“Squeaky Wheel” Optimization

David E. Joslin
i2 Technologies
909 E. Las Colinas Blvd.
Irving, TX 75039
dj@i2.com

David P. Clements
Computational Intelligence Research Laboratory
University of Oregon
Eugene, OR 97403-1269
clements@cirl.uoregon.edu

Abstract

We describe a general approach to optimization which we term “Squeaky Wheel” Optimization (SWO). In SWO, a greedy algorithm is used to construct a solution which is then analyzed to find the trouble spots, i.e., those elements, that, if improved, are likely to improve the objective function score. That analysis is used to generate new priorities that determine the order in which the greedy algorithm constructs the next solution. This Construct/Analyze/Prioritize cycle continues until some limit is reached, or an acceptable solution is found.

SWO can be viewed as operating on two search spaces: solutions and prioritizations. Successive solutions are only indirectly related, via the re-prioritization that results from analyzing the prior solution. Similarly, successive prioritizations are generated by constructing and analyzing solutions. This “coupled search” has some interesting properties, which we discuss.

We report encouraging experimental results on two domains, scheduling problems that arise in fiber-optic cable manufacturing, and graph coloring problems. The fact that these domains are very different supports our claim that SWO is a general technique for optimization.

Overview

We describe a general approach to optimization which we term “Squeaky Wheel” Optimization (SWO) (Clements *et al.* 1997). The core of SWO is a Construct/Analyze/Prioritize cycle, illustrated in Figure 1. A solution is constructed by a greedy algorithm, making decisions in an order determined by priorities assigned to the elements of the problem. That solution is then analyzed to find the elements of the problem that are “trouble makers.” The priorities of the trouble makers are then increased, causing the greedy constructor to deal with them sooner on the next iteration. This cycle repeats until a termination condition occurs.

On each iteration, the analyzer determines which elements of the problem are causing the most trouble in the current solution, and the prioritizer ensures that the constructor gives more attention to those elements on the next iteration. (“The squeaky wheel gets the grease.”) The construction, analysis and prioritization are all in terms of the elements that define a problem domain. In a scheduling domain, for example, those elements might be tasks. In graph coloring, those elements might be the nodes to be colored.

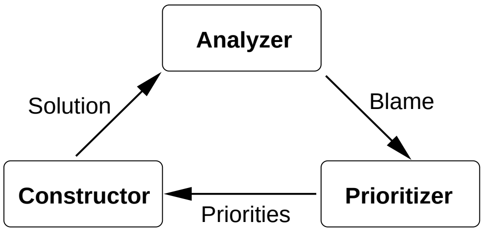

Figure 1: The Construct/Analyze/Prioritize cycle

The three main components of SWO are:

Constructor. Given a sequence of problem elements, the constructor generates a solution using a greedy algorithm, with no backtracking. The sequence determines the order in which decisions are made, and can be thought of as a “strategy” or “recipe” for constructing a new solution. (This “solution” may violate hard constraints.)

Analyzer. The analyzer assigns a numeric “blame” factor to the problem elements that contribute to flaws in the current solution. For example, if minimizing lateness in a scheduling problem is one of the objectives, then blame would be assigned to late tasks.

A key principle behind SWO is that solutions can reveal problem structure. By analyzing a solution, we can often identify elements of that solution that work well, and elements that work poorly. This information about problem structure is local, in that it may only apply to the part of the search space currently under examination, but may be useful in determining what direction the search should go next.

Prioritizer. The prioritizer uses the blame factors assigned by the analyzer to modify the previous sequence of problem elements. Elements that received blame are moved toward the front of the sequence. The higher the blame, the further the element is moved.

