Building Maze Solutions with Computational Dreaming

Scott M. Jackson

Thesis submitted to the faculty of the Virginia Polytechnic Institute and
State University in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

JoAnn M. Paul

Luiz A. DaSilva

Wen-Li Wang

July 15, 2014

Alexandria, VA

Keywords: Parallel computing, computational dreaming, MISD, bio-inspired
computer architecture

Copyright © 2014 Scott Jackson

Building Maze Solutions with Computational Dreaming

Scott M. Jackson

Abstract

Modern parallel computing techniques are subject to poor scalability. Their performance tends to suffer diminishing returns and even losses with increasing parallelism. Some methods of intelligent computing, such as neural networks and genetic algorithms, lend themselves well to massively parallel systems but come with other drawbacks that can limit their usefulness such as the requirement of a training phase and/or sensitivity to randomness. This thesis investigates the feasibility of a novel method of intelligent parallel computing by implementing a true multiple instruction stream, single data stream (MISD) computing system that is theoretically nearly perfectly scalable. Computational dreaming (CD) is inspired by the structure and dreaming process of the human brain. It examines previously observed input data during a “dream phase” and is able to develop and select a simplified model to use during the day phase of computation. Using mazes as an example problem space, a CD simulator is developed and successfully used to demonstrate the viability and robustness of CD. Experiments that focused on CD viability resulted in the CD system solving 15% of mazes (ranging from small and simple to large and complex) compared with 2.2% solved by random model selection. Results also showed that approximately 50% of successful solutions generated match up with those that would be generated by algorithms such as depth first search and Dijkstra’s algorithm. Experiments focusing on robustness performed repeated trials with identical parameters. Results demonstrated that CD is capable of achieving this result consistently, solving over 32% of mazes across 10 trials compared to only 3.6% solved by random model selection. A significant finding is that CD does not get stuck on local minima, always converging on a solution model. Thus, CD has the potential to enable significant contributions to computing by potentially finding elegant solutions to, for example, NP-hard or previously intractable problems.

Acknowledgements

I would like to thank my advisor Dr. JoAnn Paul for her seemingly endless patience, support, and advice throughout my graduate studies. I would also like to offer special thanks to my wife, Amber, for her patience and support and to our cats for their “help” in typing this thesis. Thank you also to Dr. Luiz DaSilva and Dr. Wen Li Wang for serving on my committee. Finally thank you to my friends, parents, sister, and other family members for your support through the years.

This research was supported in part by the National Science Foundation under grant 1043341. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Table of Contents

Abstract.....	ii
Acknowledgements.....	iii
List of Figures	vi
List of Tables	vii
Chapter 1 Introduction	1
1.1 Thesis Organization.....	3
Chapter 2 Background	4
2.1 Dreaming and the Brain	4
2.2 Parallel Computing.....	5
2.2.1 Parallel Architecture Classification	5
2.2.2 Maximum Speedup	9
2.2.3 Hardware Parallelism	12
2.2.4 Software Parallelism	12
2.2.5 Scalability	14
2.3 Artificial Intelligence	15
2.3.1 Software-based AI.....	16
2.3.2 Hardware-based AI	17
2.3.2.1 SpiNNaker	17
2.3.2.2 BrainScaleS.....	19
2.4 Maze Solving Overview	19
2.5 Summary	24
Chapter 3 Computational Dreaming.....	25
3.1 An Overview of Computational Dreaming	25
3.2 The Phases of Computational Dreaming	29
3.3 Challenges	31
3.4 An Overview of Maze Solving with Computational Dreaming.....	33
3.4.1 Solution Models	34
3.5 Limitations.....	36
3.6 Summary	37
Chapter 4 A Computational Dreaming Simulator	38
4.1 Simulator Architecture.....	39
4.2 Maze Generation	42
4.3 Solution Models	43
4.4 Solving Mazes with CD	46
4.4.1 Measuring Progress	49
4.4.2 Gathering Data	50
4.5 Simulation Parameters.....	52
4.6 Experimental Overview	53
Chapter 5 Goals.....	55
Chapter 6 Experimental Setup.....	58
6.1 Viability	58
6.2 Robustness	61
6.3 Flexibility	62
Chapter 7 Experimental Results.....	66
7.1 Viability	66

7.2	Robustness	70
7.3	Flexibility	74
7.4	Summary of Experimental Results	76
Chapter 8	Conclusions and Future Work	77
8.1	Future Work	78
Chapter 9	References	80
Appendix A	Glossary of Terms.....	83
Appendix B	Simulator Parameter Listing.....	84

List of Figures

Figure 1: SISD Processing	6
Figure 2: SIMD Processing.....	6
Figure 3: MISD Processing.....	7
Figure 4: MIMD Processing	7
Figure 5: Speedup achieved based on Amdahl's law.....	10
Figure 6: Speedup achieved based on Gustafson's law.	11
Figure 7: An example of a sparse direction map with the goal in the center of the maze. Adapted from [34].....	22
Figure 8: A representation of the maze is shown at left along with start (S) and goal (G) locations. To the right is the starting point for the flood fill algorithm. Adapted from [35].....	23
Figure 9: Depiction of the flow of information in dreaming and day phases. Adapted from figures in [13].	27
Figure 10: A simplified example of how higher layers contain more complex models by combining lower layers in different ways.....	28
Figure 11: Day phase data flow. Adapted from a figure in [13].....	30
Figure 12: Dream phase data flow. Adapted from a figure in [13].	31
Figure 13: Example of the first two layers of maze solving models.....	34
Figure 14: Maze solved by a model with steps: S, E, S, E, N, E, S, E. White cells are those visited by the agent. Cells shaded light gray are those observed, but not visited. Cells shaded dark gray are unknown to the agent. The upper-left cell with an S is the start point while the lower-right cell with the G is the exit or goal.	35
Figure 15: High level simulator architecture.....	41
Figure 16: Example of a maze generated by the CD maze generator	43
Figure 17: Layer 0 models	45
Figure 18: Layer 1 models	45
Figure 19: Partial example of how layer 0 models combine to form layer 1 models.	45
Figure 20: Example layer 0 and layer 1, showing how layer 1 models are constructed from those on layer 0.	46
Figure 21: Data flow within the CD portion of the simulator. Dashed lines represent the flow of information during a dream phase while solid lines represent data flow during a day phase.	47
Figure 22: Maze from Figure 16 solved by a model with steps: S, E, S, E, N, E, S, E. Cells shaded white are those visited by the agent. Cells shaded light gray are those observed, but not visited. Cells shaded dark gray are unknown to the agent.	51
Figure 23: Observed portion of maze stored by agent for use while dreaming.....	52
Figure 24: High-level view of experimental setup.	58
Figure 25: Conceptual viability experiment results	59
Figure 26: Expected robustness experimental results.....	62
Figure 27: Depiction of expected results. Simpler mazes and more data are expected to yield higher success rates. Number of mazes processed increases from left to right. Success rate increases from bottom to top.....	64
Figure 28: Viability test results.....	67
Figure 29: Robustness test results	71
Figure 30: Graphical representation of varied input parameter experiment.....	74

List of Tables

Table 1: Flynn's Taxonomy.....	6
Table 2: Robustness experimental simulation parameters	61
Table 3: Simulation parameters by trial.....	65
Table 4: Viability test results summary.....	67
Table 5: Summary of models selected by CD during viability trials and their success rates. Gray cells indicate that a model was not selected at all during the respective trial.	68
Table 6: Summary of solutions found by CD and how they compare to known maze-solving algorithms. The total number of mazes solved during this experiment was 480.....	69
Table 7: Robustness results summary	70
Table 8: Summary of models selected by CD during robustness trials and their success rates. Gray cells indicate that a model was not selected at all during the respective trial.	73
Table 9: Parameter Range Test Results Summary	75

Chapter 1 Introduction

Over the past decade, multicore processors have become widely available at a relatively low price and grown to dominate the commercial marketplace. As a result, commercial users have been able to take advantage of hardware parallelism that allows multiple tasks to run in a truly simultaneous fashion. While previous advancements in clock speed and pipelining techniques brought steady improvements to processor performance, such gains began to slow in 2002 and almost completely leveled off by 2008 [1, 2] bringing a focus back to explicit forms of parallelism. Over many decades, much research has been done on parallel computation and how best to take advantage of parallel processors and overcome the challenges associated with parallel computing. These challenges include limitations in performance gains due to the amount of code that must be executed serially [1, 3, 4], memory and bus bandwidth [5, 6], determining which portions can and should be executed in parallel, and the complexity of developing a parallel application. Another significant challenge comes in the form of scalability: how well an application is able to take advantage of additional processing resources in order to improve its performance. Amdahl states that the potential for performance improvement through the addition of processors is limited by the portion of an application that must be executed serially [3]. In contrast, Gustafson argues that additional parallel computing resources offer performance improvements by allowing larger workloads to be processed in the same amount of time [4]. This implies that as the parallel capacity of a system is scaled up (by adding more processors) that the size of the workload it can handle with no loss of performance scales up as well, seemingly without bound.

Modern multicore computing systems can process billions of operations per second, performing tasks that are too complex for humans to do in the blink of an eye. Even so, functions that are simple for biological computers (i.e., brains) to accomplish have eluded their digital counterparts for decades. All computers, even the most powerful ones, have difficulty performing tasks that humans consider simple, such as reading and understanding a newspaper article or extending a (robotic) limb to pet a cat [7]. In order for computer systems to hope to match the performance of the brain, novel approaches to scalability must be found. The thesis of this work is that novel methods of partitioning problems must be developed. Further, their execution time must remain constant (or nearly constant) as complexity increases in order to remain both manageable and feasible given the current state of computing. This thesis

attempts to do just that by taking inspiration from the human brain. Research indicates that the brain is made up of many neurons that all perform the same basic function [8]. The specific role that a given neuron plays (e.g., speech, facial recognition, vision) is determined by how it is connected to other neurons [8, 9]. Other research suggests a link between dreaming [10], or at least sleeping [11, 12], and improved problem solving. This concept is applied to computing and tested via a selection-based approach to multiple instruction stream, single data stream (MISD) computing referred to as Computational Dreaming (CD).

CD takes advantage of a brain-like architecture to improve computing performance by partitioning and simplifying the solution space. Its dream phase makes use of many relatively simple processing elements that are all capable of the same basic functions. These elements are combined into candidate “solution models” that represent a wide range of approaches to processing input data. The solution models grow in complexity but still remain far simpler than an approach that unifies all possible maze solving algorithms into a single parallel algorithm. This is achieved by two phases of processing. A dreaming phase feeds previously observed input data to all available solution models simultaneously. The output of each element is then analyzed and the best performing model is chosen for use during the subsequent day phase, during which real-world input data is processed and stored for use in the next dreaming phase. Thus CD, through dreaming, determines how best to make workloads smaller and simpler. Through the use of a MISD architecture CD offers not only an effective way to find solutions to problems, but also theoretically offers nearly perfect scalability. Here, scalability means that CD can take advantage of additional processing resources to solve more complex problems during dream phases with little to no performance penalty in real time. In doing so, it is able to minimize the amount of work required to process data during day phase. This does not, however, give any indication as to how well it processes data. Before scalability is examined, it is important to ensure that CD works at all.

The primary goal of this thesis is to demonstrate that CD is a viable method for problem solving and thus one that is worthy of further investigation. The robustness of CD is also investigated in order to show that it is capable of consistently finding success even when subjected to randomness. A simulator was constructed that uses CD to find solutions to mazes. Simulation parameters were varied to explore different aspects of the system’s performance including whether or not it can solve problems at all, the limitations of the system’s ability to solve problems in term of size and complexity, and how repeatable success is. As a result of this investigation, a primary contribution of this work is the finding that CD

does not get stuck and therefore does not require external intervention, unlike traditional Artificial Intelligence (AI) approaches.

1.1 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents background information on a number of topics relevant to this research. These topics include an overview of the organization and operation of the brain, the state of parallel computation and artificial intelligence, and an overview of existing methods of solving mazes. Chapter 3 presents computational dreaming. It discusses the system architecture and concept of operation, as well as several of the challenges that the developer of a CD system must deal with. Chapter 4 describes the simulator constructed for this research. It describes how input mazes and solution models are generated and used, simulation inputs and outputs, and how a user can modify the operation of the simulator. Chapter 5 discusses the goals of this research. Chapter 6 describes the setup for each experiment performed, while Chapter 7 includes presentation and analysis of results. Chapter 8 discusses the conclusions of this research and suggests topics for future research on CD.

Chapter 2 Background

This chapter presents discussions of several topics that are relevant to a discussion of CD. These include overviews of Dreaming, Parallel Computing, Artificial Intelligence and Maze Solving.

2.1 Dreaming and the Brain

The CD technique is inspired by the assumption that the physical structure of the brain, particularly its massively parallel nature, plays a major role in determining its performance and function. The brain is able to quickly and reliably recognize patterns and process abstract concepts [7, 9, 13]. This makes the brain an excellent example of how parallelism can enhance processing capability. If the same structural benefits can be applied to computing, it would mark a significant advancement in parallel processing. CD does just that using mazes as an example problem space. Experimental results suggest that it offers several benefits over other brain-inspired methods (neural networks), such as requiring no special training phase and little to no human oversight. Like the brain, it makes use of large scale parallelism and many instances of the same basic work units.

Researchers have observed that all areas of the brain's neocortex look structurally the same and theorized that they must all be performing the same basic operation [8, 9]. The implication, as Hawkins points out, is that what makes one area of the brain different from another is how its neurons are interconnected [9]. He sums this up saying, "If you connect regions of cortex together in a suitable hierarchy and provide a stream of input, it will learn about its environment [9]." One of the assumptions made by this thesis is that dreaming helps the brain to better organize what it has seen and use past experience to improve its own problem solving ability by developing the internal structure that is best able to solve the problem at hand. This suggests that dreaming is an important part of intelligence, one that might be leveraged to build more intelligent machines. In fact, research suggests that dreaming is something that many, if not all, animals experience [10, 14].

While the purpose of dreams is still unknown, there are several theories. One of these is that dreaming helps the brain to process and organize information as it is stored into memory. Louie and Wilson observed that neural activity in sleeping rats experiencing rapid eye movement (REM), which occurs while

dreaming, showed a correlation to neural activity while the rats were awake and running through a test track [10]. They theorized that dreaming has something to do with how the brain decides what is worth remembering, thus playing a role in how we learn [10].

Another study suggests that dreaming can help to prime the brain to solve problems. Cai et al. performed experiments where human subjects were given analogies to complete in addition to word association tests (known as a remote associates test or RAT). One RAT and one analogy session was given in the morning and one RAT in the evening [11]. In between tests, subjects either rested quietly, entered REM sleep¹, or slept without any REM activity [11]. Subjects who experienced REM sleep had a higher success rate when solving such problems in the evening compared to subjects who did not experience REM sleep [11]². Yet another experiment, performed by Wagner et al., similarly suggests that sleeping can improve problem solving capacity. Here, researchers trained subjects to perform an algorithm that transforms a string of numbers into another string [12]. The subjects who slept (and presumably dreamed) discovered a shortcut (intentionally hidden by researchers) more often than subjects who did not sleep [12]. These results suggest that sleeping, and potentially the dreaming process, are key to intelligence and may improve the brain's ability to solve problems.

2.2 Parallel Computing

Since the brain (a massively parallel system) is an inspiration for CD, a discussion of parallel computing is necessary. This section presents a brief discussion of parallel computing including current methods for both hardware and software parallelism. In doing so, the uniqueness of the CD approach is put into perspective.

2.2.1 Parallel Architecture Classification

Processor level parallelism can take on several different forms. Historically, processors have been classified in terms of their parallelism according to Flynn's taxonomy [15]. The taxonomy, shown in Table 1,

¹ REM (Rapid Eye Movement) sleep is a phase of sleep during which brain activity is high. It is the phase of sleep most commonly associated with dreaming.

² Interesting to note is that when different answers were required between morning and afternoon tests, those who experienced REM sleep performed no better or worse than those who did not. [11] D. J. Cai, S. A. Mednick, E. M. Harrison, J. C. Kanady, and S. C. Mednick, "REM, not incubation, improves creativity by priming associative networks," *Proceedings of the National Academy of Sciences*, vol. 106, pp. 10130-10134, 2009.

consists of four classes of computers, defined by the number of instruction and data streams available. These classes are useful not only for classifying, at a high level, what type of parallelism is being used, but also what types of parallelism are available for use.

Table 1: Flynn's Taxonomy

		Instruction	
		Single	Multiple
Data	Single	SISD	SIMD
	Multiple	MISD	MIMD

A SISD processor is one that operates on a single datum at any given time. As shown in Figure 1, only one instruction is issued at a time in a SISD machine. The processing unit, p , takes one instruction, i , and one datum, d , to produce result q . Examples include the early Intel 8088 microprocessor and modern microcontrollers such as the ARM9.

Figure 1: SISD Processing

Figure 2: SIMD Processing

SIMD processors are able to operate on multiple pieces of data with a single instruction. This allows a SIMD computer to perform many computations in parallel, limited by the fact that the computational steps are the same and only the data is different from processor to processor. SIMD computers exploit data-level parallelism [1] and thus are useful when doing work on vectors of data. For example, vector processors are SIMD machines [1]. Figure 2 shows a conceptual depiction of SIMD processing. There are n processing units and n data values. The same instruction is fed to all processing units at the same time to produce n results in parallel.

