15 units, the goal of the puzzle is to cross the stream (with dry feet) in the minimum number of steps.

The figure below shows a sample problem instance and the solution found by scouting. The black dots represent rocks, and the larger dots are the rocks selected to be stepped on. The initial position is at the center of the horizontal grid labeled 0, and the desired final position is the center of the 100 grid. Since maximum stride is 15 units, the largest step distance between rocks is 1½ grid spaces.

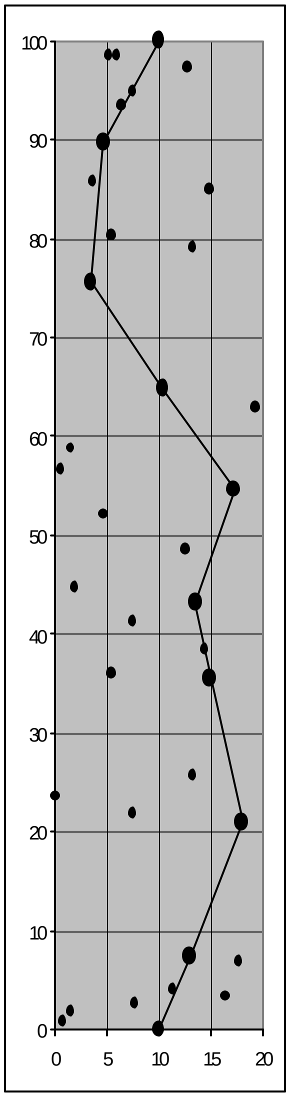

Figure 20: Stream Crossing Example and Solution

This is a good problem for visualizing the methodology of scouting. Imagine 100,000 scouts standing on the rock at the bottom of the figure and all leaving at the same time, hopping randomly from rock to reachable rock. Whichever one gets to the other side first yells out and ends the solution searching process. The path followed by that first scout represents the solution to be executed.

Stream crossing is an interesting problem because it is a real-world problem that humans solve so easily. Furthermore, the PA logic is a bit more complex than straight combinational logic, since the measurement or estimation of Euclidean distance between the rocks is required. Undoubtedly, the human visual system assists us greatly in pre-processing the data when we solve the same problem.

For this scale of stream crossing, typical solution depth is 10 moves (or 10 hops), with a branching factor averaging 5 choices per hop.

6.4 Estimating Runtime

It is now possible to develop a solution time estimate for other puzzles. A scout cycle rate of $f = 100MHz$ will be assumed (100 million moves/sec). This is approximately accurate for a present CPLD digital logic technology, and future speeds will only get faster.

In addition, it will be assumed that the n scout cycles will be executed on N physical scout circuits, and that in general $N < n$. That is, the physical scout circuits will be iterating so that each physical scout circuit will perform several, or many, scouting searches. A scouting search in this scenario will have a runtime measured by:

$$t_{run} = \frac{nd}{fN}; \quad N \le n$$

Equation 5: Scouting Runtime

This equation says that execution time goes up with solution depth and down with circuit cycle time. In addition, the execution time increases with the $\frac{n}{N}$ ratio in a three-phased but obvious way; for $n = N$, there is complete simultaneous search of all n scouts. For $N < n$, there is a time penalty as most scouts execute $\frac{n}{N}$ searches. And for $N > n$, there is no benefit since more the n scouts cannot execute any faster than n scouts would. In fact, in this case, we would run all N scouts and get the added benefit of more complete coverage.

Since a human has $N \cong 10^6$, it is reasonable to use $N = 10^3$ for a first-generation scouting engine. Using this expression, the following puzzle solution runtimes were developed.

Table 4: Runtime Estimate with Various Puzzles

Runtime Estimate, N=1000 f=100MHz						
Puzzle	**b**	**d**	**m**	**Pd**	**n**	**trun, S**
8-Puzzle	1.67	20	1	0.95	8.5E+04	1.71E-05
Garage Door	25	6	3	0.95	2.4E+08	1.46E-02
15-Puzzle	2	40	1	0.95	3.3E+12	1.32E+03
15-Puzzle reduction to 8	2	20	1	0.95	3.1E+06	6.28E-04
Missionaries and Cannibals	3	11	2	0.95	2.7E+05	0.029
Stream(20,100,36)	5	10	3	0.95	9.8E+06	9.75E-04

Obviously, all execution times are acceptable with the exception of the direct attack on the 15-puzzle. In this case, first reducing the 15-puzzle to an 8-puzzle, and then solving the remaining 8-puzzle would find the solution.

All of these puzzles have been solved with variants of the scouting simulator of Appendix A, and excellent solutions are always found. Due to the number of scout simulations required, however, it is difficult to achieve the 95% confidence level in simulation. Running actual hardware scouts in these numbers is much simpler.

6.5 Motion Planning

As an example of how scouting can be applied to much more complex problems, it is instructive to examine the problem of motion control. This section will show how scouting can be used to plan a movement in advance, thereby drastically reducing the complexity of the required control system.

A traditional motion control methodology is shown below. In this diagram, motion control is implemented as a feedback system in which the desired position $r(t)$ of the system is compared to the current position $y(t)$ of the system, and the resulting error signals $e(t)$ are used to develop actuator inputs $u(t)$. In such a scenario, a dynamic model of the system, or plant, is essentially inverted to enable the prediction of what actuator inputs would be required to achieve a particular motion.[20]

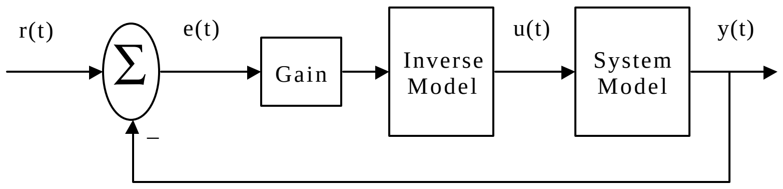

Figure 21: Traditional Motion Control

Unfortunately, this inversion of the system model is often quite difficult, and indeed the bulk of the motion control literature is concerned with *linear* system theory since linear systems are the only ones that can be analytically inverted in practice.

Nonlinear systems are quite common. In fact, most real systems demonstrate significant nonlinearity. A common example involves any jointed arm robotic system, since these systems

[20] Franklin, Powell, and Workman (1990), p. 2

have a rotational coordinate system subject to the constant linear acceleration of gravity. The simplest example of such a system is a traditional pendulum, as shown below.

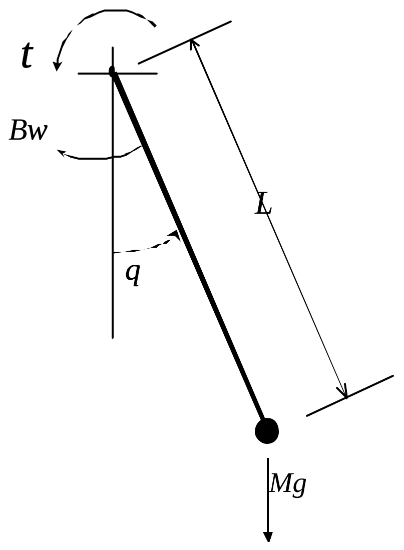

Figure 22: Pendulum System

In this simple system, a mass *M* suspended from an arm of length *L* is free to rotate completely around the upper pivot point. We consider the force exerted by gravity, *Mg*, on the mass, and assume that the pivot has a frictional drag property Bw that is proportional to the rotational velocity of the pendulum.

This second degree system has two state variables ($y_1(t)$ and $y_2(t)$) which can be called by their more common names, q and w . The angle q is the angle made by the pendulum from the vertical, and w is its rotational velocity. We consider a single input $u_1(t)$, more commonly called t , which is a torque applied about the pivot.

A multidimensional system equation is commonly decomposed into multiple 1st order differential equations for analysis purposes. The system equations for the inverted pendulum are:

$$\dot{q} = w$$

$$\dot{w} = \frac{1}{ML^2}\left(t - Bw - MgL\sin(q)\right)$$

It is the gravitational torque term including $\sin(q)$ that introduces significant nonlinearity into this otherwise simple system. A closed form solution of these equations is very difficult to obtain. For an arbitrary t not of a specific form, an analytical solution is probably impossible.[21]

Given a command input r_1 and r_2, which could for example be $q = p$ (straight up) and $w = 0$ (motionless), traditional motion control would look at the q, w error and try to generate the necessary t to move in the correct direction. The nonlinear nature of the system model makes this position-to-actuator input conversion very difficult.

Several techniques for controlling nonlinear systems are proposed in the literature. The most common techniques, however, involve piecewise linearization of the system model to enable more classical analytical approaches to be used in at least a piecewise sense to develop a control methodology. For inverting the pendulum, a half-dozen or more control regions would have to be selected over which a linear approximation to the $\sin(q)$ term would be created.

It is possible to use scouting and propose a much different approach, however. In this approach, each motion will be planned in advance, much the way the human motor cortex is known to be involved in planning long before movement begins. Motion planning involves the development of a "trajectory" which consists of a time history of $q(t)$ and $w(t)$ that leads from the intial to the

[21] As stated in Franklin, Powell, and Workman (1990), p. 518

desired final state. This trajectory is driven by a proposed $t(t)$ which would be the exact actuator input profile required to drive the system if the system model were perfect and there were no outside influences during movement.

In a real-world environment, the motion plan serves as an intelligent guideline for the motion and actuation which can be fed into a very simple feedback controller for actual execution. In fact, since the steps of the plan can be spaced arbitrarily close together, this motion plan can be used with a series of piecewise linear controllers to build extremely accurate nonlinear system controllers.

A scouting approach to motion planning requires applying arbitrary actuator inputs at a particular rate and then seeing if the model moves as was hoped. If a scout discovers an actuator input sequence that leads the system to the desired final state, then a motion plan has been found. Other scouts might find solutions with fewer "steps," which would normally be associated with faster execution time, but which might also be set up to indicate lower total power consumption or some other metric of solution "length."

A direct application of this idea, however, is doomed to fail, since applying a stream of random torques to our pendulum is extremely unlikely to, say, invert it. However, knowledge of the form of typical solutions to motion control problems can reveal a way to abandon this scout-0 approach in favor of a scout-1 or even scout-2 type formulation.

Actuator inputs in real motion control problems typically follow reasonably smooth curves. An extremely short duration actuator input spike has little effect on dynamic systems, and is also likely to cause motions that are jerky or awkward, not smooth. This by its very nature suggests that the regulation of actuator inputs will be a smooth and relatively slow-varying operation in any good solution.

In generating a "random" torque profile, then, a scout should not consider setting the torque level to a random value at each step. Rather, the scout should merely adjust the torque slew rate; that is, the rate of change of the torque. Furthermore, it makes no sense to rapidly alternate between slewing torque up and then slewing it down, unless the torque value is near zero. We can in fact reduce the number of torque slew actions to a fairly small number without losing the ability to generate complex torque profiles. For the purposes of this example, only the following five actions were considered:

1) Fast ramp increase
2) Slow ramp increase
3) Maintain current torque
4) Slow ramp decrease
5) Fast ramp decrease

Furthermore, the decision branching factor may be additionally reduced by limiting each decision to an adjustment of the slew rate of one step in either direction. This recognizes that it should not be necessary to follow a rapid torque increase with a rapid decrease. Following action 2, for example, a scout should only consider actions 1, 2, or 3. Following action 1, it should only consider actions 1 or 2. Thus, the average branching factor for the force adjustment decision problem is reduced to $\frac{3 \cdot 3 + 2 \cdot 2}{5} = 2.6$.

To determine whether a torque profile moves the system to the desired position, the system must be simulated. This is easily accomplished with numerical integration of the system equations. Though the equations may be nonlinear, this is immaterial for numerical integration since we are only simulating the system in a forward direction and not attempting to invert it.

Since the actuator inputs are specified by a slew rate, the inputs must be adjusted at each step of the numerical integration according to the slew rate selected. In this way, an accurate numerical integration, which requires small time steps, can be coupled with a more sensible slew adjustment rate, which would be expected to need updating much less frequently.

Based on this discussion, a simulation program for motion planning will now be developed.

