IMPROVING MEMORY FOR OPTIMIZATION AND LEARNING IN DYNAMIC ENVIRONMENTS

Gregory John Barlow

CMU-RI-TR-11-17

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Robotics.

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

July 2011

Thesis committee:
Stephen F. Smith (chair)
Katia Sycara
Laura Barbulescu
Jürgen Branke, University of Warwick

Abstract

Many problems considered in optimization and artificial intelligence research are static: information about the problem is known a priori, and little to no uncertainty about this information is presumed to exist. Most real problems, however, are dynamic: information about the problem is released over time, uncertain events may occur, or the requirements of the problem may change as time passes. One technique for improving optimization and learning in dynamic environments is by using information from the past. By using solutions from previous environments, it is often easier to find promising solutions in a new environment. A common way to maintain and exploit information from the past is the use of memory, where solutions are stored periodically and can be retrieved and refined when the environment changes. Memory can help search respond quickly and efficiently to changes in a dynamic problem. Despite their strengths, standard memories have many weaknesses which limit their effectiveness. This thesis explores ways to improve memory for optimization and learning in dynamic environments. The techniques presented in this thesis improve memories by incorporating probabilistic models of previous solutions into memory, storing many previous solutions in memory while keeping overhead low, building long-term models of the dynamic search space over time, allowing easy refinement of memory entries, and mapping previous solutions to the current environment for problems where solutions may become obsolete over time. To address the weaknesses and limitations of standard memory, two novel classes of memory are introduced: density-estimate memory and classifier-based memory. Density-estimate memory builds and maintains probabilistic models within memory to create density estimations of promising areas of the search space over time. Density-estimate memory allows many solutions to be stored in memory, builds long-term models of the dynamic search space, and allows memory entries to be easily refined while keeping overhead low. Density-estimate memory is applied to three dynamic problems: factory coordination, the Moving Peaks benchmark problem, and adaptive traffic signal control. For all three problems, density-estimate memory improves performance over a baseline learning or optimization algorithm as well as state-of-the-art algorithms. Classifier-based memory allows dynamic problems with shifting feasible regions to capture solutions in memory and then map these memory entries to feasible solutions in the future. By storing abstractions of solutions in the memory, information about previous solutions can be used to create solutions in a new environment, even when the old solutions are now completely obsolete or infeasible. Classifier-based memory is applied to a dynamic job shop scheduling problem with sequence-dependent setup times and machine breakdowns and repairs. Classifier-based memory improves the quality of schedules and reduces the amount of search necessary to find good schedules. The techniques presented in this this thesis improve the ability of memories to guide search quickly and efficiently to good solutions as the environment changes.

To Liddy

Acknowledgements

I would like to begin by thanking my advisor, Stephen Smith. Without his support and patience, the completion of this dissertation would not have been possible. He has helped guide my work while allowing me the freedom to try out new ideas that didn't always work out right away.

I would like to thank my committee members, Katia Sycara, Laura Barbelescu, and Jürgen Branke for their help with this dissertation. I would also like to thank Zack Rubinstein, Drew Bagnell, and Anthony Gallagher for serving on my research qualifier committee.

The work on adaptive traffic control in Chapter 8 owes a great deal to many discussions with Xiao-Feng Xie, Steve, and Zack. Maps, traffic signal timing plans, and traffic demand data were provided by the City of Pittsburgh.

I'd like to thank everyone in the Robotics Institute for a wonderful experience. I'd especially like to thank all the members of the Intelligent Coordination and Logistics Laboratory.

Many people helped me along the path toward a Ph.D. When I was a senior in high school, Edward Grant gave me the chance to do robotics research, and he helped me develop as a researcher during my years at North Carolina State University. He also fostered a spirit of excitement around research that I try my best to live up to. Choong Oh introduced me to genetic programming and provided me with an excellent topic for my master's thesis. I thoroughly enjoyed my four summers working with him at the United States Naval Research Laboratory.

For the first three years of my doctoral work, I was supported by a National Defense Science and Engineering Graduate fellowship. The Robotics Institute has also been kind enough to continue my funding even during the moments when I had doubts that I would ever finish.

Without my large, wonderful family, none of this would have been possible. I'd particularly like to thank my parents, John and Cheryl Barlow, and my three sisters, Logan, Lindsey, and Gwen. Most of all, I would like to thank my wife, Liddy, and my daughter, Philippa. Liddy has been my greatest supporter during my time at Carnegie Mellon, and I could not have done this without her.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

When confronted with a changing world, humans are apt to look not just to the future, but to the past. Drawing on knowledge from similar situations we have encountered helps us to decide what to do next. The more experience we've had with a particular situation, the better we can expect to perform when we encounter it again. When solving dynamic problems using search, it may be enough to solve the problem completely from scratch, but incorporating information from the past into optimization and learning can lead to a more adaptive search process. Like human experience, the richer the information from past events, the better we can expect dynamic optimization to perform. By creating memories capable of building rich models of past experiences, this thesis attempts to develop more effective memories for learning and optimization in dynamic environments.

Many problems considered in optimization and artificial intelligence research are static: information about the problem is known a priori, and little to no uncertainty about this information is presumed to exist. Most real problems, however, are dynamic: information about the problem is released over time, uncertain events may occur, or the requirements of the problem may change as time passes. A common approach to dynamic problems is to consider each change as the beginning of a new static problem, and solve the problem from scratch given the new information. If time is not an issue, this is a fine solution, but in many cases, finding a good solution quickly is more important than finding an optimal solution. Approaches to dynamic optimization problems must balance the speed of arriving at a solution with the fitness of the solutions produced.

There are many approaches to solving dynamic problems, most originally designed for static problems. Many techniques have been shown to help approaches designed for static problems work well on dynamic problems. One of the most common techniques is the use of information from the past to improve current performance. In a purely stochastic domain, information about the past might not be meaningful, but in many dynamic problems, the current state of the environment is often similar to previously seen states. By using information from the past, it may be easier to find promising

solutions in the new environment. The system may be more adaptive to change and perform better over time. A common way to maintain and exploit information from the past is the use of memory, where solutions are stored periodically and can be retrieved and refined when the environment changes.

Memory aids dynamic optimization in several ways. By maintaining good solutions from the past, memory may speed up search for similar solutions after a dynamic event. Memory entries provide additional points to search from after a change, and once search has converged may help inject diversity into the search process. Memory also helps to build a model of the dynamic problem over time. Memory has been used extensively for dynamic learning and optimization, and while there are many types of memory, a standard memory system for dynamic optimization has emerged. In this standard memory system, a finite number of solutions are stored and then incorporated into search as the problem changes to direct the search process toward good solutions.

For many problems, this type of memory has helped to improve the performance of dynamic optimization. However, because the memory size is finite, and typically small, it may be difficult to store enough solutions to accurately model the search space over time. It may also be difficult to refine the memory over time, as the only way to change an entry is to completely replace it. Parts of the memory system that could benefit from a good model of the dynamic fitness landscape, like retrieving from the memory and maintaining diversity in the search, are typically done in uninformed ways.

Most memory systems are limited to problems where all solutions in the search space are feasible throughout the course of optimization or learning. In some problems, the feasible solutions at a particular time are only a subset of the total search space, and the set of feasible solutions changes over the course of the dynamic problem. For example, in a dynamic rescheduling problem, jobs are completed and new jobs arrive. An old schedule containing only jobs that have been completed is not a feasible schedule for the jobs that are currently available. When the feasible region of the search space shifts over time, the typical memory system is either not applicable or not very useful.

This thesis improves optimization and learning in dynamic environments through enhanced memory systems. The improved memories presented in this thesis address and overcome the weaknesses and limitations of a standard memory system and enable memory to construct long-term probabilistic models of the dynamic search space, aggregate information from many previous solutions, easily refine the models in the memory, and extend memory to new types of dynamic problems. The enhanced memory systems presented in this thesis improve the performance of optimization and learning algorithms on dynamic problems by allowing search to respond more quickly to change and helping to locate better solutions.

1.1 Overview

This thesis explores the use of memory for improving optimization and learning in dynamic environments. Memory helps learning and optimization algorithms respond quickly and efficiently to changes in dynamic environments. Despite their many strengths, standard memories also have many weaknesses which limit their effectiveness. This thesis presents improved memories which overcome many weaknesses and limitations of previous memory systems, enhancing the performance of optimization and learning algorithms in dynamic environments.

Many prior works have investigated dynamic problems where changes are small and algorithms must track those changes quickly. However, many real problems are discontinuous, with changes that revisit previous solution areas as the problem progresses. These types of problems include dynamic scheduling and adaptive traffic signal control. By using information from previous environments, it may be easier to find promising solutions in new environments. A common way to use information from the past is memory, where good solutions are stored over time and can be retrieved when the environment changes.

Standard memory systems have been shown to help improve search algorithms on dynamic problems, but memories typically only build simple models of promising regions of the search space and store a small number of solutions located in areas where the underlying optimization or learning algorithm may find more good solutions. These memories do not capture much information about the structure of the search space over time, and memory entries cannot usually be refined without completely replacing them with a better solution. Memories are also not applicable to all types of dynamic problems, particularly those problems where solutions stored in the memory can become irrelevant or infeasible.

To address the weaknesses and limitations of standard memory, two novel classes of memory are introduced: density-estimate memory and classifier-based memory. Density-estimate memory builds and maintains probabilistic models within a memory to create rich density estimations of promising areas of the search space as it changes over time. Classifier-based memory allows dynamic problems with shifting feasible regions to capture solutions in memory and then map these memory entries to feasible solutions in the future.

Density-estimate memory solves the problem of limited storage of previous solutions, weak models of the dynamic search space, and difficulty in refining the contents of memory, all while keeping memory overhead low. Density-estimate memory maintains a similar structure to a standard memory, but instead of storing only one solution in a memory entry, density-estimate memory clusters many solutions into an entry, then uses probabilistic models to estimate the density of points in the cluster. It builds rich models of the dynamic search space efficiently and then uses that information to improve the quality of solutions returned to the search process. These models allow the memory

to capture much more information about the structure of the search space over time. By building models within the memory, the search process can continue to interact with a finite number of entries, except now these memory entries aggregate many individual solutions. The models stored in memory can be continuously refined as new solutions are stored to the memory. These models can be used to help keep search diverse in a more informed way and help retrieve the most useful solutions from the memory.

In this thesis, density-estimate memory is applied to three dynamic problems: factory coordination, the Moving Peaks benchmark problem, and adaptive traffic signal control. Though all three of these problems have dynamic environments, these problems are very different from one another. In the factory coordination problem, incoming jobs must be distributed among several machines to maintain the flow of jobs through a simulated factory. Density-estimate memory is used to boost the performance of a reinforcement learning algorithm. The Moving Peaks problem is a common benchmark from the literature that allows the nature of the search space to be highly configured. For this problem, density-estimate memory is combined with an evolutionary algorithm to improve performance on the problem. Traffic signal control is another highly dynamic problem. Traffic demands at an intersection constantly change due to the time of day, the types of vehicles, and the control of surrounding traffic signals. Experiments apply density-estimate memory to an adaptive learning algorithm on two road networks based on intersections in downtown Pittsburgh, Pennsylvania. For all three of these problems, density-estimate memory improves performance over a baseline learning or optimization algorithm. Density-estimate memory also outperforms state-of-the-art algorithms for each problem.

Classifier-based memory allows the use of memory for dynamic problems where the feasible region of the search space shifts over time. For most dynamic problems, the use of memory by an optimization or learning algorithm is straightforward. Though a solution may have been stored in the memory long ago, that solution is typically feasible no matter when it is retrieved from the memory. For some types of problems such as dynamic scheduling, the feasible region of the search space shifts over time as the problem changes. In dynamic scheduling problems, the current jobs change over time. If a solution to the problem is represented by a prioritized list of jobs to be fed to a schedule builder, any memory that stores a solution directly will quickly become irrelevant. Some jobs in the solution will be completed, other jobs will become more or less important, and new jobs that have arrived since the solution was stored will not be included at all. In classifier-based memory, solutions are classified based on the attributes of available jobs at the time the solution is stored in memory. When a memory entry is retrieved, this classification is mapped onto the currently available jobs to create a new feasible solution. This mapping allows classifier-based memory to help transfer information from previous schedules to the current environment.

Classifier-based memory is applied to a dynamic job shop scheduling problem with sequence-

dependent setup times and machine breakdowns and repairs. An evolutionary algorithm scheduler is used to build schedules. Classifier-based memory is compared to the standard evolutionary algorithm as well as several other evolutionary algorithm approaches to dynamic optimization. Classifier-based memory improves the quality of the schedules and reduces the amount of search necessary to find good schedules.

1.2 Contributions

The major goal of this thesis is to improve optimization and learning in dynamic environments using memory. Novel enhanced memory systems are presented to address the weaknesses and overcome the limitations of standard memory systems. This thesis contributes specifically to the improvement of memories for optimization and learning in dynamic environments by:

Incorporating probabilistic models of previous solutions into memory Instead of storing individual solutions separately, previous solutions are aggregated using probabilistic models.

Storing many previous solutions in memory while keeping overhead low While the use of probabilistic models allows many more solutions to be stored in memory, using a limited number of probabilistic models allows the underlying learning or optimization algorithm to interact only with the models, not with the solutions that create those models. By clustering the solutions stored in memory, only some of the probabilistic models in memory need to be rebuilt when new solutions are stored.

Building rich, long-term models of the dynamic search space over time By storing solutions across many different environments, probabilistic models provide density estimation for the search space over time. This density estimation gives a long-term model of promising areas of the search space.

Allowing easy refinement of memory entries By allowing memory entries to be updated by adding new solutions to the memory, rather having to replace previous memory entries, the memory can be refined incrementally, producing richer models of where good solutions may exist in the dynamic search space.

Mapping previous solutions to the current environment for problems where solutions may become obsolete By storing abstractions of solutions in the memory, information about previous

solutions can be used to create solutions in a new environment, even when the old solutions would be completely obsolete and infeasible.

Memories

This thesis contributes two new classes of memory for dynamic problems that make the improvements outlined above:

Density-estimate memory Density-estimate memory is introduced to allow memories to effectively store many previous solutions without substantially increasing the overhead associated with maintaining and using the memory. Density-estimate memories build and maintain probabilistic models of past solutions to improve the performance of learning and optimization in dynamic environments. While building much richer models of the dynamic search space than a standard memory, density-estimate memories can be constructed efficiently and maintained with a low amount of overhead. Density-estimate memory also builds better long-term models of the dynamic search space and makes it easier to refine memory entries.

Classifier-based memory Classifier-based memory is introduced to extend the use of memory to dynamic problems where solutions may become obsolete as the environment changes. Classifier-based memory creates an abstraction layer between feasible solutions and memory entries so that old solutions stored in memory may be mapped to solutions that are feasible in the current environment. Classifier-based memory allows dynamic problems with shifting feasible regions to use memory to improve search.

Algorithms

This thesis contributes several algorithms that implement these novel classes of memory and one algorithm for reinforcement learning of traffic signal control:

Incremental Euclidean clustering density-estimate memory This algorithm uses simple, incremental clustering to separate solutions into memory entries. The cluster centers are used as the models in the memory. This is the simplest density-estimate memory implementation.

Incremental Gaussian clustering density-estimate memory This algorithm uses incremental clustering to form memory entries, then creates Gaussian models for each memory entry. This implementation provides good density estimation of previous solutions.

Gaussian mixture model density-estimate memory For problems that have more time to build models, a Gaussian mixture model implementation of density-estimate memory is presented. This implementation requires building a model over all solutions in the memory, which is more expensive than the incremental clustering implementations.

Classifier-based memory for job-shop scheduling An implementation of classifier-based memory is presented for job-shop scheduling. This algorithm uses attributes of jobs and operations in the problem to classify schedules and allow them to stored in memory and then retrieved later.

Balanced phase utilization algorithm This thesis also contributes a traffic-responsive learning algorithm for traffic signal control, the balanced phase utilization algorithm (Chapter 8). This algorithm requires only the use of exit detectors in order to balance the splits and coordinate the offset for traffic signals.

Benchmark problems

This thesis defines several problems that may be used as benchmarks to evaluate the performance of optimization and learning algorithms in dynamic environments:

Factory coordination A distributed, dynamic factory coordination problem with long time horizons is defined in Chapter 6.

Traffic signal control An adaptive traffic control problem that models 32 intersections in the city of Pittsburgh, PA is defined in Chapter 8.

Long-term job-shop scheduling A dynamic scheduling problem with sequence-based setups and machine breakdowns is defined in Chapter 9.

1.3 Outline

This thesis is divided into three parts. Part I discusses optimization and learning in dynamic environments and how remembering information from the past can help to find new solutions. Part II introduces a new type of memory that builds density-estimate models of information from the past. Part III introduces a new abstraction layer that allows memory to be used on problems with shifting feasible regions.

Part I focuses on defining the problem of optimization and learning in dynamic environments. Chapter 2 classifies the different types of dynamic environments and discusses several dynamic benchmark problems. Chapter 3 surveys prior work that has been done in the area of dynamic optimization and learning, particularly the use of memory for dynamic optimization. Chapter 4 defines a standard memory that has been widely used in the literature for dynamic optimization and discusses the strengths and weaknesses of this approach.

Part II investigates extending the standard memory by building and storing probabilistic models within memory. Chapter 5 introduces a new class of memory called density-estimate memory. Density-estimate memory is a new memory technique that builds density estimation models to aggregate information from many solutions stored in memory. In the next three chapters, density-estimate memory is applied to optimization and learning algorithms for three dynamic problems. Chapter 6 applies density-estimate memory to a reinforcement learning algorithm on a distributed, dynamic factory coordination problem and compared to standard memory. Chapter 7 compares density-estimate memory to a variety of other techniques for improving evolutionary algorithms on dynamic problems using the Moving Peaks benchmark problem. In Chapter 8, density-estimate memory is applied to an adaptive, traffic-responsive algorithm for traffic signal control. The adaptive traffic signal controllers are compared to the real signal timing plans for a 32 intersection traffic network modeled on downtown Pittsburgh, Pennsylvania.

Part III considers the use of memory for dynamic problems with shifting feasible regions. Chapter 9 introduces a new memory technique called classifier-based memory for problems where the feasible region of the search space shifts over time. Classifier-based memory creates an abstraction layer allowing old solutions to be mapped to current feasible solutions.

The dissertation concludes in Chapter 10 with a summary of this thesis and an outlook on potential directions for future work on extending and improving memory for dynamic optimization.

Part I

Optimization and learning in dynamic environments

Chapter 2

Dynamic environments

Many of the problems considered in optimization and learning assume that solutions exist in a static, unchanging environment. If the environment does change, one may simply treat the new environment as a completely new version of the problem that can be solved as before. When a problem changes infrequently or only in small amounts, this can be a reasonable approach. However, this assumption tends to break down when the environment undergoes frequent, discontinuous changes. When this occurs, a search process may be slow to react, hurting performance in the time it takes to find a new solution. Instead of focusing only on finding the best solution to a dynamic problem, one must often balance the quality of solutions with the speed required to find good solutions.

This chapter will describe the nature of dynamic environments: those environments that change over time regardless of the actions of optimization or learning algorithms. The chapter will begin by providing a basic definition of dynamic problems, discussing the challenges of dynamic problems that separate them from static problems, and defining the terminology that will be used throughout this thesis. A classification system for dynamic environments will be presented along with a brief discussion of common types of dynamic problems and how to construct optimization and learning algorithms to respond to particular types of dynamic environments.

2.1 Dynamic problems

In a problem with a dynamic environment, the objective function, problem formulation, constraints, or some other part of the problem changes over time independent of the actions of an optimization or learning algorithm. A change in the dynamic environment produces a change in the objective value of a given solution relative to other solutions. In many cases, this means that the optimum of the problem changes as well, so effective approaches to dynamic problems must be capable of tracking the optima over time across changes to the environment. Dynamic problems may also be

known as time-varying or changing. When changes are completely random, a dynamic problem may be known as a stochastic problem. Another term used in the literature, non-stationary, may imply more than dynamics [52] and will not be used in this thesis.

Dynamic optimization and learning lends itself to problems existing within a narrow range of problem dynamics, requiring a balance between solution fitness and search speed. If a problem changes too quickly, search may be too slow to keep up with the changing problem, and reactive techniques will outperform optimization or learning approaches. If a problem changes very slowly, a balance between optimization and diversity is often no longer necessary: one may search from scratch, treating each change as a completely new static problem. Many real problems lie in this region where learning and optimization must respond quickly to changes while still finding solutions of high fitness.

When discussing problems with dynamic environments in this thesis, a certain vocabulary will be employed. The search space is the space of all possible solutions to a problem. The feasible region of the search space is a subset of the search space that contains all solutions that meet the current constraints of the problem. The term fitness will be used for the objective function value of a solution at a given time. The term fitness landscape, common in some areas of optimization, will be used to mean the landscape of objective function values for solutions in the search space at a particular time. When a dynamic event occurs, the fitness landscape changes. The term change will typically be used to mean a change in the fitness landscape. The term environment will be used to refer to the problem formulation and constraints at a given time. For example, the environment of a dynamic scheduling problem would describe the jobs to be scheduled, the machines available to schedule them on, and any constraints. Typically, a change in the environment leads to a change in the fitness landscape.

2.2 Classifying dynamic environments

Though all dynamic problems involve repeated changes to the fitness landscape, dynamic problems may differ in many ways. Some problems may have random, continuous changes that happen very frequently, while another might have severe, infrequent changes that can be partially predicted. This section will list some of the ways that a problem with dynamic environments may be classified. This classification is similar to [14].

Frequency In some problems, changes may be rare, while in others, changes may happen constantly. Problems are also not limited to one frequency of changes. Some problems may have small, frequent changes, while other have large, infrequent changes. For example, in a dynamic scheduling problem, jobs may be drawn from some distribution. The actual distribution of the jobs currently

being scheduled might change constantly as new jobs arrive, but it may differ only slightly from the underlying distribution. Less frequently, this underlying distribution might change drastically. The most important aspect of change frequency is how long a learning or optimization algorithm has to find a solution both before it has an effect on performance and before another change occurs.

Severity As mentioned, the severity of changes also defines a dynamic problem. Some problems may have changes that are small enough to be easily tracked, while others have large, discontinuous changes. Small changes may not have a large effect on the fitness landscape, while large changes may completely change the landscape.

Predictability In some problems, changes follow a specific pattern. In others, changes are completely random. A problem with small, predictable changes will require a very different algorithm than one with severe, unpredictable changes.

Detectability In some problems, changes to the fitness landscape are easy to detect; one knows exactly when a change has occurred. In others, it may take some time before it is clear that the environment has changed. This can have a large effect on how an algorithm solves a problem.

Repeatability and structure of change Some problems may be purely stochastic, with completely random changes to the environment. For some problems, the dynamic environment may cycle through a finite number of distinct configurations. For most interesting dynamic problems, the current environment will be similar to previously seen environments. This may allow information from the past to be useful when finding solutions in the current environment.

Changes to the feasible region of the search space For some problems, the structure of the environment may be similar from one moment to the next, but the region of the search space containing feasible solutions may shift with time. For example, in a dynamic scheduling problem jobs arrive in the system, are processed, and are completed. Though a similar job may arrive later, the jobs currently being scheduled change over time. A schedule containing only completed jobs is not a feasible schedule for the jobs now available.

Influence of search on the environment Though all problems with dynamic environments undergo changes independent of the results of search, search can create additional changes in the environment. For some problems, solutions have no effect on the environment. For many others, however, the specific solution will change the environment. In traffic control, both the arrival of

new vehicles into the road network and the settings of a traffic signal determine changes to the flow of traffic. In scheduling problems, completing one job may change which of the remaining jobs are allowed to be scheduled.

2.3 Responding to change

Many prior works on optimization and learning in dynamic environments have investigated problems where changes are small and algorithms must track those changes quickly. Many real problems are much more discontinuous, with changes that are not random, but revisit previous solution areas as the problem progresses. These types of problems may include scheduling and adaptive traffic control, which are investigated in this thesis.

Problems with small, frequent, continuous changes require constant refinement to the solution, but typically do not require algorithms that allow for large, rapid changes to the solution. If the optimum moves in a continuous manner, rather than jumping between areas of the search space, search algorithms must only make incremental changes to solutions in order to follow the optimum. Many algorithms, including reinforcement learning, are well suited to these types of problems.

When changes to the environment are much more severe, a very different approach is necessary. After a change in the environment occurs, the location of the global optimum may change drastically. For discontinuous problems, search must be able to find areas containing good solutions in addition to refining those solutions to find the best solutions possible. Search algorithms that are able to explore widely across the search space after a change will have an advantage over those that search only locally. If search is population-based, introducing diversity into the search may help explore the search space after a change. For many problems, changes in the search space, though discontinuous, are not completely random. Instead, the current environment is similar to previous environments. By using information from those previous environments, it may be easier to find promising solutions in the new environment. A common way to use information from the past is memory, where good solutions are stored over time and can be retrieved when the environment changes.

Chapter 3 reviews some of the previous work in responding to change in problems with dynamic environments. Many of the interesting real-world problems with dynamic environments have severe, discontinuous changes as well as repeatable changes to their environments. Memory is one of the most effective ways to improve search on these types of problems. Chapter 4 introduces a standard memory system for aiding search in dynamic environments. While this type of memory helps search, it has several weaknesses. The remainder of this thesis explores how to improve memory systems for optimization and learning in dynamic environments and then applies the improved memories to dynamic real-world problems.

2.4 Summary

Many problems in optimization and learning have dynamic environments, where changes occur over time independent of the search process. Changes in the environment lead to changes in the objective values for solutions in the search space. Optimization and learning algorithms for dynamic environments must be able to respond to change, finding good solutions after a change has occurred. Dynamic optimization and learning is useful for problems where enough time is available that search can outperform reactive approaches but not enough time is available to treat optimize from scratch.

This chapter presented a classification of dynamic environments. A dynamic environment may be classified by how frequently the problem changes, how severe or discontinuous those changes are, how easy it is to predict when changes will occur or what those changes will look like, how easy it is to detect when changes happen, whether the changing environment revisits previous environments, whether the feasible region of the problem changes over time, and whether the results of search can produce changes to the environment. Many different types of dynamic problems may be explored, but this thesis will focus on those with discontinuous, repeatable changes.

Chapter 3

Background

There are many approaches to optimization and learning in dynamic environments, and it is outside the scope of this thesis to provide a complete accounting of all the ways one may approach problems that change over time. However, a general overview of these approaches may help place the research detailed later in the context of other work on dynamic problems.

The chapter will begin by describing some of the many algorithmic approaches to dynamic problems in the literature. Then, some of the more common dynamic benchmark problems will be discussed. The remainder of the chapter will discuss approaches to improving the performance of meta-heuristics on problems with dynamic environments, particularly diversity techniques, memory, multi-population search, and anticipation.

3.1 Algorithms for dynamic problems

Like static optimization problems, dynamic optimization problems may be solved in many ways. For some problems, algorithms may be able to find globally optimal solutions for all possible environments. For other problems, an algorithm might have to adapt quickly to changes in the environment based on very little information about the problem.

If one actually knows the function to be optimized, classical methods like calculus of variations or optimal control may be applied [23]. Optimal control may allow for the construction of a control law such that optimality may be achieved across changes in the environment. Another way to find policies for optimization problems with uncertainty a priori is the use of stochastic programming [54], which uses probability distributions about the data to find policies that are always feasible and maximize the expected value of fitness. For some dynamic problems, particularly scheduling, priority-dispatch rules may be used to handle dynamic events [50]. These rules may be learned

ahead of time. Other reactive approaches may also be learned a priori to be robust in varying environments [1, 39, 74]. Approaches like reinforcement learning can move the process of learning from dynamic events into the control process [25].

One large class of algorithms commonly used for dynamic problems are meta-heuristics, including local search [94], simulated annealing, tabu search, evolutionary algorithms [52], ant colony optimization, and particle swarm optimization. Many of these algorithms were inspired by real dynamic processes like evolution, swarm behavior, or the annealing of metals. Unlike classical methods, meta-heuristics typically do not require extensive knowledge of the function to be optimized, such as the locations of good solutions or derivatives of the function to be optimized. This is particularly advantageous, since most dynamic problems are difficult specifically because of a lack of information about the function to be optimized. Though the applications of many meta-heuristics to dynamic problems have been studied in the literature, some of the most extensive work has been done on dynamic optimization with evolutionary algorithms. Though most of the results in the remainder of this section will discuss dynamic optimization with evolutionary algorithms, meta-heuristic approaches tend to encounter the same types of difficulties with dynamic problems, and many of the approaches used for evolutionary algorithms could be applied to other meta-heuristics.

3.2 Common benchmark problems with dynamic environments

Many dynamic problems have been studied in the literature. Many of these experimental problems are related to real-world dynamic problems like scheduling and routing. Others have been designed purely as benchmark problems for investigating optimization and learning in dynamic environments.

One of the more common dynamic benchmark problems used in evaluating evolutionary algorithms for dynamic optimization is the Moving Peaks problem, developed simultaneously by Branke [13] and by Morrison and DeJong [72]. This is a dynamic, multi-modal problem with a fixed number of peaks. Depending on the parameters used, the fitness landscape may change in many different ways over time. Peaks may move within the search space, as well as change in height or shape.

Another common benchmark problem is dynamic scheduling. A single benchmark scheduling problem does not exist, though most results seem to be for dynamic job shop scheduling problems [6, 7, 16, 63, 98]. While many problems consider primarily the arrival of jobs over time, some problems also include machine breakdowns and other dynamic events [3, 24]. Ouelhadj and Petrovic [76] give a survey of approaches to dynamic scheduling which includes some explanation of dynamic scheduling problems.

Other common benchmark problems include dynamic knapsack problems [18, 94] and dynamic bit-matching [91]. Farina et al. created several multi-objective dynamic benchmark problems [35]. Van

Hentenryck and Bent consider dynamic routing in addition to scheduling and knapsack problems [94]. Several dynamic problem generators have been proposed in addition to the Moving Peaks problem. Yang created a problem generator based on decomposable trap functions [105], and Jin and Sendhoff created a benchmark generator for both single objective and multi-objective problems [53].

In addition to these benchmark problems, many real-world dynamic problems have been investigated. Morley considered an example of the factory coordination problem from a General Motors plant where trucks are allocated to reconfigurable paint booths [70, 71]. Trucks arrive off the assembly line to be painted, and then wait to be assigned to a paint booth's queue. Kokai et al. [57] considered the use of particle swarm optimization to dynamically optimize adaptive array antennas. A great deal of work has been done on adaptive traffic signal control, which is a highly dynamic problem [59, 67, 79, 89]. Prior work has considered dynamic learning and optimization for single intersections, corridors, and large networks of traffic signals.

3.3 Improving performance on dynamic problems

Many meta-heuristics, including evolutionary algorithms, ant colony optimization, and particle swarm optimization, were inspired by naturally occurring responses to dynamic problems. It follows then, that many of these algorithms would be suitable for dynamic optimization [46]. These meta-heuristics are not only adaptive, but the use of population-based search allows transfer of information from the past which is often helpful in dynamic optimization. However, when meta-heuristics converge during a run, some of this adaptability is lost, which may lead to poor results after the next change. Also, while population-based search may help transfer information from the recent past, standard versions of most meta-heuristics do not use information from the more distant past, which may be helpful when the current environment is similar to one previously encountered. Last, standard meta-heuristics typically do not seek to produce solutions that are robust or flexible to changes in the environment, something that may be useful during dynamic optimization.

Prior work has shown many techniques for improving the performance of meta-heuristics on dynamic problems. [14, 52]. These techniques fall into three broad categories based on what the approach is attempting to address: keeping the population diverse in order to avoid population convergence and maintain adaptability, storing information from the past in order to improve performance after future changes, and anticipating changes in order to produce flexible solutions. Directly or indirectly, each category of approach is concerned with avoiding the loss of adaptability that comes with over-convergence of search.

3.4 Diversity

Perhaps the most straightforward approach to countering the problem of population convergence is by explicitly introducing diversity into the search process. Jin and Branke [52] group diversity approaches into two categories: generating diversity after a change and maintaining diversity throughout the run. Diversity techniques may alter the mutation rate, insert randomly initialized individuals into the population, or use a sharing or crowding mechanism.

Since the greatest need for diversity occurs immediately after a change, some of the early diversity techniques focused on generating diversity after a change. Hypermutation [26] drastically increases the mutation rate for several generations immediately after a change in the environment. Though this increases diversity, it also may replace information about previously successful individuals with random information. Variable local search [95, 96, 97] gradually increases the mutation rate to attempt to reduce some of these effects. However, since it is unclear how much diversity is useful, techniques that introduce diversity all at once may too ofter err in how much diversity is introduced.

Subsequent approaches have tended to attempt to maintain diversity throughout the run. If a population always remains diverse, then convergence may be avoided at all times, and optimization may be more adaptive to changes. The random immigrants approach [43, 42] maintains the diversity of the population by inserting randomly initialized individuals (immigrants) into the population at every generation. Instead of changing possibly useful solutions in the population via mutation, these random immigrants provide diverse genetic material that can be used in crossover to increase diversity. The thermodynamical genetic algorithm [69] uses an explicit diversity measure in combination with fitness to choose the new population. Sharing and crowding mechanisms [21] are another way to encourage diversity. A more recent approach to maintaining diversity is elitism-based immigrants [108]. Rather than generating completely random immigrants, some or all immigrants are mutated versions of the best individuals from the previous generation. This is less disruptive than random immigrants, but it keeps diversity high.

3.5 Memory

In many dynamic problems, the current state of the environment is often similar to previously seen states. Using information from the past may help to make the system more adaptive to large changes in the environment and to perform better over time. One way to maintain and exploit information from the past is the use of memory, where solutions are stored periodically and can be retrieved and refined when the environment changes.

Memory-based approaches for dynamic optimization may be divided into implicit memory and explicit memory, based on how memory is stored. Implicit memory stores information from the past

as part of an individual, whereas explicit memory stores information separate from the population, typically as a set of previously good solutions. Explicit memory has been much more widely studied and has produced much better performance on dynamic problems than implicit memory.

3.5.1 Implicit memory

Implicit memory for evolutionary algorithms stores memories in the chromosomes of individuals in the population. There are several types of implicit memory, but probably the most common is the use of multiploidy evolutionary algorithms [40, 62, 73, 86, 87]. Inspired by the large number of biological organisms with recessive genes, the creation of a multiploid chromosome with some dominance mechanism allows retention of information in the recessive portion of the chromosome. Most results have used diploid chromosomes, though polyploid chromosomes are possible.

Several other forms of implicit memory have also been created. Collard et al. created the dual genetic algorithm [27], which adds a single meta bit to a bitstring chromosome. When the meta bit is turned off, the bitstring is read as normal, but when the meta bit is turned on, the complement of the bitstring is read instead. Other implicit memories that use this concept of the dual include those by Gaspar and Collard [37] and Yang [104]. Dasgupta and McGregor created the structured genetic algorithm [31], which uses a more complex structure of meta genes.

3.5.2 Explicit memory

Explicit memory for evolutionary algorithms stores memories separate from the population in a memory bank. Explicit memories have been very popular and widely used for dynamic optimization. Specific strategies for storing and retrieving information from the memory vary between techniques, but the general structure of the memory tends to be similar. In the remainder of this thesis, a memory is defined as an explicit memory bank and a memory entry is defined as a part of a memory.

In an early use of memory, Ramsey and Grefenstette [81] created a case-based memory that stored both previous good solutions and information about the environments those solutions were created in. When a change was detected, entries with closely matching environments were found and the solutions from those entries were used to reinitialize part of the population. The dynamic problem was episodic, so memory entries were stored once per period. The memory was allowed to grow without bound and was never reduced in size.

Branke [13, 14] introduced a more general model of memory which did not require storing information about the environment. Periodically, the memory tries to store the best individual in the population. The memory has a finite size; when the memory is full a replacement strategy is used decide whether the best individual in the population should replace one of the memory entries. A

variety of replacement strategies for maintaining a diverse memory are presented in [14]. Rather than selectively retrieving some of the memory entries, the whole memory is reinserted into the population either after a change or throughout the run. Branke [13] also noted that memory is very dependent on diversity and examined several extensions to the basic memory which augment the basic memory with diversity techniques. A thorough analysis of all these approaches is given in [14].

Eggermont et al. [34] presented a memory similar to the memory in [13] with a least recently used replacement strategy. This work was extended by Eggermont and Lenaerts [33] by adding a predictor to the memory. However, this predictor was very simple and highly problem dependent.

Bendtsen and Krink [5] presented a memory that moved entries in response to the location of the best individual in the population rather than replacing entries outright. This helped track optima that move slightly. This approach outperformed Branke's memory and a standard evolutionary algorithm on an example problem.

Chang et al. [22] used case-based reasoning for scheduling with a GA. Dynamic scheduling is a difficult problem for using memory, as the available jobs change. Comparisons between job attributes were used to transition between periods. This was a periodic problem rather than a continuous dynamic problem.

Kraman et al. [55] presented a memory indexing evolutionary algorithm which stores environmental information and a distribution array of the population in each memory entry. A problem dependent measure of the environment is used to index the environment. After a change, the new environment is compared to the memory entries, and the distribution array of the closest memory entry is used to reinitialize some part of the population. This approach is similar to [81], except instead of storing the best solution, an estimate of the population distribution is stored and then sampled to create new solutions.

In a similar use of population distribution estimates, Yang [106, 107, 109] presented associative memory, which stores both a solution and a distribution estimate together in a memory entry. After a change, the solutions in all memory entries are evaluated. The distribution from the memory entry with the best solution is sampled to reinitialize part of the population. Associative memory was compared to direct memory—equivalent to Branke's memory system—and a hybrid of direct and associative memory. The use of associative memory tended to be significantly better than direct memory alone, with the hybrid version tending to perform the best of all.

Richter and Yang [82, 83, 84] presented a memory that rather than directly storing solutions, stores abstractions of the solutions by maintaining a matrix dividing the search space into cells. When a solution is added to the memory, the counter in the corresponding cell of the matrix is incremented. This allows the matrix to function as an abstract model of good solutions. After a change, solutions are retrieved from the memory by sampling from the matrix and reinitializing part of the population.

3.6 Multi-population approaches

Memory attempts to create a model of the good areas of the search space, but after an entry is created, little refinement occurs. An alternative approach has been to divide the population into several subpopulations, each of which can track a peak within the search space. This allows the evolutionary algorithm to constantly refine information about several good areas of the search space, while also trying to locate new promising areas. One of the best examples of this is self-organizing scouts [15, 14], which has been rigorously compared with other diversity and memory techniques, and shown to perform extremely well. Other multi-population models include the multi-national genetic algorithm [92, 93] and the shifting balance genetic algorithm [101].

Multi-population approaches have been very successful when compared to diversity and memory techniques, especially for problems where a peak, though moving locally and changing height, always exists in the fitness landscape. In these cases, these approaches are able to actively refine the high fitness areas of the search space and keep track of any changes, making it very simple to find the optimum after a change. However, if peaks instead disappear and reappear across time, self-organizing scouts may spend a lot of search looking for the peak after it disappears, and before it reappears, the scout population may be lost when it no longer finds that area of the search space interesting. Also, multi-population approaches spend much fewer resources on a broad search, so if the optimum is outside one of the known peaks, it may take a long time to find. Finally, like most memories, multi-population approaches are limited in the number of subpopulations.

3.7 Anticipation

While diversity techniques and memory generally attempt to respond to changes in the environment after the changes occur, anticipation attempts to create solutions that are either robust to these changes or flexible enough to allow adaptation. This makes anticipation, in many ways, a complementary approach that can be used with these other techniques. Several existing approaches to anticipation are described here, and it is assumed that anticipation could be used alongside memory and diversity techniques to create more robust and flexible solutions.

In a dynamic job shop scheduling domain with an objective of minimizing mean tardiness, Branke and Mattfeld [16, 17] framed the original problem as a multi-objective dynamic problem, where the first objective was still to minimize tardiness, but a second objective was added that measured the flexibility of a solution. The flexibility objective penalized early idle times, since schedules which push idle times toward the end of the problem are more flexible if a change occurs. This approach led to more efficient schedules and better performance than using the tardiness objective alone.

Van Hentenryck and Bent [94] have done extensive work on the use of anticipation for local search in dynamic problems. In a different approach to anticipation, distributions of future events are sampled and used to evaluate solutions. These distributions are typically available, but may be learned. Several classes of problems were investigated and the techniques described in [94] provided substantial performance benefits.

Bosman [11] has presented work on predicting dynamic problems based on past events, particularly for problems with time-linkage, where actions taken now influence future events. Bosman and La Poutre [12] have presented work on anticipation for stochastic dynamic problems.