MIMD processors execute multiple, possibly different, instructions in parallel, each operating on different data. A MIMD processor is able to exploit thread (or process) level parallelism [1]. The various

threads being executed at any given moment are not necessarily synchronized or even related to the same program. Figure 4 shows a conceptual depiction of MIMD processing. There are n processing units, n data values, and n instructions. Each processor receives an instruction and a datum at the same time, allowing the entire system to produce n total results simultaneously and in parallel. The MIMD category spans a broad range of processing architectures and organizations. For example, one example of a MIMD processor is the Intel Core2 line which features up to four physical processor cores, able to execute multiple instructions (and indeed, multiple programs) simultaneously. All cores share a common main memory bank that is equally accessible to each core in terms of speed and latency. This architecture is often called uniform memory access (UMA) [1]. In contrast is a non-uniform memory access (NUMA) MIMD system in which each processor has its own local memory (not directly accessible by other processors) and must communicate with other processors via some sort of interconnection or network mechanism [1]. An example of this style would be a computing cluster such as a Beowulf cluster. MIMD architectures are very flexible and can be exploited to run many threads of a single application in parallel to increase performance, to execute multiple entirely separate programs simultaneously, or a combination of the two [1].

Figure 3: MISD Processing

Figure 4: MIMD Processing

Finally, MISD machines are essentially the inverse of SIMD machines as they use multiple, different instructions to operate on a common datum in parallel. While one may argue that pipelined architectures and/or fault tolerant systems may be classified as MISD [16], it is more commonly accepted that no commercial MISD systems have yet been constructed and put in to practical use [1] though there has been research exploring the benefits of this architecture [17]. Figure 3 shows a conceptual depiction of MISD processing with n processing units and n instructions. The same datum is passed to all the processing units at the same time as they compute n results in parallel. An MISD system designed for in-

creased performance would be capable of issuing multiple independent and heterogeneous instructions that operate on common data. In contrast, a fault tolerant system features multiple redundant (i.e., identical) units so that if one unit fails, the system can continue to operate. In these systems, all units are performing the same instructions on the same data. Thus while it *is*, in a sense, a MISD system, a fault tolerant system is not using the MISD architecture to achieve any performance gains or perform any useful work beyond what a single SISD machine is capable of. A pipelined processor issues multiple heterogeneous “instructions” on a single datum, but the instructions do not all operate on the same datum at the same time. Furthermore, the instructions operate on the datum in a serial manner; for example the “fetch” stage must complete before the “decode” stage may begin. In contrast, based on Mountcastle’s theories [8], it could be said that the brain operates many MISD systems in parallel (e.g., for sight, hearing, taste, etc.). The CD technique explored by this thesis takes this as inspiration and constructs a true MISD machine whereby multiple independent and heterogeneous instructions operate on common data.

The MISD architecture allows a CD system to evaluate many (and, ideally, *all*) potential solutions to a given data processing problem in parallel. This evaluation takes place such that all processing units finish their job at more or less the same time with minimal overhead. The effect is that even as more processing nodes are added to the system (to handle more solution models that are potentially more complex) that they can all still be evaluated simultaneously and in roughly the same amount of time³. At the end of this evaluation process, which is referred to as the dream phase, a “best solution model” is chosen and used to solve problems. A single processing unit can then use this “best model” to solve real world mazes during a day phase. Subsequent dream phases repeat the model evaluation process, taking into account information acquired during new day phases. This allows the definition of “best solution model” to evolve over time and adapt to what is happening in the real world.

³ We say “roughly the same amount of time” because the actual time needed to evaluate a given model is dependent on the maze at hand, the complexity of the model, and what solution steps the model uses. Thus it is unrealistic to expect all processing units to finish at exactly the same time every time, but it *is* realistic to expect that there will not be a significant time penalty simply because there are more processing nodes and/or more complex solutions being evaluated.

2.2.2 Maximum Speedup

Maximum speedup refers to how much faster a given program can be executed given additional parallel computing resources. In theory, a parallel program running on an N-core system (or one that can execute N instructions simultaneously) can be sped up N times versus serial execution. In reality, this is impossible because no known real-world program is 100% parallelizable, containing *no* sections that must be executed in a serial fashion. Perfect parallel speedup also assumes that the program can be parallelized in a manner such that the execution of one thread or process does not depend on the actions of any other thread or process. In reality, most programs have sections that must be executed serially and/or depend on the results of another section. For example, one of the inputs to a multiply instruction might be the result of an addition, thus prohibiting the two instructions from executing in parallel.

Amdahl's law [3] describes how maximum speedup is limited by the proportion of a program that must be executed serially. The law states that speed up versus a completely serial program, S , for N processors can be calculated as:

$$S(f, N) = \frac{1}{(1 - f) + \frac{f}{N}} \quad (1)$$

Here, f is the fraction of the program that can be executed in parallel; it is a value between 0 and 1.

Graphing this equation for fixed values of N , as seen in Figure 5, shows that the realized speedup rapidly falls as f decreases, asymptotically approaching a speedup equal to the parallelizable fraction of the program times 100. This is because Amdahl's law assumes a fixed workload size and that the goal of adding more processors is to reduce the amount of time required to process it. For example, previous work in the area of speech recognition has noted that simply dividing a given recognition problem equally across N available processors results in limited performance gains past a certain number of processors [18, 19].

Figure 5: Speedup achieved based on Amdahl's law.

Other work by Hill and Marty explored the effects of processor organization on speedup relative to Amdahl's Law [20]. Their work explores performance from a processor level view rather than a core level one. The researchers augment Amdahl's law to consider the sequential performance of individual cores [20]. They also assume that a given processor chip can contain up to n basic core equivalent (BCE) elements which can be organized symmetrically (i.e., all cores are formed by the same number of BCEs), asymmetrically (i.e., cores may contain different numbers of BCEs), or dynamically (i.e., the processor can rearrange the architecture of its cores on the fly) [20]. Their results show that for all but the most highly parallelizable applications speedup in static (i.e., non-dynamic) systems may peak when the number of cores implemented is *less* than the maximum possible number of cores. While the authors do not specifically address it, this is presumably due to the increasing overhead (communications, resource sharing, etc.) associated with the use of more cores. Making more cores available can increase performance until the overhead associated with managing all the cores becomes a significant bottleneck resulting in decreased performance. As the researchers point out this means that, "finding parallelism is still critical [20]."

Related to Amdahl's law is Gustafson's law [4]. Gustafson, in contrast to Amdahl, assumes that the size of the workload scales with the number of processors available [4, 16]. In other words, he assumes that as more processors become available, the size of the workload increases so that more work can be done in the same amount of time. Gustafson's law is as follows:

$$S(f, N) = N + (1 - N) \times (1 - f) \quad (2)$$

The graph in Figure 6 depicting Gustafson's law shows that adding additional processors to a system can increase performance without bound. In other words, additional parallel processing resources allow a system to do more work in the same amount of time. Gustafson argues that his interpretation is more representative of the real world [4]. While Gustafson's Law may be closer to reality than Amdahl's, both laws assume a homogeneous workload. In contrast, this thesis takes the viewpoint that a heterogeneous workload is more representative of many everyday computing needs.

Figure 6: Speedup achieved based on Gustafson's law.

Other research has examined scalability from Gustafson's viewpoint [21], some even going so far as to explore it in an MISD context [17]. There are not, however, any known systems that take the approach that CD does. CD attempts to capitalize on heterogeneous parallelism by exhaustively enumerating and examining all possible solutions (a heterogeneous workload). By taking on this large and complex workload during dream phases CD can select the best performing model, a relatively small and simple workload, to use during day phases. This "best known solution model" is simply the model that has found the most success thus far, not necessarily the optimum model. As a result, daytime performance in the given problem space is improved processing requirements are reduced, freeing resources for use by other tasks. CD attempts to improve model selection over time through an aggressive implementation of trial and error inspired by the brain's dreaming process. Thus CD takes advantage of Gustafson's Law while

dreaming to essentially perform more complex trial and error. The result is a reduction of workload during day phases.

2.2.3 Hardware Parallelism

Hardware parallelism is achieved by making use of multiple physical pieces of computing hardware that are all able to execute instructions simultaneously. There are many ways to do this. The SIMD, MISD, and MIMD classifications within Flynn's taxonomy, described in section 2.2.1, represent different ways in which hardware parallelism can be implemented. Each of these architectures can be implemented in many different ways. For example, MIMD systems can be further categorized into UMA and NUMA systems (also described in section 2.2.1). Software parallelism, on the other hand, encompasses more abstract parallelism. It is concerned with what parts of a program can be executed in parallel and how the parallel execution should proceed. This can be done explicitly by the developer or implicitly such as a processor determining on the fly what instructions can be parallelized or executed out of order.

Applying the concepts of parallelism to processor design is nothing new but making additional performance gains in that arena has become prohibitively difficult [1]. Until recently, processor manufacturers have improved processor performance by taking advantage of instruction level parallelism (ILP) [1]. Hardware designers have reached the limits of ILP, however, and have instead turned to other methods of improving performance such as multi-core processors [1]. Modern multi-core processors possess the potential to increase performance via thread level parallelism (TLP). TLP allows a program to be broken down at a higher level than ILP does by considering it in terms of several separate threads of execution, each with its own set of instructions [1]. Having multiple cores available allows for the execution of many instructions simultaneously rather than serially, offering the potential for performance increases beyond those currently offered by ILP. As a result, the burden of continued performance gains has been shifted on to software developers.

2.2.4 Software Parallelism

Software parallelism refers to crafting software with the intent that portions of it will be executed in parallel either on physically separate cores or as part of a multi-tasking environment. There are many ways to achieve software parallelism. One way is to enable TLP by, as the name suggests, making use of threading constructs and/or compiler directives in the source code. When a developer specifies how to parallelize execution in this way it is referred to as explicitly parallel programming. In contrast, implicitly

parallel programming is where a compiler determines which parts of a program can be parallelized and includes the appropriate instructions to do so.

Until recently, hardware based performance improvement has received most of the attention. Software developers have traditionally relied on improvements in processor throughput for performance gains. As processor speed improvements have leveled off [1, 2], however, they must now focus on writing software that can execute faster on current hardware and/or whose performance scales well with the availability of additional processing resources. Parallelization at the software level, whether implicit or explicit, is non-trivial. One reason is that it is often difficult to determine which parts of a program can be parallelized and how best to do it. In terms of implicit parallelism, the development of compilers that can automatically determine what portions of a program can be parallelized is an active area of research but one that has yet to find widespread success [22]. In terms of explicit parallelism, one reason for this difficulty is that most software developers have not been trained to think in a parallel manner, but rather to think in a sequential one. Some application programming interfaces (APIs) and libraries, for example the message passing interface (MPI) and OpenMP, aim to help eliminate some of the complexity while not relying entirely on the compiler. MPI consists of an API and library that, as its name implies, facilitates the passing of messages between parallel processes, as well as provides functions and constructs to enforce synchronization [23]. OpenMP allows the developer to provide the compiler with “hints” about what sections of code should be parallelized and how [16]. An OpenMP aware compiler can then automatically create and manages threads as appropriate, alleviating the developer of the responsibility [16].

Even with the availability of tools such as MPI and OpenMP, there are many pitfalls and problems associated with the development of complex algorithms that may prove to be crippling if not carefully taken into consideration during the early stages of design. The CD approach is a novel method of partitioning a difficult problem such that a number of parallel systems can independently work on a problem in different ways and then determine which instance offers the best solution for the problem at hand. By doing so, CD avoids many of the issues that may snare more traditionally architected parallel programs. Previous research has demonstrated that increasing the number of threads and/or cores working on a common problem beyond a certain point does not mean increased performance and can, in fact, lead to per-

formance degradation due to limitations on memory and/or bus bandwidth [5, 6]⁴. In contrast, consider that the brain has thousands if not millions of neurons working on a given “problem” at any one time. Data travels up the layers of the brain through parallel paths until it reaches the top, where a decision is made [9]; for example, a given observed object is a person and not a tree. CD takes advantage of this approach in order to simplify parallel application development and maximize processing efficiency.

2.2.5 Scalability

For the purposes of this research, scalability refers to the effect additional processing resources have on a system’s performance. A program or system “scales well” if its overall performance increases with additional processing resources while overhead remains relatively unchanged. The ability to create programs and systems that scale well is key to improving computational performance in the future. The CD technique can theoretically provide near perfect scalability.

A system that scales well can improve performance, while one that does not can actually cause performance to degrade [1]. One reason for this is that additional overhead is required to support communications between independent processing units as the number of processors in a system increases. Amdahl’s law, driven by the assumption that the size of a workload is held constant, shows that performance gains are limited by the fraction of a program that must be executed serially [3]. It can be assumed that communication between processing threads is a serial operation, as one thread must initiate a transfer before another thread can receive anything. Since more processors generally means more communications overhead, a greater fraction of the overall application must be executed serially, thus further limiting potential performance gains. If, however, we are able to hold execution time constant and scale the workload along with the computing resources available as Gustafson suggests [4], then we can linearly increase performance without bound. This has been demonstrated previously as part of research in the area of speech recognition [18, 19]. By processing a large workload during the dream phase CD is able to select a single best performing model, resulting in a smaller workload that must be processed during the day phase.

Kent showed that better scalability can be achieved by dividing a problem into multiple, smaller, and more tightly focused problems and then assigning one processing element (core) to work on each piece [18, 19]. This research focused on speech recognition and demonstrated that dividing the corpus pre-

⁴ The point past which performance degrades depends on the program and system configuration.

sented to a speech recognition processor into more focused groups of words and phrases on the basis of context achieved a linear speedup as additional cores were added to the system [18, 19]. In contrast, simply dividing up the corpus equally amongst available cores, without regard to context, limits performance gains past a certain number of available cores [18, 19], similar to what Amdahl's law predicts as shown in Figure 5.

The CD technique is nearly perfectly scalable as its dream phase is able to easily take advantage of additional processing resources to do more work in roughly the same amount of time. This allows the performance in terms of the amount of work that can be done during the dream phase to increase without bound. CD minimizes the required communication between processing nodes to keep the parallel fraction of the application a whole as large as possible, thus maximizing execution speed under Amdahl's law. Not only will the amount of work the system is able to do in a given amount of time increase but the additional computational power will also allow it to find better, more sophisticated methods of solving problems. The best performing solution is selected for use during day phases to minimize the required workload by making use of only the best known model. While the parallelizable fraction of a CD system is not addressed by this research, this example illustrates the "more realistic ideal" that Gustafson argues for.

2.3 Artificial Intelligence

CD has the potential to enable new and better artificially intelligent systems. It offers several key advantages over existing intelligent and biologically inspired techniques such as artificial neural networks and genetic algorithms. Artificial intelligence (AI), as its name implies, tries to make computers "smarter" so that they can learn and make better decisions in the same manner as human beings. The brain is able to quickly and efficiently perform many tasks that even the most powerful computer available today struggles with [7]. For example, the brain can quickly and confidently recognize a face with ease, while a computer must perform complex mathematical operations in order to make what can only be called its best guess. The proverbial "holy grail" of AI is to create a system that is able to operate as a brain does by learning and devising new ways of approaching problems on its own. With that in mind, it is important to note that most work in the field of AI is algorithm oriented; it generally does not take physical structure into account. In contrast, the work presented in this thesis attempts to take advantage of the structure of the brain. The following sections discuss some existing work in AI that is in-

spired by biological processes, including the brain. Included in each discussion is an overview of limiting factors for each approach, along with a description of how CD addresses those limitations.

2.3.1 Software-based AI

Software-based AI refers to artificial intelligence that is implemented entirely in software. This can be done as an inherent part of algorithm design, through software architecture, or a combination of the two. While software without adequate and/or specialized hardware resources may be impractically slow, software-based approaches have an advantage in that they can be implemented on existing computing hardware. They are also flexible as it is relatively easy and inexpensive to change software relative to hardware. This section compares CD to two software-based approaches inspired by biology: genetic algorithms and artificial neural networks.

Genetic algorithms attempt to find solutions to problems by applying the principles of natural selection and evolution [24]. The goal is to evolve better solutions to a problem over time [24]. Genetic algorithms are generally used to solve problems in which the search space of solutions is fairly large and unknown [25]. They find use in optimization problems (such as circuit board layout and job-shop scheduling [26]) and in NP-hard problems [25] (such as the travelling salesman problem). While genetic algorithms have some advantages, they also have their fair share of drawbacks and limitations. It is possible that the algorithm will “prematurely converge,” meaning all members of a population have identical values for a given bit position. The only chance of recovery from this situation is mutation [24]. Another drawback is that it must be decided when the algorithm has reached the point where it is “good enough” and no further iteration is needed. One technique is to use a fitness threshold but, due to the risk of premature convergence, it is possible that the algorithm will never reach the threshold. The designer of the algorithm must also consider what to do when a problem space changes and fitness of the population suddenly changes. Using the travelling salesman problem as an example, if a road is suddenly closed due to a bridge collapse solutions that were strong before may now be among the worst performers.

