Attentional Control of Complex Systems

Thesis by

Joseph Gregory Billock

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

2001

(Submitted May 21, 2001)

Acknowledgements

It is a wonderful thing to have so many people to acknowledge and thank when completing such a large project. I want to first thank my wife Christy, for her steady support and encouragement. Her presence has been indispensable.

Special thanks goes to my advisor, Prof. Demetri Psaltis, for all his guidance, suggestions, discussions, and motivation. Also instrumental in making this work happen was Prof. Christof Koch. His energy and inspiration are much appreciated.

I worked with many colleagues on the projects described in this thesis. Thanks to Dr. George Barbastathis for the initial exploration of the *Desert Survival* game, and of other ideas further expanded upon here. Thanks also to Dr. Yaser Abu-Mostafa, Dr. Chuan Xie, Ernest Chuang, Yunping Yang, Nathan Gray, and William Peterson for their collaboration. Thanks also to Greg Steckman, Jose Mumbru, Wenhai Liu, and everyone I overlapped with in the Psaltis Group, for encouragement, support, discussions, and the occasionally short group meeting talk: Ali Adibi, Robert Denkewalter, Jean-Jacques Drolet, Geoff Burr, David Marx, Xin An, Xu Wang, Christophe Moser, George Panotopoulos, Dr. Karsten Buse, Zhiwen Liu, Irena Maravic, Michael Levene, Allen Pu, Emmanouil Fitrakis, Todd Meyrath, Jung-te Hsieh, and Martin Centurion.

I also want to acknowledge my parents, Joe and Lyn Billock, as well as Becky, Paul, and Jonathan, for their support while I've been at Caltech. Thanks also to the

many other people who I've relied on during the last few years: Mark and Earlene Papendick, Mike, Lissie, Jamie, and Kiera Quishenberry, Sheena Frye, and everyone else who would make this section way too long to mention individually.

I want to especially thank the organizations who have provided funding for this research. The Virginia Scott Steele fellowship, The National Science Foundation through its ERC program, the Center for Neuromorphic Engineering at Caltech.

A big thank you goes to Lucinda Acosta for all the excellent administrative support y conversacíon Español. Thanks also to Ya-Yun Liu for patiently maintaining the labs and equipment, and to Helen Carrier and the rest of the EE admin staff for keeping a tight ship.

Finally, I want to thank you, the readers, of whom primacy of place goes to the oft-underappreciated Caltech proofreaders, who wade through so many of these documents with such positive results.

Abstract

This thesis reports on work done in applying some of the concepts and architectures found in biological computation to computer algorithms. Biology has long inspired computer technology at the level of processing elements. This thesis explores the application of biologically inspired algorithms at a higher level—that of functional structures of the nervous system. The first chapter gives background on the attentional/awareness model of the brain, why it is important to biology and the advantages in real-time performance and in learning facilitation which we expect from applying it in computer algorithms.

The second chapter examines the application of this model to a canonical computer science problem—the bin packing problem. Approaching this NP-complete problem when limited by computational resources and time constraints means that algorithms which throw away large amounts of the information about the problem perform better than those which attempt to consider everything. The existence of an optimum in the size of a working memory needed to find the best solution under time pressure is shown. The transition between the regime of strict time constraints and more forgiving time constraints is quite sudden. Chapter 3 presents an analytical model for better understanding the performance of various bin packing algorithms.

Chapter 4 examines the application of the attentional model to a real-time computer game testbed. This testbed is explained, and results are shown which illustrate

that in a complex, unpredictable environment with tight time and resource constraints conditions, an algorithm which examines only that information which falls into a relatively small part of the playing area can win against player which addresses it all.

Chapter 5 turns to an examination of the role of reduced informational representations upon learning. Solving of various logical-kinetic puzzles by a simulated segmented arm is done by a learning system. A logic supervisory subsystem utilizes attentional/awareness methods to train, and pass control of the different control levels of the articulate arm over to, the neural networks, adaptive resonance theory networks, and declarative computer memory which it trains. Finally, chapter 6 presents an overview and evaluation of the work.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Biology has long served as an inspiration for invention. Humans strive to build machines to copy behavior we see, such as the flight of birds and the swimming abilities of dolphins. We have imitated the properties of biological materials from wool to wood. Most recently, our computers have been inspired by the phenomenal capabilities of biological brains, particularly our own. Our desire to mimic the brain stems from the abilities it possesses, which are in so many cases superior to computers and algorithms we can build today. The brain shows amazing adaptability to a wide variety of unpredictable, noisy inputs. It deals gracefully with the unexpected, in situations where a creative, ingenious solution, instead of one that is optimal, is required in real time.

Scientists and engineers have explored many ideas in response to this inspiration. Most of them have engaged biology at the neuronal level. McCulloch and Pitts [2] made an important discovery that neuron network models could be constructed to produce any logical function. Later, Rosenblatt [1] developed a mechanism for training a perceptron—a neuron modeled in the McCulloch-Pitts fashion—to do input discrimination. On this basis, later researchers described the emergent computational and memory properties of networks of similar neuron-like models (e.g. Hopfield [3]). Subsequently there has developed a rich literature and many important applications which take advantage of the capabilities of neural networks.

This thesis will explore the biological inspiration at a different level. Instead of focusing at the neural level, it will examine what computational advantages may be gained from copying the brain at the architectural, or functional level. Specifically, we are interested in the role which the important elements of attention and awareness play in the brain, and what advantages can be expected by digital computational architectures which utilize the same kinds of mechanisms in their operation.

As a starting point, we take as a broad model of the human computational system a proposal by Koch and Crick [5], [7], [6], personal communication, [4]. This model is intended to investigate the computational strategy which organisms have evolved to deal with overwhelming environmental complexity and still behave in real time.

Systems forced to behave in real time face special pressures which the more rigorous parts of algorithmic research and computer science have had less success dealing with. Confronted with a sufficiently complex problem, such as those described in the computer science literature as NP complete [32], we can in principle find an algorithm which will solve it. But as the size of the problem grows, the algorithm will take an immense amount of computational resources to find a gauranteed optimal solution. (If the problem is of sufficient size, the amount of resources may be so large as to defy any possibility of instantiating the algorithm.) I argue that biological organisms face environments of equal or greater complexity to problems devised by computer science, yet they are able to respond in these situations with adequate performance and without using astronomical quantities of supercomputer time. How is this possible?

1.1 An attentional model

The attentional model is a hypothesis about how biological brains solve this problem. At its simplest, we imagine the environment of an organism as a richly textured, complex sensory environment. We picture the computational resources of the organism as containing a high-operational-cost "logic," or "planning," unit. This "logic unit" is expensive to operate and much slower than the reflexes the animal employs to execute most decisions. In deference to its slower nature, the use of the logic resource is restricted by the organism. If the logic part of the brain were allowed to fully control the organism, real-time behavior would be radically impaired. For example, humans learning to walk on two legs require a great deal of concentration and practice to master this complex behavior. If the level of attention required to learn to walk was continuously required to move in this way, it is doubtful that humans could do so. The learning of the "walking reflex" is critical to free the expensive parts of the brain for use in other problems. After walking is learned, one only becomes aware of walking when something is wrong: after an injury, for instance, when muscles need to be retrained, or when an error occurs, such as tripping over an obstacle.

Reducing the complex environment to a small number of sensory-motor online systems makes it possible to pack a strategy for survival into fewer neurons. The question is the strategy the species takes to learning. If the species deals with environmental effects at the population level, having planning centers is not the approach taken. The online systems evolve to deal with environmental novelties. For species which invest more energy in each individual, however, being able to efficiently learn,

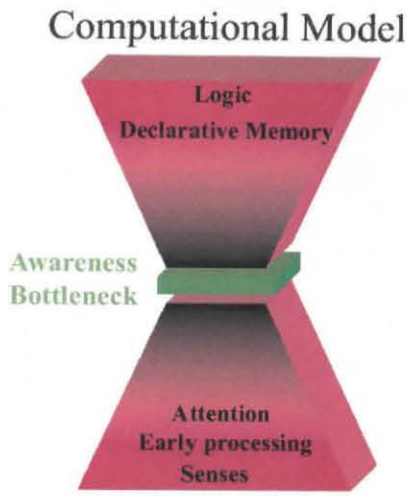

Figure 1.1: A simple computational model of the awareness bottleneck. Only a small portion of the data acquired through the senses is passed through stages of early processing and arouses the attention to pass it through the awareness bottleneck to the slower, serially operating planning and executive areas.

and deal with environmental novelties, is an imperative.

As a mechanism for learning or for size reduction, then, we believe that a reduced representation of the complex environment is critical for the real-time performance of biological systems, and that is why they exhibit the qualities of "attention" and "awareness."

1.2 Neurological Inspiration for the Awareness Model

According to Crick and Koch [6], the function of conscious awareness in biological systems is to "... produce the best current interpretation of the visual scene in the light of past experience, either of ourselves or of our ancestors (embodied in our genes), and to make this interpretation directly available, for a sufficient time, to the parts of

Image from *Virtual Hospital*
http://www.vh.org

Figure 1.2: The anterior cingulate cortex (shown highlighted) may play an important role in the coordination of attention and awareness in the brain.

the brain that contemplate and plan voluntary motor output, of one sort or another, including speech." From this biological function we abstract several principles. First, an awareness representation consists of a transformation, specifically, a reductive transformation, on current sensory data. That is, the awareness holds in a kind of cache state a version of the current information from sensory and remembered data. Second, this awareness representation is one which provides maximum utility (in some sense) to the organism. Third, the organism's higher-order planning centers utilize this represented information to construct a real-time response to its situation.

1.3 The anterior cingulate cortex

While there is no consensus on the matter of where in the brain an attentional function resides, or even on whether they reside in particular areas of the brain, or are holistically derived, there has been attention given to discovering the parts of the brain involved in attentionally providing data to, or coordinating, biological awareness. In the primate brain, this task may have an important part of its function take place in the anterior cingulate cortex. Several studies exploring tasks which rely

Base	Neutral	Congruent	Incongruent
RED	LOT	RED	RED
BLUE	DOG	BLUE	BLUE
YELLOW	FLY	YELLOW	YELLOW
BROWN	RUN	BROWN	BROWN
GREEN	CAT	GREEN	GREEN
ORANGE	SOCK	ORANGE	ORANGE

Figure 1.3: The Stroop Task. An illustration of the effects of awareness on performance in a simple task. The task is to read either the words in the list or the color in which the words are printed. The base task has no colored print, but uses the names of colors. The neutral task uses colored print, but the words are not the names of colors. In the congruent task, the names of the colors are printed in the same color, and in the incongruent task, they are printed in other colors. People find the base, neutral, and congruent tasks very easy. In the incongruent task, it is easy to read the words as printed, but very difficult to read the colors, because of the competition with the awareness of what the words say.

heavily on the ability of the brain to coordinate attentive data. For example, the Stroop task [16], which asks subjects to either read words in a list or name the colors in which the words are printed.

In this task, there is a conflict presented by the content of awareness representation and the performance desired (reading or color naming). In this and other similarly demanding tasks (like generate-use task [14] and task switching [11]), the anterior cingulate cortex is active. More recent experiments [12], [15] indicate that this function of the ACC is tied to its abilities as an error predictor. That is, the ACC modulates awareness by its hypothesis about the trustworthiness of the data which it is coordinating. When it believes that the "reflex" response will be adequate, it is less active. When it believes that the higher order brain functions are necessary to

come up with a response, it is active.

There are other psychophysical experiments which indicate that the bulk of sensory information is not represented to the higher-order brain centers. Change blindness [18] [19], in which even dramatic changes to a visual scene do not "register" in conscious awareness, is an excellent example. It appears that the brain relies on sensory information residing in a scene as a kind of memory, and so does not internally represent all that information in the more advanced stages of the visual processing pathways. This means that there is a data bottleneck between early visual processing parts of the brain (the retina, V1) and planning parts of the brain (frontal cortex). The phenomenon of blindsight [21] [20] illustrates the fact that some behaviors do not pass through this bottleneck, but are dealt with by the on-line systems. A patient with blindsight is unaware of visual stimuli, but when confronted with a forced choice on a simple visual task, performs much, much better than chance. In addition, visual tasks which are typically subconscious in everyone, such as size scaling in grasping, or orienting the hand vertically or horizontally to take an object, are performed almost normally in patients that are perceptually unable to see the objects being grasped. This indicates that there is much visual preprocessing that takes place below the level of conscious awareness, and only a select subset of sensory data is represented as the contents of awareness, for processing by the frontal cortex.

1.4 Why attention and awareness?

Crick and Koch [6] suggest that the mental mechanisms of attention and awareness are tuned to produce a reduced representation (optimal in some sense) which is then available to the energetically (and functionally) expensive planning areas of the brain to use for decision making. The evidence suggests that there are "zombie" parts of the brain which can affect motor responses without the involvement of conscious awareness. The hypothesized reason for the brain's utilization of the awareness "pathway" for processing is that it is necessary to have unified planning and decision-making by the organism. The 'awareness' subsystem, with its access to explicit memory, can deal with more complex scenarios and generate a strategy for action (see also [10]). Several reasons can be adduced for this.

First, a unified basis for decision avoids conflict or inconsistency in the actions of the organism. That is, if there are multiple "zombie" agents active within the brain, they may alternate in controlling the organism, or their conflicting commands may produce no action at all. Second, a unified decision process can be leveraged to create a superior memory apparatus, making possible higher-order responses by the organism. For example, a digger wasp (*Sphex ichneumoneus*) will leave a kill outside a hole, enter, check the hole out, then go out again and drag the prey inside. If the prey is moved, the wasp will simply repeat and repeat and repeat its (zombie-driven) actions (see [31] ref. [30]). Higher-order, unified, conscious, memory-enhanced decision making can decrease these sorts of susceptibilities. Third, a unified process allows for long-range strategic plans and forecasts. It makes possible the commitment

to a course of action within wider environmental parameters.

Having a higher "logic" planning cortex is not an unmitigated good. This part of the brain is expensive to operate, most importantly in terms of the slower reaction times it produces for the organism. Conscious responses in humans slow reactions by at least a few hundred milliseconds over the "zombie" reflex system responses. For example, startle reflexes in humans range from around 10 ms for the blink reflex to about 50 ms for the patellar reflex. In contrast, conscious processing requires several hundred ms for responses (e.g. [33]). This cost is born by the organism in increased reaction times to predator attacks, and in the inability to take advantage of faster-reacting food sources. The hypothesis is that this expensive, slower "planning machine," to be used optimally, must be guarded from the bulk of sensory input. Thus, the awareness representation must be reduced. The real-time exigencies of behavior often require the organism to choose some course of action. That is, the costs of doing nothing (sitting and thinking) are often higher than taking some action, even if that action is suboptimal. The lack of differentiation for higher-order visual resources, or "visual attentional capacity," has been the subject of investigation [24] as well. Here Lee et. al. found that concurrent discrimination of stimuli varying in form, color, and motion show that visual attention performs equally when faced with similar and dissimilar dual discrimination tasks (i.e. between two different forms versus between form and color). The expectation is that this resource limitation extends to other sensory modalities as well. That is, while the semantic content of the "awareness" representation may change with modality, as different

preprocessing steps for vision/hearing/etc. are activated, the awareness bottleneck remains relatively constant in capacity.

1.5 Awareness and Memory

The awareness mechanism also seems important for some kinds of memory, both for memorization and recall. There has been much interest in exploring the role of event-related potential as an indicator of this process (e.g., [28]). Differentiation has been made for major categories of memory including procedural memory, and semantic (or declarative) memory [25]. However, this is not the whole story, and there is evidence for subdivisions within those areas for various sensory modalities, functions of storage and recall, etc. (e.g., [27], [26]). The role of memory will vary depending on the kind of memory involved. For episodic memory, at least, attention and awareness seem to form a gateway (e.g. [34]). That is, what is attended to (placed in the awareness representation), can enter into episodic memory. What is not attended can not. This is an oversimplification, and there are intermediate-term facilitation effects from unattended stimuli which show that there must be a memory component which operates below the threshold of conscious awareness. As a rule, though, it appears that for long-term episodic memory to be formed, the contents must be put through the awareness "bottleneck."

Procedural memory also would seem to have a component sensitive to the contents of awareness. Here, though, there is a less definitive, or declarative/factual, element which can be pointed to as being represented. Instead, a kind of "concentration"

on a task seems to facilitate procedural memory formation. Once this has happened, though, attending further to the task does not help and may even inhibit performance. For example, a lot of concentration is required when humans are learning to walk, but after this has happened, attention to walking is unnecessary. Furthermore, attention can then be turned to skills which depend on walking. In many complex sports, intermediate levels of skills are learned, and consequently ignored, as more advanced skills are taken up by the attentional process to be mastered.

Why might this be? The awareness bottleneck as a prerequisite for long-term declarative memory formation could be to limit the size of declarative memory. That is, not every sensory datum collected by the organism is worthy of storage. The memory capacity of the brain could be overwhelmed, and its recall speed severely reduced. Only those data most useful for strategic planning and goal-oriented responses and activities need to be remembered. The awareness bottleneck exhibits these filtering properties.

The attentive component involved in procedural memory also appears to be important. Here more general concerns about learning may be involved: procedural learning must be very generalizable. A strong attentive component may be the important key in reducing the amount of "background noise" in the learning environment (by shutting it out as unimportant to the task) and therefore in training a much more robust, generalized, procedural response. At this level, though, even though attentiveness may be facilitatory, it is not under conscious control. A sense of "concentration" may feel demanding, but the function itself is implicit, with no conscious access to

procedural memory.

1.6 Neurological Inspiration for Algorithm Architecture

The system described by Koch and Crick is diagrammed below as by Psaltis in Figure 1.4. The sections of the diagram towards the bottom—the motor/processing modules, early processing and error generation—reside in the brain below the level of conscious awareness. This subconscious system is a kind of "inner zombie" (at least in human beings), which has fast reflexes and extensive procedural memories. The attention and awareness are the gateways which present preprocessed sensory data to the higher, more computationally expensive parts of the brain (the logic, or planning, cortices, and memory).

The brain as a whole, according to this model, contains many feedback loops between the "zombie" level parts of the brain and the planning, executive, and memory areas. The system as a whole, to maintain a coherent course of action, must be capable of suppressing either volitional control in favor of reflex-level control, or vice versa.

Ultimately, the true function of awareness in the brain is undoubtedly more complex than that described above. The current state of neuroscience of attention has been compared [29] to the state of geography in the fifteenth century: some researchers have described in detail isolated sections of new-found beach, but there is as yet no

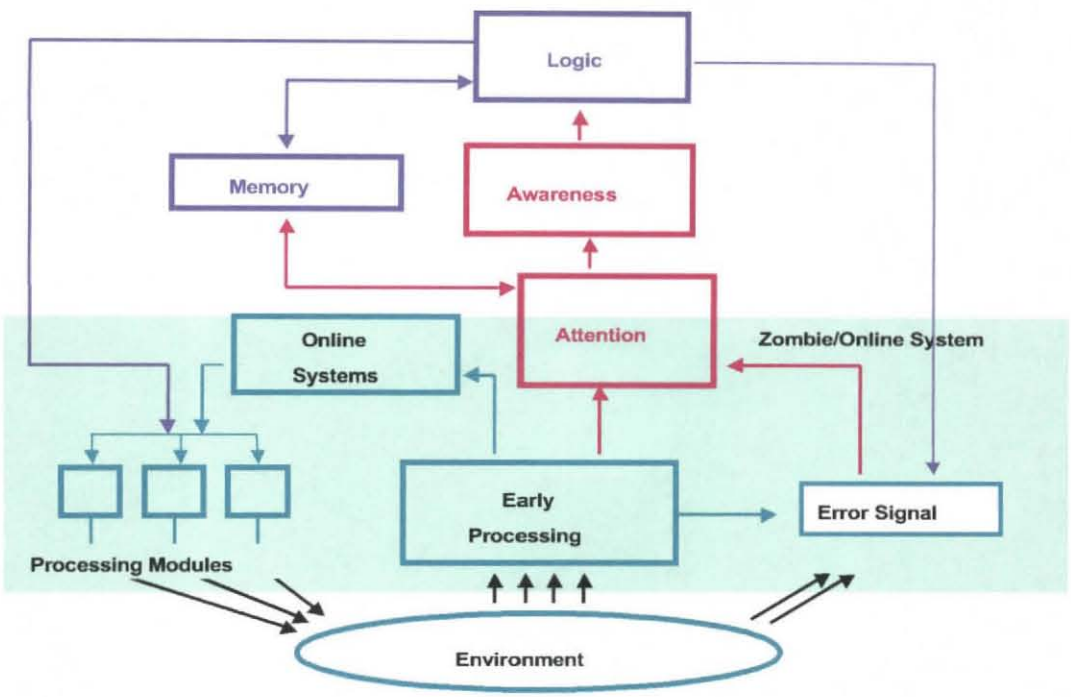

Figure 1.4: Attention/Awareness System. In this functional model of the role of attention and awareness, the pathway incorporating the awareness bottleneck operates in parallel to the faster "zombie" systems, taking their attentional cues from the "error signals" which the zombie system generates.

overall map. We believe, though, that it is not too early to begin thinking about the role of attention and awareness in the brain, trying to decipher their mechanisms and functions, and beginning to explore how to utilize the insights gained thereby to improve computer algorithms.

1.7 Applying attentional processes to computer algorithms

What we would like to abstract from the biological functions of attention and awareness is a computational architecture which can aid in performance on the same kinds of tasks in a computer system. The awareness/reduced representation/bottleneck mechanism in the brain is a very powerful tool involved in real-time, meaningful behavior and robust declarative and procedural memory. Specifically, referring to the figure, we want to explore ways in which the attentional processing framework interacts with other parts of the system.

One goal will be the demonstration that, for sufficiently complex environments, using a reduced representation of the environment allows an algorithm to function better when under time pressure. We will examine the details of this in two contexts. The first will be an application to a new algorithm for solving a traditional computer science problem—the bin packing problem. Despite being easily posed, this problem is an example of an NP hard problem, requiring immense computational resources to solve exactly for even moderately sized problems. We will propose a new heuristic

which employs a reduced representation of the problem space to produce competitive solutions under more time pressure than traditional algorithms which do not use such a reduced representation. In doing so, we will gain an understanding of the conditions under which that happens and the types of problems amenable to this approach.

The second context is an application to a real-time computer game. In this (invented) game, two players compete to take control of a mutually accessible territory. We will explore the conditions under which it is possible for a computer player which uses only a small part of the playing field information available to it can win over a competing computer player which uses all of it.

We would also like to understand better what advantages implementing such a bottleneck has for memory and machine learning. The tool we use to investigate this is a multi-leveled system wherein a computer controlled arm, using algorithms developed according to the above model, bootstrap themselves into increasing facility in the control of a segmented arm. At the bottom level, the arm's joints must be controlled to produce a desired motion. Intermediately, since the arm is moving in an environment with obstacles, the arm's motions are collected and learned as the basic "gestures," like going around an obstacle, or going straight to the target. The system also learns to collect these gestures into sequences of coherent, optimized motions which move the arm from one target point to another, effecting the solution of different sorts of abstract problems. This integrated testbed will demonstrate the effectiveness of the awareness "bottleneck" in facilitating learning at the reflex level, and in the interaction between the slower, more optimization minded "planning" units

of the model and the faster, procedural, reflex units. This example will emphasize the role of attention and awareness as a mediating bottleneck between the logic and the reflex systems, and its role as a catalyst for effective machine learning.

Bibliography

[1] Rosenblatt, F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, *Psychological Review*, **65**:6 pp. 386-408, 1958.

[2] McCulloch W., and Pitts W. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics* **7**, pp. 115 - 133, 1943.

[3] Hopfield, J. Neural Networks and Physical Systems with Emergent Collective Computational Properties. *Proceedings of the National Academy of Sciences* **79**, pp. 2554-2558, 1982.

[4] Barbastathis, G. Intelligent Holographic Databases. Ph.D. Thesis, California Institute of Technology, 1998.

[5] Crick F., and Koch C. Towards a Neurobiological Theory of Consciousness. *Seminars in Neuroscience* **2**, pp. 263-275, 1990.

[6] Crick F., and Koch C. Consciousness and Neuroscience. *Cerebral Cortex* **8**, pp. 97-107, 1998.

[7] Crick F., and Koch C. Are We Aware of Neural Activity in Primary Visual Cortex? *Nature* **375**, pp. 121-123, 1995.

[8] Crick F., and Koch C. Why Neuroscience may be able to Explain Consciousness. *Scientific American* **273**, pp. 84-85, 1995.

[9] Vanni S., Revonsuo A., Saarinen J., and Hari R. Visual Awareness of Objects Correlates with Activity of Right Occipital Cortex. *Neuroreport* **8**, pp. 183-186, 1996.

[10] Newman J., Baars B. J., and Cho S.-B. A Neural Global Workspace Model for Conscious Attention. *Neural Networks* **10**:7, pp. 1195-1206, 1997.

[11] Désposito M., Detre J. A., Alsop D. C., Shin R. K, Atlas S., and Grossman M. The Neural Basis of the Central Executive System of Working-Memory. *Nature* **378**:6554, pp. 279-281, 1995.

[12] Carter C. S., Braver T. S., Barch D. M., Botvinick M. M., Noll D., and Cohen J.D.. Anterior Cingulate Cortex, Error Detection, and the Online Monitoring of Performance. *Science* **280**:5364, pp. 747-749, 1998.

[13] Cabeza R., and Nyberg L. Imaging Cognition: An Empirical Review of PET Studies with Normal Subjects. *Journal of Cognitive Neuroscience* **9**:1, pp. 1-26, 1997.

[14] Raichle M. E., Fiez J. A., Videen T. O., Macleod A. M. K., Pardo J. V., Fox P. T., Petersen S. E. Practice-Related Changes in Human Brain Functional-Anatomy During Nonmotor Learning. *Cerebral Cortex* **4**:1, pp. 8-26, 1994.

[15] Stemmer B., Segalowitz S.J., Witzke W., Lacher S., and Schonle P.W. Error Detection and the Error-Related ERP in Patients with Lesions Involving the Anterior Cingulate and Adjacent Regions. Abstract appearing in *Psychophysiology* **37** (Suppl. 1), S95-S95, 2000. http://cogprints.soton.ac.uk/documents/disk0/00/00/01/43/index.html

[16] Posner M. I., and DiGirolamo G. J. Conflict, Target Detection and Cognitive Control. In *The Attentive Brain* Ed. Raja Parasuraman (ISBN 0-262-16172-9). Cambridge, Mass, The MIT Press, 1998. http://hebb.uoregon.edu/brainlab/online_papers/Posner&DiGi.html

[17] Fuster J. M. The Prefrontal Cortex: Anatomy, Physiology, and Neuropsychology of the Frontal Lobe, Third Ed. (ISBN 0-397-51849-8). Philadelphia, PA, Lippincott-Raven, 1997.

[18] Simons D. J., and Levin D. T. Failure to Detect Changes to Attended Objects. *Investigative Ophthalmology and Visual Science* **38**, 3273, 1997. (Conference Abstract)

[19] Simons D. J., and Chabris C. F. Gorillas in our Midst: Sustained Inattentional Blindness for Dynamic Events. *Perception* **28**:9, pp. 1059-1074, 1999.

[20] Goodale M. A., Milner A. D., Jakobson L. S., and Carey D. P. A Neurological Dissociation Between Perceiving Objects and Grasping Them. *Nature* **349**:6305, pp. 154-156, 1991.

[21] Weiskrantz L., Warrington E. K., Sanders M. D., and Marshall J. Visual Capacity in the Hemianopic Field Following a Restricted Occipital Ablation. *Brain* **97**, pp. 709-728, 1974.

[22] Itti L., and Koch C. A Saliency-Based Search Mechanism for Overt and Covert Shifts of Visual Attention. *Vision Research* **40** (10-12), pp. 1489-1506, 2000.

[23] Koch C., and Braun J. Towards the Neuronal Correlate of Visual Awareness. *Current Opinion in Neurobiology* **6**, pp. 158-164, 1996.

[24] Lee D.K., Koch C., and Braun J. Attentional Capacity is Undifferentiated: Concurrent Discrimination of Form, Color, and Motion. *Perception and Psychophysics* **61**:7, pp. 1241-1255, 1999.

[25] Schachter D.L. and Tulving E., eds. Memory Systems (ISBN 0-262-19350-7). Cambridge, MA, MIT Press, 1994.

[26] Tulving E., Kapur S., Craik F. I. M., Moscovitch M., and Houle S. Hemispheric Encoding/Retrieval Asymmetry In Episodic Memory — Positron Emission Tomography Findings. *Proceedings Of The National Academy Of Sciences* **91**:6, pp. 2016-2020, 1994.

[27] Brefczynski J.A., and DeYoe E.A. A Physiological Correlate of the 'Spotlight' of Visual Attention. *Nature Neuroscience* **2**:4, pp. 370-374, 1999.

[28] Duzel E., Yonelinas A.P., Mangun G.R., Heinze H.J., and Tulving E. Event-Related Brain Potential Correlates of Two States of Conscious Awareness in Memory. *Proceedings Of The National Academy Of Sciences* **94**, pp. 5973-5978, 1997.

[29] Kane G. Review of The Attentive Brain. *Journal of the American Medical Association* **281**:7, 1999.

[30] Wooldridge, D.E. The Machinery of the Brain (ISBN 0-070-71841-5). New York, NY, McGraw Hill, 1963.

[31] Dennett D. Elbow Room (ISBN 0-262-54042-8). Cambridge, MA, MIT Press, 1984.

[32] Garey M.R., and Johnson D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness (ISBN 0-716-71045-5). San Francisco, CA, W. H. Freeman, 1979.

[33] Libet, B. Neuronal vs. subjective timing for a conscious sensory experience. In P.A. Buser & A. Rougeul-Buser (Eds.), (1978) *Cerebral correlates of conscious experiences: INSERM Symposium* **6**, pp. 69-82. Amsterdam, Elsevier, 1977.

[34] Duyckaerts C., Suarez S., and Hauw J.J. Types of Memories: Clinicopathological Data. *Revue Neurologique* **154**: S8-S17, Suppl. 2, 1998.

Chapter 2 The Bin Packing Problem and the Match Fit Algorithm

2.1 Introduction

The computational abilities of humans and computers are in many ways complementary. Computers are superior at routine, serialized tasks where high degrees of precision are required. Humans are superior at dealing with large and dynamic data flows and unexpected stimuli where highly precise operation is less important. One contributing factor to this are the differing computational architecture used by computers and humans. Computer algorithms have been designed to approach problems with the goal of seeking exact solutions, or at least solutions which are optimal in some sense. To do this, they use as much information about the problem domain as possible. Biological computation is bound by a different set of constraints. In order to deal with a complex and ever changing environment, the computational architecture of the primate brain has evolved to select a subset of relevant information via one or more attentional processes and to make only this information available to the planning centers of the brain [17]. This bottleneck (which sets the contents of "awareness") permits humans to be very good at generalizing, dealing with novel situations, and responding in real time, but less accomplished at finding exact solutions to hard

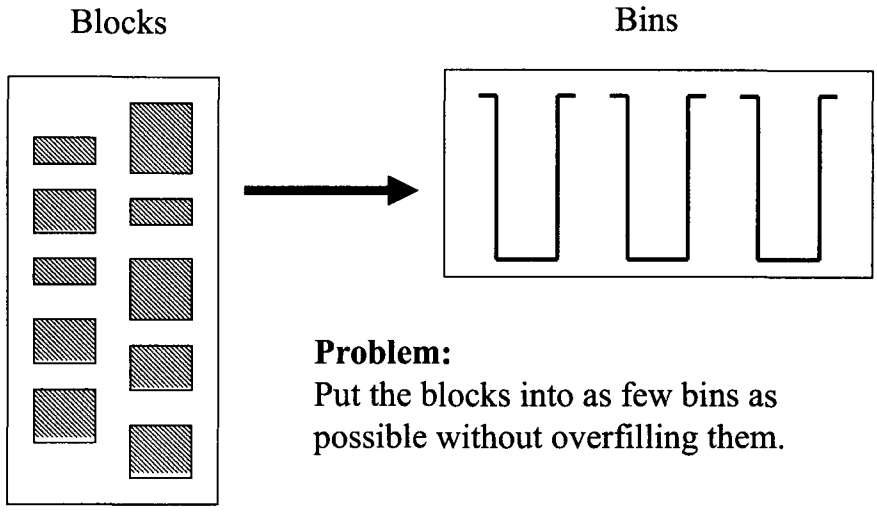

Figure 2.1: The bin packing problem. This simple problem captures the essence of what is difficult about a wide range of computational applications, from trip scheduling to cargo placement to cutting boards from logs.

problems.

We would like to explore conditions under which the demands placed on a computer are more similar to those which the brain handles, situations where highly complex problems defy exact solutions, and where external time pressure forces rapid response, and investigate how algorithms can deal with these constraints.

The bin packing problem is a promising testbed for this. It is a known NP hard problem [15], [5], and is very general, with applications to cutting stock, machine and job scheduling, parallel processing scheduling, FPGA layout, loading problems, and more [16], [12]. In its most basic form, the problem is phrased thus:

> Given a set S of real numbers in $(0, 1]$, we wish to find the smallest possible number k such that there is a partition of S into k subsets, s_i, $i = 1..k$, with $s_i \cap s_j = \emptyset$, $\cup s_i = S$, for $i \neq j$, and $\forall s_i \sum\{s_i\} \leq 1$.

One can think of this problem as an assortment of blocks of varying sizes being fit into bins of unit size. The goal is to fit the blocks into as few bins as possible without overfilling them. The elegance of the bin-packing problem has attracted much attention, including various generalizations such as applications to two and three dimensions [1], [13]. Since finding exact solutions for NP problems is believed to be computationally intractable, researchers have generally attempted to find heuristics which perform well and to analyze this performance. A large number of heuristic approaches have been suggested. These can be classified into online and meta-heuristic approaches. The online approaches (such as the Best Fit algorithm) are in general much, much faster than the meta-heuristic approaches (such as genetic algorithms or simulated annealing).

The only known way to solve the bin packing problem (or any NP-complete problem) is essentially to try every possible solution and see which one is the best. For even very small problems, the computational cost to do this can skyrocket. To solve a bin packing problem of only 20 blocks, this means that on the order of 10^{18} combinations must be tried. If one full packing could be tried every nanosecond, this would yield a solution in about 50 years.

