Fall 2009

Meta-learning computational intelligence architectures

Ryan J. Meuth

META-LEARNING COMPUTATIONAL INTELLIGENCE ARCHITECTURES

by

RYAN JAMES MEUTH

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

2009

Approved by

Donald C. Wunsch II, Advisor
Daniel Tauritz
Ganesh K. Venayagamoorthy
Randy H. Moss
Daryl Beetner

ABSTRACT

In computational intelligence, the term ‘memetic algorithm’ has come to be associated with the algorithmic pairing of a global search method with a local search method. In a sociological context, a ‘meme’ has been loosely defined as a unit of cultural information, the social analog of genes for individuals. Both of these definitions are inadequate, as ‘memetic algorithm’ is too specific, and ultimately a misnomer, as much as a ‘meme’ is defined too generally to be of scientific use. In this dissertation the notion of memes and meta-learning is extended from a computational viewpoint and the purpose, definitions, design guidelines and architecture for effective meta-learning are explored. The background and structure of meta-learning architectures is discussed, incorporating viewpoints from psychology, sociology, computational intelligence, and engineering. The benefits and limitations of meme-based learning are demonstrated through two experimental case studies – Meta-Learning Genetic Programming and Meta-Learning Traveling Salesman Problem Optimization. Additionally, the development and properties of several new algorithms are detailed, inspired by the previous case-studies. With applications ranging from cognitive science to machine learning, meta-learning has the potential to provide much-needed stimulation to the field of computational intelligence by providing a framework for higher order learning.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Donald C. Wunsch, for his support and guidance throughout my academic career. This work was supported in part by The Boeing Company, the National Science Foundation, the Missouri S&T Intelligent Systems Center, and the M.K. Finley Missouri Endowment.

I would also like to thank my advisory committee, Dr. Daniel Tauritz, Dr. Kumar Venayagamoorthy, Dr. Daryl Beetner, and Dr. Randy Moss. Their input and academic support are greatly appreciated.

Finally I would like to thank my family and friends for their love, emotional support and warm homes and hearts throughout my long academic journey.

Thank you all. I would not have been able to come this far without you.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

Over the past several years many hundreds of papers have been published on the modification and application of only a handful of core computational intelligence techniques – namely dynamic programming, evolutionary algorithms, neural networks, fuzzy logic, and data clustering methods. Algorithmically, there have been refinements and crossovers in these categories, such as heuristic dynamic programming, particle swarm optimization, evolutionary-trained fuzzy neural networks, and hybrid genetic algorithms, resulting in significant but relatively modest quality and performance gains. Beyond these modifications, the investigation of drastically different algorithm designs has been relatively slow for, while the complexity of machine learning and optimization problems has grown ever larger with the maturity of the internet, digital media, and the proliferation of data sources in all aspects of human life.

Meanwhile, advancement in hardware technology has brought about affordable and powerful computing platforms which are more easily accessible. However, it is clear that increase in computational capacity cannot even come close to addressing the challenges posed by the complexity of problems, many of which are typical of real-world scenarios [1]. More advanced and novel computational paradigms must be championed, particularly from the point of view of algorithm development. Early in the history of modern computing, algorithms by and large were able to keep up with the demands of increasing real-world problem complexity. To a certain extent, the algorithms which typically belong to the category of conventional or exact enumerative procedures were

able to surpass the complexity of problems that were typical of what people were trying to solve. Subsequently, as the complexity of problems continues to push the capability limits of algorithms, it is evident that the complexity of problems being addressed overwhelms the algorithms available. It can be envisaged that in time, the spread between complexity of problems and algorithms will widen if computational intelligence remains at status quo. There are clear signs that these issues are in the early stages of being addressed. Research should be putting emphasis not just on learning *per se*, but rather on issues pertaining to higher order learning – methods that not only solve a current instance, but are able to handle problems across instances, to learn about solving problems.

Computational intelligence to a certain extent manages to contain the gap between algorithms and problems, but we are beginning to see the growth in problem sizes outstrip computational resources and algorithms. Examples include large data-mining projects such as internet search indexing, document understanding, and the ever-present pursuit of intelligent machines. Modern day optimization techniques can rise to this challenge by incorporating not just mechanisms for adaptation during the process of solving an instance of a difficult problem, but rather automatic mechanisms for learning spanning across instances of problems encountered during the course of long-term optimization.

Many current methods utilize aspects of these mechanisms in order to increase performance, but none utilize them all, and few incorporate these features in an automatic way. A certain degree of similarity may be drawn when compared to case-based reasoning (CBR), such perceived “experiential” trait similarity in the sense that both

encompass mechanisms to draw on "experience" from previously encountered problem instances is superficial. Unlike CBR methods which rely on the need for explicit examples and ranking procedures, optimization problems are usually not amenable to such explicit case by case assessment to yield information that is potentially useful to a search algorithm [2, 3].

In artificial neural networks (ANN's), long-term optimization may be incorporated through extended training, but this often requires the intervention of users, and often this greatly affects the ANN's generalization capabilities. Similarly, evolutionary algorithms require parameter tuning, which has been automated, but often these tunings are only valid for a specific class or instance of a problem. Simple architectures of these algorithms and their limitations are further discussed in Section 2.

Emphasis should be placed on the automated construction of a body of knowledge, more specifically memes and meta-memes that collectively offer capability with a much broader problem-solving scope in order to deal with the class of problems being addressed.

As part of a historical context, Wolpert and Macready formalized the "No Free Lunch Theorem" in 1997, stated simply:

> *"Any two [non-repeating black-box search] algorithms are equivalent when their performance is averaged across all possible problems."*

The No Free Lunch (NFL) Theorem was developed specifically for discrete problems, constraints which few real-world problems satisfy. Additionally, Wolpert and Macready made the observation that in order to reduce the average cost across a set of problems and optimizers, one must methodically utilize prior or acquired information about the matching of problems to procedures, given *a priori* knowledge gained from

experience [4]. The realizations brought by both the details and the mythology of the NFL changed the research focus of the field of computational intelligence from the design of individual algorithms to the design of architectures of algorithms and parameter optimization. It is in this spirit that the development of memetic algorithms has been motivated [5-14].

Taken alone, current methods tend to be overwhelmed by large datasets and suffer from the curse of dimensionality. It is the central thesis of this document that a new class of higher order learning algorithms is needed that can autonomously discern patterns in data that exist on multiple temporal and spatial scales, and across multiple modes of input. These new algorithms can be architectures utilizing existing methods as components, but to design these architectures effectively, some design principles should be explored.

Ultimately, the curse of complexity cannot be wholly avoided. As the size or dimension of the problems increases, a greater amount of computation becomes necessary to find high quality solutions. However, such computation need not be completed at the exact time that a problem is presented. If a memory mechanism is provided that can effectively store and retrieve previously used or generalized solutions, then computation can be shifted into the past, greatly reducing the amount of computation necessary to arrive at a high quality solution at the time of problem presentation.

One of the major drawbacks of evolutionary algorithms and computational intelligence methods in general is that the solvers employed usually start from zero information, or utilize random initial states, independent of how similar the problem instance is to other instances the method has been applied to in the past. In effect, the

optimization methods typically do not incorporate any mechanisms to establish inter-instance memory. Parameter recommendations and user-seeded known-good initial evolutionary algorithm populations provide some inter-instance information, though this knowledge is provided by the operator, and gained through human experience. This random initialization property is useful for comparing different computational intelligence methods and in some cases, particularly when computation time is not an issue, is desirable as it allows the search to be more focused, thus leading to solutions that would not otherwise have been found efficiently. It is also worth noting that many real-world problem domains are composed of sub-problems that can be solved individually, and combined (often in a non-trivial way) to provide a solution for the larger problem [15, 16].

In some problem instances, such as large instances of the even parity problem, it is nearly impossible to stochastically arrive at a complete solution without utilizing generalized solutions for small instances of the problem [17]. It is simple to evolve a function that performs even parity on 2 bits using only the logical functions AND, OR and NOT as primitives, but extremely difficult to evolve a 10-bit even parity function without any *a priori* information as the space of all possible solutions is immensely larger, and even the best known solution is complex. By simply defining the general 2-bit XOR function (the even parity computation for 2 bits), the optimization method has a higher probability of combining instances of XOR to arrive at an n-bit even-parity function, greatly accelerating the optimization process.

In playing the game of Go, humans start at the top analyzing strategy and the total situation, solving a successive sequence of smaller, tractable problems to arrive at a

move. However, the learning process is bottom-up - a human player of Go first learns the legal moves of every piece, and then combines those general move capabilities into tactics, tactics into strategies and those strategies combine with the strategies of the opposing player to form a high-level view of the game and players as a whole. At each level, optimization and generalization are performed to pass information up and down the play hierarchy. This natural progression is not reflected in the methods that we utilize to computationally approach problems of this scale. The typical approach is combinatorial optimization, where a sequence of low-level moves is statistically analyzed in order to arrive at a plan of play. As a whole, this is a computationally intractable problem, and it does not even come close to resembling the way humans interact with problems of this type [18-22]. Additionally, the skills learned in Go may translate across several domains as general problem solving skills. The ability to translate knowledge from one domain to another implies the necessity of meta-learning or learning about how or what to learn – in order to recognize similar problem features in disparate environments and scenarios.

2. MEMETIC COMPUTING

Both Darwinian evolution and memetics have been sources of inspiration for classes of algorithms for problem-solving techniques with memetic algorithms being the most prominent and direct manifestation of the inspiration. In recent years, there has been a marked increase in research interests and activities in the field of Memetic Algorithms (MA). The first generation of MA refers to hybrid algorithms, the combination of population-based global search (often in the form of an evolutionary algorithm) with a cultural evolutionary stage. The first generation of MA, though it encompasses characteristics of cultural evolution (in the form of local refinement) in the search cycle, may not qualify as a true evolving system according to Universal Darwinism, since all the core principles of inheritance/memetic transmission, variation and selection are missing [23]. This suggests why the term MA stirred up criticisms and controversies among researchers when first introduced [24]. The typical design issues include i) how often should individual learning be applied, ii) on which solutions should individual learning be used, iii) how long should individual learning be run, iv) what maximum computational budget to allocate for individual learning, and v) what individual learning method or meme should be used for a particular problem, sub-problem or individual [25].

Multi-meme [26], hyper-heuristic [27] and meta-Lamarckian MA [5, 13] are referred to as second generation MA exhibiting the principles of memetic transmission and selection in their design [28]. In multi-meme MA, the memetic material is encoded as part of the genotype. Subsequently, the decoded meme of each respective individual is then used to perform a local refinement. The memetic material is then transmitted

through a simple inheritance mechanism from parent to offspring. On the other hand, in hyper-heuristic and meta-Lamarckian MA, the pool of candidate memes considered will compete, based on their past merits in generating local improvements through a reward mechanism, deciding on which meme to be selected to proceed for future local refinements. A meme having higher rewards will have greater chances of being replicated or copied subsequently. For a review on second generation MA, i.e., MA considering multiple individual learning methods within an evolutionary system, the reader is referred to [13]. Co-evolution and self-generation MAs introduced in [29] and [30] are described in [28] as 3rd generation MA where all three principles satisfying the definitions of a basic evolving system have been considered. In contrast to 2nd generation MA which assumes the pool of memes to be used is known *a priori*, a rule-based representation of local search is co-adapted alongside candidate solutions within the evolutionary system, thus capturing regular repeated features or patterns in the problem space.

From the three classes of MA outlined, memes can be seen as mechanisms that capture the essence of knowledge in the form of procedures that affect the transition of solutions during a search. The level of participation or activation of memes is typically dictated by certain indicative performance metrics, the objective being to achieve a healthy balance between local and global search. Memes instead of being performance-driven should be extended to include capacity to evolve based on the snapshots of problem instances. In the process of solving a repertoire of problem instances, memes can culminate based on the recurrence of patterns or structures. From basic patterns or structures, more complex higher level structures can arise. In this regard, a brain inspired meta-learning memetic computational system, consisting of an optimizer, a memory, a

selection mechanism, and a generalization mechanism that conceptualizes memes not just within the scope of a problem instance, but rather in a more generic contextual scope is appropriate. Such traits which are lacking in the 3rd generation MA can serve as the basis of the 4th generation class of MAs. The reader is referred to Table 2.1 for a summary of generational description of MAs. The summary, although by no means exhaustive, should serve as a useful guide on the classifications of the various traits of existing MA research.

The mammalian brain exhibits hierarchical self-similarity, where neurons, groups of neurons, regions of the brain, and even whole lobes of the brain are connected laterally and hierarchically. Biological neurons are particularly well suited to this architecture: a single neuron serves as both a selection and learning mechanism. A neuron only fires (passing a signal) when it receives significant input from one or more sources, and thus serves as a correlation detector. Additionally, it learns by modifying the weights of its inputs based on local information from firing rate, as well as global information from the chemical environment. Neurons activate when they encounter patterns that have made them fire before, and are able to adapt in delayed-reward situations due to global signals.

In laterally connected architectures, neuron groups can provide the function of clustering, as active neurons suppress the activity of their neighbors to pass their information down the processing chain, providing both selection and routing of information. The effect of this selectivity is that biological neural architectures route a spreading front of activation to different down-stream networks based on the similarity of the features present in the pattern of activation to previously presented patterns.

Table 2.1. Generational descriptions of memetic algorithms.

Generation	Characteristics	Example systems
1st	Global Search Paired with Local Search.	i) A Canonical MA [24, 31] ii) Adaptive global/local search [32] iii) MA for Combinatorial Optimization [33] iv) Evolutionary Gradient search [34] v) Large-Scale Quadratic Assignment Problem [35] vi) Evolutionary Lin-Kernighan for Traveling Salesman Problem [36] vii) Dynamic Optimization Problem [37] and many others.
2nd	Global Search with Multiple Local Optimizers. Memetic Information (Choice of Optimizer) Passed to Offspring. (Lamarckian Evolution)	i) Hyper-heuristic MA [27, 38] ii) Meta-Lamarckian MA [5] iii) Multimeme MA [7] iv) Adaptive Multi-Meme MA [13] v) Agent-based Memetic Algorithm [39, 40] vi) Diffusion Memetic Algorithm [28] and several others.
3rd	Global Search with Multiple Local Optimizers.	i) Co-evolution MA [30] ii) Self-generation MA [29]
4th	Mechanisms of Recognition, Generalization, Optimization, and Memory are utilized.	Unknown

As the activation front passes each neuron, the synaptic weights are changed based on local information – the firing rate of the neuron, the chemical environment, and the features present in the signal that activated the neuron, slightly changing how an individual neuron will respond at the next presentation of patterns [41].

Connected in loops, neurons provide short-term memory, process control and create temporally-delayed clustering. Combining loops and lateral connections at several levels of neuron groups (groups of neurons, groups of groups, etc) the neural architecture

is able to exhibit increasing levels of selection, memory, and control. This is exactly the architecture that we see in the human cortex – a single cortical column contains recursion and lateral inhibition, and these cortical columns are arranged in a similar way, progressing in a fractal learning architecture up to the level of lobes, where sections of the brain are physically separated [42]. This fractal architecture is similar to the *Nth*-order meta-learning architecture described later in Section 4.

The brain inspired meta-learning memetic computational system is thus regarded here as a 4th generation memetic computational system. The novelty of the proposed meta-learning memetic system is highlighted in the following list.

i. In contrast to the 2nd generation memetic algorithms, there is no need to pre-define a pool of memes that will be used to refine the search. Instead memes are learned automatically - they are generalized information that passed between problem instances.

ii. Since it satisfies all the three basic principles of an evolving system, it also qualifies as a 3rd generation memetic computational system. Unlike simple rule-based representation of meme used in co-evolution and self-generation MAs, the meta-learning memetic computational system models the human brain, encoding each meme as hierarchies of cortical neurons [42]. With a self-organizing cortical architecture, meaningful information from recurring real-world patterns can be captured automatically and expressed in hierarchical nested relationships. A human brain stimulated by the recurrence of patterns, builds bidirectional hierarchical structures upward. The structure starts from the sensory neurons,

through levels of cortical nodes and back down towards muscle activating neurons.

iii. There exists a memory component to store the system's generalized patterns or structures of previously encountered problems - these elements could be thought of as memes.

iv. Selection mechanisms are provided to perform association between problem features and previously generalized patterns that are likely to yield high-quality results.

v. Meta-learning about the characteristics of the problem is introduced to construct meta-memes which are stored in the selection mechanism, allowing higher-order learning to occur automatically.

vi. Memes and meta-memes in computing are conceptualized for higher-order learning as opposed to the typical definition of local search method used in all the works on MAs.

A genetic algorithm learns by passing schema (the genetic information of individuals) from generation to generation. Through natural selection and reproduction, useful schemata proliferate and are refined through genetic operators. The central concept of learning is that of the schema – a unit of information that is developed through a learning process [43-45]. The typical 'memetic algorithm' uses an additional mechanism to modify schemata during an individual's 'lifetime,' taken as the period of evaluation from the point of view of a genetic algorithm, and that refinement is able to be passed on to an individual's descendants. The concept of schemata being passable just as

behaviors or thoughts are passed on is what this document defines as memes – a meme being a unit of cultural information [5, 13, 46, 47].

Some parallels may be drawn with Learning Classifier Systems (LCS) where schemata are evolved explicitly, and individuals compete for territory in schema-space rather than explicitly with each other for survival – thus a population is evolved as a whole, rather than individuals. In this context, individuals represent a schema, with the population as a whole becoming a meme, storing the valuable accumulated experience over the evolutionary process.

Memes can be thought of as an extension of schemata – schemata that are modified and passed on over a learning process. However, this distinction is a matter of scale. In a learning method, the current content of the representation could be called a schema, but when that information is passed between methods, it is more appropriately regarded as a meme.

This is analogous to the sociological definition of a meme [48]. In this form, a meme may contain certain food preparation practices, how to build a home or which side of the road to drive on. Within the individuals of a generation, they are relatively fixed, but they are the result of a great deal of optimization, capturing the adaptations resulting from the history of a society. These cultural memes are passed from generation to generation of the population, being slightly refined at each step – new ingredients are added to the cooking methods, new building materials influence construction, traffic rules change, etc. The mechanism that allows this transformation is that of generalization [49-51]. To communicate an internal schema from one individual to another, it must be generalized into a common representation – that of language in the case of human

society. The specifics of the schema are of no great importance, as they would mean very little to an individual other than the originator due to the inherent differences between individuals. For instance, a description of the precise movements necessary to create a salad, such as the technique used to slice tomatoes and wash lettuce, is less important than the ingredients and general process of preparing the salad. The salad recipe is a meme, a generalized representation of the salad, but the recipe alone is insufficient to produce the salad. The salad recipe is expressed only when it is put through the process of preparation, of acquiring and preparing the individual ingredients, and combining them according to the salad meme.

A meme may be thought of as generalized schema. Schemata are refined for an instance; memes are generalized to the extent of being transmissible between problem instances. To resolve the potential confusion that may arise, "Memetic Computation" is loosely defined as a paradigm of computational problem-solving that encompasses the construction of a comprehensive set of memes thus extending the capability of an optimizer to quickly derive a solution to a specific problem by refining existing general solutions, rather than needing to rediscover solutions in every instance.

3. A FRAMEWORK FOR HIGHER ORDER LEARNING

A meta-learning system should be composed of four primary components – an optimizer, a memory, a selection mechanism, and a generalization mechanism, shown in Figure 3.1. The selection mechanism takes the features of a given problem as input, and performs a mapping to solutions in the memory that have an expected high quality. The memory stores previous or generalized solutions encountered by the system, and passes selected solution(s) on to the optimizer. The optimizer performs specialization and modification of solutions to optimize a given specific problem instance, while the generalization mechanism compares the resultant solution with existing solutions in memory, and either adds a new solution or modifies an existing solution. In memetic computation terms, the optimizer generates schema or modifies memes into schema, and then the generalization mechanism converts the schema back into memes for storage in memory. The selection mechanism provides a mapping on memes, providing recognition from a problem specification to a likely useful general solution, effectively utilizing internally represented meta-memes.

With these components, the architecture should be capable of exploiting information gained in previous problem sessions towards the solution of problems of increasing complexity. Integrating a cross-instance memory and a selection mechanism with an optimization method allows the recognition of a situation and the selection of previously utilized schema as likely high quality solution candidates. The optimization process then combines and refines these solution candidates to provide a good solution much faster than if the method had only random initial solutions. Once the solution is

deployed, the selection method is trained to associate the situation (stimulus) with the solution (behavior) utilizing the fitness (reward) of the solution.

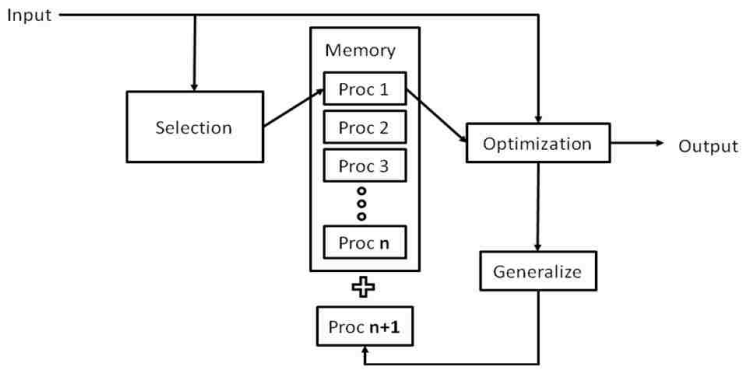


Figure 3.1. Meta-learning architecture.

The meta-learning process is itself a learning process, and thus could be augmented with increasingly higher level memory and selection methods, to allow complex, high-order solutions to be found. A sort of fractal meta-learning architecture of this type is expected to work well across a wide variety of real-world problems.

The sequence of learning sessions matters greatly to the expression of complex behavior. By starting with simple problem instances and presenting successively more complex scenarios, the problem is decomposed, allowing solutions from sub-problems to be exploited, increasing the likelihood that higher level solutions will occur. Additionally, by training these simple solution components, a wider variety of high-level solutions can be trained more rapidly. For example, when training a dog, teaching him to 'sit' decreases the amount of training necessary for both 'stay' and 'beg' behaviors. This is analogous to the automatic construction of a 'Society of Mind' as described by [52].

When constructing optimization architectures, an issue of particular relevance is that of representation – how the schemata are stored. In genetic algorithms schemata are stored as strings, while in neural networks schemata are implicitly represented as interconnection weights, clustering methods store templates for categories, etc. How these schemata are expressed (and thereby their meaning) is dependent on the expression structure. In genetic algorithms a string is decoded into a trial problem solution, while the weights in neural networks are utilized through weighted summation and passing through a transfer function. This division of representation prevents the simple utilization of schema across solution methods. To get disparate methods to work together, great care must be taken to modify all methods to utilize the same schema, which has been the subject of a great deal of research [5, 53-61].

First order learning methods consist of a single algorithm that modifies schema to optimize a system. Individually, all classical machine learning methods fall into this category. Meta-learning or second-order methods learn about the process of learning, and modify the learning method, which in turn modifies schema. A simple illustration of a meta-learning architecture is presented in Figure 3.1. In this figure, schemata are represented as "procedures", which are stored in memory. A problem is presented to the architecture, and a selection mechanism chooses likely valuable schema from memory, which are then modified to the particular problem instance. High-value schema are then generalized and stored, the selection mechanism then learns an association between characteristics of the problem instance and schema that yielded positive results.

These second order methods should be able to be combined with other methods or layers to produce third-order methods and so on to order *N*, as illustrated in Figure 3.2.

To produce higher order methods, information gained in one problem instance should be utilized to provide a partial solution to another similar problem instance allowing the system as a whole to take advantage of previous learning episodes.

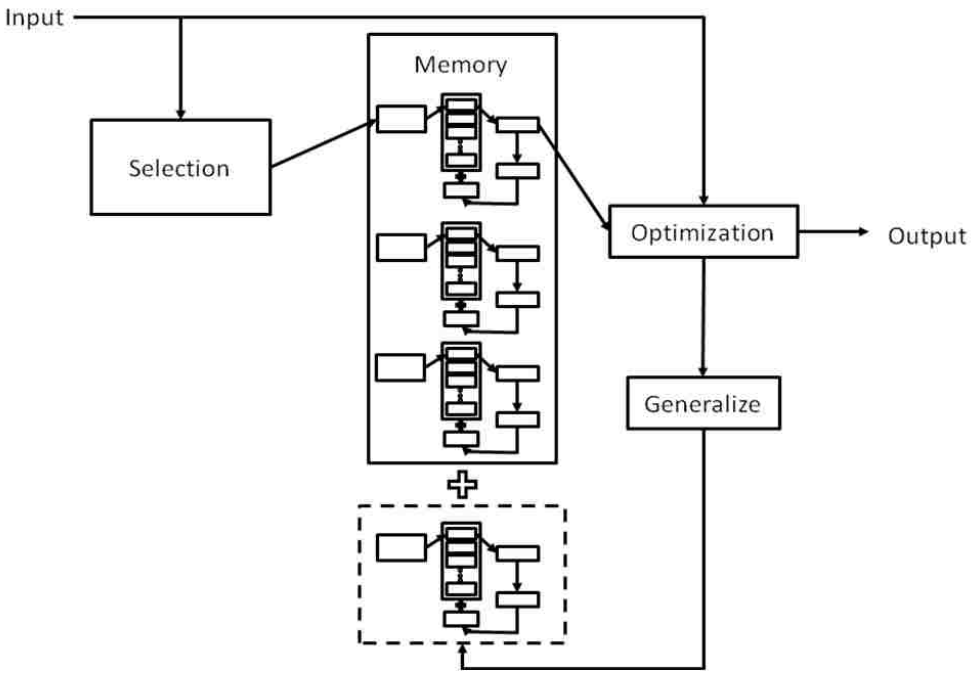


Figure 3.2. Meta-meta learning.

4. GRAMMATICAL ADAPTIVE RESONANCE THEORY

A new Adaptive Resonance Theory variant is presented that is capable of clustering variable dimension semantic inputs by creating templates that store a non-parametric distribution over the symbols and structure of a given grammar. Originally created as an automatic function definition mechanism for Genetic Programming architectures, the Gram-ART method has many other useful applications and properties. The variable cluster geometry of Gram-ART is demonstrated on a 2D clustering task. Gram-ART performance is shown to be improved compared to that of Fuzzy-ART and K-means on the benchmark IRIS and mushroom data-sets. The classification properties of Gram-ART are explored using the UNIX Users identification problem. Gram-ART demonstrates superior performance on all of these benchmarks.

The Adaptive Resonance Theory (ART) unsupervised learning method has long been a state of the art clustering tool due to its low run-time complexity and ability to scale the number of clusters that represent a data-set via a single parameter. Additionally, the seminal ART1 architecture [62] has been the subject of many research modifications, resulting in the development of Fuzzy-ART [63], Gaussian ART [64], Category Theory ART [65], and numerous others.

Genetic Programming (GP) is a rapidly growing field with increasingly valuable application to a number of important areas [66]. While this evolutionary algorithm is able to efficiently generate solutions to many problems which significantly outpace those devised by human experts, there are issues of computational cost to be addressed. In particular, this document investigates a class of GP's which tend to produce function trees

of such magnitude that the approach is rendered less effective in a very short order. The Gram-ART clustering algorithm is introduced to intelligently and dynamically adjust the size of the GP function tree in a way to satisfy the dual criteria of efficacy and computability. Designed to operate within the context of a Backus Naur Form (BNF) grammar, Gram-ART is capable of clustering variable-dimension inputs. The algorithm is based on the neural cognitive model known as Adaptive Resonance Theory, and it is the first such ART-based architecture to address variable-dimension symbolic inputs.

4.1. GENETIC PROGRAMMING

In GP, the genome of an individual is represented as a tree structure, where operations are applied at branches, leaves are constants and problem parameters [17, 67]. One advantage of GP is that the results can be easily interpreted by humans and formally verified, a quality that is not present in many other computational intelligence methods [50].

There has been some development of methods to generalize function blocks (branches in an individual's genome) that appear similarly and usefully across individuals and across generations, making those blocks available as fundamental components in the next generation of programs [17, 44, 66-69]. In this way, a library of functions are generated and customized in a meta-evolutionary way. This modification leads to greatly increased performance and reuse of structures allowing the algorithm to find solutions that it would have very little chance of finding otherwise. Additionally, by creating function blocks and removing parts of an individual's genome from active evolutionary modification, the probability of high-level architectural changes increases, as the genome is effectively shortened, and changes are only allowed on parts of the genome that have a

higher-level effect. In this way the evolutionary process starts by building and stabilizing low-level functionality, which grows to higher-level functions that exploit it. The result is a progressive, fitness-driven increase in program complexity that massively accelerates how well GP performs both in terms of quality and speed.

To simplify the dynamic function definition, a BNF grammatical definition is used to specify the set of all functions, variables, and their structural relationships. The BNF grammar is a way of expressing a language in the form of production rules. A BNF grammar consists of the tuple {N,T,P,S}, where N is the set of non-terminals such as <expr>, <op>, <preop>, corresponding to expressions, binary operators and unary operators, respectively. T is the set of terminals, such as operation symbols AND, OR and NOT. P is the set of production rules that map from N to T, and S is a seed symbol which is a member of N. An example of a simple binary BNF grammar is shown in Table 4.1.

Table 4.1. A simple binary BNF grammar.

```
N = {expr, op, pre_op, var}
T = {AND, OR, NOT, X, Y}
S = <expr>
P can be represented as:
  1. <expr> ::= <expr> <op> <expr> |
                <preop> <expr> |
                <var>
  2. <op> ::= AND | OR
  3. <pre_op> ::= NOT
  4. <var> ::= X | Y
```

Using this encoding style, grammars of arbitrary and dynamic complexity can be implemented, including the grammars of compilable languages and arbitrary functions. Note that the grammar does not specify the values that X and Y may take, nor does it

describe the conditions necessary to satisfy the meanings of AND, OR and NOT. The BNF structure merely describes the structure of the grammar, and how symbols relate to one another.

A key aspect of the GP process is defining the functions produced through the evolutionary process. Koza's early attempts at function definition utilized a rigid structure where the number of functions and arguments are fixed [70]. This limits the flexibility of the defined function and limits the complexity of evolved programs. Later attempts utilized the differential fitness of the population to determine when functions should be created. This leads to a large number of possibly similar functions, with any given function having a small chance of being selected.

To automatically generalize useful functions, it is proposed that a clustering method be utilized with differential fitness selection. The parameters of the clustering method are tuned to control the number and coarseness of functions generated, providing a simple mechanism for automatic function definition. As categories are generated on-line, the templates from each category are added to the grammar as new functions, and the GP process can then take advantage of these new elements.

For this purpose, a new clustering algorithm based on Adaptive Resonance Theory is developed that is able to utilize a variable-dimensionality representation to encode categories against a specified grammar. Currently no Adaptive Resonance Theory based - clustering method exists that is able to handle symbolic trees or variable-length representations. This new algorithm is called Grammatical Adaptive Resonance Theory, or Gram-ART.

4.2. ADAPTIVE RESONANCE THEORY

Adaptive Resonance Theory (ART) was developed by Carpenter and Grossberg as a solution to the plasticity and stability dilemma, i.e., how adaptable (plastic) should a learning system be so that it does not suffer from catastrophic forgetting of previously-learned rules (stability) [62, 63, 71]. ART can learn arbitrary input patterns in a stable, fast, and self-organizing way, thus overcoming the effect of learning instability that plagues many other competitive networks. ART is not, as is popularly imagined, a neural network architecture. It is a learning theory hypothesizing that resonance in neural circuits can trigger fast learning [72]. ART is distinguished by its use of resonance as a learning mechanism.

Adaptive Resonance Theory exhibits theoretically rigorous properties desired by neuroscientists which solved some of the major difficulties faced by modelers in the field. Chief among these properties is stability under incremental learning. In fact, it is this property which translates well to the computational domain and gives the ART1 clustering algorithm, the flavor of ART most faithful to the underlying differential Equation model, its high status among unsupervised learning algorithm researchers. At its heart, the ART1 algorithm relies on calculating a fitness level between an input and available categories.

What fundamentally differentiates ART1 from similar distance-based clustering algorithms is a second fitness calculation whereby a given category can reject the inclusion of an input if the input does not meet the category's standards as governed by a single global parameter. Cognitively, this models the brain's generation and storage of expectations in response to neuronal stimulation. The initial fitness, measuring the degree to which each input fits each of the established categories, is considered a short-term

memory trace which excites a top-down expectation from long-term memory. Computationally, this second fitness calculation acts to tune the number of categories, and it may force the creation of new categories where a k-means styled algorithm would not, thus exhibiting stronger, more nuanced, classification potential. The ART1 algorithm has enjoyed great popularity in a number of practical application areas of engineering interest. Its chief drawback is the requirement that input vectors be binary. The ART2 algorithm was first proposed to address this restriction, but in practice today it is the Fuzzy-ART modification of ART1 which powers most of the new ART research and applications.

Fuzzy-ART admits input vectors with elements in the range [0,1]. Typically a sort of preprocessing called complement coding is applied to the input vectors as well as any normalization required, mapping the data to the specified range. The Fuzzy-ART's core fitness Equations take a different form than those of ART1, leveraging the mechanics of fuzzy logic to accommodate analogue data vectors. Researchers have concocted a wide variety of ART-based architectures by modifying the fitness Equations to specialize them for a given problem domain.

For example, Gaussian ARTMAP uses the normal distribution to partition categories, with the relevant fitness Equations incorporating the Gaussian kernel. This parametric statistical approach to ART was the first in what has become a rich field of study. Other parametric methods incorporate different probability distributions or allow for alternative preprocessing schemes based on statistics. The Gram-ART architecture presented in this paper extends this body of knowledge by exploring non-parametric statistical methods for category determination.

Parametric statistics assume much about the underlying distribution of the inputs to the system. In running a standard t-test, for example, it is required that the data be generated by Gaussians or have a sufficient quantity of data to ensure the sampling distribution is normal. It is often the case in practice that such normality assumptions are invalid. Gram-ART adds to the existing probabilistic ART architectures in that it makes no such assumptions regarding the distribution of inputs (as compared to, for example, Gaussian ARTMAP.) Instead, it relies on non-parametric, or distribution-free, statistical models of the inputs when making its classifications. This allows Gram-ART to effectively handle data from small samples or about whose structure nothing is known. The interested reader is directed to [73] for further details regarding non-parametric statistical analysis.

Other specializations of ART include ARTMAP-IC [74] which allows for input data to be inconsistently labeled and is shown to work well on medical databases, Ellipsoidal ARTMAP [75] which calculates elliptical category regions and produces superior results to methods based on hyper-rectangles in a number of problem domains, and a version of ART which uses category theory to better model the storage and organization of internal knowledge [65]. Overall, Adaptive Resonance Theory enjoys much attention by those studying computational learning for both scientific and engineering purposes.

Fuzzy-ART incorporates fuzzy set theory into ART and extends the ART family by being capable of learning stable recognition clusters in response to both binary and real-valued input patterns with either fast or slow learning.

Layers F1 and F2 are connected via adaptive weights W. The orienting subsystem is controlled by the vigilance parameter ρ. The basic FA architecture consists of two-layer nodes or neurons, the feature representation field F1, and the category representation field F2, as shown in Figure 4.1. The neurons in layer F1 are activated by the input pattern, while the prototypes of the formed clusters, represented by hyper-rectangles, are stored in layer F2. The neurons in layer F2 that are already being used as representations of input patterns are said to be committed. Correspondingly, the uncommitted neuron encodes no input patterns. The two layers are connected via adaptive weights, W^j, emanating from node *j* in layer F2. After layer F2 is activated according to the winner-take-all competition between a certain number of committed neurons and one uncommitted neuron, an expectation is reflected in layer F1 and compared with the input pattern. The orienting subsystem with the pre-specified vigilance parameter ρ $(0 < \rho \leq 1)$ determines whether the expectation and the input pattern are closely matched. If the match meets the vigilance criterion, learning occurs and the weights are updated. This state is called resonance, which suggests the name of ART. On the other hand, if the vigilance criterion is not met, a reset signal is sent back to layer F2 to disable the current winning neuron for the entire duration of the presentation of this input pattern, and a new competition is performed among the remaining neurons. This new expectation is then projected into layer F1, and this process repeats until the vigilance criterion is met. In the case where an uncommitted neuron is selected for coding, a new uncommitted neuron is created to represent a potential new cluster.

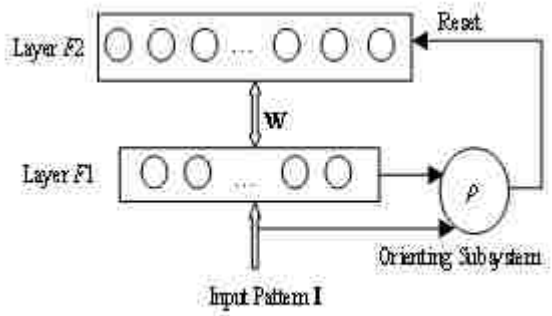


Figure 4.1. Topological structure of Fuzzy-ART.

Fuzzy-ART exhibits fast, stable, and transparent learning and atypical pattern detection. The Fuzzy-ART method has the benefit of being a highly efficient clustering method with a linear run-time complexity. Algorithmically, there are two steps to ART: category choice and vigilance test. Let *x* be the input, w^j the weights associated with category *j* and ρ be the vigilance. In category choice, the degree of match is calculated in Equation 1 for each category *j*. The vigilance test is calculated in Equation 2. The algorithm cycles between category choice and the vigilance test until resonance occurs and the winning weight is updated according to Equation 3. Fast learning occurs when $\beta = 1$.

$$T(j) = \frac{|x \vee w^j|}{|w^j|} \tag{1}$$

$$\frac{|x \vee w^j|}{|x|} \geq \rho \tag{2}$$

$$w_{(new)}^{j} = \beta\left(w_{(old)}^{j} \wedge x\right) + (1-\beta)w_{(old)}^{j} \quad (3)$$

4.3. GRAM-ART ALGORITHM

The Gram-ART algorithm is a specialization of ART designed to handle variable-length input patterns represented in a tree structure based on a BNF grammar. Let *x* be a tree under the grammar. Let w^j be a generalized tree corresponding to category *j*. Note here that the category representations in Gram-ART are themselves trees, thus abstracting the hyper-rectangular prototype forms of earlier manifestations of ART. Each node in the generalized tree has an array representing the distribution of possible symbols at that node. Here, *r* represents the number of nodes in a tree. Finally, let ρ represent the vigilance level.

To implement ART for trees, a measure of magnitude is necessary for inputs and weights. Since the size of the elements of these distributions do not correspond in a meaningful way to any sense of magnitude, the measure is defined to be simply the number of nodes present in each tree. That is, the tree-norm operator ||.|| is defined as ||*y*||= (the count of nodes in *y*). So, ||*x*|| = *n* and $||w^j|| = r$.

Initially, there are no category nodes committed. The first input vector is used to update, so no initial values of the weights need to be given. It is necessary to define a notion of overlap or intersection between the input tree and the category templates. It is not possible to use either the normal intersection operator or the fuzzy-AND operator

because x and w^j are not guaranteed to be of the same dimensionality. Therefore, the trace of x in w^j, denoted by $|x \cap w^j|$, is defined in Equation 4.

$$\left|x \cap w^j\right| = \sum_{i=0}^{r} w_{i,x_i}^{j} \tag{4}$$

The trace is the sum of the values stored in the weight corresponding to the symbols in a given input x. This has the effect of comparing root-aligned trees.

A Gram-ART tree is an ordered pair (N,R) where is N a set of nodes and R is a binary relation describing the structure of the tree. If x and y are nodes, then $xRy = 0$ if y is not a successor node of x. If $xRy = n$, for $n>0$, then this means that x is the successor node of y. The Gram-ART trees, unlike some graph theory trees, have an ordered succession.

Let $A = (N^A, R^A)$ and $B = (N^B, R^B)$ be Gram-ART trees. Then their intersection $A \cap B = (N', R')$ is defined as follows. The root node r is the same for all non-empty trees and therefore it is in N'. The rest of N' is built iteratively starting at the root. Consider all $x \in N^A$ and $y \in N^B$ such that $rRx = rRy$. These correspond to the same node and therefore this node is in the intersection N'. This process is repeated for each of these nodes until all the nodes in A and B have been so compared. The resulting intersection tree will then contain the nodes corresponding to the structural overlap between A and B.

An example is given in Figure 4.2. Given two trees, one for the function “X AND Y” and another for the function “NOT X.”

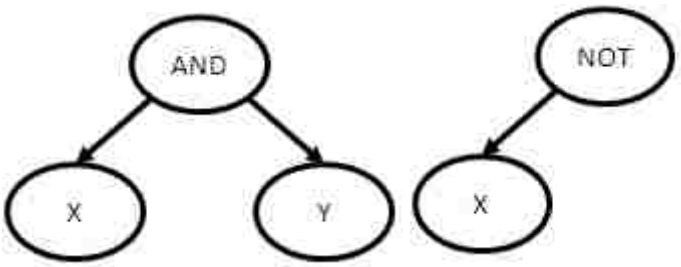


Figure 4.2. Example function trees for “X AND Y” on the left, and “NOT X” on the right.

To store the combination of the two, a type of prototype tree is created that holds a distribution over the symbols at each node, and has a variable number of children. This prototype tree is not bound by the rules of the BNF grammar, as each node is a superposition of nodes at a position that have been combined to construct the prototype. An example prototype node for the example binary grammar is shown in Figure 4.3. Note that the distribution for each of the symbols is initially zero. Combining the two trees in Figure 4.2 would result in the proto-tree shown in Figure 4.4.

