Greedy Transformation of Evolutionary Algorithm Search Spaces for Scheduling Problems

David Joslin, *Member, IEEE*, and Justin Collins

***Abstract*— Many scheduling algorithms search the space of possible solutions (schedules), but some instead search the space of permutations of the set of jobs, employing a greedy algorithm to map any such permutation to a schedule that can be evaluated by the fitness function. The search algorithm is thus simplified because knowledge about problem domain details is encapsulated in the greedy algorithm that constructs schedules, and the fitness function that evaluates them. The variety of types of algorithms for which this sort of "greedy transformation" has proven effective, and the range of successful applications, prompts us to look more closely at how such transformations may also make good solutions easier to find. In this paper we experimentally evaluate some characteristics of search spaces under greedy transformations as a first step toward understanding why this technique is effective.**

I. Introduction

Part of the "art" involved in applying evolutionary algorithms to new problem domains is in designing a problem representation that will facilitate the search for good solutions. It is well known that the choice of representation can dramatically affect the characteristics of the search space. One example would be the choice between *phenotypic* representation and *genotypic* representation, as discussed in [9], which can make a significant difference in the effect of small mutations. In this paper we look at the use of greedy algorithms as a technique for transforming search spaces. Although we focus on two scheduling domains, similar approaches apply to other types of problems as well.

When we talk about a greedy algorithm "transforming" a search space for a scheduling problem we mean that the the greedy algorithm is used to construct solutions, so that the search algorithm component of the system searches in the space of job permutations that drive the greedy algorithm, rather than in the space of possible solutions to the original scheduling problem. We observe that several effective algorithms have this sort of greedy transformation in common, although they differ substantially in the manner in which they search the transformed space. This raises the question of the extent to which the success of these algorithms is due to the greedy transformation itself. To explore this question we hope to take some first steps toward understanding how the search characteristics of search spaces under greedy transformation compare to those of the original space of problem solutions.

David Joslin is with the Computer Science and Software Engineering Dept., Seattle University, 900 Broadway, Seattle, WA 98122. (Email: joslind@seattleu.edu).

Justin Collins is with the Computer Science Dept., UCLA, 4732 Boelter Hall, Los Angeles , CA 90095 (Email: justincollins@ucla.edu)

We stress that in this paper we are not attempting to introduce a new scheduling algorithm. Our starting point is the fact that algorithms such as Optiflex [18] and Squeaky Wheel Optimization (SWO) [14], both discussed below, have already demonstrated that algorithms using greedy transformations can be highly effective on real-world scheduling problems. However, little has been done in that work to explain *why* those algorithms are effective. We believe that a better understanding of why those algorithms are effective could be very beneficial to the design of new algorithms. The problem domains we consider here are intentionally simple, in order to make a comparison of search space characteristics before and after the greedy transformation more accessible.

The next two sections outlines some characteristics of greedy transformations and hypotheses about why such transformations and searching in the transformed space may be effective, and summarize several successful scheduling algorithms that use greedy transformations. We then consider two sets of experiments. The first uses a single-machine weighted tardiness scheduling domain to try to characterize the extent to which a simple greedy transform has desirable effects on the search space. The second set of experiments, on a course scheduling problem, look at the question of whether adding more "intelligence" to a greedy transformation helps or hurts a genetic algorithm. The final section discusses at these results and discusses some tradeoffs involved in applying greedy transformations.

II. Greedy transformation overview

A greedy algorithm for a scheduling problem may be driven by a list of the jobs in some order, with each job in turn being scheduled in a "greedy" fashion taking whatever resources it needs. A greedy algorithm thus defines a mapping from a permutation of a set of jobs to a complete schedule. The ideal case for a greedy algorithm is one in which an ordering of those jobs can be found such that the greedy algorithm is guaranteed to find an optimal solution, but in practice only some very simple problem domains make this feasible.

Even when no ordering of the jobs is guaranteed to produce an optimal solution, the greedy algorithm defines a mapping from the space of permutations of a set of jobs to the space of solutions (schedules), and thus makes it possible for an algorithm to search the space of permutations of jobs, rather than in solution space. A point in that space of permutations can be evaluated by applying the greedy algorithm to generate a solution, then applying the fitness function to evaluate that solution. This often turns out to make algorithm

1-4244-1340-0/07$25.00

implementation fairly straightforward, because the details of the problem domain can be encapsulated in the greedy algorithm that constructs schedules and the fitness function that evaluates them.

We can also think of the space of permutations of jobs as a "priority space," because the permutation determines the order in which the greedy mapping places the jobs into the schedule. If we are minimizing lateness, for example, then the first job in the input sequence is guaranteed to be scheduled on time if at all possible, even if that turns out to be at the expense of other jobs, so in this sense it receives the highest possible priority.

