On
The Use of Randomness in Computing To Perform Intelligent Tasks

by

Ryan Scott Regensburger

A thesis submitted to the
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
December 2004

We the undersigned committee
hereby approve the attached thesis

On
The Use of Randomness in Computing
To Perform Intelligent Tasks

by

Ryan Scott Regensburger

Gregory Harrison, Ph.D.
Adjunct Professor
Computer Science
(Principal Advisor)

Richard James, Ph.D.
Adjunct Professor
Computer Science

David E. Clapp, Ph.D.
Associate Professor
Management

William D. Shoaff, Ph.D.
Associate Professor
Program Chair
Computer Science

Abstract

On

The Use of Randomness in Computing

To Perform Intelligent Tasks

By

Ryan Scott Regensburger

Principal Advisor: Dr. Gregory Harrison

The study of Artificial Intelligence attempts to simulate the processes of human intelligence in a set of computable algorithms. The purpose of Random Algorithms in this field is to provide a best-guess approach at identifying the unknown. In this thesis, research shows that random algorithms are able to break down many intelligent processes into a set of solvable problems. For example, solving puzzles and playing games involve the same estimating ability shown in standard problems such as the Coupon Collector problem or the Monty Hall problem. This thesis shows Random Algorithmic applications in two overlapping categories of intelligent behavior: Pattern Recognition (to solve puzzles) and Mind Simulation (to play games). The first category focuses on one of the prominent intelligent processes, recognizing patterns from randomness, which the human mind must continually and dynamically perform. The second category deals with simulating the processes of making decisions and solving problems in a more abstract and uncontrolled way, much like the unpredictable human mind.

Table of Contents

List of Figures

List of Tables

List of Symbols

log:	logarithm base 2
a mod b:	remainder in the division of a by b
O:	big-oh notation - asymptotic upper bound
Ω:	big-omega notation - asymptotic lower bound
π:	pi - the ratio of the circumference to the diameter of a circle

Acknowledgement

To my thesis advisor, Dr. Gregory Harrison, for the support and guidance throughout the development of this Thesis.

To my professors at F.I.T. for showing me the next level of computer science. Thank you Mr. Findling, Dr. Ludwig, and Mr. Slone.

To Lockheed Martin for financial and professional support.

To Dr. David Clapp, for providing firm and truthful guidance in the will to take on a Master's degree.

Dedication

This thesis is dedicated to my wife, Teresa Regensburger, and our families, The Regensburger's, The Johnson's, and The Schumann's. For, without them, the path to enlightenment and finding my own goals and dreams would not have been such a joyous one. It is ultimately from them that I have learned the true unique and random qualities about the world and the chaotic, yet humbling nature of love.

Chapter 1: Introduction and Background

1.1 Universal Purpose

Everything is random. This statement is an oxymoron in that, if the dictionary definition of random were "everything", then the true meaning and nature of the word would cease to hold any credibility. However, this paradox is ultimately true and was first revealed by Claude Shannon with his Theory of Information. The Merriam-Webster dictionary states that the meaning of random is "lack of a definite plan, purpose, or pattern", "haphazard", and/or "without aim, direction, rule, or method." Yet, when human beings perceive in the world around us, we find much purpose, planning, and patterns. How can everything be random?

In the purposes and patterns we find in the universe, there also exists uniqueness. Classical uniqueness is revealed in snowflakes, fingerprints, and DNA. These are natural occurring phenomena that 'do not occur the same way twice.' If we look further, we can find uniqueness in many other places, and ultimately, *all* other places. For example, humans intelligently define a 'tree' pattern to classify all species of tree. The basic shape and abstract qualities are outlined so humans can recognize a tree when they see one. However, no two trees are ever alike. No two trees have the exact same features because there are an infinite amount of naturally unpredictable events that determine their existence. This idea also applies to inanimate objects. A machine that molds Yo-Yos from

plastic into the exact same form each and every time still cannot create two Yo-Yos that are, in essence, equivalent.

The investigation of this phenomenon is not difficult to understand. By simply stating that there are *two* of something in this universe automatically means that they cannot be physically equivalent. Even if two objects were structurally built with the exact same atomic structure, the fact that there are *two* distinct objects means that they are different. Microscopically, different particles are used in an object's construction (and even swapped out for replacements) and macroscopically, one object may have more dust on it than the other, thus making the two objects different.

The meaning of all this uniqueness in the universe is that everything is random. The human brain perceives a universe with no redundancy. Redundancy leads to boredom, such as may happen when repeatedly watching the same television show, or having the same daily process of getting ready for work. Everything we see, smell, taste, touch, and hear is random and has meaning. Everything we perceive is pure information. The remarkable computing power of the human brain is responsible for identifying and recognizing patterns from the randomness, and reasoning with the uncertainty.

Most computer hardware and software has been developed to work in a world of determinism and control. Automated universal machines perform certain

processes efficiently, according to strict protocol, every single time they are summoned. This thesis shows how software can utilize randomness for coping with a random world. It shows classes of problems that random algorithms can solve efficiently, optimally, and approximately, by focusing mainly on pattern recognition and the simulation of intelligent processes. Random algorithmic techniques are demonstrated to provide insight into the mysteriousness of the ultimate computing machine, the human mind.

1.2 Randomness Defined

The behavior of randomized algorithms is based on the dynamic generation of numerical values that drive decisions made by the algorithm. A single number, or a sequence of numbers is difficult to classify as random. An intuition about an arbitrary number or sequence of numbers being random holds no credibility because an unknown rule can negate the assumption. For example, given the sequence 1100110, it would seem that the sequence is random since there are no easily discernable patterns. Given the additional digits of the sequence, 01100, a pattern begins to develop. The repeated pattern of 1100 emerges from the additional information, making the initial sequence no longer uncertain.

Prior to knowing the generation rules of the sequence, it contained much uncertainty. If a sequence can be deterministically generated to produce a pattern, it is no longer uncertain. Future iterations can be easily predicted and computed.

Word games and number games aim to challenge the mind to deduce patterns by cleverly hiding them with noise. This noise acts as an adversary to confuse a mind 'clouded' with complexities. In the above example, if the player were told that some pattern exists in a sequence containing 1100110, a reasonable starting point would be mathematics or logic to deduce the information. However, these paradigms are not needed to deduce or create a pattern. The player can simply copy the string and append it to the end and claim a viable pattern exists. Or, as in the case above, the player can tack on another set digits to create a pattern utilizing the information that already exists. In these two cases, the sequences 1100110-1100110 and 1100-1100-1100-1100 have been created from a seemingly random set of digits. Both are deterministic and predictable. Therefore, any intuition about a set of seemingly random digits can easily be proven false.

This intuition pitfall regarding random sequences is referred to as the Undefined Reference Sample (Whitney, 1990). No intuition can hold if an object, such as a sequence or arbitrary number, is presented with no information of where it came from. In the above case, a sequence that looked random can be shown to be deterministic. The reverse also holds true. For a seemingly deterministic sequence, a simple rule can show that it is random. For example, the number 1111 does not look random because it contains a recognizable pattern of repeated 1's. However, it is perfectly normal to choose this number in a random drawing of numbers from

0 to 5000. It is therefore difficult to label a sequence of numbers given no intimation of the rules that produced it.

1.2.1 Random Process

A process for generating a sequence of random numbers, which are independent of each other, is easier to define. The next number in a random sequence that is generated using a random process cannot be predicted by referring to the previous numbers. A random generation process has no memory of previous events to generate future events. For example, if it is stated that an evenly balanced coin will be tossed end over end into the air making a number of tumbles that are unable to be measured by the human eye, then it is safe to assume that the sequence of heads and tails will be random. This is because the complex movements of the coin are unpredictable and depend on immeasurable factors. The trajectory of the coin is extremely sensitive to the initial conditions of the event. A slightly different angle or a slightly differing wind direction can produce different outcomes, even though the coin is obeying the laws of physics. Other truly random processes include dice throwing, a roulette game, and card shuffling, assuming that no factors can unfairly affect the outcome (such as weighted dice or a sneaky dealer). These tasks rely on the unpredictability of the underlying physical processes in place, whether they are inherently random or even chaotically deterministic. Naturally-random processes are arguably random however, since they depend on an infinite amount of microscopic factors as well as larger factors such as temperature or

wind. Depending on the rules of the universe, they may also be dependent on time. For example, if it were possible to travel back in time, would seemingly natural random events occur the exact same way twice? Since time travel has yet to pan out, and the amount of natural factors involved is infinite and impossible to measure, it is acceptable that such processes are deemed random by using mathematical testing techniques. Examples include the statistical goodness-of-fit Kolmogorov-Smirnov test and the 'noise sphere' technique between triplets of random numbers (Weisstein, Kolmogorov and Noise Sphere 2004). The goal of these tests is to estimate that a random number or series of random numbers have been chosen seemingly independently from a given probability distribution.

The central rule of probability theory is that a large amount of independent events will cause a random variable to converge to some likelihood of occurrence. An infinite series of coin tosses will result in 50% heads and 50% tails with absolute certainty. The result of a single coin toss is completely uncertain. This statement is reasonable before the toss, but is completely false afterwards since the uncertainty of the event is gone. Since an infinite amount of events cannot occur in a finite time frame, a random variable is measured based on its expected value. The randomness imbedded in a random algorithm causes them to be measured by expected behavior. This allows the unpredictable algorithm behavior to be averaged and hopefully it behaves according to some expected bounds.

The built-in determinism and the ability to model only concrete mathematical equations unfortunately make generating a random sequence impossible by means of a computer algorithm. This is because no matter how sophisticated, an executed algorithm is a completely predictable series of steps that performs a task. A predictable process cannot be used to create random numbers. Computer algorithms must therefore rely on a pseudorandom process in order to obtain near-random sequences of numbers.

1.2.2 Pseudorandom Process

A pseudorandom process approximates a truly random process, yet unlike a random process, it uses previously generated values to obtain the next value in the sequence. Such an algorithm is used to deterministically generate sequences of numbers that appear random when statistically tested. Pseudorandom generators use an initial 'seed' value to begin the generation of numbers. Therefore, the same 'seed' value yields the same sequence of 'random' numbers, and is perfectly predictable. This fact becomes a burden when dealing with computer security, which relies heavily on random numbers for secret keys. Eq. 1 shows the deterministic algorithm commonly known as the 'linear congruence' method of calculating seemingly random values. In the equation, variables *A*, *C*, and *M* are non-negative integers. The initial value of *X* is the seed (Liu, 1999).

$$X_{i+1} = (A * X_i + C)(\mathrm{mod}\, M) \tag{1}$$

Successive calculations using the linear congruence method generate pseudorandom numbers that have the potential of passing statistical tests for randomness (Eastlake, Crocker, & Schiller, 1994). To avoid repeating number sequence patterns, it is useful to reference the system clock for a seed value. Turning the clock back and running the algorithm again at the same instant of time would, unfortunately, produce the same sequence. With time constantly moving forward, the seed value would be different at each future unit of time. Therefore, it is possible to look at algorithms that generate pseudorandom sequences using time as the seed value as being truly random if the universe works in similar ways in which the clock cannot be turned back. Time travel is beyond the scope of this thesis, but not beyond the scope of the system clock.

1.2.3 Recommendations

The incompleteness of a pseudorandom number generator can often be overcome by the scope of the problem. Pseudorandom numbers are useful for gaming, simulation, and sampling. When the scope of the problem requires a better solution, truly random numbers must be used. True random numbers are not difficult to obtain and are recommended for use in security applications.

Normally, hardware electronics suffer from random electromagnetic disturbances. For example, the static noise on the radio or the ‘snow’ on a television screen is the presentation of natural random energy picked up by the

antenna or receiver as a result of other electronics in the area or even what has been attributed to leftover energy from the big bang, seen as Cosmic Background Radiation. A natural way to obtain random numbers is to capture this energy at some point in time. For example, to create the effect of a random coin toss, one can choose a pixel on a black and white screen, and measure its color value through a series of frames. Figure 1 is a screenshot of a program that fills in black and white squares using pseudorandom numbers. After filling in a rectangle of pixels and processing many frames per second, the result appears like an off-broadcast television station. Although the image is completely random, can the mind still find patterns?

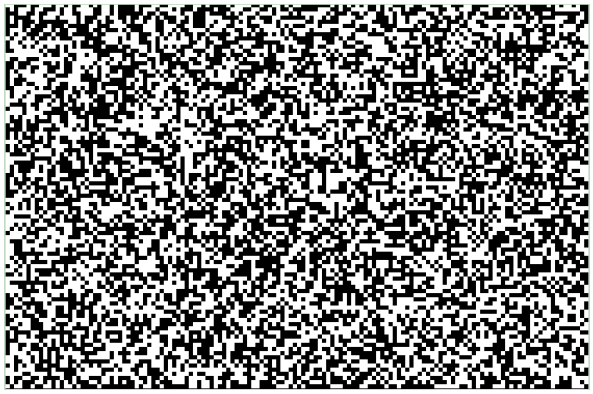


Figure 1. One frame of Random Noise.

Randomness plays a crucial role for security systems, especially in applications over the Internet (Eastlake et al., 1994). Security systems rely on cryptographic algorithms that try to foil adversaries attempting to recognize

patterns. The trick is to use algorithms that contain little to no patterns. This task is extremely difficult, given that computer systems are built on structured mathematical rules. Randomness acts as a means to provide the unpredictability needed in a secure system. Hardware provides a good level of unpredictability needed to obtain random sequences (Eastlake et al., 1994). The linear congruence method, as described earlier, may be suitable for simulations but terrible for security systems due to the ability to decipher an entire pseudorandom sequence given the initial state (Eastlake et al., 1994). A pseudorandom process does not provide the level of security required for generating secret values such as password and keys.

1.3 Early Uses

Randomization in algorithms was first used to find approximate solutions to numerical problems. Named after the city that is famous for roulette tables and probabilistic gambling in the Principality of Monaco, the development of numerical probabilistic algorithms called Monte Carlo dates to atomic bomb research in World War II (Brassard & Bratley, 1996).

Prior to WWII, numerical probabilistic algorithms were employed on a smaller scale to solve problems. Most notably, the method of "Buffon's needle" dates to the eighteenth century where Georges Louis Leclerc, comte de Buffon, used random methods to approximate the value of π with needles thrown at random

onto wooden planks (Brassard & Bratley, 1996). With the needle being half as long as the width of a plank and the plank cracks having a width of 0, the probability of a needle intersecting a crack between planks was proved to be $1/\pi$. Therefore, n throws results in n/π needles intersecting plank cracks. As the number of needles thrown moves towards infinity, the answer gets more precisely closer to the true value. However, like most numerical probabilistic algorithms, this precision gain is extremely slow. This method is described fully in Section 1.6.

The Leclerc algorithmic approach to approximating π is not practical since deterministic methods have shown to be much more precise. However, this early example was one of the first probabilistic algorithms, and stands as an intriguing example of the power and ability of such algorithms.

1.4 Random Algorithms

"An algorithm corresponds to a Turing machine that always halts" (Motwani & Raghavan, 1995, p. 17). Represented as an abstract model of computation, a *randomized* algorithm is a *probabilistic* Turing machine that always halts. This type of Turing machine chooses transitions randomly from the set of available transitions and accepts or rejects input with some probability (Motwani & Raghavan, 1995). Like a non-deterministic Turing machine, a probabilistic one has many paths to choose from, yet only follows one at a time instead of all of them in parallel ("probabilistic" & "nondeterministic", NIST 2004). The path to be chosen

is the result of a random draw. See Section 1.5.2 for the complexity classes used to organize decision problems that random algorithms solve.

At a lower level, randomizing an algorithm is a process that uses a dynamic and unpredictable mechanism to re-order input, sample a population, distribute objects, measure variations, and calculate approximations. This mechanism (e.g. random number draw) helps the algorithm make decisions to estimate problems where closed form solutions or deterministic methods are too complex and/or infeasible. These cases arise in the real world where the computation of an exact solution is not possible, in principle, because of the uncertainty of data and/or the computational resources are unable to model the needed units (e.g. irrational numbers – $\sqrt{2}$, $\sqrt{3}$, π, e) (Brassard & Bratley, 1996). In the precise world of the digital computer, an answer may be infeasible due to the amount of running time it takes to find it. Making random choices and arriving at an approximate solution may be preferably faster than a lengthy runtime search for the optimal solution. As a result of the uncertainty in random algorithm decisions, approximate answers and/or varying run times exist for a problem instance.

Deterministic methods aim to produce the same solution with each run and execute according to a fixed set of rules. Any variation or error in their behavior for a specific instance of a problem will prove that the algorithm is never suitable for that instance (division by 0, etc.). In contrast, random algorithms should behave differently from one run to the next. Variables include: the length of execution

time, as well as the result of the algorithm. Solutions may vary to a certain probability, or even be incorrect altogether. If the algorithm returns a known incorrect result, it can be executed again to hopefully arrive at a better solution. If there are multiple solutions, comparing results after a combination of runs provides an increased level of confidence.

