Combining Genetic Algorithms with Squeaky-Wheel Optimization

Justin Terada
Computer Science and Mathematics Depts.
Seattle University
Seattle, WA 98122
teradaj@seattleu.edu

Hoa Vo
Computer Science Dept.
Seattle University
Seattle, WA 98122
voh@seattleu.edu

David Joslin
Computer Science Dept.
Seattle University
Seattle, WA 98122
joslind@seattleu.edu

ABSTRACT

The AI optimization algorithm called "Squeaky-Wheel Optimization" (SWO) has proven very effective in a variety of real-world applications. Although the ideas behind SWO are more closely tied to those of local search such as hill-climbing, in some ways SWO can be thought of as an evolutionary algorithm. From that point of view SWO makes a number of design decisions that are at odds with the conventional wisdom of evolutionary algorithms, but not for any clear reasons. This suggests the possibility of improving on SWO by incorporating aspects of Genetic Algorithms that are known to be effective. We compare several algorithm variants on a set of constrained optimization benchmarks, and present some preliminary results suggesting that combining ideas from SWO with a more standard GA approach yields some significant improvements over both.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods, Scheduling

General Terms

Algorithms

Keywords

Genetic Algorithms, scheduling, optimization, hybrid algorithms

1. INTRODUCTION

The field of evolutionary algorithms encompasses a very wide variety of algorithms. Our starting point for this research is an algorithm called "Squeaky-Wheel Optimization" (SWO) [8]. SWO has historically been described as a variant of local search algorithms such as hill-climbing rather than as an evolutionary algorithm. This is partly reflective of the history of how SWO was developed, but also simply because the SWO algorithm does not readily fit into any of the usual paradigms of evolutionary algorithms.

It is possible, however, to think about what the SWO algorithm looks like from a Genetic Algorithms (GA) point of view, and we argue that doing so turns out to be instructive. In some respects design decisions made in the development of SWO start to appear arbitrarily and unnecessarily restrictive. For example, in describing SWO as a GA, we see it as a GA with a pool size of a single chromosome, obviously not what we would usually expect to see in a GA. The idea of adapting SWO to use a larger pool size then naturally arises. We similarly consider other aspects in which SWO can be viewed as a "crippled" GA, and apply a number of ideas from the GA literature to address those issues. At the same time we preserve the essential characteristics of SWO that are not commonly found in genetic algorithms, with the goal of exploring how hybrids of these algorithms might benefit from both bodies of work.

We define several algorithms representing "pure" SWO and GA implementations, as well as hybrid algorithms that combine elements of each. We evaluate these algorithms on a resource-constrained "car sequencing" benchmark problem from the *CSP Library*, a collection of constraint satisfaction problem domains [9]. With only one domain our results are obviously preliminary. We describe this as suggestive of some directions of future research.

The next section describes SWO and shows how it can be viewed as a sort of GA. We then describe the hybrid algorithms and the intuitions and motivations behind them, followed by a summary of the car sequencing problem. Our experimental results are presented next, showing that the hybrid algorithms can be significantly more effective. Finally we look at some related work, and discuss some directions for continuing this research.

2. SQUEAKY-WHEEL OPTIMIZATION

The SWO algorithm is driven by a sequence of problem elements (such as the tasks in a factory scheduling problem) that can be thought of as representing a prioritization of those tasks. A greedy polynomial-time "constructor" is used to map a sequence to a solution. In the case of factory scheduling, the constructor might schedule each task in the sequence to the earliest possible start time, the best production line for the task, etc., depending on the problem.

Once a solution has been produced, SWO applies an "analyzer," or what might in other contexts be called "critics,"

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

to identify flaws in the solution. Flaws in the solution are translated into "blame" attached to problem elements. The priority sequence is then adjusted so that the greater the blame associated with a problem element, the more that element is moved toward the front of the sequence. Moving an element closer to the front of the sequence is in effect giving it a higher priority for the constructor. For example, if a task is found to be late in a schedule and is given a higher priority in this fashion, then on the next iteration it has a better chance of allocating whatever resources it might need because the greedy constructor handles that task sooner than before.

The modified sequence is fed back into the constructor, and this cycle of construction, analysis and re-prioritization continues until some stopping condition is reached. Unlike hill-climbing, in the original form of SWO the modified sequence is used even if the solution quality was worse than the previous iteration. Of course, the best solution found is remembered and returned at the end. Although SWO has usually been described in comparison to local search algorithms, the moves made on each iteration of SWO are generally not very "local" because small priority changes can result in very different solutions from the constructor.