Figure 4.3. Prototype tree node.

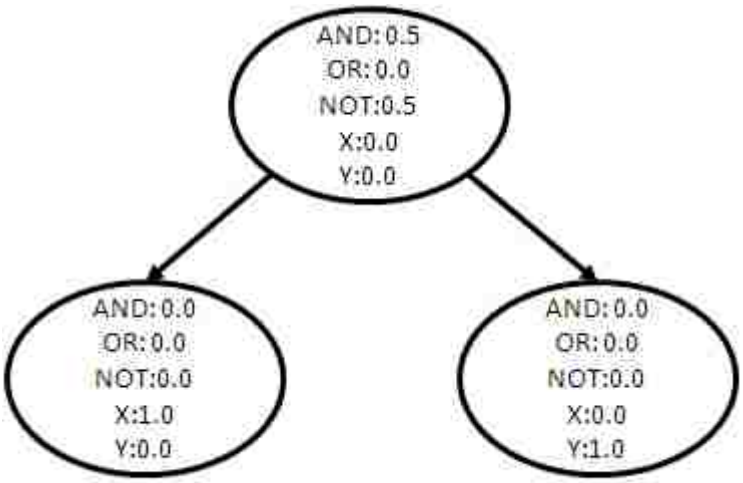


Figure 4.4. Prototype tree resulting from the combination of “X AND Y” and “NOT X” trees.

Note that the distribution at the root node is divided equally between the “AND” and “NOT” symbols. Also note that each of the child nodes has only one non-zero entry in the distribution - “X” and “Y” respectively. In the case of “X”, both trees have this symbol as the first child, updating twice, while “Y” is the child of only the “AND” tree, it is updated once.

Two node structures ProtoNode and TreeNode are outlined in Table 4.2 and Table 4.3. To compare and update trees, recursive functions that traverse both trees synchronously are outlined in Table 4.4 and Table 4.5. The ProtoNode structure is used to construct tree prototypes, which are the templates of Gram-ART. The ProtoNode holds a distribution over all symbols in a given grammar (Line 2) and the update counters for each symbol (Line 3), as well as an array of child-nodes (Line 4). The TreeNode structure, outlined in Table 4.3, holds a single symbol (Line 2) and an array of child-nodes (Line 3).

Table 4.2. Code listing for prototype node structure.

```
1  struct ProtoNode
2    double dist[];
3    int N[];
4    ProtoNode protochildren[];
5  end struct;
```

Table 4.3. Code listing for tree node structure.

```
1  struct TreeNode
2    Terminal t;
3    TreeNode children[];
4  end struct;
```

To find the trace of an input tree on a template tree, the trace process recursively descends the two trees, retrieving the distribution sample at each node that corresponds to the symbol in the current node. These distribution samples are summed over all corresponding nodes to complete the trace.

The Trace function, outlined in Table 4.4, performs the recursive process of comparing a tree with a prototype. The function first accumulates the probability of a tree's symbols occurring in the prototype (Line 2), then increments a counter that tracks the number of nodes that the trees have in common (Line 3). The function then recurses on each of the child nodes to accumulate statistics for the remainder of the tree.

Table 4.4. Trace process pseudo-code.

```
1  function Trace(TreeNode &A,
   ProtoNode &B,
   double &sum,
   double &size)
2    sum = sum + B.dist[A.t];
3    size = size + 1;
4    For each i in A.children[],
5      CompareNode(A.children[i],
               B.protochildren[i],
               sum,
               size);
6  end function;
```

4.3.1. Category Match. The first step in ART is to calculate the strength of the activations to the category nodes. We define this activation strength, or choice value, for category *j* as Equation 5.

$$T(j) = \frac{|x \cap w^j|}{||w^j||} \tag{5}$$

This quantity measures to what extent the input pattern *x* activates the category weight entries of w^j. If the elements of *x* correspond to all 1's in the rows of w^j then this is a perfect match with activation equal to 1. If the category w^j is nowhere close to the input *x* then the corresponding weight entries will be small so that the match approaches 0. Note that the template might contain more or fewer nodes than the input and this measure penalizes such mismatches. In the numerator, if the weight value does not exist to correspond to the input then the value does not get summed. In the denominator, the size of the weight is counted, lowering the resulting value if the trace has fewer entries.

4.3.2. Vigilance Test. Once $\boldsymbol{T^j}$ has been calculated for all categories, this vector is sorted and the highest category is checked for vigilance. The purpose of the vigilance test is to compare how accurately the chosen category can predict the value of the input *x*, so the following condition in Equation 6 is checked.

$$\frac{|x \cap w^j|}{||x||} \geq \rho \tag{6}$$

If this condition is satisfied, then resonance is said to occur and the weight update process is initiated. Otherwise, value T^j is reset and proceeds with the next highest category match. If none of the categories pass the vigilance test, then a new blank uncommitted node is assigned to the current input, and the input is used to initialize the new node.

4.3.3. Weight Update. Element update is a weighted sum of the frequency with which a given option has been presented and is calculated using Equation 7, where *N* is the number of updates at a node prior to the latest one and $\boldsymbol{\delta_j}$ is a characteristic function given by Equation 8.

$$w_i^j = \frac{w_i^j * N + \delta_j}{N + 1} \tag{7}$$

$$\delta_j = \left\{ \begin{array}{l} 1 \; if \; x_i = j \\ 0 \; otherwise \end{array} \right\} \tag{8}$$

The recursive process for updating w^j is described in Table 4.5. The function first updates the probability of a tree symbol occurring in that node location using Equation 7 (Line 2), then it increments the number of updates for that symbol (Line 3). The function then calls itself on each of the child nodes, recursively updating the rest of the tree.

Table 4.5. Weight update pseudo-code.

```
1  function UpdateNode( TreeNode &A,
                        ProtoNode &B)
2    B.dist[A.t] = NewWeight(B.dist, B.N);
3    B.N[A.t] = B.N + 1;
4    For each i in A.children[],
5      UpdateNode(A, B);
6  end function
```

An example of the effect of template updating process is illustrated in Figures 4.5-4.8. Three updates are applied to a template with two nodes, labeled X and Y. Each node holds a distribution over 5 symbols. The updates proceed with input pairs{X3, Y2}, {X4, Y4}, {X3, Y3}. The changes in the template distributions are shown by the shaded bars in Figure 4.5.

To further visualize template shapes, a two-dimensional activation plot is generated by calculating the category match for all possible combinations of symbols, and plotting the resulting values in a gray-scale grid, where each cell of the grid corresponds to one combination of symbols. In this visualization scheme, lighter color represents a higher activation, for example, white is fully activated, and black is zero activation.

Figures 4.6, 4.7, and 4.8 show the changes in activation patterns resulting from these updates, where brighter areas correspond to higher activation. These activation maps illustrate the effective cluster shapes. Note that the clusters have no strict geometry, as they are a non-parametric distribution over symbols.

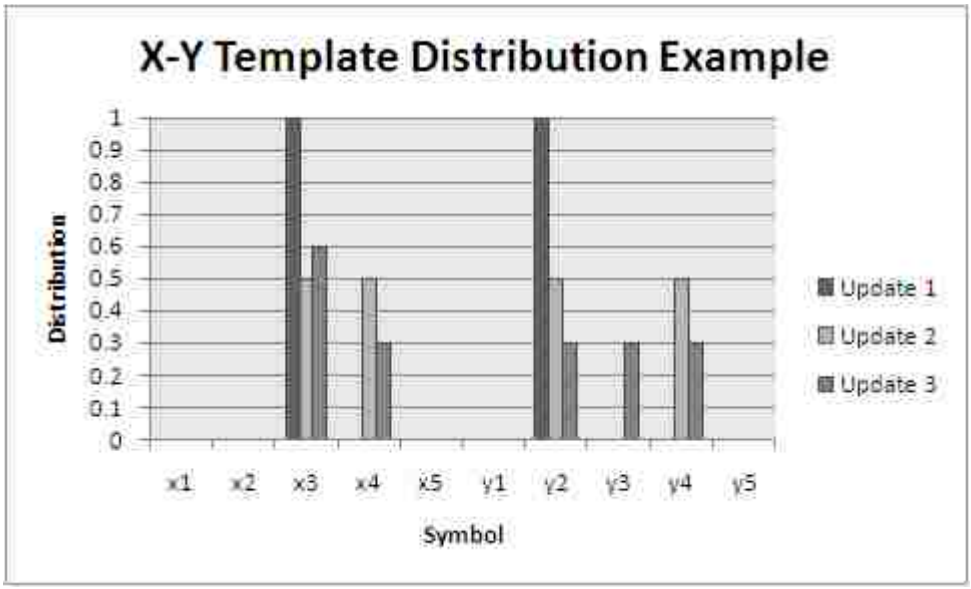


Figure 4.5. Template distribution changes across symbol updates for two nodes, “X” and “Y” on left and right, respectively, each with 5 possible symbols.

The first update {X3, Y2}, shown in black, initializes the distribution for the given symbols. The second update {X4, Y4}, shown in light gray, divides the distribution between X3, X4 and Y2, Y4 in the X and Y nodes respectively. The third update {X3,Y3}, shown in dark gray, increases the distribution at X3 due to the recurrence of the X3 symbol, also deceasing the distribution at X4. In the Y node, the distribution is again split between Y2, Y3, and Y4, as all of these symbols are equally likely to occur at this node, given the history of these three updates.

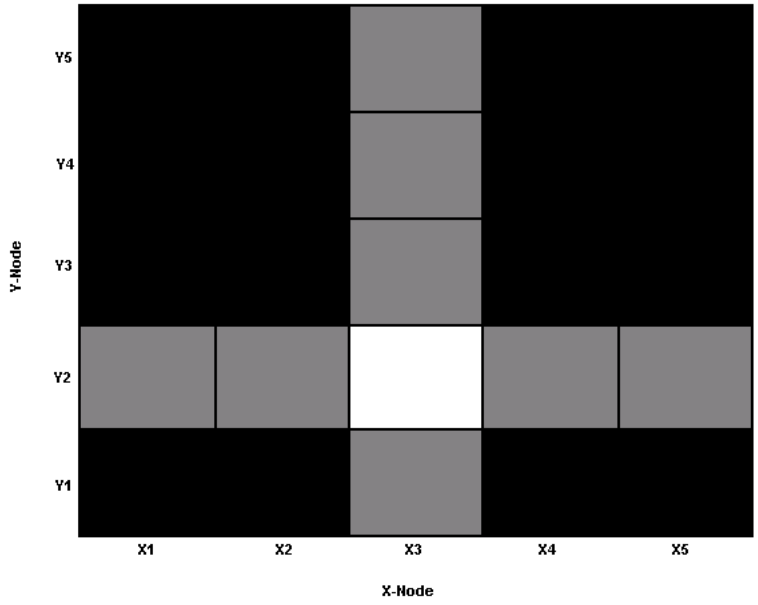


Figure 4.6. Activation plot for a two-node template after initial update of symbol pair {X3, Y2}. Lighter areas indicate locations of higher activation.

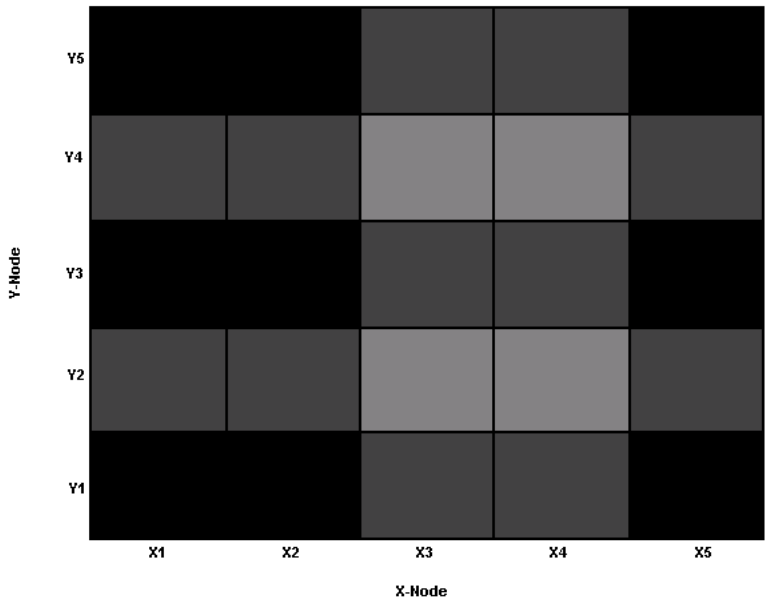


Figure 4.7. Activation plot of two-node template after update of symbol pairs {X3, Y2}, and {X4, Y4}.

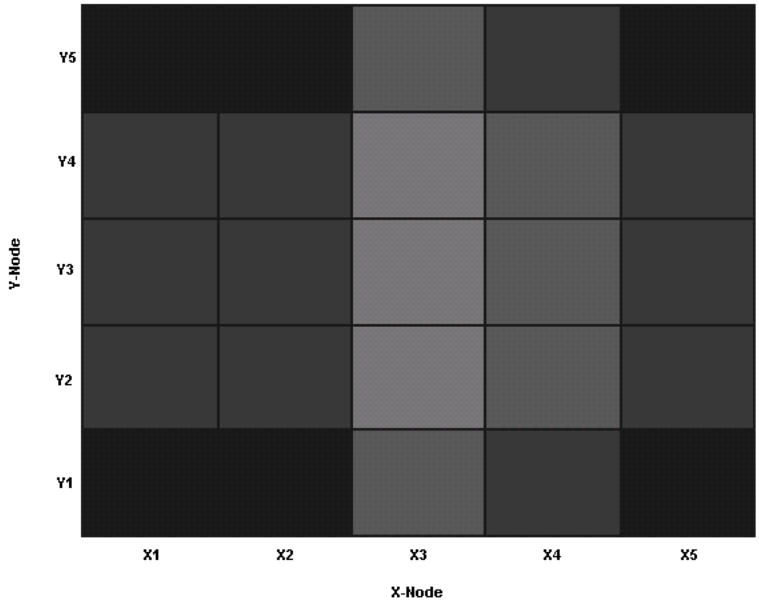


Figure 4.8. Activation plot of two-node template after updates of {X3, Y2}, {X4, Y4}, {X3, Y3}.

4.4. GRAM-ART PROPERTIES

The early papers on ART [76] contain analysis and proofs of various convergence and stability properties. These proofs focus on the dynamical system approach to understanding the network architectures; importantly, they do not reference the later algorithms designed to translate the theory into a computational tool for engineering applications. While it is true that some results, such as bounds on the sizes of category recognition regions, can derive from the algorithms of systems such as Fuzzy ART [63], the core properties of ART reveal themselves most readily when the system is formulated as differential Equations.

While much important research has been done in the areas of category region analysis for traditional geometric-cluster ART systems, the approach to analyze probabilistic ART algorithms, of which Gram-ART is one, necessarily must proceed

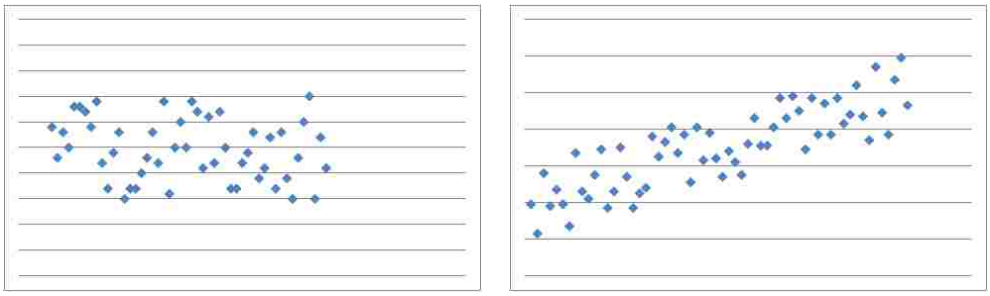
along a different axis. For example, in [77] it is shown that all Fuzzy ART templates are distinct and bounds are given on template size and the intersection of two templates given various conditions. In [78] a detailed analysis of cluster geometry is included for the purposes of determining when various input patterns have been learned. These results aid in the tuning of the system. Finally, [79] introduces new category structures based on the steps in the Fuzzy ART algorithm. These new regions are tied directly to the geometry of the category templates themselves, and the useful results necessitate the existence of such structure. While all these advances contribute greatly to the state-of-the-art in ART, they do not necessarily translate to probabilistic ART systems such as Gaussian ARTMAP and Gram-ART.

In Gaussian ARTMAP the templates are represented by the parameters for multidimensional Gaussian distributions—means, standard deviations, and counts. This probabilistic system is able to better handle noisy data without experiencing category proliferation. By moving beyond the limitations of a fixed geometry for the data clusters, the Gaussian ARTMAP system is capable of capturing data which would take several hyper-rectangles to cover. In particular, Gaussian ARTMAP is adept at clustering data with independent variance along each dimension. Where Gaussian ARTMAP requires more clusters is in the face of co-varying data. In these cases multiple Gaussian distributions are required to cover the data set, illustrated in Figure 4.9 and Figure 4.10.

Gram-ART, due to its non-parametric nature, does not share in this weakness for co-varying distributions [80]. Gram-ART requires more data to represent the category template than does Gaussian ARTMAP (which relies on the parameters of the Gaussian distribution), but the trade-off is that it is not limited by its own parameterization to

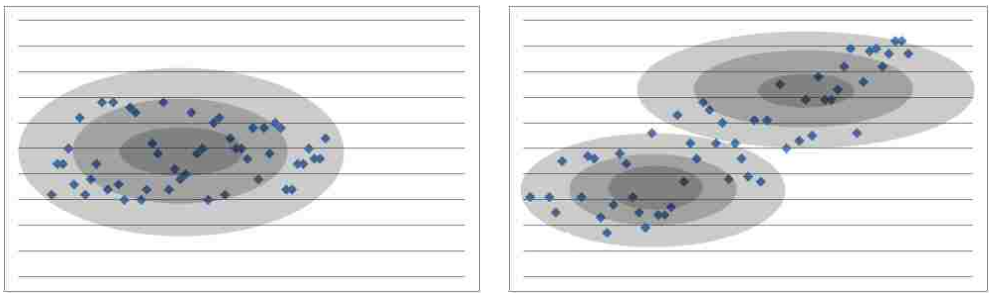
distributions satisfying certain constraints. Independent variations or co-variations are handled the same in Gram-ART, as its probability distribution category model can adapt to whatever structure the input set may present (Figure 4.11 and Figure 4.12). Note that Gaussian distributions cannot efficiently match co-varying data with only one distribution, as only the parameters μ and δ are modified. There is no analogue to rotation. Note that the match has much lower error, and this error is only tied to training samples and number of discrete columns used for representation.

Whereas Gram-ART may be able to represent a data-set using fewer clusters, the overall storage requirements of Gram-ART may be higher than that of other clustering methods. Additionally, discrete non-parametric templates are well-suited to symbolic representation, a property that very few clustering methods are able to handle naturally.



(a) (b)

Figure 4.9. 2D data with variance (a) and covariance (b).



(a) (b)

Figure 4.10. Gaussian best match to 2D varying (a) and co-varying data (b).

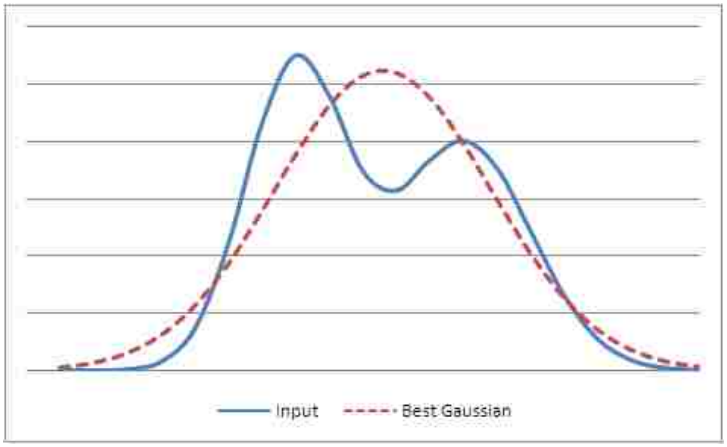


Figure 4.11. An example input and the best matching Gaussian distribution. Note the large differences between the input (solid line) and the Gaussian (dashed line).

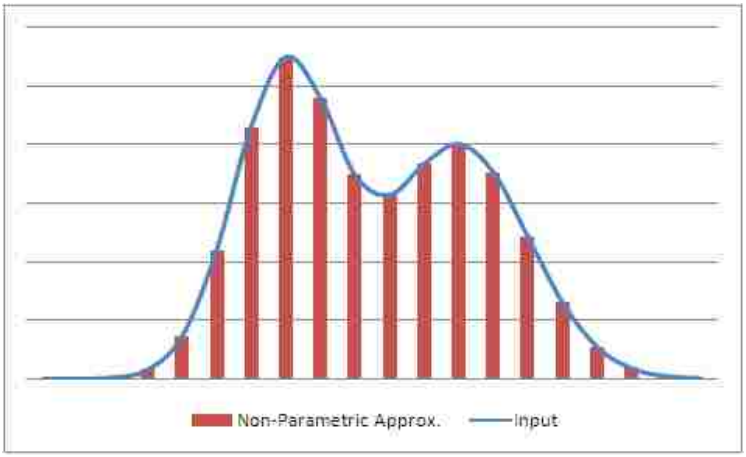


Figure 4.12. A non-parametric match to an example input.

4.4.1. Vigilance and Template Size. The Gram-ART system is constructed such that the degree of match decreases as the size difference between an input and a template increases. It is clear by examining the Gram-ART category match and vigilance match equations that there exists a maximum likelihood of update when two trees are the same size, independent of the values of the symbols that each tree holds.

Additionally, the interaction between the size of an input and the vigilance test produces a bound on the values that the vigilance value can take such that templates of sizes equal to the size of the input are able to be updated. For example, a tree with 1 node will never be updated after its initial creation, except by an identical input, so that the template can never change. For vigilance values above 0, the trace of an input on a template is either 1 or 0. A 1 corresponds to a perfect match, and the distribution is not changed. A 0 results from no match, and will not pass the vigilance test - again the template does not change. For a tree with 2 nodes, there are four possible match cases resulting in 3 possible match values, one case producing 0, 2 cases producing 0.5 and 1 case producing 1. Again, only the 0.5 match cases result in a change in weights, so vigilance has to be less than or equal to 0.5 for a 2-node tree to change. Similarly, a 3-node tree, 1 match in 3 nodes (0.33) produces a change in the template, as well as 2 in 3 (0.66...), so 0.66 is the max vigilance threshold.

These are the largest values the vigilance threshold can take before templates with certain numbers of nodes are never updated after they are initially created. Equations 9 and 10 describe the bounds that the vigilance value must take such that a template of a given non-zero size n may be updated.

$$\rho_{max} = \frac{n-1}{n} \tag{9}$$

$$\rho_{min} = \frac{1}{n} \tag{10}$$

4.4.2. Template Drift. Template drift is the phenomenon that occurs when a given category template is updated sufficiently that an input that was initially assigned to the category is no longer represented by the category template. Given the Gram-ART weight update Equation, holding the maximum change in weight for a single node it is trivial to derive Equation 11.

$$\Delta w_{max}^{ij} = \frac{1}{N+1} \tag{11}$$

It is clear to see from Equations 7, 8 and 11 that the Gram-ART algorithm is inherently an infinite window updating algorithm, and thus given infinite updates to a template, any template could be shaped to any value. This is a problem for any ART variant and is typically approached by repeated presentations of the same training set until the system reaches a stable state. In Gram-ART, the learning rate is implicitly controlled by the number of updates per node, and after the first update to a node, the maximum change in a weight is 0.5 or less, providing rapid convergence and increasing template stability.

4.4.3. Computational Complexity. Due to the excellent on-line learning properties of ART variants, their computational complexity is $\boldsymbol{O(n)}$.

As mentioned previously in the discussion on template drift, ART-based algorithms typically require several passes through a data set before templates stabilize, and Gram-ART is no exception. However, most ART-based methods utilize a fixed input size, where Gram-ART can handle variable length inputs. The space and computational requirements for search and update will necessarily be larger for certain data sets. Additionally, the Gram-ART weight update modifies every part of a distribution at a given node, but if an input and a template differ structurally, then only the nodes of the common structure are modified.

4.5. TWO-DIMENSIONAL CLUSTERING

To demonstrate arbitrary cluster geometries in Gram-ART, a data-set consisting of two-dimensional points sampled from two normal distributions is given as input, each distribution centered at the top-right and bottom left corners. To translate between a continuous 2D space and a symbolic grammar, the X and Y dimensions are evenly segmented into three symbols each, giving nine separate regions of activation. One hundred points were given as input with a vigilance value of 0.7. Two templates were produced, shown in Figure 4.13. Regions with high activation values are shown in bright areas while low activation values are shown in black.

Template 1 in Figure 4.13 corresponds to the bottom-left distribution, while Template 2 captures the top-right distribution. Nearby regions are also partially activated, illustrating the inherent arbitrary geometry of the clusters.

Arbitrary cluster geometries are further illustrated in a second experiment. The X and Y dimensions were segmented into ten symbols in each dimension which produced eleven clusters. These templates are each shown by bands of activation in the X and Y directions in Figure 4.14 and Figure 4.15. Several bands can be seen in each cluster. Many of these bands even have multiple peaks in them, indicating a complex relationship among the input data. The sample data is separable and Gram-ART is able to divide the data non-linearly into clusters. Additionally, the vigilance parameter controls how spread out the templates can become before a new template is allocated.

Template 1: Template 2:

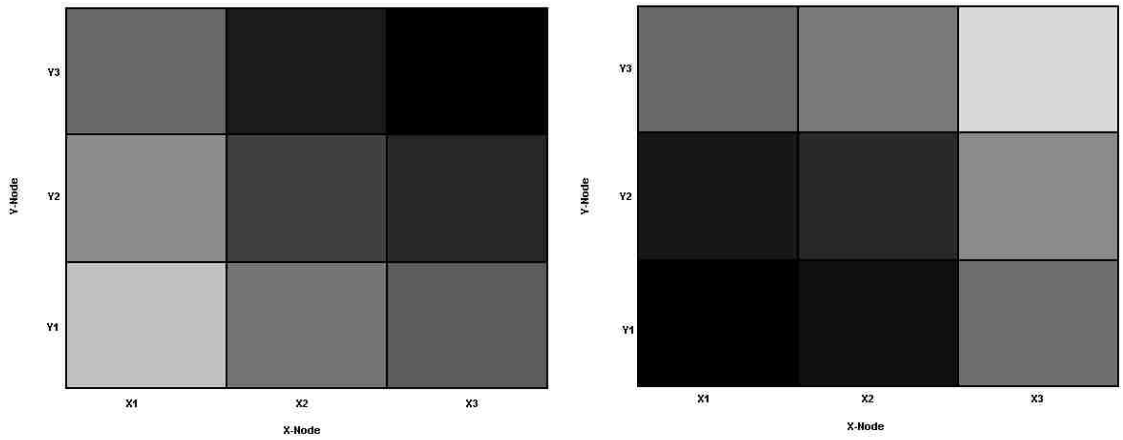
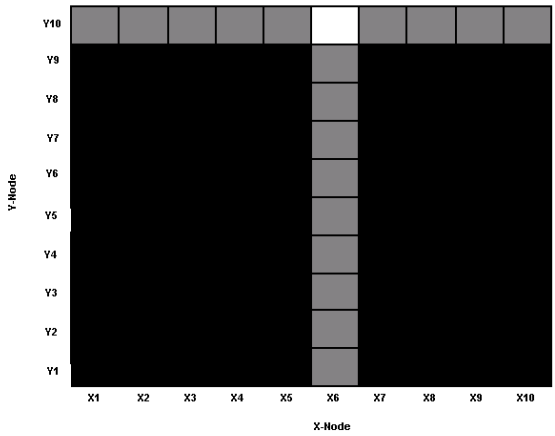
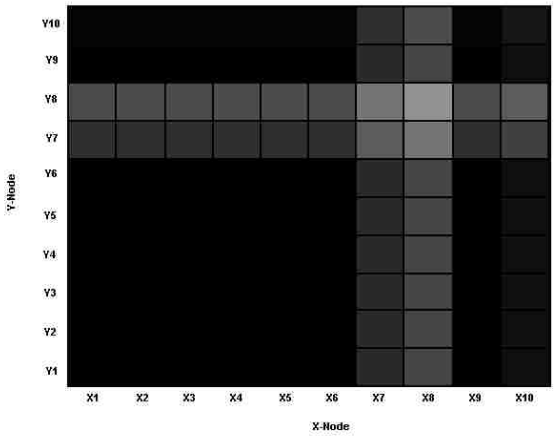


Figure 4.13. 3-Symbol 2D clustering activation plots of templates resulting from application of points sampled from two normal distributions, one centered in the bottom left (Template 1), and one centered in the top right (Template 2).

Template 1:



Template 2:



Template 3:

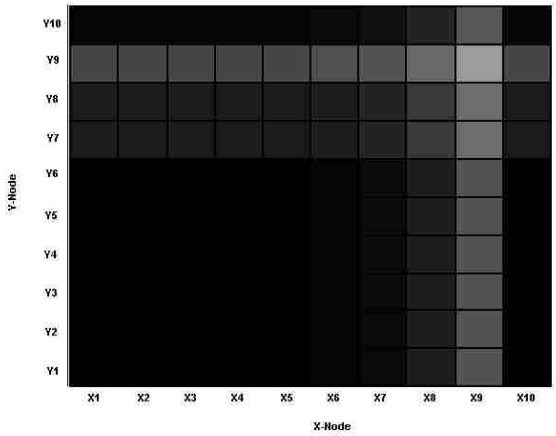
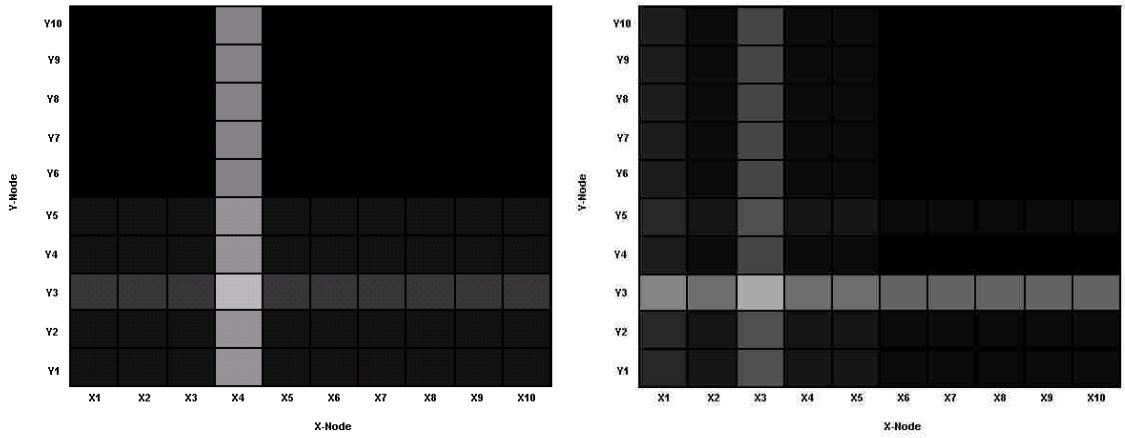


Figure 4.14. 10-symbol 2D clustering activation plots of templates resulting from application of points sampled from two normal distributions, centered in the top right (Templates 1-3).

Template 4: Template 5:



Template 6:

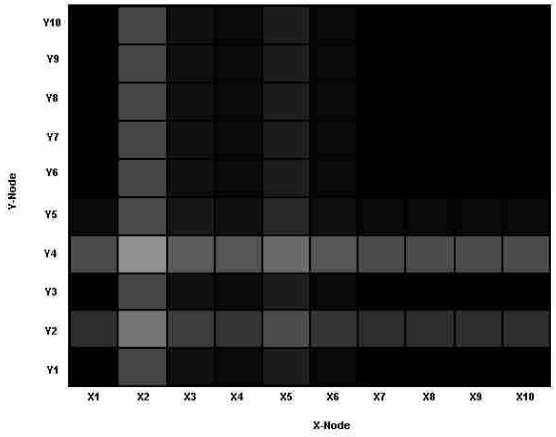


Figure 4.15. 10-symbol 2D clustering activation plots of templates resulting from application of points sampled from two normal distributions, centered in the bottom left (Templates 4-6).

4.6. IRIS DATA-SET

To evaluate the performance of the Gram-ART algorithm, the standard Fisher's IRIS data-set [81, 82] was used as a benchmark against the Fuzzy-ART and K-Means methods.

The input variables are translated into symbols by finding the max and min of each variable and then dividing into equal compartments. For instance, the Sepal Length variable has a min and max value of 2 and 4.4, respectively. Dividing this range into ten equal bins results in symbol SL1 with range 2 to 2.24, symbol SL2 with range 2.25 to 2.48, etc.

Each input variable is translated into a symbolic representation and input to Gram-ART for clustering. Combined with a fixed seed, the grammar is able to encode a fixed-length symbolic representation of the input data.

To evaluate Gram-ART on the IRIS data-set, the IRIS grammar was constructed, shown in Table 4.6.

Table 4.6. IRIS BNF grammar.

```
N = {SL, SW, PL, PW}

T = {SL1, SL2, SL3, SL4, SL5, SL6, SL7, SL8, SL9, SL10,
     SW1, SW2, SW3, SW4, SW5, SW6, SW7, SW8, SW9, SW10,
     PL1, PL2, PL3, PL4, PL5, PL6, PL7, PL8, PL9, PL10,
     PW1, PW2, PW3, PW4, PW5, PW6, PW7, PW8, PW9, PW10,}

S = <SL> <SW> <PL> <PW>

P can be represented as:
  1. <SL> ::=
    {SL1 | SL2 | SL3 | SL4 | SL5 |
     SL6 | SL7 | SL8 | SL9 | SL10}
  2. <SW> ::=
    {SW1 | SW2 | SW3 | SW4 | SW5 |
     SW6 | SW7 | SW8 | SW9 | SW10}
  3. <PL> ::=
    {PL1 | PL2 | PL3 | PL4 | PL5 |
     PL6 | PL7 | PL8 | PL9 | PL10}
  4. <PW> ::=
    {PW1 | PW2 | PW3 | PW4 | PW5 |
     PW6 | PW7 | PW8 | PW9 | PW10}
```

In order to compare Fuzzy-ART and Gram-ART it is necessary to explore the effects of discretization. The IRIS data-set was applied to the Fuzzy-ART method in two ways, first applying the raw IRIS data, and then by applying the discretizing method described previously, then substituting the symbol with the midpoint of the numerical range that the symbol represents. For example, SL2 represents the range 2.25 to 2.48, and after processing, corresponds to the value 2.364.

To evaluate the performance of each algorithm, the order of records in the data-set was randomized and applied to each method. This process was repeated 30 times per method, per parameter configuration. In the ART-based methods, the vigilance was varied from 0 to 1, in increments of 0.01. K-Means was evaluated for numbers of clusters between 2 and 30.

Confusion matrices were constructed, and the number of misclassified inputs was recorded. To produce labels for output categories of the clustering methods, the count of each actual class falling into a given category was evaluated, and the most-frequently occurring class was used as the label for the category. This is in contrast to the ARTMAP procedure - no supervisory match-tracking procedure is used here.

The performance of these algorithms is shown in Figures 4.16 - 4.18, as well as Tables 4.7 - 4.9. The confusion matrices for the best-performing vigilance values are given in Tables 4.7 - 4.9. Gram-ART performs significantly better than both variants of Fuzzy-ART, and performs equally as well as the best K-Means experiment. Fuzzy-ART and Fuzzy-ART with discretized inputs both perform very similarly, strongly suggesting that discretization has no effect. Note that symbolic representation does not affect the performance of Fuzzy-ART, but Gram-ART performs extremely well across a wide range

of vigilance values. For all methods, vigilance values below 0.5 produced 100 classification errors. For a wide range of vigilance values ($\rho>0.74$), Gram-ART produced a constant number of clusters, corresponding to high performance from Figure 4.16. For all methods, vigilance values below 0.5 produced only 1 cluster.

As can be seen, Gram-ART performed extremely well, achieving 97.3% accuracy at the best-performing vigilance value, significantly better than either K-Means or Fuzzy-ART. Additionally, at the best-performing value, Gram-ART utilized many fewer categories, indicating that the category templates generated were particularly salient.

It is worth noting that at relatively low vigilance values (less than 0.6) Gram-ART produces similar performance to that of higher vigilance values (greater than 0.75) while generating many fewer clusters. In the intervening vigilance range (0.6 to 0.75) Gram-ART errors increase, though not significantly. This is likely due to the difficulty of distinguishing species from the given data. It is also expected that if the IRIS dataset was larger, this effect would be less pronounced, and the performance of Gram-ART would be constant throughout this range.

To analyze the effect of converting the real-valued input of the raw IRIS data set to symbolic information, the IRIS data set was applied to Gram-ART with 2, 5, 10, and 100 symbols per dimension. The results are shown in Figure 4.19 and Figure 4.20. Note that there is a large difference between the performance profiles utilizing 2 and 5 symbols, a smaller difference between 5 and 10 symbols, and very little difference between 10 and 100 symbols, though the number of clusters generated has changed significantly.

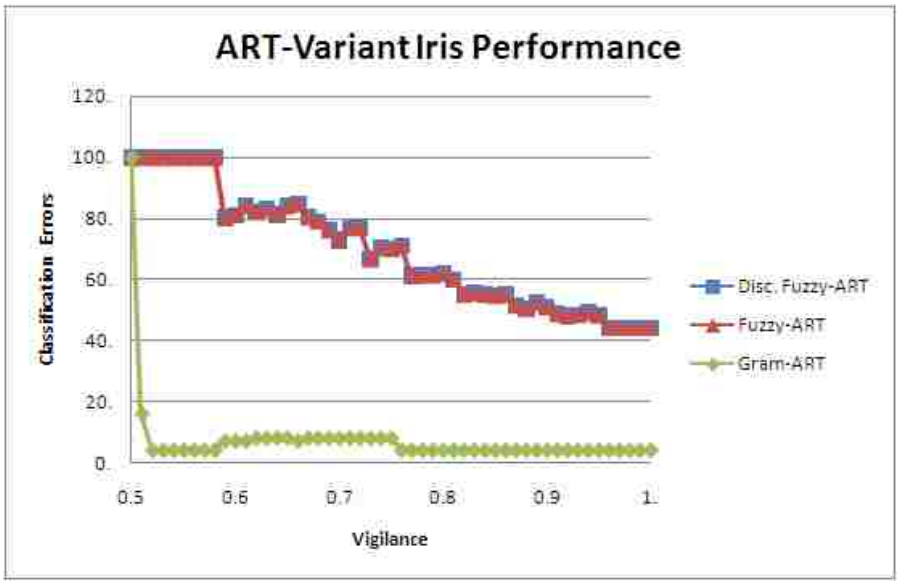


Figure 4.16. ART variant classification performance on the IRIS data-set.

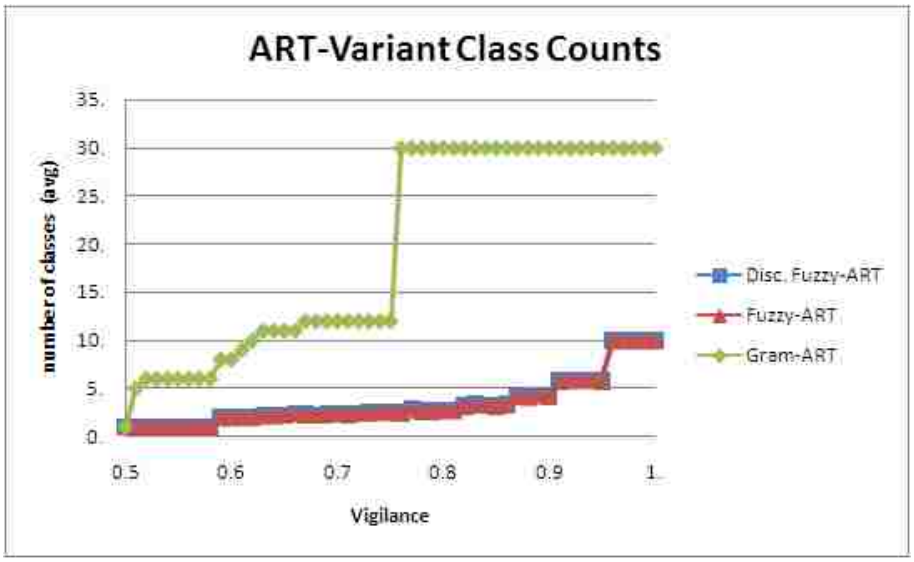


Figure 4.17. ART variant clustering profile on the IRIS data-set.