The most common random algorithms behave in such ways that are similar to human behavior. For example, using randomness in searching makes the run time of the algorithm vary from run to run. Like a human searching a telephone book for a specific name, a random algorithm can be guided by the alphabetic order, but pinpointing an entry can be an uncertain task that involves random decisions. Humans cannot search the phone book in a strictly deterministic fashion because there are other factors that our minds must perceive than just the ordered list of entries. For instance, large ads are provided in the Yellow Pages, so a search for Ryan's Surf Shop would start in the R section of the retail stores category. Some amount of randomness would lead the visual search to an advertisement instead of the list of textual entries, hopefully finding the phone number in large, bold print quicker. The advertisement stands out compared to the list of entries that all look the same with very small text.

Random algorithms help to suppress the killers of deterministic algorithms, adversaries. An adversary is an input to an algorithm that causes it to perform poorly. For example, Quicksort has a very fast O(n log n) average-case running

time. However, the adversary of an already-sorted list causes the algorithm to behave with $\Omega(n^2)$ time. Random algorithms 'foil' adversaries by making random decisions on-the-fly so that they cannot be predicted and fooled. A randomized Quicksort algorithm chooses a pivot at random and allows the expected running time to be O(n log n) for all input instances.

Much like the human mind, random algorithms are built to focus on a varying set of problem situations. Random algorithms are useful when dealing with the following problem spaces: 1) As stated above, adversary conditions that cause deterministic algorithms to perform poorly can be thwarted using random methods to reduce or eliminate their negative affect. 2) If a search space contains a large number of acceptable solutions, a random sample from the population can be used to efficiently find one of them. 3) Random sampling also helps to obtain a solution from a subset of a population to approximately model the entire population. 4) Deadlock and symmetry problems show that randomness is helpful to load balance resources and avoid or break deadlock conditions. 5) Environments where variety and uncertainty are necessary to provide training use randomness to approximate and simulate real-world effects. 6) Simulation must also be able to test scenarios and obtain statistical data, and random algorithms provide the necessary mechanisms to do so. 7) Where creativity is needed, especially in the field of artificial intelligence, random algorithms attempt to make decisions outside

of the bounds of determinism and provide controlled noise in the form of unpredictable variety.

The following pseudo-code examples display the inner workings of a few randomized algorithms. Each example utilizes a procedure ‘uniform(i..j)’ to obtain a random value in the interval ‘i’ to ‘j’. The value can then used for making a decision, testing an event, feeding an object attribute, assigning a task to a processor, etc. This results in algorithmic procedures that contain varying run times and varying answers.

Chaos Game - draw random points according to simple rules.

Input: 3 points of a perfect triangle, width and height of graphic display
Output: a visual approximation of the Sierpinski gasket (Pascal triangle with odd numbers displayed as points)

Source: (Gleick, 1987)

```
1: Draw the points of the triangle
2: Choose a random starting point: P = uniform(1..width), uniform(1..height)
3: Loop
4:      Choose a random vertex: V = uniform(1..3)
5:      Calculate new point. newP = ½ way between V and P.
6:      Draw newP.
7:      Set P = newP.
8: End Loop
```

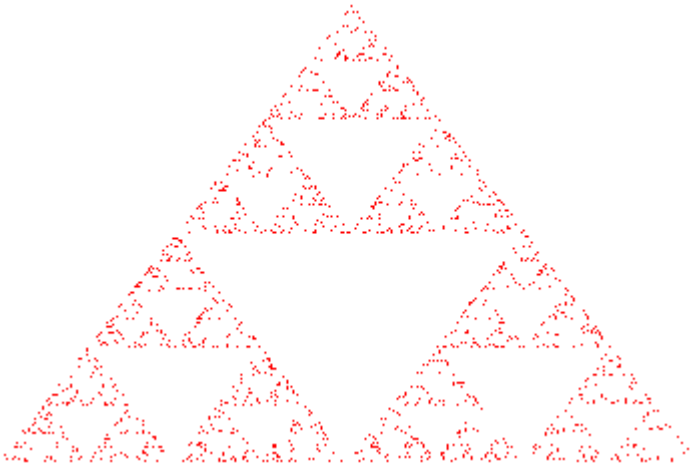


Figure 2. Chaos Game randomized algorithm with example result.

Min-Cut – find a set of edges (with minimum cardinality) to remove that breaks a connected, undirected multigraph into two or more components (cut). **Input**: connected, undirected, multigraph G **Output**: a cut (and candidate min-cut) of G **Source**: (Motwani & Raghavan, 1995)
1: While G.number_of_vertices > 2 loop 2: Pick random edge: E = uniform(1..G.number_of_vertices) 3: Contract edge vertices and preserve multi-edges 4: Remove loops 5: End loop 6: Output remaining edges

Figure 3. Min-Cut randomized algorithm.

The benefits of random algorithms are outlined throughout this paper. Two common features that random algorithms contain are speed and simplicity. One unusual feature that they may also possess is reliability. Random algorithms are reliable when confidence bounds are defined for their range of solutions or range of expected run time. These algorithms will sometimes have so small of an error that the probability of a hardware failure is more likely. Therefore, if a slow deterministic algorithm has to run longer than the hardware is reliable for, then a faster, approximate random algorithm will provide a much better solution (Brassard & Bratley, 1996).

1.5 Complexity

The universal computational machine, the Turing machine, is only as good as the software (algorithm) that is ran on it. The main challenge of developing the software is to solve problems efficiently. Gödel's incompleteness theorem shows

that there exist problems that are unsolvable due to the incompleteness of 'self' describing rules in a complex system. Also, the Church-Turing thesis shows the famous undecidable problem in which a Turing machine cannot tell ahead of time if an algorithm will halt or provide an answer. These theories show that algorithms must be analyzed and measured in time and space to determine if they are useful.

1.5.1 Standard Problem Classes

Complexity theory classifies problems based on their difficulty. The P class of languages contains decision problems that can be solved in polynomial time by a deterministic Turing Machine. Problems in this class are considered to be tractable. The NP class of languages contains decision problems that can be solved by a non-deterministic (multiple-tape) Turing Machine in polynomial time. The answer to an NP problem can be verified quickly, but not necessarily solved quickly. NP-hard refers to the class of problems that are naturally more difficult than those in NP. If a problem's correctness can be verified in polynomial time (NP), and its algorithmic solution can be translated to solve any other NP problem (NP-hard), then the problem is classified as NP-Complete**.** These problems are the hardest of the NP class.

By measuring time and space requirements for an algorithm, complexity theory expresses the existence of problems that can be classified as intractable. Algorithms built to solve these problems are slow or infeasible. For example, solving the Traveling Salesperson problem, which is NP-Hard, for a large number

of cities could take more time (thus, cost more money), than if the salesperson were to make an educated guess and be willing to accept some amount of error. Therefore, in order to provide approximate solutions to intractable classes of problems, estimation algorithms are needed. These algorithms are considered acceptable if they are efficient (i.e. polynomial), and contain a high probability of not producing terribly incorrect answers.

The human mind often uses approximation to reason and decide in the world. Instead of pulling off onto the shoulder to wait and see how a traffic situation pans out, a driver instead must sit in the traffic jam and estimate the best route without knowing the obstacles that lie ahead. This estimation ability uses probability and statistics to analyze a situation. Randomness is inherent because of the uncertainty that probability theory contains. Therefore, an algorithmic process that uses the result of a random draw to make an approximated decision has the ability to estimate reasonable solutions.

1.5.2 Random Problem Classes

The above standard problem classes can be generalized to allow probabilistic requirements for the behavior of random algorithms. Random problem classes use probabilities to describe their correctness in a polynomial run time (Motwani & Raghavan, 1995). Random Polynomial (RP) algorithms accept input with probability 50% or more if the input is a member of the language. If the input is not a member, the algorithms accept the input with zero probability. These

rules mean that the algorithm only errors for input that is a member of the language. This is known as a One-Sided Error Monte Carlo algorithm. A Two-Sided Error Monte Carlo algorithm is allowed to error for both members and non-members of the language. These Probabilistic Polynomial (PP) algorithms correctly accept input with probability greater than 50% and incorrectly accept input with probability less than 50%. Bounded-Error Probabilistic Polynomial (BPP) algorithms put tighter bounds on the error probabilities with a polynomial number of iterations to reduce the error probability further.

Random algorithmic behavior can also be classified as Zero-Error Probabilistic Polynomial (ZPP). These algorithms still make random decisions but always produce correct answers. The trade-off is a variation in run time. These algorithms are named Zero-Sided Error Las Vegas. Las Vegas algorithms are described further in Section 1.8.

1.6 Numerical Probabilistic algorithms

Numerical probabilistic algorithms are one type of random algorithm that always yields approximate answers to numerical problems. They give a probability of correctness and a given confidence interval of upper and lower bounds. These algorithms may improve on the precision of the answer along with the tightness of the bounds with increased available running time. A real-world example of this type of algorithm is an opinion poll, with its deterministic equivalent as a general

election. A general election takes a lot more time and resources to execute than a poll in which a random sample of votes is used to approximate the opinions of many. The more sampling that is performed, the more accurate the poll will be to the actual election.

An example of this type of algorithm comes from a classic technique of estimating the value of π. As stated in Section 1.3, the Buffon needle experiment uses randomness to estimate the value of π within certain boundaries that get smaller as the algorithm is repeated. For example, if only 10 needles are used, the best possible estimation of π is 3.333…. The algorithm could also possibly output an answer of 1.0 with the bad luck that only one needle intersects a plank crack. This answer is not incorrect; it is just outside of the expected bounds. Using a larger set of needles, the estimate of π becomes closer. Buffon proved that the answer would be correct if an infinite amount of needles were used. This is not possible on a computing machine with finite resources, so we must deal with the numerical probabilistic approximation.

The Buffon's Needle method works by exploiting the properties of geometry utilizing randomness to approximate area ratios. The angle of a randomly thrown needle (θ) ranges from 0 to π as measured from the center point of the needle. The distance from the center of a needle to the nearest plank crack (D) is never greater than ½ the distance between cracks. Since the length of the needle is ½ the size of the distance between plank cracks, an intersection of the

needle and crack occurs when D is less than or equal to (¼ sin θ). Figure 4 is a diagram of the experiment space. The blue line, f(x)= (¼ sin θ), represents the threshold for needle intersections with plank cracks. The area under the blue curve, from 0 to π, measures ½. The area of the entire experiment space is ½ * π. Therefore, the probability of a randomly thrown needle intersecting a plank crack is the ratio of the area under the curve to the total area (½ / (π/2)) = 1/π. In a simulation of N randomly thrown needles, the points representing θ and D are uniformly distributed over the search space, shown in pink on Figure 4. Therefore, the ratio of total points (N) to points on or below the curve (number of needles that intersect plank cracks) is approximately equal to π.

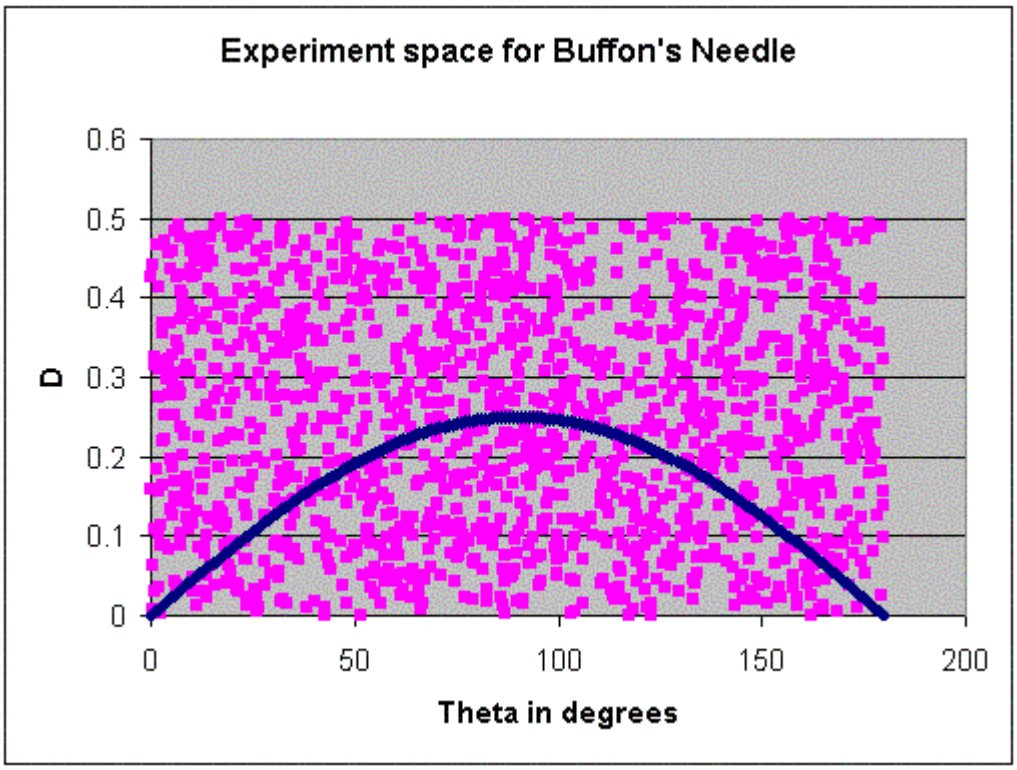


Figure 4. Buffon Needle experiment space.

As stated previously, as the number of needles increases (pink points in Figure 4), the value of π gets more accurate. Figure 5, Figure 6, and Figure 7 each show a scenario of a Buffon's Needle program with varying amounts of needles. With 10 needles, Figure 5 contains 4 intersections, and a π estimation of 2.5. The 100 needles of Figure 6 intersect planks 31 times estimating π at 3.23. The 10,000 needles of Figure 7 have 3187 intersections, thus a π estimation of 3.1377.

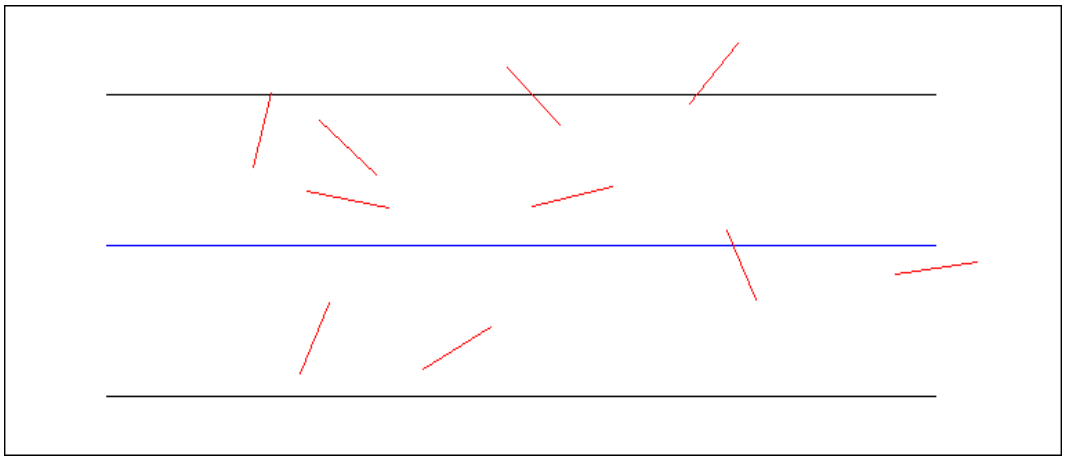


Figure 5. Buffon's Needle. 10 needles.

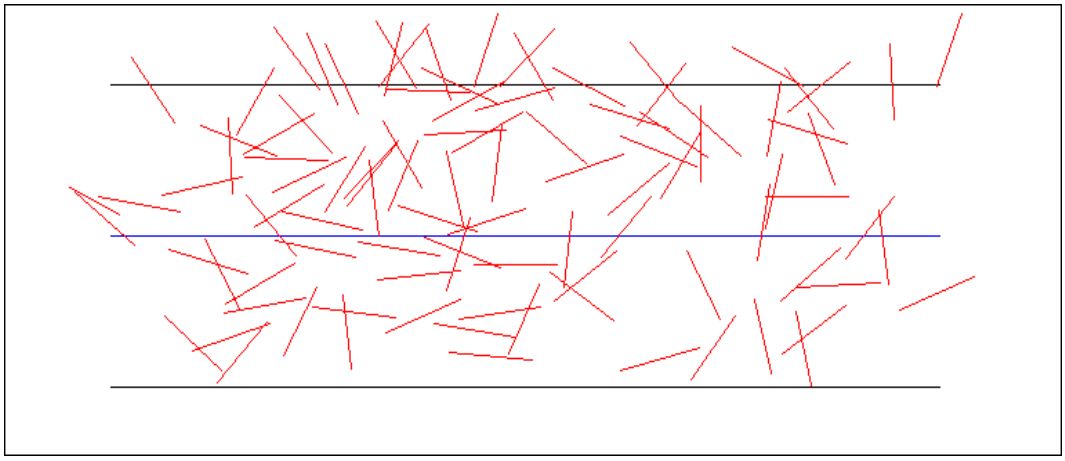


Figure 6. Buffon's Needle. 100 needles.