While not solving the $P = NP$ problem, "branch and bound" algorithms can reduce time needed to find exact solutions to NP problems. These algorithms work by cutting off problem branches when it becomes clear that no solution will be found in them. If the root of a particular branch of permutations of a bin packing problem already takes more bins than the best known solution, that set of permutations need

not be examined. These algorithms dramatically reduce the solution time, but they have the weakness of having unpredictable (and still very long) running time. If it were possible to know in advance which branches could be pruned from the search tree, the problem would already be solved. These algorithms, then, are still non-polynomial, but their computational explosion is less dramatic than the more brute-force try-every-solution approach.

If the goal of achieving an exact solution is sacrificed, meta-heuristic algorithms can be used to again reduce the amount of time needed to find a good solution. These algorithms, such as dynamic programming, genetic algorithms, simulated annealing, the various gradient descent methods, and more, all work from the basis of having the complete problem specified, and being able to reshuffle the solutions they have created as better solutions are discovered.

Online algorithms represent the next level of solution time reduction. These are the algorithms which do not start with complete knowledge of the entire problem (although they may accumulate such knowledge as they solve it). Instead, they approach the problem one piece at a time. These kinds of heuristics typically operate much, much faster than the meta-heuristics. Some (like Next Fit) operate at basically the maximum possible speed at which it is possible to solve the problem. Such algorithms have reduced performance relative to meta-heuristic algorithms, especially for harder cases which may be specifically designed to probe the weak points of online algorithms.

Since we are most interested in the time-pressured performance of algorithms,

we will concentrate on the online class of algorithms. Global algorithms, such as reduction procedures [8], genetic algorithms [10], [14], or simulated annealing approaches, typically take orders of magnitude longer to run than online algorithms. These meta-heuristics perform better, but our interest in high performance under tight time constraints rules out this class of algorithms. The four best-known online algorithms are named Best Fit, Worst Fit, First Fit, and Next Fit, and were analyzed by Johnson in [16]. Of these, the two of most interest to us are the Best Fit (BF) and Next Fit (NF) algorithms. The BF algorithm (see Figure 2.2(b)) maintains a list of all partially filled bins it has used so far and compares each new block with the space available in all previously used bins. The bin in which the new block fits with the least leftover space is the one chosen in which to place it. If no bin has enough space to fit the new block, a new bin is allocated for it. In contrast, the NF algorithm (see Figure 2.2(a) maintains no list of past bins. It considers each bin separately and fits blocks into the bin until the next one considered will not fit. It then allocates a new bin and considers it.

As a result, the Next Fit algorithm operates very, very quickly. It does only one comparison per block in the problem above the bare minimum of packing each block into its own bin. The Best Fit algorithm operates quickly compared to meta-heuristic algorithms, but comparatively much slower than Next Fit. For large problems (of 10,000 blocks), this difference can be several orders of magnitude.

To date, assessment of these various algorithms has largely taken the form of worst-case analysis, wherein the worst possible performance is identified, or statistical

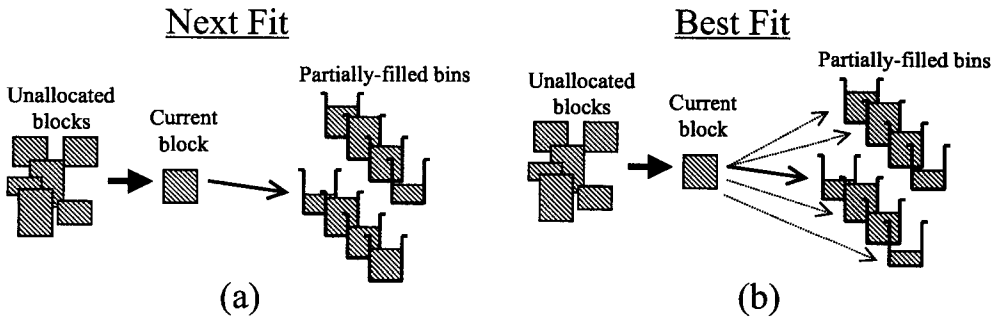

Figure 2.2: Best Fit and Next Fit algorithms. (a) shows the operation of the Next Fit algorithm. NF only checks each new incoming block against one bin in memory. If it fits, it is packed, if it doesn't, a new empty bin is selected and the block is packed into that bin. (b) shows the Best Fit algorithm. BF checks the new block against all bins in memory. It packs the block into the bin into which it fits leaving the least space. If the block will not fit in any bin, BF selects a new empty bin and packs the block into it.

analysis, where the algorithm is applied to large numbers of representative problems and conclusions drawn about the performance of the algorithm. These measures are computationally accessible, but given that the worst-case performance ratio (See Chapter 3) of online algorithms is quite low, from 1.7 for Best Fit to 2 for Next Fit [3], [7], there is not much room for algorithms which perform dramatically better than these.

The worst case performance ratio for an algorithm is the performance of that algorithm on a list of blocks specially designed to produce its worst possible performance. Thus, the algorithm is guaranteed to perform better on any real problem. (This will be discussed in more detail in Chapter 3.)

2.2 Considerations on the Best Fit algorithm

While the performance of Best Fit is already quite good, when we consider a time-constrained problem, where algorithms are under time pressure to produce the best possible packing, then we can explore algorithms which have comparable or slightly improved performance than Best Fit, but which perform at speeds nearer to that of Next Fit.

The Best Fit algorithm has received a great deal of examination, mostly in discovering facts about its performance ratio. The performance ratio is an important quantity in determining the quality of an algorithm, and is defined as

$$\frac{P = N_{actual}}{N_{optimal}} \tag{2.1}$$

where N_{actual} is the number of bins the algorithm actually takes to solve the problem, and $N_{optimal}$ is the least possible number of bins required. That is, the correct solution has a "performance ratio" of 1—it uses exactly N_{optimal} bins in the solution. The use of the performance ratio of an algorithm is hampered by the necessity of knowing the number of blocks in the optimal solution. In all but a few carefully constructed (or heavily analyzed) problems, discovering this quantity is very difficult. For some problems, this quantity can be found by trial and error, usually by finding a solution which can be shown to use the fewest possible number of bins (by adding up the sizes of the blocks in the problem). For many test cases, especially when sufficiently large, this estimate of the optimal solution is very good [8]. When this approximation applies, another way to think of the performance ratio is as the

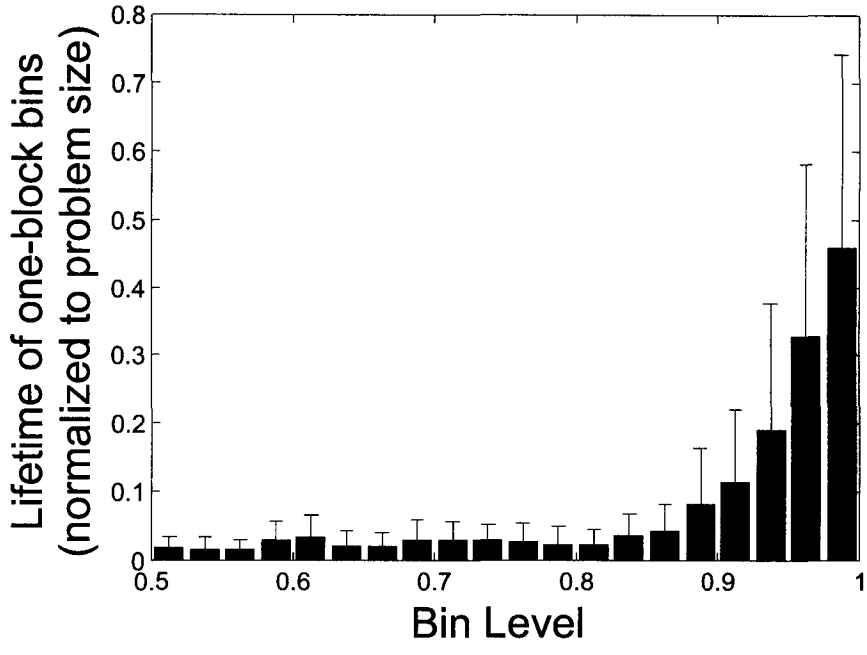

Figure 2.3: Lifetime of bins with various levels in the Best Fit algorithm solution. The bars show how long (compared to the total time taken to solve the problem) a block with one bin in it remains at a certain level. The very long residency times for bins that are quite full indicates that the algorithm is generating bins at these mostly full levels faster than it can find blocks to fill them any more. The high standard deviation results from the lifetimes of the fullest bins being taken over all the bins in the problem, those which were created at the very beginning and those created just at the end.

reciprocal of the average level of bins in the final solution.

One inspiration for our algorithm is a characteristic of the way in which Best Fit treats bins in its interim solutions (see Figure 2.3). Bins below a certain level can be thought of as "in progress," that is, actively being used by the algorithm to pack new blocks. Above a certain bin level, however, the flow of incoming bins is enough to make the algorithm unlikely to work any more with a particular bin at that level. Furthermore, the number of "in progress" bins scales well, increasing very slowly even

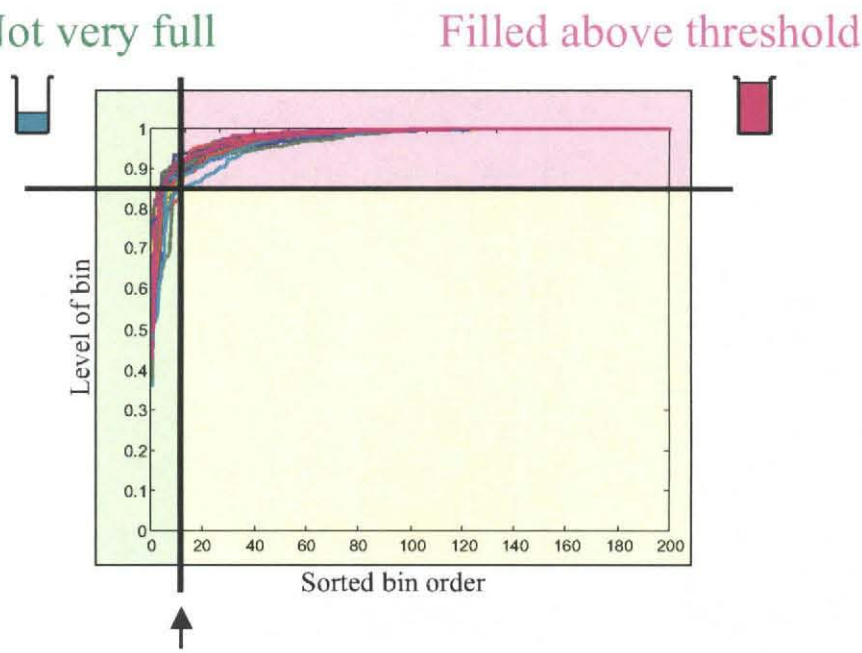

Figure 2.4: Best Fit bin level plot. The various curves are for different runs of the BF algorithm on a 500 block problem. (The algorithm uses around 250 bins to solve the problem.) After the algorithm is done, the bins in the solution are ordered by level and plotted. The bins are size unity and the blocks are uniformly distributed between sizes 0 and 1.

for large problems.

Another way to look at this characteristic of BF is shown in Figure 2.4. The level plots shown are the result of sorting the bins in a BF solution of a 500 block bin packing problem by their level (how full they are). The knee in the level plot indicates the separation of the problem into two groups by the actual operation of the algorithm. The few bins to the left of the knee are not very full, and so are in active consideration by the algorithm. New blocks are very, very likely to end up in one of these bins. Most of the bins lie to the right of the knee and are very full, and so not being used much by Best Fit in its operation. Furthermore, the new blocks which

do fit in these bins are quite small, and so if they are placed in the bins which are being actively considered, they won't have that large of an effect. The arrow marks the number of bins at the knee, which is the approximate number of bins which are being actively used at any one time by BF. This number remains quite constant even for very large problem sizes.

2.3 Match Fit algorithm

We have designed an algorithm which operates in linear time (as does Next Fit), but which uses approximately those bins and blocks which Best Fit would use, and whose performance is thus very close to that of Best Fit. It does this by maintaining at any one time a limited "short-term memory" of bins and blocks, and allocating blocks to the bins in short-term memory to fill them up as well as possible. The algorithm's operation is illustrated in Figure 2.5.

The operation of the algorithm (see Figure 2.5) maintains in memory those bins which are actively involved in the solution of the problem. It does this by limiting the size of its "short-term memory" according to parameter specification, and taking full bins out of this short-term memory to keep within that bound. We have introduced a memory for blocks, as well. The algorithm matches blocks and bins from its memory, which is usually quite small compared to the problem as a whole. Since bins packed in this working memory will not be reexamined, the criteria used for matching attempts to produce bins which are nearly full. Thus, the algorithm takes advantage of the relatively small number of bins in the "active" category for excellent time performance

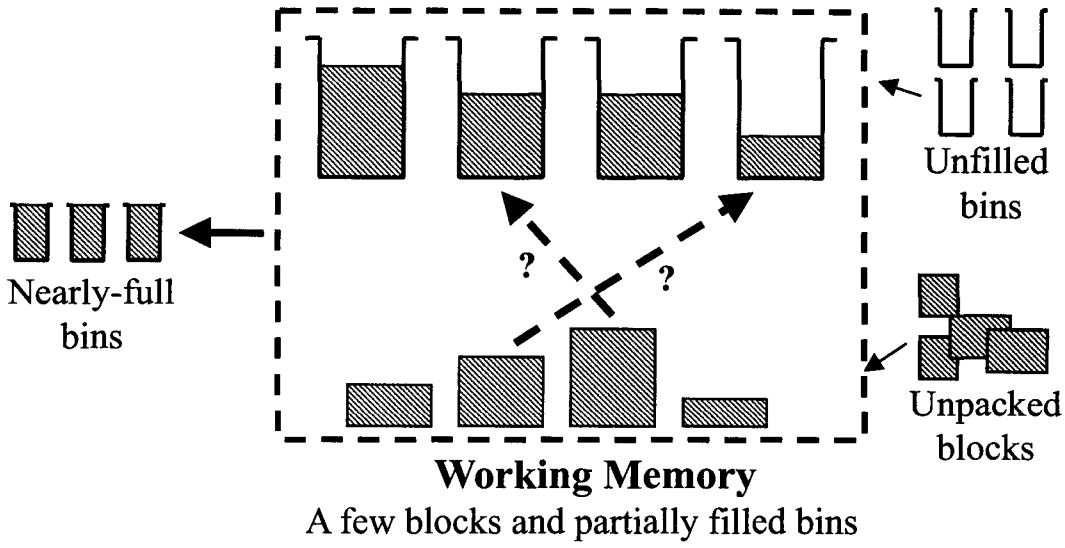

Figure 2.5: The Match Fit algorithm illustrated. The algorithm maintains a memory of bins and blocks. On each cycle it attempts to find matches between bin/block pairs which are within a small threshold of making a full bin. When it finds such a match, it removes the nearly full bin from its memory and refreshes its memory from the queue of waiting blocks and/or new, empty bins. If no such match is found, blocks are put into bins in which they don't match exactly, but the bin is not removed from memory.

	Best Fit	**Match Fit** 3 Bins, 3 Blocks	**Match Fit** 10 Blocks, 6 Bins	**Match Fit** unlimited memory
Performance Ratio (mean)	0.96	0.90	0.96	> 0.99

Table 2.1: Falkenauer data test comparison results

even for large problems (the algorithm is $O(n)$) with a relatively small ($O(1)$) memory. After each iteration, the memory is replenished from any blocks left unpacked. The bin memory is replenished with empty bins. If no suitable matches in memory can be found, the algorithm forces placement of blocks into bins which they don't fill as well, and then replenishes memory.

2.4 Performance of Match Fit algorithm

We have run Match Fit on the Falkenauer [10] test sets and compared its performance to Best Fit, as shown in Figure 2.6 and in Table 2.1. These test sets are standard collections of bin packing problems assembled by Emanuel Falkenauer. The set used in these experiments contains collections of 120, 250, 500, and 1000 block problems (20 problems each).

On the 1000 block collection, which is composed of problems with a uniform normalized block distribution on integers in [20, 100] with bin size 150, Best Fit has a mean performance of 0.96. The performance of the Match Fit algorithm varies with memory size. With a working memory of only three bins and three blocks, it has a

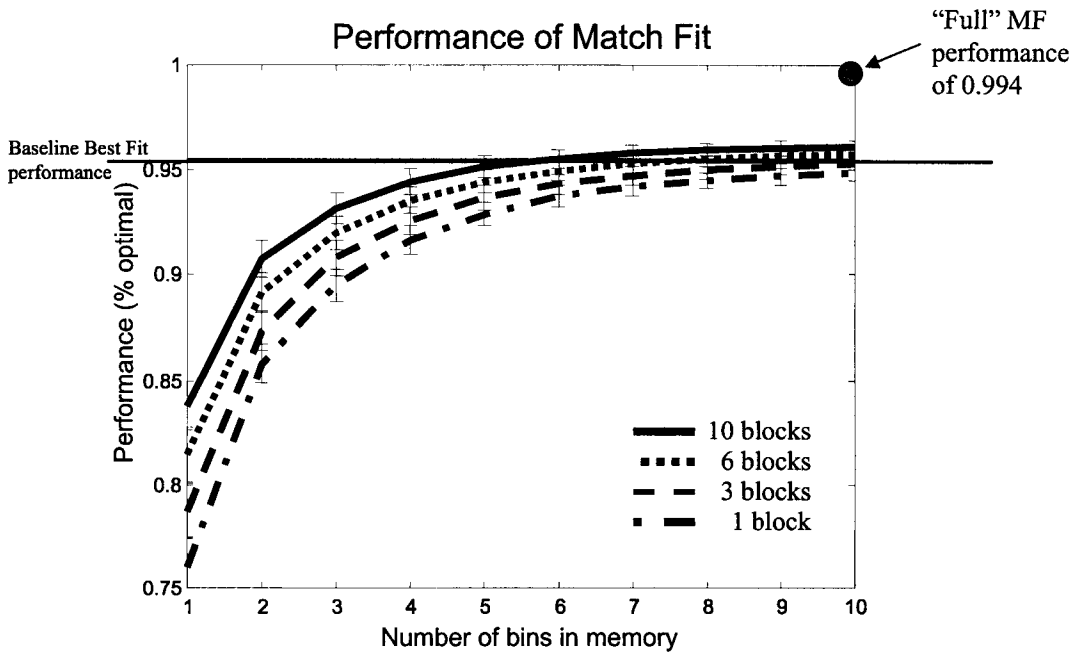

Figure 2.6: Performance of Match Fit on Falkenauer data set. The problems solved are those with 1000 blocks. The blocks are of integer size, distributed on $[20, 100]$ with bin size 150. The "full MF" performance is for the algorithm when using no working memory constraints.

mean performance of 0.90.

For working memory sizes of only ten blocks and six bins, or of six blocks and eight bins, the average performance of Match Fit was 0.96, equal to that of Best Fit. The working memory size is about 2% of the problem size for performance at parity with Best Fit. For larger working memory sizes, the performance gradually improves and outperforms Best Fit, and for very large working memory sizes (comparable to the size of the problem), Match Fit very often yields optimal solutions (which are known for these test problems), with an average performance of 0.994.

In Fig. 2.7, the algorithms performance on a very large problem (10,000 blocks;

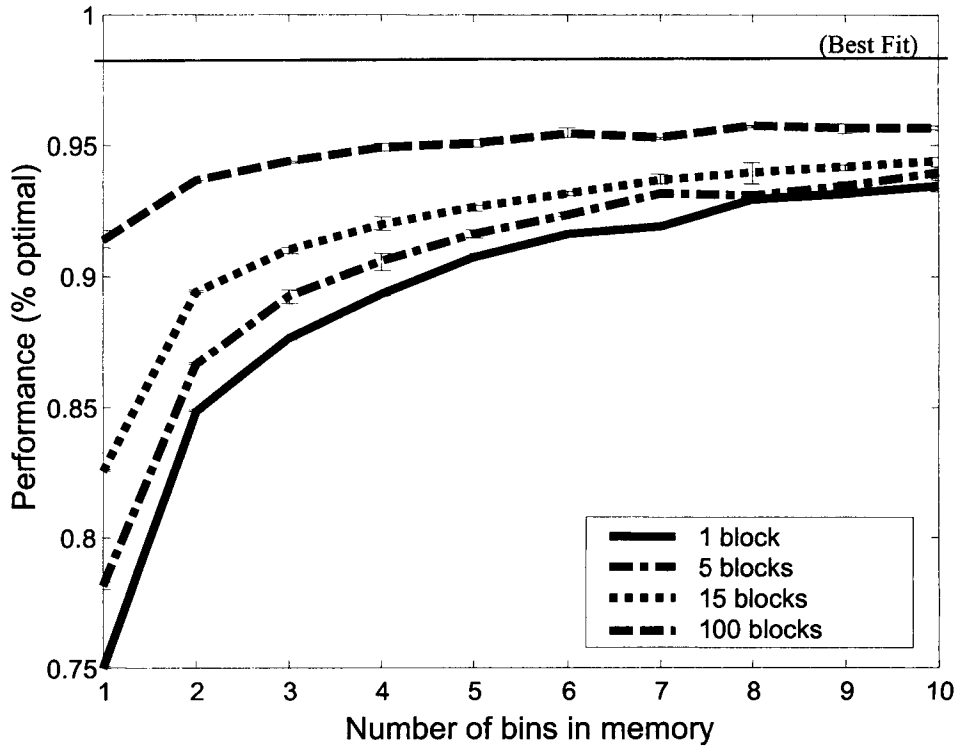

Figure 2.7: Performance of MF algorithm on large test problem. 10,000 blocks; uniform distribution $(0, 1]$ on size. The bins are of size unity. The performance increases as the working memory of the algorithm increases slowly towards the performance of Best Fit (which is 98%).

uniformly distributed in size) is shown. The early saturation of performance suggests that Match Fit will do well under time pressure. The reason is that since the MF algorithm takes a shorter time to operate, and still can produce competitive performance with BF, then when there is not much time to operate, the MF algorithm will be able to keep up and perform better than BF. The saturation is comparable when either bin or block memory is increased. Of the two, increasing block memory offers slightly better marginal performance. This suggests that bin packing algorithms which operate in very resource-limited environments would do well to expand the number of blocks they consider simultaneously alongside, or even before, they expand the number of partially filled bins they consider.

Match Fit can perform better on the Falkenauer test set because of the integer sizes of the blocks. This means that there are pairs or triples of blocks which combine to exactly fill some bins, and many of the solution bins MF finds are of this type. When the blocks have real-valued sizes, this does not happen as often, and there tends to be more space left over in the bins it packs.

The asymptotic performance of the MF algorithm on this problem can be seen in more detail in Figure 2.8. The saturation of the performance suggests that MF will do well under time constraints. From Figure 2.8, we notice that comparing bin and block memory, both have similar affects on performance. Of the two, increasing block memory offers slightly better performance. This suggests that bin packing algorithms which operate in very resource-limited environments would do well to expand the number of blocks they consider simultaneously alongside, or even before,

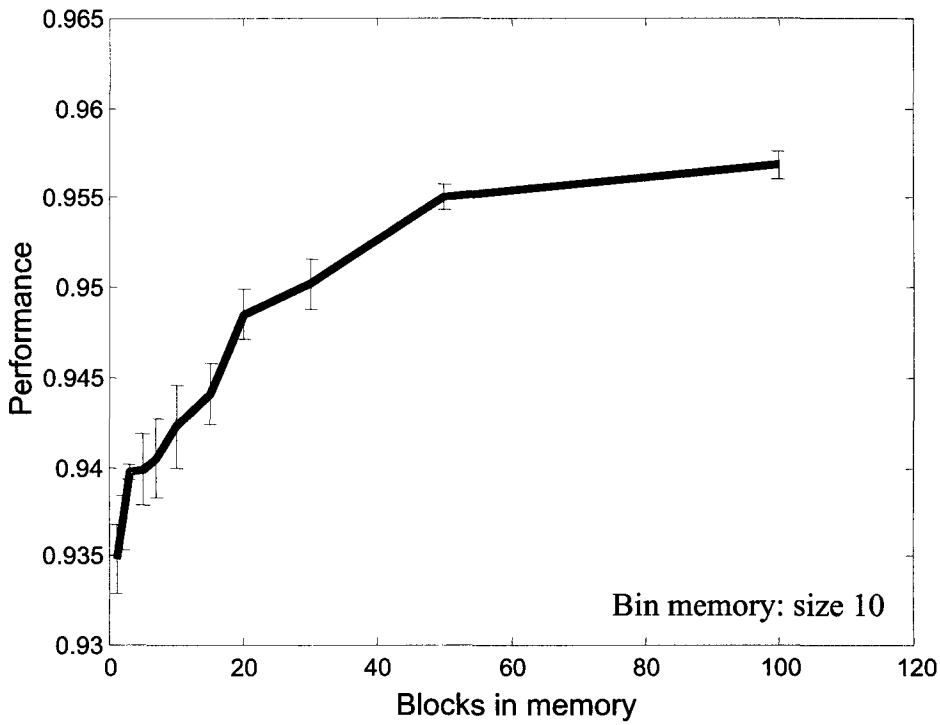

Figure 2.8: Asymptotic performance of MF algorithm on large test problem. Here the bin memory size is fixed at 10 bins. The problem is 10,000 blocks of uniformly distributed size. The bins are size unity.

they expand the number of partially filled bins they consider.

2.5 Time-pressured performance characteristics

Our primary focus in this investigation is the performance under time pressure of these various algorithms. By time pressure we mean the performance of the algorithm in a situation where the best solution is demanded from the algorithm after a particular length of time. The time constraint is enforced by an external controller, which allows the algorithms to run on a problem for a fixed amount of time, and then allocates any remaining blocks unpacked by the algorithm at one block per bin. This is equivalent to blocks passing a real-life packing machine operated by one of these algorithms. If the algorithm could not consider a particular block as it passed (that is, if the blocks passed too quickly), then that block would pass outside the working area of the machine and be placed into its own bin. Figure 2.9 shows a performance comparison between the BF, NF, and a few configurations of the MF algorithm (with five bins and a varying numbers of blocks available to its working memory). The problems being solved by the algorithms are the packing of the 10,000-block problem (uniform $(0, 1]$ distribution on block size) into bins of size unity. As can be seen, the performance of the MF algorithm is intermediate to BF and NF performance.

An examination of the performance characteristics for the MF algorithm indicates that when the algorithm is in its most interesting performance region in terms of its time-pressured performance—that is, performing well, but not yet at its optimum where it would be best to choose the most possible memory—there is an optimum

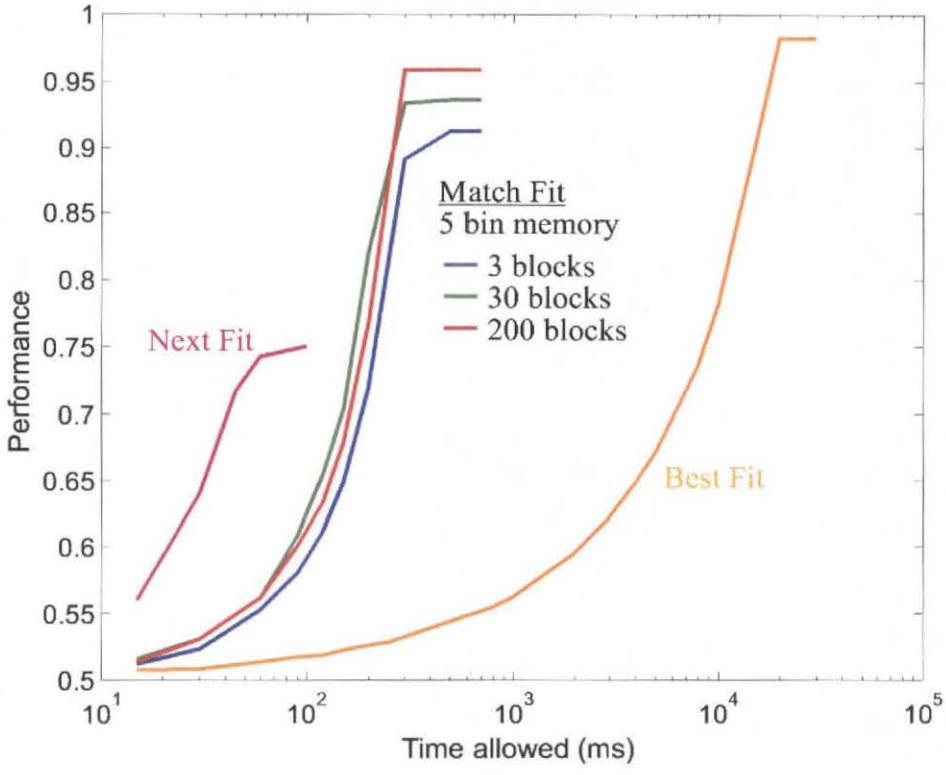

Figure 2.9: Comparison of time-pressured performance of three bin-packing algorithms. The performance of Match-Fit is intermediate to Next Fit and Best Fit. The time allowed for the problem solution is shown in milliseconds, but will scale if a different processor is used for the problem, while maintaining the general shape of the curves. The problem is packing 10,000 blocks uniformly distributed in size over $(0, 1]$ into unity-sized bins.

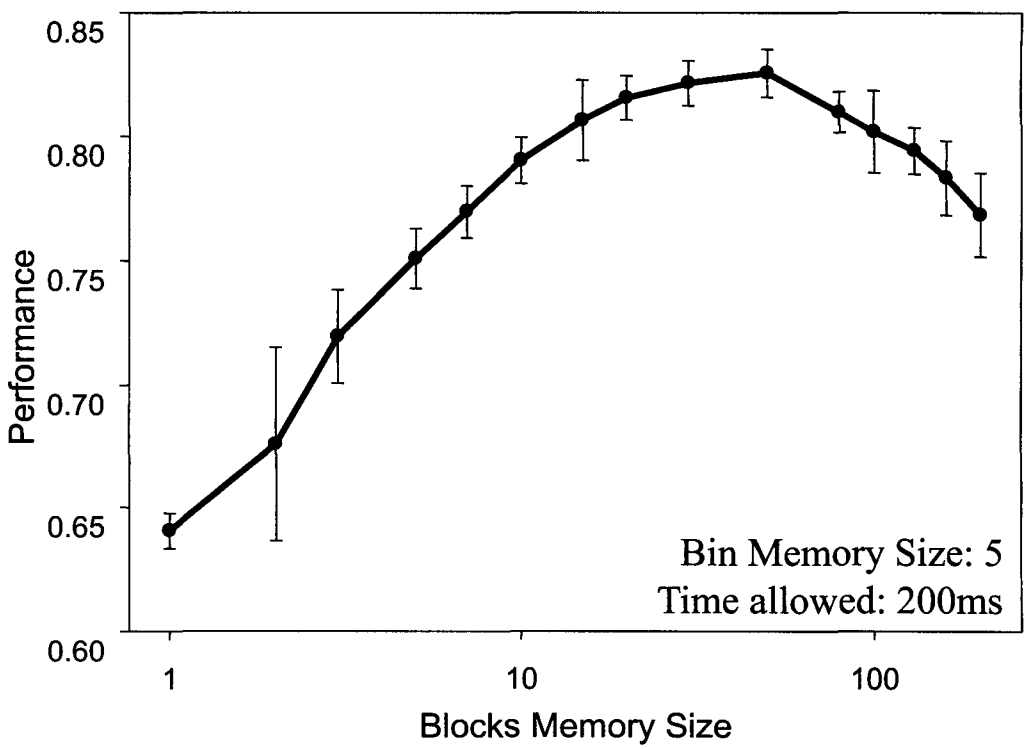

Figure 2.10: Optimality in memory size for time-pressured MF algorithm. Under time pressure, there is a performance optimum for the MF algorithm. Too many items in memory slows the algorithm down too much, whereas with too few items, it does not perform as well. The problem is packing 10,000 blocks uniformly distributed in size over $(0, 1]$ into unity-sized bins. The time pressure here is 90ms allowed per game.

in the amount of working memory the algorithm uses. This is shown more explicitly in Figure 2.10. The optimal for this case (with five bins) is about 25-30 blocks in memory. When fewer blocks are used, the performance is worse because the algorithm doesn't have as good a chance of finding good packings for blocks. When the memory uses more blocks, the performance also decreases, because although good packings are being found, it takes the algorithm longer to find them and runs out of time. In the case of the BF algorithm, which is equivalent to a limiting case of the MF algorithm where an almost unlimited bin memory is allowed (but using a single block

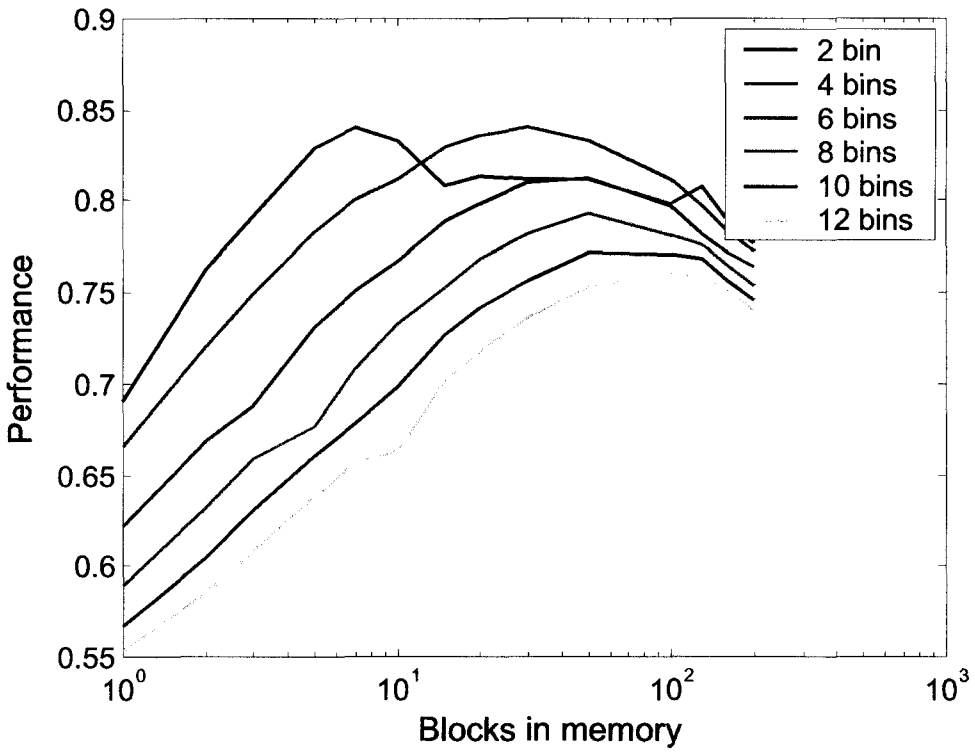

Figure 2.11: Optimality in memory size for time-pressured MF algorithm. Representative curves are shown for various numbers of bins used in working memory. The problem is packing 10,000 blocks uniformly distributed in size over $(0, 1]$ into unity-sized bins.

in memory at a time), the packing is superior, but the time taken is roughly two orders of magnitude more.