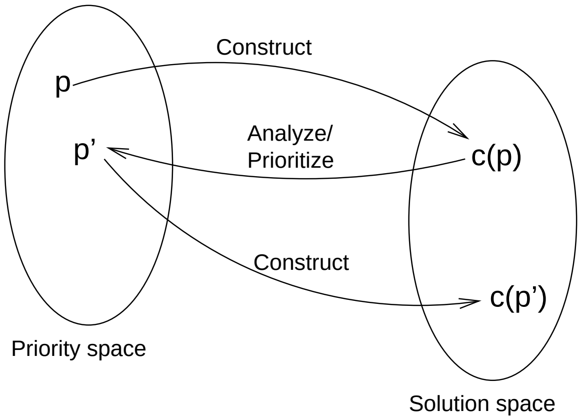

Figure 2: Coupled search spaces

The priority sequence plays a key role in SWO. As a difficult problem element moves forward in the sequence it is handled sooner by the constructor. It also tends to be handled better, thus decreasing its blame factor. Difficult elements rise rapidly to a place in the sequence where they are handled well. Once there, the blame assigned to them drops, causing them to slowly sink in the sequence as other parts of the problem that are not handled as well are given increased priority. Eventually, difficult elements sink back to the point where they are no longer handled well, causing them to receive higher blame and to move forward in the sequence again. Elements that are always easy to handle sink to the end of the sequence and stay there.

Real problems often combine some elements that are difficult to get right, plus others that are easy. In the scheduling problems presented below, some tasks can be assigned to just a few production lines, while others allow for much more flexibility. Some have due dates close to their release time, while others have a lot of leeway. It is sometimes possible to identify "difficult" elements of a problem with static analysis, but interactions can be complex, and elements that are causing difficulty in one part of the search space may be no trouble at all in another. Rather than trying to identify elements that are globally difficult by analyzing the entire problem, we analyze individual solutions in order to find elements that are *locally* difficult. Globally difficult elements tend to be identified over time, as they are difficult across wide parts of the search space.

It is useful to think of SWO as searching two coupled spaces, as illustrated in Figure 2. One search space is the familiar solution space, and the other is *priority space*. Moves in the solution space are made indirectly, via the re-prioritization that results from analyzing the prior solution. Similarly, successive prioritizations are generated by constructing and analyzing a solution, and then using the blame that results from that analysis to modify the previous prioritization.

One consequence of the coupled search spaces is that a small change in the sequence of elements generated by the prioritizer may correspond to a large change in the corresponding solution generated by the constructor, compared to the solution from the previous iteration. Moving an element forward in the sequence may change its state in the resulting solution. In addition, any elements that now occur after it in the sequence must accommodate that element's state. For example, in the scheduling domain, moving a task earlier in the priority sequence may allow it to find a better place in the schedule, with lower-priority tasks "filling in the gaps" after that task has been placed.

The result is a large move that is "coherent" in the sense that it is similar to what we might expect from moving the higher priority task, then propagating the effects of that change by moving lower priority tasks as needed. This single move may correspond to a large number of moves for a search algorithm that only looks at local changes to the solution, and it may thus be difficult for such an algorithm to find.

The fact that SWO makes large moves in both search spaces is one obvious difference between SWO and traditional local search techniques, such as WSAT (Selman, Kautz, & Cohen 1993). Another difference is that with SWO, moves are never selected based on their effect on the objective function. Instead, unlike hillclimbing techniques, each move is made in response to "trouble spots" found in the current solution. The resulting move may be uphill or downhill with respect to the objective function. In effect, SWO deals with local optima in the solution space by ignoring them.

In priority space the only "local optima" are those in which all elements of a solution are assigned equal blame. SWO tends to avoid getting trapped in local optima, because analysis and prioritization will always (in practice) suggest changes in the sequence, thus changing the solution generated on the next iteration. This does not guarantee that SWO will not become trapped in a small cycle, however. In our implementations we have introduced small amounts of randomness in the basic cycle. We also restart SWO periodically with a new initial sequence.