3.8 Other approaches

Other meta-heuristic algorithms have been used for dynamic problems. Ant colony optimization has been used for the dynamic traveling salesman problem [45, 44] with several methods to repair and reinitialize the pheromone matrix after a change occurs. Particle swarm optimization has been widely used for dynamic optimization [9, 8, 10, 20, 49, 51, 78]. This meta-heuristic has similar problems with convergence to those seen in evolutionary algorithms, but these problems are approached in different ways due to the nature of particle swarm optimization. Some approaches to using particle swarm optimization include reinitializing particles after a change [49] or introducing charged particles [9]. These approaches tend to be focused on maintaining diversity. Since particle swarms already have some concept of memory, this may be helpful [20], though it does not seem to be as rich as the explicit memory described above. Multi-swarm approaches have also been proposed [10, 78] as parallels of multi-population approaches.

Another meta-heuristic which has seen some recent use for dynamic problems is estimation of distribution algorithms [36, 56, 109, 110]. Estimation of distribution algorithms are an outgrowth of evolutionary algorithms where instead of operations like crossover or mutation, the algorithm functions by learning and sampling the probability distribution of the best individuals in the population at each iteration [60]. The work to date on using estimation of distribution algorithms for dynamic problems seems focused on how to respond to a change, essentially the same diversity problem that has been encountered before. Yang and Yao [109] have also considered the use of associative memory with an estimation of distribution algorithm for dynamic problems.

Chapter 4

A standard memory for optimization and learning in dynamic environments

Prior work has shown that retaining information from the past often helps dynamic optimization adapt more quickly to changing environments. Though memories developed in the literature differ widely, most memories can be considered as variants of a standard memory system. In the first part of this chapter, a standard memory system will be defined to provide an established system that has been tested on many problems, and whose strengths and weaknesses we can analyze in order to determine how memory could perform better.

Incorporating memory into algorithms for solving dynamic problems requires a balancing act. While some types of memory may be capable of drastically improving the quality of solutions, improvements may come at the expense of how quickly these solutions can be found. Memories may bias algorithms toward good solutions for commonly seen environments, but this may be at the expense of maintaining a diverse memory that performs well in all environments. Previously investigated memories have many strengths on dynamic problems, but also have many weaknesses. The remainder of this chapter will attempt to examine the strengths and weaknesses of memory.

4.1 Overview of the standard memory

Many variants of memory for optimization and learning in dynamic environments exist in the literature. While implementations differ, most can be considered as variants of an explicit memory described in [14] and [107]. In this chapter, a standard memory system for learning and optimization algorithms is defined.

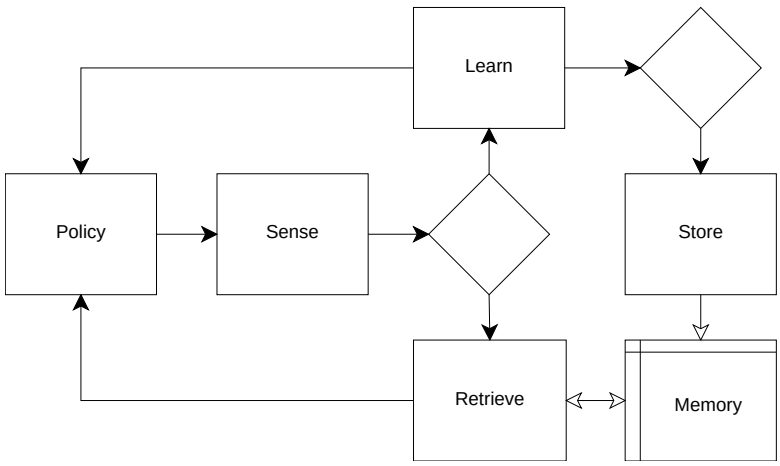

Figure 4.1: A diagram showing a basic learning algorithm with memory. The algorithm executes a policy, senses data from the environment, and then uses those data to choose either to adapt the policy or to retrieve a policy from the memory. After the learning process adapts a policy, it may be stored in the memory for future use.

The standard memory stores a finite number of solutions or policies generated by the search algorithm separately from the main optimization or learning process. These memory entries may then be retrieved from the memory and reinserted into the search process at a later time. The standard memory system can be altered to suit particular dynamic problems. For this reason, the standard memory may be seen as a building block for more complex memories and as a first step toward memories that build rich models of the dynamic fitness landscape.

Figure 4.1 shows a simplified diagram of a learning algorithm with memory. Given the current policy generated by the learning algorithm, the algorithm senses the current state of the environment. If the conditions for retrieving from the memory are met, the current policy is replaced by a policy from the memory. If conditions for retrieving from the memory are not met, the learning algorithm adapts the current policy based on the state of the environment. If conditions for storing to the memory are met, this new policy is stored in the memory. The new policy then replaces the old policy.

Figure 4.2 shows a simplified diagram of a population-based search (such as evolutionary algorithms, particle swarm optimization, or beam search) with memory. Search operations transform a parent population into a new child population. If the best individual in the child population meets some criteria, this individual is stored in the memory. Then individuals in the memory are retrieved and combined with the child population to form the parent population for the next generation.

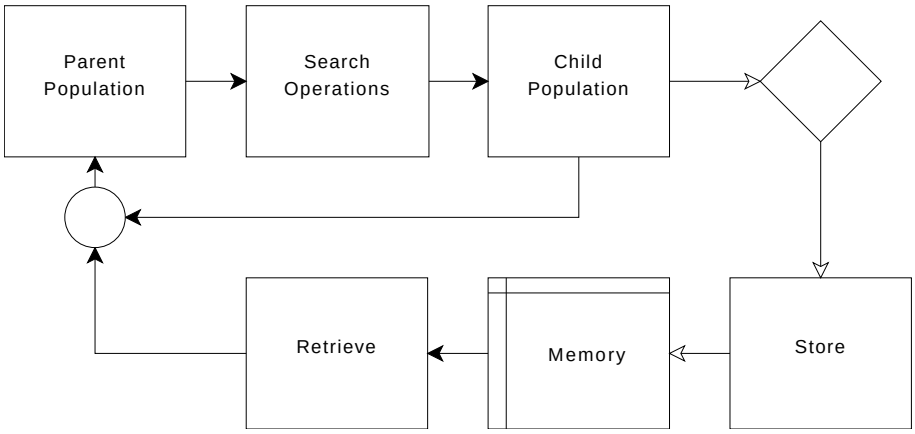

Figure 4.2: A diagram showing a population-based search algorithm with memory. The algorithm begins with a parent population of solutions. These solutions are transformed using search operations into a child population. The child population is combined with individuals retrieved from the memory to form the parent population for the next iteration of the search algorithm. Individuals from the child population may be selected to be stored in the memory.

4.2 Structure of the standard memory

A memory stores a finite number of entries containing information produced by the search process that may be used to aid search after changes in the environment. Memory functions as a sophisticated version of elitism—where good solutions are maintained in the population regardless of the results of search. The memory has a fixed size. When used with population-based search the size is generally small relative to the total size of the population. In a dynamic problem, all individuals must be evaluated at every step of the search process, including those individuals stored in the memory. By keeping the memory small, most evaluations are reserved for the main search process. The most common practice in the literature is to reserve $\frac{1}{10}$ of the allowed population size for memory. If the total population size is set as p, the memory size would be set as $m = \frac{p}{10}$. The memory is stored separately from the population.

Figure 4.3 shows the structure of the memory. Information stored in a memory entry can be divided into two categories: environmental information and control information. Environmental information helps capture the current state of the problem. As the problem changes, the environmental information may be used to determine similarities between the current environment and the environments when memory entries were stored. Environmental information is used to maintain the memory, deciding which entries should be stored in the memory when new entries become available. Typically, environmental information is used to calculate the distance between memory entries so a diverse memory may be maintained. The control information includes information that might be used by the search process. For example, the control information might just be a solution stored

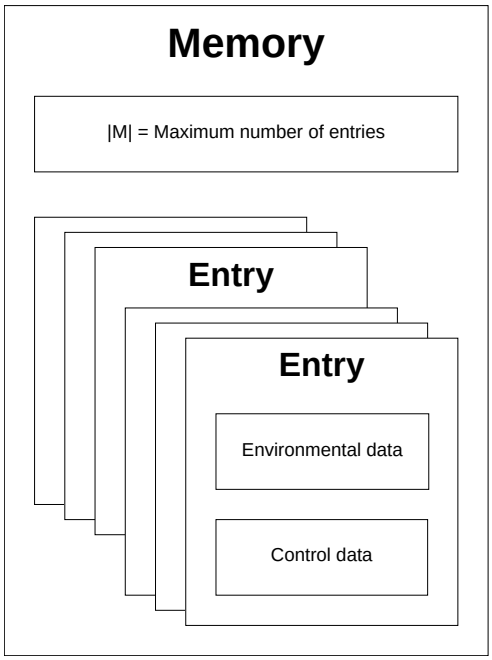

Figure 4.3: A standard memory is made up of a finite number of entries $|M|$. Each entry contains environmental and control data from the time the entry was stored.

at a previous time which can then be reinserted into the population, or it might contain a probabilistic model of the population of an evolutionary algorithm that could be used to reinitialize the population after a change.

In its simplest form, each memory entry is only an individual from the population—a solution to the problem. This solution is used for both environmental and control information in a memory entry. The location of the solution in the search space as well as the current fitness of the solution are used as environmental information to decide whether a memory entry should remain in memory. The solution itself is the control information for a memory entry, and may simply be reinserted into the population. Unless otherwise noted, this is the default structure of a memory.

In many examples from the literature, additional information is stored in an entry for improving performance. For example, in associative memory [106], an individual from the population is stored as environmental information and an estimate of the population distribution is stored as control information. When a change occurs, the fitness of the individuals stored in each entry are calculated. The individual stored in an entry—the state information—provides information about how well the entry might perform in the current environment. The entry whose individual has the highest fitness is chosen to reinitialize part of the population. The estimate of the population distribution—the control information—is then sampled to create new solutions to insert into the population.

4.3 Storing solutions in the standard memory

Storing and maintaining the entries in the memory has been one of the largest problems examined in prior work. First, it must be decided how often to update the memory. Second, what should continue to be stored in the memory as one tries to add new entries.

Storage can happen either prior to a change or periodically throughout a run; the latter is by far the most common. To avoid the possibility of always trying to store a good individual right after a change—a time when individuals tend to be quite poor—the tendency is to update the memory in a stochastic time pattern; after each update, the time until the next update would be set randomly within some range.

Given that a memory has a finite size, if one wishes to store new information in the memory, one of the existing entries must be discarded. The mechanism used to decide whether the candidate entry should be included in the memory, and if so, which of the old entries should it replace, is called the replacement strategy. A variety of replacement strategies have been proposed [13, 34, 14]. Many are designed to maintain the most diverse memory: for example, one replacement strategy might find the two closest entries out of the existing memory and the candidate memory, compare the fitness of the two entries, and discard the entry with the lower fitness. No single replacement strategy can be seen as a default, though *mindist2* and *similar* from [14] are probably most common.

4.4 Retrieving solutions from the standard memory

Memory can be retrieved in one of two ways during population-based search: after a change or throughout the run. If memory is retrieved after a change, then typically the worst m individuals in the population are replaced by copies of the m memory entries for population-based search. When memory is retrieved throughout the run, memory entries are often retrieved from the memory every generation. In this case, the working population size is $r = p - m$, where p is the total allowed population size and m is the size of the memory. At every generation, the population of r solutions produced by the last generation is combined with copies of the m solutions from the memory to form a population of p solutions available to the search operators. The search operators—in the case of an evolutionary algorithm, crossover and mutation—then produce a new population of r solutions.

For learning algorithms that are not population-based, a solution is retrieved from the memory when a problem-specific event is triggered. This may happen when a change is detected or when the current environment is very similar to one of the memory entries. Unlike population-based search, a single-point search or learning algorithm cannot retrieve from the memory at every step, as this would not allow the underlying search algorithm to make progress in refining the solution retrieved from memory.

Memory remains separate from the search process. While search or learning operators may alter copies of memory entries reinserted into the population, the stored entries in the memory are not changed. In most memory implementations, the individuals retrieved from the memory are exactly the same individuals that were stored there. However, there are several examples of memory where new solutions are generated based on the solutions stored in the memory entries—often by sampling a stored distribution of solutions—and then inserted into the population [55, 106]. For population-based search, it is also feasible to only retrieve some of the individuals from the memory at a given time rather than all memory entries.

4.5 Strengths of the standard memory

Memory has proved to be useful for dynamic optimization, giving improved performance over stock evolutionary algorithms on dynamic optimization problems. This success can be attributed to several aspects of memory. The standard memory helps search adapt to changes in the environment, maintain diversity in the population, build a simple model of where good areas exist in the search space, and limit the overhead of building and maintaining the memory.

Adapting to change Since finding a good solution quickly in a dynamic problem is often more important than finding the absolute best solution, leading search toward promising areas soon after a change may be very helpful. Since many problems are not completely stochastic, but often encounter similar states to those seen previously, memories have proved to be very useful. Since memory is typically limited to storing a finite number of good solutions which may be relatively small compared to the number of contexts, maintaining a diverse memory may allow the search process to quickly move toward the general area of the new optimum.

Maintaining diversity Population convergence can limit the adaptability of an algorithm for a dynamic problem, but memory helps inject diversity into the population whenever the memory is accessed. Since a memory is typically maintained to be as diverse as possible, even a highly converged population should be able to diversify after a change given a good memory. The use of memory does not provide as much diversity as dedicated techniques like random immigrants. However, such diversity techniques often produce useless individuals, where diversity produced by memory may more often have the potential to be useful, since the solutions were once good.

Building a model of the dynamic search space On a similar note, standard memory provides a model of where good solutions exist over time. Though this model is immediately useful for adapting to recent changes in the environment, a model of the dynamic landscape over time can also

help with anticipating future environments, developing better diversity mechanisms, or improving other areas of the search process. Given a small memory size, this model is very crude, but it does provide an idea of where solutions have been best over time.

Limiting overhead of the memory The standard memory accomplishes all this with limited overhead. Memory is an investment in search: if some amount of computation time is taken from search and given to memory, then it should provide at least as much improvement as increasing the amount of search would. For the standard memory, the main overhead is evaluating the fitness of the entries in the memory, which may be done when the memory is updated if we only retrieve after a change or at every generation if we retrieve throughout the run. Storing and retrieving from the memory also requires some amount of computation, but this is generally small compared to the time required to evaluate solutions. The use of memory also requires very little physical memory; this is not a limiting factor.

4.6 Weaknesses of the standard memory

While memory has many strengths, the standard memory model also has many weaknesses. Some of those weaknesses have been addressed in prior work, but usually only in a piecemeal way. The next section will discuss how this thesis will improve memory for optimization and learning in dynamic environments by addressing the weaknesses of the standard memory.

Limited model of the dynamic search space The standard memory builds a simple model of the good areas of the dynamic fitness problem over time. While this is true, this model is very limited. Since the memory size must be small in comparison to the population, a multi-modal landscape that reoccurs often in a dynamic problem may be modeled in the memory using only a single memory entry. Richter and Yang [83, 82, 84] have developed an abstract memory that constructs a much better model of the dynamic problem over time by creating a grid over the space solutions. Storing to the memory means incrementing the counter at the corresponding grid point. This enables the estimation of the distribution of good solutions. The system is not based on the standard memory, and so loses some of the advantages that go with it, including storing actual solutions, rather than just their abstractions. The model that is constructed in this abstract memory is also limited by the grid-based structure of the memory. A logical next step would be a memory capable of building a model of the dynamic problem over time within the structure of the standard memory.

Small memory size While the standard memory accomplishes a great deal with a very small memory size, this limits the number of areas a memory can cover. When the number of peaks in a

dynamic problem increases beyond the size of the memory, the good areas may no longer be well covered by the memory. It is also possible for the memory to become more volatile; as the number of good areas increases, an individual that is being stored is less likely to be similar to an entry already in the memory.

Difficult to refine memory entries Though memory often leads search toward promising areas, there may be times where memory actually hinders search. In many of these cases, memory leads search to several suboptimal areas, taking time away from the search that ends up leading to the best solutions. This may be due to several factors, but one cause may be that standard memory typically cannot refine memory entries after storing them: the only way to change a memory entry is by replacing it. Multi-population techniques like self-organizing scouts [15, 14] help to counter this problem, though these approaches typically draw resources away from search in other ways.

Limited diversity Though memory does help inject diversity into the population after a change, this diversity is limited to those areas that have been searched in the past. Thus, memory relies heavily on the underlying optimization algorithm to find diverse solutions, something that is not always possible. For this reason, memory must often be accompanied by diversity techniques to be useful [14]. However, diversity techniques are typically not designed specifically to work well alongside memory. Diversity techniques can be disruptive to search, and memory can often provide a great deal of diversity. In an ideal situation, when the memory is not sufficiently diverse, a diversity technique should inject a great deal of new genetic material into the population. However, once the memory has become more diverse, the diversity technique should respond by decreasing its role.

Limited applicability Finally, at present, memory is limited to certain types of problems. In problems where the feasible region of the search space never changes—where peaks move within the search space, but may return to exactly the same location—memory has been widely tested. However, problems like dynamic scheduling, where the tasks to be scheduled change over time, the feasible region of the search space also changes with time. Other problems like this include dynamic routing and some dynamic knapsack problems where the number of items available to place in the knapsacks change over time. Memories might be helpful for these types of problems, but little work has been done on extensions that would allow memory to be used.

4.7 Improving memory

The goal of this thesis is to improve memory for optimization and learning in dynamic environments. The remainder of this thesis will describe improvements to the standard memory that address many of the weaknesses described above while maintaining the strengths of this memory system.

Part II introduces a new class of memories called density-estimate memory. Density-estimate memory builds probabilistic models of past solutions within the memory in order to build a much richer model of the dynamic search space over time. Though many more solutions may be stored in a density-estimate memory than in the standard memory, the overhead of using the memory remains low because the search process can interact with the probabilistic models, rather than with every solution stored in the memory. Since many more solutions can be stored in the memory, refining memory entries becomes much easier. Also, with the richer models of the search space created by a density-estimate memory, diversity techniques can be more sophisticated.

Part III introduces another new class of memories called classifier-based memory. Classifier-based memory extends memory to problems where the feasible region of the search space changes over time, such as dynamic rescheduling. Instead of storing an exact solution to a dynamic problem, classifier-based memory stores a abstraction of a solution which can be mapped to a feasible solution at any time. This allows memory to be applied to many new problems without compromising the strengths of the standard memory.

4.8 Summary

This chapter defined a standard memory similar to many memory systems that have been used to improve the performance of optimization processes like evolutionary algorithms on problems with dynamic environments. This standard memory is composed of a finite number of memory entries that provide a simple model of the search space over time. As the search process discovers good solutions, the memory considers whether a new solution will improve the quality of solutions currently stored in the memory and if so, the new solution replaces one of the current memory entries. Solutions from the memory may be retrieved in a variety of ways, though population-based search like evolutionary algorithms typically retrieve all solutions from the memory at every iteration of the search process.

Memory has been shown to help lead optimization to promising areas of the search space in dynamic environments. Memory also helps to inject diversity into the search process, helping optimization avoid over-convergence. The standard memory does all of this with little overhead required to build or maintain the memory.

The standard memory has a number of weaknesses. While the standard memory builds a model of the search space over time, this model is usually very simple since the memory typically has a small number of entries. This standard memory cannot refine memory entries without completely replacing them, so outdated entries may lead search astray. While memory helps to inject diversity, that diversity only comes from areas the algorithm has searched before, so memory relies heavily on the underlying algorithm to search widely. The standard memory is also not applicable to all dynamic problems. In particular, the standard memory is not useful for problems where the feasible region of the search space changes over time.

Part II

Building probabilistic models in memory

Chapter 5

Density-estimate memory

This chapter introduces density-estimate memory, a new class of memory for optimization and learning in dynamic environments. Density-estimate memory is designed to address many of the weaknesses of standard memory while keeping the overhead associated with memory low. Density-estimate memory helps to guide search algorithms on dynamic problems by using probabilistic models of previous good solutions. Instead of storing single solutions separately in memory, density-estimate memory uses these probabilistic models to create density estimations of the search space over time. These models can reveal more about the landscape of the search space than the individual points stored in a standard memory. Even though density-estimate memory aggregates information from many more solutions than standard memory, the overhead of using memory remains low. Instead of having to interact with every solution stored in the memory, density-estimate memory allows a search algorithm to interact only with the models stored in the memory. Density-estimate memory also allows memory entries to be more easily refined than the memory entries in standard memory. The richer, long-term model of the dynamic search space also allows diversity techniques to be more informed and sophisticated.

5.1 Improving memory by building probabilistic models

The standard memory described in Chapter 4 builds a very simple model of the location of good solutions over time by storing individual previous solutions. These solutions may be retrieved by an optimization or learning algorithm to lead search back toward those areas that have help good solutions in the past. For simple search spaces, this approach can be quite successful at improving the ability of an evolutionary algorithm to find better solutions. As the fitness landscape becomes more complex, the model of the search space built by standard memory becomes weaker. When only a small number of memory entries are allowed, entire areas containing good solutions may not

be represented in the memory. Even increasing the number of memory entries may have little effect, since the standard memory does not aggregate information from multiple solutions.

Instead of storing individual solutions, density-estimate memory aggregates information from many solutions by building and storing probabilistic models. By storing many more of the good solutions discovered by the search process, density-estimate memory creates a long-term model of good solutions in the search space. Density-estimate memories are inspired in part by estimation of distribution algorithms [60]. Estimation of distribution algorithms search by learning and sampling the probability distribution of the best individuals in the population at each iteration. Repeating the learning and sampling process allows estimation of distribution algorithms to refine models of good solutions. Unlike standard memory, which can only change a memory entry by completely replacing it, density-estimate memory allows the model of a peak in the fitness landscape to be refined. The addition of new points to a memory refines the probabilistic model and creates a feedback loop between the memory and the underlying learning or optimization algorithm. As the quality of solutions retrieved from the memory improves, the underlying algorithm can spend more time in finding the best solutions rather than finding areas containing good solutions. This leads to better solutions being stored in the memory. Since density-estimate memory can store many more of the good solutions produced by the underlying search algorithm, it is less likely to discard solutions from rare environments which reoccur less frequently in the problem. Density-estimate memory is less dependent on the new optimum having recently been in a high fitness area, so it is able to guide search more effectively for problems where changes in the environment are very discontinuous.

The structure of a density-estimate memory is similar to standard memory. The memory is divided into a fixed number of memory entries. In density-estimate memory, each of these entries may contain many points; each point is a solution stored in the memory at some time. Periodically, a solution produced by the underlying learning or optimization algorithm is stored in the memory. When a new point is stored, models in the memory are updated. Solutions may be retrieved from the memory and used by the underlying algorithm at any time. Since density-estimate memory maintains the same interface as standard memory—a relatively small number of memory entries—the overhead of the memory remains low. The underlying algorithm only needs to interact with the models stored in a density-estimate memory, not all of the points used to create those models.

Density-estimate memory is a class of related memories. Many different density-estimation techniques may be used to build the models used by a density-estimate memory. Though many memories from the literature have been designed specifically for use with evolutionary algorithms, density-estimate memory may be easily adapted for a variety of underlying optimization and learning algorithms on many different problems. This chapter will describe a general implementation of a density-estimate memory, but some of the experiments described in later chapters will test variants of this implementation.

5.2 Other methods for improving memory

Some methods from the literature discussed in Chapter 3 have addressed some of the same weaknesses in the standard memory as density-estimate memory. While many methods have improved performance over the standard memory, most introduce limitations on the applicability of the standard memory. This section discusses two of these techniques that build richer models of the dynamic search space and allow the memory to be more easily refined.

Self-organizing scouts [14] creates a constantly adapting memory by devoting most of an evolutionary algorithm population to scout populations which search limited areas of the search space. This allows solutions in good areas of the search space to be quickly refined as the dynamic environment changes. By dividing the search space, exploration can ignore areas covered by the scout populations. Self-organizing scouts is particularly useful for problems with continuous changes where several areas of the search space contain good solutions and the global optimum is almost always in these areas.

Self-organizing scouts requires a population-based search such as an evolutionary algorithm, so it is not suitable for many types of dynamic problems. Self-organizing scouts also limits the amount of exploratory search, since many of the individuals in the population are dedicated to refining good solutions in the scout populations. For discontinuous problems with many areas of good solutions, scout populations may not contain sufficiently good solutions to lead to the global optimum.

Richter and Yang [83, 82, 84] developed an abstract memory that aggregates the locations of good solutions over time. The search space is divided discretely and counts are kept to measure how often good solutions occur in each segment. This abstract memory is not limited by any particular model for the distribution of good solutions in the search space. This memory can also better represent complex search spaces than the standard memory.

This abstract memory does require choices about the grid used within the memory. The choice of how to divide the search space is very important; large segments may not accurately capture the structure of the space, while small segments may not aggregate solutions enough to be useful. Also, this abstract memory may be difficult to adapt to solutions representations that are not real-valued.

While density-estimate memory will not outperform these methods for every problem, it is expected to do better for many problems, particularly those with very discontinuous changes and complex search spaces. Density-estimate memory also has many fewer limitations. Density-estimate memory may be used with a wide variety of optimization and learning algorithms, can use a wide variety of probabilistic models, can adjust its models based on the structure of good solutions in the search space, and can be adapted to problems where the representation is not real-valued.

5.3 Structure of a density-estimate memory

A density-estimate memory stores points in a finite number of memory entries. Each point is composed of environmental information and control information. The control information captures a solution to the problem while the environmental information captures the state of the dynamic environment at the time the point was stored. In the standard memory, each memory entry could store a single point; in density-estimate memory, each entry may store many points.

A memory may have up to $|M|$ entries. Each entry contains a collection of points, a model of the environmental information in those points, and a model of the control information in those points. Figure 5.1 shows the structure of a density-estimate memory. The models in a density-estimate memory may be as simple as the average of all points or may use complex probabilistic models. In this thesis, the most common models used to represent an entry are a multi-variate Gaussian model and a simple clustering model. In the Gaussian version, the mean and sample covariance of all the points in the memory entry are used to describe the model. When entries are first created, there may not be enough points to calculate a valid covariance. In this case, random points around the mean are temporarily added to fill out the model. In the simple clustering version, only the mean is used. In Chapter 8, a density-estimate memory using Gaussian mixture models will also be considered.

Though a memory entry contains many points, only the environmental model and control model are necessary when interacting with the entry. Since entries might contain thousands of points, this keeps the overhead of a density-estimate memory low while still allowing these many entries to contribute the density-estimation captured in the models. The only time the individual points are used is when the models must be recomputed after a new point is added to the entry.

Figure 5.2 shows an example of how density-estimate memory differs from standard memory. In this example, the one-dimensional search space has four peaks of varying shape and fitness. Suppose that the underlying search algorithm has tried to store the points marked with a solid line to the memory over time. The standard memory will choose the point with the best fitness; for these simple peaks, this point will always be the closest point to the true center that the memory has seen. The density-estimate memory will cluster these points by peak and then build a model for each cluster. A Gaussian model for each peak is shown at the top of the figure. For this example, the mean calculated by each cluster has higher fitness and is closer to the true peak maximum than the corresponding memory entry in the standard memory. Rather than using only Euclidean distance, the Gaussian models provide more information when adding a new point to the memory. The peak at the far left is quite wide, and much more likely to accept points far from the mean than the peak at the far right, which is quite narrow. For the clusters that do not yet have an accurate model of the true peak, the model may be refined by adding more points to the memory. As the models become more accurate, the solutions retrieved from the memory will be better. As long as the underlying

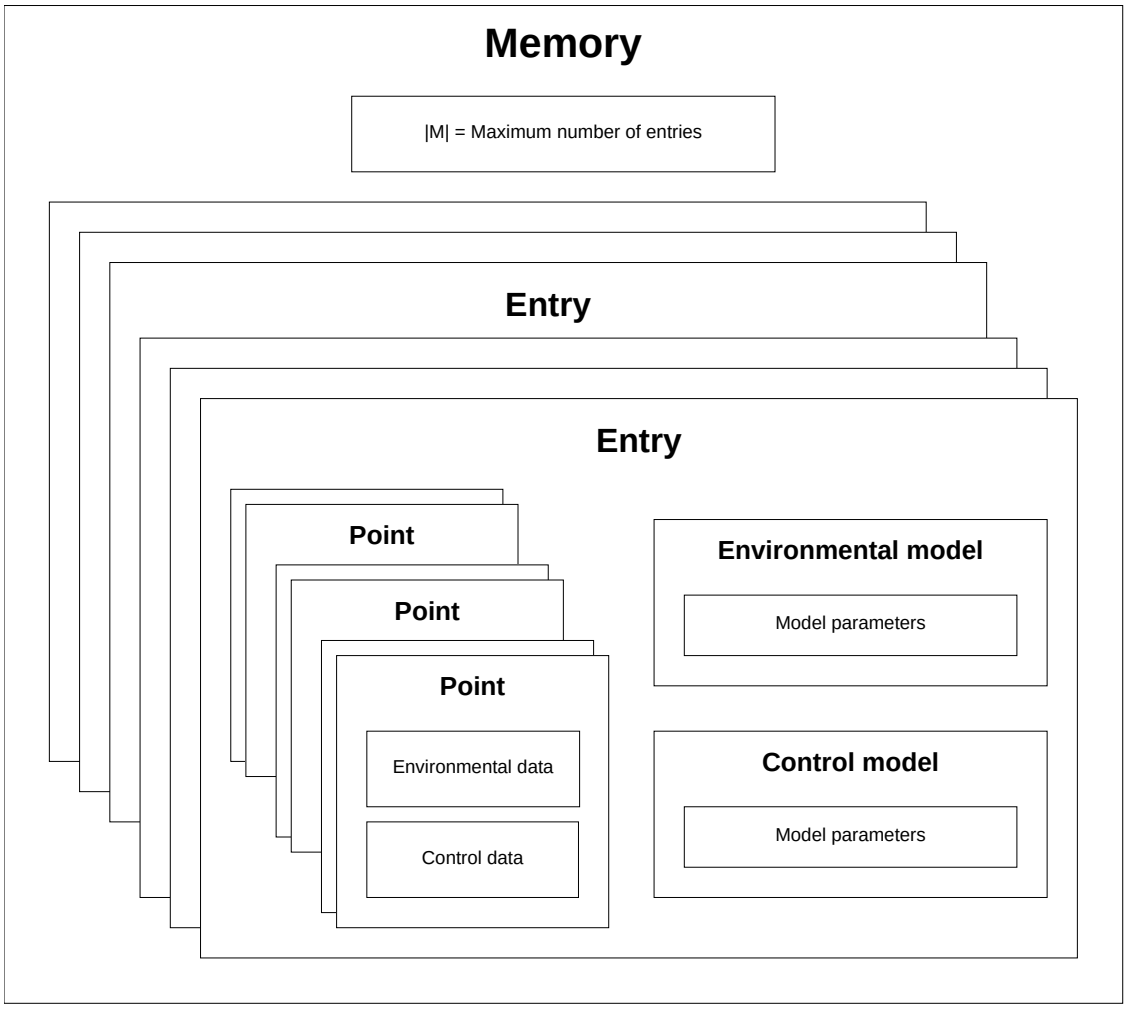

Figure 5.1: A density-estimate memory is made up of a finite number of entries $|M|$. Each entry contains a collection of points where each point contains environmental and control data from when the point was stored. These points are used to construct an environmental model and a control model for the memory entry.

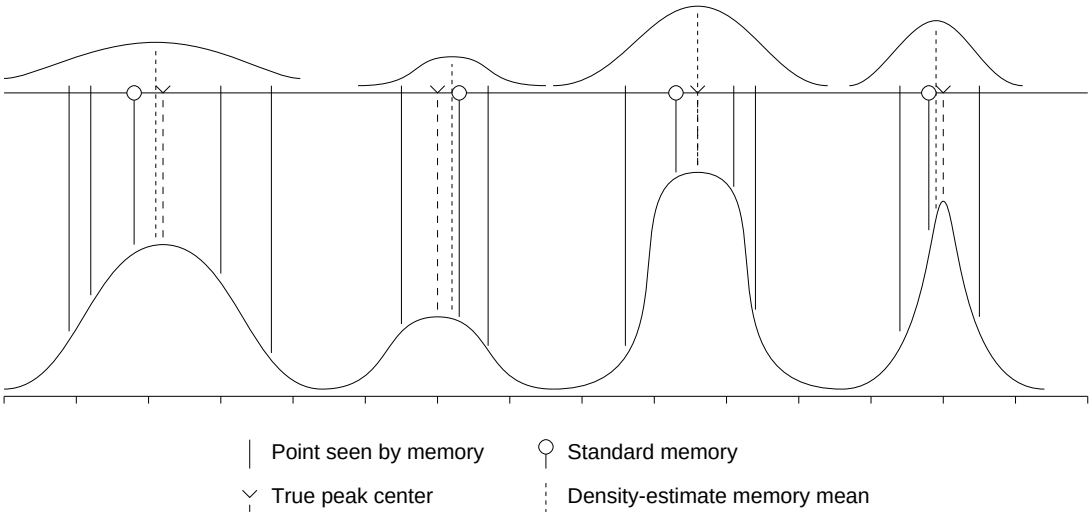

Figure 5.2: Density-estimate memory example of a search space with four different peaks. Both standard and density-estimate memory have seen the same points; the memory entries for the standard memory are shown as dots, while the Gaussian models calculated for each cluster in the density-estimate memory are shown above the search space.

search algorithm can continue to improve on the solutions retrieved from the memory, this feedback loop will continue to improve the models stored in the memory.

5.4 Storing solutions in a density-estimate memory

In a standard memory, a point is stored in the memory using a replacement strategy that decides whether the new point should replace one of the existing memory entries. Most replacement strategies are designed to maintain the most diverse memory possible. For example, a replacement strategy typically adds the new point to the memory, finds the two memory entries closest to one another, and discards the one with the lower fitness.

Rather than discarding entries, a density-estimate memory merges memory entries together. First, the new point being added to the memory is used to create a new memory entry. After the models are built for this new entry, the two most similar entries in the whole memory are found and merged together. Similarity is measured using the environmental models of each entry. After clusters are merged, the models for the new cluster are rebuilt using the combined set of points in the new memory entry. In most cases, the new point will end up being merged into an existing memory entry. When a new area of the search space is included for the first time, the new point will then be used to begin modeling this area within the memory.

By incrementally clustering points, new areas of the search space can be modeled as the dynamic

environment changes. As clusters merge together, clusters contain more points. As the density of those points increases, an appropriate model will be able to provide increasingly good estimates of the location of good solutions in the area.

It should be noted that density-estimate memory, like standard memory, depends heavily on the underlying learning or optimization algorithm. Density-estimate memory builds models based on the points stored in the memory, but if the underlying algorithm stores poorly performing solutions, then the models within the memory will not represent good solutions. If good solutions are found by the underlying algorithm, density-estimate memory will help the underlying algorithm to explore the promising area, but the algorithm must be able to perform the search in the area.

When to store new points in the memory depends largely on the underlying learning or optimization algorithm and the type of problem. When used with an evolutionary algorithm, a new point may be stored every few generations. When used with a reinforcement learning algorithm, it may be best to store new points once good solutions have been learned. For some problems where changes can be easily detected, it may be best to store solutions right before a change. For other problems where changes are more gradual, it may be best to store periodically rather than storing when certain conditions occur. It is rarely desirable to store a solution right after a change since solutions have not had time to adapt and tend to be quite poor. When updating periodically, storing in a stochastic time pattern can help avoid repeatedly storing right after a change. After a new point is stored in the memory, the time of the next update may be set randomly within some range.

The computational overhead of using a density-estimate memory depends on the problem as well as the probabilistic models used by the memory. When incrementally clustering points, choosing the two entries to merge can be done in $O\left(m^2\right)$, where m is the number of memory entries. This computation time is similar to that required for a replacement strategy with the standard memory. When using Gaussian models within a density-estimate memory, the sample covariance may be computed in $O\left(nd^2\right)$, where n is the number of points in the memory entry, and d is the number of dimensions in the environmental or control data. Since the memory is divided into clusters, n is typically much smaller than the total number of points in the memory. Though the use of memory does have some computational overhead, the time necessary to evaluate solutions typically dominates the time needed to maintain the memory. Since density-estimate memory does not require additional solution evaluations, the computational overhead of a density-estimate memory comes only from the construction and maintenance of the probabilistic models in memory.

For some problems where either the environmental or control data have a large number of dimensions, it may take a long time to build up the memory before retrieving from memory can be effective at improving performance. The larger the dimensionality of the data, the more points are required in order to give a good density estimation. For these types of problems, it is advantageous to pre-build a memory rather than starting with an empty one. Results from other domains have shown that

biasing a solution in this way is helpful, even if the training data are only similar to the testing data [2, 39].

5.5 Retrieving solutions from a density-estimate memory

Solutions are retrieved from a density-estimate memory in much the same way as solutions are retrieved from a standard memory. For population-based search algorithms, solutions may be retrieved from the memory either throughout a run or just after a change is detected. Most commonly, solutions are retrieved from the memory at every generation of an evolutionary algorithm. Though the memory entries only change when new points are stored to the memory, the solutions produced by retrieving from the memory can then be used at each generation by the underlying search algorithm. For learning algorithms that search from a single point, solutions must be retrieved from the memory less frequently and more selectively. Since retrieving from the memory at every step would make it impossible for the underlying algorithm to refine the solution retrieved from the memory, solutions are retrieved either at a problem-specific trigger or periodically. In this thesis, a solution is typically retrieved from the memory when the current solution is poor and the current environment closely matches one of the memory entries.

Unlike standard memory, the solutions retrieved from the memory may not be the same as any of the individuals stored in the memory. By aggregating many points into a single memory entry, a solution retrieved from memory is influenced by many previous solutions that performed well in similar environments in the past. For problems with a real-valued encoding, the easiest way to retrieve a solution from a density-estimate memory entry is to average the control information for all the points in the entry. For problems with other encodings that do not allow this—such as binary strings or feature vectors—solutions could be generated by sampling from a model of the control information in the memory entry, by finding the most commonly occurring control information among the points in the entry, or by other methods that use the models generated by the density-estimate memory.

Even if it is possible to simply average the points in the entry to find a solution to retrieve, density-estimate memory could be used in other ways to generate new solutions based on the models stored in memory. There are several examples from the literature where memory stores not just solutions but a distribution that can be used to regenerate the population after a change [55, 106]. Density-estimate memory could be used in a similar way in cases where changes could easily be detected. After a change, the probabilistic model of control information in a memory entry could be sampled to generate many solutions to be introduced into the search population.

5.6 Summary

This chapter introduced a new type of memory for learning and optimization in dynamic environments called density-estimate memory. Density-estimate memories are an extension of the standard memory that use probabilistic models to aggregate information from many solutions produced by a search algorithm. Rather than just storing and comparing single points, density-estimate memories build a rich model of the search space over time. As the search process discovers good solutions, the models stored in the memory are constantly refined. As the model develops, the solutions that can be retrieved from the memory become more useful to the search process. Density-estimate memory is especially well suited to discontinuous problems with large changes in the location of optimal solutions.

Though many more points are stored in the memory, the overhead associated with building, maintaining, and interacting with the memory remains low. Density-estimate memories cluster points into a finite number of memory entries. Each entry maintains separate models that only need to be rebuilt when new points are added to the entry. An incremental clustering process allows the memory to adapt as promising areas of the search space are discovered by the search process.

Density-estimate memories may use a variety of probabilistic models within the memory. Simple clustering models may be sufficient for some problems, while more complex mixture models might be better for others. Since each type of probabilistic model has a different amount of overhead required to build and maintain the models in memory, density-estimate memories offer a great deal of flexibility in choosing model types depending on the requirements of the dynamic problem.

Density-estimate memory stores good solutions produced by optimization and learning algorithms and allows these solutions to be reinserted into the search process, improving the ability of search to quickly adapt to changing environments. Density-estimate memory addresses many of the weaknesses of standard memory by allowing many more good solutions to be stored, building long-term models of the dynamic search space, and constantly refining and improving memory entries as new points are added to the memory, all while keeping the overhead of memory low.

5.7 Outline

The remainder of Part II considers the application of density-estimate memory to three dynamic problems: factory coordination, dynamic optimization with evolutionary algorithms, and adaptive traffic control. Though all three have dynamic environments, these problems are very different from one another. For all three of these problems, density-estimate memory improved performance over a baseline learning or optimization algorithm. Density-estimate memory also outperformed state-of-the-art algorithms for each problem.