The characteristics of this optimum can be seen in more detail in Figure 2.11. The nature of the optimum in working memory size is present for all the combinations of bins and blocks used in the memory.

We are also interested in how the optimal strategy changes as the time pressure eases. Figure 2.12 illustrates that there is a quite abrupt change in the optimal

approach to solving the problem as the time pressure is slowly varied. In this figure, the memory parameters of the algorithm were varied widely (from 1 bin and 1 block to 200 blocks in memory and 20 bins in memory). For each value of time pressure, the various Match Fit algorithms using their memory parameters were run, and the best performer was examined. The top plot of Figure 2.12 shows the best performance of any of the Match Fit algorithm configurations in solving the problem. The bottom plot shows the working memory size (the addition of blocks and bins in memory) at this optimal point. We observe a sharp threshold in the parameter space of the optimal configuration. Below this threshold, the optimal approach to solving the problem is for the algorithm to use a small working memory to best advantage. This remains true as the time pressure eases off and the algorithm is able to perform better and better. When the performance becomes close to its asymptotic limit, however, there is a transition. For time pressures less than this transitional value, the algorithm is better off to use basically as much memory as is available to it (the saturation in working memory size shows reflects the maximum size of 250 used in the simulations). Before the threshold, the performance curves exhibit the clear optimum we anticipate for a system solving a demanding problem in real time: there is an optimum in the amount of resources it should dedicate to the task. As the time pressure eases off, this optimum becomes less pronounced, and the approaches which use more resources start to become attractive.

Why is this threshold so steep? Figure 2.13 presents a way to answer this question.

Before the threshold, the performance curves exhibit the clear optimum we antic-

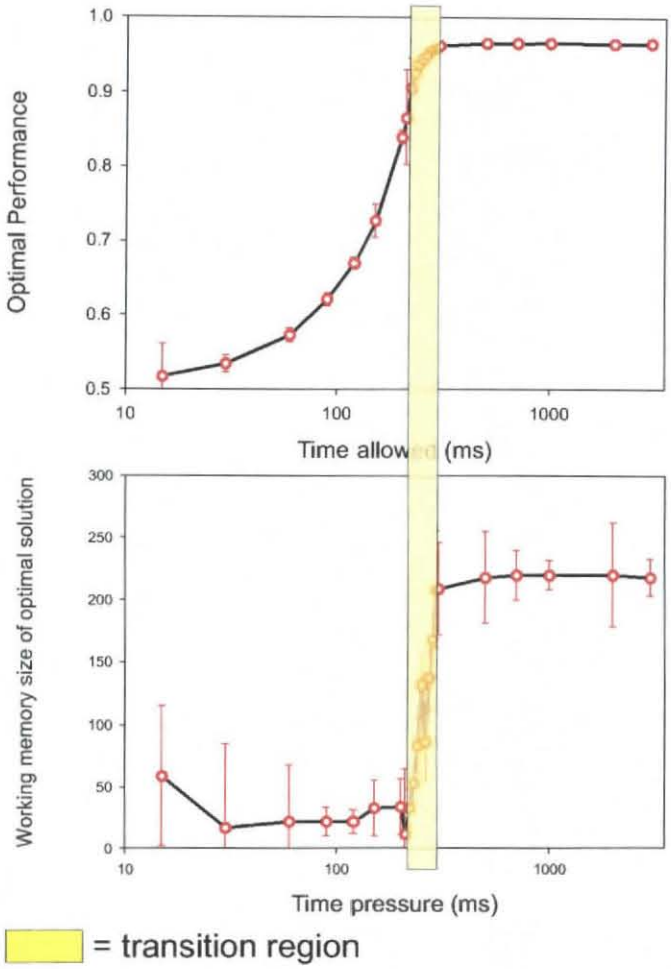

Figure 2.12: The performance of MF as a function of time pressure and the corresponding sudden change in strategy needed to achieve optimal performance. For time pressures such that the algorithm cannot perform at its asymptotic level, the optimal strategy is to use a relatively small working memory. The transition between this regime—that where it is optimal to use a very small working memory and that where it is optimal to use a very large working memory—is extremely sharp. The error bars in the top plot show the standard deviation in performance of the memory configuration with the highest mean performance over 10 runs of the simulation. The error bars in the bottom plot indicate the standard deviations for those values of working memory size for which the performance at a given time pressure was ever the best in any simulation run, and so are quite pessimistic.

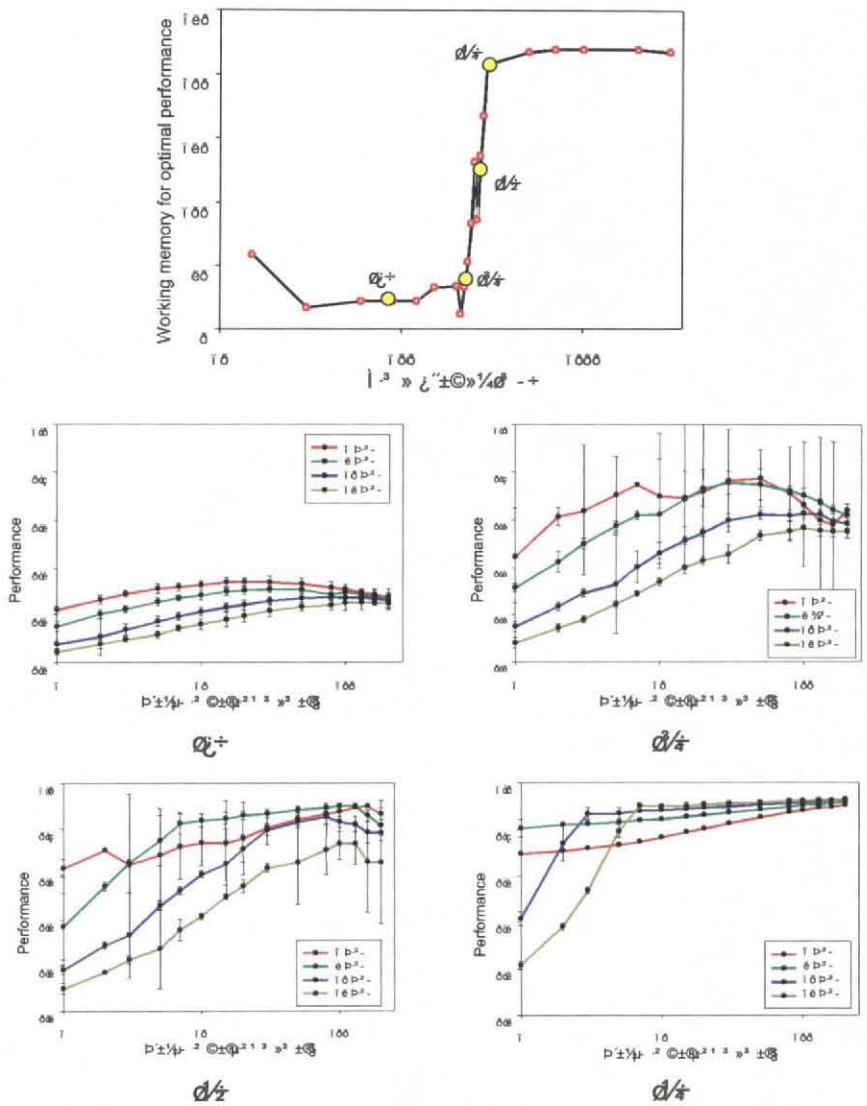

Figure 2.13: Details of MF performance at the phase transition. (a) shows performance curves at 90ms; (b) shows performance curves at 210ms; (c) shows performance curves at 260ms; (d) shows performance curves at 300ms.

ipate for a system solving a demanding problem in real time: there is an optimum in the amount of resources it should dedicate to the task. As the time pressure eases off, this optimum becomes less pronounced, and the approaches which use more resources start to become attractive. Their performance curves no longer have humps, but start to bend (see Figure 2.13(c)) and eventually simply rise asymptotically (as do some curves in Figure 2.13(c) and all in Figure 2.13(d)). When these asymptotic curves outpace the curves with maxima, it is more advantageous to select a resource-intensive algorithm. There effect which drives this process is the fact that the algorithm has an asymptotic performance-with a specific memory size the hump in the performance curve flattens as time pressure eases and overall performance increases. As this happens, it suddenly becomes advantageous to use lots of memory, because the asymptotic performance becomes higher.

2.6 Per-block computational constraints

There is another way to implement a time pressure on the algorithms. Instead of limiting the time pressure on the solution as a whole, we can limit the resources the algorithms spend on a finer scale, at the per-block level. This is straightforward to do with the Best Fit algorithm, we simply give it a certain number of computational "tokens" with which to pack every block and require it to use tokens to do memory accesses, comparisons, stores, etc. For example, it might cost one token to examine the level of a bin (corresponding to a memory access), another token to compare the level with a current working block, and so on). Since Match Fit can use multiple

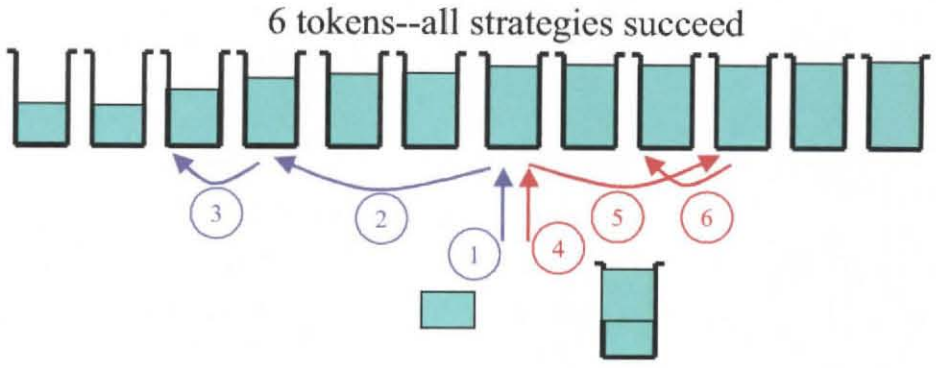

Figure 2.14: Operation of Best Fit when time constraints are low. The algorithm can succeed in keeping an ordered list, and so needs no mechanism to handle situations when it fails to do so.

blocks at once, we can become quite fine grained, but not quite to the per-block level. Instead, we allocate a certain number of tokens for each block it takes into its memory, and let it store up tokens until its memory is full and use them as it solves the problem. The same operations require the Match Fit to give up tokens, and when it runs out of tokens when processing the blocks and bins in its memory, it must start getting new blocks in.

What the algorithms do when they run out of time is an interesting decision. For the Match Fit, when it runs out of time, it is forced to put all the blocks remaining in its working memory into new, empty bins using one block per bin. For example, if it was packing six blocks, and got ten blocks per token, it would start with six blocks in working memory and sixty tokens. If it could only pack four of the blocks with the sixty tokens, the other two would be packed at one block per bin, and then the memory would be reloaded. The Best Fit algorithm we run with a variety of exit strategies. Figure 2.14 shows the situation Best Fit is intended to face—enough computational

resources to perform searches and maintain an ordered list in its memory.

In the strictest exit strategy (shown in Figure 2.15(a)), the algorithm uses half its tokens to conduct a binary search for the best bin. It then packs the block, and uses the rest of the tokens to do the insert. When the problem becomes large enough so that it cannot find the best fit, it gives up and packs the block in a new empty bin. In another strategy (Figure 2.15(b)), the algorithm is not constrained to use half the tokens on searching and half on inserting. It is allowed to do as much searching as it wants, even knowing that it will not be able to re-insert an augmented bin. A third strategy (Figure 2.15(c)) allows BF to keep two lists, ordered and unordered. The ordered list grows until the time pressure doesn't allow bins to be replaced. The bins are then put in an unordered list, and the ordered list will start to shrink. The algorithm behaves in such a way to keep the size of the ordered list right about the point where it can do both searches and replacements. If the list grows too large, it does not have enough time to replace bins, and so the list shrinks as it packs new blocks into the bins from the ordered list. If the list shrinks, the algorithm has enough time to finish searching it and to put in newly created bins with blocks that can't be packed from the list. The fourth strategy (Figure 2.15(d)) has Best Fit give up on keeping an ordered list. As it does its searches and replacements, it simply operates until it runs out of tokens and then takes either the best fit found so far (for packing) or the best location for the bin found so far (for replacement). For large problems, this quickly leads to a list which is barely ordered at all, and "searches" and "replacements" are essentially random. The fifth strategy (Figure 2.15(e)) gives

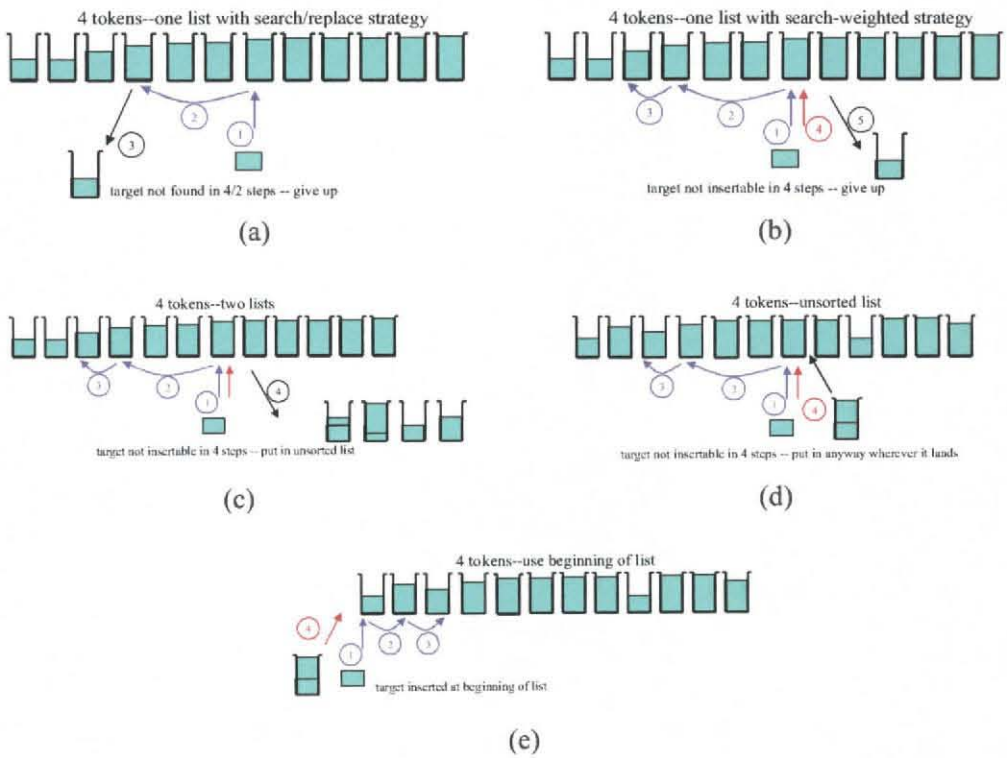

Figure 2.15: Various exit strategies we implement for Best Fit in a resource-constrained environment. (a) shows a strategy where half the resources are allocated for searching and half for replacing repacked bins. When the resources are used up, the new block is assigned to an empty bin. (b) diagrams the case similar to (a), but where the algorithm will use all of its tokens if necessary doing the initial search. (c) is the strategy where the algorithm maintains an unordered list as well and can do searches without replacements if it runs out of resources. In (d) Best Fit does not maintain an ordered list any more, but aborts its searches and replacements when it runs out of resources, taking the best result up until then. (e) shows the case when Best Fit isn't maintaining an ordered list or doing binary searches, but simply using the first part of its list as a "working memory."

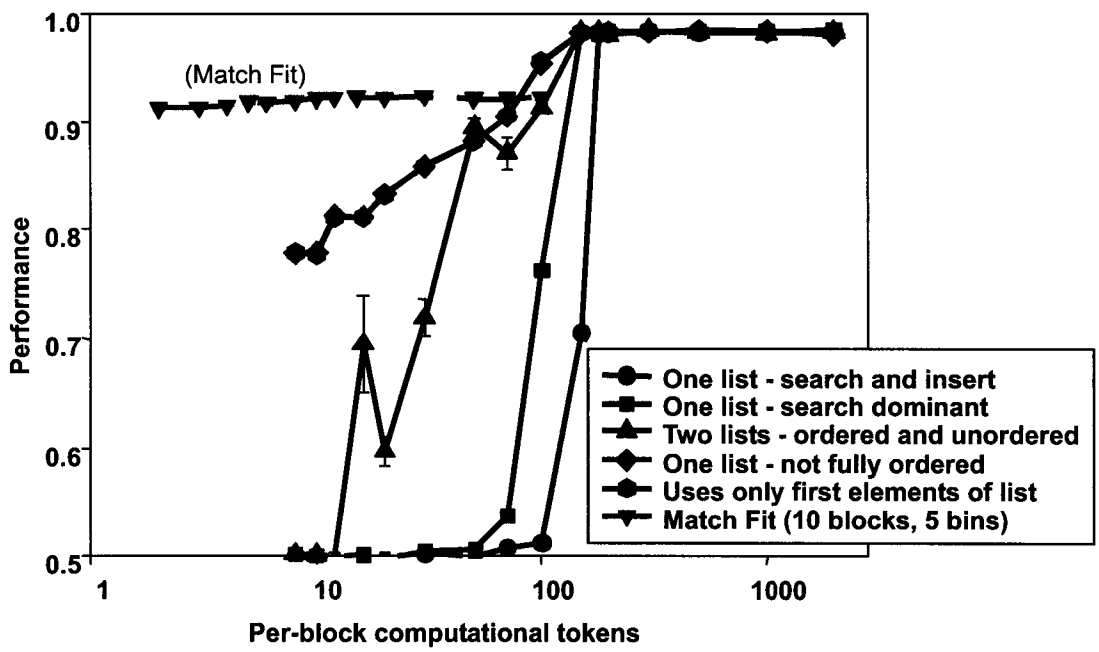

Figure 2.16: Plots of performance as a function of the number of tokens allowed per block (the time pressure as implemented through the use of computational "tokens.") The problem is packing 10,000 blocks uniformly distributed on $(0, 1]$ with unity sized bins.

up completely on an ordered list and maintains a system whereby all the tokens are used to search the first few bins in the list. Newly created bins are placed at the front of the list, while replacement bins (which are assumed to be getting fuller) are placed at the end.

The performance of these strategies is shown in Figure 2.16. The different strategies mentioned perform increasing well. The cutoff in performance to 0.5 is due to the eventual inability of some exit strategies to perform their searches and replacements with the computational resources as given. Those strategies applied to Best Fit which do not give up after their resources are exhausted, but instead compromise the assumptions of Best Fit by not doing a complete search, do better. The Match

Fit algorithm using a memory of 10 blocks and five bins outperforms all the variants of Best Fit under more intense time pressure. As the time pressure eases off, the Best Fit strategies improve to eventually all outperform MF.

The better the Best Fit strategies perform under time pressure, the more they resemble the Match Fit algorithm in the sense of using limited information. The maintenance of two lists-an ordered and unordered list-is similar already to the idea of using a reduced representation of what is important about the problem. In this case, the question of which bins are important is arising from the order in which the blocks happen to be presented to the problem. The strategies which give up on an ordered list, and either operate on essentially random bins in memory, or on the bins clustered towards the front of the problem, perform better, but are even more similar to Match Fit. The items they examine (whether systematically spread through memory or clustered at the beginning) are an even more dramatic reduced representation of the memory used by the BF algorithm.

Figure 2.17 shows how these performance differences evolve during the solution of the problems. At the beginning of the problem, the performance increases until the memory required overwhelms the available computational resources. At that point, the performance either degrades rapidly to 0.5 (for the strategies which give up when they cannot maintain an ordered list), or approach some threshold performance level for those strategies which abandon the Best Fit memory organization. Match Fit rapidly reaches its asymptotic performance level, since it requires a constant amount of computational resources, and given those, it reaches asymptotic performance by

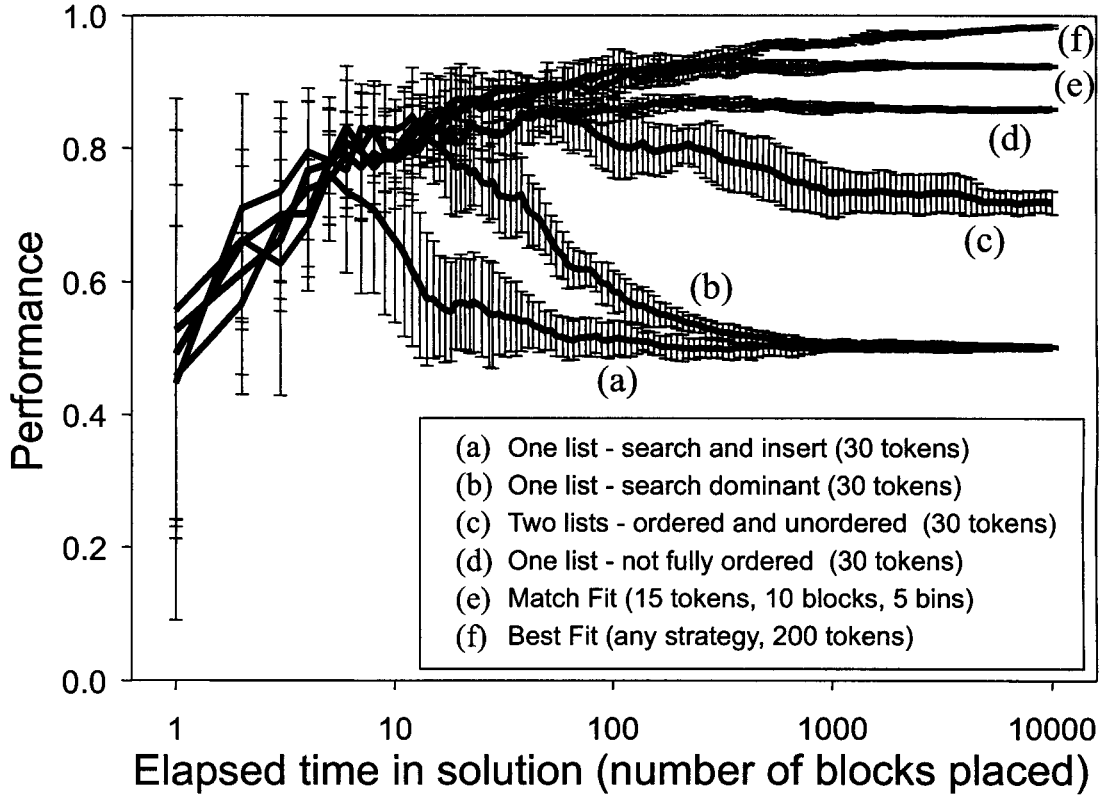

Figure 2.17: Plots of performance throughout the solution of a bin packing problem. The problem is packing 10,000 blocks uniformly distributed on $(0, 1]$ with unity sized bins. The solution time corresponds to the number of blocks packed out of the total problem size of 10,000, so the horizontal axis marks the elapsed time as the problem is being solved. Mean and standard deviation shown for 9 trials (BF) and 5 trials (MF).

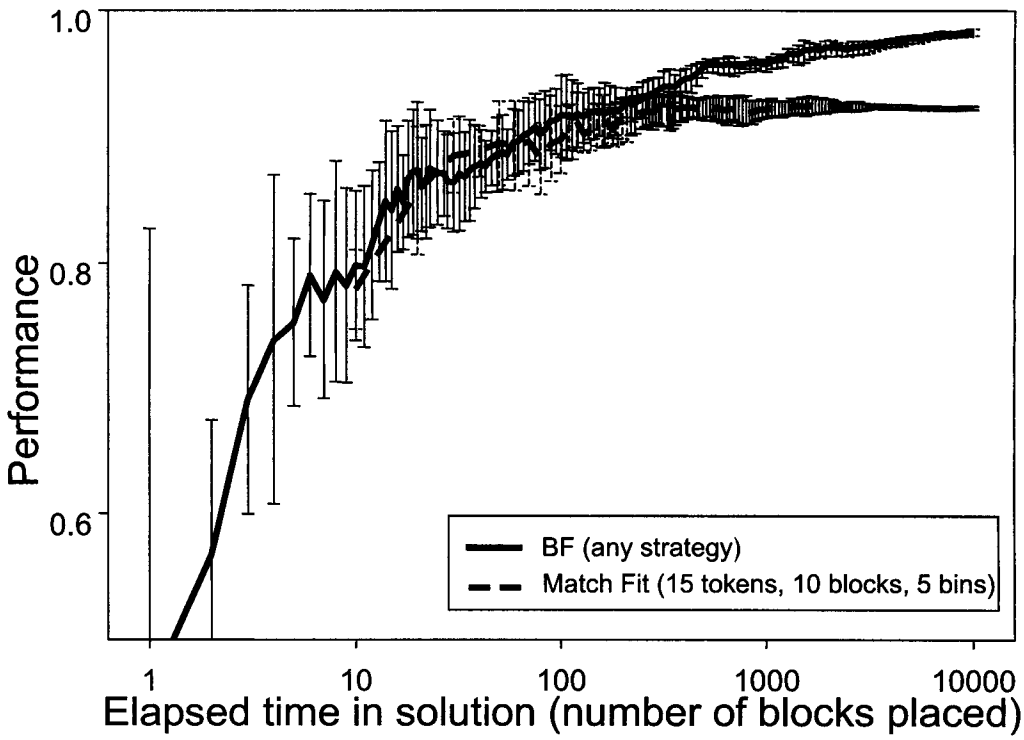

Figure 2.18: Plots of asymptotic performance in the solution of a bin packing problem. The problem is packing 10,000 blocks uniformly distributed on $(0, 1]$ with unity sized bins. The solution time corresponds to the number of blocks packed out of the total problem size of 10,000. Mean and standard deviation shown for 9 trials (BF) and 5 trials (MF).

the time it finishes processing the first few batches of blocks it reads. Given enough computational resources, of course, any BF strategy will look about the same, and the plot is shown for using 200 tokens is representative of any of the strategies. Figure 2.18 shows another view of this case, where the computational resources available are ample for all algorithms.

2.7 Conclusions

The Match Fit algorithm provides an interpolation between several of the interesting online approaches to solving the bin packing problem, a classic NP-complete problem which has analogs to many other interesting and important computational problems. We noted that the bin packing problem is one where we observe a decreasing importance on additional resources utilized in solving the problem, where the information neglected is the "correct" kind. That is, neglecting much of the problem state in the functioning of the algorithm need not lead to great performance deficits. We then devised an algorithm which would take advantage of this idea by utilizing a user-defined amount of computational resources in its solution of the problem.

Since we are able to vary the size of the memory available to the MF algorithm fairly smoothly, we can smoothly observe the impacts of strategies to solution which are resource-intensive and those which do not require so many resources. When not under time pressure, we observe classic asymptotic performance behavior with respect to the amount of resources used, and this asymptotic performance lies between the extremes of the Next Fit algorithm (which uses virtually no resources) and the Best Fit algorithm, which uses more resources. When the algorithm is under time pressure, however, there is an optimum in the computational resources used by the algorithm. This corresponds to the difference between the usual computer and human, or, more general, neurobiological computational strengths. When there is less time pressure, and when exact solutions are desired, the approach which uses the most information about the problem is favored. Under conditions where time pressures are important,

it is best to severely restrict the amount of information considered by the planning parts of the algorithm.

Under increasing time pressure, we observe that there is a very sharp threshold between the optimal performance of the two approaches. When time pressure is severe, that is, there is not enough time to quite get to near-asymptotic performance, it is advantageous to select a strategy which uses very few computational resources. When the time pressure is not so severe, it very quickly becomes advantageous to use very large (relative to the previous case) amounts of computational resources. The reason for this is that under intense time pressure, the system switches from performance being asymptotic in the amount of resources used to having an optimum. Instead of providing a performance boost, having more resources available simply "distracts" the algorithm and slows it down as it has to take time to take the extra information into account.

Furthermore, we observe that under per-block time pressure, those strategies that are reasonable for the Best Fit algorithm to take to increase its performance begin to look more and more similar to algorithms which systematically limit their use of computational resources by using only the information in the problem which is most valuable. The Match Fit algorithm still outperforms Best Fit in this regime, though, because it takes a more systematic view of what parts of the interim solution are important and which are not.

These three lessons—an optimum in the performance vs. resource utilization curve when the algorithm operates in time constrained environment, and the sudden transi-

tion from a limited-memory optimum to a large-memory optimum, and the transition of the best strategy for a more comprehensive strategy to take under time pressure looking more and more like a reduced representation-using strategy—we believe are extensible to a wide variety of computationally interesting problems. The bin packing problem shares with many other interesting problems the characteristic that it is very hard to solve exactly, but relatively easy to get close. When there is this smooth measure on performance (instead of the more binary case of, say, the k-SAT [11] problem), we expect to observe these three phenomena in real-time algorithms.

2.8 Appendix

Pseudocode of Match Fit Algorithm

```
procedure MatchFit
    while (there are unassigned blocks)
        refill block working memory from unassigned blocks
        call FitBlocks
        for (any blocks left unassigned)
            call GauranteeFitBlock
        endfor
    endwhile
endprocedure MatchFit
procedure FitBlocks
    for (each bin in working memory)
        for (each other bin in working memory; not counting already compared pairs)
            if (this pair of bins has combined fill levels which cross threshold into "nearly full")
                combine bins, remove both from working memory
            endif
        endfor
    endfor
    for (each block in working memory)
        for (each other block in working memory; not counting already compared pairs)
```

```
            if (this pair of blocks has combined size which are above threshold to "nearly full")
                combine blocks into new bin, remove both from working memory
            endif
        endfor
        for (each bin in working memory)
            if (this block would fill this bin to above threshold for "nearly full")
                add this block to this bin; remove both from working memory
            endif
        endfor
    endfor
endprocedure FitBlocks
procedure GauranteeFitBlock
    set B as Best Fit bin for this block
    (unallocated bin with least empty space remaining when block is inserted)
    if (B exists)
        add block to B
    else
        add block to new, empty Bin
        if (exists empty slot in working memory for new Bin)
            insert new Bin in first empty slot found
        else
            remove fullest bin from working memory
```

```
            insert new Bin in just-freed spot
        endif
    endif
endprocedure GauranteeFitBlock
```

Bibliography

[1] Csirik, J., J. B. G. Frenk, and M. Labbé. Two-Dimensional Rectangle Packing: On-line Methods and Results. *Discrete Applied Mathematics* **45**, pp. 197-204, 1993.

[2] Coffman, E. G., M. R. Garey, and D.S. Johnson. Approximation Algorithms for Bin Packing: An Updated Survey, in *Algorithm Design for Computer System Design*, G. Ausiello, M. Lucertini, and P. Serafini eds., Springer-Verlag, NY, pp. 49-106, 1984.

[3] Johnson, D. S., A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM Journal of Computing* **3**, pp. 299-326, 1974.

[4] Johnson, D. S. Fast Algorithms for Bin-Packing. *Journal of Computer Systems Science* **8**, pp. 272-314, 1974.

[5] Karp, R. M. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher eds. Plenum Press, NY. pp. 85-104, 1972.

[6] Ivkovic, Z., and E. L. Lloyd. Fully Dynamic Algorithms for Bin Packing: Being (Mostly) Myopic Helps. *SIAM Journal on Computing* **28**:2, pp. 574-611, 1998.

[7] Mao, W. Tight Worst-case Performance Bounds for Next-k-Fit Bin Packing. *SIAM Journal on Computing* **22**:1, pp. 46-56, 1993.

[8] Martello, S., and P. Toth. Lower Bounds and Reduction Procedures for the Bin Packing Problem. *Discrete Applied Mathematics* **28**, pp. 59-70, 1990.

[9] Eilon, S. and N. Christofides. The Loading Problem. *Management Science* **17**, pp. 259-267, 1971.

[10] Falkenauer, E. A Hybrid Grouping Genetic Algorithm for Bin Packing. Working paper *CRIF Industrial Management and Automation, CP 106 - P4*, 50 av. F.D.Roosevelt, B-1050 Brussels, Belgium, 1996.

[11] Monasson, R., R. Zecchina, S. Kirkpatrick, B. Selman and L. Troyansky. Determining Computational Complexity from Characteristic 'Phase Transitions'. *Nature* **400**:6740, pp. 133-137, 1999.

[12] Izumi T., T. Yokomaru, A. Takahashi, and Y. Kajitani. Computational Complexity Analysis of Set-Bin-Packing Problem. *IEICE Transactions on Fundamentals Of Electronics Communications and Computer Sciences* **5**, pp. 842-849, 1998.

[13] Lodi, A., S. Martello and D. Vigo. Approximation Algorithms for the Oriented Two-Dimensional Bin Packing Problem. *European Journal of Operational Research* **112**:1, pp. 158-166, 1999.

[14] Reeves, C. Hybrid Genetic Algorithms for Bin-Packing and Related Problems. *Annals Of Operations Research* **63**, pp. 371-396, 1996.

[15] Garey, M. R. and D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, CA, 1979.

[16] Johnson, D. S. Near-Optimal Bin Packing Algorithms. Ph. D. Thesis, Technical Report Mac TR-109, Massachusetts Institute of Technology, Cambridge, MA, 1973.

[17] Crick, F. and C. Koch. Consciousness and Neuroscience. *Cerebral Cortex* **8**, pp. 97-107, 1998.

Chapter 3 Modeling of Online Bin Packing Algorithms

3.1 Introduction

As mentioned in Chapter 2, most analysis of online bin packing heuristics has focused either on the mathematically accessible worst-case performance of the algorithms (e.g., [3], [1], [7]) or on statistical approaches (e.g., [6]). In this chapter, we will be concerned with modeling the online algorithms we've discussed in terms of both their behavior under time pressure and, in the case of the Best Fit algorithm, in developing a way to estimate the expected performance of the algorithm.

3.2 Worst-case performance bounds

The worst-case performance bound for a bin packing algorithm was explored by Johnson et al. [1] and subsequently others, such as [8]. The worst-case performance ratio is defined for a bin-packing heuristic as the least value ν such that $\frac{\nu \geq P_h(L)}{OPT(L)}$ for all L, where $P_h(L)$ is the performance of heuristic h on the list of blocks L, and $OPT(L)$ is the optimal performance for that list.

