Generalized Density-Estimate Memory for Dynamic Problems

Gregory John Barlow

CMU-RI-TR-09-21

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

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

June 2009

Abstract

Optimization systems traditionally focus on static problems, also known as offline or a priori optimization problems. Many real-world problems may be better modeled as dynamic optimization problems, also known as stochastic, in situ, or online optimization problems. In these types of problems, the fitness landscape of the search space changes over time. While it may be possible to learn policies offline for every potential scenario, doing so may not be tractable or sufficiently robust. Instead, recent work has focused on optimization concurrent with dynamic processes in order to track moving optima.

One part of the field that has seen a great deal of work in the last decade is dynamic optimization with evolutionary algorithms (EAs). The population-based search used in most EAs has proven very useful for dynamic problems, since the population helps to ease optimization after changes in the fitness landscape. However, normal EAs encounter several difficulties with dynamic problems; foremost, the convergence of a population on a single peak in the fitness landscape depletes the diversity of the population, making adaptation after a change in the fitness landscape more difficult. One method for countering this problem has been the use of memory to retain good solutions for future use.

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 also provide additional points to search from after a change, and when the population has converged, may add a great deal of diversity to the population. Memory has been used extensively for dynamic optimization, and while there are many variations, a standard memory model has emerged. In this model, a small portion of the population is reserved for memory. For many problems, this type of memory has helped to improve the performance of dynamic optimization. However, because the memory size is finite, and usually small, it may be difficult to store enough solutions to accurately model the search space over time. Also, for problems where the feasible region of the search space changes with time, memory entries may become infeasible, making the memory less useful.

This proposal examines the development of more generalized models of memory for dynamic problems. A density-estimate memory for dynamic problems which 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 is proposed. By building models within the memory, the search process can continue to interact with a limited number of entries, except now these entries are models incorporating many individual solutions. The models stored in memory can be continuously refined by updating the models 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. An extension of memory to problems where the feasible region of the search space changes over time through the development of an indirect memory layer is also proposed, which will open up the use of memory to many dynamic problems which could not use information from the past. The indirect memory layer will use environmental information from the problem to create an indirect representation of a solution which can be stored in memory. The stored entry can be mapped to a feasible solution at a future time. The work proposed here will lead to a generalized memory which can be applied to many dynamic problems.

Contents

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, without the benefit of information from the past, but incorporating this kind of information into search 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 a memory capable of building rich models of past experiences, this work proposes to develop a more effective memory for dynamic optimization.

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, many 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 and fitness landscape 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 optimization, and while there are many variations, a standard memory model has emerged where a finite number of solutions are stored and incorporated into search as the problem changes.

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 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. Also, for problems where the feasible region of the search space changes with time, memory entries may become infeasible, making the memory less useful.

This work proposes the development of a density-estimate memory for dynamic problems, one which 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. By building models within the memory, the search process can continue to interact with a finite number of entries, except now these entries are models incorporating many individual solutions. The models stored in memory can be continuously refined by updating the models 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.

This work also proposes the extension of memory to problems where the feasible region of the search space changes over time through the development of an indirect memory layer. The indirect memory layer

will use environmental information from the problem to create an indirect representation of a solution which can be stored in memory. The stored entry can be mapped to a feasible solution at a future time. Creating this indirect memory layer will open up the use of memory to many dynamic problems which could not use information from the past.

This work will begin by discussing prior work in the area of dynamic optimization in Section 2. A great deal of work has been done on using evolutionary algorithms (EAs) for dynamic optimization, and in developing techniques which improve the performance of EAs on dynamic problems. An extensive overview of the use of memory for dynamic optimization is given.

The proposed work is to develop a generalized memory system to help solve dynamic problems. In Section 3, a standard memory is defined that can be used to describe most memories in the literature and is a first step toward a generalized memory. The strengths and weaknesses of the standard memory system are presented.

Sections 4.1 and 4.2 describe completed work addressing two weaknesses of the standard memory. Section 4.1 presents the extension of memory to dynamic scheduling with evolutionary algorithms, a problem where the feasible region of the search spaces changes over time. In Section 4.2, several model-building memories are proposed and memory is extended to a multi-agent reinforcement learning algorithm. This is the first step toward creating rich models of the dynamic problem in memory.

Section 5 presents proposed research and contributions. The overall proposed contribution is to develop a density-estimate memory system which addresses as strongly as possible the weaknesses that exist in the standard memory system. To that end, experiments are proposed and potential contributions discussed. Finally, a proposed research plan and schedule is presented in Section 6

2 Background

Dynamic optimization with evolutionary algorithms (EAs) 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 approaches. If a problem changes very slowly, a balance between optimization and diversity is no longer necessary: one may search from scratch, treating each change as a completely new problem. Many problems do lie in this region where optimization must respond quickly to changes while still finding solutions of high fitness.

2.1 Dynamic problems

In a dynamic problem, the objective function, problem formulation, constraints, or some other part of the problem changes over time such that the objective value of a given solution changes relative to some other solution. In many cases, this means that the optimum of the problem changes as well, so a good approach to dynamic problems must be capable of tracking the optima over time. 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 [40] and will not be used in this work.

When discussing dynamic problems in this work, a certain vocabulary will be employed. 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. A change will often 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.

One of the more common dynamic benchmarks problems for EAs is the moving peaks problem, developed simultaneously by Branke [11] and by Morrison and DeJong [53]. 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 [5, 47, 4, 72, 13]. While many problems consider primarily the arrival of jobs over time, some problems also include machine breakdowns and other dynamic events [21, 2]. Ouelhadj and Petrovic [57] give a survey of approaches to dynamic scheduling which includes some explanation of dynamic scheduling problems.

Other common benchmark problems include dynamic knapsack problems [15, 69] and dynamic bit-matching [65]. Farina et al. created several multiobjective dynamic benchmark problems [28]. Van Hentenryck and Bent consider dynamic routing in addition to scheduling and knapsack problems [69]. Several dynamic problem generators have been proposed in addition to the Moving Peaks problem. Yang created a problem generator based on decomposable trap functions [78], and Jin and Sendhoff created a benchmark generator for both single objective and multiobjective problems [41].

