

Genetic Algorithm Parameter Control: Application to Scheduling with Sequence-Dependent Setups

Vincent A. Cicirello
Computer Science and Information Systems
Stockton University
101 Vera King Farris Drive
Galloway, NJ, 08205
cicirelv@stockton.edu

ABSTRACT

Genetic algorithms, and other forms of evolutionary computation, are controlled by numerous parameters, such as crossover and mutation rates, population size, among others depending upon the specific form of evolutionary computation as well as which operators are employed. Setting the values for these parameters is no simple task. In this paper, we develop a genetic algorithm with adaptive control parameters for an NP-Hard scheduling problem known as weighted tardiness scheduling with sequence-dependent setups. Our genetic algorithm uses the permutation representation along with the non-wrapping order crossover and insertion mutation operators. We encode the control parameters within the members of the population and evolve these during search using Gaussian mutation. We demonstrate this approach out-performs a manually tuned genetic algorithm for the problem, and that it converges upon effective parameter values very early in the run.

Categories and Subject Descriptors

I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search—*heuristic methods, scheduling*; I.2.6 [Artificial Intelligence]: Learning—*parameter learning*; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—*sequencing and scheduling, computations on discrete structures*; G.2.1 [Discrete Mathematics]: Combinatorics—*combinatorial algorithms, permutations and combinations*

General Terms

Algorithms, Experimentation, Performance

Keywords

Genetic algorithm, parameter control, parameter optimization, permutation operators, weighted tardiness scheduling, sequence-dependent setups

1. INTRODUCTION

The Genetic Algorithm (GA) and other forms of evolutionary computation are typically controlled by several parameters. For example, the simplest form of GA is controlled by a crossover rate, mutation rate, and population size; while more sophisticated forms have additional parameters such as elitism rate, scaling window, generation gap, or use parameterized operators such as uniform crossover or k -point crossover. Even operator choice can be viewed as a parameter (e.g., single-point vs two-point vs uniform crossover). The most common approach to parameter tuning is manual tuning—i.e., the GA implementer uses a tedious trial-and-error approach. Oftentimes, published results report the parameter values, but do not explain how they were derived.

Others take a more rigorous approach to control parameter tuning. De Jong offers the earliest example of formal analysis of GA control parameters, providing an empirically determined “optimal” set of parameter values for a specific class of function optimization problem [17]; while others look to automate the process, such as Grefenstette’s introduction of the idea that a GA can be used to optimize GA control parameters [21]. There have since been a variety of meta-optimization approaches for optimizing GA parameters (e.g., [25, 13, 4]), as well as the parameters of other metaheuristics (e.g., [25, 32]) and systems (e.g., [26]). Others argue that control parameters are not to be tuned a priori, but rather should adapt dynamically using feedback from search progress (e.g., [22, 18, 38, 1, 5, 15]).

In this paper, we explore an approach to dynamically adapting GA control parameters for an NP-Hard single-machine scheduling problem known as weighted tardiness scheduling with sequence-dependent setups. Our GA uses a permutation representation, rather than the classic bit-string. Many problems, such as the scheduling problem we explore here, are more naturally represented as permutations of a set—in this case a set of jobs. The permutation represents the order to process the jobs. To adapt the control parameters, we augment the representation to include the permutation as well as the crossover and mutation rates, allowing the GA to evolve not only the problem’s solution, but also its own parameters. In our experiments, we see potential to use the approach for parameter tuning, in addition to parameter control. That is, though we focus on dynamic parameter adaptation, the final evolved parameters can potentially be used to control future GA runs.

We begin by providing background and related work (Section 2), including on the GA permutation representation and

relevant operators (Section 2.1) as well as on the target problem, weighted tardiness scheduling with sequence-dependent setups (Section 2.2). We present the technical details of our adaptive GA in Section 3 and discuss our experimental results in Section 4. We offer concluding remarks in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 GA Permutation Operators

The permutation representation requires specialized GA operators, capable of producing valid permutations from population members. Crossover must be capable of recombining parts of two parent permutations to produce two valid child permutations; and mutation must be capable of producing a new permutation that is a variation of another.

There are many crossover operators available that attempt to recombine different permutation properties, and are thus relevant for different problem types. For some problems, the positions of the elements within the permutation are most important, such as assignment problems where an optimal one-to-one mapping from the elements of one set to the elements of another is sought (e.g., largest common subgraph and other isomorphism related problems [36, 13]). For such problems, crossover must focus on retaining absolute positions of elements in the parent permutations when forming children. Cycle Crossover (CX) is the best available example that does just that [29]. Other relevant, though more disruptive, operators for this problem class include Partially Matched Crossover (PMX) [20] and Uniform Partially Matched Crossover (UPMX) [13]. For other problems, crossover must attempt to retain relative positions of elements (i.e., which elements are adjacent), such as the traveling salesperson, and other routing and scheduling problems. Crossover operators for relative position problems include Order Crossover (OX) [16], Non-Wrapping Order Crossover (NWOX) [7], and Uniform Order Based Crossover (UOBX) [33]. Most relevant to this paper is NWOX, as it retains edges while minimizing positional deviation relative to the original parent permutations, unlike OX which keeps edges but tends to displace elements large distances from locations in parents. For scheduling with sequence-dependent setups, edges directly impact fitness; but for the weighted tardiness scheduling objective, the general position of the permutation elements also impacts due date achievement, and thus fitness. NWOX is therefore ideal for this problem. For yet other problems, crossover must retain general pairwise element precedences, and not simply edges, in order to produce children phenotypically similar to the parents. Precedence Preservative Crossover (PPX) [3] is designed to do this, while others that may work well for such problems include NWOX, UOBX, and CX. There are yet other crossover operators that introduce problem-dependent knowledge into crossover (e.g., [28, 37, 9]).

