A Memory Enhanced Evolutionary Algorithm for Dynamic Scheduling Problems

Gregory J. Barlow and Stephen F. Smith

Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA
gjb@cmu.edu, sfs@cs.cmu.edu

Abstract. This paper describes a memory enhanced evolutionary algorithm (EA) approach to the dynamic job shop scheduling problem. Memory enhanced EAs have been widely investigated for other dynamic optimization problems with changing fitness landscapes, but only when associated with a fixed search space. In dynamic scheduling, the search space shifts as jobs are completed and new jobs arrive, so memory entries that describe specific points in the search space will become infeasible over time. The relative importances of jobs in the schedule also change over time, so previously good points become increasingly irrelevant. We describe a classifier-based memory for abstracting and storing information about schedules that can be used to build similar schedules at future times. We compared the memory enhanced EA with a standard EA and several common EA diversity techniques both with and without memory. The memory enhanced EA improved performance over the standard EA, while diversity techniques decreased performance.

1 Introduction

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.

Prior work has shown many techniques for improving the performance of evolutionary algorithms on dynamic problems of this sort [1, 2]. Approaches to these types of problems must consider two competing objectives: improving the fitness of solutions, and decreasing the search time necessary to reach good solutions. There are three broad categories of approaches for improving evolutionary algorithms on dynamic problems: keeping the population diverse to avoid population convergence [3], using a memory or multiple populations to maintain good solutions for future use [1, 4], and anticipating changes to produce solutions that will be robust to change [5, 6]. The first two categories address the issue of population convergence, which can hinder the ability of the EA to find better solutions after a change. Approaches in the third category produce solutions that not only have high fitness in the current environment, but are robust to environmental changes.

A variety of dynamic benchmark problems have been considered, including the moving peaks problem [4, 7], the dynamic knapsack problem, dynamic bit-matching, dynamic scheduling, and others [1, 2]. 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 [8, 9], problems with machine breakdowns and redundant resources [10], improved genetic operators [11], heuristic reduction of the search space [12], and anticipation to create robust schedules [5, 6]. While Louis and McDonnell [13] 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 as the landscape is changing, so a memory may store individuals directly from the population [4]. 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 time-dependent constraints that shift the feasible region of the 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 paper, 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 paper, 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.

2 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.

3 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. In static problems, these are permutations of all jobs; 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 [14] 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 [15] 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 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 classifications, 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 [1]. 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

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

Fig. 1. Classifier-based memory example

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 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 paper, 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.

5 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 [1]. Hence, we also consider a standard EA with 25 random immigrants [3] per generation (RI) and the same approach with classifier-based memory (RIm). Finally, we consider the memory/search approach of [1], 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 paper, 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 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 [5, 6, 9], 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.

6 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 much 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 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%

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.

For both fitness and search, the addition of classifier-based memory improved performance over the standard EA. While large improvement in fitness was only

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%

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.

7 Conclusion

This paper 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 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 [1]. 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.

As this is a preliminary investigation of the use of memory for dynamic scheduling, there are many avenues for future work in dynamic scheduling with evolutionary algorithms. 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. Applying other approaches from the literature, like self-organizing scouts [1], to dynamic scheduling problems might also be a good area for future work.

References

1. Branke, J.: Evolutionary Optimization in Dynamic Environments. Kluwer (2002)
2. Jin, Y., Branke, J.: Evolutionary optimization in uncertain environments—a survey. IEEE Transactions on Evolutionary Computation **9**(3) (2005) 303–317
3. Grefenstette, J.J.: Genetic algorithms for changing environments. In: Parallel Problem Solving from Nature. (1992) 137–144
4. Branke, J.: Memory enhanced evolutionary algorithms for changing optimization problems. In: Congress on Evolutionary Computation. (1999) 1875–1882
5. Branke, J., Mattfeld, D.C.: Anticipatory scheduling for dynamic job shop problems. In: AIPS Workshop on On-line Planning and Scheduling. (2002) 3–10
6. Branke, J., Mattfeld, D.C.: Anticipation and flexibility in dynamic scheduling. International Journal of Production Research **43**(15) (2005) 3103–3129
7. Morrison, R., DeJong, K.: A test problem generator for non-stationary environments. In: Congress on Evolutionary Computation. (1999) 2047–2053
8. Lin, S.C., Goodman, E.D., William F. Punch, I.: A genetic algorithm approach to dynamic job shop scheduling problems. In: International Conference on Genetic Algorithms. (1997) 481–488
9. Bierwirth, C., Mattfeld, D.C.: Production scheduling and rescheduling with genetic algorithms. Evolutionary Computation **7**(1) (1999) 1–17
10. Chryssolouris, G., Subramaniam, V.: Dynamic scheduling of manufacturing job shops using genetic algorithms. Journal of Intelligent Manufacturing **12** (2001) 281–293
11. Vazquez, M., Whitley, L.D.: A comparison of genetic algorithms for the dynamic job shop scheduling problem. In: Genetic and Evolutionary Computation Conference. (2000) 1011–1018
12. Mattfeld, D.C., Bierwirth, C.: An efficient genetic algorithm for job shop scheduling with tardiness objectives. European Journal of Operations Research **155** (2004) 616–630
13. Louis, S.J., McDonnell, J.: Learning with case-injected genetic algorithms. IEEE Transactions on Evolutionary Computation **8**(4) (2004) 316–328
14. Giffler, B., Thompson, G.L.: Algorithms for solving production scheduling problems. Operations Research **8**(4) (1960) 487–503
15. Bierwirth, C., Mattfeld, D.C., Kopfer, H.: On permutation representations for scheduling problems. In: Parallel Problem Solving from Nature. (1996) 310–318