6.5.1 Situation And Action Structure

Situation will be represented by theta and omega, and as the simulation progresses it will update the future time so it is easy to see how far in the future a given situation is.

```
// Situation
//     theta,omega,t
#define theta s[0]
#define omega s[1]
#define t s[2]
```

The action structure keeps track of the current tau and the dTau slew rate. The direction of the tau slew is reprented as an integer from –2 to 2, representing from fast decrease to fast increase.

```
// Action
//     dTau,Tau,dt
#define dir a[0]
#define dTau a[1]
#define Tau a[2]
#define dt a[3]
```

6.5.2 Pick Action

The PickAction simulation code simply looks at the last action, which specifies a slew direction from –2 to 2. It adjusts this value up to one step in either direction without going below –2 or above 2. This slew index is then multiplied by a motion constant to create the physical t slew rate. In addition, the PickAction code anticipates what the final torque will be after the next set of numerical integrations. If the slew rate chosen would cause the torque to go out of physical limit, it adjusts the slew rate such that the torque will just barely stay in limit.

```
void PendulumSimulator::PickAction(double* a)
{
       int curDir=dir;
       int d;
       if(curDir==-2)
       {
              d=rand()%2;
       }
       else if(curDir==2)
       {
              d=rand()%2-1;
       }
       else
              d=rand()%3-1;

       curDir+=d;
       dir=curDir;

       dTau=(double)(curDir)/2.*tauSlew;

       // anticipate and prevent overshooting Tau limits
       double tauDist=dTau*(double)integrationsPerStep;
       if(Tau+tauDist>maxTau)
              dTau=(maxTau-Tau)/integrationsPerStep;
       else
              if(Tau+tauDist<minTau)
                     dTau=(minTau-Tau)/integrationsPerStep;
}
```

6.5.3 Next Situation

Given the current model position, the NextSituation simulation code integrates the system model a programmable number of times to move the model forward. This is a simple Euler integration; more advanced techniques are possible but not necessary for small enough step sizes.

```
void PendulumSimulator::NextSituation(double* s,double* a)
{
       for(int i=0; i<integrationsPerStep; i++)
       {
              // straight Euler technique; walk the system model
              Tau += dTau; // keep Tau ramping

              double thetadot = omega;
              double omegadot = MLLinv * (Tau - B*omega –
                                   MgL*sin(theta));

              theta = theta + thetadot*dt;
              omega = omega + omegadot*dt;

              t+=dt;
       }
}
```

6.5.4 Goal Check

The GoalCheck simulation simply checks to see if the final theta and omega of the model are within limits, although two other commented out possibilities are shown in the simulation code. One example forces the solution to occur within a particular time window. This would cause a good position/velocity to fail unless it was inside the specified time window. The other modification shows steady state checking, in which the model would be required to be inside the final position/velocity window for several time periods before a successful trajectory would be declared.

```
BOOL PendulumSimulator::GoalCheck(double* s)
{
       // EXAMPLE: Force solution at a specific time
       //if(t<4.5 || t>5.0) return FALSE;

       // STANDARD:  Are we in position/velocity spec
       if(fabs(theta-finalTheta)<finalThetaTol &&
              fabs(omega-finalOmega)<finalOmegaTol)
              return TRUE;
       else
              return FALSE;

       // EXAMPLE:  Force steady state solution: check to see
       //  if we have been in the final state for at least K timeslots
       //if(fabs(theta-finalTheta)<finalThetaTol &&
       //     fabs(omega-finalOmega)<finalOmegaTol)
       //{
       //     nConforming++;
       //     if(nConforming>K) return TRUE;
       //     else return FALSE;
       //}
       //else
       //{
       //     nConforming=0;
       //     return FALSE;
       //}
}
```

6.5.5 Simulation with Plenty of Torque

A complete simulation was developed and tested.[22] The simulation is virtually identical to the earlier 8-puzzle simula tion with the replacement of the PickAction, NextSituation, and GoalCheck functional units as described above.

The complete system was simulated for several different types of constraints. In each case, the goal of inverting a motionless hanging pendulum was used. This corresponds to an intial condition of $q = w = 0$ and a final condition of $q = p$, $w = 0$. A goal tolerance of $\pm 2°$ position and rotational velocity $\pm 2° / s$ was allowed. M and g were set to 1, and L was set to 2. The system model was integrated for eight 0.025 s steps after each torque slew choice was made. Thus, the motion plan is updated at a 5 Hz rate, and the model simulation proceeds at 40 Hz.

The following motion plan resulted from running 30,000 scouts with torque constrained to a range of –8 to +8. The "best" scout was chosen based on number of actions, which translates in this case to the actual runtime of the motion.

This represents a 15-step solution in which the torque slew rate is adjusted 15 times. Since each slew choice is maintained for eight 0.025 s integrations, an adjustment choice is made every 0.2 S and the entire solution takes 3.0 s. As expected, the winning plan involves slewing torque to the maximum as fast as possible, and then slewing to a large negative amplitude at the right moment to arrest motion just as the pendulum hits the apex.

[22] Lecky (1999). To be published in Conference Proceedings.

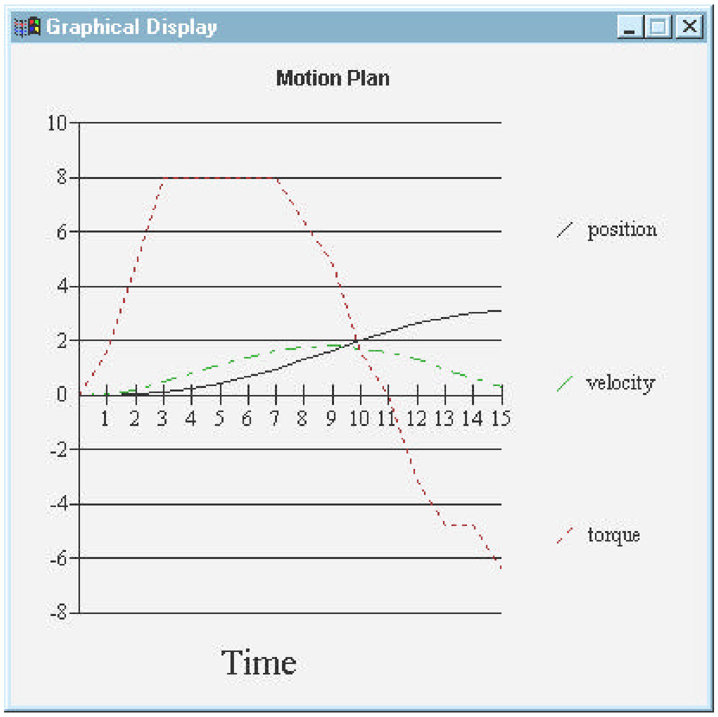

Figure 23: Minimally Constrained Motion Plan

To actually execute this motion plan, the $q(t)$ and $w(t)$ profiles would be fed to a simple motion controller. This controller could make use of the $t(t)$ plan actuator input and merely adjust torque slightly up or down from that level based on positional deviation from plan. In this regard, the motion controller becomes much less complex for the planned motions generated by the scouting approach.

6.5.6 Simulation with Torque Severely Constrained

The solution becomes much more interesting when torque is constrained to stay between –1.5 and 1.5. The best results of 30,000 scouts with all other parameters held constant is shown below.

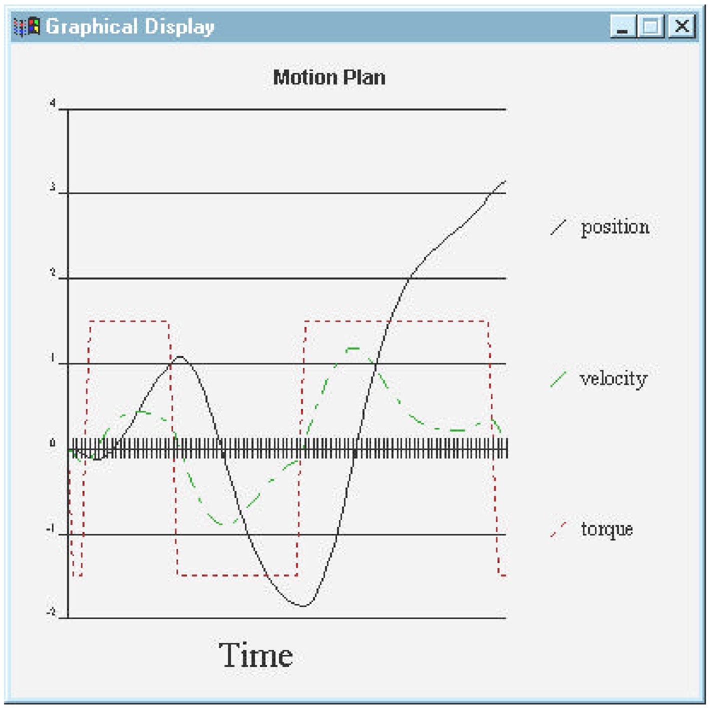

Figure 24: Symmetrically Constrained Motion Plan

Here, the maximum torque of 1.5 is not enough force to lift the mass directly to the vertical position. The found solution instead swings the weight backward, forward, backward, and then forward again to achieve inversion. Developing an analytical technique that would have discovered this solution is difficult to imagine. Using scouting, however, if there is a way to achieve the goal, it will eventually be discovered. The solution displayed required 100 scout actions, representing an elapsed time of 20 s to achieve the goal.

6.5.7 Simulation with Assymetric Torque

If torque is constrained asymmetrically from –6 to 2, a sensible solution is also found.

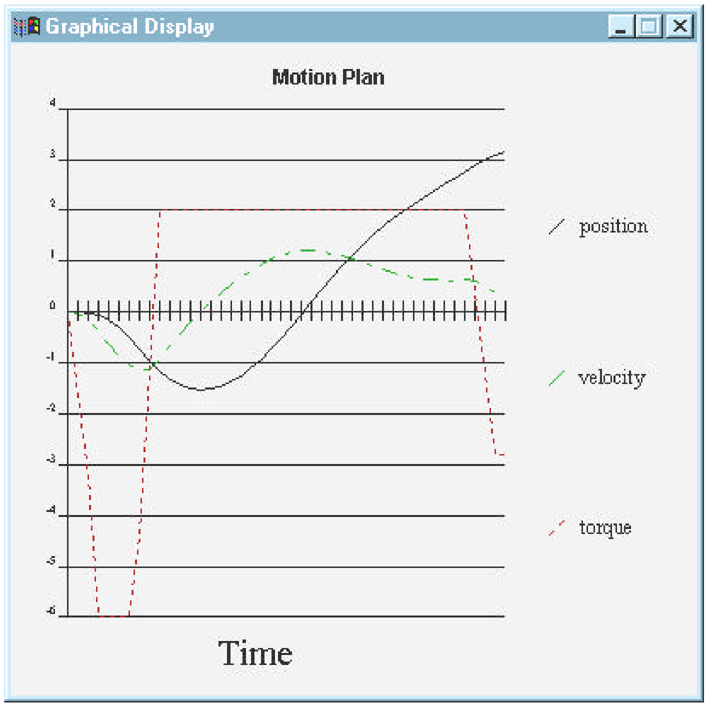

Figure 25: Assymetrically Constrained Motion Plan

This solution required 44 scout actions, representing 8.6 s motion time. The strong negative torque is used to rock the pendulum back approximately 90 degrees (1.5 rad), and then the torque is slewed to its maximum value to swing forward and up to the vertical position.

6.5.8 Simulation with Nonlinear Damping

As a final illustration of the power of forward simulation, this example uses non-linear damping of the form:

$$B(t) = \begin{cases} 1.5(5-t) & t \le 5 \\ 0 & otherwise \end{cases}$$

Torque is constrained to –1.5 to 1.5, and the following 49-step (9.8 s) solution was found by 10,000 scouts:

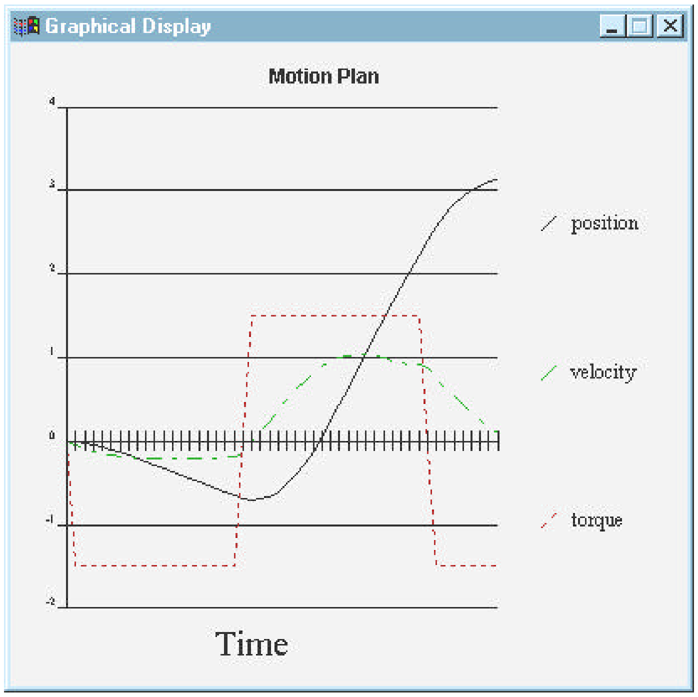

Figure 26: Constrained Motion with Non-linear Damping Plan

Here, since damping decreases with time, it makes most sense to move backwards at the most negative torque until nearly the midpoint of the motion, at which time damping is nearing 0. Now, we slew to maximum torque and rapidly hit maximum velocity. Braking torque is applied earlier than in the previous examples since there is no longer any braking assistance from the damping.

6.5.9 Motion Planning Summary

The motion plans developed by this simple example program are in many ways quite surprising. Without knowing how they were developed, an observer would be tempted to say that they exhibited insight, cunning, grace, and elegance. Human beings, masters at fluid motion control representing elegance and grace, are often surprised that a machine could "think" of such a smooth and efficient solution.