3. COMPARING SWO AND GA

On two points we can find some common ground between SWO and GAs, although in both cases the connection is to aspects that are outside the mainstream of GAs:

Chromosome. The priority sequence in SWO is analogous to a chromosome for a GA. The mapping from the genotype (priority sequence) to the phenotype (a solution) is more complex than is typically the case in GAs because the constructor algorithm is non-trivial, but this alone doesn't take us outside the realm of Genetic Algorithms. It does mean that search in the space of chromosomes is at a more abstract level than is typically the case for GAs.

Mutation. It is unusual but not unheard of for GAs to use directed mutation or "genetic engineering" in which analysis of the phenotype is used to modify the genotype in ways that are intended to help steer the search toward better solutions. SWO introduces some noise into the calculation of the distance to move an element in the sequence during prioritization, so there is some degree of random mutation as well. But SWO does not use pure random mutation as would most commonly be the case with a GA.

In several other ways SWO differs substantially from GA approaches, although not in ways that seem well motivated:

Fitness. SWO doesn't pay attention to solution quality directly except in remembering the best solution found on any iteration. From a GA point of view it is natural to define the fitness of a chromosome to be the fitness of the solution generated by the constructor given that chromosome as input.

Elitism. As mentioned above, on each iteration the modified sequence is used to generate a new solution regardless of the change in solution quality. Elitism is important to GAs, and a preference for improvement is important in most other optimization algorithms. The rationale for ignoring this conventional wisdom in SWO is not clear. In comparison to hill-climbing and GAs, SWO avoids the common problems of local optima and premature convergence, but even so it seems counter-intuitive that it should be desirable to always avoid elitism.

	Options		
Class	**A (1/3)**	**B (2/3)**	**C (1/2)**
1	Yes	Yes	Yes
2	No	Yes	No
3	No	No	Yes
2	No	Yes	No
1	Yes	Yes	Yes

Figure 1: Simple example solution

Population. SWO effectively has a pool size of one, because the algorithm repeatedly modifies a single priority sequence. When we look at SWO from a GA point of view an obvious idea is to allow a pool of sequences (chromosomes) to persist, and combining this with the important idea of elitism, to use competitive selection based on fitness.

Crossover. With a pool size of one, SWO had no opportunity to take advantage of crossover operators. This is another obvious idea for possibly improving the performance of GA/SWO hybrid algorithms.

4. THE CAR SEQUENCING PROBLEM

The car sequencing problem is a resource-constrained scheduling problem based on a simple model for automobile production line scheduling [9]. The problem is known to be NP-complete [3]. The cars in this problem are distinguished by the options they require, with cars grouped into classes that share the same set of options. For each option, there is a corresponding station on the production line, constrained to handle at most m cars with that option in any subsequence of n cars, where m and n depend on the option. A valid solution is one that has no violations of this constraint.

As a very simple example, suppose we have three options, air conditioning (option A), a deluxe sound system (option B), and a CD changer (option C). The assembly station for option A can handle at most one car in every three, i.e, for any "window" of three consecutive cars in a schedule, at most one is allowed to have option A. For example, if the first car in the sequence requires that option, then the next two cars cannot require that option. The assembly station for option B can handle two cars out of any three with that option. And finally, at most one car in every two may have option C.

Suppose we have three classes of cars. Class 1 has all three options. Class 2 has just option B, and Class 3 has just option C. We must schedule two cars from each of the first two classes, and one from the third.

Figure 1 shows a valid solution. Each row represents one car in the scheduled sequence, with the class number for that car indicated followed by a column for each option. By inspection we can see that the frequency constraints are satisfied for each option. For example, we are allowed to have two cars in every three with option B. The first two cars in the sequence have that option, so the third must not have it, and by putting a car from Class 3 in that slot we avoid a constraint violation. For any three consecutive cars we can also see that this constraint is satisfied.

Problems can be generated with varying degrees of difficulty by adjusting the utilization percentage. A set of benchmarks constructed in this fashion are provided as part of the

CSP Library, with seven sets of ten problems each and utilization percentages ranging from 60 to 90 [9]. We use these benchmarks in the experiments described below.

5. HYBRID ALGORITHMS

To explore ways that ideas from SWO and GAs can be combined, we implemented several algorithms that combine elements of each. For all algorithms, the chromosome (or priority sequence for SWO) was a vector of floating point values used to define a permutation. Each position in the vector corresponds to one of the cars to be scheduled, and when we want the permutation we sort the cars by the corresponding values. The SWO-like algorithms kept a single chromosome (pool size of one), and the GA-like algorithms had a pool size of fifty and were steady-state GAs using tournament selection.