SWO for scheduling

This section describes an application of SWO to a fiber-optic production line scheduling problem, derived from data provided by Lucent Technologies. In this particular plant, a cable may be assembled on any one of 13 parallel production lines. For each cable type, only a subset of the production lines are compatible, and the time required to produce the cable will depend on which of the compatible lines is selected. Each cable also has a setup time, which depends on its own cable type and that of its predecessor. Setups between certain pairs of cable types are infeasible. Task preemption is not allowed, i.e. once a cable has started processing on a line, it finishes without interruption.

Each cable is assigned a release time and due date. Production cannot begin before the release time. The objective function includes a penalty for missing due dates, and a penalty for setup times.

Implementation

We describe the implementation in terms of the three main components of SWO:

Constructor. The constructor builds a schedule by adding tasks one at a time, in the order they occur in the priority sequence. A task is added by selecting a line and a position relative to the tasks already in that line. A task may be inserted between any two tasks already in the line or at the beginning or end of that line's schedule. Changes to the relative positions of the tasks already in the line are not considered. Each task in the line is then assigned to its earliest possible start time, subject to the ordering, i.e., a task starts at either its release time, or immediately after the previous task on that line, whichever is greater.

For each of the possible insertion points in the schedule, relative to the tasks already in each line, the constructor calculates the effect on the objective function, and the task is placed at the best-scoring location. Ties are broken randomly. After all tasks have been placed, the constructor applies SWO to the individual line schedules, attempting to improve the score for each line by reordering the cables that were assigned to it.

Analyzer. To assign blame to each task in the current schedule, the analyzer first calculates a lower bound on the minimum possible cost that each task could contribute to any schedule. For example, if a task has a release time that is later than its due date, then it will be late in every schedule, and the minimum possible cost already includes that penalty. Minimum possible setup costs are also included. For a given schedule, the blame assigned to each task is its "excess cost," the difference between its actual cost and its minimum possible cost. Excess lateness costs are assigned to tasks that are late, and excess setup costs are split between adjacent tasks.

Prioritizer. Once the blame has been assigned, the prioritizer modifies the previous sequence of tasks by moving tasks with non-zero blame factors forward in the sequence. Tasks are moved forward a distance that increases with the magnitude of the blame. To move from the back of the sequence to the front, a task must have a high blame factor over several iterations.

Our current implementation has considerable room for improvement. The analysis and feedback currently being used are very simple, and the construction of schedules could take various heuristics into account, such as preferring to place a task in a line that has more "slack," all other things being equal.

Experimental results

We have six sets of test data, ranging in size from 40 to 297 tasks, all with 13 parallel production lines. The largest problem was the largest that the manufacturer required in practice. We compare the following solution methods:

- SWO Applies the SWO architecture to the problem, running for a fixed number of iterations and returning the best schedule it finds.
- TABU Uses TABU search (Glover & Laguna 1997), a local search algorithm in which moves that increase cost are permitted to avoid getting trapped at local optima. To avoid cycling, when an "uphill" move is made, it is not allowed to be immediately undone.
- IP Applies an Integer Programming (IP) solver, using an encoding described in (Clements *et al.* 1997).

		SWO		TABU		IP	
Data Set	Best Obj	Avg Obj	Avg Time	Obj	Time	Obj	Time
40	1890	1890	48	1911	425	1934	20
50	3101	3156	57	3292	732	3221	175
60	2580	2584	87	2837	1325	2729	6144
70	2713	2727	124	2878	2046	2897	4950
148	8869	8927	431	10421	17260	—	—
297	17503	17696	1300	—	—	—	—

Table 1: Experimental results: scheduling

On the 297 task problem, SWO was far more effective than either TABU or IP. TABU, for example, failed to find a feasible schedule after running for over 24 hours. On the smallest problems, TABU and IP were able to find solutions, but SWO outperformed both by a substantial margin.