Next, consider artificial neural networks (or simply, “neural networks”). The name “neural network” literally refers to the fact that such networks are designed to mimic biological neurons [26]. While artificial neural networks do not have the same complexity or capacity of biological ones, both are made up of relatively basic units (neurons) organized in a highly interconnected fashion [26, 27]. Each neuron associates weights with its inputs and determines its output based on an activation function specified by the network designer. A neural network “learns” through the adjustment of the weights and biases of each

neuron [26] during a special training phase. Generally, the weights and biases in a network are initialized to small random values before training begins [27]. Neural networks are used to solve a wide variety of problems including pattern classification, function approximation, target tracking, speech and letter recognition, optimization, and more [27, 28]. They are not without their drawbacks, however. The randomness used in determining initial neuron weights makes neural networks susceptible to premature convergence, just as genetic algorithms are. It is also possible that a network may “over fit” a solution if a designer attempts to use too many neurons [26]. A network that has been over fit may only find a solution to a few specific inputs (usually similar to those presented during training) as opposed to finding success over a range of inputs.

An important contribution of this work is showing that CD has several advantages over both neural networks and genetic algorithms. It requires no special training phase and presents no risk of premature convergence or overfitting. CD also guarantees that a model (even if it is not the best model) will be selected at the end of every dream phase. This is due to the nature of CD’s dream phase which is always taking in new data that is used to improve upon model selections. As a result, even though CD may initially select poor solution models, it is expected that the model choice will improve over time as new data are considered. This allows CD to adapt to changes in the problem space (see the earlier travelling salesman example) without the burden of re-training, re-initialization, or re-design.

2.3.2 Hardware-based AI

Hardware-based AI refers to artificial intelligence that is made possible by, if not implemented entirely in, hardware. Such systems are architected to mimic the structure and operation of a biological brain. While such systems may execute software programs to solve specific problems, the idea behind them is that the physical structure and operation of a machine plays a significant role in determining its ability to exhibit intelligent behavior. Along these lines, this thesis asserts that the structure and operation of the human brain (i.e., how neurons fire and interact with each other) is critical to human intelligence and thus may enable computer systems to act more intelligently. As with many topics in computing, there are different approaches to hardware-based AI. CD as presented in this thesis is a part architectural and part software approach. The subsections below briefly examine two other hardware approaches.

2.3.2.1 *SpiNNaker*

The SpiNNaker project aims to construct a digital model of the human brain on a massive scale [7, 29]. One of the stated goals of the project, according to Furber [7, 29], is to build, “a system that would be

capable of modeling brain activity in real time . . . yet be as programmable as a general-purpose digital computer [7].” As mentioned previously, biological neurons encode information in terms of the timing of electrical impulses or spikes [29, 30]. The information, to the best of our knowledge, is encoded in the timing of these spikes; the size and shape do not carry any information [29]. Traditional neural networks as discussed above use mathematical functions to generate values that serve as the means of communication between neurons. The SpiNNaker project, on the other hand, is an attempt to implement a spiking neural network [29]. Taking advantage of the inherent parallelism in neural networks, SpiNNaker operates on a large array of parallel systems in order to simulate its neural network [29, 30]. A SpiNNaker chip consists of twenty ARM968E-S cores along with an SDRAM interface and a network router [29, 30]. This router controls a network on chip (NoC) system that allows the cores on a given chip to communicate with each other, a local SDRAM (external to the chip) and other chips and core in the larger SpiNNaker chip network [29]. Each node (a SpiNNaker chip and SDRAM) is connected to six neighboring nodes to form a large, 2D hexagonal mesh [30]. The designers of SpiNNaker estimate that each core will simulate 1,000 neurons firing at a rate of approximately 10Hz [29]. This design enables SpiNNaker to simulate the massively parallel nature of a biological brain in contrast to more traditional neural networks that operate on a smaller scale. SpiNNaker is still in the development stage; its designers have only recently received the first units of custom hardware and, while initial experiments have indicated that the hardware design is working as intended, there remains much work to be done to evaluate how well the system performs.

While the SpiNNaker project attempts to build a model of the brain, CD takes inspiration from the structure of the brain to influence computer architecture. Instead of modeling large numbers of neurons, CD generates models of solutions to a given problem type and tries to determine which is best. These solutions are constructed by combining problem solving steps into more and more complex solution models, mimicking the way that the layers of the brain operate on increasingly abstract concepts [9]. Furthermore, SpiNNaker’s goals include the ability to support real time modeling and general purpose programmability [7, 29]. In contrast, the investigation presented in this thesis is concerned with building a system that can find solutions to complex problems by using novel methods of partitioning. For the purposes of this work, there are no real timing constraints beyond the ability to execute experiments in a reasonable amount of time.

2.3.2.2 BrainScaleS

While the SpiNNaker project attempts to model a network of neurons using digital circuits, the BrainScaleS project attempts to model neurons using analog circuits [7]. This is done using the “Spikey chip” which implements spiking neurons, including interconnections, in an analog fashion [31, 32]. The chip can be reconfigured for modeling different networks though over 2,900 different parameters [32]. An FPGA is used to control the operation of the Spikey chip in real time during experiments [32]. Researchers have already made use of the chip to model several different neural network systems including a portion of an insect’s olfactory system [32]. Spikey chips must be subjected to a calibration process, which can vary depending on the experiment at hand, to account for variability in the manufacturing process [32]. Without calibration this variation can cause repeated iterations of the same experiment to produce different results [32]. Like SpiNNaker, however, the project is still in its early stages and there is much work to be done with Spikey chip and within the BrainScaleS project.

The BrainScaleS project, like SpiNNaker, focuses on making a model of the brain. In the case of BrainScaleS, however, the focus is on using analog circuits to mimic the brain more directly. CD is also inspired by the brain’s structure, but its goal is not to emulate the function of the brain directly. Instead, it is inspired by the layered structure of the brain and the way that each layer combines inputs from lower layers into more complex models of a solution to a problem. This is achieved by enumerating all possible models (within user specified limits) while the system is offline and not being used to solve real world problems. Moreover, while the BrainScaleS approach shows promise, calibration procedures are required to ensure that experiments are repeatable [32]. CD, in contrast, requires no special calibration or setup beyond what is needed to implement the initial system, making it easier to use.

2.4 Maze Solving Overview

For the purposes of this research, mazes were chosen as an example problem space. Demonstrating that CD can be used to successfully solve mazes suggests that CD can also be successfully applied to many other problem types. There are several reasons for using mazes in this initial research, including:

- There are many existing algorithms that can solve mazes, thus providing a basis for performance comparison and selection
- It is easy to guarantee that a maze always has at least one solution

- An agent⁵ attempting to solve a maze does not need much information in order to get started
- Maze solving is closely related to path finding, a problem with many real world applications

First is that there are many existing methods to find a solution (or solutions) to a maze. They differ in terms of performance by the solution space they target, i.e., the type of maze. The solutions generated by the CD system can be compared to solutions found by these other algorithms to determine if it has derived a known algorithm via the dreaming process. The next favorable property of mazes is that they can be easily constructed such that there is always at least one solution. This guarantees that there is indeed something that can be solved, ensuring that the CD system is not wasting effort attempting to find a solution where there is none. A maze is also a problem that can be set up with very little initial knowledge on the part of the solver; only the starting location and how to identify the goal location need to be known (as opposed to requiring an a priori knowledge of the entire maze layout). This places more emphasis on the usage of previously gained knowledge during the dreaming phase. Finally, mazes represent something of a “real world,” and thus relevant, problem. There exist competitions to build maze navigating robots as well as video game AI that must know how to find a path from point A to point B.

While the previous sections in this chapter discuss generalized methods of biologically inspired AI, this section briefly explores a few methods of solving mazes in an “intelligent” manner. There are many algorithms that can be used to solve mazes from breadth-first search to Dijkstra’s algorithm [33] to A* search and beyond. In each case, however, the computer must be told what steps to take in order to find a solution beforehand; there is no room for the system to evolve in an attempt to find new and potentially better solutions.

For the purposes of this research, maze-solving algorithms can be categorized into different groups as follows:

⁵ An agent is a computational unit (i.e., a thread or process) that uses a single solution model in order to process inputs in an attempt to derive a solution. Thus an agent is not itself a solution model, but rather a process that makes use of a solution model by applying it to input data in order to generate solutions.

- How much knowledge of the maze is required up front? That is, must any information about the maze be known from the start (other than starting location)?
- Does the algorithm simply return *a* path through the maze (even if the path is inefficient) or the shortest path?
- Can the algorithm accommodate loops in a maze (i.e., a maze with multiple solutions) or not?

As presented in this thesis, CD requires no prior knowledge of a maze (save for the starting location). It is also aimed at finding *any* solution to a given maze (as opposed to the shortest or fastest solution) and is able to deal with mazes with multiple solutions. The remainder of this section examines some other maze solving algorithms that share these characteristics and can thus be easily compared to the solutions generated by CD. For the purposes of this comparison, algorithms that find the shortest path (versus any path) are also discussed.

Since a maze can essentially be reduced to a graph traversal problem, we can turn to graph theory for answers on how to find either the shortest path through a maze or any path at all. Breadth and depth first search are algorithms that most (if not all) computer engineering and computer science students examine in their undergraduate studies. These algorithms exhaustively explore a graph to find all the vertices and edges. While this can find *a* path through a maze, other algorithms, such as Dijkstra's algorithm [33], can be used to find the *shortest* path through a maze. These methods do not represent any sort of artificial intelligence but they can be used as either a starting point for the development of maze solving AI algorithms or a point of comparison.

An example of an AI algorithm used to solve mazes is the method proposed by Gordon and Matley [34] which takes advantage of genetic algorithms. Their approach divides a maze into several large sectors. Each sector is comprised of a number of maze cells and is assigned a direction. When an agent moves into any of the cells in that sector, it is instructed to move in the direction assigned to all cells in the sector. The authors refer to this as a sparse direction map [34]. An example is shown in Figure 7. Once a maze has been divided up, a genetic algorithm is used to gradually evolve the direction map until one that is most effective is found. One drawback to this approach is that it can take many generations to evolve a suitable direction map. In fact, the authors note that for a complex maze over 1200 generations were required to find an adequate direction map [34]. Another potential problem is that a particular direction map is only guaranteed to be good for the maze on which it was evolved. If the maze changes

then the algorithm must start over again. Finally, this algorithm requires knowledge of the entire maze up front, information which may not always be available. In contrast, the maze-solving CD implementation developed for this thesis does not need complete knowledge of a maze, nor does it need nearly as many iterations to find an adequate solution model.

Figure 7: An example of a sparse direction map with the goal in the center of the maze. Adapted from [34].

The IEEE micro mouse competition asks teams to build a robot to navigate an unknown maze. The competition has resulted in a number of papers describing different approaches to solving the maze. One of these [35] examines three popular methods: the wall follower, Dijkstra's algorithm, and the flood fill algorithm. The authors briefly describe each algorithm and then discuss the relevant drawbacks.

The wall follower algorithm is simply an implementation of a depth first search. The authors point out that this algorithm falls short in that it does not incorporate any actual intelligence and is too inefficient to solve mazes with multiple solutions or high complexity [35]. Significantly, a wall follower cannot deal with a maze with loops (it may very well fail to find a solution to such a maze) and will not necessarily find the shortest path.

The authors go on to note that while Dijkstra's algorithm can provide an optimal solution it requires that the robot fully explore the maze first, a process that consumes a lot of time and power [35]. In contrast, CD can arrive at solution models that can deal with complex mazes, including those that contain loops.

Moreover, it does not require a maze to be fully explored in order to find a solution, unlike Dijkstra’s algorithm.

Finally the flood fill method is discussed. This algorithm works by “flooding” the maze and assigning each cell a score that represents its distance from the goal. The authors state that this approach allows the robot to solve the maze as it is mapping it, producing an optimal solution while requiring less time than Dijkstra’s algorithm [35]. Prior to solving the maze, the robot knows the size of the maze, its starting position, where the goal is, and how many moves from each cell to the goal. Figure 8 shows an example of a maze and the initial grid stored by the flood fill algorithm. As the robot moves through the maze and encounters walls, it updates this map and thus its perceived best path through the maze. CD, on the other hand, does not need any prior knowledge of a maze at all. It can discover optimal solution models on-the-fly with the only data gathered from observation available. The flood fill algorithm is similar to the A* search algorithm in that both take the estimated distance to the goal in to account. A* improves upon this concept by also taking into account the distance travelled thus far and incorporating the use of a heuristic function. The heuristic function is used to estimated the distance from a given cell to the goal and can be used to speed up algorithm execution at the potential cost of finding a suboptimal solution.

Figure 8: A representation of the maze is shown at left along with start (S) and goal (G) locations. To the right is the starting point for the flood fill algorithm. Adapted from [35].

Research has also been done using Q-learning to solve mazes [36, 37]. The Q-learning algorithm uses reinforcement learning to learn what the best actions to take are over time and is thus geared toward

finding the optimal solution to a given maze [37]. This means that, similar to how genetic algorithms evolve a solution or how a neural network must be trained, many iterations are required in order to establish a background knowledge that allows an agent to select an optimal path. Osmankovic and Konjicija showed that for even a small 5x5 maze over 2,000 iterations were required while larger mazes (19x19 and larger) can require over 30,000 iterations [37]. Gunady and Gomaa present a modified Q-learning approach, called state aggregation Q-learning, which is designed to reduce the time needed for Q-learning to converge [36]. While the results do indeed show an improvement over standard Q-learning, the authors note that as a maze becomes more complex their approach is more or less reduced to the performance of more traditional Q-learning algorithms [36]. CD, on the other hand, requires far fewer iterations in order to find reliable methods of solving mazes. By examining a large number of potential solution models simultaneously it is able to converge on a successful model with much less iteration.

There are many algorithms for solving mazes, each with its own particular advantages and drawbacks. This research uses maze solving as an example problem space to demonstrate how CD can equal (and in some cases surpass) the performance of existing algorithms. In doing so, it is able to avoid drawbacks that affect other methods while also providing the same advantages. While there is nothing maze-specific about CD's core operation, demonstrating that it is an effective method in one problem space (mazes, in this case) suggests that it would be effective in other problem spaces as well.

2.5 Summary

As multi-core processors have become the norm, there has been a renewed focus on parallel computing in recent years. Many architectures and techniques that have been researched and developed that specifically target multi and many core computing [7, 15, 16, 23] yet none have emerged as widely adopted options for improving performance. The brain has been an inspiration for other efforts in parallel [7, 29, 32] and intelligent [27] computing but the approach taken by CD is believed to be a novel one. Unlike previous hardware-based solutions, the focus of CD is to emulate the brain's architecture and dreaming process as opposed to directly modeling them. In doing so, it takes advantage of the apparent benefits of dreaming and sleep [10-12] to both improve performance in a nearly perfectly scalable manner and build an intelligent computing platform that does not suffer from the drawbacks of other biologically inspired techniques [24, 27]. This research shows that CD is a viable technique through the use of mazes as an example problem space.

Chapter 3 Computational Dreaming

The brain is essentially a massively parallel processing machine with billions of processing nodes in the form of neurons [9]. The brain's neurons are organized into several layers that process increasingly complex concepts [9]. Modern computer processors are similar to the brain in that they are made up of multiple cores, all capable of performing the same basic functions. Unlike the brain, however, these cores are not connected in a layered architecture. Software developers, within the limitations of available hardware, are able to impose their own organizational structure between cores. Popular parallel development libraries and techniques, however, place much of the burden on the developer to specify how this should be done, as well as making the developer responsible for data integrity and avoiding situations like deadlock [16, 23].

Designing, developing, and especially debugging parallel software is a significant undertaking, so much so that manual development can become unrealistic. Moreover, traditional methods tend to increase required overhead as more processor cores (or threads/processes) are added to the system. The more overhead required by a parallel application, the slower it will be. This limits scalability and has been a big barrier to improving performance through parallel computation. CD presents a way to not only minimize the overhead associated with parallel applications but also to simplify the development of systems that make use of many cores. It takes inspiration from both the brain's structure and its dreaming processes. In doing so, it provides a method of nearly perfectly scalable parallel computing.

3.1 An Overview of Computational Dreaming

Researchers have observed that various areas of the brain's cortex all look identical and suggested that each neuron in the brain actually performs the same basic function [8]. Hawkins later noted that the implication of this idea is that what makes different areas of the brain respond to different stimuli (e.g., auditory versus visual) is how neurons are interconnected [9]. The idea is that each layer takes inputs from the layer below it and passes outputs to the layer above with the lowest layer getting its input from the environment in the form of sound, light, heat, and so on [9]. Thus each layer aggregates inputs from lower layers in different ways, eventually reaching the highest layer, which decides which interpretation of the sensory stimuli is "best." Then, it selects the best partition which is also greatly simplified

over a version of a maze solver which would include all possible solution methods. CD attempts to take advantage of these ideas in order to minimize parallel processing overhead and maximize scalability.

CD also takes inspiration from the brain's ability to dream. Dreaming is a phenomenon that occurs in all animals [10, 14]. While dreaming, the brain ignores inputs from the surrounding environment and essentially stimulates itself [13]. Research suggests that the brain's problem solving ability can be improved through dreaming or at least periods of unconsciousness [11, 12, 38, 39]. In different experiments, both Cai et al. and Wagner et al. found evidence that suggests that unconsciousness (and presumably dreaming) enabled test subjects to more easily and quickly solve problems [11, 12]. (See section 2.1 for a more detailed discussion.) These concepts inspire the dreaming process used by the CD technique. Theoretically the brain's billions of neurons all operate with perfect (or nearly perfect) parallelism. Part of the thesis of CD is that the process of dreaming helps the brain to reconfigure itself to optimize and select a set of candidate solutions. Over time, this would allow the brain to scale its processing power up in order to handle more complex scenarios. CD attempts to mimic these characteristics in order to form a method that is not only scalable but also provides intelligent data processing. At the same time, it avoids problems that afflict other biologically inspired techniques such as neural networks [27] and genetic algorithms [24].

