Heuristic Sequencing Crossover: Integrating Problem Dependent Heuristic Knowledge into a Genetic Algorithm

Vincent A. Cicirello
Computer Science and Information Systems
Richard Stockton College
Pomona, NJ 08240

Abstract

Dispatch scheduling policies offer a computationally inexpensive approach to many scheduling problems as an alternative to more processor intensive search algorithms. They are especially useful in dynamic situations where problem solving may be temporally constrained. However, dispatch heuristics have also been effectively used for guidance within more intensive search algorithms, expanding their applicability well beyond simple myopic job or task selection. In this paper, we present a new crossover operator for genetic algorithms (GA) where a permutation representation is employed. Our new operator directly integrates problem-dependent knowledge in the form of a dispatch scheduling policy into the crossover step of a GA which is usually a problem-independent recombination. Our novel crossover operator, Heuristic Sequencing Crossover (HeurX), recombines elements of M parents into a single child and is specifically designed for steady-state GAs. We empirically demonstrate its effectiveness on a strongly NP-Hard scheduling problem known as Weighted Tardiness Scheduling with Sequence-Dependent Setups. We compare our approach to a generational GA using a problem-independent crossover operator. We show the potential power of integrating problem-dependent knowledge via a dispatch heuristic into what is traditionally a problem independent problem solver.

Introduction

Genetic Algorithms (GA) and other Evolutionary Computation algorithms are approaches to problem solving inspired by natural evolutionary processes (Goldberg 1989). GAs are widely used for solving combinatorial optimization problems. The key operators of the genetic search are known as crossover and mutation and GAs are usually referred to as population-based as the search maintains a population of candidate solutions to the problem. Crossover operators recombine elements from pairs (usually) of individuals of the population into children that replace individuals in the population. Mutation operators introduce small random changes to individuals. Crossover and mutation operators are almost always problem independent operators that require zero knowledge of the problem. The only problem specific knowledge required by the typical GA is a fitness function that is used to evaluate the quality of the members of the population, and which is used in the selection operator as the source of a metaphor for the survival of the fittest. The more common type of GA is generational where the evolutionary operators are iterated over a series of generations, each generation comprised of children (some mutated) of the fittest members of the previous generation. In this paper, we actually employ what is known as a steady-state GA, which differs from the generational model in that at each step a small number of the least fit members of the population are replaced by children of other members of the population.

Despite minimal knowledge of the problem to be solved, GAs are often effective general problem solvers. One of our objectives is to explore potential benefits of integrating knowledge of the problem into a GA crossover operator. In this paper, we introduce a crossover operator called Heuristic Sequencing Crossover (HeurX). We define it in a problem independent way. One of its inputs, however, is a heuristic with problem dependent knowledge. HeurX treats the heuristic as a black-box, enabling the design of a GA with the typical structure that is disconnected from the application, until provided with a fitness function, and in our case, a heuristic. Additionally, HeurX is designed for GAs that use the permutation representation and not for the more commonly employed bitstrings. The permutation representation of a GA represents individuals in the population by a permutation of elements and is particularly suitable for scheduling problems where the task is to find an ordering over a set of jobs or tasks to optimize some objective. The permutation representation requires specialized operators, and there is an extensive literature on problem independent crossover operators for permutations (Oliver, Smith, and Holland 1987; Davis 1985; Goldberg and Lingle 1985; Watson et al. 1998; Moraglio and Poli 2005; Cicirello 2006; Ho and Chen 2000; Starkweather et al. 1991).

In order to apply the HeurX operator within a GA to solve a scheduling problem or some other combinatorial optimization problem, we require a heuristic with knowledge of the problem. For many scheduling problems, there exist what are referred to as dispatch scheduling policies (Morton and Pentico 1993). A dispatch policy is a type of heuristic most useful in dynamic situations. Their intended application is the myopic selection of a job or task to next process on a machine. Each of the available jobs are evaluated by the dispatch policy and the job with highest evaluation is selected and processed. If time is available, one can use a dispatch policy to first sequence the jobs and then use a local search to locally improve it. But if time is highly constrained, often the dispatch policy is used to simply myopically select the next job. Many of the available dispatch policies for common scheduling objectives were designed around human expert knowledge of the particular problem, and as such are a source of problem specific heuristic knowledge. In this paper, we will specifically use our HeurX operator to integrate the problem dependent heuristic knowledge inherent in dispatch scheduling policies into a steady-state GA.