All of the algorithms used the same greedy, polynomial-time constructor. After the chromosome is used to generate a permutation of the cars to be scheduled, the constructor adds them to the schedule one at a time at the earliest possible time that violates none of the constraints on the frequency of the options. When the constructor finishes, the resulting schedule may have unfilled slots; if so the solution is invalid. A valid solution has been found when the constructor returns a schedule that has no empty slots.

SWO depends on "genetic engineering" operators. We can think of these as implementing a sort of "intelligent" mutation, although calling them "intelligent" may be misleading. With SWO, and therefore in our implementation, these operators only need to make changes to the priority sequence that may tend to improve a particular constraint violation, ignoring interactions with other constraints. But of course ignoring interactions and making such priority changes myopically will very often mean that the change does not have the desired effect. The hope of course is that in spite of the imperfect nature of these genetic engineering operators, an algorithm such as SWO or a GA can use them to search for good solutions.

We defined three genetic engineering operators:

Overflow. As described above, the constructor may generate a plan that has empty slots. The schedule is then also longer than a valid schedule by that number of slots, and we call the excess slots beyond that valid length the "overflow" for a schedule. The car classes found in that overflow may have a better chance of being scheduled successfully if they are considered earlier by the constructor. We adjust the corresponding weights in the chromosome for the overflow cars.

Unfillable Slots. Not every empty slot in a completed schedule is constrained by the nearby cars so that no class of cars could be scheduled in that slot, but intuitively such slots identify the worst sort of constraint conflicts that can occur. Once such a conflict has occurred no reordering of the rest of the permutation can yield a legal schedule. The conflict must be the result of only the cars within the maximum "window" size, for the windows that define the scheduling constraints for each option. This operator adjusts the weights of the cars within that window size of an unfillable slot.

Option Utilization. This operator examines an empty slot and determines the options that could prevent cars from being scheduled there. The corresponding weights for cars with those options are adjusted to give them an earlier chance of filling such slots.

The original implementations of SWO all applied whatever genetic engineering operators were defined on every iteration. Intuitively this seemed unlikely to be preferable to allowing each operator to be applied individually. However, as discussed below, that intuition turned out to be wrong for these experiments. Except where otherwise noted the three above operators were applied consecutively as a single compound operator.

We also defined a crossover operator, implementing a standard single-point crossover. In the experiments presented here we did not incorporate random mutation, even though random mutation would normally be included in a GA and could be added to any of the other algorithms. In experiments with various rates of mutation we did not find random mutation to be helpful. For the two best algorithms described below adding a 1% mutation rate caused performance to deteriorate slightly, eventually catching up but not exceeding the non-mutation performance. For this reason we report on the algorithms with random mutation omitted.

All of the algorithms selected an operator, selected an appropriate number of chromosomes from the pool then applied the operator to generate one new chromosome. In the SWO-like algorithms the "pool" is a single chromosome, and the only applicable operators take one chromosome as input and produce one chromosome as output. The new chromosome(s) are evaluated by running the constructor on the permutation defined by the chromosome, and counting the constraint violations in the resulting solution. A score of zero means that a valid solution was found.

With the exception of the "pure" SWO algorithm, the new chromosome(s) were added to the pool only if they could replace less-fit chromosomes. In one SWO-like algorithm, following the original design of SWO, the new chromosome always replaced the previous one even if the fitness decreased. In the other, it replaced that chromosome only if it improved the fitness.

We define our various algorithms in terms of the following components:

SWO This indicates that the algorithm uses the genetic algorithm operators described above. Unless otherwise noted (the "Multi" variant) the three operators were treated as a single compound operator and applied consecutively.

HC This stands for "hill-climbing," although in the context of genetic algorithms "elitism" might be a better term. With a population size of one this simply means that on each iteration the new chromosome replaces the old only if it has equal or greater fitness. With a larger population size it means that the new chromosome survives only if it displaces a member of that population when the population is sorted by fitness. If not specified it means that the chromosome produced by an operator replaces its parent (or one of its parents in the case of crossover) whether it has better fitness or not.

Pool This means that a pool or population of chromosomes is maintained. In these experiments the size of that

```
CarSchedule(SWO,Multi,HC,Pool,Xov)
    If Pool
        Set size of population to 50
    Else
        Set size of population to 1
    Fill population with random chromosomes

    availableOperators = {}
    If SWO
        If Multi
            add three genetic engineering operators to availableOperators
        Else
            add compound genetic engineering operator to availableOperators
    If Xov
        add Crossover to availableOperators

    While iterations < maxIterations and optimal solution not found
        operator = select operator randomly from availableOperators
        parentChromosomes = select chromosome(s) from population using tournament selection
        childChromosome = apply operator to parentChromosome(s)

        If HC
            if childChromosome is better than worst in population, replace worst with childChromosome
        Else
            add childChromosome to population, replacing parent

    Return best chromosome found
```

Figure 2: Pseudocode summarizing algorithm variants

population is 50. If not specified, it means that there is only one chromosome.