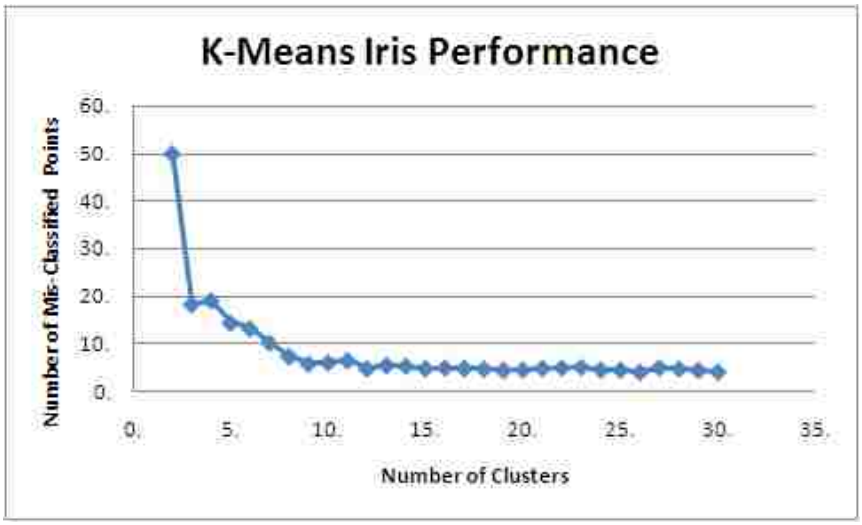


Figure 4.18. K-Means IRIS data-set performance.

Table 4.7. Confusion matrix for best-performing discretized Fuzzy-ART on IRIS dataset. Vigilance value is 0.96.

	D. Fuzzy-ART	Actual		
Predicted		Setosa	Versicolor	Virginica
	Setosa	0	5	1
	Versicolor	0	25	8
	Virginica	0	20	41

Table 4.8. Confusion matrix for best-performing classical Fuzzy-ART on IRIS data-set. Vigilance value is 0.96.

	Fuzzy-ART	Actual		
Predicted		Setosa	Versicolor	Virginica
	Setosa	7	11	1
	Versicolor	3	19	8
	Virginica	0	20	41

Table 4.9. Confusion matrix for best-performing Gram-ART on IRIS data-set. Vigilance value is 0.52.

	Gram-ART	Actual		
Predicted		Setosa	Versicolor	Virginica
	Setosa	0	0	0
	Versicolor	0	49	3
	Virginica	0	1	47

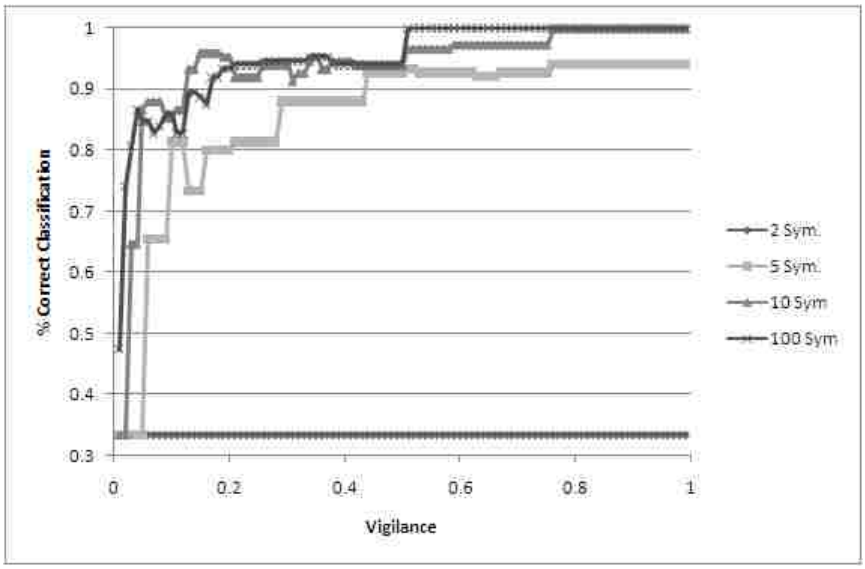


Figure 4.19. Gram-ART Performance on the IRIS data set over varying vigilance and number of symbols per data dimension.

For 2 symbols per data dimension, this is not likely enough information to capture the complexity of the problem, leading to only a few clusters, and poor performance, regardless of vigilance. Five symbols per dimension produce much greater capability, but Gram-ART is not able to produce a high-value classification for any vigilance setting. At 10 symbols per dimension Gram-ART is able to make very accurate classifications with moderate cluster proliferation. Similarly, 100 symbols per dimension results in accurate classification, but many clusters are generated.

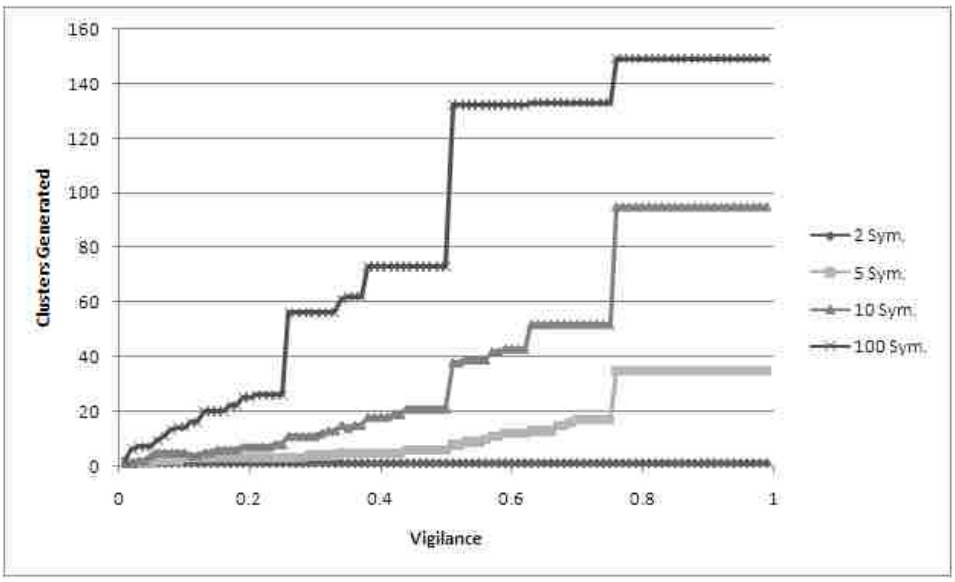


Figure 4.20. Gram-ART class counts generated on the IRIS data set over varying vigilance and number of symbols per data dimension.

This example illustrates the fact that when dealing with symbolic information, even when it is based on real-valued measurements, there is no measurement of proximity between symbols. In order to handle symbolic patterns, Gram-ART trades a real-valued proximity measure for a temporal proximity measure by accumulating the frequency of symbol occurrences. Even though the symbol-sets are derived from the same data set, they can produce radically different results. This also suggests that when converting real-valued information to symbolic information, there may be 'sweet-spots' - particular parameter settings that result in efficient clustering.

4.7. MUSHROOM DATA-SET

The Mushroom Database contains information about 8124 species of mushrooms. It classifies each as poisonous or edible and gives 22 attributes to identify the species.

Attributes include qualities like odor, shape, color, population and habitat [83]. Each of these attributes is given as a character that represents one type of the attribute. For example, odor is represented by almond (a), anise (l), creosote (c), fishy (y), foul (f), musty (m), none (n), pungent (p), or spicy (s).

The mushroom data-set is challenging for most clustering methods due to the symbolic representation of attribute values. This greatly limits the effectiveness of typical clustering methods, as it is difficult to express the symbols in a format that is able to be simply processed [57, 84-86]. For example, a previous implementation utilizing ARTMAP used a large binary vector to represent all combinations of traits [87]. As Gram-ART is fundamentally a symbolic clustering method, the only difficulty lies in constructing the grammar, which can be easily derived from the data-set specification.

The mushroom data-set grammar is shown in Table 4.10 and Table 4.11. To produce labels for output categories of the clustering methods, the count of each actual class falling into a given category was evaluated, and the most-frequently occurring class was used as the label for the category. Again, this is in contrast to the ARTMAP procedure - no supervisory procedure is used.

For comparison, the procedure outlined in [87] was used to apply the mushroom data set to Fuzzy-ART, the only difference being a supervisory signal is not applied, Gram-ART and Fuzzy-ART are compared directly.

Table 4.10. Mushroom BNF grammar, non-terms, seed and terminals.

```
N = {cap-shape, cap-surface, cap-color,
 bruises, odor, gill-attachment, gill-spacing,
 gill-size, gill-color, stalk-shape, stalk-root,
 stalk-surface-above-ring, stalk-surface-below-ring,
 stalk-color-below-ring, veil-type, veil-color,
 ring-number, ring-type, spore-print-color,
 population, habitat}

T = { bell, conical, convex, flat, knobbed, sunken,
 fibrous, grooves, scaly, smooth, brown, buff,
 cinnamon, gray, green, pink, purple, red, white,
 yellow, bruises, no_bruises, almond, anise,
 creosote, fishy, foul, musty, none, pungent, spicy,
 attached, descending, free, notched, close,
 crowded, distant, broad, narrow, black, brown,
 buff, chocolate, gray, orange, enlarging, tapering,
 bulbous, club, cup, equal, rhizomorphs, rooted,
 missing, fibrous, scaly, silky, partial, universal,
 one, two, cobwebby, evanescent, flaring, large,
 pendant, sheathing, zone, abundant, clustered,
 numerous, scattered, several, solitary, grasses,
 leaves, meadows, paths, urban, waste, woods }

S = <cap-shape> <cap-surface> <cap-color> <bruises>
 <odor> <gill-attachment> <gill-spacing> <gill-size>
 <gill-color> <stalk-shape> <stalk-root>
 <stalk-surface-above-ring>
 <stalk-surface-below-ring>
 <stalk-color-below-ring> <veil-type> <veil-color>
 <ring-number> <ring-type> <spore-print-color>
 <population> <habitat>
```

The clustering profile and performance over varying vigilance for Gram-ART and Fuzzy-ART is shown in Figure 4.21 and Figure 4.22. Using a vigilance value of 0.73, Gram-ART was able to perfectly categorize the mushrooms as poisonous or not. Gram-ART generated 24 clusters at this vigilance value.

Table 4.11. Mushroom BNF grammar production rules.

```
P can be represented as:

1. <cap-shape>::= {bell | conical | convex | flat |
  knobbed | sunken }
2. <cap-surface>::= {fibrous | grooves | scaly |
  smooth }
3. <cap-color>::= {brown | buff | cinnamon | gray |
  green | pink | purple | red | white | yellow }
4. <bruises>::= {bruises | no_bruises}
5. <odor>::= {almond | anise | creosote | fishy | foul
  | musty | none | pungent | spicy}
6. <gill-attachment>::= {attached | descending | free |
  notched }
7. <gill-spacing>::= {close | crowded | distant}
8. <gill-size>::= {broad | narrow}
9. <gill-color>::= {black | brown | buff | chocolate |
  gray | green | orange | pink | purple | red | white |
  yellow}
10. <stalk-shape>::= {enlarging | tapering}
11. <stalk-root>::= {bulbous | club | cup | equal |
  rhizomorphs | rooted | missing}
12. <stalk-surface-above-ring>::= {fibrous | scaly |
  silky | smooth }
13. <stalk-surface-below-ring>::= {fibrous | scaly |
  silky | smooth }
14. <stalk-color-above-ring>::= {brown | buff |
  cinnamon | gray | orange | pink | red | white |
  yellow }
15. <stalk-color-below-ring>::= {brown | buff |
  cinnamon | gray | orange | pink | red | white |
  yellow }
16. <veil-type>::= { partial | universal }
17. <veil-color>::= {brown | orange | white | yellow }
18. <ring-number>::= {none | one | two }
19. <ring-type>::= { cobwebby | evanescent | flaring |
  large | none | pendant | sheathing | zone }
20. <spore-print-color>::= { black | brown | buff |
  chocolate | green | orange | purple | white | yellow}
21. <population>::= { abundant | clustered | numerous |
  scattered | several | solitary }
22. <habitat>::= { grasses | leaves | meadows | paths |
  urban | waste | woods }
```

By contrast, Fuzzy-ART was not able to accurately classify the data until vigilance reached 0.93 (98.3% mean accuracy), at which point an average of 913 clusters were generated. For Fuzzy-ART, evaluation had to be halted at this point due to limited computational resources. At vigilance values greater than 0.85, Gram-ART generated over 1000 clusters, which was set as the maximum threshold. Due to this, Gram-ART's classification error began to rise as the resonance procedure was broken.

This example demonstrates that Gram-ART is particularly well-suited to processing symbolic information, achieving high accuracy with fewer clusters and at lower vigilance values than the popular Fuzzy-ART method. This improvement is likely due to Gram-ART's method of representation and update rule. To use symbolic information in Fuzzy-ART, a single large vector containing all possible symbol values is created and updated as one - the 22 dimensions with various discrete values of the mushroom data-set become a single 126 dimensional space. Gram-ARTs representation treats each dimension separately - maintaining 22 dimensions and storing the frequency of symbol occurrence within each of those dimensions. Additionally, Gram-ART's update rule is explicitly probabilistic, while Fuzzy-ART utilizes a probabilistic approximation based on a fixed learning rule. The Gram-ART representation method and update rule thus enable higher resolution templates and increased capability for discerning categories in data.

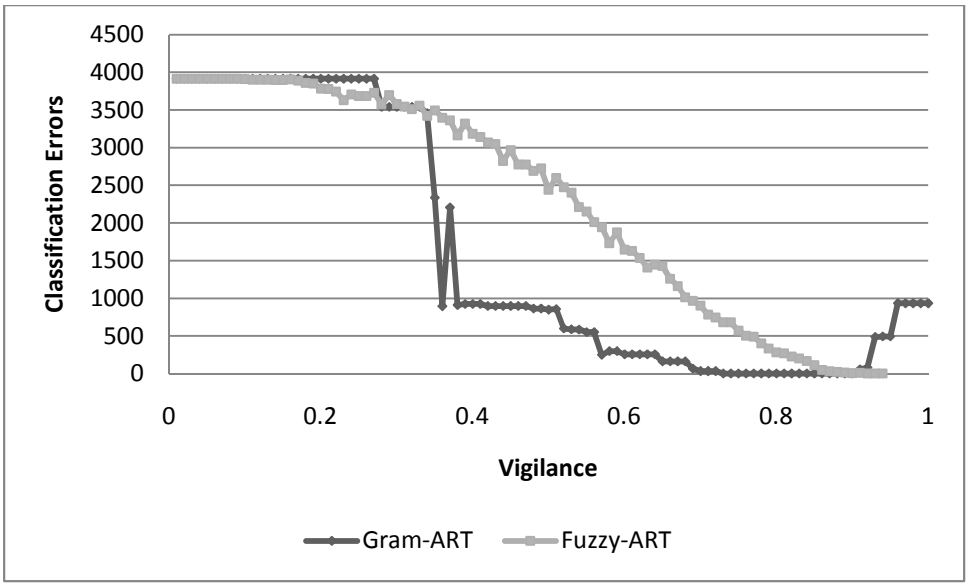


Figure 4.21. Gram-ART and Fuzzy-ART classification performance on the mushroom data-set.

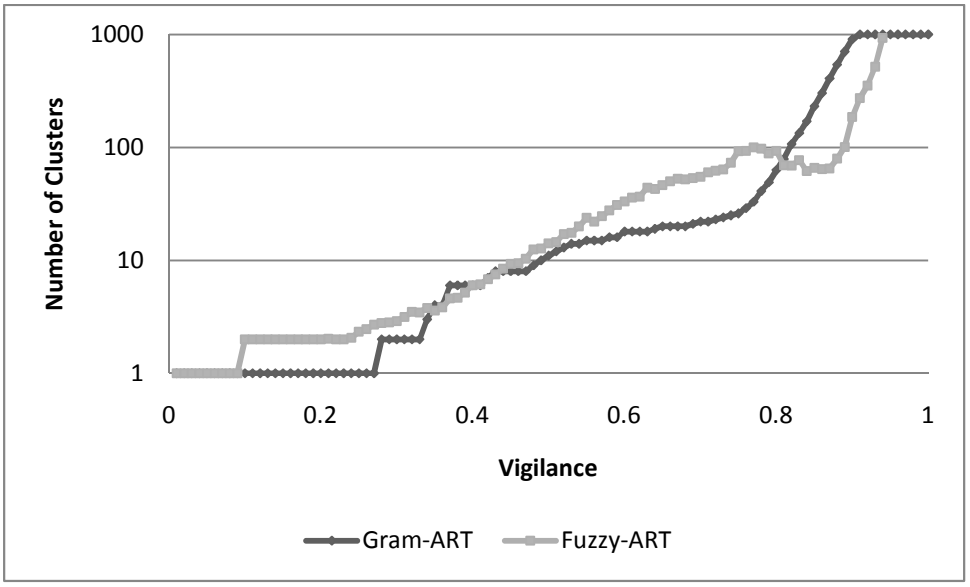


Figure 4.22. Gram-ART and Fuzzy-ART clustering profile on the mushroom data-set.

4.8. UNIX USER DATA-SET

To further illustrate the capabilities of Gram-ART, a final experiment was conducted on the UNIX user data set archived at the UCI machine learning repository [88]. The UNIX user data contains the transcripts of nine UNIX system users, originally intended for testing intrusion detection systems. The data set has been sanitized to remove file names, user names, directory structures and any other possibly identifying items, while command names, flags, and shell meta-characters have been preserved, as well as typos. This data set is interesting due to its inherently symbolic nature (each command is a symbol - over 500 commands possible), as well as its sequential nature and size (40,000+ sessions constitute the data set). Additionally, the data-set has been studied extensively in literature, with no previous method achieving over 83.8% correct classification [89, 90].

Gram-ART was executed utilizing a simple grammar with one non-term – "<command>" that can take one of 590 values, each corresponding to a UNIX shell command, flag, or meta-character, such as "dir" or "pwd." Similarly, there was only one production rule, specifying that a command may follow another command. Gram-ART was tested by randomly selecting 1000 sessions from the data-set and verifying their labeling. This is the same experimental configuration in [89, 90], and it is used here in order to facilitate a direct comparison of results.

The data was evaluated over varying vigilance values from 0.1 to 0.99, with performance shown in Figure 4.23. At a vigilance value of 0.9, Gram-ART was able to achieve 96.5% accuracy on the UNIX user data set, misidentifying the user of only 33 sessions out of the 1000 test sessions. This is a significant increase over the performance

reported in the literature for methods such as voting K-Nearest Neighbor (a popular non-parametric method), various discriminant analysis, and vector methods.

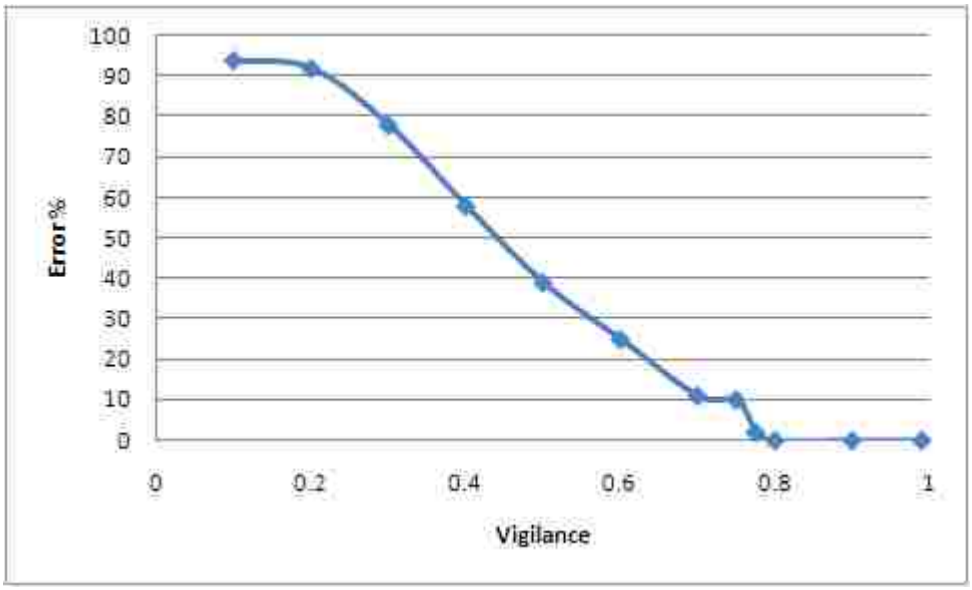


Figure 4.23. Gram-ART performance on the UNIX user data set over varying vigilance settings.

This experiment demonstrates the full capability and benefit of Gram-ART - variable data dimensionality, symbolic information, and large corpus of data. The Gram-ART structure was able to achieve very high performance on this data set as compared to historical methods due to non-parametric statistical foundations, adaptive-resonance design, and ability to operate on variable dimension and sequential data.

4.9. DISCUSSION

Gram-ART, a new Adaptive Resonance Theory variant, has been developed with many valuable properties, including the ability to cluster symbolic information and not

only data but the structure of data relative to a grammar. Additionally, the Gram-ART method is able to develop non-geometrically constrained cluster shapes, which enables an increased ability to model complex data. This ability was demonstrated through the application of Gram-ART to several clustering problems, ranging from illustrative 2D clustering, fixed-dimensionality, real valued standard IRIS database, fixed dimensionality symbolic Mushroom database, and variable dimensionality, sequential, symbolic UNIX user data set.

Future research directions of Gram-ART development could include modification of the trace operator, such as the use of median or minimum of distribution samples, and investigating the effect on performance. Additionally, the weight update mechanism may be modified to utilize a kind of Bayesian update rule.

5. META-LEARNING GENETIC PROGRAMMING

To demonstrate the principles and advantages of meta-learning, its application to the even and odd parity problems, standard benchmarks for GP and automatic function definition methods [70] are examined. A hypothetical GP system is proposed utilizing a set of Boolean operators to construct individuals implementing the even or odd parity functions (XOR and XNOR, respectively). Two cases of evolution of the three-input XOR function are analyzed, both starting with populations implementing the two-input XOR function, with and without the abstraction that is inherent in a meta-learning system. A third case is presented illustrating the functionality of a simple selection mechanism on the odd-parity function.

5.1. EVEN-PARITY PROBLEM OVERVIEW

Koza described the even parity problem succinctly:

> *"The Boolean even-parity function of k Boolean arguments returns* `T` *(True) if an odd number of its arguments are* `T`*, and otherwise returns* `NIL` *(False). The concatenation of this returned bit to the original string making the total string even, hence even-parity.*
>
> *In applyito the even-parity function of k arguments, the terminal set* `T` *consists of the k Boolean arguments* `D0`*,* `D1`*,* `D2`*, ... involved in the problem, so that*
>
> *T = {D0, D1, D2, ...}.*
>
> *The function set F for all the examples herein consists of the following computationally complete set of four two-argument primitive Boolean functions:*
>
> *F = {AND, OR, NAND, NOR, NOT}.*
>
> *The Boolean even-parity functions appear to be the most difficult Boolean functions to find via a blind random generative search of expressions using the above function set F and the terminal set T. For example, even though there are only 256 different Boolean functions with three arguments and one output, the Boolean even-3-parity function is so difficult to find via a blind random generative search that we did not encounter it at all after randomly generating 10,000,000 expressions using this function set F and terminal set T. In addition, the even-parity function appears to be the most difficult to learn using genetic programming using this function set F and terminal set T [67]."*

The odd-parity function is similarly constructed, returning true if an even number of its arguments are true, and otherwise returning false. In GP, the genome of an individual is represented as a tree structure, where operations are applied at branches, and the leaves are constants and problem parameters. An illustration of a functional represented as tree structure is shown in Figure 5.1 [17, 67]. One advantage of GP is that the results are easily human interpretable and formally verifiable, a quality that is not present in many other computational intelligence methods [50].

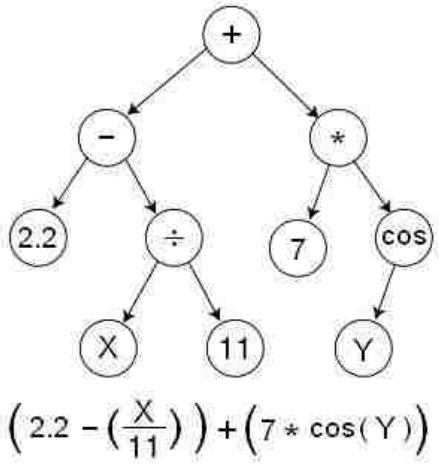


Figure 5.1. Illustration of function representation as tree structure.

The even-2-parity function is simply the XOR function, which is itself a composition of the terminal set functions in one simple possible configuration. Using a tree representation, the XOR function is shown in Figure 5.2.

a XOR b = (a OR b) AND (a NAND b)

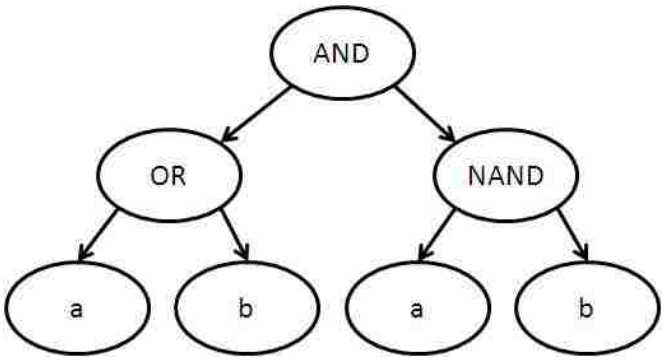


Figure 5.2. XOR tree representation.

Constructing the even-3-parity function using only these primitives is more difficult, but follows a similar pattern, illustrated in Figure 5.3.

XOR (a, b, c) = (((a OR b) AND (a NAND b)) OR c) AND

(((a OR b) AND (a NAND b)) NAND c)

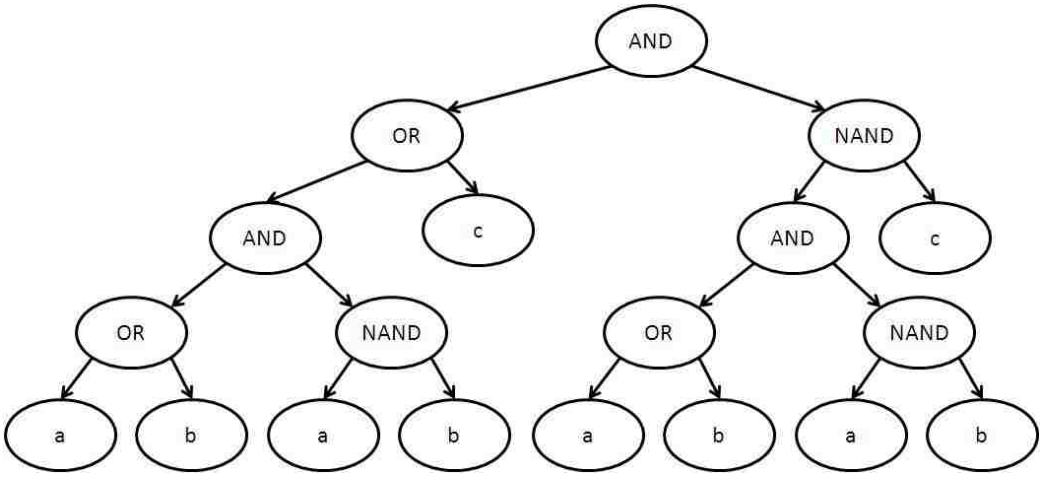


Figure 5.3. Three-input XOR tree representation.

That the three-input XOR structure relies on the recursive use of the two-input XOR function, replacing the 'a' nodes with XOR nodes, and re-assigning the top-level 'b' nodes to be the 'c' variable. If a 2-bit XOR function is defined explicitly as in Figure 5.4, the even-3-parity function becomes greatly simplified, as shown in Figure 5.5.

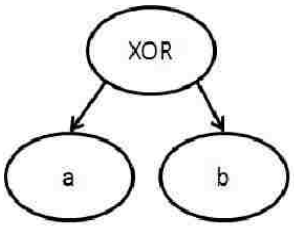


Figure 5.4. Simplified two-input XOR.

XOR(a, b, c) = (a XOR b) XOR c

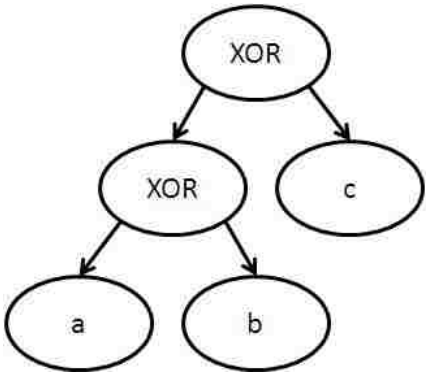


Figure 5.5. Simplified three-input XOR.

5.1.1. Case 1 – Non-Meta XOR3 Evolution. Taking a GP as an example, in a non-meta learning system, evolution of the XOR3 function must proceed through at least two generations.

To further expand on this illustration, consider the best case scenario whereby all the individuals in the population incorporate the simplified XOR function, as shown in Figure 5.6.

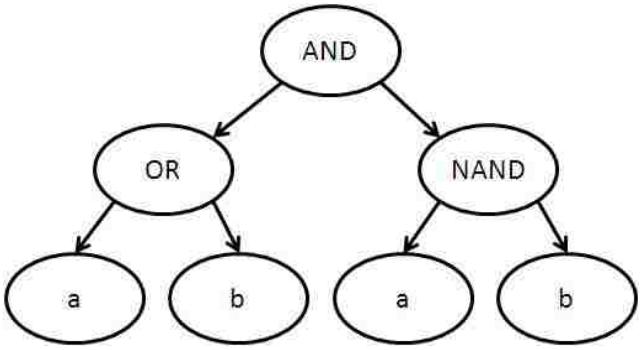


Figure 5.6. Initial non-meta learning XOR2 individual.

As there are 4 leaf nodes out of 7 total nodes, the probability of selecting a leaf node for crossover (P_{L1}) is 4/7. Assuming a uniform population of individuals implementing XOR2 (translating to a 100% probability of choosing another XOR2 individual for crossover) the probability of selecting the root node of another individual to replace the selected leaf node is (P_{F1}) 1/7.

Then the evolutionary process must select one of the two top-level 'b' nodes for mutation from the tree which has a total of thirteen nodes, thus the probability of selecting one correct leaf for mutation (P_{M1}) is 2/13. Choosing from the eight possible

node types (the combination of terminal set and functional set), the probability of selecting the correct 'c' variable (P_{V1}) is 1/8.

At this point the evolutionary reproduction steps are completed, and the individual shown in Figure 5.7 is evaluated. This partial XOR3 function is not yet complete, but it correctly completes one test case more than the XOR2 function, which may give it an evolutionary advantage. Assuming that the individual survives to the next generation and is again selected as a parent with 100% probability, an additional reproduction step must be completed to yield an XOR3 function.

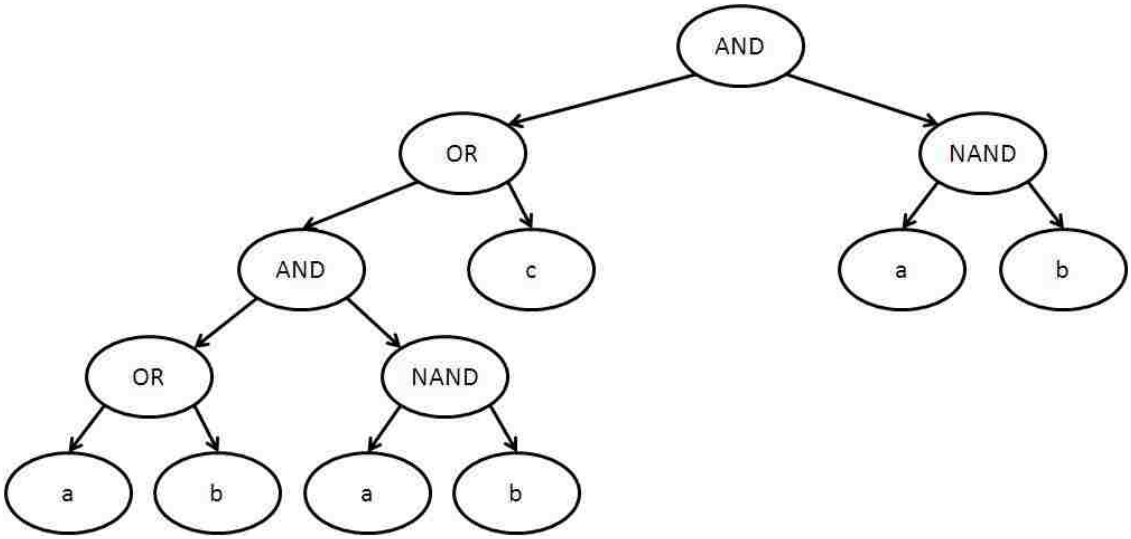


Figure 5.7. Intermediate step in development of 3-bit XOR function after a single generation.

Now the correct leaf node must be selected for crossover, but this time there is only one node, the 'a' node at a depth of three, from the thirteen possible nodes, so the probability of selecting the correct leaf node for crossover (P_{L2}) is 1/13. Once again,

assuming all other individuals in the population still implement the XOR2 function in Figure 5.2, the probability of selecting the root of another XOR2 individual to replace the leaf (P_{F2}) is 1/7. At the completion of crossover, the total number of nodes in the tree becomes eighteen. At the mutation step, the remaining 'b' node at depth three must be selected, and the probability of selecting correct leaf for mutation (P_{M2}) is 1/18. Completing the XOR3, the probability of selecting the correct variable from the total set of node types (P_{V2}) is 1/8. The completed three-input XOR function is illustrated earlier in Figure 5.3.

Ignoring changes in the population and evolutionary survivability, the probability of transitioning from XOR2 to XOR3 in two generations without meta-learning is calculated:

$$P_{xor3\_nonmeta} = P_{L1}*P_{F1}*P_{M1}*P_{V1}*P_{L2}*P_{F2}*P_{M2}*P_{V2} = 1.19 \times 10^{-7}$$

Where P_{L1} is the probability of a leaf node selection for crossover during the first generation, P_{F1} is the probability of functional root selection for crossover during the first generation, P_{M1} is the probability of proper leaf selection for mutation during the first generation, P_{V1} is the probability of proper variable selection for mutation during the first generation, P_{L2} is the probability of a leaf node selection for crossover during the second generation, P_{F2} is the probability of functional root selection for crossover during the second generation, P_{M2} is the probability of proper leaf selection for mutation during the second generation, P_{V2} is the probability of proper variable selection for mutation during the second generation.

Note that this ignores the significant influence of relative fitness, generational selection, parent selection, probability of application of crossover/mutation operators and population influence and may be interpreted as a kind of upper-bound on the probability that a two-input XOR individual will develop into a three-input XOR without the abstraction capability of meta-learning.

5.1.2. Case 2 – Meta-Learning XOR3 Evolution. In this case a meta-learning system is assumed that has already learned a two-input XOR function, performed generalization and added this to the function set (F = AND, OR, NAND, NOR, NOT, XOR2). The probability that the system will transition from XOR2 to XOR3 is calculated using only the mutation step.

With a population uniformly initialized with the two-input XOR and an individual selected from this population, illustrated in Figure 5.2, the probability of selecting a leaf node for mutation (P_L) is 2/3 as the simplified XOR tree has only 3 nodes, and two of them are terminals. Having selected a terminal, the probability of selecting the XOR2 function from the node set of six functions and three terminals to replace the leaf node (P_F) is 1/9. Assuming a recursive mutation process, two new leaf nodes must be selected, and they must contain variables not yet used by the tree to produce a three-input XOR. The probability of selecting the correct terminal node is 1/9, and this process must be repeated twice, so the probability of selecting two correct terminal nodes (P_V) is $(1/9)^2$ or 1/81. Using only one generation the three-input XOR can be developed in a meta-learning system:

Probability of XOR3 from XOR2: $P_{xor3\_meta} = P_L * P_F * P_V = 0.000914$

Where P_L is the probability of a leaf node selection for mutation, P_F is the probability of XOR2 function selection for mutation, P_V is the probability of proper leaf selection for mutation.

Note that using meta-learning, the three-input XOR can also occur with a crossover and a mutation, where the non-meta learning system must utilize two full generations. Though the size of the functional set has increased, the number of changes necessary to place an upper-bound on the probability of a three-input XOR occurring has been substantially decreased, allowing the evolutionary process to focus on high-level changes.

In a large population, the XOR3 function may occur in a single generation with a meta-learning system, where a non-meta learning system must take at least two generation and probably many thousands of evaluations to evolve an XOR3.

5.1.3. Case 3 – Selection and Odd-Parity Evolution. To demonstrate the advantages of the complete meta-learning procedure, the 2-bit even-parity problem is first presented to a theoretical meta-learning system, then the 2-bit odd-parity problem, and finally the 3-bit even-parity problem. The selection mechanism shall have 2 inputs – the first is activated only when the system is operating on the even-parity problem, the second is activated only when operating on the odd-parity problem. Initially, the memory is empty, so the optimizer is initialized with random solutions.

Presented with the even-2-parity problem, the optimizer outputs a resulting solution that performs the XOR function – "D0 XOR D1", where D0 and D1 are the Boolean arguments of the input. This function is passed to the generalization mechanism, which removes the absolute references to the Boolean arguments, replacing

them with dummy variables ‘A’ and ‘B’, resulting in the function “A XOR B”. This generalized XOR function is then added to the memory, making the function available as a primitive. The functional set becomes:

F = {AND, OR, NAND, NOR, NOT, XOR}.

The selection mechanism is updated to learn an association between the active ‘even-parity’ input and the new memory element. At this point the procedure and difference in optimization would be no different than if the optimizer were operating without the rest of the meta-learning architecture.

Next, the odd-2-parity problem is presented, the ‘odd-parity’ input is activated on the selector mechanism, and having no other elements to select, the sole item in memory (the generalized “A XOR B” function) is selected to initialize the state of the optimizer. The optimizer replaces the dummy variables with references to the Boolean arguments and begins optimization. As only a small modification is necessary, the addition of the NOT primitive function at a high-level to create an XNOR function, the optimizer has a high probability of quickly finding a perfect solution to the odd-2-parity problem. This differs from a randomly initialized optimizer as there would be a lower probability of finding a good solution due to the need to explore more modifications. Once the meta-learning optimizer finds the solution, the generalization, memory insert, and selection training steps are repeated for the XNOR function:

F = {AND, OR, NAND, NOR, NOT, XOR, XNOR}.

Finally, the even-3-parity problem is presented to the meta-learning architecture. The selection ‘even-parity’ input is activated, and the associated XOR memory element is used to initialize the optimizer state. The optimizer replaces the XOR dummy variables

with argument references, and begins the optimization process. The optimizer need only make the relatively small change of cascading the XOR function to produce a 3-input XOR function, where a raw optimization function without a memory or selection method would need to evaluate and modify many combinations of the original 5 functional primitives to arrive at a good solution. The meta-learning architecture should be able to arrive at high-value solutions rapidly by exploiting previously generated solution to construct high-level solutions.

In this example the memory component stores generalized solutions to previously encountered problems - these elements could be thought of as memes, as they are solutions that are passed between problem instances. The selection mechanism performs association between problem features and solutions that are likely to yield high-value results. By not only providing the input data to the problem, but additional meta-data about the characteristics of the problem, the meta-learning architecture should be able to construct meta-memes and store them using the memory mechanism, allowing higher-order learning to occur automatically.

5.2. META-GP ARCHITECTURE

A GP Meta-Learning system is constructed by augmenting the Automatic Function Definition GP with a neural network method that is trained to map between a parametric description of a given task and the function-categories created by the Gram-ART method described in Section 5. The output of this mapping is used to probabilistically bias the use of functions in the initial generation of the GP process. By seeding the population with genetic information that has been useful in similar situations

in the past, it is expected that the GP will be able to more quickly find a high-quality solution. If a high-quality solution is not found, the exploration/exploitation feedback mechanism will drive the system towards new solutions, which will be incorporated into the function library at the end of training. An illustration of the Meta-GP architecture is provided in Figure 5.8.

This stored learning should allow high-quality solutions to be rapidly found when encountering similar scenarios, and with careful construction of successive training scenarios, the method should be able to find good solutions in increasingly complex scenarios with less computational expense than a classical algorithm.

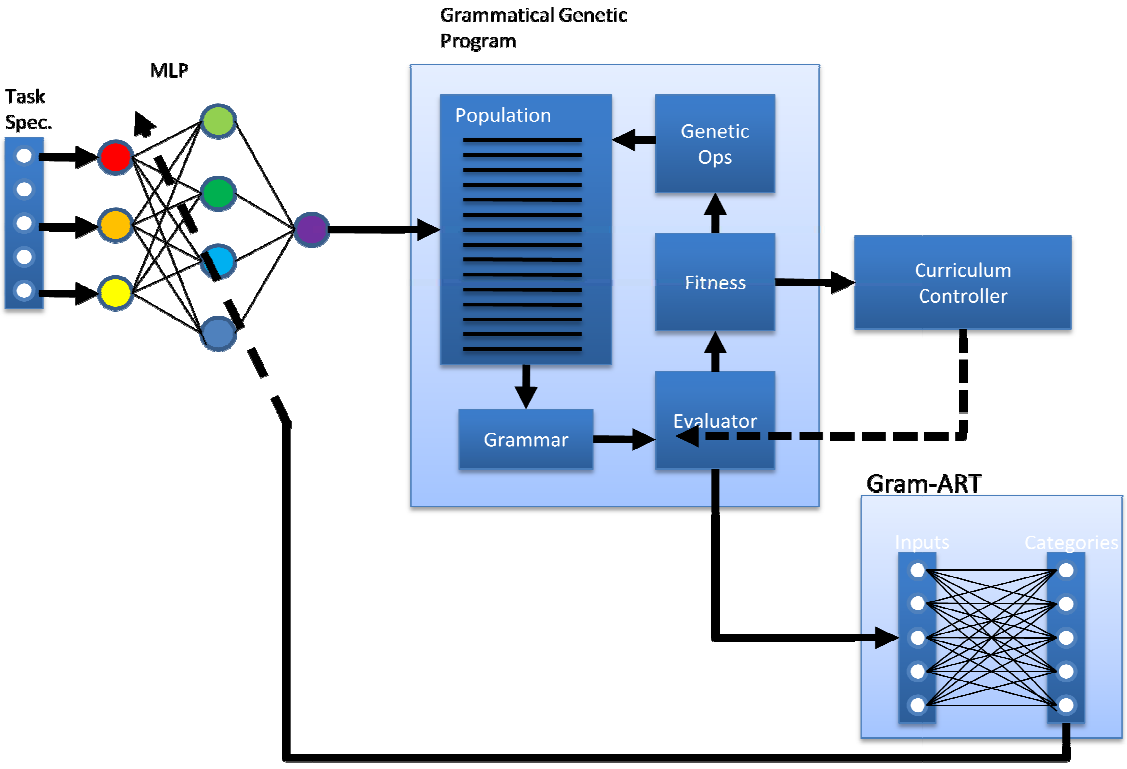


Figure 5.8. Meta-learning GP architecture.

