Using Memory Models to Improve Adaptive Efficiency in Dynamic Problems

Gregory J. Barlow and Stephen F. Smith

***Abstract*—Many real-world problems involve the coordination of multiple agents in dynamic environments, where characteristics of the problem being solved change over time. In such problems, adaptive, self-organizing agent approaches have been shown to provide very robust solutions. However, these approaches often require non-trivial amounts of time to respond to large environmental shifts. Considering this limitation, we observe that environmental changes in a given dynamic problem are generally not completely random; similar states in the environment tend to reappear over time. Memory is one way to leverage this past information and improve the adaptive efficiency of the agent system. In this paper, we explore the use of memory as a means of boosting the performance of self-organizing agents in solving dynamic coordination problems. We consider the specific problem of coordinating product flows in a factory that is subject to changing job mixes over time, which has been previously solved using a computational model of the task allocation behavior of wasps. We augment this base procedure with a number of memory systems, the most sophisticated of which exploit memory models inspired by estimation of distribution algorithms (EDAs) to manage computational cost. An experimental analysis is presented which demonstrates the advantage of using memory. Configurations using the EDA-inspired memory models are shown to substantially outperform configurations with more standard and infinite-sized memory models, and all are shown to improve the performance of the baseline task allocation procedure.**

I. Introduction

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: by completely resolving the problem from scratch each time a change occurs, by creating a model that can learn and adapt as changes occur [1], by using optimization to maintain a good solution over time [2], or by many other 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.

Using information from the past to improve current performance is a common technique for dynamic problems. The current state of the environment in a dynamic problem 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 well studied approach is explicit memory—directly storing a finite number of previous solutions that can be retrieved later when the environment changes [3].

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 paper, we introduce several model-building memory systems inspired by estimation of distribution algorithms that improve upon standard memory systems while avoiding large increases in overhead.

The remainder of this paper is organized as follows. 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 the baseline approach on the new dynamic distributed factory coordination problem and present several novel model-building memory systems. We will then compare the performance of the baseline system with the memory-augmented systems, showing that the model-building memories outperform both the baseline system and standard memory.

II. 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. 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 [4], [5]. Trucks arrive off the assembly line to

Gregory J. Barlow and Stephen F. Smith are with the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA (email: {gjb@cmu.edu, sfs@cs.cmu.edu}

be painted, and then wait to be assigned to a paint booth's queue. The color of each truck is determined probabilistically based on a distribution of colors. Booths each have a queue of trucks waiting to be painted. All booths can paint a truck any of the available colors, but when a booth switches between colors it must flush out the previous paint color, causing a setup time delay as well as incurring the cost of the wasted paint. Booths that specialize in painting a single color, at least for a few trucks in a row, incur fewer setups. Morley demonstrated that a market-based approach, where the paint booths bid against each other for trucks coming off the assembly line, could outperform the centralized scheduler previously used by the real paint shop [4]. This system saved almost a million dollars in the first nine months of use [5]. In this approach, a paint booth bids on a truck based on the current length of the booth's queue and whether a setup delay would be required to process this truck.

Campos et al. [6] and Cicirello and Smith [1] 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. [7] and others examined similar approaches.

Cicirello and Smith [1] 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 [1] 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. Recurring changes in the job type distribution highlight the need for adaptability to large environmental changes, which R-Wasps may sometimes lack.

III. Problem

In this paper, we examine a dynamic extension of the distributed factory coordination problem similar to the variant from Cicirello and Smith [1] described above. Performance is evaluated over a much longer time horizon, and the underlying distribution of the job types changes many times.

A. 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 $[10, 20]$. The objectives are to maximize the number of jobs processed by the factory and to minimize the number of setups.

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 [1], we chose a mean arrival rate per machine of 0.05 to represent a medium to heavily loaded factory. 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 $\lambda = 0.05N\ell$. 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.