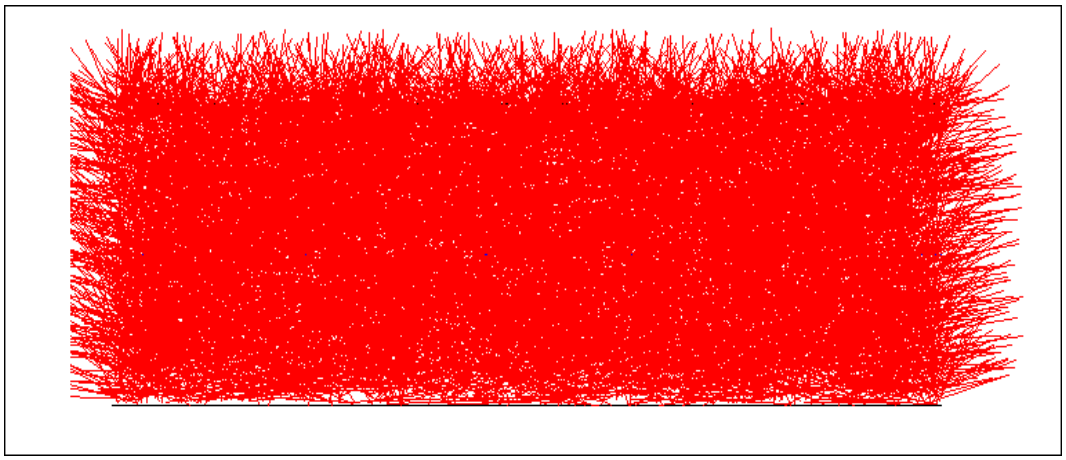


Figure 7. Buffon's Needle. 10,000 needles.

A very similar example to Buffon's Needle is the estimation of π using the ratio of a unit circle encompassed by a unit square. As shown in Figure 8, the number of randomly plotted points that fall in the circle area divided by the number of total points in the square area is approximately equal to π divided by 4. This is based on the fact that the ratio of the area of the circle to the area of the square is exactly π divided by 4. An algorithm that plots random points in a unit square and computes the ratio of points in the unit circle to those in the square is numerical probabilistic. The algorithm attempts to uniformly fill in the areas and obtain a better answer with more trials.

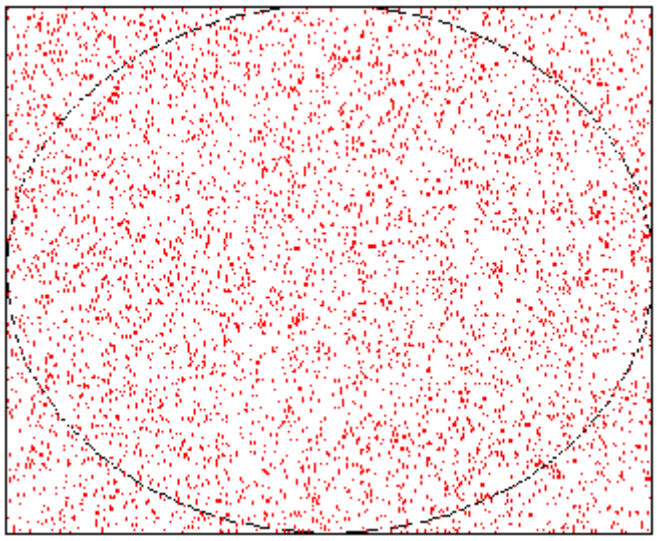


Figure 8. Random points occupying a unit square and unit circle.

Numerical probabilistic algorithms relate to intelligent ways of searching, recognizing patterns, and simulating the mind. These algorithms continuously sample a population and attempt to provide an estimate. In the field of artificial intelligence, it is important to use clever techniques to approximate difficult problems. Numerical probabilistic algorithms help by simulating the 'thinking' of a mind that wishes to take some amount of time to build and improve the answer. For example, while quickly trying to do mathematics, the mind may first take some time and estimate the values to obtain a quick, rough answer. The mind may then process the numbers further and improve upon their estimation. Like the performance of numerical probabilistic algorithms, the run time of a thought must

be analyzed ahead of time to get an idea of how long it will take to obtain a solution, and how accurate that solution will be.

Brainstorming is an intelligent technique that attempts to extract an unknown solution from a set of related ideas. In reference to Buffon's Needle, the search area is represented by the topic of a brainstorm activity. Every random needle thrown represents an idea, uniformly covering the topic area. The resulting solution via ratio comparison is the consideration of a subset of developed ideas, within some boundary, to the whole set. The result of the brainstorm activity is an average, core idea. As with Buffon's Needle, the preferred scope or 'accuracy' of the brainstorm is improved with repetition. As the amount of generated ideas that are random and uniform grows, the more effective the solution turns out since repeated trials seek to fully cover the search space. Newly generated ideas while brainstorming can be stimulated by previously generated ideas, thus improving the solution and tightening the bounds of variation.

Numerical probabilistic algorithms are often referred to as Monte Carlo. This thesis regards Monte Carlo algorithms as a similar technique where there exists the possibility of obtaining an incorrect answer.

1.7 Monte Carlo algorithms

True Monte Carlo algorithms produce answers with a high probability of correctness on every instance, but unlike numerical probabilistic algorithms, run

the risk of producing incorrect answers. These types of random algorithms must be able to handle all instances of a problem with none having a high probability of error. Some Monte Carlo algorithms "… allow *p* [the probability of correctness] to depend on the instance size but never on the instance itself." (Brassard & Bratley, 1996, p. 341).

If a Monte Carlo algorithm is unable to determine if an incorrect answer has resulted, allowing it more running time may reduce the probability of error. On the other hand, some Monte Carlo algorithms have the ability to produce an answer that will positively be known to be correct. If this answer is obtained, then it is certain that the correct solution has been reached. If the answer is not obtained, then repetition of the algorithm may yield a higher confidence interval, and/or a wider search for the definitive answer. Allowing a Monte Carlo algorithm more time to produce a more confident answer is known as "amplifying the stochastic advantage" (Brassard & Bratley, 1996, p. 341).

An example of this certainty is represented in the verification of matrix multiplication algorithm known as Freivalds (Brassard & Bratley, 1996) that may output a 'false positive'. When the algorithm returns false, the answer is guaranteed to express that two multiplied matrices do not equal a third (Brassard & Bratley, 1996). Repetition in the absence of a guaranteed answer drops the probability of error in the algorithm. Many Monte Carlo algorithms that attempt to

solve decision problems are known for their rapid convergence to approximate equilibrium.

The min-cut algorithm specified above in Section 1.4 Figure 3, is a Monte Carlo algorithm that has the possibility of returning a candidate that is not a min-cut. For the graph in Figure 9, a valid min-cut occurs when removing edges 0 and 1, or 4 and 5. Many repetitions of the Monte Carlo algorithm produce approximately a 66% chance that one of the valid solutions is found. The other 34% of solutions output by the algorithm are incorrect and of cardinality 4, such as edges 0-2-3-5 or 0-2-3-4.

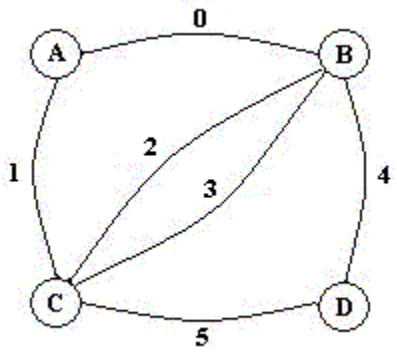


Figure 9. Connected, undirected multigraph.

A Monte Carlo method notable for testing a very large odd integer for primality is popular because no deterministic method is known to be optimal (Brassard & Bratley, 1996). Application of such an algorithm is important to security encryption methods where large prime numbers are used for key values.

Monte Carlo algorithms relate to the workings of the human mind, which is known to produce incorrect answers from time to time. Whether attempting to recognize a pattern or recall some memory, the mind produces a probability of correctness with error bounds, and runs the risk of being wrong.

1.8 Las Vegas algorithms

A Las Vegas algorithm is a type of randomized algorithm that uses a random value to make probabilistic choices and never produces an incorrect answer. The choices made during computation attempt to guide the algorithm to the desired solution faster than other methods. This is possible because of the ability of randomness to avoid adversary conditions that may lead to a lengthy exhaustive search for the correct solution.

A simple Las Vegas algorithm example is a sock-sorting program. This program works with the problem of selecting matching socks from a drawer. Any deterministic algorithm to accomplish this task will take O(n) worse-case execution time when faced with a bad instance. For example, a deterministic algorithm could loop through an array of socks and attempt to find a match for the very first sock selected. As new unmatched socks are picked up, they are continually eliminated. With an instance of input such that the only matching pair is the first and last elements, this deterministic approach must search through the entire list, thus O(n) behavior.

The randomized, Las Vegas version loops through the array and collects pairs of socks. If a match is not found, it randomly eliminates one of the chosen socks. This algorithm has a very small possibility of taking just as long as the deterministic approach worst-case due to bad decisions. It also shares the possibility of failing to find a matching pair with the deterministic approach. However, simulations of the random search show the expected behavior of 4 choices for every input instance. The randomized version has eliminated the adversary input instances that cause poor running time in a deterministic method.

One generic type of Las Vegas algorithm may perform efficient correctness checks and, rather than producing an incorrect answer, output no answer at all (or better, an apology message). These error cases can be handled by repeatedly running the algorithm until a successful answer is found. For a deterministic algorithm, this behavior is unacceptable. It is acceptable for Las Vegas algorithms if the probability of a dead end is not high or if an efficient deterministic method does not exist, such as large integer factorization (Brassard & Bratley, 1996).

Instead of occasionally returning no answer, other Las Vegas algorithms are always guaranteed to return an answer, but could suffer long running times due to poor choices. These algorithms are usually used when a known deterministic algorithm can solve a problem quickly on average, but suffers major setbacks when encountering a specific input instance. The randomness in the Las Vegas versions of these algorithms is used to reduce or remove the probability of these instances

from occurring. The worst case is not prevented, but instead the association is removed between the bad instance(s) and their probability of occurrence.

Las Vegas algorithms cause the phenomenon known as the Robin Hood effect (Brassard & Bratley, 1996). That is, when the deterministic method counterpart solves an instance very slowly, the Las Vegas algorithm performs quickly. On the other hand, when the deterministic method is fast on an instance, the Las Vegas method slows it down. Similar to Robin Hood, Las Vegas algorithms steal from the rich (fast deterministic instances) and give to the poor (slow deterministic instances). However, the average case behavior of such algorithms over any instance of the problem results in good expected performance.

A more common and useful example of a Las Vegas algorithm to address the selection and sort problem is called selectionLV (Brassard & Bratley, 1996). The problem of finding the k-th smallest element in an array can be handled deterministically by partitioning the array using a pivot point, and repeatedly searching each sub-array. This technique known as divide-and-conquer is most efficient when the pivot point is as closest to the median of the elements. Calculating the exact median is not efficient because the process involves a special case of the problem at hand. Deterministically choosing a pseudo-median avoids the "infinite recursion", but is still inefficient (Brassard & Bratley, 1996). Choosing the pivot as the first element is better, with average linear execution time, but has worst-case quadratic time. Therefore, deterministic approaches with linear

worst-case times are not optimal because of hidden constants, and simple deterministic approaches require quadratic worst-case time.

The selectionLV algorithm chooses the pivot randomly to avoid the pitfalls of the deterministic worst-case instance. The execution time is now only dependent on the size of the instance, instead of the instance itself. Any instance of the problem results in linear expected time, although quadratic time is possible. The possibility of quadratic behavior results from poor random decisions, and becomes very small as the instance size grows.

The same idea is used for the popular sorting algorithm known as Quicksort. This deterministic algorithm has a very fast O(n log n) average case running time. However, in the worst-case of an already-sorted list, the algorithm behaves with $\Omega(n^2)$ time. The recursive nature of splitting an array according to a pivot is optimal for Quicksort if the pivot splits the array into same size sub-arrays. The result of choosing a pivot at random causes the expected running time to be O(n log n) for all instances under consideration.

Chapter 2: Theory

2.1 Approximation and Simulation

Problems that are NP-complete and/or NP-hard are unlikely to be optimally solved using a polynomial running time algorithm. The intractability of finding an exact solution can possibly be solved by a number of interesting methods including the following: using an exponential running time algorithm, isolating special instances, or using a polynomial running-time algorithm that outputs near-optimal solutions (Cormen, 2001). The mentioned polynomial, near-optimal method of providing approximate answers is usually good enough for situations where it is reasonable to sacrifice optimality for a feasible, efficient solution.

The National Institute of Standards and Technology defines an approximation algorithm as: "An algorithm to solve an optimization problem that runs in polynomial time in the length of the input and outputs a solution that is guaranteed to be close to the optimal solution. "Close" has some well-defined sense called the performance guarantee" ("approximation", NIST 2004). Randomization in algorithms is one of many methods used for the approximation of problem solutions. Monte Carlo and numerical probabilistic algorithms both produce approximate answers. They specify a type of 'performance guarantee' in terms of a probability of correctness and/or confidence intervals.

Simulation is an approximation technique used to model the real world. Using randomness to abstract details, repeated statistical tests are executed to narrow in on a solution with a sense of accuracy. It is a powerful way to study complex problems without analytically studying fine details. These details are abstracted and the resulting solution is an estimated proportion. For example, a simulation of the weather may find that the probability of rain is 80% when a cold front moves through. A weather simulation does not model every atomic detail of the wind, pressure, and temperature conditions at every point in space. Many factors are estimated, which could lead to the forecast being incorrect. Although a simulation could be slow and costly, it could also save lives and money for sensitive systems where extra analysis is never a bad thing. The alternative to simulation is an even costlier experimentation effort consisting of trial and error.

Simulation allows choices to be made without actually making them. Choices are 'virtually' made, and the results are studied to see the behavior of a real system under approximately the same environment. The simulation can then be run over and over with different arrangements to study the effects. Once the effects are acceptable, the variables in the simulation can be applied to the real world, and a real decision can be made with confidence that the expected behavior is known.

Simulation is a type of random algorithm that is solely responsible for approximating and analyzing. A simulation contains an approximation mechanism that causes results similar to a random algorithm. Like random algorithms,

simulations can be wrong. Weather simulations are often incorrect for the path of a hurricane, or the movement of a cold front. These imperfections arise because the simulation itself is imperfect. However, if a simulation were constructed to measure every detail, it would be very costly and serve as a useless redundant system. Simulation attempts to approximate the unimportant details and pinpoint the end result.

2.2 Performance

The theoretical study of random algorithms is an important and necessary science due to the uncertainty contained in the computational process. While deterministic algorithms are analyzed for their worst-case time performance, random algorithmic performance presents a different problem. Numerical probabilistic algorithms produce different answers on repeated runs; Monte Carlo algorithms can be wrong; and Las Vegas algorithms produce varying execution times. Therefore, the analysis of these algorithms cannot be defined by solid rules like deterministic algorithms. The complexity analysis of these algorithms must be performed using estimations of the expected behavior.

2.2.1 Complexity Analysis

Analyzing algorithms that use random methods is quite different from analyzing deterministic complexity. The notion of averages and expectations is appropriate for random algorithms since these algorithms work with a random

variable chosen from some probability distribution whose value is uncertain. Therefore, analyzing such algorithms must take into account the average or expected value of the random decisions.

The average running time of deterministic algorithms is the measurement of likely behavior of the algorithm over a series of problem instances. This measurement assumes that each possible instance of a problem is equally likely to occur at random. The problem with this approach is that if some instances are more likely to occur, which occurs quite frequently in some problems, then the average behavior can be misleading since the instance probability distribution is not uniform. For example, updating a checking account history usually involves inserting the newest transactions into a pre-sorted list. The entries themselves must be sorted and inserted so the list is in correct order. Such algorithms like Insertion Sort can do this computation much faster than its average case, which in this case, is misleading (Brassard & Bratley, 1996).

In order to make average case analysis useful in deterministic algorithms, random methods can be used to modify the instance probability distribution. A deterministic algorithm that performs well under the average case, yet has a bad worst case, can be altered to make the worst-case instance very unlikely or impossible to occur. By incorporating a random variable, the algorithm can become less prone to the worst-case instance.

