

# Heuristic Optimization: A hybrid AI/OR approach

David P. Clements\*    James M. Crawford<sup>†,\*</sup>    David E. Joslin\*  
George L. Nemhauser<sup>‡</sup>    Markus E. Puttlitz<sup>‡</sup>  
Martin W. P. Savelsbergh<sup>‡</sup>

\*Computational Intelligence Research Laboratory  
University of Oregon  
Eugene, OR 97403-1269  
{clements, joslin}@cir.uoregon.edu

<sup>†</sup>i2 Technologies  
909 E. Las Colinas Blvd.  
Irving, TX 75038  
jc@i2.com

<sup>‡</sup>School of Industrial and Systems Engineering  
Georgia Institute of Technology  
Atlanta, GA 30332-0205  
{george.nemhauser, markus, martin.savelsbergh}@isye.gatech.edu

October 4, 1997

## Abstract

We have developed a hybrid architecture, H-OPT, that combines Integer Programming (IP) for global optimization, and heuristic search techniques. Our hybrid approach captures the most desirable features of each. A heuristic local search algorithm generates a large number of good feasible solutions quickly, and the IP solver is then used to combine the elements from those solutions into a better solution than the local search approach was able to find. Preliminary experimental results are very encouraging.

In developing the heuristic component for H-OPT, we have generalized several existing, highly effective scheduling algorithms. The generalization is based on two principles that we have found to be key: (1) local search benefits from the ability to make large, coherent moves in the search space, and (2) good solutions can be “taken apart” to reveal structure in the (local) search space. Our solver has served as a component of H-OPT, but is also a very good scheduling algorithm in its own right.

The techniques we have developed are very general, and should be applicable to a wide range of problems. Here we report very promising results on a scheduling problem that arises in fiber-optic cable manufacturing. The heuristic approach can generate good solutions very quickly by itself, but in combination with the global IP optimization significant further improvement is possible. The hybrid approach also produces better quality solutions than a TABU search [8] algorithm, and runs faster as well.

## 1 Introduction

Both heuristic and exact optimization techniques have been applied to difficult combinatorial optimization problems. Typically, heuristics have the advantage of speed and size of instances that can be handled, while exact methods have the advantage of quality. We present a hybrid approach that integrates heuristics and exact optimization techniques with the goal of capturing the desirable features of both.

We call our hybrid approach *heuristic optimization* (H-OPT). One component of H-OPT heuristically generates an initial set of good schedules, and the other uses an Integer Programming algorithm to globally optimize those results, producing better schedules by combining elements of the schedules in the initial set.

Our approach to heuristically generating good schedules, *“squeaky wheel” optimization* (SWO), is a generalization of several existing, highly effective scheduling algorithms, including Doubleback Optimization [3], and the patented algorithm used in Optiflex, a commercial scheduler [13]. In SWO, solutions are analyzed to provide feedback for a local search algorithm. The algorithm is designed to make large “coherent” moves in the search space, thus helping to avoid local optima without relying entirely on random moves.

In the optimization component, a linear program (LP), which is a relaxation of an IP, is solved. Each column in the LP represents a feasible solution to a subproblem; in the problems used for the experiments for this paper, each column represents a feasible schedule for a single production line in a multi-line facility. Since there are a huge number of feasible schedules for each line, it is not practical to work with the whole LP. Instead, we use a local search heuristic to generate high-quality schedules.

A branch-and-bound solver is then used to obtain “good” integer solutions to the overall problem, i.e., finding the optimal combination of columns (line schedules) from the heuristically-generated schedules. Given a set of columns, the LP solver finds optimal primal and dual solutions to the LP relaxation. In future work, the optimal dual values will be used to guide a local search algorithm that will produce new columns for the LP throughout the search. In the scheduling problem, for example, this feedback indicates which jobs are most “difficult” to schedule.

The local search heuristic generates good solutions very quickly by itself, but in combination with the IP optimization considerable improvements are obtained. The hybrid approach also produces better quality solutions than an existing TABU search algorithm, and runs faster as well.

In the next section, we present the scheduling problem, and in Section 3 we present the framework of H-OPT. In Section 4, we compare the results produced with H-OPT to those obtained by the local search heuristic and a steepest ascent/TABU search heuristic. Section 5 gives conclusions and describes work in progress.