As noted in Chapter 2, the worst-case performance bound for the Next Fit algo-

rithm is 2, and the bound for Best Fit is 1.7. The trick to constructing these lists lies in two facts about these online algorithms. The first is that they have to handle the blocks in the order they are presented. This is part of what makes them "on-line" algorithms, and means that the order of the blocks in the list is very important. The second fact is that these algorithms do not reassign blocks based on subsequent information.

The strategy for composing the worst-case list for Next Fit is shown in Figure 3.1.

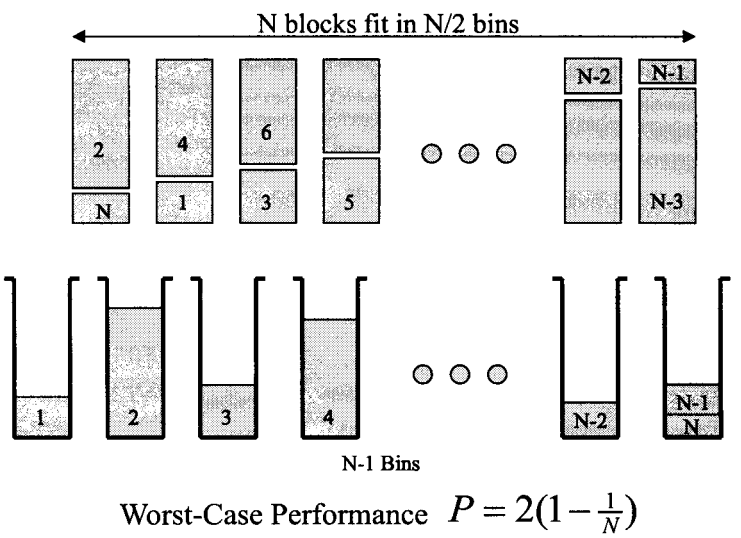

Figure 3.1: Worst-case list construction for the Next Fit algorithm. The blocks are constructed as indicated from bins which are completely filled and then ordered in presentation as indicated by the numbers on the blocks. The bottom half of the figure shows how the blocks are placed in bins by the algorithm.

The Next Fit algorithm can be induced to perform very badly, because it only takes into account one bin at a time. This results in very fast operation, but in the worst case, it can be fooled into packing many bins almost empty.

The partial strategy for composing the worst-case list for Best Fit is shown in

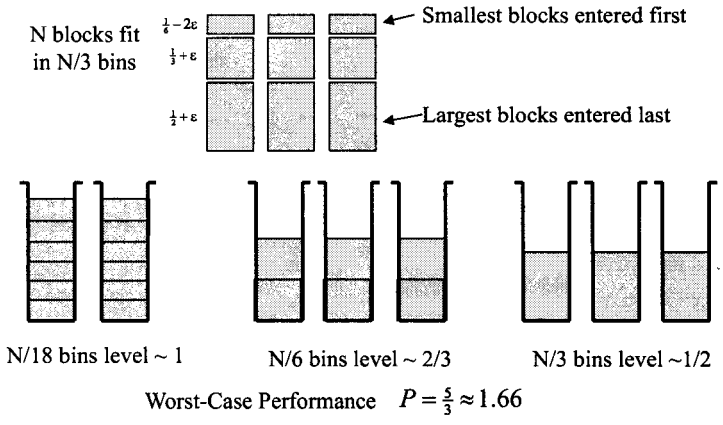

Figure 3.2: Worst-case list construction for the Best Fit algorithm. The blocks are constructed as indicated and then arranged from smallest to largest. ϵ is some very small number.

Figure 3.2.

Here the situation is more complex. Since Best Fit has a memory, it will not fall into the same trap of packing many bins mostly empty. It will find such bins in its memory and fill them better. (In fact, BF will perform optimally on the worst-case NF problem.) The worst that BF will do is when presented with blocks very close to $1/N$ where N is an integer. Figure 3.2 shows the trick. The blocks are chosen such that the smaller blocks are slightly under the $1/N$ point and the larger blocks are slightly over. When presented in order, this means BF will pack the smaller blocks all together, and then when it comes to the larger blocks, it will have to leave the bins more empty (up to half empty for blocks sized $1/2+\epsilon$). The case shown produces a performance of $5/3$ but it can be extended to produce worst-case performance bound of 1.7.

This same strategy can be used for Match Fit, and, in general, for any online

algorithm with finite block memory (provided it is sufficiently high performing). This means that the worst-case performance bound for Match Fit, and for any bin packing non-reallocating algorithm which uses a finite memory size, is 1.7. Match Fit will perform better than Best Fit on the transitions between the different block sizes for this list, but asymptotically, that does not impact the worst-case performance.

The reason is that we can always prepare sample problems following the strategy of Johnson, which exploit the non-reallocating nature of the algorithm by tricking it into packing many bins only half-full. In the case of the Next Fit algorithm, the worst-case tricks the algorithm into packing many bins less than half full, producing a worst-case performance ratio of 2. When addressed to (sufficiently high-performing) algorithms with memory, this strategy cannot work. The algorithm is not susceptible to the method of presenting blocks alternating between large and small, because it has a memory: it will find the bin created two steps previously and use the still-mostly-empty bin. Thus, it cannot be fooled into packing many blocks mostly empty—if it has a memory greater than Next Fit (which is the smallest possible memory for an online algorithm to have), it will not fall into the trap which NF does. However, if the algorithm has a finite memory, a problem large enough exists so that the method used by Johnson et al. [1] can apply.

For algorithms with unlimited memory which nevertheless pack a new block as soon as it comes in (such as BF) the bound will hold. For those which have unlimited memory but do not pack a block as soon as it comes in (such as Best Fit Decreasing, where the blocks in the problem are sorted in decreasing order of size and then BF

is run on the resulting list), this bound will not hold. However, such algorithms are not "online" any more, in the sense that they must deal with the problem as a whole (in order to sort it, assign genetic algorithm markers to the blocks, etc.).

Even reallocating algorithms are subject to this bound, so long as they use a finite working memory. Asymptotically, the problem can be made large enough to swamp their memory, and since all the blocks in the list are the same size, there is no benefit in reallocating the packing. These algorithms have the potential to perform better on the transitions between block sizes than BF, but again, this does not change the asymptotic bound.

For a large class of online algorithms, then, including Match Fit and Best Fit, the best possible worst-case performance bound is 1.7. For a larger class, that is, those that use their memory advantageously, the worst-case performance bound is less than 2.

3.3 Time-constrained performance modeling

We know turn to constructing time-constrained performance models for the three algorithms, Next Fit, Best Fit, and Match Fit for the case where time constraints are implemented as an overall problem time constraint. The time pressure considered here is the same as the problem-level constraint in Chapter 2—the time constraint is enforced by only allowing the algorithms to run for a fixed amount of time, after which any unallocated blocks are packed at one block per bin.

3.3.1 Next Fit

For NF, the time order of the algorithm is $P \sim t$. So for a problem size of N blocks, where NF has an asymptotic performance P_0, the performance will be

$$P = \frac{N}{\frac{1}{P_0}N_{NF} + 2(N - N_{NF})} \tag{3.1}$$

This means that the overall performance will be a weighted average of the performance of NF on the portion of the problem that it solves, and the performance on the portion it does not solve (which is 2).

The size of the problem processed by NF in time t is

$$N_{NF} = \frac{N}{t_0}t \tag{3.2}$$

where t_0 is the scaling time which is the time needed by NF to solve the problem when there are no time constraints imposed. Combining,

$$P = \frac{1}{2 + \frac{t}{t_0}(\frac{1}{P_0} - 2)} \tag{3.3}$$

As can be seen in Figure 3.3, this fit corresponds very closely to the observed data.

3.3.2 Best Fit

For Best Fit, we have the same kind of analysis, with

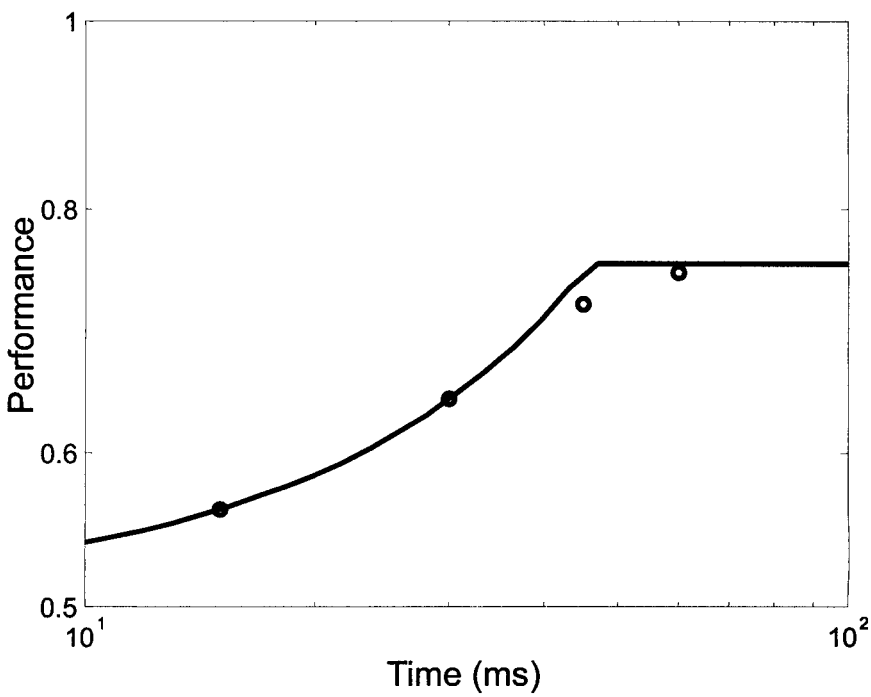

Figure 3.3: Time-constrained performance model for Next Fit. The line is the model and the circles are the simulation data.

$$P = \frac{N}{\frac{1}{P_0}N_{BF} + 2(N - N_{BF})} \tag{3.4}$$

and

$$N_{BF} = \alpha N\sqrt{\frac{t}{t_0}} \tag{3.5}$$

which give

$$P = \frac{1}{2 - \alpha\sqrt{\frac{t}{t_0}}(2 - \frac{1}{P_0})} \tag{3.6}$$

The fit of this to the simulation is also very close, as in Figure 3.4, this fit corresponds very closely to the observed data.

3.3.3 Match Fit

The situation for Match Fit is more complicated. The algorithm doesn't always perform the same actions in an iteration, since what it does depends on the blocks and bins it is actually working with (and what it had on a previous iteration), so the performance model is not simply quadratic or linear with the size of the working memory. We can use a fitted model to understand better how the performance of the algorithm, and the time it takes to execute, vary with respect to the size of its working memory in blocks and bins.

The model used for fitting the time order of Match Fit is a cubic log model:

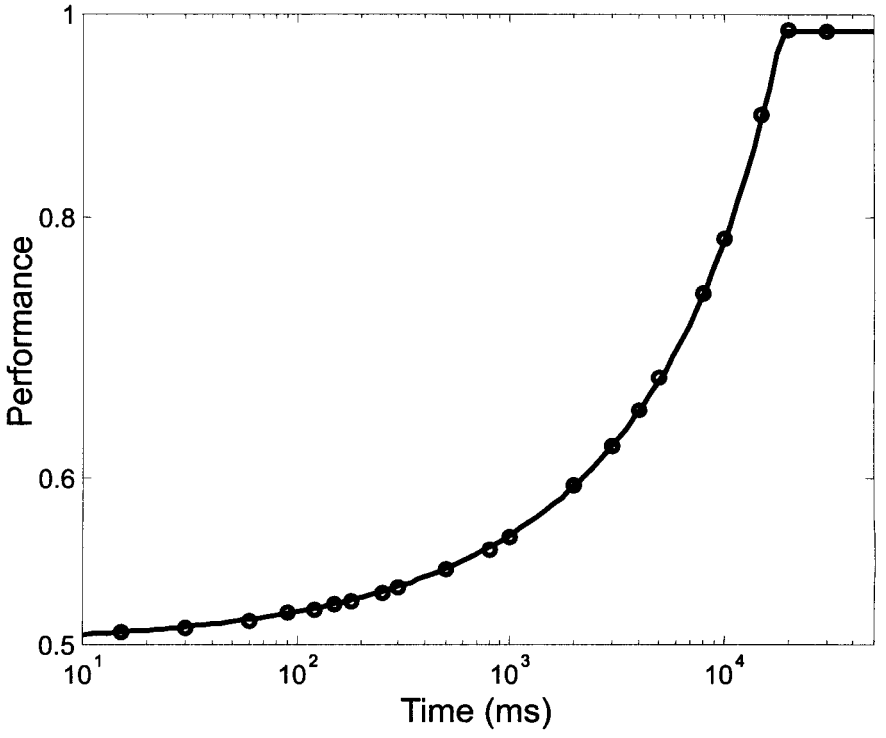

Figure 3.4: Time-constrained performance model for Best Fit. The line is the model and the circles are the simulation data.

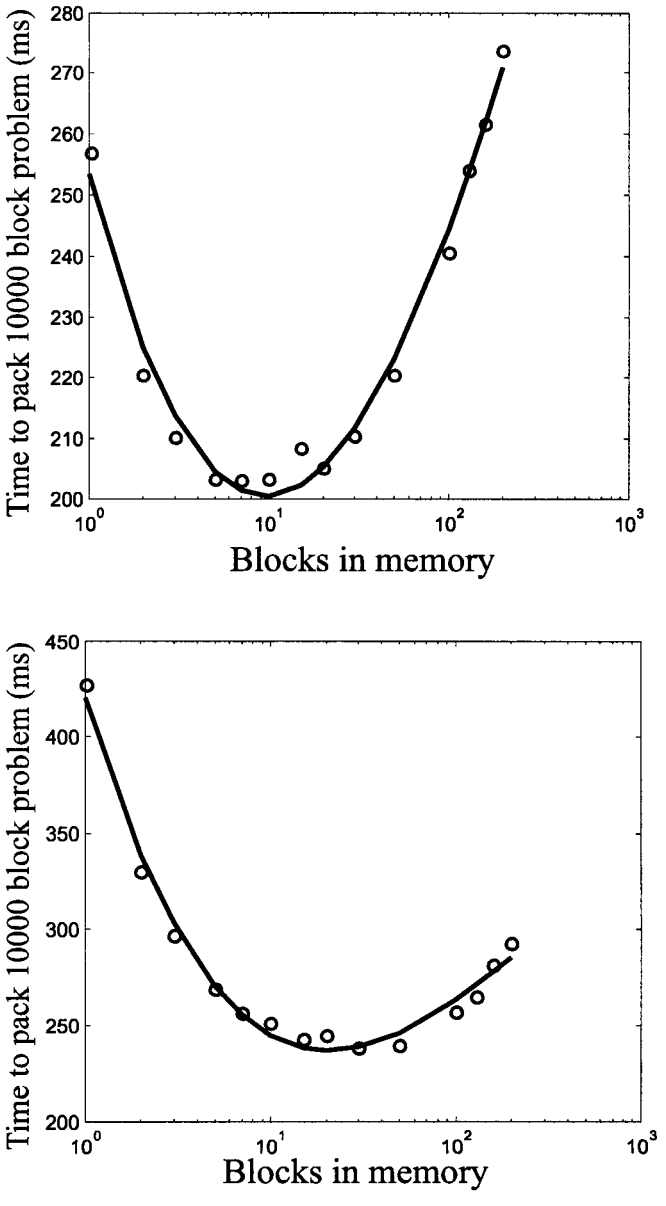

Figure 3.5: Time order model for Match Fit. Plots are shown for one bin in working memory, and for seven bins in working memory. The solid lines are the model fits, and the circles are the simulation data.

$$t = a(\log \text{memory blocks})^3 + b(\log \text{memory blocks})^2 + c \log \text{memory blocks} + d \quad (3.7)$$

From inspecting the algorithm, it is not immediately clear that this model should provide a good approximation to the algorithm execution time. However, the execution time is close to quadratic when seen on the log scale. The reason for this is that there are two competing effects in the algorithm. First, as the number of blocks in working memory grows, the algorithm takes longer per cycle to work through its memory. This is offset, though, by the "short-circuit" effect of the analysis: if the algorithm finds a good match which places a bin over the threshold into the "very-nearly-full" category, it does not reexamine that block or bin—it packs the block and moves on, and does not need to reassess that block or bin in the future. As the number of blocks in working memory grows, it becomes more likely to find such short-circuiting matches. This effect is especially strong when there are small numbers of bins in memory. As the working memory grows larger, though, despite finding some short circuits, the algorithm still has to examine large numbers of blocks. This is especially true when there are few bins in working memory, and for the case of one bin, the curve is very nearly quadratic on the log scale. As there get to be more bins in the memory, the quadratic effect of still having to process through those blocks which don't find matches becomes much less important, since there is more and more of a chance that all will find good matches. So as the number of bins in memory increases, the algorithm settles to close to steady state performance corresponding to

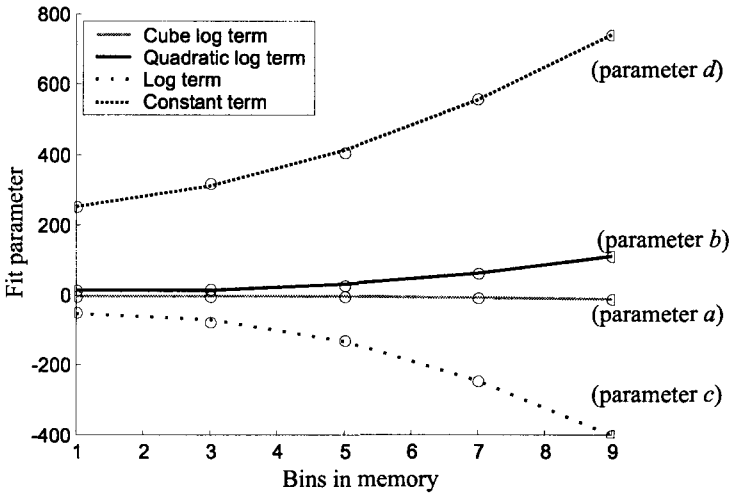

Figure 3.6: Fits of the time model parameters with bin memory size for Match Fit.

good probabilistic match-finding for all blocks in working memory.

The reason the curve is most clear on the log scale is due to the combinatorics involved with the distribution of sizes on the blocks in the problem. If the blocks were distributed differently, the scaling would change somewhat.

Figure 3.6 shows quadratic fits to the time order parameters discussed previously, as the number of bins in memory changes. The quadratic changes with bin memory size relate to the operation of the algorithm: the pair-wise matching of the bins to blocks in memory indicates that changes to the parameters will come quadratically.

These considerations leave us with a 12-parameter model which does quite well in approximating the time-order performance of Match Fit. Such a complex empirical model is only necessary due to the fact that the algorithm doesn't always behave in the same way. Given different problems, it may take less time to operate. Nevertheless, we can place a firm $O(N)$ boundary on the execution time, which grows quadratically

as the memory size increases. This is because the algorithm will make at most $W^2/2$ comparisons, where W is the size of the working memory used.

We now turn to an empirical model of the performance attained by Match Fit.

The performance model for the performance data shown in Figure 3.7 is exponential:

$$P = ae^{b(\text{memory blocks})} + c \tag{3.8}$$

An exponential model of performance matches the above interpretation of the model for algorithm execution time. Since performance depends on the ability of the algorithm to find matches between bins and blocks, then as the size of working memory increases, the performance should improve as well, and the rate of improvement should go proportionately to the size of the memory. Since the performance must saturate as memory grows larger, a single exponential is the simplest model. A double-exponential model fits the data even better at this level, perhaps reflecting the modes of matching performed by the algorithm: blocks with other blocks and blocks with bins. We do not observe the same smooth change of fitting parameters as the number of bins is varied, however, so we achieve better overall modeling with the single exponential approximation.

The time and performance models are combined for Match Fit in the same was as for Best Fit and Next Fit:

$$P = \frac{N}{\frac{1}{P_0}N_{MF} + 2(N - N_{MF})} \tag{3.9}$$

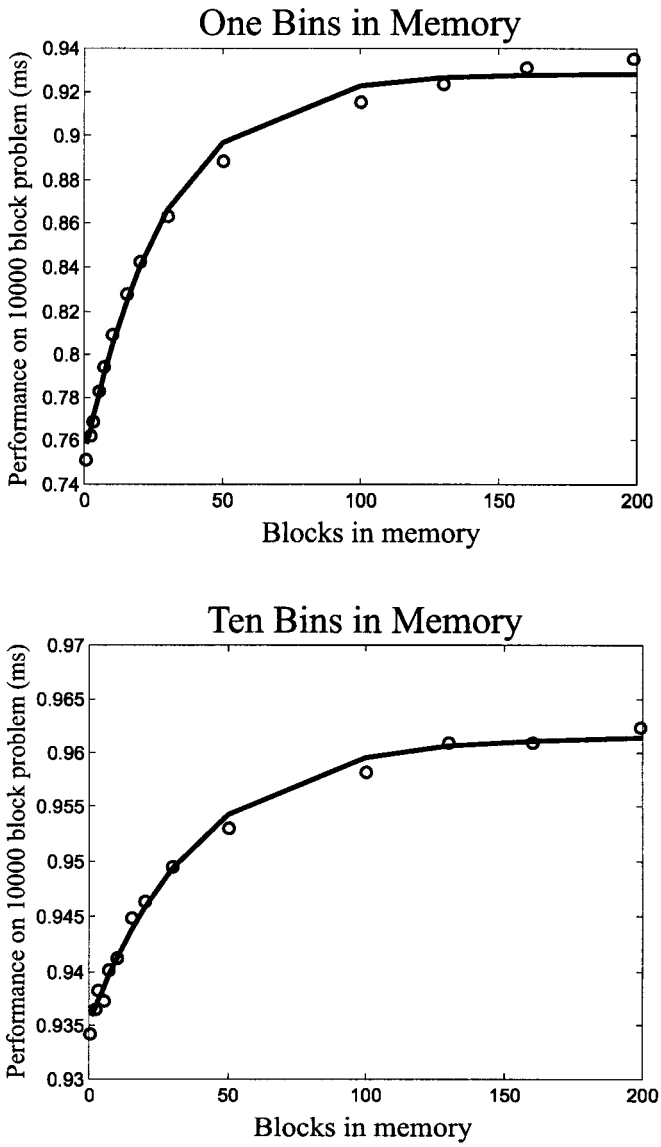

Figure 3.7: Performance model for performance vs. block memory of MF algorithm. The fits are exponential.

Here P_0 is gotten from the performance model, and

$$N_{MF} = N\frac{t}{t_0} \tag{3.10}$$

where t_0 is the value from the time model discussed above.

These models give a very good account of the behavior of Match Fit, as shown in Figure 3.8

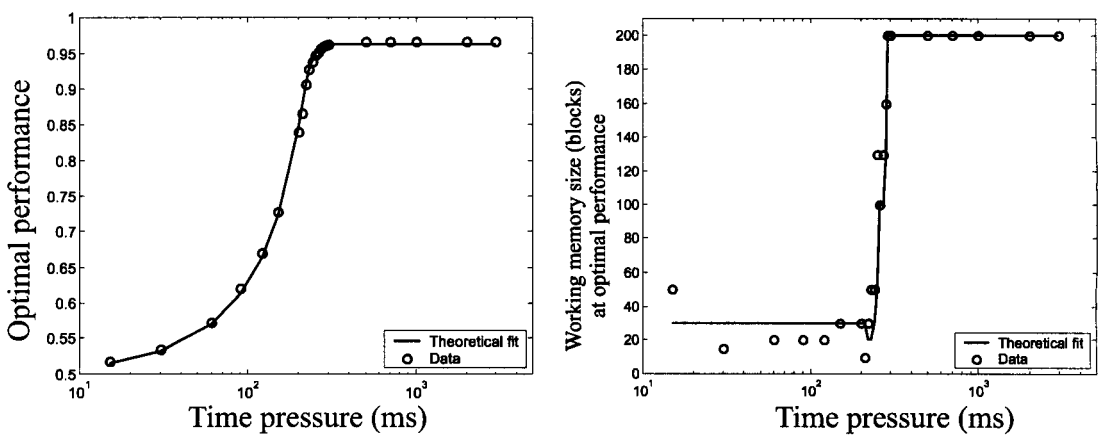

Figure 3.8: Results of the Match Fit model. The plots are the optimal performance for various memory configurations of MF vs. time pressure, and the number of blocks in working memory at that optimal point.

As can be seen, the model predicts very accurately the optimal performance behavior, as well as the threshold in optimal strategy as time pressure eases off.

3.4 Expected performance of Best Fit

The Best Fit algorithm has received a great deal of examination with regards to finding worst-case performance bounds and empirical performance. Since most realistic

problems do not resemble worst-case examples closely, another important measure of the quality of an algorithm is probabilistic: what is the expected performance ratio of the algorithm. Although the Next Fit algorithm has been successfully examined using this criterion [2], there has not been a successful analysis on the expected performance of BF. In this section, we will develop a method for estimating the performance of Best Fit on problems of blocks with sizes uniformly distributed in $(0, 1]$ and bin sizes of unity.

We begin our probabilistic analysis of the BF algorithm in an examination of the expected performance ratio assuming a uniform probability distribution on the sizes of the input blocks. By the *level* of a bin, we will refer to the sum of the sizes of all blocks that have been placed in that bin. A key element in our analysis will be the idea of a 'gap' in the level diagram. If we arrange the bins currently being used by the BF algorithm by ascending order in their level, we will have a curve which looks something like Figure 2. A gap in the level diagram, then, is the space between two successive levels in this sorted arrangement. For example, a gap of 0.1 would arise if, in their sorted order, there was a bin with level l and the next bin had level $l + 0.1$ (See Figure 3.9(c)).

Referring to the curves of Figure 3.9, we can think about the levels of the bins as falling into three regions. In the fullest region (above the knee of the curve), the sorted levels will be closest together after the algorithm has been running for a while. In the complementary region, located at levels less than one minus the level of the knee, there will be no bins: blocks which would fill the bin to this low level would

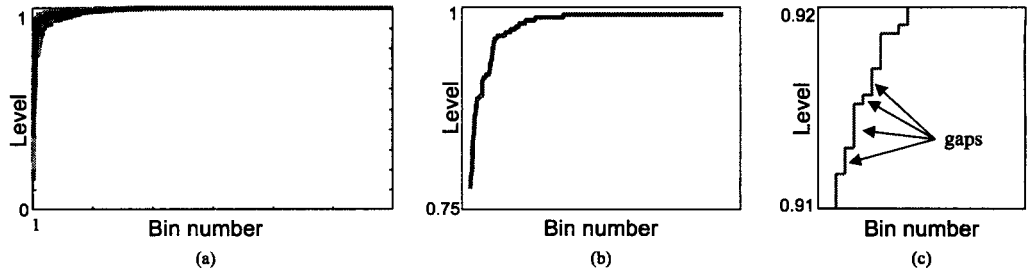

Figure 3.9: Bin level diagrams for ascending-fullness-sorted bin levels in a Best Fit algorithm solution of the bin packing problem. (a) The colors correspond to different trial runs (b) One specific trial run (c) Illustration of gaps in level diagram.

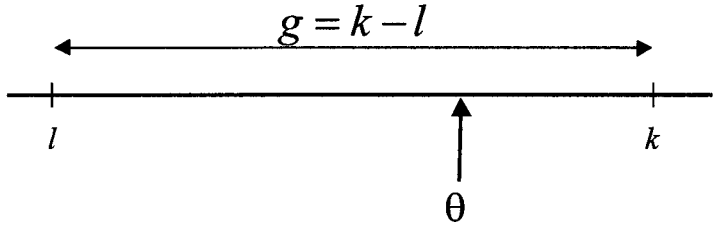

Figure 3.10: The gap size g around a position θ is given by finding the levels (k and l) of the bins with levels adjacent to θ (when the bins sorted by ascending order by level). The gap about θ is the difference in level between those two bins.

instead be placed by the BF algorithm in the many bins that are almost full. Between these two areas is a region of more active interest. Here an equilibrium will develop as the algorithm operates: if the bins are too dense (that is, the gaps are small), the algorithm will deplete the region of bins as it finds good matches and moves the bins higher in the sorted order (very near the top or nearly full). If the bins are too sparse in this region, the algorithm will find blocks for which a good match is unavailable, and will start to populate the area.

Referring to Figure 3.10, we would like to derive the form of the distribution on

the gaps occurring in the BF algorithm. Specifically, for some point θ which is in the "region of active interest," that is, greater than one half (only one bin can have level less than one half at any given time) but less than the threshold value, we'd like to know $f(g)$, the probability distribution function (PDF) on g, the size of the gap which surrounds θ.

To find this distribution, we approach from the problem probabilistically. That is, we can write that

$$f_+(g) = \int f(g, x, b) dx db \tag{3.11}$$

or, using the marginal distributions:

$$f_+(g) = \int f(g|x, b) f(x) q(b) dx db \tag{3.12}$$

where $f_+(g)$ is the PDF *after* some block has been placed, and $f(g)$ is the PDF *before* this block has been placed. x is the gap about θ before the block placement, and b is the size of the block. $q(b)$ is the PDF of the blocks, as set up by the problem statement (here we will consider a uniform distribution on the block size).

Since we know $q(b)$, and we will assume that $f_+(g) \rightarrow f(g)$, all we need is to write down an equation for $f(g|x, b)$. There are regions of the values of x and b which change the form of this distribution. For $x < g$, we have the situation as diagrammed in Figure 3.11. The gap size before block placement is smaller than that after, so the gap about θ must increase. In order for this to happen, the block must be placed by

BF into either the bin at level l or the one at level k. That is, it must fall either into the gap about θ, or into the gap above this one.

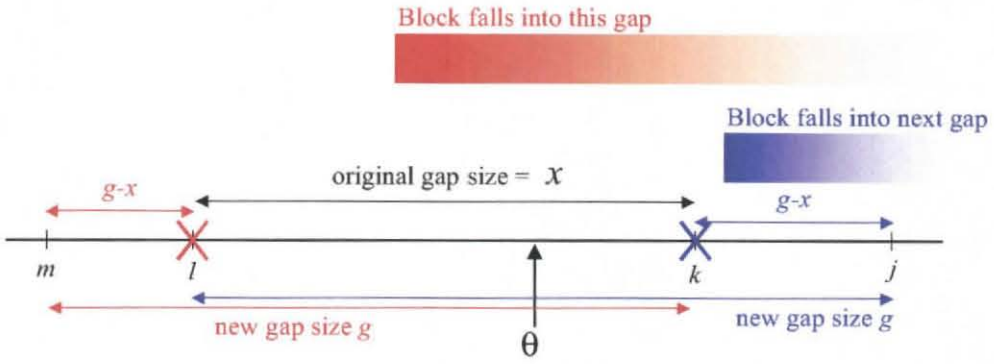

Figure 3.11: If the post-fitting gap size g is **more than** the prefitting gap size x, it must be because a block either fit into the gap about θ, or into the gap adjacent to that one.

For the PDF for $x < g$ (see Figure 3.11), then, we have $f(g|x,b) = f(g-x)$, which is the probability that the adjacent gap in one direction has size $g - x$ or that the one in the opposite direction does. These cases have different requirements on the value of b, however. In the first, $b \in (l,k)$; in the second, $b \in (k,j)$. These cases are summed to give

$$f_+(g)|_{x<g} = \int_0^g f(x)f(g-x)\int_l^j q(b)dbdx \tag{3.13}$$

Taking $q(b) = 1$ (the uniform distribution), and knowing that $j - l = g$ gives

$$f_+(g)|_{x<g} = \int_0^g gf(x)f(g-x)dx \tag{3.14}$$

Or, looking at it another way, when $x < g$, $f_+(g)$ is the product of the probability

of a block falling into either the gap in question or the next one, and the probability of that gap being the right size to sum to g.

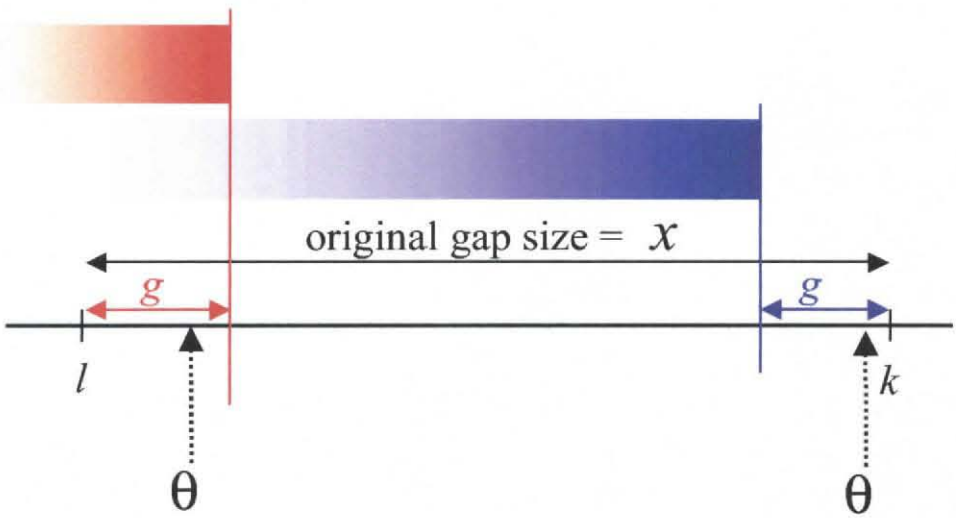

Figure 3.12: The post-fitting gap size g is **less than** the prefitting gap size x about θ. One way for this to happen is for a new block of just the right size to be packed in a new bin.