Of course, once the ghost in the machine is understood, the elegance is easily explained. The scouting system has tried literally thousands of torque profiles and merely selected the one that achieved the goal as quickly as possible. The interesting thing about these solutions is that they were directly generated from nothing more than an actuator-to-movement model of the system, which can be learned purely from experimental data. In effect, this solution proposes a model that explains how an unintelligent system can develop extremely insightful solutions using nothing more than a model of experience developed from trial-and-error.

While this example of scouting behavior is in some ways more impressive than the 8-puzzle problems used to illustrate the technique earlier in the dissertation, there is really nothing different about this application of scouting. The only creative part of the process involves how the problem is represented to the architecture. Of course, other complex problems could be similarly structured to use scouting. There is nothing special about solving motion planning problems, just as there is nothing special about solving 8-puzzles.

7 Looking Forward

This initial look at scouting revealed a technique with broad application in a wide variety of difficult situations. Whether the model is closely aligned with biological processing or not will be shown in years to come. In any event, scouting represents an important processing technique that can take non-deterministic steps to develop solutions to problems that are very difficult to solve.

The theoretical developments in this dissertation allow estimation of scouting performance on arbitrary problems. This capability is critical to estimating both the likelihood of success and required solution time for various problems.

Certain aspects of this dissertation seem a bit regenerative; the more answers you get, the more questions become obvious. Scouting as a general strategy will need to be implemented and applied on many fronts before all of the details become apparent. Several key issues are so tempting to address that they must at least be introduced now. Each of these represents a complete research topic for future study.

7.1 Improving Data Representation

This dissertation focuses on the computational aspect of intelligence; how information is stored is likely to play an important role as well. It is highly likely that an intelligent data storage model is used in the brain, and that this storage model enhances our ability to seem intelligent. How a problem is represented can have many impacts on the complexity and efficiency of scouting, and has implications for learning and memory as well. These ideas are briefly explored next.

7.1.1 Optimizing for Minimal Hardware

At the scouting level, we consider memory only as it relates to remembering the rules and goals of a problem. The actual implementation of the PA, GC, and NS logic developed for our simple games requires quite sophisticated combinational logic that would require many neurons to simulate. It is possible that redundant information in the form of representing the situation in several alternate formats could reduce the complexity of this logic.

Consider that a checkerboard can be represented linearly, in that the pieces could be laid out in a straight line 32 positions long (ignoring the positions in the 8x8 board that never contain a piece). Moves would then be represented by moving the checkers 7 or 9 positions left or right (although there are also rules about watching the linear position mod 8 to make sure you don't move off the right edge of the board and onto the left). Similarly, jumps would be represented by moves of 14 or 18 spaces left or right, as long as the position half-way between the initial position and the target position is occupied by the opponent (with the same rules of not passing off one side and onto the other applied).

Playing checkers in this format would probably be impossible for humans. The human visual cortex develops many alternate representations of vision data which contain different information useful for different purposes. This redundant data could be used to develop smaller or more efficient versions of the combinational logic.

7.1.2 Assisting in Learning

A good problem representation will certainly reduce the amount of hardware necessary for implementing scouts particular to a given problem. In addition, the right data representation could allow new rules to be added to the rule database. These would be "rules-of-thumb" noticed during problem solving, or noticed during failure, and then added to the given rules of the problem to enhance future solutions.

This second item is obviously of extreme interest. By developing relationships between various data representations, and correlating them with success and failure, it may be possible for a machine to effectively increase its own PA or GC circuitry to improve the quality of its own play. This provides a mechanism for experiential development of reasonable subgoals for solving complicated problems, as was found to be important for the 15-Puzzle.

More generally, learning is important in humans in three major areas:

1) Learning more raw information (declarative information)

2) Learning how to do things (procedural information)

3) Remembering the past (so we can choose to repeat it or not) (episodic information)

Some interesting insight into learning comes from the study of "experts." Humans can become expert at specific tasks through practice and repetition. Expert chess players, for example, can remember actual chess game board situations much better than non-experts. They do no better than non-experts at remembering *random* chess board arrangements, however.[23]

This suggests that an important part of learning is the representation of the problem in an efficient way. Mathematically, we can think of representing vectors in linear algebra as weighted sums of a particular set basis vectors. If a memory consists of *n* simultaneous inputs, we could represent it as:

[23] Matlin (1998), p. 364 cites the work of DeGroot (1966), Simon & Chase (1973), Vincente & DeGroot (1990)

$$M_1 = \sum_{k=0}^{n-1} a_k b_k$$

where a_k are weights, and b_k are the n-dimensional basis vectors. If the basis vectors b_k are chosen well, many different memories could be represented efficiently by a fairly small number of non-zero a_k coefficients. The development of many different sets of basis vectors could be directed simply by combining commonly recurring patterns in sensory input, which might correspond to phonemes in auditory recognition, for example.

It has been stated that associative memory is the “fruit fly” or “Bohr atom” problem of neural computing.[24] Human memory performs lookup by association, not by direct addressing. A vector-basis representation of memory combined with scouting yields associative memory as an indirect consequence. By finding the memory with the closest set of a_k coefficients to the current situation, we have identified a similar memory, irrespective of the fact that the actual sensory inputs coming from the new situation might be vastly different from those that created the stored memory.

The quality of the associativity, or the “expertness,” of the analysis, is based on the quality and richness of the basis vectors

7.2 A Complete Thinking Machine

A design for a complete “thinking machine” is possible based on these ideas.. A good design would require the ability to issue multiple simultaneous searches. That is, it should be able to walk and chew gum at the same time, or at least listen and prepare a response at the same time.

[24] Hertz (1991), p. 11

This can be handled by partitioning the scouts into groups, and sizing the groups based on the priority of the present tasks.

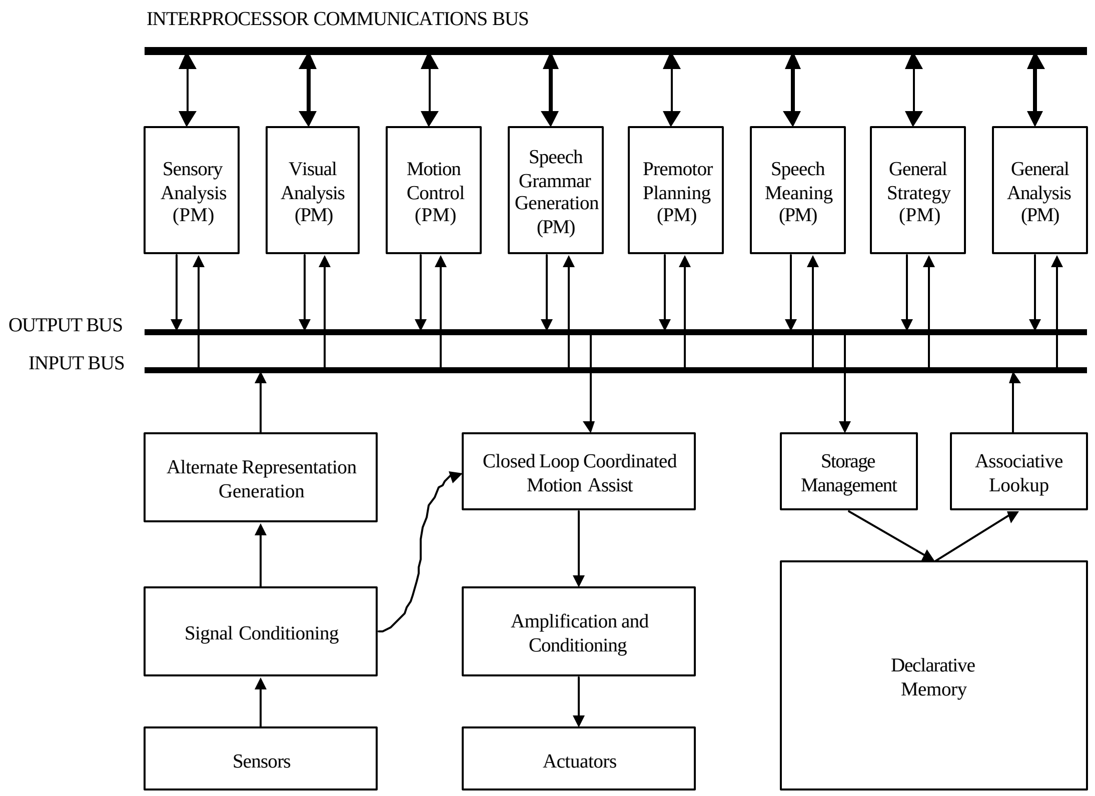

Figure 27: Comprehensive Brain Processing Model

Implementation of such a design is within reach, especially using electronically programmable CPLDs. This reprogrammable ASIC technology would allow reprogramming of the scouts on the fly, permitting dynamic load balancing across system functions, while eliminating the requirement for huge neural fan-in. In this sense, the thinking machine would actually have completely uniform scout macrocolumns which could be redistributed to different functions based on some overall master control program which is not itself yet obvious.

7.3 In Closing

Scouting paired with reasonably efficient memory storage strategies gives a complete solution to intelligent computing; the only question is whether (or how soon) a machine will be built that can contain enough scouts, and have enough scout-memory communications bandwidth, to duplicate human intelligence. Give the current state of electronics and its rate of improvement, this goal may not be too far off.

8 Bibliography

Much of the anatomical and physiological information in this dissertation comes from *Gray's Anatomy*, 38th Edition (1995), published by Churchill Livingstone, and edited by Bannister, Berry, Collins, Dussek, Dyson, Ellis, Gabella, Salmons, Soames, and Standring. This is really THE authority and state-of-knowledge on human anatomy.

Other References:

Allport, Susan (1986). *Explorers of the Black Box*. W.W. Norton & Co.

Bardell, Paul, William H. McAnney, and Jacob Savir (1987). *Built-In Test for VLSI: Pseudorandom Techniques*. John Wiley and Sons.

Cairns-Smith, A. G. (1996). *Evolving the Mind*. Cambridge University Press.

Caudill, Maureen and Charles Butler (1990). *Naturally Intelligent Systems*. MIT Press.

Chen, C. H. (Ed) (1996). *Fuzzy Logic and Neural Network Handbook*. McGraw-Hill.

Cherkassky, Vladimir and Filip Mulier (1998). *Learning from Data*. John Wiley and Sons.

De Becker, Gavin (1997). *The Gift of Fear*. Little, Brown, and Co.

Epp, Susanna S. (1995). *Discrete Mathematics with Applications*. Brooks/Cole Publishing.

Franklin, Gene F., Powell, J. David and Workman, Michael L. (1990). *Digital Control of Dynamic Systems*, 2nd Edition. Addision-Wesley.

Greenfield, Susan A. (1997). *The Human Brain*. Harper Collins.

Hertz, John, Anders Krogh, and Richard G. Palmer (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley.

Hill, Frederick J, and Gerald R. Peterson (1978). *Digital Systems: Hardware Organization and Design*. John Wiley and Sons.

Khanna, Tarun (1990). *Foundations of Neural Networks.* Addison-Wesley.

Kirk, Donald E. (1970). *Optimal Control Theory*. Prentice-Hall.

Kurzweil, Ray (1999). *The Age of Spiritual Machines*. Penguin Books.

Lecky, John E. (1999). *Using Scout Clusters for Motion Planning*. International Joint Conference on Neural Networks. To be presented 7/99.

LLin·s, Rodolfo R (Ed.) (1988). *The Biology of the Brain. Readings from Scientific American*. W.H. Freeman and Co.

Matlin, Margaret W. (1998). *Cognition*, 4th Edition. Harcourt Brace.

Polivka, Raymond P. and Sandra Pakin (1975). *APL: The Language and Its Usage*. Prentice-Hall.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery (1992). *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press.

Rietman, Ed (1988). *Experiments in Artificial Neural Networks.* Tab Books.

Russell, Stuart and Peter Norvig (1995). *Artificial Intelligence*. Prentice Hall.

Stark, Henry and John Woods (1986*). Probability, Random Processes, and Estimation Theory for Engineers*. Prentice-Hall.

Sedgewick, Robert (1990). *Algorithms in C*. Addison-Wesley.

Squire, Larry R. (1987). *Memory and Brain*. Oxford University Press.

Welstead, Stephen T. (1994). *Neural Network and Fuzzy Logic Applications in C/C++*. John Wiley and Sons.