Based on this inspiration from the brain, CD has the following high level characteristics:

1. Makes use of a layered architecture in which higher layers operate on more complex concepts, representing combinations of simpler concepts contained in lower layers
2. Allows for many possible approaches to be developed and selected in solving a problem, thereby partitioning the solution space
3. Employs a process (dreaming) to determine which approach tends to work the best, allowing for a single, relatively simple approach to be used when not dreaming
4. Allows for a large number of problem solving approaches to be developed and evaluated independently and in parallel simultaneously

CD executes over a series of iterations. Each iteration consists of a day phase followed by a dream phase. A CD system observes and records information about inputs it receives and processes during the day phase. Later, during the dreaming phase, the system draws upon these records and plays a random subset of them back. This playback simultaneously presents stored inputs to multiple processing units in

parallel. Each unit uses a different method (model) to process the inputs. The output from each processing unit is then evaluated against all other outputs. Whichever processing unit is deemed to produce the “best” outputs is then used in a subsequent day phase and the process repeats, eventually converging on an optimal processing model. As with neurons in the brain, these models are organized into layers. During the dream phase, all models from all layers are exercised in parallel simultaneously. At the end of the dream phase, the model that tends to perform the best is selected for use during the next day phase. Figure 9 shows a graphical representation of the dreaming and day phases.

Figure 9: Depiction of the flow of information in dreaming and day phases. Adapted from figures in [13].

Each layer of models effectively combines models from lower layers in different ways in order to produce new, more complex models. As an example, consider a CD system designed to solve mazes. Models on the lowest layer may consider only one direction of movement; one model may only try to move north while another only moves east. Models on higher layers combine the directions on lower layers in different ways to create more complex solutions; for example a model on a much higher layer may move north, east, south, east, south, and north as it tries to solve a maze. A simplified version of this is illustrated in Figure 10. While some models may be nonsensical and some may overlap (or even duplicate) the functionality of other models, it is expected that many models will present potentially valid solutions to a given problem.

Figure 10: A simplified example of how higher layers contain more complex models by combining lower layers in different ways.

There are a number of implications associated with the concepts described in this chapter and depicted in Figure 9. Many of them address problems that exist with parallel computing today as described in section 2.2 and at the beginning of this chapter. First, looking at the dreaming phase, all solution models (represented by shaded/hatched circles) operate on the same inputs simultaneously and in parallel. Each model processes the data differently from other models and thus generates its own solution to the given problem. This method of processing is the definition of a true MISD computer (see section 2.2.1), something believed to be novel [1]. This architecture allows all solution models to operate completely independently; there is no need to share resources or data between models and thus no communication is needed (save for that which communicates problem data to a model and retrieves solution information from it). As more parallel processor cores are made available more solution models can be evaluated in the same amount of time. Since no communication or resource sharing amongst processors is required this makes the CD system nearly perfectly scalable. This scalability allows a CD system to take advantage of additional processing power while dreaming to solve increasingly complex problems with almost no penalty, something that traditional methods have been unable to achieve.

Another implication is that CD largely takes much of the complexity of parallel software development out of the hands of a developer. Traditionally, parallel software development has been regarded as challenging at best. A developer needs to consider how to safely share data across parallel processes in order to ensure that no deadlocks occur and that data integrity is not compromised. Constructs such as

mutexes and semaphores exist for this purpose, but it is up to the developer to use them consistently. Finding and fixing problems when they do occur can be very difficult. Indeed, Pankratius et al. noted that large parallel applications can quickly become too complex for manual optimization and management and that attempts at automating such processes have been largely unsuccessful [40]. CD avoids all of this by implementing parallelism in such a way that sharing data between processes is not necessary in the first place. Each model that is evaluated during the dream phase executes completely independently of other models. Input data is read from a common source and results are stored such that they do not overlap each other. Thus the developer of a CD system is responsible only for deciding how input data is distributed and results collected; the dream process automatically manages thread creation. This makes the development of a CD system much simpler than the development of a more traditional parallel application.

The end result of CD iteration is the selection of a single model that is able to reliably and consistently process inputs. In comparison to the large number of possible models evaluated by the dreaming phase, this single model is relatively simple and easy to execute, thus minimizing the use of resources during the day. Adding additional processing resources to a CD system makes it possible to examine many more models in the same amount of time during a dreaming phase. This allows CD to easily scale to handle more complex problems with minimal additional overhead.

3.2 The Phases of Computational Dreaming

Each CD iteration consists of a day phase followed by a dream phase. During the day phase, the CD system uses a single selected model to solve new problems presented by users. As the system attempts to solve each problem, it stores some information about the problem. The details of the information stored depend on the type of problem at hand but in all cases enough information is stored to allow the system to recreate the same problem later. Problem information is always stored, even if the problem was not solved successfully. Figure 11 depicts this flow of information. Model 1 was the selected model for this iteration. It operates on user input and generates a potential solution, which the system translates into actions. At the same time, it stores what it observes about a problem into a problem information database.

Figure 11: Day phase data flow. Adapted from a figure in [13].

The dream phase is what makes CD interesting. Figure 12 shows a graphical representation of data flow in a dream phase. While dreaming, the system does not accept user inputs and operates solely on information stored regarding problems observed during previous day phases. A number of agents, one for each available solution model, each attempt to simultaneously process stored data. Each agent generates its own potential solution to the problem independently of all other models. This is an important feature of CD. As more processors (or processor cores) are added to a CD system, a greater number of solution models can be evaluated in a given amount of time with almost perfect scalability. This is possible because all agents operate completely independently of one another. Simultaneously employing a number of different models in parallel and feeding them all the same input data creates an MISD system, something which we believe is novel [1].

Some of these potential solutions may be completely wrong, while others may be successful. Among the successful solutions, some solutions will be better (e.g., faster, more accurate, etc.) than others. The characteristics that make one solution “better” than another depend on the given problem. At the end of the dream phase, a performance comparison function is used to compare the quality of the various solutions generated during the phase. Based on the available performance data, the best performing model is selected for use during the next day phase. The ultimate goal is to construct a system that is

able to select a context-specific, and thus simpler, solution model that effectively functions as an algorithm for reliably solving a problem or for processing data (mazes in the case of this thesis).

Figure 12: Dream phase data flow. Adapted from a figure in [13].

The number of iterations that are executed is up to the user. Over time, CD will converge on a solution model that is able to reliably provide solutions to the given problem type. Due to the iterative nature of CD, the system will be able to select better and better models, ideally converging on an optimal one. A CD system will also be able to adapt to changes in the problem space over time.

As an example, consider a CD system that is constructed to predict traffic congestion in a given area. Over time, this system will converge on a reliable model of the traffic. If a new road is built, however, then traffic patterns change and the previously “best” model may now perform much worse. As the CD system continues to iterate, collecting new information each day, it will eventually select a new model that provides solutions that better fit the new environment.

3.3 Challenges

There are a number of challenges associated with the development of any CD system. Developing solution models is one of them. This can be divided into two parts. The first is to determine what the fundamental elements of a solution model should be. It is a decision that is problem dependent and may not necessarily be clear cut. As this decision can have a profound impact on the performance of the CD system and how other challenges are addressed, it is important to carefully consider available options.

Once the fundamental elements have been selected, the next problem is to develop a way to combine

(and recombine) these elements together in different ways to form solution models. Related to this issue is whether or not available solution models should remain static or be allowed to change over time. Yet another question is to decide how many solution models the CD system should develop and/or know about at any one time. The answers to these questions can have a dramatic effect on how the CD system performs.

Another challenge is to devise a method of evaluating the quality of a given model solution. A CD system must have a way to compare models to each other so that it can select the best one for use during day phases. This challenge is similar to the challenge that a genetic algorithm designer faces when developing a fitness function. What constitutes a “good” solution is different for each problem type. It can also depend on what the designer’s priorities are (e.g., a faster solution versus one that uses less memory). Related to the definition of what constitutes “good” is establishing a definition of what constitutes “bad.” It is possible that a solution model simply cannot solve some instances of a problem. For example, a certain maze may require all left turns to solve but the model being used only contains right turns. Thus it is important to devise a way to tell when a model should give up trying to solve a problem in order to avoid an infinite loop or otherwise unacceptably long execution time.

Once these challenges above are addressed, more important challenges are investigated. Chief among these is showing that CD is a viable method of processing data. Viability means showing that the CD technique is able to consistently and successfully process data and solve problems. Going a step further, if it can be shown that CD can arrive at reliable and efficient methods of solving problems then this suggests that a CD system may be able to derive algorithms that are as efficient as those devised by humans, if not more so. For example, if a maze solving CD system can select, on its own, a solution model (or models) that consistently produces solutions that are like those that a well known algorithm would produce (such as Dijkstra’s algorithm), then this suggests that CD has the capacity to arrive at efficient solutions to a wide variety of other problems. Moreover, it suggests that CD could discover algorithms that humans as yet have not.

Once viability is shown, another challenge is to show that CD can perform consistently. Existing biologically inspired AI such as neural networks and genetic algorithms are prone to “getting stuck” on a local minima due to random chance, potentially never converging on a solution at all. If this happens then a user must intervene to, for example, restart an algorithm’s training sequence. One of the goals of this

research is to show that CD is robust in that it can consistently find success and that it is not prevented from doing so by random chance. If successful in this regard, it would show that CD needs no oversight or user intervention and thus has an edge over both neural networks and genetic algorithms. The design of CD is such that the system exhaustively examines all possible solutions to a problem during the dream phase. Some of these models will likely produce suboptimal solutions that represent local minima. Since all possible solutions are examined, however, CD will still choose the models that successfully solve the most mazes and, should two or more models be equally successful, the model that tends to produce the shortest solutions.

The final challenge addressed in this research is to determine how variations in input data affect the performance of CD. While CD might be both viable and robust, if it is only successful when processing data sets with certain characteristics, its applicability will be limited. A more detailed investigation of how CD performs when presented with data sets of varying size and complexity would provide not only an initial characterization of CD's abilities but also a basis for planning future research and development.

This thesis explores CD using mazes as an example problem space. Its primary goal is to demonstrate that CD is viable and can reliably select solution models during dream phases that find success in day phases. Showing that CD is a viable maze solving technique suggests that it will likely find success in other problem spaces as well. Supporting experiments examining the robustness and performance of CD (again using mazes) are also performed to provide a stronger foundation for additional CD research.

While this work addresses a number of challenges, it does not necessarily fully investigate them. Fundamental solution elements and a method of determining whether a solution is "good" or "bad" are defined out of necessity but neither topic is explored in depth. The question of "how long does it take CD to adapt to changes in the problem space" is not covered by this research, nor is scalability. These and other topics are left to a future effort in favor of first showing that CD is capable of finding success at all.

3.4 An Overview of Maze Solving with Computational Dreaming

As described previously, this thesis investigates CD using mazes as an example problem space. The goal is to not only demonstrate that CD is viable, but also to show that it is robust and useful when applied to a wide range of problems. CD generates models that it uses to process data and attempt to solve problems. Generated models are organized into layers such that simpler models are found on lower layers

and more complex models on higher layers. Complex models consist of combinations of simpler models. For a given level of complexity, all possible solution models are enumerated. Let us assume that from each cell it is possible to move in four directions: north, south, east, and west. Figure 13 shows an example of the first two layers for a system with these basic solution elements.

Figure 13: Example of the first two layers of maze solving models

On the lowest layer, layer 0, are the simplest models which each contain only one direction. Layer 1 contains more complex models with two steps in each model. These more complex models are constructed by combining directions from simpler models in different ways such that all possible combinations of steps on layer 0 are enumerated. Layer 2 would then enumerate all possible combinations of models from layer 1; layer 3 would enumerate combinations of models on layer 2, and so on. This gives the system a wide array of potential ways that it can solve a problem, each of which is made up of the same basic building blocks. The only limit on the number of models that can be enumerated, and thus the size of the search space, is the availability of processing resources. CD is used to select which of the available models is “best” given what it has seen in the past. As described previously, CD does this over a number of iterations.

3.4.1 Solution Models

Solution models are used to attempt to solve generated mazes. As described above, each model contains one or more steps to take in order to attempt to solve a maze. Each step consists of one of the four possible movement directions (fundamental solution elements): north, east, south, or west. Models are used to attempt to solve mazes by repeatedly executing the directions within the model in the order that they are stored. The mechanism that uses a solution model in such a way is referred to hereafter as an agent, and the agent executes the steps within the chosen model in order. At the start of a new

maze, the agent only knows its starting location and what cells it can observe from said location; it does not know the location of the exit, the number of solutions available, or the size of the maze at hand. If a wall blocks the agent's path when it attempts to execute one of the steps, that step is simply skipped. The agent will execute the steps within its model over and over again until it reaches the maze's goal location or it determines that it is stuck and not making any progress. (Section 4.4.1 describes how an agent determines whether or not it is stuck.) If the agent gets stuck it gives up trying to solve the maze. If the agent finds the exit, it returns a complete solution to the maze.

Figure 14: Maze solved by a model with steps: S, E, S, E, N, E, S, E. White cells are those visited by the agent. Cells shaded light gray are those observed, but not visited. Cells shaded dark gray are unknown to the agent. The upper-left cell with an S is the start point while the lower-right cell with the G is the exit or goal.

As an example, consider an agent whose model contains the following eight steps: South, East, South, East, North, East, South, East. Figure 14 shows an example of a maze that the agent has solved. The agent starts at the cell marked S and proceeds to repeatedly follow the steps in the model as best as it can. From the starting cell, it observes that there are no walls to the East and South and thus knows that open cells exist in both directions. The agent first moves South (as this is the first step in the model) and then attempts to move East but is blocked by a wall. Since it was not possible to move East, the agent looks to the next direction in the sequence and again moves South. The agent then continues with the remaining directions in the model, skipping all East directions (as cells contain walls to the East) and moving North and then South again. At the end of one iteration using this model, the agent is located

two cells South of the start position. The agent has visited three cells (the start cell and the two cells to the South of it) and observed an additional two cells (to the East of the start position and to the South of where it ended the first iteration). After executing the sequence of steps in the model several more times, the agent produces a solution with a total of 12 visited cells and six observed cells as depicted in the figure. White cells are those visited by the agent at least one time, while those shaded light gray are cells observed, but not visited, by the agent. Dark gray shaded cells are those that the agent has no knowledge of.

3.5 Limitations

CD offers several advantages over existing approaches including nearly perfect scalability, the ability to adapt to changes in a problem space, and the ability to avoid premature convergence on local minima. There are also, however, limitations to the technique. One limitation is processing resource requirements. Ideally, a CD system will have one processor core available per solution model. Since the number of models grows exponentially with the number of layers (the four layers used in this research contain 65,812 total models), however, this ideal quickly becomes impractical. Multiple models can be processed by each core at the expense of overall execution time, meaning that a system designer must carefully evaluate the tradeoff between execution time and the complexity of enumerated models.

Depending on the problem space, memory requirements may be another limitation of the CD approach. If, for example, a problem required relatively large and complex solution models even on its lowest layers, the memory required for all models may become an issue. As with processor requirements, this is especially true as more models are enumerated.

Finally, while CD theoretically guarantees that the optimum solution for a given problem space will ultimately be selected, it makes no guarantees as to how long this selection process might take. It is possible that hundreds or thousands of iterations may be required in order to find the best solution model. Similarly, when a problem space changes, CD is able to adapt but makes no guarantees as to how long the process will take.

While these are currently limitations, CD points to a level of parallelism and memory capacity inspired by the brain. Thus, these current limitations may eventually become insignificant as computing seeks to

re-visit its placement of infrastructural overhead and radically new architectures are needed that can take advantage of anticipated levels of parallelism.

3.6 Summary

CD takes inspiration from the brain's layered architecture and dreaming process to create a technique that is able to evolve better solutions to data processing problems over time while avoiding the pitfalls of methods like neural networks and genetic algorithms. Its MISD architecture enables near perfect scalability, allowing it to easily take advantage of additional cores to solve larger problems. A number of challenges have been discussed but none are insurmountable. While this chapter describes how CD can be used to solve mazes the next focuses on how a complete maze-solving simulator was implemented to facilitate this research.

Chapter 4 A Computational Dreaming Simulator

CD promises several key benefits over existing methods of parallel computation. One is near perfect scalability. By allowing parallel computation to occur on a massive scale in a dream phase with little to no overhead and thus simplify algorithms used for daytime operation CD has the potential to address increasingly large and complex problems as more and more processing resources become available, scaling workloads to fit the system. Another is a robustness that other biologically inspired approaches such as neural networks and genetic algorithms do not possess since they have the potential to get stuck in solutions that require user intervention. Through a dreaming process, CD determines which model performs best and uses it to process data presented by the user. No special training phase is required and there are no random mutations or selections.

The main goal of this thesis is to illustrate that CD can be a viable solution which warrants further investigation. The completely novel simulator is then used to demonstrate that CD is consistent with nearly perfect scalability and not subject to getting stuck in local minima, unlike conventional AI approaches.

The simulator is designed to solve mazes using CD. Several challenges were addressed during simulator development, including:

- Where the input data will come from
- How mazes will be represented within the simulation
- How solution models will be represented within the simulation
- How solution models will be employed to solve mazes
- How the “best” model is defined
- What the outputs of the simulator should be
- What parameters should be made available to the user

The remainder of this chapter describes how a CD simulator was constructed, how it operates, and how these challenges were met.