Chapter 6 examines the problem of dynamic, distributed factory coordination. In the factory coordination problem, incoming jobs must be distributed among several machines to maintain the flow of jobs through a simulated factory. Machines may be configured to perform any job, but changing the configuration of a machine requires a setup time. Density-estimate memory allows a reinforcement learning algorithm to adapt more quickly to changing jobs distributions.

Chapter 7 considers dynamic optimization with evolutionary algorithms on the Moving Peaks benchmark problem. The Moving Peaks problem is a common benchmark from the literature that allows the nature of the search space to be highly configured. Density-estimate memories are compared to a wide variety of approaches to improving a standard evolutionary algorithm on dynamic problems. On a highly discontinuous variant of the Moving Peaks problem, density-estimate memory outperforms the state-of-the-art self-organizing scouts technique.

Chapter 8 applies density-estimate memory to an adaptive algorithm for controlling traffic signals in a urban road network. Traffic signal control is a highly dynamic problem. Traffic demands at an intersection constantly change due to the time of day, the types of vehicles, and the control of surrounding traffic signals. Experiments consider two road networks based on intersections in downtown Pittsburgh, Pennsylvania. The second experiment uses thirty-two intersections and realistic traffic flows based on traffic counts for different times of day. Density-estimate memory improves the performance of an adaptive traffic signal controller enough to be competitive with the actual fixed timing plan used on these intersections.

Chapter 6

Factory coordination

Many real-world problems involve the coordination of multiple agents in dynamic environments. Machines in a factory may need to coordinate the scheduling and execution of jobs to ensure smooth operation as customer demands shift. Teams of robots may need to coordinate exploration and task allocation in order to operate in new and changing environments. Web-based agents may need to coordinate services like information gathering as types of input or demand change. One may approach dynamic problems in many ways, depending on the nature of the problem. Change may be dealt with through completely resolving the problem from scratch each time a change occurs, using centralized optimization to maintain a good solution over time, learning a fixed distributed model that can adapt to changes, creating a model that can learn and adapt as changes occur, or a combination of approaches.

In the domain of factory operations, adaptive, self-organizing agent-based approaches have been shown to provide very robust solutions. A factory is a complex dynamic environment with constant changes in product demand and resource availability. These types of changes often conflict with attempts to build schedules in advance. By using adaptive approaches, a scheduler can be sensitive to unexpected events and can avoid invalid schedules. However, these adaptive approaches may require non-trivial amounts of time to respond to large environmental shifts.

Techniques exist that have been shown to help many different approaches perform better when problems are dynamic. One common technique is the use of information from the past to improve current performance. In many dynamic problems, the current state of the environment is often similar to previously seen states. Using information from the past may help to make the system more adaptive to large changes in the environment and to perform better over time. One way to maintain and exploit information from the past is the use of memory, where solutions are stored periodically and can be retrieved and refined when the environment changes. For more dynamic problems, one well studied approach is explicit memory—directly storing a finite number of previous solutions to

be retrieved later.

As the number of possible environmental states increases in a dynamic problem, a memory of fixed size has a more difficult time modeling the dynamic landscape of solutions. While the memory size can be increased, the overhead associated with maintaining and using the memory limits how large the memory can be. This chapter introduces several density-estimate memory systems inspired by estimation of distribution algorithms that improve upon standard memory systems while avoiding large increases in overhead.

The performance of these density-estimate memories is evaluated on a dynamic, distributed factory coordination problem [25]. This problem requires the dynamic assignment of jobs to machines in a simulated factory. Products of several different types arrive over time and must be allocated to a machine for processing. When a machine switches from processing one type of product to another, a setup time is incurred.

This chapter will explain the distributed factory coordination problem and several agent-based approaches to the problem. It will then describe a new dynamic variant of this problem, a baseline agent-based approach to the problem, and the weaknesses of the baseline approach. It will introduce the use of memory to improve the performance of on the new dynamic distributed factory coordination problem and present several novel density-estimate memory systems. Finally, the performance of the baseline system will be compared with the memory-augmented systems.

6.1 Background

Manufacturing processes provide many interesting examples of dynamic problems. For example, the factory coordination problem, also known as the dynamic task allocation problem, involves assigning jobs to machines for processing. Jobs are released over time and the scheduler has little prior information about the jobs. Given the lack of a priori information, there has been success in designing adaptive scheduling systems for these types of problems instead of using a centralized scheduler for computing optimal schedules.

Morley presented an example of the factory coordination problem from a General Motors plant: allocating trucks to paint booths [71, 70]. Trucks arrive off the assembly line to be painted, and then wait to be assigned to a paint booth's queue. The color of each truck is determined probabilistically based on a distribution of colors. Booths each have a queue of trucks waiting to be painted. All booths can paint a truck any of the available colors, but when a booth switches between colors it must flush out the previous paint color, causing a setup time delay as well as incurring the cost of the wasted paint. Booths that specialize in painting a single color, at least for a few trucks in a row, incur fewer setups.

Morley demonstrated that a market-based approach, where the paint booths bid against each other for trucks coming off the assembly line, could outperform the centralized scheduler previously used by the real paint shop [71]. This system saved almost a million dollars in the first nine months of use [70]. In this approach, a paint booth bids on a truck based on the current length of the booth's queue and the whether a setup delay would be required to process this truck.

Campos et al. [19] and Cicirello and Smith [25] independently developed distributed, agent-based approaches for the factory coordination problem inspired by the self-organized task allocation of social insects like ants and wasps. Like Morley's approach, booths still bid against one another for trucks, but instead of a fixed policy, agents representing each booth use reinforcement learning to develop policies. Agents use the concept of response thresholds to determine a bid for each truck. Though similar in inspiration, there are several major differences in these approaches. Nouyan et al. [75] and others examine these and similar approaches.

Cicirello and Smith [25] compared their system, R-Wasps, to Morley's system and Campos et al.'s system on six problems: the original problem, a version with more significant setup times, versions with two different probabilities of machine breakdown, a version with an alternate truck color distribution, and a version where the truck color distribution changes in the middle of a scenario. R-Wasps was shown to be superior to the other approaches, particularly in minimizing the number of setups required.

Cicirello and Smith [25] also presented a variant of the paint shop problem to allow for better analysis of algorithm behavior. In this problem, jobs of N types are processed by M multi-purpose machines operating in parallel. Jobs arrive probabilistically over time based on a distribution of job types, and each job has a length of $15 + N(0,1)$ time units. Each machine is allowed an infinite queue. Setups for a machine to switch between jobs require 30 time units. Cicirello and Smith examined problems with 2 job types and both 2 and 4 machines. They examined scenarios with both a single job type distribution and a switch between two job type distributions. One of the findings was that this approach may be slow to adapt to changes in the job type distribution.

6.2 A dynamic, distributed factory coordination problem

In this chapter, we examine a dynamic extension of the distributed factory coordination problem similar to the variant from Cicirello and Smith [25] described above. In previous versions of this problem, algorithms were evaluated over relatively short time horizons with very few changes in the distribution of jobs. In the dynamic factory coordination problem described here, performance is evaluated over a much longer time horizon and the underlying distribution of the job types changes many times [4].

In this problem, factories produce N products (N job types) which are processed by M parallel multi-purpose machines which can process any job type. The length of a machine's job queue is unlimited. The setup time to reconfigure a machine for a different job type is 30 time units. The process time of each job is $15 + N(0,1)$ time units. Process times greater than 15 time units are rounded up to the nearest integer, while process times less than 15 are rounded down. The process time is also bounded in the interval $[10,20]$.

Jobs are released to the factory floor according to a distribution of job types; this distribution changes over time. For example, given two job types with a 60/40 mix and a 10% chance of any new job arriving at each time unit, the distribution would be $D = \begin{bmatrix} 0.06 & 0.04 \end{bmatrix}$. Only one job may arrive at each time unit. From [25], we chose a mean arrival rate per machine of 0.05 to represent a medium to heavily loaded factory. For convenience, we define the term ℓ as the loading on the system, where $\ell = 1.00$ indicates the normal amount of load, and larger values may place the system into an overloaded state. The mean arrival time for a given scenario is $\lambda = 0.05M\ell$. A scenario lasts for 150000 time units and is divided into 50 periods of 3000 time units; at the beginning of each period the job type distribution changes.

6.3 R-Wasps agent-based learning algorithm

We use R-Wasps as described in [25] as a baseline approach on this problem. In R-Wasps, each machine is associated with a routing wasp agent in charge of bidding on jobs for possible assignment to the machine's queue. Each agent has a set of response thresholds:

$$\Theta_w = \{\theta_{w,0}, \ldots, \theta_{w,N-1}\} \tag{6.1}$$

where $\theta_{w,j}$ is the threshold of wasp w to jobs of type j. Unassigned jobs broadcast a stimulus S_j proportional to the length of time the job has waited for assignment that indicates the job type. An agent will bid on a job emitting stimulus S_j with probability

$$P(bid|\theta_{w,j}, S_j) = \frac{S_j^2}{S_j^2 + \theta_{w,j}^2} \tag{6.2}$$

Otherwise, the agent will not bid on the job. The lower the threshold value for a particular job type, the more likely an agent is to bid for a job of that type. Threshold values may vary in the interval $[\theta_{min}, \theta_{max}]$. Each routing wasp agent is completely aware of the state of the machine, but not the states of other machines in the factory. The knowledge of machine state is used to adjust the thresholds at each time step according to several rules. If the machine is processing or setting up to

process a job of type j, then

$$\theta_{w,j} = \theta_{w,j} - \delta_1 \tag{6.3}$$

If the machine is processing or setting up to process a job other than type j, then

$$\theta_{w,j} = \theta_{w,j} + \delta_2 \tag{6.4}$$

If the machine has been idle for t time units and has an empty queue, then for all job types j

$$\theta_{w,j} = \theta_{w,j} - \delta_3^t$$

The values of the system parameters used here are $\theta_{min} = 1$, $\theta_{max} = 1000$, $\delta_1 = 2$, $\delta_2 = 1$, and $\delta_3 = 1.001$.

When more than one agent bids on a job, a dominance contest is held. Define the force F_w of an agent as

$$F_w = 1.0 + T_p + T_s \tag{6.5}$$

where T_p and T_s are the sum of the process times and setup times respectively of all jobs in the machine's queue. Let F_1 and F_2 be the forces of agents 1 and 2. Then, agent 1 will win the dominance contest with probability

$$P(Agent\,1\,wins|F_1,F_2) = \frac{F_2^2}{F_1^2 + F_2^2} \tag{6.6}$$

If more than two agents bid on a job, a single elimination tournament of dominance contests is used to determine the winning bid. Seeding is done by force variable, and when the number of bidders is not a power of 2, the top $2^{\lceil \log_2 C \rceil} - C$ seeds receive a first round bye. Further explanation of the R-Wasps algorithm may be found in [25].

6.4 Weaknesses of the R-Wasps algorithm

R-Wasps performs well on the distributed factory coordination problem, but since it takes time to learn the thresholds, it may be slow to adapt to changes in the job type distribution. If the underlying job type distribution remains the same for a long period of time, the system will generally correct any problems that this slow adaptation creates. However, in the short term, this may have a large negative impact on performance.

As an example, take a problem with four machines, two job types, and three time periods, each 4000 time units long. In the first time period, 85% of jobs arriving are of type 1 and 15% are of type 2. At the beginning, each machine adapts its thresholds to accept jobs efficiently. In the second period,

when the underlying distribution of arriving jobs changes to 15% of type 1 and 85% of type 2, each machine adapts to the new distribution. In the third period, the distribution returns to 85% of type 1 and 15% of type 2. As Figure 6.1 shows, this may take longer from the change in distributions than the initial adaptation took from the beginning of the scenario. This behavior occurs for several reasons. First, the visible distribution changes more slowly than the underlying distribution, since jobs produced by the distribution in the first time period are still unprocessed at the beginning of the second time period. Second, distribution changes may lead to queue explosions. This can be seen in Figure 6.2, where the queue for machine 2 grows very large after the first distribution change and the queue for machine 3 grows large after the second distribution change. This often occurs when a machine has specialized in processing one job type while few others have. If this job type becomes common, the machine will win bids on those jobs until the other machines have had a chance to exhaust their queues and become idle. At that point, other machines will specialize and the machine with the queue explosion will stop accepting jobs and finish off the jobs already in the queue. This may lead to idleness in the system, but the real problem is that cycle times may become large and the system will thus be less adaptive. One of the major reasons for this is that when the job type distribution changes, all machines have jobs in their queue. In the example shown, three machines specialize on job type 1 during the first interval. When the distribution changes, job type 2 becomes much more prevalent, and machine 2, which had previously specialized on this type, has the advantage when bidding on jobs of this type, because the threshold for bidding is low compared to the other machines. By the time the other machines have exhausted their queues, machine 2's queue has exploded, leading to high cycle times for jobs it has queued. The same thing happens to machine 3 during the third period. This behavior stems from some of the mechanisms of R-Wasps that make it so successful during the majority of the time when the distribution is not changing. Improving performance over the baseline version of R-Wasps will probably involve reducing this adaptation time associated with these changes in the underlying distribution of job types. While it might be possible to change the mechanisms of R-Wasps directly, an approach that augments R-Wasps instead of changing it would keep performance the same for the majority of the run.

6.5 Memory-enhanced R-Wasps

As noted above, systems like R-Wasps may have trouble adapting quickly when the underlying distribution of job types changes. Finding a way to improve factory performance around these major changes could potentially improve throughput, reduce the number of setups, and in the end make the system more responsive. Fortunately, the possible states of the underlying distribution are not completely random. In cases where a new distribution of job types resembles a previous distribution, a repository of past states could be leveraged to provide a shortcut for learning thresholds for the new distribution. This section proposes adding a memory to R-Wasps for use in the dynamic factory

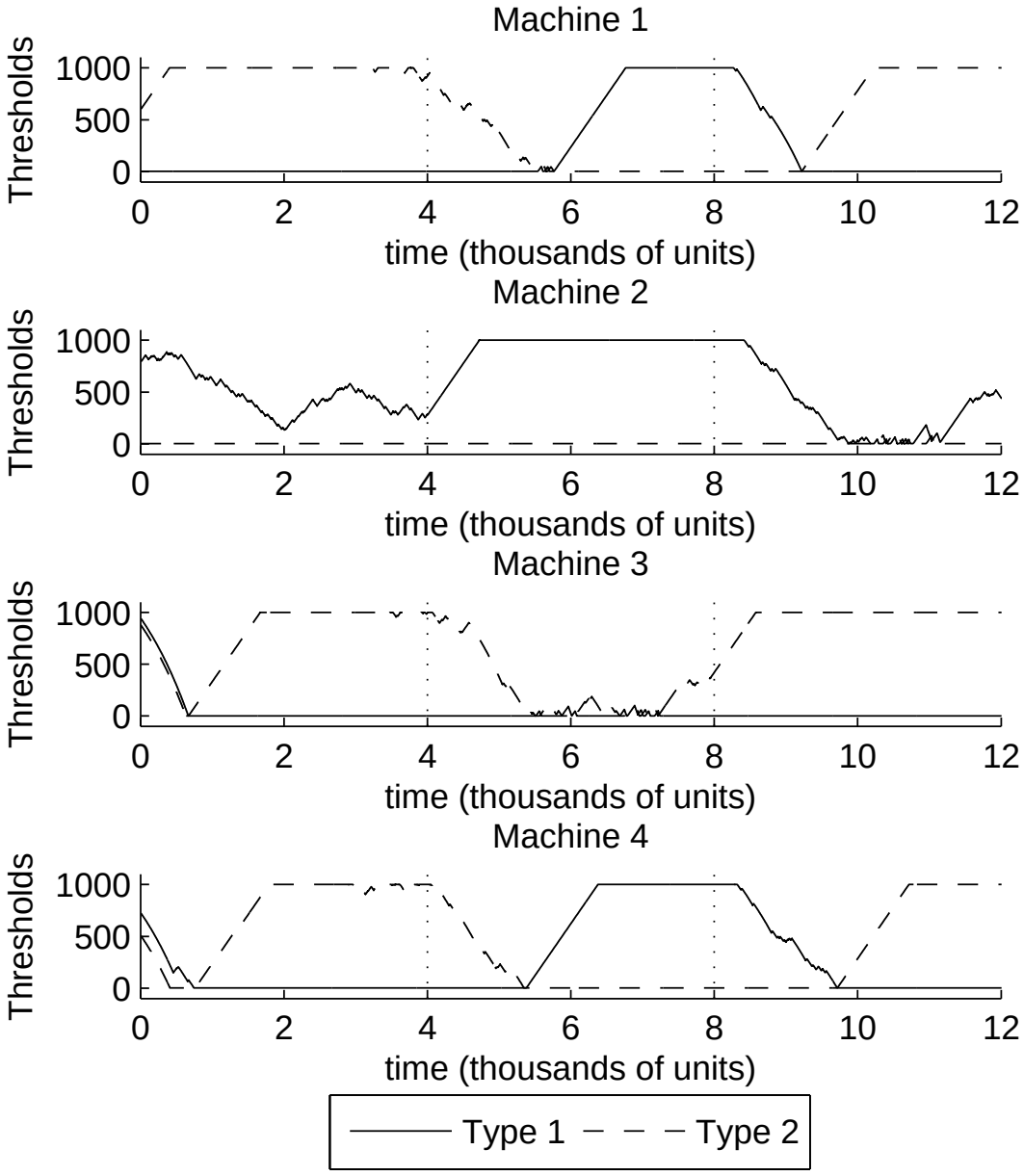

Figure 6.1: Sample run with four machines shows how long it can take to adapt the thresholds for a machine after a change occurs.

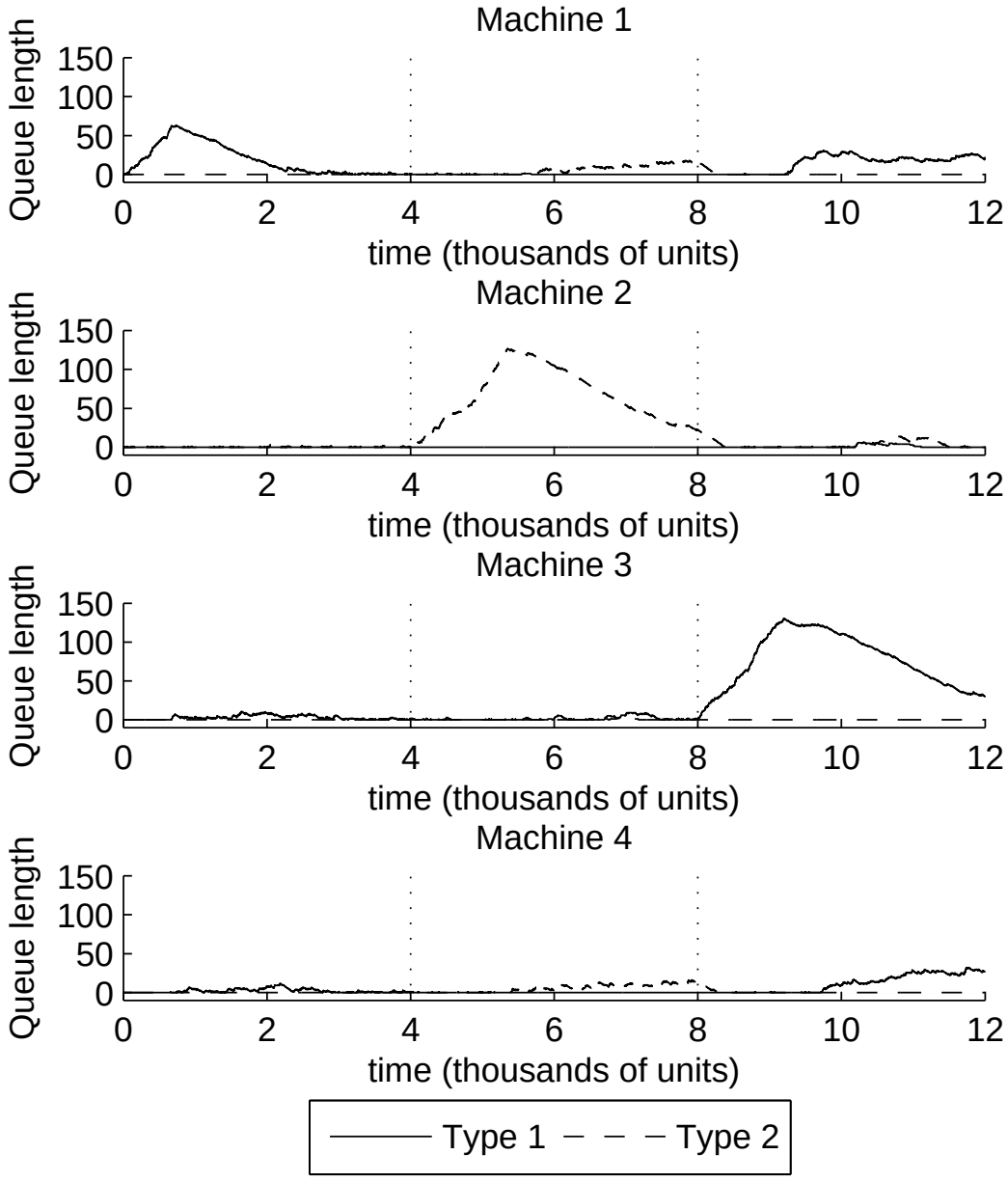

Figure 6.2: Sample run with four machines shows that if thresholds on one machine take a long time to adapt, queues can become very large on other machines.

coordination problem described in Section 6.2. It also proposes several density-estimate memory systems inspired by EDAs that allow the use of many past states while keeping overhead low. By sampling each machine's state as R-Wasps learns response thresholds, we can build a model of the solution space over time which can be used when a new distribution is detected.

One of the weaknesses of the general memory systems that have been studied is the typical small size of the memory. This is because the memory must be reinserted into the population, so it must be much smaller than the population size. In addition, as memory grows, the computational overhead of the memory can become quite large and detract from the resources devoted to optimization.

For the distributed factory coordination problem, only one solution can be retrieved at a time, so the memory can be of any size. Though the length of a time step is not strictly defined for this problem, one can assume it to be on the order of seconds. An infinitely large memory might require enough computation time to detract from the learning process, but even very large memories do not require enough computational overhead to test the limits of a problem changing at this speed. The overhead required of any memory described here is not significant with respect to the length of a time step since the probabilistic models used by the density-estimate memories do not require any search to build the models, as would be necessary for more complex options like mixture models.

6.5.1 Standard memory

We begin by defining a standard memory system, which the other memories will be based upon. This memory system will be denoted as Memory throughout the experiments. In this system, each machine has a memory with a finite number of memory entries β. Each memory entry stores a machine's state at a point in time: the response thresholds Θ_w and the job type distribution D_t. The job type distribution is used as environmental information, while the response thresholds are used as control information. Machines have no knowledge of the true job type distribution, so the distribution must be estimated over some window of time. The current distribution D_t is estimated over the interval $[t-\omega_1, t]$ where t is the current time and ω_1 is the number of time steps to estimate the distribution over. The throughput rate since the last distribution change—the number of jobs completed divided by the time since the last change—is also stored for this standard memory system. Every $1000+N(0,250)$ time units, a machine's memory takes a snapshot of the machine's state and tries to store it in the memory. If the memory is full, a replacement strategy determines whether the new point should replace one of the memory entries. The replacement strategy maintains diversity in the memory [14]. The new point is added to the memory, and the two entries which have the most similar job type distributions are found. The entry with the lower throughput rate is removed from the memory. Each machine has its own memory, and there is no communication between the memories. Machines do not update their memories simultaneously, so memories are not the same for each machine (though the distributed memories tend to be similar).

Since the goal is to retrieve entries from memory after changes in the underlying job type distribution, the memory contains a system for detecting those changes. We can detect changes by comparing the current distribution to one in the past—D_{t-w_2}, computed over the interval $[t-\omega_1-\omega_2, t-\omega_2]$ where ω_2 is the number of time steps in the past the distribution was calculated. Both the current and past distributions can be easily maintained by the agent as new jobs arrive. If the difference between D_t and D_{t-w_2} is large enough, we know the job type distribution has changed. The threshold for a change is ϕ times the mean value of $|D_t - D_{t-\omega_2}|$. If a change is detected, the current job type distribution is compared to the distributions of each memory entry. If the closest entry is less than a distance ε from the current distribution, the machine's thresholds are changed to those of the closest entry.

6.5.2 Density-estimate memory

Instead of storing only single points in memory, we propose to store clusters of points in each memory entry and to create a model of the points in each cluster. Though we will be able to store many more points, the computation overhead required for the memory will remain low. Unless stated otherwise, all of these density-estimate memories use the same mechanisms as the standard memory model described above.

The first density-estimate memory system, DEMc, uses incremental Euclidean clustering to store points in memory. Each machine's memory contains a finite number of memory entries β. Each memory entry is a cluster of stored points—machine states at a particular time. These points are equivalent to the stored points for the standard memory: the response thresholds Θ_w and the job type distribution D_t. Each memory entry averages these values over all the points in its cluster to create a simple cluster model composed of a distribution center c_d and a threshold center c_θ. These values are used when interacting with the memory entry. The memory is updated with the same frequency as for the standard memory. When saving a new point, we create a new memory entry containing only that point and add it to the memory. Then we find the two entries in the memory whose distribution centers are closest together and merge their clusters into a single memory entry, recalculating c_d and c_θ. An entry is retrieved in the same way as in the standard memory model, but instead of using the thresholds from a single point, we change the machine's thresholds to c_θ.

The second density-estimate memory, DEMg, uses incremental Gaussian clustering to store points in memory. Using a Gaussian model in place of Euclidean clustering improves the retrieval of memory entries after a distribution change. In addition to c_d and c_θ, a Gaussian model of the job type distributions in the cluster, m_d, is created. For clusters with fewer than 10 points, the model is padded by adding random points around c_d (uniformly distributed in each dimension in the interval $[-0.125, 0.125]$). Instead of computing the distance between the current job type distribution and the distribution in each memory entry, the Gaussian model in each entry is used to calculate the

probability that the memory entry belongs to the Gaussian (and hence, to the cluster in the memory entry). The entry with the highest probability is selected and the machine's thresholds are changed to c_θ. All the other parts of the memory system are the same as in DEMc.

The third density-estimate memory, DEMgw, extends DEMg by using the Gaussian model of the job type distribution throughout the memory system. In addition to using this environmental model when retrieving an entry from the memory, the model is used to calculate probability when adding new points to the memory. Instead of measuring distance between entries when deciding which to merge, a mean probability is computed. The mean probability of each point in entry 1 being in the model for entry 2 is added to the mean probability of each point in entry 2 being in the model for entry 1. The two entries with the highest probability are merged. Like DEMg, the Gaussian model in DEMgw is also used to retrieve an entry after a change in the underlying distribution is detected. Instead of changing the machine's thresholds to c_θ, a new weighted center is calculated. For each point in the memory entry, a weight w_j is calculated by finding the probability that the job type distribution for point j is part of m_d. The weights are then normalized by dividing them by the sum of all weights. A set of weighted thresholds is formed by multiplying the weight for each point by its thresholds Θ_w. The weighted center wc_θ is the mean of all of these weighted thresholds.

6.6 Experiments

We compared standard R-Wasps to the memory-enhanced versions of R-Wasps described in Section 6.5: Memory, Memory-∞, DEMc, DEMg, and DEMgw. Memory-∞ is exactly the same as the standard memory system, but with no limit on the number of memory entries. Each of the other memories were allowed to store a maximum of 5 entries (5 clusters for the density-estimate memories).

We examined problems with four machines and four job types. Each scenario lasted 150000 time units split into 50 periods of 3000 time units. At the beginning of every period, the job type distribution used to generate new job arrivals changes to a new distribution. This distribution is chosen at random from ten distributions generated at the beginning of the scenario. The distribution for this period is then randomly perturbed, so distributions are not repeated exactly.

For detecting distribution changes, we used $\phi = 2.5$, $\varepsilon = 0.25$, and $\omega_1 = \omega_2 = 100N$, where N is the number of job types. The parameter values for R-Wasps are as described in Section 6.2. We ran scenarios with three values of $\ell \in \{1.00, 1.25, 1.50\}$ to test performance over a variety of loads from normal to overloaded.

To evaluate scenarios, we measure four statistics: throughput, setups, cycle time, and queue length. The throughput statistic measures the percentage of all jobs in the scenario that have been processed

by a machine. The setups statistic is the total number of setups performed by all machines in the system. The cycle time is the average time a job spends in the system from when it arrives until it is finished being processed. The queue length is the average number of jobs in a machine's queue over the entire scenario.

6.7 Results

Tables 6.1, 6.3, and 6.5 show average results from 20 scenarios with loading values $\ell = \{1.00, 1.25, 1.50\}$. Tables 6.2, 6.4, and 6.6 compare the approaches, showing the percent improvement of memory approaches. When results are statistically significant, the result is marked accordingly. For all experiments in this chapter, the statistical significance of the results has been evaluated using the Kruskal-Wallis test, considering a confidence of 95% ($p = 0.05$). The Kruskal-Wallis test, a one-way analysis of variance by ranks, is a nonparametric equivalent to the classical one-way analysis of variance (ANOVA) that does not assume data are drawn from a normal distribution [29].

When $\ell = 1.00$, the system has a medium to high load, and all six approaches had throughputs above 98%. Incomplete jobs remaining at the end of the scenario existed mostly because of jobs that arrived too late to be processed, since jobs could potentially arrive one time unit prior to the end of the scenario. Since the standard version of R-Wasps completed over 99% of jobs, there was not much room for improvement in throughput. The density-estimate memories greatly reduced the number of setups as well as the cycle time and queue length for these scenarios. The standard memory actually hurt performance in all areas.

When $\ell = 1.25$, the system has a high load. The results for the six approaches varied quite a bit more for the more highly loaded system than they did when $\ell = 1.00$. The standard R-Wasps approach only completed 89.84% of jobs on average, while the addition of memory raised this average above 93% for every type of memory. Once again, the density-estimate memories gave the best performance, with large reductions in number of setups, average cycle time, and average queue length.

When $\ell = 1.50$, the system is overloaded. Compared to scenarios with lighter loads, the throughput decreased for all approaches, with an average throughput of under 80% for standard R-Wasps. The addition of memory still resulted in significant improvement over standard R-Wasps, particularly for the two more complex density-estimate memories, DEMg and DEMgw. DEMg had the best throughput and cycle time, while DEMgw had the fewest setups and smallest average queue size, but the differences in performance between these two approaches was small. The largest area of improvement in these scenarios over standard R-Wasps was in reducing the number of setups.

Approach	throughput	setups	cycle time	queue length
R-Wasps	99.05	2029.25	20.02	59.04
Memory	98.64	2478.20	22.96	68.94
Memory-∞	99.27	1913.45	16.20	48.03
DEMc	99.43	1371.45	11.31	33.76
DEMg	99.65	1213.10	10.10	29.80
DEMgw	99.36	1550.30	13.93	41.71

Table 6.1: Average results for scenarios with $\ell = 1.00$

Approach 1	Approach 2	throughput	setups	cycle time	queue length
Memory	R-Wasps	-0.41	-22.12 (-)	-14.66	-16.76
Memory-∞	R-Wasps	0.22	5.71	19.10	18.66
Memory-∞	Memory	0.64	22.79 (+)	29.44 (+)	30.33 (+)
DEMc	R-Wasps	0.38	32.42 (+)	43.53 (+)	42.83 (+)
DEMc	Memory	0.80 (+)	44.66 (+)	50.75 (+)	51.03 (+)
DEMc	Memory-∞	0.16	28.33 (+)	30.19 (+)	29.71
DEMg	R-Wasps	0.60 (+)	40.22 (+)	49.58 (+)	49.52 (+)
DEMg	Memory	1.02 (+)	51.05 (+)	56.02 (+)	56.77 (+)
DEMg	Memory-∞	0.38 (+)	36.60 (+)	37.67 (+)	37.95 (+)
DEMg	DEMc	0.22	11.55	10.71	11.72
DEMgw	R-Wasps	0.32	23.60 (+)	30.45 (+)	29.36
DEMgw	Memory	0.74 (+)	37.44 (+)	39.34 (+)	39.50 (+)
DEMgw	Memory-∞	0.09	18.98	14.03	13.16
DEMgw	DEMc	-0.07	-13.04	-23.16	-23.55
DEMgw	DEMg	-0.28	-27.80	-37.93	-39.95

Table 6.2: Percent improvement of approach 1 over approach 2 for each metric with $\ell = 1.00$ (results that are statistically significant to 95% confidence are noted with a + or -)

Approach	throughput	setups	cycle time	queue length
R-Wasps	89.84	2514.05	137.70	538.06
Memory	93.16	1821.00	106.79	403.87
Memory-∞	93.10	1841.45	110.37	419.28
DEMc	95.29	1208.10	85.84	323.91
DEMg	94.86	1299.05	90.79	344.41
DEMgw	96.50	820.95	61.17	234.51

Table 6.3: Average results for scenarios with $\ell = 1.25$

Approach 1	Approach 2	throughput		setups		cycle time		queue length	
Memory	R-Wasps	3.70	(+)	27.57	(+)	22.44	(+)	24.34	(+)
Memory-∞	R-Wasps	3.63	(+)	26.75		19.85		22.98	
Memory-∞	Memory	-0.07		-1.12		-3.35		-1.79	
DEMc	R-Wasps	6.07	(+)	51.95	(+)	37.66	(+)	38.63	(+)
DEMc	Memory	2.29	(+)	33.66	(+)	19.62		18.89	
DEMc	Memory-∞	2.36	(+)	34.39	(+)	22.22		20.32	(+)
DEMg	R-Wasps	5.59	(+)	48.33	(+)	34.06	(+)	35.60	(+)
DEMg	Memory	1.82	(+)	28.66	(+)	14.98		14.88	
DEMg	Memory-∞	1.89	(+)	29.46	(+)	17.74		16.38	
DEMg	DEMc	-0.45		-7.53		-5.77		-4.94	
DEMgw	R-Wasps	7.41	(+)	67.35	(+)	55.58	(+)	49.64	(+)
DEMgw	Memory	3.58	(+)	54.92	(+)	42.72	(+)	33.45	(+)
DEMgw	Memory-∞	3.65	(+)	55.42	(+)	44.58	(+)	34.62	(+)
DEMgw	DEMc	1.26		32.05	(+)	28.75	(+)	17.95	(+)
DEMgw	DEMg	1.73		36.80		32.63	(+)	21.81	(+)

Table 6.4: Percent improvement of approach 1 over approach 2 for each metric with $\ell = 1.25$ (results that are statistically significant to 95% confidence are noted with a + or -)

Approach	throughput	setups	cycle time	queue length
R-Wasps	79.33	1935.15	262.83	1229.65
Memory	82.71	1169.85	233.78	1073.85
Memory-∞	81.10	1489.70	244.26	1148.97
DEMc	81.71	1374.00	241.27	1114.86
DEMg	83.08	1057.15	225.57	1051.82
DEMgw	83.06	1046.00	226.87	1041.66

Table 6.5: Average results for scenarios with $\ell = 1.50$

Approach 1	Approach 2	throughput	setups	cycle time	queue length
Memory	R-Wasps	4.27 (+)	39.55 (+)	11.05	12.67
Memory-∞	R-Wasps	2.23	23.02	7.06	6.56
Memory-∞	Memory	-1.95	-27.34	-4.49	-7.00
DEMc	R-Wasps	3.00	29.00	8.20	9.34
DEMc	Memory	-1.21	-17.45	-3.20	-3.82
DEMc	Memory-∞	0.75	7.77	1.23	2.97
DEMg	R-Wasps	4.73 (+)	45.37 (+)	14.18 (+)	14.46 (+)
DEMg	Memory	0.44	9.63	3.51	2.05
DEMg	Memory-∞	2.44	29.04	7.65	8.45
DEMg	DEMc	1.67	23.06	6.51	5.65
DEMgw	R-Wasps	4.71 (+)	45.95 (+)	13.68 (+)	15.29 (+)
DEMgw	Memory	0.42	10.59	2.95	3.00
DEMgw	Memory-∞	2.42	29.78	7.12	9.34
DEMgw	DEMc	1.66	23.87	5.97	6.57
DEMgw	DEMg	-0.02	1.05	-0.58	0.97

Table 6.6: Percent improvement of approach 1 over approach 2 for each metric with $\ell = 1.50$ (results that are statistically significant to 95% confidence are noted with a + or -)

6.8 Discussion

Based upon these results, the memory-enhanced versions of R-Wasps exhibit better performance than the standard version of R-Wasps for the dynamic distributed factory coordination problem. Though all of the memories performed well, the density-estimate memories introduced here consistently outperformed both the standard memory and the infinite-sized standard memory.

The standard memory system, Memory, improved performance over R-Wasps for higher loads, but that improvement was only significant at the highest load tested. In fact, at the lowest load levels, Memory actually hurt performance when compared to R-Wasps, with a significant increase in the number of setups required. Given the limitations of the fixed-size memory explained earlier, this is not surprising. The infinite-sized memory, Memory-∞, also improved performance over R-Wasps for higher loads, though without the drop in performance that Memory showed at lower loads. However, the increase in overhead did not allow Memory-∞ to outperform the density-estimate memories.

DEMc improved significantly on the standard memory models under the two lower loads. Despite using only a very simply model—clustering points and using the centers of each cluster to interact with the memory entry—enhancing R-Wasps with this type of memory significantly improved performance. By aggregating many solutions, the memory was able to overcome the noise inherent in detecting the current job type distribution.

When compared with standard R-Wasps, DEMg was the only one of the five memory approaches that showed statistically significant improvement for all four statistics on all three load scenarios. In addition to being the most consistent, DEMg was the best memory for $\ell = 1.00$. Though it did not always outperform DEMc, the addition of the Gaussian model used to choose which memory entry to retrieve seems to have made this approach more consistent.

DEMgw, the most complex model, showed statistically significant improvement for all four statistics on the two higher loads. It was the best approach when $\ell = 1.25$, with improvement over all other approaches on all statistics—the majority of improvements were statistically significant. However, it was outperformed by the other two density-estimate memories when $\ell = 1.00$. Under lighter loads, the estimation of the current job type distribution is noisier, since fewer jobs arrive during the time window used to estimate the distribution. Since DEMgw tries to exploit more information from the points in memory than DEMc or DEMg, it is more susceptible to this noise. As estimates of the distribution get better, performance improves.

6.9 Summary

For dynamic problems, using information from the past can help improve performance when the current state of the environment is similar to a previous state. One way to exploit past information is through the use of memory. Standard memory models exist, but have a limited ability to model dynamic solution landscapes. In this chapter, we have introduced three density-estimate memory systems that improve upon standard memory without large increases in the overhead required to maintain and use the memory.

By enhancing R-Wasps with memory, performance improves on the dynamic distributed factory coordination problem. Each agent has a separate memory, so the distributed agent-based solution is preserved, which improving adaptability when changes in the underlying job type distribution occur. R-Wasps also maintains control of the system except immediately after changes in the distribution, so the system remains flexible.

The density-estimate memories outperformed both the standard R-Wasps algorithm as well as R-Wasps enhanced with a standard memory. In particular, the density-estimate memories significantly reduced the number of setups required. These density-estimate approaches produce more robust memories with very little increase in overhead.

Chapter 7

Dynamic optimization with evolutionary algorithms

While the use of historical data is common in learning and optimization, building and maintaining an explicit memory of past solutions has not been common for use by most dynamic learning or optimization processes. In the area of dynamic optimization with evolutionary algorithms, the use of explicitly constructed memories has been widely explored in recent literature. Evolutionary algorithms encompass a variety of stochastic optimization techniques inspired by natural selection and biological evolution. Typically, evolutionary algorithms perform a population-based search using operators like recombination (crossover) and mutation. Population-based search is well suited to use with a memory, as multiple elements from the memory can be reintroduced into the search process without interrupting search on promising areas of the landscape. Thus, when considering a new type of memory like density-estimate memory, dynamic optimization with evolutionary algorithms offers multiple state-of-the-art techniques for comparison.

One of the most commonly used benchmark problems for dynamic optimization with evolutionary algorithms is the Moving Peaks problem, designed by Jürgen Branke [13, 14]. The Moving Peaks problem has a multimodal, multidimensional fitness landscape. Each time a change in the problem environment occurs, the height, width, and position of each peak changes. The problem is highly configurable and allows a great deal of control over the search space; because of this, Moving Peaks is a very suitable benchmark problem despite not being analogous to any dynamic environment in the real world. Due to its use in experiments in the literature, the Moving Peaks problem also allows comparison to existing techniques.