There are relatively few commonly used permutation mutation operators [19, 31, 35, 12]. Among these are swap, which exchanges a random pair of elements; insertion, which removes a random element and reinserts it at a random location; reversal, which reverses the order of a random sub-permutation; and scramble, which randomizes a random sub-permutation. There also exist window-limited variants of these, which constrain distance between random element selection [10]. See [11] for a comprehensive fitness landscape analysis of mutation operator behavior on permutation land-

scapes. One of the results of that study showed insertion to be the dominant choice when directed edges most directly impact fitness, and when small positional movement is important. Thus, for the same reasons that we have chosen NWOX, we use insertion mutation in our GA.

2.2 Sequence-Dependent Setup Scheduling

The weighted tardiness scheduling problem with sequence-dependent setups consists of N jobs $J = \{j_1, j_2, \dots, j_N\}$. Each job j_k has weight w_k , due date d_k , and process time p_k . For each pair of jobs, $s_{i,k}$ is the setup time required prior to processing job j_k if it immediately follows job j_i . Setup times are asymmetric—i.e., it is not necessarily the case that $s_{i,k} = s_{k,i}$. $s_{0,k}$ is the initial setup time required if job j_k is processed first. The weighted tardiness objective is to sequence the set of jobs J to minimize:

$$T = \sum_{k=1}^N w_k T_k = \sum_{k=1}^N w_k \max(c_k - d_k, 0), \quad (1)$$

where T_k and c_k are the tardiness and completion time of job j_k . The completion time c_k is the sum of the process times and setup times of all jobs that come before j_k plus the setup time and process time of j_k . Let $\pi(k)$ be the position in the sequence of job j_k , then c_k is defined as follows:

$$c_k = \sum_{\pi(x) \leq \pi(k), \pi(x) = \pi(y)+1} (p_x + s_{y,x}). \quad (2)$$

Single-machine scheduling to optimize weighted tardiness is NP-Hard even if setups are independent of job ordering [27]. Sen and Bagchi show that sequence-dependent setups induce a non-order-preserving property of the evaluation function that greatly magnifies problem difficulty [30]. The current best available exact solver for the problem, Tanaka and Araki's Successive Sublimation Dynamic Programming, is capable of solving all of the available benchmark instances, but requires over two weeks of memory-intensive CPU time to solve the harder instances [34]. Therefore, it is desirable to turn to alternative approaches, such as genetic algorithms as well as other metaheuristics, that are able to more efficiently find sufficiently-optimal solutions.

In our experiments, we use the standard benchmark set for the problem, which we previously introduced [5, 6].¹ This benchmark set has been used by many researchers for a variety of search algorithms, such as dynamic programming [34], neighborhood search [24], iterated local search [39], value-biased stochastic sampling [14], genetic algorithms [7], simulated annealing [8], ant colony optimization [23], etc.

3. ADAPTIVE GENETIC ALGORITHM

Instance Preprocessing. To minimize the impact of setup times on problem solving performance, we transform each job, j_k , increasing process time by the job's minimum setup time, and reducing all setup times accordingly:

$$s_k^{\min} = \min_{0 \leq i \leq N, i \neq k} s_{i,k}, \quad (3)$$

$$p_k = p_k + s_k^{\min}, \quad (4)$$

$$s_{i,k} = s_{i,k} - s_k^{\min}, \forall i, i \neq k, 0 \leq i \leq N. \quad (5)$$

¹Currently maintained at <http://loki.stockton.edu/~cicirelv/benchmarks/>.

Additionally, following Tanaka and Araki [34] as well as others, we eliminate any job j_k , with weight $w_k = 0$, provided $\forall x \forall y, x \neq y, s_{x,k} + p_k + s_{k,y} \geq s_{x,y}$.

Representation. Let Pop refer to the GA population, and let $PopSize$ be the size of the population. Each individual member i of the population is defined as a 4-tuple as follows: $Pop_i = \langle P_i, C_i, M_i, \sigma_i \rangle$. P_i is a permutation—in the case of the scheduling problem we consider here, it is a permutation of the set of jobs. C_i and M_i are the crossover and mutation rates, respectively, for member i of the population; and σ_i is a parameter related to mutating the crossover and mutation rates, which we will discuss in detail below.

Fitness Calculation. The fitness of population member Pop_i depends explicitly only on permutation P_i , and not on any of the GA control parameters embedded in Pop_i . Let $T(P_i)$ be the weighted tardiness (Equation 1) for permutation P_i ; and define fitness as follows:

$$fitness(Pop_i) = 1 - T(P_i) + \max_{1 \leq k \leq PopSize} T(P_k). \quad (6)$$

Our objective is to minimize $T(P_i)$. By defining fitness in this way, higher fitness values correspond to better schedules, and the least fit individual has fitness equal to 1.

Selection. We use elitism to select the E most fit population members containing unique permutations. The elite members do not undergo crossover or mutation, and are copied into the next generation as is, ensuring that the population always contains the best solution found thus far, and also preventing convergence upon a single solution since the population always contains at least E unique permutations.

We use Stochastic Universal Sampling (SUS) [2] to select the remaining $PopSize - E$ members of the population for the next generation. All $PopSize$ members of the current population are available for selection by SUS, including the elite members. SUS selects Pop_i with probability proportional to $fitness(Pop_i)$ just like the more common fitness proportionate selection (i.e., “weighted roulette wheel”). However, SUS is analogous to spinning a wheel with k equidistant pointers a single time to select k members simultaneously, whereas fitness proportionate selection spins a 1-pointer k times to select k members. Baker showed that SUS reduces selection bias [2]; and it is also more efficient (e.g., only 1 random number need be generated to select an entire population).