2.2 Methods for dynamic optimization

There are many approaches to dynamic optimization, and it is outside the scope of this work 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.

If one actually knows the function to be optimized, classical methods like calculus of variations or optimal control may be applied [20]. 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 [42], 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 [38]. These rules may be learned ahead of time. Other reactive approaches may also be learned a priori to be robust in varying environments [55, 1]. Approaches like reinforcement learning move that process of learning from dynamic events into the control process [22].

One large class of approaches that are commonly used for dynamic problems are metaheuristics, including local search [69], simulated annealing, tabu search, evolutionary algorithms [40], ant colony optimization, particle swarm optimization, and many others. Many of these algorithms were inspired by real dynamic processes like evolution, swarm behavior, or annealing. Metaheuristics are often advantageous because they typically do not require extensive knowledge of the function to be optimized, such as the derivatives of the function (some local search algorithms do require derivatives). In particular, dynamic optimization with evolutionary algorithms has been extensively studied in the literature. Though most of the results in the remainder of this section will discuss dynamic optimization with evolutionary algorithms, metaheuristic approaches tend to encounter the same types of difficulties with dynamic problems, and many of the approaches used for EAs could be applied to other metaheuristics.

2.3 Improving performance on dynamic problems

Many metaheuristics, 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. These metaheuristics 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 metaheuristics 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 metaheuristics 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 metaheuristics 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 metaheuristics on dynamic problems. [12, 40]. 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 overconvergence of search.

2.4 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 [40] 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 [23] 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 [70, 71] 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 [34, 33] 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 [50] uses an explicit diversity measure in combination with fitness to choose the new population. Sharing and crowding mechanisms [18] are another way to encourage diversity. A more recent approach to maintaining diversity is elitism-based immigrants [75]. 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, while keeping diversity high.

2.5 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.

2.5.1 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 EAs [32, 54, 63, 64, 46]. 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 GA [24], 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 [30] and Yang [77]. Dasgupta and McGregor created the structured GA [25], which uses a more complex structure of meta genes.

2.5.2 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 work, a memory is defined as an explicit memory bank and a memory entry is defined as a stored point within the memory.

In an early use of memory, Ramsey and Grefenstette [60] 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 [11, 12] 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 [12]. 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 [11] 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 [12].

Eggermont et al. [27] presented a memory similar to the memory in [11] with a least recently used replacement strategy. This work was extended by Eggermont and Lenaerts [26] by adding a predictor to the memory. However, this predictor was very simple and highly problem dependent.

Bendtsen and Krink [3] 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 EA on an example problem.

Chang et al. [19] 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. [43] presented a memory indexing EA 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 [60], 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 [74, 79, 76] 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 [62, 61] 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.

2.6 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 [10, 12], 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 GA [67, 68] and the shifting balance GA [73].

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.

2.7 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 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 [13, 14] 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 [69] 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 [69] provided substantial performance benefits.

Bosman [59] 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 [9] have presented work on anticipation for stochastic dynamic problems.

2.8 Other approaches

Other metaheuristic algorithms have been used for dynamic problems. Ant colony optimization has been used for the dynamic traveling salesman problem [36, 35] with several methods to repair and reinitialize the pheromone matrix after a change occurs. Particle swarm optimization has been widely used for dynamic optimization [17, 37, 7, 6, 8, 39, 58]. This metaheuristic 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 [37] or introducing charged particles [7]. These approaches tend to be focused on maintaining diversity. Since particle swarms already have some concept of memory, this may be helpful [17], though it does not seem to be as rich as the explicit memory described above. Multi-swarm approaches have also been proposed [8, 58] as parallels of multi-population approaches.

Another metaheuristic which has seen some recent use for dynamic problems is estimation of distribution algorithms (EDAs) [44, 80, 29, 76]. EDAs are an outgrowth of EAs 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 [45]. The work to date on using EDAs 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 [76] have also considered the use of associative memory with an EDA for dynamic problems.

3 Memory for dynamic optimization

Prior work has shown that retaining information from the past often helps dynamic optimization adapt better 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 section, 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. This standard memory will also give the reader some idea where the work in this proposal is headed.

Incorporating memory into algorithms for solving dynamic problems requires a balancing act. While some types of memory may be capable of drastically improving the fitness of solutions, improvements may come at the expense of how quickly these fitter 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. This remainder of the section will attempt to examine the strengths and weaknesses of memory.

3.1 Standard memory system

Many variants of memory exist in the literature, but most of these variants are based on a standard model of explicit memory, described in [12] and [79]. In this section, a standard memory system for population-based search algorithms is defined and default settings are given. Since most uses of this type of memory in the literature use EAs, the memory will be defined in terms of its use for dynamic optimization with EAs. However, as will be shown later in this work, this standard memory system can be extended to other optimization and learning algorithms. This standard memory may be seen as a building block that can be altered in many ways to improve performance and as a first step toward a density-estimate memory capable of providing rich models of the dynamic fitness landscape.

3.1.1 Structure

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 which is generally small relative to the total size of the population—the most common practice 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.

Information stored in a memory entry can be divided into two categories: state information and control information. State information is used to maintain the memory, deciding which entries should be stored in the memory when new entries become available. Typically, state information is used to calculate the distance between memory entries so a diverse memory may be maintained. Control information is used to have some effect on the normal optimization process. For example, in population-based search, the control information might just be a solution stored at a previous time which is then reinserted into the population.

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 state 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 state 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 improve performance. For example, in associative memory [74], an individual from the population is stored as state 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.

3.1.2 Storing solutions in the 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.

First, 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 [11, 27, 12]. 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 [12] are probably most common.

3.1.3 Retrieving solutions from the memory

Memory can be retrieved in one of two ways: 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. 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. Memory remains separate from the search process. While search operators may alter copies of memory entries reinserted into the population, the stored entries in the memory are not changed. Most memories seem to retrieve after a change, so that will be used as a default.

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 individuals are generated based on the individuals stored in the memory entries—often by sampling a stored distribution of solutions—and then inserted into the population [43, 74]. It is also feasible to only retrieve some of the individuals from the memory at a given time.