4.1 Simulator Architecture

One of the first challenges that was addressed in designing the simulator was to determine where CD input data should come from. It was decided that presenting a wide variety of inputs was important so that the simulator does not inadvertently converge on a solution that, for example, can only solve mazes with no loops or only solve mazes that are smaller than 10x10. A maze generator was developed to fulfill this purpose. This generator (described in more detail in section 4.2) operates within user-specified simulation parameters to randomly create new mazes to feed to CD during day phases. This random generation is meant to re-create the input data variability that one would expect to encounter in real world problems. Although there may be some similarities, we do not expect to encounter exactly the same situations during each day phase and thus neither should the simulation operate on the same set of input mazes over and over again. Related to determining the source of CD input data was the challenge of determining what parameters a user should be able to control. Parameters are divided into two categories: operational parameters that control how the CD implementation operates and simulation parameters that control how the simulation runs.

Here, operational parameters are examined first. Prior to beginning an experiment, it is impossible to know how many iterations will be required before a viable model is found, nor how many iterations it might take to discover better models. Moreover, while one goal is to select a relatively simple model, there is no method to predict how complex a model must be before it is able to produce working solutions. Thus, there are two operational parameters: how many total day/dream iterations to execute and how many layers of models to generate.

Next, in terms of simulation parameters, there are several things to consider. First, there is no way to predict (for a given set of simulation parameters) how complex a maze can get before it is beyond the capability of the simulator. Another is that it may be useful to control how variable the aforementioned randomly generated mazes are. For example, mazes could be widely variable in terms of size and number of possible solutions or more tightly constrained, encompassing only a narrow range of possible sizes and complexities. Moreover, it is impossible to determine how many mazes the system must see during each day/dream phase before it is able to reliably converge on solutions. Since these parameters cannot be calculated in advance it was determined that users should be able to adjust all critical aspects of maze generation, as well as how much “time” CD has to operate while in day and dream phases.

Finally the simulation outputs were determined. Outputs must include not only the outputs of the CD system being simulated (operational outputs) but also information on how the simulator runs. This information will provide insight into how the system performs during the simulation. In addition to the CD outputs of observed day mazes and discovered solutions, the following monitoring outputs are also generated:

- How many times each model was selected for use during the day (*Times selected*)
- How many total mazes each model attempted to solve and how many were solved successfully during day phases (*Day success rate*)
- How many total mazes each model attempted to solve and how many were solved successfully during dream phases (*Dream success rate*)
- The average length of generated solutions (*Avg. solution length*)

A high level simulation architecture was also developed. This architecture, shown in Figure 15, has four main parts: day maze (input) generation, model generation, a maze solving CD implementation, and a simulation monitor. The model generator creates the entire set of solution models that will be available to the CD system at the start of simulation. The maze generator (*day maze generator*) creates new maze data that is passed to the CD implementation during day phases. Both of these pieces are controlled by the parameters described above. CD operates on generated input mazes using the solution models generated at the start of simulation. During the course of the simulation, CD generates outputs in the form of maze solutions and a “database” of maze data observed during day phases. The *simulation monitor* is designed to collect data about the operation of the simulator and the CD implementation and to produce output at the end of simulation. These outputs are meant to be used in performance analysis to gauge how CD performs and include a dump of all mazes that were observed (*observed mazes*) and solved (*solved mazes*) during the simulation.

Figure 15: High level simulator architecture.

Each simulation parameter is specified as a range. While the simulation is in progress, random numbers are generated within the user specified bounds to dictate how different pieces of the simulation run. For example, the user might specify that all mazes generated should be at least 5 cells wide, but no more than 50 cells wide (using the *minDim* and *maxDim* parameters) and have between 2 and 6 possible solutions (using the *minPaths* and *maxPaths* parameters). The number of available solution models created by the *model generator* is a function of the number of layers as specified by the *layers* parameter. The user can also control the length of day and dream phases by specifying the number of mazes to process during each through the *min/maxDay* and *min/maxDream* parameters, respectively. Similarly, the total number of day/dream iterations is specified using the *cd-iter* parameter.

A particular parameter can be forced to a certain value by simply setting the minimum and maximum values to be equal. This allows the input data to the simulation to be narrowly constrained (e.g., so that only the ability to solve a specific subset of mazes is tested) or widely variable (e.g., to test the adaptability of the system).

4.2 Maze Generation

A CD system operates on “real world” input data during each day phase. Observed input data are then saved for use during the dream phase as various models are evaluated. In the case of this research, the “real world” data consists of randomly generated mazes. This random generation is done so that the CD implementation is exposed to a wide variety of different mazes within user specified bounds (e.g., dimensions and number of solutions to each maze). The goal is to ensure that the CD system does not converge on a model that, for example, is only able to solve mazes without loops, or one that is good at solving small mazes, but fails when presented with larger ones. Instead it ensures that the best models will be those with the capability to solve a broad range of mazes. This section focuses on the technical details of random maze generation.

Random values, governed by user specified limits, are generated to specify the height and width of each new maze. The choice to use random numbers was made because it allows the simulator to generate a wide variety of input data. Maze generation can be easily constrained by narrowing the difference between specified minimums and maximums or setting them to be equal, thus forcing generated mazes to have a certain characteristic. This is an important aspect to be able to control since the effects of input variability were unknown.

A grid of cells is generated as a starting point. All cells initially have four walls blocking passage in any direction. An algorithm based on depth-first search (DFS) is then employed to create a path through the maze. The algorithm chooses a random cell within the grid and removes a randomly chosen wall. It then moves into the newly available cell and chooses another random wall to remove. While the selection is random, it is governed by a few rules to ensure that a path is generated through the maze. Only neighboring cells that have all four walls intact are eligible for selection. If the algorithm reaches a point where such a selection isn’t possible, it backs up the travelled path to a point where an untouched neighbor *is* available. Once all cells have been explored, the algorithm terminates. The algorithm repeats this process of removing walls and visiting new cells until each cell in the maze has had at least one wall removed. Cells are only visited once and a wall is always removed at each step. At this point the generated maze has exactly one solution.

Figure 16: Example of a maze generated by the CD maze generator

A maze with only one solution isn't very interesting and not representative of real world problems that may have many potential solutions. To that end, the algorithm also features code that generates additional solutions. A number of cells are selected at random and each has a wall randomly removed to create a new path through the maze. The number of randomly generated paths for a given maze is governed by user specified maximum and minimum values.

Finally, an entrance and exit must be assigned to the maze. For the purposes of this research, this selection process has been simplified such that the entrance is always in the upper-leftmost corner of the maze (at location 0, 0) and the exit is always in the lower-rightmost corner (at location width-1, height-1). This was chosen to ensure that solutions would require several steps and would never be too easy (e.g., an exit that is only one or two moves from the entrance). Figure 16 shows an example of a maze generated using the algorithm described above. The "S" marks the entrance (start) cell and the "G" marks the exit (goal) cell.

4.3 Solution Models

The extensive enumeration of solution models, along with their logical organization, is a key feature of CD. All possible models (within a user specified bound) are enumerated and organized into layers of in-

creasing complexity. One of the challenges associated with developing models is determining the basic elements. The topic of how basic elements should be generated for specific algorithms and what impact they have on the performance of a CD system is beyond the scope of this research. In general however, they are derived from fundamental properties of time and space. For the purposes of demonstrating viability, the basic solution model elements have been arbitrarily chosen to be the four cardinal directions: north, east, south, and west. We considered these primitive spatial elements for mazes. Mazes are generated such that each cell can be entered and exited from at least one of these four directions.

Another challenge is to determine how many layers of models should be generated. There must be enough layers to ensure that a good solution will be found but not so many as to make the runtime requirements of the simulation impractical. The simulator allows the user to specify the number of layers to generate. An investigation of the optimal number of layers to generate is beyond the scope of this thesis. Four layers of models were generated for all experiments. Four layers were assumed to be sufficient for the purposes of this research as they provide over 65,000 different models.

A key feature of CD is that structure and organization guide how solution models are generated, dictating that models should be organized into layers of increasing complexity, similar to organization of neurons into layers found in the human brain [9]. The CD concept does not define how many models should exist within each layer or the details of how less complex models should be combined into more complex ones. For the purposes of this research, models on a given layer are generated by simply enumerating all possible combinations of two models from the next lowest layer. There are many other ways that models could be generated such as by combining three or four models from the next lowest layer or combining models from several different lower layers. Indeed, it is possible that such combinations could produce better results than the chosen approach. Always combining two models creates a wide range of models with varying complexity and is sufficient for an initial investigation and evaluation of CD.

Models on the lowest layer, layer 0, each contain only one of the four steps (thus there are only four models in layer 0). Subsequent layers are populated by models consisting of a combination of two models from the previous layer. That is, the models on layer n are combinations of models on layer $n-1$. Thus models on layer 1 contain two steps; those on layer 2 contain four steps, and so on. The number of steps

in a model, s , on a given layer, n , can be calculated as shown below in (3). The number of models, m , in a given layer can also be easily calculated using basic principles of counting as shown in (4).

$$s(n) = 2^n \quad (3)$$

$$m(n) = 4^{s(n)} = 4^{2^n} \quad (4)$$

As an example, consider a CDS with two layers. The first layer, layer 0, would have four solution models, each with one step (Figure 17). The second layer, layer 1, would then be composed of all possible combinations of two steps from the first layer, resulting in 16 different models (Figure 18).

Figure 17: Layer 0 models

Figure 18: Layer 1 models

Models on layer 1 are combinations of models from layer 0, while models on layer 2 are combinations of models on layer 1 and so on. This is inspired by the way that neurons in the brain aggregate and/or combine information from neurons on lower layers [9]. Figure 19 shows a partial example how layer 1 models are constructed from those available on layer 0. It shows all possible combinations of models from layer 0 in which N is the first step.

Figure 19: Partial example of how layer 0 models combine to form layer 1 models.

The concept shown in Figure 19 is applied to all other layer 0 models such that there are four models with N as the first step, four with S as the first step, and so on for a total of sixteen models. Figure 20

shows a complete depiction of how layer 1 models are derived from those on layer 0.

Figure 20: Example layer 0 and layer 1, showing how layer 1 models are constructed from those on layer 0.

The simulator developed for this thesis implements a layering scheme whereby higher numbered layers contain more complex solution models. Models are automatically generated such that each layer is completely populated with all possible solution models with respect to (3). Since this research is focused on evaluating the viability of CD in the context of mazes, all models that will be available during the lifetime of a simulation are generated up front; no new models are created, nor are existing models recombined during the course of a simulation.

Significantly, while models may seem to require significant connectivity amongst layers, the representation is for initial model creation which occurs only once at the start of the CD process. It is meant to depict how complex models are derived by combining and recombining simpler ones. Once models are established the connectivity between layers is not a factor. Similarly, the brain has connectivity that aggregates information from lower layers of neurons into more and more complex concepts in higher layers. The potential for connectivity seems to be the dominant factor in human intelligence, to include pruning when the best models are chosen.

4.4 Solving Mazes with CD

One of the decisions that must be made in the design of any CD system is how solution models will be used to process input data. This section focuses on details concerning how the maze-specific CD implementation developed for this research makes use of available models. Figure 21 depicts the flow of information between different parts of the developed simulator. Solid lines show the flow of data during day phases while dashed lines show the flow of data during dream phases. Circles represent the different solution models used by agents during simulation. The circle within the “day agent” box represents

the selected solution model based on aggregate dream results as they were after the most recent dream phase. Note that the flow of information is such that all solution models across all layers can be evaluated simultaneously in parallel and without any interaction with each other. This effectively implements an MISD system and it is this functionality that enables a CD system to scale nearly perfectly. The only overhead required is to give all models access to input data at the start of a dream iteration and to collect results at the end.

Figure 21: Data flow within the CD portion of the simulator. Dashed lines represent the flow of information during a dream phase while solid lines represent data flow during a day phase.

Recall that, as depicted in Figure 15 on page 41, the solution models are generated separately from the CD process at the beginning of a simulation. In doing so, the CD system builds all the different instruction sets (the “multiple instruction stream” part of MISD) to operate on common maze input data (the “single data stream” part). Models are generated in such a way that all possible combinations of a given complexity (that is, in a given layer) are created. This is where CD attempts to take advantage of structure in order to facilitate more intelligent processing.

Once all the input models have been created the day and dream phases that make up the CD process begin. The day phase consists of a single solution model being used to solve a random number of randomly generated mazes. As described earlier in this chapter, the number of mazes that are generated, as well as the characteristics of the maze (size, number of solutions) are random within user specified bounds. The observed portion of a maze must be stored to what is effectively a database of known mazes for use during the dreaming phase. This database essentially represents the system's "memory," storing pieces of problems that it has seen before. This is meant to mimic the way that the brain's hippocampus stores imagery and experiences from daily life and uses them later in the formation of dreams [41]. The first time the day phase is executed a solution model is chosen randomly from all available models and used to attempt to solve all mazes for that day. Subsequent day phases will use the solution model chosen by the model selector after each dream phase completes.

Once the day phase is completed, the dream phase begins. During the dream phase, mazes from the database of known mazes are randomly selected and presented to *all* available solution models simultaneously. As with the number of mazes generated during the day, the number of mazes that are attempted during dream phases to solve at night is randomly chosen within user specified bounds. For each solution model that has been enumerated, statistics are kept on how many mazes the node has attempted to solve, how many it has solved successfully, how many it has failed to solve, and the average number of steps in each completed solution.

At the end of a dream phase, the best performing solution model is selected for use in the following day phase. One of the aforementioned challenges with CD development is determining how the "best" model is defined. There are many possible ways that this could be done; it could be the model that generates the fewest steps, the one that finds a solution the fastest, or the one that solves the most mazes. For the purposes of this research, the goal is to demonstrate that CD is able to successfully develop solutions to a problem. Thus, the "best" model is defined as the one that has solved the most mazes. If there is a tie between two or more models, then the model with the lowest average number of steps per solution is selected. In the case where two or more models have the same number of solved mazes and the same number of average steps per solution, then they are essentially equal and so it is a reasonable choice to simply select a model at random from among these essentially equivalent options.

The simulator continues executing the CD process for a user specified number of iterations. Over time, the dream phases will converge upon a “best” model for use during the day phases. The idea is that this allows CD to improve its abilities through dreaming just as the brain does [11, 12] by playing back previously seen events from the day [10] and allowing many different models to process the data, better and better models will be selected over time.

4.4.1 Measuring Progress

Some mazes cannot be solved by a given solution model. Therefore it is important for a CD implementation to be able to determine when a solution model will not be able to successfully solve the current maze so that it can give up and avoid wasting time and resources. For the purposes of this thesis’ investigation into the viability of CD, the logic used to determine whether or not it should give up trying to solve a maze is relatively straightforward.

An agent does not know anything about a maze besides where it has been and what cells it has observed as being adjacent to cells it has visited. Additionally, if an agent attempts to move in a given direction but is blocked by a wall, it remains in its current location and attempt to execute the next step in its solution model. Thus, an agent knows when an attempt to move to a different location has failed. Finally, while an agent’s knowledge of a maze is limited, it *does* know how many steps are in its solution model. These four pieces of information (number of steps in the solution model, if an attempt to move was successful, what cells have been visited, and what cells have been observed but not visited) can be used together to determine whether or not an agent should give up. As the agent attempts to execute the steps in its solution model over and over, it keeps track of three pieces of information:

1. How many moves have been attempted since the agent successfully moved to a different location
2. How many moves have been made since the agent last visited a cell that was previously unvisited
3. How many moves have been made since the agent last observed a new, previously unknown cell in the maze

A count is reset to zero whenever a successful move to a different location is made, a previously unvisited cell is visited, or a new cell is observed respectively. The agent gives up if either:

- The number of steps executed without moving to a different cell equals or exceeds the number of steps in the solution model

- Both the number of steps executed since a previously unvisited cell was visited AND the number of steps since a previously unknown cell was added to the known map equal or exceed the number of steps in the solution model

This is a valid criteria for the first case as it is easy to see that an agent that has not been able to move to a new location after attempting to execute all available steps will not succeed in moving by executing the same steps again. The second case ensures that an agent does not get stuck in an infinite loop by continually moving through the same few cells over and over again. The third case functions as an extension of the second case. It attempts to ensure that the agent is at least observing new cells as it moves through the maze, even if it is backtracking through previously visited locations. If the agent observes no new locations nor has it visited any previously unvisited locations after executing all the available steps at least once, it is reasonable to conclude that no real progress is being made with the current model and thus the agent can safely give up.

If an agent gives up, no solution is generated and an attempt with no success is stored in simulation results. If the agent gave up during a dream phase, then nothing more happens. If it gives up during the day phase, however, the observed portion of the maze is stored for use during the dream phase. It is first “closed up” to ensure that cells that were observed but not visited have walls blocking all unknown cells. An exit is also artificially created such that models have a chance to find success during the dream phase. The process for preparing and storing this data is described in the next section.

4.4.2 Gathering Data

During the day phase, whether an agent gives up or finds the maze’s exit it always stores a map of the maze based on what it has observed. This allows for the collection of maze data to be used during the dream phase in order to evaluate the performance of other models, possibly finding that another model is able to find a shorter path. For the purposes of this research, the agent can see one cell beyond its current location in all directions. When the agent has completed its attempt at solving the maze, it has knowledge of only the portion of the maze it observed. This partial maze must be “closed up” such that there are walls blocking passage to portions of the maze that were not observed and are thus unknown. This creates a “complete” maze that can then be examined by other agents using different models. It is this closed up maze that is stored for usage during the dream phase.