9 Appendix A: Scouting Simulator for the 8-puzzle

The simulation code below is a complete example which runs both 0-scout and 1-scouts on the specific test puzzle mentioned in the text, as well as randomly generated puzzles. A sample output run is included in Appendix B.

```
// Scout Cluster Simulation
// 8-puzzle
// Dissertation Example
// 1/18/99  John E. Lecky

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

#define TRUE -1
#define FALSE 0


// DATA REPRESENTATION
// Situation representation
// int situation[10];
//  position map 0 1 2
//               3 4 5
//               6 7 8
//  9 is redundant location of blank
// Thus the 8-puzzle:
//  1 8 3
//    4 2
//  5 7 6
// is represented as {1,8,3,0,4,2,5,7,6,3}


// Action representation
// int action[2];
//  0= move blank from position...
//  1= to position
// Thus if the first move in the example puzzle above is to move
// tile 5 up, the action is {3,6}



// specific test puzzle from the dissertation
int testPuzzle[]={
    1,8,3,0,4,2,5,7,6,
    3};

// desired goal situation
int Sg[]={
    // 1 2 3
    // 8 0 4
    // 7 6 5  with blank in pos 4
    1,2,3,8,0,4,7,6,5,
    4};

// RULES
// Rules are implemented as a list of possibilites
int rndMove8[][5]={
    {2,1,3,0,0},  // with blank in pos 0, there are 2 moves, 1 and 3
    {3,0,2,4,0},  // with blank in pos 1, there are 3 moves, 0, 2, and 4
    {2,1,5,0,0},  // with blank in pos 2, there are 2 moves, 1 and 5
    {3,0,4,6,0},  // etc.
    {4,1,3,5,7},
    {3,2,4,8,0},
    {2,3,7,0,0},
    {3,4,6,8,0},
    {2,5,7,0,0}};

// FUNCTION PROTOTYPES
int VRand(int max);
void ScramblePuzzle(int* s,int n);
void ShowSituation(int* s);
int GC(int* s);
void PA_Scout0(int* a,int* s);
void PA_Scout1(int* a,int* s);
void NS(int* s, int* a);
int Scout(int* s,int d0);
int ScoutCluster(int* s,int d0);
void Solve(int* s,int n,int k,int d0);

// Function pointer to allow use of 0-scout or 1-scout
void (*PA)(int* a,int* s);

// Global monitoring variables
char soln[1024];      // solution proposed by current scout
char bestSoln[1024]; // best solution found so far
int bestIter;             // scout number that found bestSoln

// Return a pseudo random integer 0 - max-1
// NOTE: C rand() function returns 0 to 32767.  This implementation
//   doubles the range since it calls rand() an odd number of times
int VRand(int max)
{
    return (rand() | rand()<<15 | (rand()&1)<<30)%max;
}

// Make n random moves in a puzzle; useful for generating test puzzles
void ScramblePuzzle(int* situation,int n)
{
    int action[2];
    action[0]=-1;

    for(int i=0; i<n; i++)
    {
         PA_Scout1(action,situation);
         NS(situation,action);
    }
}

// Display a game situation
void ShowSituation(int* s)
{
    printf("-----\n");
    for(int i=0; i<3; i++)
    {
         char buf[28];
         buf[3]=0;
         for(int j=0; j<3; j++)
              buf[j]='0'+s[i*3+j];
         printf("|%s|\n",buf);
    }
    printf("----%1d\n\n",s[9]);  // shows redundant blank pointer
}

// Is situation in the goal situation?
int GC(int* s)
{
    return 0==memcmp(s,Sg,9*sizeof(int));
```

```
}

// Given the current situation, pick a random action
// 0-scout version; no looking back at previous moves
void PA_Scout0(int* a,int* s)
{
    // pick a random move using the possible move table
    int blankPos=s[9];
    int n=rndMove8[blankPos][0];
    int move=rndMove8[blankPos][VRand(n)+1];

    // build action = from,to
    a[0]=blankPos;
    a[1]=move;
}

// Given the current situation, pick a random action
// 1-scout version; looking back at last move only
void PA_Scout1(int* a,int* s)
{
    int blankPos=s[9];
    int lastBlankPos=a[0];

    // keep picking random moves until we aren't backtracking
    int move=lastBlankPos;
    while(move==lastBlankPos)
    {
          int n=rndMove8[blankPos][0];
          move=rndMove8[blankPos][VRand(n)+1];
    }

    // build action = from,to
    a[0]=blankPos;
    a[1]=move;
}

// Perform action a on situation s
void NS(int* s, int* a)
{
    int from=a[0];
    int to=a[1];

    s[from]=s[to];  // move the blank
    s[to]=0;
    s[9]=to;         // update redundant position
}

// Scout Simulator
// return depth of found solution or -1 if no solution found
int Scout(int* Si,int d0)
{
    int s[10];
    memcpy(s,Si,10*sizeof(int));

    int a[2];
    a[0]=-1;  // invalid initial action to be ignored

    int d=0;
    soln[0]=0;

    char tbuf[2];
    tbuf[1]=0;
    while(!GC(s) && d<d0)
    {
          (*PA)(a,s);
          NS(s,a);
          tbuf[0]=a[1]+'0';
          strcat(soln,tbuf);
          d++;
    }

    if(d<d0)
          return d;
    else
          return -1;
}

// Implement the complete scout cluster
int ScoutCluster(int* s,int n,int d0)
{
    int initd0=d0;
    bestIter=0;
    bestSoln[0]=0;

    for(int i=0; i<n; i++)
    {
          int d=Scout(s,d0-1);
          if(d>=0)
          {
                printf("scout %d found solution d=%d\n",i,d);
                d0=d;
                bestIter=i;
                strcpy(bestSoln,soln);
          }
    }

    if(d0<initd0) return d0;
    else return -1;
}

// Solve puzzle s n times with k scouts looking to a maxdepth d0
void Solve(int* s,int n,int k,int d0)
{
    for(int i=0; i<n; i++)
    {
          int d=ScoutCluster(s,k,d0);
          printf("best d=%d scout=%d ",d,bestIter);
          printf("moves=%s\n",bestSoln);
    }
}

// Run the complete simulation
int main()
{
    srand(1);

    // create a few random puzzles
    int sTest[3][10];
    for(int p=0; p<3; p++)
    {
          // build a random puzzle by making 1000000 random moves
          // starting from the goal situation
          memcpy(sTest[p],Sg,10*sizeof(int));
          ScramblePuzzle(sTest[p],1000001);
    }


    // run the examples with scout-0 first, then scout-1
    for(int i=0; i<2; i++)
    {
          if(i==0)
          {
                PA=&PA_Scout0;
                printf("Scout-0 TESTING\n");
          }
          else
          {
                PA=&PA_Scout1;
                printf("Scout-1 TESTING\n");
          }

          // run the test example from the dissertation
          int s[10];
          memcpy(s,testPuzzle,10*sizeof(int));
          ShowSituation(s);
          Solve(s,10,100000,100);

          // run the random examples
          for(p=0; p<3; p++)
          {
                ShowSituation(sTest[p]);
                Solve(sTest[p],10,100000,100);
          }
    }

    return 0;
}
```

10 Appendix B: Sample Run of the Scouting Simulator

This appendix provides a complete output listing generated by the scout cluster simulator in Appendix A.

The listing is divided into 2 sections, scout-0 testing and scout-1 testing. Each puzzle simulation starts with a graphic of the initial puzzle situation followed by listing reporting each scout that finds a successively better solution. The last scout improvement represents the "best" solution, which is reported along with the complete list of moves found to solve the puzzle.

The move list is interpreted as the sequence of positions through which the blank space moves.

10.1 Scout-0 Testing

```
Scout-0 TESTING
-----
|183|
|042|
|576|
----3

scout 7125 found solution d=81
scout 13068 found solution d=79
scout 20133 found solution d=49
scout 26410 found solution d=39
best d=39 scout=26410 moves=036367452147634367430125874103454587674
scout 7414 found solution d=61
scout 38348 found solution d=59
scout 42699 found solution d=47
scout 88603 found solution d=45
best d=45 scout=88603 moves=678785436785478525214147476363012585854121034
scout 7449 found solution d=83
scout 19100 found solution d=61
best d=61 scout=19100
moves=4103454341210101036343634103676787452103014301254345478541034
scout 1820 found solution d=83
scout 3136 found solution d=67
best d=67 scout=3136
moves=4541474585878521414301258787434767636785410367678541436745876763034
scout 1770 found solution d=85
scout 5895 found solution d=57
scout 21361 found solution d=51
scout 36452 found solution d=45
best d=45 scout=36452 moves=010345252147474141478763476301212541036347854
scout 8297 found solution d=61
scout 18587 found solution d=47
best d=47 scout=18587 moves=67458521010141434303012121034301254101034785254
scout 4082 found solution d=87
scout 27764 found solution d=51
best d=51 scout=27764 moves=636785854543036767874101258745210367630303630147634
scout 9109 found solution d=67
scout 62410 found solution d=63
scout 64681 found solution d=47
best d=47 scout=64681 moves=67636785874585852141030147674301254767852541034
scout 11494 found solution d=79
scout 14716 found solution d=49
best d=49 scout=14716 moves=4763452101476301258547852585212543014301030121014
scout 13519 found solution d=81
scout 14447 found solution d=75
scout 70490 found solution d=73
scout 94940 found solution d=63
best d=63 scout=94940
moves=012125858545254367634367636747854125410101212543674525876763014
-----
|764|
|830|
|521|
----5

scout 39516 found solution d=81
best d=81 scout=39516
moves=214521036763676367854525874543436747858741478587854303410367454363636785410363034
scout 1283 found solution d=85
best d=85 scout=1283
moves=4525478585454763010343012143458587678541014303012125258587452103014143036767414521
214
scout 13856 found solution d=91
scout 39977 found solution d=49
best d=49 scout=39977 moves=2141436367852145476303630121252147630147874345874
scout 43425 found solution d=89
best d=89 scout=43425
moves=4301474547878543036343414767634103634741254767678547878525854101410143412587430363
4147854
scout 15247 found solution d=91
best d=91 scout=15247
moves=8787630145214301254145878545258587858587436347636741014141030121436367676787874587
852101014
scout 90865 found solution d=61
best d=61 scout=90865
moves=4125854145852543010147678525254763630147852543678587854763014
scout 5351 found solution d=83
scout 25999 found solution d=67
scout 42846 found solution d=57
best d=57 scout=42846 moves=254301254763012143434521014143430125874525452143636301254
scout 930 found solution d=75
best d=75 scout=930
moves=214787636767852585410125254521014585412547630301474363012121434125878743674
scout 1742 found solution d=93
scout 60392 found solution d=83
scout 92704 found solution d=77
best d=77 scout=92704
moves=43036303414125876341452125476741474121010125458763030367412547634543012547634
scout 29829 found solution d=83
best d=83 scout=29829
moves=2585878521454763678545252545434301478587636367630143676301258767678525454101012587
4
-----
|251|
|048|
|376|
----3

scout 1066 found solution d=53
best d=53 scout=1066 moves=67878743012585430367634341210363410125874745876741034
scout 36120 found solution d=53
best d=53 scout=36120 moves=67476767474525210341034521212121014547634367854301214
scout 9726 found solution d=73
scout 45734 found solution d=45
best d=45 scout=45734 moves=010363636785214521014301014367678521476303634
scout 10226 found solution d=87
scout 22144 found solution d=79
scout 44579 found solution d=77
best d=77 scout=44579
moves=67878785214545212103430341034367874787878525858785478587858763036345878585214
scout 16189 found solution d=51
best d=51 scout=16189 moves=452143674521254767430125412587874301474101414587874
scout 7512 found solution d=93
scout 21816 found solution d=71
best d=71 scout=21816
moves=45436303010121034545212547474787854745210103434525876345852103478541254
scout 16268 found solution d=97
scout 28735 found solution d=77
scout 37430 found solution d=61
best d=61 scout=37430
moves=4345210367410343430101210345452103474525476787876767852103634
scout 3509 found solution d=49
best d=49 scout=3509 moves=6745214587874301214347876747878525214301010147854
scout 82208 found solution d=85
scout 87503 found solution d=49
best d=49 scout=87503 moves=4125410367854541034347452521414341010345252141034
scout 9089 found solution d=79
scout 59049 found solution d=57
best d=57 scout=59049 moves=678541034125430125454743636785858521254767878763030341254
-----
|167|
|584|
|302|
----7

scout 45959 found solution d=93
best d=93 scout=45959
moves=4763030301474587674787454587878547678743012525252587630141030121254521012141436301
01258767634
scout 7648 found solution d=87
scout 52425 found solution d=81
scout 54065 found solution d=77
best d=77 scout=54065
moves=87630125412143030147634525430101258525878525858585214143458763474525258521214
scout 69194 found solution d=77
best d=77 scout=69194
moves=87634587858787476787674767874303476741258747878767876341012545258741436367674
scout 19350 found solution d=83
scout 87178 found solution d=71
best d=71 scout=87178
moves=63474587452121014547676787874545210347434121252145876763630125210125874
scout 3548 found solution d=85
scout 34631 found solution d=77
best d=77 scout=34631
moves=63458547630367874767410147852141474525452103636743636743452125258745436301214
scout 12869 found solution d=55
best d=55 scout=12869 moves=6343634785254585852147876345874303636785214125214145874
scout 54214 found solution d=75
best d=75 scout=54214
moves=636347410121254545876347874587854125852521012585252125876787436367410101254
scout 3397 found solution d=95
scout 78690 found solution d=87
best d=87 scout=78690
moves=6303452147630363678525214747876341452547854303034343010301010347676743676785852585
25214
scout 13445 found solution d=71
scout 83515 found solution d=69
best d=69 scout=83515
moves=674521252521252103036745854587858587876767476743014367874341254125874
best d=-1 scout=0 moves=
```