2.2.2 Expected Run Time

With random algorithms, the complexity analysis heuristic that is widely used is the *expected* running time. This refers to the mean running time of a randomized algorithm on a particular instance, multiple times. Unlike the average time of a deterministic algorithm, the expected time of a random algorithm is "defined on each individual instance" (Brassard & Bratley, 1996, p. 331).

Since random choices are under direct control of the algorithm, it is not useful to measure the unfortunate case where an algorithm is inefficient due to bad choices. Unlike deterministic algorithms, no one particular instance causes worst-case behavior in a random algorithm. While one instance may be a victim of the algorithm's terrible choices and have a long running time, the next may be solved quickly due to better or more flexible choices. The analysis of random algorithms measures the expected equilibrium behavior over multiple runs on the same problem instance, similar to the expected equilibrium value of a random variable.

Instead of relying on a probability distribution of input instances, where some may occur more or less often than others, random algorithms attempt to treat every instance in an equal manner. The expected behavior is therefore applicable for all instances instead of a subset that must be averaged. The distribution of the random decisions made internal to the algorithm governs its behavior. Understanding this distribution is important for analyzing the expected behavior. However, it is still helpful to measure the average and worst case expected

behavior. These measurements refer to the analysis of the expected running time of an algorithm with respect to the average and the worst instance of a given size as opposed to the behavior based on lucky or unlucky decisions.

2.2.3 Unknown Run Time

Another interesting category of analyzing random algorithms concerns the consideration of run times that may be unknown. While the benefits of this realization may be minimal, the importance can be seen when considering some human-like processes. For example, it is obvious that humans do not perform linear searches for retrieving memories. We instead can only guess at how we can lose a thought and regain it at some later time, such as remembering a dream that occurred a week ago. How are we to analyze such performance? The uncertainty of thought recollection processes causes the run-time of a memory search to be unknown. The uncertainty built into a random algorithm can cause them to perform in the same manner. For example, when drawing a number of random black and white pixels on the screen to simulate an off-broadcast television channel, how long must we wait until every other pixel is black, and every other pixel is white? Is it even possible to estimate this? Probability theory states that this event will never occur (probability of 0). This does not rule out that it can happen since it is a legitimate distribution of pixel configurations. However, the continuous random variable takes on the value so rarely that its proportion of occurrence is reduced to 0.

Randomness translates to a level of uncertainty, which cannot be tolerated in an environment that is required to be highly reliable. Randomized algorithms produce approximate and uncertain answers, and have uncertain run times. In critical real-time applications such as avionics or biotechnology, the use of random algorithms should be limited. It is ironic that the computer has seemingly been developed to function as a slave in which simple deterministic algorithms, that are reliable and fast, are trusted to perform tasks better than that of their creator.

2.3 Probability and Game Theory

Probability and Game Theory are the best mechanisms for explaining randomness. Probability problems can be both intuitive and perplexing, even at the same time. Game Theory uses probabilities to analyze games and predict the behavior of players. Random algorithms can be both designed and analyzed using the rules of these theories to determine expected behavior and rational strategies.

2.3.1 Intuitive Probability Problems

Random methods are intuitive to the algorithm designer when the order of decision operations does not matter and there is no supporting evidence of choosing one path over the next. For example, situations that cause a deadlock condition have arrived in a state of equilibrium where it is not important who/what takes precedence. What matters is that the deadlock must be broken to avoid losing processing time. A processor with four pending tasks that have arrived at the same

time, and determined to be equal in size, has no gain in picking one over another. The goal is to get out of the deadlock state and process any of them. Picking the order at random is a fair way of breaking the deadlock.

A deterministic algorithm is unable to fairly break the symmetry because it contains no variation in its choosing ability. Deterministic algorithms are 'dumb' in that they cannot make decisions to ward off adversaries. This is analogous to a pinball machine that continually shoots the ball in an unbreakable circular path in which the player can accumulate a large amount of points with absolutely no interaction. The user has found a built-in adversary, and has to eventually wait for the inherent randomness in the universe to free the ball. Without a variation in bounce speeds, or perhaps a random spin of the ball, the machine is 'dumb' and cannot adapt to fight this condition.

The conscious decision of choosing something randomly raises an interesting question in the debate of mind versus machine. Any decision in which a human must "just pick one" involves some sort of random generation. How do we perform such a task? Do we base it on other events? Are the other events independent of each other? For example, did I decide to put on my left sock first because I happened to stub my right toe yesterday on a box that was delivered incorrectly to my house because the delivery man was upset because his right sock had a hole in it? Just pondering about one simple case makes it mind boggling to think of the large amount of randomness that one uses per day.

2.3.2 Counter Intuitive Probability Problems

Probability problems must be analyzed fully to realize their value since they may present counter-intuitive results. A probability problem that is not properly analyzed could cause haphazard decisions because of incorrect assumptions. For example, consider a version of the Prisoner's Dilemma described by Frederick Mosteller (Weisstein, Prisoner 2004). Three prisoners apply for parole and only two of them are to be released. One prisoner asks a knowledgeable warder for the name of one of the lucky prisoners. The warder tells the prisoner the name of one of the two other prisoners. While the prisoner now would think that his chances of being released are 50% because only two prisoners remain, they are actually still 66.666…%, the same as they were if he did not have the extra knowledge. The prisoner's incorrect assumption is due to the misuse of extra knowledge in a probabilistic environment.

Now consider a similar, yet opposite problem known as the Monty Hall problem. Named after the famous television game show host, this problem involves three doors in which only one contains a prize behind it. A player chooses one door that they believe contains the prize. The problem exists when the host displays a booby prize behind one door, and asks the player if they want to switch their guess to the remaining door. Since the strategy for the first choice was rationally random, it would seem that the second choice of switching would also be. However, statistical analysis shows that this is not the case. Probability theory

shows that switching doors results in a 66.666…% percent chance of choosing the correct door. When the player does not switch, the chance of choosing the correct door is only 33.333…%.

It seems that these two problems are identical, yet have drastically differing rational solutions. In the Prisoner's Dilemma, it seems rational that the extra knowledge would allow for better chances. In the Monty Hall problem, it seems rational to either stick with your initial 'gut' instinct and not switch your decision or randomly decide to switch your decision. However, in these cases, the intuitions are false. Extra knowledge in the Monty Hall problem is beneficial, and extra knowledge in the Prisoner Dilemma is irrelevant.

Programming these problems reveals that they are not so tricky after all. Figure 10 and Figure 11 combine a random number generator and logic statements to show that the Prisoner Dilemma is simply a random choice between three numbers, and that the Monty Hall problem results in two common solutions (switching doors, 66.666…%) and one lone solution (sticking, 33.333…%).

Prisoner Demo – performs a guessing game with two of three prisoners to be released. Repeated iterations of this program will result in a probability of 66.666…% release rate since (C==H) 33.333…% of the time. **Input**: **Output**: boolean – true if released
1: Identify a prisoners to be held: H = uniform(1..3) 2: Identify a curious prisoner: C = uniform (1..3) 3: Release one prisoner != H 4: Release other prisoner != H 5: Return (If C was released)

Figure 10. Prisoner demonstration algorithm.

Monty Hall Demo – performs a guessing game with three doors and switches the guess when a dummy door is opened. Repeated iterations of this program will result in a probability of 66.666…% win rate since (G==D) 33.333…% of the time. **Input**: **Output**: boolean – true if win
1: Choose a door that hides the prize: D = uniform(1..3) 2: Choose a guess of the prize door: G = uniform(1..3) 3: If: G == D switch guess to incorrect door – return false 4: Else: switch guess to correct door – return true

Figure 11. Monty Hall demonstration algorithm.

It is important to notice how probability theory has certain properties that can lead to misuse. Some non-intuitive properties of a random process were pointed out in Section 1.2.1. The above mentioned Prisoner's Dilemma points out another useful property: predicting future events according to probability theory

does not follow the intuitive Law of Averages (Whitney, 1990). An independent random event is completely random on every single instance. This is why a random flip of a coin could possibly turn up heads five times in a row. Using past knowledge does not change the probability of an independent event to occur. The Law of Averages only applies to past events and probability theory shows that the likeliness of an event is applicable for an infinite number of trials. Since the future is not known, the Law of Averages cannot be used. Therefore, probability theory is simultaneously claiming to describe the future with complete certainty (probability of an event occurring) and to describe the future with complete uncertainty (randomness, anything can happen).

Like a magician and his bag of tricks, these counter intuitive situations occur because of uncertainty in the problem domain. The magician takes advantage of their naïve audience and amazes them with the unexplained. The audience's lack of understanding causes them to accept what they see because they do not know any better. The information that the magician is allowing them to perceive does not fully explain the situation. The magician attempts to render the intuitions of the audience false through the use of illusions. For example, it is assumed that a lady sawed in half cannot continue to smile and wave at the audience, but the magician is able to cleverly shock the audience by disguising the event with 'smoke and mirrors'. Misuse of the illusions presented by magicians

(sawing a person in half is bad) shows how the illusions of probability theory can lead to incorrect assumptions.

2.3.3 Minimax Principle

With the realization of probability theory benefits and pitfalls, it becomes clear that randomness and probability are the perfect mechanisms for analyzing and playing games. The analysis of games and the role that chance plays highly depends on the study of probability. Games are designed to challenge players and allow them to devise strategies in order to win. Ideally, the most rational strategies will win. Ironically, randomization helps to determine which strategies are the most rational.

A zero-sum game is one in which a player benefits only at the expense of other players. At any point in the game, the net-amount won and lost for all players is zero, such as in Chess or Poker. In these games, it is ideal for an offensive player to choose a strategy that will maximize their payoff outcome, while a defensive player would ideally like to minimize it. An optimal strategy allows the player to guarantee a payoff amount that they will be satisfied with, no matter what action the opponent takes. In games where there is no definite strategy that will produce optimal results for any player, a randomization method can be used to choose a rational strategy. A definite strategy would, in essence, make the game boring and cause the player to have no sense of creativity. A devised strategy according to a probability distribution would allow for interesting play.

Using the payoff matrix in Figure 12, the optimal choice for the row player in order to maximize the minimal payoff amount is strategy b. The optimal choice for the column player in order to minimize the maximal payoff amount is strategy b also. This game has a solution in which a player can deterministically establish their optimal strategy and place a bound on their payoff amount. Each player is guaranteed to payoff 0 no matter what the other player chooses to do.

	a	**b**
a	0	-1
b	1	**0**

Figure 12. Payoff Matrix with solution.

The payoff matrix in Figure 13 shows why probabilities must be used to find a strategy in the absence of a solution. Each player does not have a clearly defined optimal strategy. If a player were to choose one strategy and stick with it, they could end up losing many points to the other player. Assigning probabilities to the strategies allows for a more interesting game. If each player chooses a strategy with a 50% probability, the expected payoff of the game is 0. Also note that if the row player chooses strategy **'a'** with 90% probability, and the column player chooses **'a'** with 90% probability, the expected payoff is now 0.64 points in favor of the row player. The expected payoff is calculated by summing the row and column probabilities multiplied by the point value (Motwani & Raghavan, 1995).

	a	**b**
a	1	-1
b	-1	1

Figure 13. Payoff Matrix with no solution.

John von Neumann's Minimax Theorem shows that using probabilities to determine a strategy causes the guaranteed maximized expected payoff amount to equal the guaranteed minimal expected payoff amount, thus any two-person zero-sum game always has a solution (Motwani & Raghavan, 1995). The offensive and defensive players agree to disagree about optimal strategies since they are designed to disrupt one another (Ruelle, 1991).

Using randomness when playing games is a perfectly rational way to devise a clever strategy against an opponent. In an uncertain competitive environment, the best strategy to use is random. A baseball pitcher can choose a random set of pitches to use to confuse a hitter. A boxer can devise a random set of punches to throw. A tennis player can serve over a random area and keep their opponents return strategy to a best-guess. The more random the activity, the more randomness the opponent needs to use to combat it. Therefore, in theory, the player with the most amount of randomness wins.

Acting according to probabilities allows for approximate answers, but more freedom. Always making the optimal choice can lead a strategy to a dead-end because of unforeseen information. For example, it is optimal for a boxer to hit as

hard as possible, but an unforeseen consequence is the stamina of the player. A boxer hitting with maximum strength can usually only last a few rounds. A boxer that randomly mixes strength, agility, and awareness is more rational. Using these principles, algorithms can act erratically and still produce rational solutions.

2.3.4 Lower Bound Performance

The Minimax Theorem provides more information about random algorithms than simply that of a mechanism to play a game. The idea of studying player strategies to accomplish some optimal goal coincides with the wish to place bounds on a random algorithm. A random algorithm is a struggle between an input distribution and an algorithm distribution. Since a random algorithm is eventually executed on a logical computing machine, it must follow the rules of a standard, deterministic state machine. Just like the rules of probability are obsolete after an uncertain event, a random algorithm execution path is deterministic after all random numbers have been generated. That is, the same random numbers will produce the same output. Therefore, a random algorithm with a finite number of states and finite input can be viewed as a distribution of deterministic algorithms. This fact allows a random algorithm to be analyzed for a lower bound using the Minimax Theorem.

For analyzing such a random algorithm, the payoff matrix contains rows of different input and columns of the deterministic algorithms contained within the random algorithm. The payoff amount specified in the matrix is some measure of

the algorithm such as run time or memory. Just like a game payoff matrix, the row player (the adversary choosing an input) would like to maximize the payoff and cause the algorithm to perform poorly. The column player (the mechanism choosing which algorithm to use) would like to minimize the payoff and execute efficiently. If a deterministic strategy were used to choose an algorithm, then the optimal strategy would result in the worst-case performance. This is because if an adversary input is chosen, the algorithm will guarantee an upper bound of the behavior. No matter what input is chosen, the algorithm will perform no worse than the chosen strategy. However, using a mixed (random) strategy results in a probability distribution over the set of deterministic algorithms resulting in a Las Vegas, randomized algorithm (Motwani & Raghavan, 1995).

For this random algorithm, represented as a set of deterministic algorithms, it is possible to define a lower bound on its behavior. Choosing the worst possible input distribution and the best algorithm, the complexity is smaller than the pure strategy (deterministic worst-case) because the input distribution is known (Motwani & Raghavan, 1995). Using the Minimax Theorem, this game has a solution, and the solution reflects that the complexity of the best algorithm for the worst input is a lower bound for the expected run time of any randomized algorithm. For any input, the expected run time of the optimal algorithm is a lower bound of the optimal Las Vegas algorithm since the randomness allows for other algorithms to perform the task at hand. To prove the lower bound, any input

distribution can be used to find the best behaving expected run time of the deterministic algorithms. This lower bound is useful for random algorithms in which the computation time is finite, the number of algorithms is finite, the number of inputs is finite, and the size of every input is finite.

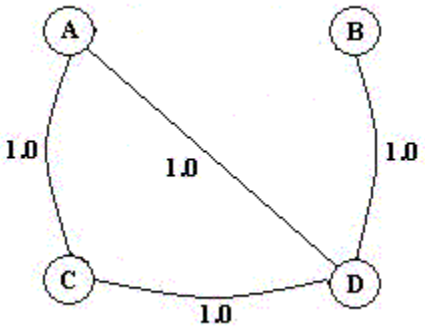
The Minimax Theorem shows that the input distribution does not matter when proving a lower bound. This is because the algorithm must be able to handle the worst possible input. Given the worst input, the Las Vegas algorithm can perform no better than its best deterministic computation behavior. This reduction to determinism is beneficial because the input distribution is known, which is an advantage in any game.

2.4 Random Walk

Modeling probabilities and games over time is achieved by using a construct known as the random walk. Any system can be modeled by a set of states and transitions. The behavior and design of random algorithms benefit from the patterns that can be discovered walking through a probabilistic state machine.

A deterministic state machine has the knowledge of exactly how it arrived in a state, and all future states are determined by the transitions of the past. There is no variation of behavior because the movement from state to state strictly follows the rules of logic. The machine's final state can be completely predicted given an initial state and a sequence of actions.

A state machine that uses probabilities to determine state transitions allows for a more dynamic and flexible machine. Given the initial state of the machine, and a set of actions, the final state cannot be completely predicted. The transition from state to state is based on the result of a random draw. Navigation through such a machine is called a random walk. A random walk can be performed on any connected, undirected graph where the next state to visit is chosen uniformly at random from the set of neighbors. Figure 14 shows a random walk and associated transition matrix. The weights of all edges are equal, so the probability of transition between vertices is calculated by 1/(# neighbors). In Figure 14, vertex D has three neighbors, so the probability of transition to each one is 0.333…, which sums to 1.