In this chapter, evolutionary algorithm density-estimate memory techniques are compared with state-of-the-art methods from the literature, including a variety of memory techniques. In particular, density-estimate memory is compared to self-organizing scouts [14], a state-of-the-art technique

that uses the population-based search of evolutionary algorithms as a de facto memory capable of constant refinement. All methods are applied to the dynamic optimization of the Moving Peaks benchmark problem.

7.1 Moving Peaks benchmark problem

The Moving Peaks benchmark[1] is a multimodal, multidimensional dynamic problem. Proposed by Branke [13] in 1999, it is very similar to another benchmark problem proposed independently by Morrison and DeJong [72]. In Moving Peaks, the landscape is composed of m peaks in an n-dimensional real-valued space. At each point, the fitness is defined as the maximum over all m peak functions. This fitness can be formulated as

$$F(\vec{x},t) = \max_{i=1\ldots m} P(\vec{x}, h_i(t), w_i(t), \vec{p}_i(t))$$

where $P(\ldots)$ is a function[2] describing the fitness of a given point ($\vec{x}$) for a peak described by height (h), width (w), and peak position ($\vec{p}$).

Every Δe evaluations, the height, width, and position are changed for each peak, changing the state of the environment. The height and width of each peak are changed by the addition of Gaussian random variables scaled by height severity (hs) and width severity (ws) parameters. The position is shifted using a shift length s and a correlation factor λ. The shift length controls how far the peak moves, while the correlation factor determines how random a peak's motion will be. If $\lambda = 0.0$, the motion of a peak will be completely random, but if $\lambda = 1.0$, the peak will always move in the same direction until it reaches a boundary of the coordinate space where its path reflects like a ray of light. At the time of a change in the environment, the changes in a single peak can be described as

$$\begin{aligned} \sigma &\in N(0,1) \\ h_i(t) &= h_i(t-1) + hsev \cdot \sigma \\ w_i(t) &= w_i(t-1) + wsev \cdot \sigma \\ \vec{p}_i(t) &= \vec{p}_i(t-1) + \vec{v}_i(t) \end{aligned}$$

The shift vector $\vec{v}_i(t)$ combines a random vector $\vec{r}$ with the previous shift vector $\vec{v}_i(t-1)$. The random vector is created by drawing uniformly from $[0,1]$ for each dimension and then scaling the vector to have length s.

$$\vec{v}_i(t) = \frac{s}{|\vec{r} + \vec{v}_i(t-1)|}\left((1-\lambda)\vec{r} + \lambda\vec{v}_i(t-1)\right)$$

[1] The C code for this benchmark is currently available at http://people.aifb.kit.edu/jbr/MovPeaks/

[2] The definition of $F(\vec{x},t)$ in [14] includes an optional time-invariant basis function, $B(\vec{x})$. No basis function is used for these experiments so it has been removed for clarity.

The height, width, and position of each peak are randomly initialized within constrained ranges. A number of peak functions are available, but in this chapter, only the cone function is used. This peak function is defined as

$$P(\vec{x}, h(t), w(t), \vec{p}(t)) = h(t) - w(t) \cdot \sqrt{\sum_{j=1\ldots n} (x_j - p_j)^2}$$

It should be noted that the width of a peak in this benchmark is defined in what may be a confusing manner. The $\sqrt{\cdots}$ portion of the peak function is the distance between the given point and the position of the peak. As the width increases, the peak becomes narrower, and when $\vec{x} \neq \vec{p}$, the value of the peak function decreases. As $w \to \infty$, the peak function converges to a delta function. Likewise, as $w \to 0$, the peak function converges to a constant basis function. While this confusion could be removed by inverting the width parameter, then comparison with other results using this benchmark problem would be more difficult.

7.2 Evolutionary algorithms

Evolutionary algorithms encompass a large class of biologically-inspired, stochastic search methods based on principles of natural evolution [48]. Though implementations of evolutionary algorithms vary widely, in general an evolutionary algorithm applies search operations like recombination (crossover) and mutation to a population of candidate solutions. Selective pressure is applied through the use of an objective function (fitness function). As better performing solutions are more likely to be chosen for genetic operations and contribute to future generations of the population, the population "evolves" and the average performance of an individual in the population increases. Evolutionary algorithms are well suited for optimization of dynamic problems for a variety of reasons, in particular because the population functions as a short term memory, potentially keeping individuals in many areas that can be searched after a change occurs in the environment. However, the population of an evolutionary algorithm may sometimes converge into one area of the search space, leaving the algorithm ineffective after a change. For this reason, many evolutionary algorithm variants specifically designed for dynamic problems have been investigated (e.g. hypermutation [42] and memory [81, 13]).

Comparing different evolutionary algorithm variants on dynamic problems is challenging for a number of reasons. Some benchmark problems may be better suited for one approach over another, and the stochastic nature of evolutionary algorithm search may mask true performance differences if enough evaluations are not performed. Evolutionary algorithms also have a large number of tunable parameters, and the same set of parameters may not be optimal for different evolutionary algorithm variants. Optimal parameter settings cannot typically be derived theoretically, but must often be

Algorithm 7.1 Basic operations of the evolutionary algorithm

```
init(Pop(0)) initialize the population
eval(Pop(0))
t = 1
WHILE (termination criteria not fulfilled)
    Pop(t) = ∅
    Parents(t) =
    Pop(t−1) ∪ Memory combine old population with memory
    WHILE |Pop(t)| < popsize
        M(t) =
        select(Parents(t)) select individuals and copy to mating pool
        M'(t) = crossover(M(t)) perform crossover
        M''(t) = mutation(M'(t)) perform mutation
        Pop(t) = Pop(t) ∪ M''(t) update population
    eval(Pop(t)) evaluate individuals in the population
    t = t + 1
```

tuned through experimentation. This tuning may also make an evolutionary algorithm variant more brittle as the type of dynamic problem changes. Probably the most common approach is to use the same parameter settings for all algorithms being compared and not to fine tune those parameters for any particular variant, instead choosing reasonable parameters that generally perform well on the problem. This approach has been adopted in most previous work in the area of dynamic optimization with evolutionary algorithms.

The parameter settings used here are adopted from those used in [14]. The basic outline of the evolutionary algorithm is shown in Algorithm 7.1 and the parameters are shown in Table 7.1. The evolutionary algorithm uses a real-valued encoding and generational replacement with one elite (the best individual in a generation is copied to the next generation unchanged). Two-point crossover is used with probability 0.6. Then, mutation takes place with probability $\frac{1}{n}$ where n is the length of the solution vector (chromosome). In these experiments, the Moving Peaks problem has five dimensions, so the mutation probability is 0.2. Mutation takes place by adding a Gaussian random variable to each vector entry (allele), i.e. $x_i \leftarrow x_i + \delta$ with $\delta \in N(0, 3.3)$. The evolutionary algorithm uses a population size of 100 individuals, which includes the number of memory entries if a memory is used. Since all individuals in the population are evaluated every generation, including those in memory, each generation requires 100 evaluations of the objective function. If a memory is used, a new point may be stored every 10 generations and individuals are retrieved from the memory every generation.

The experiments in this chapter compare several evolutionary algorithm variants from the literature,

Parameter	Value
Population size	100
Crossover probability	0.6
Mutation probability	$\frac{1}{n} = 0.2$
Mutation distribution	$\delta \in N(0, 3.3)$
Memory size	10
Number of random immigrants	25

Table 7.1: Evolutionary algorithm parameter settings

all based on the standard evolutionary algorithm (SEA) described above. Results for these variants on several dynamic problems, including the Moving Peaks problem, can be found in [14]. While many other specialized evolutionary algorithms have been designed for use on dynamic problems, the variants considered here provide a good basis of comparison for density-estimate memory.

Standard evolutionary algorithm with memory The standard evolutionary algorithm is supplemented with a standard memory as described in Chapter 4 for the standard evolutionary algorithm with memory (SEAm) approach. The *mindist2* replacement strategy is used to maintain the memory.

Random immigrants The random immigrants technique [43, 42] was designed to increase the diversity of a population to avoid over-convergence. Random immigrants are randomly generated solutions that are inserted into the population. In these experiments, random immigrants replace the worst $\frac{1}{4}$ of the individuals in the population. In addition to a SEA with random immigrants (RI), a variant that also uses a standard memory (RIm) is considered.

Memory/search Sometimes, using memory alone can inhibit exploration of the search space by finding good, but suboptimal areas of the search space quickly. While a random restart or hypermutation after a change in the environment can lead to more thorough exploration of the search space, these approaches may increase the amount of time necessary to find good solutions. In memory/search techniques (memsearch), the total population size is divided into multiple populations, some of which are devoted to search, some to the use of memory [14]. In these experiments, an approach with two populations is considered as shown in Figure 7.1. The search population can store solutions to the memory and is randomly initialized whenever a change in the environment is detected. The memory population can both store and retrieve solutions from the memory. The memory population maintains and refines good solutions and ensures that the quality of the best solution in the entire population remains high after a change has occurred. The search population searches widely through the environment and introduces diversity into the memory.

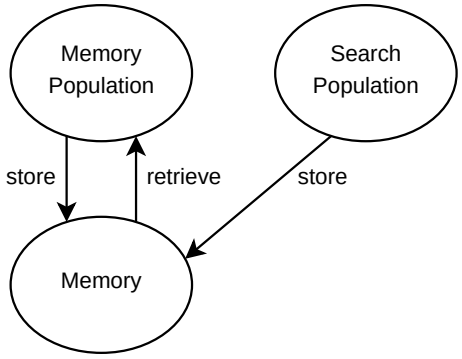

Figure 7.1: Memory/search multi-population technique. The population is divided into a memory population and a search population. The memory population can store solutions to and retrieve solutions from the memory. The search population can only store solutions to the memory and is randomly reinitialized when a change to the environments is detected.

Self-organizing scouts While a standard memory stores solutions that have been good in the past, improving the quality of these memory entries can be difficult. To refine an entry, it must be replaced completely by another solution. Often, information in the memory becomes obsolete over time, reducing the effective size of the memory and making it less useful. Self-organizing scouts (SOS) is a state-of-the-art multi-population evolutionary algorithm approach designed to overcome this limitation of a standard memory [15, 14]. SOS begins with some number of base subpopulations searching for good solutions. When a peak (region with good solutions) has been found, the population splits. A scout population is formed to keep track of the peak, while the base population resumes searching for other peaks. If a peak is being followed by a scout population, the population can refine itself as the peak moves and changes. Since the total population size is limited, individuals in a scout population may be reassigned to other subpopulations if new areas of interest are found, or another peak being followed by a scout population looks more promising. If a peak becomes very unpromising, the scout population following that peak may be abandoned. SOS can be considered an evolutionary algorithm with memory allowing constant refinement of the memory entries. SOS only allows one subpopulation to search a particular area at a time, effectively dividing the search space and avoiding overlapping search. The area covered by a particular subpopulation changes depending on the individuals in the subpopulation. By partitioning the search space, the base populations can avoid duplicating the search of the scout populations and can maintain search diversity. The default parameters for SOS in these experiments are the same as in [14]. Selected parameters are shown in Table 7.2. Unlike the other memory methods in these experiments, which are only allowed a maximum of 10 memory entries, SOS may form up to 20 scout populations.

Parameter	Value
Total population size	100
Number of base populations	2
Minimum size of a base population	8
Maximum number of scout populations	20
Minimum size of a scout population	4
Minimum radius of a scout population	6.0
Maximum radius of a scout population	12.0

Table 7.2: Selected parameters for self-organizing scouts

7.3 Density-estimate memory

Density-estimate memory (DEM), described in Chapter 5, is designed to overcome many of the weaknesses of the standard memory, particularly a limited model of the dynamic search space, difficulty in refining memory entries, and ability to store information about a few good solutions. This is accomplished by aggregating information from many good solutions by building probabilistic models within the memory. Density-estimate memories may be implemented in many ways. In this chapter, some basic density-estimate memory methods are compared to the evolutionary algorithm variants described above. Like all of the evolutionary algorithm variants described above, the density-estimate memory algorithms use the standard evolutionary algorithm as a basis. Since the Moving Peaks problem has no available environmental information, the solution vector itself is used both as environmental and control information for a density-estimate memory.

Euclidean clustering Two types of clustering were considered for density-estimate memories. The first, Euclidean clustering, computes only cluster centers and measures the Euclidean distance between clusters. This clustering method requires very little overhead, as the only part of the model that must be computed is the mean of the points in a memory entry. However, this approach does not provide a true density-estimate for use by the memory, since no information about the distribution of points is available when accessing the memory. Euclidean clustering is indicated with a *c* (e.g. SEA with a Euclidean clustering density-estimate memory would be denoted as DEMc).

Gaussian clustering The second type of clustering for density-estimate memories, Gaussian clustering, provides richer density-estimate models for use by the memory. In this approach, each memory entry computes a Gaussian model of the points in the entry. The mean and covariance matrix may then be used to calculate the probability that a new point belongs to an existing cluster or the probability that two clusters should merge. Gaussian clustering is indicated with a *g*.

Single population Though single population evolutionary algorithms have not typically performed as well on the Moving Peaks problem as multi-population techniques, single population density-estimate memory methods are tested as a baseline. These methods augment SEA with a density-estimate memory and are indicated with an *s*.

Memory/search In [14], memory/search outperformed the single population approach (SEAm) for the standard memory system. Memory/search improves the quality of the memory by adding diversity to the search, something that is also necessary for density-estimate memory. For the experiments in this chapter, density-estimate memories use the memory/search approach unless the single population approach is indicated.

Random immigrants Since memory techniques often benefit from increased diversity, random immigrants in combination with density-estimate memories are tested. Since memory/search techniques already include diversity via the search population, random immigrants are only used for single population evolutionary algorithms. Random immigrants are indicated by *i*.

Informed diversity While diversity is often useful to a memory, diversity techniques like the search population in memory/search can sometimes add redundancy to the search process by initializing new individuals in areas well covered by the memory. A standard memory only provides information about the location of single points, while a density-estimate memory can provide information about the density of good solutions in a particular area. This informed diversity technique replaces the random initialization of the search population in a memory/search method with initialization that tries to create random points in areas not well covered by the memory. In each memory entry, the maximum distance between the mean of the entry and all the points contained in the entry is saved as part of the model. When reinitializing the search population after a change, each new individual is checked to see that it is further from the mean of each memory entry than that entry's maximum distance. The use of informed diversity is indicated with *d*.

Reclustering With the incremental clustering used for density-estimate memory, sometimes one or two very large entries come to dominate the memory, since entries can only merge, not split. Splitting a large entry into multiple entries or combining several smaller entries into a single entry may improve the quality of the density-estimate stored in the memory. One way of making this possible is through periodic reclustering. A periodic k-means reclustering was used to investigate the effects of reclustering on the performance of the memory. K-means is a simple clustering algorithm with low overhead. Reclustering is indicated by *r*.

Including fitness in environmental model Since the Moving Peaks problem has no environmental information aside from the location of a point, including any additional information available might enrich the models built in memory. Since solution fitness is the only other available piece of data, including the fitness of a solution at the time it is stored in memory might be beneficial. In these experiments, the control data remain the same, but the environmental data of a point include both the location of that point and the fitness at storage time. This fitness is not updated, as the number of evaluations would be prohibitive. When fitness is included in the environmental model, the experiment is indicated by *f*.

7.4 Experiments

Many prior works have investigated problems where changes are small and algorithms must track those changes quickly. However, many real problems are much more discontinuous, with changes that are not random, but revisit previous solution areas as the problem progresses. These types of problems may include scheduling and adaptive traffic control, which are investigated elsewhere in this thesis. This benchmark allows fine control over the parameters to investigate what problem types density-estimate memory is especially suited for.

The search space of an instance of the Moving Peaks benchmark problem is described by a large number of parameters. The main set of experiments in this chapter uses a single set of parameters to generate search spaces with a high density of good solutions and very severe changes. The effects of varying one or more parameters are also considered. Four parameters were considered: the frequency of changes, the severity of changes in height, the number of peaks, and the maximum value for peak width (as mentioned in Section 7.1, the width parameter is actually inversely proportional to the width of a peak, so this should really be considered the minimum peak width). Table 7.3 shows the default settings for the Moving Peaks benchmark problem in these experiments. These values are very similar to the default values of Branke in [14], though peaks move less and have a larger range of heights. The values for the four parameters that vary are shown in Table 7.4. Change frequency is the number of evaluations performed before the environment changes. Height severity is the maximum change in height of a peak during one change in the environment. Peak width is the minimum width of a peak, where larger values produce narrower peaks. The final parameter that is varied is the number of peaks in the search space. The default parameter values here are different than those used by Branke. The search spaces produced by these default parameters have very severe, discontinuous changes and numerous wide peaks.

The importance of each of these four parameters on the performance of density-estimate memory was evaluated by varying one parameter at a time in the ranges shown in Table 7.4. While changes in height severity and change frequency are mostly independent of the other parameters, the peak

Parameter	Value
Number of dimensions (n)	5
Coordinate range	$[0, 100]$
Shift length (s)	0.5
Correlation factor (λ)	0.5
Change frequency (Δe)	varies
Number of peaks (m)	varies
Peak height range	[10,90]
Height change severity ($hsev$)	varies
Peak width range	[0.5, varies]
Width change severity ($wsev$)	1.0

Table 7.3: Default settings for the Moving Peaks benchmark problem

Parameter	Default	Branke	Parameter ranges
Change frequency	5000	5000	{ 1000 3000 5000 }
Height severity	21.0	7.0	{ 7.0 14.0 21.0 28.0 }
Peak width	4.0	12.0	{ 2.0 3.0 4.0 6.0 8.0 12.0 }
Number of peaks	60	10	{ 10 30 60 90 }

Table 7.4: Parameter values for a dense, discontinuous version of the Moving Peaks benchmark

width and number of peaks together describe the density of the search space. To investigate the effects of search space density, these parameters were also varied simultaneously with the peak width in {4, 8, 12} and the number of peaks in {30, 60, 90}.

As mentioned, all methods used the same underlying evolutionary algorithm. A variety of techniques from the literature were evaluated along with many variants of density-estimate memory. Table 7.5 explains the abbreviations used for evolutionary algorithm variants. Offline error is used to evaluate the performance of algorithm variants. At each generation, the fitness of the best individual in the population is subtracted from the current global optimum (the height of the highest peak, a value not available to the optimization process). The average over all generations is the offline error for a particular run. Each run lasts 20,000 generations; with a change frequency of 5000 and a population size of 100, the environment changes every 50 generations for a total of 399 changes. Memory techniques build memories over the course of a single run, beginning with an empty memory. Error values for a given technique are averaged over 100 independent runs.

7.5 Results

Table 7.6 shows the average offline error values (over 100 runs) of a selection of evolutionary algorithm methods on the default Moving Peaks problem defined by the parameters in Tables 7.3

Abbreviation	Description
SEA	standard evolutionary algorithm
RI	random immigrants (25% of population)
memsearch	two population memory/search
SOS	self-organizing scouts
DEM	density-estimate memory (two population)
m	standard memory
g	Gaussian clustering density-estimate memory
c	Euclidean clustering density-estimate memory
s	single population
i	random immigrants (25% of population)
r	reclustering
d	informed diversity
f	include fitness in model

Table 7.5: Abbreviations for evolutionary algorithm methods

and 7.4. As expected, the standard algorithm (SEA) performed worst. The addition of memory (SEAm) and diversity (RI) improved performance. The single population density-estimate memories (DEMgs and DEMcs) were better than the standard memory, but due to a lack of diversity, these density-estimate memory methods were outperformed by the combination of random immigrants and memory (RIm). The two density-estimate memory methods using the memory/search approach (DEMg and DEMc) had better average error values than self-organizing scouts and memory/search with a standard memory. The differences in error between the methods shown here were all statistically significant. For all experiments in this chapter, the statistical significance of the results has been evaluated using the Kruskal-Wallis test, considering a confidence of 95% ($p = 0.05$). The Kruskal-Wallis test, a one-way analysis of variance by ranks, is a nonparametric equivalent to the classical one-way analysis of variance (ANOVA) that does not assume data are drawn from a normal distribution [29].

One of the reasons that memory is useful here is due to the specific dynamics of the problem. For this version of the Moving Peaks benchmark problem, peaks move slowly in the search space, but the height of a peak may change drastically. When a change occurs in the environment, the global optimum often jumps between peaks, rather than following the peak with the previously optimal solution. The factory coordination problem in Chapter 6 has similar dynamics. A memory can help search move quickly to a new area of the search space, so when the global optimum jumps, memory lets search follow.

Many approaches to maintaining the diversity of solutions in the population were also considered. Table 7.7 shows the average error values for density-estimate memories with and without diversity measures. The combination of density-estimate memory with any of the diversity techniques

Method	Average error
SEA	18.3717
RI	11.1810
SEAm	10.3838
DEMgs	9.9276
DEMcs	9.6100
RIm	9.2397
memsearch	7.7636
SOS	7.6966
DEMc	5.9584
DEMg	5.6768

Table 7.6: Average offline error values on the default Moving Peaks problem

Method	Average error
DEMgs	9.9276
DEMcs	9.6100
DEMcsi	8.4909
DEMgsi	8.3516
DEMcd	5.9670
DEMc	5.9584
DEMg	5.6768
DEMgd	5.6628

Table 7.7: Average offline error values for diversity methods with density-estimate memories on the default Moving Peaks problem

outperformed the single population density-estimate memory evolutionary algorithms (DEMcs and DEMgs). When both were combined with density-estimate memory, the memory/search approach outperformed random immigrants. All differences in error are significant except between DEMg and DEMgd and between DEMc and DEMcd. Though the informed diversity technique slightly improved average error values, this improvement was not significant, at least for this problem.

Though the incremental method of building density-estimate memories described here may sometimes lead to poor clustering, introducing periodic reclustering with a simple clustering algorithm like k-means did not improve performance. Table 7.8 shows the average offline error values for density-estimate memories that use reclustering. These results show that the more frequently a density-estimate memory is reclustered, the poorer the performance for both Gaussian and Euclidean clustered density-estimate memories. Though density-estimate memories with reclustering still performed better than self-organizing scouts, it seems strange that reclustering based on all available data would perform worse than the incremental clustering normally used in density-estimate memory. The methods without reclustering were significantly better than those with reclus-

Method	Frequency	Average error
DEMg	500	6.2087
DEMg	1000	6.1965
DEMc	500	6.1566
DEMc	1000	6.1439
DEMg	2000	6.0932
DEMc	2000	6.0637
DEMc	5000	5.9897
DEMg	5000	5.9800
DEMc	never	5.9584
DEMg	never	5.6768

Table 7.8: Average offline error values for density-estimate memories with reclustering on the default Moving Peaks problem

tering. In these density-estimate memories, once a point is added to memory, it is never removed. In some cases, poorly performing solutions and other outliers may be added to the memory. With incremental clustering, these outliers have less of an effect on how clusters are divided than during a k-means reclustering. Reclustering by distance alone also does not fit well with the models in DEMg. The probabilistic models used for these density-estimate memories may also affect the performance of reclustering as much as the type of clustering. Both the Euclidean and Gaussian incremental clustering density-estimate memories use sample means, and the Gaussian density-estimate memory uses sample covariance. Though these are common and easy to compute, they are sensitive to outliers. Using different probabilistic models might make reclustering worthwhile for this problem.

Table 7.9 shows the average offline error values for density-estimate memories that include fitness in the environmental model. In comparing the difference in error between a particular method with and without fitness in the environmental model, the only statistically significant differences are between DEMg and DEMgf and DEMgd and DEMgdf. Including fitness did not give any consistent benefit, though it did help slightly for Euclidean clustering density-estimate memories as long as reclustering was not also used.

7.5.1 Examining the effects of varying a single parameter at a time

The results using the default parameters show that density-estimate memory using Gaussian clustering performed better than the state-of-the-art self-organizing scout method for a version of the Moving Peaks problem with many wide peaks that may move severely. To examine the effects of the fitness landscape on the relative performance of these two algorithms (DEMg and SOS) parameters controlling the fitness landscape were varied. Results for the standard evolutionary algorithm

Method	Average error	Method	Average error
DEMgrdf	6.2825	DEMcrdf	6.2707
DEMgrd	6.2194	DEMcrd	6.2355
DEMgr	6.2087	DEMcrf	6.2056
DEMgrf	6.1914	DEMcr	6.1566
DEMgdf	5.9171	DEMcd	5.9670
DEMgf	5.8312	DEMcdf	5.9650
DEMg	5.6768	DEMc	5.9584
DEMgd	5.6628	DEMcf	5.9160

Table 7.9: Average offline error values for density-estimate memories including fitness in the environmental models on the default Moving Peaks problem

Height severity	SEA	DEMg	SOS	DEMg–SOS
7.0	13.4419	4.0079	3.2568	(-)
14.0	16.4488	5.0198	5.8330	(+)
21.0	18.3717	5.6768	7.6966	(+)
28.0	19.6253	6.0021	8.6015	(+)

Table 7.10: Average offline error values when varying height severity

(SEA) are also included as a reference. As mentioned, four parameters were considered: the frequency of changes, the severity of changes in height, the number of peaks, and the maximum value for peak width. In each table of results, the default value of the parameter is shown in bold. If the comparison between DEMg and SOS is significant, it is marked with (+) if DEMg is better or (-) if SOS is better.

In Table 7.10, the height severity was varied. Though all methods performed best with less discontinuous changes (lower height severity), the performance of density-estimate memory degrades more gradually than the other methods as changes become more severe. Density-estimate memory creates a long-term model of good solutions in the search space, and is less dependent on the new optimum having been in a high fitness area immediately before the change occurred. In cases where changes are more gradual, self-organizing scouts outperformed density-estimate memory, as the scout populations can refine good solutions for peaks that may potentially become the new global optimum.

In Table 7.11, the peak width was varied. The peak width, along with the number of peaks, defines the density of good solutions in the search space. The smaller the peak width parameter, the wider the peak can become. Density-estimate memory performed best on wider peaks compared to self-organizing scouts. As peaks become narrow, the performance of density-estimate memory degrades, and self-organizing scouts performs best. This makes sense, as self-organizing scouts is well designed for sparser spaces, sacrificing some search ability to maintain scout populations on

	Peak width	SEA	DEMg	SOS	DEMg–SOS
	2.0	15.1876	4.1358	6.4885	(+)
wide	3.0	16.9322	4.9558	7.1249	(+)
↑	**4.0**	18.3717	5.6768	7.6966	(+)
↓	6.0	20.5322	6.7438	7.8582	(+)
narrow	8.0	22.5249	7.6800	7.7212	
	12.0	25.0266	9.1744	7.8629	(-)

Table 7.11: Average offline error values when varying peak width

Frequency	SEA	DEMg	SOS	DEMg–SOS
1000	24.1577	13.7914	12.0930	(-)
3000	19.6700	7.0613	8.6617	(+)
5000	18.3717	5.6768	7.6966	(+)

Table 7.12: Average offline error values when varying change frequency

more sparsely distributed peaks.

In Table 7.12, the change frequency was varied. The more frequently changes occur, the more difficult the problem, leading to higher offline error. Density-estimate memory was comparable to self-organizing scouts for all frequencies, though at lower frequencies, self-organizing scouts began to dominate. Density-estimate memory relies on the underlying evolutionary algorithm to produce good solutions, and the quality of solutions degrades as frequency of changes increases. Self-organizing scouts, on the other hand, is designed to refine solutions within scout populations, so it may be able to improve on solutions that density-estimate memory cannot. For most change frequencies, density-estimate memory performs better.

In Table 7.13, the number of peaks in the search space were varied. For a small number of peaks, self-organizing scouts performed best, but as the number of peaks increases, both the standard evolutionary algorithm and density-estimate memory improved while self-organizing scouts got worse. Self-organizing scouts is designed to track a smaller number of peaks through scout populations. When the interesting peaks are small in number, this is a good strategy. However, when many peaks can produce interesting solutions, the models in density-estimate memory are designed to

Peaks	SEA	DEMg	SOS	DEMg–SOS
10	23.1275	5.1124	2.6900	(-)
30	20.5928	6.0417	5.7060	(-)
60	18.3717	5.6768	7.6966	(+)
90	17.0958	5.3890	7.9876	(+)

Table 7.13: Average offline error values when varying the number of peaks

Peaks	Width	SEA	DEMg	SOS	DEMg-SOS
30	4.0	20.4133	6.0337	5.8371	(-)
60	4.0	18.3717	5.6768	7.6966	(+)
90	4.0	17.0587	5.3815	8.0069	(+)
30	8.0	25.9858	8.7797	6.4518	(-)
60	8.0	22.3916	7.7148	7.7330	
90	8.0	20.3070	7.0485	8.9135	(+)
30	12.0	28.2473	10.8441	7.2463	(-)
60	12.0	25.0725	9.1747	7.7781	(-)
90	12.0	22.5908	8.3026	8.5474	

Table 7.14: Average offline error values when varying both peak width and number of peaks

	Peak width		
Peaks	4.0	8.0	12.0
30	-0.1966	-2.3279	-3.5978
60	2.0198	0.0182	-1.3966
90	2.6254	1.8650	0.2448

Table 7.15: Offline error value difference between self-organizing scouts and Gaussian density-estimate memory when varying peak width and number of peaks

include as many peaks as are in the search space without adding the overhead of tracking each peak individually.

7.5.2 Examining the effects of varying multiple parameters

In Table 7.14, the peak width and number of peaks were both varied to examine the effects of changing the density of solutions in the search space. A sparser search space would have fewer, narrower peaks (the minimum width parameter is inversely proportional to the width of a peak), while a denser space would have many wide peaks. Table 7.15 shows a direct comparison between self-organizing scouts and density-estimate memory. Positive values indicate parameter settings of the problem where density-estimate memory was better; negative values indicate areas where self-organizing scouts was better. The results show that the denser the space (the lower left corner is the densest), the better density-estimate memory performed relative to self-organizing scouts. Along the diagonal of Table 7.15, the difference between self-organizing scouts and density-estimate memory is low, and the confidence values are also low. Of the results on the diagonal, only the case with 30 peaks and a peak width of 4.0 produced a statistically significant result, and the confidence was much lower than the results off the diagonal.

7.6 Summary

Density-estimate memory was compared to a variety of evolutionary algorithm techniques on a version of the Moving Peaks problem with very severe changes in the search space and a large number of wide peaks. In this setting, density-estimate memory outperformed all other methods, including the state-of-the-art self-organizing scouts method. Density-estimate memories with Gaussian clustering outperformed those using the simpler Euclidean clustering. While density-estimate memories perform well on this problem, they do require diversity measures such as a multi-population memory/search model.

While it showed a slight improvement in average offline error, the informed diversity measure is not significantly better than the normal uninformed diversity included in the memory/search model. Periodic reclustering using a k-means algorithm made density-estimate memory performance worse on this problem. Including the fitness of a solution in the environmental model within the memory also did not help performance.

When parameters controlling the search space were varied, density-estimate memory continued to outperform self-organizing scouts as long as changes continued to be severe and the search space was composed of many wide peaks. By varying the peak width and number of peaks, both of which control the density of good solutions in the search space, a clear crossover point was shown between areas where density-estimate memory performs best and areas where self-organizing scouts performs best. Based on these results, density-estimate memory seems best suited for problems with a high density of good solutions and severe changes in the environment.

Considering the weak environmental information available in this problem, density-estimate memory performed very well. In a problem where the environmental information available was both stronger and decoupled from the solution, it is reasonable to assume that fitness differences between methods might be quite different. In particular, one would expect density-estimate memory methods to be even stronger, particularly those using informed diversity.

Chapter 8

Adaptive traffic signal control

One real-world problem with dynamic environments is traffic signal control. Controlling traffic congestion is not a new problem. In ancient Rome, congestion was bad enough that wagons were banned from the roads at certain times of day [47]. A recent study on 439 urban areas in the United States [88] concluded that in 2009, urban drivers in the United States traveled 4.8 billion more hours and purchased an additional 3.9 billion gallons of fuel due to congestion, for a total congestion cost of 115 billion dollars. The study also suggests that the problem is getting better, not worse. Many solutions, including increased public transportation and newer, wider roads, may help alleviate the problem. Improving the flow of vehicles through intersections offers enormous opportunities to reduce congestion, as poorly performing intersections may propagate delays into other areas of a traffic network.

There are many approaches to managing the movement of vehicles with conflicting paths through an intersection. Traffic signals, stop signs, roundabouts (also known as traffic circles or rotaries), and grade-separated interchanges all exist in many traffic networks. Most urban intersections use traffic signals to manage conflicting traffic flows. Typically, traffic signals operate on schedules that have been determined using historical data about the vehicle flows. Since vehicle flows change, traffic lights are typically reprogrammed every few years. One solution for reducing congestion that has become increasingly feasible with faster and cheaper computing power is the adaptive control of traffic signals so that traffic signals can constantly respond to changing vehicle flows.

This chapter investigates the potential for memory to improve the performance of traffic signal control methods. An adaptive traffic signal control algorithm is presented, as well as a density-estimate memory system to help improve the responsiveness of the adaptive algorithm. These methods are evaluated on several different road networks, including a traffic model of 32 intersections in downtown Pittsburgh, Pennsylvania. The results show that density-estimate memory improves the adaptive algorithm enough to outperform the fixed timing plans actually in use on these 32 intersections.

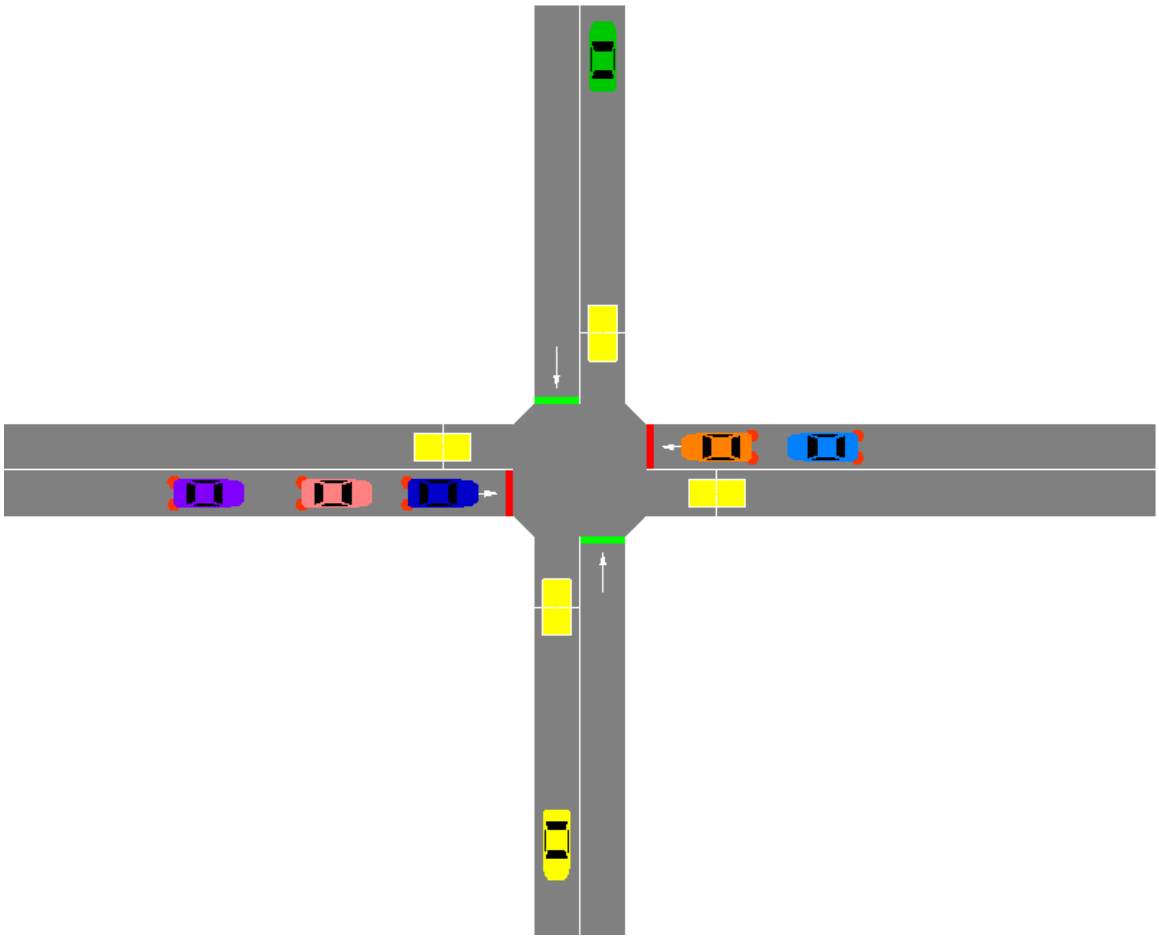

Figure 8.1: Simple intersection connecting four road segments with one lane on each input and output edge

8.1 Traffic signal control for urban road networks

The basic design of road networks is almost certainly familiar to the reader. Roads travel from one place to another, and where roads cross at-grade, intersections are created. Though highways may separate roads, only allowing vehicles to move between them via ramps, for most intersections the roads are at-grade and share the same space. In urban networks, an intersection is an area of a road network where vehicles with conflicting paths cross. Vehicles enter an intersection via entry and exit edges. Edges are made up of lanes, where a lane is wide enough for one vehicle to pass at a time. A two-way road segment connecting two intersections is a combination of two edges, one in each direction. Figure 8.1 shows a simple intersection connecting four road segments, each with an input and an output edge, with each edge containing a single lane. Figure 8.2 shows a larger intersection where many of the edges have multiple lanes.

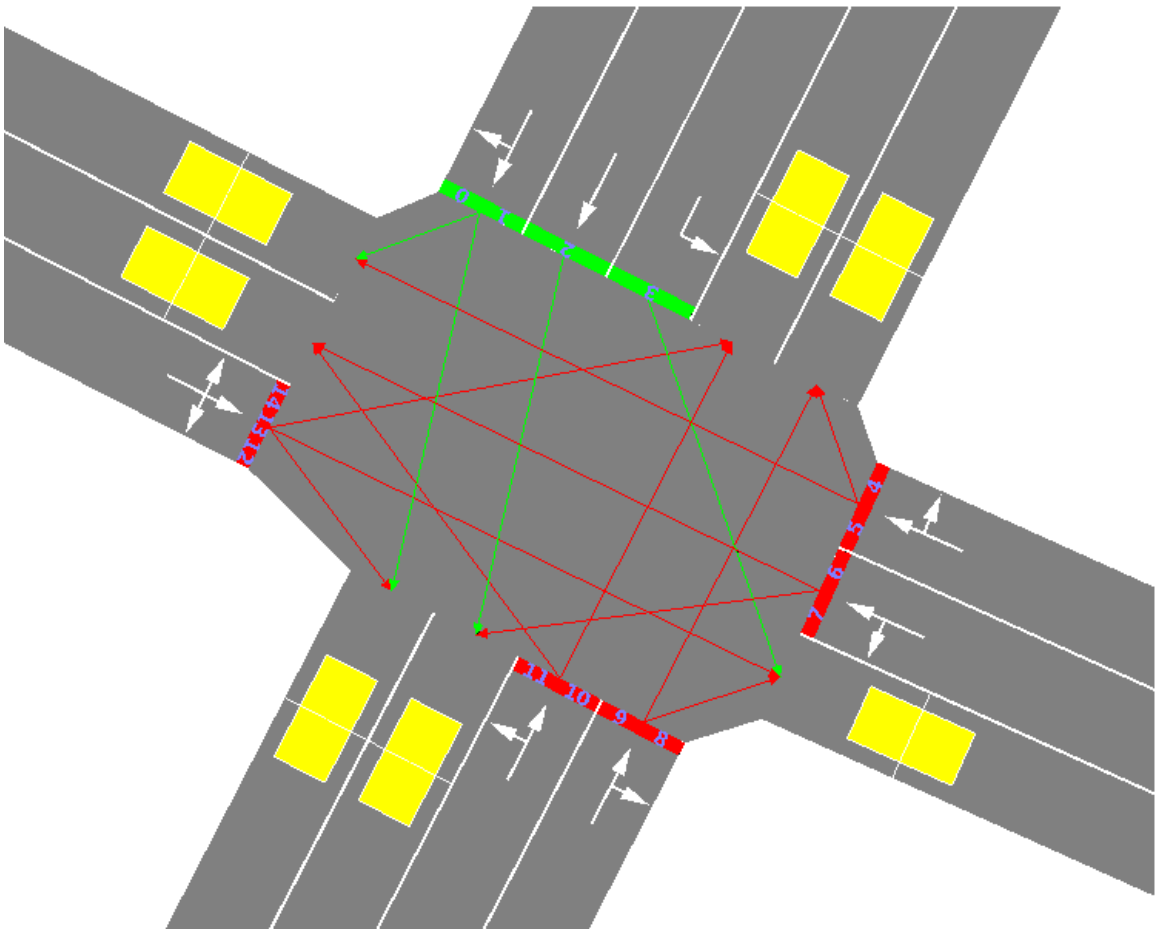

Figure 8.2: Complex intersection showing turning movements for each lane

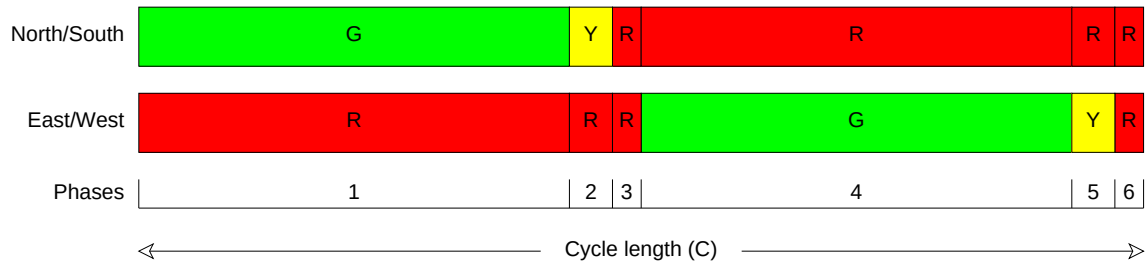

Figure 8.3: Example traffic signal phase diagram showing a cycle with six phases. The green phases (1 and 4) may have variable lengths, while the yellow phases (2 and 5) and all-red phases (3 and 6) have fixed lengths.