The Meta-GP process begins with the presentation of a parametric description of a problem to the Multi-Layer Perceptron (MLP). This problem description consists of a real-valued vector with elements in the domain [0,1]. The problem description vector activates the MLP, producing an output vector with the same dimensionality as the number of learned, stored functions – the number of categories in the Gram-ART Structure. Each element of the output vector is interpreted as the probability that a corresponding function category will be useful in the genetic programming optimization process. The initial GP population is seeded with automatically defined functions utilizing fitness proportional selection based on the 'perceived fitness' provided by the MLP output. Using this initial population, the GP optimization process is initiated, and solutions to the provided problem are generated, evaluated, and evolved. The differential fitness between a trial individual and its parents is utilized to determine if an individual should be added to the function library. When the stopping conditions for the optimization process are met (such as population convergence, or a fitness threshold) the occurrence frequency of library functions is calculated for the fittest individuals in the population. This frequency is used to train the MLP to associate the current problem description with the functions that are most likely to provide fit solutions. This process is repeated across several instances and sizes of different problems.

The Meta-GP architecture is evaluated on two test-bed problems, the Parity Problem described in Section 5.1, and variations of the game Pac-Man. The even and odd-parity problem stack is a standard GP function approximation benchmark, and a good demonstration problem for functional usage as 2 Bit Even Parity is a sub-problem of 4-Bit even parity is a sub-problem of 6-bit, etc [17]. The game of Pac-Man is a

standard benchmark for the study of evolution of autonomous agents in changing environments. The Pac-Man scenario allows the demonstration of behaviors such as task-prioritization (eating dots vs. avoiding ghosts), adaptability, and robustness. Board size, additional incentives (fruits, extra lives, etc) and monsters can all be modified for increasing problem complexities. Additionally the Pac-Man framework can be modified for multi-agent optimization by introducing additional Pac-Men [58], though this is out of the scope of the current experiments.

5.3. META-GP EVALUATION: THE PARITY PROBLEM

Evaluation of the Meta-GP architecture on the parity problem was completed through a series of experiments designed to provide information about the additional effect of each component on the optimization process. This was accomplished through three primary experiments, each one executing on the even and odd-parity problems, each ranging in size from 2 to 10 bits. Evolved individuals were evaluated by applying all possible input combinations, and comparing the output to the correct (even or odd) output. The number of correct cases was used as the fitness value. This is in contrast to Koza's evaluation method, where a relatively small subset of outputs was selected as evaluation cases.

The first experiment investigates the performance of the GP process alone, without automatic function definition, and without the influence of the MLP. A second experiment examines the effect of adding automatic function definition on the system performance. The third experiment utilizes the full Meta-GP architecture, including

automatic function definition, and recognition MLP to explore the influence of these components on the optimization process.

5.3.1. Genetic Programming Baseline. A baseline performance level was set by applying the GP process alone to the even and odd parity problem with size ranging from 2 to 10 bits. The experiments were carried out with a population size of 1000 individuals, and 10K maximum generations. The maximum genome size (number of nodes) was fixed at 100. One hundred parents were selected from the population using roulette-wheel selection, generating 100 children. Children were generated using either mutation or recombination (but not both for a given child) with equal probability, based on empirical evidence. Diversity control was performed utilizing a mass extinction method with elitism [85]. Mass extinction was initiated after the average fitness of the population was equal to the best fitness of the population for a set number of generations, indicating convergence. Mass extinction with elitism was implemented by saving the top n individuals, then re-initializing the population. For all experiments with diversity control, the threshold for initiating mass extinction was 100 generations with 100 elitist individuals.

To obtain a statistical sampling, experiments were repeated with the same configuration 30 times, recording the maximum number of generations to completely solve the presented parity problem. The results are shown in Figure 5.9 for problem sizes between 2 bits and 5 bits, reported as the percent correct cases achieved. The values of best fitness individuals are shown using solid lines, while the population average is shown using dashed lines. The population averages vary greatly due to the use of the

mass extinction mechanism, which resets the population when convergence criteria are detected.

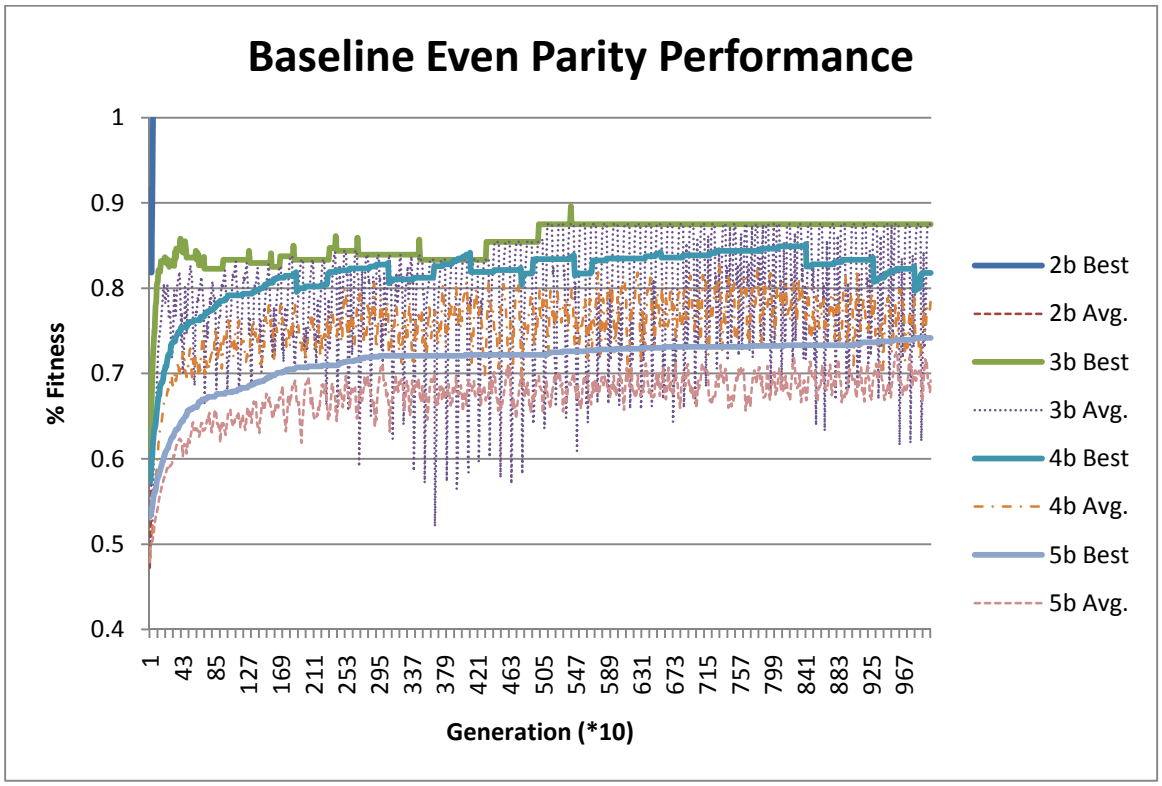


Figure 5.9. Baseline even parity performance.

Note that smaller problem sizes have much higher performance, particularly the 2-bit problems, which achieve perfect performance within 10 generations. Larger problem sizes take much longer to produce fully-fit individuals, and indeed the 5-bit problem size fails to produce an individual with 100% fitness in any evaluated run. This trend follows through the larger problem sizes. The performance data for problem sizes 6 through 10 is omitted here for brevity, as the GP process was not able to achieve over

60% fitness for a problem size larger than 5 bits. The baseline performance of the odd-parity problem is effectively identical, as would be expected, and is omitted here for brevity.

These results illustrate the challenge of directly evolving individuals that implement parity functionality for large problem instances.

5.3.2. Automatic Function Definition. In Grammatical Evolution architectures, the Gram-ART unit can serve the purpose of dynamic function definition, providing a library of generalized functions as cluster templates. If an individual has a non-zero differential fitness between itself and the higher fitness of its two parents, a search is initiated to find a sub-tree in the individual that differs from that of its parents. When found, this sub-tree is passed as input to the Gram-ART method, where it is matched to a category and modifies a template. The templates are then extracted from Gram-ART and added to the grammar as high-level functions that are available for new individuals to utilize. In this way, useful sub-trees are removed from the evolutionary process, and the genetic operations are then focused on increasingly high-level modifications to the programs. This type of mechanism maintains population diversity and is able to counteract the bloat of individuals that causes fitness stagnation [70].

The 3-bit even parity problem was used to evaluate Gram-ART as an automatic function definition method. The Gram-ART method was evaluated over 11 vigilance values, ranging from 0.0 to 1.0. Note that the vigilance value of 1.0 corresponds to creating a new template for each not-equal input, corresponding to classical automatic function definition methods in literature.

The experiments were carried out with a population size of 1000 individuals, and 10K maximum generations. The maximum genome size (number of nodes) was fixed at 100. One hundred parents were selected from the population using roulette-wheel selection, generating 100 children. Children were generated using either mutation or recombination (but not both for a given child) with equal probability, based on empirical evidence. Diversity control was performed utilizing a mass extinction method with elitism [85]. Mass extinction was initiated after the average fitness of the population was equal to the best fitness of the population for a set number of generations, indicating convergence. Mass extinction with elitism was implemented by saving the top *n* individuals, then re-initializing the population. For all experiments with diversity control, the threshold for initiating mass extinction was 100 generations with 100 elitist individuals.

Experiments were performed by varying two factors – utilization of diversity control, utilization of Gram-ART. For experiments utilizing Gram-ART, the experiment was repeated for 11 values of vigilance. To obtain a statistical sampling, experiments were repeated with the same configuration 30 times, recording the maximum number of generations to completely solve the 3-bit even parity problem. The results are shown in Figure 5.10 and Figure 5.11. The dashed lines of Figure 5.10 represent baseline performance measures for the evolutionary process without automatic function definition. Note that the evolutionary process without diversity control, utilizing Gram-ART with a vigilance value of 0.9 out-performs all other configurations.

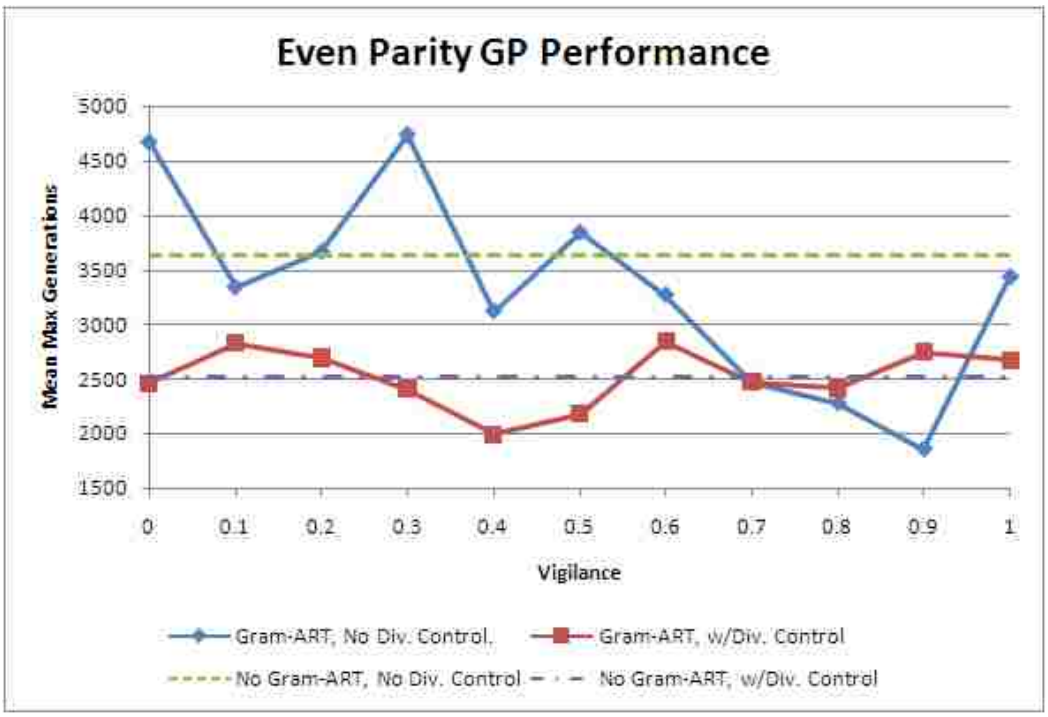


Figure 5.10. Evolutionary profiles for the 3-bit even parity problem utilizing GP with combinations of Gram-ART and diversity control.

Figure 5.11 shows the effect of diversity control on cluster generation. Under diversity control, the GP regularly resets, creating a new population and retaining only a small subset of highly fit individuals. As the new population evolves, many more individuals are increasing in fitness, creating more functions for utilization. Without diversity control, this process only occurs at the beginning of the evolutionary process, and thus function creation stagnates.

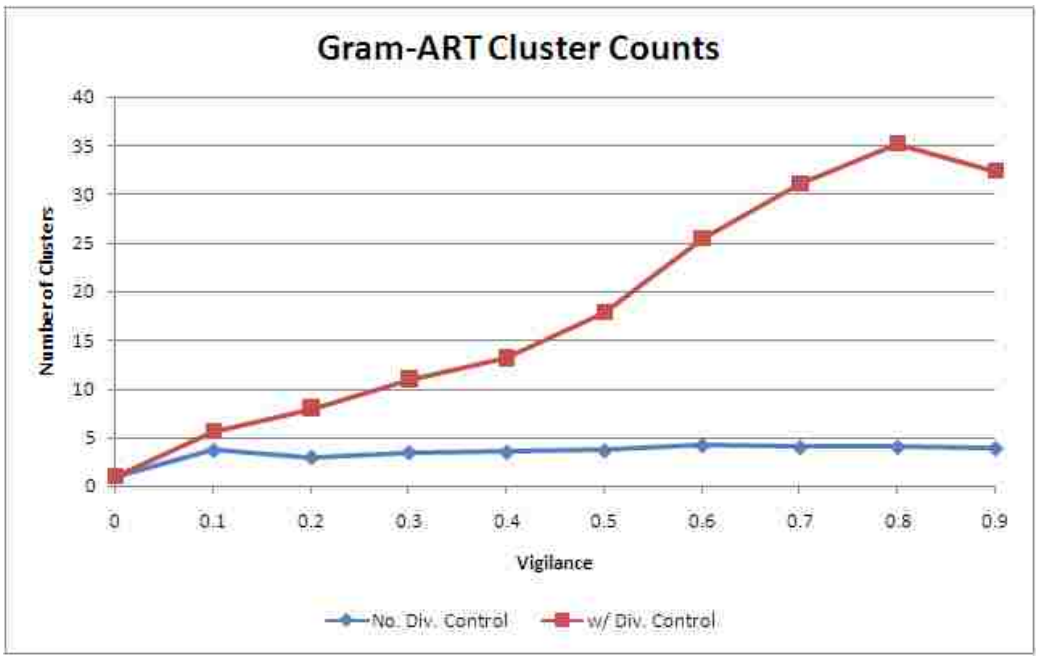


Figure 5.11. Clustering profiles for Gram-ART on the even parity GP problem.

Figure 5.12 and Figure 5.13 show the normalized distribution of function utilization by the fittest individuals after completion of the 3-bit even parity problem. It can be seen that early dynamically generated functions are highly utilized.

For a vigilance value of 0.7, the number of clusters generated using Gram-ART as automatic function definition is many orders of magnitude smaller than the number of functions generated without clustering. Less than a hundred functions were generated using clustering, compared to tens of thousands of functions generated without clustering. In the genetic process, this increases the probability that any given high-value function will be selected for utilization in an individual.

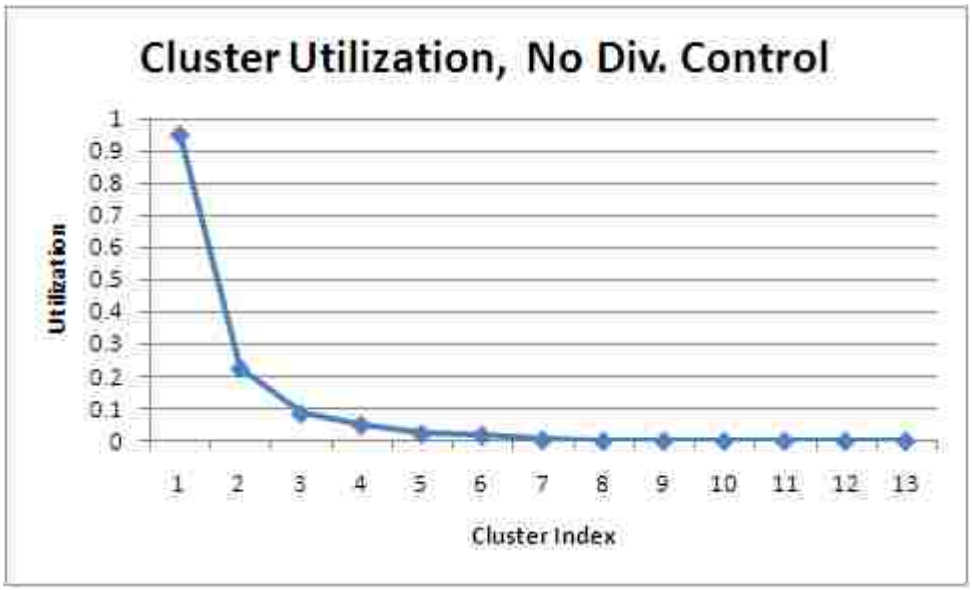


Figure 5.12. Example function utilization after dynamic function definition.

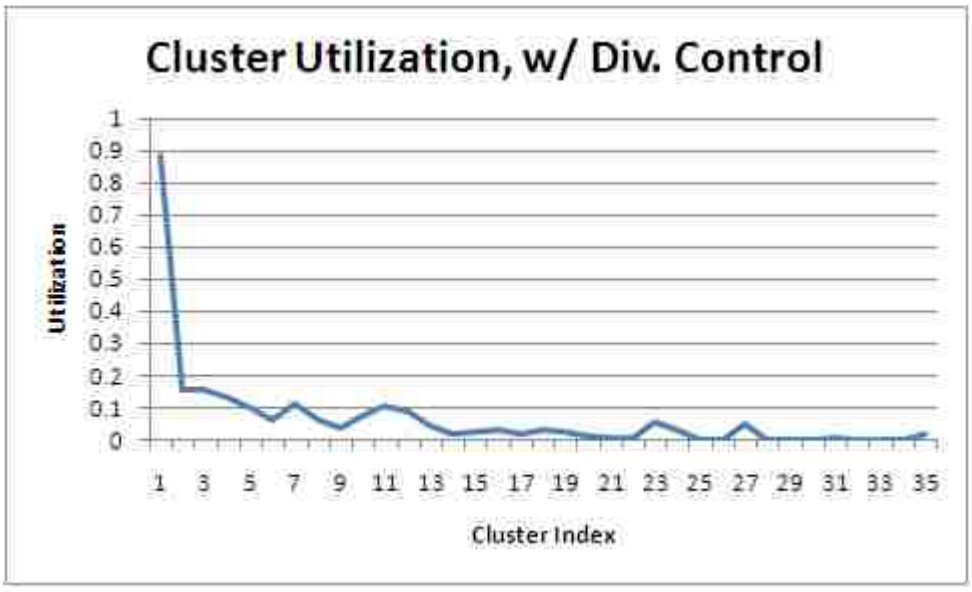


Figure 5.13. Example function utilization after dynamic function and diversity control.

An analysis of variance was performed, showing that utilizing Gram-ART with the best vigilance settings (0.9 without diversity control, 0.4 with diversity control), the

evolutionary process was significantly improved over all other evolutionary configurations for a confidence value of $\alpha = 0.05$.

To see how Automatic Function Definition (ADFS) with Gram-ART compares with traditional methods, the data already collected need be examined. When the vigilance value equals zero, this is equivalent to the evolutionary process without any ADFS, as only one cluster is generated, and it is continually over-written. Similarly, a vigilance value of 1 is equivalent to traditional ADFS where every generated sub-tree is kept, and never modified. It can be seen from the results that neither of these extremes is the best performing on the Even Parity problem - a middle ground must be found, and Gram-ART fulfills these requirements by contributing symbolic, variable dimension, structural clustering. Additionally Gram-ART's infinite-window updating ensures short term template stability, and enables long-term evolutionary modification.

5.3.3. Parity Evaluation. To evaluate the performance of the full Meta-GP architecture on the parity problem, three experiments were conducted. The first applied only the GP optimizer to the 10-bit even parity problem to acquire a baseline performance metric. The second experiment integrated the memory unit as described previously as an automatic function definition method.

In the second experiment, the problem size was increased from 2 bits to 10 bits. Problem size increases were triggered as the optimizing system produced individuals that achieved100% fitness for the current problem size, or a threshold of 1000 generations had passed, whichever came first. This increasing problem scheme is known as curriculum control. The third experiment utilized the full Meta-GP architecture by applying the even parity problem for problem sizes between 2 and 5, using curriculum control, then

applying the system to the odd-parity problem for problem sizes between 2 and 5, before applying the even-parity problem again for problem sizes between 2 and 10 for final evaluation. For each problem type in the meta-learning experiment, the corresponding neural network input was trained on the problem specification (a binary vector of 2 inputs, one for even-parity, one for odd-parity) using back-propagation with a learning rate of 0.01 and a training threshold of 0.03 mean squared error.

The experiments were completed using the parameter settings described in Section 5.3.2., and the system was allowed to evolve for 10,000 generations before terminating. To obtain a statistical sample, the experiments were each completed 1000 times. Table 5.1 shows an analysis of variance comparing the baseline performance to the optimizer utilizing automatic function definition and curriculum control. For a confidence value of $\alpha = 0.05$, there is a very significant difference between the optimizer utilizing memory, and the baseline optimizer. Indeed, the optimizer utilizing memory was able to significantly out-perform the baseline by over 100 cases. Table 5.2 compares the baseline and full Meta-GP configurations, again for a confidence value of $\alpha = 0.05$. The Meta-GP optimizer drastically out-performs the baseline by an average of over 250 cases. This is most likely due to the extensive training and library of training that was constructed and stored in the system's function library. For completeness, the analysis between the optimizer using memory and the Meta-GP optimizer are compared in Table 5.3. Again, there is significant difference between the two optimizers, though it is much closer than the difference between both optimizers and the baseline.

Figure 5.14 shows the average evolutionary profiles for the different algorithm configurations. It is worth noting that no algorithm configuration was able to approach

perfect performance on the 10-bit even parity function, further demonstrating the difficulty of evolving such a complex function.

Table 5.1. Analysis of variance comparing baseline GP optimizer and optimizer with automatic function definition and curriculum control.

Groups	*Count*	*Sum*	*Average*	*Variance*		
Baseline	1000	521096.5	521.0965	7.071009		
w/Memory	1000	623339	623.339	67504.14		
Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	30691889	1	30691889	909.2383	**6E-165**	3.846117
Within Groups	67443696	1998	33755.6			
Total	98135586	1999				

Table 5.2. Analysis of variance comparing baseline GP optimizer and meta-learning optimizer with selection, automatic function definition and curriculum control.

Groups	*Count*	*Sum*	*Average*	*Variance*		
Baseline	1000	521096.5	521.0965	7.071009		
Meta-Learning	1000	782666.7	782.6667	103230.6		
Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	19489263	1	19489263	377.5612	**3.58E-77**	3.846117
Within Groups	1.03E+08	1998	51618.82			
Total	1.23E+08	1999				

Table 5.3. Analysis of variance comparing GP optimizer with only automatic function definition, and meta-learning optimizer with selection, automatic function definition and curriculum control.

Groups	*Count*	*Sum*	*Average*	*Variance*		
w/Memory	1000	623339	623.339	67504.14		
Meta-Learning	1000	782666.7	782.6667	103230.6		
Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	1266439	1	1266439	14.83516	**0.000121**	3.846117
Within Groups	1.71E+08	1998	85367.35			
Total	1.72E+08	1999				

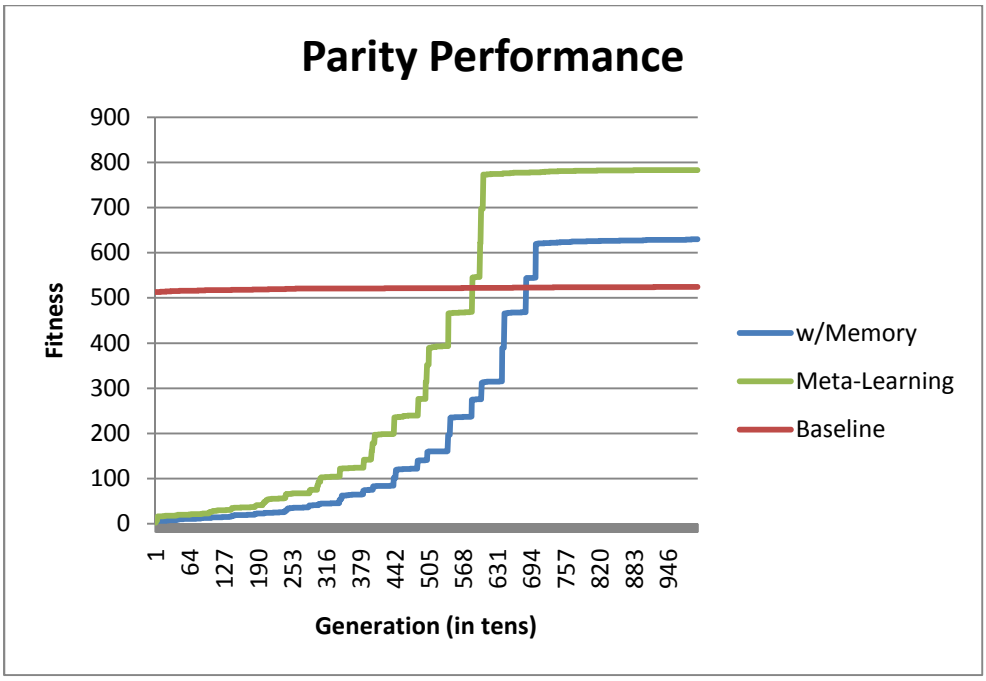


Figure 5.14. Parity performance of Meta-GP algorithm configurations.

5.4. META-GP EVALUATION: PAC-MAN

The Meta-GP architecture was evaluated on the game of PAC-MAN by first identifying the inputs and outputs of the PAC-MAN agent. Clearly, the agent has outputs or actions of 'move-up,' 'move-down,' 'move-left,' and 'move-right.' Inputs are less obvious, as the human player has total knowledge of the game state, including dots remaining to be eaten, location of ghosts, and the appearance of special items. In this conception of the PAC-MAN game, the agent is blind to all but its immediate surroundings. The agent inputs include whether or not a wall blocks each of the cardinal directions, whether or not a dot exists in one of the non-blocked cardinal directions, whether or not a ghost exists in one of the non-blocked cardinal directions, and whether or not a pill exists in one of the non-blocked cardinal directions. To further simplify the

game, special items (fruit) do not appear. The board size and shape is maintained for all experiments, but the existence of ghosts is controlled for training purposes. A PAC-MAN grammar is constructed, shown in Table 5.4.

Table 5.4. The PAC-MAN grammar.

```
N = {expr, op, pre_op, cond, var, term}
T = {AND, OR, NOT, IF
     MOVE_UP, MOVE_DOWN, MOVE_LEFT, MOVE_RIGHT,
     WALL_UP, WALL_DOWN, WALL_LEFT, WALL RIGHT,
     GHOST_UP, GHOST_DOWN, GHOST_LEFT, GHOST_RIGHT,
     DOT_UP, DOT_DOWN, DOT_LEFT, DOT_RIGHT,
     PWRUP_UP, PWRUP_DOWN, PWRUP_LEFT, PWRUP_RIGHT}
S = <expr>
P can be represented as:
  1. <expr> ::= <expr> <op> <expr> |
                <preop> <expr> |
                <cond> <expr> <expr> <expr> |
                <var> |
                <term>
  2. <op> ::= AND | OR
  3. <pre_op> ::= NOT
  4. <var> ::= WALL_UP | WALL_DOWN | WALL_LEFT | WALL RIGHT|
     GHOST_UP | GHOST_DOWN | GHOST_LEFT | GHOST_RIGHT |
     DOT_UP | DOT_DOWN | DOT_LEFT | DOT_RIGHT | PWRUP_UP |
     PWRUP_DOWN | PWRUP_LEFT | PWRUP_RIGHT
  5. <term> ::= MOVE_UP | MOVE_DOWN | MOVE_LEFT | MOVE_RIGHT
  6. <cond> ::= IF
```

Note that several new symbols and types of symbols have been introduced, particularly the conditional non-term ‘cond’, and the output non-term ‘term’. The conditional works like an ‘IF’ statement, evaluating the first child-expression for truth, then passing evaluation to either the second or the third expression, but never both, depending on the outcome of the first expression.

5.4.1. Experimental Setup. The PAC-MAN benchmark was evaluated using three primary experimental configurations, analogous to the configurations detailed in Section 5.3.3. The baseline configuration utilizes only the bare GP optimizer, the memory configuration adds automatic function definition, and the meta-learning configuration utilizes context recognition and memory along with curriculum control.

In the PAC-MAN benchmark, curriculum control takes the form of task prioritization – the goal is switched between finding and consuming dots, to avoiding ghosts, finding power-ups, and capturing ghosts. The memory configuration utilizes no curriculum control. The meta-learning curriculum control takes the form of the optimizer first being exposed to the dots-only game priority. In this scenario, ghosts and power-ups are removed from the game. After 1000 generations of exposure to the dots-only scenario, the priority is switched to avoiding and hunting ghosts, removing dots from the game. This is executed for another 1000 generations. For each of these configurations, a two-bit priority vector is presented to the MLP, one bit for dot priority, and one bit for ghost priority. At the end of each training phase, high-value functions are associated with each input vector. During the evaluation runs, both MLP inputs are set to 1, indicating that behaviors for both priorities should be utilized. The system is then allowed to evolve players that participate in the full game, including both dots and ghosts.

For all algorithm configurations, fitness is measured by the points accrued, according to the rules of PAC-MAN. To obtain a statistical sample, each experimental configuration was repeated 75 times.

5.4.2. Results. The results show that the addition of memory, and the training and integration of separately learned skills can significantly increase the fitness of evolved individuals playing the game of PAC-MAN. Tables 5.5 - 5.7 display the analysis of variance of algorithm configuration on evolved player performance. In all cases, a confidence value of $\boldsymbol{\alpha = 0.05}$ was used. Comparing the baseline configuration to the memory configuration in Table 5.5, the difference is small but significant for the given confidence value. The difference between baseline, memory, and meta- learning configurations is much greater, shown in Table 5.6 and Table 5.7. This strongly suggests that the curriculum and recognition methods significantly improve the performance of evolved PAC-MAN players, further illustrating the power and capabilities of the Meta-GP architecture, and meta-learning in general.

This trend is confirmed in Figure 5.15, where the average evolutionary profiles for the algorithm configurations are displayed. Once again, it is clear that the meta-learning configuration drastically improves performance.

Table 5.5. Analysis of variance comparing baseline and memory configurations.

Groups	*Count*	*Sum*	*Average*	*Variance*		
w/Memory	75	182116.7	2428.223	6236.895		
Baseline	75	163445.2	2179.27	3577.224		
Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	23241.64	1	23241.64	4.736367	**0.031117**	3.90506
Within Groups	726244.8	148	4907.059			
Total	749486.4	149				

Table 5.6. Analysis of variance comparing memory and meta-learning configurations.

Groups	*Count*	*Sum*	*Average*	*Variance*		
w/Memory	75	182116.7	2428.223	6236.895		
Meta-Learning	75	311396.2	4151.949	23482.34		
Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	1114212	1	1114212	74.98258	**7.49E-15**	3.90506
Within Groups	2199223	148	14859.62			
Total	3313436	149				

Table 5.7. Analysis of variance comparing baseline and meta-learning configurations.

Groups	*Count*	*Sum*	*Average*	*Variance*		
Baseline	75	163445.2	2179.27	3577.224		
Meta-Learning	75	311396.2	4151.949	23482.34		
Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	1459299	1	1459299	107.8583	**2.55E-19**	3.90506
Within Groups	2002408	148	13529.78			
Total	3461707	149				

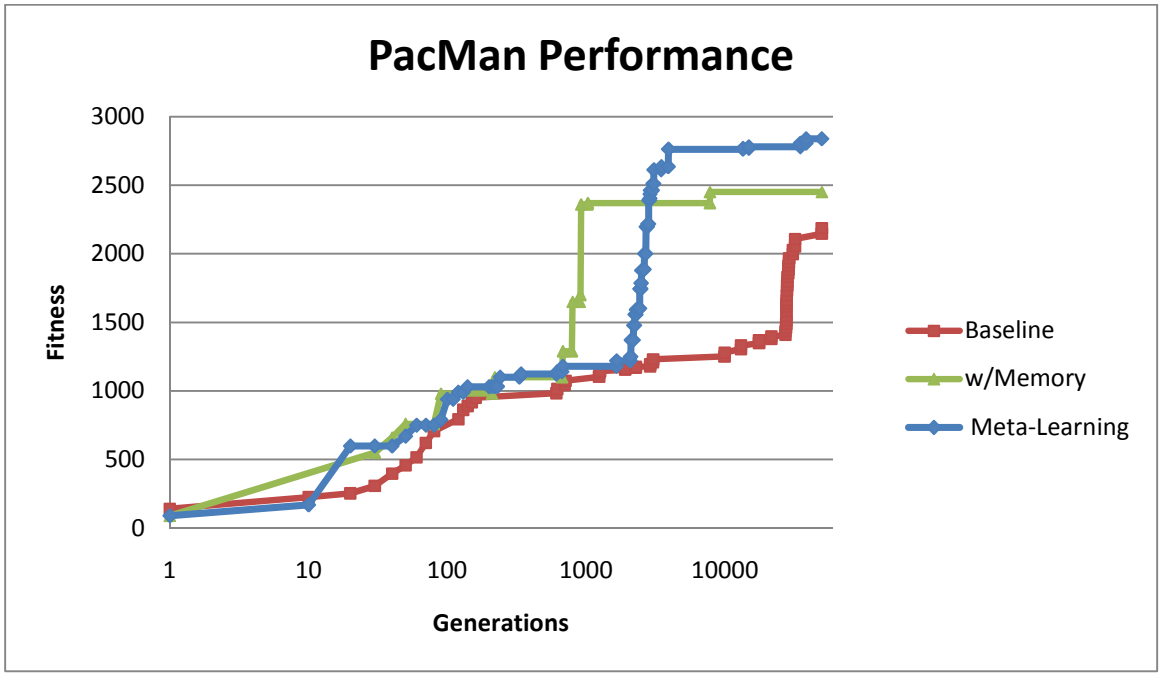


Figure 5.15. Meta-GP performance on the PAC-MAN benchmark.

6. META-LEARNING TRAVELING SALESMAN SOLUTION

The Traveling Salesman Problem (TSP) is a standard combinatorial optimization problem used to evaluate optimization methods [56, 57, 60, 61, 73, 91-98]. TSP optimization algorithms have a wide range of applications including job scheduling, DNA sequencing, traffic management, and robotic path planning. To further illustrate the capabilities of the meta-learning design paradigm, an example is presented using instances of the TSP. In many of these applications, it is not completely necessary to determine the optimal solution, thus heuristic methods are used to provide a good quality solution as fast as possible. It is with this in mind that meta-learning is applied to the traveling salesman problem.

To apply meta-learning to the TSP problem, the schema of the problem must be identified. Here the schema takes the form of the ordering of points in a tour. The addition of a clustering method to divide and conquer the TSP has been shown to greatly accelerate the solution of the TSP [36]. With this addition, the overall schema for the optimizer consists of the combination of cluster templates, tour point ordering, and the locations of points. This schema must be generalized to create a meme, which is trivial for the cluster templates, but more challenging for the tour ordering and point locations. The problem is further complicated by the necessity to generalize tours to be applicable over multiple scales.

For this application, a meme consists of an ordered tour. To create the meme, the centroid of the group is calculated and subtracted from each point, making the centroid the origin of the group. The coordinates of each point are then normalized by distance from the origin. This projects the points into unit-space, and allows comparisons across

multiple scales. Each TSP-meme serves as a pre-optimized tour template. Each point in the TSP-meme can represent a real point in the problem instance, or the centroid of a group of points, itself represented by a meme.

Given an instance of the TSP, the meta-TSP algorithm utilizes a clustering method to divide the problem into sub-problems, and divides those sub-problems into sub-sub problems and so on, until a threshold for sub-problem size is reached. The relationships between sub-problems are recorded in a tree-representation. Each of these sub-problems is generalized, and compared against the recorded memes for existing solutions.

The recognition mechanism must be able to detect structurally similar sub-problems. The matching mechanism compares two normalized sub-problems by finding the nearest corresponding points between the memes, and calculating the mean error between these points.

If a match is found in memory, the existing meme-solution (a point ordering) is copied to the current sub-problem. If no match exists in memory, the sub-problem is solved as accurately as possible. With a small enough problem threshold, exact solutions to sub-problems can be found, depending on computational resources available. The sub-problem is then stored in memory as a new meme. After all the sub-problems are solved, they are combined into a global tour by collapsing the problem-tree, and utilizing a simple constant-time merge algorithm.

To illustrate this process, an example is given utilizing a simple instance of the TSP, shown in Figure 6.1. A first pass of clustering is shown in Figure 6.2. Note that cluster M3 contains many points, and that a single point has been left out of the clusters

for illustrative purposes. A second pass further divides cluster M3 into clusters M5, M6, and M7, as shown in Figure 6.3. The final clustering pass assigns all clusters to a global cluster, M8, in Figure 6.4. The hierarchy of clusters, and thereby sub-problems, is denoted by the cluster tree in Figure 6.5.

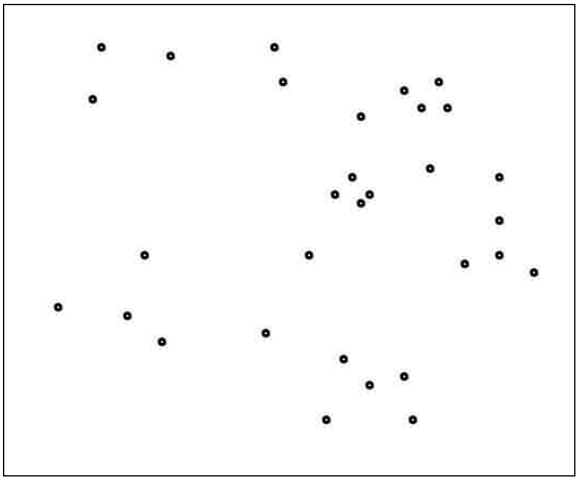


Figure 6.1. Small TSP instance of approximately 30 points.

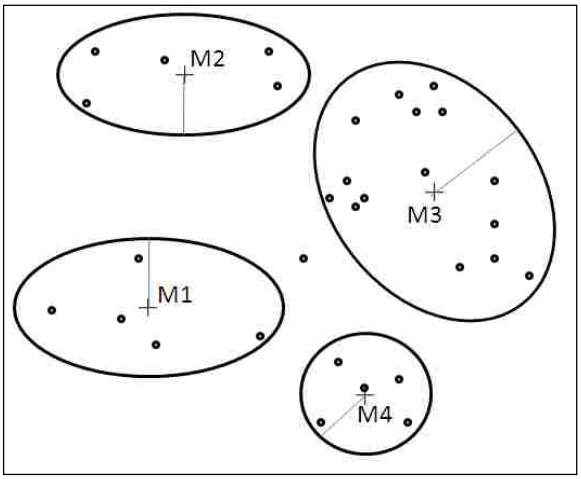


Figure 6.2. TSP instance after first clustering pass. Each cluster initializes a meme, labeled with "M#" and a "+" denoting the centroid.

At this stage, each sub-problem is optimized independently, as shown in Figure 6.6. Note that some of the sub-problems contain references to other sub-problems, particularly M3 and M8. The centroids of sub-problems are utilized for optimization and solution, representing sub-problems as a whole. During the course of optimization, each sub-problem is normalized, and compared with previously computed, normalized solutions in the memory. These memes can be stored across instances, building a large library of pre-computed solutions that can be deployed to yield high quality solutions rapidly. Sub-problems of a global problem instance can be thought of as new problem instances, and pre-computed solutions that are generated during the calculation of a global instance can be applied across sub-problems.