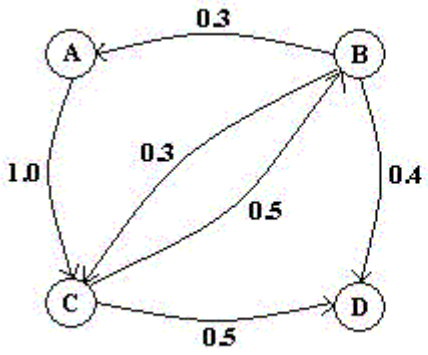


$$T = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0.0 & 0.0 & 0.5 & 0.5 \\ B & 0.0 & 0.0 & 0.0 & 1.0 \\ C & 0.5 & 0.0 & 0.0 & 0.5 \\ D & 0.\overline{3} & 0.\overline{3} & 0.\overline{3} & 0.0 \end{array}$$

Figure 14. Random Walk graph with transition matrix.

A Markov chain is an abstraction of a random walk over a graph that contains weighted, directed edges. For each vertex, the transition to the next state

is calculated by normalizing the outgoing edges to create a matrix of transition probabilities. The higher the weight, relative to the other outgoing edges, the more probable the state transition. The normalized outgoing edges therefore sum to 1. Figure 15 shows a Markov Chain with edge weights that are already normalized. Unlike a deterministic state machine, a Markov chain is memoryless. Future states of the system are determined on-the-fly and are only dependent on the current state of the system, not the previous states.



T =

	A	B	C	D
A	0.0	0.0	1.0	0.0
B	0.3	0.0	0.3	0.4
C	0.0	0.5	0.0	0.5
D	0.0	0.0	0.0	0.0

Figure 15. Markov chain graph and transition matrix.

As an abstraction of a random walk, a Markov chain is also an abstraction of a random algorithm. A random algorithm contains a number of states, and the draw of a random number determines the movement along the states, according to a distribution, to a solution. Although random walks contain uncertainty, there are measurable properties that can be used to determine the expected behavior of a

given probabilistic state machine. These properties are, in turn, useful for analyzing the behavior of random algorithms.

The measurable properties of a random walk can be divided into two main areas discussed in the following sections. The first area deals with the distribution of the expected final state in random walks of a certain length. The second area groups together measures of the expected number of cycles to transition between states and includes the complicated and mysterious cover time.

2.4.1 Endpoints

A random walk by itself is, by definition, very unpredictable. It does not express anything about the nature of the state machine. However, a series of random walks on the same graph can extract interesting patterns and measure commonalities that may not be obvious. As the quantity of random walks increases, the distribution given by the transition probabilities emerges. The study of these patterns extracts the expected behavior of the state machine, even though one run can vary greatly from the next. Analyzing a random walk by measuring the distribution of the final states gives a good indication of the nature of the machine.

Given a random algorithm that transitions between states on a line, it is beneficial to determine the expected final state of the algorithm after N cycles. For example, a random walk can be performed to simulate the movement of an ant towards food. The ant moves left or right with 50% probability. If the food were

located only towards the outreaches of the ants range, then the only ants that would survive would be the most adventurous ones. Starting in the middle of the line with a large number of ants, only few would reach the food since the distribution of endpoints in this type of random walk is similar to a normal curve. These ants must make the same directional move many times in a row. This is the same as a random sequence of 1's and 0's turning out 000000 or 111111. It is rare, but not impossible.

Figure 16 is a summary of a random walk simulation with 10,000 random walks of length 10 along a line. The random walk begins with the center vertex and the movement is binary (left or right). The number of random walks that completed at a particular vertex is shown as vertical bars with the exact quantity specified in red. Because the length of the walk is even, no random walk can finish at the even numbered vertices. The odd numbered vertices show that most walks ended towards the center starting point, with few walks reaching the ends.

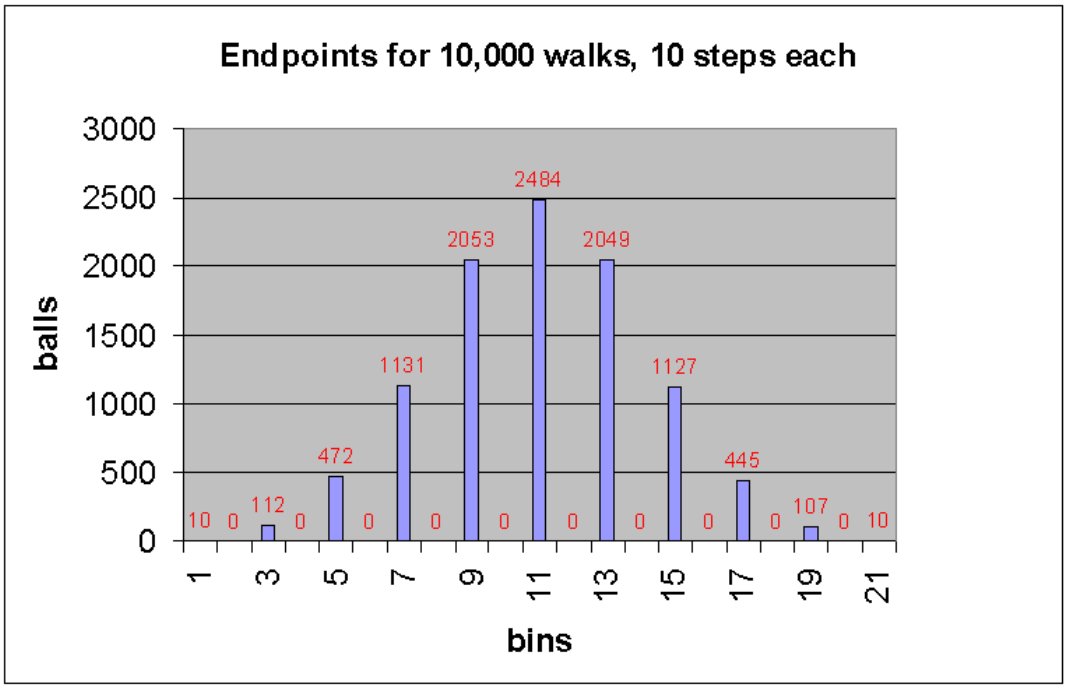


Figure 16. Distribution of random walk endpoints on a line.

Another way of performing the above random walk model to produce the familiar endpoint distribution pattern is to use a Galton Board. A Galton board is a structure of pegs arranged in a triangle with row N having N pegs. A random walk on a Galton board can be expressed by dropping a ball from the top, and watching it bounce to the bottom with the probability of 50% falling to the right or left of one peg in each row. The number of paths to the bottom row of bins in a Galton board is equal to the values of the Pascal Triangle (Whitney, 1990). Therefore, it is not surprising that a large number of random walks produce an approximate binomial distribution of balls into bins, like the numbers of the Pascal Triangle. With a large amount of walks, the number of balls in bins can be normalized and approach the

proportions of the Pascal Triangle numbers. This is the same result for the random walk along a line since the Galton board is, in essence, equivalent. The Galton board shows that the random walk will tend to follow the most common paths. The middle bins are easily accessible, and the outer bins are difficult to reach.

To obtain accurate values of the Pascal triangle, a very large amount of random walks on a Galton Board must take place. As the number of bins increases, the amount of random walks must drastically increase to get a good approximation. This is why Figure 17 shows only up to the ninth row since it took 1,679,616 random walks to achieve the accurate Pascal triangle results of the bottom row.

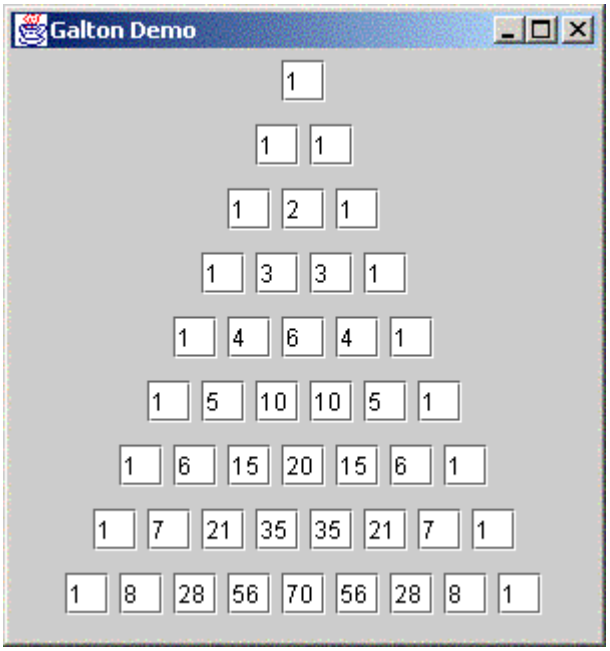


Figure 17. Rows derived by a Galton Board random walk.

2.4.2 Routes and the Cover Time

Other important measures of random walks involve the expected number of cycles needed to visit a range of vertices. The hitting time (expected number of cycles to travel from one vertex to another), the commute time (expected number of cycles to travel from one vertex to another and back), and the difference time (the delta between the two (to and from) paths in the commute time) are measures that provide addition information about the patterns of a random walk. The cover time is a unique and difficult measure of a random walk. The cover time measures the expected number of cycles needed to exhaustively cover all states. Although it is not known if the cover time can be computed exactly or approximately in deterministic polynomial time, Feige and Rabinovich (2003) outline a deterministic algorithm that can approximate the cover time with polylogarithmic factors. This deterministic algorithm is an alternative to a polynomial numerical probabilistic algorithm that simulates a series of random walks and approximates the cover time. The numerical probabilistic version calculates random numbers and walks along the graph counting state transitions until all vertices are covered. The simulation then repeats the experiment and calculates the approximate result. Like any numerical probabilistic algorithm, the precision gain is very small for a large increase in simulations. Therefore, for a large amount of vertices, the algorithm is not reliable.

The application of a random walk and the cover time relates to the intelligent task of decision-making. A random walk acts as a good mechanism to create or find a decision in which an expectation value exists. Many random walks along the problem space converge to some set of expected decisions. However, some decision paths will stray from the norm from time to time. The cover time is an estimate of the expected time to visit all states of the decision-making process, thus could be used to produce a well-informed decision. As well as decision-making, a random walk and its properties also apply to the problem of allocation.

2.5 Allocating Balls into Bins

Many problems of chance deal with the ability to estimate the distribution of balls into bins. How many rounds of poker are necessary to obtain a royal flush? How can a number of tasks be distributed to complete them all within a given time frame? A distributed computer network, a microprocessor, or even a workflow software application must be able to handle load balancing of processes to resources efficiently. With a completely random interface to the world, these systems have no knowledge of incoming processes (balls) that need allocating. They must distribute the processes to a number of resources (bins) in a rational and efficient manner.

2.5.1 Allocation Problem

Randomness can benefit the allocation process and eliminate the need for a central point of control. It provides variation, needed to minimize the load over resources, and distribution, to reduce the number of decisions required to allocate a process. A deterministic allocation scheme has limited flexibility and requires a 'global controller' to assign balls to bins.

Like a deterministic allocation process that can calculate and guarantee loads, a random one can similarly set bounds on load balancing. The 'classical allocation process' assigns a ball to a bin by choosing the bin uniformly at random and has an average allocation time of 1 (Czumaj & Stemann, 2001). This method is still not ideal due to the possibility of wasting resources because of unlucky random draws. For example, the first bin could be chosen five times in a row to allocate processes, while all the other bins are idle. Therefore, it is useful to study 'adaptive allocation processes' that still use randomness, but provide a better tradeoff between the maximum load, the maximum allocation time, and the average allocation time (Czumaj & Stemann, 2001). Another useful strategy for allocation is to reassign processes to improve efficiency. Reallocating processes to resources can provide an ideal balance, but the operations involved are usually expensive and should be limited (Czumaj & Stemann, 2001).

A random walk, described in Section 2.4, acts as a simulation of the allocation process. Processes can be assigned to resources by being distributed by

some random walk scheme where the expected patterns are known, and the variation from the expected path is acceptable and possibly beneficial. For example, if four processors were available with a binomial distribution of computing power (2 slow machines and 2 fast), a Galton board random walk would allocate more processes to the fast machine if they are arranged like the bottom row of a 4 level Pascal triangle with the slow machines on the outside and fast machine in the middle (1,3,3,1).

Algorithms that schedule and allocate processes to resources are susceptible to adversaries, especially in distributed, fail-prone environments (Chlebus & Kowalski, 2004). Distributed computing allows for the execution of independent tasks concurrently. A distributed system must deal with processors that fail, or crash, and be able to reallocate tasks. Adversaries decide which distributed processors to fail and when. 'Weakly-adaptive' adversaries must choose the processors to fail prior to execution. Therefore, randomization during execution can be used to disguise the assignments of processes to resources (Chlebus & Kowalski, 2004). Since the adversaries are unable to predict the uncertainty, they cannot produce an unfair failure strategy. This allows randomized algorithms to solve the problem of performing tasks reliably in a distributed environment to be more efficient than deterministic methods (Chlebus & Kowalski, 2004).

2.5.2 Coupon Collector Problem

When using randomness to distribute processes to resources, it is important to place bounds on the allocation time. This, and other problems that are suitable for random algorithmic techniques to perform intelligent tasks, such as game playing, fall into a category that is described by the Coupon Collector problem. This problem estimates the number of trials that are necessary to randomly allocate balls into a set of bins such that each bin has at least one ball. The name is derived from a collector of coupons that wishes to have at least one coupon of each available type by randomly choosing them. The problem encompasses the most influential areas of analyzing random algorithms such as the cover time, random walk, and allocation time.

The main problem with randomness is the uncertainty in the value of a random variable over very few problem instances. The definition of the variable being random causes the value to be completely unknown, even if it is chosen from a known distribution. The run time for waiting for a particular value of a random variable is completely unknown. The only help in this area comes from the Coupon Collector Problem. The solution to this problem tells us when all the bins will be full opposed to when one of them will be filled. For example, when waiting for the number '3' when uniformly picking a random number between '1' and '10', the only accurate estimate that can be made (other than the probability of 0.1 of being chosen) is derived from the Coupon Collector Problem. The solution to the

problem produces an expected upper bound of about 29 trials before the number 3 is chosen. Therefore, the number 3 is expected to be chosen anywhere from the first choice, to the twenty-ninth choice, and possibly more.

The solution to the Coupon Collector Problem comes from harmonic numbers. Motwani and Raghavan (1995) show that the expected value of the number of trials to collect at least one of every type of coupon is n*H(n) where H(n) is the nth harmonic number. The nth harmonic number is defined in Eq. 2. Therefore, the expected number of trials to collect all N coupons is given in Eq. 3.

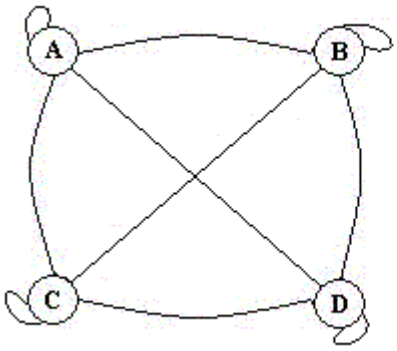
$$H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \tag{2}$$

$$\mathrm{E}[x] = 1 + \frac{n}{n-1} + \frac{n}{n-2} + \ldots + n \tag{3}$$

Although the solution to the coupon collector problem can be calculated easily for small values of N, the calculation gets messy for large N. Therefore, for large N, the solution must be approximated. Motwani and Raghavan (1995) put a sharp threshold on the approximation of the solution.

The Coupon Collector problem provides expected upper bounds for the cover time of a random walk on a complete graph with self-loops, as shown in Figure 18. Many simulations of a random walk along this graph converge to the actual solution for the quantity of vertices (bins). For example, in a simulation of

the graph in Figure 18 with four vertices, 100 random walks were averaged to produce an estimate of 8.76 while 10,000 walks estimated 8.3156. As the quantity of random walks increases, the solution becomes closer to the actual value of 8.333….



T =

	A	B	C	D
A	0.25	0.25	0.25	0.25
B	0.25	0.25	0.25	0.25
C	0.25	0.25	0.25	0.25
D	0.25	0.25	0.25	0.25

Figure 18. The Coupon Collector random walk graph and transition matrix.

The application of the Coupon Collector problem in intelligence problems is the ability to provide expected upper bounds and obtain a controlled grasp of uncertainty. A random search with replacement is bounded from above by the expected number of uniform random choices to make to exhaustively cover all possibilities. The bounds provide good worst-case analysis for guessing games and chance games where other analysis is too complex or infeasible. For the allocation problem, these bounds provide a randomized load-balancing algorithm with the expected number of task assignments to keep all resources occupied.

2.6 Search and Fingerprinting

Searching is the most important technique to finding patterns and devising strategies. The quickest and most efficient search technique is classified as the most intelligent technique. Accordingly, the ability to make the computer extremely fast and efficient when searching is one of many possible methods to demonstrate intelligence.

The role of randomness in searching is an ironic one. The benefit that uncertainty brings is the ability to be flexible and non-biased. Random algorithmic processes are not restricted to an orderly search path that could allow adversaries to take advantage of.