3.2 Memory strengths

Memory has proven 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. Memory helps to provide diversity in the population, leads search toward promising areas, develops a simple model of where good areas exist in the search space, and does all this with a limited amount of overhead.

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.

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 proven 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.

On a similar note, standard memory provides a model of where good solutions exist over time. Though this model is immediately useful for leading search toward promising areas, a model of the dynamic landscape over time might be helpful in anticipating future landscapes, developing better diversity mechanisms, or any number of other improvements to dynamic optimization. Given a small memory size, this model is very crude, but it does provide an idea of where solutions have been best over time.

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.

3.3 Memory weaknesses

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 piecemeal. It should be possible to address many or all of these weaknesses, so this section can be seen as a road map for where memory could go in the future.

It was mentioned above that memory helps to build a model of the good areas of the dynamic fitness problem over time. While this is true, the model the standard memory builds 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 [62] have actually 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.

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.

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. Techniques like self-organizing scouts [10, 12] help to counter this problem, though multi-population approaches draw resources away from search in other ways.

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 [12]. 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.

The standard memory just reinserts the individuals in the memory entries into the population. As some memories have shown [74, 79], it may often be helpful to be more selective about which entries are returned. By selecting only those entries which look the most useful, however, diversity may be lost, and memory may actually be less useful. How to choose the memory entries which not only best suit the current context, but provide diversity to help lead search toward better solutions is still an open question.

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 Completed research

This section will describe two completed areas of research intended to begin to address the weaknesses of standard memory models. First, for problems with variable-length solutions where the feasible region of the search space shifts over time, solutions stored by a standard memory quickly become infeasible and useless in helping to solve a changing problem. An indirect memory layer was developed to use information from old solutions to create solutions in a changed environment. This indirect memory—classifier-based memory—

was applied to an evolutionary algorithm approach to a dynamic job shop scheduling problem. Second, many weaknesses of the standard memory model stem from the limited ability of the memory to model the fitness landscape over time. By building probabilistic models based on many solutions and storing those in the memory, rather than storing just a few solutions, memory could potentially be much more powerful. A first generation of these density estimation memories were developed and applied to a multi-agent reinforcement learning algorithm for a dynamic scheduling problem.

4.1 Memory for problems with shifting feasible regions

A variety of dynamic benchmark problems for EAs have been considered, including the moving peaks problem [11, 53], the dynamic knapsack problem, dynamic bit-matching, dynamic scheduling, and others [12, 40]. 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 [47, 4], problems with machine breakdowns and redundant resources [21], improved genetic operators [72], heuristic reduction of the search space [49], and anticipation to create robust schedules [13, 14]. While Louis and McDonnell [48] have shown that case-based memory is useful given similar static scheduling problems, there has been no work on memory for dynamic scheduling. Since the addition of memory has been successful in improving the performance of EAs 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 [11]. 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.

In this section, we present one such memory for dynamic scheduling, which we call 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. In the remainder of this section, we will describe classifier-based memory for dynamic scheduling problems and compare it to both a standard EA and to other approaches from the literature.

4.1.1 Dynamic job shop scheduling

The dynamic job shop scheduling problem used for our 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 we will refer 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. $\lambda = \bar{P}/(mU)$. 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. We consider a single objective, weighted tardiness. 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, we 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.

4.1.2 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. We will call the uncompleted operations of these jobs the set of pending operations $P = \{o_{j,k} \,|r_j \leq t, \neg complete(o_{j,k})\}$ 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, we say that the operation is schedulable. We define the set of schedulable operations as $S = \{o_{j,k} \,|o_{j,k} \in P, complete(o_{j,k-1})\}$.

Like most EA approaches to scheduling problems, we encode solutions 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.

We use the well known Giffler and Thompson algorithm [31] to build active schedules from a prioritized list. First, from the set of pending operations P we create the set of schedulable operations S. From S, we find the operation o' with the earliest completion time t_c. We 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. We then 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 EA is generational with a population of 100 individuals. We use the PPX crossover operator [5] 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 EA runs until the best individual in the population remains the same for 10 generations.

4.1.3 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 we 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 we 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. We present one such memory for dynamic scheduling, which we call classifier-based memory. Instead of storing prioritized lists of operations, we use an indirect representation, 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 several 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. Rather than storing the prioritized list of operations, we store a prioritized list of classifications as a memory entry. To retrieve an individual from a memory entry, we map the pending operations to a prioritized list of classifications. We rank each operation according to the same attributes, then determine quantiles and classify each operation. Then, for each of these new classification, we find the best match among the classifications in the memory entry. We assign each pending operation a sort 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, we use the average of the positions as the sort key. Then, we sort the pending operations by these sort keys to create a prioritized list of operations which can be used as an individual for the EA.

The basic mechanisms for interacting with the memory are the same as those for other explicit memories used for dynamic optimization with evolutionary algorithms. At every generation, we create an individual from each memory entry and insert the individuals 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 [12]. To maintain diversity in the memory, we determine 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. We then choose the less fit list j 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, we use the mean of the positions. 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, we replace classification list j 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 1 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 $class(A) \rightarrow 101$. At time 10000, we would like to use the memory entry to create a prioritized list from the four pending operations. These new operations are classified and given a score based on their best match within the memory entry, creating a new individual to be inserted into the population.

This classifier-based memory also allows new jobs to be ordered alongside older jobs that may have been

At $t = 400$, store $[C, B, A, D]$	At $t = 10000$, get $[011, 000, 101, 110]$
$A = \{dd : 800, pt : 100, w : 7\} \rightarrow 101$	$W = \{dd : 10400, pt : 80, w : 1\} \rightarrow 110 \rightarrow (3)$
$B = \{dd : 450, pt : 110, w : 5\} \rightarrow 000$	$X = \{dd : 10100, pt : 70, w : 5\} \rightarrow 011 \rightarrow (0)$
$C = \{dd : 500, pt : 130, w : 9\} \rightarrow 011$	$Y = \{dd : 10500, pt : 50, w : 6\} \rightarrow 101 \rightarrow (2)$
$D = \{dd : 900, pt : 150, w : 3\} \rightarrow 110$	$Z = \{dd : 10070, pt : 60, w : 2\} \rightarrow 000 \rightarrow (1)$
$[011, 000, 101, 110] \rightarrow$ memory	$[X, Z, Y, W] \rightarrow$ population

Figure 1: Classifier-based memory example

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 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.