This is certainly not the first use of dispatch scheduling policies to guide a more intensive search procedure. For example, stochastic sampling search procedures search a randomized neighborhood of a dispatch policy's suggested solution (Bresina 1996; Cicirello and Smith 2005). This type of heuristic has also been used to guide backtracking search in constrained optimization (Franck and Neumann 1996). Applications of Ant Colony Optimization (ACO) (Dorigo and Di Caro 1999) to scheduling problems also often use dispatch heuristics for guidance. However, we believe that our HeurX operator is the first GA crossover operator designed to integrate dispatch scheduling policies into a GA.

To demonstrate the benefit of introducing problem dependent knowledge into a GA crossover operator, we turn to a problem known as Weighted Tardiness Scheduling with Sequence-Dependent Setups, which is NP-Hard in the strong sense (Garey and Johnson 1979). A particularly challenging aspect of the problem are the sequence-dependent setup constraints, which greatly magnify the computational hardness of the problem (Sen and Bagchi 1996). We empirically compare our steady-state GA using the HeurX operator to one of the current best meta-heuristics for the problem, a generational GA that uses the permutation crossover operator Non-Wrapping Order Crossover (NWOX) (Cicirello 2006). We show that HeurX enables us to effectively integrate problem dependent knowledge into a GA as compared to using a problem independent crossover operator.

The Heuristic Sequencing Crossover Operator

The Heuristic Sequencing Crossover (HeurX) operator is an adaptation of the general procedure usually employed to schedule a set of jobs via dispatch policy. Before presenting the HeurX operator, we first briefly summarize the typical application of a dispatch policy. Figure 1 provides pseudocode for a simple dispatch scheduling procedure. A dispatch scheduling procedure iteratively selects the job from the set of remaining jobs J that is most highly valued by the dispatch heuristic H. This job is placed in the next available position in the sequence. One can view the dispatch heuristic H as a heuristic approximation of the priority of the job. Jobs with high values of the dispatch heuristic are believed by the heuristic to be most critical to schedule next.

Figure 2 provides pseudocode for the Heuristic Sequencing Crossover (HeurX) operator. In addition to the set of elements J (e.g., jobs in a scheduling problem) that we need to sequence and a dispatch heuristic H, the HeurX operator also takes as input a set P of parent permutations. Each $p \in P$ is a permutation of the elements of J. Although most GA crossover operators recombine elements of 2 parents, the HeurX operator can be given any number of parents $M \geq 2$. Regardless of the number of parents M, the HeurX operator produces a single child permutation C.

```
Dispatch Scheduling
Inputs: An unordered set of elements $J$.
        A dispatch heuristic $H$.
Outputs: An ordered sequence $S$.
$S \longleftarrow \emptyset$
while $J \neq \emptyset$ do
    next $\longleftarrow \arg\max_{j \in J} H(j)$
    Add next to the end of $S$
    $J \longleftarrow J - \textit{next}$
end
```

Figure 1: Sequencing a set of jobs via a dispatch policy.

```
The HeurX Operator
Inputs: An unordered set of elements $J$.
        A dispatch heuristic $H$.
        A set $P$ of $M$ parent permutations.
        Each $p \in P$ is a permutation of the elements of $J$.
Outputs: A child permutation $C$.
$C \longleftarrow \emptyset$
while $J \neq \emptyset$ do
    eligible $\longleftarrow \emptyset$
    for each $p \in P$ do
        $e \longleftarrow$ find first element of $p$ that is still in $J$
        eligible $\longleftarrow$ eligible $\cup \{e\}$
    end
    next $\longleftarrow \arg\max_{j \in \textit{eligible}} H(j)$
    Add next to the end of $C$
    $J \longleftarrow J - \textit{next}$
end
```

Figure 2: The HeurX operator: pseudocode.

The HeurX operator constructs the child permutation similar to the typical application of a dispatch policy. However, prior to the selection of the next element to add to the child sequence C, the HeurX operator uses the set of parents P to form a restricted set of "eligible" elements, rather than selecting from the entire set of remaining elements in J. Specifically, the first element e in each of the M parents that has not yet been added to the child C is added to a set of "eligible" elements. The dispatch heuristic H is then used to select an element from this smaller set.