As an example, consider the maze from Figure 16. This maze is redrawn in Figure 22 and color coded to show which portions of the maze were visited (white), observed (lightly shaded), and unobserved (dark shading) by an agent. Cells shaded in white and light gray would be stored as the “known maze” for use in a subsequent dream phase. It is this partial maze that would be closed up by the CD implementation. Consider the cell marked “A” in Figure 22. The agent would add walls to the West and North sides of this cell as they lead to unobserved portions of the maze. Figure 23 depicts the partial maze that the agent would store based on its observations.

Figure 22: Maze from Figure 16 solved by a model with steps: S, E, S, E, N, E, S, E. Cells shaded white are those visited by the agent. Cells shaded light gray are those observed, but not visited. Cells shaded dark gray are unknown to the agent.

Many times during a day phase an agent may not find any solution to a maze at all. In these cases it is still desirable to make the solution useful during a dream phase so that it can be used to test available models. Thus CD can determine which other models are able to make at least as much progress as the model that originally attempted the maze during the day. Thus models which are not as good will be less likely to be selected while those models which are at least as good as a previously selected model remain strong contenders for future selection. In order for this to happen, however, it is necessary to make the maze solvable. This is done by assigning an exit cell. For the purposes of this research, when

an exit cell is not observed during the day, the lower-rightmost cell in the observed portion of the maze is marked as the exit for dreaming purposes.

Figure 23: Observed portion of maze stored by agent for use while dreaming.

4.5 Simulation Parameters

There are a number of parameters that can be used to control and monitor the behavior of the simulation. The major parameters relevant to the experiments described later in this thesis are briefly discussed below. A full listing of configurable parameters, along with default values, can be found in Appendix B.

First, there are several parameters for which the user can specify minimum and maximum bounds (inclusive). These include maze dimensions (*min/maxDim*), number of solutions per maze (*min/maxPaths*), the number of mazes to generate during a day phase (*min/maxDay*), and the number of mazes to attempt to solve during a dream phase (*min/maxDream*). As the simulation executes, random numbers are generated such that they fall within the specified bounds. The user can effectively eliminate randomization by specifying the same value for both the minimum and the maximum. For example, if the user specifies that both the minimum and maximum maze dimension is 15, then only 15x15 mazes will be generated during day phases.

There are also several parameters that the user can specify up front as fixed values, not subject to randomization. These include the number of layers of solution models to generate, the number of CD iterations to execute, and the number of threads to use during the dream phase. These values can have a significant impact on the length of time it takes a simulation to run. For example, more layers mean that dream phases will last longer as there are more solution models to evaluate. The simulator uses multi-threading to take advantage of multiple processor cores in order to speed up simulation time. Each thread is assigned a number of solution models to evaluate during a dream phase such that all cores in the system are doing more or less the same amount of work. Care must be taken when setting this parameter as making it too large or too small will yield sub-optimal performance and can make the simulation take longer than necessary to execute. The optimal number of threads to use depends on the hardware available in the host system and some experimentation may be necessary to determine an appropriate value for this parameter. Given the computer used for the experiments described in this thesis, the optimal number of threads was empirically determined to be eight.

4.6 Experimental Overview

Finding a solution to a maze is essentially a graph theory problem; demonstrating that a given system is able to solve a maze suggests that it can be effectively applied to other areas of graph theory. If it can be shown that the maze-specific CD implementation can reliably select optimal maze solving models then it is reasonable to hypothesize that the CD approach could be used to solve more complex problems well beyond mazes. In order to test the viability of CD, experiments were conducted to determine if a) the system can reliably select a solution model within a reasonable amount of time and b) if the model that is selected mimics a known maze solving algorithm (e.g., if it produces solutions that are essentially equivalent to what a breadth first search would produce).

Since CD is a new concept and, as described earlier in this chapter, no previous implementations have been constructed prior to this research. There is currently no way to predict what effect each simulation parameter will have on the results produced. As a result, experiments were designed to explore a variety of parameter settings. The experiments executed for this research can be divided into three categories:

1. Experiments designed to test whether or not CD truly is a viable method of solving mazes
2. Experiments designed to test the robustness of the CD implementation

3. Experiments designed to test the repeatability of CD

The following chapters discuss the experiments and results for each of these categories in more detail

Chapter 5 Goals

While scalability to levels of complexity and parallelism found in the human brain is a primary motivating factor of CD, proving unlimited scalability is beyond the scope of this thesis. Viability of the CD approach in solving mazes is more tractable, but its significance could be deemed to be limited in scope. Therefore since CD is an approach to automatic algorithm development and is therefore in the broad category of AI techniques, demonstration of the robustness of CD in not being susceptible to getting stuck in local minima and thus requiring human intervention is a major goal of this work. As discussed in previous sections of this thesis, existing biologically inspired computing methods such as genetic algorithms and artificial neural networks offer a number of benefits but are each subject to becoming trapped in sub-optimal solutions or stuck completely. Since a key attribute of intelligence is recognition of each of these conditions, a primary goal of this work is to demonstrate that Computational Dreaming is not subject to these limitations.

For genetic algorithms a significant problem is premature convergence whereby the algorithm, due to the random chance involved in mutation, gets “stuck” on a sub-optimal solution and is unable to recover [24]. In the case of neural networks, a significant problem is that the algorithm is subject to becoming trapped in local solutions, depending upon the weights selected during the training process [42]. For example, Sexton and Gupta note that when using backpropagation to train a neural network, “If these initial weights are located on a local grade, which is probable, the [backpropagation] algorithm will likely become trapped in a local solution that may or may not be the global solution [42].” When this happens, it is up to a user to recognize that the algorithm has become stuck and initiate partial or full retraining. With neural networks, it may even become necessary to adjust the network design and try again.

An algorithm that gets stuck is, at best, forcing the user to waste time on retraining efforts that will, hopefully, not also become stuck. At worst, it could mean (potentially extensive) redesign of the algorithm itself. In both cases human supervision is required, which is clearly a serious limitation for AI algorithms. True AI would not require any human oversight and thus would not be prone to getting stuck. A robust algorithm that does not get stuck and thus requires no human intervention would possess the potential to be a true AI technique. To be successful, a technique must not only be robust, but also via-

ble and applicable to a wide range of problems. Thus, this research has three goals regarding the maze solving CD implementation:

- The primary goal: to show that it is **robust**
- To show that it is **viable**
- To show that it is **flexible**

The primary goal of this research is to demonstrate that CD is not subject to getting “stuck” on a sub-optimal solution due to random chance or poor selection of initial parameters. Doing so would be a key piece of evidence that CD is capable of enabling a truly artificially intelligent system. For the remainder of this work, algorithms that do not get stuck will be referred to as robust. Both the CD simulator and the maze-solving CD implementation make use of some randomness. Chapter 4 describes how the simulator randomly generates input mazes during each day phase within the bounds of user supplied simulation parameters. The robustness of the maze solving CD implementation is examined by executing repeated CD simulations, all with the same parameters. Though each input maze will still be randomly generated, the parameters are set up so as to constrain generated inputs to a narrow range of size and complexity. This is done in order to minimize the amount of randomness in the simulation and ensure that all trials generate mazes with the same characteristics (e.g., 6x6 and five solutions). Forcing all trials to generate similar mazes makes it easier to evaluate the consistency of CD’s performance.

A similar set of trials will then be executed using random solution model selection instead of the more intelligent CD process. We define robustness as the ability of CD to reliably produce successful results despite these elements of randomness. Thus, CD would not be subject to the limitations of genetic algorithms and neural networks. While CD was not investigated for algorithms other than maze solving, the results suggest that any CD implementation would be unaffected by such limitations and thus CD is a candidate for a robust computing technique.

A technique that does not get stuck is not useful unless it also can find success over a wide range of problem sizes and complexities. To this end, another goal of this work is to demonstrate the viability of CD. Viability is examined by executing a number of trials with parameters that permit generation of a wide variety of input mazes. A parallel set of trials is executed in which solution models are selected randomly instead of using CD. Showing that maze solving CD consistently and significantly outperforms random selection indicates that it is viable, at least in the context of mazes.

While this thesis deals only with maze solving, the goal is to lay the foundation for broader applications of CD. There is nothing maze specific about the core CD processes, however. The models and evaluation methods developed for this research are maze specific, but the algorithms that drive day and dream phases are ignorant of the specifics of what models contain and the contents of model evaluation and usage functions. This does not prove CD will always be a viable technique but suggests that CD will also be viable in other applications well beyond maze solving.

Finally, the effect of simulation parameter values on simulation outputs is examined. By using simulation parameters that result in generated mazes ranging from small and simple to large and complex, the goal is to show that the CD maze solver is flexible and consistently able to find some measure of success. The setup for this experiment is similar to the experiment in robustness except that three sets of tightly constrained parameters are used instead of one. As described above, there is nothing special about the core CD processes that enable it to find success with mazes. Demonstrating that the CD maze solver can find at least some success when processing data of various sizes and complexities suggests that CD can be applied to a wide range of data processing applications. This would suggest that any CD system, regardless of application, would exhibit the same flexibility.

Chapter 6 Experimental Setup

This chapter discusses the experimental setup used to fulfill each of the goals described in the previous chapter. Three experiments are discussed, one for each of the goals of this research. While each experiment uses the same general setup, simulation parameters are different for each scenario, allowing results analysis to better focus on a particular goal.

The general setup shared by all experiments is depicted in Figure 24. The boxes at the bottom of the diagram encapsulate the parameters that were changed for different experiments. The parameters that affect generated input maze size are *minDim* and *maxDim*. These control the minimum and maximum maze dimensions respectively. The density of mazes is controlled by parameters which specify how many possible solutions each generated maze has. These parameters similarly define minimum and maximum values and are referred to as *minPaths* and *maxPaths*. The *layers* parameter controls the number of layers of solution models generated by CD. The *cd-iter* parameter controls how many total day/dream iterations are executed. At the end of each simulation, the overall day phase success rate is recorded and compared to the success rates of other simulations.

Figure 24: High-level view of experimental setup.

6.1 Viability

A technique is viable if it has the ability to consistently find success over a wide range of inputs. By definition, a technique that isn't viable is not very useful in real world situations. The first experiment pre-

sented in this work focuses on the viability of CD. Viability is investigated by running multiple trials using simulation parameters that allow for a wide range of maze sizes and complexities to be generated. Two sets of trials are run: one using CD to determine the best solution model and one using random chance. Showing that the maze solving CD implementation consistently and significantly outperforms random chance demonstrates that it is viable.

Obtaining a successful result from the experiment will demonstrate viability. Here, a “successful result” is one in which the system was able to select a solution model that was able to solve one or more mazes during one or more day phases. Simply showing that CD is able to select models that solve mazes during day phases does not, in and of itself, mean that CD is viable. Therefore a second experiment was performed in which exactly the same simulation parameters were used but instead of attempting to intelligently select solution models using a dream phase (i.e., using CD) a new model was selected randomly at the end of each day phase. The proportion of mazes solved by CD is then compared to the proportion of mazes solved by the random selection process using a two proportion statistical test. Conceptual results are shown in Figure 25. Once viability has been shown, a brief analysis focusing on whether or not a particular model is clearly “best” and how closely CD generated solutions resemble those generated by established maze solving algorithms is presented.

Figure 25: Conceptual viability experiment results

As described previously, there is no way to quantify the effect of each parameter on the outcome of the simulation, and some situations are more significant than others, regardless. Enumerating as many layers as possible, and thus enumerating as many different ways to solve a maze as possible, would in-

crease the chances of success. For this research, all experiments make use of four layers of models. No additional layers were used due to constraints imposed by certain data structures used inside the simulation software, as well as available memory. This was not seen as a problem since four layers contain a total of 65,812 solution models, a number which was deemed to be more than sufficient to achieve the goals of this research.

Similarly, observing more mazes during day phases and examining as many mazes as possible during dream phases (ideally, such that all known maze information has been examined) would logically seem to maximize chances for success. With all this in mind, the simulator was programmed to randomly generate between 25 and 40 new mazes during each day phase, and to examine all known mazes during dream phases.⁶ Maze generation parameters were chosen such that mazes of small to relatively large size (from 4x4 to 20x20) and complexity (3 to 10 possible solutions) would be generated. A total of ten day/dream iterations were executed for each trial run, with a total of ten trials performed. These values were arbitrarily chosen in order to ensure that the CD simulations would be exposed to a wide variety of different problems, but would still finish within a reasonable amount of time (i.e., no more than a few hours of simulation time on the computer used for simulation⁷ which is consistent with the amount of time a computer systems might spend dreaming while its owner is doing the same).

Finally, maze generation parameters must be specified. These parameters govern the dimensions and number of possible solutions that each randomly generated maze has. Recall from Chapter 4 that these parameters are specified as a range and that a random value within the range is selected for each generated maze. Maze generation parameters were, like other simulation parameters, chosen to maximize the chances of success. Maze dimensions were specified to be at least 4x4 but no more than 20x20 with between 3 and 10 possible solutions. These values were chosen in an attempt to ensure that mazes would not be too easy or too difficult. Obtaining results that show that the CD maze solver consistently

⁶ The minimum and maximum number of mazes to examine while dreaming was set to 40. The CD implementation used for this research will thus examine a maximum of 40 mazes while dreaming. If all available known maze data is examined before 40 mazes have been attempted, the dream phase will stop.

⁷ The computer used for simulation runs Windows 7 64-bit on an Intel Core i5-2500 (a quad core processor) at 3.3GHz with 8.0GB of RAM.

finds significantly better success than random chance indicates demonstrates that the maze specific implementation is viable. This, in turn, suggests that CD as a technique in general is viable.

6.2 Robustness

As described previously, the primary goal of this research is to demonstrate that maze solving CD is robust and not subject to getting stuck in contrast to neural networks and genetic algorithms. By showing that a maze specific CD implementation is robust, it suggests that any CD implementation would also be robust. Thus a CD system would be able to function successfully without the need for a human in the loop, potentially creating a path to a true AI system.

In order to demonstrate this, a series of identical trials were executed. All iterations used the simulation parameters shown in Table 2. While all mazes generated were 6x6 and had five possible solutions each individual maze was still randomly generated based on a random seed that was different for each simulation.

Table 2: Robustness experimental simulation parameters

	Parameter	Value
Input Generation	<i>min/maxDim</i>	6
	<i>min/maxPath</i>	5
	<i>min/maxDay</i>	30
Operational	<i>min/maxDream</i>	30
	<i>layers</i>	4
	<i>cd-iter</i>	10

Success rates would ideally all be identical but due to the random nature of the maze generation process, some variation is to be expected. Moreover, it is expected that CD will *always* produce results that significantly outperform random chance. A graph of the expected results is shown in Figure 26.

Figure 26: Expected robustness experimental results

After executing ten trials using CD, an additional ten trials were executed with the same input parameters. Instead of using CD to intelligently select a solution model for use during each day phase, however, a new model was randomly selected at the start of each day phase. Random selection was performed such that each available solution model had an equal chance of being selected for daytime use. This was done to provide a basis for comparison of consistency; that is, how consistently CD finds success versus the consistency of random selection. Demonstration that maze specific CD is always able to select successful solution models at a significantly better rate than random chance with the same simulation parameters and over repeated trials shows that it is not prone to prematurely converging on local minima as a result of randomness in either input data or the CD process itself.

CD has a key advantage over neural networks and genetic algorithms in that the dreaming process makes use of an algorithm to deterministically select the best performing models despite the randomly selected initial model. This suggests that CD is a technique that not only does not require a special training phase, but also requires no continual monitoring. This is the primary contribution of this work. A system designer can make use of CD and be assured that the system will always find a successful solution model and will continue to select successful models indefinitely.

6.3 Flexibility

While the other experiments discussed in this work focus on showing the robustness and viability of the CD approach, neither examines the performance of the maze-solving CD system in depth. In other words, they look at the system's performance in a broad sense without any focus on how individual

simulation parameters affect results. A data processing technique that is only successful when used on a narrow range of data or problem sets (e.g., small and simple or large and complex) is not very useful. The goal of this experiment supports previous goals as it seeks to demonstrate that the maze solving CD implementation can find some success across a wide range of inputs. Simulation parameters are set to force the generation of mazes that are (relatively) small and simple, large and complex, and somewhere in between. In doing so, a case is made that any system using CD would share the same traits. This, in turn, suggests that CD is a flexible technique for use in a wide variety of applications.

Bagnall and Zatuchna note that maze size, density, and distance to food (or in the case of this research, distance to exit or goal) are the most important characteristics that determine how difficult a maze is for an agent to solve [43]. Thus, parameters that affect the maze size, density, and distance to goal were chosen as the problem complexity variables for this experiment. Maze size was controlled through the *minDim* and *maxDim* parameters. The density of mazes was varied by adjusting the *minPaths* and *maxPaths* parameters, which control the number of solutions each generated maze has. The final factor affecting maze difficulty, distance to goal, was not directly varied. Since the day maze generator always places the maze entrance in the upper-leftmost corner and exit in the lower right, distance to goal is indirectly influenced by both maze size and maze density. In a more general sense, changing any or all of these parameters affects the size and complexity of input data presented to the CD system. While additional research is still required, showing that the maze specific CD implementation finds success with large and complex problems as well as with small and simple ones indicates that any CD system would show the same, robust problem solving characteristics. It is expected that much higher success rates will be achieved with simpler mazes, but maze difficulty is not the only input that can affect simulation results. Again, we anticipate that CD would exhibit similar results for non-maze applications.