Table 1 presents results on each problem for SWO, TABU and IP. For SWO, ten trials were run and the results averaged. The TABU and IP implementations were deterministic, so only the results of a single run are shown. The second column of the table shows the best objective function value we have ever observed on each problem. The remaining columns show the objective function value and running times for SWO, TABU and IP. All but the IP experiments were run on a Sun Sparcstation 10 Model 50. The IP experiments were run on an IBM RS6000 Model 590 (a faster machine).

The best values observed have been the result of combining SWO with IP, as reported in (Clements *et al.* 1997). In that work, SWO generated solutions, running until it had produced a number of "good" schedules. An IP solver was then invoked to re-combine elements of those solutions into a better solution. Although the improvements achieved by the IP solver were relatively small, on the order of 1.5%, it achieved this improvement quickly, and SWO was unable to achieve the same degree of optimization even when given substantially more time. While noting that the hybrid approach can be more effective than SWO alone, and much more effective than IP alone, here we focus on the performance of the individual techniques.

We also note that our very first, fairly naive implementation of SWO for these scheduling problems already outperformed both TABU and IP. Moreover, our improved implementation, reported above, is still fairly simple, and is successful without relying on domain-dependent heuristics. We take this as evidence that the effectiveness of our approach is not due to cleverness in the construction, analysis and prioritization techniques, but due to the effectiveness of the SWO cycle at identifying and responding to whatever elements of the problem happen to be causing difficulty in the local region of the search.

SWO for graph coloring

We have also applied SWO to a a very different domain, graph coloring. Here the objective is to color the nodes of a graph such that no two adjoining nodes have the same color, minimizing the number of colors.

Implementation

The priority sequence for graph coloring consists of an ordered list of nodes. The solver is always trying to produce a coloring that uses colors from the *target set*, which has one less color than was used to color the best solution so far.

Constructor. The constructor assigns colors to nodes in priority sequence order. If a node's color in the previous solution is still available (i.e. no adjacent node is using it yet), and is in the target set, then that color is assigned. If that fails, it tries to assign a color in the current target set, picking the color that is least constraining on adjacent uncolored nodes, i.e. the color that reduces the adjacent nodes' remaining color choices the least. If none of the target colors are available, the constructor tries to "grab" a color in the target set from its neighbors. A color can only be grabbed if all neighbor nodes with that color have at least one other choice within the target set. If multiple colors can be grabbed, then the least constraining one is picked. If no color in the target set can be grabbed then a color outside the target set is assigned.

Nodes that are early in the priority sequence are more likely to have a wide range of colors to pick from. Nodes that come later may grab colors from earlier nodes, but only if the earlier nodes have other color options within the target set.

Analyzer. Blame is assigned to each node whose assigned color is outside the target set. We ran experiments with several different variations of color-based analysis. All of them performed reasonably.

Prioritizer. The prioritizer modifies the previous sequence of nodes by moving nodes with blame forward in the sequence according to how much blame each received. This is done the same way it is done for the scheduling problems. The initial sequence is a list of nodes sorted in decreasing degree order, with some noise added to slightly shuffle the sort.