In this section, we use four attributes ($a = 4$): job due-date, job weight, operation processing time, and operation order within the job. We divide rankings 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.

4.1.4 Experiments

To examine the effects of classifier-based memory on schedule fitness and search time, we compare several common approaches. We use a standard evolutionary algorithm (SEA) as a baseline, since we don't know the optimal schedules for any of the problem instances, and we also consider the standard EA with classifier-based memory (SEAm). Prior results on benchmarks like the moving peaks problem suggest that memory-based approaches work better when combined with a diversity strategy [12]. Hence, we also consider a standard EA with 25 random immigrants [33] per generation (RI) and the same approach with classifier-based memory (RIm). Finally, we consider the memory/search approach of [12], also using the classifier-based memory. 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, we select the utilization rate so that jobs arrive at approximately replacement rate, so the number of jobs available at time 0, ρ, is also the expected schedule size. We would like to be able to vary the due-date tightness to change the difficulty of the problem, so we use a due-date tightness parameter τ, the percentage of jobs we expect 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 $d_j = r_j + \bar{P} + \left[0, 2\rho\bar{P}\tau\right]$. The setup time severity is given by $\eta = \bar{s}/\bar{p}$ 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 section, we used the following settings 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, we varied the due-date tightness of the jobs. As the due-date tightness changes, the types of situations the scheduler faces

Table 1: Fitness improvement over the standard EA

	$\tau = 0.5$	$\tau = 0.8$	$\tau = 1.1$
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%

also change. We 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 we expect most jobs to be on time. For each value of τ, we created 10 problem instances, for a total of 30 problem instances.

Rather than rebuild the memory from scratch on every problem instance during our experiments, we pre-built several seed memories using SEAm over a larger number of jobs, varying the due-date tightnesses of the jobs. Though we test over a limited number of jobs, if actually implemented in a scheduling system, the EA would work over a long time horizon, and so we are more interested in the steady state performance of the EA. By pre-building the memory, we better simulate this state of the algorithm. 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 EA with memory, one of the pre-built 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.

We performed simulation runs for each EA variant on each of the 30 problem instances. Since this is a dynamic problem, we are interested not just in fitness improvements but in improvements in the speed of search. As in [13, 14, 4], we attempt to measure only the steady state performance by discarding the first 100 and last 100 jobs. We use the summed weighted tardiness of the middle 300 jobs as the fitness. We measure search 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. We can compare search more fairly by comparing the number of optional generations per event.

4.1.5 Results

Table 1 shows the percentage of improvement in average fitness over the standard EA. 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 there the improvement for medium due-dates is slightly better than that for tight due-dates.

Table 2 shows the percentage improvement in average optional generations per event over the standard EA. 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 we consider 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.

Table 2: Search improvement over the standard EA

	$\tau = 0.5$	$\tau = 0.8$	$\tau = 1.1$
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%

For both fitness and search, the addition of classifier-based memory improved performance over the standard EA. While significant improvement in fitness was only evident for loose due-dates, search improved for most problem instances. We saw improvement using SEAm for all three values of τ, but we saw the least improvement for $\tau = 0.8$. Our belief is that of the three, 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 individual to include in the memory, is at a disadvantage in the steady state environment we are interested in, though this approach might still be useful for pre-building memories, where search time is not an issue.

4.1.6 Conclusions

This section describes 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. We describe a classifier-based memory that enables the mapping of information about jobs at one point in time to the creation of valid schedules at another point in time. We compared several EA variants, with and without memory, on problem instances of varied difficulty. Our results show that classifier-based memory can improve both schedule fitness and the speed of search over a standard evolutionary algorithm. Our 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 we investigated.

We did not consider 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. We have also made no attempt to finely tune the EA used by each approach, following the example of [12]. Given the lack of prior work on memory enhanced EAs for dynamic scheduling, 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.

4.2 Building distribution models in memory

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. In this section, we introduce several density-estimate memory systems inspired by estimation of distribution algorithms that improve upon standard memory systems while avoiding large increases in overhead.

We evaluate the performance of these new types of memories on an agent-based system for distributed factory coordination [22]. 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 one type of product to another, a setup time is incurred.

In this section, we will explain the distributed factory coordination problem and several agent-based approaches to the problem. We 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. We will introduce the use of memory to improve the performance of R-Wasps on the new dynamic distributed factory coordination problem and present several novel density-estimate memory systems. We will then compare the performance of the baseline system with the memory-augmented systems.

4.2.1 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 [52, 51]. 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 determine 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 [52]. This system saved almost a million dollars in the first nine months of use [51]. 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. [16] and Cicirello and Smith [22] 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. [56] and others examine these and similar approaches.

Cicirello and Smith [22] 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 [22] 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 examine problems with 2 job types and both 2 and 4 machines. They examine 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.

4.2.2 Problem

In this section, we examine a dynamic extension of the distributed factory coordination problem similar to the variant from Cicirello and Smith [22] described above. The performance of R-Wasps is evaluated over a much longer time horizon, where the underlying distribution of the job types changes many times.

Dynamic factory coordination 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 [22], 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.