B. R-Wasps

We use R-Wasps as described in [1] 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, w, 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 in this paper are the same as those from [1]: $\theta_{min} = 1$, $\theta_{max} = 1000$, $\delta_1 = 2$, $\delta_2 = 1$, 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. Thresholds are initialized uniformly at random in the interval $\left[\frac{\theta_{max}}{2}, \theta_{max}\right]$ unless $w = j$, in which case $\theta_{w,j} = \theta_{min}$. Further explanation of the R-Wasps algorithm may be found in [1].

C. R-Wasp Weaknesses

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

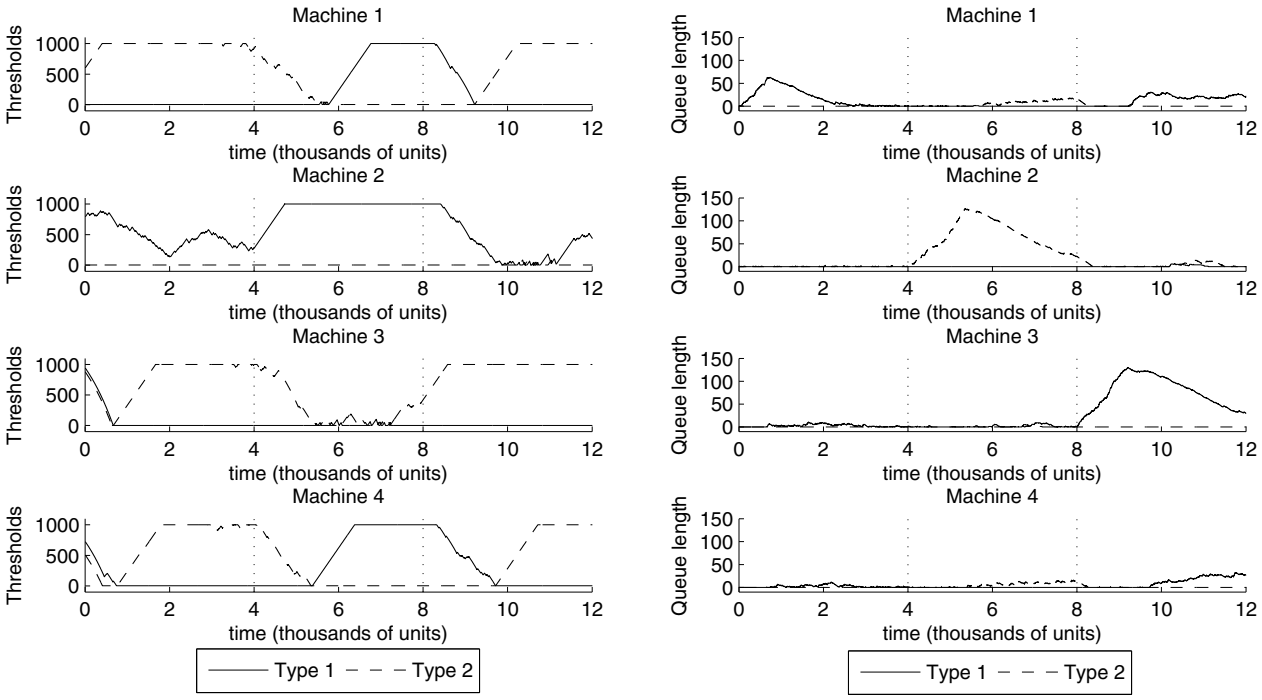
As an example, take a problem with four machines, two job types, and three time periods, each 4000 time units long. In the first time period, 85% of jobs arriving are of type 1 and 15% are of type 2. At the beginning, each machine adapts its thresholds to accept jobs efficiently. In the second period, 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 1(a) shows, this may take longer from the change in distributions than the initial adaptation took from the beginning of the scenario. For example, by $t = 650$, three of the four machine have the lowest possible threshold for accepting the more common job type 1. After the first change, it takes until $t = 5350$, 1350 time units after the change, for a second machine to begin accepting the now more common job type 2. 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 1(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 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.