## 2 Problem description

This section describes our formulation of a fiber-optic production line scheduling problem, derived from real data provided by the manufacturer. Instances of this problem are used in the experimental evaluation of our architecture. This is a rather generic scheduling problem so the methodology developed should be applicable to a wide variety of scheduling problems and many other logistics problems.

This multi-job, parallel machine scheduling problem with lateness and changeover costs originated in a fiber-optic cable plant. A cable consists of up to 216 *optical fibers*. The *sheathing* operation involves joining the fibers and wrapping a protective rubber sheathing around them. This operation can be performed on one of 13 parallel *sheathing lines*. Typically, the number of cables in the set of orders is much larger than the number of sheathing lines. Every ordered cable has a release time and a due date. Production cannot begin before the release time, and the objective function includes a penalty for not completing a cable by the due date.

The production lines are heterogeneous in the types of cables they are able to produce, and the speeds at which they operate. For each cable, only a subset of the production lines will be compatible, and the time required to produce the cable will depend on which of the compatible lines is selected. Job preemption is not allowed, i.e. once a cable has started processing on a line, it finishes without interruption.

We need to make two types of decisions, namely how to assign cables, hereafter called jobs, to lines and how to sequence the jobs assigned to each line. Objectives are minimization of the number of late jobs and minimization of the sum of the setup times between jobs. This is a strongly NP-hard combinatorial optimization problem.

Our overall approach is to formulate the problem as an IP and to solve it by a branch-and-bound algorithm. Hence we need a “good” IP formulation, an efficient method for solving the linear programming (LP) relaxation, and an algorithm to generate integral solutions.

One concept of modeling discrete optimization problems with complicated constraints that has been shown to work well in practice is known as *set partitioning (SP)*. Suppose we assign *schedules* to lines (rather than single jobs). Let a *line schedule* be a feasible assignment of a group of jobs to a line, including a sequencing and the associated objective cost. Notice that the computation of the objective function value of one line is independent of all other lines. To solve the problem, we need to find a min-cost subset of the set of all line schedules that uses each line at most once and includes each job in exactly one line schedule.

Let  $x_{lm}$  be the 0/1 decision variable which is 1 if line schedule  $l$  is assigned to line  $m$ . Associated with this variable will be a column  $a_{lm}$  representing:

- A set of jobs assigned to line  $m$ , represented by 0/1 indicators  $a_{lm}^j$ , which are equal to 1 if job  $j$  is in line schedule  $l$  and 0 otherwise. Column  $a_{lm} = \{a_{lm}^j\}$  will then be the characteristic vector of the jobs in line schedule  $l$  for line  $m$ .
- An ordering of that set resulting in a cost  $c_{lm}$  associated with that line schedule. For a given set of jobs, we would ideally like to find a line schedule that minimizes  $c_{lm}$ , but solving this problem is NP-hard, and in practice we usually must apply heuristic methods.

This leads to the SP problem

$$\begin{aligned} & \text{Minimize} && \sum_{m \in M} \sum_{l \in L_m} c_{lm} x_{lm} \\ & \text{subject to} && \sum_{m \in M} \sum_{l \in L_m} a_{lm}^j x_{lm} = 1 && \forall j \in J \end{aligned}$$

$$\sum_{l \in L_m} x_{lm} \leq 1 \quad \forall m \in M$$

$$x_{lm} \in \{0, 1\} \quad \forall l \in \bigcup L_m, m \in M$$

where  $L_m$  is the set of feasible line schedules for line  $m$ ,  $J$  is the set of jobs, and  $M$  is the set of available production lines.

The SP formulation comprises two types of constraints. The first forces the solution to the scheduling problem to include each job exactly once. The second makes sure that for each line at most one line schedule can be part of the solution. Note that the constraints that determine whether or not a line schedule is feasible are not represented in the SP formulation; since only feasible line schedules are generated by the heuristic solver, the SP formulation does not need to take these constraints into account.

Although fairly large instances of SP problems can be solved efficiently, the algorithmic challenge is to devise methods for solving SPs with a huge number of columns. In our scheduling problem, the SP has a column for every possible line schedule for every line. The number of such columns is generally exponential in the number of jobs. Fortunately, as explained below, it is possible to approximate the SP so that only a relatively small number of line schedules are considered.