R-Wasps We use R-Wasps as described in [22] 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:

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

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

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

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

$$\theta_{w,j} = \theta_{w,j} - \delta_1 \tag{3}$$

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

$$\theta_{w,j} = \theta_{w,j} + \delta_2 \tag{4}$$

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

$$F_w = 1.0 + T_p + T_s \tag{5}$$

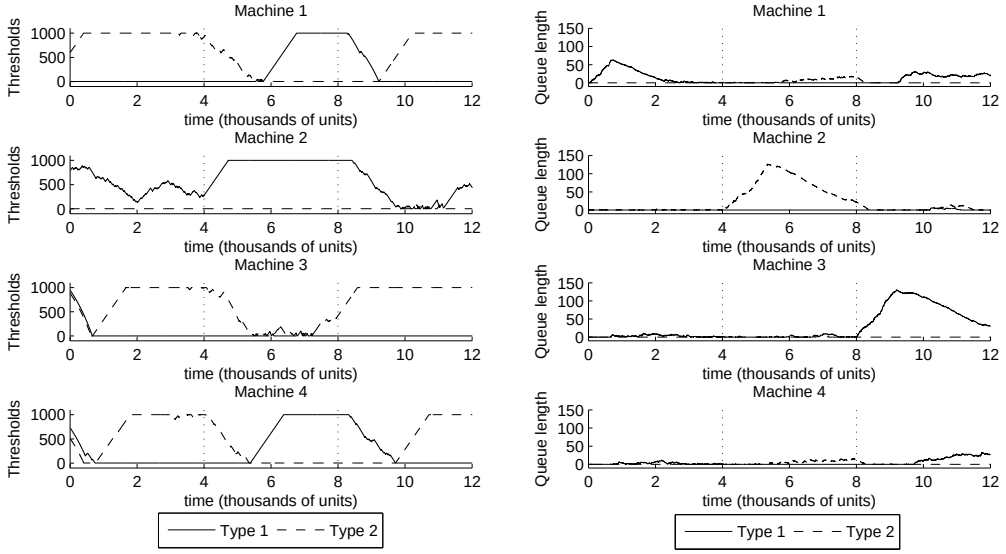
where T_p and T_s are the sum of the process times and setup times respectively of all jobs in the machine's queue. Let F_1 and F_2 be the forces of agents 1 and 2. Then, agent 1 will win the dominance contest with probability

$$P(Agent\,1\,wins|F_1, F_2) = \frac{F_2^2}{F_1^2 + F_2^2} \tag{6}$$

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 [22].

R-Wasp Weaknesses 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 2(a) 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 2(b), 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



(a) Thresholds

(b) Queue lengths

Figure 2: Thresholds and queue lengths for an example run

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.

4.2.3 Memory

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. We propose adding a memory to R-Wasps for use in the dynamic factory coordination problem described in Section 4.2.2.

Previous uses of memory Memory has been used extensively in dynamic problems, primarily for dynamic optimization with evolutionary algorithms (EAs). Memory helps to maintain diversity in the search after a change and to lead the search into promising regions. One early approach by Ramsey and Grefenstette was the use of case-based memory, where memory is used to initialize the population of an EA after a change [60]. Branke introduced a general explicit memory model [11] for dynamic optimization with EAs and then refined it in several ways, including a multi-population memory/search model [12]. Eggermont et al. introduced a case-based memory similar to Branke's, but with several options for retrieving entries from the memory [27, 26]. Chang et al. used case-based reasoning for period-based scheduling with an EA [19]. Barlow and Smith [2] investigated abstract memory for dynamic scheduling with an EA.

Several investigations into the use of memory have used or been inspired by estimation of distribution algorithms (EDAs), though primarily to completely reinitialize a population after a change. EDAs are an outgrowth of EAs 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 [45]. Karaman et al. [43] introduced a memory indexing EA and Yang [74, 79] introduced an associative memory, both of which use estimations of a population's distribution to reinitialize the population after a change. Richter and Yang [62] investigated an abstract memory for less structured problems where the general memory model was weak. So far, EDAs have only been used to strengthen memories learned over periods when the environment was not changing, but similar techniques used to learn solution spaces across changes could create more effective memories.

Memory-enhanced R-Wasps 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, we cannot make memories infinitely large because of the increase in overhead, particularly the computation required to choose a memory entry to retrieve. We propose 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.

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. Machines have no knowledge of the true job type distribution, so this 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 memory system (though not for the others). 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 [12]. 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.

Density-estimate memories 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, Model-C, has a finite number number of memory entries β for each machine's memory. Each memory entry is a cluster of stored points—machine states at a particular time. These points are equivalent to the stored points for 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 two centers: a distribution center c_d and a threshold center c_θ. These values are used to interact with the memory entry. The memory is updated with the same frequency. 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, Model-G, improves the retrieval of memory entries after a distribution change over Model-C. 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 Model-C.

The third density-estimate memory, Model-GW, uses a Gaussian model of the job type distribution throughout the memory system. In addition to using this model to compute probability that a point is part of the Gaussian for 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 Model-G, the Gaussian model 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.

4.2.4 Experiments

We compared standard R-Wasps to the memory-enhanced versions of R-Wasps described in Section 4.2.3: Memory, Memory-∞, Model-C, Model-G, and Model-GW. 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.

Table 3: Average results for scenarios with $\ell = 1.00$

Statistic	R-Wasps	Memory	Memory-∞	Model-C	Model-G	Model-GW
throughput	99.05	98.64	99.27	99.43	99.65	99.36
setups	2029.25	2478.20	1913.45	1371.45	1213.10	1550.30
cycle time	20.02	22.96	16.20	11.31	10.10	13.93
queue length	59.04	68.94	48.03	33.76	29.80	41.71

Table 4: 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 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
Model-C	R-Wasps	0.38	32.42 (+)	43.53 (+)	42.83 (+)
Model-C	Memory	0.80	44.66 (+)	50.75 (+)	51.03 (+)
Model-C	Memory-∞	0.16	28.33 (+)	30.19	29.71
Model-G	R-Wasps	0.60 (+)	40.22 (+)	49.58 (+)	49.52 (+)
Model-G	Memory	1.02 (+)	51.05 (+)	56.02 (+)	56.77 (+)
Model-G	Memory-∞	0.38	36.60 (+)	37.67 (+)	37.95 (+)
Model-G	Model-C	0.22	11.55	10.71	11.72
Model-GW	R-Wasps	0.32	23.60 (+)	30.45	29.36
Model-GW	Memory	0.74	37.44 (+)	39.34 (+)	39.50 (+)
Model-GW	Memory-∞	0.09	18.98 (+)	14.03	13.16
Model-GW	Model-C	-0.07	-13.04	-23.16	-23.55
Model-GW	Model-G	-0.28	-27.80	-37.93	-39.95

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 4.2.2. We ran scenarios with three values of $\ell \in \{1.00, 1.25, 1.50\}$ to test performance over a variety of loads.

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.