Intuitively, the greedy transformation can in some cases make the search for good solutions easier. Consider a single-machine scheduling problem that includes some jobs with deadlines far in the future, and others with very near-term deadlines, in a domain that requires us to minimize lateness. Solutions that put the far-deadline jobs early in the schedule may tend to be of low quality because those jobs may displace others that need to be early in the schedule to avoid lateness. Consider instead the effect of having far-deadline jobs early in the input sequence when we are searching in priority space. If a job with a deadline far in the future appears early in the input sequence, the greedy algorithm can schedule it without lateness and still give it a late starting time. Other jobs will then be able to fill in earlier slots in the schedule.

In this way the greedy transformation creates a search space that is more "forgiving." It also creates a many-to-one mapping from points in priority space to points in solution space, and this many-to-one mapping will sometimes tend to make low-quality regions of the solution space unreachable, while making higher-quality regions of the solution space reachable from multiple points in priority space. But this is not guaranteed, and it is possible that a greedy algorithm might also make the *best* points in the solution space unreachable, or might make the best points reachable in a theoretical sense but make it difficult for search algorithms to find those points by iterative improvement. In short, it is not immediately obvious that greedy transformations will be helpful when combined with algorithms that attempt to search in priority space.

III. Related work

A number of successful algorithms have been based on this approach, leading us to conclude that further investigation is merited. One example is Optiflex, a commercial application based in part on work by Syswerda [18]. In this approach the chromosome is a sequence of jobs. A deterministic "schedule builder" takes a sequence of jobs and attempts to construct a valid schedule, respecting the hard constraints of the problem. A Genetic Algorithm (GA) is used to search the space of chromosomes.

In the terms we introduce above, Optiflex is searching entirely in priority space. A chromosome defines a point in priority space, and mutation and crossover operations are operations within priority space. The fitness of a permutation is evaluated by applying the greedy schedule construction algorithm and then applying the original domain fitness function to the resulting schedule, but as far as the GA itself is concerned these together comprise the fitness function and the GA never needs to know about the schedule.

Another technique for using a greedy transformation is found in Squeaky Wheel Optimization (SWO) [14]. Like Syswerda's approach, SWO uses a greedy algorithm that is given a sequence of jobs and constructs a schedule. Rather than using a GA to search the space of job sequences, SWO uses only a "directed mutation" or "genetic engineering" approach. The solution produced by the constructor is analyzed to find flaws, and changes are made to the job sequence in a way that tends to help avoid those flaws in the next iteration. For example, a job that is completed after its due date may be moved earlier in the input sequence, so that the greedy constructor is more likely to schedule that job successfully on the next iteration, possibly at the expense of other jobs. The standard SWO approach repeatedly modifies a single job sequence in this fashion, remembering the best solution found. The "directed mutation" in SWO can be viewed as modifying a priority-space permutation according to flaws identified in the corresponding point in solution space.

Although SWO is not a traditional evolutionary algorithm, it has some elements in common with them and hybrid approaches that combine ideas from SWO and Genetic Algorithms are possible [19]. It been successfully applied to a wide variety of domains, including factory scheduling and graph coloring [14], satellite downlink scheduling [1], satellite observation scheduling [13], project scheduling [17], and scheduling of airborne astronomical observations [12].

GRASP (Greedy Randomized Adaptive Search Procedures) [10] is another approach that incorporates a greedy algorithm, although in this case the greedy algorithm is not driven by a permutation of the problem elements, but instead selects them from a set of candidates ordered by a greedy function, with some randomness imposed on the selection according to a tunable parameter. Local search is used to try to improve upon the resulting solution. GRASP has been successfully used in a very wide variety of applications [11], and although GRASP is not an evolutionary algorithm, it is worth noting as another example of an application of a greedy algorithm to optimization problems. The local search that is performed by GRASP is a solution-space search, and the steps that generate the solution that serves as a starting point for the local search are performed in something similar to the priority space of SWO and Optiflex, but dynamically rather than statically determined.

In this paper we focus on trying to understand the foundation that Optiflex and SWO (and to a lesser extent, GRASP) have in common, namely that they operate at least partially in a transformed search space created by a greedy algorithm that maps from permutations to solutions. Our goal here is not to develop better algorithms based on this approach, but to use simple examples to begin characterizing the nature of this search space transformation. We hope that a better

understanding of this transformation will in turn lead to more effective applications of this technique.

IV. Single-machine weighted tardiness domain

We begin with a set of experiments in the *single-machine weighted tardiness* (SMWT) domain [6], [7], [15] using data from the OR-Library [3], [4]. As defined in [3], an SMWT problem has jobs $j_1, j_2, ..., j_n$ to be processed on a single machine starting at time $t = 0$. Each job j requires processing time $p(j)$, and has a due date $d(j)$ and positive weight $w(j)$. For job j completed at time $C(j)$ the tardiness $T(j)$ is $C(j) - d(j)$ if $C(j) > d(j)$, and otherwise zero. The objective is to minimize the total weighted tardiness, $\sum_{j=1}^{n} w(j)T(j)$. The experiments described below all use the largest data sets from [3], giving us 125 problem instances with 100 jobs each.