Xov This indicates that a crossover operator is used. Obviously it only makes sense with a population size greater than one.

It makes no sense to have a crossover operator if the population size is one, but otherwise most of the combinations can be tested. We have the following algorithms:

SWO This is the "pure" SWO algorithm. With none of the other components we have only the genetic-engineering operators, a population size of one, and the new chromosome always replaces the previous one whether or not it is an improvement. This follows the design of the original SWO described in [8].

SWO+Multi The "Multi" option indicates that the genetic engineering operators were treated as three separate operators, rather than as a single compound operator. We expected this to be an improvement over SWO because the changes to the chromosome could be more "fine tuned" to the flaws detected in the solution. It turned out to be less effective than SWO, however, and other experiments didn't show it helping any of the other algorithms, so it only shows up in this one variant.

SWO+HC This algorithm differs from SWO only in that the new chromosome is accepted only if it is at least as good as the previous one. We can think of it as a hill-climbing variant of SWO.

SWO+Pool This is SWO without the hill-climbing option, but with a population size of 50. Without crossover the members of the population do not interact. The potential value of this change to SWO is that because of tournament selection, the better chromosomes will receive more attention. We could think of this as a "time-sharing" SWO with a bias toward trying to improve most fit members of the population.

SWO+Pool+HC This adds the "hill-climbing" variant to the previous algorithm. The members of the population all still evolve independently without interaction, with a bias toward giving attention to the most fit members of the population, but a child must have fitness at least as good as the parent in order to replace the parent.

SWO+Pool+Xov Adding the Crossover operator to SWO+Pool adds the potential for interaction between the members of the population. The child resulting from Crossover replaces one of the parents (the first one that was selected) whether the child is an improvement or not.

SWO+Pool+HC+Xov This is now the equivalent of the GA algorithm with the genetic engineering operator added.

GA The GA algorithm can be described as Pool+HC+Xov. It differs from the previous algorithm only in that it does not include the genetic engineering operators.