	IG		SWO	
Data set	Colors	Time	Colors	Time
DSJC125.5	18.9	4.2	18.4	2.7
DSJC250.5	32.8	11.3	31.7	11.8
DSJC500.5	58.6	30.3	56.3	51.7
DSJC1000.5	104.2	112.1	101.6	280.0
C2000.5	190.0	451.5	185.6	1446.4
C4000.5	346.9	1747.1	341.9	7184.6
R125.1	5.0	3.4	5.0	0.3
R125.1c	46.0	1.8	46.0	11.8
R125.5	36.9	3.1	36.0	5.7
R250.1	8.0	11.6	8.0	0.8
R250.1c	64.0	7.6	64.0	62.9
R250.5	68.4	13.8	65.0	27.1
DSJR500.1	12.0	35.0	12.0	3.0
DSJR500.1c	85.0	24.1	85.0	122.2
DSJR500.5	129.6	43.3	124.0	112.7
R1000.1	20.6	144.5	20.0	11.9
R1000.1c	98.8	80.7	101.4	556.6
R1000.5	253.2	170.5	239.0	801.3
flat300_20_0	20.2	6.2	25.3	20.7
flat300_26_0	37.1	12.7	36.0	15.9
flat300_28_0	37.0	15.8	35.8	16.5
flat1000_50_0	65.6	242.5	100.0	267.5
flat1000_60_0	102.5	144.7	100.2	265.2
flat1000_76_0	103.6	131.9	100.8	266.4
latin_sqr_10	106.7	99.0	111.4	486.7
le450_15a	17.9	28.1	15.0	8.9
le450_15b	17.9	26.9	15.0	9.4
le450_15c	25.6	24.0	21.3	9.4
le450_15d	25.8	22.5	21.4	9.6
mulsol.i.1	49.0	6.9	49.0	13.5
school1	14.0	17.4	14.0	11.8
school1_nsh	14.1	14.8	14.0	10.4

Table 2: Experimental results: graph coloring

Experimental results

We applied SWO to a standard set of graph coloring problems, including random graphs and application graphs that model register allocation and class scheduling problems. These were collected for the Second DIMACS Implementation Challenge (Johnson & Trick 1996), which includes results for several algorithms on these problems (Culberson & Luo 1996; Glover, Parker, & Ryan 1996; Lewandowski & Condon 1996; Morgenstern 1996). Problems range from 125 nodes with 209 edges to 4000 nodes with 4,000,268 edges.

(Glover, Parker, & Ryan 1996) is the only paper that uses a general search technique, TABU with branch and bound, rather than a graph coloring specific algorithm. This approach had the worst reported average results in the group. (Morgenstern 1996) used a distributed IMPASSE algorithm and had the best overall colorings, but also required that the target number of colors, as well as several other problem specific parameters be passed to the solver. (Lewandowski & Condon 1996) also found good solutions for this problem set. Their approach used a hybrid of parallel IMPASSE and systematic search on a 32 processor CM-5. (Culberson & Luo 1996) used an Iterated Greedy (IG) algorithm that bears some similarity to SWO. IG is the simplest algorithm in the group. It's solution quality falls between the IMPASSE algorithms and TABU but solves the entire set in 1 to 2 percent of the time taken by the other methods. Both IG and IMPASSE are discussed further under related work.

Table 2 compares SWO with the results for IG, the best serial algorithm in (Johnson & Trick 1996), and the only one with results for all 32 problems. The table shows average results and run times (in CPU seconds) of ten independent runs for each algorithm. Both algorithms terminated after 1000 iterations.

The times shown for IG are those reported in (Culberson & Luo 1996), normalized to our times using the DIMACS benchmarking program *dfmax*, provided for this purpose. Therefore, timing comparisons are only approximate. Our machine, a Pentium Pro 200Mhz workstation running Linux, ran the *dfmax r500.5* benchmark in 142.49 seconds, and their machine, a Sun Sparcstation 10/40, ran it in 192.60 seconds; in the table, we multiplied their times by 0.74.

As the table shows, SWO generally achieves better results than IG but, also generally, takes more time. The difference between them is less than 0.5 colors on 9 of the problems. Of those with a difference of 0.5 or greater, IG does better on 4 of the graphs and SWO does better on 19 of them.

We also note, as with the scheduling work, that our first, naive implementation produced respectable results. Even without color reuse, color grabbing, or the least constraining heuristic (the first free color found was picked), SWO matched IG on 6 problems and beat it on 10. However, on half of the remaining problems IG did better by 10 or more colors.