Tables 3, 5, and 7 show average results from 20 scenarios with $\ell = \{1.00, 1.25, 1.50\}$. Tables 4, 6, and 8 compare the approaches, showing the percent improvement of one approach over another. If these results are statistically significant with a confidence of 95%, the result is marked accordingly.

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

Table 5: Average results for scenarios with $\ell = 1.25$

Statistic	R-Wasps	Memory	Memory-∞	Model-C	Model-G	Model-GW
throughput	89.84	93.16	93.10	95.29	94.86	96.50
setups	2514.05	1821.00	1841.45	1208.10	1299.05	820.95
cycle time	137.70	106.79	110.37	85.84	90.79	61.17
queue length	538.06	403.87	419.28	323.91	344.41	234.51

Table 6: 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 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
Model-C	R-Wasps	6.07 (+)	51.95 (+)	37.66 (+)	38.63 (+)
Model-C	Memory	2.29	33.66 (+)	19.62	18.89 (+)
Model-C	Memory-∞	2.36 (+)	34.39 (+)	22.22	20.32 (+)
Model-G	R-Wasps	5.59 (+)	48.33 (+)	34.06 (+)	35.60 (+)
Model-G	Memory	1.82	28.66 (+)	14.98	14.88
Model-G	Memory-∞	1.89	29.46 (+)	17.74	16.38
Model-G	Model-C	-0.45	-7.53	-5.77	-4.94
Model-GW	R-Wasps	7.41 (+)	67.35 (+)	55.58 (+)	49.64 (+)
Model-GW	Memory	3.58 (+)	54.92 (+)	42.72 (+)	33.45 (+)
Model-GW	Memory-∞	3.65 (+)	55.42 (+)	44.58 (+)	34.62 (+)
Model-GW	Model-C	1.26	32.05 (+)	28.75 (+)	17.95 (+)
Model-GW	Model-G	1.73	36.80 (+)	32.63 (+)	21.81 (+)

Table 7: Average results for scenarios with $\ell = 1.50$

Statistic	R-Wasps	Memory	Memory-∞	Model-C	Model-G	Model-GW
throughput	79.33	82.71	81.10	81.71	83.08	83.06
setups	1935.15	1169.85	1489.70	1374.00	1057.15	1046.00
cycle time	262.83	233.78	244.26	241.27	225.57	226.87
queue length	1229.65	1073.85	1148.97	1114.86	1051.82	1041.66

Table 8: 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 -)

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
Model-C	R-Wasps	3.00	29.00	8.20	9.34
Model-C	Memory	-1.21	-17.45	-3.20	-3.82
Model-C	Memory-∞	0.75	7.77	1.23	2.97
Model-G	R-Wasps	4.73 (+)	45.37 (+)	14.18 (+)	14.46 (+)
Model-G	Memory	0.44	9.63	3.51	2.05
Model-G	Memory-∞	2.44	29.04	7.65	8.45
Model-G	Model-C	1.67	23.06	6.51	5.65
Model-GW	R-Wasps	4.71 (+)	45.95 (+)	13.68 (+)	15.29 (+)
Model-GW	Memory	0.42	10.59	2.95	3.00
Model-GW	Memory-∞	2.42	29.78	7.12	9.34
Model-GW	Model-C	1.66	23.87	5.97	6.57
Model-GW	Model-G	-0.02	1.05	-0.58	0.97

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, Model-G and Model-GW. Model-G had the best throughput and cycle time, while Model-GW 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.

4.2.5 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.

Model-C 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, Model-G 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, Model-G was the best memory for $\ell = 1.00$. Though it did not always outperform Model-C, the addition of the Gaussian model used to choose which memory entry to retrieve seems to have made this approach more consistent.

Model-GW, 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 Model-GW tries to exploit more information from the points in memory than Model-C or Model-G, it is more susceptible to this noise. As estimates of the distribution get better, performance improves.

4.2.6 Conclusions

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. Once 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 section, 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.

5 Proposed research

Using memory to help improve performance on dynamic problems has been successful, but previously implemented memory systems do have limitations. Sections 4.1 and 4.2 address some of these limitations, and approaches to other weaknesses in standard memory systems have been investigated by others. The goal of the proposed research detailed here is the development of a density-estimate memory system for dynamic problems. This system should be able to build a memory that provides a rich model of complex, realistic dynamic fitness landscapes, maintain the memory as the problem changes, use information from the memory to quickly locate promising areas of search, and be readily applicable to a wide variety of dynamic problems without having to extensively alter the parameters of the memory system. This section will identify several limitations of current memory systems and propose areas of research to address them. These areas of research are roughly divided into three areas: building rich models in memory, using density-estimate memory effectively, and generalizing memory to new problems and approaches.

5.1 Building rich models in memory

As discussed in Sections 3 and 4.2, the standard memory model stores a finite number of solutions, providing a simple model of the promising areas of the search space over time. This type of system performs well on simple problems, but the difficulty of the dynamic problem may increase in many ways. The number of distinct states of the dynamic problem may increase beyond the number of solutions a standard memory system can store. The fitness landscape at each environmental state and the overall landscape of good solutions over time may be multi-modal and too complex to accurately model using a small number of points.

Section 4.2 shows the first completed work on building models in memory over time. Instead of storing and interacting with only a few solutions, many more solutions are combined into probabilistic models. These models allow the memory to better represent promising areas of the search space. This first generation of density-estimate memories does have some potential shortcomings. Though much better suited than the standard memory model to dynamic problems where the number of promising areas exceeds the number of memory entries, the systems described in Section 4.2 are only first attempts at building models in memory, and still have many limitations. This first generation of density-estimate memory relies heavily on the clusters that are created incrementally over time, but no reclustering every occurs, which may cause problems. These systems also use simple models, typically a single Gaussian for each cluster. A cluster may end up including solutions from multiple peaks, which may be poorly modeled by a single Gaussian.

I plan to develop more sophisticated density-estimate memories for dynamic problems. First, I propose to use more powerful models to provide better representations of more complex dynamic problems. Many

potential multivariate models exist in the estimation of distribution algorithm and machine learning literature. In addition to better modeling multi-modal landscapes, these more sophisticated models may also be capable of learning linkages between variables in solutions.