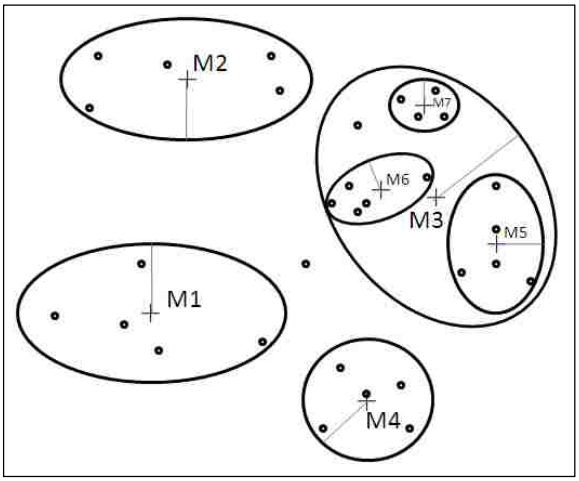


Figure 6.3. Second clustering pass. Note the new clusters, M5, M6, and M7.

For example, the normalized versions of M2 and M4 would be very similar in structure, and once M2 is computed, the structural similarity of the sub-problems would

be recognized, and the ordering of points for M4 need not to be computed, only copied from M2 to M4. The same process applies across scales and global problem instances.

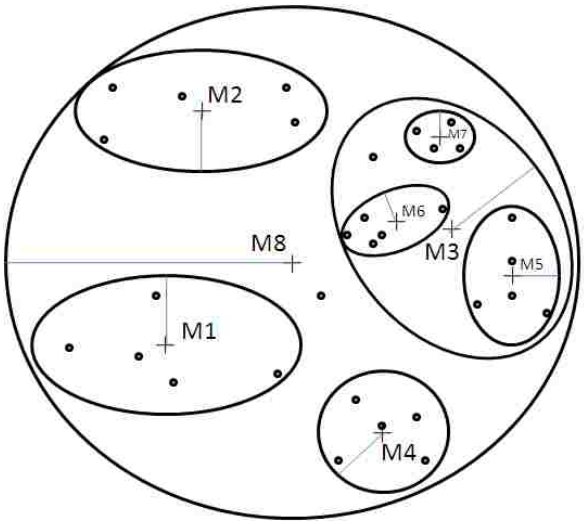


Figure 6.4. Final clustering pass, with global cluster M8

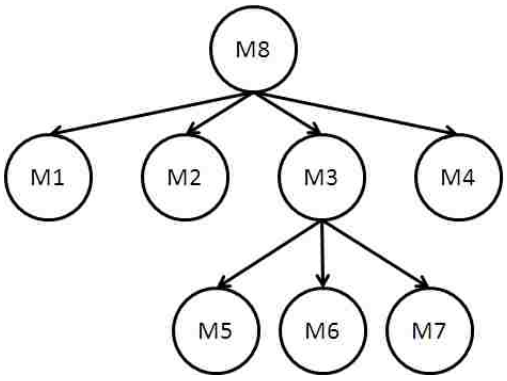


Figure 6.5. Tree of sub-problems (clusters).

When all sub-problems are completed, the problem hierarchy is collapsed by de-referencing sub-problems and incrementally merging them with higher level tours. Figure 6.7 shows the final merge of all complete sub-tours into a final tour. The completed tour is shown in Figure 6.8.

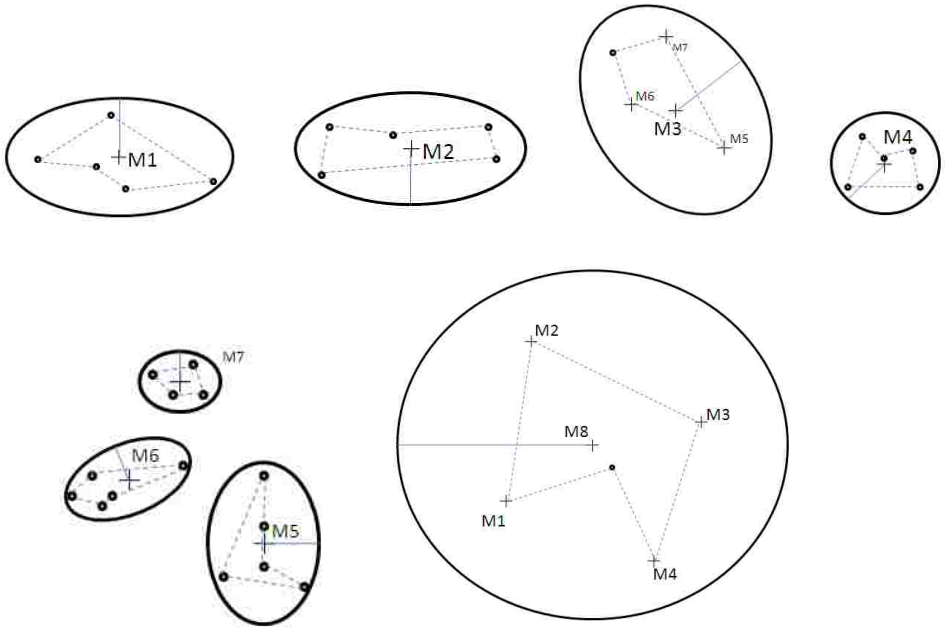


Figure 6.6. Completed memes, M1 through M8. Super-clusters reference the centroids of sub-clusters. Note that memes M2 and M4 are similar in structure, but not scale.

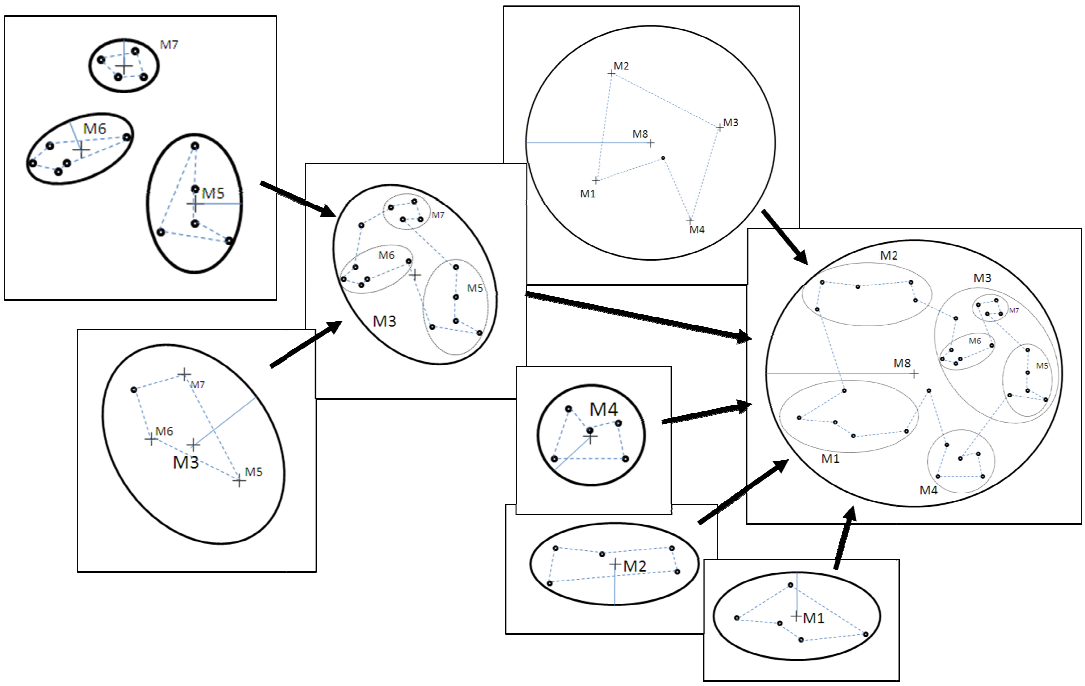


Figure 6.7. Merger of memes into a final tour.

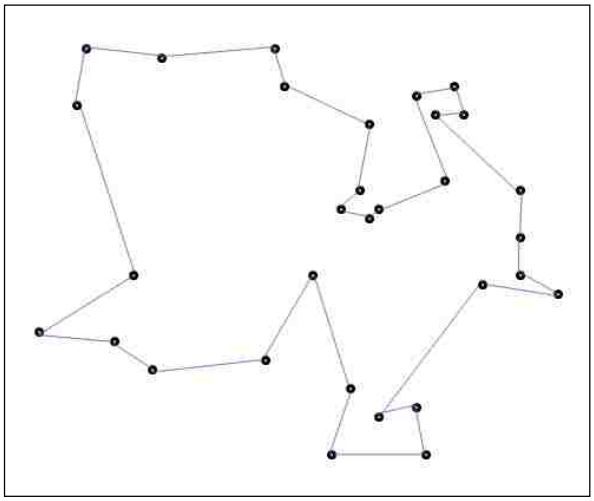


Figure 6.8. Completed tour.

6.1. META-TSP ALGORITHM

The design of the Meta-TSP algorithm is based on the concept of a centroid – the mean component values of all points in a set, this is also known as a center-of-mass. The centroid is used in the Meta-TSP algorithm to represent a set of points, and it is an assumption (and indeed source of error) in the algorithm design that calculations operating on a centroid result in useful information that can be used to make decisions about how the set represented by the centroid can be organized in large-scale structures. The primary failing of the centroid representation is that outliers can significantly bias the value of the centroid. It is vitally important that point-sets are chosen such that outliers are minimized, and for this reason an ART variant 2D point clustering method is utilized.

Another assumption of the algorithm is that the most-valuable ordering of a point set is 'transportable' to another point set of equal size and similar, if not equal, position. It is a subject of future research to analytically determine the thresholds and qualities of

similarity to which this assertion does and does not apply. Here, this assertion is studied experimentally by varying the threshold of similarity that is accepted to activate a stored tour/ordering pair.

The Meta-TSP algorithm consists of three primary stages – recursive generation of a hierarchy of sub-tours, local optimization of sub-tours, and merging of sub-tours into a final tour. There are a few operations that are used in several places in the algorithm – Normalization, casting the components of a point-set into the [0,1] domain, Hash, used to calculate a key to store and retrieve a tour-ordering, and comparison, determining the similarity between two normalized tours. Additionally, the segmentation method, an Adaptive Resonance Theory variant for 2D Euclidean spaces is presented, as well as analysis of expected algorithmic complexity and overhead of memory operations.

These algorithmic components are described, and then evaluated on a set of standard TSP instances from the TSPLIB database. Experiments are also conducted to study the influence of adding memory to the optimization process, comparing memory-less Meta-TSP instances, and the effect of the order of presentation in memory on optimization. These results are presented and discussed at the end of this section.

6.1.1. Tour Normalization. Tour normalization is performed to provide a general representation of a tour; its ordering, and the relative positions of the component points.

Towards this goal, the tour normalization procedure transforms the components of a set of points from $\Re$ to the range [0,1]. Given a non-empty set of points *P,* and the sets of their components *X* and *Y* such that $p = (x, y), x \in X, y \in Y, \forall p \in P$. *P* must also have more than one element. The minimum elements of X and Y are first found, denoted as $x_{min} \in X, y_{min} \in Y$, where x_{min} and y_{min} are calculated in Equation 12.

$$x_{min} = \min(X)$$
$$y_{min} = \min(Y) \tag{12}$$

For each point $p \in P$, the set P' is calculated by subtracting x_{min}, y_{min} from each p, detailed in Equation 13, this shifts every point into the positive quadrant. From the set P', the maximum magnitude is found using Equation 14. This maximum magnitude then divides the components of P' shown in Equation 15.

$$p' = (p_x - x_{min},\ p_y - y_{min})$$
$$p' \in P', \forall p \in P \tag{13}$$

$$m_{max} = \max\left(\sqrt{{p'_x}^2 + {p'_y}^2}\right), \quad \forall p' \in P' \tag{14}$$

$$||p|| = \left(\frac{p'_x}{m_{max}}, \frac{p'_y}{m_{max}}\right)$$
$$p' \in P', ||p|| \in ||P|| \tag{15}$$

This results in the normalized tour $||P||$, with no component or point magnitude outside of the range [0,1]. This normalization procedure places bounds on the values that a set of points can take, allowing simple comparison, while preserving the structure of the point set and their relative positions.

6.1.2. Tour Hash. In order to store and retrieve tour-orderings efficiently, a hash multi-map data structure is utilized.

To take advantage of the hash-map, a method for generating keys based on tours must be developed. Ideally any key-generating method should generate keys as

uniformly and as uniquely as possible. As the hash-map is being used as a memory, which will require the activations of map entries by similar input tours, the uniqueness requirement may be relaxed somewhat. The components of the tours, and the tours themselves by extension, are likely to be randomly scattered throughout the tour-space, satisfying the uniformity requirement.

A common method of calculating the key of a data element is the Zobrist hash, most often used in chess-playing programs to encode and store the values of game-states [99]. In the Zobrist hash, every possible combination of piece and position on the chess board is assigned a random value. A key is generated by applying the XOR function to the values of the pieces on the current board-state. This method served as the inspiration for the Tour-Hash method.

Similar to the Zobrist method, the Tour-Hash first creates a table of 64 randomly generated values. These 64 partial keys each represent one bin in the normalized tour-space. To encode a point, the axes of the normalized space are divided into 8 bins each, a point's position on each axis encoded by 3 bits corresponding to the bin in which the point-component falls. This is calculated by multiplying each normalized point component by 8, and flooring the result. For example, the point (0.1, 0.7) is mapped to the binary values (000, 011). These 3-bit pairs are then concatenated into a single 6-bit value with the x-value in the most significant location. These bin-codes are calculated for each point in a given input tour. The bin-code of each point is used to retrieve the corresponding entry from the partial key table. For each tour-point, the XOR function is applied to these partial keys to produce the final key.

The tour-hash is not guaranteed unique, as the randomly generated partial key table may have repeats, though this is not likely. Additionally, the XOR combination of partial keys may produce the same result for different tours. To compensate for this, a multi-map is used, and secondary checks for matching tour-sizes and similarity thresholds ensure that recovered stored tours match the input tour as closely as necessary. Using a 32 bit (long) representation, the key format can store billions of unique keys.

6.1.2.1 Hash overhead. As the tour-hash calculation procedure is simple, and the hash multi-map data structure has amortized constant time insertion and retrieval, the computation overhead for utilizing the hash-map is very low. The overhead of other data-structures such as lists would be much greater, as retrieval in a list is linear with the number of elements, and the comparison operation itself is linear with the size of tours.

6.1.2.2 Maximum difference of collisions. The discrete nature of the tour-hash method places a bound on the possible difference between two tours of the same size, generating the same key. This difference is based on the size of the bins that divide the normalized tour-space, and the fact that bins are square. The maximum collision difference is calculated by finding the distance between two opposite corners of a bin. This is most easily calculated using the (0, 0) bin, developed in Equation 16. This maximum difference is multiplicative for the size of a tour.

$$d_{max} = \sqrt{\left(\frac{1}{8} - 0\right)^2 + \left(\frac{1}{8} - 0\right)^2} = \sqrt{2\left(\frac{1}{64}\right)} = 0.176 \tag{16}$$

Conversely, two points separated by less than d_{max} are not guaranteed to fall in the bin. Indeed, two points can be separated by a very small amount if they fall on

opposite sides of a bin-border. This is a flaw in the tour-hash method that it is hoped will be addressed with further research.

6.1.3. Tour Comparison. The tour comparison procedure is used to verify that tours retrieved from memory are sufficiently similar in structure to an input tour. If the input and retrieved tours are of the same size, the best ordering of the input onto the retrieved tour is also discovered. This ordering can then be used to reorder the input tour, taking advantage of stored computation in the form of a memorized tour.

To measure the similarity between two tours, it is necessary to first check that they are the same size. Tours of differing size, even with similar structures, can have greatly differing optimal orderings, thus comparisons are restricted to tours of the same size. Given two tours of the same size, each tour point should be paired with the closest corresponding point from the opposite tour. This problem is analogous to the assignment problem, and is itself a combinatorial optimization problem.

In order to avoid combinatorial explosion, and thereby erasing all advantage of memory, a naïve comparison method is utilized that compares a given point to each unassigned point in the other tour. The given point is assigned to the opposing tour-point that is closest. This process of comparison and assignment is repeated for all points, resulting in a $O(n^2)$ comparison process, rather than a guaranteed optimal $O(n!)$ process.

6.1.4. 2D Euclidean ART. In order to divide and conquer a large tour into smaller sub-problems, a method of tour subdivision is needed.

For clustering 2D points, several methods have been used previously, most commonly the K-means algorithm [73], as this method can utilize any metric for comparing data elements.

However, K-means has the drawback of requiring the number of clusters be set beforehand, reducing the capability of the method to be wholly data-driven. Additionally, K-means is inherently an $O(n^2)$ algorithm. A much more desirable, and indeed more popular framework is that of Adaptive Resonance Theory (ART), which exhibits fast ($O(n)$ complexity), stable, and completely data-driven operation. However, ART frameworks do not typically deal well with data points represented in a 2D Euclidean space. For this reason a modification of ART is presented here, enabling fast, data-driven clustering with minimal user-set parameters.

The 2D Euclidean ART (E-ART) modification operates similarly to that of the typical ART structure, with F1 and F2 nodes, category match, vigilance test, and resonance phases. The modifications to E-ART lie in the format of input, and the way that inputs are compared. E-ART takes as input an unordered set of points that have been normalized using the tour normalization method described previously. This normalization and bounding of components into the [0,1] range allows scale-independent comparisons that are bounded to the [0,1] range. This enables the vigilance parameter to remain in the same [0,1] range.

The E-ART comparison method replaces binary or fuzzy operators with the Euclidean distance calculation, shown in Equation 17, where the input is represented by I, and the template is represented by w^j.

$$T(j) = 1 - \sqrt{(w_x^j - I_x)^2 + (w_y^j - I_y)^2} \tag{17}$$

As this Euclidean comparison produces a result that is already in the [0,1] range, a separate Equation for calculating vigilance match is not needed, so the best category

match is simply tested for being greater than the given vigilance value. If this best category does not pass the vigilance test, a new node is committed using the input as the initial value.

6.1.5. Tour-Hierarchy Creation. The normalized Euclidean ART method can be used to construct a hierarchy of sub-tours by recursively normalizing and further subdividing clusters using the same vigilance value. The process begins by normalizing the entirety of a problem instance, and E-ART is used to divide the problem instance into a set of clusters, each cluster containing a sub-set of the total problem instance, illustrated in Figure 6.9. For each sub-set, a centroid is calculated using the non-normalized points. This centroid is used to represent the subdivided point-sets. If a sub-set is larger than a maximum threshold, the sub-set is then normalized and clustered using a new E-ART unit, repeating recursively, creating a tree of sub-tours. This process is illustrated in Figure 6.9. The lines connect the centroids of the colored sub-tours, showing the hierarchy structure. Division hierarchy increases from left to right, top to bottom. In the top-left of Figure 6.9, a first level division is shown, in top-right, a second level division, in the bottom-left, a third-level division, and finally in bottom-right the multi-level division of a TSP instance, with maximum sub-problem threshold of ten points.

The set of centroids at each level of the hierarchy are used as tours, representing the optimization of high-level structure, while the final sub-threshold tour sets, at the bottom of the hierarchy are optimized as low-level detail. Each level of the tour-hierarchy (and thereby a tour of centroids, called *hierarchy tours* from this point forward) adds to the total set of sub-problems to be optimized, but it is expected that the hierarchal

structure, optimization and merge procedures will mitigate the increased total problem size.

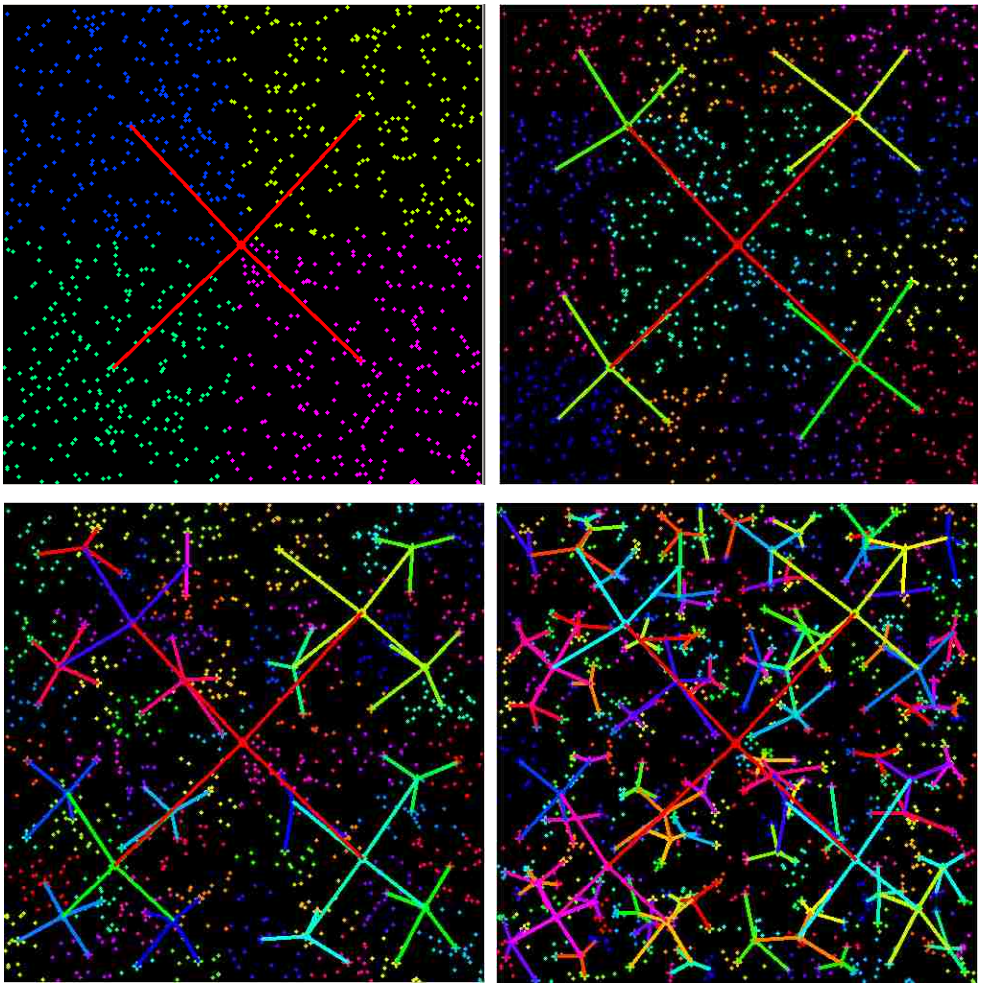


Figure 6.9. Division of a TSP instance.

6.1.6. Exhaustive Optimization. The goal of the hierarchal subdivision process is to divide the TSP instance into small pieces that may be precisely optimized, while maintaining structural information.

It then becomes possible to use exact TSP solution methods. In the current implementation, a simple permutation-based algorithm is utilized to optimize sub-tours.

This results in optimal solutions to sub-problems, but limits the size of a sub-problem to be less than 12 points for reasonable computation on modern hardware. It is left as a topic of future research to explore the effect on structure and performance of more efficient exact methods (such as branch and bound, or dynamic programming) as well as heuristic methods.

In the optimization phase, hierarchy tours are solved from the top-down, starting with the first-level tour-cycle, the tour-tree is traversed depth-first, optimizing tours at each node. It should be noted that through hierarchal construction, only the first-level hierarchy tour is a cyclic tour. All lower-level tours, hierarchy tours or sub-tours, are *tour segments* – where the problem switches from finding the shortest cycle to finding the shortest path between two fixed points. In a tour segment, the endpoints of a path are specified. These endpoints are chosen to be the ordered centroids of adjacent higher-level hierarchal tours. The optimization of the top-level tour influences the optimization of the tour-segments in the levels below, continuing until the bottom-most sub-tour segments are reached. This process is illustrated in Figure 6.10. Colored lines connect ordered tours within hierarchy levels.

6.1.7. Merge Operation. Once all tour sub-problems have been solved, they are merged into a final tour that contains all points in the original problem instance.

Due to the proliferation of Divide and Conquer-based algorithms in the TSP literature there are many methods for merging sub-tours, but very few methods reference the merging of tour *segments*. Existing methods for merging tours have $O(n^2)$ complexity, as they compare all combinations edges in the two tours to be merged in

order to find the best merge process. This is further improved by enforcing a small neighborhood [60].

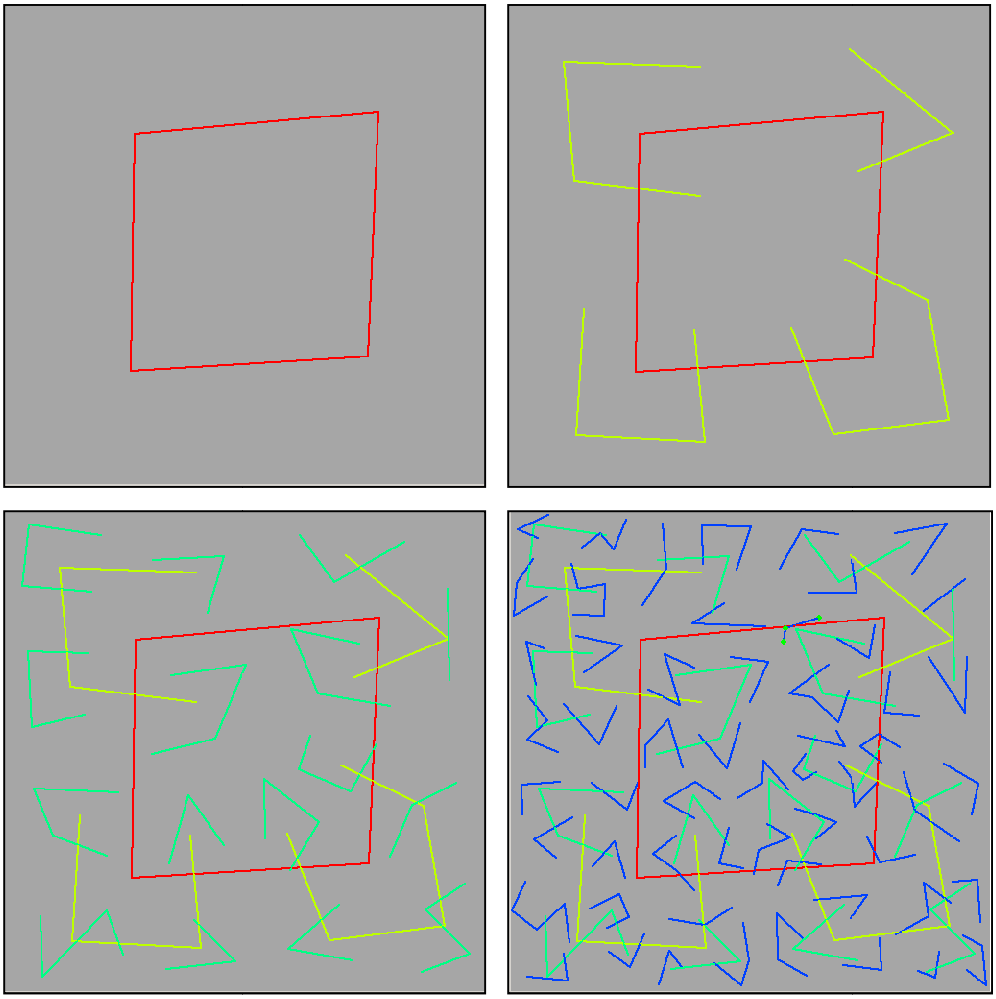


Figure 6.10. Optimization descending the tour hierarchy, left to right, top to bottom.

In Meta-TSP, tour-segments are already ordered by higher-level optimization, so the merge operation naturally follows by simply connecting endpoints of segments at every level. This leads to a very fast, constant-time merge operation at each hierarchy level, so that the overall complexity of the merge operation is $O(n \log(n))$, where n is the

total problem size, and *m* is the maximum sub-problem limit. An example of this process is illustrated in Figure 6.11.

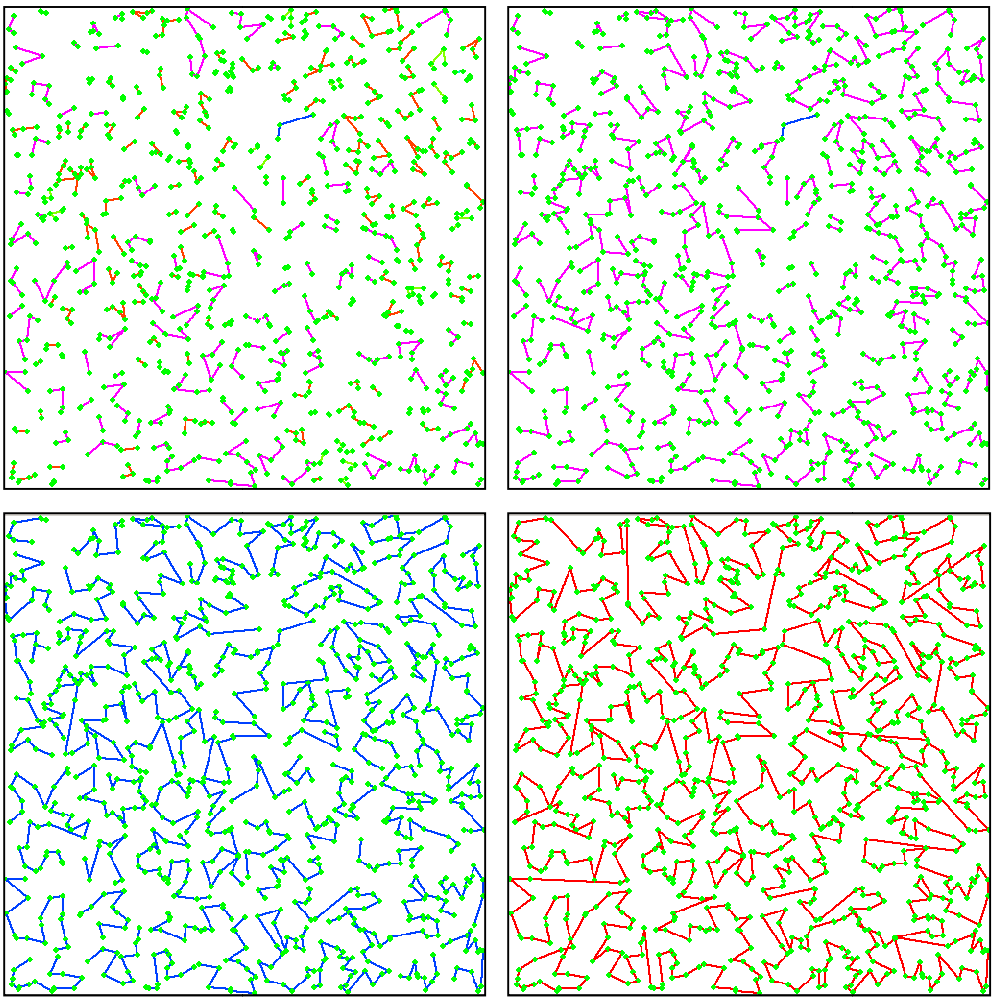


Figure 6.11. Bottom-up merging of tours.

Tour-Points are marked in green, colored lines represent the ordering of the sub-tours. The merge process ascends the hierarchy from left to right images, top to bottom, starting with the bottom-most hierarchy level (blue and violet lines), to the top-most, final tour (red).

6.1.8. Algorithmic Complexity Analysis. The run-time complexity of the Meta-TSP algorithm is non-trivial to calculate, as the run-time greatly depends on the interplay of the clustering method with the characteristics of a tour-instance. Integrating memory further complicates the analysis, as the complexity of the algorithm is then affected not only by the current problem instance, but the algorithm's *experience* or history of problem solution. The complexity of the algorithmic components has been discussed separately, so all that remains is to examine how these components interact in order to determine the overall run-time complexity. For notational convenience, several parameters are defined in Table 6.1.

Table 6.1. Definition of complexity analysis parameters.

Parameter	Definition
n	Size of TSP instance.
m	Sub-tour limit. The maximum allowed size of a sub-tour.
l	Memory size.
p	Memory hit-rate.

In Section 6.1.5 it was shown that the tour hierarchy construction process is completed in $O\left(n \log_m(n)\right)$ time. This tour-hierarchy can be thought of as an m-ary tree with height $\log_m(n)$ and the maximum number of internal (non-leaf, non-root) nodes is $m^{\log_m(n)} - 1$ [100]. By the hierarchal construction process, the number of leaves is $\frac{n}{m}$. Thus, the sum of leaf tours and hierarchy tours is $m^{\log_m(n)} - 1 + \frac{n}{m}$. Utilizing a

permutation based exact solution method, each of these sub-tours takes $O(m!)$ to solve, giving $O\left(n\log_m(n) + \left(m^{\log_m(n)} - 1 + \frac{n}{m}\right)m!\right)$ time to create and solve the tree of sub-tours. To reach a final tour, these sub-tours must be merged together, a simple operation depending only on the number of nodes at a given level. As this merge procedure is applied at all non-leaf depths, the time complexity becomes Equation 18.

$$O\left(n\log_m(n) + \left(m^{\log_m(n)} - 1 + \frac{n}{m}\right)m! + \left(m^{\log_m(n)} - 1\right)\right) \tag{18}$$

This is the complexity of the algorithm without the use of memory. Memory is integrated during the optimization phase, where a successful retrieval from takes the place of a tour optimization. However, the computational overhead of hash-key calculation and tour-comparison must be added. As a hash-map is utilized, the memory retrieval process is amortized constant time, so it may be effectively ignored. The hash-key calculation is linear with the maximum sub-tour size, $O(m)$. If a corresponding tour exists in memory, the input and memory tours are compared utilizing a $O(m^2)$ operation. It is likely that these two operations occur for every sub-tour. The frequency of memory access and retrieval resulting in a successfully remembered tour is represented by the memory hit-rate, p, which is influenced by the memory threshold algorithm parameter. By integrating memory affects into the overall complexity calculation, Equation 19 results.

$$O\left(n\log_m(n) + \left((1-p)\left(m^{\log_m(n)} - 1 + \frac{n}{m}\right)\right)m! + \left(m^{\log_m(n)} - 1\right) + m\left(m^{\log_m(n)} - 1 + \frac{n}{m}\right) + m^2\left(m^{\log_m(n)} - 1 + \frac{n}{m}\right)\right) \tag{19}$$

To find the overall complexity class of the algorithm some simplification is necessary. It can be seen that the total number of tours, $\left(m^{\log_m(n)} - 1 + \frac{n}{m}\right)$ can be simplified greatly, as *m* is a constant, and $m^{\log_m(n)} = n$, giving $O(n - 1 + n) \approx O(n)$. Applying this simplification throughout results in Equation 20.

$$O\big(n \log_m(n) + \big((1-p)n\big)m! + \mathrm{n}\big) \tag{20}$$

Examining Equation 20, it is seen that as the hit-rate improves, the optimization phase of the algorithm should become less significant and the other two phases of hierarchy creation and merger dominate the run-time complexity. Most notably, for large problem instances, the run-time complexity is primarily driven by the hierarchy creation procedure, placing the Meta-TSP algorithm firmly in the $O(n \log(n))$ complexity class. Unfortunately, the memory hit-rate is dependent on the history of an implementation's experience, and the structure of the data itself, making more precise analysis extremely difficult.

6.2. META-TSP EVALUATION

In order to evaluate the actual run-time performance of the Meta-TSP algorithm, a test-bed consisting of 20 problem instances was sampled from the TSPLIB benchmark library [101]. The test-bed consists of problem instances from a variety of sources, natural and artificial, ranging in size from 48 to 100K points. All but the mona-lisa100k problem have been solved to optimality, and even this remaining problem has a solution that is within 0.0029% of the best-known lower-bound. This body of data allows for a variety of benchmark comparisons, including time-to-solve, and how close the tested

methods come to the optimal solutions. Table 6.2 provides a brief description of the test-bed problems.

The real-world TSP instances are particularly interesting, as they exhibit inherent structure and form not found in the artificially generated TSP instances. Most notable of the real-world problems is the Mona Lisa problem, created by Robert Bosch as a 100,000-point continuous line-drawing, shown in Figure 6.12.

Table 6.2. TSP instance descriptions.

Problem Name	Size	Source / Description	Optimal Cost
att48	48	48 capitals of the U.S.	33523.7
eil51	51	Artificial	429.98
st70	70	Artificial	678.59
eil76	76	Artificial	545.38
pr76	76	Artificial	108159.43
kroA100	100	Artificial	21285.44
kroC100	100	Artificial	20750.76
kroD100	100	Artificial	21294.29
rd100	100	Artificial	7910.85
eil101	101	Artificial	642.30
lin105	105	Artificial	14385.99
ch130	130	Artificial	6110.86
ch150	150	Artificial	6532.28
tsp225	225	Artificial	3916.00
a280	280	PCB Drilling Problem	2586.76
pcb442	442	PCB Drilling Problem	50783.54
pr1002	1002	Artificial	259045.61
pr2392	2392	Artificial	378062.82
pla85900	85900	Programmed Logic Array	142382641
mona-lisa100k	100000	DaVinci's Mona Lisa	~5757005

6.2.1. Experimental Setup. To study the Meta-TSP algorithm and the effect of memory on TSP optimization three initial experimental configurations were used – Meta-TSP with no memory component, Meta-TSP using memory with ascending problem size, and Meta-TSP using memory with descending problem size.

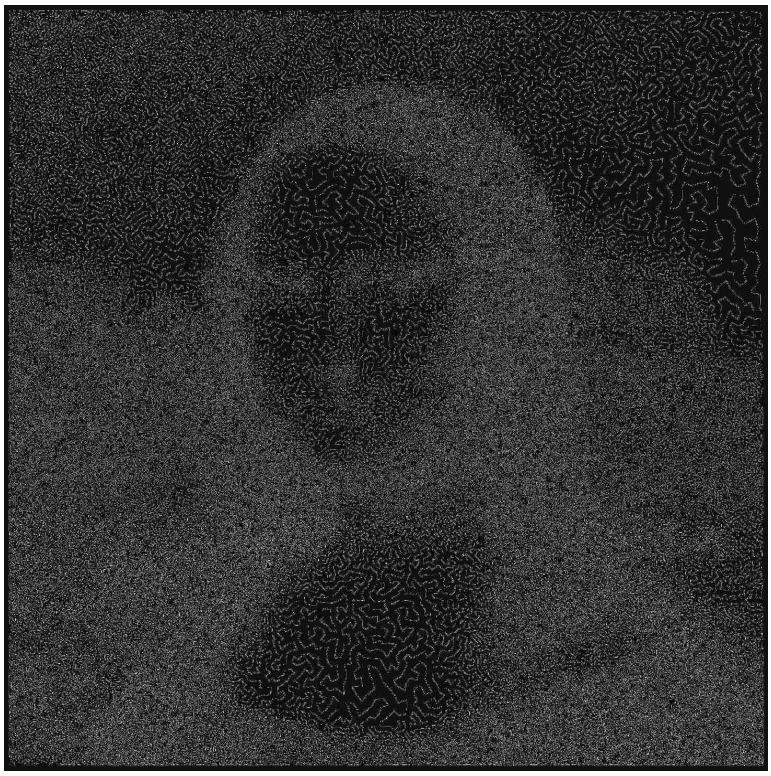


Figure 6.12. Leonardo da Vinci's Mona Lisa as a continuous line drawing.

The ordering of problem presentation was varied to examine the effect that this would have on the optimization process. In ascending presentation, the largest problems are presented after smaller problems have been solved and committed to memory, making a body of pre-optimized sub-tours available for rapid deployment. Conversely, in descending presentation the larger problems are presented without memory, and must perform optimization without a set of pre-optimized tours.

To examine the biasing effect of memory, a fourth configuration is utilized where the Meta-TSP method is *trained* on small problem instances by applying the optimal tour and allowing the algorithm to decompose and integrate sub-tour solutions into memory without utilizing the optimization procedure. Larger tours are then presented and the

algorithm is allowed to attempt solution utilizing known-optimal memory, recording the system performance.

For each of the major problem configurations, the algorithm parameters of maximum sub-tour size limit, E-ART vigilance, and for configurations with memory, the memory match vigilance were varied. The range and resolution of these parameters is presented in Table 6.3.

Each configuration of algorithm, parameters, and TSP instance was evaluated 30 times in order to obtain a reasonable statistical sample, as the E-ART method introduces a stochastic element to the otherwise deterministic Meta-TSP method. For each configuration the recorded performance metrics include the height of the generated tree, the sum of all sub-tour sizes (hierarchal and leaf tours), the count of sub-tours (hierarchal and leaf tours), completed tour cost, execution time, and for configurations using memory, the memory size, count of memory accesses, and count of memory hits.

Table 6.3. Meta-TSP experimental parameter configurations.

Parameter	**Range**	**Resolution**
Maximum Tour Size	3 to 8	1
E-ART Vigilance	0.2 to 1.0	0.1
Memory Match Threshold	0.1 to 1.0	0.1

With the availability of optimal TSP solutions for most of the benchmark problems and the integration of memory into TSP solution, it becomes possible to construct a supervisory TSP solution architecture. The supervisory TSP solution process takes as training a set of pre-optimized tours, deconstructs them into component sub-tours using the hierarchal clustering procedure, and then saves those sub-tours in memory for

later use. The evaluation phase proceeds as normal, except that the memory is already pre-initialized with known-optimal tours. This configuration is examined as a fourth experimental configuration. The supervisory TSP architecture is trained on the optimal tours of TSP instances listed in Table 6.4. After training, the full test-bed is applied in ascending order. This process is completed for all combinations of parameters, as detailed previously.