Related work

The importance of prioritization in greedy algorithms is not a new idea. The "First Fit" algorithm for bin packing, for example, relies on placing items into bins in decreasing order of size (Garey & Johnson 1979). Another example is GRASP (Greedy Randomized Adaptive Search Procedure) (Feo & Resende 1995). GRASP differs from our approach in several ways. First, the prioritization and construction aspects are more closely coupled in GRASP. After each element is added to the solution being constructed, the remaining elements are re-evaluated by some heuristic. Thus the order in which elements are added to the solution may depend on previous decisions. Second, the order in which elements are selected in each trial is determined only by the heuristic (and randomization), so the trials are independent. There is no learning from iteration to iteration in GRASP.

Doubleback Optimization (DBO) (Crawford 1996) was to some extent the inspiration for both SWO and another similar algorithm, Abstract Local Search (ALS) (Crawford, Dalal, & Walser 1998). In designing SWO, we began by looking at DBO, because it had been extremely successful in solving a standard type of scheduling problem. However, DBO is only useful when the objective is to minimize makespan, and is also limited in the types of constraints it can handle. Because of these limitations, we began thinking about the principles behind DBO, looking for an effective generalization of that approach. DBO can, in fact, be viewed as an instance of SWO. DBO begins by performing a "right shift" on a schedule, shifting all tasks as far to the right as they can go, up to some boundary. In the resulting right-shifted schedule, the left-most tasks are, to some extent, those tasks that are most critical. This corresponds to analysis in SWO. Tasks are then removed from the right-shifted schedule, taking left-most tasks first. This ordering corresponds to the prioritization in SWO. As each task is removed, it is placed in a new schedule at the earliest possible start time, i.e., greedy construction.

Like SWO, ALS was the result of an attempt to generalize DBO. ALS views priority space (to use the terminology from SWO) as a space of "abstract schedules," and performs a local search in that space. Unlike SWO, if a prioritization is modified, and the corresponding move in solution space is downhill (away from optimal), then the modified prioritization is discarded, and the old prioritization is restored. As is usual with local search, ALS also sometimes makes random moves, in order to escape local minima.

ALS, and also List Scheduling (Pinson, Prins, & Rullier 1994), are scheduling algorithms that deal with domains that include precedence constraints on tasks. Both accommodate precedence constraints by constructing schedules left-to-right temporally. A task cannot be placed in the schedule until all of its predecessors have been placed. In order for the analysis, prioritization and construction to be appropriately coupled, it is not sufficient to simply increase the priority of a task that is late, because the constructor may not be able to place that task until after a lot of other decisions have been made. Consequently, some amount of blame must be propagated to the task's predecessors.

In contrast, our schedule constructor is able to place tasks in any order because it commits to placing a task on a specific production line, and to a relative ordering on that line, without committing to a specific start time for each task. The ability for the constructor to make decisions in "best first" order allows the analysis and prioritization to be kept very simple, but obviously complicates the constructor. Our intuition is that it is more important to keep the analysis simple, with a constructor that is able to respond flexibly to the results of prioritization. We believe this approach also has the most potential for generalization.

The commercial scheduler OPTIFLEX (Syswerda 1994) uses a genetic algorithm approach to modify a sequence of tasks, and a constraint-based schedule constructor that generates schedules from those sequences.

OPTIFLEX can also be viewed as an instance of SWO, with a genetic algorithm replacing analysis. In effect, the "analysis" instead emerges from the relative fitness of the members of the population.

Two graph coloring algorithms also bear some similarity to SWO. Impasse Class Coloration Neighborhood Search (IMPASSE) (Morgenstern 1996; Lewandowski & Condon 1996), like SWO, maintains a target set of colors and produces only feasible colorings. Given a coloring, IMPASSE places any nodes that are colored outside of the target set into an impasse set. On each iteration a node is selected from the impasse set, using a noisy degree-based heuristic, and assigned a random color from the target set. Any neighbor nodes that are now in conflict are moved to the impasse set.