Early in the construction of a child, HeurX uses the myopic dispatch policy to select from at most M jobs rather than from all jobs. The set of eligible jobs can be smaller than M if the next non-scheduled job is the same in more than one parent. Many dispatch policies tend to be more accurate in job selection toward the end when fewer jobs remain. If a dispatch policy makes a poor selection, it is more likely (in most cases) to be toward the beginning of the sequence. If the job that is not really as important as the heuristic believes it to be is not at the very beginning of any of the parents, then HeurX will not place it at the beginning

```
A Steady-State GA Utilizing the HeurX Operator
Inputs: An unordered set of elements $J$.
        A dispatch heuristic $H$.
        Number of parents $M$ to use during crossover.
        The population size $S$.
        The max number, Evals, of crossover operations.
        The optimization objective, $F$, to minimize.
        Optional: A local search $L$ to apply to the children
                  produced during crossover.
Outputs: The ordered sequence, MinS, with lowest value
         of $F(i)$ of all sequences $i$ found by the search.
Population $\longleftarrow$ $S$ random permutations of $J$
MinS $\longleftarrow \arg\min_{p \in Population} F(p)$
for $i$ from 1 to Evals do
    parents $\longleftarrow$ $M$ random members of Population
    $C \longleftarrow$ HeurX(J,H,parents)
    Optional Step: Locally optimize $C$ using $L$.
    Population $\longleftarrow$
        (Population $- \arg\max_{p \in Population} F(p)) \cup \{C\}$
    if $F(C) < F(MinS)$ then MinS $\longleftarrow C$
end
```

Figure 3: Steady-state GA utilizing the HeurX operator.

of the child as it will not be in the eligible set of elements.

Later in the construction of the child, the number of remaining jobs begins to decline well below M, there is an increasing chance that the eligible set will contain all or nearly all of the remaining jobs. When this occurs, HeurX will reduce to dispatch scheduling. This is desirable given that dispatch policies are most accurate when there are fewer jobs to schedule. The higher the number of parents M, the earlier HeurX reduces to dispatch scheduling during child construction. The lower the number of parents M, the more the evolutionary processes of the GA affect the search. The runtime complexity of the HeurX operator is $\bigcirc(M \cdot |J|)$.

HeurX and a Steady-State Genetic Algorithm

Although there are likely to be other appropriate structures for a GA that uses our HeurX operator, we recommend its use in a steady-state GA. The property of the HeurX operator where a single child is produced from several (M) parents does not lend itself well to the generational GA. In the typical generational GA, after selection the members of the population are paired. Some pairs undergo crossover producing pairs of children to replace the parents. With the HeurX operator, M parents produce 1 child, and thus does not naturally map into the structure of a generational GA.

Instead, we use a steady-state model (Figure 3). We assume we must minimize the objective F. We begin with S random permutations. We then perform *Evals* iterations, where each iteration consists of the following. M parents are selected randomly from the population with uniform probability. The HeurX operator produces a single child C from those M parents. If desired, one can locally optimize C using a local search such as a hill climber. The final step of an iteration replaces the worst individual (i.e., with largest F value) of the population with C. Throughout, we keep track of the best sequence (i.e., with minimum F value).

The cost of an iteration is $\bigcirc(M + M \cdot |J| + \log(S)) = \bigcirc(\max(M \cdot |J|, \log(S)))$. If we store the population in a heap, each of the M parent selections can be done in constant time since they would be simple array accesses, and both the removal of the worst member and addition of the new child can be done in $\bigcirc(\log(S))$. It is also possible to decrease the constant factor of that term by combining the remove worst and add child into a single operation. Specifically, one can replace the root of the heap (the worst member of the population) with the newly constructed child, and then execute a heapify on the root (see any algorithms book for details of heapify, such as (Cormen, Leiserson, and Rivest 1990)). The alternative would be to perform a remove which would involve replacing the root with the element in the last position of the array used by the heap (one of the leaves) and executing heapify on the root, followed by the addition of the child to the heap involving placement of the child as a leaf and then percolating it up the heap. The $M \cdot |J|$ term is the cost of the call to HeurX. Unless the population size S is large, the dominant term is the cost of the call to HeurX, making the cost of a single iteration of the GA $\bigcirc(M \cdot |J|)$ with the cost of the overall GA $\bigcirc(\mathit{Evals} \cdot M \cdot |J|)$.