For all configurations and experiments, recorded measurements included the resulting cost, evaluation time, tour size (total points in tour, including those added by hierarchy), count of sub-tours, hierarchy depth, memory size, count of memory accesses, and count of memory hits.

Table 6.4. TSP instances used for supervisory training.

Training TSP Instances
att48
eil51
st70
eil76
pr76
kroA100
kroC100
kroD100
eil101
lin105
ch130
ch150
tsp225
a280
pcb442
pr2392

6.2.2. Results. The effect of algorithm parameters of ART vigilance, memory vigilance, and sub-tour size limit are examined relative to solution cost and execution time. Next, the effect of problem presentation order on cost and evaluation time is explored. Finally, the effect of integrating memory and overall algorithm performance is investigated.

Where analysis of variance is performed, tables present the probability that the null-hypothesis is true. The coloring of table cells is based on a confidence threshold of $\alpha = 0.05$, marked green if the content of the cell is below this threshold, indicating significance, or marked red if the content falls above this threshold indicating lack of significance. Each table presents the significance of effect (either cost or execution time) driven by an algorithmic control parameter. For example, Table 6.5 analyzes the effect of ART-vigilance on tour-cost. Each row corresponds to one vigilance setting, as does each column. The cell at the intersection of a row and column contains the probability that the null-hypothesis is true – the probability that the effect produced by the row-column parameter settings is insignificant.

6.2.2.1 Effect of ART vigilance. As the E-ART method is principle to the construction of the tour hierarchy, it was expected that the E-ART vigilance value would have a pronounced effect on the count of sub-tours, and similarly, the total tour size as well. Examining the results across all algorithm configurations, problem instances and controlling for E-ART vigilance, this is not the case, as there is only slight significant difference between the tour size and count of sub-tours between vigilance values. The results of this analysis are shown in Tables 6.5 - 6.7. For all algorithm configurations, most E-ART vigilance values do not significantly influence the resulting tour-cost.

However, for high vigilance values, this effect is significant, but undesirable, as cost begins to increase as clusters become small, and instance information is lost in the hierarchy construction. This trend is illustrated in Figure 6.13.

For large tours such as pla85900, the E-ART vigilance has a significant effect on the hierarchy depth, as shown in Figure 6.14. For vigilance values less than 0.5, the E-ART method is indiscriminant, creating few clusters at each level, necessitating more levels to create sub-problems below the sub-tour size threshold. At vigilance values greater than 0.9, many sub-tours would be created near the root, but the sub-tour size threshold limits the number of sub-tours created at each level. The high vigilance threshold then causes many levels to be created at the bottom of the tree, as clusters are further sub-divided. The ideal vigilance threshold range for short trees (and thus small hierarchies) appears to be the range 0.4 to 0.8.

Figure 6.15 details the proportion of total tour size after hierarchy creation by E-ART vigilance. Note that regardless of tour size, the proportion of input tour size to hierarchy tour size is generally between 1.5 and 1.65 with few exceptions. This indicates that the tour-hierarchy creation has relatively constant overhead, regardless of tour-size and vigilance setting. The effect of E-ART vigilance on execution time is illustrated in Figures 6.16 - 6.19. A general trend can be discerned from these figures that as E-ART vigilance increases, the execution time appears to follow proportional to the hierarchy depth in Figure 6.13. This is likely due to the interplay between sub-tour size and hierarchy depth. As sub-tours get larger, the hierarchy depth is decreased, and execution time increases by the factorial of sub-tour size due to the use of an exhaustive

optimization method. The analyses in Tables 6.8 - 6.10 confirm the significance of E-ART vigilance on execution time.

Table 6.5. Analysis of variance on effect of ART vigilance on cost for Meta-TSP without memory.

Art Vig	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.2	9.06E-01	8.53E-01	8.65E-01	9.65E-01	7.84E-01	4.19E-01	1.04E-01	1.85E-02
0.3		9.46E-01	9.58E-01	9.42E-01	6.95E-01	3.55E-01	8.20E-02	1.36E-02
0.4			9.88E-01	8.88E-01	6.46E-01	3.22E-01	7.11E-02	1.14E-02
0.5				9.00E-01	6.57E-01	3.29E-01	7.34E-02	1.18E-02
0.6					7.50E-01	3.95E-01	9.56E-02	1.65E-02
0.7						5.94E-01	1.76E-01	3.67E-02
0.8							4.10E-01	1.17E-01
0.9								4.51E-01

Table 6.6. Analysis of variance on effect of ART Vigilance on cost for Meta-TSP with memory and ascending order.

ART Vig	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.2	9.24E-01	7.53E-01	5.24E-01	2.98E-01	1.15E-01	1.40E-02	5.22E-05	1.63E-15
0.3		8.27E-01	5.88E-01	3.45E-01	1.39E-01	1.81E-02	7.76E-05	3.24E-15
0.4			7.47E-01	4.69E-01	2.07E-01	3.19E-02	1.85E-04	1.50E-14
0.5				6.88E-01	3.48E-01	6.80E-02	6.18E-04	1.35E-13
0.6					5.91E-01	1.54E-01	2.45E-03	1.90E-12
0.7						3.74E-01	1.24E-02	5.53E-11
0.8							1.05E-01	9.73E-09
0.9								2.71E-05

Table 6.7. Analysis of variance on effect of ART vigilance on cost for Meta-TSP with memory and descending order.

ART Vig	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.2	9.25E-01	7.41E-01	5.19E-01	2.95E-01	1.13E-01	1.32E-02	5.00E-05	1.87E-15
0.3		8.14E-01	5.81E-01	3.41E-01	1.36E-01	1.71E-02	7.38E-05	3.65E-15
0.4			7.52E-01	4.74E-01	2.09E-01	3.15E-02	1.88E-04	1.88E-14
0.5				6.88E-01	3.47E-01	6.62E-02	6.12E-04	1.60E-13
0.6					5.89E-01	1.50E-01	2.42E-03	2.22E-12
0.7						3.69E-01	1.24E-02	6.55E-11
0.8							1.07E-01	1.19E-08
0.9								3.04E-05

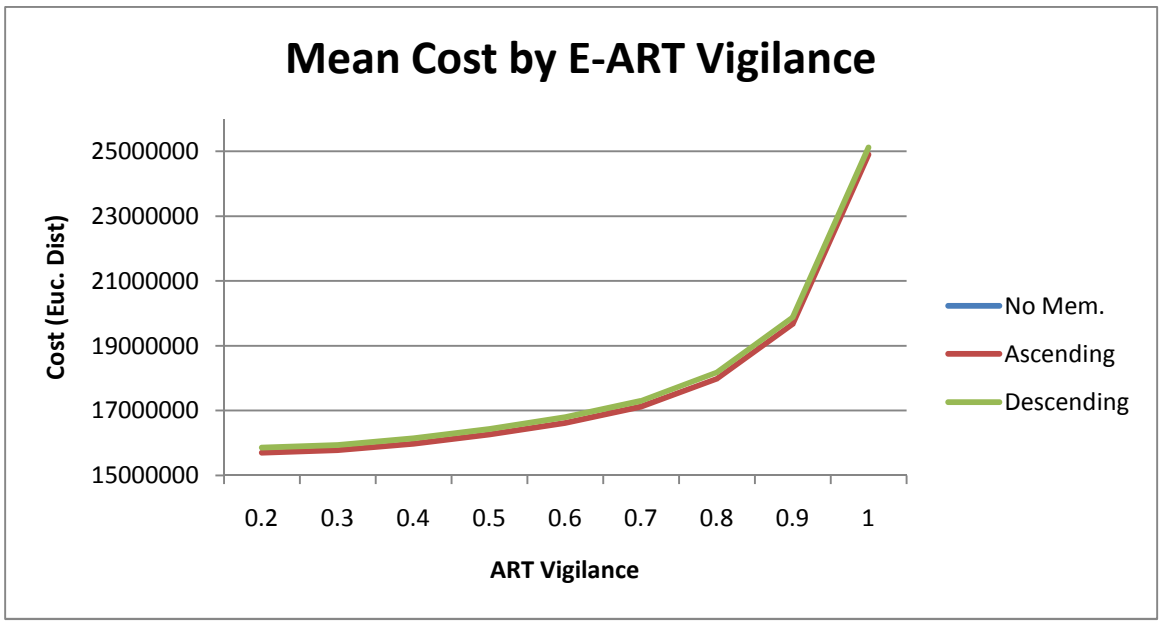


Figure 6.13. Cost by E-ART vigilance for algorithm configurations.

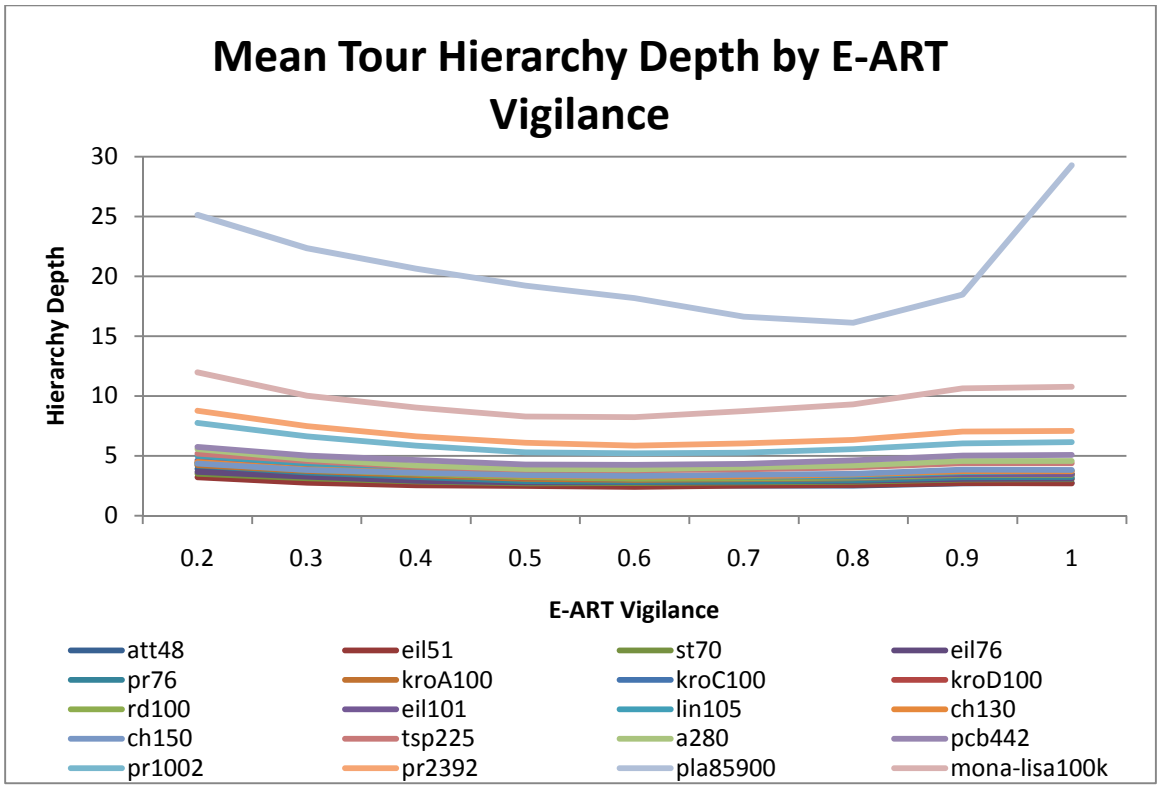


Figure 6.14. Mean tour hierarchy depth by E-ART vigilance.

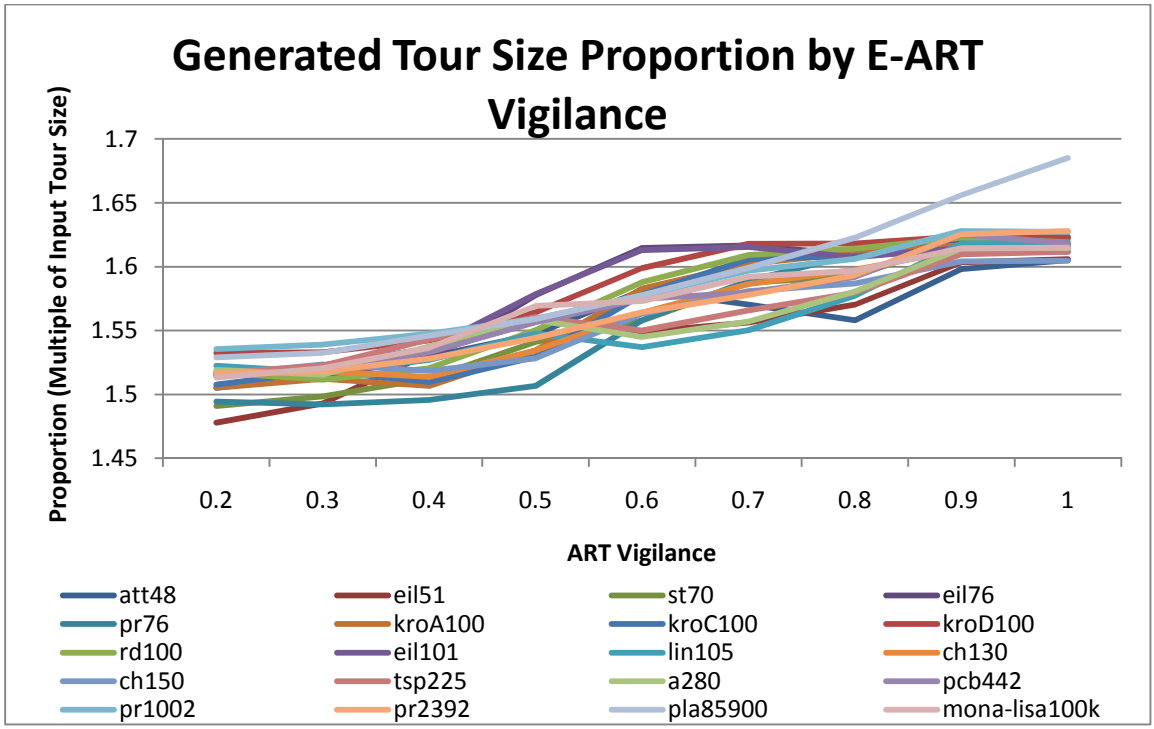


Figure 6.15. Generated tour size proportion by E-ART vigilance.

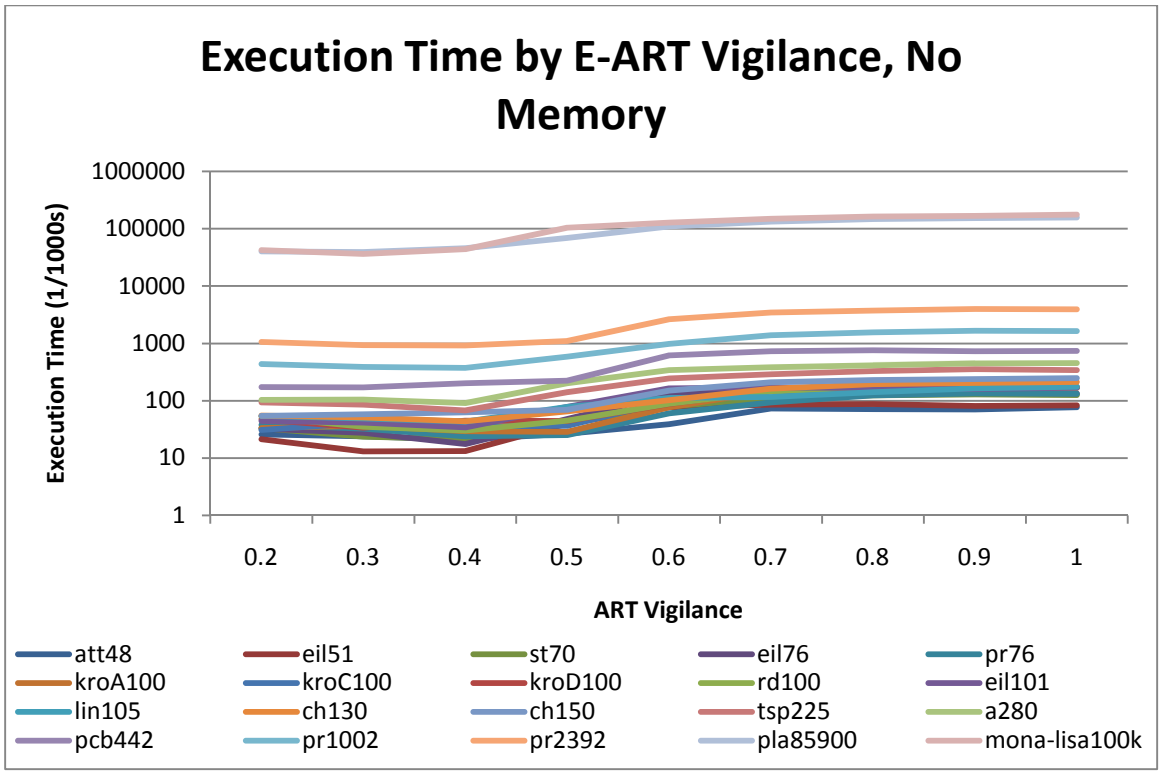


Figure 6.16. Execution time by E-ART vigilance, without memory.

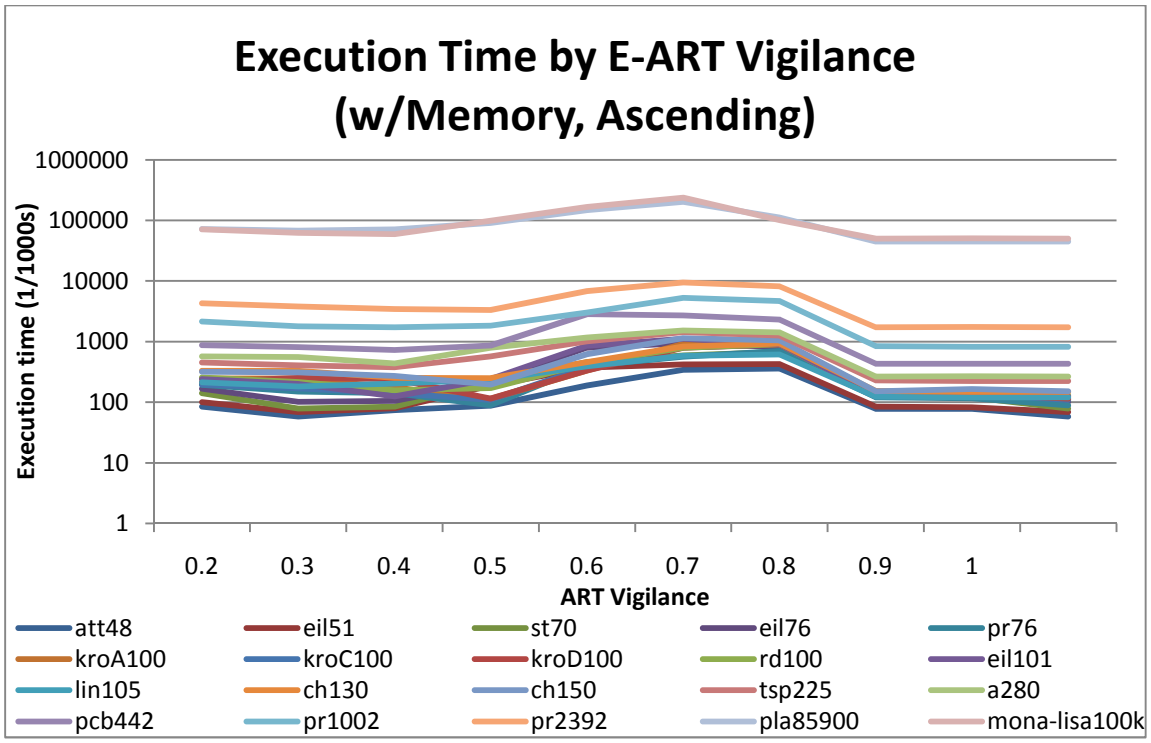


Figure 6.17. Execution time by E-ART vigilance, with memory, ascending order.

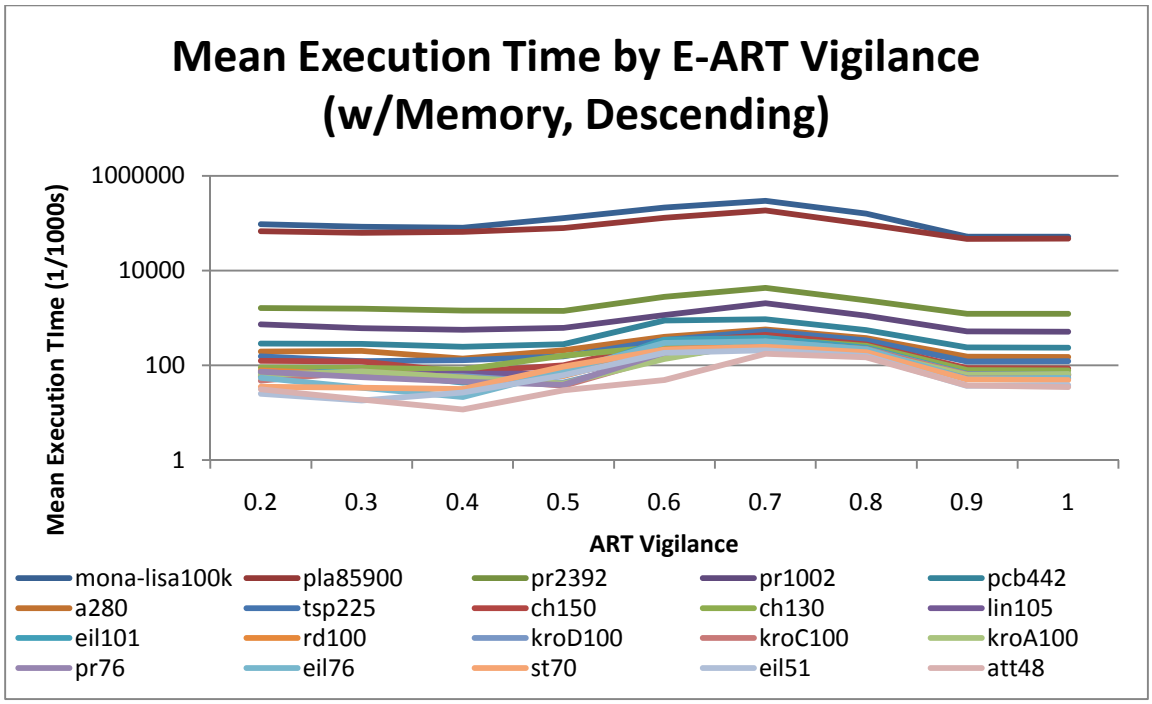


Figure 6.18. Execution time by E-ART vigilance, with memory, descending order.

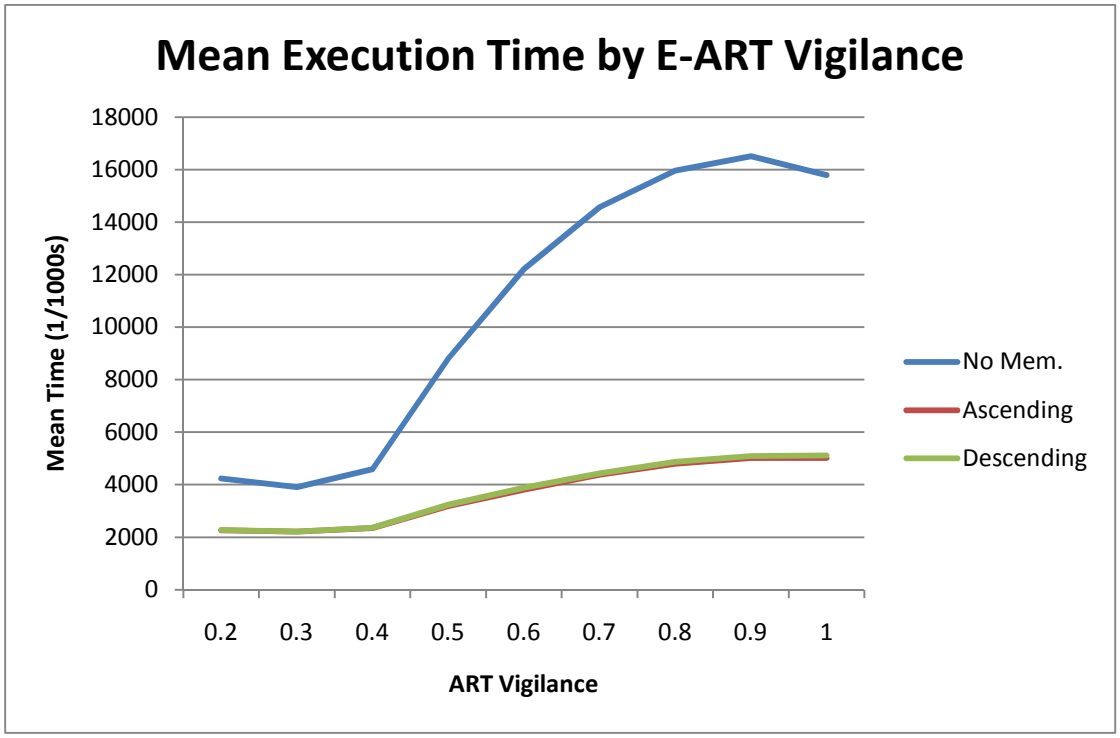


Figure 6.19. Execution time by E-ART vigilance for algorithm configurations.

Table 6.8. Analysis of variance on effect of ART Vigilance on execution time for Meta-TSP without memory.

Art Vig	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.2	5.20E-01	5.28E-01	2.26E-06	4.25E-10	2.13E-11	6.78E-12	4.26E-12	9.62E-11
0.3		1.93E-01	2.36E-07	4.91E-11	3.34E-12	1.26E-12	8.38E-13	2.15E-11
0.4			1.39E-05	2.61E-09	1.06E-10	2.98E-11	1.80E-11	3.66E-10
0.5				2.33E-02	8.82E-04	1.42E-04	7.03E-05	3.47E-04
0.6					2.20E-01	6.79E-02	4.13E-02	9.18E-02
0.7						5.30E-01	3.93E-01	5.94E-01
0.8							8.19E-01	9.41E-01
0.9								7.68E-01

Table 6.9. Analysis of variance on effect of ART vigilance on execution time for Meta-TSP with memory and ascending order.

ART Vig	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.2	7.24E-01	6.31E-01	2.80E-04	1.02E-06	2.41E-08	2.46E-09	7.31E-10	3.97E-10
0.3		3.99E-01	9.06E-05	3.10E-07	7.99E-09	8.73E-10	2.66E-10	1.39E-10
0.4			9.96E-04	4.03E-06	8.84E-08	8.52E-09	2.47E-09	1.39E-09
0.5				9.40E-02	5.03E-03	5.59E-04	1.69E-04	1.40E-04
0.6					2.16E-01	4.87E-02	2.00E-02	1.89E-02
0.7						4.45E-01	2.56E-01	2.54E-01
0.8							7.07E-01	7.09E-01
0.9								9.96E-01

Table 6.10. Analysis of variance on effect of ART vigilance on execution time for Meta-TSP with memory and descending order.

ART Vig	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.2	6.82E-01	6.37E-01	1.96E-04	5.84E-07	1.88E-08	2.05E-09	5.36E-10	2.26E-10
0.3		3.73E-01	5.36E-05	1.52E-07	5.46E-09	6.50E-10	1.73E-10	6.98E-11
0.4			6.91E-04	2.24E-06	6.63E-08	6.75E-09	1.74E-09	7.59E-10
0.5				8.40E-02	5.15E-03	5.56E-04	1.65E-04	1.10E-04
0.6					2.39E-01	5.34E-02	2.23E-02	1.81E-02
0.7						4.34E-01	2.51E-01	2.25E-01
0.8							7.14E-01	9.58E-01
0.9								6.73E-01

6.2.2.2 Effect of sub-tour limit. The sub-tour size limit parameter has a similar effect to that of E-ART vigilance. This is expected, as both parameters control the construction of the tour hierarchy. The sub-tour limit has no measureable effect on the resulting tour cost. This is supported by the analysis presented in Tables 6.11 - 6.13.

However, the sub-tour limit has a significant effect on execution time, as can be seen in Figures 6.20 - 6.23. The general trend is that execution time increases as sub-tour sizes increase. Once again, this can be explained by the exhaustive optimization method used. For small sub-tours, execution time is very low, but increases rapidly as sub-tour size increases. The significance of this relationship is confirmed in Tables 6.14 - 6.16. Of interesting note is the bowl-shaped profile for algorithm configurations utilizing memory. This is likely due to memory overhead. For low sub-tour sizes, memorized tours proliferate, increasing memory access time. For medium sub-tour sizes, less tours need to be memorized, and the search process becomes much more efficient. For large sub-tour sizes, the complexity of the optimization method greatly overshadows memory overhead.

Table 6.11. Analysis of variance on effect of sub-tour limit on cost for Meta-TSP without memory.

Max Tour	4	5	6	7	8
3	5.01E-01	3.02E-01	2.03E-01	1.50E-01	1.08E-01
4		7.19E-01	5.48E-01	4.42E-01	3.48E-01
5			8.10E-01	6.82E-01	5.63E-01
6				8.65E-01	8.67E-01
7					7.35E-01

Table 6.12. Analysis of variance on effect of sub-tour limit on cost for Meta-TSP with memory and ascending order.

Max Tour	4	5	6	7	8
3	0.888464	0.701251	0.495235	0.353027	0.2403
4		0.807986	0.588559	0.431076	0.301787
5			0.765839	0.586225	0.429696
6				0.805183	0.62252
7					0.805907

Table 6.13. Analysis of variance on effect of sub-tour limit on cost for Meta-TSP with memory and descending order.

Max Tour	4	5	6	7	8
3	8.51E-01	6.47E-01	4.22E-01	2.88E-01	1.77E-01
4		7.88E-01	5.39E-01	3.83E-01	2.46E-01
5			7.30E-01	5.46E-01	3.72E-01
6				7.96E-01	5.84E-01
7					7.73E-01

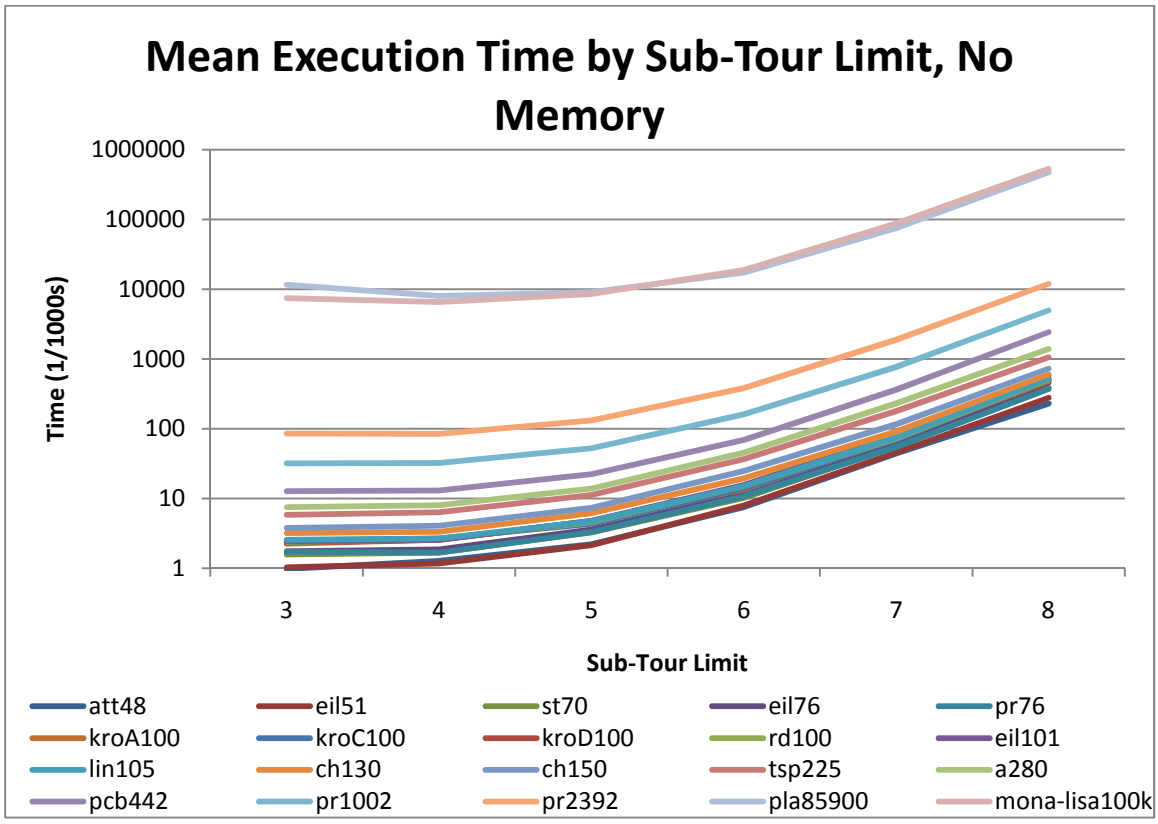


Figure 6.20. Execution time by sub-tour limit, without memory.

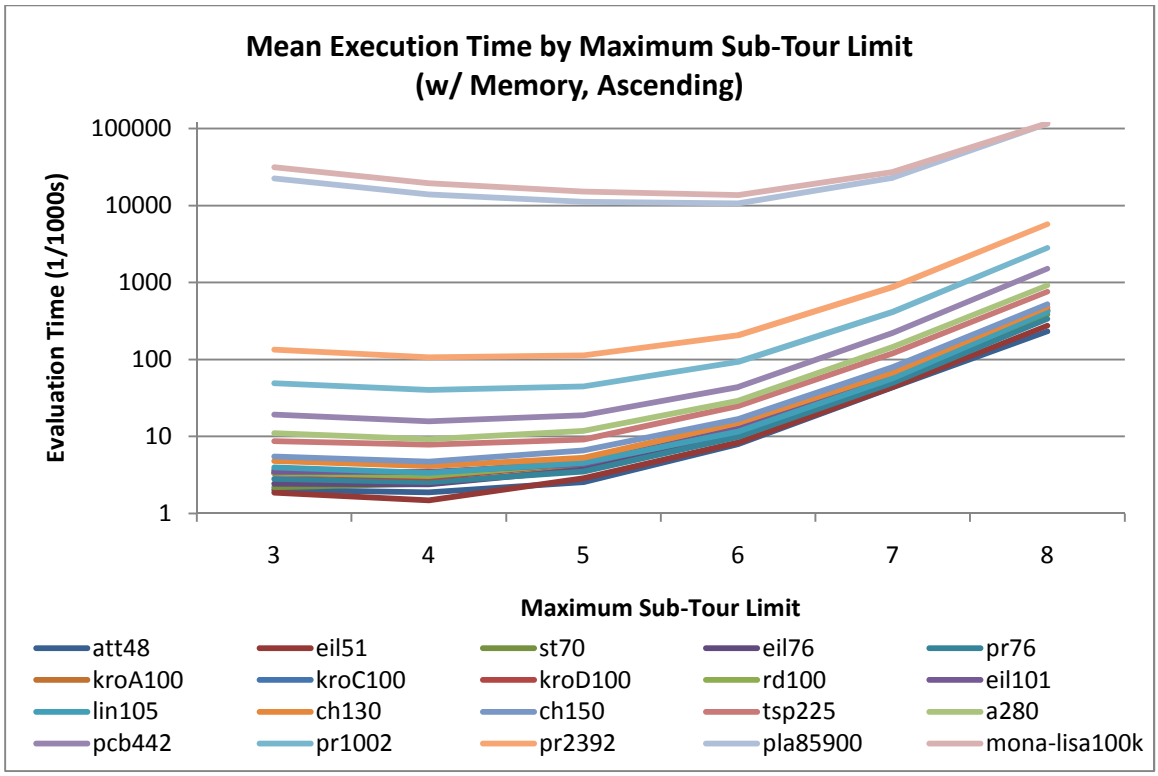


Figure 6.21. Execution time by sub-tour limit, with memory, ascending order.

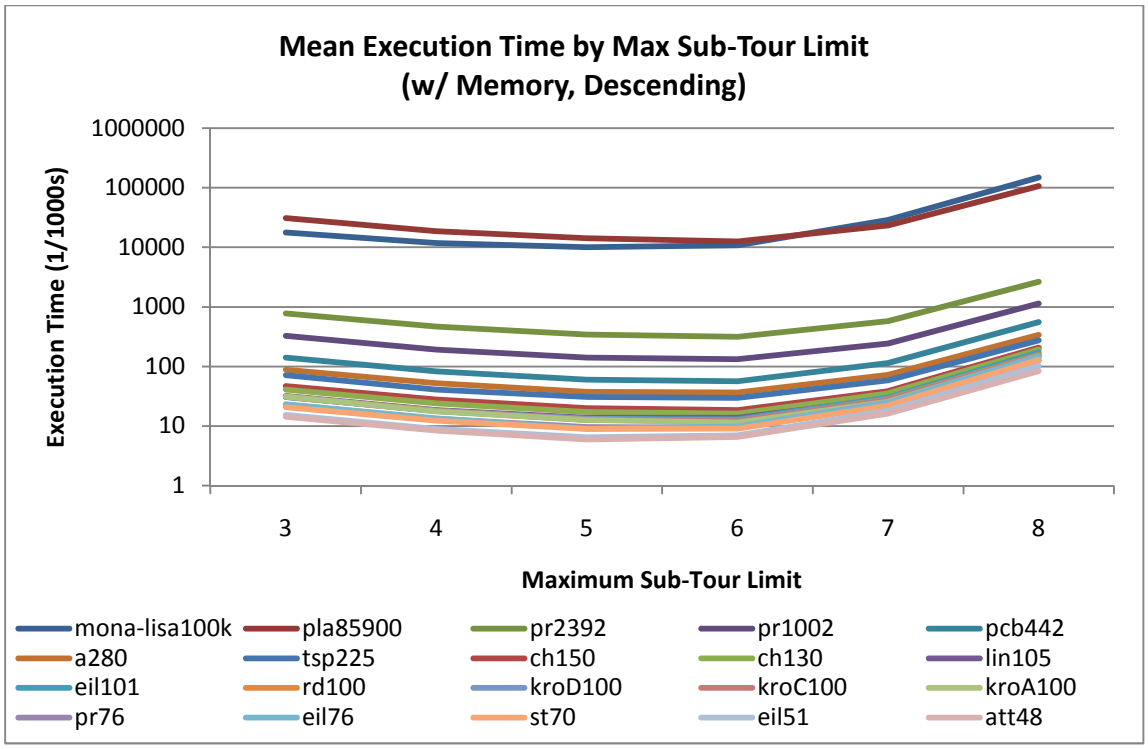


Figure 6.22. Execution time by sub-tour limit, with memory, descending order.

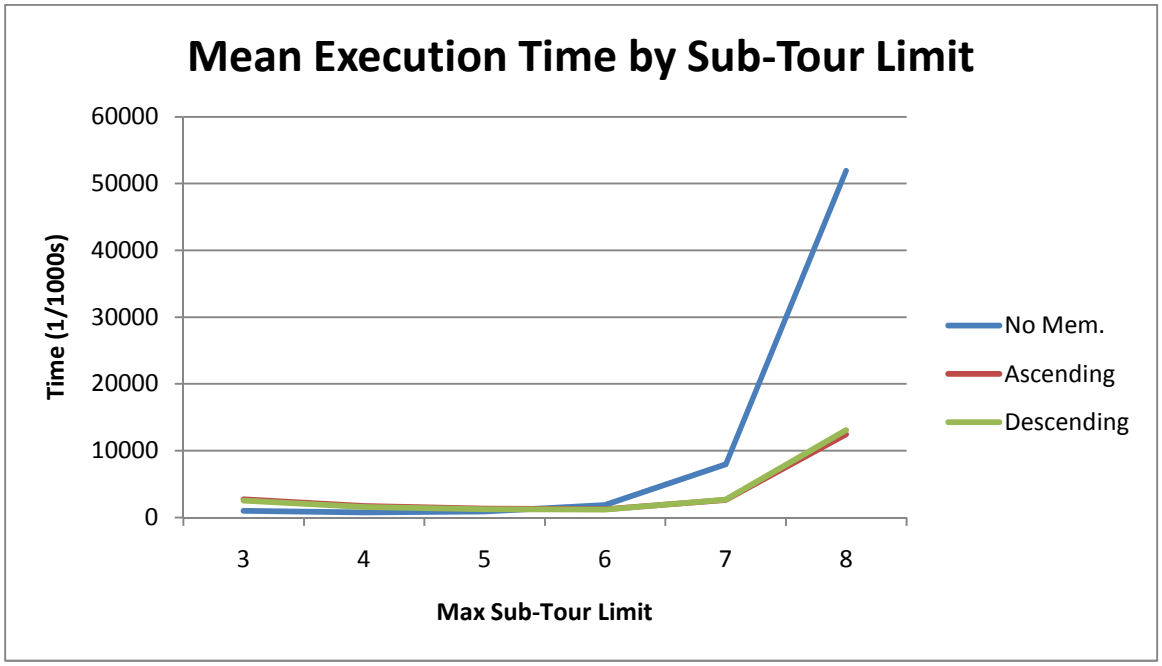


Figure 6.23. Execution time by sub-tour limit for algorithm configurations.

Table 6.14. Analysis of variance on effect of sub-tour limit on execution time for Meta-TSP without memory.