Crossover. In each generation, the $PopSize - E$ non-elite members are paired randomly. For each pair, Pop_i, Pop_j , an arbitrary member is chosen (e.g., Pop_i from this hypothetical pair). With probability C_i (the crossover rate from the chosen member of the pair), NWOX [7] is applied to the permutations contained in the pair. Figure 1 illustrates the behavior of both OX and NWOX for comparison. Two random cross points are chosen, similar to a 2-point bit-string crossover. Child C_i gets the cross region elements from parent P_j in the same positions, and likewise for C_j . For the original OX [16], the remaining elements for C_i are taken in the same relative order as they appear in P_i , filled into C_i beginning just after the cross region, wrapping around to the start. By contrast, NWOX fills in these remaining elements beginning at the left end of C_i , skipping over the cross region, and continuing to its end. NWOX has the ef-

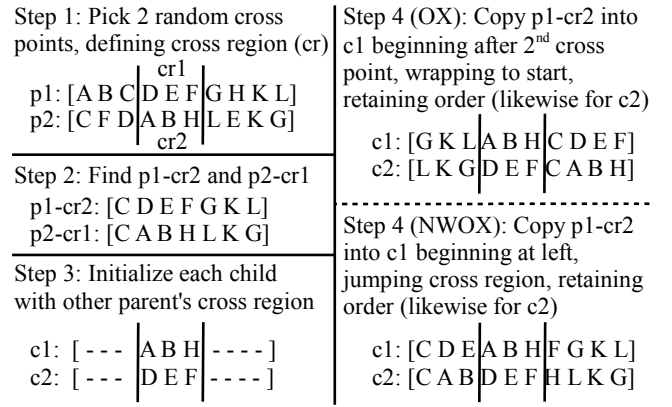


Figure 1: The NWOX and OX operators.

fect of better keeping elements near their original locations in the parents, whereas OX tends to displace elements from one end to the other of the permutation. For a scheduling problem like the one we consider here, the smaller positional displacements of NWOX is beneficial. We do not apply any crossover operation to the adaptive GA parameters.

Mutation. In each generation, for each of the $PopSize - E$ non-elite members Pop_i , we apply insertion mutation with probability M_i , the mutation rate. Insertion mutation removes a random element, and then reinserts it at a different randomly chosen location, shifting the elements between the removal and reinsertion points one place each. In our prior work, we compared its performance on this very scheduling problem to alternatives both in a GA [7] as well as within simulated annealing [8]. As shown in our prior work on permutation search landscape analysis [11], insertion mutation is ideally suited to permutation problems with directed edges (e.g., the asymmetric and sequence-dependent setups) and where positional information also influences fitness (e.g., general position within permutation affects job tardiness).

Parameter Initialization and Adaptation. The initial values for the C_i and M_i are generated uniformly at random from $[0.1, 1.0]$. These evolve during the search with Gaussian mutation [22], controlled by σ_i . Hinterding recommends initializing these around 0.1 [22]; and thus, we randomly generate the initial σ_i uniformly from $[0.05, 0.15]$.

In each generation, C_i, M_i , and σ_i of each of the $PopSize - E$ non-elite members undergo Gaussian mutation as follows:

$$C_i = C_i + N(0, \sigma_i), \quad (7)$$

$$M_i = M_i + N(0, \sigma_i), \quad (8)$$

$$\sigma_i = \sigma_i + N(0, 0.01), \quad (9)$$

where $N(0, \sigma)$ is a normally distributed random variable with mean 0 and standard deviation σ . If C_i is greater than 1, it is reset to 1 (likewise for M_i) to ensure it remains a valid probability. If C_i is less than 0.1, it is reset to 0.1 (likewise for M_i). Similarly, σ_i is allowed to vary only within $[0.01, 0.2]$. The range for σ_i and its adaptation rule (Equation 9) differ from Hinterding [22] but are consistent with his recommendation that the standard deviation of the Gaussian mutation be approximately 0.1 times the maximum value of the real-valued parameter. The maximum

values of C_i and M_i are both 1; and the mid-point of the allowable range of σ_i is approximately 0.1, which is further mutated with a Gaussian with standard deviation 0.01.

4. EXPERIMENTS

4.1 Experimental Design

The set of benchmark instances for the weighted tardiness problem with sequence-dependent setups consists of 120 instances, 40 each of loose due dates, medium due dates, and tight due dates. Of these, 22 instances have an optimal solution with weighted tardiness equal to 0 (all of these are loose due date instances). We use the following commonly employed metrics in the analysis of our experiments for this problem. Most commonly reported is the average percentage deviation from the optimal solutions, averaged only across the instances with non-zero optimal values:

$$\% \Delta \text{Opt} = \frac{100}{N} \sum_{i=1}^N \frac{(S_i - O_i)}{O_i}, \quad (10)$$

where S_i and O_i are the value of the solution found for problem instance i and its optimal solution, respectively. One problem with this metric is that it ignores the 22 problem instances whose optimal solutions have weighted tardiness equal to 0. Thus, we also report the percentage deviation of the sum across the problem instances relative to the sum of the optimal solutions:

$$\% \Delta \text{OptSum} = 100 \frac{\sum_{i=1}^N S_i - \sum_{i=1}^N O_i}{\sum_{i=1}^N O_i}. \quad (11)$$

We consider the following run lengths (in maximum number of generations): $\{10^2, 10^3, 10^4, 10^5, 10^6\}$. For each alternative algorithm in our experiments, we solve each instance 10 times for each run length. The reported values of $\% \Delta \text{Opt}$ are thus averages of $10N$ runs (for N instances), while the reported $\% \Delta \text{OptSum}$ are 10 run averages. We use t-tests to test the significance of the $\% \Delta \text{OptSum}$ results. However, since $\% \Delta \text{Opt}$ is an average across multiple problem instances with values of varying scale, the normality requirement for the t-test is not met. So we test the significance of the $\% \Delta \text{Opt}$ results using the Wilcoxon signed rank test.