Experimental Analysis

The Problem: Weighted Tardiness Scheduling

Weighted tardiness scheduling with sequence-dependent setups consists of a set of jobs $J = \{j_0, j_1, \ldots, j_N\}$. Each job j has a weight w_j, duedate d_j, and process time p_j. Furthermore, $s_{i,j}$ is the setup time required immediately prior to the start of processing job j if it follows job i. It is not necessarily the case that $s_{i,j} = s_{j,i}$. The 0-th "job" is the start of the problem ($p_0 = 0$, $d_0 = 0$, $s_{i,0} = 0$, $w_0 = 0$). Its purpose is to specify the setup of each job if sequenced first.

The weighted tardiness objective is to minimize:

$$T = \sum_{j \in J} w_j T_j = \sum_{j \in J} w_j \max(c_j - d_j, 0), \qquad (1)$$

where T_j is the tardiness of job j; and c_j, d_j are the completion time and duedate of job j. The completion time is the sum of the process times and setup times of all jobs that come before j in the sequence plus that of j. If $\pi(j)$ is the position of job j in the sequence, then c_j is:

$$c_j = \sum_{i,k \in J, \pi(i) <= \pi(j), \pi(i) = \pi(k)+1} p_i + s_{k,i}. \qquad (2)$$

Single-machine scheduling to optimize weighted tardiness is NP-Hard even if setups are independent of job ordering (Morton and Pentico 1993). The challenge is greatly magnified by the sequence-dependent setup constraints. Sen and Bagchi discuss the challenge of sequence-dependent setups (Sen and Bagchi 1996). Specifically, they discuss how they induce a non-order-preserving property of the evaluation function. At the time of their writing, exact solution procedures such as A*, Branch-and-Bound, and their own GREC (Sen and Bagchi 1996) for sequencing problems with sequence-dependent setups were limited to instances with no more than approximately 25-30 jobs, even for easier objective functions. Problem instances of larger size require inexact solution procedures such as metaheuristics.

Applying Our Approach to the Problem

The first step in applying our GA with the HeurX operator to a problem is the identification of an appropriate dispatch scheduling policy. There are numerous dispatch policies available for the setup-free version of the weighted tardiness problem (Morton and Pentico 1993). There are far fewer available for the variation that we consider consisting of sequence-dependent setups. One of the strongest dispatch policies available when sequence-dependent constraints are involved is the *Apparent Tardiness Cost with Setups (ATCS)* heuristic (Lee, Bhaskaran, and Pinedo 1997), defined as:

$$\text{ATCS}_j(t,l) = \frac{w_j}{p_j}\exp\left(-\frac{\max(d_j - p_j - t, 0)}{k_1\bar{p}} - \frac{s_{l,j}}{k_2\bar{s}}\right), \tag{3}$$

where t is the current time (or the sum of the process and setup times of jobs already sequenced); l is the index of the job most recently added to the schedule; $\bar{p}$ is average processing time of all jobs; and $\bar{s}$ is average setup time.

The k_1 and k_2 are parameters that are automatically tuned based on characteristics of the problem instance as follows.

$$k_1 = \begin{cases} 4.5 + R & \text{if } R \leq 0.5 \\ 6.0 - 2R & \text{otherwise} \end{cases}, \tag{4}$$

where R is the duedate range factor which is an indicator of how spread the duedates of the jobs are, and is defined as: $R = \frac{d_{\max} - d_{\min}}{C_{\max}}$ where $d_{\max}$, $d_{\min}$ are the maximum and minimum duedates, and $C_{\max}$ is an estimate of the makespan (or completion time of the last job).

$$k_2 = \frac{\tau}{2\sqrt{\eta}}. \tag{5}$$

where τ is the duedate tightness factor, defined by $\tau = 1 - \frac{\bar{d}}{C_{\max}}$; and η is the setup time severity factor, defined by $\eta = \frac{\bar{s}}{\bar{p}}$. The τ is an indicator of how urgent the duedates of the problem are and η is an indicator of how large an effect the setup times can have on problem solving.

In their publication of the ATCS heuristic, Lee et al describe a simple local search that they recommend applying to the solution given by ATCS (Lee, Bhaskaran, and Pinedo 1997). In our experimental results we consider two alternatives: using Lee et al's local search as the optional local search step of the algorithm of Figure 3, and not using that optional local search.

Lee et al's local search works as follows. Find the job that contributes the most to the cost of the solution. If removing that job and reinserting it into an earlier position in the sequence decreases the weighted tardiness of the solution, then perform the removal of that job and reinsertion into the point that decreases the cost the greatest. If there is no reinsertion point for that job that would decrease the cost of the solution, then the local search is done. Otherwise, repeat.