Max Tour	4	5	6	7	8
3	3.09E-67	3.11E-67	3.19E-67	8.55E-67	1.16E-66
4		8.37E-04	2.18E-41	1.77E-86	2.28E-102
5			1.89E-29	1.23E-82	9.32E-102
6				2.50E-60	4.99E-98
7					1.98E-74

Table 6.15. Analysis of variance on effect of sub-tour limit on execution time for Meta-TSP with memory and ascending order.

Max Tour	4	5	6	7	8
3	2.24E-18	4.76E-36	2.19E-40	5.98E-01	3.47E-61
4		4.74E-07	5.13E-10	1.80E-14	1.37E-75
5			1.65E-01	6.50E-29	6.82E-81
6				1.48E-32	4.55E-82
7					6.98E-62

Table 6.16. Analysis of Variance on effect of sub-tour limit on cost for Meta-TSP with memory and descending order.

Max Tour	4	5	6	7	8
3	2.76E-21	6.35E-41	1.80E-43	2.64E-01	7.14E-69
4		2.65E-07	5.35E-09	1.36E-20	1.65E-82
5			3.84E-01	9.00E-36	2.71E-87
6				8.20E-38	6.01E-88
7					3.20E-66

6.2.2.3 Effect of memory vigilance. To examine the effect of memory vigilance, only the algorithm configurations utilizing initially empty memory were used, namely the ascending and descending configurations.

The effect of memory vigilance on resultant tour cost is first examined. It can be seen from Table 6.17 and Table 6.18 that only memory vigilance values greater than 0.7 have a significant effect on tour cost. As will be examined later, tour-cost decreases as memory vigilance approaches 1.0. At a memory vigilance value of 1.0, only tours that match perfectly can be retrieved from memory. The likelihood of this occurring is very low, thus most sub-tours will be optimized. The fact that final tour cost decreases as memory vigilance increases indicates that the stored tours are not always optimal for the contexts that activate them, particularly for low vigilance values.

Conversely, Figure 6.24 and Figure 6.25 show that as memory vigilance increases, execution time increases as well. This is explained by the same mechanism described previously. As memory vigilance increases, less stored tours are activated out of memory, and more tours thus need to be optimized, increasing evaluation time. This relationship is confirmed by analysis in Table 6.19 and Table 6.20. Figure 6.24 and Figure 6.25 also illustrate an effect of TSP instance presentation order on execution time.

Note that in the ascending case, (Figure 6.24) the spacing between problem profiles is relatively wide, while in the descending case (Figure 6.25), the spacing between problem profiles is very narrow. In the ascending case, this is due to the 'bottom-up' order of presentation, each successive tour has only a small body of memorized tours to utilize, while in the ascending case presentation proceeds from the 'top-down', and the majority of memorized tours (and thus computation) are generated in the initial instance (mona-lisa100K). This provides a full library for utilization by successive optimization instances, decreasing the amount of computation required to solve a new instance.

For these measurements, memory is defined as the library of stored generalized tours and memory size is measure in the count of generalized tours in the library, in contrast to physical system memory, which is not measured directly for these experiments.

This difference between ascending and descending TSP instance presentation is further illustrated by examining the effect of memory vigilance on memory size, shown in Figure 6.26 and Figure 6.27. In both cases, memory size increases with memory vigilance, as expected and explained previously. In the ascending case, memory is increased incrementally, and the amount of increase is dependent on the interaction between tour-size and memory vigilance. In the descending case most of the memory contents are created initially with the mona-lisa100k TSP instance, and successive instances contribute very few additional memory elements.

Inspection of the trend of memory growth in Figure 6.28 and Figure 6.29 provides more evidence of the effect of memory vigilance. In these figures, the change in memory size is plotted against the TSP instances with each series representing a different memory

vigilance value. The instances are listed left to right in order of presentation for the respective algorithm configurations. In the ascending case (Figure 6.28) there is the expected 'fan-out' of memory growth, as each successive tour must add proportionally more sub-tours to the memory. Contrarily, the descending procession (Figure 6.29) adds a relatively small quantity of memory elements at each successive instance.

Examining the hit rate for the two memory configurations in Figure 6.30 and Figure 6.31 yields additional insight into the operation of the memory mechanism. The hit-rate is measured by dividing the number of tours restored from memory (hits) by the number of memory accesses. Predictably, as memory vigilance increases, the hit-rate decreases in both algorithm configurations. As further evidence of the effect of problem ordering, in the ascending configuration the hit rate rises gradually by problem presentation, while hit rate in the descending case remains relatively constant over all problem instances. This should have the effect of improved execution time, as high hit rates indicate less new computation.

A clear picture of the overall effect of memory vigilance on cost and time performance is provided when these metrics are averaged over all problem instances, as shown in Figure 6.32 and Figure 6.33. These figures show that overall there is very little difference in either performance or cost between the two algorithm configurations.

Table 6.17. Analysis of variance on effect of memory vigilance on cost for Meta-TSP with memory and ascending order.

Mem Vig	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.1	9.90E-01	9.72E-01	9.58E-01	9.15E-01	8.08E-01	5.46E-01	3.45E-02	8.65E-11	4.19E-21
0.2		9.82E-01	9.68E-01	9.24E-01	8.17E-01	5.54E-01	3.55E-02	9.28E-11	4.56E-21
0.3			9.85E-01	9.42E-01	8.34E-01	5.69E-01	3.74E-02	1.05E-10	5.31E-21
0.4				9.56E-01	8.48E-01	5.81E-01	3.91E-02	1.17E-10	6.06E-21
0.5					8.91E-01	6.19E-01	4.44E-02	1.59E-10	8.87E-21
0.6						7.18E-01	6.08E-02	3.48E-10	2.38E-20
0.7							1.29E-01	2.69E-09	3.39E-19
0.8								5.87E-06	1.67E-14
0.9									7.56E-04

Table 6.18. Analysis of variance on effect of memory vigilance on cost for Meta-TSP with memory and descending order.

Mem Vig	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.1	9.83E-01	9.76E-01	9.75E-01	9.38E-01	8.51E-01	6.68E-01	1.49E-01	3.17E-09	2.88E-21
0.2		9.92E-01	9.91E-01	9.54E-01	8.67E-01	6.82E-01	1.55E-01	3.52E-09	3.28E-21
0.3			9.98E-01	9.60E-01	8.73E-01	6.88E-01	1.57E-01	3.68E-09	3.48E-21
0.4				9.61E-01	8.74E-01	6.89E-01	1.58E-01	3.71E-09	3.53E-21
0.5					9.12E-01	7.24E-01	1.72E-01	4.74E-09	4.83E-21
0.6						8.08E-01	2.08E-01	8.49E-09	1.05E-20
0.7							3.09E-01	3.05E-08	6.10E-20
0.8								4.59E-06	1.07E-16
0.9									6.00E-05

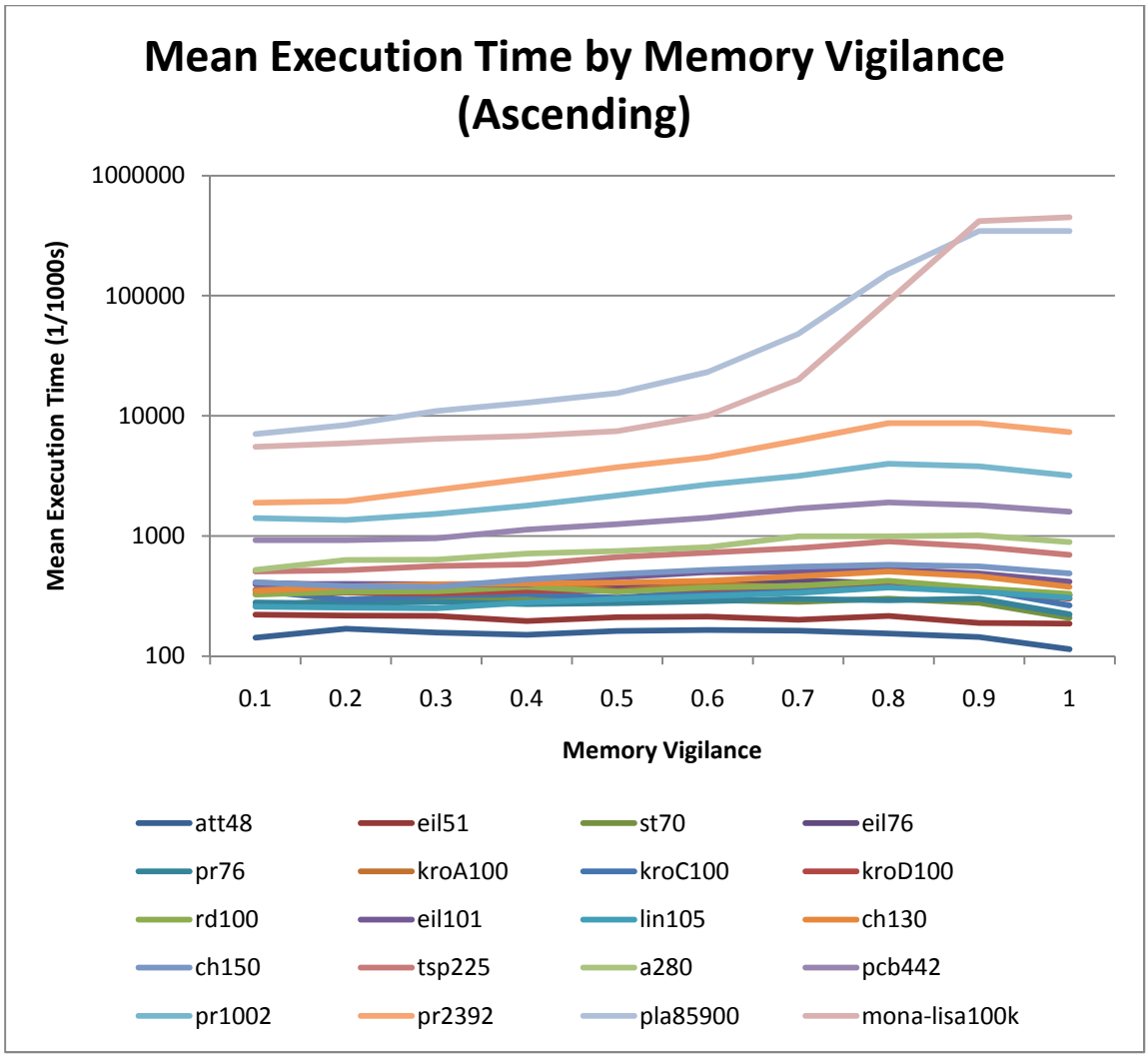


Figure 6.24. Execution time by memory vigilance, ascending order.

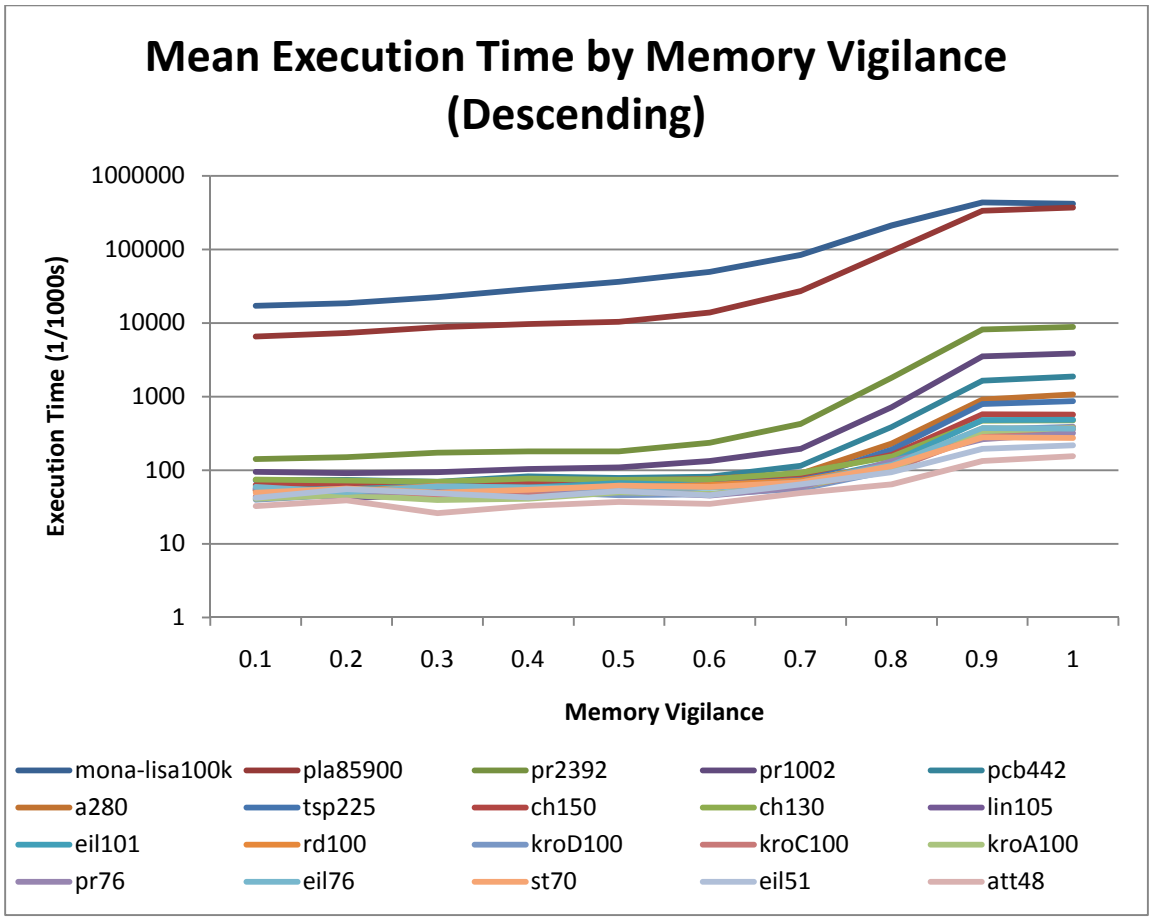


Figure 6.25. Execution time by memory vigilance, descending order.

Table 6.19. Analysis of variance on effect of memory vigilance on execution time for Meta-TSP with memory and ascending order.

Mem Vig	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.1	3.47E-01	1.36E-02	1.33E-04	6.74E-09	1.13E-22	1.42E-56	4.51E-71	4.73E-65	1.78E-80
0.2		1.24E-01	3.80E-03	1.07E-06	3.64E-19	1.05E-52	6.95E-70	9.22E-65	3.34E-80
0.3			1.72E-01	7.72E-04	4.89E-14	1.66E-46	7.03E-68	2.88E-64	9.76E-80
0.4				4.34E-02	3.41E-10	2.98E-41	4.65E-66	8.22E-64	2.63E-79
0.5					1.25E-05	1.10E-33	2.99E-63	4.21E-63	1.23E-78
0.6						2.17E-18	1.95E-56	2.37E-61	5.58E-77
0.7							1.54E-37	4.29E-56	5.78E-72
0.8								4.70E-33	5.72E-49
0.9									3.36E-04

Table 6.20. Analysis of variance on effect of memory vigilance on execution time for Meta-TSP with memory and descending order.

Mem Vig	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.1	3.87E-01	1.26E-02	2.45E-05	4.21E-09	4.97E-18	1.39E-36	9.07E-53	8.52E-62	8.26E-77
0.2		1.00E-01	6.54E-04	3.07E-07	1.08E-15	1.79E-34	6.27E-52	1.71E-61	1.64E-76
0.3			6.98E-02	3.10E-04	1.88E-11	2.77E-30	3.25E-50	7.15E-61	6.64E-76
0.4				6.60E-02	2.92E-07	2.29E-25	4.42E-48	4.24E-60	3.81E-75
0.5					6.64E-04	2.89E-20	1.24E-45	3.36E-59	2.91E-74
0.6						3.55E-11	3.28E-40	3.76E-57	3.05E-72
0.7							2.44E-25	7.23E-51	5.49E-66
0.8								5.67E-27	7.71E-41
0.9									2.65E-03

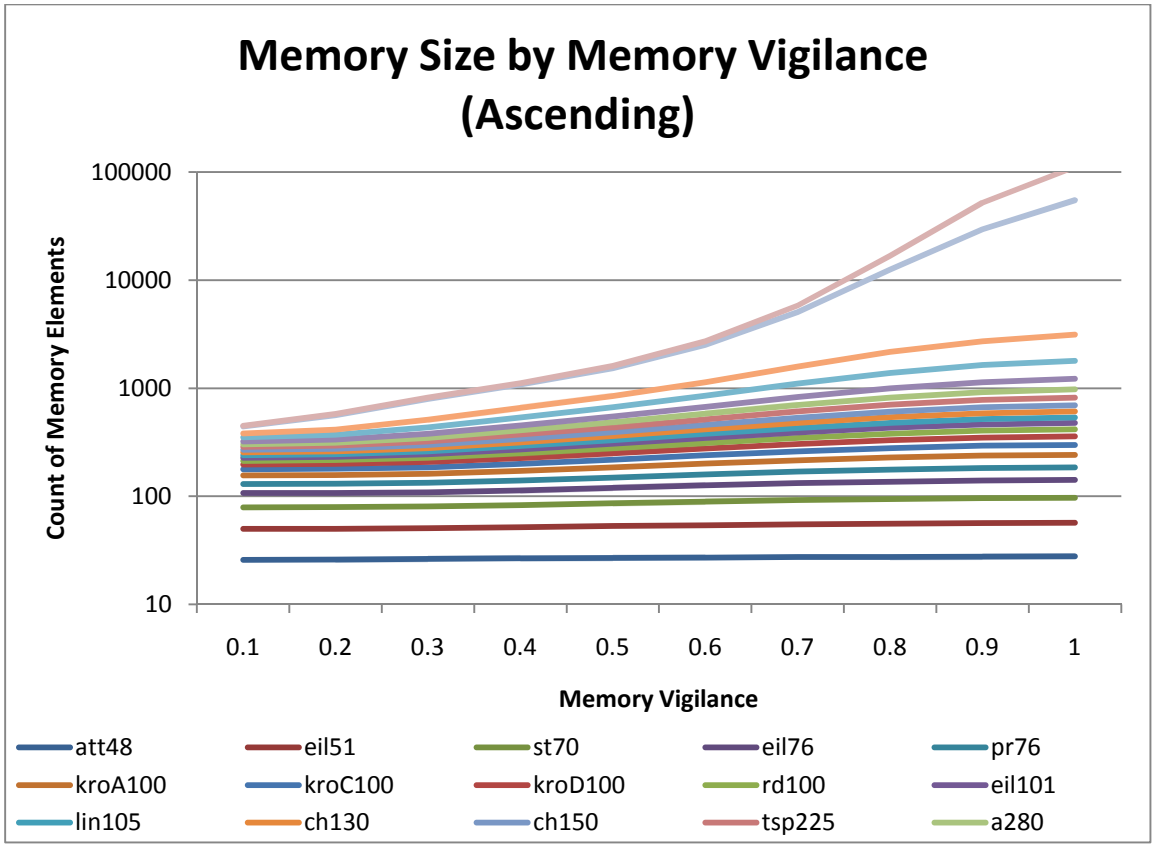


Figure 6.26. Memory size by memory vigilance, ascending order.

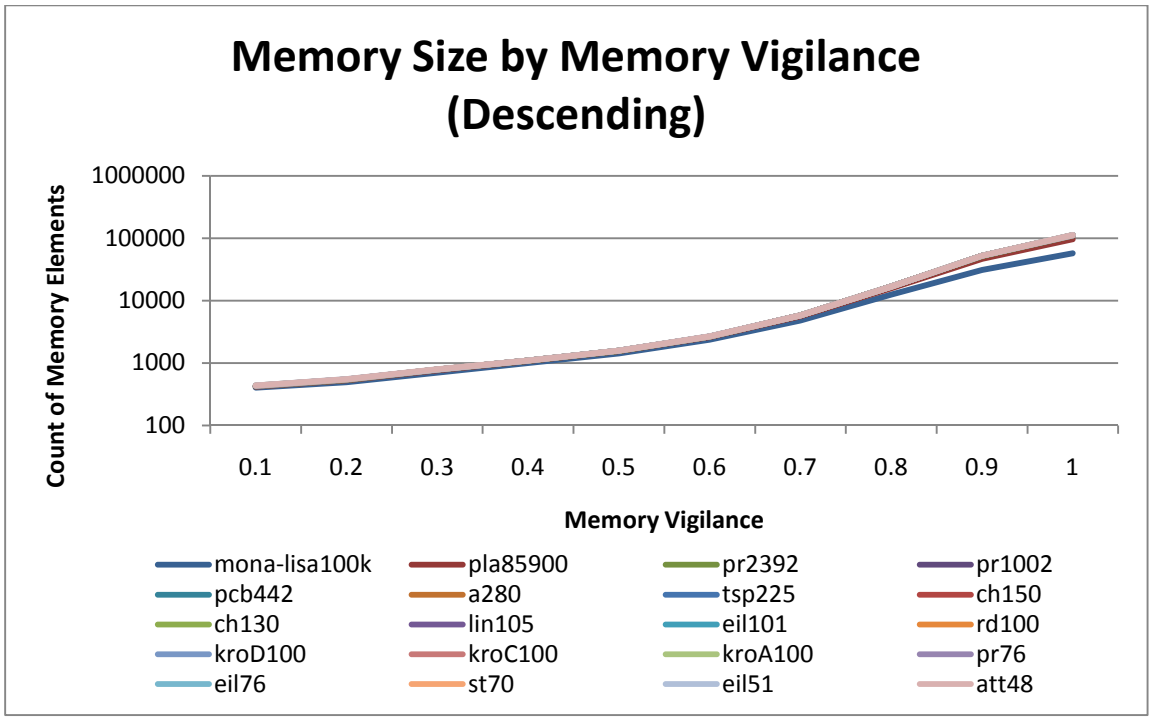


Figure 6.27. Memory size by memory vigilance, descending order.

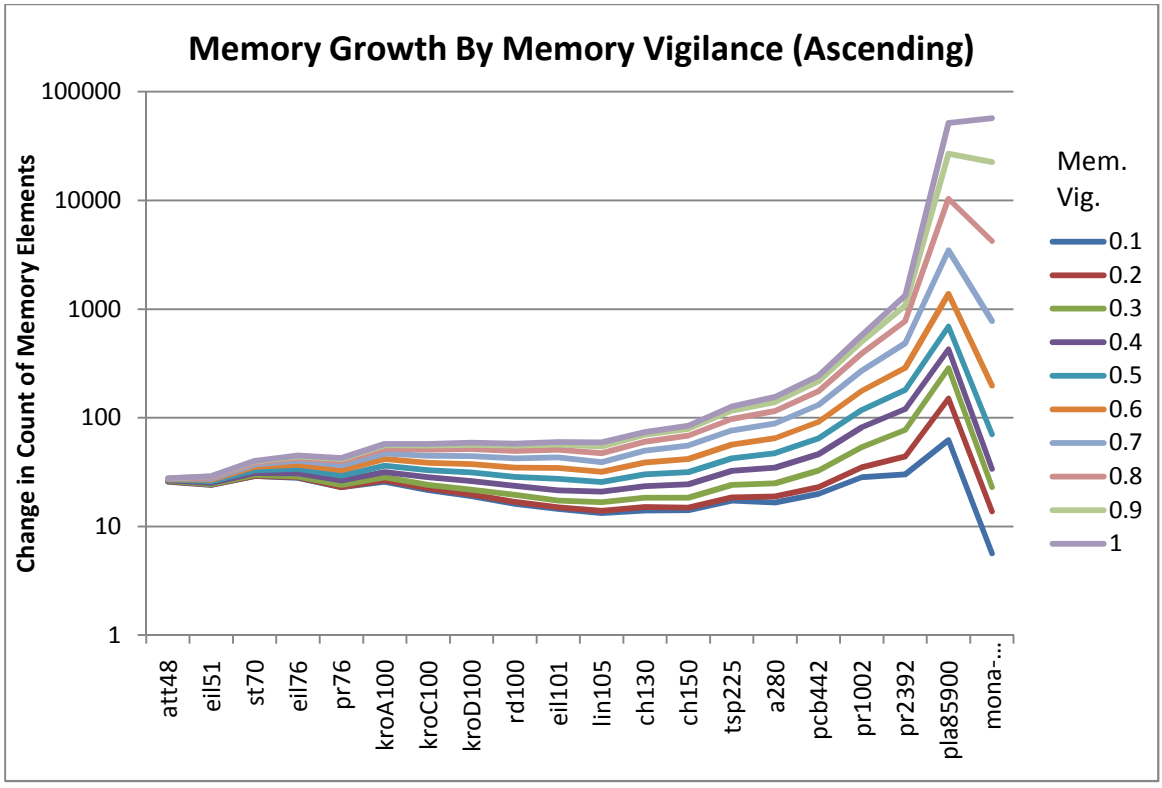


Figure 6.28. Memory growth by memory vigilance, ascending order.

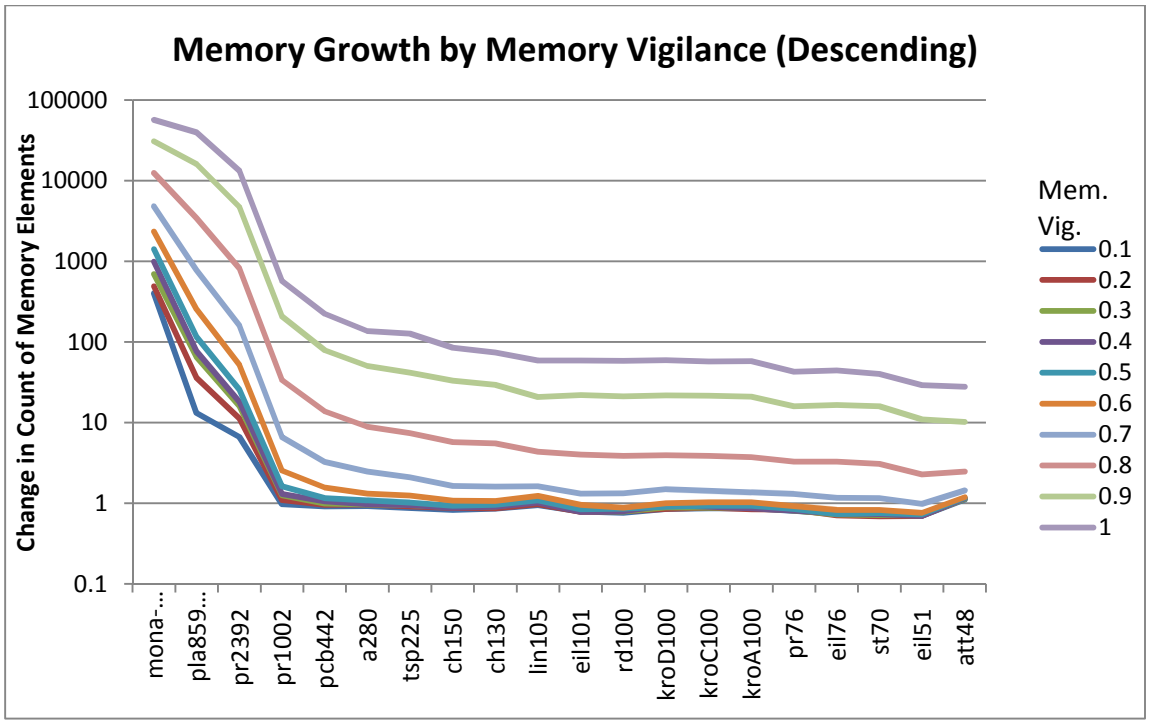


Figure 6.29. Memory growth by memory vigilance, descending order.

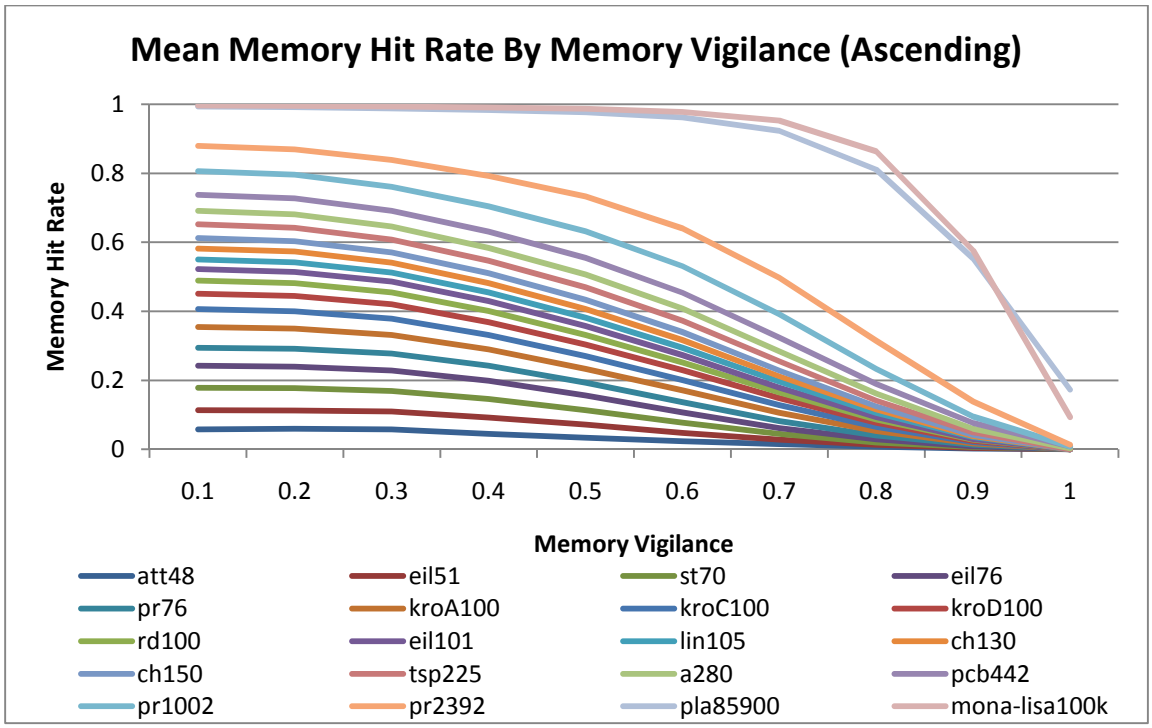


Figure 6.30. Memory hit rate by memory vigilance, ascending order.

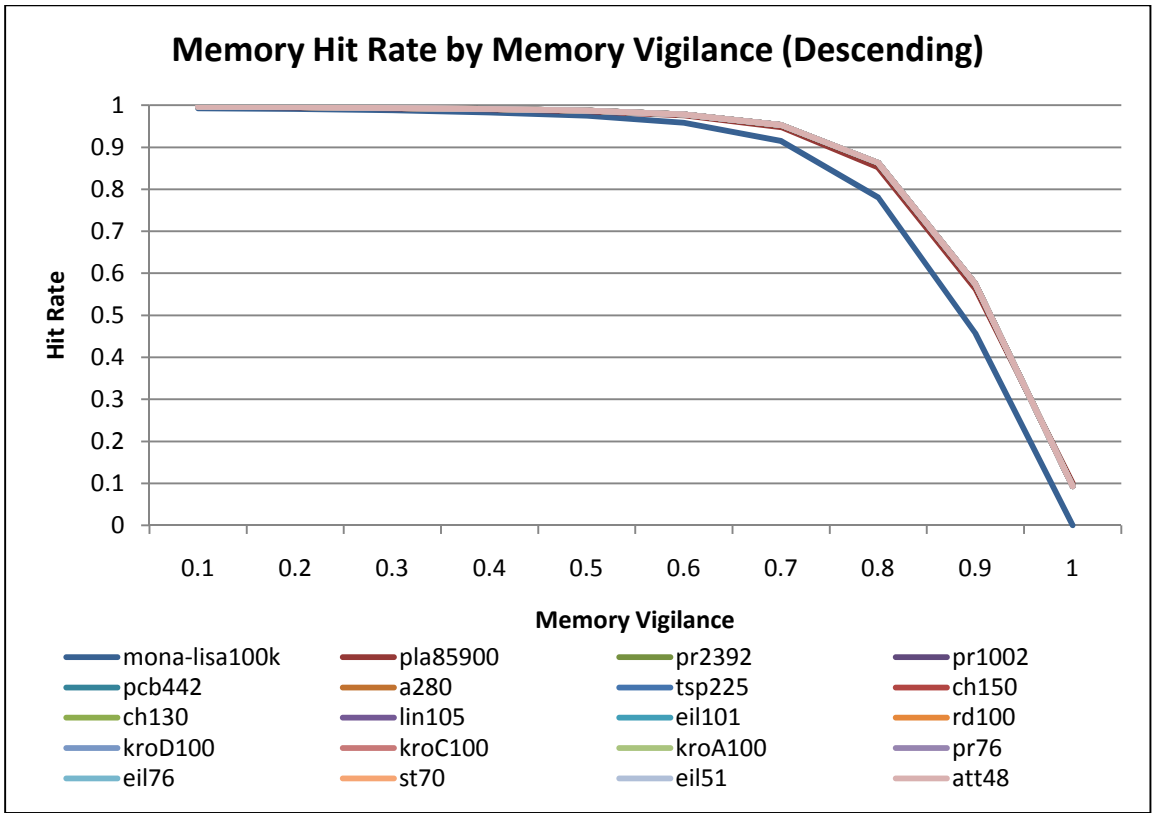


Figure 6.31. Memory hit rate by memory vigilance, descending order.

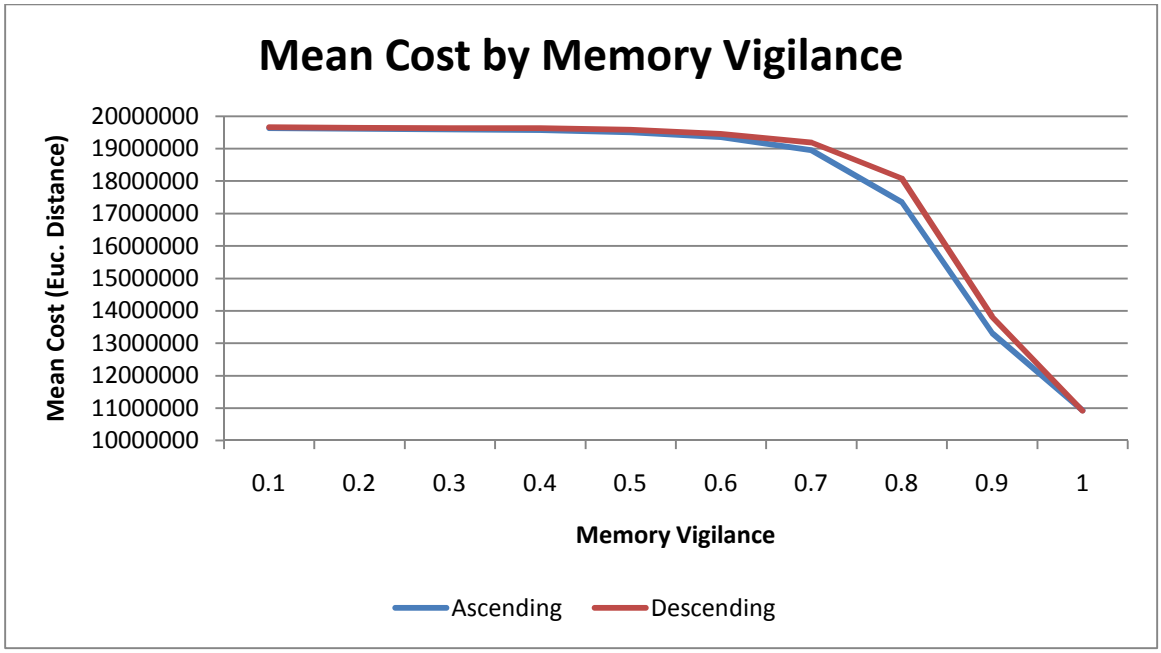


Figure 6.32. Cost by memory vigilance for algorithm configurations.

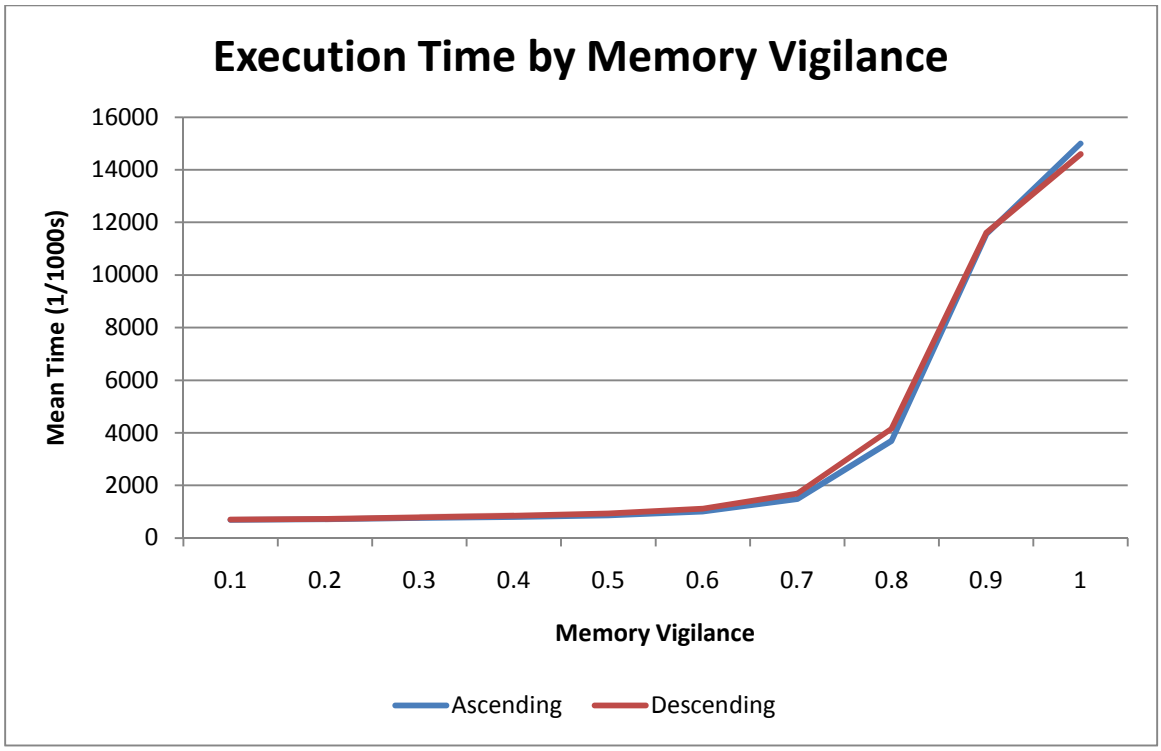


Figure 6.33. Execution time by memory vigilance for algorithm configurations.

6.2.2.4 Effect of algorithm configuration. Some indirect comparison between the algorithm configurations has already been completed in the previous sections, particularly the difference in memory characteristics based on order of instance presentation. In this section, the performance of all algorithm configurations are compared directly in terms of execution time and cost.

The comparison between configurations is completed first across all parameter combinations and between problem instances. Table 6.21 contains the result of the analysis between algorithm configurations and problems based on cost, while Table 6.22 contains the result of the analysis between algorithm configurations and problems based on execution time. In both analyses, the vast majority of configuration and problem

combinations are significantly different. Notable exceptions include the evaluation time between ascending and memory-less configurations. This is expected, as the ascending configuration starts with no elements in memory, giving similar performance to that of the memory-less configuration.

Table 6.21. Analysis of variance on effect of algorithm configuration on cost by test-bed problem.

	Ascending vs. No Mem	Descending vs. No Mem	Ascending vs. Descending	Supervised vs. Ascending
att48	2.03E-04	0.00E+00	0.00E+00	1.96E-08
eil51	7.60E-102	0.00E+00	0.00E+00	1.30E-18
st70	2.42E-219	0.00E+00	0.00E+00	1.79E-06
eil76	0.00E+00	0.00E+00	0.00E+00	4.34E-02
pr76	0.00E+00	0.00E+00	0.00E+00	1.19E-04
kroA100	0.00E+00	0.00E+00	0.00E+00	2.36E-01
kroC100	0.00E+00	0.00E+00	0.00E+00	3.73E-02
kroD100	0.00E+00	0.00E+00	0.00E+00	2.79E-04
rd100	0.00E+00	0.00E+00	0.00E+00	4.31E-05
eil101	0.00E+00	0.00E+00	0.00E+00	3.04E-06
lin105	0.00E+00	0.00E+00	0.00E+00	3.59E-04
ch130	0.00E+00	0.00E+00	0.00E+00	2.88E-06
ch150	0.00E+00	0.00E+00	0.00E+00	3.49E-07
tsp225	0.00E+00	0.00E+00	0.00E+00	1.84E-07
a280	0.00E+00	0.00E+00	0.00E+00	1.30E-06
pcb442	0.00E+00	0.00E+00	0.00E+00	9.42E-08
pr1002	0.00E+00	0.00E+00	1.17E-277	9.04E-07
pr2392	0.00E+00	0.00E+00	4.15E-172	8.39E-05
pla85900	0.00E+00	0.00E+00	2.13E-02	3.01E-03
mona_lisa100K	0.00E+00	0.00E+00	8.45E-16	6.69E-03

The optimality of the average tours generated by the algorithm configurations is given in Table 6.23. The optimality measure is calculated by finding the percent difference between the average instance cost for a configuration, and the optimal (or in the case of mona_lista100k, the best known lower bound) tour cost. For TSP instances

smaller than 200 points, most of the tour costs come within 10% of the optimal cost, with the memory-less configuration achieving optimality on the ‘st70’ TSP instance.