When $x > g$, the gap has shrunk, so a new bin must have been created which falls into the previous gap. There are two ways this can happen. First, a new block might be created that is larger than 0.5, and so fits in a new bin which has a level between l and k (Figure 3.12). Second, the block might fit in an existing bin and cause its new level to be between l and k (Figure 3.13). Considering the first case, there are two possibilities for how the gap about θ could change to size g. One is that the new bin might be at level $k - g$, and $\theta \in (k - g, g)$, the second is that the bin might have level $l + g$ with $\theta \in (l, l + g)$. For either case, if we assume that the probability distribution of the position of the θ within the gap is uniform, the probability that θ will satisfy the requirements is $\frac{g}{x}$. (Recall that $x > g$.) Since there

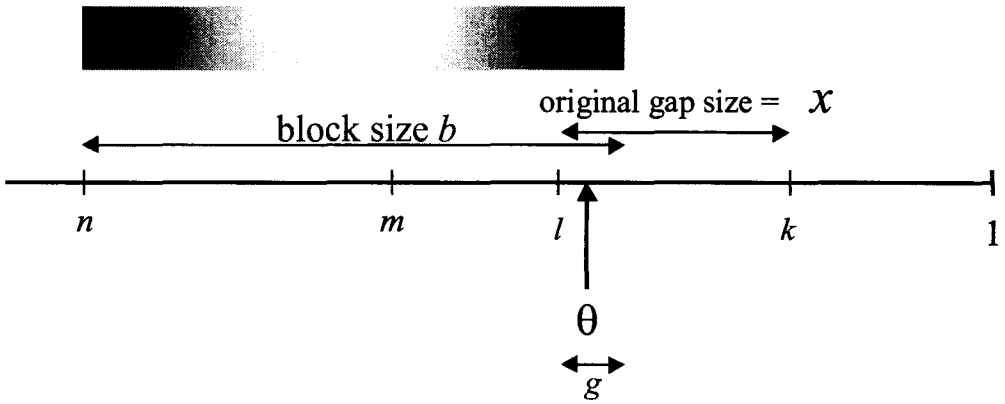

Figure 3.13: The post-fitting gap size g is **less than** the prefitting gap size x about θ. In the second case, this has happened because a block has fit into a bin of lower level.

are two possibilities, $f(g|x,b) = \frac{g}{x}\delta(b-(k-g)) + \frac{g}{x}\delta(b-(l+g))$. This means

$$f_+(g)|_{x>g,case1} = \int_g^\infty f(x)\frac{g}{x}\int_{1-k}^{1-l}\left[\delta(b-(k-g)) + \delta(b-(l+g))\right]q(b)dbdx \tag{3.15}$$

The limits of integration encompass the δ functions, so

$$f_+(g)|_{x>g,case1} = \int_g^\infty f(x)\frac{g}{x}\left[q(k-g) + q(l+g)\right]dx \tag{3.16}$$

Taking a uniform distribution again ($q(b) = 1$) gives us

$$f_+(g)|_{x>g,case1} = 2g\int_g^\infty \frac{f(x)}{x}dx \tag{3.17}$$

The second case is more difficult to analyze. So far in our analysis, $f_+(g)$ has been independent of θ. In the present case, this is no longer true. To see why, we need to

examine the way in which a new block will combine with an existing bin to impact the gap of interest.

In order for this to happen, the block must not be able to fit in the current gap, that is, $b > 1-l > 1-k > 0$. This implies that $b > x$. Also, since the block fits in the gap described by bins with levels n, m, we have that $n + b > l$. Combined with the fact that $l \geq m$, this indicates that $b > m - n$. We can now examine the probability of this case by integrating over suitable ranges for the block size and the size of the gap into which the block falls. But how large can the gap into which the block falls be? This depends on θ. Intuitively, when θ is small, it is easier to find combinations of block sizes and gap sizes to combine to impact the gap. When θ is large, there is less of a chance of doing so.

Mathematically, we can say that the probability of getting a block/bin combination at the right place in the gap about θ is equal to the product of the probability of a certain sized block coming along and the probability that this bin will land in a gap which will have the property of moving the bin on its low side to the right level. Since we're examining the marginal distribution $f(g|x, b)$, we know b (we'll integrate it out later), so we are faced with the problem of finding the probability that the gap described by b and x will actually have this property. This probability, since the BF algorithm guarantees that a block will be placed in a bin of a particular level (which we're calling n), is the probability that the gap which has bin level n as one of its endpoints will be of the right size to make the assumptions meaningful. If this gap is too small, then the block wouldn't have landed in the gap in the first place. If it

gets too big, it will bump against the gap about θ which is the target. This places bounds on the size of the gap, and since we know $f(g)$ the distribution on the size of the gap, we can just integrate to find the probability of the gap being that big.

The inequalities forming the bounding conditions on the gap at n are

$$b > 1 - m \tag{3.18}$$

$$m \leq l \tag{3.19}$$

The first inequality comes from the larger condition that $1 - n > b > 1 - m$, which is the condition that the block end up in the gap at n in the first place. Let $h = m - n$ be the size of the gap at n, then the first inequality above gives:

$b > 1 - m$

$b + n > 1 - m + n = 1 - (m - n) = 1 - h$

$h > 1 - b - n = 1 - (b + n) = 1 - (l + g) = 1 - l - g$

$h > 1 - l - g$

Where we have used $n + b = l + g$. Taking the second inequality yields:

$m \leq l$

$m \leq n + b - g$

$m - n \leq b - g$

$h \leq b - g$

So we have an inequality that $b - g > h > 1 - l - g$ and the probability we seek is $\int_{1-l-g}^{b-g} f(h)dh$. This will yield a function of b, l, and g. We will integrate over b, or, more precisely, over $b > x$, since we derived this constraint on b earlier. We need to

integrate out the dependence on l however. We have assumed that θ occurs randomly within its gap, which means that in the gap after block placement, θ can be anywhere in $(l, l+g)$. (Note that this is not strictly true, since if $\theta + g > 1$ then l must be limited. We maintain the assumption since we're interested in values of θ beneath the threshold.) This means that the PDF of l is uniform on $(\theta - g, \theta)$. Integrating out l, then, introduces the dependence on θ which we expected:

$$\int_{\theta-g}^{\theta} \frac{1}{g} \int_{1-l-g}^{b-g} f(h)dhdl \tag{3.20}$$

This is the equation for one possibility of producing a gap of size g. Following a similar approach for the case where $n + b = k - g$ yields this probability:

$$\int_{\theta}^{\theta+g} \frac{1}{g} \int_{1-k+g}^{b-g} f(h)dhdk \tag{3.21}$$

Combining terms and integrating gives

$$f_+(g)|_{x>g,case2} = \int_g^{\infty} f(x) \int_x^1 \left[\int_{\theta-g}^{\theta} \frac{1}{g} \int_{1-l-g}^{b-g} f(h)dhdl + \int_{\theta}^{\theta+g} \frac{1}{g} \int_{1-k+g}^{b-g} f(h)dhdk \right] dbdx \tag{3.22}$$

We now turn to the case where $x = g$. For this case, $f(g|x, b) = \delta(g - x)$ so long as the block size b is "correct," meaning that it doesn't combine with existing bin levels to change the size of the gap about θ. Where does this happen? If we imagine the possibilities for b, when b is small, $b < 1 - j$, it will not interfere with the gap about θ because it will go into a bin with a level higher than the one at the top of the

this gap. For $1-j<b<1-l$, the block will fit into either the bin at the bottom of the gap (of size l), or into the bin at the top of the gap. Either occurrence will cause the gap about θ to increase in size. The interval between $1-l$ and $1-k$ is simply x. The interval between $1-k$ and $1-j$ can be integrated out: it is a gap of unknown size, and so has the probabilistic value $\bar{g}$. Similarly, when b is larger than k, it will go into a new bin of its own with a level large enough not to interfere with the gap about θ. This will also be true for $\frac{1}{2}<b<l$. For $l<b<k$, however, the new bin will come into the gap about θ and cause this gap size to decrease (this was the first case of $x>g$ above).

For $1-l<b<\frac{1}{2}$ the story is more complicated: this is the regime of the second case above of when $x>g$, and to assess whether the block combines with an existing bin to change the size of the gap about θ follows analysis of the same sort which we employed in that situation. From the inequalities

$b>1-m$

$m\leq l$

$l<b+n<k$

we find that $1-k<h<b$ are the limits on h, the size of the gap into which the block falls. This yields a probability of $\int_{\theta}^{\theta+g}\frac{1}{g}\int_{1-k}^{b}f(h)dhdl$ for this range of values for the block size. Putting these together, then, we have

$$f_{+}(g)|_{x=g}=\int_{0}^{1}\int f(x)f(g|x,b)\delta(g-x)dxdb=f(g)\int_{0}^{1}f(g|b)db \tag{3.23}$$

And

$$f(g|b) = \left\{ \begin{array}{ll} 1 & 0 < b < 1-j \\ 0 & 1-j < b < 1-l \\ \int_{\theta}^{\theta+g} \frac{1}{g} \int_{1-l-g}^{b} f(h)dhdl & 1-l < b < \frac{1}{2} \\ 1 & \frac{1}{2} < b < l \\ 0 & l < b < k \\ 1 & k < b < 1 \end{array} \right\} \tag{3.24}$$

So

$$f_+(g)|_{x=g} = f(g) \left\{ 1 - 2g - \bar{g} - \int_{\theta-g}^{\theta} \frac{1}{g} \int_{1-l}^{\frac{1}{2}} \int_{1-l-g}^{b} f(h)dhdbdl \right\} \tag{3.25}$$

Putting these together gives this result

$$\begin{aligned} f_+(g)|_{x=g} = f(g) \left\{ 1 - 2g - \bar{g} - \int_{\theta-g}^{\theta} \frac{1}{g} \int_{1-l}^{\frac{1}{2}} \int_{1-l-g}^{b} f(h)dhdbdl \right\} + \\ f_+(g) = \int_0^g gf(x)f(g-x)dx + \\ 2g \int_g^{\infty} \frac{f(x)}{x} dx + \\ \int_g^{\infty} f(x) \int_x^1 \left[\int_{\theta-g}^{\theta} \frac{1}{g} \int_{1-l-g}^{b-g} f(h)dhdl + \int_{\theta}^{\theta+g} \frac{1}{g} \int_{1-k+g}^{b-g} f(h)dhdk \right] dbdx \end{aligned} \tag{3.26}$$

If we assume that packing a single block will make negligible changes to $f(g)$, we have an integral expression for $f(g)$ which depends only on the distribution on the sizes of the input blocks (here, the uniform distribution has been folded into the equation).

Once we have the distribution on the gaps in the region below threshold, we can use that to find the contribution of blocks falling into this region make to the PDF of the levels of bins around threshold. If we denote this PDF as $h(l)$, then we know that a block must land in a gap larger than $1 - l$ in order to end up with a bin of level l. This means that

$$h(l) = \int_{1-l}^{\infty} \frac{1}{g} g f(g) dg = \int_{1-l}^{\infty} f(g) dg \tag{3.27}$$

because $gf(g)$ is the probability of the block landing in a gap of size g, and the block size will be distributed uniformly within this gap distance.

Since the contribution of blocks of sizes larger than the threshold, or smaller than one minus the threshold, will be uniform, it will be this contribution to the PDF from blocks coming from below threshold which will determine the shape of the level diagram.

The gaps in the level diagram occurring as the BF algorithm operates will have approximately an exponential distribution, as shown in the example case of Figure 3.14. If we consider the function $N(x)$ as the number of gaps of size x, then ΔN will be proportional to the gap size x itself, since for a uniform distribution, the chance of finding a new block which will be inserted into that gap (and thus removing it) is just proportional to x. Since the distribution is approximated by $\frac{N(x)}{N_{total}}$, it too will have an exponential distribution (N_{total} is the total number of gaps in the approximation).

For larger gaps, the deviation from the predicted curve is larger than for the smaller gaps. This is for two reasons. First of all, there are not as many gaps during

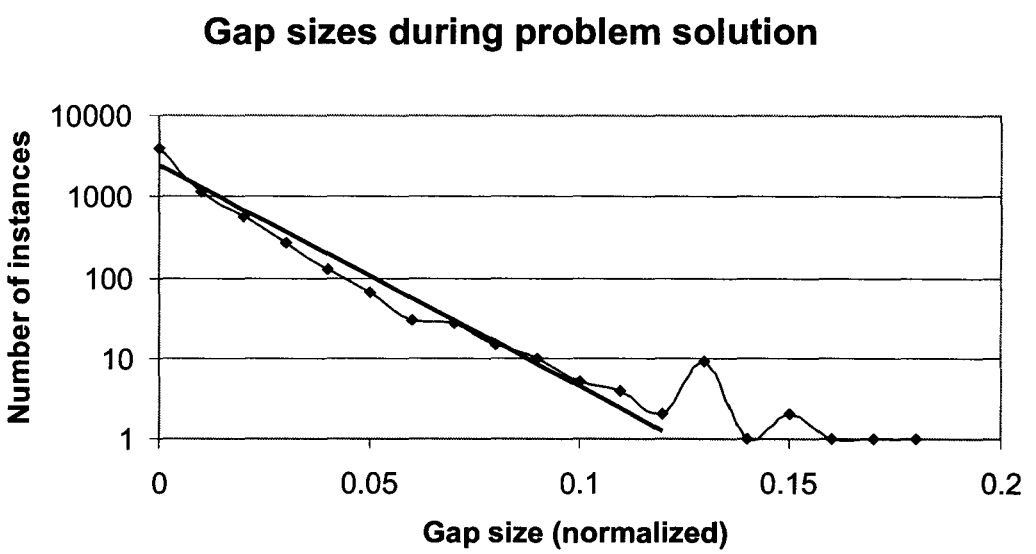

Figure 3.14: Distribution of the gap sizes during the running of the Best Fit algorithm. The fit is with an exponential.

the statistical run of a single test (which our example records) and so deviations will be larger. Second, the assumption we made — that when a gap is removed by a block it will disappear without affecting the gaps in this region of interest — is strictly true only when the gap itself is small, and so a block which is placed into the bin by the BF algorithm fills it very full, and places it above the knee of the curve and out of our region of interest. For gaps of sizes larger than this, there is a probability related to its excess size that it may remain (after having a block added) in the region of interest, and so our neglect of this factor is more important.

The distribution of gaps in the region of interest, then, is given by

$$f(x) = \frac{1}{\alpha} e^{-\alpha x} \tag{3.28}$$

where α is the exponential decay constant.

Bibliography

[1] Johnson, D. S., A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM Journal of Computing* **3**, pp. 299-326, 1974.

[2] Ong, H. L., M. J. Magazine, and T. S. Wee. Probabilistic Analysis of Bin Packing Heuristics. *Operations Research* **32**:5, pp. 983-998, 1984.

[3] Mao, W. Tight Worst-case Performance Bounds for Next-k-Fit Bin Packing. *SIAM Journal on Computing* **22**:1, pp. 46-56, 1993.

[4] Garey, M. R. and D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, CA, 1979.

[5] Johnson, D. S. Near-Optimal Bin Packing Algorithms. Ph. D. Thesis, Technical Report Mac TR-109, Massachusetts Institute of Technology, Cambridge, MA, 1973.

[6] Albers, S., and M. Mitzenmacher. Average-Case Analyses of First Fit and Random Fit Bin Packing, *Random Structures and Algorithms* **16**:3, pp. 240-259, 2000.

[7] Zhang, G. C., X. Q. Cai, and C. K. Wong. Linear Time-Approximation Algorithms for Bin Packing. *Operations Research Letters* **26**:5, pp. 217-222, 2000.

[8] SimchiLevi, D. New Worst-Case Results for the Bin-Packing Problem. *Naval Research Logistics* **41**:4, pp. 579-585, 1994.

Chapter 4 The Desert Survival Game

4.1 Introduction

Computer games are an attractive testbed for cognitive modeling (e.g., [9]). Agre and Chapmen [10] point out some of the benefits in using a computer game (Pengo, in their case) as a testbed for such modeling: the environment is complex but specified by the experimenter, there are real-time demands placed on any player of the game, the play is uncertain, meaning that predicting the actions of other players, or artifacts, in the game is nondeterministic. Especially when playing against an unknown algorithm, the actions of the opposing player are unpredictable. While computer games are artificial, these qualities which they abstract from the real world are a large part of what makes them interesting to humans, and makes them valuable as testbeds for exploring the behavior of cognitive models.

Our interest in using a computer game parallels that of Chapman and Agre. For our purposes, we would like to test out some of the ideas entailed by the model of attention and awareness proposed by Crick and Koch [2]. Here we are interested in the lower level aspects of reduced representations: using the idea of throwing away large parts of the environment or collapsing them in importance, in order to enable an otherwise expensive strategy to operate faster on complex, uncertain data in real time.

4.2 Description of the Desert Survival game

The testbed for this experiment is an artificial "Camel Game"—Desert Survival. This game is based on that described by Barbastathis [1] with certain simplifications.

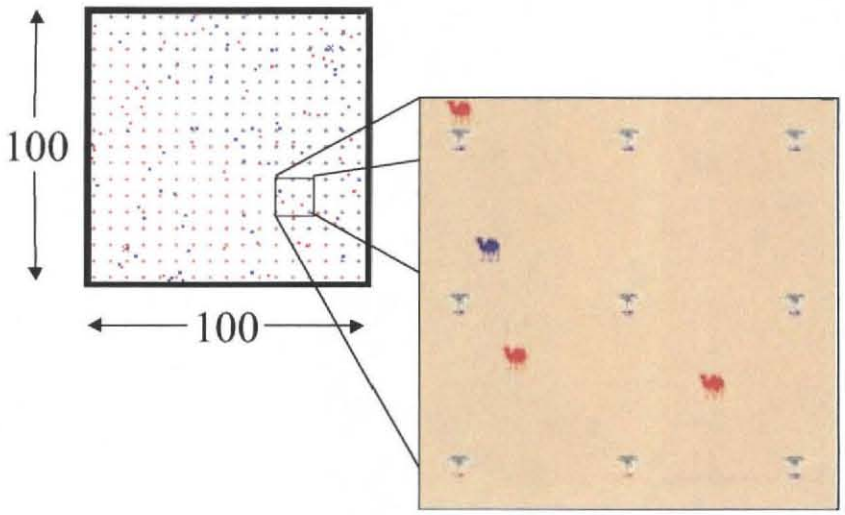

Figure 4.1: Schematic of the Desert Survival game. The full playing grid is constructed with territory markers—oases—arranged according to some method (here the arrangement is in a regular grid). There are two players, red and blue, and each starts with some movable pieces (camels) which they use to attempt to capture their opponents territory.

In essence, the Camel Game is a territorial one: two players vie against each other for control of resource production in a simulated environment. The environment, in this case, is a desert of a certain size (100 by 100 grid squares by default), and resource production occurs at specific territory markers of the desert oases. As tools in the capture of desert territory, both players have a certain number of camels (50) under their command. To capture an oasis if it is empty, the player must move two camels simultaneously onto the grid square occupied by the oasis. If the opposing player

has one camel on that grid square, the player must have three camels simultaneously present. In general, to capture an oasis requires two more camels than the opponent has in an oasis square.

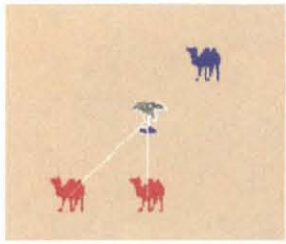

Figure 4.2: Red player captures a blue oasis. To capture an oasis, the capturing player must have two camels at the oasis more than the player which currently owns the oasis. If the opposing player has no camels there, then two will capture it. If the opposing player has one camel, it takes three to capture the territory.

Game play proceeds in turns. Each turn, both players have a chance to move their camels one square in any direction (laterally or diagonally). A player may also provide a camel with a "goal." In the absence of a direct command by a player in a given turn, each camel piece will consult its goal and move in a best-straight-line fashion towards it.

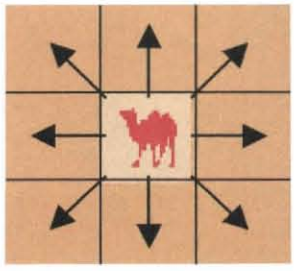

Figure 4.3: Camel navigation. The desert is played on a grid of squares. On each turn, camels may move one square horizontally, vertically, or diagonally.

In the desert, the camels require water to survive. Each turn in which a camel moves, it uses up a certain amount of water. If the camel doesn't move, it uses no water. Being camels, there is a certain maximum amount of water they can carry for movement between oases (enough for 200 turns). To give a camel more water, a player must move it to an oasis which they own. There, when not moving, the camel will drink enough water per turn to allow it to move for 15 turns (until it reaches maximum water capacity). If a camel runs out of water between oases, it will die and no longer be available to the player.

Figure 4.4: An oasis. The territory in the game is marked by oases. An oasis is always owned by one player or another. The moving pieces (camels) replenish their water at friendly oases.

The game lasts a certain number of turns (by default 840), after which the player who controls the most oases is declared the winner. A turn consists of offering the opportunity to each player to give orders to their playing pieces (the camels). These orders consist of instructing the camels to head towards one spot in the desert (which is typically occupied by an oasis). Ordinarily, the turns are controlled by a clock to be of a fixed duration, that is, the players are allowed a fixed time within which to give their orders, after which the camels are moved by the game engine whether or not the player has finished with their turn. This time pressure can be adjusted and even turned off entirely, to allow turns of arbitrary length, in which each player has

as much time as desired to give their camels commands.

4.3 Explanation of the rules

The Camel Game, ultimately, is a several-times-generalization of a competitive traveling salesman problem (e.g., [3]). The traveling salesman problem is to determine the fastest way to route a traveler (the salesman) between cities separated by varying travel times. One generalization, then, would be to route several travelers among the cities. Another variant would be to specify that at least two travelers be present in each city at once. A competitive version would add in the "city capture" effect, and then playing with traveling distance specifications can be coerced to mimic the water consumption of camels. Ultimately, though, this means that an optimal strategy for play involves moving one's camels as efficiently as possible between oases that are not yet controlled, under the constraints of water resources and with an eye to the strategy of the opponent. The strategic element of the game comes from the NP-complete nature of the problem. In computer science, problems are segmented by difficulty into several main classes. "P" complexity problems are those with an exact solution which is calculable in polynomial time. This means that as the number of elements in the problem grows, the amount of time it takes a certain computer to solve the problem grows polynomially (linearly, quadratically, etc.) For most real problems, the polynomial is of a small power (such as one, two, three, eight, etc.). Another way of looking at it is that this polynomial growth means that a computer can "keep up" with increasingly large problems by increasing either the amount of time spent solv-

ing the problem, or the amount of resources dedicated to it, in a polynomial fashion. In practice, this allows very large "P" problems to be solved in short time frames by relatively simple computers. For example, sorting lists with n elements by some order measure is a "P" problem with complexity $n \log n$ (algorithms with complexity $n \log \log n$ are also known [4]).

"NP" problems, on the other hand, are not so amenable to such efficient solution. Problems in the NP class have the property that as the size of the problem increases, the amount of computational resources required to find an exact solution scales faster than a polynomial of any power. Many problems of the NP type are known, isomorphic to each other. For example, the SAT problem (finding terms to satisfy a binary expression, such as $abc + bde + cbf + bef = 1$, where adjacency signifies the AND operation and + signifies OR), or the bin packing problem (finding arrangements of blocks in bins so as to use as few bins as possible without overfilling), and so forth.

If the camel game were much, much simpler, a clear solution might exist which by using few computational resources would perform optimally. Since the game is large and the problem is very difficult, such a simple solution is unknown. The number of fast-changing parameters (positions, water resources of the camels) is very large, and operates against a dynamic backdrop of many more slowly changing parameters (oasis ownership). Given that it is a generalization of a known NP problem, exact solutions are completely intractable using any known methods, and so some kind of heuristic algorithm must be deployed.

The rules of the Camel Game make the heuristics more interesting. The necessity

of using two more camels than an opponent has in an oasis to capture it implies a pressure to bunch up camels. Since the size of the environment is large, bunching camels up too much is counterproductive: the opponent can let a large "herd" intrude at will, while splitting its own camels up into smaller "squads" and thus capturing more territory. In general, this rule gives the edge to defense, while the size of the playing field makes a purely defensive strategy unattractive.

Similarly, the ability of the camels to carry fairly large amounts of water gives a lot of "path dependence" to the game, where an accounting of the current state of one's own and the opponent's pieces is important. A camel which runs out of water is lost, and so the future strategic value of the camels themselves can be assessed against their present ability to be moved and capture territory.

The ability for camels to be moderately "self-guided" presents the player with interesting strategic options. On the one hand, giving a camel a distant goal can save a lot of work in giving intermediate commands to the camel. On the other hand, the farther the journey (and so the more work saved), the higher fraction of its stored water the camel will use. The "work" referred to is the amount of computational resources it takes to control a player's camels on each move. These resources are limited—the algorithm only has a fixed time to give its orders to all the camels it is controlling. Thus, it is an attractive option to use the self-guidance of the camels to avoid giving them orders except when necessary. On the other hand, the camels themselves have no notion of strategy or planning, and so will not naturally behave in any kind of reliably optimal way. The scale of the environment is such that a camel

will frequently use up much (or all) of its water supply making a journey across even say one half of the full size of the desert if it cannot stop at intermediate locations to re-water. This means that there is a trade-off between the necessity to "micromanage" camels and their ability to carry out long commands unaided without running out of water.

The game rules used in this scenario differ in some important ways from the game described by Barbastathis [1]. First of all, in this game there is no money involved. In the original set of rules, the camels maintained money and water, and could trade money for water at enemy oases. Also, the oases maintained balances of money which could be captured, and the scoring incorporated this differential valuation on oases. In this version of the game, these rules are simplified: the camels may only drink at oases owned by their own team and never at enemy oases. There is no money at oases—they are all of equal value. The basic geometry of the game is unchanged: sheiks start out with the same number of camels, on a board that looks the same. Details in the navigational algorithm of the individual camels are consistent. Another larger change is the removal of the incremental navigational ability of the camels. In the original game, the camels would not always move each turn-they would "hesitate" with some probability that became smaller as they had more experience moving. In the new version, this is removed; the camels always move every game cycle, except when they are within an oasis where they still have a probability of leaving that is less than one.

The biggest difference from the original game is the removal of the "computational

tokens" which were used to control how much computational time an algorithm could spend per game cycle. In the new version, this mechanism is replaced by a straightforward measure: real computer time used. Instead of spending tokens to compute saliency and give camels orders, the competing algorithms can do whatever they are programmed to do within the constraints of a game cycle, but the time spent per game cycle is fixed. Thus, the measure of how much computational resources an algorithm needs is not dependent on the choice of where to assign computational token usage, but in actual execution time.

4.4 Computational resources and strategies in the camel game

The Camel Game shares some characteristics with other complicated environments: it is a large, fast-changing system with an overwhelming number of "sensory" inputs (or parameters) and no obvious model with which to deal with all the data and make optimal command decisions. In this context, there can be many classes of heuristic approaches (or strategies), and those can incorporate a large number of more finely grained differences within them.

The way we make computational resource usage a large factor in the Camel Game is by setting the game cycles, or how fast the turns run, to proceed independently of each player's actions. That is, the game is a "real-time" game where if a player does not move the pieces within a certain time limit, the game goes on anyway. This

creates another trade-off in player strategy in which we are very interested: what differences are there between good strategies when time pressure is light (that is, when much time is allowed per turn) and when it is intense (that is, when very little time is allowed per turn).

To examine these effects, we compare three strategies.

4.4.1 Unguided camels strategy

The first is the 'unguided' strategy: this player algorithm lets its camels wander more or less at random. The "more or less" is that the player sends its camels to oases, and lets them arrive before sending them somewhere else. All camels are controlled independently. In effect, this is the most random strategy which still incorporates the camels modest ability to "self-guide" themselves to targets.

4.4.2 Savvy player strategy

The second strategy is the 'savvy' player algorithm. This algorithm has three stages of operation. First, a saliency map [6] of the desert environment is calculated. This saliency is calculated around grid squares where there are oases (the rest of the desert has no resource value), as illustrated in Figure 4.5.

The number of oases and camels in the neighborhood (here a 3 x 3 grid) of the oasis for which saliency is being calculated are counted. Then the number of friendly and enemy oases and camels are assigned a weight. In the example, friendly camels and oases are weighted negatively and enemy camels and oases are weighted positively

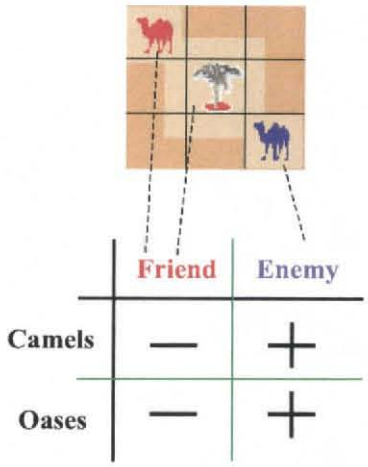

Figure 4.5: Saliency measure as used by 'savvy' algorithm. The area within a certain horizon of each oasis is evaluated for the presence of friendly and enemy camels and oases. These numbers are assigned saliency weights according to a map such as that shown (for the red player).

(this would make places where there are a lot more enemy oases and camels than friendly oases and camels more salient). Already, then, there are several parameters available for adjustment in the algorithm: the size of the neighborhood, and the weights given to enemy and friendly camels/oases.

The second stage is to average this saliency map over some characteristic size. This is done essentially by convolving the saliency map with a square of fixed dimensions. An example of the result is shown in Figure 4.6. The reddest areas are those with the highest averaged saliency values. Lighter red correspond to saliencies nearer zero, fading into light green for small negative saliencies and darker green for very negative saliencies. The area of the desert surrounding the highest value in the averaged saliency map is passed on to the next stage.

The third stage of the algorithm is to give commands to camels which are in this

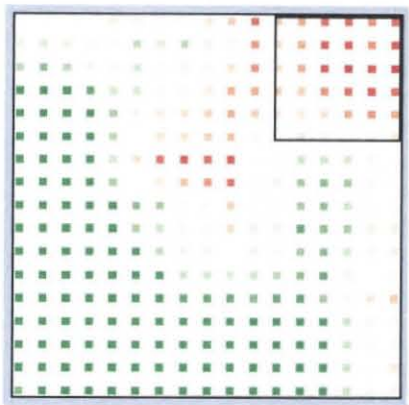

Figure 4.6: An averaged saliency map. The saliency assigned to each oasis desert location is calculated by analyzing how many friendly and enemy camels and oases there are nearby.

smaller area of the desert. Any camel outside this area is left to follow previous commands or to choose a destination at random, if it has reached its target. In addition, these commands will be to target an oasis also within the smaller area. In effect, then, the algorithm focuses in on a smaller playing environment and commands its camels there as if that were the entire scope of the game.

The commands given are chosen in the following way. For each camel, if α and β are parameters, and d_j is the distance from the camel to the j'th oasis in the small area, and s_j is the saliency of that oasis, then the probability that the camel will be given a command to head toward oasis j is

$$p_{\text{camel targets oasis j}} = \frac{e^{\alpha s_j}}{1 + \beta d_j} \tag{4.1}$$

That is, the greater the value of α, the more the camel tends to head for high-

saliency oases. The higher the relative value of β, the more it tends to look for nearer oases.

Of the three additional variable parameters in these stages, the most interesting to us is the size of the small area (the "awareness window") which the algorithm considers. The values of the command parameters α and β are held fixed at $\alpha = 0.1$ and $\beta = 1.0$. The behavior of the game is not very sensitive to the choice of these parameters.

4.4.3 "Deep Blue" Strategy

The third strategy we will examine we refer to as the "Deep Blue" algorithm, in reference to the IBM chess-playing supercomputer [11]. This algorithm also works in a series of stages. The first stage is to simulate internally the future state of the entire environment for some set number of turns (and assuming no additional commands given to the camels). That is, by knowing the targets of the camels on the board, the system predicts how they will move, whether any oases will be captured and by whom, whether any camels will need water or run out of water.

In the second stage, the algorithm assigns some of its camels to defense. What this means is that if the opponent is directing camels towards a particular oasis and will be taking it over within the look-ahead time frame, the algorithm will attempt to find enough camels to send to the endangered oasis to defend it from a takeover. If there are not enough friendly camels close enough to save the oasis, the algorithm doesn't defend it and lets it be captured.

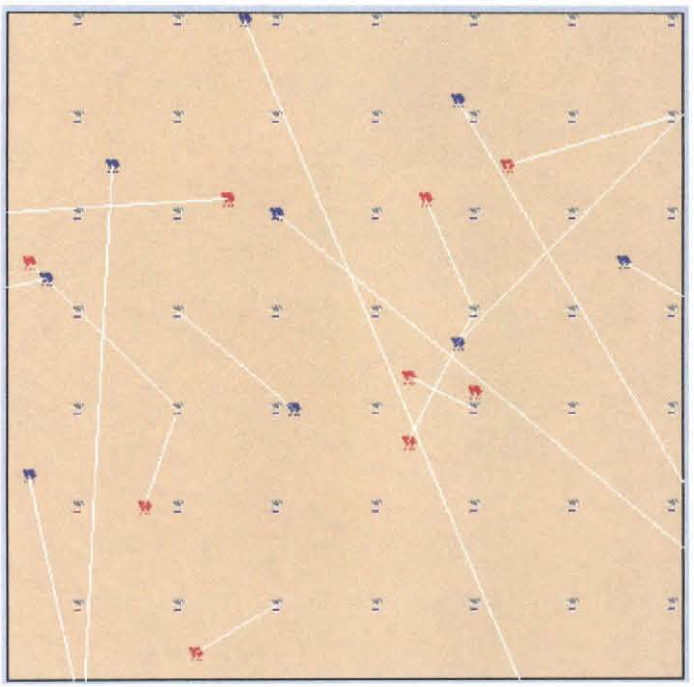

Figure 4.7: Illustration of Deep Blue algorithm. The Deep Blue player simulates in advance the motions of all the camels on the entire playing field and commands its camels accordingly.

In the third stage, the algorithm makes sure its own camels are not going to run out of water: if any are in danger, it sends them to the nearest friendly oasis to replenish their supply.

The fourth stage is the offensive stage. The algorithm takes all the healthy camels and assigns them to capture enemy oases. It does so exhaustively, calculating a very good candidate of assignments to capture the most oases in the minimum amount of time. The way it does this is by calculating the distances from every camel to every oasis it does not yet control. Then it goes through this matrix of distances and calculates what the first oasis it could capture would be and which camels are needed to do it. It assigns those camels and moves on to the next-earliest captured oasis. It continues until it runs out of either enemy oases or unallocated friendly camels.

This algorithm is not provably optimal (as indicated, the problem is complex enough to elude that possibility). It is, however, a very capable heuristic, with a few key customizable parameters, such as the depth in the game to look ahead and the water levels at which it will send camels to replenish their own water supplies instead of to take a defensive or offensive role. For example, since it takes fewer camels to defend an oasis than to capture a new one, it makes sense to allocate camels to defense first. Capturing as many oases as soon as possible makes sense as a goal: this will make the job of the opponent harder, and the game exhibits some positive feedback effects: having a lot of territory means there are many places to replenish camel water supplies, giving more resource production and thus an advantage in the game.