### 3 Description of H-OPT framework

#### 3.1 Scheduling by local search

Some of the most effective approaches for solving systems of constraints in recent years have been based on local search. GSAT [12] and WSAT [11] apply local search techniques to SAT solvers, and WSAT has been used as the solver for the SATPLAN [9] planning system. CIRL's scheduling technology uses *Doubleback Optimization*, which performs a kind of local search to improve a "seed" schedule over a number of iterations [3]. The commercially successful scheduler OPTIFLEX from *i2 Technologies* is based on a patented approach that uses genetic algorithms [13]. Although these approaches differ substantially in the details, there has been a clear movement toward the use of local search in AI approaches to optimization problems.

In designing our local search algorithm, we began by looking at the Doubleback algorithm, because it had been extremely successful in solving a standard type of scheduling problem. (On one benchmark related to aircraft manufacture [6], CIRL's scheduler produces the best-known solutions by a substantial margin, and finds them faster than the closest competitors.) However, the Doubleback algorithm is only useful when the objective is to minimize makespan. The problem domain we wanted to use to test the hybrid architecture required a different objective function, using a weighted sum of several factors. The problems also used constraints that are more complex than could be handled by the current Doubleback algorithm. Because of this, we began thinking about the principles behind Doubleback, looking for an effective generalization of that approach.

The architecture that emerged has three components:

**Prioritizer** Generates a sequence of jobs, with higher "priority" jobs being earlier in the sequence. Uses feedback from the Analyzer to modify previously generated sequences.

**Constructor** Given a sequence of jobs, constructs a schedule. Uses “greedy” scheduling for each job, in the order they occur in the sequence, without backtracking.

**Analyzer** Given a schedule, analyzes that schedule to find the “trouble spots.” This feedback is provided to the Prioritizer.

We call this architecture “Squeaky Wheel” Optimization (SWO), from the aphorism “The squeaky wheel gets the grease.” The picture is that on each iteration, the Analyzer determines which jobs are causing the most trouble in the current schedule, and the Prioritizer ensures that the Constructor gives more attention to those jobs on the next iteration.

In the current implementation, the Analyzer “assigns blame” to each of the jobs in the current schedule. For each job we calculate the minimum possible cost that each job could contribute to any schedule. For example, if a job 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. The minimum possible setup costs are also included. Then, for a given a schedule, the penalty assigned to each job is its “excess cost,” the difference between its actual cost and its minimum possible cost. The setup time penalty for each pair of adjacent jobs is shared between the two jobs, and the penalty for lateness is charged only to the late job itself.

Once these penalties have been assigned, the Prioritizer modifies the previous sequence of jobs by moving jobs with high penalties forward in the sequence. We currently move jobs forward in the sequence a distance that increases with the magnitude of the penalty, such that to move from the back of the sequence to the front, a job must have a high penalty over several iterations. (Sorting the jobs by their assigned penalty is simpler, and turns out to be almost as effective.) As a job moves forward in the sequence, its penalty will tend to decrease, and if it decreases sufficiently the job may then tend to drift back down the sequence as other jobs are moved ahead of it. If it sinks too far down the sequence, of course, its penalty may tend to increase, resulting in a forward move.

The Constructor builds a schedule by adding jobs one at a time, in the order they occur in the sequence. A job is added by selecting a line, and a position relative to the jobs already in that line. A job may be inserted between any two jobs already in the line, or at the beginning or end of that schedule, but changes to the relative positions of the jobs already in the line are not considered. Each job in the line is then assigned to its earliest possible start time, subject to the ordering, i.e., a job starts at the minimum of either its release time, or immediately after the previous job on that line, with the appropriate setup time between them.

For each of the possible insertion points in the schedule, relative to the jobs already in each line, the Constructor calculates the effect on the objective function, and the job is placed at the best-scoring location. Ties are broken randomly. After all of the jobs in the sequence have been placed, the Constructor tries to improve on the completed schedule with a small amount of local search. Currently, we only consider reordering jobs within a line.