We conduct our experiments on an Ubuntu 14.04 Server, with 32GB memory and two Intel Xeon L5520 Quad-Core CPUs (2.27GHz). The L5520 supports hyper-threading with two threads per core, so our server has a total of 16 logical cores. We implement our experiments using Java 8 and the Java HotSpot 64-bit Server VM. Our GA is not implemented with multi-threading, so it does not explicitly utilize the multi-core architecture of our server, though the VM would certainly do so for garbage collection.

We compare three GA schemes ($\text{PopSize} = 100$ in each):

Manually Tuned: As a baseline for comparison, we use our prior GA for the problem in which we manually tuned the GA parameters using a small set of instances not contained in the benchmark [7]. This GA also uses NWOX and Insertion for crossover and mutation, as well as SUS selection. The manually tuned parameters are $E = 3$, $C = 0.95$, and $M = 0.65$. A mutation rate of 0.65 would be unusually high for a bit-string GA where it is a per-bit mutation rate. However, the mutation rate for a permutation-based GA is a per population member rate (i.e., the probability that a single

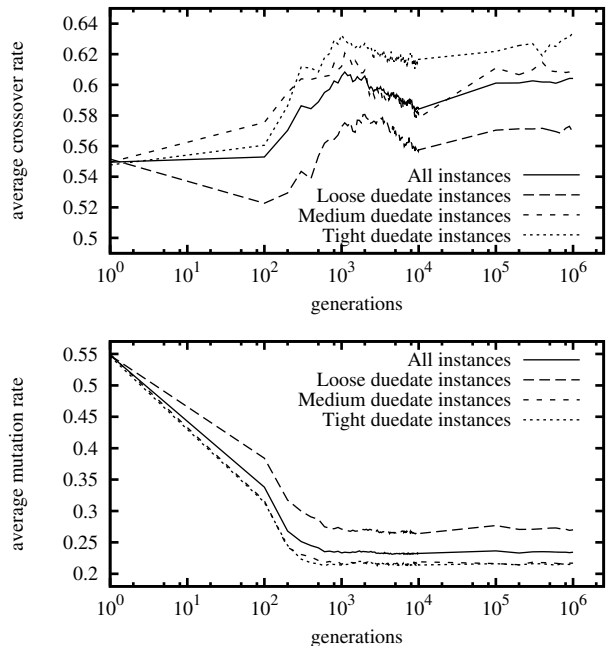


Figure 2: GA control parameter evolution: population averages across instances of designated classes.

mutation is applied to a population member), so mutation rates tend to be higher than encountered for bit-strings.

Adaptive: This is our adaptive GA as described in this paper, where each population member has its own C_i and M_i , which evolve during search. The elitism parameter was tuned manually using a small set of instances outside the benchmark, and our experiments use $E = 5$ retaining the 5 most fit population members unaltered in each generation.

Evolved: As a third option, we consider a GA with fixed parameters derived from the final parameters of a 1000000 generation run of the GA. Figure 2 shows the evolution of the average population C and M averaged across all benchmark instances, as well as the three instance classes. For the first 10000 generations at 100 generation intervals, we computed C as the average of the C_i for the population, and M as the average of the M_i for the population. We then continued this at 100000 generation intervals for the remainder of the run. Note that we did not perform any cross validation and we used the very benchmark instances, so no generalizations can be made with regard to anticipated performance on future instances. This parameter set is deliberately over-fitted to serve as a performance bound with which to compare our adaptive GA—e.g., if we had access to a clairvoyant oracle that could tell us what parameters we should use, how well would we do? The evolved parameters are: $C = 0.60$ and $M = 0.23$ (we still use $E = 5$ as above).

4.2 Results

The dynamic tracking of the crossover and mutation rates (Figure 2) show interesting behavior. First, the average mutation rate decreases steadily for around the first 1000 generations, and then remains constant; and the average crossover rate does not stabilize until after 10000 generations. Second, observe that for loose due dates, the adaptive GA settles

Table 1: Comparison of manually tuned parameters and adaptive parameters: % Δ OptSum and % Δ Opt.

Parameters (# gens.)	% Δ OptSum		% Δ Opt	
	Ave.	Best	Ave.	Best
Manual	59.34%	48.23%	533.79%	405.57%
Adaptive (10^2)	49.25%	38.78%	392.79%	285.28%
	p: $4.5 * 10^{-15}$		p: 0.00	
Manual	21.71%	16.01%	154.39%	115.39%
Adaptive (10^3)	14.78%	9.81%	78.71%	52.33%
	p: $1.1 * 10^{-15}$		p: 0.00	
Manual	11.13%	7.32%	62.71%	42.24%
Adaptive (10^4)	9.95%	6.04%	48.17%	30.08%
	p: $1.1 * 10^{-07}$		p: $8.9 * 10^{-15}$	
Manual	7.30%	4.30%	39.64%	24.84%
Adaptive (10^5)	6.85%	3.99%	33.37%	20.87%
	p: 0.001		p: $6.0 * 10^{-11}$	
Manual	4.83%	2.46%	26.98%	16.60%
Adaptive (10^6)	4.98%	2.43%	22.61%	13.44%
	p: 0.133		p: 0.010	

Table 2: Comparison of manually tuned parameters and adaptive parameters: #Opt and CPU time.