A. Greedy transformation

This is a convenient problem to start with because a permutation of the jobs can be mapped very directly to a solution, with the jobs scheduled sequentially with no gaps between them. In this problem domain no schedule with gaps between the jobs can have better quality (lower weighted tardiness) than a schedule with no such gaps. Thus we can define a simple mapping from permutations to schedules that does not make use of a greedy schedule constructor to use as a baseline for comparisons. We call this the "linear" mapping.

Many greedy algorithms would be possible for this or any other domain. As used by approaches such as GRASP, SWO and Optiflex, we need a greedy algorithm that is given a sequence of jobs and constructs a schedule by adding each job in the order given. The algorithm will be "greedy" in the sense that placement of each job should be the best possible for that job with respect to the objective function (minimizing tardiness for that job), but no greedier.

We define a simple greedy algorithm as follows. Given a sequence of jobs, we add each job to the schedule without changing the assigned start time for any job already in the schedule. In other words, the new job can only be scheduled within an existing gap in a partial schedule. If it is possible to schedule a job so that it finished prior to its due date, then pick the latest possible start time, thus minimizing the impact this job has on any jobs that are yet to be scheduled. If, however, it is impossible for the job to be scheduled on time, then schedule it at the earliest feasible start time to minimize the penalty for tardiness.

The resulting schedule may have gaps that reduce the quality of the solution, but this is easily resolved by performing a "left shift" operation (as used in the Doubleback algorithm [8]) that shifts all jobs as early as possible, preserving the order. This also gives us the same schedule that the linear mapping would have produced, given the ordering of jobs in the final schedule. Some of the experiments below rely on this comparison.

B. Experiment: random sampling

For this first experiment we looked at random points in the space of permutations under both mappings, greedy and linear. We ran 100 trials for each of the 125 problem instances. Taking the mean score over 100 random points in the input space for each mapping we found that the greedy mapping outperformed the linear mapping for all 125 problem instances. Moreover, the *worst* solution found by the greedy mapping was better than the mean of the 100 trials for the linear mapping 64.0% of the time, and the worst solution found by the greedy mapping was better than the best solution found by the linear mapping 37.6% of the time.

Not surprisingly, random points in the input space for the greedy mapping yield significantly better solutions than random points under the linear mapping. If we consider the linear mapping to be a one-to-one mapping from permutations to schedules (the alternative being to consider the solution space to include schedules with unhelpful gaps between jobs), then the greedy mapping creates a search space that has a significantly greater density of good solutions. This fits with the idea discussed above that the greedy mapping is more "forgiving" of permutations that don't fully reflect the fact that it may be much more important to give an early start time to some jobs than others.

C. Experiment: simple heuristics

Increasing the density of good solutions is a positive step, but we also need to ensure that a greedy transformation preserves the sort of search space structure that search algorithms need to exploit. As a first step in exploring this question we look at several job ordering heuristics. The point in these experiments is not to find the best heuristics for use with a greedy transform. The point is to verify that heuristic-driven search remains effective under the greedy transformation, as an indication that the search-space structure remains well behaved under that transformation.

We began by generating input sequences using two simple heuristics: the *weight* heuristic sorts the jobs in decreasing order by weight, and the *due date* heuristic sorts them in increasing order by due date. Because a stable sorting algorithm will preserve the order in case of ties, and because there are sometimes duplicated values in the problem instance definitions, we generated 100 random permutations of the set of jobs, then applied the sorting heuristics to the randomized sequences.

With the weight heuristic, the greedy mapping resulted in a better solution than the linear mapping 99.4% of the time, with the remaining fraction of a percent being a tie between the two mappings. The due date heuristic was less effective with the greedy mapping, but greedy still outperformed linear 35.2% of the time, with ties 56.0% of the time, and the linear mapping superior to the greedy mapping only 8.8% of the time.

We might expect that the weight heuristic would be more naturally suited to the greedy mapping because it allows the highest-weight jobs to be scheduled without tardiness, with

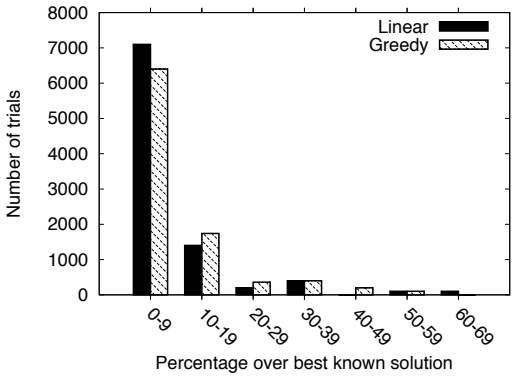

Fig. 1. WMDD heuristic rule

lower-weight jobs filling in gaps in the schedule as much as possible. So it is not surprising that for the greedy mapping the weight heuristic outperformed the due date heuristic 75.4% of the time. But the relative success of the greedy mapping with both heuristics also gives some support for the idea that the greedy mapping preserves useful structure in the search space, of a kind that optimization algorithms can exploit.