4.5 Characterizing the strategies

To characterize these strategies and interpret the results, we arrange two players utilizing different strategies (or the same strategy with differing parameters) to play the Camel Game against each other. During the game, the players capture territory from each other or defend their initial territory. The game is repeated many times, so that the statistics of the relative performances can be assessed. The performance of each player is the number of oases it holds at the end of the game. To report this, we will use the "performance margin", which is the percentage of oases an algorithm holds more than its opponent. Thus, in a game with 100 oases, a performance margin of 20% would mean that the winning player held 60 oases. This performance margin measure is calculated as follows, where **current_oases** is the number of oases owned by the player under test, and **enemy_oases** is the number of oases owned by the algorithm which that player is playing against:

$$PM = 100\frac{2(\text{current\_oases}) - \text{total\_oases}}{\text{total\_oases}} = 100\frac{\text{current\_oases} - \text{enemy\_oases}}{\text{total\_oases}} \quad (4.2)$$

When the current player (the test player) owns all the oases the performance margin is equal to 100. When the enemy player owns all the oases, the performance margin is -100. The game, if let run long enough, will end in one of these two states. In order to resolve finer detail of performance, then, we do not allow the game to run until this happens, but stop it after a fixed time.

4.5.1 Savvy player vs. unguided camels

A key expectation of our model is that there be an optimum in the "awareness" size associated with the 'savvy' player algorithm which uses the awareness analogue as a guiding mechanism. The reason for this is that if the "awareness" size is too small, then although the algorithm will run quickly, it will not have much to work with, and so not perform very well. At the other extreme, if the amount of information is too large, then the algorithm will start to slow down and miss turns, or give outdated instructions to its camels, and so the performance will again degrade. An optimum in the performance curve means that there is a benefit to the algorithm in having an awareness-like mechanism which throws away most information and preserves only the most important.

Figure 4.8 shows the performance margin of the savvy player when playing against the unguided algorithm under time pressure. When the awareness window is small, the savvy player performs on a par with the unguided player. This is because the algorithm has very little data to work with, and so although it runs quickly, it cannot give its camels meaningful instructions. The small awareness size means that most of its camels do not receive any directions, and so the outcome has no performance advantage for either player.

As the awareness size increases, the savvy player sees more and more of the playing field, and is able to command more of its camels and give them more meaningful instructions. Consequently, its performance margin increases. Why does the performance margin again drop off when the awareness size gets too large? The savvy

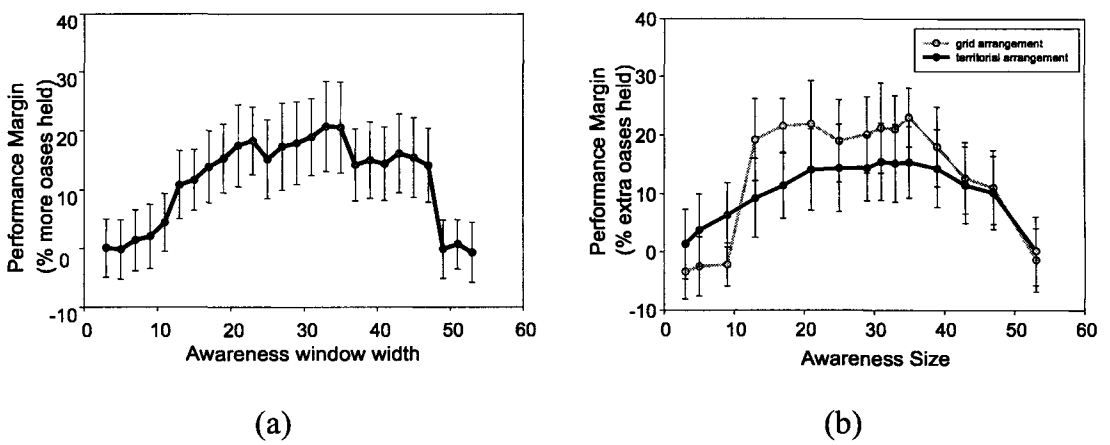

Figure 4.8: Performance of Savvy algorithm vs. unguided player. When resources are limited, there is an optimal in the size of the awareness window. In (a) the savvy player plays with a limitation in computational tokens. Tokens are required by it to calculate the awareness window for a given size, and more tokens for each command given to a camel. In (b) there are no computational tokens, but a guideline based on real computer time taken by the algorithm is used. The abrupt changes in the performance curve are seen to be due to the grid arrangement of the oases. When they are arranged more smoothly, the performance curve is more smooth (See Figure 4.11).

algorithm with a larger awareness size begins to spend more and more time calculating commands to give its camels. In doing so, it falls behind in the game and ends up giving instructions that are out of date. In the extreme case, it would be unable to give many instructions at all during the course of the game, so its camels would behave on a par with unguided camels.

The local maximum in the performance, then, reflects an optimum in the amount of "awareness," or in the size of the reduced representation for the algorithm. On the low end, the optimum is bounded by the inability to give the algorithm enough data to make good decisions. On the high end, the optimum is bounded by the oversupply of information: the time pressure placed on the algorithm makes keeping up with the game important. The demands of behaving in real time make too much information equally deleterious.

Figure 4.8(b) shows a comparison between a desert map with oases arranged on a grid and oases arranged randomly. The grid arrangement shows a dramatic increase in performance when the awareness size increases above ten, at which point more than one oasis is visible to the algorithm. Using randomly distributed oases produces a much smoother curve, since there is no sudden threshold on the low end. On the high end, where time limitations are more important, the two curves drop simultaneously, as expected.

4.5.2 Results for different parametric configurations

These results are robust to a large degree to various algorithmic details of the saliency measure and the values of the instructional parameters used to give directions to the camels. In Figure 4.9, we see that the choice of saliency measure must have an element of offensive play in order to win against a simple opponent using unguided camels.

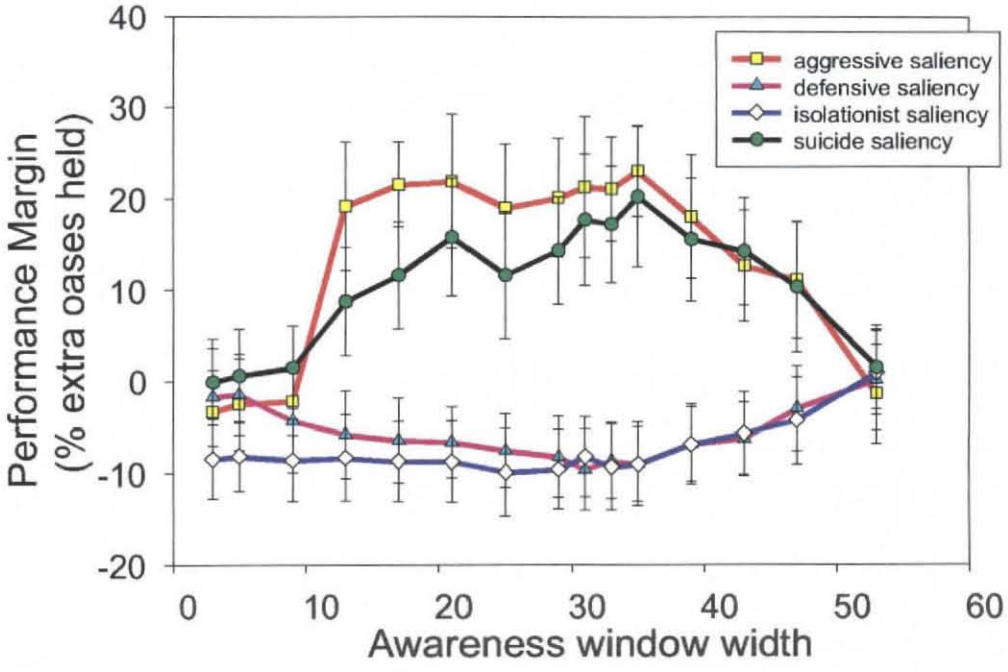

Figure 4.9: Performance of various saliency measures played against unguided camel opponent.

The names of the saliency measures correspond to the calculation as shown in Figure 4.10.

The weightings given to the various game pieces by these saliency measures reflect different approaches about what "looks interesting" to the algorithm. The "aggressive" measure wants to see spots on the map where it has camels and the enemy has

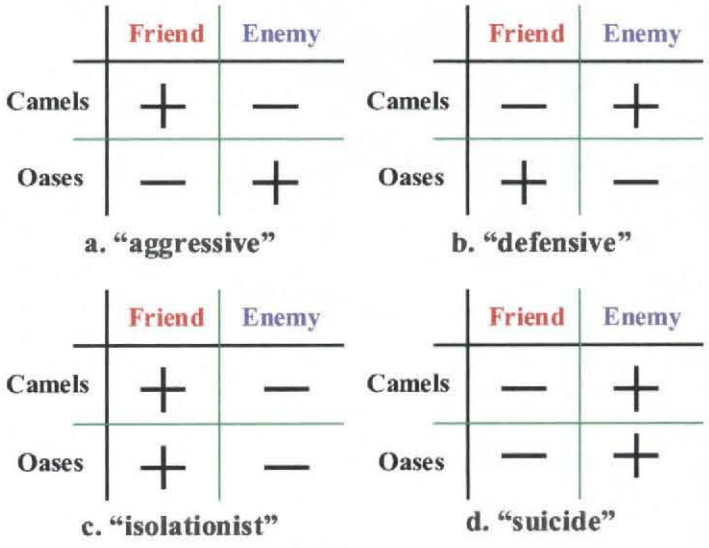

Figure 4.10: Diagram of saliency measures. The (a) and (d) saliency measures have an offensive component, in that they will tend to attract the savvy player's field of view to areas where the opponent has territory. The (b) and (c) measures, in contrast, will tend not to look at such areas, and therefore never go on the offensive.

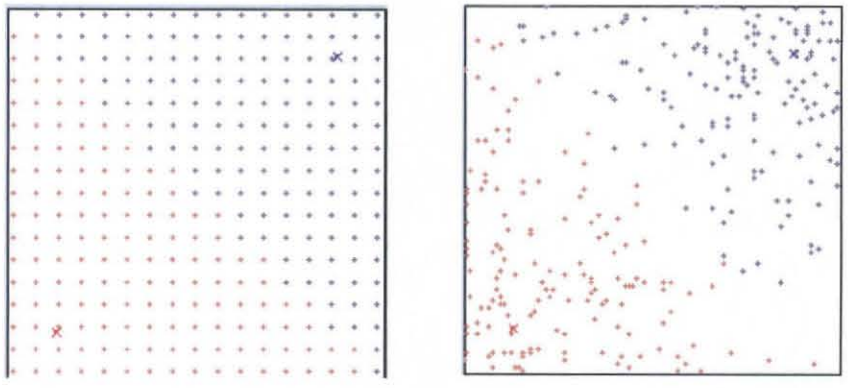

Figure 4.11: Examples of the grid (a) and territorial (b) oasis arrangements.

oases. The "defensive" measure, on the other hand, looks for spots where it has oases and the enemy has camels. The "isolationist" measure looks for friendly camels and oases, not seeking out the enemy at all, and the "suicide" measure looks for places where there are both lots of enemy oases and camels.

Referring to the performance curves, we see that the importance a saliency measure gives to oases is more important than that given to camels. The aggressive saliency approach (which is the typical measure used) does somewhat better than the suicidal measure. Both, however, exhibit the optimum in their performance curves indicating that they are giving somewhat reasonable (at least) instructions to their camels. The other two measures, on the other hand, perform at or even below random chance. The reason is that in giving their camels instructions, they tend to tell them not to capture new oases. The isolationist measure particularly doesn't even *look in* places where the enemy oases are. These two saliency measures are thus detrimental to the performance of the savvy algorithm.

This overall pattern is true even for significant changes in the underlying rules of the game. If the way oases are arranged is varied from a grid to a "territorial" arrangement, the overall performance remains the same. The grid arrangement (as shown in Figure 4.11) is that oases owned by each side are arranged in a regular grid. In the territorial arrangement, they are distributed more randomly in a clustered pattern around a "home base" and thinning out towards the initial territorial boundary.

The smoother performance curves in the territorial arrangement are a result of the closer spacing between oases over large parts of the desert. This means that finer differences in the number of oases captured are visible, as seen in Figure 4.12.

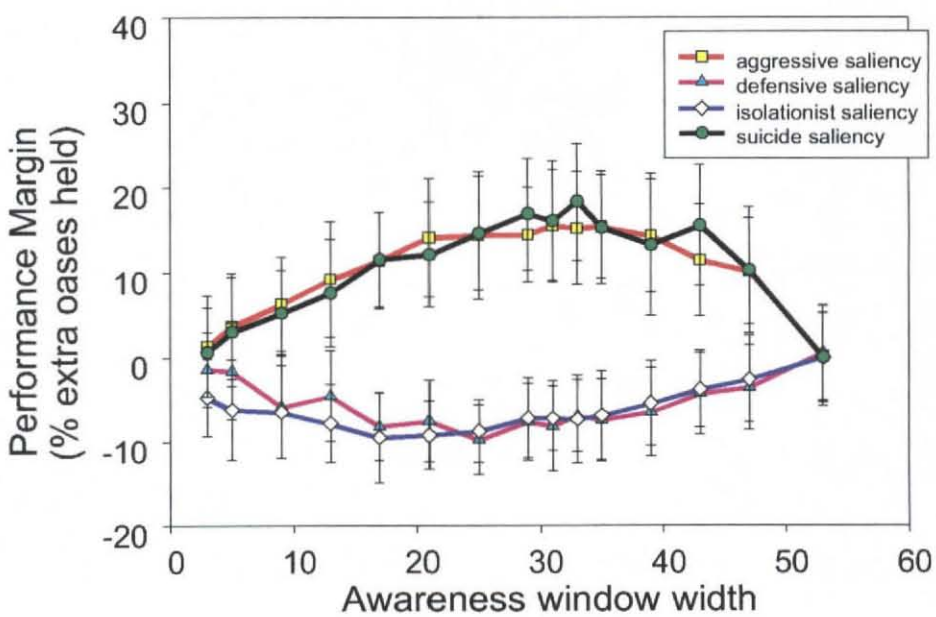

Figure 4.12: Savvy player performance against unguided camels with a *territorial* oasis arrangement.

These overall patterns hold qualitatively when players use the savvy algorithm

with different parameters against each other. As an example, in Figure 13 is the performance of a savvy algorithm using the aggressive saliency measure competing with the same algorithm using the defensive saliency measure.

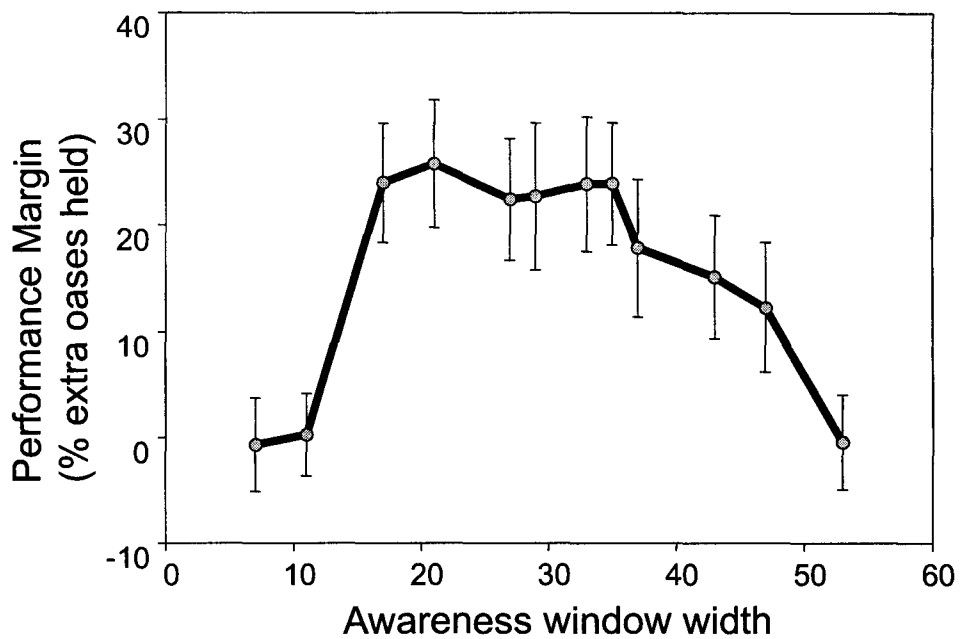

Figure 4.13: Performance of the savvy player using an "aggressive" saliency measure vs. a savvy player using a "defensive" saliency measure.

Figure 4.14 shows a comparison of games played under different sets of underlying game conditions.

As can be seen, the performance is both qualitatively and quantitatively the same under a variety of underlying game conditions: the basic result of the experiment—that both too little information and too much information lead to poorer performance—is consistent even under many permutations of the game rules. The red-outlined graph is a run for the rules that have been explained above. (b) shows a game where camels, instead of starting randomly distributed on the board, begin the game at a "home"

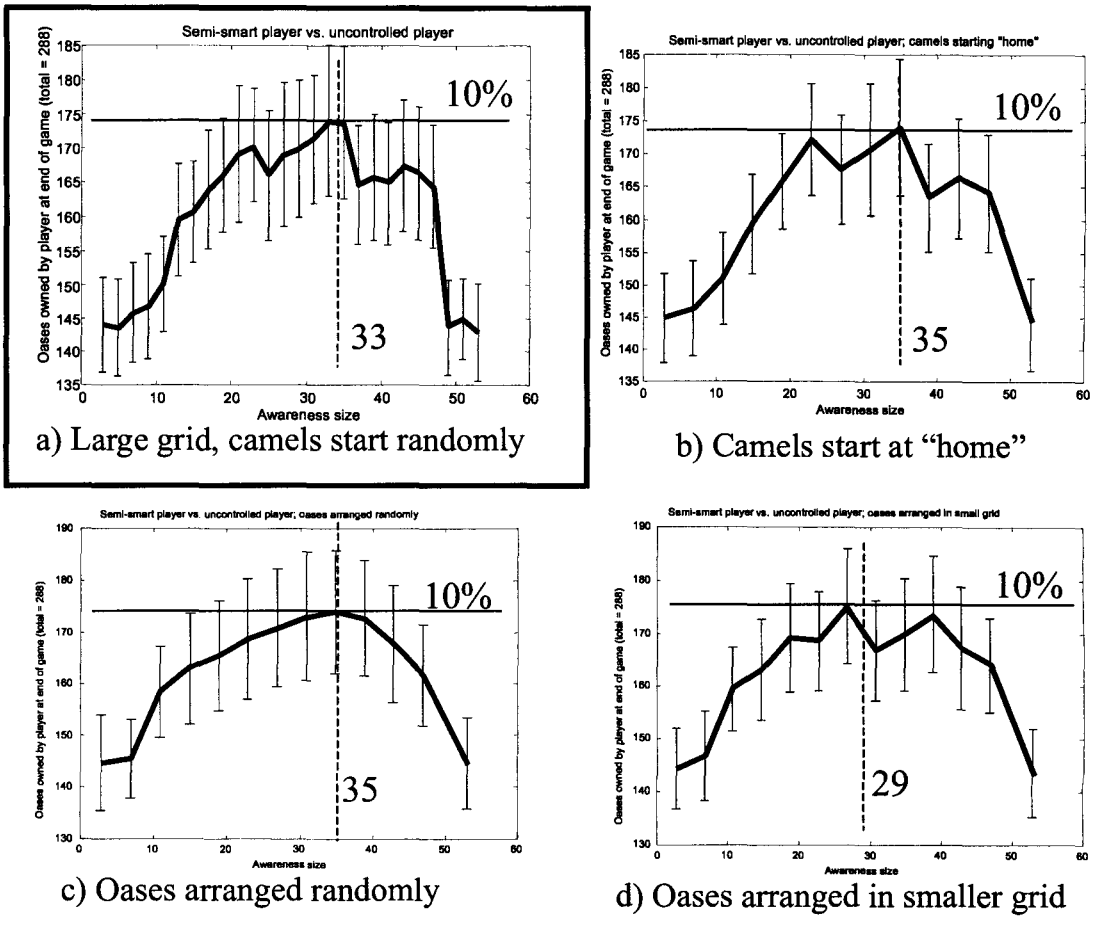

Figure 4.14: Desert Survival played under a range of game situations. The savvy player is playing the player with unguided camels. Details in the rules have a minimal effect on the overall performance characteristic of the algorithm.

location in each player's territory. (c) is an outcome where oases are arranged randomly instead of in a regular grid. (d) changes the details of the regular grid, making the spacing smaller. The significant qualitative difference is the much smoother curve when the oases are arranged randomly. This is because the regular grid has a sharper increase in available information when the window of awareness enlarges to cover more than just one oasis at a time. In a random arrangement this isn't the case, so

the increase in performance is more smooth. The small decrease in optimal attention size for the case of smaller oasis separation is not really statistically significant.

4.5.3 "Deep Blue" vs. unguided camels

The performance of the "Deep Blue" algorithm playing against the unguided strategy is quite overwhelming. Even under tight time constraints, the performance margin of the Deep Blue player goes very rapidly to 100%.

4.5.4 "Deep Blue" vs. savvy player

We are very interested in the performance margin of the savvy player playing the "Deep Blue" player. Figure 4.15 shows what happens under this condition.

As can be seen, under time constraints, the savvy player (here with an awareness size of 33 and using the "suicide" saliency rule) can still outperform the "Deep Blue" player. How does this happen? When the time allowed per turn is very low (close to zero), the two strategies perform at near chance. This is because neither has time to issue any commands to their camels, and so the camels of both pretty much wander randomly. When time pressure is eased off some, though, the savvy player gains the advantage. Using an awareness size of 33 is in the highest zone of its performance against unguided camels, meaning here it is in a good position to utilize its trade-off of neglecting some parts of the map in favor of speed of play. The time allowed per turn is given in units of milliseconds. This is for a particular computer used for testing. For another machine, this axis might scale one way or another depending on

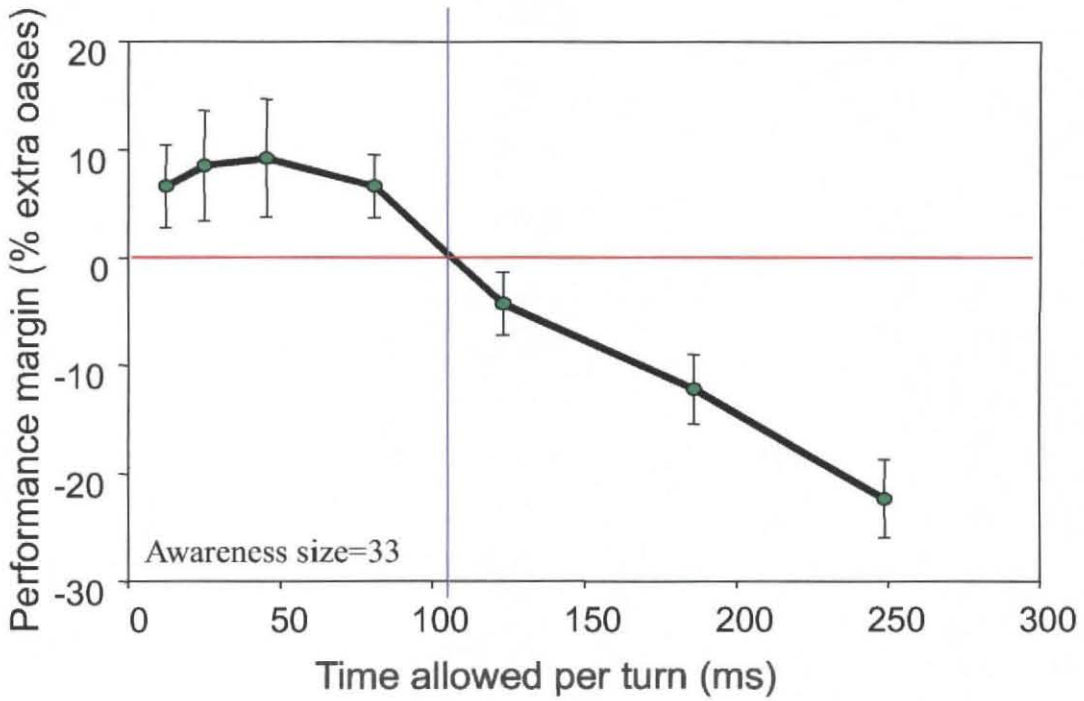

Figure 4.15: Savvy player plays the "Deep Blue" player under varying time constraints. When the time allowed is very low, the savvy player wins. As the time allowed increases, "Deep Blue" begins to dominate.

the relative performance, but the shape of the curve would remain consistent.

And that is how it can win against a much stronger opponent. When the time pressure is high, the savvy player can stay on top of the game. It's strategy is to ignore much of the board and concentrate on giving adequate commands to those camels in its "field of view." The "Deep Blue" player can give better commands, but it is a much more computationally demanding algorithm. The penalty, then, is twofold. First, it doesn't have the chance to give as many commands to its pieces. This means that more of its camels are wandering unguided for more time. Second, when it does give commands, they may be outdated, reflecting information that is already stale.

If its forward trace of the game to determine vulnerable oases does not happen much faster than the evolution of the game itself, it ends up giving commands, for example, to defend an oasis which has already long been captured by the savvy player. The computationally intensive attempt to give near optimal commands, then, backfires under time pressure. A much simpler algorithm, which just ignores 90% (and more, as we will see) of the playing area, can actually win when the timing of the game is fast enough.

As the time pressure is eased, however, the advantages of the savvy player decay. The strength of the "Deep Blue" algorithm becomes more important in terms of its not falling behind and its ability to give lots of commands to camels. At some point, the time agility of the savvy player and the expensive strength of the "Deep Blue" player are matched, and as time pressure is eased off even more, the Deep Blue player begins to win more and more decisively. When no time pressure at all is applied, the Deep Blue player often ends up with a performance margin of 100%.

This general behavior holds true even when the parameters are changed. Figure 4.16 shows curves similar to that above under different experimental conditions. The processor is slightly (33%) faster; the algorithm utilizes the "aggressive" rather than the "suicide" saliency measure, and the awareness dimension is varied.

For this set of experiments, the faster computer (as well as the aggressive saliency map) means a more decisive victory for the savvy player when time pressure is high (that is, low time allowed per turn). The same analysis applies, however, to the overall performance trend. The differences in exact performance margin between the different

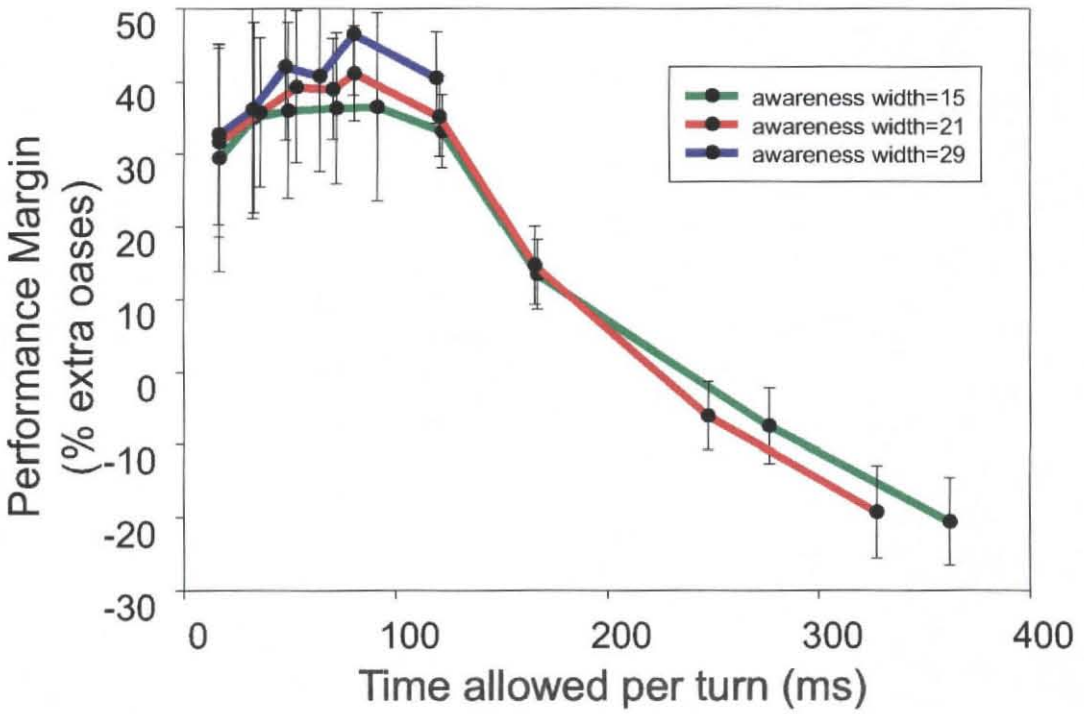

Figure 4.16: Performance of savvy player against Deep Blue player for varying awareness window sizes. The width indicated is the measure of the edge of the awareness window.

awareness dimension sizes are hard to attribute with any statistical significance to that parameter variation. If there is an effect, it is quite small, as might be expected by the overall smoothness of the performance curve when looking at performance as a function of awareness size when playing the unguided player.

4.5.5 "Deep Blue" vs. "Deep Blue"

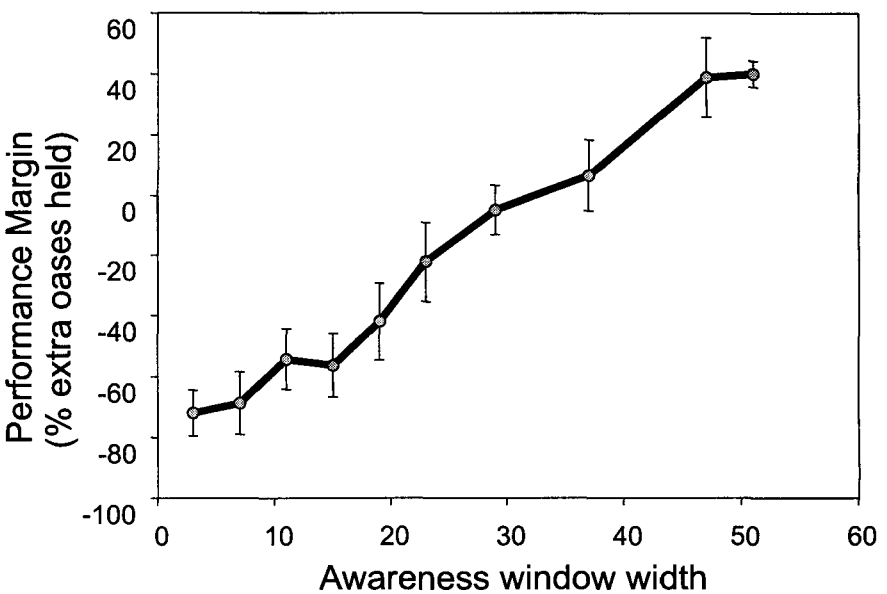

Figure 4.17: Deep Blue with an "awareness window" vs. Deep Blue playing on entire field. The player using an awareness window gains an advantage when the window is large enough. (50 ms per turn time constraints are present in this game. The advantage of the awareness window player stems from its magnifying an initial lead.

Figure 4.17 helps us to understand better the values of a reduced representation in terms of the more extensive algorithm alone. Here, a version of the "Deep Blue" algorithm is run on a smaller area of the desert, and competes with the full algorithm run on the entire playing environment. (There is moderate time pressure for this

experiment.)

With some time pressure, the algorithm using the awareness model loses when the awareness size is small. When the size is at about a third of the edge of the playing field, however, the awareness-model player starts to win. This means that by playing with only about 10% of the data, the "Deep Blue" player is able to outcompete a version of itself playing with 100% of the data. Here, especially, since the only difference between the players is the amount of data they use to play (each using the same algorithm), the conclusion is that under time constraints, the neglect of the vast majority of the problem can actually be beneficial.

4.6 Conclusions

An important criterion for the evaluation of biological computational strategies is the necessity to behave in real time. An apparently sub-optimal approach may in fact be the best, given the constraints imposed by having limited (expensive) computational resources and the necessity of reacting immediately. When assessing the options available to a biological (or artificial) system, in realistic scenarios the cost of doing nothing—of sitting and thinking—must be considered alongside the costs of making the wrong choice. When the cost of doing nothing is comparable to that of a wrong choice, it can even be better to act randomly, but fast, than to deliberate.

For the case of a computer game with fairly simple rules, but playing on a large-sized board, the addition of time constraints leads to an optimum in the amount of information an awareness-model-inspired player uses to make play decisions. Too

little, and the player is handicapped by insufficient data; too much, and enough time is required to process the data that the player falls behind and gives too few, and even outdated, orders to its pieces.

When playing a much more computationally imposing opponent, the awareness-model-inspired player can still win if the time constraints are heavy. As behaving in real time becomes less important (that is, as the time pressure on the players is eased), the awareness-model player's edge deteriorates, and the much more nearly optimal, but computationally expensive, algorithm takes the upper hand.

Bibliography

[1] Barbastathis G. Intelligent Holographic Databases. Ph.D. Thesis, California Institute of Technology, 1998.

[2] Crick F., and Koch C. Towards a Neurobiological Theory of Consciousness. *Seminars in Neuroscience* **2**, pp. 263-275, 1990.

[3] Garey M. R., and Johnson D. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, CA, 1979.

[4] Nilsson S. The Fastest Sorting Algorithm? *Dr. Dobbs Journal* **25**:4, pp. 38–45, 2000.

[5] Clearwater S. H., Huberman B. A., and Hogg T. Cooperative Solution of Constrained Satisfaction Problems. *Science* 254, pp. 1181–1183, 1991.

[6] Koch C., and Ullman S. Shifts in Selective Visual Attention: Towards the Underlying Neural Circuitry. *Human Neurobiology* 4, pp. 219–227, 1985.

[7] Braun J., and Julesz B. Withdrawing Attention at Little or No Cost: Detection and Discrimination Tasks. *Perception and Psychophysics* **60**:1, 1998.

[8] Itti L., Braun J., Lee D. K., and Koch C. Attentional Modulation of Human Pattern Discrimination Psychophysics Reproduced by a Quantitative Model. *Ad-*

vances in Neural Information Processing Systems, Kearns, M.S., Solla, S.A. and Cohn, D.A. (eds.) **10**, pp. 789-95, MIT Press, Cambridge, MA, 1999.

[9] McCarthy J. *Partial Formalizations and the Lemmings Game*, http://wwwformal.stanford.edu/jmc/lemmings/lemmings.html, 1994.

[10] Agre P., and Chapman D. PENGI: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence* (AAAI-87), pp. 268-272, Seattle, WA, 1987.

[11] Clark D. Deep thoughts on Deep Blue. *IEEE Expert-Intelligent Systems and Their Applications* **12**:4 p. 31, 1997. See also http://www.chess.ibm.com.

Chapter 5 An Attentional Learning Model and its Application to Control of an Articulated Arm

5.1 Introduction

There is evidence that attentional mechanisms in humans are important for memory formation (e.g., [19], [2]). Machine learning, then, is one area where we expect algorithms inspired by attentional functions to outperform conventional ones. There are several ways in which attention might facilitate learning. One is in the important area of generalization. In biology, the brain must segment the data to be learned away from the background. Remembering a phone number should not be linked to the color of the paper on which it is written, or whether a male or female voice spoke it. Such segmentation will also help in a second way: reducing the amount of data that must be memorized, thus improving learning speed. Picking the right information to be learned, and ignoring the rest is a job well suited for an informational bottleneck within the brain, and indeed, such a bottleneck appears to be necessary for the utilization of some kinds of memory.