At an intersection, each lane has a set of allowed turning movements. In Figure 8.2, the southbound edge entering the intersection has three lanes. The leftmost lane may only turn left, the middle lane may only go straight, and the right lane may either go straight or turn right. The figure shows the allowed turning movements for each lane as well as the possible connections between lanes.

Since the paths allowed by different turning movements may conflict, priority rules must exist to prevent accidents. In this chapter, typical priority rules are in effect, e.g. turning cars must yield to those going straight. For many ways of managing intersections, such as stop signs and roundabouts, it is necessary for a vehicle to slow down or even stop every time it reaches an intersection in order to follow priority rules. Traffic signals, which are also known as traffic lights or stop lights, can potentially reduce or remove this need to stop by alternating between different groups of allowed turning movements. For example, in Figure 8.1, the green light allows traffic to move north and south. When the light changes, traffic flowing east and west will be allowed to proceed. Each period of time that allows one group of turning movements is called a *phase*. All the phases of a traffic signal combined together constitute the *cycle* of the signal. Figure 8.3 shows a traffic signal with six phases. In the first phase, traffic is allowed to flow north and south. In the second phase, traffic is warned, using a yellow light, to slow down since the direction of traffic is about to change. The third phase, called the all-red phase, is optional. In this phase, no traffic is allowed to flow. This allows time to clear the intersection of remaining vehicles. In the fourth phase, traffic may flow from the east and west. The fifth phase is a yellow phase, and the sixth phase is an all-red phase. Each phase i runs for a length of time P_i. The sum of all the phase times is the cycle time C, as shown in Equation 8.1.

$$C = \sum_{i=1}^{|P|} P_i \tag{8.1}$$

Once the last phase is complete, the first phase begins again. This sequence of phases can be seen as a schedule for the operation of a traffic signal.

While the sequence of phases is sufficient to control a single light, additional information is nec-

essary to properly coordinate one intersection with another. A car may leave one intersection, but arrive at another during the middle of a phase with a red light, delaying the car until the light turns green. By offsetting the start of the cycle of the first signal from the start of the second light's signal, this car might be able to drive through both intersections without stopping. In this chapter, the combination of a sequence of phases with an *offset* will be called a *timing plan*.

Timing plans for traffic signals may be set in many ways. Most commonly, traffic signals have prebuilt timing plans. A traffic signal might always have the same timing plan no matter the time of day, or a signal might instead have specific plans for different times of day, particularly morning and evening commuting times. These plans may be optimized using computer modeling software such as Synchro[1]. The United States Federal Highway Administration divides traffic signal systems into six categories in the Traffic Control Systems Handbook [41]: isolated intersection control, time base coordination, interconnected control, traffic adjusted control, traffic responsive control, and traffic adaptive control.

In isolated intersection control, signals are assumed to be sufficiently isolated from other signals such that coordination is not necessary, though the traffic signal may have different time-of-day plans. Time base coordinated signals use a common synchronized time to allow many intersections to coordinate switching between predefined time-of-day timing plans. Interconnected control adds networking to time base coordination to allow operators to switch between timing plans based on events or traffic conditions in addition to the normal time-of-day plans.

The last three signal control categories use information from traffic detectors to change the signal timing plan. A wide range of sensors can be used as detectors in traffic networks including inductive loops, cameras, magnetometers, infrared, and radar. The most common type of detector—the type assumed in this chapter—is the inductive loop: one or more turns of insulated wire embedded in the pavement of a road. When vehicles pass over the loop, the induction changes, allowing these detectors to sense a wide variety of information including the number of vehicles that pass the detector, the amount of time the detector is occupied by vehicles, and the estimated speed of vehicles passing the detectors. Inductive loops may be located on various parts of an edge as seen in Figure 8.4. Advance detectors are located ahead of the intersection on an edge to detect traffic flowing toward the intersection. Stop-line detectors are located where vehicles stop at the intersection to measure traffic flowing into the intersection. Exit detectors are located at the exits of an intersection to measure traffic flowing out of an intersection.

Traffic adjusted control senses current traffic conditions and switches between a set of predefined timing plans. This allows the signal to respond to traffic conditions as the conditions change, rather than switching between timing plans at the same times every day. Traffic adjusted control requires

[1] Synchro is commercial software for optimizing traffic signal timing plans (http://www.trafficware.com)

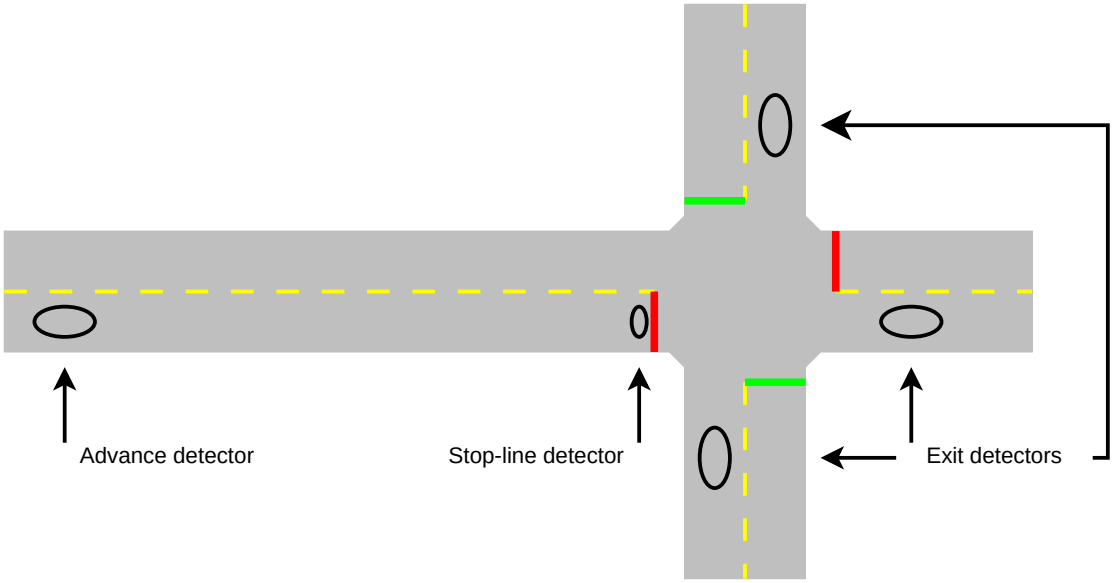

Figure 8.4: Detector locations relative to an intersection. Exit detectors are located near the intersection on exit lanes. Advance detectors are located on entry lanes far from the intersection. Stop-line detectors are located on entry lanes right before the intersection begins.

timing plans to be precomputed. If new traffic patterns occur, it is possible that none of the prebuilt timing plans will be suitable.

Unlike controllers in the first four categories of traffic signal control, traffic responsive controllers are able to generate new timing plans using detector data. Traffic responsive control continues to use a fixed cycle length, changing the offset and phase durations to respond to traffic conditions. The Split Cycle Offset Optimization Technique (SCOOT) [85] is an urban traffic control system developed in the United Kingdom in the 1980s. SCOOT incrementally optimizes timing plans based on an on-line model of vehicle queues. With installations in over 200 cities, SCOOT is used worldwide, though many installations are in the United Kingdom where the system was developed. The United States Federal Highway Administration initiated the development of ACS Lite [89] as an affordable traffic responsive system that did not require extensive infrastructure for its installation. ACS Lite is designed for arterial networks of 30 or fewer traffic signals. Traffic responsive control has also been a continuing research direction. Recently, Prothmann et al. [79] have developed an organic traffic control architecture that combines off-line optimization and on-line adaptation.

Traffic adaptive control dispenses with a fixed cycle length. Instead, detector data is used to forecast the arrival of individual vehicles at an intersection and then phase lengths are adapted proactively. Since traffic adaptive controllers do not have set cycle lengths, coordination using offsets is usually not possible. Instead, coordination is self-organizing based on the arrival of individual vehicles. The Sydney Coordinated Adaptive Traffic System [90] was one of the first traffic adaptive control

systems. As of March 2011, SCATS was in use in 146 cities in 24 countries. In the 1990s, the United States Federal Highway Administration funded the development of several traffic adaptive controllers [30] including the RHODES hierarchical control system [67]. These systems performed well in field trials, but could be expensive and complex to deploy. More recently, Lammer and Helbing [59] developed a self-organizing system that anticipates queues based on future arrivals of vehicles. Xie et al. [103, 102] developed an extension to Lammer and Helbing's work that accounts for vehicles not part of anticipated queues when controlling traffic signals.

Despite the nomenclature, both traffic responsive control and traffic adaptive control are often referred to as being adaptive. The key distinction between the two types of control is how coordination between signals is achieved. Traffic responsive systems adapt the split between phases while continuing to use set cycle lengths that are offset from other traffic signals. Traffic adaptive systems formulate the problem differently, dispensing with the notion of timing plans and instead adapting phases individually based on the predicted arrival of vehicles. In this chapter, the use of the term adaptive does not imply traffic adaptive control, only that the control system adapts in real-time based on detector data.

8.2 A traffic-responsive learning algorithm for traffic signal control

Density-estimate memory requires a learning algorithm to produce good solutions that can be aggregated in memory. While density-estimate memory is capable of making large jumps in the solution space of a problem, the underlying algorithm must be capable of refining the solution drawn from the memory. As the work in Chapter 6 shows, combining density-estimate memory with a reinforcement learning algorithm can improve solution quality. In this section, a traffic-responsive learning algorithm for controlling traffic signals is proposed as a baseline for exploring this hypothesis.

As mentioned above, traffic responsive systems change the timing plan of a traffic signal based on current traffic conditions. The balanced phase utilization (BPU) algorithm is a distributed learning algorithm that changes the timing plan for a traffic signal in small increments over time. The algorithm can be divided into several steps: phase balancing, coordination between traffic signals, offset calculation, and calculation of the new timing plan.

The cycle for each intersection begins with a green phase and ends with an all-red phase. During the last phase of a cycle, the BPU algorithm runs to determine the parameters of the timing plan for the next cycle. The offset of the signal is adjusted by slight changes in the cycle time as necessary.

8.2.1 Definitions

The signal timing plan is composed of the cycle time, phase lengths, and offset time. The BPU algorithm assumes a common cycle length C so that a traffic signal may coordinate with any of its neighboring signals. Each phase n has a minimum duration $\underline{P_n}$ and a variable split V_n. There are two types of phases: controllable and fixed. Controllable phases may vary in length as long as $P_n \geq \underline{P_n}$. Fixed phases may not vary, so $P_n = \underline{P_n}$ and $V_n = 0$. For a typical traffic signal, the green phases are controllable, while yellow, all-red, and pedestrian phases are fixed in length. The free cycle time is defined as

$$C_{free} = C - \sum_{n=1}^{|P|} \underline{P_n} \tag{8.2}$$

This time is allocated to controllable phases with

$$P_n = \underline{P_n} + V_n \cdot C_{free} \cdot controllable(n) \tag{8.3}$$

where

$$controllable(n) = \begin{cases} 1 & \text{if phase } n \text{ is controllable} \\ 0 & \text{otherwise} \end{cases} \tag{8.4}$$

$$0 \leq V_n \leq 1 \tag{8.5}$$

$$\sum_{n=1}^{|P|} V_n = 1.0 \tag{8.6}$$

For example, a traffic signal with a timing plan like the one in Figure 8.3 might have $\mathbf{V} = [0.5, 0, 0, 0.5, 0, 0]$, giving half the green time to north-south traffic (phase 1) and half the green time to east-west traffic (phase 4).

8.2.2 Phase utilization

A metric called *phase utilization* is used throughout the BPU algorithm. This metric estimates how efficiently each phase is being used. If a phase is much longer than is necessary to serve all the vehicles that pass the intersection during that phase, its utilization will be low. If a phase is over-saturated and a queue begins to build, the utilization will be high. The concept of phase utilization, defined in a similar way, is used by ACS Lite [89] for phase balancing. ACS Lite assumes the existence of stop-line detectors, allowing the utilization to be measured for each edge during each phase. Based on the location of inductive loop detectors in downtown Pittsburgh, these experiments assume stop-line detectors are not available, only exit detectors. This forces utilization to be measured for all edges during a phase.

To calculate the phase utilization, data from the exit detectors at the intersection i are collected in a rolling window of length ω. The phase utilization ρ_n for phase n is defined as

$$\rho_n = \frac{O_n}{G_n - t_{loss}} \tag{8.7}$$

where O_n is the occupancy time during the time window, G_n is the elapsed time of phase n during the time window, and t_{loss} is a constant loss time allowed for the beginning of each phase. The occupancy time is calculated as

$$O_n = \sum_{\tau=0}^{\omega} occ_i(t-\tau) \cdot ctrl_i(t-\tau, n) \tag{8.8}$$

where $occ_i(t)$ and $ctrl_i(t,n)$ are defined as

$$occ_i(t) = \begin{cases} 1 & \text{if any exit sensor at intersection } i \text{ is occupied by a vehicle at time } t \\ 0 & \text{otherwise} \end{cases} \tag{8.9}$$

$$ctrl_i(t,n) = \begin{cases} 1 & \text{if phase } n \text{ controls the signal at intersection } i \text{ at time } t \\ 0 & \text{otherwise} \end{cases} \tag{8.10}$$

The elapsed time of phase n during the time window is

$$G_n = \sum_{\tau=0}^{\omega} ctrl_i(t-\tau, n) \tag{8.11}$$

While traffic may flow in several directions during a given phase, as long as one direction is flowing efficiently the phase utilization is high. This prevents bias toward edges with more lanes, temporary delays when turning traffic blocks a lane while waiting for cross traffic to pass, and other events that don't detract from the efficiency of the phase.

8.2.3 Phase balancing

In the first step of the BPU algorithm, the phase times are balanced so that vehicles are properly served. The goal of the algorithm is to allocate green time such that all phases of the light are equally efficient. The vector of phase utilizations ρ is used to calculate the phase change vector Θ, which is used to update the variable split vector $\mathbf{V}$. The phase change vector has a sum of zero, since if any phase is lengthened, another must be shortened. First, the utilization difference vector—measuring the difference from the mean utilization for each phase—is calculated in Equation 8.12.

$$\Delta\rho_n = \left(\rho_n - \frac{\sum_{k=1}^{|\rho|} \rho_k}{\sum_{k=1}^{|\rho|} controllable(k)} \right) controllable(n) \tag{8.12}$$

where $controllable(n)$ is defined in Equation 8.4. The utilization difference for a controllable phase is the difference between ρ_n—the utilization of phase n—and the average utilization of the controllable phases. Fixed phases, such as yellow and all-red phases, are not controllable, and will always have $\rho_n = 0$ and $\Delta\rho_n = 0$. The utilization difference vector is used to calculate the phase change vector

$$\Theta_n = \frac{\Delta\rho_n}{|\max \Delta p|} \tag{8.13}$$

where $|\max \Delta p|$, the size of the largest difference in phase utilization, is used to normalize each value in the phase utilization vector. The phase change vector Θ is used to update the values for the variable splits. The phase change vector is scaled by a discount factor γ to regulate how much the timing plan may change from one cycle to the next.

$$\mathbf{V} \leftarrow \mathbf{V} + \gamma\Theta \tag{8.14}$$

If necessary, the variable split vector $\mathbf{V}$ is normalized to satisfy the requirements of Equations 8.5 and 8.6.

8.2.4 Coordination between traffic signals

The first step in calculating the offsets between traffic lights is to determine how signals should be coordinated. The BPU algorithm performs a simple calculation at each intersection to determine which of its neighbors an intersection should coordinate with. The coordination method does not require cooperation or negotiation; an intersection simply chooses another intersection to follow and asks for the necessary information to be communicated.

The first step is to find the edge with the highest input flow during the time window ω

$$edge = \underset{e \in inputs_i}{\operatorname{argmax}} \left(\sum_{\tau=0}^{\omega} flow(t-\tau, e) \right) \tag{8.15}$$

where $inputs_i$ is the set of input edges to intersection i, t is the current time, and

$$flow(t,e) = \text{number of vehicles passing all detectors on edge } e \text{ during time step } t \tag{8.16}$$

The second step is to find the neighboring intersection connected to intersection i via $edge$.

$$j = neighbor_i(edge) \tag{8.17}$$

The third step is to find the synchronized phase of intersection i with the highest output flow during the time window

$$sp_i = \underset{p \in phases}{\operatorname{argmax}} \left(\sum_{\tau=0}^{\omega} \sum_{e \in outputs_i(edge)} flow(t-\tau, e) \cdot ctrl_i(t-\tau, p) \right) \tag{8.18}$$

where $outputs_j(edge)$ is the set of output edges from $edge$.

Finally, the fourth step is to find the synchronized phase of the neighboring intersection j with the highest output flow during the time window

$$sp_j = \underset{p \in phases}{\operatorname{argmax}} \left(\sum_{\tau=0}^{\omega} \sum_{e \in outputs_j(edge)} flow(t-\tau, e) \cdot ctrl_j(t-\tau, p) \right) \tag{8.19}$$

8.2.5 Offset calculation

The offset o_i of an intersection i when coordinated with an intersection j depends on:

- The phase delay $pd_i(sp_i)$ for the synchronized phase sp_i of intersection i
- The queue clearing time $q_i(sp_i)$ for the synchronized phase sp_i of intersection i
- The offset o_j of coordinated intersection j
- The phase delay $pd_j(sp_j)$ for the synchronized phase sp_j of the coordinated intersection j
- The travel time $d_{j,i}$ between coordinated intersection j and intersection i at the speed limit

The phase delay of the synchronized phase of intersection i is the time between the start of the cycle and the start of the phase: the sum of all phases preceding the synchronized phase.

$$pd(sp) = \sum_{k=1}^{sp-1} P_k \tag{8.20}$$

A very simple queue clearing time calculation can be made by estimating the number of cars in the queue during phase sp_i and applying the following formula

$$q_i(s) = \frac{l_v}{v_v} \frac{n_{queue}(sp_i)}{n_{lanes}(i)} \tag{8.21}$$

where l_v is an estimated vehicle length, v_v is an estimated vehicle velocity, $n_{queue}(sp_i)$ is the estimated number of cars in the queue for phase sp_i, and $n_{lanes}(sp_i)$ is the number of lanes that are active during phase sp_i.

The desired offset o_i^* is calculated as

$$o_i^* = (o_j + pd_j(sp_j) + d_{j,i} - pd_i(sp_i) - q_i(sp_i)) \bmod C \tag{8.22}$$

similar to the offset calculation in [79]. The actual offset at time t is

$$o_i(t) = t_{cycle} \bmod C \tag{8.23}$$

where t_{cycle} is the time the current cycle began.

8.2.6 Calculation of the new timing plan

Given the desired offset o_i^*, the actual offset $o_i(t)$, the ideal cycle time C^*, and the variable splits $\mathbf{V}$, the timing plan for the next cycle can be calculated. The offset difference Δo_i is the difference between the desired and current offsets.

$$\Delta o_i = o_i^* - o_i(t) \tag{8.24}$$

This offset difference along with the ideal cycle time is used to determine the cycle time for the next phase.

$$C = C^* + \Delta o_i \tag{8.25}$$

This new cycle time is used to determine the free cycle time; the amount of time allowed for splits between controllable phases.

$$C_{free} = C - \sum_i \underline{P_i} \tag{8.26}$$

Then, this free cycle time is divided up using the variable splits.

$$P_i = \underline{P_i} + V_i \cdot C_{free} \tag{8.27}$$

These phase times are used to control the traffic signal until the final phase of the cycle is reached. Once the final phase is reached, the BPU algorithm uses the new windowed detector data to compute the timing plan for the next cycle.

8.3 Density-estimate memory

The previous chapters have described several implementations of density-estimate memory. Adding density-estimate memory to an algorithm for controlling traffic signals is similar to the problem

described in Chapter 6. The memory is distributed since each intersection is controlled separately; each intersection builds and maintains its own memory. Since the BPU algorithm searches from a single point, rather than using a population, solutions can only be retrieved from the memory periodically to avoid disrupting the learning algorithm too often. A version of the incremental clustering density-estimate memory used in other chapters was tested on the traffic control problem, along with a density-estimate memory using Gaussian mixture models. Gaussian mixture models are one of many possible probabilistic models that can be used in density-estimate memories.

8.3.1 Structure

For the experiments in this chapter, the environmental information for an intersection is a vector of input traffic flows to that intersection and to all of its neighbors. Each flow is a windowed count over the previous ω seconds of the number of cars passing over advance detectors on an input edge to an intersection. An intersection with four input edges connected to four neighbors—where all neighbors also have four input edges each—has an environmental information vector with 20 traffic flows. The length of this vector varies for intersections with different geometries, particularly those on the outside of the modeled road network that do not have a neighboring intersection for every input edge. The control information is composed of a variable split vector, $\mathbf{V}$, and the tuple containing all information describing the offset coordination for the intersection i: the neighboring intersection j to be coordinated with and the synchronized phases for both intersections (j, sp_i, sp_j).

Incremental Gaussian clustering

In the incremental clustering version of density-estimate memory, points are added to clusters as those points are stored in memory. When a point is different enough from existing clusters that a new cluster should be created, two existing clusters are merged together so the number of clusters remains the same. Each cluster contains a Gaussian model of the environmental information, an aggregated variable split vector $\mathbf{V}$, and an aggregated offset tuple. The aggregated split vector is created by averaging the values of $\mathbf{V}$ for all points in the cluster and then scaling to satisfy Equations 8.5 and 8.6. The aggregated offset tuple is created by choosing the distinct offset tuple that occurs most frequently among points within the cluster.

Gaussian mixture model

The incremental clustering approach used by density-estimate memories throughout this thesis provides a strong model within the memory while keeping the overhead of maintaining the memory low. Since clusters are built incrementally, the complexity of adding a new point to the memory

depends on the number of clusters, not the total number of points. While incremental clustering performs very well for a variety of dynamic environments, sometimes this approach can lead to a large imbalance in the size of clusters. If most good solutions are concentrated in one area of the search space, that area may only be represented by one or two clusters, even if the real fitness landscape in the area is very complex. As the results from Chapter 7 show, reclustering doesn't necessarily help improve performance. Rather than using the same model of completely separate clusters, another option is the use a mixture model.

Mixture models are a common class of probabilistic techniques for representing unlabeled data that fall into different classes [66]. The applications of mixture models are widespread, but when used in density-estimate memory, a mixture model helps separate points stored in the memory by environmental information. Information from the model can then be used to aggregate control information. To create a mixture model, the model parameters are learned from the known points. The most common method for learning the model parameters is the expectation maximization algorithm [32]. In this chapter, expectation maximization is used to build a Gaussian mixture model within a density-estimate memory.

For the dynamic environments in Chapters 6 and 7, maintaining a low overhead for density-estimate memory kept the memory from slowing down the optimization or learning algorithm. For the traffic control problem, as defined in this chapter, the control information is only updated once per cycle, leaving plenty of computation time to build more complex models that might not be suitable for problems when building density-estimate models within the memory would reduce the amount of computation time available to the search process.

While a Gaussian mixture model of the environmental information for all points is built within the density-estimate memory, control information is still collected in clusters. Using the mixture model, each point is assigned to a best fitting cluster. Then, the control information is aggregated in the same way as for the incremental clustering density-estimate memory.

8.3.2 Storage

The memory is populated by periodically storing the current state of the traffic signal controller. Density-estimate memory relies on the underlying learning algorithm to produce good solutions that can be stored in the memory. In a good solution, the phase utilizations are well balanced. Algorithm 8.1 describes the computation of a relative efficiency vector for measuring the performance of the current state of the underlying BPU algorithm. First, the phase utilization vector ρ is computed as in Section 8.2.2. Each entry in ρ describes the phase utilization of a single phase of the intersection. Next, phases with utilizations higher than a threshold ϕ_{eff} are added to the efficiency vector. For some intersections, phases exist that see very little traffic and require only the minimum phase

Algorithm 8.1 Calculation of the relative efficiency vector for traffic signal control

$\rho =$ `phase utilization vector`
eff $= []$
`FOR` n `IN` ρ

    `IF` $\rho_n > \phi_{eff}$
        $eff_c = \rho_n$

`FOR` i `IN` **eff**

    $efficiency_i = \frac{length(\mathbf{eff})}{\sum \mathbf{eff}} \cdot eff_i$

duration. The efficiency of the current phase balancing depends only on those phases with enough traffic to require some of the free cycle time. After removing low traffic phases, the efficiency vector is normalized by the sum of the vector and then scaled by the length of the vector so that the average value is 1. Each entry in the vector measures how far the phase utilization varies from the mean. A perfectly balanced signal would result in all entries being equal to 1.

Algorithm 8.2 describes the memory insertion procedure for use by the density-estimate memory. The criteria for inserting a solution into the memory changes with time. As the time since the last storage to memory increases, the threshold required to store a new point becomes less constraining. To avoid storing too frequently to the memory, insertions must occur at least t_A apart. From t_A to t_B, variations in the relative efficiency vector must be less than the threshold ϕ_A. From t_B to t_C, variations in the relative efficiency vector must be less than the threshold ϕ_B, where $\phi_A < \phi_B$. From t_C to t_D, the largest variation in the relative efficiency vector must be smaller than the largest variation at the last storage time. Finally, if the time since the last storage exceeds t_D, a new point is stored regardless of the quality of the solution. Based on this algorithm, the memory may be updated as frequently as every t_A seconds, but no more infrequently than every t_D seconds.

8.3.3 Retrieval

A solution should be retrieved from the memory when the current state of the controller is far from the optimal solution and a sufficiently good match within the memory can be found. Solutions should not be retrieved from the memory too frequently, so a sigmoid function is used to create a probability of retrieving from the memory dependent on the time since the last retrieval

$$p_t = \frac{1}{1 + e^{-\frac{1}{S}(t-T)}} \tag{8.28}$$

Algorithm 8.2 Memory insertion for traffic signal control

$t =$ `time since last memory storage`
$\mathbf{eff}_0 =$ `efficiency at last storage time`
$\rho =$ `current phase utilization vector`
$\mathbf{eff} = efficiency(\rho)$ `using Algorithm 8.1`
`IF` $t > t_A$ `THEN`

```
    IF max eff < (1 + φ_A) AND min eff > (1 − φ_A) THEN
        STORE TO MEMORY
    ELSE IF t > t_B THEN
        IF max eff < (1 + φ_B) AND min eff > (1 − φ_B) THEN
            STORE TO MEMORY
        ELSE IF t > t_C THEN
            IF |eff − 1| < |eff_0 − 1| THEN
              STORE TO MEMORY
            ELSE IF t > t_D
              STORE TO MEMORY
```

where t is the time since a solution was last retrieved from the memory and S and T are scaling factors for the sigmoid function.

The probability that the current environment is a good fit for a cluster in the memory is measured by p_e. For the incremental clustering density-estimate memory, p_e is the probability that the current environment fits the model in the chosen entry. For the Gaussian mixture model density-estimate memory, p_e is the probability that the current environment fits the model multiplied by the posterior probability that the chosen entry is the best fit for the current environment. The cluster with the highest p_e value is always chosen.

When a change in demand occurs in a traffic network, it may take a long time for the demand to propagate throughout the network. Since memory is distributed by intersection, including the behavior of neighboring intersections can help encourage retrieval from the memory to propagate faster, allowing quicker reactions to changes in traffic demand. The probability of retrieving from the memory is computed as

$$p_r = \begin{cases} \sqrt{p_e \cdot p_t} & \text{if any neighbors have retrieved within } N \text{ seconds} \\ p_e \cdot p_t & otherwise \end{cases} \tag{8.29}$$

where intersections with neighbors that have retrieved recently have higher probabilities of retrieval. Algorithm 8.3 describes the memory retrieval decision process. If a random number p_{random} is less

Algorithm 8.3 Memory retrieval for traffic signal control

```
IF p_random < p_r THEN
    eff = efficiency(ρ) using Algorithm 8.1
    IF max eff > (1 + φ_R) OR min eff < (1 − φ_R) THEN
        RETRIEVE FROM MEMORY
```

than p_r and the relative efficiency vector varies within ϕ_R of the mean efficiency, then the control information from the cluster that best matches the current state is used to set $\mathbf{V}$ and the offset tuple for the intersection.

8.4 Experiments

All experiments were conducted using the SUMO (Simulation of Urban MObility)[2] traffic simulator. SUMO is a microscopic road traffic simulator designed to accurately model vehicle traffic on large road networks [58]. SUMO is open source software and is mainly developed at the Institute of Transportation Systems at the German Aerospace Center. SUMO has been used to create accurate models of traffic in several German cities, including Cologne and Nuremberg. SUMO includes a traffic control interface (TraCI) that allows the simulator to be controlled by an external program [100]. TraCI allows interaction with many areas of the simulation including interaction at the level of individual vehicles. For these experiments, only a few parts of TraCI are necessary: traffic signals within the simulation may be controlled and detector values may be read at each time step. All controllers were implemented using the Python Adaptive Traffic Signal Control Interface (PATSCI)[3] which was designed for these experiments. PATSCI implements the general controller model for traffic signals described in this chapter as well as all control algorithms to be compared.

Two metrics were used to evaluate the performance of a traffic signal control algorithm on a scenario: average speed and average wait time for all vehicles. For each vehicle, the average speed can be calculated as the total length of the route divided by the time necessary to traverse the route. The wait time is the number of seconds the vehicle had a speed less than 0.1 meters per second. Though less waiting tends to lead to higher speeds, these two metrics are not perfectly correlated. For example, poorly coordinated signals may require vehicles to stop at every intersection, increasing wait time. If congestion is low, the speed may still be higher than a situation with well coordinated signals but a lot of congestion. The goal is to maximize speed while minimizing wait time. While both metrics are important, speed is more important than wait time as a measure of performance.

[2] More information about SUMO, as well as the source code, are available from http://sumo.sourceforge.net

[3] The source code for PATSCI is available from http://www.ozone.ri.cmu.edu/projects/traffic/patsci/

For these experiments, a single set of parameters was chosen for the BPU algorithm. The discount factor used by BPU was $\gamma = 0.05$, the ideal cycle length was $C^* = 100$ seconds, the time window for data collection was $\omega = 300$ seconds (5 minutes), and the loss time was $t_{loss} = 5$ seconds. For density-estimate memory, 10 clusters were allowed and $\phi_{eff} = 0.1$. For Algorithm 8.2, $t_A = 15$ minutes, $t_B = 30$ minutes, $t_C = 45$ minutes, $t_D = 60$ minutes, $\phi_A = 0.1$, and $\phi_B = 0.25$. For Algorithm 8.3, $S = 300$ (5 minutes), $T = 1200$ (20 minutes), $N = 300$ (5 minutes), and $\phi_R = 0.1$.

Based on the location of inductive loop detectors in downtown Pittsburgh, these experiments assume the existence of exit detectors only. An exit detector for one intersection is used as an advance detector for the neighboring intersection. No stop-line detectors are used for these experiments, though including stop-line detectors can improve the accuracy of computing phase utilization. For intersections that border the edge of the network, the input edges have advance detectors that are assumed to be exit detectors for intersections not modeled in the network.

Two experiments were performed to examine the performance of the BPU algorithm with and without memory. The first experiment simulates five hours of traffic on a simple grid traffic network with six intersections. The BPU algorithm was compared to the combination of BPU and density-estimate memory. The second experiment simulates entire days of traffic with realistic traffic demands and turning ratios on a 32 intersection network modeled on downtown Pittsburgh, Pennsylvania. The real fixed timing plan used to control these traffic signals in Pittsburgh is compared to BPU as well as density-estimate memory combined with BPU.

8.4.1 Six intersection grid

The first experiment considers a simple traffic network of six intersections arranged in a grid. The topography of this network, shown in Figure 8.5, is modeled on six intersections in downtown Pittsburgh, Pennsylvania. Unlike the real network, each road in the simulation has only a single lane running in each direction. Fort Duquesne Boulevard and Penn Avenue run east and west, while Sixth Street, Seventh Street, and Ninth Street run north and south.

The vehicle routes for this experiment are not based on real traffic patterns. Instead, traffic flows are constructed artificially using turn probabilities and input flows to examine the performance of density-estimate memory as traffic changes over time. On average a vehicle at an intersection turns left 10% of the time, goes straight 70% of the time, and turns right 20% of the time. Each turning proportion is allowed to vary slightly, and the underlying turning distribution changes every 30 minutes; random noise is added to each default turning probability with $N(0, 0.05)$ to produce the turning proportions used for that period. The sum of turning probabilities for each edge always sums to one. Different traffic demands are modeled by changing the input flows for each of the 10 edges that lead into the grid network. Table 8.1 shows a variety of traffic flow patterns for the network

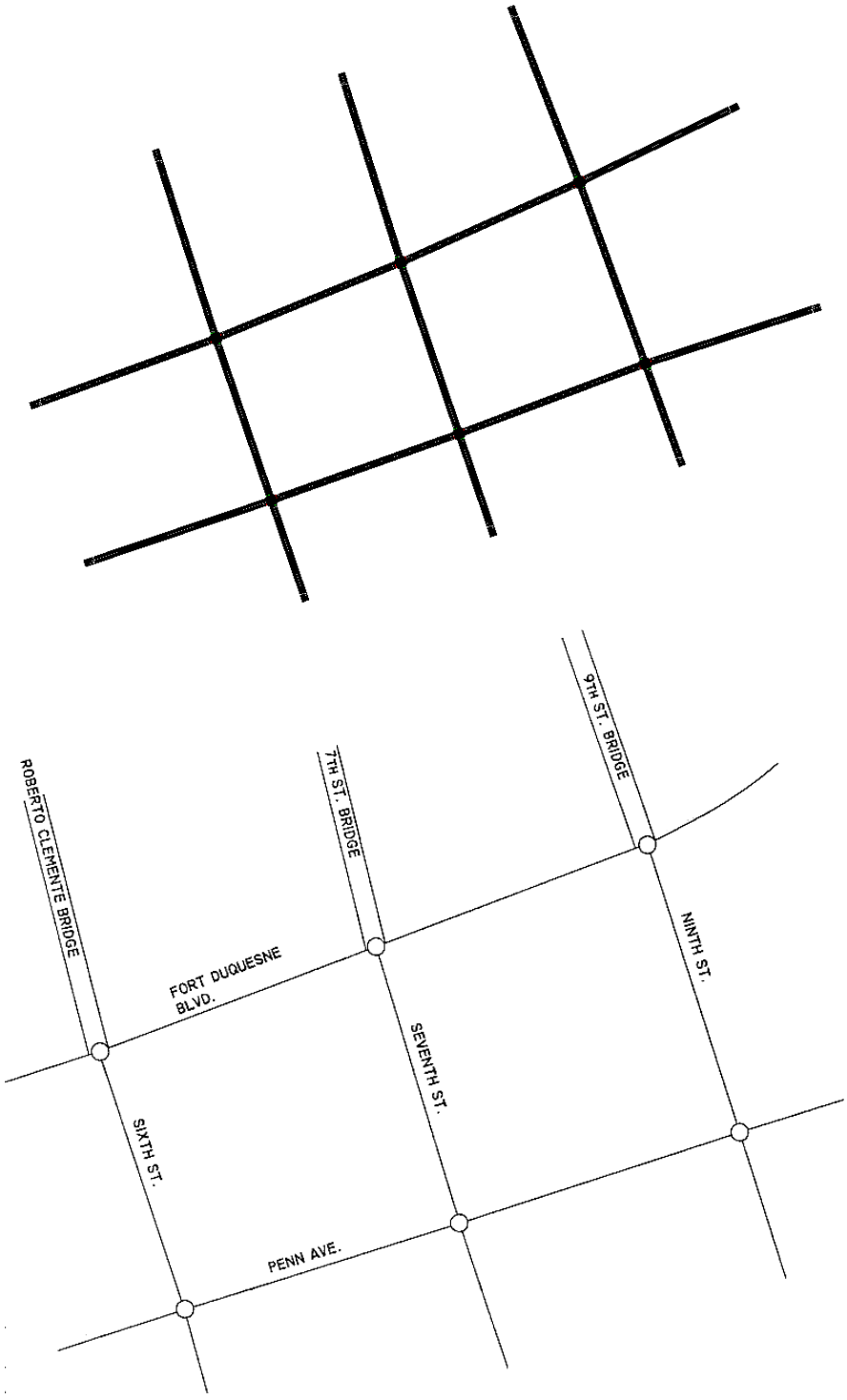

Figure 8.5: Six intersection grid traffic network in the SUMO simulator based on six intersections in downtown Pittsburgh, Pennsylvania. Fort Duquesne Boulevard and Penn Avenue run east and west, while Sixth Street, Seventh Street, and Ninth Street run north and south.

		Traffic flow by input (each unit is $1/36$ of the total flow)							
Street	Dir.	*east*	*west*	*north*	*south*	*duqE*	*duqW*	*ninthN*	*ninthS*
Ft. Duquesne	East	9	3	3	3	12	3	3	3
Ft. Duquesne	West	3	9	3	3	3	12	3	3
Penn	East	9	3	3	3	6	3	3	3
Penn	West	3	9	3	3	3	6	3	3
Sixth	North	2	2	6	2	2	2	4	2
Sixth	South	2	2	2	6	2	2	2	4
Seventh	North	2	2	6	2	2	2	4	2
Seventh	South	2	2	2	6	2	2	2	4
Ninth	North	2	2	6	2	2	2	10	2
Ninth	South	2	2	2	6	2	2	2	10

Table 8.1: Input flows for traffic patterns on the grid network

where an individual flow n produces $nF/36$ vehicles where F is the total number of vehicles in the scenario. The traffic patterns in Figure 8.1 include patterns where particular directions dominate traffic flow (*east*, *west*, *north*, *south*) and patterns where particular roads dominate traffic flow (*duqE*, *duqW*, *ninthN*, *ninthS*). SUMO uses the turning proportions and vehicle flows to produce routes for all vehicles that will be part of the simulation.

To evaluate the performance of the BPU algorithm with and without memory, a five hour scenario was constructed to simulate the effects of an accident that requires the temporary closure of a lane of traffic as well as changing traffic flows over the course of time. Figure 8.6 shows the changing traffic patterns in the scenario. The scenario is divided into five hour long periods: an initial hour to populate the network, three periods to model the effects of the lane closure, and a final hour to see the effects of congestion created by the lane closure. The scenario begins with an hour of traffic to populate the network with vehicles using the *east* traffic pattern. In Period 1, the dominant flow is east on Fort Duquesne Boulevard. Period 2 simulates the closure of Fort Duquesne Boulevard east of Ninth Street; all traffic intended for that lane is diverted to Penn Avenue. The input traffic flows remain the same, but the turning proportions at the two intersections on Ninth Street are changed. In Period 3, the dominant traffic flow changes to be south on Ninth Street, the closed lane is reopened, and the default turning proportions are used at all intersections. Finally, the last hour uses the *south* traffic pattern. The traffic demand is constant with 2480 vehicles arriving every hour.