Parameters (# gens.)	#Opt	CPU Time
Manual (10^2)	3	0.06
Adaptive (10^2)	7	0.05
Manual (10^3)	14	0.30
Adaptive (10^3)	121	0.19
Manual (10^4)	145	1.52
Adaptive (10^4)	160	0.87
Manual (10^5)	164	10.20
Adaptive (10^5)	168	7.15
Manual (10^6)	169	94.92
Adaptive (10^6)	176	68.51

upon a higher mutation rate and a lower crossover rate than it does for medium and tight due dates. In general, crossover rate is higher the tighter the due dates are for the instance, so crossover appears more productive than mutation for the harder, tight due date instances.

Tables 1 and 2 summarize the experimental data comparing the manually tuned and the adaptive parameters. The CPU times are in seconds, averaged across 1200 runs (10 runs on each of 120 problem instances). In addition to the averages for % Δ OptSum and % Δ Opt, we also show the results of the best runs. The #Opt is the number of runs (out of 1200) where the optimal solution was found.

Figures 3 and 4 show % Δ OptSum and % Δ Opt, respectively, averaged over all problem instances as well as by due date tightness, as the number of generations increases. The graphs are at log-log scale.

For short runs (100 generations), the evolved fixed parameters do lead to slightly better results compared to our adaptive GA across all instances of the benchmark set (difficult to see on the graphs, though p-value less than 10^{-10} for % Δ Opt and less than 10^{-7} for % Δ OptSum, shows differences to be extremely statistically significant for 100 generation runs). However, for longer run lengths, there is no statistical significance between the performance of these two

algorithms (with the exception of loose due date instances and runs between 10000 and 100000 generations where the fixed parameters perform slightly better). This shows that the adaptive GA needs only a small amount of time to evolve parameters that lead to effective problem solving; and that the use of sub-optimal parameters early in the run have a negligible effect on the run as a whole.

The adaptive and the evolved fixed parameters greatly out-perform the manually tuned parameters on both metrics with the exception of very long runs (10^6 generations) on tight due date instances, where the manually tuned parameters lead to slightly better performance. The differences for % Δ Opt between the adaptive GA and the manually tuned GA, are statistically significant for all problem classes and all run lengths other than 10^5 generation runs on tight due date instances (p-value is 0.13 in that case, but is otherwise no higher than 0.01, and in some cases less than 10^{-14}). The % Δ OptSum results are similar—i.e., statistically significant except for 10^5 generations on tight due date instances, and at 10^6 generations across the entire benchmark ($p = 0.133$).

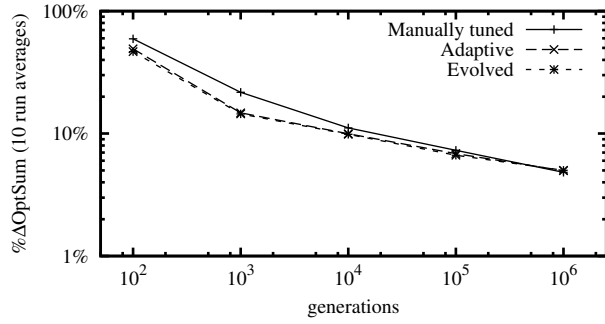
The results are more dramatic when computational time is considered. Figure 5 shows CPU time in seconds, as a function of number of generations, averaged across all instances as well as separated out by due date tightness (x-axis is at log scale). Although the adaptive GA has the added computation required to adapt the control parameters, it requires an overall lower CPU time compared to the manually tuned parameters. The manually tuned parameters are higher crossover and mutation rates, so more GA operations are performed in the same number of generations as compared to the adaptive GA. The evolved fixed parameters obviously require the least amount of CPU time for an equivalent number of generations. Recall that this option uses the final population average parameter values evolved by the adaptive GA, but right from the start without the added overhead of parameter adaptation.

Figure 6 shows % Δ OptSum and % Δ Opt, computed across the entire benchmark set, as functions of CPU time, rather than number of generations. The performance separation is more pronounced when CPU time is considered (e.g., contrast with Figures 3a and 4a, respectively).

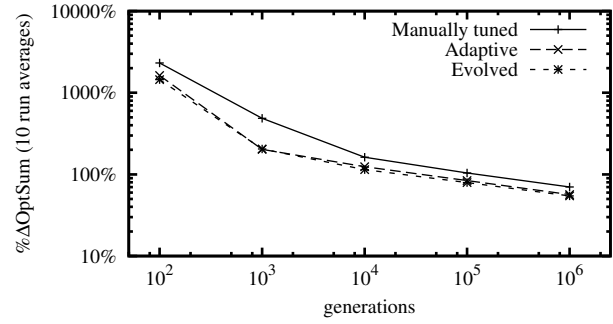
5. CONCLUSIONS

In this paper, we presented an adaptive GA for an NP-Hard problem: weighted tardiness scheduling with sequence-dependent setups. GA control parameters, such as mutation and crossover rates, are all too often tuned in a tedious, ad hoc trial-and-error manner. Our GA evolves the control parameters simultaneously with the solution to the problem, thus eliminating the need to tune the control parameters ahead of time. An additional advantage is that the parameters can be tuned to the problem instance at hand dynamically during search. For example, we saw in Figure 2 and its associated discussion that the adaptive GA evolved a higher mutation rate and a lower crossover rate for loose due date instances as compared to tight due date instances.