Iterated Greedy (IG) (Culberson & Luo 1996), like SWO, uses a sequence of nodes to create a new coloring on each iteration, and then uses that coloring to produce a new sequence for the next iteration. The method used to generate each new sequence differs from SWO. The key observation behind IG is that if all nodes with the same color in the current solution are grouped together in the next sequence (i.e. adjacent to each other), then the next solution will be no worse than the current solution. IG achieves improvement by manipulating the order in which the groups occur in the new sequence, using several heuristics.

Conclusions and future work

Our experience has been that it is fairly straightforward to implement SWO in a new domain, because there are usually fairly obvious ways to construct greedy solutions, and to analyze a solution to assign "blame" to some of the elements. Naive implementations of SWO tend to perform reasonably well.

We have found the view of SWO as performing a "coupled search" over two different search spaces to be very informative. It has been helpful to characterize the kinds of moves that SWO makes in each of the search spaces, and the effect this has on avoiding local optima, etc. We hope that by continuing to gain a deeper understanding of what makes SWO work we will be able to say more about the effective design of SWO algorithms.

Although the ability to make large, coherent moves is a strength of the approach, it is also a weakness. SWO is poor at making small "tuning" moves in the solution space, but the coupled-search view of SWO suggests an obvious remedy. SWO could be combined with local search in the solution space, to look for improvements in the vicinity of good solutions. Similarly, making small changes to a prioritization would generally result in smaller moves in the solution space than result from going through the full analysis and reprioritization cycle. Yet another alternative is genetic algorithm techniques for "crossover" and other types of mutation to a pool of nodes, as is done in OPTIFLEX. Many hybrid approaches are possible, and we believe that the coupled-search view of SWO helps to identify some interesting strategies for combining moves of various sizes and kinds, in both search spaces, adapting dynamically to relative solution qualities.

While SWO uses fast, greedy algorithms for constructing solutions, and we have demonstrated its effectiveness on problems of realistic size, the greatest threat to the scalability of SWO is that it constructs a new solution from scratch on each iteration. An obvious solution to this problem is to develop an incremental version of SWO. The graph coloring solver, with its selective reuse of colors from the previous iteration, is a small step in this direction. It allows the constructor to avoid spending time evaluating other alternatives when the previous choice still works. More generally, it may be possible to look at the changes made to a prioritization, and modify the corresponding solution in a way that generates the same solution that would be constructed from scratch based on the new prioritization. It seems feasible that this could be done for some domains, at least for small changes to the prioritization, because there may be large portions of a solution that are unaffected.

A more interesting possibility is based on the view of SWO as performing local search plus a certain kind of propagation. A small change in priorities may correspond to a large change in the solution. For example, increasing the priority of one task in a scheduling problem may change its position in the schedule, and, as a consequence, some lower priority tasks may have to be shuffled around to accommodate that change. This is similar to what we might expect from moving the higher priority task, then propagating the effects of that change by moving lower priority tasks as well. This single move may correspond to a large number of moves in a search algorithm that only looks at local changes to the schedule, and may thus be difficult for such an algorithm to find.

Based on this view, we are investigating an algorithm we call "Priority-Limited Propagation" (PLP). With PLP, local changes are made to the solution, and then propagation is allowed to occur, subject to the current prioritization. Propagation is only allowed to occur in the direction of lower-priority elements. In effect, a small change is made, and then the consequences of that change are allowed to "ripple" through the plan. Because propagation can only occur in directions of decreasing priority, these ripples of propagation decrease in magnitude until no more propagation is possible. A new prioritization is then generated by analyzing the resulting solution. (It should be possible to do this analysis incrementally, as well.) The resulting approach is not identical to SWO, but has many of its interesting characteristics.

Another potential pitfall for SWO is that analysis, prioritization and construction must all work together to improve the quality of solutions. We have already discussed the complications that can arise when constraints are placed on the order in which the constructor can make decisions, as is the case for Line Scheduling and ALS, where construction is done strictly left-to-right. Without more complex analysis, the search spaces can effectively become uncoupled, so that changes in priority don't cause the constructor to fix problems discovered by analysis.