10.2 Scout-1 Testing

```
Scout-1 TESTING
```

```
-----
|183|
|042|
|576|
----3

scout 4925 found solution d=69
scout 11692 found solution d=57
scout 15129 found solution d=29
scout 30450 found solution d=23
scout 56727 found solution d=21
best d=21 scout=56727 moves=678543674125876345214
scout 4442 found solution d=21
best d=21 scout=4442 moves=678543674125876345214
scout 7 found solution d=81
scout 1529 found solution d=53
scout 7324 found solution d=33
scout 7558 found solution d=31
scout 27527 found solution d=25
scout 32459 found solution d=21
best d=21 scout=32459 moves=674521430125410347854
scout 286 found solution d=39
scout 6834 found solution d=25
scout 15897 found solution d=23
scout 81253 found solution d=21
best d=21 scout=81253 moves=678543674125876345214
scout 70 found solution d=23
scout 14736 found solution d=21
best d=21 scout=14736 moves=678543674125876345214
scout 7712 found solution d=55
scout 9887 found solution d=45
scout 10950 found solution d=39
scout 12868 found solution d=33
scout 15688 found solution d=25
best d=25 scout=15688 moves=4103458763478543014367854
scout 3189 found solution d=73
scout 4944 found solution d=35
scout 13214 found solution d=25
scout 71902 found solution d=23
scout 94812 found solution d=21
best d=21 scout=94812 moves=678543674125876345214
scout 1878 found solution d=87
scout 2424 found solution d=49
scout 5684 found solution d=39
scout 21233 found solution d=25
scout 33940 found solution d=23
best d=23 scout=33940 moves=67452143012587458741034
scout 2721 found solution d=93
scout 2768 found solution d=31
scout 18600 found solution d=23
scout 20525 found solution d=21
best d=21 scout=20525 moves=678543674125876345214
scout 459 found solution d=69
scout 6867 found solution d=29
scout 9654 found solution d=21
best d=21 scout=9654 moves=678543674125876345214
-----
|764|
|830|
|521|
----5

scout 665 found solution d=71
scout 1794 found solution d=41
scout 37105 found solution d=27
best d=27 scout=37105 moves=874587630143012587436741254
scout 45 found solution d=85
scout 389 found solution d=41
scout 1557 found solution d=39
scout 9417 found solution d=35
scout 10934 found solution d=29
scout 43174 found solution d=27
best d=27 scout=43174 moves=874103412587634125876301254
scout 4232 found solution d=93
scout 4829 found solution d=71
scout 5243 found solution d=69
scout 8896 found solution d=33
best d=33 scout=8896 moves=214587430125876301430125430125874
scout 1629 found solution d=57
scout 3302 found solution d=49
scout 5933 found solution d=41
scout 31424 found solution d=31
scout 39981 found solution d=27
best d=27 scout=39981 moves=874103412587634125876301254
scout 3371 found solution d=31
scout 99360 found solution d=29
best d=29 scout=99360 moves=87634587630143674301452143674
scout 6171 found solution d=51
scout 7175 found solution d=45
scout 11919 found solution d=43
scout 19264 found solution d=39
scout 35400 found solution d=31
scout 40437 found solution d=27
best d=27 scout=40437 moves=874125876341258763012547634
scout 1318 found solution d=91
scout 6094 found solution d=65
scout 8036 found solution d=41
scout 15549 found solution d=27
best d=27 scout=15549 moves=874125876341258763012547634
scout 9 found solution d=41
scout 27029 found solution d=39
scout 54468 found solution d=31
best d=31 scout=54468 moves=8741258763012587430125874367854
scout 75 found solution d=95
scout 2138 found solution d=91
scout 3515 found solution d=85
scout 4904 found solution d=63
scout 11586 found solution d=57
scout 15369 found solution d=39
scout 17161 found solution d=31
best d=31 scout=17161 moves=4103412587452147634587630145214
scout 2737 found solution d=43
scout 3505 found solution d=41
scout 14415 found solution d=29
scout 50264 found solution d=27
best d=27 scout=50264 moves=874521458743012587634125874
-----
|251|
|048|
|376|
----3

scout 2466 found solution d=45
scout 3532 found solution d=35
scout 41768 found solution d=27
best d=27 scout=41768 moves=452103674521034785210345214
scout 1072 found solution d=53
scout 13351 found solution d=49
scout 15678 found solution d=35
scout 66633 found solution d=31
scout 86565 found solution d=29
best d=29 scout=86565 moves=67430125410345214521478541034
scout 4212 found solution d=31
scout 6038 found solution d=29
scout 22841 found solution d=27
best d=27 scout=22841 moves=678521452103410367852147634
scout 43 found solution d=95
scout 2505 found solution d=91
scout 8923 found solution d=39
scout 11751 found solution d=33
scout 44085 found solution d=31
scout 45372 found solution d=29
best d=29 scout=45372 moves=45210341034521436743014785214
scout 3721 found solution d=79
scout 5936 found solution d=65
scout 12410 found solution d=31
scout 31301 found solution d=27
best d=27 scout=31301 moves=678521452143014367852147634
scout 1416 found solution d=59
scout 7156 found solution d=45
scout 7820 found solution d=41
scout 11272 found solution d=39
scout 18612 found solution d=37
scout 47223 found solution d=35
scout 64548 found solution d=31
scout 79601 found solution d=25
best d=25 scout=79601 moves=6785412543014367852147634
scout 2387 found solution d=53
scout 6901 found solution d=39
scout 12390 found solution d=29
scout 48525 found solution d=27
best d=27 scout=48525 moves=412541036785210345210345214
scout 1376 found solution d=97
scout 2881 found solution d=73
scout 3472 found solution d=59
scout 3481 found solution d=55
scout 6453 found solution d=39
scout 7857 found solution d=25
best d=25 scout=7857 moves=6745214301254125430147854
scout 2051 found solution d=85
scout 14442 found solution d=79
scout 17723 found solution d=67
scout 18116 found solution d=51
scout 31376 found solution d=47
scout 31525 found solution d=35
scout 51130 found solution d=27
best d=27 scout=51130 moves=452103452103678521430145214
scout 8023 found solution d=59
scout 11107 found solution d=51
scout 19321 found solution d=39
scout 21889 found solution d=35
scout 73478 found solution d=23
best d=23 scout=73478 moves=67452143014521430147854
-----
|167|
|584|
|302|
----7

scout 347 found solution d=47
scout 1322 found solution d=41
scout 29745 found solution d=31
scout 33737 found solution d=27
scout 44673 found solution d=25
best d=25 scout=44673 moves=6341258763412587412543674
scout 7575 found solution d=63
scout 11827 found solution d=33
scout 86507 found solution d=29
best d=29 scout=86507 moves=45214587634125478547634125874
scout 438 found solution d=57
scout 6860 found solution d=27
scout 28315 found solution d=25
best d=25 scout=28315 moves=6341258763412587412543674
scout 224 found solution d=27
scout 62429 found solution d=25
best d=25 scout=62429 moves=6347852145210347630145874
scout 381 found solution d=55
scout 1105 found solution d=39
scout 52940 found solution d=33
scout 59867 found solution d=29
best d=29 scout=59867 moves=63012587410367436741258741254
scout 358 found solution d=63
scout 1233 found solution d=43
scout 14220 found solution d=31
best d=31 scout=14220 moves=4125436785436741034763014785214
scout 126 found solution d=71
scout 4450 found solution d=63
scout 7650 found solution d=43
scout 15377 found solution d=33
scout 73967 found solution d=31
scout 78176 found solution d=29
best d=29 scout=78176 moves=43674125874367410347630145214
scout 397 found solution d=83
scout 2460 found solution d=41
scout 3122 found solution d=27
best d=27 scout=3122 moves=412587634125876345876341254
scout 508 found solution d=59
scout 2092 found solution d=43
scout 9188 found solution d=41
scout 14879 found solution d=27
best d=27 scout=14879 moves=634125876341258763412547634
scout 1631 found solution d=77
scout 3855 found solution d=45
scout 23830 found solution d=37
scout 27292 found solution d=27
best d=27 scout=27292 moves=412587634125876345876341254
```

11 Appendix C: Theory Verifier

This appendix includes the complete source listing of a theory tester designed to experimentally verify equations 1-3 in the text. The program runs *n* tosses of a *k*-sided die and compares the resulting experimental data with the theoretical predictions.

A complete output listing is provided in Appendix D.

```
// Coverage Theory Verification
// Dissertation Coding Example
// J.E. (Ned) Lecky
// 1/19/99

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <math.h>

const int TARGET=1;
const int VISIT=2;

// Generate random number from 0 to max-1
int VRand(int max)
{
    return rand()%max;
}

// Simulate n tosses of a k-sided die
//   - insert nTargets desired targets
//   - store actual coverage and number of target hits
void StateCounter(int k,int n,int nTargets,double* cList,int* nTargetList)
{
    int sizeD=k-nTargets;

    // allocate visit tally vector
    unsigned char* state=new unsigned char[k];
    if(state==NULL)
    {
          printf("Can't allocate state array.\n");
          return;
    }

    // clear visit data
    memset(state,0,k*sizeof(char));

    // pick nTargets random sites as "targets"
    int tSet=0;
    while(tSet<nTargets)
    {
          int r=VRand(k);
          if(state[r]==0)
          {
                state[r]=TARGET;
                tSet++;
          }
    }

    // run the simulation
    int nHits=0;
    int nVisits=0;
    for(int i=0; i<n; i++)
    {
          // store away stats for later analysis
          double Cactual=(double)nVisits/(double)k;
          cList[i]=Cactual;
          nTargetList[i]=nHits;

          // now make the toss
          int r=VRand(k);

          // if we hit a target, increment count
          if(state[r]&TARGET) nHits++;

          // if we have not already hit this state, coverage is up
          if(!(state[r]&VISIT))
          {
                state[r]|=VISIT;
                nVisits++;
          }
    }

    delete state;
}

// Run the complete simulation
int main()
{
    srand(1);

    // all experimental data goes into this file
    FILE* fp=fopen("\\cover.dat","w");

    // Running tests at various k
    for(int test=0; test<6; test++)
    {
          // select k (number of sides on the die)
          int k;
          // select nTargets (number of target sides)
          int nTargets=1;
          switch(test)
          {
          case 0: k=2; break;
          case 1: k=3; break;
          case 2: k=6; nTargets=3; break;
          case 3: k=40; nTargets=2; break;
          case 4: k=1000; nTargets=4; break;
          case 5: k=4000; break;
          }

          // select a reasonable number of tosses known to generate
          //   high coverage
          int n=k*5;

          // number of simulation runs
          int iterCount=200;

          // number of individual tests to print
          int iterPrint=3;
          if(iterPrint>iterCount) iterPrint=iterCount;

          // allocate and clear coverage and target hit arrays
          double** cList=new double*[iterCount];
          int** nTargetList=new int*[iterCount];
          for(int i=0; i<iterCount; i++)
          {
                cList[i]=new double[n];
                nTargetList[i]=new int[n];
                for(int j=0; j<n; j++)
```

```
		{
			cList[i][j]=0.;
			nTargetList[i][j]=0;
		}
	}

	// print simulation column headers
	fprintf(fp,"Theory Testing  k=%d n=%d sims=%d m=%d\n",
		k,n,iterCount,nTargets);
	fprintf(fp,"    i,Ctheo,CtheoA, Cact,nTtheo, nTact,Pdtheo, Pdact");
	for(i=0; i<iterPrint; i++)
		fprintf(fp,",C%dact,T%dact",i,i);
	fprintf(fp,"\n");


	// run iterCount complete simulations
	for(i=0; i<iterCount; i++)
		StateCounter(k,n,nTargets,cList[i],nTargetList[i]);

	// print approximately 25 lines from the simulation data
	int printmod=n/25;
	if(printmod==0) printmod=1;

	// double precision versions
	double kd=k;
	double nTargetsd=nTargets;

	// analysis loop
	for(i=0; i<n; i++)
	{
		// print every so many lines, or on last line
		if(i%printmod==0 || i==n-1)
		{
			double nd=i;
			double cTheo = 1. - pow((1. - 1./kd),nd);
			double cTheoApprox = 1. - exp(-nd/kd);
			double PdTheo = 1. - pow((1. - nTargetsd/kd),nd);
			double nTargetTheo = nd/kd * nTargetsd;

			// average all tests
			double cSum=0.;
			double nTsum=0.;
			double nHit=0.;
			for(int j=0; j<iterCount; j++)
			{
				cSum+=cList[j][i];
				nTsum+=nTargetList[j][i];
				if(nTargetList[j][i]>0) nHit++;
			}
			double I=iterCount;

			// experimental averages
			double Cavg=cSum/I;
			double nTavg=nTsum/I;
			double Pd=nHit/I;

			fprintf(fp,"%5d, %4.02lf, %5.02lf, %4.02lf",
				i,cTheo,cTheoApprox,Cavg,);
			fprintf(fp,", %5.02lf, %5.02lf, %5.02lf, %5.02lf",
				nTargetTheo,nTavg,PdTheo,Pd);
			for(j=0; j<iterPrint; j++)
				fprintf(fp,", %4.02lf, %4d",cList[j][i],nTargetList[j][i]);
			fprintf(fp,"\n");
		}
	}

	// deallocate storage
	for(i=0; i<iterCount; i++)
	{
```

```
            delete cList[i];
            delete nTargetList[i];
        }

        delete nTargetList;
        delete cList;
    }

    // close output file and exit
    fclose(fp);

    return 0;
}
```

12 Appendix D: Sample Run of the Theory Verifier

This appendix provides a complete output listing of the program in Appendix C. The listing consists of several sections headed "Theory Testing." Each section represents 200 simulations of n tosses of a k-sided die, where m sides are considered targets. The variables n, k, and m vary from section to section.