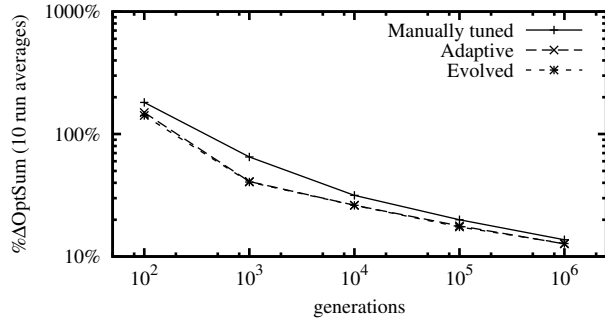
Although there is computational overhead associated with evolving the control parameters, it seems to be negligible. For example, we saw that the adaptive GA finds significantly better solutions for the scheduling problem of this paper as compared to an existing manually tuned GA that uses the same genetic operators; and that it does so with less computational cost since the evolved parameters are lower



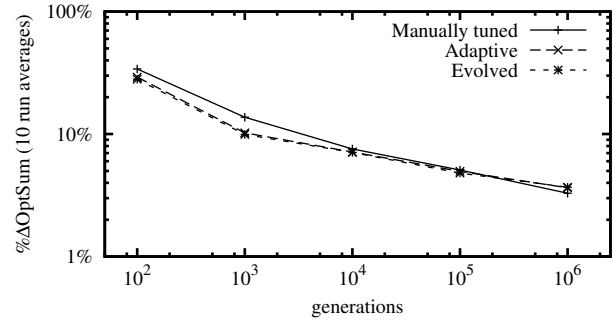
(a) All instances



(b) Loose due-date instances

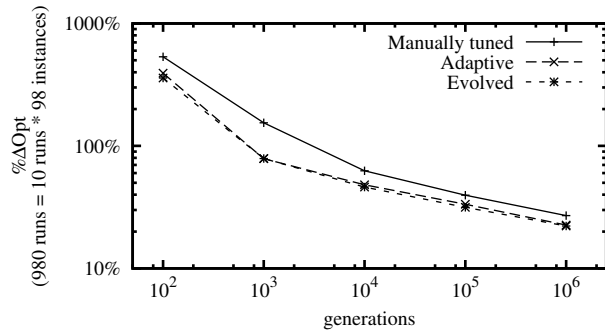


(c) Medium due-date instances

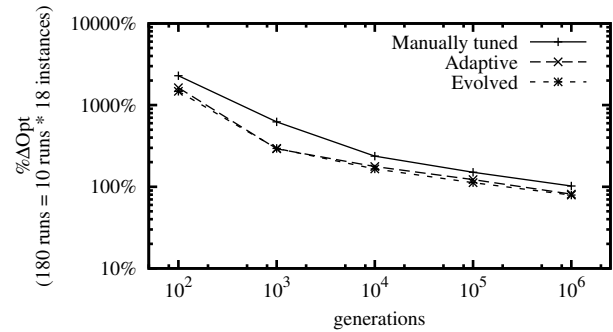


(d) Tight due-date instances

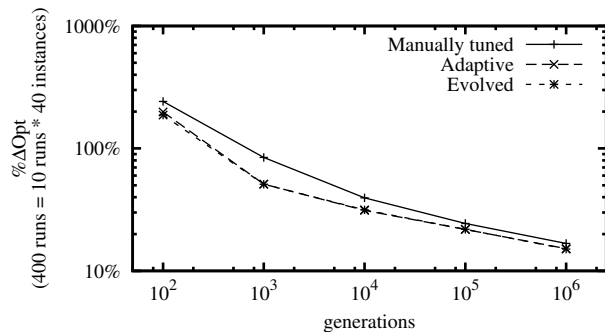
Figure 3: $\% \Delta \text{OptSum}$ (10 run averages) over: (a) all 120 benchmark instances, (b) all 40 loose due-date instances, (c) all 40 medium due-date instances, and (d) all 40 tight due-date instances.



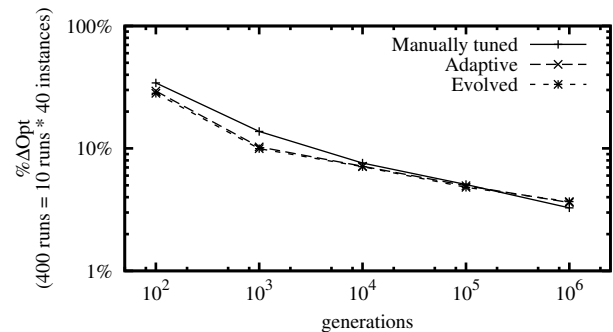
(a) All instances



(b) Loose due-date instances



(c) Medium due-date instances



(d) Tight due-date instances

Figure 4: $\% \Delta \text{Opt}$ (10 runs per instance) over: (a) the 98 instances with optimal > 0 , (b) the 18 loose due-date instances with optimal > 0 , (c) all 40 medium due-date instances, and (d) all 40 tight due-date instances.