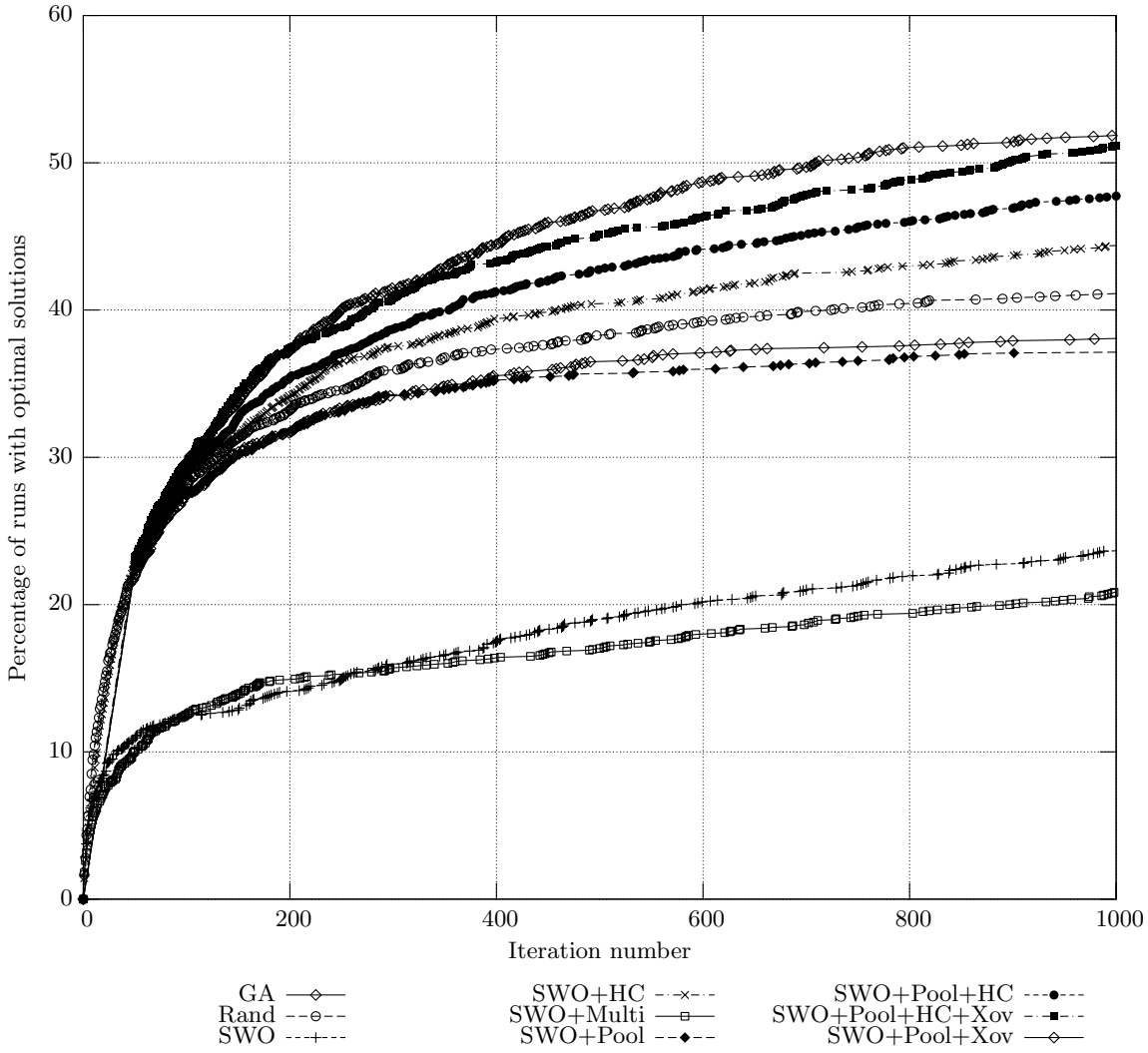

Figure 3: Solution quality

Rand Finally, to establish a base-line we ran an algorithm with a single chromosome, and an operator that ignores the previous solution and returns a random vector for the new chromosome. The idea here was to try to identify problems that were so easy that even random probes of the search space would find solutions quickly. In such cases the flexibility of the polynomial-time constructor is doing a lot of the work, and this algorithm helps to separate that effect from the contributions of the other aspects of the algorithms.

Figure 2 summarizes the algorithms. As noted above, the GA algorithm results from selecting the *Pool*, *HC*, and *Xov* options.

6. EXPERIMENTAL RESULTS

We used a benchmark set of 70 satisfiable problems, consisting of ten problems each for resource utilization percentages of 60, 65, 70, 75, 80, 85 and 90 [9]. Increasing the resource utilization percentage increases the difficulty of the problem. The "60" series of problems are very easy, while the "90" series of problems tend to be very hard. In fact, one of the 90 series happens to be very easy as well, both for our algorithms and for other reported results. We began with 24 runs of 1000 iterations each for all of the algorithms described above. We considered a run successful only if it found a valid solution.

Figure 3 graphs the percentage of valid solutions found across all of the runs against the number of iterations. To show the comparison more clearly the y-axis is truncated at sixty percent. We truncate the y-axis at the same point for all of the graphs that follow.

Most strikingly, the SWO and SWO+Multi algorithms lag all of the others. They perform substantially below even the Rand algorithm. As noted above, contrary to our intuitions applying all of the genetic engineering operators on every iteration turned out to give SWO a small edge over SWO+Multi, but not enough to make it competitive even with Rand.

An obvious question is whether the implementation of SWO might just be a poor implementation. It is possible,