We are interested in how a supervisor for a learning system employing this kind

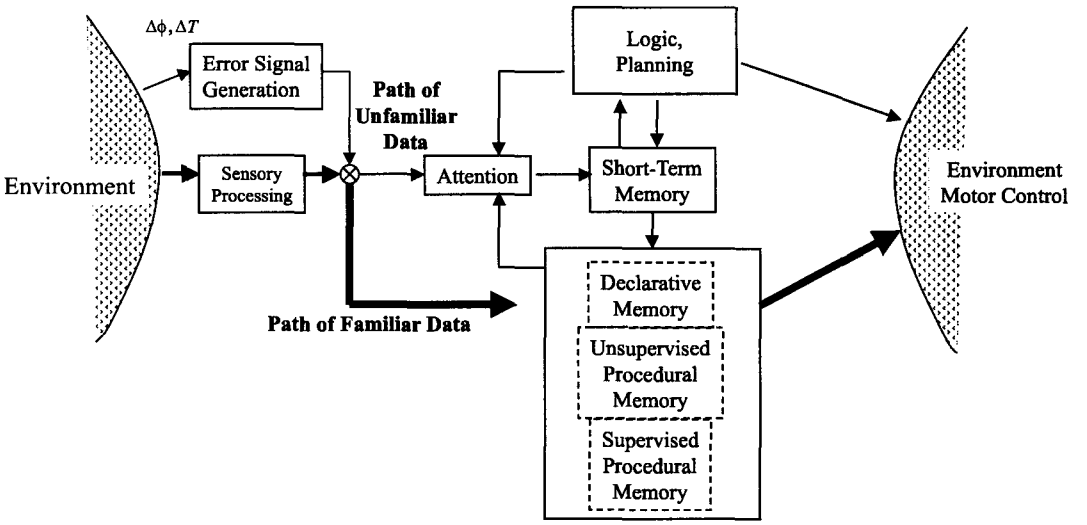

Figure 5.1: Attentional learning architecture. At first, the responses of the system are calculated by the logic/planning unit. After the system has learned how to control the arm sufficiently well, control is transferred gradually away from the logic/planning areas to memory. The error signal arouses the attention function, which lets the logic know that the memory needs more guidance in learning the current situation.

of informational bottleneck assists learning. The application used as a testbed is a demanding problem of control of a segmented arm. The function of attention is utilized for several different important functions in the problem solution. It is used to separate out the important elements from the unimportant in the field of obstacles for better generalization when learning gestures. It is also used to divide the task into various levels of expertise. As the most basic and generalizable elements are learned first, and the harder elements later, the most abstract, high-level factors are left in the domain of the logic subsystem.

Figure 5.1 shows the learning model used for these experiments. When the system is new to the problem, it uses its logic/planning functions to solve the control problem. This results in performance which is largely error-free, but very slow, since

complicated optimization problems must be solved. As the logic/planning subsystem solves these control problems, the reduced representation of the environment, which the logic determines to be important in solving the problem, is then presented to the memory for learning. There are several different kinds of memory involved which correspond to the kinds of memory humans employ to solve different kinds of problems. There is a procedural memory which learns from examples to reproduce forces on the arm segments to cause desired motions. There is an unsupervised memory which accumulates example of common "gestures." There is a declarative memory which stores sequences of these gestures as "directions" of how to go from one target to another.

As the system becomes more and more trained, the memory gradually takes over control of the arm from the logic unit. This happens independently at the various levels. Thus, once the networks which learn the arm force motion are trained, the logic does not compute those forces and torques any longer, and leaves them up to the memory. This frees it to spend more of its time training the other parts of memory. Or equivalently, it allows the system to start behaving more rapidly. As gestures and the higher-level directions between targets are learned, the logic/planning subsystem delegates more to the memory and spends more of its time deciding what to do in the solution of the abstract problem solution.

This can be interrupted, however, by the attentional mechanism. When there is an error—that is, if the arm seems to have gotten stuck in its motion or goes to the wrong place or hits an obstacle—the attention mechanism is triggered, which makes

the logic subsystem then dedicate time to solving the current situation of the arm in order to better train the memory subsystems.

5.2 Application environment—the articulated arm

At the most abstract level, the arm is used to solve various kinds of puzzles. We explored two different logical problems with the system. The first is a problem of ordering various targets in a target location. This is equivalent to the Tower of Hanoi problem [4], [21] (note that "Claus" is an anagram for Edouard Lucas, who invented the problem in 1883). Despite the long history of this problem, there is as yet no solution proven optimal, although algorithms exist which are widely suspected to be optimal (e.g., [8], [17]). The second is the bin packing problem, a common computer science problem which is non-polynomial in nature [7]. With the multi-level learning system we use, changing between problems like this can be done with no changes at all to the program at the memory, attentional, and short-term memory levels.

In the sorting (Tower of Hanoi) problem, we begin with an allotment of disks of various sizes on three pegs. The goal is to arrange the disks in order of increasing size on the target peg. The restrictions are that the disks must be moved one at a time and at no time may a larger disk rest on top of a smaller disk.

In our version of the problem (see Figure 5.3), the pegs are placed on a two-dimensional board, and the problem is imagined to be solved by an articulated arm which moves around the board and physically takes the top disk from each peg and moves it to another. Various obstacles are placed randomly on the board, through

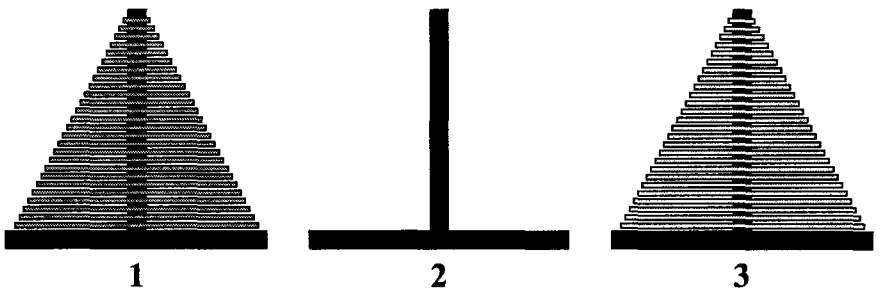

Figure 5.2: The Tower of Hanoi problem. The objective is to move the stack of disks from peg 1 to peg 3, only moving one disk at a time, and never placing a larger disk over a smaller. It is said that there is a Hindu temple where monks work a problem utilizing 64 golden disks, and when the last disk is placed, the world will end.

which the arm cannot pass. The only constraint on the pegs and obstacles is that they aren't placed such that the arm can't move between the pegs. That is, the use of the arm to physically sort the disks can't be impossible. The arm's segments can overlap each other, and for our purposes, it is assumed that the end effector, if placed over a peg, takes or releases the disk by itself. Our problem, then, is to move the end effector of the arm between the appropriate pegs in the correct order so as to solve the abstract problem.

The three distinct levels of responses to be learned are as described above. At the most basic, a "motor control network" must be trained to control the motion of the arm. That is, when the second segment of the arm needs to move clockwise, the proper forces have to be applied to do that. At the top level, the right sequence of movements between the pegs must be undertaken, or the problem will never be solved. In between, there are motions, or "gestures" which are generalizable between problems and which can be learned by a controller, such as "going around an obstacle

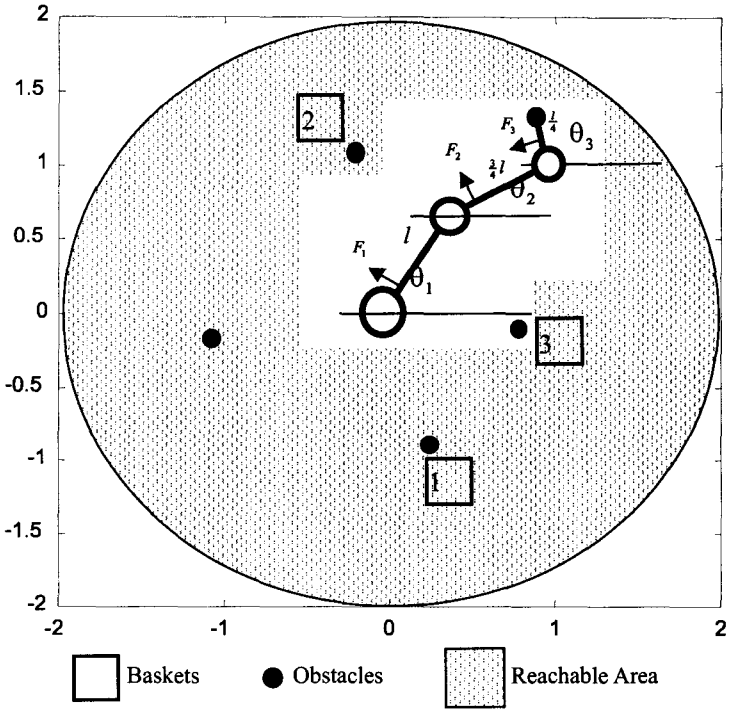

Figure 5.3: The Tower of Hanoi problem arranged in two dimensions for solution by an articulated arm. The arm must move between the marked baskets without colliding with the solid obstacles. (The outlines squares are the positions of the target baskets. The solid circles are obstacles.)

clockwise" or "extend the arm a bit to get around the obstacle next to the peg before going to the peg."

These three levels of generalization can be compared to things humans learn. While driving, for instance, the basic "gestures" of driving—turning left, changing to the right lane, exiting a freeway–are generalized between episodes of driving. The basic motor controls of steering are even more basic. At the top level, the directions one memorizes to navigate take as given the grammar of driving gestures and motor control.

5.3 Articulated arm control

The details of the articulated arm, the playing board, and targets is shown in Figure 5.3. The arm is composed of three segments. The first has length 1, the second length $\frac{3}{4}$ and the third $\frac{1}{4}$. The end effector is located at the end of the third segment, and therefore can be moved anywhere within a radius 2 of the origin, which is presumed to be fixed. The position of the arm is specified by three angles, $\theta_1, \theta_2, \theta_3$, which describe each segment's orientation with respect to the x axis.

The strategies that humans actually use to optimally control arm movements are quite complex [22], [20]. For our purposes, we give the arm segments a minimal dynamics involving a maximum torque and a momentum/friction decay characteristic. The equations of motion which describe the arms' dynamics are:

$$\dot{\theta}_i^{(t)} = \alpha_i \dot{\theta}_i^{(t-1)} + \frac{F_i^{(t-1)}}{I_i^{(t-1)}} \tag{5.1}$$

$$\theta_i^{(t)} = \theta_i^{(t-1)} + \dot{\theta}_i^{(t)} \tag{5.2}$$

Here the θ_i are the angles of the various joints in the arm. The $\dot{\theta}_i$ are their angular velocities. F_i is the torque on the joint and I_i is its moment of inertia. α is the constant which sets the magnitude of the momentum/friction decay, and is set to 0.96. The (t) and $(t-1)$ superscripts refer to the discretization of time. That is, $\theta_i^{(t)}$ is the i^{th} angle at time t, and $\theta_i^{(t-1)}$ is the i^{th} angle at time $t-1$.

In optimizing arm motion in the environment within these parameters, our logic subsystem employs a minimum torque-change heuristic similar to that described by Uno et al. [22] The three-segment arm has a complicated inverse kinematics, diagrammed in Figure 5.4, which requires an expensive optimization process to find the best trajectory to move from a present position to a target position. The minimum torque-change model breaks the degeneracy of the problem by selecting the configuration out of the possible solutions which requires the minimum change to achieve. This is the most costly step for the logic in deciding on how to move the arm. For a much simpler problem (a one-segment arm, for instance), there is no degeneracy, and so much less need to utilize memory to speed behavior. Similarly, if there is no pressure to behave quickly, the expensive computational process to find the optimal trajectory is not a burden. Learning only becomes imperative when the complexity of the situation becomes large and real-time behavior is still required.

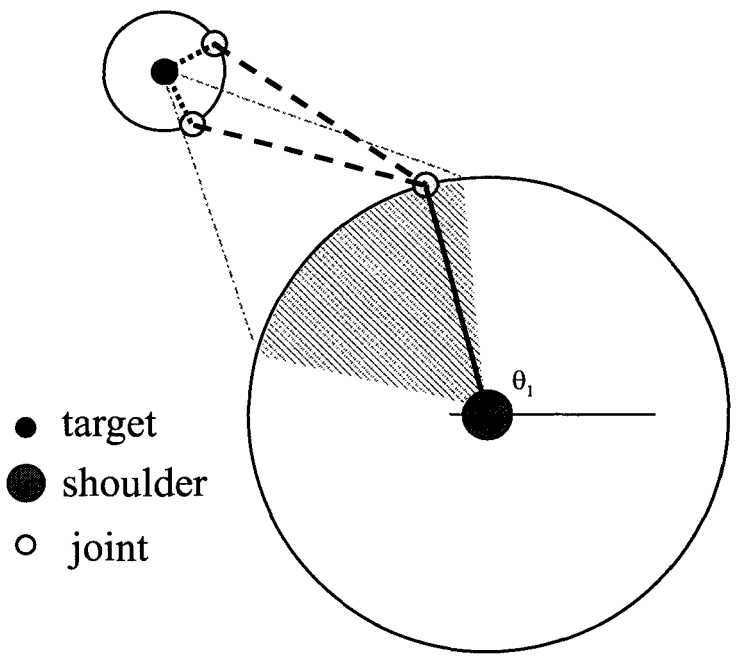

Figure 5.4: Inverse kinematics for a three-segment arm. A two-segment arm has a twofold degeneracy in deciding how to place the end effector on a target. The three-segment arm has a range of values for one segment, within which for every value, there is a twofold degeneracy.

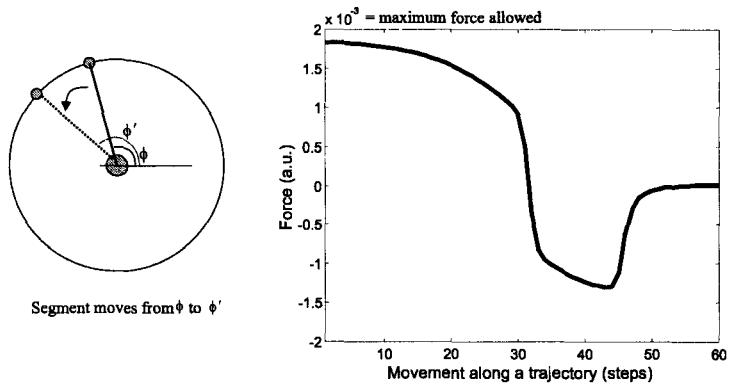

Figure 5.5: The motor learning problem for an arm segment as learned by the neural network.

The pegs (baskets) and obstacles are placed on the playing board in such a way that the arm can still move freely and not too close to each other such that the arm can't find a way to avoid the obstacles and reach a basket. Each basket has an obstacle between it and the origin, which must be reached around to enter the basket.

5.4 Learning arm kinematics

As the puzzle is solved, the system is simultaneously learning at all the various levels. The arm kinematics are learned by a network responding to forces as shown in Figure 5.5. The force curve shown is one generated to move the arm as quickly as possible (given a maximum torque) from one angle to another. In our system, the force models as originally computed by the logic/planning subsystem are learned by a three-layer neural network with four units in the hidden layer. The units use a hyperbolic tangent activation function and are fully interconnected. There are two input parameters: the current distance from the computed goal angle and the angular velocity of the arm

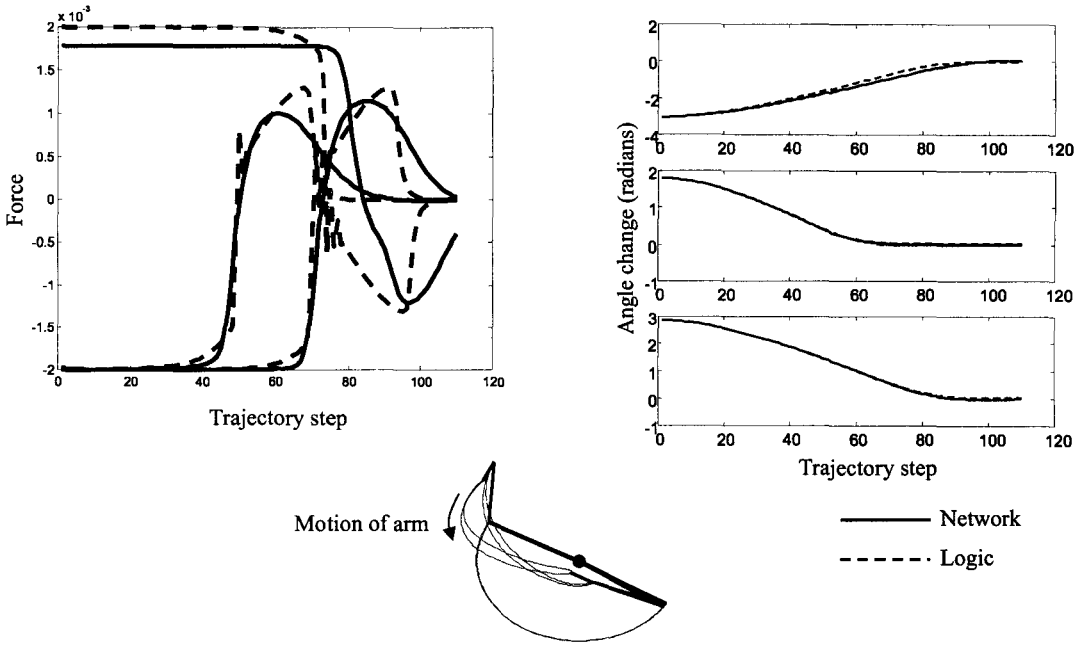

Figure 5.6: The motor network learning the motion problem. The forces and angles of the arm under the control of the logic are shown as dashed lines. The solid curves are the forces and angles of the arm under the control of the motion network. The plots on the left are the forces throughout a trajectory. The plots on the right are the angular change by segment on this trajectory. The diagram below is the illustrated history of how the arm moves under the control of the two systems. "Trajectory step" refers to the discretized time step as the arm performs the motion.

segment. The output is the torque to be placed on the segment joint. The motion learning network starts its training once a sufficient number of samples (about 40 trajectories) have been collected so that it doesn't fall into a shallow local minimum and fail to learn the motor curves. Once it begins learning, we use the Levenberg-Marquardt learning algorithm (an optimization method which uses a modified Gauss-Newton method to estimate the second derivative of the error surface and improve convergence speed), two iterations per motion, which learns quite fast over the next two dozen or so trajectories, after which it has trained sufficiently well that it stops.

Figure 5.6 shows this learning process for actual trajectories. Each arm segment is controlled by the same steering process, and the comparison between forces (and angles) of the arm under the control of the logic and network show a very close correspondence.

During training, the control of the arm is shared by the network and the logic. This sharing is adjusted based on the current training error level of the network multiplied by a quality criterion calculated as logistic function of the size of the training data. This product is the quality parameter indicating how well the network has learned the data, and how much data there was to learn. If the quality parameter is 0.5, the network will be used for controlling the motion of the arm half of the time. If it is over 0.98, it will be used 100% of the time. As the network is trained, the quality parameters rises from 0 towards 1 as more and more data are accumulated and the network learns better and better to model it. After this threshold of 0.98, if the network fails to steer the arm to the required target, an error signal triggers the attentional mechanism, and the logic takes over and does the motion, thus producing more training data. While control is shared, it is switched randomly from the logic to the network-in-training during motion of the arm through a trajectory. This allows the arm to accumulate more data to improve its force modeling as it takes over more and more arm control.

Figure 5.7 shows these processes for one example. When 4000 examples are gathered, the network begins training. With fewer than that is more likely to result in fast initial training but result in a network stuck in a local minimum from which it

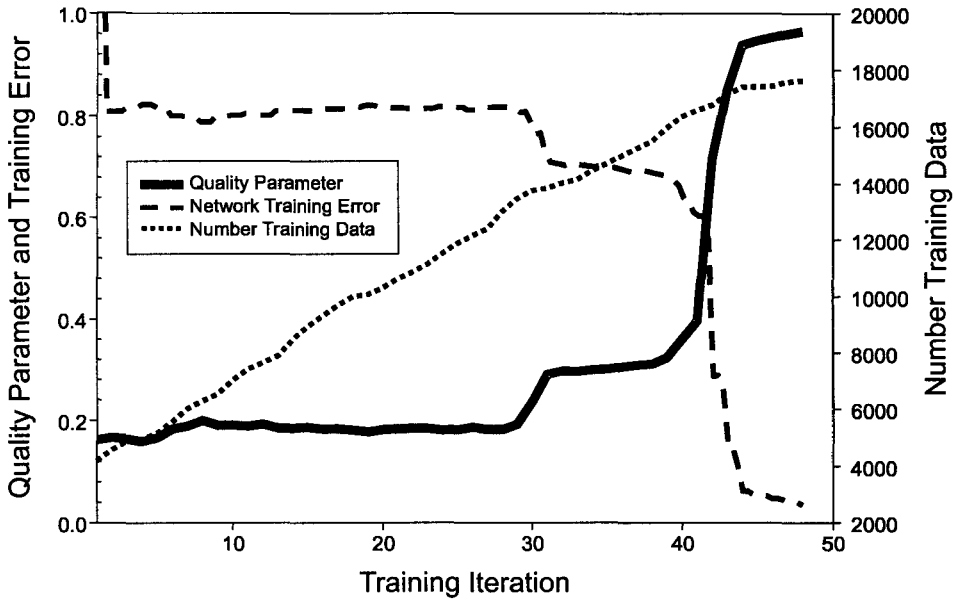

Figure 5.7: Details of kinetic network learning. The number of examples being trained on is shown in the right-hand axis. The training error and the resulting quality parameter are shown relative to the left-hand axis.

cannot escape. This is because since training is done on real motions of the arm, and uses a fast-converging learning algorithm, the network will overlearn the data from the first few trajectories and not generalize. If a slower learning algorithm is used (such as gradient descent), this is less of a problem. As training goes on (two iterations of Levenberg-Marquardt per arm motion), the training error drops. Also, more data is added and the two effects result in an improved quality parameter, meaning that the network begins to take over control of the arm movement. Generalization is excellent—this network, with a training error of .0346, has a test set error (over the whole input field) of .0017 (both taken in the least-squared sense). For actual trajectories the test set error is still very low but is higher (around that of the training set error). This is because for actual trajectories, there tend to be fewer points

θ_2	θ_3	θ_{target}	r_{target}	$\theta_{obstacle}$	$r_{obstacle}$
Angle of second segment with first segment of arm	Angle of third segment with first segment	Angle of target (from horizontal)	Distance from origin to the target	Angle of present obstacle (from horizontal)	Distance from origin to the obstacle

Table 5.1: Gesture memory input dimensions

scattered through the space, and more clustered in the areas where performance is more critical, such as when the arm segment is close to its target location.

5.5 Learning gestures with Adaptive Resonance Theory

For the middle level, we utilize an ART-like network as described by Grossberg and Carpenter [10], [11] to learn the various gestures utilized by the arm as it behaves in different problem environments. The ART architecture is an unsupervised learning model which auto-clusters the data which the logic presents to it during actual problem solving, and learns models for those clusters which can then be introduced into the control loop and, eventually, greatly reduce the need for the logic to compute the trajectories for the gestures itself.

Figure 5.8 diagrams the function of the ART network. The input space is the reduced representation of the environment which the logic has determined to be important in performing a particular gesture. (Going around an obstacle to the left doesn't

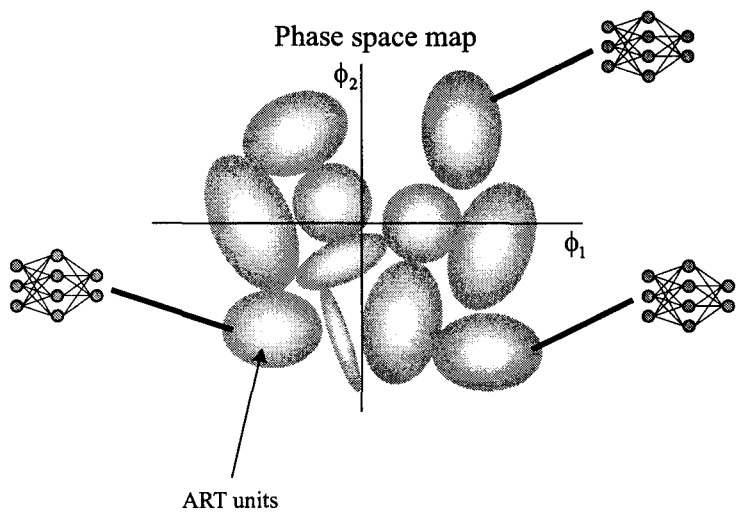

Figure 5.8: The Adaptive Resonance Theory network in diagram. Each unit exists in a part of the parameter space, where it can be in resonance with the current problem status. Resonance results in either training (if the logic unit is generating examples); failure of resonance results in the creation of a new ART network unit. ϕ_1 and ϕ_2 are representative phase dimensions. An ART unit is in resonance with the problem status when those status parameters are inside its extent within the parameter space.

depend on another obstacle on the other side of the board.) The learning problem, then, is pared down from a very-high dimensional one to a lower-dimensional one which only considers the next obstacle to deal with (or the basket to move to, if no obstacles are in the way). This results in faster learning and better generalization by the ART network. The network works in a six-dimensional space set by the relative angles of the second and third arm segments (relative to that of the first), the relative angles of the final target and the present obstacle, and their distances from the origin. These are shown in Table 5.1. Each unit maintains a mean and covariance matrix of the current data is has learned or is learning, and uses a gaussian distribution to compute a resonance with each new gesture to be learned. This resonance is computed as the probability of that gesture being within the mean/covariance model of the unit. The network as a whole uses a "winner-take-all" mechanism to decide which ART unit is most in resonance with the current input. If the resonance is greater than 0.5 (that is, if the new input is within two standard deviations of the mean of the winner-take-all-selected unit), then the input is considered to be in resonance with that unit, and is picked up by it.

Each unit is associated with a linear neural network which it uses to model the gesture parameters of the data it is in resonance with. These linear networks have six inputs (for the six dimensions of the data), and six hidden units. Once a unit has more than three data points, it begins to train its associated neural network to model the data. This training also uses Levenberg-Marquardt and operates up to ten training epochs for every new gesture data point added to it. Typically, though, fewer epochs

are required before achieving a minimum error of 5×10^{-5}. If a new data point ruins the ability of the existing network to model the data well, it is rejected and forms a new ART unit of its own (where it will compete with the existing units). If the new data point can be learned well, though (the usual case), then it is incorporated into a new estimate of the mean and covariance which is computed (using a gaussian model) of the unit's resonance region. The outputs of the neural network are relative coordinates for the segments of the arm to steer towards in completing the gesture. These coordinates are then mappable directly on the kinematic level for controlling the arm joints to move the arm to that configuration.

Once an ART unit has more than six data points, it is allowed to begin to respond to the environment itself, and if it is in resonance with the current environment (here, the resonance threshold is set to 1, that is, one standard deviation from the mean of the unit's data), it will control the choice of the next gesture. Each unit, then, corresponds to a "gesture" that the system has learned or is learning how to make. As enough examples are gathered to provide a good training base, the logic shifts control to the ART network and only calculates new gesture trajectories when no resonance with any unit is strong enough. Training and operation overlap: if resonance occurs with one of the existing units, and that unit has sufficiently good performance, it is used to construct the next trajectory. If resonance occurs, but that unit needs more data to train with, it is given the resulting data by the logic unit. If no resonance occurs, then a new unit is created, which will be added to and trained and used as that gesture is utilized by the system.

It only takes a few examples to build up a good gesture model (the trajectory models are quite linear for the parameter ranges within a gesture), so only six or seven trajectories are needed until they can start to be used. This linearity permits the networks associated with the ART units to usually achieve least squared training set errors below 10^{-3}, and frequently learn to the training threshold of 5×10^{-5} without significant overtraining. On the other hand, similar gestures may not be repeated by every movement from one basket to another, so it may take a while (in puzzle-solution time) to accumulate the few examples needed from real solutions. An antidote to this would be "directed play." That is, the generation of scenarios which are in resonance with the ART unit that we want to train, but which don't have anything specifically to do with an example puzzle. These scenarios, though, would be generalizable to real puzzle solution.

5.6 Learning directions

The sequence to follow when moving from one peg to another is memorized using "declarative memory." These are basically memorized series of gestures: "to go from peg 1 to peg 3, first go clockwise around obstacle 2, then counter-clockwise around obstacle 6, then straight on to peg 3." These directions are learned as the logic solves the puzzle and continue to evolve as play proceeds. If a sequence of directions doesn't result in success, then we go back to the originating basket and try again by generating a new set of directions. These will replace the original set. Since it only takes one example for this memory to be useful, the sequence memory comes

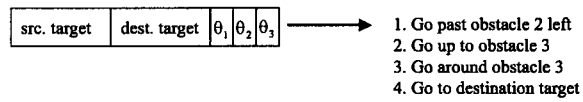

Figure 5.9: Diagram of the structure of sequence, or directions, memory used by the system. For a particular motion which the abstract puzzle solver indicates, memory is searched to find sets of directions which correspond to the desired movement between targets. The memory that matches the current position of the arm the best is selected.

into play quite quickly. On the other hand, it is not generalizable at all, except to different arrangements of the items to be sorted on the same playing board (meaning the same pegs and obstacles in the same position, just a redistribution of the disks to be moved).

The sequences of directions are remembered in a conventional array type memory. The initial point of the arm position is stored along with the target we are going from and the target to which we are going. When recalling these memories, only those which match the targets between which we want to move (for example, from target 3 to target 2) are considered, and the sequence memory closest to the present arm position is selected. If the variation from the current position is more than 10 degrees away, no memory is recalled and a new one must be generated by the logic. Figure 5.9 illustrates the structure of the directions memory.

If following a memorized set of directions results in an error (that is, when the directions have been followed, the arm is still more than 0.06 distance units from the target), the logic is alerted by the error signal to generate a new path for the arm by going back to the previous starting point and starting over. The new sequence which is generated will replace the existing sequence in the memory.

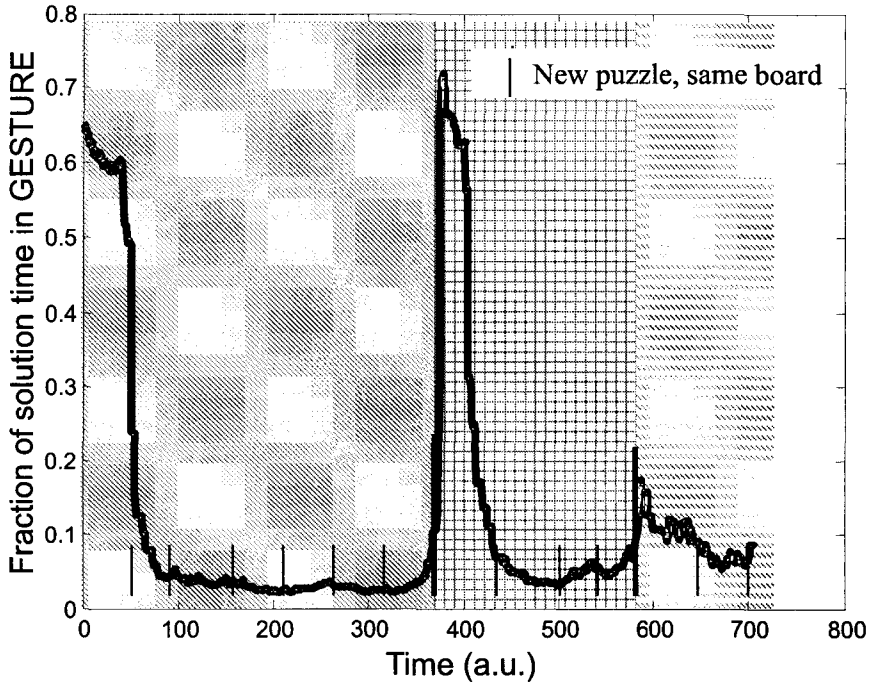

Figure 5.10: Learning gestures. Here is shown the learning progression of the system as it learns to perform the gestures needed in solving the problem. The vertical lines separate the different puzzles the system is solving. The data is averaged over a 40-move moving average. (A puzzle typically takes between 40 and 60 moves to solve.) A move is moving a disk from one peg to another and is an aggregate of several gestures. The different backgrounds correspond to different arrangements of obstacles and pegs. The smaller lines separate puzzles solved on the same board arrangement (that is, the same pegs/obstacles but a different initial distribution of the disks).

5.7 Dynamics of learning

Figure 5.10 shows the results of gesture learning as the arm solves the problem. When the arm begins playing, the logic unit is doing all the planning, as well as gesture computation and arm kinematics. As the arm behaves, its learning mechanisms began to take over operation. This figure shows how the unsupervised ART-like gesture learning system takes over the planning of gestures to go from one peg to another. During the first few puzzles, the logic has to spend a lot of the solution

time (around 60%) planning these gestures. After that, however, the system begins to learn commonly repeated gestures (like moving clockwise a short distance around an obstacle) and takes over the gesture planning. This reduces the total time spent in problem solution, and also dramatically reduces the amount of time the logic spends "thinking about" gestures—it drops to less than 10%.

There is a comparable reduction in the amount of time the logic has to dedicate to training the network that learns arm kinematics (in a supervised way), and learning the gesture sequences to go from one basket to another. The learning of gestures is the most dramatic example, however. After these networks are trained and operating, the logic unit spends its time figuring out which overall motion to perform next (go to peg 3, then to peg 1), and in dealing with error signals propagating up from its reflex networks when they make a mistake and it has to assist them.