For the purposes of this discussion only the overall day phase success rate is of interest. This is because the goal is determining whether CD can solve problems at all as opposed to how well or how consistently they are solved. A higher success rate means that CD was successfully able to determine which models would best solve mazes during each day phase. This indicates that CD is well suited to working with a wide variety of input data. Both the difficulty of input mazes presented to CD during day phases and the length of each phase were varied by adjusting simulation parameters. This was done to model a variety of problem complexities and available processing times respectively. The maze specific implementation

is able to find success given varied problem complexity and with varying amounts of available data. This suggests that the CD technique will find success beyond mazes.

Other important simulation inputs, specific to all CD implementations and not just mazes, are how much data is processed during each dream phase and day phase (*min/maxDay* and *min/maxDream* respectively). Producing more mazes during day phases gives dream phases more data to process. The expectation is that the more data that is processed during dream phases, the more data that the maze specific CD system has to refer to when selecting a new solution model and the better the model will be. The same expectation holds for any CD systems focused on other types of problems. Figure 27 shows a graph of conceptual results, depicting how it is expected that simpler mazes and more mazes per day/dream phase (i.e., more observation and dreaming time) are expected to yield better success rates.

Figure 27: Depiction of expected results. Simpler mazes and more data are expected to yield higher success rates. Number of mazes processed increases from left to right. Success rate increases from bottom to top.

Table 3 below summarizes the simulation parameters used for each of the six trials performed. Like the viability experiment, all trials used four layers of solution models and ten day/dream iterations. For this set of trials, the simulation was configured to always generate a specific number of mazes during day phases and to examine all known mazes during each dream phase using the *min/maxDay* and

min/maxDream parameters respectively. Thus only one number is listed under the “*min/max Day/Dream*” column. The “*Dim*” columns list the values for *minDim* and *maxDim* parameters controlling maze size. Similarly, the “*Paths*” columns list the values for *minPaths* and *maxPaths* parameters controlling maze density.

Table 3: Simulation parameters by trial.

Trial	Simulation Parameters				
	<i>Dim</i>		<i>Paths</i>		<i>min/max Day/Dream</i>
	<i>min</i>	<i>max</i>	<i>min</i>	<i>max</i>	
1	4	5	1	3	15
2					50
3	8	12	3	6	15
4					50
5	18	20	6	10	15
6					50

Ideally, the maze-solving CD implementation would be able to achieve success rates near 100% for all size and complexity combinations. Based on the results of the viability experiment in Chapter 5 and the notion that maze difficulty increases with size and is inversely proportional to density [43] a more realistic expectation is that trials 1 and 2 will produce the highest success rates and that rates will taper off as trials progress. Moreover, it is expected that processing more mazes in each dream phase will result in higher success rates for a given maze size/complexity. This is a reasonable expectation since processing more mazes means that CD has more “experience” in the form of background data to look at when determining what the best solution model is based on the maze characteristics observed during day phases. These expectations also hold for any CD system. Thus, it is expected that the best success rates would be found with relatively small and simple problems and that more available background information will only help to increase the rate of success.

Chapter 7 Experimental Results

This chapter presents the results of the experiments described in Chapter 6 along with a brief discussion of their implications.

7.1 Viability

The first experiment investigates the viability of the maze-solving CD implementation developed for this research. Viability of the CD maze solver means that it is able to find success across a wide range of data sizes and complexities.

The simulation statistics show how many mazes were attempted during day phases and how many of those were actually solved. Table 4 summarizes the results from each of ten trials executed using both CD and random model selection. These results ignore which specific models were selected and instead look at the overall success rates for each trial. The summary row at the bottom of the table shows the total number of mazes attempted and solved using each approach, as well as the overall percentage of mazes solved by each approach.

These results indicate that CD has a much better success rate than simply selecting a model at random; CD solved approximately 15% of mazes seen during the daytime while random selection only solved about 2.2%. A two proportion Z-test can be used to verify that CD is indeed a significant improvement over random model selection. This test is a statistical tool that can be used to determine whether or not two proportions (or means) are significantly different from one another. It calculates a normalized score from both proportions that can then be compared to a critical z-value to determine whether or not the two are significantly different. The critical value is a normalized value calculated based on what the test is attempting to demonstrate. Using 1% level of significance and attempting to show that CD results in higher success rates than random chance, the critical z-value is 2.33. Our calculated z-score is approximately 18.43. This means that we can say with 99% certainty that intelligent model selection using CD's dreaming process is significantly better than randomly selecting solution models. This confirms that the dreaming process employed by CD is indeed a viable approach to problem solving. Table 4 shows a detailed summary of results while Figure 28 charts the variation in overall success rates by trial.

Table 4: Viability test results summary

		CD			Random Selection		
		Attempts	Successes	% Solved	Attempts	Successes	% Solved
Trial	1	339	58	17.11%	316	4	1.27%
	2	329	31	9.45%	325	6	1.85%
	3	324	53	16.36%	313	10	3.19%
	4	312	63	20.19%	303	1	0.33%
	5	336	48	14.29%	306	14	4.58%
	6	304	56	18.42%	336	4	1.19%
	7	317	46	14.51%	361	15	4.16%
	8	323	31	9.60%	322	4	1.24%
	9	328	36	10.98%	333	2	0.60%
	10	297	58	19.53%	334	10	2.99%
Summary		3209	480	14.96%	3249	70	2.15%

Figure 28: Viability test results

Several other interesting aspects of the CD maze solver's performance were also examined as part of this experiment. One of these is whether or not there are any models that consistently perform better than all others. Table 5 shows a list of which solution models were selected during the viability experiment. Models are listed by layer and an arbitrarily assigned model number. The columns to the right of each model designator show the proportion of mazes that were successfully solved by each model during all day phases for which it was the selected model. Cells shaded gray indicate that a model was not selected during the respective trial run. A value of zero percent means that a model was selected for use

during one or more day phases, but that it did not successfully solve any mazes. When looking at this data, recall that the model selected for the very first day phase is selected at random.

Looking at the success rates for each model (and ignoring those with a success rate of zero), we note that most of the selected models performed significantly better (i.e., success rates greater than about 5%) than random chance in terms of the proportion of mazes solved. This suggests that, given enough time and input data, CD will eventually converge on a reliable model. That being said, it is also interesting to note that only two models (5862 and 6630, both on layer 3) were selected during more than one trial. Thus, there were no models that consistently performed better than other models. This is interesting because it suggests that there was too much variability in input and thus the system was unable to converge on any one particular model and/or that allowing the system to run for longer (i.e., increase the iteration limit) might result in more consistent model selection.

Table 5: Summary of models selected by CD during viability trials and their success rates. Gray cells indicate that a model was not selected at all during the respective trial.

Layer, Model	Trial									
	0	1	2	3	4	5	6	7	8	9
2,6								0.00%		
2,41										11.11%
3,102		0.00%								
3,358								0.00%		
3,406									0.00%	
3,1102									0.00%	
3,1641		12.06%								
3,4518								13.60%		
3,5862				32.00%						17.86%
3,6630				21.24%						23.36%
3,6839				0.00%						
3,7057							0.00%			
3,7993							0.00%			
3,13558		0.00%								
3,14850										0.00%
3,18361	7.69%									
3,19088					2.70%					
3,20693								0.00%		
3,22110			17.51%							

3,26118									13.69%	
3,26509			3.70%							
3,26574						0.00%				
3,26886					15.72%					
3,27790						0.00%				
3,28257	19.86%									
3,31110	21.26%									
3,31715	0.00%									
3,40545						22.49%				
3,42513							17.83%			

Another interesting question (and one that has been mentioned previously in this thesis) is whether or not any selected models produce results similar to a known maze solving algorithm. Regularly selecting solution models that mimic the behavior of known algorithms suggests that CD is able to, on its own, converge on known methods of solving mazes. For the purposes of this research, the solutions generated by selected models were compared to solutions that breadth-first search (BFS), depth-first search (DFS), and Dijkstra's algorithm would generate for the same input maze. Table 6 was generated using a tool designed specifically to compare solutions generated by CD to solutions that might be generated by one of these known algorithms.

Table 6: Summary of solutions found by CD and how they compare to known maze-solving algorithms. The total number of mazes solved during this experiment was 480.

Algorithm	# Solutions Matching	% Solutions Matching
BFS	240	50.00%
DFS	249	51.88%
Dijkstra	240	50.00%

It is important to point out that, in many cases, a given solution generated by CD matched a solution generated by more than one algorithm. That being said, these data show that nearly half of all solutions generated by CD are solutions that known algorithms would generate. This indicates that CD is capable of determining, without any guidance from a user or software developer, that certain methods will consistently produce better results than others.

7.2 Robustness

The second experiment focuses on the main contribution of this work: that the CD maze solver does not get stuck in contrast to neural networks and genetic algorithms. This, in turn, suggests that any CD implementation would exhibit the same robustness since there is nothing special about the maze-specific implementation that would affect the robustness of the technique.

As with the viability experiment, the number of mazes attempted and solved across all day phases is of particular interest for this experiment. Table 7 shows the results from all ten trials executed as part of this experiment while Figure 29 charts success rates for each trial. On average, nearly 32% of mazes were solved across all trials using CD. In contrast randomly selecting models to solve mazes yielded an overall success rate of about 4%. This shows that CD is consistently able to outperform random chance and indicates that it is not subject to limitations that afflict both neural networks and genetic algorithms.

It should be noted, however, that the overall success rates generated at the end of each trial showed a lot of variation, ranging from as low as 19.7% to as high as 41.3%. This wide variation in results was not expected but is not necessarily bad. Such variation could very well be due to the limited number of CD iterations that were executed as part of this experiment. Executing more iterations and/or processing more data per iteration may allow the simulation to converge on one or more solution models that perform better and more consistently.

Table 7: Robustness results summary

		CD			Random Selection		
		Attempts	Successes	% Solved	Attempts	Successes	% Solved
Trial	1	300	86	28.67%	300	7	2.33%
	2	300	124	41.33%	300	9	3.00%
	3	300	115	38.33%	300	15	5.00%
	4	300	98	32.67%	300	0	0.00%
	5	300	113	37.67%	300	35	11.67%
	6	300	70	23.33%	300	14	4.67%
	7	300	59	19.67%	300	0	0.00%
	8	300	124	41.33%	300	16	5.33%
	9	300	98	32.67%	300	11	3.67%
	10	300	63	21.00%	300	2	0.67%
Summary		3000	950	31.67%	3000	109	3.63%

Figure 29: Robustness test results

A two-proportion Z-test was used to determine the statistical significance of the CD success rates versus the success rates from random selection. The Z-test result yields a score of 28.5, well above the critical score of 2.33 for a 1% level of significance. In fact, performing the same statistical test comparing the worst performing CD trial (trial 7 with a success rate of 19.67%) with the best performing random trial (trial 5 with a success rate of 11.67%) shows that CD still performed significantly better than random selection (Z-score: 2.69).

In addition to the raw success rate, results analysis also examined whether or not some models tended to be more successful than others. Table 8 (page 73) shows which models were selected during each trial. Gray cells indicate that a model was not selected all while cells with a zero percent indicate that a model was selected but did not successfully solve any mazes⁸.

The data gathered during this experiment show that the maze solving CD implementation is robust in the sense that the maze solving implementation always had success rates significantly better than random chance. Since there is nothing maze-specific about the high level dreaming process employed by CD, these results also indicate that any CD implementation would not be prone to getting “stuck” due to random “bad luck” as a neural network or genetic algorithm might [24, 27]. This is a key advantage. It

⁸ A zero percent success rate for a given model could possibly be because it was randomly selected at the start of the simulation and thus not based on any empirical data.

suggests that CD systems will reliably select successful solution models despite any randomness inherent in either the surrounding environment or the CD system itself.

Also interesting to note is that no model was selected during more than one trial. This suggests that there is no one “best” model (or group of models) for solving mazes generated according to the simulation parameters used in this experiment. This is likely due to the relatively low complexity of the mazes and/or because the number of potential solutions generated for each maze allowed a wider range of models to find success. It is also possible that allowing the system to process more data during each dream phase and/or to run for more day/dream iterations overall would result in some models showing a clear advantage over others. Considering applications beyond mazes, this could indicate that CD may find a different “best solution” each time it runs. While all solutions may be technically correct, such variation from run to run may not be tolerable in certain applications. It also means, however, that CD may be able to find new “best” ways of solving problems that have not yet been discovered. If so, this means that CD systems may make significant contributions to many areas of computing, including (but not limited to) solving previously intractable problems as well as finding better algorithms to process NP-hard problems. Additional investigation into these areas is beyond the scope of this work and is left to future efforts.

Table 8: Summary of models selected by CD during robustness trials and their success rates. Gray cells indicate that a model was not selected at all during the respective trial.

Layer, Model	Trial									
	1	2	3	4	5	6	7	8	9	10
2,6										0.00%
3,358						0.00%				
3,479										0.00%
3,617								10.00%		
3,850							0.00%			
3,1005	0.00%									
3,1154				0.00%						
3,2469								30.00%		
3,2705	16.67%									
3,4507							0.00%			
3,4518										30.00%
3,5286			36.67%							
3,5782			36.67%							
3,6630								53.33%		
3,7062		40.00%								
3,11531								0.00%		
3,11814		0.00%								
3,11846					20.00%					
3,17833									36.67%	
3,20905									33.33%	
3,21938				31.67%						
3,22130				37.62%						
3,22180									30.00%	
3,22806						29.17%				
3,22826			33.33%							
3,22861						0.00%				
3,24990			39.52%							
3,25686							24.58%			
3,27718										0.00%
3,28257		53.33%								
3,39825					39.63%					
3,40545		46.67%								
3,43317	33.75%									

7.3 Flexibility

The final experiment performed as part of this research examines the effect of simulation parameters on end results. While the previous two experiments focus on the viability and robustness of CD, neither examines its ability to deal with data sets of various size and complexity in any detail. This experiment focuses on showing that CD is effective on a wide range of data set sizes and complexities as a supporting goal. This is referred to as flexibility.

Again, the only simulation statistics that were of concern are how many mazes were attempted during day phases and how many of those were actually solved. These results ignore which specific models were selected and instead look at the overall success rates for each trial. This was done because the focus is on whether or not the maze solving CD implementation is able to solve mazes at all as opposed to the quality of solutions. Results were examined to see how success rates are affected by maze size and complexity as well as how they were affected by the number of mazes processed while dreaming. In a more general sense, the focus of results analysis is on how well CD deals with problems of various difficulties and sizes (from small and simple to large and complex) and on how much data is available to examine during dream phases.

Figure 30: Graphical representation of varied input parameter experiment.

Figure 30 shows a graphical summary of results while Table 9 shows more detail. The graph shows that the highest success rates were achieved with the small, simple mazes. Moreover, processing fifty mazes per dream phase yielded improved success rates, at least with simple mazes. One interpretation of this

result is that the maze solving CD implementation, as set up for this experiment, is clearly well suited to solving relatively small and simple problems and that, as expected, success rates improved with longer dream phases. It is likewise expected that CD systems in general would exhibit the same behavior, no matter what the problem at hand. Indeed, even longer dream phases would likely improve success rates further.

Success rates fell as mazes grew more complex (larger and denser). Interestingly, in the last four trials the percentage of mazes solved during day phases fell slightly as the number of mazes processed while dreaming increased. Since the difference between trial success rates is small ($\leq 2\%$), this is likely because larger and more complex mazes are pushing up against the limits of what the four layers of solution models used in this research can handle effectively. Indeed, it is reasonable to expect that large and complex problems presented to any CD system, whether it be designed for mazes or some other type of data processing, will require the availability of larger and more complex solution models. As a result, processing more data does not have the same positive effect that it does on smaller, simpler input data sets. It is reasonable to say that this phenomenon is not specific to solving mazes. Any simple problem would likely be solved by relatively simple models while complex problems would require complex models to solve efficiently.

Table 9: Parameter Range Test Results Summary

Trial	Maze Parameters			Results		
	Dim	Paths	Generated / Examined	Attempts	Successes	% Solved
1	4-5	1-3	15	150	59	39.33%
2			50	500	266	53.20%
3	8-12	3-6	15	150	24	16.00%
4			50	500	70	14.00%
5	18-20	6-10	15	150	1	0.67%
6			50	500	3	0.60%

The fact that the large mazes saw very low success rates overall ($< 1\%$) suggests that mazes of this size are at the limit of what a four layer system can solve. This does not, however, mean that CD is poorly suited to solving large and/or complex problems. A likely cause for the lack of success with large mazes is that the four layers of solution models generated for these experiments were simply not enough to solve larger and more complex mazes. As stated above, it is reasonable to expect that any CD system would require the availability of more complex solution models in order to successfully solve more com-

plex problems. Another reason could be that perhaps the known portion of mazes stored during day phases did not contain enough data for dream phases to adequately determine which model was best suited for solving larger and more complex mazes. Indeed, it would be difficult to determine the best way to solve any type of problem (not just mazes) if only limited observation and experience was available. Thus, while it has been shown that the maze solving CD implementation can find success across a range of inputs, there is still additional work to be done in order to fully characterize the simulator's performance.