Along with considering more powerful models, the structure of density-estimate memory systems will also be investigated. Though the work described in Section 4.2 has made progress in answering some questions, many remain. Should the memory be reclustered occasionally? Though this would add overhead, it might create more efficient clusters. Should clusters even be used? One might create a mixture model instead that could be used to model all solutions in memory, but this doesn't guarantee better performance or necessarily give the same low overhead. In the reinforcement learning domain explored in Section 4.2 fitness values for individual solutions were not available, but that information will often be available, though sometimes at some computational cost. How can this fitness information be useful in creating more useful models? These are just some of the questions that will be investigated as new density-estimate memory systems are developed and tested.

5.2 Using density-estimate memory effectively

Effectively using information stored in a memory may be as important a research area as how to store information in the memory. With the increased amount of information available from a memory that stores probabilistic models, one must consider how best to use that information. I will investigate two questions: how to retrieve solutions from the memory and how to use information from the memory to help maintain diversity in the search.

Retrieving solutions from memory In Section 4.2, it was only possible to retrieve a single solution from the memory on each machine, so the mean of the cluster that best matched the current environment was retrieved (Model-GW used a weighted mean). However, instead of using the mean, it would have been possible to randomly sample a single point from the model created from that cluster. For population-based search, we can use some combination of retrieving specific solutions from memory or generating new solutions based on the models stored in the memory. Though typically only one of these approaches is used, Yang [79] has shown that a hybrid approach works well for associative memory. Since the models in density-estimate memory may potentially provide much more information than other memory variants capable of generating new solutions, I plan to investigate the effects different retrieval strategies have on the effectiveness of density-estimate memories.

Maintaining diversity in search When used in population-based search, memory has often been combined with diversity techniques, since memory does not always provide sufficient population diversity alone. Often, these diversity techniques are completely uninformed, and may generate new solutions in areas already well modeled by the memory. Using the information from the models in memory to decide where to create new solutions may help diversity techniques to be more efficient and more effective. This idea is similar in some ways to the forking genetic algorithm (fGA) [66] and self-organizing scouts (SOS) [12], which divide a population into subpopulations which restricted search areas. However, since the information about where to search is stored not in the population, as in fGA and SOS, but in the memory, much more of the population is available for search, rather than being needed to maintain the subpopulations.

5.3 Generalizing density-estimate memory

Standard memory has been applied to a wide variety of dynamic problems with many different representations. Memory has also been applied to multiple approaches to solving dynamic problems, though evolutionary algorithms have been by far the most common. While density-estimate memory may go a long way toward making memory a more powerful tool by building probabilistic models in the memory, it is important to create a framework so that density-estimate memory can be generalized to many different types of

dynamic problems and can provide helpful information to many different types of approaches to solving dynamic problems. The requirements for a generalized memory system may be divided into three parts. First, memory should be extended to problems where solutions may change in length or the feasible region of the search space shifts over time. Second, the density estimation used in the memory should be able to accomodate models for many solution representations. Third, the memory should be general enough to be useful not only for population-based search algorithms like evolutionary algorithms but for a variety of approaches to solving dynamic problems.

Indirect memory layer At present, memory cannot be applied to many types of dynamic problems. In general, these problems use a variable length solution representation and the feasible region of the search space shifts over time. In Section 4.1, the first approach to using memory for dynamic problems with shifting feasible regions—classifier-based memory—was investigated, and proved successful on a dynamic scheduling problem. However, this approach does have some weaknesses. Classifier-based memory may not always provide much detail about the differences between two similar jobs, since the number of possible classifications is limited. Also, the current implementation does not always help with very specific job ordering, which might make it unsuitable to other similar problems like dynamic vehicle routing. I propose to create an improved indirect memory layer to address some of the shortcomings in classifier-based memory. I also propose to show how this indirect memory layer could be used for a problem other than dynamic scheduling.

Representations Different representations—bitstrings, permutations, etc.—may require different types of probabilistic models. This has been a subject of much research in the estimation of distribution algorithms literature. So far, work on density-estimate memory has focused on real numbers. To effectively apply memories that build probabilistic models to a wide range of problems with different solution representations, I plan to test density-estimate memory on a scheduling problem that uses a permutation representation and at least one benchmark problem that uses a bitstring representation.

Approaches There are many metaheuristics that could benefit from memory when solving dynamic problems. In Section 4.2, density-estimate memory has already been applied to a reinforcement learning algorithm. Most of the research on memory for dynamic optimization has focused on EAs, so I will begin by creating a density-estimate memory for an EA that can be compared to other types of memory. I also plan to create a framework so density-estimate memories can be used with other metaheuristics.

6 Research plan

Summer 2009

- Finish testing first generation density-estimate memories for evolutionary algorithms on the moving peaks benchmark problem
- Develop second generation density-estimate memories
- Test second generation density-estimate memory for evolutionary algorithms on benchmark problems

Fall 2009

- Finish tests of second generation density-estimate memory for evolutionary algorithms on benchmark problems
- Develop integrated diversity measure that uses information from density-estimate memory
- Test integrated diversity measure on benchmark problems

- Develop third generation multivariate density-estimate memory

Spring 2010

- Test third generation multivariate density-estimate memory for evolutionary algorithms on benchmark problems
- Develop experiments to compare memory retrieval strategies when using density-estimate memory
- Test multivariate density-estimate memory on other types of metaheuristics
- Complete experiments comparing memory retrieval strategies
- Develop indirect memory layer for a generalized memory system using classifier-based memory as a basis

Summer and Fall 2010

- Test combination of indirect memory layer and multivariate density-estimate memory on scheduling problems
- Complete any remaining experiments
- Write thesis and defend

References

[1] Gregory J. Barlow and Choong K. Oh. Robustness analysis of genetic programming controllers for unmanned aerial vehicles. In *Proceedings of the 2006 Genetic and Evolutionary Computation Conference*, Seattle, WA, July 2006.

[2] 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.

[3] C. N. Bendtsen and T. Krink. Dynamic memory model for non-stationary optimization. In *Congress on Evolutionary Computation*, pages 145–150. IEEE, 2002.