Since a maximum k of 4000 is used in this simulation, the limitation of the C rand() function in generating numbers only from 0 to 32767 is of no concern.

Column headings have the following significance:

i	number of scouts simulated so far
Ctheo	theoretical coverage expected (Equation 1)
CtheoA	approximate coverage expected (Equation 2)
Cact	measured coverage from experimental data
nTtheo	theoretical number of times a target should have been hit
nTact	experimental number of times a target has been hit
Pdtheo	Theoretical probability of discovery (Equation 3)
Pdact	measured probability of discovery
Cjact	raw coverage data from simulation j
Tjact	raw targets hit data from simulation j

The raw simulation run data is only shown for the first three out of the 200 simulations.

As an aside, nTtheo is simply the theoretical number of times that a target should have been hit. This expression is not generally required in scouting analysis, but for completeness is given below. It says that, for example, if one rolls a k-sided die k times, each side should be seen once. The factor of m corrects for the possibility of multiple targets.

$$n_T = \frac{nm}{k}$$

All data agree very closely with the theoretical values. CtheoA is most different from Ctheo for the small problems with $k = 2$ (analogous to a coin-toss) or $k = 3$ as expected. These differences are shown to give an example of the approximate nature of Equation 2.

Most real scouting problems would have k values in the thousands or higher, making the approximation of Equation 2 extremely accurate. Note also that the probabilities of discovery track extremely closely with the theoretical values.

```
Theory Testing  k=2 n=10 sims=200 m=1
    i,Ctheo,CtheoA, Cact,nTtheo, nTact,Pdtheo, Pdact,C0act,T0act,C1act,T1act,C2act,T2act
    0, 0.00,  0.00, 0.00,  0.00,  0.00,  0.00,  0.00, 0.00,    0, 0.00,    0, 0.00,    0
    1, 0.50,  0.39, 0.50,  0.50,  0.46,  0.50,  0.46, 0.50,    1, 0.50,    1, 0.50,    0
    2, 0.75,  0.63, 0.73,  1.00,  0.99,  0.75,  0.72, 1.00,    1, 0.50,    2, 1.00,    1
    3, 0.88,  0.78, 0.86,  1.50,  1.49,  0.88,  0.85, 1.00,    1, 0.50,    3, 1.00,    2
    4, 0.94,  0.86, 0.93,  2.00,  1.95,  0.94,  0.94, 1.00,    2, 0.50,    4, 1.00,    2
    5, 0.97,  0.92, 0.98,  2.50,  2.44,  0.97,  0.98, 1.00,    2, 0.50,    5, 1.00,    3
    6, 0.98,  0.95, 0.99,  3.00,  2.96,  0.98,  0.98, 1.00,    2, 1.00,    5, 1.00,    4
    7, 0.99,  0.97, 0.99,  3.50,  3.44,  0.99,  0.99, 1.00,    2, 1.00,    6, 1.00,    4
    8, 1.00,  0.98, 1.00,  4.00,  3.95,  1.00,  1.00, 1.00,    2, 1.00,    6, 1.00,    4
    9, 1.00,  0.99, 1.00,  4.50,  4.42,  1.00,  1.00, 1.00,    2, 1.00,    7, 1.00,    5
Theory Testing  k=3 n=15 sims=200 m=1
    i,Ctheo,CtheoA, Cact,nTtheo, nTact,Pdtheo, Pdact,C0act,T0act,C1act,T1act,C2act,T2act
    0, 0.00,  0.00, 0.00,  0.00,  0.00,  0.00,  0.00, 0.00,    0, 0.00,    0, 0.00,    0
    1, 0.33,  0.28, 0.33,  0.33,  0.35,  0.33,  0.35, 0.33,    0, 0.33,    0, 0.33,    1
    2, 0.56,  0.49, 0.55,  0.67,  0.64,  0.56,  0.53, 0.33,    0, 0.67,    0, 0.33,    2
    3, 0.70,  0.63, 0.70,  1.00,  1.02,  0.70,  0.71, 0.33,    0, 0.67,    0, 0.67,    2
    4, 0.80,  0.74, 0.81,  1.33,  1.30,  0.80,  0.79, 0.67,    1, 0.67,    0, 0.67,    2
    5, 0.87,  0.81, 0.87,  1.67,  1.62,  0.87,  0.85, 0.67,    2, 0.67,    0, 0.67,    3
    6, 0.91,  0.86, 0.90,  2.00,  1.95,  0.91,  0.89, 1.00,    2, 0.67,    0, 1.00,    3
    7, 0.94,  0.90, 0.95,  2.33,  2.26,  0.94,  0.94, 1.00,    3, 1.00,    1, 1.00,    3
    8, 0.96,  0.93, 0.96,  2.67,  2.56,  0.96,  0.96, 1.00,    3, 1.00,    1, 1.00,    3
    9, 0.97,  0.95, 0.97,  3.00,  2.94,  0.97,  0.97, 1.00,    3, 1.00,    2, 1.00,    3
   10, 0.98,  0.96, 0.98,  3.33,  3.27,  0.98,  0.98, 1.00,    3, 1.00,    2, 1.00,    3
   11, 0.99,  0.97, 0.99,  3.67,  3.59,  0.99,  0.98, 1.00,    3, 1.00,    3, 1.00,    3
   12, 0.99,  0.98, 0.99,  4.00,  3.94,  0.99,  0.99, 1.00,    3, 1.00,    3, 1.00,    3
   13, 0.99,  0.99, 0.99,  4.33,  4.26,  0.99,  1.00, 1.00,    3, 1.00,    4, 1.00,    3
   14, 1.00,  0.99, 1.00,  4.67,  4.62,  1.00,  1.00, 1.00,    3, 1.00,    4, 1.00,    3
Theory Testing  k=6 n=30 sims=200 m=3
    i,Ctheo,CtheoA, Cact,nTtheo, nTact,Pdtheo, Pdact,C0act,T0act,C1act,T1act,C2act,T2act
    0, 0.00,  0.00, 0.00,  0.00,  0.00,  0.00,  0.00, 0.00,    0, 0.00,    0, 0.00,    0
    1, 0.17,  0.15, 0.17,  0.50,  0.48,  0.50,  0.48, 0.17,    1, 0.17,    1, 0.17,    0
    2, 0.31,  0.28, 0.31,  1.00,  0.97,  0.75,  0.72, 0.17,    2, 0.33,    2, 0.33,    1
    3, 0.42,  0.39, 0.42,  1.50,  1.52,  0.88,  0.89, 0.33,    3, 0.50,    2, 0.50,    2
    4, 0.52,  0.49, 0.53,  2.00,  2.02,  0.94,  0.95, 0.50,    3, 0.50,    3, 0.50,    3
    5, 0.60,  0.57, 0.61,  2.50,  2.48,  0.97,  0.98, 0.50,    3, 0.67,    4, 0.50,    4
    6, 0.67,  0.63, 0.67,  3.00,  2.93,  0.98,  0.99, 0.67,    3, 0.67,    5, 0.67,    5
    7, 0.72,  0.69, 0.73,  3.50,  3.44,  0.99,  1.00, 0.83,    3, 0.67,    6, 0.67,    5
    8, 0.77,  0.74, 0.77,  4.00,  3.97,  1.00,  1.00, 1.00,    4, 0.67,    7, 0.67,    5
    9, 0.81,  0.78, 0.81,  4.50,  4.49,  1.00,  1.00, 1.00,    5, 0.67,    8, 0.67,    6
   10, 0.84,  0.81, 0.84,  5.00,  4.98,  1.00,  1.00, 1.00,    6, 0.67,    9, 0.83,    6
   11, 0.87,  0.84, 0.87,  5.50,  5.50,  1.00,  1.00, 1.00,    7, 0.67,    9, 0.83,    6
   12, 0.89,  0.86, 0.89,  6.00,  6.05,  1.00,  1.00, 1.00,    7, 0.83,    9, 0.83,    6
   13, 0.91,  0.89, 0.91,  6.50,  6.52,  1.00,  1.00, 1.00,    7, 0.83,   10, 0.83,    7
   14, 0.92,  0.90, 0.92,  7.00,  7.04,  1.00,  1.00, 1.00,    8, 0.83,   11, 0.83,    8
   15, 0.94,  0.92, 0.93,  7.50,  7.55,  1.00,  1.00, 1.00,    9, 0.83,   12, 1.00,    8
   16, 0.95,  0.93, 0.94,  8.00,  8.07,  1.00,  1.00, 1.00,    9, 0.83,   12, 1.00,    9
   17, 0.95,  0.94, 0.95,  8.50,  8.57,  1.00,  1.00, 1.00,   10, 1.00,   12, 1.00,   10
   18, 0.96,  0.95, 0.96,  9.00,  9.10,  1.00,  1.00, 1.00,   10, 1.00,   12, 1.00,   11
   19, 0.97,  0.96, 0.97,  9.50,  9.58,  1.00,  1.00, 1.00,   10, 1.00,   13, 1.00,   11
   20, 0.97,  0.96, 0.97, 10.00, 10.08,  1.00,  1.00, 1.00,   10, 1.00,   13, 1.00,   12
   21, 0.98,  0.97, 0.98, 10.50, 10.59,  1.00,  1.00, 1.00,   11, 1.00,   14, 1.00,   13
   22, 0.98,  0.97, 0.98, 11.00, 11.06,  1.00,  1.00, 1.00,   11, 1.00,   15, 1.00,   13
   23, 0.98,  0.98, 0.98, 11.50, 11.59,  1.00,  1.00, 1.00,   11, 1.00,   16, 1.00,   14
   24, 0.99,  0.98, 0.99, 12.00, 12.10,  1.00,  1.00, 1.00,   12, 1.00,   17, 1.00,   14
   25, 0.99,  0.98, 0.99, 12.50, 12.66,  1.00,  1.00, 1.00,   13, 1.00,   18, 1.00,   15
   26, 0.99,  0.99, 0.99, 13.00, 13.18,  1.00,  1.00, 1.00,   13, 1.00,   18, 1.00,   15
   27, 0.99,  0.99, 0.99, 13.50, 13.71,  1.00,  1.00, 1.00,   13, 1.00,   18, 1.00,   16
   28, 0.99,  0.99, 0.99, 14.00, 14.20,  1.00,  1.00, 1.00,   13, 1.00,   18, 1.00,   16
   29, 0.99,  0.99, 0.99, 14.50, 14.73,  1.00,  1.00, 1.00,   14, 1.00,   18, 1.00,   17
Theory Testing  k=40 n=200 sims=200 m=2
    i,Ctheo,CtheoA, Cact,nTtheo, nTact,Pdtheo, Pdact,C0act,T0act,C1act,T1act,C2act,T2act
    0, 0.00,  0.00, 0.00,  0.00,  0.00,  0.00,  0.00, 0.00,    0, 0.00,    0, 0.00,    0