2.6.1 Random Search

Random Search is a method of searching with no planned structure or memory. This method has the same intent of the typical intuitive search, but the opposite strategy to achieve it. A random technique may be preferred if it takes more time to devise a better technique, or the additional work involved is negligible. For example, the Coupon Collector problem shows that a random search is bounded from above by the expected number of trials to exhaustively visit all vertices of a graph. If the extra factor associated with the random search is acceptable then the simplicity of it can be utilized.

Numerical probabilistic algorithms use random search when zeroing in on a target value. For example, approximating the area of a figure or approximating the value of π is a random search. Iterations of the algorithm loop independently to contribute to a random sample. At the end of all iterations, the sampled values are combined and usually averaged to output an approximated value. This random search is not exact, but instead is a random sampling technique that uses a subset of information to approximate a whole. Theoretically, an exhaustive random search, in this case, will lead to the entire population represented, thus emerging the exact, correct value. An exhaustive random search for a numerical probabilistic algorithm is one who's run time is infinite, like the probability of 'heads' or 'tails' for an infinite number of coin tosses.

With numerical probabilistic problems, the search is conducted for some unknown value that must be approximated. The random search is continuous because the accuracy of the answer is related to the number of repetitions. However, only an infinite number of repetitions will provide the exact answer. Better answers are provided with a large sample size that obtains more coverage of the search space. Random search is also beneficial where the solution pattern is known and can be verified so the search can end. This method is closer to the standard search scheme, where the goal is realized, yet performs the search in an ad hoc manner. This Las Vegas type of searching allows for creativity and adversary elimination. Since the search path is not known is advance, an adversary cannot be

planted ahead of time to cause the search to perform its worst. This ability also allows the algorithm to be free of any constraints such as leading down the wrong path and having to backtrack.

It is worthy to note the downfalls of a random search because it is certainly not the best technique for all situations. Random search has a very long upper-bound expected run time for a large search space, as defined by the coupon collector problem. It is only acceptable where the possibility of an exhaustive search is tolerable or not likely to occur. Random search may even take longer than a deterministic exhaustive search due to entries being revisited any number of times with replacement.

2.6.2 Fingerprinting

Fingerprinting is a mechanism for randomly mapping members of a population into a smaller population to allow for fast, approximate string matching (searching). The mapping of a member in the new, smaller population is called its fingerprint. This is a misnomer because the fingerprint is not exactly unique. This quality causes searches to be approximate when using fingerprint values for comparisons.

Motwani and Raghavan (1995) show a pattern-matching algorithm that uses the basic fingerprint function ‘mod’ to hash data to almost-unique values to match against. Therefore, unique values in the original population can take on non-unique

values in the new population. To obtain more unique values and reduce the occurrence of a false match while searching, it helps to choose larger values to hash against.

The role of randomness in fingerprinting is to fight adversary input that can cause many false matches, thus the search algorithm must verify each one and run longer. For example, if it was known that the fingerprint function was always (Z mod 3), then in a search space of '2222222', the algorithm would have to verify each occurrence of 2 to determine if it was really a '2' or a '5' (they both hash to 2).

Instead, if the value to hash against were chosen randomly, then it would be difficult to produce an input that is known to cause the algorithm to slow down for verification purposes. Also, if the value were chosen from a large set of primes, the probability of overlap, and thus a false match, would decrease. When using a prime number 'p' to hash against, a 'p' value that is equal to or smaller than the maximum integer value in the original population will produce the possibility of overlap. Over-lapping values in the new population is not a terrible thing and can be very beneficial for the algorithm if it does not happen too often. Motwani and Raghavan (1995) show a method of calculating the probability of a false match by bounding the set of primes by a threshold value.

The process of fingerprinting is presumed to be human-like because of the relations that are set up. Fingerprinting converts and breaks down (hashes) a data set into pieces, and assigns the pieces to a smaller set of recognizable values. Similar data is converted or abstracted to the same value. While the computer considers 'similar' to be mathematical structure, the human mind can consider 'similar' to be visual, aural, or even emotional structure. When doing pattern matching, the mind can compare against the abstract set of information instead of the entire original population.

This human-like quality of fingerprinting can be expressed in a problem such as recognizing a word in a sentence. For example, instead of recognizing an entire word by each individual character, the mind could convert similar 'looking' words into non-unique fingerprint values. Therefore, words like 'tree' and 'the' can have the same fingerprint: 'te'.

The fingerprinting function for this concept could be any number of manipulations such as removing random vowels or characters or removing redundant letters. Randomness could help by fighting adversary input that attempts to confuse the reader (searcher). For example, an adversary sentence could contain a lot of 'noise', like a word-find puzzle does, in the form of misspelled words or structurally similar words. Randomness could be used to hash these words into fingerprints that can be deciphered, yet sometimes can be interpreted incorrectly, like a tongue twister.

Chapter 3: Concept Demonstration

3.1 Concept categories

This thesis considers random algorithms to be loosely divided into three main categories. Any program that makes a decision using a random number is considered a Random Algorithm. However, the devised categories are split to express the purpose of the random draw. First, truly 'randomized' algorithms are those that are built upon randomness to perform some task. Many of these algorithms have been summarized in the above chapter. The second category of algorithms contains programs with 'injected' randomness to provide a level of variation. The randomness in these programs is not necessary to accomplish the primary goal, but instead acts as abstraction of details, or a simulation of reality. These features are beneficial for training environments where reactions are measured against unpredictable events. The third category combines inherent and injected randomness to attempt to solve problems intelligently using pattern recognition and puzzle solving. The following programs were written to demonstrate the second and third categories.

3.2 Abstracting Reality: Projectile Simulation

The projectile program demonstrates how randomness can be injected to express variety and even simulate the real world by abstracting fine details about the environment. The program is a physics simulation where projectiles are fired at

one another. If they collide, they will set off an explosion and disperse a number of new projectiles. The randomness is injected in the explosion where the particle velocities and angles use a random number generator to determine their future deterministic paths via physical equations. Therefore, each run of the simulation with the exact same initial projectiles will result in an explosion that looks different, yet is bounded by the probability distribution of the calculated explosion.

3.2.1 Determinism

A deterministic simulation contains no variation of behavior. This makes it difficult to create approximations of the real world where events are haphazard in nature. A version of the graphical projectile program using pure determinism to plot the explosion shows no variation in distribution. Also, determinism and preprogrammed paths will cause the explosion to look the same every time. Figure 19 shows a run of the program where the exploded projectile speed is fixed, and the angle is based on (array) index. Therefore, the red colored projectiles are the first 25 and have low angles; the next 25 are yellow with higher angles, and so on. Realistic explosions do not seem to exhibit this kind of behavior.

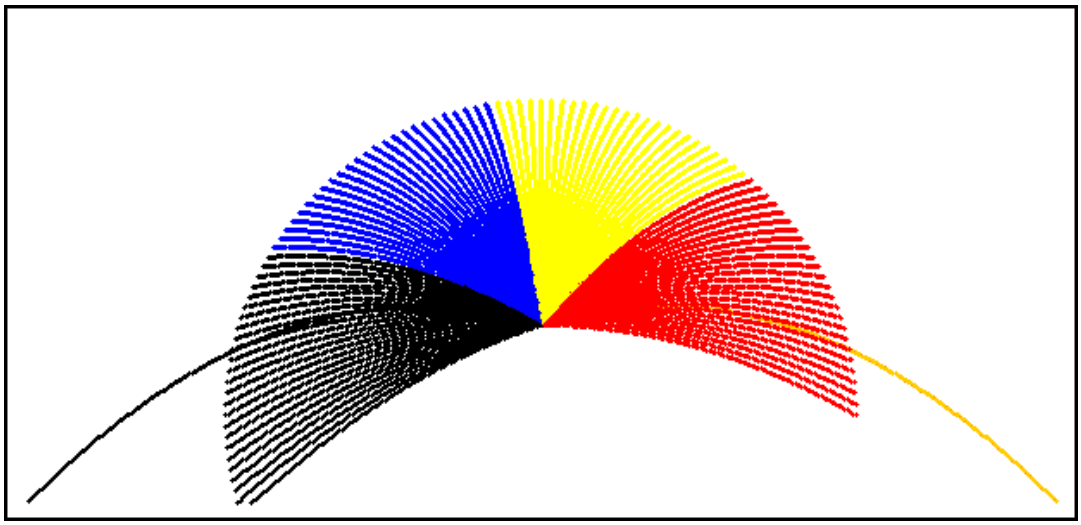


Figure 19. Projectiles following a deterministic path.

3.2.2 Randomness

An easy method of producing realistic results is to subject the projectiles to a random draw upon explosion. An explosion projectile's speed and angle can be randomly bounded using a pseudorandom number generator to obtain uncertain values from 0 to 35 meters per second, and 0 to 360 degrees, respectively. The distribution of the blast is now uniform and realistic. Figure 14 shows an example of using randomness. The colored projectiles of the deterministic explosion are now scattered all around in an uncertain manner. Repeated simulations will cause different 'looking' explosions, thus providing the uncertainty needed to study and train against similar behavior in the real world.

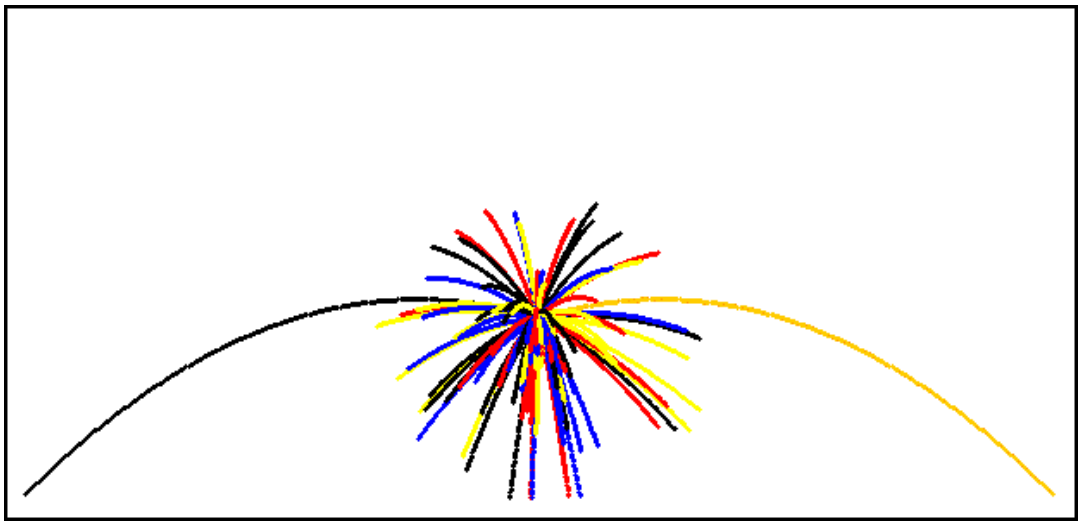


Figure 20. Projectiles using randomness to determine their destiny.

For a simulation to be 'accurate', it must be 'uncertain'. A simulation of the real world must contain some of the uncertainties that are expected. If a simulation was deterministic where the 'players' quickly recognize how events occur and why events occur, they will be able to tell the future (predict), fool the system (create adversaries), and cause unwanted symmetry (deadlock). If uncertainty is absent, then the 'players' are unable to train or learn the skills to handle unexpected events.

3.3 Intelligent Puzzle Solving: Word-Find

The classical word-find puzzle contains many features common to intelligence problems that involve randomness. The object of the puzzle is to

search and locate a set of words from apparent chaos. The intelligent tasks used to solve this problem are search, pattern recognition, and memory.

3.3.1 Determinism

Searching the puzzle can easily be solved using deterministic methods. The player can traverse one character at a time from the top left corner to the bottom right corner and search for words starting with that character. The number of positions searched is always the width multiplied by the height. This use of determinism does not satisfy the typical player because the challenge of the game is removed.

The devised deterministic solution is equivalent to the following randomized version, except the flow of the search through the puzzle is deterministic from top-left to bottom-right. The search ends when all words have been found. Therefore, an adversary condition exists when words are placed near the bottom of the puzzle. The deterministic algorithm must always traverse the entire character array to find them. This occurs quite frequently in standard word-find puzzles with a uniform distribution of words.

3.3.2 Randomness

When presented with a table full of seemingly random characters, and no other guiding help, a reasonable search method is a random one. Randomly

jumping around the board and looking for patterns provides the player with a more uncertain, yet challenging game play.

Using a random approach to solving the puzzle could possibly force the problem to be more difficult and take longer to solve. However, analysis of this problem shows that random techniques can be quicker than deterministic ones due to the presence of adversaries, and luck. As stated above, if all the words are grouped in the lower right corner, the deterministic algorithm will not find them until the end. The deterministic algorithm must search the entire puzzle in the worst-case and waste time by visiting empty 'white space' in the puzzle where words are not hiding. The random version selects bits and pieces from all over, and on lucky runs, is able to ignore much of the white space.

The devised solution to the word-find puzzle, expressed in pseudo-code in Figure 21, is based on a completely random search. The program continually loops and chooses a random character in the puzzle (step 2). This is the extent of randomization in the program, leaving all other processing as deterministic. The program then searches in eight directions for a series of characters to determine a list of possible candidate matches. For example, if the puzzle randomly chose the character 'a', and found the character 't' in one of the eight directions, then a candidate match could be 'hat' or 'cat'. Once a word is found, it is removed from the candidate list.

Random Word-Find - walk randomly along a puzzle and find words

Input: puzzle of characters P[][],list of words to find W[]
Output: empty word list

```
1: While (W.length > 0) Loop
2:      Choose a random character in P[][]: RC
3:      For (int I in up, up-right, right, down-right, down, down-left, left, up-left) Loop
4:          Gather next character in direction I: NC
5:          Append character RC to NC: SS
6:          Search W for words containing SS forwards: FSS[]
7:          Search W for words containing SS backwards: BSS[]
8:          For (int J in FSS.length) Loop
9:              Gather additional characters to create string of length J.length: T
10:             If (T==J) remove J from W
11:         Loop
12:         For (int K in BSS.length) Loop
13:             Gather additional characters to create string of length K.length: T
14:             If (T==J) remove J from W
15:         Loop
16: End Loop
```

Figure 21. Random Word-Find pseudo-code.

The key result, compared with the deterministic version, is the measurable execution time of the initial character search. The deterministic program visits all characters in the worst case while the random program visits a seemingly unknown amount of characters. The presumed problem with the random program is that the same character can be visited over and over, while the deterministic program visits each character only once. The worst-case expected runtime of a random search over the characters in a word-find puzzle is bounded by the results of the coupon collector problem.

In the random solution, each character represents a coupon to be collected. The deterministic program shows a guarantee that once all the characters have been

visited, all the words have been found. Therefore, when collecting random characters with replacement, the algorithm is simulating the coupon collector problem and its search time should be measured by the expected amount of trials to choose each character at least once. This gives an expected upper bound to how many characters are visited. Choosing every character at least once is the very worst case and will take a very long time for a large puzzle. Table 1 shows the coupon collector solutions for up to 25 coupons. The table expresses that, in a puzzle containing only 25 characters, a random search will take 95 trials to cover each character at least once.

Table 1. Coupon Collector solutions for up to 25 coupons

coupons	expected number of trials to collect at least one coupon of each type
1	1.00
2	3.00
3	5.50
4	8.33
5	11.42
6	14.70
7	18.15
8	21.74
9	25.46
10	29.29
11	33.22
12	37.24
13	41.34
14	45.52
15	49.77
16	54.09
17	58.47
18	62.91
19	67.41
20	71.95
21	76.55
22	81.20
23	85.89
24	90.62
25	95.40

While the coupon collector expected number of trials (nlnn+O(n)) is asymptotically greater than the deterministic search time (n), experiments while running the randomized program resulted in, on average, much fewer choices. This shows that the random solution is a good candidate for solving the puzzle and provides benefits that the deterministic solution does not. For an average amount

of words that are typically distributed (do not crowd the game board), the random solution usually searches faster, yet sometimes does perform more trials than there are characters. The randomized search time is therefore dependent on the quantity and size of words. The more crowded the puzzle is with words, the more characters the random algorithm must find. The smaller the length of words, the less likely it is that the word will be found. For example finding the word 'cat' could take longer than the word 'simulation' because the algorithm has a better chance of finding the larger word since the characters span over a larger area. Table 2 shows the quantity of characters visited to find all words for 20 random search trials on a 15X15 character puzzle. The varying amount of words demonstrates that the more crowded the game board, the longer the search takes. Also, each configuration averaged a search time less than the worst-case deterministic search time of around 225. However, trial 7 with 16 words and numerous trials with 22 words showed longer search times.