Tuning the Control Parameters

Before applying the HeurX operator and our Steady-State GA to a new problem, we must determine values for the control parameters. We must determine the number of parents M to use and the size S of the population for the GA. There is a trade-off with choosing values for each of these. The computational cost of the HeurX operator grows linearly with M. If we double the number of parents M, then we limit ourselves to the ability to execute half as many iterations of the GA. Additionally, the higher the value of M, the more confidence we place on the knowledge encoded in the dispatch heuristic. The lower the value of M, the closer the algorithm gets to a random uninformed search.

The cost associated with increasing the size S of the population is largely negligible (compared to increasing M). Given our steady-state model, we are not iterating over the entire population as a generational GA does. The removal of the worst member of the population and insertion of the newly formed child is inexpensive given our use of a heap—logarithmic in S. In our preliminary parameter tuning experiments, we considered population sizes into the 100,000s with little effect on runtimes. However, large S does have a time cost at the start of the GA. Before beginning the iterations of the GA, the initial random population must be generated. This operation scales linearly with S. For longer runs of the GA, this may be considered a minor point; however, for short runs, this $\bigcirc(S)$ time initialization of the population may necessitate the use of a smaller population.

With these tradeoffs in mind, we set out to find a balance. We generated 36 random problem instances with 60 jobs each using the structural properties suggested by Lee et al (Lee, Bhaskaran, and Pinedo 1997). This small set of problem instances were only used for control parameter tuning, and were not in any way used for our comparisons with other approaches. Our parameter tuning method was mostly automated and considered values for the number of parents M in the range $[2, 100]$ and values for the size S of the population in the range $[100, 100000]$. We focused our tuning of the algorithm without the optional local search step.

During control parameter tuning, we discovered that the choice of control parameters also depends on how long of a search is planned. For example, longer single runs will stagnate if the population size is not sufficiently large. However, if the population size is too large, then shorter runs are unable to effectively focus the search in promising areas. We chose to tune the algorithm for short run lengths of approximately 2 seconds in length (on a Dell Dimension 8400 with a 3.0 GHz Pentium 4 CPU and 1GB memory). We implemented the HeurX operator and the steady state GA in Java 1.6. If additional computing time is available, we restart the GA with a new random population.

The result of the control parameter tuning is as follows: population size $S = 14000$, number of parents $M = 59$, and approximately 1500 iterations before restarting a longer run. Decreasing M allows us to execute more iterations in the same length of time, but our tuning procedure showed lower values of M to ineffectively utilize the heuristic's guidance. To effectively integrate the knowledge inherent in the dispatch policy, M cannot be set too low.

Table 1: Local search step vs no local search for short runs of the GA with HeurX. $\%\Delta B$ is average percentage deviation from the best known solutions to the benchmarks. 95% confidence intervals and T-Test results are shown.

	With Local	Without Local	P-value
$\%\Delta B$	0.255 ± 0.034	0.488 ± 0.079	$< 1 * 10^{-18}$
Time	2.05 seconds	2.01 seconds	

Experimental Results

We begin by comparing a couple variations of our approach: with and without using the optional local search step, and restarting a short run vs executing a single longer run. Then, we compare to a generational-style GA using the permutation crossover operator NWOX, an algorithm that had found several best known solutions to benchmarks.

In comparing our approach to others, we use a set of benchmark instances (Cicirello and Smith 2005; Cicirello 2007). This set of benchmarks have been used by several for a wide range of approaches, including GAs, simulated annealers, ACO, truncated branch-and-bound, among others. The set consists of 120 problem instances, 10 instances from each of the 12 categories of Lee et al (Lee, Bhaskaran, and Pinedo 1997). The benchmark set includes data on best known solutions found by any approach so far.

The Benefits of the Optional Local Search Step: We first report on the potential benefit from applying the optional local search step described earlier. We begin by comparing:

- The steady-state GA with HeurX operator without the optional local search step.
- The steady-state GA with HeurX operator using the simple (and fast) local search suggested by Lee et al in their original presentation of the ATCS dispatch policy. No additional tuning was performed for the local search version.