7.4 Summary of Experimental Results

The goal of showing that the CD maze solver is viable by demonstrating that it consistently and significantly outperforms random solution model selection was achieved successfully and presented at the beginning of this chapter. Results show that CD was able to select solution models that correctly solved 15% of daytime mazes. In contrast, only 2.2% of mazes were solved when solution models were selected at random. Next, the experiment focusing on demonstrating robustness (the primary goal of this research) was presented and discussed. Results show that the maze solving CD implementation consistently and reliably selects solution models that significantly outperform random chance. The CD system solved about 32% of day mazes across all trials while random selection solved only about 4%. These results show that maze-solving CD does not "get stuck" and suggest that any CD implementation would share the same robustness. Finally, experimental results concerning the supporting goal of showing CD flexibility were presented. Results from this experiment indicate that CD is indeed a flexible technique. Some success was found across a wide range of data sizes and complexities. While additional research is needed to more fully investigate the capabilities of CD, these results demonstrate that it is a robust, viable, and flexible technique that merits further study.

Chapter 8 Conclusions and Future Work

This thesis has demonstrated that computational dreaming (CD) is a robust, viable, and flexible approach to finding efficient ways to solve small mazes. In doing so, it demonstrates that not only is CD a worthwhile technique for use in solving mazes and maze-like problems, but that it is likely to find success in a variety of other application areas as well.

The primary contribution of this work is demonstrating that a maze-specific CD implementation is robust, does not get stuck, and thus does not require human intervention—unlike neural networks and genetic algorithms. Results showed that on average approximately 32% of day phase mazes were solved across all experimental trials. In contrast, random solution model selection solved only about 4% of all mazes. This also suggests that any CD implementation would share this same robustness. By not requiring any human intervention or oversight in order to find success, CD has the potential to enable systems that are truly artificially intelligent.

In analyzing the results of the robustness experiment, it is interesting to note that there were not one or two solution models that were clearly superior to others. One implication of this finding is that systems using CD may be able to find new “best” ways of solving problems that humans have not yet discovered. This suggests that CD may find success in many areas of computing, including research regarding algorithms to work on NP-hard and/or NP-complete problems.

The goal of demonstrating the viability of CD was also met and is one of the supporting contributions of this work. Experimental trials were executed in which a wide range of input size and complexity was presented to the maze-specific CD implementation. Results showed that the CD system was able to solve approximately 15% of all day mazes, significantly more than the 2% solved by random chance. These results show that the maze solving CD implementation is viable as it is consistently able to find success even when presented with input data that varies across a relatively wide range of size and complexity. Since there is nothing maze specific about the fundamental CD process these results also suggest that any CD implementation could also be viable.

Results from the viability experiment also looked at how the solutions found through CD model selection compare to those that would be found by well-established algorithms (BFS, DFS, and Dijkstra's algorithm). Analysis showed that approximately 50% of all solutions generated by the CD implementation matched those that would be found by a known algorithm (approximately 52% in the case of DFS). This is significant as it indicates that CD is capable of determining, without any human intervention, that certain methods will consistently produce good results.

The last experiment executed focused on the goal of showing flexibility. Here, experimental trials focused specifically on three data sets: small and simple, large and complex, and those of moderate size and complexity. In all cases, the CD maze solver was able to find some success. While the degree of success varied over a wide range, the results indicate that CD has the potential to be successfully applied to a variety of applications, regardless of how large or complex the input data might be. Thus another supporting contribution of this work is the demonstration that maze solving CD is flexible, suggesting that any other CD implementation would share a similar flexibility.

The demonstrations of the viability, robustness, and flexibility of this maze-specific system are important contributions as they indicate that the maze-specific CD implementation would find success in real world applications. There is nothing maze-specific about the way the CD technique operates, however. Therefore these contributions also suggest that non-maze specific CD implementations would also find success. Thus this work provides a foundation for future research on the CD technique.

8.1 Future Work

The scalability of a CD system is a key topic of interest. Previous chapters in this thesis described how CD is theoretically nearly perfectly scalable, but stopped short of implementing a system to test how scalable it really is. The implementation of a CD system on a machine with more processing resources than the one used for this research would provide invaluable insight into the scalability of our approach. Demonstrating near perfect scalability would constitute an important contribution to the field of parallel computing and is recommended as the next CD topic of investigation.

Another important topic for future research is the application CD to problem sets other than mazes in order to examine how well other types of problems are handled. Earlier chapters of this thesis describe how CD offers a number of advantages over biologically inspired methods of computing such as genetic

algorithms and neural networks. An interesting avenue of research would be to implement a CD system that solves the same problem as a neural network of genetic algorithm enabled system and then compare the results. Empirical results showing that CD can perform as well as or better than other techniques, in addition to not being subject to the same pitfalls, would provide solid evidence that not only is CD superior to other biologically inspired methods but also that it has the potential to enable powerful new intelligent systems.

Related, but perhaps less obvious, is whether or not CD can successfully develop its own new solution models over time based on day phase experiences and dream phase experimentation. In doing so, the CD system would not be constrained by the solution models generated at the start of operation and would instead be free to develop its own new models over time. Along the same lines, the system would have the freedom to discard models that are determined to consistently be poor performers (e.g., a model that only ever attempts to go north). This would require a new CD implementation that does not just enumerate a number of solution models at startup, but also attempts to create new models by combining and recombining existing ones. Show success in this area would provide additional evidence that CD has strong potential for use in artificially intelligent systems.

Finally, the CD simulator developed for this research is only one example of how CD might be implemented. This work does not claim that this is the best way to implement the technique. Therefore additional research on alternate implementation methods is needed in order to determine how the system may be optimally implemented. Such experiments may, for example, investigate alternate ways of constructing solution models, evaluating model performance, and/or presenting observed data to models during dream phases.

Chapter 9 References

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed.: Morgan Kaufman, 2007.
- [2] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server Engineering Insights for Large-Scale Online Services," *Micro, IEEE*, vol. 30, pp. 8-19, 2010.
- [3] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *American Federation of Information Processing Societies*, 1967, pp. 483-485.
- [4] J. L. Gustafson. (1988) Reevaluating Amdahl's Law. *Communications of the ACM*. pp. 532-533.
- [5] V. Gupta, H. Kim, and K. Schwan, "Evaluating Scalability of Multi-threaded Applications on a Many-core Platform," Georgia Institute of Technology 2012.
- [6] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, Seattle, WA, USA, 2008, pp. 277-286.
- [7] S. Furber. (August 2012) To Build a Brain. *IEEE Spectrum*. pp. 45-49.
- [8] V. Mountcastle, "An Organizing Principle for Cerebral Function: The Unit Model and the Distributed System," in *The Mindful Brain*, Cambridge, Mass.: MIT Press, 1978.
- [9] J. Hawkins and S. Blakeslee, *On Intelligence*: Times Books, 2004.
- [10] K. Louie and M. A. Wilson, "Temporally structured replay of awake hippocampal ensemble activity during rapid eye movement sleep," *Neuron*, vol. 29, pp. 145-156, 2001.
- [11] D. J. Cai, S. A. Mednick, E. M. Harrison, J. C. Kanady, and S. C. Mednick, "REM, not incubation, improves creativity by priming associative networks," *Proceedings of the National Academy of Sciences*, vol. 106, pp. 10130-10134, 2009.
- [12] U. Wagner, S. Gais, H. Haider, R. Verleger, and J. Born, "Sleep inspires insight," *Nature*, vol. 427, pp. 352-355, 2004.
- [13] J. M. Paul. (2010, Accessed: 06 Dec 2012). "Computational Dreaming." [Whitepaper]. Available: <http://www.ece.vt.edu/jmpaul/cd-pubs/cd-pub5.7.pdf>
- [14] M. Mott. (2010, Accessed: 9 December 2012). "Do dogs and cats really dream?" *USA Today*. Available: http://usatoday30.usatoday.com/life/lifestyle/pets/2010-06-29-dogs-cats-dream_N.htm

- [15] M. J. Flynn, "Very High Speed Computers," *Proceedings of the IEEE*, vol. 54, pp. 1901-1909, 1966.
- [16] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, 2nd ed.: Pearson Prentice Hall, 2005.
- [17] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. Snove, and O. R. Birkeland, "A recursive MISD architecture for pattern matching," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, pp. 727-734, 2004.
- [18] C. G. Kent, "Personalized Computer Architecture as Contextual Partitioning for Speech Recognition," M.S. Computer Engineering, Electrical and Computer Engineering, Virginia Tech, 2009.
- [19] C. G. Kent and J. M. Paul, "Contextual partitioning for speech recognition," *ACM Trans. Embed. Comput. Syst.*, vol. 13, pp. 1-20, 2013.
- [20] M. D. Hill and M. R. Marty. (2008, 33-38) Amdahl's Law in the Multicore Era. *IEEE Computer*.
- [21] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater Scalability for Parallel Molecular Dynamics," *Journal of Computational Physics*, vol. 151, pp. 283-312, 1999.
- [22] G. Tournavitis, Z. Wang, B. Franke, and M. F. P. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, Dublin, Ireland, 2009, pp. 177-187.
- [23] P. S. Pacheco, *Parallel Programming with MPI*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1997.
- [24] D. Whitley, "A Genetic Algorithm Tutorial," *Statistics and Computing*, vol. 4, pp. 65-85, 1994.
- [25] K. De Jong, "Learning with Genetic Algorithms: An Overview," *Machine Learning*, pp. 121-138, 1988.
- [26] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed.: Prentice Hall, 2003.
- [27] M. Hagan, H. Demuth, and M. Beale, *Neural Network Design*. Boulder, CO: Campus Publishing Service, Colorado University Bookstore, 2002.
- [28] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: A tutorial," *IEEE computer*, vol. 29, pp. 31-44, 1996.
- [29] S. Furber, S. Temple, and A. Brown, "High-Performance Computing for Systems of Spiking Neurons," in *Proceedings of AISB '06 workshop on GC5: Architecture of Brain and Mind*, Bristol, UK, 2006, pp. 29-36.

- [30] X. Jin, S. Furber, and J. Woods, "Efficient Modelling of Spiking Neural networks on a Scalable Chip Multiprocessor," in *IEEE International Joint Conference on Neural Networks*, 2008, pp. 2812-2819.
- [31] M. Marshall. (2012, Accessed: 1 December 2012). "Brain-like chip outstrips normal computers." *New Scientist* (2892). Available: <http://www.newscientist.com/article/mg21628925.100-brainlike-chip-outstrips-normal-computers.html>
- [32] T. Pfeil, P. Muller, A. Grubl, M. Petrovici, J. Schemmel, S. Jeltsch, M. Schmuker, K. Meier, E. Muller, and D. Bruderle. (2012, Accessed: 29 October 2012). "Six networks on a universal neuromorphic computing substrate." Available: <http://arxiv.org/pdf/1210.7083v4>
- [33] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, pp. 269-271, 1959.
- [34] V. S. Gordon and Z. Matley, "Evolving sparse direction maps for maze pathfinding," in *Evolutionary Computation, 2004. CEC2004. Congress on*, 2004, pp. 835-838.
- [35] S. Mishra and P. Bande, "Maze Solving Algorithms for Micro Mouse," in *IEEE International Conference on Signal Image Technology and Internet Based Systems*, 2008, pp. 86-93.
- [36] M. K. Gunady and W. Gomaa, "Reinforcement learning generalization using state aggregation with a maze-solving problem " in *Japan-Egypt Conference on Electronics, Communications and Computers*, 2012, pp. 157-162.
- [37] D. Osmankovic and S. Konjicija, "Implementation of Q-Learning algorithm for solving maze problem," in *MIPRO, 2011 Proceedings of the 34th International Convention*, 2011, pp. 1619-1622.
- [38] S. Diekelmann and J. Born, "The memory function of sleep," *Nature Reviews: Neuroscience*, vol. 11, pp. 114-126, 2010.
- [39] M. Walker and R. Stickgold, "Overnight alchemy: sleep-dependent memory evolution," *Nature Reviews: Neuroscience*, vol. 11, p. 218, 2010.
- [40] V. Pankratiy, C. Schaefer, A. Jannesari, and W. F. Tichy, "Software Engineering for Multicore Systems: An Experience Report," in *Proceedings of the 1st International Workshop on Multicore Software Engineering*, 2008, pp. 53-60.
- [41] R. Stickgold, J. A. Hobson, R. Fosse, and M. Fosse, "Sleep, Learning, and Dreams: Off-Line Memory Reprocessing," *Science*, vol. 294, pp. 1052-1057, 2001.
- [42] R. S. Sexton and J. N. D. Gupta, "Comparative evaluation of genetic algorithm and backpropagation for training neural networks," *Information Sciences*, vol. 129, pp. 45-49, 2000.
- [43] A. Bagnall and Z. Zatuchna, "On the Classification of Maze Problems," in *Foundations of Learning Classifier Systems*. vol. 183, L. Bull and T. Kovacs, Eds.: Springer Berlin Heidelberg, 2005, pp. 305-316.

Appendix A

Glossary of Terms

Agent	For the purposes of this thesis, an agent is the metaphorical entity that attempts to solve a maze by applying the steps in a given solution model.
AI	Artificial intelligence
Awake Phase	See Day Phase
CD	Computational dreaming
CDS	Computational dreaming simulator
Day Phase	The part of the CD process during which a single solution model is used to solve newly presented problems. During the day phase, the system accumulates new problem data for use during the dream phase.
Dream Phase	The part of the CD process during which <i>all</i> solution models attempt to solve mazes from the database of observed problem information. At the end of the dream phase, a new solution model is chosen for use during the subsequent day phase.
Iteration	A single CD iteration consists of exactly one day phase followed by exactly one dream phase.
MISD	Multiple Instruction-stream, Single Data stream
Solution Model	A series of steps that can be used to attempt to solve a given maze.
Viable	In the context of this document, a system is said to be “viable” if it exhibits performance that can be statistically shown to be better than random chance.

Appendix B Simulator Parameter Listing

This section lists all simulator parameters along with a brief description of the parameter's function along with default, min, and max values. The default, minimum, and maximum values listed are for the purposes of the simulator only and do not necessarily represent any limitations of CD in general. Note that many parameters are primarily for debug and development purposes and were thus not discussed in the main text. UINT_MAX represents the largest value that can be held by an unsigned integer.

<u>Parameter</u>	<u>Description</u>
min-dim	Specifies the minimum dimension (height and width) that mazes generated during day phases may have. Default = 3 Range = [2-UINT_MAX]
max-dim	Specifies the maximum dimension (height and width) that mazes generated during day phases may have. Default = 10 Range = [2-UINT_MAX]
min-paths	Specifies the minimum number of solutions that mazes generated during day phases should have. Default = 2 Range = [1-UINT_MAX]
max-paths	Specifies the maximum number of solutions that mazes generated during day phases should have. Default = 10 Range = [1-UINT_MAX]
min-day	Specifies the minimum number of mazes to generate during day phases. Default = 2 Range = [1-UINT_MAX]
max-day	Specifies the maximum number of mazes to generate during day phases. Default = 10 Range = [1-UINT_MAX]
min-dream	Specifies the minimum number of known mazes to process during dream phases. Default = 2 Range = [1-UINT_MAX]
max-dream	Specifies the maximum number of known mazes to process during dream phases. Default = 10 Range = [1-UINT_MAX]

layers	Specifies the number of layers of solution models to generate. Default = 4 Range = [1- UINT_MAX]
cd-iter	Specifies the number of CD iterations that should be executed during this simulation. Default = 3 Range = [1- UINT_MAX]
dbg_save-gen	Specifies whether or not all mazes generated during day phases should be saved to disk. This parameter is intended for use in debugging and/or data analysis. Default = true Range = true, false
dbg_save-dayAtt	Specifies whether or not the observed portion of mazes seen during day phases should be saved to disk, even if no solution was found. This parameter is intended for use in debugging and/or data analysis. Default = false Range = true, false
dbg_save-daySol	Specifies whether or not solutions found during day phases should be saved to disk. This parameter is intended for use in debugging and/or data analysis. Default = true Range = true, false
dbg_save-daySolFull	When a solution model successfully solves a maze during a day phase, this parameter specifies whether or not a copy of the originally generated maze (i.e., the complete maze, including any unobserved portions) should be saved to disk. This parameter is intended for use in debugging and/or data analysis. Default = false Range = true, false
dbg_save-dreamAtt	Specifies whether or not the observed portion of mazes seen during dream phases should be saved to disk, even if no solution was found. This parameter is intended for use in debugging and/or data analysis. Default = false Range = true, false
dbg_save-dreamSol	Specifies whether or not solutions found during dream phases should be saved to disk. This parameter is intended for use in debugging and/or data analysis. Default = true Range = true, false
dbg_show-solve	Specifies whether or not a message should be printed to the output terminal whenever a maze is successfully solved during the course of simulation. Default = false Range = true, false
dbg_show-attempt	Specifies whether or not a message should be printed to the output

	terminal whenever a maze is attempted but not solved during the course of simulation. WARNING: Setting this parameter to true will generate a LOT of output. Default = false Range = true, false
random-model-selection	Specifies whether or not models used during day phases should be selected at random (instead of using the CD process to intelligently select a model). Setting this parameter to true will cause the simulation to skip all dream phases and simply randomly select a new solution model at the start of each day phase. This parameter is intended to be used to generate data on how well mazes can be solved using random chance model selection. Default = false Range = true, false
max-threads	Specifies the number of threads to be used during dream phases. Models are split up more or less evenly among all threads. This parameter has no effect on the actual results of a simulation and is intended to be used to help speed up the execution time of a simulation Default = 1 Range = [1- UINT_MAX]