Table 2. Word-Find program result

number of words	3	16	22
trial 1	83	181	257
trial 2	143	163	163
trial 3	117	183	188
trial 4	36	138	195
trial 5	63	80	234
trial 6	106	151	92
trial 7	102	314	169
trial 8	40	180	103
trial 9	78	98	133
trial 10	100	98	239
trial 11	195	78	201
trial 12	106	195	195
trial 13	21	106	221
trial 14	18	122	103
trial 15	68	54	186
trial 16	127	175	331
trial 17	95	77	89
trial 18	88	71	240
trial 19	86	86	264
trial 20	30	164	128
total	1702	2714	3731
average	**85.1**	**135.7**	**186.55**

The random search of the puzzle is subject to the same uncertainty of any probabilistic trial. For one trial, the results are completely random. Therefore, the random search time can be anywhere from 1 to the expected coupon collector worst case, or even more. A search time of 1 is the very lucky case that the algorithm chooses a letter in which all words intersect. This is usually not the typical case, so the lower bound is on the order of the number of words (taking into account

overlaps). The expected worst case given by the coupon collector problem is considered to be unlucky, with a longer search time being very unlucky.

For argument's sake, we can 'write-off' the extra runtime factor and label it a 'challenge' or 'fun' factor. If an intelligent being were to attempt solving this puzzle, they would take the challenge out of the game by using a deterministic strategy. Even a 'deterministic' strategy from a human point of view may not be perfect because of mistakes in processing. The proposed random search strategy allows for the emergence of patterns from uncertainty and provides fun, mind-teasing game play.

Chapter 4: Applications

4.1 Pattern Recognition

The goal of this thesis is to understand how randomness plays a role in computing in order to perform intelligent tasks. The primary lesson to be learned is the uniqueness of all matter and energy in the universe. Intelligent beings are able to function in this world of randomness by defining reality as a set of patterns that abstract information. Intelligence is built on the ability to hypothesize and test, in order to recognize patterns.

4.1.1 Order from Uncertainty

Intelligent software needs to handle random information as input and make decisions in such random environments as puzzles or mazes. Pattern recognition is the process that intelligent beings use to make sense of randomness. It is inherent in the universe that processes are random, including the processes involved with every living being. As game theory suggests, in order to strategize and make sense from randomness, some random element must be used. If your opponent is using a random strategy, the best strategy to combat it involves randomness. For example, consider the task of peering into a room of items for some period of time, then recalling these items from memory at some future point. The viewer initially considers the items in the room to be completely random. It is their responsibility to then recognize or create patterns and store analogies into memory for extraction at a later time. The best way to handle the randomness of the items in the room is a

random process of perceiving. It is beneficial to the viewer to use their 'internal' randomness generator to decide on where and how to start memorizing items. The presence of adversaries could cause a deterministic method to break down. The arrangement of the room could be purposefully set up to confuse someone who starts left-to-right, or there could be so many items that trying to store the large amount of data in short-term memory will not be successful. Randomness provides a way to break the potential deadlock and offers a fresh, unbiased way of sampling data.

The process of random searching a random environment is most applicable to solving puzzles. Repetition in such processes is encouraged because the purpose is to remember redundancy and likenesses in order to classify patterns. As shown in Section 3.3, randomness is a good method of searching a word-find puzzle for words. In a word list, if the most common letter is 'e', then a random scan of the puzzle can find and store the different locations of 'e'. These locations can then be returned to in the future when testing different words.

4.1.2 Memory and Reconstruction

It is obvious that the human mind does not store and recall data like that of a typical computer system. For example, if words are stored in a database, it is trivial to enumerate every word. Imagine reading a dictionary, then being asked to list all the words from A-Z. This seems like a very complex task. However, the

task of recognizing if a word exists in the dictionary is a relatively simple task that is expressed in games like Scrabble®.

It is intuitive that the mind does not waste time storing all the words, but rather uses a more efficient method to construct or recognize a word such as storing grammatical rules or using hardware-like in-line functions. Then, when asked to verify a word, the mind uses the faster technique and comes up with a probability of correctness. Instead of storing the exact data (the dictionary of words), it is easier to approximately recognize them. Recognizing the data and matching it to a stored pattern can be accomplished using randomness, similar to the fingerprint hashing technique. Also, like fingerprinting, storing and recognizing words in the mind can produce incorrect output in which the meaning of a word is mistaken, or a word is constructed (spelled) incorrectly.

The recollection of a thought could be performed in a similar fashion. Instead of re-thinking a thought, the mind could store enough information to approximately re-create it. For example, children can easily remember the faces of their parents. They sometimes makes mistakes, but after some thought (gathering of supplementary data), realize that they made an incorrect assumption. The pattern matching that they are doing is thus not perfect. Some data is missing and must be reconstructed. Adults also can have similar pattern matching "mistaken identities". It seems that the reconstruction of the child's parent's face is from a subset of data. A computer algorithm may be able to act in the same way by using

a sample of random data to approximate or re-create a thought pattern and make a decision.

4.1.3 Security

Information security deals with the ability of a computer to protect data and hide patterns. The more random a stream of data, the more impossible it is to predict. No computing power has the ability to predict the next 'bit' in a sequence of random bits (Ruelle, 1991). Although transmitted data is always random to the receiver, the patterns that are deduced by the receiver are what they perceive as useful information. This is the information that must be protected. If an eavesdropper were to access transmitted data and recognize patterns, the system's security is compromised. Therefore, the more random the transmitted data is, the less likely it is to be recognized by unwanted receivers.

Steganography is a technology that embeds information in already noticeable patterns. Instead of garbling data so it is unreadable, steganography utilizes an existing pattern of information to hide another. For example, the pixels of an image can be slightly altered to contain additional information available to a party that knows it's there, yet unsuspecting parties just see a close approximation to the original picture. This is similar to the form of security that exists in monetary artifacts. A dollar bill has several 'hidden' features to ensure its authenticity, yet these features are only noticeable and recognizable to those who

are aware of it. Some features are noticeable by the human eye, others with a microscope, and even others that are invisible such as magnetic materials.

Securing data from unwanted recipients is important in computer security, but the Internet also requires a means to secure data from automated programs. Automated programs can take advantage of systems that are intended to register e-mail addresses and purchase tickets. These services provided through the Internet are intended for human use only, not SPAM e-mail or ticket scalping programs. Therefore, a reverse Turing Test must be used to prove if the user is human. This test is known as Captcha (Completely Automated Public Turing test to tell Computers and Humans Apart) ("Captcha", 2004). One common method used for Captcha implementations is randomness.

The Captcha concept is an exploitation of the realization that algorithms have a difficult time recognizing even the slightest distorted pattern. A typical implementation of Captcha creates a random amount of background noise to overlay onto an image of a word. The user is then asked to verify the word. The human senses can deal with distortion and 'see' through all the noise to reproduce the word. The noise acts as an adversary for programs that cannot decipher the pattern. Figure 22 shows a word that has been masked by a Captcha type process.

Figure 22. Captcha masked word.

For a pattern recognition algorithm to show a level of intelligence similar to humans, it must be able to solve this problem. Captcha shows that computers are very far from true human-like pattern recognition intelligence. This is especially true when non-textual patterns are distorted. The problem of identifying an image within a set of noise is much harder than text. For example, while it is difficult for an algorithm to recognize the distorted word ‘couch’, it is much harder for an algorithm to recognize a distorted image of a man sitting on a couch and output the description: ‘man sitting on couch’. This adversary condition also leads to the idea of presenting a series of distorted text or images to the user and have them solve a simple mind teaser puzzle. For example, the reverse Turing test could display Figure 23 and expect the user to type the word ‘beautiful’. The difficulty involved here for pattern recognition algorithms could be orders of magnitude higher than simply recognizing text and images.

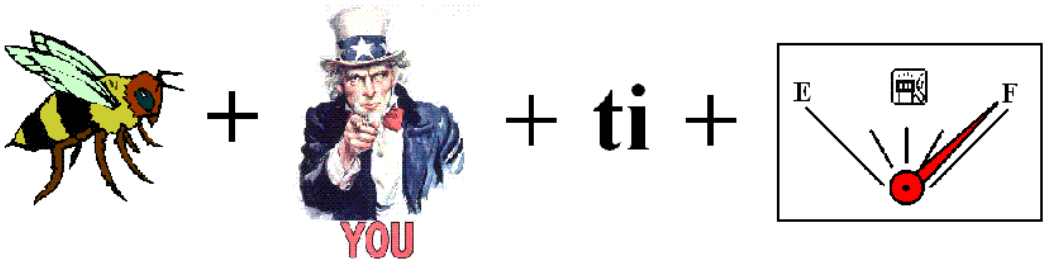


Figure 23. Beautiful.

Breaking the Captcha concept is not an easy task. Certain built-in accidental pitfalls help the breaking process, such as using common words that can be guessed, but Captcha was designed to create a test that it could not pass itself. This irreversibility makes for difficult problem solving that, instead of devising clever ways to create adversaries, deals more with uncovering the mysteries of artificial intelligence ("Captcha", 2004). Intuitive methods of solving such word puzzles involve some level randomness to avoid adversaries and deadlock. Random sampling and developing probabilities of correctness are examples of candidate methods for recognizing distorted patterns.

4.1.4 Word Problems

One of the main communication methods between humans is language. Therefore, the words we use are of utmost importance to express ideas. Pattern recognizing a set of words is one of the most important intellectual properties of humans. Word games are played to challenge the mind and, although many languages are redundant, mastering them is a difficult task to perform. Many

phrases and sentences are written and/or spoken with more information than is needed to comprehend the expression. According to Claude Shannon's Information Theory, the redundancy of English is about 50% (Shannon, 1948). This helps the mind that has to multitask and share computation power with other processes, so a missed word here or there will not take too much away from the meaning. Only a subset of information is actually perceived and understood. For example, consider the following sentence: *It is cold outside and I want to go play football with the neighbors*. The sentence could really be arranged to express similar meaning in much less words: *Cold out, want play football with neighbors*. The sentence could also be recognized by the typical human eye while missing data: *It is cld outsde nd I wnt to go ply footbal wth the neghbors*.

Word problems like the above use randomness to play with the redundancy of language and the ability of human intelligence to pattern recognize through noise and missing data. As Shannon (1948) stated, any two-dimensional array of letters in a language with 0% redundancy produces a complete crossword puzzle. The same is true for a word-find puzzle; yet there is no search to be performed since every combination of characters would be a valid word. These word games would not be as interesting since they contain no uncertainty.

E-mail known as SPAM is becoming a large problem over the Internet because it takes up valuable memory space and disturbs unwanted recipients. As SPAM filters are being developed, there are systems creating SPAM that can

quickly learn to create an adversary to fool the filter and bypass it successfully. The use of randomness to foil adversaries is an advantage that can be used to develop SPAM filters. Therefore, a SPAM creating system will not know the exact method used to identify e-mail as SPAM. For example, instead of a deterministic algorithm looking for words such as "viagra" or "debt consolidation", a random algorithm similar to the random searching of the word-find problem in Section 3.3 can be used to find hidden words or phrases or even contextual meanings that recipients do not wish to view.

4.2 Mind Simulation

Simulation and playing games are analogous to real life. Reality and life are a type of game in which a conscious being struggles and searches for meaning. Even if the world does not contain any inherent randomness, Chaos Theory shows that the massive amount of data perceived in the world is enough to affect some end result by stirring up variables with uncertainty. Players in any game must deal with uncertainty and use probabilities and random help to make decisions.

Random algorithms are useful for dealing with problems that the mind has to approximate because of the uncertainty of the environment, or more specific, the lack of understanding of the environment. The ability of random algorithms to deal with uncertainty and still make rational decisions allows them to simulate the intuitive processes that the mind performs.

The mind must deal with many uncertainties in order to make decisions. In estimating the trajectory of a tennis serve, the mind must process the behavior of the ball in such a way to stimulate a reaction and return it. The velocity of the ball is so fast that the mind must either attempt to slow down time and analyze every three dimensional position at intervals in time, or the mind could estimate the trajectory based on initial position, wind, and other factors. The player must also estimate what the results will be in some error deviation range. Too hard of a return will render the ball out of bounds, while too soft will hit the net. A reasonable return is the result of the analysis of many uncertain parameters used to place the ball in an approximate target area in order to provide the opponent with seemingly random information to hide patterns that could be exploited to win the game. The mind processes this analysis and makes complex decisions in the blink of an eye.

4.2.1 Human-like Artificial Intelligence

The brain's ability to develop scientific and mathematical knowledge does not seem to have been an evolutionary artifact (Ruelle, 1991). The brain does not keep time, memorize a mass of information, or perform mathematics very well. Instead, the brain is better suited to devise strategies to gather, fight, and hunt. However, the brain does comprehend mathematics, logic, and computing, and can build machines that could possibly surpass it in power. Although we do not understand the reason we are able to discover mathematical truth, and although

Gödel's theorem does not guarantee we will find solutions, we continue to work on and solve problems (Ruelle, 1991). This includes the problem of artificial intelligence.

One branch of artificial intelligence aims to produce computing machines that exhibit human-like qualities. This may be the most understandable view of artificial intelligence since human intelligence is all that is known from the perspective of our species. Unlike the application of the rules of logic, humans make irrational decisions. Human-like artificial intelligence research questions if irrational decisions are reasonable enough to exhibit repeatable, reliable intelligence. This thesis shows that randomness can be used in algorithms to simulate processes and make intelligent decisions. Randomness may be most applicable to the thought processes that are executed in the mind to make decisions and solve problems. Simulation of mind processes can help understand how and why decisions are made.

4.2.2 Decision Making, Playing Games

It is assumed that making useful decisions is the ultimate expression of intelligence. Random algorithms make choices somewhat erratically by depending on the uncertainty of a random draw. As seen in game theory, in an environment of uncertainty, this strategy is not irrational.

Simulation and numerical probabilistic algorithms use probability theory to make decisions and narrow in on some expectation value. The decisions that the mind makes are similar in nature. The mind could continually simulate a decision before arriving at an average or preferred conclusion. Some decisions, like Monte Carlo algorithms, run the chance of producing incorrect conclusions. Other decisions can be formulated using a type of Las Vegas algorithm where the solution can always be correct, yet takes a varying amount of time to compute (think of). For example, when searching for the television remote, the path of decisions could be haphazard, but always lead to successfully finding it.

Imagine the process of making a decision as passing through a graph of mental states to arrive at a conclusion. The steps involved with navigating these states are probabilistic in nature because decisions depend on many factors that can dynamically change. Therefore, the process of making a decision is similar to a random walk on the graph. Many random walks will converge to an average result and patterns will emerge (do not touch a hot stove). However, single random walks are unpredictable and can cause decisions to seem erratic. It is logical to conclude that the mind uses similar constructs to random algorithms, and there may in fact be a built in random generator to drive the system.

4.2.3 Personality and Behavior

Personality and behavior are ignored qualities when it comes to computing languages and logic. Without these traits, programmed machines are not able to

express the emotional side of man. Machines that express emotion will be more relatable and understandable. They will be able to use feeling to describe environments in which humanity does not have to personally witness to understand. Randomness provides a certain level of variation that is more relatable as opposed to deterministic, slave-like cyborgs.

A deterministic algorithm that continually performs the same operation with no variation is seen as 'dumb'. Using the analogy of weapons, 'smart' bombs are able to react while on the way to the target and display some sense of variation. They adapt based on the environment. 'Dumb' bombs follow the deterministic path of a projectile that is unchangeable and predictable once released from the host. Adversaries such as weather can cause the 'dumb' bomb to miss the target because it cannot adapt and change its path. To create a program that performs in a more human-like manner, the designer would have to include factors that cause its output, or behavior, to be unpredictable, yet adaptive and rational.

A 1969 article titled "The Art of Using Computers to Model Human Personality" (Dorf, 1974) is one of the many sources that express a need for machines to interact with humans more naturally. Because humans are influenced by other conscious beings that they come in contact with, a machine with a personality could have the same effect. Like intelligence, the definition of personality is not specific enough to model perfectly. Therefore, a computer that models personality can only be an approximation measured by the expression of its

behavior, and the reactions to it. The problem lies in the machine's ability to predict which actions and behaviors are appropriate for a personality in an approximated environment. This is a subset of general artificial intelligence where the machine tries to exhibit behavior patterns that are intelligent. Personality modeling involves three main areas of research with a fourth that has still yet to be discovered (Dorf, 1974). First, task-oriented models combine subcomponents of human personality, such as games and music, and aim to build systems that are highly specialized in many different areas. The problem lies in the complexity of the interface between components. Processing cycles and memory units are quickly occupied. Second, heuristic programming is used as a mechanism for machines to learn. Third, intelligent processes such as pattern recognition, inference, and hypothesis formation are grouped in a category labeled 'simulation of concept formation'. Finally, the fourth and still undiscovered concept to artificial intelligence and personality modeling is a new way of thinking and problem solving to provide solutions to the problems that were originally thought to be simple, yet turned out to be highly complex. It is unknown if we will ever have the knowledge to program the uncertainty of human behavior.