For each of these variations, we solve each of the benchmark problem instances 10 times providing us with 1200 data points for each version. The addition of the simple local search step results in a marginal increase in CPU requirements (from 2.01 seconds on average to 2.05 seconds). With the rather restricted move set, this local search hits a local optima in a relatively few number of operations.

Table 1 shows the results. $\%\Delta B$ is the average percentage deviation from the best known solutions to the benchmark instances. The local search step produces an algorithm that deviates from the best known solutions by approximately 25.5% on average compared to nearly 49% if we do not use the optional local search step. This result is extremely statistically significant (P-value $< 1 * 10^{-18}$).

Restarting a Short Run vs Executing a Single Longer Run: Next we consider the alternatives of restarting a short run vs executing a single longer run. Both versions use the optional local search step. We compare:

- Steady-state GA with HeurX operator and local search step. Restart with new random population every 1500 iterations. We use 11 restarts to produce a run that requires 20.5 seconds of CPU time on average on our test machine.
- Steady-state GA with HeurX operator and local search, but retuned for a longer run length. We repeated the tuning phase described earlier but for runs of approximately 20 seconds in length. The result is population size $S = 100000$, number of parents $M = 44$, and 21000 iterations to reach the desired run length on our test machine. This results in a 20.6 second run on average.

Table 2: Restarting a short search vs running a single longer search. $\%\Delta B$ is average percentage deviation from the best known solutions to the benchmarks. 95% confidence intervals are shown as well as the result of a T-Test.

	11 Short Restarts	1 Long Search	P-value
$\%\Delta B$	0.182 ± 0.023	0.193 ± 0.024	0.0018
Time	20.5 seconds	20.6 seconds	

Table 3: Comparing the HeurX-GA with the NWOX-GA for different length runs. $\%\Delta B$ is average percentage deviation from the best known solutions to the benchmarks. 95% confidence intervals are shown as well as the result of a T-Test.

	HeurX-GA (1 short run)	NWOX-GA (5000 gens)	P-value
$\%\Delta B$	0.255 ± 0.034	0.291 ± 0.069	0.11
Time	2.05 seconds	2.389 seconds	

	HeurX-GA (2 restarts)	NWOX-GA (10000 gens)	P-value
$\%\Delta B$	0.209 ± 0.026	0.250 ± 0.059	0.04
Time	3.90 seconds	4.762 seconds	

Table 2 shows the results. Restarting a short search and taking the best solution across restarts is found to outperform a single longer search, deviating from the best known solutions to the benchmarks by 18.2% on average as compared to 19.3% for the single longer run (P-value of 0.0018).

Comparison to a Generational GA: We now compare our approach to one of the current best performing metaheuristics for the problem. We compare:

- HeurX-GA: Our steady-state GA using the HeurX operator and optional local search step. Every 1500 iterations, we restart with a new random population. We consider different length runs in terms of number of restarts.
- NWOX-GA: A genetic algorithm (generational model) using a permutation representation and the permutation crossover known as Non-Wrapping Order Crossover (NWOX) (Cicirello 2006). This GA previously had found several of the best known solutions to the benchmarks.

Table 3 shows a comparison of our GA with the HeurX operator to NWOX-GA for two different length runs.[1] Timing was done on the same computer architecture for both algorithms. We compare a single short run of our HeurX-GA to a 5000 generation run of NWOX-GA (slightly longer in

[1]The $\%\Delta B$ for NWOX-GA have been updated to reflect improvements to best knowns since its publication (Cicirello 2006).

Table 4: New Best Known Solutions Found by HeurX-GA for Benchmark Instances. Column P is the problem instance number.

P	Old	New	P	Old	New
8	298	201	97	418995	418753
11	5088	5037	99	374607	372786
13	6147	6092	102	495094	494019
17	387	271	111	348796	347882
19	239	60	120	399700	399383

CPU time than the run of HeurX-GA). HeurX-GA deviates on average from the best knowns by 25.5% as compared to approximately 29% for NWOX-GA. Although an improvement, this result is not statistically significant (P-value 0.11).

However, if we increase the run length, comparing 2 restarts of HeurX-GA (3.9 CPU seconds) to a 10000 generation run of NWOX-GA (4.8 CPU seconds), we see that HeurX-GA begins outperforming the GA with problem-independent operators despite using less compute time (significant with P-value of 0.04). HeurX-GA deviates from the best known solutions by around 21% for this run length as compared to 25% for NWOX-GA.