D. Experiment: WMDD heuristic

One heuristic reported to be effective on weighted tardiness scheduling problems is the "Weighted Modified Due Date" (WMDD) heuristic [15]. The WMDD for job i is defined to be $\mathrm{WMDD}_i = \max(p_i, d_i - t)/w_i$ where w_i is the weight for job i, p_i is the processing time, d_i is the due date, and t is the current time. The jobs are processed in non-decreasing order by WMDD, but unlike the two simple heuristics described above, WMDD is used as a *dispatch heuristic*. The next job to process is selected by recalculating the WMDD for each job, hence the inclusion of t as the current time in the formula.

To apply this rule under a greedy transformation we first need to decide what the "current time" parameter, t, should mean. If we define it to be the completion time of the last job in the current partial schedule then we don't take into account that the schedule may contain very large gaps. On the other hand the completion time of the job most recently added to the schedule makes even less sense, because that time jumps around as jobs are added. We chose to define the t parameter to be the sum of the processing times of the jobs already in the schedule. This is simple and sufficient for our purposes here.

We again ran 100 trials for each problem starting from a random permutation of the jobs, to avoid bias due to the original ordering of the jobs in the case of ties. Figure 1 shows a histogram of the greedy and linear scores as the percentages by which they exceeded the best known solution, omitting instances for which a solution with zero tardiness was known. Linear has more solutions within 10% of the known best than greedy, but the worst solutions for greedy are not as bad as the worst solutions for linear, and in general both greedy and linear demonstrate comparable performance on these problem instances.

On 18 of the 125 problem instances a solution with zero tardiness was known, and on these greedy and linear both found an optimal solution in 13 of the 18 instances. In two cases greedy found an optimal solution and linear did not, in one case linear found an optimal solution and greedy did not, and in the remaining two cases neither one did.

The WMDD heuristic was designed specifically for building a schedule in linear order, so there's no reason to expect that it would work well under a greedy mapping. The fact that the WMDD heuristic, adapted as described above, continues to give fairly good results under the greedy mapping suggests that the greedy mapping preserves a lot of the structure of the search space that WMDD was designed to exploit.

E. Experiment: single-inversion neighborhood

Search algorithms based on iterative improvement depend on the principle that good solutions will often be in the same "neighborhood" as other good and possibly better solutions. The meaning of "neighborhood" depends on the algorithm. We begin by looking at neighborhoods defined by a single inversion, i.e., swapping the order of two adjacent jobs, such as might be used in a simple local search algorithm or a simple mutation operator for a genetic algorithm.

For each of the 125 problem instances we ran 25 trials, trying every possible neighboring point that differed only in a single inversion. These problems all have 100 jobs, so there are 99 such neighbors. The linear mapping almost always produced more neighbors that were improvements over the starting point, 82.2% of the time compared to 2.3% of the time for greedy. Although the linear mapping consistently had more "good" neighbors than the greedy mapping, the difference was usually fairly small; 53.8% of the cases differed by ten percent or less.

The greedy mapping tended to produce fewer neighbors that were improvements, but the ones that it did produce tended to be of better quality. Of the neighbors explored for both mappings the greedy mapping found the best-quality neighbor 62.0% of the time, and the linear mapping 11.1% of the time, with the remainder being ties.

F. Experiment: larger neighborhoods

As mentioned above, the meaning of "neighborhood" depends on the algorithm being employed. A simple local search algorithm might perhaps use single adjacent swaps, as in the previous experiment, but other local moves are possible as well. A genetic algorithm may employ a similar technique to implement mutation, but for crossover we cannot define a "neighborhood" in terms of a single point in the solution space, because the crossover operation involves two (or more) individuals, although as populations converge the individuals being combined will be increasingly similar. For an algorithm like SWO the concept of "neighborhood"

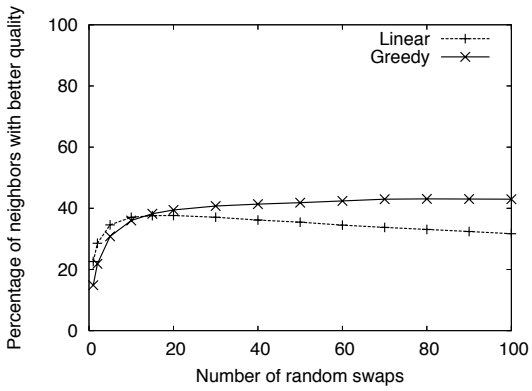

Fig. 2. Multiple-inversion neighborhoods

applies even less, because on each iteration the previous results is discarded and a new, and often very different, solution is constructed.