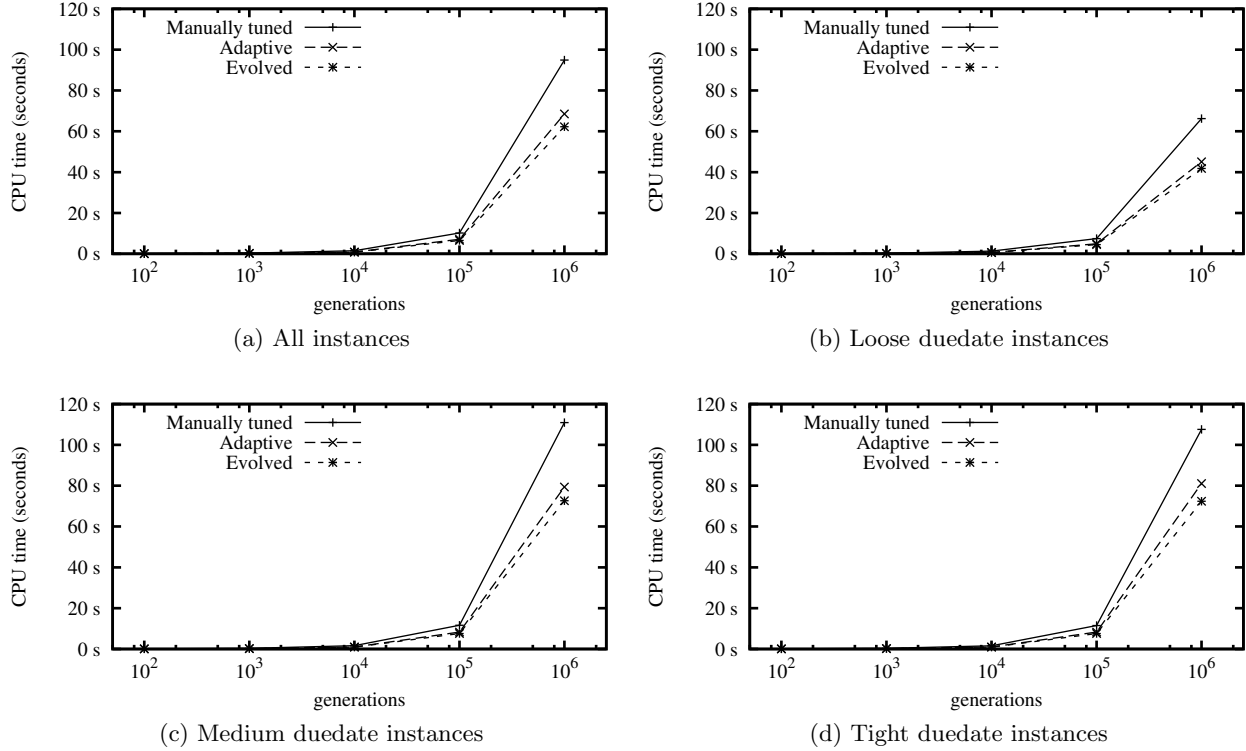


Figure 5: CPU time in seconds (10 run averages) over: (a) all 120 benchmark instances, (b) all 40 loose duedate instances, (c) all 40 medium duedate instances, and (d) all 40 tight duedate instances.

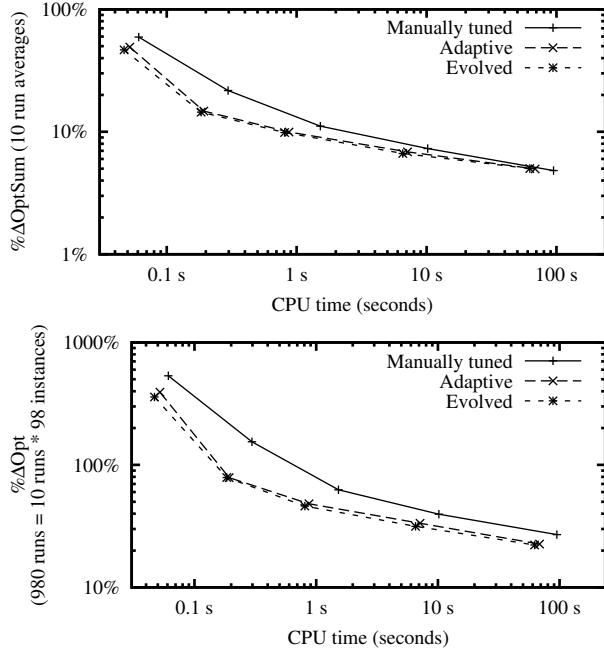


Figure 6: $\% \Delta \text{OptSum}$ and $\% \Delta \text{Opt}$ expressed as functions of CPU time in seconds (log-log scale).

rates of crossover and mutation. We also saw that the solutions found by the adaptive GA are no worse than if we somehow had access to the final evolved control parameter values from the beginning. Thus, the approach efficiently converges upon effective parameter values.

We plan to further investigate the potential effectiveness of our approach for parameter tuning. For example, we saw that if we use the final average population parameter values evolved by the adaptive GA right from the start of the search, that we achieve equivalent performance to the adaptive GA but without the overhead of parameter evolution. We will investigate whether this generalizes to other yet unseen problem instances. We will also investigate whether it is possible to bootstrap the parameter adaptation using the evolved parameters from earlier runs.