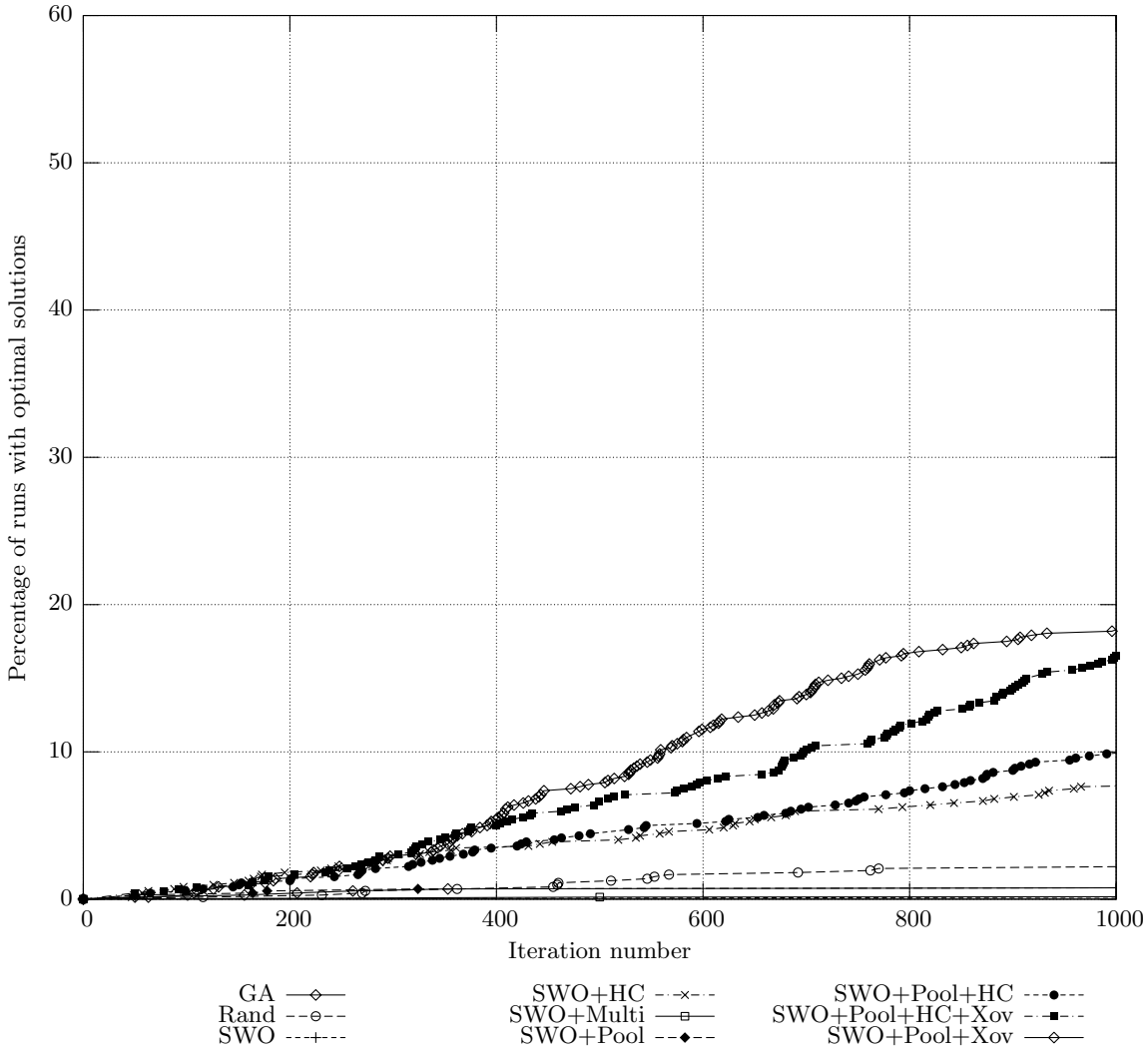

Figure 4: Solution quality for difficult problems

of course, that a different implementation of SWO for these problems could be highly effective, and the poor results seen here do not prove otherwise. We note, however, that these operators are evidently helpful in some of the other algorithm variants considered here (not in this graph, but seen in results presented below), and they were designed based on prior experience with successful SWO implementations.

The GA algorithm is superior to all of the others in this initial experiment, although the SWO+Pool+HC+Xov algorithm has essentially caught up by the 1000-iteration point. We note that in general, the greater the similarity to the GA algorithm, the better the performance. Since SWO+Pool+HC+Xov and GA differ only in the genetic engineering operators, based on these data alone the genetic engineering operators would appear to be slightly detrimental.

For the next step in our analysis we eliminated the "easy" problems, which we defined to be those that the Rand algorithm solved more than half the time in 5000 iterations. One of the "90" series of problems was easy, and solved by Rand. The rest of the "90" series were too difficult for any of our algorithms, and so we eliminated them from the experiments. The GA approach to the same problems reported by [12] also failed to solve any but the one easy problem in that subset. We had much better success on that subset when using a domain-specific post-processor, but because it has nothing to do with either SWO or GA approaches specifically we omitted it from the results reported here. The post-processor improved every algorithm, but the comparisons between the algorithms remained the same with GA performing better than SWO.

After eliminating the "90" series, there were 30 problems remaining that were not easy for Rand to solve. These consisted of problem 70-09, and all of the problems from the "75" through "85" sets except 85-03.