Nevertheless, many search algorithms rely on being able to find increasingly better solutions as they progress through the search space. They rely, in other words, on the idea that if you find a good solution it is likely to be in the vicinity of an even better solution. We therefore sought in this experiment to look at a more general notion of "neighborhood" under both mappings. Rather than allowing just one inversion, we performed multiple random inversions on a permutation in order to generate a neighbor. The greater the number of inversions, the larger the neighborhood being explored.

We varied the number of swaps from 10 to 100 in steps of 10, plus some additional values (1, 2, 5 and 15) to help fill in detail in the graph. At each of these levels we ran 100 trials for each of the 125 problems. For each trial, a random input sequence was generated, and the greedy mapping was used to generate a solution. The sequence of jobs in that solution defines an input sequence that the linear algorithm maps to the same solution. I.e., we have one randomly-generated sequence that takes the greedy mapping to a solution, and another that takes the linear mapping to exactly the same point in the solution space. This was done to avoid the bias demonstrated in the random sampling experiment.

For each trial, we then generated 100 samples from the greedy mapping input sequence by performing the desired number of adjacent swaps on that sequence. The quality of the solutions resulting from those neighboring sequences was compared to that of the starting sequence. The same was done for 100 samples generated from the linear mapping input sequence. The goal here was to look at points that are "near" the input sequences for both mappings, with both starting from the same point in the solution space, and therefore with both input sequences corresponding to the same solution quality when mapped by their respective algorithms.

Figure 2 shows the results, with the number of inversions on the x-axis defining the size of the neighborhood, and the y-axis showing percentage of samples from that neighborhood that are better than the starting point. As the graph shows, the linear mapping initially shows neighborhoods with a slightly higher percentage of better neighbors, but as the neighborhoods get larger the advantage of the greedy mapping becomes greater, with the crossover point at roughly fifteen random swaps.

Out of the total number of 62,500 trials for neighborhoods of up to 15 swaps, greedy found the best neighbor 67.7% of the time, linear 16.9% of the time, and they were tied 15.4% of the time. Additional data for neighborhoods allowing up to 500 swaps showed the percentage of better neighbors continuing to hold steady at around 43%. Under the linear mapping, the percentage of better neighbors continued to decline, down to around 21%.

These results suggest that the greedy mapping creates a search space that, in addition to having a greater density of good solutions, has the sort of structure required by search algorithms that rely on iterative improvement.

V. Course scheduling domain

The previous set of experiments looked at a simple greedy mapping *vs.* a linear mapping, and found evidence that under the greedy mapping the search space has characteristics conducive to effective search. But as already noted, a wide range of greedy algorithms could be defined for a domain, ranging from simple algorithms to those that attempt to be more "intelligent." In this section we explore a set of greedy construction algorithms for a university course scheduling domain, reviewing and extending work presented in [5].

This domain represents the problem of college course scheduling from the point of view of a student planning how they might satisfy a set of degree requirements. For the first problem described here the degree requirements were for a BS in Computer Science with a Math minor, based on actual degree requirements and course schedules.

The hard constraints for this problem were: (a) the student must satisfy all degree requirements, including all required courses and elective requirements, (b) a course may only be taken during a quarter in which it is offered, and (c) a maximum of 18 credits may be taken per quarter.

The soft constraints define student preferences. We tried to define a set of realistic preferences as follows: (a) the schedule should require as few quarters as possible over the expected four years (twelve academic quarters), (b) for the sake of "balance" we want to minimize the number of quarters with more than two CS, math, or physics courses, and (c) we want to minimize the number of classes in the schedule that are not necessary for graduation.

The soft constraints are treated hierarchically, so that, for example, any schedule that is within the desired threshold length is preferable to one that violates the length constraint in order to achieve more balanced quarters. The preference against extraneous classes could be given a higher priority, but the other soft constraints already tend to minimize extraneous classes so this arrangement seemed to work out well.

We defined five distinct greedy algorithms, designated here *C1* to *C5*, that take a chromosome consisting of a permutation of courses as input, and construct a candidate course schedule. Constructor *C1*, the simplest case, iterates over the courses in the chromosome scheduling each course in the first non-full academic period in which the course is offered. This constructor, as well as all of the others, halts as soon as all degree requirements have been met, so that extraneous courses at the end of the chromosome are ignored.