The experiments for this scenario are run over 20 different instances for each control algorithm; each algorithm is tested over the same 20 instances. Vehicle routes for each instance are generated by SUMO using a different random seed. Since building an effective memory can take a long time, a memory was prebuilt for this experiment by alternating between different traffic patterns shown in Table 8.1. The turning proportions vary as in the experiment. The traffic flows seen in Period 2, when Fort Duquesne Boulevard is closed, are never shown to the memory as it is being built.

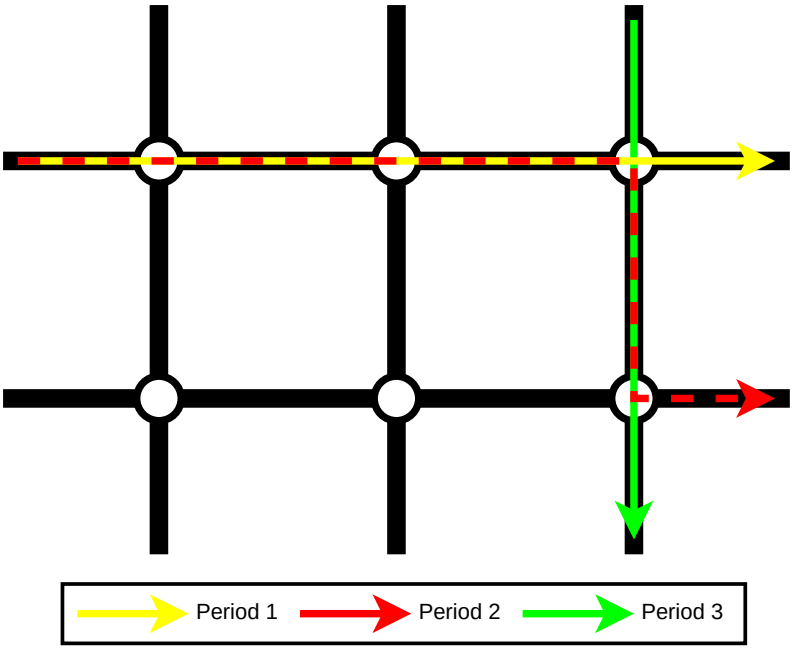

Figure 8.6: Varying demands for the six intersection grid traffic network. During the first period, the dominant traffic flow follows Fort Duquesne Boulevard eastbound. During the second period, the dominant traffic flow of traffic begins on Fort Duquesne Boulevard eastbound, turns right onto Ninth Street and then turns left onto Penn Avenue. During the third period, the dominant traffic flow follows Ninth Street southbound.

8.4.2 Downtown Pittsburgh

The second experiment simulates traffic on thirty-two intersections in downtown Pittsburgh, Pennsylvania using realistic traffic patterns. Traffic in and out of six parking garages is also included. Each simulation lasts for an entire day, and the traffic flows change to simulate differences in traffic at different times of day. The real traffic signal timing plans used at these intersections are compared to the BPU algorithm with and without memory.

The road network in SUMO for this experiment, shown in Figure 8.7, is bounded by Grant Street, Penn Avenue, and Fifth Avenue. The topology of the road network was adapted from OpenStreetMap[4]. This area of downtown Pittsburgh contains both grid traffic and high density corridors, such as Grant Street and Liberty Avenue. The traffic flows change both volume and direction over the course of the day. When measuring traffic volume for the purpose of creating the real fixed timing plans for these traffic signals, the city of Pittsburgh divides a day into several demand profiles: morning (AM), midday (MD), afternoon (PM), and off-peak. The off-peak traffic is further divided into profiles for before and after sporting events at the hockey arena, football stadium, or baseball stadium adjacent to downtown and off-peak traffic when no event occurs. For this experiment, only the non-event (NE) off-peak traffic is considered. Figures 8.8, 8.9, 8.10, and 8.11 show the traffic flows for the four time-of-day profiles used in this experiment. A scenario simulating an entire day combines these demand profiles as shown in Figure 8.12. The scenario begins with a non-event period to populate the network with vehicles. The next hour transitions from non-event to morning traffic demands. This allows the transition between traffic flows and turning proportions to be gradual, rather than very sudden. For the next two hours, the morning traffic profile generates vehicles. After an hour of transition from morning to midday traffic, the midday traffic profile generates traffic for five hours, with increased traffic flow during lunch. After the midday period, the demand profile transitions to the afternoon rush period for an hour. The afternoon rush lasts for two hours, and then transitions to the evening non-event period.

The traffic flows and turning proportions for each route profile are based on counts of real traffic in Pittsburgh. These counts, shown in Figures 8.8, 8.9, 8.10, and 8.11, are then converted to input flows and turning proportions. Like the smaller experiment, the underlying turning proportions also vary slightly within a period using a single route profile; every 30 minutes, random noise is added to each default turning probability with $N(0, 0.05)$ to produce the turning proportions used for that period. The sum of turning probabilities for each edge always sums to one. Due to limitations in the current version of SUMO that cause gridlock under very high traffic flows such as those that occur during the afternoon rush, all traffic flows for this experiment were reduced to 70% of the

[4]OpenStreetMap is a free editable street map of the world available at www.openstreetmap.org. The data from OpenStreetMap was adapted to correct connections between lanes, the number of lanes on each road, and to include parking garages.

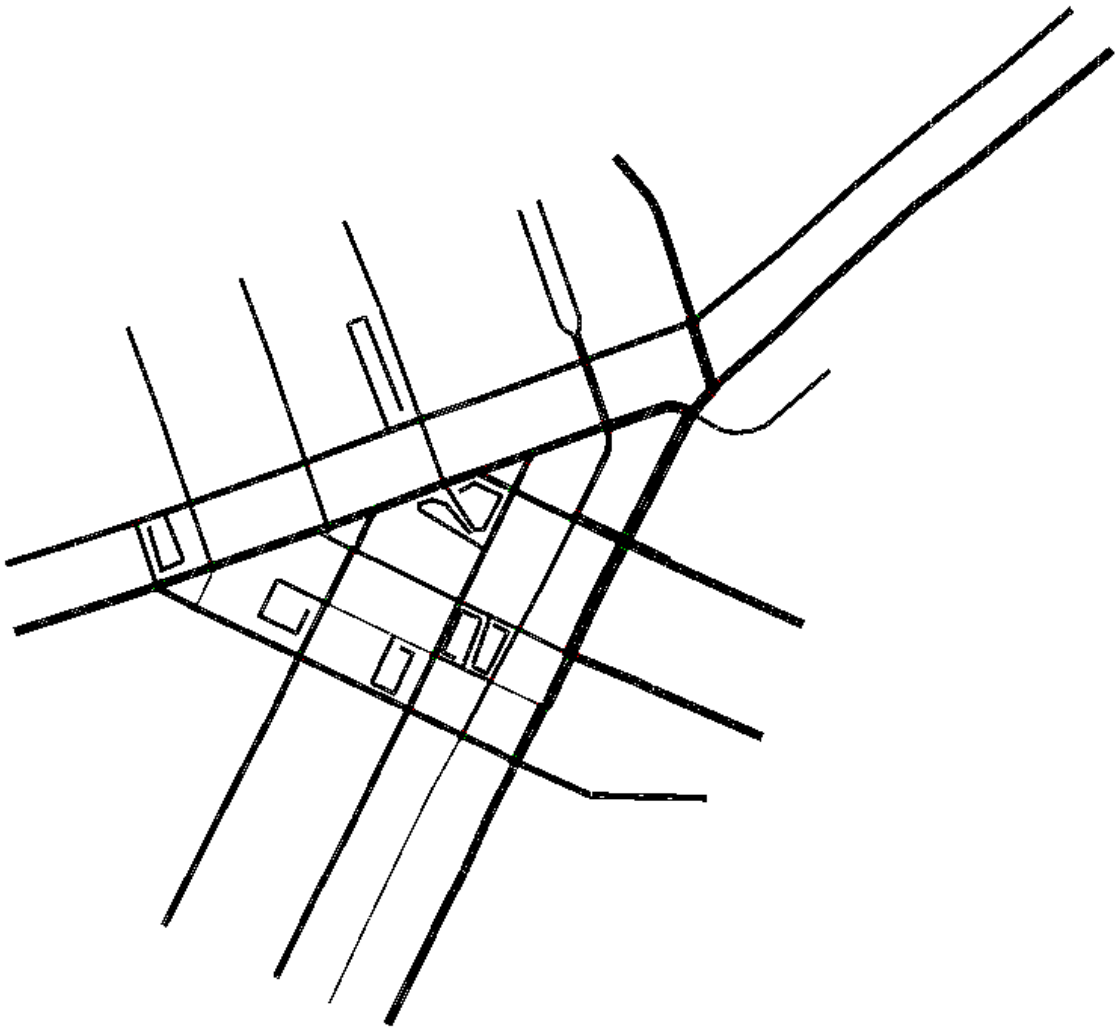

Figure 8.7: Downtown Pittsburgh road network in the SUMO simulator. The network models 32 intersections and 6 parking garages.

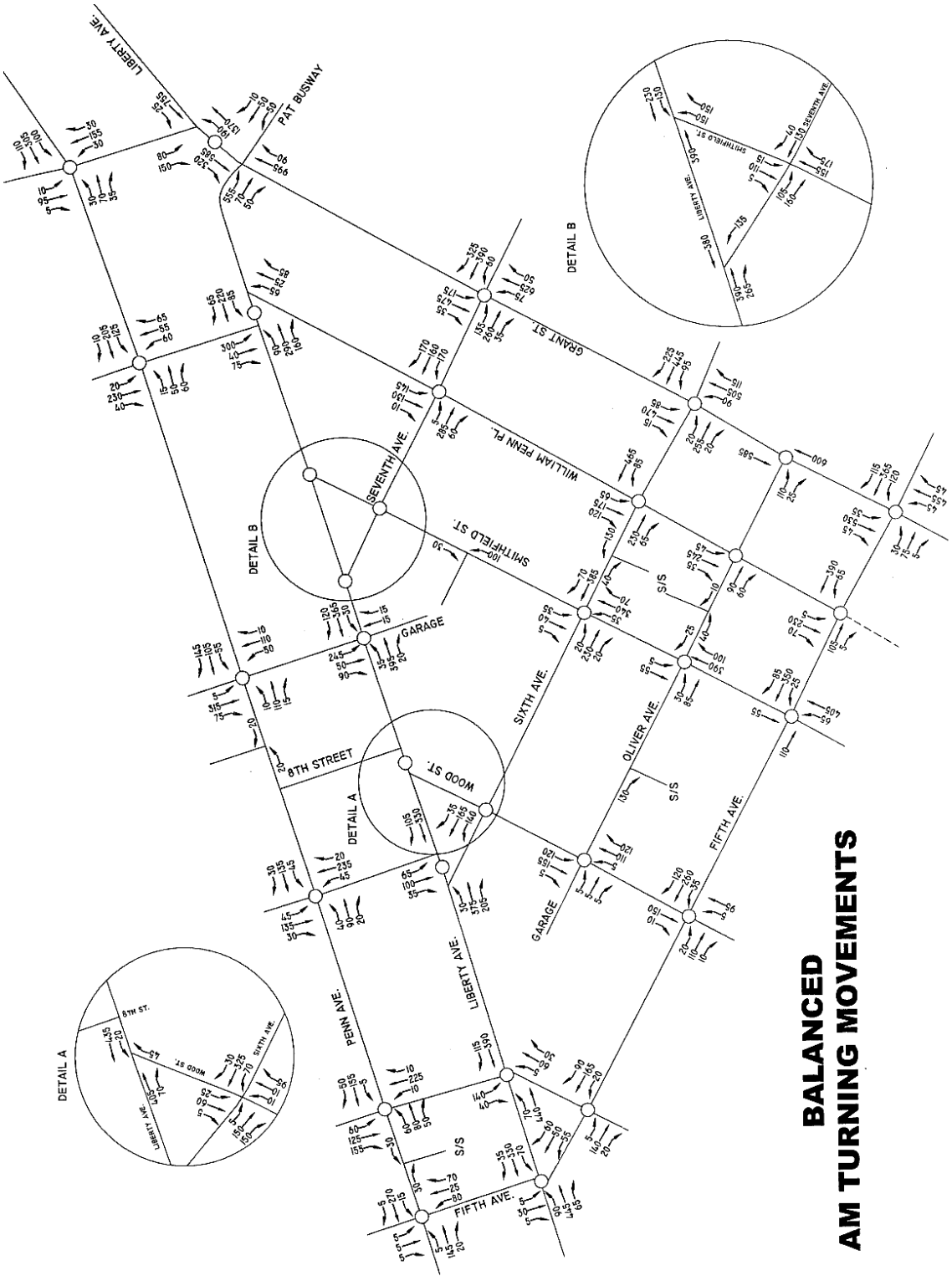

Figure 8.8: Traffic flows and turning movements for the downtown Pittsburgh road network during the morning rush time period

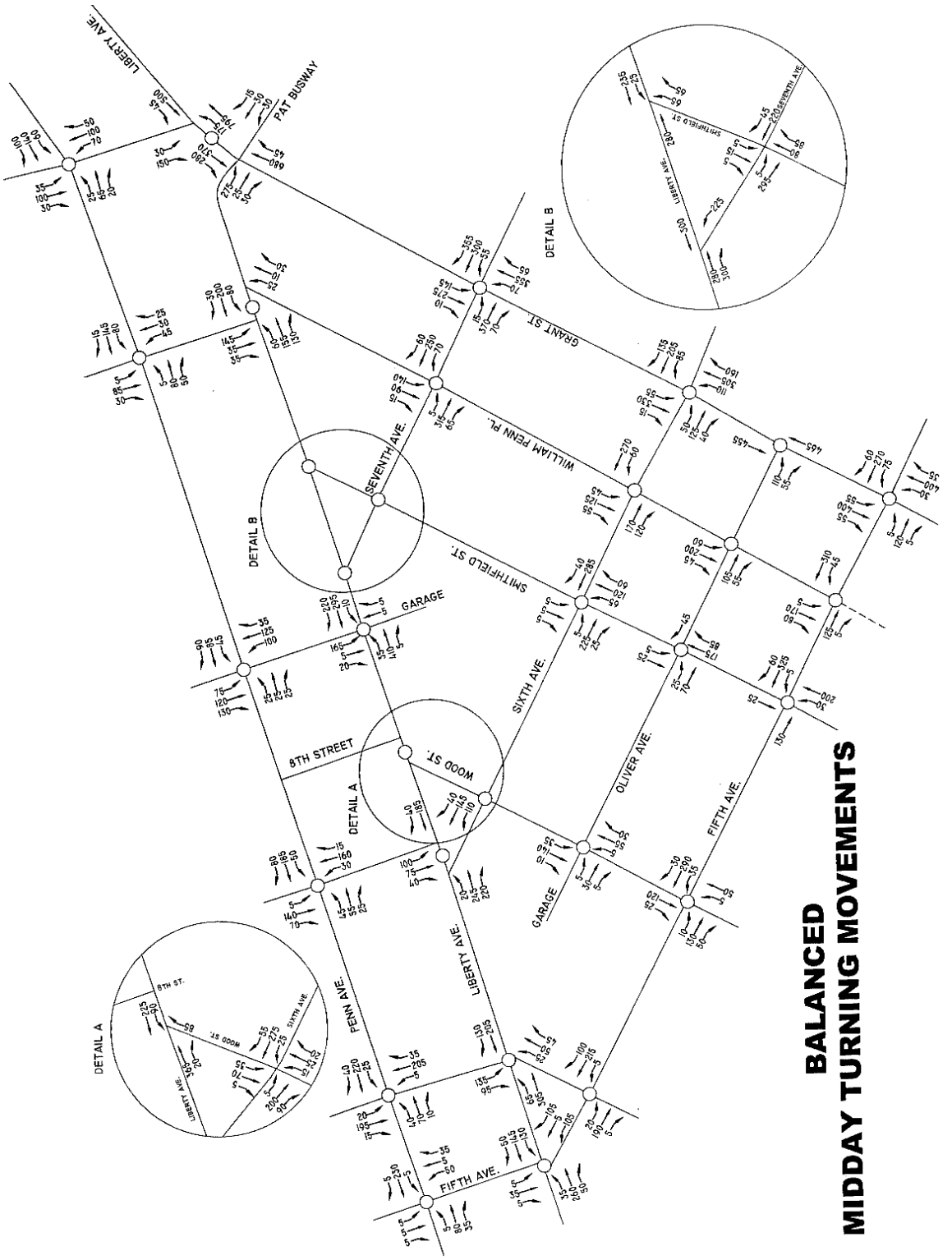

Figure 8.9: Traffic flows and turning movements for the downtown Pittsburgh road network during the midday time period

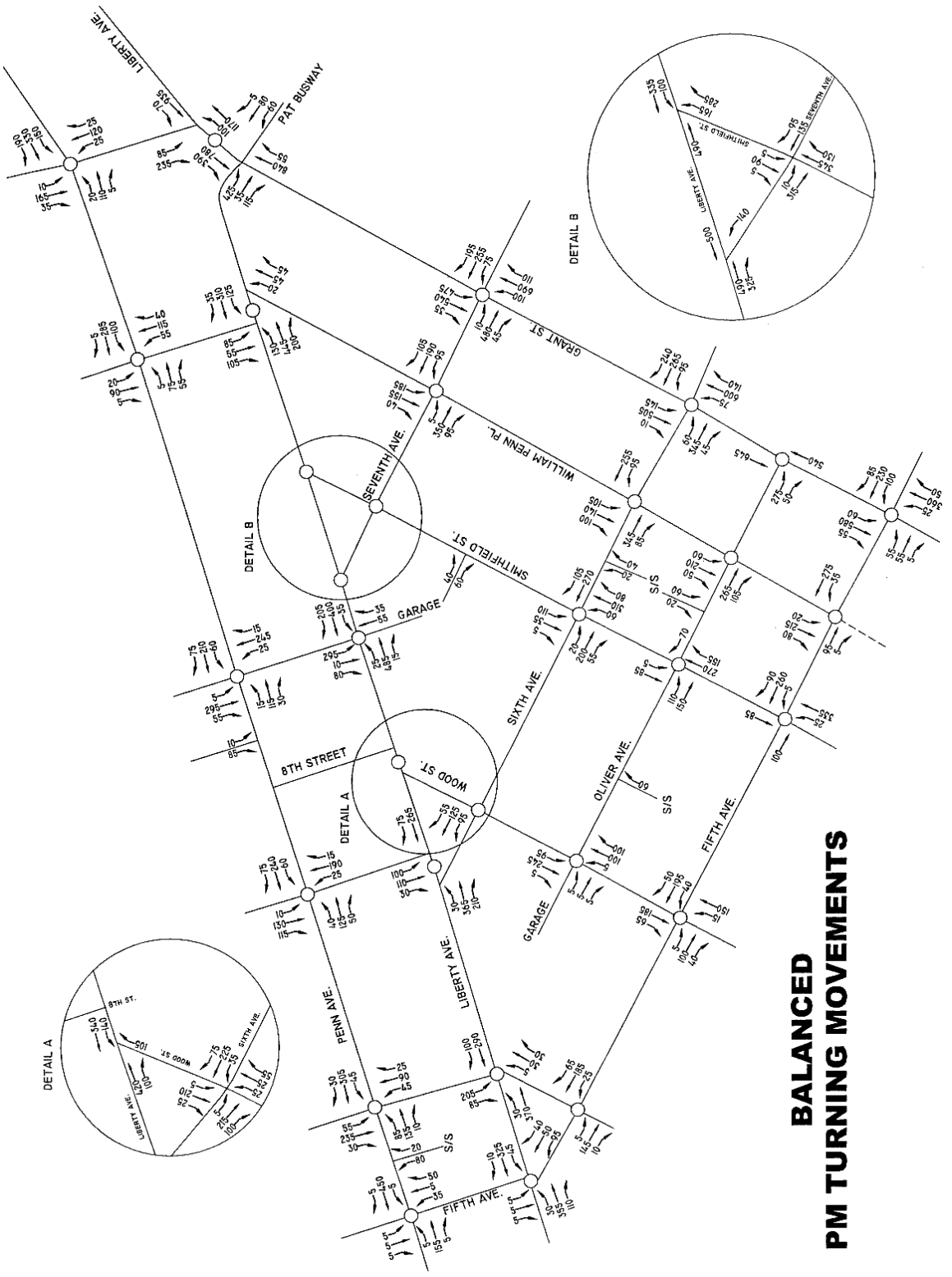

Figure 8.10: Traffic flows and turning movements for the downtown Pittsburgh road network during the afternoon rush time period

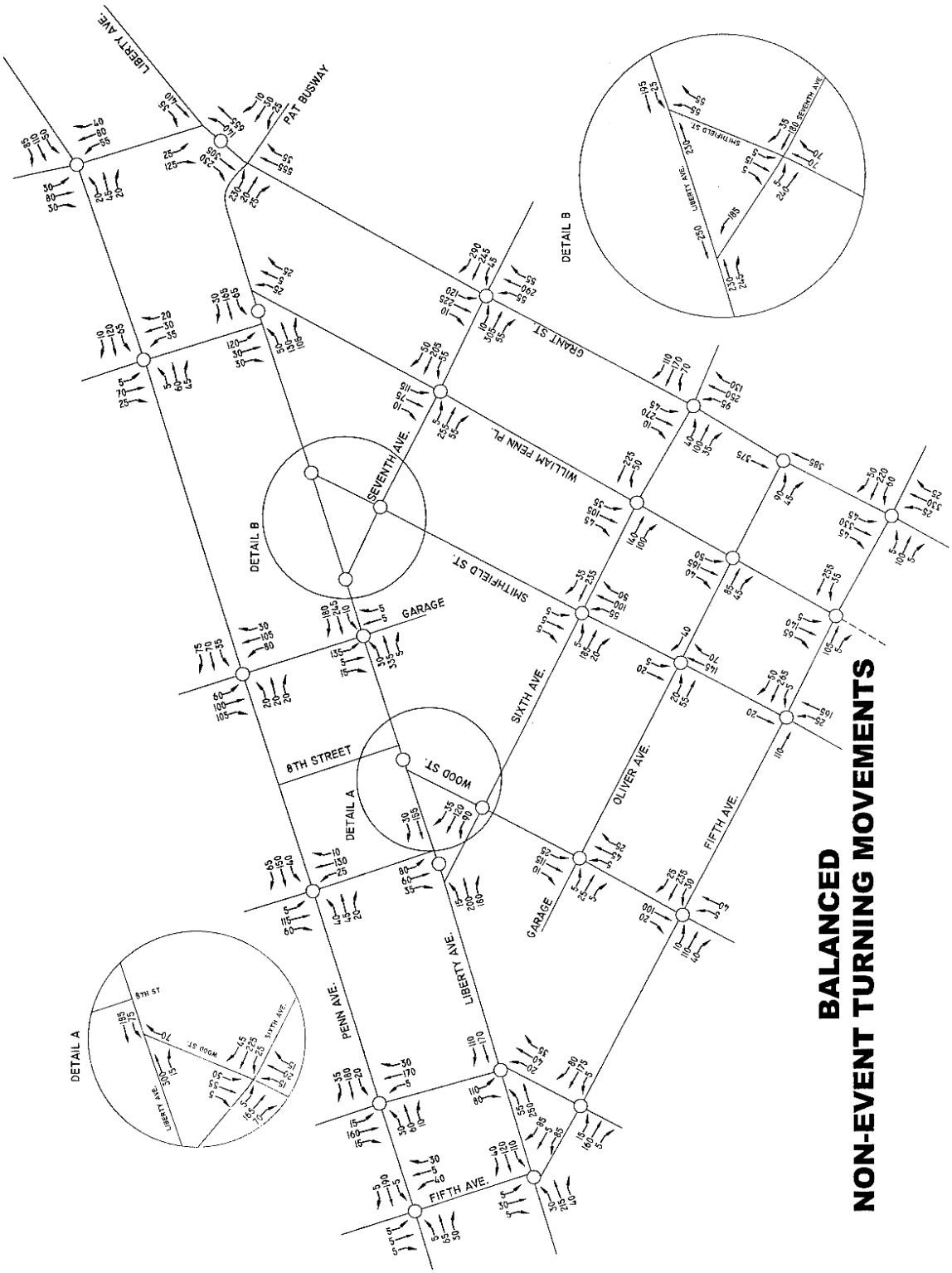

Figure 8.11: Traffic flows and turning movements for the downtown Pittsburgh road network during the non-event off-peak time period

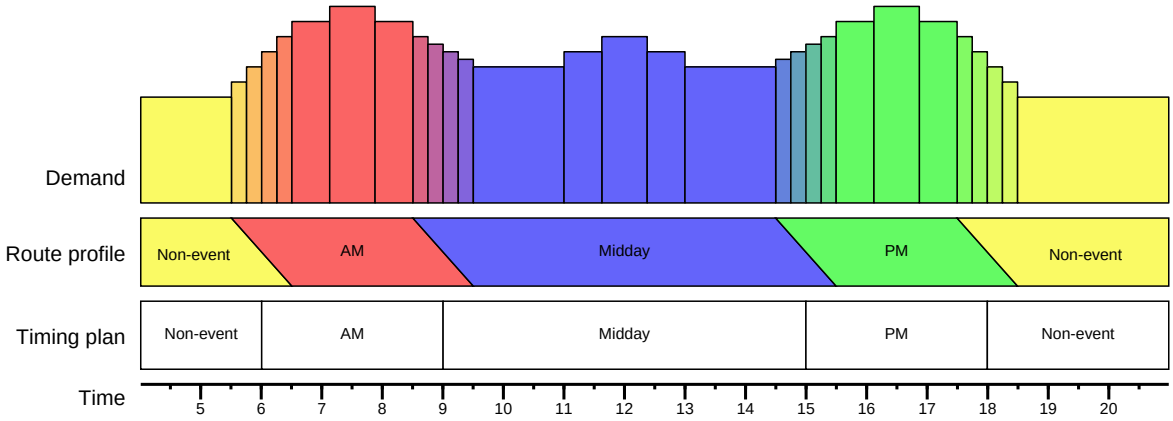

Figure 8.12: Traffic demand profile for a day-long scenario on the downtown Pittsburgh road network. Traffic begins using the non-event profile then transitions to the AM profile, the midday profile, the PM profile, and then back to the non-event profile. When a fixed timing plan is used, the time-of-day plan changes along with the demand profile.

true values. For demand profiles that do not lead to gridlock, such as the midday profile, relative performance of the algorithms were similar at 70% and 100% of true traffic flows.

The fixed timing plan used on the real intersections has four time-of-day plans corresponding to different demand profiles. The controller is time based: intersections switch between timing plans simultaneously using a synchronized clock. The non-event plan is used before 6 AM, the AM plan is used from 6 AM to 9 PM, the Midday plan is used from 9 AM to 3 PM, the PM plan is used from 3 PM to 6 PM, and the non-event plan is used after 6 PM. Since real traffic conditions don't always line up exactly with these times, the length of the first non-event period is used to vary how late or early the fixed plan changes. The length of the first non-event period for a particular day is uniformly random in $[30, 120]$ minutes. On average, the AM timing plan begins at the start of the transition from the non-event traffic profile to the morning traffic profile. At the earliest, the timing plan switches an hour before the transition begins. At the latest, the timing plan switches at the end of the transition to the AM profile.

The experiments for this scenario are repeated for 30 days of traffic for each control algorithm. Each day of traffic is different, and the same 30 days are used for all control algorithms. Since building an effective memory can take a long time, a memory was prebuilt for this experiment using DEM-GMM over 20 days of traffic. This memory was used by both DEM and DEM-GMM algorithms for the experiment.

Method	Speed	Wait time
BPU	3.6994	174.0090
DEM	3.7470	252.1891
DEM-GMM	3.7873	91.3353

Table 8.2: Average speed in meters per second and wait time in seconds for the six intersection grid scenario

Method 1	Method 2	Speed	Wait time
DEM	BPU	1.29 +	-44.93 -
DEM-GMM	BPU	2.38 +	47.51 +
DEM-GMM	DEM	1.08	63.78 +

Table 8.3: Percent improvement of method 1 over method 2 on the six intersection grid scenario (results that are statistically significant to 95% confidence are noted with a + or -)

8.5 Results

On both experiments, the use of density-estimate memory improved the performance of the BPU algorithm. The Gaussian mixture model variant of density-estimate memory performed as well or better than the incremental Gaussian clustering method for both problems. For the downtown Pittsburgh experiment, the use of memory allowed adaptive traffic signal control to outperform the fixed timing plan used by the city of Pittsburgh.

8.5.1 Six intersection grid

Table 8.2 shows the average speed and average wait time for the six intersection grid scenario over 20 instances. The use of density-estimate memory improved average vehicle speed. The Gaussian mixture model density-estimate memory improved speeds by 2.38% over the BPU algorithm alone. Using incremental clusters to retrieve from the memory led to higher average wait times, while using the Gaussian mixture model of the points in the memory improved average wait time over BPU by almost 50%. Table 8.3 shows the percent improvement of the memory methods as well as whether the differences are statistically significant. As for other experiments in this thesis, the statistical significance of these results has been evaluated using the Kruskal-Wallis test, considering a confidence of 95% ($p = 0.05$). For both density-estimate memory methods, the difference in results from plain BPU are significant. When comparing the two density-estimate memory methods, only the difference in wait time is significant, though the Gaussian mixture model method performs better on average than the method that uses clusters. The incremental clustering density-estimate memory (DEM) leads to very high waiting times for this problem, significantly worse than BPU. This suggests that the aggregate offset information retrieved from the memory is poor in quality.

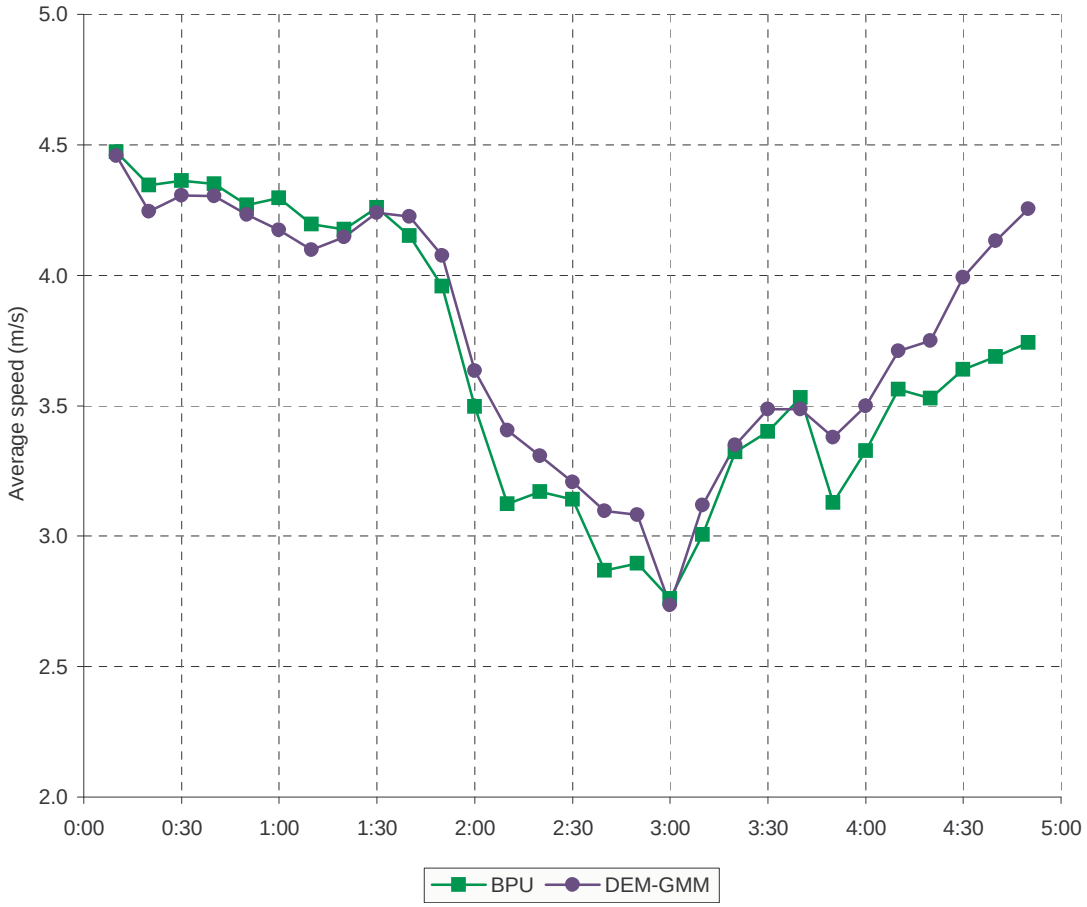

Figure 8.13: Average speed in meters per second over time for the six intersection grid scenario

Figures 8.13 and 8.14 show hourly plots of average speed and average wait time respectively for this scenario. Each data point at a particular time is the average value of the metric for cars that arrive in the simulation during the ten minute period beginning at that time. While BPU and DEM-GMM have similar performance during the early portion of the scenario, once the simulated accident occurs (at 2:00 hours into the simulation) the density-estimate memory shows a quicker response, with higher speeds and less waiting time. Congestion is also reduced, as can be seen by the rapidly growing wait times for BPU further into the scenario.

Based on these results, augmenting the BPU algorithm with a density-estimate memory increases average speed. When the Gaussian mixture model density-estimate memory is used, average wait time also decreases. The use of the Gaussian mixture model density-estimate memory reduces congestion and allows the traffic signals to continue to be responsive to changing demands.

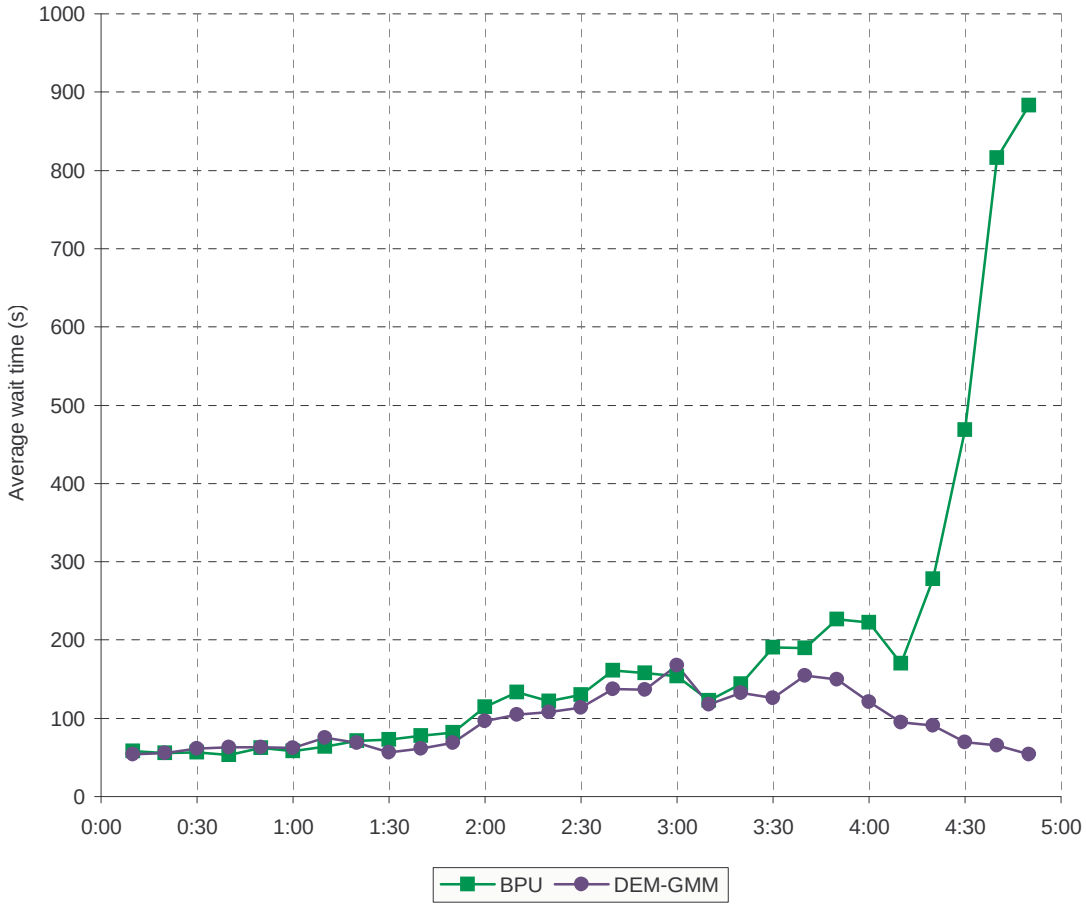

Figure 8.14: Average wait time in seconds over time for the six intersection grid scenario

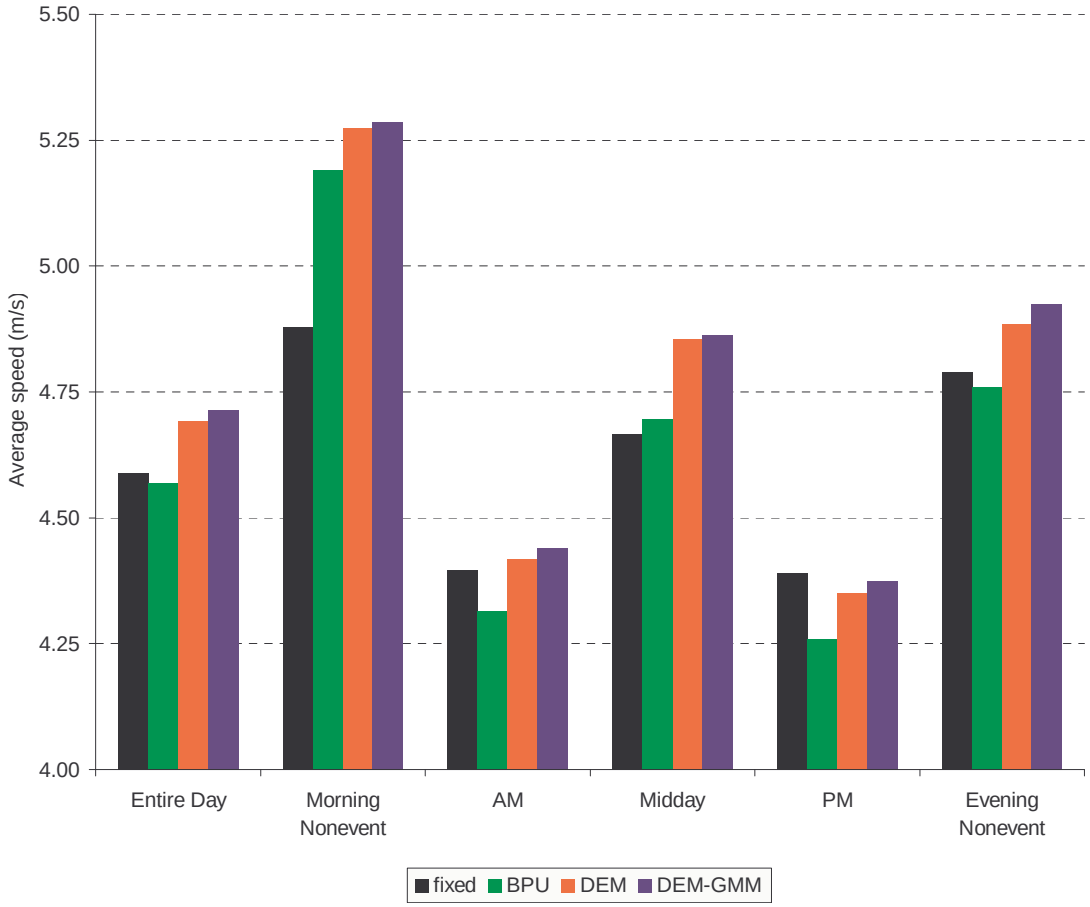

Figure 8.15: Average speed in meters per second by period for the downtown Pittsburgh scenario

8.5.2 Downtown Pittsburgh

The second experiment models 32 intersections in downtown Pittsburgh and uses much longer scenarios than the first experiment on the six intersection grid. Tables 8.4 and 8.5 show the average speed and average wait time by period for the four traffic signal control algorithms. Tables 8.6 and 8.7 compare these algorithms, showing percent improvement and statistical significance if it exists. Again, the Kruskal-Wallis test is used to test significance with 95% confidence. Figures 8.15 and 8.16 show the performance of each method overall and on the five periods of the day: NE1, the morning off-peak period; AM, the morning rush; MD, the midday period; PM, afternoon rush; and NE2, the evening off-peak period.