Figure 4 looks at the subset of the previous data corresponding to these thirty "hard" problems. Again the y-axis is truncated to show more detail, and the percentage shown is the percentage of the thirty problems used. Rand shows up because of the problems that it solved less than ten per-

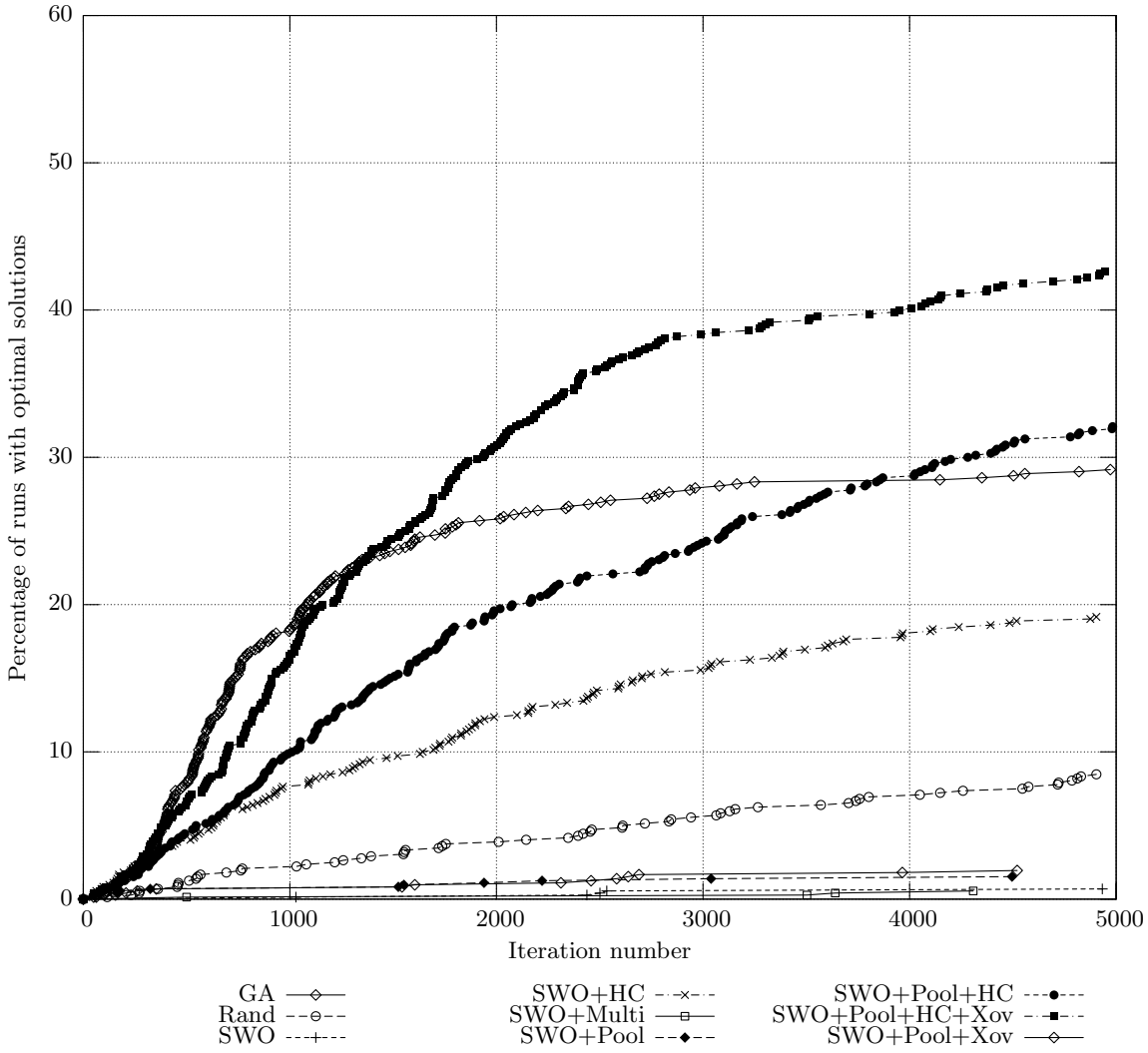

Figure 5: Solution quality on difficult problems, 5000 iterations

cent of the time. SWO, SWO+Multi, and SWO+Pool do not show up because they solved no problems that Rand didn't find easy. Again, the GA algorithm dominates, and the next closest is the algorithm that is most similar to GA, and so on.

We next took the four best algorithms (SWO+HC, SWO+Pool+HC, SWO+Pool+HC+Xov, and GA) and compared them with 5000 iterations on the set of thirty "hard" problems. The results are shown in Figure 5. It turned out that the SWO+Pool+HC+Xov algorithm (i.e., GA with genetic engineering operators) was more effective in the long run. Eliminating any other component from the algorithm, however, made it less effective than the GA.

As one reviewer suggested, this may show that the GA has converged at 1000 iterations and that the genetic engineering operators are preserving diversity. Although a GA with restarts may have addressed this concern, SWO also has an analogous problem where after many iterations, the operators no longer make a difference significant enough to allow for much progress.