    8, 0.18,  0.18, 0.19,  0.40,  0.40,  0.34,  0.33, 0.20,    0, 0.15,    2, 0.15,    0
   16, 0.33,  0.33, 0.34,  0.80,  0.77,  0.56,  0.52, 0.33,    0, 0.35,    2, 0.33,    0
   24, 0.46,  0.45, 0.46,  1.20,  1.10,  0.71,  0.69, 0.42,    0, 0.45,    2, 0.42,    1
   32, 0.56,  0.55, 0.55,  1.60,  1.49,  0.81,  0.77, 0.53,    1, 0.50,    3, 0.53,    2
   40, 0.64,  0.63, 0.63,  2.00,  1.85,  0.87,  0.83, 0.57,    1, 0.63,    4, 0.63,    3
   48, 0.70,  0.70, 0.70,  2.40,  2.29,  0.91,  0.91, 0.60,    1, 0.72,    5, 0.72,    3
   56, 0.76,  0.75, 0.76,  2.80,  2.79,  0.94,  0.95, 0.68,    1, 0.82,    6, 0.80,    4
   64, 0.80,  0.80, 0.80,  3.20,  3.17,  0.96,  0.96, 0.78,    1, 0.85,    6, 0.80,    5
   72, 0.84,  0.83, 0.84,  3.60,  3.62,  0.98,  0.97, 0.85,    1, 0.90,    7, 0.80,    5
   80, 0.87,  0.86, 0.87,  4.00,  4.01,  0.98,  0.99, 0.85,    2, 0.93,    7, 0.80,    6
   88, 0.89,  0.89, 0.89,  4.40,  4.46,  0.99,  1.00, 0.90,    3, 0.95,    7, 0.88,    6
   96, 0.91,  0.91, 0.91,  4.80,  4.84,  0.99,  1.00, 0.90,    4, 0.95,    7, 0.90,    7
  104, 0.93,  0.93, 0.93,  5.20,  5.32,  1.00,  1.00, 0.93,    4, 0.95,    9, 0.93,    8
  112, 0.94,  0.94, 0.94,  5.60,  5.68,  1.00,  1.00, 0.95,    4, 0.95,   11, 0.95,    8
  120, 0.95,  0.95, 0.95,  6.00,  6.19,  1.00,  1.00, 0.95,    4, 0.95,   11, 0.95,    8
  128, 0.96,  0.96, 0.96,  6.40,  6.60,  1.00,  1.00, 0.97,    4, 0.95,   11, 0.97,    9
  136, 0.97,  0.97, 0.97,  6.80,  6.98,  1.00,  1.00, 0.97,    4, 0.95,   11, 0.97,    9
  144, 0.97,  0.97, 0.98,  7.20,  7.28,  1.00,  1.00, 1.00,    4, 0.95,   12, 0.97,    9
  152, 0.98,  0.98, 0.98,  7.60,  7.62,  1.00,  1.00, 1.00,    5, 0.95,   12, 0.97,   10
  160, 0.98,  0.98, 0.98,  8.00,  8.05,  1.00,  1.00, 1.00,    5, 0.95,   12, 0.97,   11
  168, 0.99,  0.99, 0.99,  8.40,  8.46,  1.00,  1.00, 1.00,    5, 0.97,   12, 0.97,   12
  176, 0.99,  0.99, 0.99,  8.80,  8.89,  1.00,  1.00, 1.00,    5, 1.00,   12, 0.97,   12
  184, 0.99,  0.99, 0.99,  9.20,  9.25,  1.00,  1.00, 1.00,    5, 1.00,   13, 0.97,   12
  192, 0.99,  0.99, 0.99,  9.60,  9.68,  1.00,  1.00, 1.00,    5, 1.00,   13, 0.97,   12
  199, 0.99,  0.99, 0.99,  9.95, 10.00,  1.00,  1.00, 1.00,    5, 1.00,   13, 0.97,   12
Theory Testing  k=1000 n=5000 sims=200 m=4
    i,Ctheo,CtheoA, Cact,nTtheo, nTact,Pdtheo, Pdact,C0act,T0act,C1act,T1act,C2act,T2act
    0, 0.00,  0.00, 0.00,  0.00,  0.00,  0.00,  0.00, 0.00,    0, 0.00,    0, 0.00,    0
  200, 0.18,  0.18, 0.18,  0.80,  0.76,  0.55,  0.54, 0.18,    0, 0.18,    0, 0.18,    0
  400, 0.33,  0.33, 0.33,  1.60,  1.55,  0.80,  0.76, 0.33,    0, 0.32,    1, 0.33,    3
  600, 0.45,  0.45, 0.45,  2.40,  2.40,  0.91,  0.92, 0.45,    0, 0.44,    3, 0.45,    3
  800, 0.55,  0.55, 0.55,  3.20,  3.15,  0.96,  0.96, 0.56,    1, 0.55,    3, 0.56,    4
 1000, 0.63,  0.63, 0.63,  4.00,  4.02,  0.98,  0.98, 0.63,    2, 0.63,    5, 0.65,    5
 1200, 0.70,  0.70, 0.70,  4.80,  4.92,  0.99,  1.00, 0.69,    4, 0.69,    6, 0.72,    6
 1400, 0.75,  0.75, 0.75,  5.60,  5.75,  1.00,  1.00, 0.75,    4, 0.76,    7, 0.76,    6
 1600, 0.80,  0.80, 0.80,  6.40,  6.63,  1.00,  1.00, 0.79,    4, 0.80,    8, 0.81,    8
 1800, 0.83,  0.83, 0.83,  7.20,  7.50,  1.00,  1.00, 0.84,    6, 0.83,    9, 0.84,    9
 2000, 0.86,  0.86, 0.86,  8.00,  8.14,  1.00,  1.00, 0.87,    7, 0.87,   10, 0.86,    9
 2200, 0.89,  0.89, 0.89,  8.80,  8.97,  1.00,  1.00, 0.89,   10, 0.90,   10, 0.89,    9
 2400, 0.91,  0.91, 0.91,  9.60,  9.81,  1.00,  1.00, 0.91,   11, 0.92,   12, 0.91,   10
 2600, 0.93,  0.93, 0.93, 10.40, 10.71,  1.00,  1.00, 0.92,   11, 0.94,   13, 0.93,   11
 2800, 0.94,  0.94, 0.94, 11.20, 11.45,  1.00,  1.00, 0.93,   11, 0.95,   14, 0.94,   11
 3000, 0.95,  0.95, 0.95, 12.00, 12.20,  1.00,  1.00, 0.95,   13, 0.96,   15, 0.95,   11
 3200, 0.96,  0.96, 0.96, 12.80, 13.07,  1.00,  1.00, 0.96,   14, 0.96,   17, 0.96,   11
 3400, 0.97,  0.97, 0.97, 13.60, 13.79,  1.00,  1.00, 0.96,   17, 0.97,   18, 0.97,   12
 3600, 0.97,  0.97, 0.97, 14.40, 14.58,  1.00,  1.00, 0.98,   17, 0.98,   18, 0.97,   12
 3800, 0.98,  0.98, 0.98, 15.20, 15.42,  1.00,  1.00, 0.98,   18, 0.98,   19, 0.98,   12
 4000, 0.98,  0.98, 0.98, 16.00, 16.19,  1.00,  1.00, 0.98,   19, 0.98,   21, 0.98,   12
 4200, 0.99,  0.99, 0.99, 16.80, 16.99,  1.00,  1.00, 0.99,   19, 0.99,   21, 0.99,   14
 4400, 0.99,  0.99, 0.99, 17.60, 17.81,  1.00,  1.00, 0.99,   20, 0.99,   21, 0.99,   15
 4600, 0.99,  0.99, 0.99, 18.40, 18.64,  1.00,  1.00, 0.99,   21, 0.99,   21, 0.99,   15
 4800, 0.99,  0.99, 0.99, 19.20, 19.43,  1.00,  1.00, 1.00,   22, 0.99,   21, 0.99,   16
 4999, 0.99,  0.99, 0.99, 20.00, 20.20,  1.00,  1.00, 1.00,   24, 1.00,   22, 0.99,   16
Theory Testing  k=4000 n=20000 sims=200 m=1
    i,Ctheo,CtheoA, Cact,nTtheo, nTact,Pdtheo, Pdact,C0act,T0act,C1act,T1act,C2act,T2act
    0, 0.00,  0.00, 0.00,  0.00,  0.00,  0.00,  0.00, 0.00,    0, 0.00,    0, 0.00,    0
  800, 0.18,  0.18, 0.18,  0.20,  0.22,  0.18,  0.20, 0.18,    0, 0.18,    0, 0.18,    0
 1600, 0.33,  0.33, 0.33,  0.40,  0.44,  0.33,  0.39, 0.32,    0, 0.33,    0, 0.33,    0
 2400, 0.45,  0.45, 0.45,  0.60,  0.73,  0.45,  0.53, 0.45,    0, 0.45,    0, 0.45,    0
 3200, 0.55,  0.55, 0.55,  0.80,  0.97,  0.55,  0.62, 0.55,    0, 0.55,    0, 0.55,    0
 4000, 0.63,  0.63, 0.63,  1.00,  1.16,  0.63,  0.69, 0.64,    0, 0.63,    0, 0.64,    0
 4800, 0.70,  0.70, 0.70,  1.20,  1.34,  0.70,  0.76, 0.71,    1, 0.70,    0, 0.70,    0
 5600, 0.75,  0.75, 0.75,  1.40,  1.52,  0.75,  0.80, 0.76,    1, 0.75,    0, 0.76,    1
 6400, 0.80,  0.80, 0.80,  1.60,  1.76,  0.80,  0.83, 0.81,    1, 0.79,    0, 0.80,    1
 7200, 0.83,  0.83, 0.83,  1.80,  1.92,  0.83,  0.86, 0.84,    1, 0.83,    0, 0.84,    2
 8000, 0.86,  0.86, 0.86,  2.00,  2.07,  0.86,  0.91, 0.87,    1, 0.87,    0, 0.87,    2
 8800, 0.89,  0.89, 0.89,  2.20,  2.23,  0.89,  0.91, 0.90,    1, 0.89,    0, 0.90,    2
 9600, 0.91,  0.91, 0.91,  2.40,  2.44,  0.91,  0.92, 0.92,    1, 0.91,    0, 0.91,    2
10400, 0.93,  0.93, 0.93,  2.60,  2.66,  0.93,  0.94, 0.93,    1, 0.92,    0, 0.93,    2
11200, 0.94,  0.94, 0.94,  2.80,  2.85,  0.94,  0.96, 0.94,    2, 0.94,    0, 0.94,    2
12000, 0.95,  0.95, 0.95,  3.00,  3.06,  0.95,  0.97, 0.95,    2, 0.95,    1, 0.95,    2
12800, 0.96,  0.96, 0.96,  3.20,  3.27,  0.96,  0.98, 0.96,    2, 0.96,    1, 0.96,    3
13600, 0.97,  0.97, 0.97,  3.40,  3.50,  0.97,  0.98, 0.97,    3, 0.97,    1, 0.97,    3
14400, 0.97,  0.97, 0.97,  3.60,  3.71,  0.97,  0.98, 0.97,    3, 0.97,    1, 0.97,    4
15200, 0.98,  0.98, 0.98,  3.80,  3.97,  0.98,  0.98, 0.98,    3, 0.98,    1, 0.98,    5
16000, 0.98,  0.98, 0.98,  4.00,  4.11,  0.98,  0.99, 0.98,    3, 0.98,    1, 0.98,    5
16800, 0.99,  0.99, 0.98,  4.20,  4.34,  0.99,  1.00, 0.98,    3, 0.98,    1, 0.98,    5
17600, 0.99,  0.99, 0.99,  4.40,  4.59,  0.99,  1.00, 0.99,    3, 0.99,    2, 0.99,    7
18400, 0.99,  0.99, 0.99,  4.60,  4.77,  0.99,  1.00, 0.99,    3, 0.99,    2, 0.99,    7
19200, 0.99,  0.99, 0.99,  4.80,  4.92,  0.99,  1.00, 0.99,    3, 0.99,    5, 0.99,    7
19999, 0.99,  0.99, 0.99,  5.00,  5.17,  0.99,  1.00, 0.99,    3, 0.99,    6, 0.99,    7
```