The results show that while the fixed timing plan outperforms BPU over the entire day, the introduction of memory produces higher average speeds than either BPU or the fixed timing plan. The wait time is improved over BPU but is still worse than for the fixed timing plan. During the five periods of the day—morning non-event, morning rush, midday, afternoon rush, and evening non-

Method	Entire Day	NE1	AM	MD	PM	NE2
fixed	4.5890	4.8788	4.3969	4.6666	4.3885	4.7883
BPU	4.5697	5.1909	4.3144	4.6954	4.2585	4.7583
DEM	4.6918	5.2734	4.4173	4.8546	4.3512	4.8834
DEM-GMM	4.7128	5.2858	4.4396	4.8620	4.3739	4.9240

Table 8.4: Average speed in meters per second by period for the downtown Pittsburgh scenario

Method	Entire Day	NE1	AM	MD	PM	NE2
fixed	97.4584	83.6390	108.6159	92.0562	108.8533	86.8457
BPU	118.5739	73.3603	134.9804	100.4871	159.3537	103.9774
DEM	107.2045	67.3089	119.5445	87.7726	149.1963	96.2740
DEM-GMM	104.4146	67.2763	116.5712	87.5253	145.6777	89.5775

Table 8.5: Average wait time in seconds by period for the downtown Pittsburgh scenario

Methods		Day	NE1	AM	MD	PM	NE2
BPU	fixed	-0.42	6.40 +	-1.88 -	0.62	-2.96 -	-0.63
DEM	fixed	2.24 +	8.09 +	0.46	4.03 +	-0.85	1.99 +
DEM-GMM	fixed	2.70 +	8.34 +	0.97	4.19 +	-0.33	2.83 +
DEM	BPU	2.67 +	1.59 +	2.39 +	3.39 +	2.18 +	2.63 +
DEM-GMM	BPU	3.13 +	1.83 +	2.90 +	3.55 +	2.71 +	3.48 +
DEM-GMM	DEM	0.45	0.23	0.50	0.15	0.52	0.83

Table 8.6: Percent speed improvement of method 1 over method 2 on the downtown Pittsburgh scenario (results that are statistically significant to 95% confidence are noted with a + or -)

Methods		Day	NE1	AM	MD	PM	NE2
BPU	fixed	-21.7 -	12.29 +	-24.3 -	-9.16 -	-46.4 -	-19.7 -
DEM	fixed	-10.0 -	19.52 +	-10.1 -	4.65 +	-37.1 -	-10.9 -
DEM-GMM	fixed	-7.14 -	19.56 +	-7.32 -	4.92 +	-33.8 -	-3.15
DEM	BPU	9.59 +	8.25 +	11.44 +	12.65 +	6.37	7.41 +
DEM-GMM	BPU	11.94 +	8.29 +	13.64 +	12.90 +	8.58 +	13.85 +
DEM-GMM	DEM	2.60	0.048	2.49	0.28	2.36	6.96 +

Table 8.7: Percent wait time improvement of method 1 over method 2 on the downtown Pittsburgh scenario (results that are statistically significant to 95% confidence are noted with a + or -)

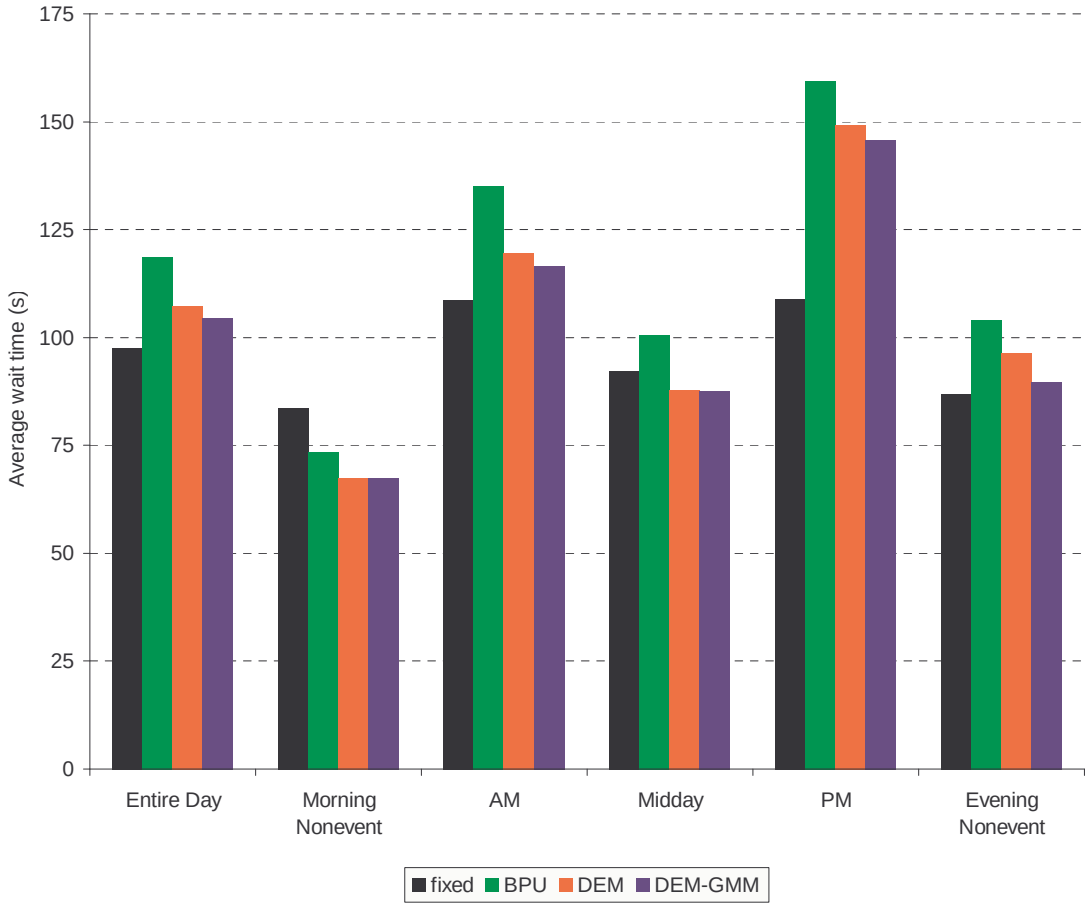

Figure 8.16: Average wait time by period for the downtown Pittsburgh scenario

event—the use of memory improves both speed and wait time significantly over BPU used alone. While BPU is only significantly better than the fixed timing plan during the morning non-event period, the density-estimate memory methods have significantly higher speeds than the fixed timing plan during the midday period and both morning and afternoon non-event periods. The memory methods also have significantly better wait times than the fixed timing plan during the morning non-event and midday periods. In the cases where the density-estimate memory methods are not able to outperform the fixed timing plan, the BPU algorithm is significantly worse than the fixed timing plan. Though density-estimate memory is able to improve performance of a learning algorithm, the memory relies on the underlying algorithm to find and then refine good solutions. When the algorithm exhibits very poor performance, memory may not very beneficial.

For the rush periods, AM and PM, BPU has lower speeds than the fixed timing plan. One likely explanation for this is that the offset time calculation is not properly designed for very congested periods where it takes cars longer to traverse a given edge. This is suggested by the very high wait times produced by BPU. Though the memory methods have higher average speeds than the fixed timing plan for the AM period, the improvement is not significant. For the PM period, the BPU algorithm is never able to find good enough solutions that the memory can improve enough to compete with the fixed timing plan.

Figures 8.17 and 8.18 show the hourly speed and wait times. Each point at a given time averages the values for all vehicles that arrived in the simulation during the half-hour period beginning at that time. These figures show dips in performance for all four algorithms during the peaks of demand, particularly at 8:00 AM and 5:00 PM during the morning and afternoon rushes, but also at 12:30 PM during the midday peak. The performance of DEM and DEM-GMM are very similar, though DEM-GMM is slightly quicker to adapt in periods of transition. Both memory methods respond quicker than BPU during transitions and have higher speeds and lower wait times. Though BPU produces higher average speeds for much of the simulation, the fixed timing plan is better during the morning and afternoon rush periods. Since the memory methods can only build on the BPU algorithm, the memory methods are outperformed by the fixed timing plan during the peak rush periods.

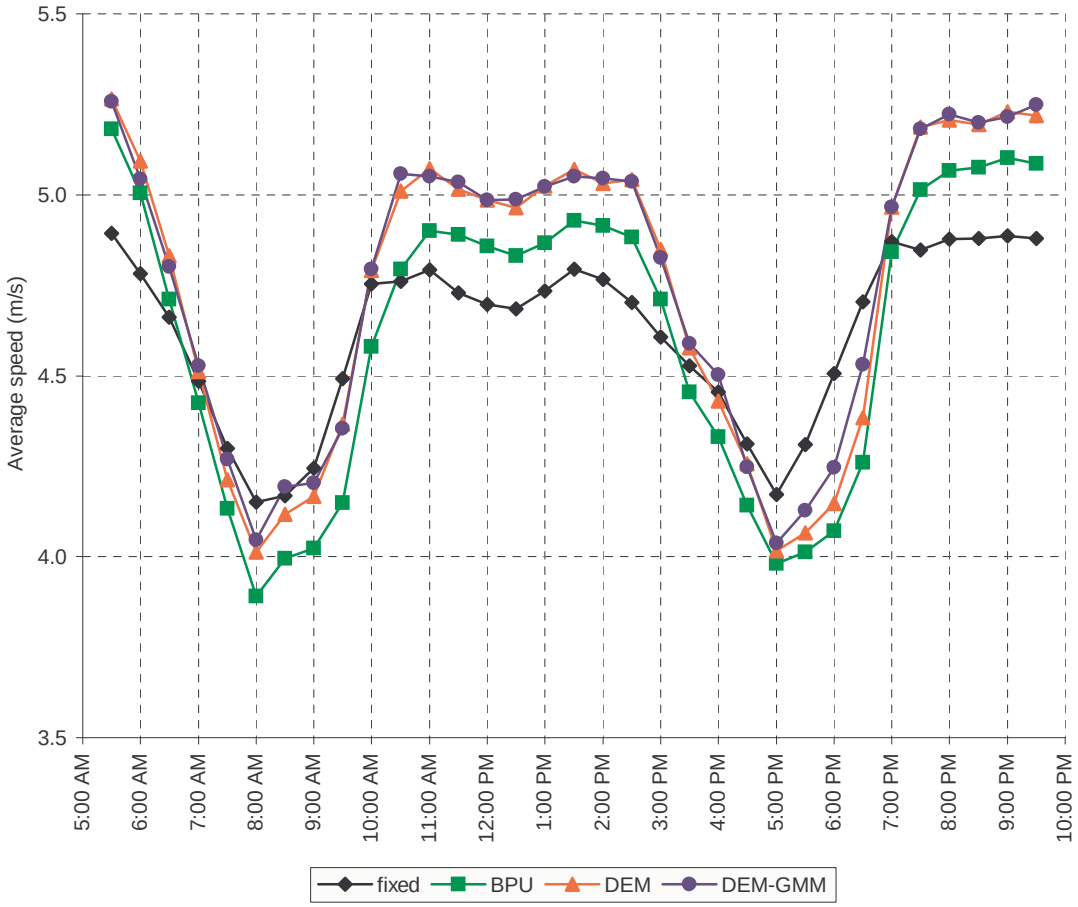

Figure 8.17: Average speed in meters per second over time for the downtown Pittsburgh scenario

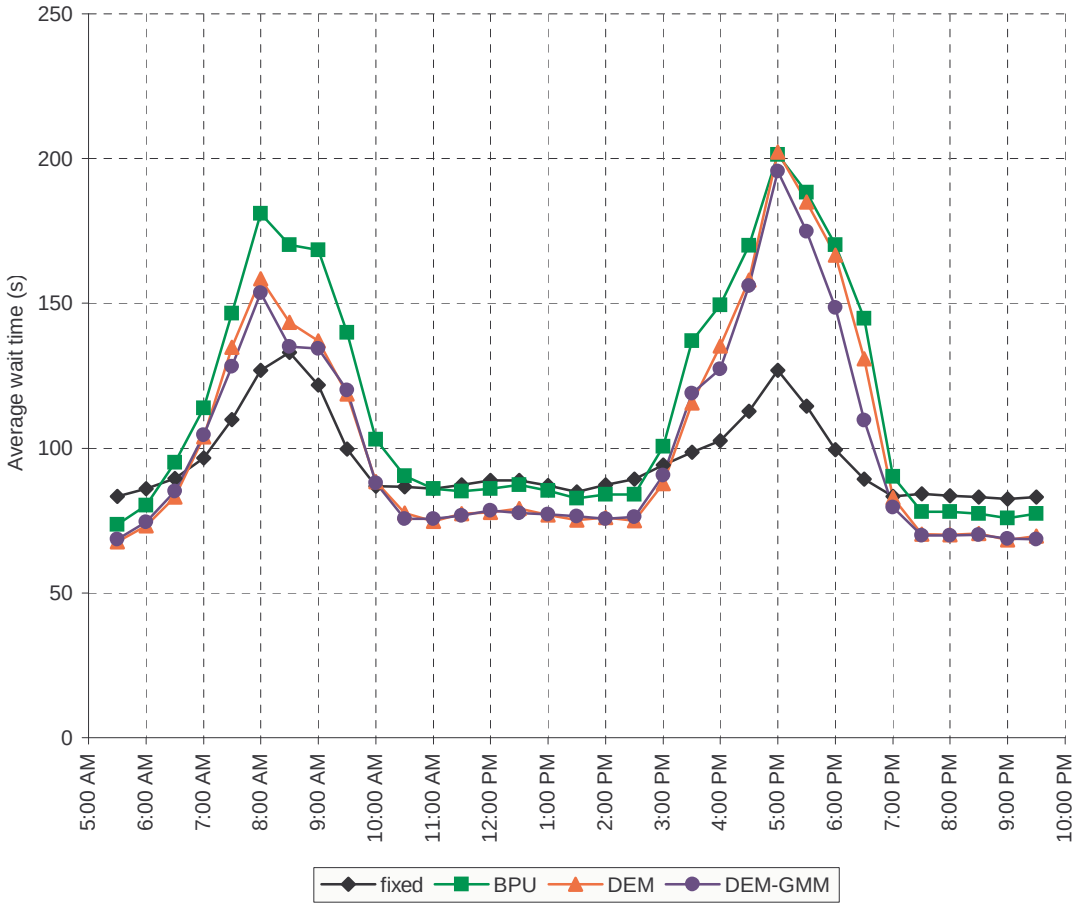

Figure 8.18: Average wait time in seconds over time for the downtown Pittsburgh scenario

8.6 Discussion

Based on these results, the addition of density-estimate memory improves the performance of an adaptive algorithm for traffic signal control. For most traffic demands, density-estimate memory increases the speed of vehicles over the BPU algorithm and decreases the wait time. Though the results are not as significant, the Gaussian mixture model density-estimate memory outperformed the incremental Gaussian clustering memory. When compared to a real fixed timing plan, the BPU algorithm is competitive for many types of traffic demands. The combination of density-estimate memory with the BPU algorithm often outperforms the fixed timing plan, though the limitations of the BPU algorithm prevent density-estimate memory from always producing the best performance.

One of the biggest limitations on the performance of a density-estimate memory is the quality of the underlying search or learning algorithm. While the BPU algorithm performs well under many conditions, it follows a very simple model. BPU only accounts for the departure of vehicles from the intersection, so it is susceptible to conditions where the phases are balanced, but where the growth of queues is uneven. For example, if traffic is saturated in several directions, all phases may be very efficient, but traffic from one direction may be backing up much faster than in other directions. In this case, it may be advantageous to have an unbalanced signal to avoid creating large queues. A more sophisticated algorithm might take into account queue lengths, arrival of new vehicles, occupancy on sensors, or other data that are available.

Another limitation of this version of BPU is the placement of detectors. Most phases allow at least one turning movement on multiple edges. With only exit detectors available, the traffic from each edge cannot be distinguished easily from one another. Since each edge has its own queue, the queues being served by a particular phase are likely to be unbalanced. The use of stop-line detectors would allow each edge to be separated from the other edges being served during a phase. This would allow phase utilization to be calculated for each edge, as is done in ACS Lite [89].

Finally, the offset calculation used by BPU is very simple, not taking into account the actual speed of vehicles, a sophisticated estimate of queue clearing time, or knowledge of specific vehicles. The coordination used to choose offsets is also very simple. The results suggest that these poor offset time calculations lead to higher waiting times. While density-estimate memory produced better vehicle speeds for most traffic demands, wait times were often worse. Many techniques exist to choose offsets that could potentially improve the performance of BPU.

A wide variety of adaptive algorithms can be implemented within the framework outlined in this chapter and many could benefit from the addition of density-estimate memory. The BPU algorithm, though simple, was competitive with the real fixed timing plan used in downtown Pittsburgh. BPU provides a good baseline for this problem, and with a better adaptive algorithm, one would expect density-estimate memories to produce even better results.

8.7 Summary

In this chapter, the dynamic problem of traffic signal control was presented as a possible application of density-estimate memory. Like other dynamic problems considered in this thesis, the traffic signal control problem includes both large, drastic changes (changes in the underlying traffic demand) and small, continuous changes (normal variability in traffic). An adaptive, traffic-responsive algorithm called balanced phase utilization (BPU) was introduced. Two implementations of density-estimate memory—incremental clustering and Gaussian mixture model—were described and used to augment the BPU algorithm. Two experiments were used to compare the performance of these algorithms. The first used a six intersection grid network with one lane in each direction for each road. The second used a realistic network modeled on 32 intersections in downtown Pittsburgh, Pennsylvania. This network used realistic traffic patterns and the adaptive methods were compared to the real fixed timing plan used on these intersections.

By augmenting the BPU algorithm with density-estimate memory, average vehicle speed increased and average wait time decreased. Each traffic signal maintained its own control system and memory; very little information needs to be communicated between intersections for this approach. Memory improved performance both during transitions and overall.

Though the BPU algorithm was slightly worse than the fixed timing plan over an entire day on average, the use of density-estimate memory allowed the adaptive system to outperform the fixed timing plan. For some periods, such as the afternoon rush, the fixed timing plan was so much better than BPU that memory could not improve performance enough to be competitive. When the underlying BPU algorithm was competitive, the use of density-estimate memory produced higher speeds than the fixed timing plan. These results suggest that adding memory to a better-performing adaptive algorithm would make it possible to outperform the fixed timing plan during these periods.

Part III

Memory for problems with shifting feasible regions

Chapter 9

Classifier-based memory

A variety of dynamic benchmark problems for evolutionary algorithms have been considered, including the moving peaks problem [13, 72], the dynamic knapsack problem, dynamic bit-matching, dynamic scheduling, and others [14, 52]. The commonality between most benchmark problems is that while the fitness landscape changes, the search space does not. For example, in the moving peaks problem, any point in the landscape—represented by a vector of real numbers—is always a feasible solution. One exception to this among common benchmark problems is dynamic scheduling, where the pending jobs change over time as jobs are completed and new jobs arrive. Given a feasible schedule at a particular time, the same schedule will not be feasible at some future time when the pending jobs are completely different. Previous work on evolutionary algorithms for dynamic scheduling problems have focused primarily on extending schedulers designed for static problems to dynamic problems [63, 6], problems with machine breakdowns and redundant resources [24], improved genetic operators [98], heuristic reduction of the search space [65], and anticipation to create robust schedules [16, 17]. Louis and McDonnell [64] have shown that case-based memory is useful given similar static scheduling problems. Chapter 6 described the application of memory to a factory coordination problem, but instead of storing actual schedules, the memory stored control parameters for the scheduler. Miyashita and Sycara [68] used case-based reasoning to improve schedule quality for a dynamic job-shop rescheduling problem, though each case represented a schedule repair action for a single operation of a job rather than an entire schedule. Little work has been done on using a memory to capture whole schedules for use in dynamic rescheduling. Since the addition of memory has been successful in improving the performance of evolutionary algorithms on other dynamic problems, there is a strong case for using memory for dynamic scheduling problems as well.

In most dynamic optimization problems, the use of an explicit memory is relatively straightforward. Stored points in the landscape remain viable as solutions even though the landscape is changing, so

a memory may store individuals directly from the population [13]. In dynamic scheduling problems, the jobs available for scheduling change over time, as do the attributes of any given job relative to the other pending jobs. If an individual in the population represents a prioritized list of pending jobs to be fed to a schedule builder, any memory that stores an individual directly will quickly become irrelevant. Some or all jobs in the memory may be complete, the jobs that remain may be more or less important than in the past, and the ordering of jobs that have arrived since the memory was created will not be addressed by the memory at all. For these types of problems that have both a dynamic fitness landscape and a shifting search space, a memory should provide some indirect representation of jobs in terms of their properties to allow mapping to similar solutions in future scheduling states.

This chapter presents one such memory for dynamic scheduling called classifier-based memory. Instead of storing a list of specific jobs, a memory entry stores a list of classifications which can be mapped to the pending jobs at any time. The remainder of this chapter describes classifier-based memory for dynamic scheduling problems and compares it to both a standard evolutionary algorithm and to other approaches from the literature.

9.1 Dynamic job shop scheduling

The dynamic job shop scheduling problem used for these experiments is an extension of the standard job shop problem. In this problem, n jobs must be scheduled on m machines of mt machine types with $m > mt$. Processing a job on a particular machine is referred to as an operation. There are a limited number of distinct operations ot which will be referred to as operation types. Operation types are defined by processing times p_j and setup times s_{ij}. If operation j follows operation i on a given machine, a setup time s_{ij} is incurred. Setup times are sequence dependent—so s_{ij} is not necessarily equal to s_{ik} or s_{kj} $(i \neq j \neq k)$—and are not symmetric—so s_{ij} is not necessarily equal to s_{ji}. Each job is composed of k ordered operations; a job's total processing time is simply the sum of all setup times and processing times of a job's operations. Jobs have prescribed due-dates d_j, weights w_j, and release times r_j. The release of jobs is a non-stationary Poisson process, so the job inter-arrival times are exponentially distributed with mean λ. The mean inter-arrival time λ is determined by dividing the mean job processing time $\bar{P}$ by the number of machines m and a desired utilization rate U, i.e.

$$\lambda = \frac{\bar{P}}{mU}$$

The mean job processing time is

$$\bar{P} = (\varsigma + \bar{p})\,\bar{k}$$

where ς is an expected setup time, $\bar{p}$ is the mean operation processing time, and $\bar{k}$ is the mean number of operations per job. There are ρ jobs with release times of 0, and new jobs arrive non-deterministically over time. The scheduler is completely unaware of a job prior to the job's release time. Job routing is random and operations are uniformly distributed over machine types; if an operation requires a specific machine type, the operation can be processed on any machine of that type in the shop. The completion time of the last operation in the job is the job completion time c_j. A single objective, weighted tardiness, is considered. The tardiness is the positive difference between the completion time and the due-date of a job,

$$T_j = \max(c_j - d_j, 0)$$

The weighted tardiness is

$$WT_j = w_j T_j$$

As an additional dynamic event, the experiments model machine failure and repair. A machine fails at a specific time—the *breakdown time*—and remains unavailable for some length of time—the *repair time*. The frequency of machine failures is determined by the percentage downtime of a machine—the breakdown rate γ. Repair times are determined using the mean repair time ε. Breakdown times and repair times are not known a priori by the scheduler.

9.2 Evolutionary algorithms for dynamic scheduling

At a given point in time, the scheduler is aware of the set of jobs that have been released but not yet completed. The uncompleted operations of these jobs will be called the set of pending operations

$$P = \left\{ o_{j,k} \,\middle|\, r_j \leq t, \neg complete(o_{j,k}) \right\}$$

where $o_{j,k}$ is operation k of job j. Operations have precedence constraints, and operation $o_{j,k}$ cannot start until operation $o_{j,k-1}$ is complete (operation $o_{j,-1}$ is complete $\forall j$, since operation $o_{j,0}$ has no predecessors). When the immediate predecessor of an operation is complete, one can say that the operation is schedulable. The set of schedulable operations is defined as

$$S = \left\{ o_{j,k} \,\middle|\, o_{j,k} \in P, complete(o_{j,k-1}) \right\}$$

Like most evolutionary algorithm approaches to scheduling problems, solutions are encoded as prioritized lists of operations. Since this is a dynamic problem where jobs arrive over time, a solution is a prioritized list of only the pending operations at a particular time. Since the pending operations change over time, each individual in the population is updated at every time step of the simulator.

When operations are completed, they are removed from every individual in the population, and when new jobs arrive, the operations in the job are randomly inserted into each individual in the population.

The well known Giffler and Thompson algorithm [38] is used to build active schedules from a prioritized list. First, from the set of pending operations P create the set of schedulable operations S. From S, find the operation o' with the earliest completion time t_c. Select the first operation from the prioritized list which is schedulable, can run on the same machine as o', and can start before t_c. Update S and continue until all jobs are scheduled.

1. Build the set of schedulable operations S
2. (a) Find o' on machine M' with the earliest completion time t_c
   (b) Select the operation $o^*_{i,k}$ from S which occurs earliest in the prioritized list, can run on M', and can start before t_c
3. Add $o^*_{i,k}$ to the schedule and calculate its starting time
4. Remove $o^*_{i,k}$ from S and if $o^*_{i,k+1} \in E$, add $o^*_{i,k+1}$it to S
5. While S is not empty, go to step 2

The evolutionary algorithm is generational with a population of 100 individuals. The PPX crossover operator [7] is used with probability 0.6, a swap mutation operator with probability 0.2, elitism of size 1, and linear rank-based selection. Rescheduling is event driven; whenever a new job arrives, a machine fails, or a machine is repaired, the evolutionary algorithm runs until the best individual in the population remains the same for 10 generations. Figure 9.1 shows the evolutionary algorithm scheduling system.

9.3 Classifier-based memory for scheduling problems

The use of a population-based search algorithm allows us to carry over good solutions from the immediate past, but how can one use information from good solutions developed in the more distant past? Some or all of the jobs that were available in the past may be complete, there may be many new jobs, or a job that was a low priority may now be urgent. Unlike many dynamic optimization problems, this shifting search space means one cannot store individuals directly for later recall. Instead, a memory should allow us to map the qualities of good solutions in the past to solutions in the new environment. This chapter presents one such memory for dynamic scheduling called classifier-based memory. Instead of storing prioritized lists of operations, an indirect representation

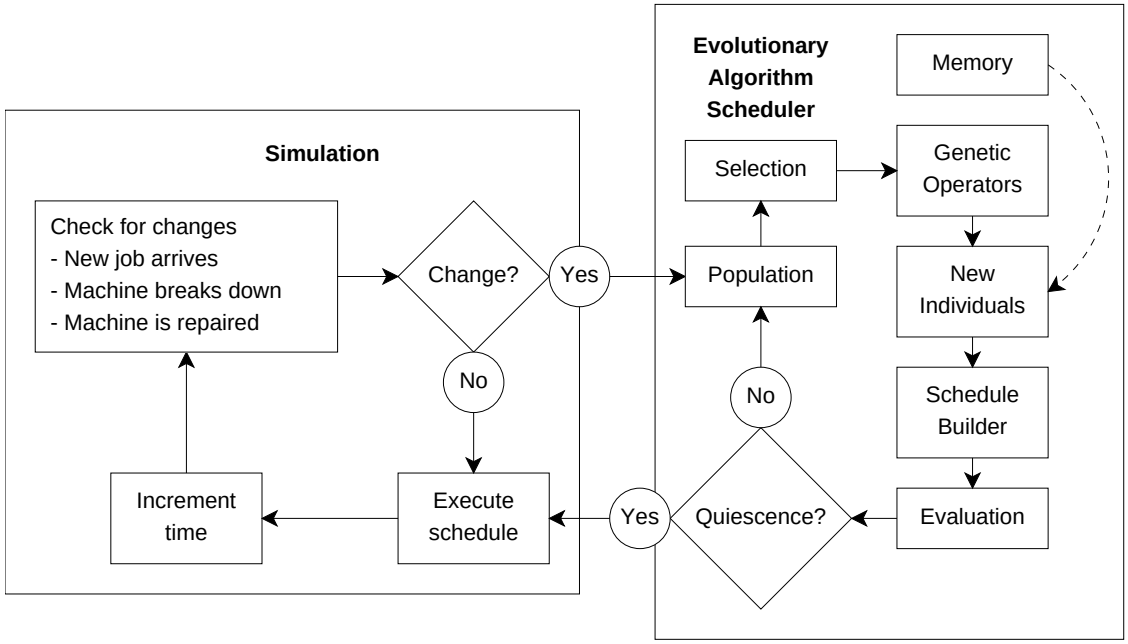

Figure 9.1: Evolutionary algorithm scheduling system in simulation. The simulator executes a schedule until a change in the environment is detected. Then the evolutionary algorithm scheduler evolves a new schedule to return to the simulation.

is used, storing a prioritized list of classifications of operations. To access a memory entry at a future time, the pending jobs are classified and matched to the classifications in the memory entry, producing a prioritized list of operations.

A memory entry is created directly from a prioritized list of pending operations. First, operations are ranked according to a set of attributes and then quantiles are determined for each ranking in order to classify each operation with respect to each attribute. The number of attributes a and the number of subsets q determine the total number of possible classifications q^a. For example, operations might be ranked by the due-date of the job and the operation processing time ($a = 2$) and then classified using quartiles ($q = 4$). This produces $q^a = 4^2 = 16$ possible classifications. A sample operation might be part of a job very near its due-date, such that it was in the first quartile of the due-date ranking. That operation might take very long to process, such that it was in the fourth quartile of the operation processing time ranking. This would produce a classification of $\{0,3\}$.

Through this process, the prioritized list of operations is converted to a prioritized list of classifications. This list of classifications may then be stored as a memory entry. To retrieve a valid schedule from a memory entry, the pending operations are mapped to the prioritized list of classifications stored in that entry. Each operation is ranked according to the same attributes, then quantiles are determined to classify each operation. Then, for each of these new classifications, the best match is found among the classifications in the memory entry. Each pending operation is assigned a sort

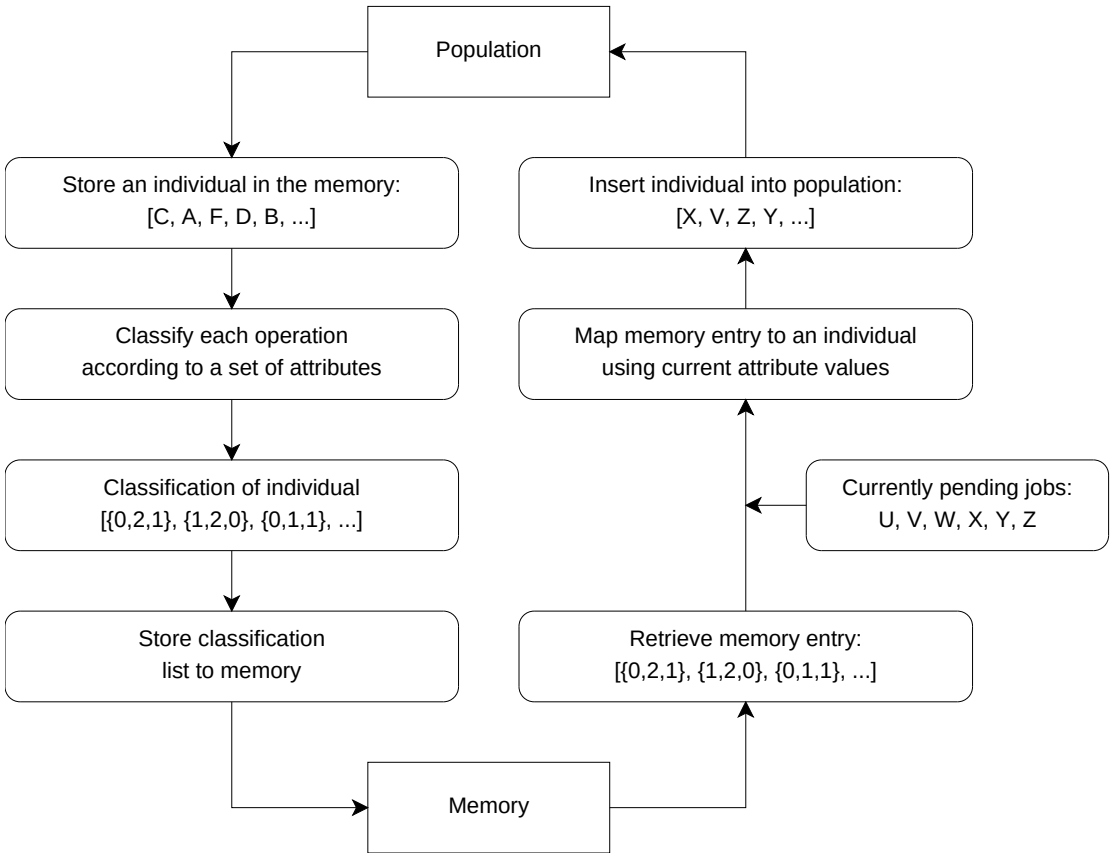

Figure 9.2: Overview of storage and retrieval operations of the classifier-based memory. Individuals to be stored in the memory are classified and their classification lists are stored in the memory. To retrieve a memory entry, the entry is mapped to an individual using currently pending jobs and their attributes. This individual can then be inserted into the population.

key based on the position of its classification's best match within the prioritized list in memory. The sort key for classification x in memory entry Y is j such that

$$\min_{j=0, j<|Y|} \sum_{i=0}^{a} |x_i - Y(j)_i|$$

where $Y(j)$ is classification j in list Y. If there is more than one best match, the average of the positions is used as the sort key. Then, the pending operations are sorted by these sort keys to create a prioritized list of operations which can be used as an individual for the evolutionary algorithm.

The basic mechanisms for interacting with the memory are the same as those for other explicit memories used for dynamic optimization with evolutionary algorithms. Figure 9.2 shows an overview of how individuals from the population are classified and stored in the memory and how memory entries are mapped to valid individuals and inserted in the population. At every generation, an indi-

vidual is created from each memory entry and the individuals are inserted into the population. Every φ generations and at the end of every rescheduling cycle, a replacement strategy chooses whether to insert the best individual in the population into memory. If the memory is full, the classification list of this best individual replaces a current memory entry using the *mindist2* replacement strategy [14]. To maintain diversity in the memory, the replacement strategy determines the two classification lists i and j that are closest together among the classification of the best individual in the population and all of the memory entries. The less fit list j is then chosen as a candidate for replacement. The distance between two classification lists S and T is the sum of the differences between a classification's position in one list and the position of its best match in the other list. As before, if there is more than one best match, the mean of the positions is used. Since this is not symmetric, it is done for both lists. If S has length s and T has length t, then

$$d = \sum_{i=0}^{s} |i - bestmatch(S(i), T)| + \sum_{i=0}^{t} |i - bestmatch(T(i), S)|$$

As long as the classification of the best individual in the population is not the candidate for replacement, classification list j is replaced with the new classification list when

$$f_j \frac{d_{ij}}{d_{max}} \leq f_{best}$$

where f_x is the fitness of the prioritized list produced by the classification list x, d_{ij} is the distance between classification lists i and j, and d_{max} is the maximum possible distance.

Figure 9.3 shows a simplified example. Suppose we have a memory with $q = 2$ and $a = 3$ and the following attributes: job due-date (dd), operation processing time (pt), and job weight (w). At time 400, we have a prioritized list of four operations that we'd like to store in the memory. With $q = 2$ and four operations, the lower two values for each attribute receive a classification of 0 and the higher two values a classification of 1. So job A has a due-date classification of 1, a process time classification of 0, and a weight classification of 1, for an overall classification of $class(A) \rightarrow 101$. After classification, the prioritized list $[C, B, A, D]$, becomes $[011, 000, 101, 110]$. This classification list is stored in memory. Suppose that at time 10000, we would like to retrieve this memory entry to create a prioritized list from the four pending operations W, X, Y, and Z. These new operations are classified and then given a score based on their best match within the memory entry. The pending operations are sorted by these scores in order to create a new prioritized list that may be inserted into the population as a new individual.

This classifier-based memory also allows new jobs to be ordered alongside older jobs that may have been available when the memory entry was created; the classifier-based memory does not store specific information about operations, only how a particular operation compares to other pending operations at a specific point in time. A memory entry may place a particular operation at different

At $t = 400$, store $[C,B,A,D]$		Due-date	Process time	Weight
$A = \{$dd: 800, pt: 100, w: 7$\} \rightarrow 101$	$\uparrow$	450	100	3
$B = \{$dd: 450, pt: 110, w: 5$\} \rightarrow 000$	0	500	110	5
$C = \{$dd: 500, pt: 130, w: 9$\} \rightarrow 011$	1	800	130	7
$D = \{$dd: 900, pt: 150, w: 3$\} \rightarrow 110$	$\downarrow$	900	150	9

$$[C,B,A,D] \rightarrow [011,000,101,110] \rightarrow \text{memory}$$

At $t = 10000$, retrieve $[011,000,101,110]$		Due-date	Process time	Weight
$W = \{$dd: 10400, pt: 80, w: 1$\} \rightarrow 110 \rightarrow (3)$	$\uparrow$	10070	50	1
$X = \{$dd: 10100, pt: 70, w: 5$\} \rightarrow 011 \rightarrow (0)$	0	10100	60	2
$Y = \{$dd: 10500, pt: 50, w: 6$\} \rightarrow 101 \rightarrow (2)$	1	10400	70	5
$Z = \{$dd: 10070, pt: 60, w: 2$\} \rightarrow 000 \rightarrow (1)$	$\downarrow$	10500	80	6

$$[011,000,101,110] \rightarrow [X,Z,Y,W] \rightarrow \text{population}$$

Figure 9.3: An example of classifier-based memory. Given a memory with $q = 2$, $a = 3$, and the following attributes: job due-date (dd), operation processing time (pt), and job weight (w). At $t = 400$, an individual is stored in the memory. A prioritized list of operations is classified based on the attributes of all pending operations at $t = 400$. The classification list is then stored in memory. If that memory entry is retrieved at $t = 10000$, the entry needs to be mapped to the currently pending jobs. Each of the pending jobs is classified and then matched to a position in the memory entry. This mapping can then be inserted into the population as a new individual.

positions in the prioritized list as its due-date becomes more imminent or as the mix of pending operations changes the operation's relative importance.

Introducing this abstraction layer into memory requires additional computational overhead. Storing a solution to memory requires sorting n pending operations by a attributes, which can be done in $O(a \cdot n \log n)$. Retrieving a solution from the memory requires classifying n pending operations by a attributes, but also matching each operation to a position in a memory entry of length e, which can be done in $O(a \cdot e \cdot n \log n)$. Though the use of classifier-based memory requires additional computation over the standard EA, this computation is not significant in comparison to the time necessary to evaluate a solution.

In this chapter, classifier-based memory uses four attributes ($a = 4$): job due-date, job weight, operation processing time, and operation order within the job. Rankings are divided into quartiles ($q = 4$) for a total of 256 possible classifications. Many other attributes exist that could easily be included, as this approach does not depend on a particular set of attributes.

9.4 Experiments

To examine the effects of classifier-based memory on schedule fitness and search time, several common approaches are compared. A standard evolutionary algorithm (SEA) is used as a baseline, since the optimal schedules are not known for any of the problem instances. The standard evolutionary algorithm with classifier-based memory (SEAm) is also considered. Prior results on benchmarks like the moving peaks problem suggest that memory-based approaches work better when combined with a diversity strategy [14]. Hence, a standard evolutionary algorithm with 25 random immigrants [42] per generation (RI) and the same approach with classifier-based memory (RIm) are also considered. Finally, the memory/search approach of [14], also using the classifier-based memory, is included in the comparison. In memory/search, the population is divided into a memory subpopulation and a search subpopulation. The memory population can both store individuals to the memory and retrieve memory entries. The search population can only store items to the memory, and the population is re-initialized randomly every time the problem changes.

When creating problem instances, the utilization rate was selected so that jobs arrive at approximately replacement rate. The number of jobs available at time 0, ρ, is then the expected schedule size. By changing the due-date tightness, the difficulty of the problem may vary. The due-date tightness parameter τ is the percentage of jobs expected to meet their due-dates. The expected waiting time before job completion is the expected number of jobs in the schedule times the mean job completion time $\rho\bar{P}$. Due-dates are generated by

$$d_j = r_j + \bar{P} + [0, 2\rho\bar{P}\tau]$$

The setup time severity is given by

$$\eta = \frac{\bar{s}}{\bar{p}}$$

where $\bar{s}$ is the mean setup time and $\bar{p}$ is the mean operation processing time. The number of breakdowns per machine is uniformly distributed with the mean number of breakdowns per machine equal to $\frac{n\bar{P}}{m}\frac{\gamma}{\varepsilon}$. Breakdown times for each machine are uniformly distributed over $\left[0, \frac{n\bar{P}}{m}\right]$. Repair times are uniformly distributed over $\left[\frac{1}{2}\varepsilon, \frac{3}{2}\varepsilon\right]$.