Constructor *C2* also schedules courses in the first non-full period they are offered, but first checks for prerequisites. If the prerequisites are not yet met for a particular course, the scheduling of that course is deferred until its prerequisite courses have been scheduled. *C3* checks for prerequisites, but rather than deferring the scheduling of the course for which the prerequisites have not been met it recursively schedules prerequisites until the course can be scheduled.

Constructor *C4* is like *C3*, with the additional idea of filtering out courses that do not directly satisfy a degree requirement and are not a prerequisite for another course we are trying to schedule. Note that a course may not be needed at the point where it is encountered in the chromosome, in which case *C4* will initially ignore it, but it may be added later as the prerequisite for some other course. With *C1* and *C2* omitting such a class would mean that if a later class that needs it as a prerequisite is encountered, the schedule would end up violating the prerequisite hard constraint.

Constructor *C5* also applies recursive prerequisite scheduling, and ignores courses that do not contribute to the degree. When scheduling a course (either because it was next in the chromosome or when recursively handling prerequisites) the constructor begins at the first quarter at which all the prerequisites would be satisfied, then scans future periods checking what penalties it would receive for each period. It then schedules the course in the period with the lowest penalty. Note that although *C4* and *C5* attempt to filter out useless courses, they do not guarantee zero extraneous courses due to dependencies between optional courses.

This search for the "earliest-best" point at which to schedule a class was implemented by providing the constructor with a list of functions that can be applied to a partial schedule, with one function per soft constraint. These are all black-box functions to the constructor. They do nothing but return a penalty value, and *C5* does not need to know what constraints they represent. The intention was to move toward an architecture that is easily adaptable and more extensible than the other constructors. Below we describe one experiment that uses significantly different soft constraints, with encouraging results for *C5*.

A. Experiment: constructor comparison

Each constructor was run for 10,000 iterations using a population size of 100, maximum gene length of 100, and mutation rate of 1%. The genetic algorithm used is steady-state with tournament selection and single-point random crossover. The initial pool is random. The fitness of a chromosome is evaluated by first passing it to a constructor which produces a schedule; the schedule is then evaluated for validity, length, balance, and extraneous courses, the results of which form a ranked tuple indicating the fitness of the chromosome. We used a messy GA ([2], [16]) with random keys to represent a permutation of a subset of the available courses, but with the maximum chromosome size set high enough that it is generally not a limiting factor.

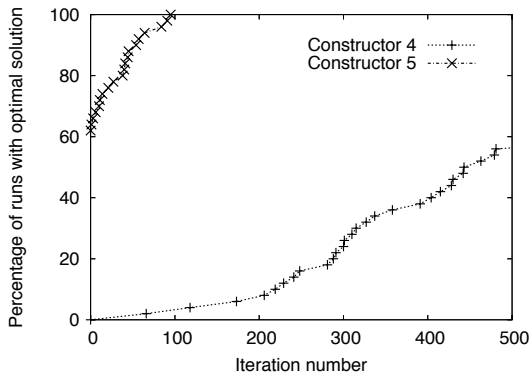

Fig. 3. Performance comparison for Constructors *C4* and *C5*

Solution quality	C3	C4	C5
Valid solution	100%	100%	100%
Valid + 1 soft constraint	.0023%	76.62%	47.77%
Valid + 2 soft constraints	None	13.81%	10.13%
Optimal	None	.0043%	.89%

Fig. 4. Analysis of one million random chromosomes

Constructor *C1* never found a valid solution. *C2* finds valid solutions but rarely finds solutions that satisfy even one soft constraint, and never solutions that satisfy more than the first soft constraint. Constructor *C3* finds valid solutions quickly, and gradually finds solutions that satisfy one and sometimes two of the soft constraints, but none that satisfy all three.

Figure 3 shows the relative performance of *C4* and *C5*. The graph shows the percentage of optimal solutions (satisfying all soft constraints) as a function of the number of iterations. Constructor *C4* generaly finds solutions that satisfy all of the soft constraints fairly easily. After 1000 iterations a solution satisfying all of the soft constraints had been found in 72% of the runs. Little improvement was seen after that point. Even with 10,000 iterations *C4* does not always find an optimal solution. Constructor *C5* found optimal solutions in the initial random pool over 60% of the time, and it always found an optimal solution within 100 iterations.

B. Experiment: random sampling

To better understand the way in which the various constructors transform the search space, we generated one million random chromosomes for each constructor and counted

the number of solutions at each quality level (valid solutions, and valid solutions with one, two or three soft constraints satisfied) that resulted from applying the constructor. The table in Figure 4 shows the results. Columns for *C1*, which never found a valid solution, and *C2*, which always found valid solutions but never found solutions that satisfied any soft constraints, are omitted to save space.