IV. 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 III-A.

A. Previous uses of memory

Memory has been used extensively in dynamic problems, primarily for dynamic optimization with evolutionary algo-



(a) Thresholds

(b) Queue lengths

Fig. 1. Thresholds and queue lengths for an example run of R-Wasps with four machines and two job types

rithms (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 used a case-based memory, where memory was used to initialize the population of an EA after a change [8]. Branke introduced a standard memory model [3] for dynamic optimization with EAs; many other uses of memory from the literature are based on this model. Eggermont et al. introduced a case-based memory similar to Branke's, but with several options for retrieving entries from the memory [9]. Branke refined his standard memory in several ways, including a multi-population memory/search model [2]. Chang et al. used case-based reasoning for period-based scheduling with an EA [10]. Barlow and Smith [11] investigated an indirect 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 [12]. Karaman et al. [13] introduced a memory indexing EA and Yang [14], [15] introduced an associative memory, both of which use estimations of a population's distribution to reinitialize the population after a change. Richter and Yang [16] investigated an abstract memory for less structured problems where the standard 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.

B. Memory-enhanced R-Wasps

One of the weaknesses of the standard memory systems from the literature is the typically 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 model-building 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 [2], 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 probability of job type *j* in D_t is the number of jobs of type *j* that arrived during this time window divided by the total number of jobs that arrived during the time window. 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 S 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 [2]. The new point is added to the memory, and the two entries which have the most similar job type distributions are found using Euclidean distance between distribution vectors to measure similarity. 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. If no memory entry is within ε of the current distribution, then retrieving from the memory might actually hinder R-Wasps.

C. Model-building 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 model-building memories use the same mechanisms as the standard memory model described above.

The first model-building memory system, Model-C, has a finite number of memory entries β for each machine's memory. Each memory entry is a cluster of stored points. 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 (mean vectors): a distribution center c_d and a threshold center c_θ. These values are used to interact with the memory entry. 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. Instead of replacing points as in the standard memory, we cluster similar points together and aggregate information from them. 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 model-building memory, Model-G, attempts to improve the process of retrieving entries from the memory using a Gaussian model of each cluster. Clusters may have very different densities, and the richer model that the Gaussians provide may help improve performance when more than one entry is a good match for the current environment. In addition to c_d and c_θ, a Gaussian model of the job type distributions in the cluster, m_θ, is created. For clusters with fewer than 10 points, the model is padded by adding random points around c_θ (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, for each memory entry we calculate the probability that the current distribution belongs to the entry's Gaussian model using the probability density function of the model and choose the entry with the highest probability. All the other parts of the memory system are the same as in Model-C.

The third model-building memory, Model-GW, uses Gaussian models of the job type distribution throughout the memory system. In addition to using the models for retrieving entries from the memory, the models are used when adding new points to the memory with the goal of using cluster density information to improve the quality of entries created by the memory. Instead of measuring distance between entries when deciding which to merge, a match 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 pair of entries with the highest match 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 to attempt to reduce the influence of outliers in the cluster when an entry is retrieved from memory. 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_θ. 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.

V. Experiments

We compared standard R-Wasps to the memory-enhanced versions of R-Wasps described in Section IV: 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 model-building memories).

We examined problems with four machines and four job types. Each scenario lasted 150000 time units split into 50 periods of 3000 time units. At the beginning of every period, the job type distribution used to generate new job arrivals changes to a new distribution. This distribution is chosen at random from ten distributions generated at the beginning of the scenario. The distribution for this period is then randomly perturbed, so distributions are not repeated exactly. For scenarios where memory was used, each machine updated its memory every $S = 1000 + N(0, 250)$ time units.

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 III-A. 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 I, III, and V show average results from 20 scenarios with $\ell = \{1.00, 1.25, 1.50\}$. Tables II, IV, and VI compare the approaches, showing the percent improvement of one approach over another. If 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 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 model-building 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. Throughputs were lower, with larger differences between methods. 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 model-building 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 model-building 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.

VI. 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 model-building 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, Memory-∞ did not outperform the model-building memories even though the infinite-sized memory required much more computational overhead.