For the experiments in this chapter, the following settings were used to create problem instances. The job shop contains 2 machines each of $mt = 3$ machine types, for a total of $m = 6$ machines. There are 50 operation types, with mean operation processing time $\bar{p} = 100$ and processing times uniformly distributed over $[50, 150]$. The setup time severity is $\eta = 0.5$, so the mean setup time is $\bar{s} = 50$. The setup times are uniformly distributed over $[0, 2\bar{s}]$. The estimated setup time is $\varsigma = 35$. A problem instance consists of 500 jobs, each with $k = 3$ operations. There are $\rho = 25$ jobs with release times of 0. Job weights are uniformly distributed over $[1, 10]$. The utilization rate

is $U = 0.7$, and the breakdown rate is $\gamma = 0.1$, for a total utilization of 0.8. The mean repair time is $\varepsilon = 10\bar{p} = 1000$. To control the problem difficulty, the due-date tightness of the jobs varied. As the due-date tightness changes, the types of situations the scheduler faces also change. These experiments tested with due-date tightnesses $\tau \in \{0.5, 0.8, 1.1\}$, from very tight due-dates where many jobs will be late, to loose due-dates where most jobs are expected to be on time. For each value of τ, 10 problem instances were created, for a total of 30 problem instances.

Rather than rebuild the memory from scratch on every problem instance during these experiments, several seed memories were prebuilt using SEAm over a larger number of jobs, varying the due-date tightnesses of the jobs. Though these experiments test over a limited number of jobs, if actually implemented in a scheduling system, the evolutionary algorithm would work over a long time horizon, and so the steady state performance of the evolutionary algorithm is of interest. By prebuilding the memory, this state of the algorithm may be better simulated. The memory may still change with the same replacement strategy, but after seeing a large number of jobs, the stability of the memory is much higher than if the memory was built from scratch for every problem instance. For each run of an evolutionary algorithm with memory, one of the prebuilt memories was chosen at random as a seed memory. Updating of the memory occurred as normal: memory replacement took place every $\varphi = 10$ generations and at the end of each rescheduling cycle.

Simulation runs for each evolutionary algorithm variant were performed on each of the 30 problem instances. Since this is a dynamic problem, one is interested not just in fitness improvements but in improvements in the speed of search. As in [6, 16, 17], the steady state performance is measured by discarding the first 100 and last 100 jobs. The summed weighted tardiness of the middle 300 jobs is used as the fitness. Search is measured in a similar way, by only including optional search generations that occur while the middle 300 jobs are among the pending jobs in the system. At the end of every rescheduling event, the scheduler is required to search for 10 generations where the best individual does not improve. Any generations per rescheduling event aside from these 10 constitute the optional search. Also, the number of rescheduling events is made up both of new job arrivals and machine breakdowns. Since the scheduler performance determines how long this period lasts, the number of machine breakdowns during this period is not fixed, so neither is the total number of rescheduling events. The search required for each algorithm can be compared more fairly by comparing only the number of optional generations per event.

The standard evolutionary algorithm was also compared to a suite of priority-dispatch rules to evaluate the benefits of search for this problem. Six rules were evaluated on the same scenarios as the evolutionary algorithm scheduler: earliest due-date (EDD) [28], earliest weighted due-date (EWDD), shortest processing time (SPT), apparent tardiness cost (ATC) [99, 77], Raman's rule [80], and apparent tardiness cost with setups (ATCS) [61]. The priority-dispatch rules were evaluated as a suite, always using the best performing rule for each scenario.

$\tau = 0.5$	$\tau = 0.8$	$\tau = 1.1$
54.66% (+)	68,57% (+)	50.94% (+)

Table 9.1: Fitness improvement of the standard evolutionary algorithm scheduler over a suite of priority-dispatch rules (statistically significant results are noted with a + or -)

	$\tau = 0.5$	$\tau = 0.8$	$\tau = 1.1$
Standard EA with memory	0.9% (+)	1.5%	15.7% (+)
Random immigrants	-5.7% (-)	-49.6%	-30.3% (-)
Random immigrants with memory	-8.3%	-51.2% (-)	13.1% (+)
Memory/Search	0.2% (+)	-18.0%	2.1% (+)

Table 9.2: Fitness improvement over the standard evolutionary algorithm (statistically significant results are noted with a + or -)

9.5 Results

Table shows the percentage of improvement in average fitness of the evolutionary algorithm scheduler over a suite of priority-dispatch rules. As for other experiments in this thesis, the statistical significance of these results has been evaluated using the Kruskal-Wallis test, considering a confidence of 95%. The evolutionary algorithm scheduler significantly improved schedule quality over priority-dispatch rules for all due-date tightnesses.

Table 9.2 shows the percentage of improvement in average fitness over the standard evolutionary algorithm. SEAm performs slightly better than SEA with tight and medium due-dates. When the due-dates are loose, SEAm performs significantly better than SEA. When diversity measures are introduced, performance actually drops. With just random immigrants, fitness worsens for all τ, but especially for medium due-dates. With RIm, performance on loose due-dates actually improves over SEA, though not over SEAm. With memory/search, performance gains are very slight for tight and loose due-dates, but performance worsens for medium tightness. The improvement (or lack thereof) for each of the approaches is worst with $\tau = 0.8$, except for SEAm where the improvement for medium due-dates is slightly better than that for tight due-dates.

Table 9.3 shows the percentage improvement in average optional generations per event over the standard evolutionary algorithm. SEAm shows good search reduction for tight and loose due-dates, with very slight improvement for medium due-dates. RIm actually improves search speed over SEAm for medium due-dates, but if one considers how much worse fitness was in this case, this improvement is not really meaningful. Of all the approaches, memory/search is the only one that fails to improve search speed for any due-date tightness. Again, medium due-dates show the worst performance in three of the four approaches, with RI as the only exception.

For both fitness and search, the addition of classifier-based memory improved performance over

	$\tau = 0.5$	$\tau = 0.8$	$\tau = 1.1$
Standard EA with memory	10.0% (+)	2.9% (+)	22.8% (+)
Random immigrants	9.4%	-5.3% (-)	-13.5% (-)
Random immigrants with memory	9.5% (+)	8.5% (+)	23.4% (+)
Memory/Search	-18.5% (-)	-35.6%	-16.8% (-)

Table 9.3: Search improvement over the standard evolutionary algorithm (statistically significant results are noted with a + or -)

the standard evolutionary algorithm. While significant improvement in fitness was only evident for loose due-dates, search improved for most problem instances. SEAm showed improvement for all three values of τ but the least improvement for $\tau = 0.8$. Based on these results, of the three types of due-dates, medium due-dates present search landscapes that are larger and more difficult to search than those for the other due-date tightnesses.

While the combination of memory and diversity techniques has yielded good results for most dynamic benchmark problems, for this dynamic scheduling problem none of the diversity approaches performed well. Perhaps due to the shape of the search landscape, diversity techniques are simply disruptive, rather than helpful in finding areas of high fitness. Memory/search, which devotes half of its population to searching for new individuals to include in the memory, is at a disadvantage in the steady state environment considered here. This approach might still be useful for prebuilding memories, where search time is not an issue.

9.6 Summary

This chapter described a memory enhanced evolutionary algorithm approach to the dynamic job shop scheduling problem. Memory enhanced evolutionary algorithms have been widely investigated for other dynamic optimization problems, but not for problems like dynamic scheduling where changes in the fitness landscape are accompanied by shifts in the search space. A classifier-based memory was described that enables the mapping of information about jobs at one point in time to the creation of valid schedules at another point in time. Several evolutionary algorithm variants were compared, with and without memory, on problem instances of varied difficulty. These results show that classifier-based memory can improve both schedule fitness and the speed of search over a standard evolutionary algorithm. These results also show that diversity techniques, which have had success on other dynamic benchmark problems, show decreased fitness and search speed for the dynamic scheduling problem investigated here.

Areas not considered in this chapter include anticipation of robust or flexible schedules, heuristic reduction of the search space, or other approaches from previous work for improving performance

on dynamic scheduling problems, because these approaches are complementary to the use of memory. Unlike more complementary techniques, density-estimate memory would be more difficult to combine with classifier-based memory. When aggregating multiple points, the different lengths of each point would have to be considered. One possible approach would be to extend all memory points to the length of the longest point. The discrete classifier used in this chapter also would not be ideal for use by a density-estimate memory when aggregating multiple points. A real-valued classifier would be more suitable, though the performance of this type of classifier-based memory is unclear.

Given the lack of prior work on storing information about complete schedules in memory for dynamic rescheduling, these experiments were an attempt to determine the potential of classifier-based memory. Comparing the performance of classifier-based memory using different attributes, a variety of quantile sizes, larger memories, or other changes in the memory structure would shed more light on the potential of classifier-based memories for dynamic scheduling. Also, other memory types could be constructed to include ways to retain information about setup times, periodic changes in the mix of operation types over time, or other types of information that this memory cannot easily capture.

Chapter 10

Conclusions

Many real-world problems are dynamic, undergoing changes that affect the quality of solutions. As changes occur, optimization and learning algorithms must adapt quickly to maintain high solution quality. Balancing the speed of arriving at good solutions with the time necessary to find the best solutions often requires quite different approaches than might be taken for static problems.

Most dynamic problems do not experience completely random changes. Instead, the current environment is often similar to previous environments. By using information from the past in the form of a memory, a search algorithm can often exploit these similarities to adapt more quickly after changes occur in the environment.

Memory has been used extensively to help the performance of optimization and learning algorithms on problems with dynamic environments. Many memories are quite simple, storing only a small number of solutions. This thesis has explored the improvement of memory systems to make memory more powerful and useful for more problems. Two new classes of memories—density-estimate memory and classifier-based memory—were introduced. Density-estimate memory builds and stores probabilistic models of many good solutions in memory, creating richer models of the dynamic search space. Classifier-based memory extends the use of memory to problems with shifting feasible regions.

10.1 Summary

Part I discussed optimization and learning for dynamic problems. Chapter 2 began by defining the concept of a dynamic environment. Since problems may be dynamic in many ways, a classification system for dynamic environments was defined. A variety of benchmark problems with dynamic environments were also discussed. Chapter 3 surveyed prior work in the area of dynamic optimization and learning, particularly the use of memory for dynamic optimization. Chapter 4 defined a

standard memory that has been widely used in the literature for dynamic optimization and discussed the strengths and weaknesses of this approach.

The remainder of this thesis focused on two new classes of memory that address many of the weaknesses in the standard memory. Density-estimate memories build and store probabilistic models of good solutions within memory. Density-estimate memory builds rich models of the dynamic search space that can be more easily refined over time than can those in the standard memory. Density-estimate memory typically stores many more previous solutions than a standard memory while keeping overhead low. Classifier-based memories allow memory to be used on problems with shifting feasible regions. Classifier-based memory adds an abstraction layer to memory that allows old solutions to be mapped to current feasible solutions.

Part II described the general class of density-estimate memories and specific implementations of density-estimate memory on three dynamic problems. Chapter 5 introduced density-estimate memories. Density-estimate memory is a new memory technique that builds density estimation models to aggregate information from many solutions stored in memory. As the search process discovers good solutions, the models stored in the memory are constantly refined. Though many more points are stored in the memory, the overhead associated with building, maintaining, and interacting with the memory remains low. Density-estimate memories may use a variety of probabilistic models within the memory. Since each type of probabilistic model has a different amount of overhead required to build and maintain the models in memory, density-estimate memories offer a great deal of flexibility in choosing model types depending the the requirements of the dynamic problem. The remainder of Part II considered the application of density-estimate memory to three dynamic problems: factory coordination, dynamic optimization with evolutionary algorithms, and adaptive traffic control.

Chapter 6 applied density-estimate memory to a reinforcement learning algorithm on a distributed, dynamic factory coordination problem. Three density-estimate memory implementations were compared to a standard memory, an infinite-sized standard memory, and the learning algorithm alone. All memory-enhanced algorithms improved performance over the baseline learning algorithm by enabling the learning algorithm to adapt more quickly to large, drastic changes in the underlying distribution of jobs in the factory. Though all of the memories performed well, the density-estimate memories consistently outperformed all the other methods. Among the density-estimate memories, the two using Gaussian models tended to outperform the memory using a simpler clustering model.

Chapter 7 compared density-estimate memory to a variety of other techniques for improving evolutionary algorithms on dynamic problems using a version of the Moving Peaks benchmark problem with very severe changes in the search space and a large number of wide peaks. Density-estimate memory outperformed all other methods, including the state-of-the-art self-organizing scouts method. Density-estimate memories with Gaussian clustering outperformed those using the

simpler Euclidean clustering.

Several variations of the default density-estimate memory were tested to explore other ways the density-estimate models stored in the memory might improve performance. An informed diversity measure used these models to replace the normal generation of the search population in the memory/search multi-population algorithm. While it showed a slight improvement in average fitness, the improvement of the informed diversity measure over the normal uninformed diversity technique was not significant. A simple reclustering algorithm and the introduction of fitness values both had negative effects on performance.

When parameters controlling the search space were varied, density-estimate memory continued to outperform self-organizing scouts while changes continued to be severe and the search space was composed of many wide peaks. By varying the peak width and number of peaks, both of which control the density of good solutions in the search space, a clear crossover point was shown between areas where density-estimate memory performs best and areas where self-organizing scouts performs best. Based on these results, density-estimate memory seems best suited for problems with a high density of good solutions and severe changes in the environment.

Chapter 8 applied density-estimate memory to an adaptive, traffic-responsive algorithm for traffic signal control. Two density-estimate memory implementations—incremental Gaussian clustering and a Gaussian mixture model—were described. Adaptive traffic signal controllers were compared to the real signal timing plans for a 32 intersection traffic network modeled on downtown Pittsburgh, Pennsylvania. By augmenting the adaptive algorithm with density-estimate memory, average vehicle speed increased and average wait time decreased. Each traffic signal maintained its own control system and memory; very little information needs to be communicated between intersections for this approach. Memory improved performance both during transitions and overall.

Though the adaptive algorithm was slightly worse than the fixed timing plan over an entire day on average, the use of density-estimate memory allowed the adaptive system to outperform the fixed timing plan. For some periods, such as the afternoon rush, the fixed timing plan was so much better than the adaptive algorithm that memory could not improve performance enough to be competitive. When the underlying adaptive algorithm was competitive, the use of density-estimate memory produced higher speeds than the fixed timing plan. These results suggest that adding memory to a better-performing adaptive algorithm would make it possible to outperform the fixed timing plan during these periods.

The experiments in Part II show that density-estimate memory can improve both optimization and learning algorithms on problems with dynamic environments. Density-estimate memories outperform standard memories as well as other state-of-the-art techniques for dynamic problems by providing rich models of the dynamic search space. These density-estimate models are refined over time as new solutions are added to the memory. Though density-estimate memories incorporate

many previous solutions, the overhead required to store to and retrieve from the memory remains consistently low. Only the computation required to rebuild models increases, and for incremental clustering density-estimate memories, this computation is not expensive.

Dynamic problems are often composed of several types of changes. Both the factory coordination problem and the adaptive traffic control problem have both small, continuous changes—changes in the distribution of jobs currently waiting to be processed or traffic flows of the cars currently at an intersection—and severe, discontinuous changes—large shifts in underlying distribution of job types or large time-of-day shifts in traffic flows. Optimization and learning algorithms tend to be well suited to adapting solutions for smaller changes, but may take a long time to adapt to more severe changes. Memory may help considerably in speeding up the adaptation to severe, discontinuous changes. The results in Part II suggest that density-estimate memories are best suited for problems with a high density of good solutions and severe, discontinuous changes in the dynamic environment.

Part III considered the use of memory for dynamic problems with shifting feasible regions. Chapter 9 introduced classifier-based memory for problems where the feasible region of the search space shifts over time. This chapter described a memory enhanced evolutionary algorithm approach to the dynamic job shop scheduling problem. Classifier-based memory enables the mapping of information about jobs at one point in time to the creation of valid schedules at another point in time. The results from this chapter show that classifier-based memory can improve both the quality of schedules and the speed of search over a standard evolutionary algorithm. These results also show that diversity techniques, which have had success on many other dynamic problems, show decreased fitness and search speed for the dynamic scheduling problem investigated here.

10.2 Contributions

The major contributions of this thesis are methods for enhancing memory for improving optimization and learning in dynamic environments. Novel enhanced memory systems were presented to address the weaknesses and overcome the limitations of standard memory systems. The techniques presented in this thesis improve memories by incorporating probabilistic models of previous solutions into memory, storing many previous solutions in memory while keeping overhead low, building long-term models of the dynamic search space over time, allowing easy refinement of memory entries, and mapping previous solutions to the current environment for problems where solutions may become obsolete. This thesis contributes two new classes of memory that enable these improvements: density-estimate memory and classifier-based memory.

Density-estimate memory was introduced to allow memories to effectively store many previous solutions without substantially increasing the overhead associated with maintaining and using the

memory. Density-estimate memories build and maintain probabilistic models of past solutions to improve the performance of learning and optimization in dynamic environments. While building much richer models of the dynamic search space than a standard memory, density-estimate memories can be constructed efficiently and maintained with a low amount of overhead. Density-estimate memory also builds better long-term models of the dynamic search space and makes it easier to refine memory entries.

Classifier-based memory was introduced to extend the use of memory to dynamic problems where solutions may become obsolete as the environment changes. Classifier-based memory creates an abstraction layer between feasible solutions and memory entries so that old solutions stored in memory may be mapped to solutions that are feasible in the current environment. Classifier-based memory allows dynamic problems with shifting feasible regions to use memory to improve search.

This thesis contributes several implementations of density-estimate memory: incremental Euclidean clustering density-estimate memory, incremental Gaussian clustering density-estimate memory, and Gaussian mixture model density-estimate memory. An implementation of classifier-based memory for job-shop scheduling is presented. This thesis also contributes a traffic-responsive learning algorithm for traffic signal control, the balanced phase utilization algorithm.

Finally, this thesis defines several problems that may be used as benchmarks to evaluate the performance of optimization and learning algorithms in dynamic environments. A distributed, dynamic factory coordination problem with long time horizons is defined in Chapter 6. An adaptive traffic control problem that models 32 intersections in the city of Pittsburgh, PA is defined in Chapter 8. A dynamic scheduling problem with sequence-based setups and machine breakdowns is defined in Chapter 9.

10.3 Outlook

This thesis presented two novel classes of memory that improve optimization and learning algorithms in dynamic environments. Though the memories implemented in this thesis outperformed state-of-the-art techniques, there is room for improvement in both density-estimate memory and classifier-based memory. This final section discusses several avenues for future work using the techniques introduced in this thesis.

Most density-estimate memory experiments in this thesis used relatively simple Euclidean and Gaussian clustering models to perform density-estimation. More sophisticated probabilistic models from the estimation of distribution algorithm literature—such as Gaussian mixture models, hierarchical Bayes networks, and dependency trees—could be used to improve the quality of density-estimate memory. For the experiments in Chapter 8, a Gaussian mixture model density-estimate

memory improved performance over an incremental Gaussian clustering density-estimate memory. Using more complex models does have the potential to increase the overhead required to maintain the memory, but for problems that allow sufficient time to rebuild more complex density-estimate models, these models could provide further improvements in the performance of density-estimate memory.

One of the strengths of density-estimate memory is the long-term model of the dynamic search space that it creates over time. By retrieving solutions from this model, density-estimate memory is able to improve the performance of an underlying search algorithm. However, this model could be used in many other ways. More informed diversity methods than the one developed in Chapter 7 could help population-based search, as could sampling from the models in memory.

Since density-estimate memory can store so many solutions, the effect of adding a new solution to memory decreases the more points have previously been stored. In addition to adding points to memory, it might be beneficial to remove points that become outliers to the model or are very old. Thinning out the memory by removing points very close to one another might also be beneficial for some problems.

For the adaptive traffic control experiments in Chapter 8, the underlying balanced phase utilization algorithm was one of the largest limitations on the performance of density-estimate memory. This algorithm performed well, but a better learning algorithm would probably lead to better performance of the density-estimate memory.

This thesis presented one possible implementation of classifier-based memory, other approaches to classifying jobs and mapping those classifications back to valid schedule might improve the quality of the information stored in memory. Finally, future work might combine classifier-based memory with other techniques like anticipation or density-estimate memory.

Bibliography

[1] Gregory J. Barlow and Choong K. Oh. Robustness analysis of genetic programming controllers for unmanned aerial vehicles. In *Genetic and Evolutionary Computation Conference*, pages 135–142, 2006.

[2] Gregory J. Barlow and Choong K. Oh. Evolved navigation control for unmanned aerial vehicles. In *Frontiers in Evolutionary Robotics*, chapter 20, pages 353–378. I-Tech Education and Publishing, 2008.

[3] Gregory J. Barlow and Stephen F. Smith. A memory enhanced evolutionary algorithm for dynamic scheduling problems. In *Applications of Evolutionary Computation: EvoWorkshops 2008*, pages 606–615, 2008.

[4] Gregory J. Barlow and Stephen F. Smith. Using memory models to improve adaptive efficiency in dynamic problems. In *IEEE Symposium on Computational Intelligence in Scheduling*, pages 7–14, 2009.

[5] Claus N. Bendtsen and Thiemo Krink. Dynamic memory model for non-stationary optimization. In *Congress on Evolutionary Computation*, pages 145–150, 2002.

[6] Christian Bierwirth and Dirk C. Mattfeld. Production scheduling and rescheduling with genetic algorithms. *Evolutionary Computation*, 7(1):1–17, 1999.

[7] Christian Bierwirth, Dirk C. Mattfeld, and Herbert Kopfer. On permutation representations for scheduling problems. In *Parallel Problem Solving from Nature*, pages 310–318, 1996.

[8] Tim Blackwell. Swarms in dynamic environments. In *Genetic and Evolutionary Computation Conference*, pages 1–12, 2003.

[9] Tim Blackwell and Peter J. Bentley. Dynamic search with charged swarms. In *Genetic and Evolutionary Computation Conference*, pages 19–26, 2002.

[10] Tim Blackwell and Jürgen Branke. Multi-swarm optimization in dynamic environments. In *Applications of Evolutionary Computation: EvoWorkshops 2004*, pages 489–500, 2004.

[11] Peter A.N. Bosman. Learning and anticipation in online dynamic optimization. In *Evolutionary Computation in Dynamic and Uncertain Environments*, pages 129–152. Springer, 2007.

[12] Peter A.N. Bosman and Han La Poutre. Learning and anticipation in online dynamic optimization with evolutionary algorithms. In *Genetic and Evolutionary Computation Conference*, pages 1165–1172, 2007.

[13] Jürgen Branke. Memory enhanced evolutionary algorithms for changing optimization problems. In *Congress on Evolutionary Computation*, pages 1875–1882, 1999.

[14] Jürgen Branke. *Evolutionary Optimization in Dynamic Environments*. Kluwer, 2002.

[15] Jürgen Branke, Thomas Kaußler, Christian Schmidt, and Hartmut Schmeck. A multi-population approach to dynamic optimization problems. In *Adaptive Computing in Design and Manufacturing*, pages 299–308, 2000.

[16] Jürgen Branke and Dirk C. Mattfeld. Anticipatory scheduling for dynamic job shop problems. In *AIPS Workshop on On-line Planning and Scheduling*, pages 3–10, 2002.

[17] Jürgen Branke and Dirk C. Mattfeld. Anticipation and flexibility in dynamic scheduling. *International Journal of Production Research*, 43(15):3103–3129, 2005.

[18] Jürgen Branke, Merve Orbayi, and Sima Uyar. The role of representations in dynamic knapsack problems. In *Applications of Evolutionary Computing: EvoWorkshops 2006*, pages 764–775, 2006.

[19] Mike Campos, Eric Bonabeau, Guy Theraulaz, and Jean-Louis Deneubourg. Dynamic scheduling and division of labor in social insects. *Adaptive Behavior*, 8(2):83–96, 2000.

[20] Anthony Carlisle and Gerry Dozier. Adapting particle swarm optimisation to dynamic environments. In *International Conference on Artificial Intelligence*, pages 429–434, 2000.

[21] Walter Cedeno and V. Rao Vemuri. On the use of niching for dynamic landscapes. In *International Conference on Evolutionary Computation*, pages 361–366, 1997.

[22] Pei-Chann Chang, Jih-Chang Hsieh, and Yen-Wen Wang. Genetic algorithm and case-based reasoning applied to production scheduling. In *Knowledge Incorporation in Evolutionary Computation*, pages 215–236. Springer, 2005.

[23] Alpha C. Chiang. *Elements of Dynamic Optimization*. Waveland Press, 1992.

[24] George Chryssolouris and Velusamy Subramaniam. Dynamic scheduling of manufacturing job shops using genetic algorithms. *Journal of Intelligent Manufacturing*, 12:281–293, 2001.

[25] Vincent A. Cicirello and Stephen F. Smith. Wasp-like agents for distributed factory coordination. *Autonomous Agents and Multi-Agent Systems*, 8(3):237–266, 2004.

[26] Helen G. Cobb. An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments. Technical Report AIC-90-001, Naval Research Laboratory, Washington, DC, 1990.

[27] Phillipe Collard, Cathy Escazut, and Alessio Gaspar. An evolutionary approach for time dependent optimization. *International Journal on Artificial Intelligence Tools*, 6(4):665–695, 1997.

[28] Richard W. Conway, William L. Maxwell, and Louis W. Miller. *Theory of Scheduling*. Dover, 1967.

[29] Gregory W. Corder and Dale I. Foreman. *Nonparametric Statistics for Non-Statisticians*. Wiley, 2009.

[30] Deborah Curtis. Adaptive control software. Technical Report HRTS-04-037, U.S. Department of Transportation, Federal Highway Administration, 2004.

[31] Dipankar Dasgupta and Douglas R. McGregor. Nonstationary function optimization using the structured genetic algorithm. In *Parallel Problem Solving from Nature*, pages 145–154, 1992.

[32] Arthur P. Dempster, Nan M. Laird, and Donald B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[33] Jeroen Eggermont and Tom Lenaerts. Dynamic optimization using evolutionary algorithms with a case-based memory. In *Belgium-Netherlands Conference on Artificial Intelligence*, pages 107–114, 2002.

[34] Jeroen Eggermont, Tom Lenaerts, Sanna Poyhonen, and Alexandre Termier. Raising the dead: Extending evolutionary algorithms with a case-based memory. In *European Conference on Genetic Programming*, pages 280–290, 2001.

[35] Marco Farina, Kalyanmoy Deb, and Faolo Amato. Dynamic multiobjective optimization problems: Test cases, approximations, and applications. *IEEE Transactions on Evolutionary Computation*, 8(5):425–442, 2004.

[36] Carlos M. Fernandes, Cláudio F. Lima, and Agostinho C. Rosa. UMDAs for dynamic optimization problems. In *Genetic and Evolutionary Computation Conference*, pages 399–406, 2008.

[37] Alessio Gaspar and Phillipe Collard. Time dependent optimization with a folding genetic algorithm. In *International Conference on Tools for Artificial Intelligence*, pages 207–214, 1997.

[38] Bernard Giffler and Gerald L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8(4):487–503, 1960.

[39] Matthew R. Glickman and Katia Sycara. Evolutionary search, stochastic policies with memory, and reinforcement learning with hidden state. In *International Conference on Machine Learning*, pages 194–201, 2001.

[40] David E. Goldberg and Robert E. Smith. Nonstationary function optimization using genetic algorithm with dominance and diploidy. In *International Conference on Genetic Algorithms*, pages 59–68, 1987.

[41] Robert L. Gordon and Warren Tighe. Traffic control systems handbook. Technical Report FHWA-HOP-06-006, U.S. Department of Transportation, Federal Highway Administration, 2005.

[42] John J. Grefenstette. Genetic algorithms for changing environments. In *Parallel Problem Solving from Nature*, pages 137–144, 1992.

[43] John J. Grefenstette and Connie L. Ramsey. An approach to anytime learning. In *International Conference on Machine Learning*, pages 189–195, 1992.

[44] Michael Guntsch. *Ant Algorithms in Stochastic and Multi-Criteria Environments*. PhD thesis, Universität Karlsruhe (TH), Institut AIFB, Karlsruhe, Germany, 2004.

[45] Michael Guntsch and Martin Middendorf. Applying population based ACO to dynamic optimization problems. In *International Workshop on Ant Algorithms*, pages 111–122, 2002.

[46] Tim Hendtlass, Irene Moser, and Marcus Randall. Dynamic problems and nature inspired meta-heuristics. In *Biologically-Inspired Optimisation Methods*, volume 210 of *Studies in Computational Intelligence*, pages 79–109. Springer Berlin / Heidelberg, 2009.

[47] Robert Herman. Technology, human interaction, and complexity: Reflections on vehicular traffic science. *Operations Research*, 40(2):199–212, 1992.

[48] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[49] Xiaohui Hu and Russell C. Eberhart. Adaptive particle swarm optimisation: detection and response to dynamic systems. In *Congress on Evolutionary Computation*, pages 1666–1670, 2002.

[50] Mohsen Jahangirian and G V. Conroy. Intelligent dynamic scheduling system: the application of genetic algorithms. *Intelligent Manufacturing Systems*, 11(4):247–257, 2000.

[51] Stefan Janson and Martin Middendorf. A hierachical particle swarm optimizer for dynamic optimization problems. In *Applications of Evolutionary Computing: EvoWorkshops 2004*, pages 513–524, 2004.

[52] Yaochu Jin and Jürgen Branke. Evolutionary optimization in uncertain environments—a survey. *IEEE Transactions on Evolutionary Computation*, 9(3):303–317, 2005.

[53] Yaochu Jin and Bernard Sendhoff. Constructing dynamic optimization test problems using the multi-objective optimization concept. In *Applications of Evolutionary Computation: EvoWorkshops 2004*, pages 525–536, 2004.

[54] Peter Kall and Stein W. Wallace. *Stochastic Programming*. John Wiley and Sons, 1994.

[55] Aydin Karaman, Sima Uyar, and Gülsen Eryigit. The memory indexing evolutionary algorithm for dynamic environments. In *Applications of Evolutionary Computation: EvoWorkshops 2005*, pages 563–573, 2005.

[56] Milos Kobliha, Josef Schwarz, and Jiri Ocenasek:. Bayesian optimization algorithms for dynamic problems. In *Applications of Evolutionary Computation: EvoWorkshops 2006*, pages 800–804, 2006.

[57] Gabriella Kokai, Tonia Christ, and Hans Holm Frhauf. Using hardware-based particle swarm method for dynamic optimization of adaptive array antennas. In *Adaptive Hardware and Systems*, pages 51–58, 2006.

[58] Daniel Krajzewicz, Georg Hertkorn, Christian Rossel, and Peter Wagner. SUMO (Simulation of Urban MObility); an open-source traffic simulation. In *Middle East Symposium on Simulation and Modelling*, pages 183–187, 2002.

[59] Stefan Lämmer and Dirk Helbing. Self-control of traffic lights and vehicle flows in urban road networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(4):P04019, 2008.

[60] Pedro Larranaga and Jose A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Springer, 2002.

[61] Young Hoon Lee, Kuman Bhaskaran, and Michael Pinedo. A heuristic to minimize the total weighted tardiness with sequence dependent setups. *IIE Transactions*, 29(1):45–52, 1997.

[62] Jonathan Lewis, Emma Hart, and Graeme Ritchie. A comparison of dominance mechanisms and simple mutation on non-stationary problems. In *Parallel Problem Solving from Nature*, pages 139–148, 1998.

[63] Shyh-Chang Lin, Erik D. Goodman, and William F. Punch. A genetic algorithm approach to dynamic job shop scheduling problems. In *International Conference on Genetic Algorithms*, pages 481–488, 1997.

[64] Sushil J. Louis and John McDonnell. Learning with case-injected genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 8(4):316–328, 2004.

[65] Dirk C. Mattfeld and Christian Bierwirth. An efficient genetic algorithm for job shop scheduling with tardiness objectives. *European Journal of Operations Research*, 155:616–630, 2004.

[66] Geoffrey McLachlan and David Peel. *Finite mixture models*. Wiley, 2000.

[67] Pitu Mirchandani and Larry Head. A real-time traffic signal control system: architecture, algorithms, and analysis. *Transportation Research Part C — Emerging Technologies*, 9(6):415–432, 2001.

[68] Kazuo Miyashita and Katia Sycara. CABINS: a framework of knowledge acquisition and iterative revision for schedule improvement and reactive repair. *Artificial Intelligence*, 76(1-2):377–426, 1995.

[69] Naoki Mori, Hajime Kita, and Yoshikazu Nishikawa. Adaptation to a changing environment by means of the thermodynamical genetic algorithm. In *Parallel Problem Solving from Nature*, pages 513–522, 1996.

[70] Richard Morley. Painting trucks at General Motors: The effectiveness of a complexity-based approach. In *Embracing Complexity: Exploring the Application of Complex Adaptive System to Business*, pages 53–58. Ernst and Young Center for Business Innovation, 1996.

[71] Richard Morley and C. Schelberg. An analysis of a plant-specific dynamic scheduler. In *Proceedings of the NSF Workshop on Dynamic Scheduling*, pages 115–122, 1993.

[72] Ronald Morrison and Kenneth DeJong. A test problem generator for non-stationary environments. In *Congress on Evolutionary Computation*, pages 2047–2053, 1999.

[73] Khim Peow Ng and Kok Cheong Wong. A new diploid scheme and dominance change mechanism for non-stationary function optimization. In *International Conference on Genetic Algorithms*, pages 159–166, 1995.

[74] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics*. MIT Press, 2000.

[75] Shervin Nouyan, Roberto Ghizzioli, Mauro Birattari, and Marco Dorigo. An insect-based algorithm for the dynamic task allocation problem. *Künstliche Intelligenz*, 4:25–31, 2005.

[76] Djamila Ouelhadj and Sanja Petrovic. Survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*, 12(4):417–431, 2009.

[77] Peng Si Ow and Thomas E. Morton. The single machine early/tardy problem. *Management Science*, 35(2):177–191, 1989.

[78] Daniel Parrott and Xiaodong Li. A particle swarm model for tracking multiple peaks in a dynamic environment using speciation. In *Congress on Evolutionary Computation*, pages 98–103, 2004.

[79] Holger Prothmann, Jürgen Branke, Hartmut Schmeck, Sven Tomforde, Fabian Rochner, Jörg Hähner, and Christian Müller-Schloer. Organic traffic light control for urban road networks. *International Journal of Autonomous and Adaptive Communications Systems*, 3(2):203–225, 2009.

[80] Narayan Raman, Ram V. Rachamadugu, and F. Brian Talbot. Real-time scheduling of an automated manufacturing center. *European Journal of Operational Research*, 40(2):222–242, 1989.

[81] Connie L. Ramsey and John J. Grefenstette. Case-based initialization of genetic algorithms. In *International Conference on Genetic Algorithms*, pages 84–91, 1993.

[82] Hendrik Richter and Shengxiang Yang. Learning in abstract memory schemes for dynamic optimization. In *International Conference on Natural Computation*, pages 86–91, 2008.

[83] Hendrik Richter and Shengxiang Yang. Memory based on abstraction for dynamic fitness functions. In *Applications of Evolutionary Computing: EvoWorkshops 2008*, pages 596–605, 2008.

[84] Hendrik Richter and Shengxiang Yang. Learning behavior in abstract memory schemes for dynamic optimization problems. *Soft Computing*, 13(12):1163–1173, 2009.

[85] Dennis I. Robertson and R. David Bretherton. Optimizing networks of traffic signals in real time—the SCOOT method. *IEEE Transactions on Vehicular Technology*, 40(1):11–15, 1991.

[86] Conor Ryan. The degree of oneness. In *European Conference on Artificial Intelligence Workshop on Genetic Algorithms*, 1994.

[87] Conor Ryan. Diploidy without dominance. In *Nordic Workshop on Genetic Algorithms*, pages 45–52, 1997.

[88] David Schrank, Tim Lomax, and Shawn Turner. Annual urban mobility report. Technical report, Texas Transportation Institute, Texas A&M University System, 2010.

[89] Steven G. Shelby, Darcy M. Bullock, Doug Gettman, Raj S. Ghaman, Ziad A. Sabra, and Nils Soyke. An overview and performance evaluation of ACS Lite — a low cost adaptive signal control system. In *Transportation Research Board Annual Meeting*, 2008.

[90] Arthur G. Sims and K. W. Dobinson. The Sydney coordinated adaptive traffic (SCAT) system philosophy and benefits. *IEEE Transactions on Vehicular Technology*, 29(2):130–137, 1980.

[91] Stephen A. Stanhope and Jason M. Daida. (1+1) genetic algorithm fitness dynamics in a changing environment. In *Congress on Evolutionary Computation*, pages 1851–1858, 1999.

[92] Rasmus K. Ursem. Multinational evolutionary algorithms. In *Congress on Evolutionary Computation*, pages 1633–1640, 1999.

[93] Rasmus K. Ursem. Multinational GAs: Multimodal optimization techniques in dynamic environments. In *Genetic and Evolutionary Computation Conference*, pages 19–26, 2000.

[94] Pascal Van Hentenryck and Russell Bent. *Online Stochastic Combinatorial Optimization*. MIT Press, 2006.

[95] Frank Vavak, Terrence C. Fogarty, and Ken Jukes. A genetic algorithm with variable range of local search for tracking changing environments. In *Parallel Problem Solving from Nature*, pages 376–385, 1996.

[96] Frank Vavak, Ken Jukes, and Terrence C. Fogarty. Learning the local search range for genetic optimisation in nonstationary environments. In *IEEE International Conference on Evolutionary Computation*, pages 355–360, 1997.

[97] Frank Vavak, Ken Jukes, and Terrence C. Fogarty. Performance of a genetic algorithm with variable local search range relative to frequency for the environmental changes. In *International Conference on Genetic Programming*, pages 602–608, 1998.

[98] Manuel Vazquez and L. Darrell Whitle. A comparison of genetic algorithms for the dynamic job shop scheduling problem. In *Genetic and Evolutionary Computation Conference*, pages 1011–1018, 2000.

[99] Ari P.J. Vepsalainen and Thomas E. Morton. Priority rules for job shops with weighted tardiness costs. *Management Science*, 33(8):1035–1047, 1987.

[100] Axel Wegener, Michal Piórkowski, Maxim Raya, Horst Hellbrück, Stefan Fischer, and Jean-Pierre Hubaux. TraCI: An interface for coupling road traffic and network simulators. In *Communications and Networking Simulation Symposium*, pages 155–163, 2008.

[101] Mark Wineberg and Franz Oppacher. Enhancing the GA's ability to cope with dynamic environments. In *Genetic and Evolutionary Computation Conference*, pages 3–10, 2000.

[102] Xiao-Feng Xie, Gregory J. Barlow, Stephen F. Smith, and Zachary B. Rubinstein. Platoon-based self-scheduling for real-time traffic signal control. In *IEEE Conference on Intelligent Transportation Systems*, 2011.

[103] Xiao-Feng Xie, Gregory J. Barlow, Stephen F. Smith, and Zachary B. Rubinstein. Self-scheduling agents for real-time traffic signal control. Technical Report TR-RI-11-06, Robotics Institute, Carnegie Mellon University, 2011.

[104] Shengxiang Yang. Non-stationary problems optimization using the primal-dual genetic algorithm. In *Congress on Evolutionary Computation*, pages 2246–2253, 2003.

[105] Shengxiang Yang. Constructing dynamic test environments for genetic algorithms based on problem difficulty. In *Congress on Evolutionary Computation*, pages 1262–1269, 2004.

[106] Shengxiang Yang. Associative memory scheme for genetic algorithms in dynamic environments. In *Applications of Evolutionary Computing: EvoWorkshops 2006*, pages 788–799, 2006.

[107] Shengxiang Yang. Explicit memory schemes for evolutionary algorithms in dynamic environments. In *Evolutionary Computation in Dynamic and Uncertain Environments*. Springer, 2007.

[108] Shengxiang Yang. Genetic algorithms with elitism-based immigrants for changing optimization problems. In *Applications of Evolutionary Computing: EvoWorkshops 2007*, pages 627–636, 2007.

[109] Shengxiang Yang and Xin Yao. Population-based incremental learning with associative memory for dynamic environments. *IEEE Transactions on Evolutionary Computation*, 12(5):542–561, 2008.

[110] Bo Yuan, Maria Orlowska, and Shazia Sadiq. Extending a class of continuous estimation of distribution algorithms to dynamic problems. *Optimization Letters*, 2(3):433–443, 2008.