6. REFERENCES

- [1] A. Aleti and I. Moser. Entropy-based adaptive range parameter control for evolutionary algorithms. In *Proc. GECCO*, pages 1501–1508. ACM, 2013.
- [2] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proc. ICGA*, pages 14–21, 1987.
- [3] C. Bierwirth, D. Mattfeld, and H. Kopfer. On permutation representations for scheduling problems. In *Proc. PPSN*, pages 310–318. Springer, 1996.
- [4] J. Branke and J. A. Elomari. Meta-optimization for parameter tuning with a flexible computing budget. In *Proc. GECCO*, pages 1245–1252. ACM, 2012.
- [5] V. A. Cicirello. *Boosting Stochastic Problem Solvers Through Online Self-Analysis of Performance*. PhD

- thesis, Robotics, Carnegie Mellon University, 2003.
- [6] V. A. Cicirello. Weighted tardiness scheduling with sequence-dependent setups: A benchmark library. Tech. report, ICL Lab, CMU, Feb. 2003. <http://loki.stockton.edu/~cicirelv/benchmarks/>.
- [7] V. A. Cicirello. Non-wrapping order crossover: An order preserving crossover operator that respects absolute position. In *Proc. GECCO 2006*, pages 1125–1131. ACM, July 2006.
- [8] V. A. Cicirello. On the design of an adaptive simulated annealing algorithm. In *Proc. CP 2007 First Workshop on Autonomous Search*. AAAI Press, Sept. 2007.
- [9] V. A. Cicirello. Heuristic sequencing crossover: Integrating problem dependent heuristic knowledge into a genetic algorithm. In *Proc. 23rd FLAIRS*, pages 14–19. AAAI Press, 2010.
- [10] V. A. Cicirello. On the effects of window-limits on the distance profiles of permutation neighborhood operators. In *Proc. Int. Conf. Bioinspired Information and Communications Technologies*, pages 28–35, 2014.
- [11] V. A. Cicirello. The permutation in a haystack problem and the calculus of search landscapes. *IEEE Transactions on Evolutionary Computation*, Forthcoming 2016.
- [12] V. A. Cicirello and R. Cernera. Profiling the distance characteristics of mutation operators for permutation-based genetic algorithms. In *Proc. 26th FLAIRS*, pages 46–51. AAAI Press, May 2013.
- [13] V. A. Cicirello and S. F. Smith. Modeling GA performance for control parameter optimization. In *Proc. GECCO*, pages 235–242, July 2000.
- [14] V. A. Cicirello and S. F. Smith. Enhancing stochastic search performance by value-biased randomization of heuristics. *Journal of Heuristics*, 11(1):5–34, 2005.
- [15] V. A. Cicirello and S. F. Smith. The max k -armed bandit: A new model of exploration applied to search heuristic selection. In *Proc. 20th Nat. Conf. Artificial Intelligence*, pages 1355–1361. AAAI Press, 2005.
- [16] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proc. IJCAI*, pages 162–164, 1985.
- [17] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, Ann Arbor, MI, 1975.
- [18] A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.
- [19] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.
- [20] D. E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In *Proc. ICGA*, pages 154–159, 1985.
- [21] J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.
- [22] R. Hinterding. Gaussian mutation and self-adaption for numeric genetic algorithms. In *IEEE CEC*, pages 384–389, Nov 1995.
- [23] C.-J. Liao and H.-C. Juan. An ant colony optimization for single-machine tardiness scheduling with sequence-dependent setups. *Computers and Operations Research*, 34(7):1899–1909, 2007.
- [24] C.-J. Liao, H.-H. Tsou, and K.-L. Huang. Neighborhood search procedures for single machine tardiness scheduling with sequence-dependent setups. *Theoretical Computer Science*, 434:45–52, 2012.
- [25] S. Luke and A. K. A. Talukder. Is the meta-ea a viable optimization method? In *Proc. GECCO*, pages 1533–1540. ACM, 2013.
- [26] L. D. Merkle. Metaoptimization of the in-lining priority function for a compiler targeting a polymorphous computing architecture. In *Proc. GECCO*, pages 1921–1928. ACM, 2008.
- [27] T. E. Morton and D. W. Pentico. *Heuristic Scheduling Systems: With Applications to Production Systems and Project Management*. John Wiley and Sons, 1993.
- [28] Y. Nagata and S. Kobayashi. Edge assembly crossover: A high-power genetic algorithm for the travelling salesman problem. In *Proc. ICGA*, pages 450–457, 1997.
- [29] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proc. ICGA*, pages 224–230, 1987.
- [30] A. K. Sen and A. Bagchi. Graph search methods for non-order-preserving evaluation functions: Applications to job sequencing problems. *Artificial Intelligence*, 86(1):43–73, 1996.
- [31] M. Serpell and J. E. Smith. Self-adaptation of mutation operator and probability for permutation representations in genetic algorithms. *Evolutionary Computation*, 18(3):491–514, 2010.
- [32] A. Sinha, P. Malo, P. Xu, and K. Deb. A bilevel optimization approach to automated parameter tuning. In *Proc. GECCO*, pages 847–854. ACM, 2014.
- [33] G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
- [34] S. Tanaka and M. Araki. An exact algorithm for the single-machine total weighted tardiness problem with sequence-dependent setup times. *Computers and Operations Research*, 40(1):344–352, 2013.
- [35] C. L. Valenzuela. A study of permutation operators for minimum span frequency assignment using an order based representation. *Journal of Heuristics*, 7(1):5–21, 2001.
- [36] Y.-K. Wang, K.-C. Fan, and J.-T. Horng. Genetic-based search for error-correcting graph isomorphism. *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, 27(4):588–597, 1997.
- [37] J.-P. Watson, C. Ross, V. Eisele, J. Denton, J. Bins, C. Guerra, L. D. Whitley, and A. E. Howe. The traveling salesrep problem, edge assembly crossover, and 2-opt. In *Proc. PPSN*, pages 823–834, 1998.
- [38] S. Wessing, M. Preuss, and G. Rudolph. When parameter tuning actually is parameter control. In *Proc. GECCO*, pages 821–828. ACM, 2011.
- [39] H. Xu, Z. Lü, and T. C. Cheng. Iterated local search for single-machine scheduling with sequence-dependent setup times to minimize total weighted tardiness. *J. of Scheduling*, 17(3):271–287, 2014.