Conclusions

In this paper, we demonstrated the potential benefits of integrating problem specific knowledge in the form of a dispatch scheduling policy into a genetic algorithm. Specifically, we presented a novel crossover operator, Heuristic Sequencing Crossover, designed to enable this integration of dispatch scheduling with a GA. A steady-state GA using HeurX was presented and we validated its effectiveness on an NP-Hard scheduling problem. Our approach applied to a sequence-dependent setup version of the weighted tardiness scheduling problem is competitive with some of the current best metaheuristics for the problem. In particular, we showed there to be an advantage over problem-independent metaheuristics for shorter length searches. This is perhaps due to the dispatch policy helping to focus the search early on a promising region, rather than requiring the search to first narrow in on that region.

During the course of this study, the HeurX-GA was able to improve upon the best known solutions to 10 of the benchmark instances. Table 4 provides the objective values for the new best known solutions for those instances as well as the old best known for comparison purposes.

References

Bresina, J. L. 1996. Heuristic-biased stochastic sampling. In *Proc of the 13th Nat Conf on Artificial Intelligence*, 271–278. AAAI Press.

Cicirello, V. A., and Smith, S. F. 2005. Enhancing stochastic search performance by value-biased randomization of heuristics. *Journal of Heuristics* 11(1):5–34.

Cicirello, V. A. 2006. Non-wrapping order crossover: An order preserving crossover operator that respects absolute position. In *Proc of the Genetic and Evolutionary Computation Conf*, 1125–1131. ACM Press.

Cicirello, V. A. 2007. The challenge of sequence-dependent setups. In *Proc of the ICAPS Workshop on Scheduling a Scheduling Competition*. AAAI Press.

Cormen, T. H.; Leiserson, C. E.; and Rivest, R. L. 1990. *Introduction to Algorithms*. McGraw-Hill.

Davis, L. 1985. Applying adaptive algorithms to epistatic domains. In *Proc of the IJCAI*, 162–164.

Dorigo, M., and Di Caro, G. 1999. The ant colony optimization meta-heuristic. In *New Ideas in Optimization*. McGraw-Hill. 11–32.

Franck, B., and Neumann, K. 1996. Priority-rule methods for the resource-constrained project scheduling problem with minimal and maximal time lags – an empirical analysis. In *The 5th Int Workshop on Project Management and Scheduling*, 88–91.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*.

Goldberg, D. E., and Lingle, R. 1985. Alleles, loci, and the traveling salesman problem. In *Proc of the 1st Int Conf on Genetic Algorithms*, 154–159.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley.

Ho, S.-Y., and Chen, J.-H. 2000. A ga-based systematic reasoning approach for solving traveling salesman problems using an orthogonal array crossover. In *HPC '00: Proc of the The 4th Int Conf on High-Performance Computing in the Asia-Pacific Region*, 659–663.

Lee, Y. H.; Bhaskaran, K.; and Pinedo, M. 1997. A heuristic to minimize the total weighted tardiness with sequence-dependent setups. *IIE Transactions* 29:45–52.

Moraglio, A., and Poli, R. 2005. Topological crossover for the permutation representation. In *Workshop Program of the Genetic and Evolutionary Computation Conference*, 332–338. ACM Press.

Morton, T. E., and Pentico, D. W. 1993. *Heuristic Scheduling Systems: With Applications to Production Systems and Project Management*. John Wiley and Sons.

Oliver, I. M.; Smith, D. J.; and Holland, J. R. C. 1987. A study of permutation crossover operators on the traveling salesman problem. In *Proc of the 2nd Int Conf on Genetic Algorithms*, 224–230. Lawrence Erlbaum Associates, Inc.

Sen, A. K., and Bagchi, A. 1996. Graph search methods for non-order-preserving evaluation functions: Applications to job sequencing problems. *Artificial Intelligence* 86:43–73.

Starkweather, T.; McDaniel, S.; Mathias, K.; Whitley, C.; and Whitley, D. 1991. A comparative study of genetic sequencing operators. In *Proc of the 4th Int Conf on Genetic Algorithms*, 69–76.

Watson, J.-P.; Ross, C.; Eisele, V.; Denton, J.; Bins, J.; Guerra, C.; Whitley, L. D.; and Howe, A. E. 1998. The traveling salesrep problem, edge assembly crossover, and 2-opt. In *Proc of the 5th Int Conf on Parallel Problem Solving from Nature*, 823–834. Springer-Verlag.