Figure 5.11 shows an example of the arm's behavior after several puzzles have been solved and the arm is more trained. The bars indicate times when the error signal arouses the attention of the logic unit because of an error in the way the memory subsystem has directed the arm. The error signal is generated as a difference in position from the target (peg 2) and the current state of the arm after following the directions in memory. If this difference is large enough, it means the arm hasn't been directed to the place the logic expected it to go, and so triggers an attentional "interrupt." Then, the logic recalculates the gestures and/or the directions needed to control the movement for which there was an error. Sometimes, this process doesn't take a lot of extra time. This happens when the required motion is fairly easy to

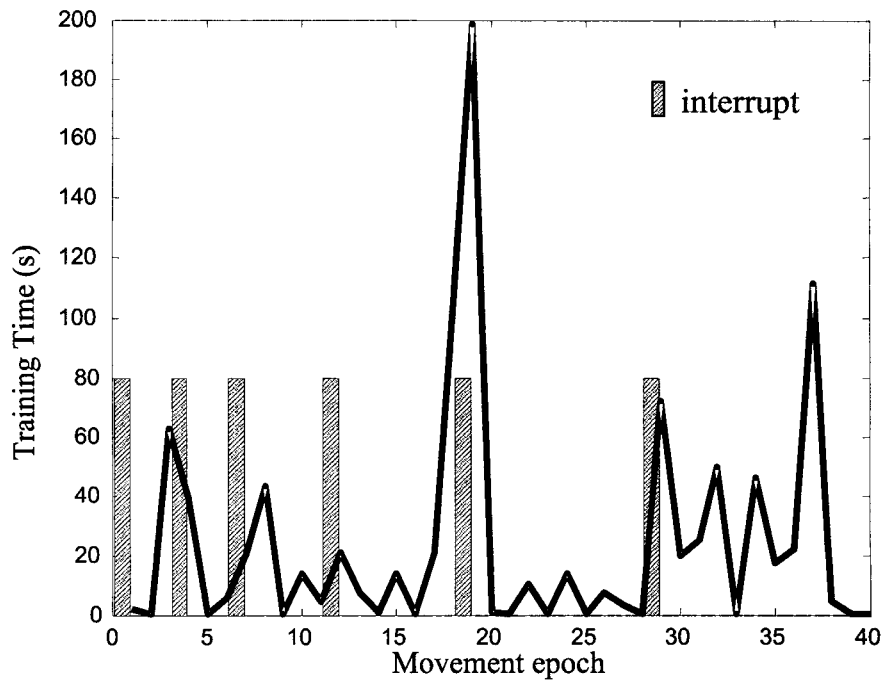

Figure 5.11: Interrupt-driven training. The bars locate interrupts to the logic when the performance error signal tells it that the memory-controlled subsystem needs more training. The solid line is the training time spent. Sometimes, the error is corrected without very much additional training time. Other times, much additional time is needed to recalculate the gestures and directions needed for a particular movement. Other times, a motion requires the attention of the logic, but is not being signaled by an interrupt.

optimize, or when there was an error in the directions, and the gestures required can simply be pulled from memory and reassembled. Other times, an interrupt requires quite a bit of extra time from the logic. Also, there are times when the logic is spending time training the memory without an error interrupt. This is when the motion hasn't been learned yet, so the logic subsystem is still teaching that gesture, or set of directions, to the memory.

We now return to a discussion of the operation of the system in terms of the blocks of Figure 5.1. The angular position and velocities of the arm segments, and the positions of obstacles, baskets, and disks, form the environment. From this environment, the job of the controller as a whole is to move the arm in the best way to solve the puzzle. The solution of the puzzle at the abstract level (that is, which move to perform next with a view to solving the problem) always remains the province of the logic. Before the network is trained, the logic/planning subsystem feeds back to the attentional process to select only the relevant parts of the environment, thus dramatically simplifying the optimization task it performs and assisting memory formation speed and improving generalization. There is some sacrifice in optimality here, but when time-pressured, the behavior is much faster than would be the case with global optimization.

When training is more complete, the attention mechanism is used to propagate "error" messages up to the logic, telling it when the (mostly trained) zombie levels have failed to do what the logic thought they would, so that it can improve their training. From the perspective of the logic, this training consists of either rememo-

rizing a sequence of directions, at the top level, or "practicing" the movement again for training the lower levels. The lower levels are not explicitly memorized by the logic unit into declarative memory. They are trained through behavior. The motion level learns in a supervised fashion as the logic controls the arm. The gesture level learns in an unsupervised fashion from the behavior of the logic in choosing gestures to execute. This illustrates the role of the reduced representation of the short-term memory in interfacing to different kinds of memory. For declarative memory, the bottleneck reduces the amount of information necessary for it to learn useful patterns. For procedural memory, whether supervised or unsupervised, it assists the logic unit by pruning out the information in the environment that is less relevant, making the resulting patterns lower dimension and more rapidly learned by the procedural memories.

As the game is played, then, the expensive logic unit is used more and more optimally: at first, it has to not only figure out which pegs to move the arm to next in order to solve the problem, but also figure out which obstacles it must get around in what order, what series of gestures it needs to do that, compute trajectories for each gesture, and then guide the arm through the trajectories. As it learns, the memory subsystems take over more and more control, learning the motor sequences necessary to guide the arm, the gestures needed to bypass obstacles and get the end effector near the target pegs, and the directions to get from one peg to another, leaving the logic unit free to operate at the abstract level of puzzle solution. This greatly improves the speed of performance: figuring out arm ballistics and computing near-

optimal gestures are hard problems, and when these responses have been learned, performance (in terms of speed) can be greater by an order of magnitude and more.

The problem of using an arm to solve a puzzle is one in which neither pure logic nor traditional all-network learning is very good. When the logic has to plan out in detail all the motions of the arm, it can take a very long time to solve the problem. The conventional learning problem is intractable. For even quite small problems, there are dozens of dimensions in the learning problem, generating error signals is hard, and learning is very slow, if it would work at all.

Figure 5.12 shows an example of how the arm moves during the solution of a particular sorting problem. As can be seen, the arm in its history navigates all its joints successfully to avoid the obstacles. The kinks mark the separation between different gestures the arm makes as it moves from target point to target point.

5.8 Bin packing problem

The hierarchical learning method allows us to make changes in the abstract problem solving level and use the same learning structure to quickly change the task the system learns to solve. We did this for the bin packing problem [7]. The puzzle is to put a given number of blocks (we use puzzles of 35 blocks), with random sizes from .01 to 1, into as few bins (of size 1) as possible without overfilling any bin. The system must move the arm's end effector between the spots on the board where blocks to be packed are placed, and the spots on the board where the bins are in which they are to be put. The three situations of interest to us are diagramed in Figure 5.13. Every

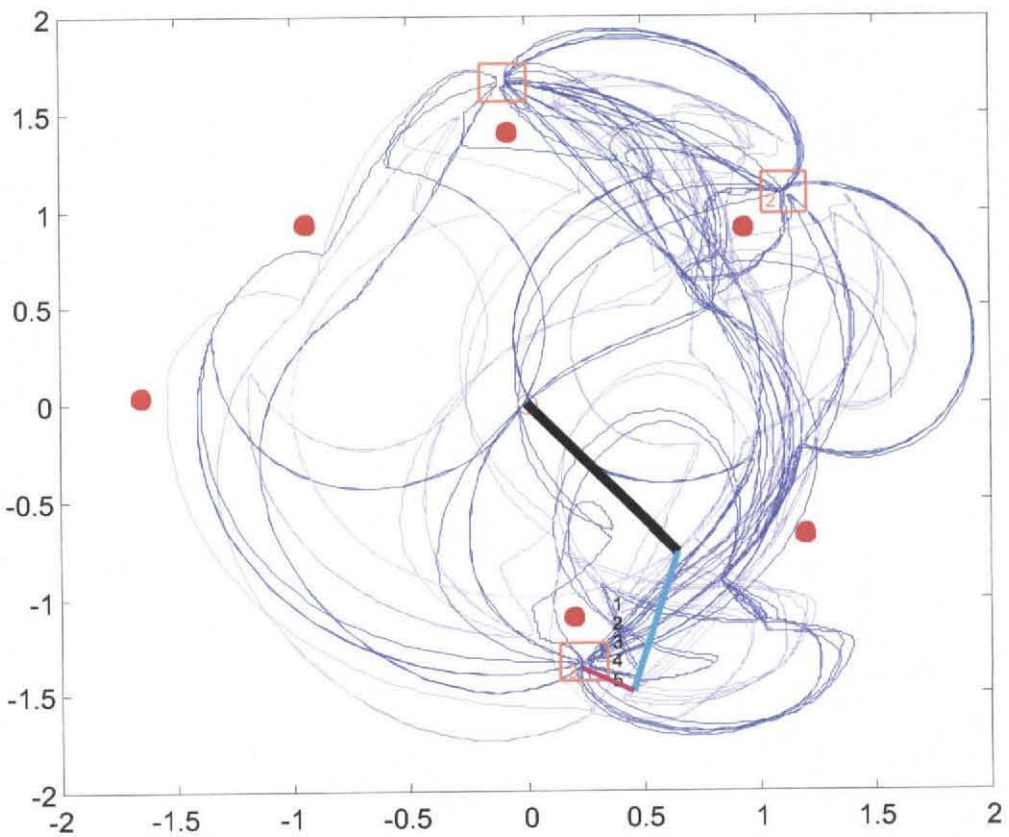

Figure 5.12: The traces from the arm motions as it solves a Tower of Hanoi sorting problem. The solid circles are obstacles (they only touch the lines because they've been enlarged for easier identification). The squares mark the locations of the pegs.

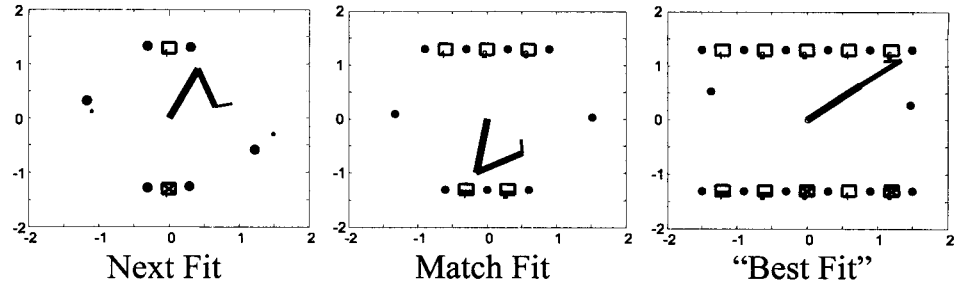

Figure 5.13: Three bin packing puzzles. The Next Fit puzzle uses a single spot for a block and another single spot for a bin. The second case (Match Fit) considers two blocks and three bins at once. The third case (close to Best Fit) utilizes five blocks and five bin locations. The blocks come in at the bottom of the playing field. They are shown as X's in the target locations, with the size of the block in the examples indicating the size of the block. The target bins are located at the top of the playing field. There are obstacles between all the target bin locations and the block locations, as well as two obstacles to the left and right.

40 seconds (in puzzle solution time), if the system hasn't yet picked up a block to place, one is removed from the set of blocks with which it is working. This enforces a real-time behavioral performance constraint on the system: if it cannot behave within this time limit, those blocks it fails to handle will be placed one per bin in the final solution.

At the abstract level, the algorithm used to decide where to place blocks and bin is the Match Fit algorithm, described in Chapter 2. As discussed there, when there are only one block and one bin location used by the algorithm, the performance is equivalent to that of the Next Fit algorithm. As the number of bins used grows, the performance becomes that of the Best Fit algorithm. In between, the Match Fit algorithm, by making use of the most important parts of the interim solution, can achieve performances near that of Best Fit, while using fewer computational resources,

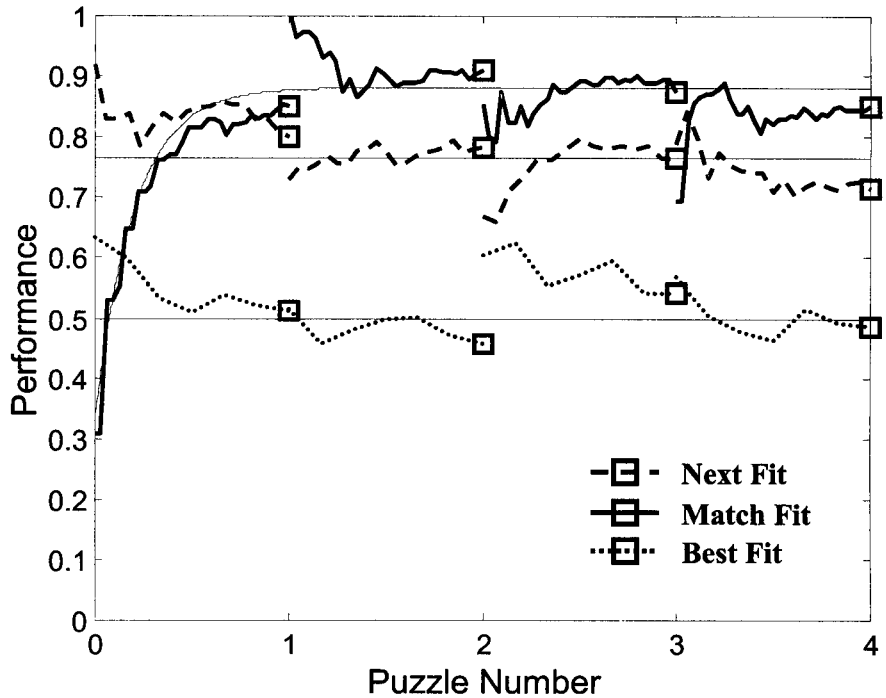

Figure 5.14: Comparison of performance on the bin packing puzzle in three situations. The Next Fit situation results in the expected performance of 0.75. In the Match Fit situation, the system learns the gestures and arm force movements during the first game, and thereafter can perform better: at a level of 0.88. In the Best Fit situation, the system is much, much slower in learning the many more motions, and so performance remains stuck at the level of 0.5. Data are shown for a typical trial run, so the starting values of the performance for all three algorithms are largely noise, and depend on the (random) sizes of the first few blocks to be packed.

and therefore does much better in time-constrained situation.

That same comparison holds when memory is involved, as can be seen in Figure 5.14. In the Next Fit case, learning is very quick, and the performance is stable at the expected value of 0.75. Here, there are only two motions for the arm to learn: going from the only block location to the only bin location. The transition from logic control of motion to network control is very smooth, since because of the regularity, the data is very consistent and easy for the system to model.

In the Match Fit case, learning is slower, and the interim performance can be seen rising during the first game as the system must spend longer optimizing the individual movements of the arm, and therefore loses some of the blocks to the time limit. After the first game, though, enough of the motions have been learned so that very few blocks are lost, and performance stabilizes at a mean of 0.88.

The Best Fit case shows, though, that simply adding more elements to the mix can harm performance. Here, there are five block locations and five bin locations. The logic unit must spend a lot more time choosing which motion is the best and then optimizing gestures to complete that motion. The gestures tend to be more complicated, and thus take longer to learn. Waiting a long time would result in eventual rises in performance by the system as the more complicated gestures were learned, but this is much slower than for the Match Fit case, and here performance stays at 0.5 as almost all the blocks are lost to the time limit.

Figure 5.15 shows how the time taken to do individual moves (from one spot to another on the playing board) decreases as the arm becomes more proficient in its movements. As the trained memory takes over from the logic/planning subsystem, the learned optimizations are easier to achieve, and so motion speeds up dramatically. In this case, by a factor of about 5. In problems like the Tower of Hanoi puzzles, which require more arm movement, this speed-up is greater (a factor of 10 and higher).

Figure 5.16 shows the fraction of time the system spends in the logic/planning subsystem and the training and recall of various memory subsystems. During solution of the first puzzle, the system spends almost 100% of its resources training the

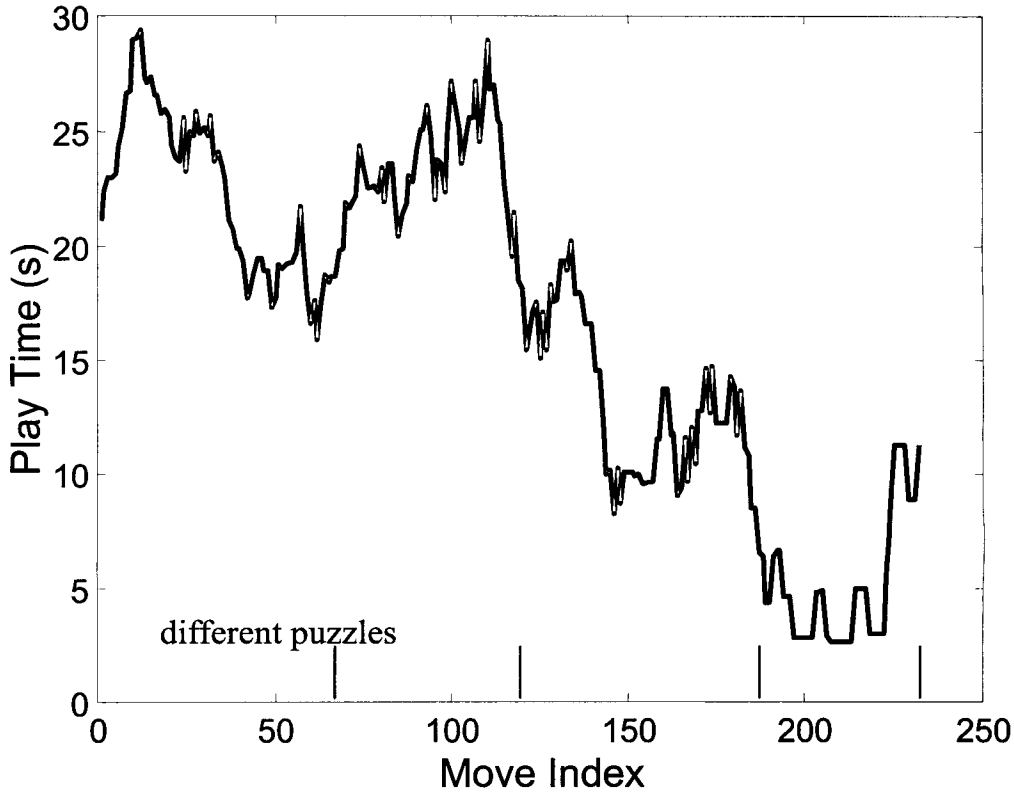

Figure 5.15: Time performance over several puzzle solutions (Match Fit case). The time shown is the time to complete one move in the problem solution. The problems take between 60 and 70 moves to solve. The time shown is a running average over 15 moves.

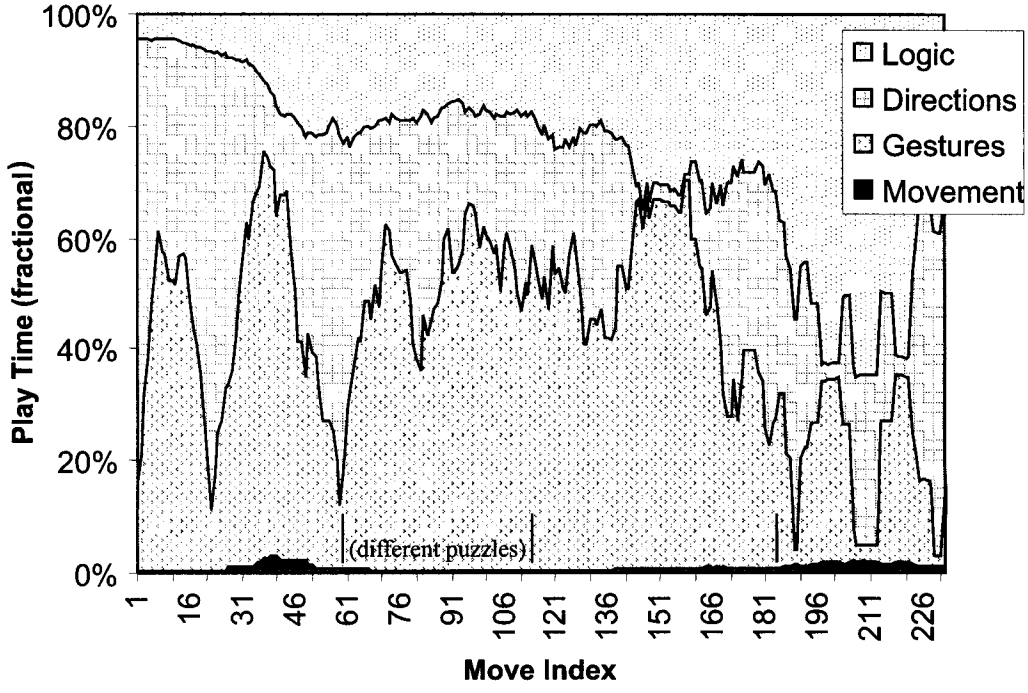

Figure 5.16: Fractional time spent by the system as a whole within the various subsystems (movement, gesture learning and recall, directions learning and recall, and the overhead of operating the logic). The time shown is a running average over 15 moves.

gesture and directions memories. It spends a few percent of its resources training the movement memory as well, but, as mentioned above, this is not so resource intensive, and training is rather rapid once sufficient data is collected. As the system operates, however, the fraction of time it spends training the gestures and direction sequence memories decreases, and more time is spent doing logic/planning overhead tasks (figuring out where to play next, etc.). By the time the system is solving the fourth puzzle, resource utilization is taken up mostly by the logic/planning subsystem. Total solution time, as discussed above, has dropped to below 20% of what it required in the first problem, and so responding to interrupts when the memory subsystems fail to guide the arm correctly accounts for almost all of the time spent in these subsystems (using them in recall mode is hundreds of times faster than the time spent doing the optimization needed to provide them with new training data).

Figure 5.17 shows that these subsystem resource utilization times as in Figure 5.16 are consistent on the average. Here, data from 19 trials are averaged, showing a consistent decrease in system resources dedicated to movement, gesture, and directions computation and learning as time goes on. The figure gives cumulative times, so that the time spent on computing or learning gestures is the difference between the line with this label and the line labeled "movement." The error bars shown are standard deviations from the mean. The standard deviation remains high even after the system has been operating for 200 moves and more because in this regime, the system time is dominated by the "interrupts" caused when the learned response causes an error, attracting the "attention" of the logic subsystem. The decreasing amount of

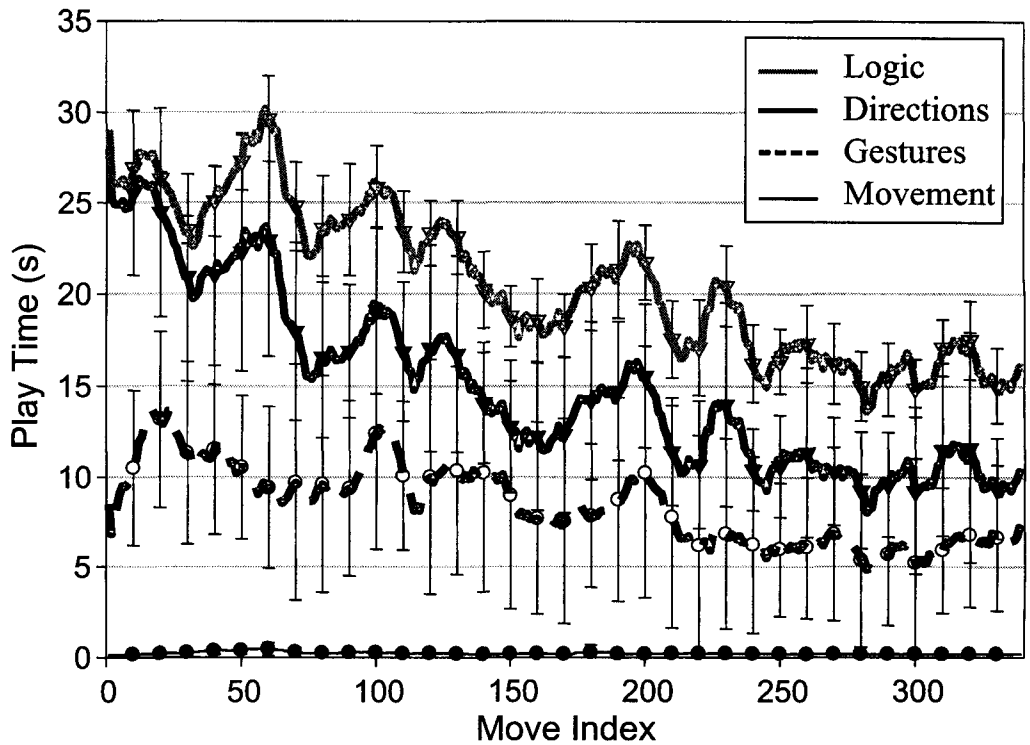

Figure 5.17: Cumulative average time spent by the system (over 19 trials) in various subsystems while solving a bin packing problem. Time is shown cumulatively, so that the distance between lines is amount of time spent on that subsystem. Error bars are shown only every 10 moves. The time shown is a running average over 15 moves.

time spent in the average indicates the decreasing frequency of these interrupts, but as handling these interrupts becomes the dominant way the system spends time in the subsystem, the variance in that time spent remains large.

5.9 Conclusions

Learning with the assistance of a mechanism which uses attentional processes assist a logic/planning unit in training a hierarchical memory has several benefits. First, it dramatically reduces the dimensions on the input space. The sorting problem as described has some 30 dimensions, with dependencies that are ill-suited to learning by a traditional neural network. By segmenting the process and learning simpler approximations in a hierarchical way, using the reduced representations as utilized by the logic/planning subsystem to solve the problem at different levels, it becomes possible to present tractable problems to the neural networks.

Second, the hierarchical organization of skills enables those learned the fastest to be assumed during the learning of higher-level skills.

Third, the attentional mechanism as employed allows for cooperation between the memory subsystems and the logic/planning subsystems. When the faster network-based subsystems can respond, they do so. It is only when they make errors that the logic subsystem is aroused and spends time correcting the error.

Bibliography

[1] Avnimelech, R. and N. Intrator. Boosted Mixture of Experts: An Ensemble Learning Scheme. *Neural Computation* **11**, pp. 483-497, 1999.

[2] Buckner, R. L., W. H. Kelley, and S. E. Petersen. Frontal Cortex Contributes to Human Memory Formation. *Nature Neuroscience* **2**, pp. 311–314, 1999.

[3] Chen K., X. Yu, and H. S. Chi. Combining Linear Discriminant Functions with Neural Networks for Supervised Learning. *Neural Computing and Applications* **6**, pp. 19–41, 1997.

[4] Claus, N. La Tour d'Hanoi: Jeu de Calcul. *Science et Nature* **1**, pp. 127–128, 1884.

[5] Dempster, A. P., N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm *J. of the Royal Statistical Society B* **39**, pp. 1–38, 1977.

[6] Funahashi, K. On the Approximate Realization of Continuous Mappings by Neural Networks, *Neural Networks* **2**, pp. 183–192, 1989.

[7] Garey, M.R. and D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, CA, 1979.

[8] Gault, D. and M. Clint. A Fast Algorithm for the Towers of Hanoi Problem. *The Computer Journal* **30**, pp. 376–378, 1987.

[9] Grossberg, S. The Link Between Brain Learning, Attention, and Consciousness. *Consciousness and Cognition* **8**, pp. 1–44, 1999.

[10] Grossberg, S. and G. A. Carpenter Pattern Recognition by Self-Organizing Neural Networks. MIT Press, Cambridge, MA, 1991.

[11] Carpenter, G. A. and S. A. Grossberg. A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine. *Computer Vision, Graphics, and Image Processing* **37**, pp. 54–115, 1987.

[12] Grossberg, S. and R. W. Paine. A Neural Model of Cortico-Cerebellar Interactions during Attentive Imitation and Predictive Learning of Sequential Handwriting Movements. *Neural Networks* **13**, pp. 999–1046, 2000.

[13] Hirano, T., M. Sase, and Y. Kosugi. Bidirectional Feature Map for Robotic Arm Control. *Systems and Computers in Japan* **25**, pp. 91–100, 1994.

[14] Jacobs, R. A., M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive Mixtures of Local Experts. *Neural Computation* **3**, pp. 79–87, 1991.

[15] Jordan, M. I. and R. A. Jacobs. Hierarchical Mixtures of Experts and the EM algorithm. *Neural Computation* **6**, pp. 181–214, 1994.

[16] Jordan, M. I. and L. Xu. Convergence Results for the EM Approach to Mixtures of Experts Architectures *Neural Networks* **8**, pp. 1409–1431, 1995.

[17] Kaykobad M., S. T. U. Rahman, R. A. Bakhtiar, and A. A. K. Majumdar. A Recursive Algorithm for the Multi-Peg Tower of Hanoi Problem. *International Journal of Computer Mathematics* **57**, pp. 67–73, 1995.

[18] Martin P., J. D. R. Millan. Learning of Sensor-Based Arm Motions while Executing High-Level Descriptions of Tasks. *Autonomous Robots* **7**, pp. 57–75, 1999.

[19] Naveh-Benjamin, M. and J. Guez. Effects of Divided Attention on Encoding and Retrieval Processes: Assessment of Attentional Costs and a Componential Analysis *J. of Experiemental Psychology–Learning, Memory, and Cognition* **26**, pp. 1461–1482, 2000.

[20] Sabes, P. N. The Planning and Control of Reaching Movements. *Current Opinion in Neurobiology* **10**, pp. 740–746, 2000.

[21] Schoute, P. H. De Ringen van Brahma. *Eigen Haard* **22**, pp. 274–276, 1884.

[22] Uno, Y., M. Kawato, and R. Suzuki. Formation and Control of Optimal Trajectory in Human Multijoint Arm Movement: Minimum Torque-Change Model. *Biological Cybernetics* **61**, pp. 89–101, 1989.

Chapter 6 Conclusions

We have explored several aspects of the application of attentional and awareness mechanisms, observed to function with such efficacy in biology, to computer algorithms. There are many benefits of such systems in biology. The most interesting for our purposes are the way in which they facilitate the abilities of humans to function in complex, unpredictable, real-time environments without overloading the brain with processing demands, and the way in which attentional processes aid in learning and memory formation.

To explore the first characteristic, we tested algorithms which drew from the model of awareness to produce reduced representations of their complex domains for real-time solution in testbeds consisting of a traditional computer science problem—bin packing—as well as a simulated computer strategy game. The generality of our approach is dependent on some features of the problem which are true for a wide, although not universal, class of other hard NP problems. First of all, the problem is amenable to partial solution. That is, there is a way to assign a value to a non-perfect solution. This is widely true for NP problems, but does not describe them all. For example, the traveling salesman problem has this quality: performance can be assigned to non-perfect solutions on the basis of how far the salesman must travel. The SAT problem is not amenable to this real-valued performance criteria, however. A solution either satisfies the SAT problem or it does not.

Secondly, the bin packing problem has the property, explored in Chapters 2 and 3, which makes additional information about the problem carry less and less importance. This is the property that allowed us to create a new algorithm which uses very limited computational resources to achieve performance comparable to more resource-intensive existing algorithms. We argue that this is related to a commonly observed property of NP and other hard problems, which is that while finding an exact solution is very, very difficult, finding a bad solution is much easier. Again, this is a property shared by many problems but not all. (The same examples of traveling salesman and SAT apply.)

The common (although not universal) occurence of these properties support the argument, for which there is yet no formal proof, that there is a wide class of problems for which algorithms exist between maximally greedy and full-blown combinatorial search which perform well under time pressure and exhibit this optimum in a cache memory size. The ability to perform at less than perfect means that such heuristics exist, and the decreasing importance of having all information about the problem in hand at all times leads to an performance optimum with cache memory size. This means that when forced to behave under time pressure, such algorithms will perform best when they ignore large parts of the problem. If this "awareness paradigm" were invalid, there should be a pressure to use more and more information and this optimum would not exist. The details of the composition of this class of problems is unknown, and an area for further research.

This same conclusion was apparent in the application of the awareness model

to the computer game. Again, when resources were constrained, the performance characteristic of an algorithm which uses the awareness model has an optimum. In this testbed, we can see the awareness model compete directly with an algorithm which takes the entire playing area into account as it chooses what moves to make. Such an algorithm wins with no time constraints, but under time pressure, the awareness-inspired algorithm wins the game.

Another feature to come out of the bin packing testbed was the very fast transition between the situation when the awareness strategy was optimal (for cases where there was more time pressure) to the situation where indeed the pressure is to use more and more problem information. The conclusion here is that for even relatively mildly time-constrained situations, where optimal solution-finding is only just barely impossible, it is still better to use fast awareness-analog heuristics which use a reduced representation of the problem. We believe that this transition is reflective of the possibilities inherent in solving hard problems of this type. Given a reduced representation of the data, more time is put to better use by fully exploiting that reduced data, rather than spending it processing more data. It may be that this has implications for information processing pressures on the evolutionary development of the brain's attentional mechanism.

We also applied the awareness model to a learning problem. When assisted by a supervisor which limits the amount of information used to train a neural network, we find that the network can actually learn faster than when presented with the raw training data. There are other added benefits, such as the ability to adaptively

determine which subfunctions need the most computational resources and a robust resistance to noisy input data. Here, the bottleneck of the attentional mechanism resulted in fast, robust learning. In a situation where a learning system faces complex, high-dimensional data and a need to behave in real time, an architecture similar to the one we describe can make learning feasible.

There are many directions in which this work can be extended. Testbeds make convincing arguments, but real-world applications of these architectural ideas are needed to make a more developed case for awareness-inspired algorithms. There are many areas in the real world where time sensitivity is important, the environment is complex and uncertain, and the problems to be solved (or learned) are of the kind where only some of the information matters at any given time. Driving, for example, is such a case. Most of the time, a driver can allocate resources to other tasks, such as talking on a phone or listening to the radio or music. Sometimes, however, such tasks are dangerous distractions. An application of the attentional/awareness architecture here could be made in two areas. First, in an assistive capacity to aid the driver in identifying situations where full attention is needed for driving, and when, in contrast, the driving task is undemanding and resources can be used in other pursuits. Second, the model could be applied to the car's systems themselves, in an active capacity allowing the car to react in real-time in an autonomous driving situation.

There is much more that can be done by way of exploring the connection between the learning aspects of the attentional model and the more real-time behavioral aspects. A combined system would ideally use attentional mechanisms to learn what

kinds of triggers and information define important, or salient, parts of its environment, depending on its current goals, and then to use that knowledge to produce reduced representations appropriate for time-critical processing and decision-making. An example of such a system might be a game-playing system which bootstrapped itself from very little knowledge of the rules of a real-time strategy game similar to the Desert Survival game used in our testbed to play the game well. Also, the layered, hierarchical system we present could be extended in several ways. One interesting direction of exploration would be the development of a "meta-learning" interrupt mechanism which would trigger the addition of additional layers.

As more is learned about the neurological components of the awareness and attentional systems in humans, there will undoubtedly be further enhancements and refinements which are suggested in the model we explored here. Of particular interest are the feedback mechanisms whereby the higher-order brain functions, such as planning, decision-making, etc., can influence the lower-order functions, such as attentional processing, to provide them with signals related to what the system is interested in. These feedback loops are important to a fuller understanding of the function of awareness and attention in the human brain and will certainly provide much inspiration for future computer algorithm implementations.