With *C3*, solutions that satisfy one soft constraint can be found a small fraction of a percent of the time, and the GA is fairly effective at finding solutions that satisfy two constraints. *C4* creates a dramatic jump over *C3* in the percentage of "good" solutions represented in the search space. With *C4* some optimal solutions were found out of the million random trials, and the search space is also such that a modest number of iterations of the GA are sufficient to find an optimal solution most of the time.

Constructor *C5* shows further improvement over *C4* for the percentage of optimal solutions. That improvement explains why the initial random pool has an optimal solution as often as it does.

C. Experiment: new soft constraints

An obvious question is to what degree we have simply engineered an algorithm specific to the hard and soft constraints that define this problem. After defining the five constructors, we tried to come up with a different but still realistic set of soft constraints that would change the characteristics of optimal schedules significantly. The scenario we came up with was that of a part-time student who works for an accountant. The hard constraints are the same. The new soft constraints are as follows: (a) there is preference for fewer credit hours in the winter and spring, because our hypothetical student works longer hours during tax season; (b) three "hard" classes are acceptable in the fall, but lighter loads are preferred in winter and spring; (c) they're willing to take up to five years (15 quarters) to graduate; and (d) as before, extraneous classes are undesirable.

The definition of a valid schedule did not change, but obviously the characteristics of a good schedule are substantially different under the new set of soft constraints. A perfectly balanced schedule is not possible due to the structure of the degree requirements, but a schedule that violates that soft constraint for only one quarter is possible. The best schedule we found was one quarter longer than the desired length, with the known lower bound on balance and no other soft constraint violations.

Constructors *C1*, *C2* and *C3* never found a solution matching this best-known fitness level. *C4* performed much worse with these soft constraints than with the earlier set, very rarely finding a solution with the best-known fitness.

Constructor *C5* adapted to this new set of soft constraints with no changes other than replacing the list of soft constraint evaluation functions with the new functions described above. In fact, it found this problem easier than the first, converging on the best-known fitness within an average of seven iterations. In this case the constructor is so effective that the GA is hardly needed because a solution of the best-known quality was found in the initial random pool 68% of the time. Although the focus of this paper is not on the implementation architecture, it's very encouraging to see the flexibility that results from using a list of "black-box" functions to define the soft constraints in Constructor *C5*.

The concern behind all of the experiments in this course scheduling domain is that increasingly intelligent greedy mappings might nullify efforts of the search algorithm to search in the transformed space. Certainly this would be the case of a greedy mapping so "intelligent" that it essentially ignored the advice of the input prioritization. However, in these experiments the constructors with the most "intelligence" and the most domain-specific heuristics performed the best in terms of finding a good solution quickly.

VI. Discussion and conclusions

The preceeding experiments suggest several ways in which a greedy transformation creates a new search space with some desirable characteristics. The transformed search space can then be searched using simple, standard algorithms. Perhaps the key advantage to using a greedy transform is the extent to which it can simplify the implementation of search algorithms. Under a greedy transformation we have a set of jobs and want to find a permutation of those jobs that is of high quality according to a fitness function. We can then directly apply any standard techniques for dealing with permutation problems. The details of the actual scheduling domain may be quite messy, but as long as a reasonable greedy algorithm can be defined for constructing schedules in that domain those details don't need to be known outside of the greedy algorithm and fitness function. The greedy algorithm provides a simple way of defining a chromosome for what may be a complex problem domain.

The potential value of simplicity of implementation is only of interest if the resulting algorithm is effective. Here we observe that the many-to-one mapping created by a greedy search space mapping has the potentially useful effect of making poor regions of the solution space unreachable, and in exchange making better regions of the solution space reachable from multiple points in priority space. In a scheduling problem such as the one first considered in this paper, this happens because a greedy algorithm will give a job the resources it needs, but no more than that. Thus, for example, a job with a very late deadline will never be placed at the beginning of the schedule, no matter how early it might appear in the permutation ordering. The random sampling experiments support the idea that under the greedy mapping there are more "good" points in the transformed space than if we were instead searching in the solution space.

The idea that the greedy mapping may make good solutions easier to find may sound like a violation of the "No Free Lunch" Theorem (NFLT) [20]. No algorithm can show this kind of improvement over all possible cost functions. But in practice greedy mappings are specific to some problem domain and specific to some particular cost function. In fact, that cost function defines the sense in which the greedy algorithm is being greedy. There is no violation

of the NFLT because such algorithms already commit to optimizing over some particular cost function or family of similar cost functions. Clearly there will not be any general-purpose greedy mapping that works well over all possible cost functions. The key point is that SWO, Optiflex, and GRASP do not define general-purpose algorithms, but rather they describe approaches or patterns for algorithm design, requiring domain-specific customization. The NFLT does not say that greedy mappings cannot be effectively customized for different cost functions.