However, for larger tours the optimality drops rapidly. This is likely due to error incurred by poor high-level planning in the tour-structure, which becomes more significant as the tour hierarchy increases in depth.

Table 6.22. Analysis of variance on effect of algorithm configuration on execution time by test-bed problem.

	Ascending vs. No Mem	Descending vs. No Mem	Ascending vs. Descending	Supervised vs. Ascending
att48	6.90E-02	8.64E-37	2.71E-54	3.53E-06
eil51	2.55E-02	2.64E-31	5.17E-51	4.57E-06
st70	1.07E-01	9.61E-40	1.56E-56	2.81E-06
eil76	3.16E-01	6.65E-41	9.85E-55	7.22E-06
pr76	6.96E-01	1.45E-33	1.35E-41	9.27E-06
kroA100	8.17E-01	3.18E-39	3.33E-48	1.02E-05
kroC100	3.02E-01	3.30E-40	4.68E-43	1.07E-05
kroD100	1.89E-01	4.00E-46	5.82E-47	4.23E-06
rd100	1.11E-01	1.37E-43	4.44E-43	6.72E-06
eil101	4.10E-02	3.44E-48	1.29E-44	7.72E-06
lin105	1.31E-03	1.49E-53	2.08E-42	1.86E-06
ch130	1.81E-02	9.52E-39	3.48E-33	6.95E-06
ch150	2.45E-06	2.22E-51	1.88E-33	7.86E-06
tsp225	7.68E-07	2.58E-56	4.23E-36	6.10E-06
a280	3.67E-11	7.71E-63	4.79E-33	8.10E-06
pcb442	3.22E-13	8.30E-63	1.16E-30	2.13E-05
pr1002	1.10E-20	1.16E-62	3.03E-19	3.52E-05
pr2392	2.07E-32	2.04E-64	1.24E-09	1.98E-04
pla85900	2.23E-81	4.16E-77	3.74E-01	1.41E-03
mona_lisa100K	0.00E+00	0.00E+00	7.44E-01	6.65E-04

The average cost and execution time by problem is charted in Figure 6.34 and Figure 6.35. Notice that in comparing the cost performance of the algorithm configurations (Figure 6.34) the memory-less configuration generally approaches closest to optimality, followed by the supervised configuration, ascending, and descending

configuration has generally the worst cost performance. Examining the execution time performance (Figure 6.35) of the algorithm configurations, the memory-less configuration exhibits the worst general performance, particularly on large problem sizes, followed by ascending, descending, and supervisory with the best time performance. It should be noted that the supervisory execution time performance does not include training time.

Table 6.23. Optimality of mean costs by algorithm configuration.

Problem	Ascending	Descending	No Memory
att48	5.78%	1.82%	4.43%
eil51	0.57%	0.95%	0.40%
st70	0.94%	0.80%	**0.00%**
eil76	5.18%	3.47%	4.15%
pr76	0.92%	2.48%	3.21%
kroA100	1.77%	4.00%	1.00%
kroC100	0.18%	1.60%	0.26%
kroD100	2.24%	0.06%	1.76%
rd100	2.47%	3.55%	2.98%
eil101	5.74%	7.23%	4.19%
lin105	0.25%	1.00%	0.48%
ch130	4.45%	5.88%	5.06%
ch150	7.91%	7.80%	6.82%
tsp225	7.84%	8.03%	9.51%
a280	13.31%	12.98%	13.87%
pcb442	14.93%	15.26%	13.43%
pr1002	14.84%	15.04%	13.85%
pr2392	18.82%	19.26%	17.02%
pla85900	21.51%	21.49%	20.45%
mona_lisa100K	11.58%	11.60%	11.70%

The statistics for the training phase of the supervisory configuration are shown in Table 6.24. It should be noted that the hit-rate during training is zero for most training instances, as the memory vigilance used for training was set to its maximum value. Even so some tours matched perfectly for the larger training instances.

Table 6.24. Training statistics for supervised Meta-TSP configuration.

Problem	Hier. Depth	Tour Size	Sub-Tours	Mem. Size	Mem. Accesses	Mem. Hits
att48	3.46	70.46	23.46	8.08	8.08	0.00
eil51	2.23	74.69	24.69	16.15	16.15	0.00
st70	3.23	101.08	32.08	35.62	35.62	0.00
eil76	2.46	109.62	34.62	44.54	44.54	0.00
pr76	4.69	113.00	38.00	59.77	59.77	0.00
kroA100	3.23	142.69	43.69	77.69	77.69	0.00
kroC100	4.23	151.54	52.54	99.69	99.69	0.00
kroD100	3.00	147.15	48.15	121.00	121.00	0.00
eil101	3.46	152.08	52.08	134.62	134.62	0.00
lin105	3.69	148.54	44.54	160.85	160.85	0.00
ch130	4.46	195.69	66.69	184.85	184.85	0.00
ch150	4.46	227.46	78.46	220.77	220.77	0.00
tsp225	5.23	338.31	114.31	260.15	260.15	0.00
a280	5.46	419.38	140.38	314.00	314.23	0.23
pcb442	5.92	653.38	212.38	396.85	398.08	1.23
pr2392	12.46	3576.00	1185.00	912.00	928.38	16.38

Figure 6.34 shows the overall execution time by problem size for the algorithm configurations. Of primary interest is the fact that all of the execution time profiles are *linear* with the problem size, up to the largest evaluated instance at 100,000 points. This strongly suggests that the overall time-complexity for Meta-TSP optimization variants is $O(n)$. Also note the effects of memory, and the contents of memory on algorithm run-time. In the ascending case, memory is constructed over time, starting with similar performance to the memory-less configuration, eventually progressing to performance similar to that of the descending configuration. Note that the descending configuration *starts* with improved time performance, as the memory store is used internally to process complete the largest instance, ‘mona-lisa100k’. Finally, the supervisory configuration

exhibits the best time performance, likely through the utilization of the store of pre-optimized tours, enabling the configuration to complete instances rapidly with little computation beyond the construction of the tour hierarchy and memory access overhead.

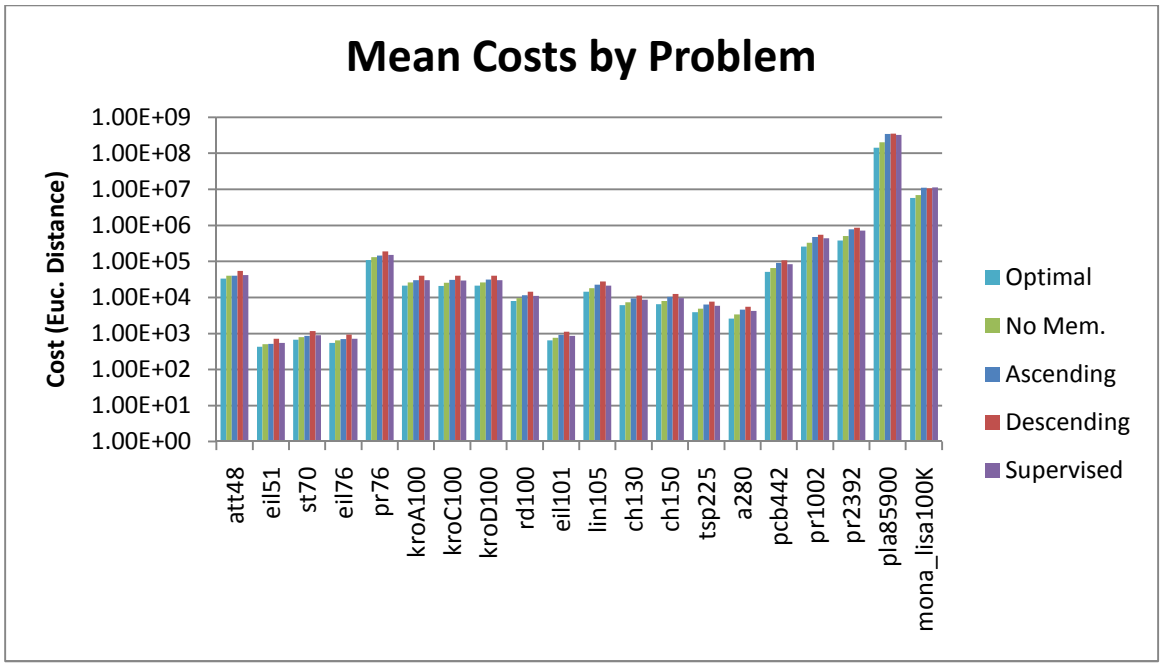


Figure 6.34. Costs by problem across all algorithm configurations, and optimality.

6.2.3. Discussion. The results demonstrate that the Meta-TSP algorithm is capable of generating good solutions to the TSP in linear time, and that the addition of memory can greatly influence and in some cases greatly improve the execution time of an algorithm without large impact on the value of produced solutions.

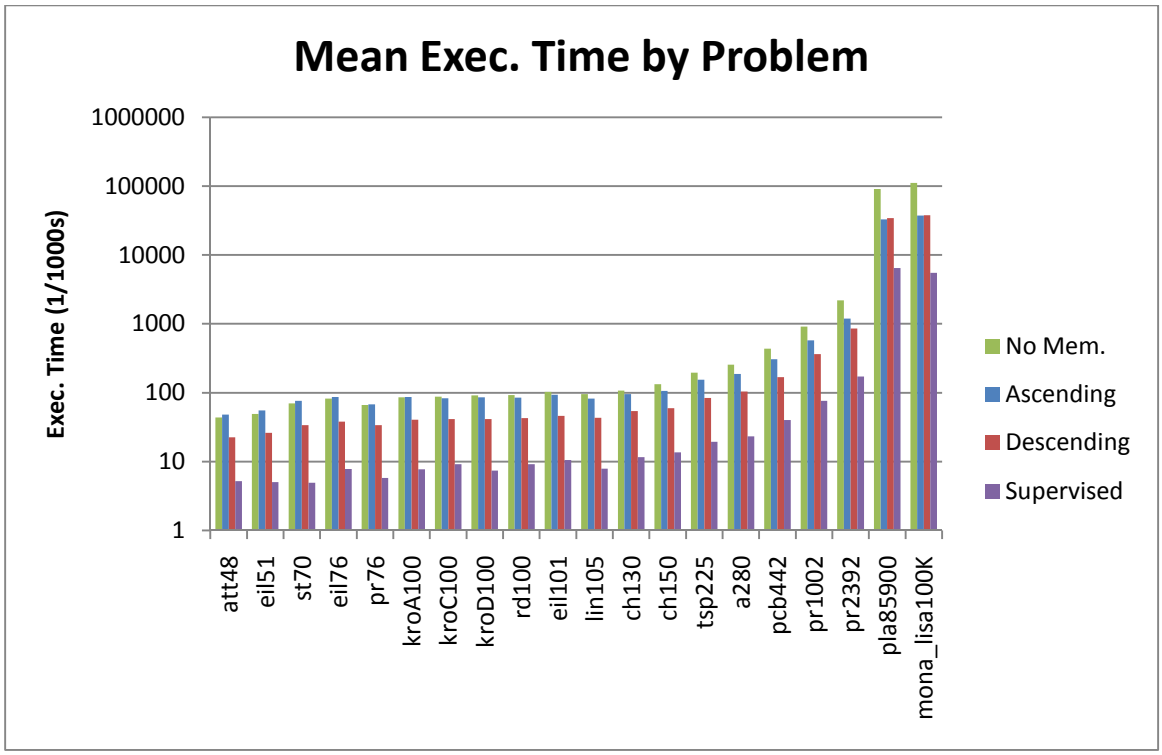


Figure 6.35. Execution time by problem, across all algorithm configurations.

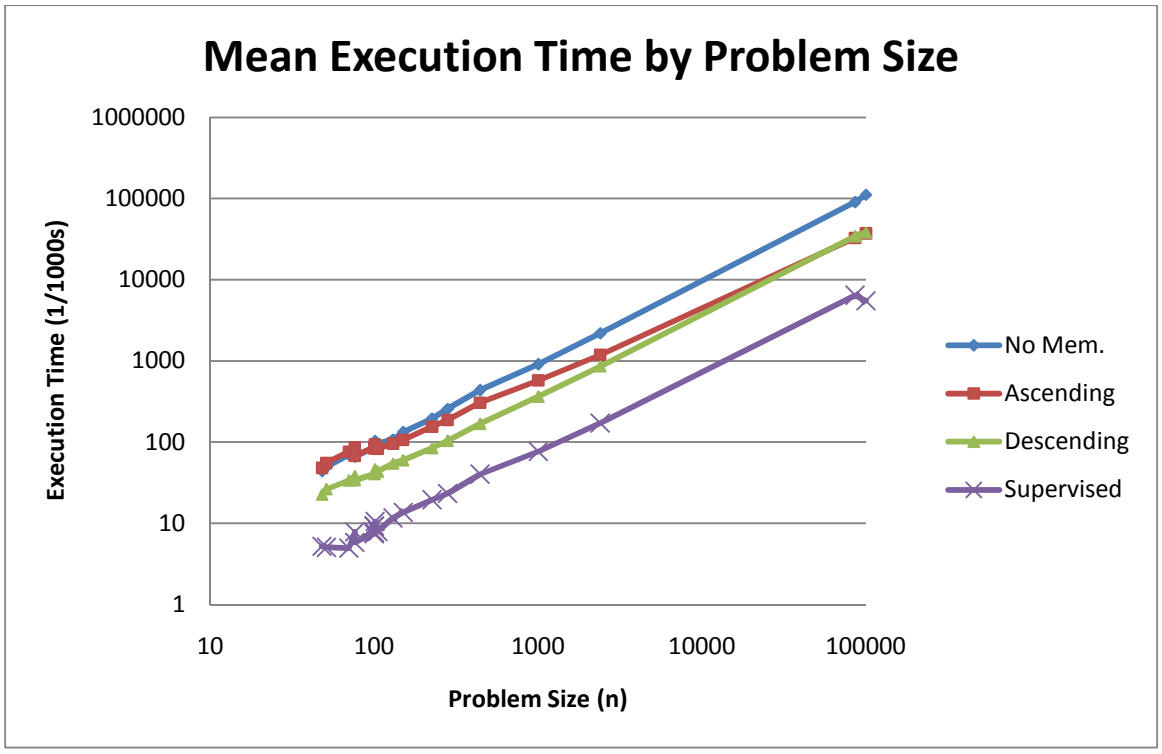


Figure 6.36. Execution time by problem size and algorithm configuration.

In a time-cost comparison with other TSP solution methods (Figure 6.37), the Meta-TSP algorithm is faster than all methods, including greedy methods like Nearest-Neighbor tour construction. This efficiency comes at the cost of tour-quality. Though Meta-TSP produces good quality tours – comparable to those produced by k-opt based methods, the tours are rarely optimal, especially for larger instances. Even so, the Meta-TSP method is a significant increase in heuristic TSP solution methods, as it comes closer to the ideal algorithm efficiency (upper left corner of Figure 6.37) than any other compared method.

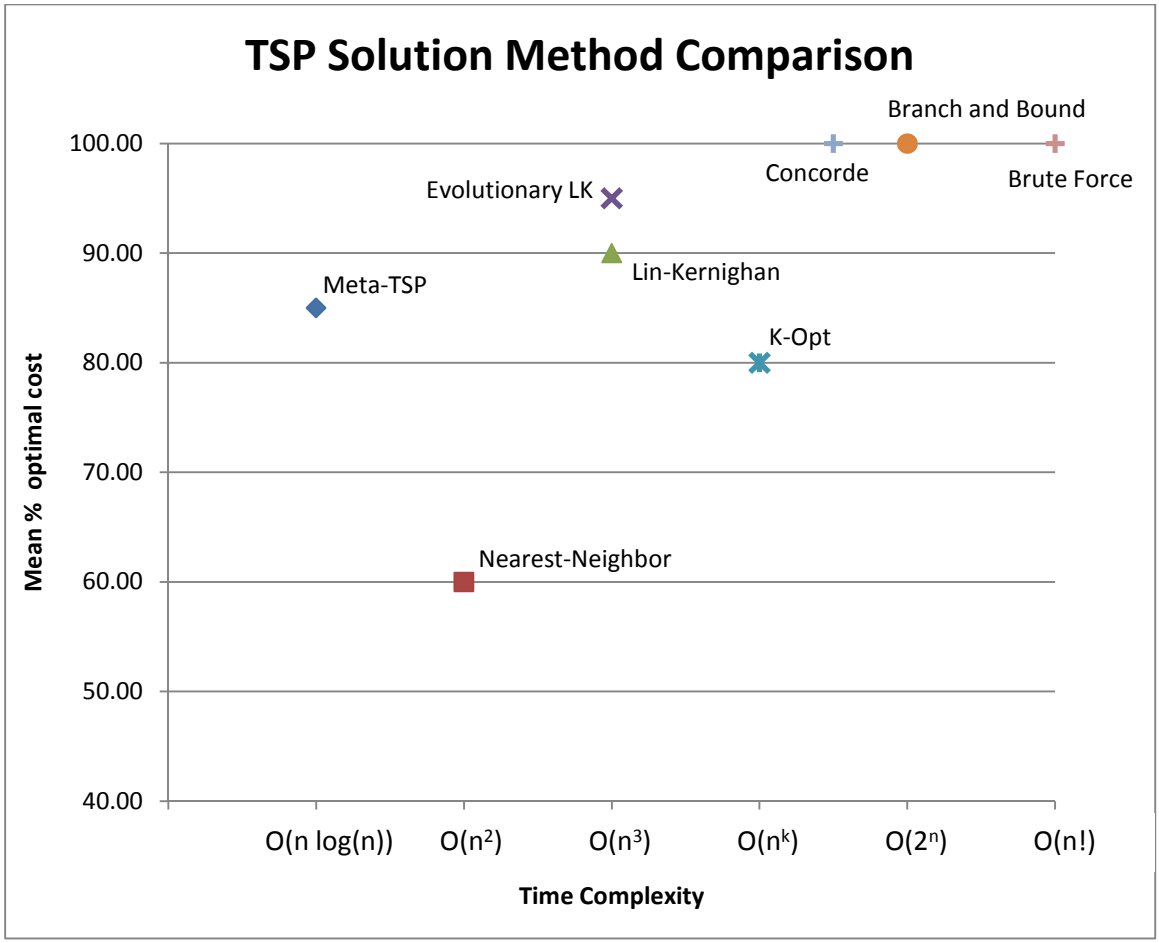


Figure 6.37. Time and Cost comparison of TSP solution methods.

6.3. FUTURE DEVELOPMENT

Though the performance of the Meta-TSP algorithm has excellent performance relative to existing methods, the method can be greatly improved through several avenues of research. The greatest contribution may be to analytically determine the thresholds and critical points of tours, in order to develop a better method for comparing and storing tours. One avenue for this investigation may be to examine how lower-bounds are calculated and employ a method for determining which points contribute most to the lower bound.

Additionally, the tour hash method can be greatly improved, as the discrete tour-hash used here is severely flawed. The core optimizer may also be improved from a $O(n!)$ method to a more efficient $O(2^n)$ operation while still maintaining optimal sub-tour performance. Non-optimal heuristic methods could also be used to further increase the sub-tour size, and it would be interesting to investigate the effect this has on overall tour optimization.

7. DISCUSSION

7.1. PRESENTED CAPABILITIES

The desire for a new and robust computational intelligence paradigm spans many problem domains, including real time robotic systems which must deal with increasing complexity on a daily basis, deep data mining such as natural language processing with applications in information retrieval and machine understanding, human-computer interaction, and long-term optimization. These new, complex frontiers of machine learning and optimization could all benefit from the higher order memetic computing methods described here.

The primary difficulty of designing meta-learning systems lies in the construction of valid representations which enable the construction of selection, generalization, and memory mechanisms. By providing generalization, memory, optimization, and selection mechanisms, a meta-learning architecture can operate on high-level features of a problem instance, selecting generalized solutions that have been used previously with high utility in the problem context. Utilizing these features, a system should be able to learn not only the solution to a problem, but learn *about* solving problems. Such systems may enable a quantum leap in the performance of real-world adaptive systems as they provide the central components of higher level, meta-adaptive systems to be constructed.

An overview of important definitions and architectures in memetic computing has been presented, and the power of next-generation memetic algorithms has been demonstrated through the development of two meta-learning architectures – Meta-GP and Meta-TSP. In the Meta-GP architecture, memory, selection, and recognition components

were added to a core evolutionary optimizer, enabling increased performance through the abstraction and selection of solution components. The Meta-TSP architecture integrated similar memory, recognition and selection components around an exact optimizer, yielding results similar to human performance on the TSP.

In both of these architectures, the integration of memory provides a trade-off of speed for performance. Additionally, in the Meta-TSP architecture it was demonstrated that by providing high-fitness memories from a supervisory source, the gains of speed can be realized with very little cost to performance.

7.2. LIMITATIONS

In any case, a first-order meta-learning method will suffer decreased performance on problem instances or types that are significantly different from previously experienced instances. Due to memory bias, performance may be decreased compared to a similar non-meta learning method, or even a meta-learning method with empty memory, as memes that are poorly suited to a situation are activated, producing poor results. However, the likelihood of this situation occurring decreases as the breadth of curriculum increases, as more memes are available across a wider range of situations. Thus the careful construction of curricula is critical to deriving high performance from a meta-learning system.

Additional, the total time to develop a system, including training time and evaluation time, may be longer than that of a simpler system for a given level of performance. For instance, in the Meta-GP parity experiment, the training time for the full meta-learning method was twice as long as the baseline GP evaluation process, and

ultimately took several thousand generations longer to reach the same level of performance as the baseline. However, the meta-learning process was then able to further improve its performance, while the baseline method stagnated.

Evaluated on the time scale of a single instance, the examined meta-learning systems outperform their non meta-learning counterparts, though this may not be the case when the entire training history is considered.

7.3. RESEARCH CHALLENGES

Future research challenges in the study of memetic and meta-learning algorithms span a wide range of topics. In the field of cognitive modeling, the biological models for memetic computational processes can be developed. On a more design front, the design and construction of increasingly high-level, *n*-meta learning architectures present a significant challenge to design, representation, implementation and evaluation. One avenue for *n*-meta learning architectures may be to investigate the use of a uniform computational component. In the architectures presented here, different computational components such as Artificial Neural Networks, Adaptive Resonance Theory, and Evolutionary Algorithms are integrated together in a cohesive architecture. This limits both the scope of problems that a given architecture can approach efficiently, and the amount and type of information that can be stored and shared. By utilizing a uniform computational component and thus a uniform representation, information should be able to be stored, retrieved, modified and shared between computational components with much greater flexibility. As the complexity of computational architectures grows, the

computational requirements also grow dramatically. The use of uniform computational components may lend itself well to parallel processing implementations.

Overall, the study and design of memetic and meta-learning methods is in its infancy with great opportunities for development, scientific exploration, and rewarding applications.

BIBLIOGRAPHY

[1] J.-L. Gaudiot, J.-Y. Kang, and W. W. Ro, "Techniques to Improve Performance Beyond Pipelining: Superpipelining, Superscalar, and VLIW, ," *Advances in Computers,* 2005.

[2] J. Kolodner, *Case-Based Reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[3] L. M. Wills and J. Kolodner, "Towards More Creative Case-Based Design Systems," *Proceedings of the Twelfth Annual National Conference on Artificial Intelligence (AAAI-94),* pp. 50-55, 1994.

[4] D. H. Woldpert and W. G. Macready, "No Free Lunch Theorms for Optimization," *IEEE Transactions on Evolutionary Computation,* vol. 1, pp. 67-82, 1997.

[5] Y. S. Ong and A. J. Keane, "Meta-Lamarckian learning in memetic algorithms," *IEEE Transactions on Evolutionary Computation,* vol. 8, pp. 99--110, 2004.

[6] O. Francois and C. Lavergne, "Design of evolutionary algorithms-A statistical perspective," *Evolutionary Computation, IEEE Transactions on,* vol. 5, pp. 129-148, 2001.

[7] N. Krasnogor and J. Smith, "A tutorial for competent memetic algorithms: model, taxonomy, and design issues," *Evolutionary Computation, IEEE Transactions on,* vol. 9, pp. 474-488, 2005.

[8] Y. Xin, "Evolving artificial neural networks," *Proceedings of the IEEE,* vol. 87, pp. 1423-1447, 1999.

[9] L. I. Kuncheva and L. C. Jain, "Designing classifier fusion systems by genetic algorithms," *Evolutionary Computation, IEEE Transactions on,* vol. 4, pp. 327-336, 2000.

[10] J. T. Lee, E. Lau, and H. Yu-Chi, "The Witsenhausen counterexample: a hierarchical search approach for nonconvex optimization problems," *Automatic Control, IEEE Transactions on,* vol. 46, pp. 382-397, 2001.

[11] S. A. Kazarlis, S. E. Papadakis, J. B. Theocharis, and V. Petridis, "Microgenetic algorithms as generalized hill-climbing operators for GA optimization," *Evolutionary Computation, IEEE Transactions on,* vol. 5, pp. 204-217, 2001.

[12] M. Milano and A. Roli, "MAGMA: a multiagent architecture for metaheuristics," *Systems, Man, and Cybernetics, Part B, IEEE Transactions on,* vol. 34, pp. 925-941, 2004.

[13] Y.-S. Ong, M.-H. Lim, N. Zhu, and K.-W. Wong, "Classification of Adaptive Memetic Algorithms: A Comparative Study," *IEEE Transactions on Systems, Man and Cybernetics Part B,* vol. 36, 2006.

[14] M.-H. Lim, S. Gustafson, N. Krasnogor, and Y.-S. Ong, "Editorial to the first issue," *Memetic Computing Journal,* 2008.

[15] D. Shahaf and E. Amir, "Towards a Theory of AI Completeness," *8th Interational Symposium on Logic Formalizations of Commonsense Reasoning,* 2007.

[16] D. Lenat and R. V. Guha, *Building Large Knowledge-Based Systems*: Addison-Wesley, 1989.

[17] J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in *International Joint Conference on Artificial Intelligence*, 1989, pp. 768-774.

[18] A. D. De Groot and F. Gobet, *Perception and Memory in Chess*: Assen: Van Gorcum, 1996.

[19] Z. Pizlo, E. Stefanov, J. Saalweachter, Z. Li, Y. Haxhimusa, and W. G. Kropatsch, "Traveling Salesman Problem: A Foveating Pyramid Model," *The Journal of Problem Solving,* vol. 1, 2006.

[20] F. Gobet, *Moves in Mind*. East Sussex: Psychology Press, 2004.

[21] D. Holding, *The Psychology of the Chess Skill*. Hillsdale: L. Erlbaum Assoc., 1985.

[22] P. Saariluouma, *Chess Players' Thinking*. New York: Routlege, 1995.

[23] D. Dennett, *Darwin's Dangerous Idea*. New York: Touchstone Press, 2005.

[24] P. Moscato, "On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms," Caltech Concurrent Computation Program, C3P Report, 826.1989.

[25] Q. H. Nguyen, Y. S. Ong, and N. Krasnogor, "A study on the design issues of Memetic Algorithm " presented at the IEEE Congress on Evolutionary Computation Singapore, 2007.

[26] N. Krasnogor, "Studies on the Theory and Design Space of Memetic Algorithms," Ph.D, Faculty Comput., Math. Eng, Univ.West of England, Bristol, U.K. , 2002.

[27] G. Kendall, E. Soubeiga, and P. Cowling, "Choice function and random hyperheuristics," presented at the 4th Asia-Pacific Conference on Simulated Evolution And Learning, 2002.

[28] Q.-H. Nguyen, Y.-S. Ong, and M.-H. Lim, "Non-Genetic Transmission of Memes by Diffusion," presented at the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO'08), Atlanta, GA, 2008.

[29] N. Krasnogor and S. Gustafson, "A Study on the use of "self-generation" in memetic algorithms," *Natural Computing,* vol. 3, pp. 53 - 76, 2004.

[30] J. E. Smith, " Coevolving Memetic Algorithms: A Review and Progress Report," *IEEE Transactions on Systems, Man, and Cybernetics, Part B,* vol. 37, pp. 6-17, 2007.

[31] M. G. Norman and P. Moscato, "A Competitive and Cooperative Approach to Comple Combinatorial Search," CalTech Concurrent Computation Program, C3P Report 7901989.

[32] W. E. Hart, "Adaptive Global Optimization with Local Search," University of California, 1994.

[33] M. W. S. Land, "Evolutionary Algorithms with Local Search for Combinatorial Optimization," UNIVERSITY OF CALIFORNIA, 1998.

[34] D. V. a. S. Arnold, R., "Evolutionary Gradient Search Revisited," *IEEE Transactions on Evolutionary Computation,* vol. 11, pp. 480 - 495, 2007.

[35] J. Tang, M. H. Lim, and Y. S. Ong, "Diversity-Adaptive Parallel Memetic Algorithm for Solving Large Scale Combinatorial Optimization Problems," *Soft Computing,* vol. 11, pp. 873–888, 2007.

[36] R. J. Meuth and D. C. Wunsch, II, "Divide and Conquer Evolutionary TSP Solution for Vehicle Path Planning," presented at the Congress on Evolutionary Computation (WCCI'08), 2008.

[37] H. Wang, D. Wang, and S. Yang, "A memetic algorithm with adaptive hill climbing strategy for dynamic optimization problems," *Soft Computing,* 2009.

[38] G. Gutin and D. Karapetyan, "A selection of useful theoretical tools for the design and analysis of optimization heuristics," *Memetic Computing Journal,* 2008.

[39] A. S. S. M. B. Ullah, R. A. Sarker, D. Cornforth, and C. Lokan, "An agent-based memetic algorithm (AMA) for solving constrained optimazation problems," in *IEEE Congress on Evolutionary Computation*, Singapore, 2007, pp. 999-1006.

[40] S. M. K. Hasan, R. Sarker, D. Essam, and D. Cornforth, "Memetic Algorithms for Solving Job-Shop Scheduling Problems," *Memetic Computing Journal,* 2008.

[41] E. L. Beinenstock, L. Cooper, and P. Munro, "Theory for the development of neuron selectivity: orientation specifity and binocular interaction in the visual cortex," *Journal of Neuroscience,* vol. 2, pp. 32-48, 1982.

[42] C. Johansson and A. Lansner, "Towards cortex sized artificial neural systems," *Neural Networks,* vol. 20, pp. 48-61, 2007.

[43] J. H. Holland, *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press, 1975.

[44] R. Poli, "Exact Schema Theory for Genetic Programming and Variable-Length Genetic Algorithms with One-Point Crossover," *Genetic Programming and Evolvable Machines* vol. 2, pp. 123-163, 2001.

[45] D. E. Rumelhart, "Schemata: The building blocks of cognition. ," in *Theoretical Issues in Reading And Comprehension*, B. B. R.J. Sprio, & W.F. Brewer, Ed., ed NJ: Erlbaum, 1980.

[46] W. Smart and M. Zhang, "Applying Online Gradient Descent Search to Genetic Programming for Object Recognition," in *Second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, Dunedin, New Zealand, 2004.

[47] A. Topchy and W. F. Punsch, "Faster Genetic Programming based on Local Gradient Search of Numeric Leaf Values," in *Genetic and Evolutionary Computation Conference*, 2001.

[48] R. Dawkins, *The Selfish Gene*: Oxford University Press, USA, 1989.

[49] M. O'Neill and C. Ryan, "Automatic Generation of High Level Functions using Evolutionary Algorithms," in *1st International Workshop on Soft Computing Applied to Software Engineering*, 1999.

[50] J. P. Rosca, "Genetic Programming Exploratory Power and the Discovery of Functions," in *Conference on Evolutionary Programming*, 1995, pp. 719-736.

[51] M. O'Neill and C. Ryan, "Grammatical Evolution," *IEEE Transactions on Evolutionary Computation,* vol. 5, pp. 349-358, 2001.

[52] M. Minsky, *The Society of Mind*: Simon & Schuster Inc., 1986.

[53] M. Abramson and H. Wechsler, "Competitive reinforcement learning for combinatorial problems," in *International Joint Conference on Neural Networks, 2001. Proceedings. IJCNN '01.* , 2001, pp. 2333-2338 vol.4.

[54] A. Agarwal, M.-H. Lim, M.-J. Er, and C.-Y. Chew, "ACO for a new TSP in region coverage," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 1717-1722.

[55] P. J. Angeline, "Evolutionary Algorithms and Emergent Intelligence," Doctoral Thesis, Ohio State University, Columbus, OH, 1993.

[56] R. Baraglia, J. I. Hidalgo, and R. Perego, "A hybrid heuristic for the traveling salesman problem," *IEEE Transactions on Evolutionary Computation,* vol. 5, pp. 613-622, 2001.

[57] J. Dang and Z. Zhang, "A Polynomial Time Evolutionary Algorithm for the Traveling Salesman Problem," in *International Conference on Neural Networks and Brain*, 2005, pp. 47-49.

[58] J. R. Koza, "Evolution and co-evolution of computer programs to control independent-acting agents," in *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, 1991.

[59] P. Merz and B. Freisleben, "Genetic Local Search for the TSP: new results," in *IEEE Conference on Evoluationary Computation*, 1997, pp. 159-164.

[60] S. Mulder and D. C. Wunsch, "Million City Traveling Salesman Problem Solution by Divide and Conquer Clustering with Adaptive Resonance Neural Networks," *Neural Networks,* July 2003.

[61] H. D. Nguyen, I. Yoshihara, K. Yamamori, and M. Yasunaga, "Modified Edge Recombination Operators of Genetic Algorithms for the Traveling Salesman Problem," in *26th Annual Conference of the IEEE Industrial Electronics Society, 2000*, 2000, pp. 2815-2820.

[62] G. A. Carpenter and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision, Graphics, and Image Processing,* vol. 37, pp. 54-115, 1987.

[63] G. A. Carpenter and S. Grossberg, "Fuzzy ART: Fast Stable Learning and Categorization of analog patters by an adaptive resonance system," *Neural Networks,* vol. 4, pp. 759-771, 1991.

[64] J. R. Williamson, "Gaussian ARTMAP: A neural network for fast incremental learning of noisy multidimensional maps.," *Neural Networks,* vol. 9, pp. 881-897, 1996.

[65] M. J. Healy, R. D. Olinger, R. J. Young, T. P. Caudell, and K. W. Larson, "Modification of the ART1 architecture based on category theoretic design principles," *Neural Networks,* vol. 1, pp. 457-462, 2005.

[66] W. B. Langdon and R. Poli, *Foundations of Genetic Programming*. New York: Springer Verlag, 2002.

[67] J. R. Koza, "The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems," in *Dynamic, Genetic and Chaotic Programming*, ed: John Wiley, 1992, pp. 201-321.

[68] J. R. Koza, "Simultaneous Discovery of Detectors and a Way of Using the Detectors via Genetic Programming," in *International Conference on Neural Networks*, 1993.

[69] J. P. Rosca, "Hierarchical Learning with Procedural Abstraction Mechanisms," 1997.

[70] J. R. Koza, "Hierarchical Automatic Function Definition in Genetic Programming," in *Foundations of Genetic Algorithms 2*, ed: Morgan Kaufmann, 1992, pp. 297-318.

[71] I. Dempsey, M. O'Neill, and A. Barbazon, *Foundations in Grammatical Evolution for Dynamic Environments*: Springer, 2009.

[72] G. A. Carpenter, S. Grossberg, and D. B. Rosen, "ART 2-A: An adaptive resonance algorithm for rapid category learning and recognition," *Neural Networks,* vol. 4, pp. 493-504, 1991.

[73] R. Xu and D. Wunsch, II, "Survey of clustering algorithms," *Neural Networks, IEEE Transactions on,* vol. 16, pp. 645-678, 2005.

[74] G. A. Carpenter and N. Markuzon, "ARTMAP-IC and medical diagnosis: Instance counting and inconsistent cases," *Neural Networks,* vol. 11, pp. 323-336, 1998.

[75] G. C. Anagnostopoulos and M. Georgiopoulos, "Ellipsoidal ART and ARTMAP for incremental clustering and classification," in *Int. Joint Conference on Neural Networks*, 2001, pp. 1221-1226.

[76] S. Grossberg, "Adaptive Pattern Recognition and Universal encoding II: Feedback, Expectation, Olfaction and Illusions," *Biological Cybernetics,* vol. 23, pp. 187-202, 1976.

[77] J. Huang, M. Georgiopoulos, and G. Heileman, "Fuzzy ART Properties," *Neural Networks,* vol. 8, pp. 203-213, 1995.

[78] M. Georgiopoulos, I. Dagher, G. L. Heileman, and G. Bebis, "Properties of Learning in a Fuzzy ART Variant," *Neural Networks,* vol. 12, pp. 837-850, 1999.

[79] G. Anagnostopoulos and M. Georgiopoulos, "Category regions as new geometrical concepts in Fuzzy-ART and Fuzzy-ARTMAP," *Neural Networks,* vol. 15, pp. 1205-1221, 2002.

[80] W. J. Conover, *Practical Nonparametric Statistics*. New York: John Wiley and Sons, 1999.

[81] A. Asuncion and D. J. Newman. Fisher's Iris Dataset [Online]. Available: http://www.ics.uci.edu/~mlearn/MLRepository.html

[82] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annual Eugenics,* vol. 7, pp. 179-188, 1936.

[83] J. Schlimmer, G. H. Lincoff, and A. A. Knopf. Mushroom Database [Online]. Available: http://www.ics.uci.edu/~mlearn/MLRepository.html

[84] W. Duch, R. Adamczak, and K. Grabczewski, "Extraction of Logical Rules from Training data using backpropagation networks," *Proceedings of the 1st Online Workshop on Soft Computing,* pp. 25-30, 1996.

[85] B. d. l. Iglesia, G. Richards, M. S. Philpott, and V. J. Rayward-Smith, "The application and effectiveness of a multi-objective metaheuristic algorithm for partial classification," *European Journal of Operational Research,* vol. 169, pp. 898-917, 2006.

[86] J. Wang and G. Karypis, "On efficiently summarizing categorical databases," *Knowledge and Information Systems,* vol. 9, pp. 19-37, 2006.

[87] G. A. Carpenter, S. Grossberg, and J. H. Reynolds, "ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network.," *Neural Networks,* vol. 4, pp. 565-588, 1991.

[88] C. L. Blake and C. J. Merz. UNIX User Dataset [Online]. Available: http://archive.ics.uci.edu/ml/datasets/UNIX+User+Data

[89] S. Aeberhard, D. Coomans, and O. d. Vel, "The Performance of Statistical Pattern Recognition Methods in High Dimensional Settings," presented at the IEEE Signal Processing Workshop on Higher Order Statistics, Ceasarea, 1994.

[90] D. Sculley and C. E. Brodley, "Compression and machine learning: a new perspective on feature space vectors," in *Data Compression Conference, 2006. DCC 2006. Proceedings*, 2006, pp. 332-341.

[91] S. Lin and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Operations Research,* vol. 21, pp. 498-516, March-April 1973 1973.

[92] D. C. Wunsch and S. Mulder, "Using Adaptive Resonance Theory and Local optimization to divide and conquer large scale traveling salesman problems," in *International Joint Conference on Neural Networks*, 2003, pp. 1408-1411.

[93] H.-K. Tsai, J.-M. Yang, and C.-Y. Kao, "An evolutionary algorithm for large traveling salesman problems," *IEEE Transactions on Systems, Man and Cybernetics Part B,* vol. 34, pp. 1718-1729, August 2004.

[94] H.-K. Tsai, J.-M. Yang, and C.-Y. Kao, "Solving traveling salesman problems by combining global and local search mechanisms," in *Conference on Evolutionary Computation*, 2002, pp. 12-17.

[95] L. Wang, A. A. Maciejewski, H. J. Seigel, and V. P. Roychowdhury, "A comparitive study of five parallel genetic algorithms using the traveling salesman problem," in *First Merged International Conference and Symposium on Parallel and Distributed Processing*, 1998, pp. 345-349.

[96] H. D. Nguyen, I. Yoshihara, K. Yamamori, and M. Yasunaga, "Implementation of an Effective Hybrid GA for Large Scale Traveling Salesman Problems," *IEEE Transactions on Systems, Man and Cybernetics Part B,* vol. 37, pp. 92-99, 2007.

[97] D. Applegate, W. Cook, and A. Rohe, "Chained Lin-Kernighan for Large Traveling Salesman Problems," *INFORMS Journal on Computing,* vol. 15, pp. 82-92, 2003.

[98] A. Agarwal, M.-H. Lim, M. J. Er, and T. N. Nguyen, "Rectilinear workspace partitioning for parallel coverage using multiple UAVs," *Advanced Robotics,* vol. 21, 2007.

[99] A. L. Zobrist, "A Hashing Method with Applications for Game Playing," University of Wisconsin, Madison, Wisconsin1969.

[100] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*: Prentice-Hall, 2001.

[101] G. Reinelt, "TSPLIB - A Traveling Salesman Problem Library," *ORSA Journal on Computing,* vol. 3, pp. 376-384, 1991.

VITA

Ryan James Meuth was born April 11, 1982. He received his Bachelors and Masters degrees in Computer Engineering from Missouri University of Science and Technology (formerly the University of Missouri – Rolla) in 2005 and 2007, respectively. He received his PhD in Computer Engineering from Missouri University of Science and Technology in December 2009. His research interests include optimizing the behavior of robot swarms, large scale optimization problems such as computer Go, and high performance computing methods utilizing video game consoles and graphics processing units.