4.2.4 Agent-based Modeling

Agent-based modeling is a paradigm that measures complex systems by extracting patterns from the interaction of chaotic events. Behaviors of agents are programmed to simulate personalities that reflect research and experience. The

agents learn and use knowledge to achieve some goal. Ant behavior can be analyzed by measuring the interactions within a colony. Terrorist behavior can be analyzed by measuring the interactions with society.

Randomness can be programmed into agent-based modeling, and is also inherent based on the unpredictable interactions. An agent can base its actions from a random draw. The behavior of a group of agents is unpredictable and emerges from cooperative interactions much like the emergence of patterns in a random walk. The emergence of patterns from randomly interacting agents is used to understand the expected behavior of the entire population.

4.2.5 Genetic Algorithms

Like algorithms that simulate the processes of the mind, genetic algorithms simulate the processes of natural evolution to solve problems. Random methods are inherent to evolution because so much is unknown about the process. It is known that during reproduction, parent genes recombine to form a new chromosome. During copying of parent genes, random errors occur in the form of mutation. The new cell then goes on to survive and reproduce like its predecessors. According to natural selection, the fittest members survive. The term ‘fittest’ is an approximation in that surviving members could be ’lucky’ and not necessarily the optimal choices for continuation. This built-in anomaly allows for diversity, and a diverse population adapts better to the environment. As a whole, the population evolves a solution and finds purpose in an uncertain world.

Genetic algorithms are built to exploit the properties of natural evolution, using randomness as an advantage. Instead of using randomness as a haphazard method of guessing or searching, genetic algorithms use it to build on previous knowledge. Randomness introduces variety and leads the algorithm to a solution, adapting its behavior along the way.

Genetic algorithms simulate the reproduction process by first starting with a population of possible solutions (members). The fitness of each member is calculated using a 'fitness function'. This function ranks the members relative to each other, according to their ability to survive. Members are chosen based on their ranking and placed into the mating pool. Members that are more fit are more likely to be chosen for the mating pool. The crossover process then chooses two members from the mating pool and randomly determines if recombination should take place. If so, the members are spliced and combined in some manner, producing new members that are added to the population. If crossover is not performed, then the members are directly copied to the new population. This continues until a new population has formed. The ranking, selecting, and combining process continues until the best (most fit) solutions are identified in the midst of noise. Another source of noise, in addition to the selection and combination methods, is the process of mutation. Mutation randomly changes parts of a member somewhere in the process. Like natural mutation, this introduces diversity to a population and avoids deadlock or stagnant situations.

The flexibility and strength of genetic algorithms is a result of the emergence of 'fit' members. This occurs because the features that a fit member exhibits have a good chance of surviving through bits of randomness used to weed out less-fit features. Although crossover, splicing, and mutation could modify them, the features are able to survive throughout the process. This is the reason that crossover should not always occur, and mutation rate should be low. Too much modification may lead to the changing or destruction of strong features.

Randomness plays a crucial role in the performance of genetic algorithms. Random decisions are made to select, crossover, and mutate members. Cantú-Paz (2004) shows that using a pseudorandom number generator that contains minor variation can produce large deviations in algorithm performance. Cantú-Paz (2004) also compares the results of using true random numbers and a "good" pseudorandom number generator and finds no difference in performance. Accurate results from the simulation of the reproduction process are a result of a suitable source of randomness that does not affect the process.

4.3 Others

Other than pattern recognition and mind simulation, the application of random algorithms is unlimited because of the similarities between their behavior and the behavior of the universe. So many processes remain a mystery. Therefore, while we search for acceptable explanations, we must utilize randomness to provide

a best-guess approximation. This is the case for physics, biology, astronomy, etc. In physics, the Uncertainty Principle limits knowledge of the microscopic. In biology, the mysteries of life remain unbounded. In astronomy, we can only imagine the structure of dark matter, while the randomness of the stars is organized into constellation patterns. While randomness in machines can be used to approximate and analyze our world, it must be used wisely when presenting output to the users, and creators.

4.3.1 Physics, Quantum Mechanics, Genetics

Uncertain environments make for a good application of random algorithms. Randomness in computing plays a role in physics and genetics since these fields work directly with the uncertainty of the universe.

The field of quantum mechanics has shown that the study of very small objects creates an uncertainty in which probabilities must be used to describe. Known as the Uncertainty Principle discovered by Heisenberg, the position and velocity of a particle cannot be known with complete certainty simultaneously.

The process of genetics, reproduction, and life are other fascinating examples of inherent randomness. Genetic algorithms show how randomness can help to evolve solutions. Sexual reproduction shows that there is some regularity in the universe since genes are able to combine and create new life (Ruelle, 1991). Yet, the recombination of genes is not perfect. The path of life contains much

uncertainty. Less fit individuals do not survive, yet some do because they are 'lucky'. The path is assumed to not be the result of a deterministic and predictable set of actions. For example, in a typical video game, a character's path is determined by a set of button presses. With no image displayed on the screen, a player that has memorized the timing of a deterministic game can produce the same results by following a script of button presses. The process of reproduction and life are assumed to contain much randomness, of which random algorithms can help approximate.

Computing in these fields must not be limited to traditional deterministic methods involving mass storage and exhaustive search techniques. The available data is much too large and the underlying rules of these systems are unknown. Therefore, using the lessons of game theory, rational techniques used to understand and model such systems should be probabilistic.

4.3.2 Human Computer Interface

The obvious dilemma when using randomness in any system is the user interaction. Living beings are able to interact with the universe because, although everything is random and unique within it, intelligence is able to make sense of it through patterns. A system with randomness must be governed by a set of rules that produce output to the user in recognizable form. While every user is different in their comprehension, there is a level of commonality that they are expected to have. For example, watching a television screen with static is not enjoyable

because the random black and white pixels do not contain relatable information. Watching a television program is enjoyable because the characters and scenery, albeit randomly transmitted to aural and visual senses, are physically and emotionally recognizable. Cartoons are more relatable to children, while adults understand and relate to drama programs.

A computing machine that uses randomness can very easily confuse the user. If the user is expecting a set of menu options, and a complete random draw is used to determine which ones are available, the user will quickly get discouraged. However, probabilistic techniques can be used to determine the frequency of used menu options, and provide the user with an approximated set of options that they wish to see and are predicted to use.

Current technologies that contain uncertainty test the limits of human frustration. An Internet surfer that witnesses random page errors while visiting a site is quickly turned off. For example, a banking site that incorrectly manages cookies and causes arbitrary page errors during the transfer of funds causes the web surfer much grief in understanding the success or failure of a transaction. Or, an Internet surfer may happen upon a randomly generated advertisement, and follow the link. Upon a browser crash, the surfer is unable to navigate back to the link since it is now a different random permutation. Uncertain behavior is frustrating and confusing to the user. An operating system that is not stable or does not produce consistent feedback is not user-friendly. For example, if the behavior of an

operating system while deleting a file is non-deterministic, the user becomes frustrated because they do not know how long they must wait for the task to succeed. Also, in the case of the previously mentioned Captcha technique, if the verification image is so distorted that it is unreadable to even a human, then the user becomes frustrated with the system because they are unable to prove their consciousness! These scenarios show that there is certain levels of computing that are expected to be deterministic, while others, such as a programmed personality, are assumed to be acceptable if they contain randomness.

Chapter 5: Conclusion and Suggestions for Future Work

5.1 Problems

The use of randomness to accomplish artificially intelligent machines is arguable since random algorithms are ultimately executed on a deterministic computation device. However, this determinism does not impair the uncertainty that the algorithms exhibit. In Chaos Theory, deterministic equations are used to produce chaotic results. Fractal images that display similar complexity to natural forms are evolved from determinism, and the boundary of turbulence can be simulated with deterministic equations like those of the Lorenz butterfly.

Michael Barnsley, the developer of the technique known as the Chaos Game used to draw natural, fractal-like images using randomness admitted, "Randomness is a red herring." (Gleick, 1987, p. 239). In many situations, randomness is simply a tool that attempts to approximate a result that already exists, and can be deterministically found. Like mathematics, the result is not invented, but discovered. Random algorithms seek to discover solutions faster and more intuitively than deterministic methods.

In the field of artificial intelligence, randomness does not provide a machine with understanding or consciousness any more than is available from pure determinism. While the output and behavior of the machine is more flexible and the power of creativity is expressed, the features of a conscious being are not

available. It is assumed that no one feature is expected to result in a conscious machine. A random number generator or the injection of an unknown sequence of bits does not satisfy the requirements of a conscious machine, nor does the combination of randomness and logical statements. The features and properties of the combination of randomness and algorithms do provide the building blocks for intelligent activities. The problem of artificially supplying a programmable entity with the qualities exhibited by a natural, intelligent being is to be continued. The combination of intuitive logic, intelligent processing, and complex problem solving, along with the flexibility and creativity of randomness, are assumed to play a vital role in the discovery.

5.2 Recommendations

Recommendations for continued research in the field of random algorithms and artificial intelligence involve the core concepts introduced in this paper, and others that have not been discussed. Parallel and distributed computing technologies fit well with the multitasking nature of the mind. Randomness in these fields will help uncover how the mind deals with noise and learns with uncertainty. It is recommended that randomness be controlled to perform useful tasks. Noise in a neural network or genetic algorithm helps to introduce variation, but too much uncertainty leads to erratic results, like that of a ‘snowy’ television screen. Large problem spaces are recommended for further research. Some examples in this paper showed useful random algorithmic solutions with bounds on

the problem space. When the search space gets large with few solution witnesses, or the accuracy of an approximation needs to be precise, random methods begin performing poorly. Improvements to the word-find problem in Section 3.3 are recommended to place tighter bounds on performance through search strategies or variations in pattern recognition.

5.3 Conclusions

This thesis has shown a summary of popular random algorithms that perform intelligent tasks and behave in a more flexible manner than deterministic algorithms. The random techniques used apply to artificial intelligence and the interaction between man and machine. Algorithms are built to simulate the processes and decisions of the mind, and are then analyzed to relate their behavior back to the assumed workings of the mind. While random algorithmic analysis focuses on pure mathematics, the results do not fully express their ultimate applications. Random algorithms have more flexibility and more creativity than standard logic statements and deterministic algorithms. The originality that is supplied by random algorithms provides the ability to create and simulate intelligent processes and lead to a new way of thinking about building intelligent machines.

If everything in the universe is unique, then everything perceived has a random nature and is pure information according to Claude Shannon and

Information Theory. If everything is pure information with no redundancy, then intelligent beings need some mechanism to classify common patterns and recognize those patterns. One method to begin classifying patterns is to use a random technique. Random techniques help to avoid deadlock (where to start or other confusion), foil adversaries (blockage or very similar patterns), and allow room for more intelligence and creativity (new thoughts, new ideas, new reasoning). Thus, for a machine to be intelligent, it should use random techniques in its algorithmic processes.

If used to search, random algorithms are good for eliminating adversaries and sampling a population fairly. If used to decide, random algorithms are good for producing estimations and approximations within reasonable bounds. If used to adapt, random algorithms can continually change their behavior. If used for creativity, random algorithms are able to exhibit unpredictability.

A player in a game is confronted with uncertainty and must decipher and translate it into patterns to express intelligence. In a word-find puzzle, the player must search through random letters and find recognizable patterns. In a baseball game, the player must hit a randomly thrown pitch and the fielders must interpret a randomly hit ball. In surfing, the wave-rider must continually deal with the uncertainties of the water to exhibit grace and style. To translate uncertainty into patterns, and be able to recognize and learn from patterns, is an ultimate expression of intelligence.

The goal of Artificial Intelligence is to simulate human mind processes such as: learning, rationalizing, recognizing patterns, and decision-making. By definition, the word simulation equates to approximation. Artificial Intelligence is therefore an approximation to the mind and what humanity considers rational thinking. The study of Artificial Intelligence is bounded by approximations because we simply do not know how the mind works. We cannot view the rule set that allows us to learn sheet music, recognize our parent's faces, or decide what to have for breakfast. These background processes and algorithms are hidden from our conscious understanding. Whether this is planned or just a chaotic result of the system (life), we may never know.

Like all problems, the study of Artificial Intelligence is bounded by uncertainty. Science and research in this field must continue because, although brain processes are unknown, it has not been proven impossible to create an artificially conscious machine that expresses intelligence. In this sense, the pursuit of 'strong' artificial intelligence is analogous to the search for extra-terrestrial life. For all we currently know, it may not exist in the universe. However, we continue our search because there is nothing that tells us that it CANNOT exist. If it were proven that 'strong' artificial intelligence is impossible, then the problem would not exist in the first place. For both problems, there is even plenty of evidence to suggest otherwise. Water is known to exist all over the universe, and water is needed for life. Calculations and computations occur in the human mind and allow

for tasks to be accomplished. Unlike the search for aliens however, the problem of artificial intelligence and the theory of the computational machine exist because we are in search of ourselves.

The correlation between the human mind and artificial intelligence should be studied with caution. The current direction of man and machine has not changed a great deal from the original creation of the Turing Machine and Babbage's Analytical Engine. The development of computers and algorithms has progressed a great deal, but many capabilities still require the logic and reasoning of the human mind. For example, a human is still needed to provide verification of an algorithm to perform fingerprint pattern recognition. A computer algorithm has the capability to rapidly store and search fingerprints, but can only output best guess "scores" when searching for matches. The machine has no conscious self and thus can make no sense of its own 1's and 0's. The machine's human operator must be the master, and deduce the output.

The study of artificial intelligence must not neglect the aspects of humanity and rationalism that involve consciousness, emotion, and the self. The study of computer science is constantly testing technologies to provide insight into these constructs. Random algorithms are one of the many technologies that can continue to lead to the path of the truly interactive man and machine.

List of References

"approximation algorithm." NIST Dictionary of Algorithms and Data Structures. Online. 11 Nov. 2004. http://www.nist.gov/dads/HTML/approximatin.html

Brassard, Gilles, and Paul Bratley. Fundamentals of Algorithmics. New Jersey: Prentice Hall, 1996.

Cantú-Paz, Erick. "On Random Numbers and the Performance of Genetic Algorithms." Online. 11 Nov. 2004. http://www.llnl.gov/casc/sapphire/pubs/146850.pdf

"Captcha". Wikipedia. Online. 11 Nov 2004. http://en.wikipedia.org/wiki/Captcha

Chlebus, Bogdan S., and Dariusz R. Kowalski. "Randomization Helps to Perform Independent Tasks Reliably" Random Structures and Algorithms 24.1 (2004): 11-41.

Cormen, Thomas H., et al. Introduction to Algorithms. Cambridge, Massachusetts: The MIT Press, 2001.

Czumaj, Artur, and Volker Stemann. "Randomized Allocation Processes" Random Structures and Algorithms 18.4 (2001): 297-331

Dorf, Richard C. Computers and Man. San Francisco: Boyd & Fraser Publishing Company, 1974.

Eastlake, D., S. Crocker, and J. Schiller. "Randomness Recommendations for Security." IETF – Request For Comments (Dec. 1994). Online. 11 Nov. 2004. http://www.ietf.org/rfc/rfc1750.txt

Feige, Uriel, and Yuri Rabinovich. "Deterministic Approximation of the Cover Time" Random Structures and Algorithms 23.1 (2003): 1-22.

Gleick, James. Chaos. Making A New Science. New York: Penguin Books, 1987.

Liu, Baoding. Uncertain programming. New York: John Wiley & Sons, 1999.

Motwani, Rajeev, and Prabhakar Raghavan. Randomized Algorithms. United Kingdom: Cambridge UP, 1995.

"nondeterministic Turing machine" NIST Dictionary of Algorithms and Data Structures. Online. 11 Nov. 2004. http://www.nist.gov/dads/HTML/nondetermTuringMach.html

"probabilistic Turing machine." NIST Dictionary of Algorithms and Data Structures. Online. 11 Nov. 2004. http://www.nist.gov/dads/HTML/probablturng.html

Ruelle, David. Chance and Chaos. Princeton, N.J.: Princeton University Press, 1991.

Shannon, Claude E. "A Mathematical Theory of Communication." The Bell Systems Technical Journal. 27 (1948): 379-423, 623-656. Online. 11 Nov. 2004. http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

Weisstein, Eric W. "Kolmogorov-Smirnov Test." MathWorld-A Wolfram Web Resource. Online. 11 Nov. 2004. http://mathworld.wolfram.com/Kolmogorov-SmirnovTest.html

Weisstein, Eric W. "Noise Sphere." MathWorld-A Wolfram Web Resource. Online. 11 Nov. 2004. http://mathworld.wolfram.com/NoiseSphere.html

Weisstein, Eric W. "Prisoner's Dilemma." MathWorld-A Wolfram Web Resource. Online. 11 Nov. 2004. http://mathworld.wolfram.com/PrisonersDilemma.html

Whitney, Charles Allen. Random Processes in Physical Systems : an Introduction to Probability-based Computer Simulations. New York: Wiley, 1990.