The idea of a transform that gives us a search space with a higher density and more clustering of "good" solutions is encouraging, but still not sufficient unless the greedy mapping preserves (and perhaps improves) the sort of neighbor relationships that the desired search algorithm exploits. What this means depends very much on the particular search algorithms we are considering, but the experiments with small neighborhoods (as might be relevant to a mutation operator) and larger neighborhoods are again very encouraging. And the experiment showing that the WMDD heuristic is still effective, even though that heuristic was specifically designed for searching in solution space, is encouraging because it suggests that the greedy algorithm preserves the sort of search space structure that optimization algorithms often exploit.

The first set of experiments looked only at a single greedy transformation, but obviously on any given problem domain we expect that there can be many different approaches to designing greedy algorithms. For example, our greedy algorithm for the SMWT domain had very little "intelligence" to it and much more sophisticated greedy mappings would be possible. As the second set of experiments suggests, more "intelligent" greedy transformations may improve the characteristics of the search space dramatically over simpler transformations.

These results are admittedly preliminary, but still encouraging. Our experiments here explore a number of ways in which the greedy constructor creates a transformed search space with some desirable characteristics. We conjecture that much of the success of scheduling algorithms based on greedy transformations, such as SWO [14], Optiflex [18], and variants on these [19], is actually due to the benefits of the greedy transformation itself.

What would it mean if this conjecture is correct? It would suggest that a better understanding of the characteristics of this transform could be extremely helpful in designing better algorithms. Changes to the search algorithms that sit on top of the greedy transform explore the benefits of the transformed search space only indirectly. But as in the course scheduling example here and in [5], it is usually the case that a wide range of greedy algorithms are possible for a problem domain. Time spent tuning the greedy algorithm itself might be more effective than time spent tuning the algorithms that search the transformed space. This would be just the opposite of the usual approach taken in applying these algorithms to new domains.

We look forward to exploring the nature of greedy transformations in other domains, and toward a better understanding of how to make the most effective use of greedy mappings in scheduling algorithms.

REFERENCES

[1] L. Barbulescu, D. Whitley, and A. Howe. Leap before you look: An effective strategy in an oversubscribed scheduling problem. In *Proc. of the 19th National Conference on Artificial Intelligence*, 2004.

[2] J. C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6(2):154–160, 1994.

[3] J. E. Beasley. Weighted tardiness. Data sets in OR-Library, `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/wtinfo.html`.

[4] J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990. `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

[5] J. Collins and D. Joslin. Improving genetic algorithm performance with intelligent mappings from chromosomes to solutions (poster). In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1407–1408, New York, NY, USA, 2006. ACM Press.

[6] R. K. Congram, C. N. Potts, and S. L. van de Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14(1):52–67, 2002.

[7] H. A. J. Crauwels, C. N. Potts, and L. N. Van Wassenhove. Local search heuristics for the single machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 10(3):341–350, 1998.

[8] J. M. Crawford. An approach to resource constrained project scheduling. In *Proceedings of the 1996 Artificial Intelligence and Manufacturing Research Planning Workshop*, 1996.

[9] K. A. De Jong. *Evolutionary Computation: a Unified Approach*. MIT Press, 2006.

[10] T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[11] P. Festa and M. Resende. An annotated bibliography of GRASP. Technical Report TD-5WYSEW, AT&T Labs, 2004. `http://www.research.att.com/~mgcr/grasp/gannbib/gannbib.html`.

[12] J. Frank and E. Kürklü. Mixed discrete and continuous algorithms for scheduling airborne astronomy observations. In *Proc. of the 2nd Intl. Conference on Constraint Programming, Artificial Intelligence and Operations Research*, 2005.

[13] A. Globus, J. Crawford, J. Lohn, and A. Pryor. A comparison of techniques for scheduling earth observing satellites. In *Proc. of the 16th Conference on the Innovative Applications of Artificial Intelligence*, 2004.

[14] D. E. Joslin and D. P. Clements. Squeaky wheel optimization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), Madison, WI*, pages 340–346, 1998.

[15] J. J. Kanet and X. Li. A weighted modified due date rule for sequencing to minimize weighted tardiness. *Journal of Scheduling*, 7(4):261–276, 2004.

[16] D. Knjazew and D. E. Goldberg. OMEGA – ordering messy GA : Solving permutation problems with the fast messy genetic algorithm and random keys. Technical Report 2000004, Illinois Genetic Algorithms Laboratory (IlliGAL), 2000.

[17] T. Smith and J. Pyle. An effective algorithm for project scheduling with arbitrary temporal constraints. In *Proc. of the 19th National Conference on Artificial Intelligence*, 2004.

[18] G. P. Syswerda. Generation of schedules using a genetic procedure, 1994. U.S. Patent number 5,319,781.

[19] J. Terada, H. Vo, and D. Joslin. Combining genetic algorithms with squeaky-wheel optimization. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1329–1336, New York, NY, USA, 2006. ACM Press.

[20] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.