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 memory approach 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 the performance of Model-G 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 model-building memories when $\ell = 1.00$. Under lighter loads,

TABLE I
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 II
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	

TABLE III
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 IV
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 V
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 VI
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

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.

VII. 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. One way to exploit past information is through the use of memory. Standard memory models exist, but have a limited ability to model dynamic solution landscapes. In this paper, we have introduced three model-building memory systems. By aggregating information from many points in time, model-building memories improve performance over 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, while the memory improves adaptability when changes in the underlying job type distribution occur. R-Wasps maintains control of the system except immediately after changes in the distribution, so the system remains flexible.

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

While thus far only tested on this problem, these types of model-building memories could be applied to many different approaches to dynamic problems. In future work, we plan to develop richer and more complex memory models. We also plan to extend these model-building memories to other problems and approaches.

REFERENCES

[1] V. A. Cicirello and S. F. Smith, "Wasp-like agents for distributed factory coordination," *Autonomous Agents and Multi-Agent Systems*, vol. 8, pp. 237–266, 2004.

[2] J. Branke, *Evolutionary Optimization in Dynamic Environments*. Kluwer, 2002.

[3] ——, "Memory enhanced evolutionary algorithms for changing optimization problems," in *Congress on Evolutionary Computation*, 1999, pp. 1875–1882.

[4] R. Morley and C. Schelberg, "An analysis of a plant-specific dynamic scheduler," in *Proceedings of the NSF Workshop on Dynamic Scheduling*, 1993, pp. 115 – 122.

[5] R. 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*. Ernst and Young Center for Business Innovation, 1996, pp. 53 – 58.

[6] M. Campos, E. Bonabeau, G. Theraulaz, and J.-L. Deneubourg, "Dynamic scheduling and division of labor in social insects," *Adaptive Behavior*, vol. 8, no. 2, pp. 83 – 96, 2000.

[7] S. Nouyan, R. Ghizzioli, M. Birattari, and M. Dorigo, "An insect-based algorithm for the dynamic task allocation problem," *Künstliche Intelligenz*, vol. 4, pp. 25 – 31, 2005.

[8] C. L. Ramsey and J. J. Grefenstette, "Case-based initialization of genetic algorithms," in *Proceedings of the International Conference on Genetic Algorithms*, 1993, pp. 84–91.

[9] J. Eggermont, T. Lenaerts, S. Poyhonen, and A. Termier, "Raising the dead: Extending evolutionary algorithms with a case-based memory," in *European Conference on Genetic Programming*, 2001, pp. 280–290.

[10] P.-C. Chang, J.-C. Hsieh, and Y.-W. Wang, "Genetic algorithm and case-based reasoning applied to production scheduling," in *Knowledge Incorporation in Evolutionary Computation*. Springer, 2005.

[11] G. J. Barlow and S. F. Smith, "A memory enhanced evolutionary algorithm for dynamic scheduling problems," in *Applications of Evolutionary Computation: EvoWorkshops 2008*, 2008, pp. 606–615.

[12] P. Larranaga and J. A. Lozano, *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Springer, 2002.

[13] A. Karaman, S. Uyar, and G. Eryigit:, "The memory indexing evolutionary algorithm for dynamic environments," in *Applications of Evolutionary Computation: EvoWorkshops 2005*, 2005, pp. 563–573.

[14] S. Yang, "Associative memory scheme for genetic algorithms in dynamic environments," in *Applications of Evolutionary Computing: EvoWorkshops 2006*, 2006, pp. 788–799.

[15] ——, "Explicit memory schemes for evolutionary algorithms in dynamic environments," in *Evolutionary Computation in Dynamic and Uncertain Environments*, S. Yang, Y.-S. Ong, and Y. Jin, Eds. Springer, 2007.

[16] H. Richter and S. Yang, "Memory based on abstraction for dynamic fitness functions," in *Applications of Evolutionary Computing: EvoWorkshops 2008*, 2008, pp. 596–605.