Another way the search can become uncoupled is related to the notion of "excess cost," discussed for the scheduling implementation. The calculation of excess cost in the analyzer turned out to be a key idea for improving the performance of SWO. However. problems sometimes have tasks that must be handled badly in order to achieve a good overall solution. One of the scheduling problems described previously has two such "sacrificial" tasks. Whenever a good solution is found, the analyzer assigns high blame to these sacrificial tasks, and the constructor handles them well on the next iteration. This means that the resulting solution is of poor overall quality, and it is not until other flaws cause other tasks to move ahead of the sacrificial tasks in the priority sequence that SWO can again, briefly, explore the space of good solutions. In such cases, to some extent the analysis is actually hurting the ability of SWO to converge on good solutions.

Ideally, we would like to generalize the notion of excess cost to recognize sacrificial tasks, and allow those tasks to be handled badly without receiving proportionate blame. For problems in which a task must be sacrificed in *all* solutions, it may be possible to use a learning mechanism that would accomplish this.

As the number of directions for future research suggests, we have only begun to scratch the surface of "Squeaky Wheel" Optimization.

Acknowledgments. The authors wish to thank Robert Stubbs of Lucent Technologies for providing the data used for the scheduling experiments. The authors also wish to thank George L. Nemhauser, Markus E. Puttlitz and Martin W. P. Savelsbergh with whom we collaborated on using SWO in a hybrid AI/OR approach. Many useful discussions came out of that collaboration, and without them we would not have had access to the Lucent problems. Markus also wrote the framework for the scheduling experiments and the TABU and IP implementations.

The authors also thank the members of CIRL, and James Crawford at i2 Technologies, for their helpful comments and suggestions. We would like to thank Andrew Parkes in particular for suggestions and insights in the graph coloring domain.

This effort was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-96-1-0335; by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreements F30602-95-1-0023 and F30602-97-1-0294; and by the National Science Foundation under grant number CDA-9625755.

Most of the work reported in this paper was done while both authors were at CIRL.

References

Clements, D.; Crawford, J.; Joslin, D.; Nemhauser, G.; Puttlitz, M.; and Savelsbergh, M. 1997. Heuristic optimization: A hybrid AI/OR approach. In *Proceedings of the Workshop on Industrial Constraint-Directed Scheduling*.

Crawford, J.; Dalal, M.; and Walser, J. 1998. Abstract local search. Unpublished.

Crawford, J. M. 1996. An approach to resource constrained project scheduling. In *Artificial Intelligence and Manufacturing Research Planning Workshop*.

Culberson, J. C., and Luo, F. 1996. Exploring the k–colorable landscape with iterated greedy. In *(Johnson & Trick 1996)*, 245–284.

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

Garey, M. R., and Johnson, D. S. 1979. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman.

Glover, F., and Laguna, M. 1997. *Tabu Search*. Kluwer.

Glover, F.; Parker, M.; and Ryan, J. 1996. Coloring by tabu branch and bound. In *(Johnson & Trick 1996)*, 285–307.

Johnson, D. S., and Trick, M. A., eds. 1996. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society.

Lewandowski, G., and Condon, A. 1996. Experiments with parallel graph coloring heuristics and applications of graph coloring. In *(Johnson & Trick 1996)*, 309–334.

Morgenstern, C. 1996. Distributed coloration neighborhood search. In *(Johnson & Trick 1996)*, 335–357.

Pinson, E.; Prins, C.; and Rullier, F. 1994. Using tabu search for solving the resource-constrained project scheduling problem. In *EURO-WG PMS 4 (EURO Working Group on Project Management and Scheduling)*, 102–106.

Selman, B.; Kautz, H. A.; and Cohen, B. 1993. Local search strategies for satisfiability testing. In *(Johnson & Trick 1996)*, 521–531.

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