13 Appendix E: Solution Likelihood Verifier

The sample program below is a replacement for the main( ) routine of the program in Appendix A. This version generates a database of puzzles with solutions from 18 to 20 moves by randomly scrambling the goal state and then running 200,000 scout-1s to determine the near-optimal solution. Only puzzles with solutions between 18- and 20-moves are kept. The puzzles are written into a database for subsequent use.

Additional runs of the program perform probability testing to determine the likelihood of finding the optimal solution of each puzzle. This structure allows various testing methodologies to be tried with the exact same set of test puzzles every time.

```
// Scout Cluster Simulation
// Variation to verify equation 4
// 8-puzzle
// Dissertation Example
// 1/20/99  John E. Lecky

// ALL OTHER CODE SAME AS APPENDIX A EXCEPT MAIN ROUTINE BELOW

// Run the complete simulation
int main()
{
    // select the scout-1 PA block
    srand(1);
    PA=&PA_Scout1;

    int s[10];

    // automatically generate the puzzles if the puzzles don't
    //   yet exist
    FILE* fp=fopen("pdata.dat","r");
    if(fp==NULL)
    {
          printf("Generating test puzzle database.\n");

          fp=fopen("pdata.dat","w");
          int nPuzzles=10;
          int dSum=0;
          int dMin=9999;
          int dMax=0;
          fprintf(fp,"%d\n",nPuzzles);
          for(int i=0; i<nPuzzles; i++)
          {
                // create the scrambled test puzzle
retry:          memcpy(s,Sg,10*sizeof(int));
                ScramblePuzzle(s,16 + VRand(10));
                printf("Looking for database puzzle %d\n",i+1);
                ShowSituation(s,stdout);

                // solve it with 200,000 scouts-1s.  This should almost
                //   always find the optimal solution
                int d=ScoutCluster(s,200000,100);
                printf("solution d=%d\n",d);

                // restrict database to puzzles with solutions 18-20 moves
                if(d<18 || d>20) goto retry;

                // save puzzle and solution length to database
                for(int j=0; j<10; j++)
                      fprintf(fp,"%d ",s[j]);
                fprintf(fp,"%d\n",d);

                // maintain overall puzzle statistics
                dSum+=d;
                if(d<dMin) dMin=d;
                if(d>dMax) dMax=d;
          }

          // add the stats to the puzzle database file
          double dAvg=(double)dSum/(double)nPuzzles;
          printf("dAvg=%.1lf dMin=%d dMax=%d\n",dAvg,dMin,dMax);
          fprintf(fp,"%.1lf %d %d\n",dAvg,dMin,dMax);
          fclose(fp);
    }

    // read in puzzle database
    fp=fopen("pdata.dat","r");
    if(fp==NULL)
    {
          printf("ERROR: cannot find puzzle database.\n");
          return(1);
    }

    int nPuzzles;
    double dAvg;
    int dMin,dMax;

    int testS[100][10];
    int solution[100];

    // scan in actual file fields
    if(1!=fscanf(fp,"%d",&nPuzzles))
    {
error:
          fclose(fp);
          printf("FILE ERROR\n");
          return(1);
    }
    for(int i=0; i<nPuzzles; i++)
    {
          for(int j=0; j<10; j++)
                if(1!=fscanf(fp,"%d",&testS[i][j])) goto error;
          if(1!=fscanf(fp,"%d",&solution[i])) goto error;
    }
    if(3!=fscanf(fp,"%lf %d %d",&dAvg,&dMin,&dMax)) goto error;
    fclose(fp);

    // print simulation header
    printf("Puzzle Database: nPuzzles=%d dAvg=%.1lf dMin=%d dMax=%d\n",
          nPuzzles,dAvg,dMin,dMax);

    fp=fopen("pdtest.dat","w");

    // problem assumptions
    PA=&PA_Scout1;  // use scout-1 (b=1.67)
    double b=1.67;  // estimate branching factor
    double d=dAvg;  // estimated solution length
    double m=1.0;   // number of optimal solutions/puzzle; usually only one

    // Run simulations at various desired Pd
    int incr=10;
    for(int Pdtarget=10; Pdtarget<=99; Pdtarget+=incr)
    {
          // adjust incr
          if(Pdtarget==90) incr=5;
          if(Pdtarget==95) incr=2;

          double Pd=Pdtarget/100.;

          // compute theoretical n for this Pd
          double nd = -pow(b,d)/m * log(1. - Pd);
          int n = (int)nd;

          fprintf(fp,"PdTarget=%.02lf n=%d\n",Pd,n);
          printf("PdTarget=%.02lf n=%d\n",Pd,n);

          // run for all puzzles
          int nTotalBest=0;
          int nTotalPuzzles=0;
          for(int p=0; p<nPuzzles; p++)
          {
                fprintf(fp," p%d s=%d",p+1,solution[p]);
                printf("%d ",p+1);

                // solve each puzzle repeatedly with n scouts
                int count=20;
                int track[500];
                for(int it=0; it<count; it++)
                {
                      int d=ScoutCluster(testS[p],(int)n,100);
                      nTotalPuzzles++;
                      track[it]=d;
                }

                // analyze how many times the optimal solution was found
                int nBest=0;
                for(it=0; it<count; it++)
                      if(track[it]>0 && track[it]<=solution[p])
                      {
                            if(track[it]<solution[p])
                                  printf("b%d=%d\n",p+1,track[it]);
                            nBest++;
                            nTotalBest++;
                      }

                // compute Pd for this single puzzle and display
                double Pdact=(double)nBest/(double)count;
                fprintf(fp," %d/%d Pd=%.2lf\n",nBest,count,Pdact);
          }

          // compute overal Pd for entire set of puzzles at given n
          double Pdavg=(double)nTotalBest/(double)(nTotalPuzzles);
          fprintf(fp," Pdavg=%.2lf Pdest=%.2lf\n",Pdavg,Pd);
          printf(" Pdact %d/%d=%.2lf Pdest=%.2lf\n",
                nTotalBest,nTotalPuzzles,Pdavg,Pd);
          fflush(fp);
    }

    fclose(fp);
    return 0;
}
```

14 Appendix F: Sample Run of the Solution Likelihood Verifier

The output generated by the program in Appendix E is presented here. This output consists of two parts.

The initial run of the program generates a database of 10 random 8-puzzles which are restricted to have optimal solutions between 18- and 20-moves. Subsequent runs of the program use this puzzle database as test input to measure the actual probability of discovery of the optimal solution for different values of expected probability.

14.1 The Test Puzzles

The following data file was generated by the program listed in Appendix E. This represents 10 8-puzzles with solutions ranging from 18 to 20 moves. The average solution depth is 18.7 moves.

The 10 data lines show the initial contents of the nine 8-puzzle grid squares in row-column order:

```
0 1 2
3 4 5
6 7 8
```

The tenth item is a redundant listing of the square where the blank tile is initially located, and the final entry is the best solution found during a 200,000 scout-1 run.

```
10
6 7 0 8 1 3 2 5 4 2 18
0 1 2 5 4 6 3 8 7 0 18
0 6 1 7 5 2 8 4 3 0 18
0 8 1 5 7 3 2 4 6 0 18
8 5 1 4 0 3 6 2 7 4 18
1 4 0 8 6 2 5 3 7 2 18
6 4 3 8 1 5 2 7 0 8 20
8 2 4 0 3 5 7 6 1 3 19
0 2 3 7 1 5 8 4 6 0 20
6 7 8 3 0 2 4 1 5 4 20
18.7 18 20
```

14.2 Solution Data

After creating the database above, the program in Appendix E estimates the *n* required to achieve various probabilities of discovery and then tests each puzzle in the database with that number of scouts. The output file below is generated.

```
PdTarget=0.10 n=1539
 p1 s=18 2/20 Pd=0.10
 p2 s=18 2/20 Pd=0.10
 p3 s=18 3/20 Pd=0.15
 p4 s=18 7/20 Pd=0.35
 p5 s=18 0/20 Pd=0.00
 p6 s=18 2/20 Pd=0.10
 p7 s=20 2/20 Pd=0.10
 p8 s=19 3/20 Pd=0.15
 p9 s=20 2/20 Pd=0.10
 p10 s=20 1/20 Pd=0.05
 Pdavg=0.12 Pdest=0.10
PdTarget=0.20 n=3261
 p1 s=18 4/20 Pd=0.20
 p2 s=18 5/20 Pd=0.25
 p3 s=18 6/20 Pd=0.30
 p4 s=18 14/20 Pd=0.70
 p5 s=18 0/20 Pd=0.00
 p6 s=18 6/20 Pd=0.30
 p7 s=20 1/20 Pd=0.05
 p8 s=19 2/20 Pd=0.10
 p9 s=20 2/20 Pd=0.10
 p10 s=20 1/20 Pd=0.05
 Pdavg=0.20 Pdest=0.20
PdTarget=0.30 n=5212
 p1 s=18 6/20 Pd=0.30
 p2 s=18 4/20 Pd=0.20
 p3 s=18 11/20 Pd=0.55
 p4 s=18 10/20 Pd=0.50
 p5 s=18 3/20 Pd=0.15
 p6 s=18 7/20 Pd=0.35
 p7 s=20 2/20 Pd=0.10
 p8 s=19 7/20 Pd=0.35
 p9 s=20 4/20 Pd=0.20
 p10 s=20 5/20 Pd=0.25
 Pdavg=0.29 Pdest=0.30
PdTarget=0.40 n=7465
 p1 s=18 9/20 Pd=0.45
 p2 s=18 10/20 Pd=0.50
 p3 s=18 10/20 Pd=0.50
 p4 s=18 13/20 Pd=0.65
 p5 s=18 3/20 Pd=0.15
 p6 s=18 6/20 Pd=0.30
 p7 s=20 7/20 Pd=0.35
 p8 s=19 11/20 Pd=0.55
 p9 s=20 9/20 Pd=0.45
 p10 s=20 4/20 Pd=0.20
 Pdavg=0.41 Pdest=0.40
PdTarget=0.50 n=10130
 p1 s=18 14/20 Pd=0.70
 p2 s=18 4/20 Pd=0.20
 p3 s=18 11/20 Pd=0.55
 p4 s=18 17/20 Pd=0.85
 p5 s=18 5/20 Pd=0.25
 p6 s=18 7/20 Pd=0.35
 p7 s=20 5/20 Pd=0.25
 p8 s=19 10/20 Pd=0.50
 p9 s=20 11/20 Pd=0.55
 p10 s=20 6/20 Pd=0.30
 Pdavg=0.45 Pdest=0.50
PdTarget=0.60 n=13391
 p1 s=18 13/20 Pd=0.65
 p2 s=18 8/20 Pd=0.40
 p3 s=18 13/20 Pd=0.65
 p4 s=18 19/20 Pd=0.95
 p5 s=18 3/20 Pd=0.15
 p6 s=18 8/20 Pd=0.40
 p7 s=20 10/20 Pd=0.50
 p8 s=19 14/20 Pd=0.70
 p9 s=20 13/20 Pd=0.65
 p10 s=20 5/20 Pd=0.25
 Pdavg=0.53 Pdest=0.60
PdTarget=0.70 n=17596
 p1 s=18 19/20 Pd=0.95
 p2 s=18 11/20 Pd=0.55
 p3 s=18 16/20 Pd=0.80
 p4 s=18 19/20 Pd=0.95
 p5 s=18 4/20 Pd=0.20
 p6 s=18 14/20 Pd=0.70
 p7 s=20 4/20 Pd=0.20
 p8 s=19 17/20 Pd=0.85
 p9 s=20 15/20 Pd=0.75
 p10 s=20 3/20 Pd=0.15
 Pdavg=0.61 Pdest=0.70
PdTarget=0.80 n=23521
 p1 s=18 17/20 Pd=0.85
 p2 s=18 15/20 Pd=0.75
 p3 s=18 19/20 Pd=0.95
 p4 s=18 20/20 Pd=1.00
 p5 s=18 7/20 Pd=0.35
 p6 s=18 17/20 Pd=0.85
 p7 s=20 7/20 Pd=0.35
 p8 s=19 19/20 Pd=0.95
 p9 s=20 16/20 Pd=0.80
 p10 s=20 8/20 Pd=0.40
 Pdavg=0.72 Pdest=0.80
PdTarget=0.90 n=33652
 p1 s=18 19/20 Pd=0.95
 p2 s=18 17/20 Pd=0.85
 p3 s=18 20/20 Pd=1.00
 p4 s=18 20/20 Pd=1.00
 p5 s=18 13/20 Pd=0.65
 p6 s=18 19/20 Pd=0.95
 p7 s=20 13/20 Pd=0.65
 p8 s=19 19/20 Pd=0.95
 p9 s=20 20/20 Pd=1.00
 p10 s=20 14/20 Pd=0.70
 Pdavg=0.87 Pdest=0.90
PdTarget=0.95 n=43782
 p1 s=18 20/20 Pd=1.00
 p2 s=18 14/20 Pd=0.70
 p3 s=18 19/20 Pd=0.95
 p4 s=18 20/20 Pd=1.00
 p5 s=18 12/20 Pd=0.60
 p6 s=18 20/20 Pd=1.00
 p7 s=20 16/20 Pd=0.80
 p8 s=19 19/20 Pd=0.95
 p9 s=20 19/20 Pd=0.95
 p10 s=20 11/20 Pd=0.55
 Pdavg=0.85 Pdest=0.95
PdTarget=0.97 n=51248
 p1 s=18 20/20 Pd=1.00
 p2 s=18 15/20 Pd=0.75
 p3 s=18 20/20 Pd=1.00
 p4 s=18 20/20 Pd=1.00
 p5 s=18 11/20 Pd=0.55
 p6 s=18 19/20 Pd=0.95
 p7 s=20 18/20 Pd=0.90
 p8 s=19 20/20 Pd=1.00
 p9 s=20 20/20 Pd=1.00
 p10 s=20 11/20 Pd=0.55
 Pdavg=0.87 Pdest=0.97
PdTarget=0.99 n=67304
 p1 s=18 20/20 Pd=1.00
 p2 s=18 17/20 Pd=0.85
 p3 s=18 20/20 Pd=1.00
 p4 s=18 20/20 Pd=1.00
 p5 s=18 15/20 Pd=0.75
 p6 s=18 20/20 Pd=1.00
 p7 s=20 19/20 Pd=0.95
 p8 s=19 20/20 Pd=1.00
 p9 s=20 20/20 Pd=1.00
 p10 s=20 18/20 Pd=0.90
 Pdavg=0.94 Pdest=0.99
```

Achieving a particular P_d requires an estimate of the average solution depth and how many optimal solutions exist. Most 8-puzzles have just a single optimal solution, so $m = 1$.

As an example, to achieve $P_d = 0.1$ we would have:

$$\begin{aligned} m &= 1.0 \\ b &= 1.67 \\ d &= 18.7 \end{aligned}$$

By Equation 4, we then have:

$$n \cong \frac{-1.67^{18.7}}{1} \ln(1-0.1) = 1539.8$$

These scout counts are simply truncated to their integer value without introduction of any significant error.

The output file should be interpreted as follows. Starting at line one, the simulation begins by attempting to achieve a P_d of 0.10. As seen in the example above, this implies that 1539 scouts will be required.

The next line in the file specifies that the first puzzle in the database has a solution of length 18, and this solution was found in 2 out of 20 attempts with the 1539 scouts. This gives a $P_d = 0.10$ exactly as anticipated for this one puzzle. The other nine puzzles follow in similar fashion.

In this case, the best puzzle is solved 35% of the time, while the worst one is never solved.

The average P_d is reported on the last line of each section, as well as a repetition of the original target P_d.