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

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

[6] T. Blackwell. Swarms in dynamic environments. In E. Cantu-Paz, editor, *Genetic and Evolutionary Computation Conference*, volume 2723 of *LNCS*, pages 1–12. Springer, 2003.

[7] T. M. Blackwell and P. J. Bentley. Dynamic search with charged swarms. In W. B. Langdon et al., editor, *Genetic and Evolutionary Computation Conference*, pages 19–26. Morgan Kaufmann, 2002.

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

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

[10] J. Branke, T. Kaußler, C. Schmidt, and H. Schmeck. A multi-population approach to dynamic optimization problems. In *Adaptive Computing in Design and Manufacturing 2000*. Springer, 2000.

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] A. Carlisle and G. Dozier. Adapting particle swarm optimisation to dynamic environments. In *Proc of int conference on artificial intelligence*, pages 429–434, 2000.

[18] W. Cedeno and V. R. Vemuri. On the use of niching for dynamic landscapes. In *International Conference on Evolutionary Computation*. IEEE, 1997.

[19] Pei-Chann Chang, Jih-Chang Hsieh, and Yen-Wen Wang. Genetic algorithm and case-based reasoning applied to production scheduling. In Yaochu Jin, editor, *Knowledge Incorporation in Evolutionary Computation*. Springer, 2005.

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

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

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

[23] 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.

[24] P. Collard, C. Escazut, and A. Gaspar. An evolutionary approach for time dependent optimization. *International Journal on Artificial Intelligence Tools*, 6(4):665–695, 1997.

[25] D. Dasgupta and D. R. McGregor. Nonstationary function optimization using the structured genetic algorithm. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature*, pages 145–154. Elsevier Science Publisher, 1992.

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

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

[28] 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, October 2004.

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

[30] A. Gaspar and P. Collard. Time dependent optimization with a folding genetic algorithm. In *International Conference on Tools for Artificial Intelligence*, pages 207–214. IEEE Computer Society Press, 1997.

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

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

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

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

[35] Michael Guntsch. *Ant Algorithms in Stochastic and Multi-Criteria Environments*. PhD thesis, Universität Karlsruhe (TH), Universität Karlsruhe (TH), Institut AIFB, D-76128 Karlsruhe, 2004. Prof. Dr. H. Schmeck.

[36] Michael Guntsch and Martin Middendorf. Applying population based aco to dynamic optimization problems. In *Ant Algorithms, Proceedings of Third International Workshop ANTS 2002*, volume 2463 of *LNCS*, pages 111–122, 2002.

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

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

[39] S. Janson and M. Middendorf. A hierachical particle swarm optimizer for dynamic optimization problems. In G. R. Raidl, editor, *Applications of evolutionary computing*, volume 3005 of *LNCS*, pages 513–524. Springer, 2004.

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

[41] 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.

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

[43] 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.

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

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

[46] J. Lewis, E. Hart, and G. Ritchie. A comparison of dominance mechanisms and simple mutation on non-stationary problems. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 139–148. Springer, 1998.

[47] 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.

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

[49] 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.

[50] N. Mori, H. Kita, and Y. Nishikawa. Adaptation to a changing environment by means of the thermodynamical genetic algorithm. In H.-M. Voigt, editor, *Parallel Problem Solving from Nature*, number 1141 in LNCS, pages 513–522. Springer Verlag Berlin, 1996.

[51] 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.

[52] 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.

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

[54] K. P. Ng and K. C. Wong. A new diploid scheme and dominance change mechanism for non-stationary function optimization. In *Sixth International Conference on Genetic Algorithms*, pages 159–166. Morgan Kaufmann, 1995.

[55] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics*. MIT Press, Cambridge, MA, 2000.

[56] 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.

[57] Djamila Ouelhadj and Sanja Petrovic. Survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*, 2008.

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

[59] Bosman Peter A.N. Learning and anticipation in online dynamic optimization. In Shenxiang Yang, Yew-Soon Ong, and Yaochu Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, pages 129–152. Springer, 2007.

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

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

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

[63] C. Ryan. The degree of oneness. In *First Online Workshop on Soft Computing*, pages 43–49, 1996.

[64] C. Ryan. Diploidy without dominance. In J. T. Alander, editor, *Third Nordic Workshop on Genetic Algorithms*, pages 45–52, 1997.

[65] Stephen A. Stanhope and Jason M. Daida. (1+1) genetic algorithm fitness dynamics in a changing environment.

[66] S. Tsutsui, Y. Fujimoto, and A. Ghosh. Forking genetic algorithms: GAs with search space division schemes. *Evolutionary Computation*, 5(1):61–80, 1997.

[67] Rasmus K. Ursem. Multinational evolutionary algorithms. In *Proceedings of the Congress on Evolutionary Computation*, 1999.

[68] Rasmus K. Ursem. Multinational gas: Multimodal optimization techniques in dynamic environments. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2000.

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

[70] F. Vavak, T. C. Fogarty, and K. Jukes. A genetic algorithm with variable range of local search for tracking changing environments. In H.-M. Voigt, editor, *Parallel Problem Solving from Nature*, number 1141 in LNCS. Springer Verlag Berlin, 1996.

[71] F. Vavak, K. Jukes, and T. C. Fogarty. Adaptive combustion balancing in multiple burner boiler using a genetic algorithm with variable range of local search. In T. Bäck, editor, *International Conference on Genetic Algorithms*, pages 719–726. Morgan Kaufmann, 1997.

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

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

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

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

[76] Shengxiang Yang and Xin Yao. Population-based incremental learning with associative memory for dynamic environments. *IEEE Transactions on Evolutionary Computation*, 2008.

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

[78] Shenxiang Yang. Constructing dynamic test environments for genetic algorithms based on problem difficulty. In *Proceedings of the Congress on Evolutionary Computation*, pages 1262–1269, 2004.

[79] Shenxiang Yang. Explicit memory schemes for evolutionary algorithms in dynamic environments. In Shenxiang Yang, Yew-Soon Ong, and Yaochu Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*. Springer, 2007.

[80] Bo Yuan, Maria Orlowska, and Shazia Sadiq. Extending a class of continuous estimation of distribution algorithms to dynamic problems. *Optimization Letters*, 2007.