7. RELATED WORK

Examples of previous work on the car sequencing problem include the application of greedy local search [5], ant colony optimization [5, 6], integer programming [6], CSP search [6], heuristic repair, TABU search, and genetic algorithms with hill-climbing to try to improve upon each offspring produced by crossover [12].

It is hard to compare our results directly against the genetic algorithms approach reported in [12]. Their approach involves using both a genetic algorithm and hill-climbing local search on each chromosome, and the structure of their algorithm also makes any attempt to compare their performance as a function of the number of iterations to ours infeasible. We can however look at their results on the three hardest benchmark sets (utility percentages of 80, 85 and 90), for a very rough comparison. They report solving 61%, 21% and 1% of 100 total runs (ten runs each for ten problems) in each category. We also solved 1% in the "90" set, because one of those problems happens to be very easy. Even with 5000 iterations the GA algorithm only solved 16% com-

pared to their 61% for the "80" set, and 2% compared to their 21% for the "85" set. We did, however, have a domain-specific post-processing step that raised our performance in those three hardest subsets to 89%, 48% and 19% respectively. This was not used in any of the experiments reported here.

The use of a non-trivial "constructor" that maps chromosomes to solutions is not unheard of with genetic algorithms, although certainly it is more common that the mapping from chromosomes to solutions be very simple and direct. A scheduling algorithm described in a patent by Syswerda [11] uses a "schedule builder" to construct a schedule given a sequence of tasks. A GA searches the space of chromosomes (task sequences), and no "genetic engineering" is used.

An algorithm for experiment planning for planetary rovers used a GA to search a space of chromosomes that represented parameters describing a "plan strategy," which was then evaluated by simulating the execution of that strategy [7]. The simulation in this case could be thought of as evaluating a dynamic constructor in a simulated environment. The plan strategy was not a plan itself, but as with a task sequence for SWO or Syswerda's algorithm, it represents an abstraction that a complex algorithm must map to a solution in order to evaluate the fitness of the chromosome.

SWO has been successfully applied to a wide variety of domains, including factory scheduling and graph coloring [8], satellite downlink scheduling [1], satellite observation scheduling [4], project scheduling [10], and scheduling of airborne astronomical observations [2].

8. CONCLUSIONS

As mentioned above, SWO has been successfully applied to a wide variety of domains. It has been patented, used for at least one commercially successful product, and used successfully in several real-world applications, not just simplified problems and benchmarks. In light of that, some of the design decisions that define SWO are surprising and counter-intuitive, at least once we start viewing it in light of evolutionary computation techniques that are widely used and effective.

Our experimental results suggest that the SWO approach can benefit from incorporation of standard GA techniques such as the use of a population larger than one, the use of a crossover operator, and elitism. In fact in these experiments the closer a hybrid algorithm is to GA, the better it is likely to do. In some cases the incorporation of the genetic engineering operators that characterize SWO was harmful, and an advantage over GA emerged only with a sufficiently large number of iterations.

Any conclusions based on a single domain have to be viewed as very tentative, but we believe that these initial results are significant and provocative. Given the success of SWO in spite of what are arguably some serious design flaws, the improvements we see with hybrid algorithms combining elements of GA and SWO are very encouraging. We look forward to further exploring this intersection of two promising fields of research.

9. 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. 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.

[3] I. P. Gent. Two results on car-sequencing problems. Technical Report APES-02-1998, Dept. of Computer Science, University of Strathclyde, Glasgow, UK, 1998.

[4] 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.

[5] J. Gottlieb, M. Puchta, and C. Solnon. A study of greedy, local search and ant colony optimization approaches for car sequencing problems. In G. R. R. *et.al.*, editor, *Applications of Evolutionary Computing, EvoWorkshops2003*, volume 2611 of *LNCS*, pages 247–258. Springer-Verlag, 2003.

[6] M. Gravel, C. Gagné, and W. L. Price. Review and comparison of three methods for the solution of the car sequencing problem. *Journal of the Operational Research Society*, pages 1287-1295, November 2005.

[7] D. Joslin, J. Frank, A. Jónsson, and D. Smith. Simulation-based planning for planetary rover experiments. In *Proceedings of the Winter Simulation Conference*, 2005.

[8] 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.

[9] B. Smith. Car sequencing. In I. P. Gent and T. Walsh, editors, *CSPLib: A Problem Library for Constraints*. `http://www.cse.unsw.edu.au/~tw/csplib/prob/prob001/index.html`, 2004.

[10] 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.

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

[12] T. Warwick and E. P. K. Tsang. Tackling car sequencing problems using a generic genetic algorithm. *Evolutionary Computation*, 3(3):267–298, 1996.