The design of the local search architecture was influenced by two key insights:

- *Good solutions can reveal problem structure.* By analyzing a good solution, we can often identify elements of that solution that work well, and elements that work

poorly. A resource that is used at full capacity, for example, may represent a bottleneck. This information about problem structure is local, in the sense that it may only apply to some part of the search space currently under examination, but may be extremely useful in helping figure out what direction the search should go next.

- *Local search can benefit from the ability to make large, coherent moves.* It is well known that local search techniques tend to become trapped in local optima, from which it may take a large number of moves to escape. Random moves are a partial remedy, and in addition, most local search algorithms periodically just start over with a new random assignment. While random moves, small or large, are helpful, we believe our architecture works, in part, because of its ability to also make large *coherent* moves. A small change in the sequence of tasks generated by the Prioritizer may correspond to a large change in the corresponding schedule generated by the Constructor. Exchanging the positions of two tasks in the sequence given to the Constructor may change the positions of those two tasks in the schedule, and, in addition, allow some of the lower priority tasks, later in the sequence, to be shuffled around to accommodate those changes. This 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 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.

This architecture can be thought of as searching two coupled spaces: the space of solutions, and the space of job sequences. Note that in the space of job sequences, the only “local optima” are those in which all jobs are assigned the same penalty, which in practice does not occur. Because of this, the architecture tends to avoid getting trapped in local optima in the solutions generated by the Constructor, since analysis and prioritization will always (in practice) suggest changes in the sequence, thus changing the solution generated on the next iteration. The randomization used in tie breaking will also tend to help avoid local optima.

Note that this architecture is a general framework, and not itself a specific algorithm. Doubleback can be viewed as an instance of this architecture, for example [3]. The OPTIFLEX scheduler [13] can also be viewed as an instance, with a genetic algorithm replacing the analysis phase. (In effect, the “analysis” instead emerges from the relative fitness of the members of the population.) These schedulers may appear to have little in common, but we believe that we have uncovered some principles that underlie both of these approaches. In the case of the OPTIFLEX scheduler, for example, we hypothesize that it is not genetic algorithms *per se* that make the scheduler so effective, but rather the manner in which prioritization and greedy construction are combined.

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 [7]. Another example would be GRASP (Greedy Randomized Adaptive Search Procedure) [5]. GRASP differs from our approach in several ways. First, the prioritization and construction aspects are more closely coupled in GRASP. After each element (here, a task) is added to the solution being constructed (here, a schedule), 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. GRASP has no mechanism analogous to the dynamic prioritization used in SWO, and consequently lacks the ability to search the space of sequences of elements, which we believe to be a key aspect of our architecture because it allows the local search to make large, coherent moves.

Our architecture has been applied to both the single-line subproblem of generating new columns for an IP solver, and to the solution of the full scheduling problem. We currently focus on the use of this approach for solving the full problem; these solutions are also used to generate the initial set of columns for an LP/IP solver.

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 job in a line that has more “slack,” all other things being equal.

### 3.2 Solving the set partitioning problem

The heuristic local search algorithm generates as many good schedules as it can within a specified time limit. Each of these schedules contains one line schedule for each production line. Each individual line schedule becomes a column in the LP/IP formulation of a set partitioning problem, as previously discussed. A branch-and-bound solver is then used to find the optimal combination of columns. The solver used in our implementation is MINTO [10], a general purpose mixed integer optimizer that can be customized to exploit special problem structure through application functions. Master LPs are solved using CPLEX [2], a general purpose LP solver. The heuristic problem solver provides an initial set of columns to the LP relaxation at the root node so that a feasible solution to the LP relaxation is assured.

### 3.3 Column generation

Although we currently use MINTO only for the re-combination of columns generated by SWO, the H-OPT architecture also allows for introduction of new columns in response to the solutions produced by MINTO.

To solve the LP relaxation of SP, called the *master problem*, we use Dantzig-Wolfe column generation [4], which means that the LP is solved with all of its rows, but not all of its columns present. The LP relaxation of SP including only a subset of the columns is called the *restricted master problem*. There are two ways in which columns can be introduced into the LP. Currently we only use one – providing an initial set of columns to the LP. This resulting LP is then solved to optimality. In future work we will also use the second method, incrementally adding new columns to the LP based on feedback from the LP solver. The same heuristic techniques that are currently used to generate the initial seed columns will be applied to the problem of generating additional columns. The problem of generating new columns from a current optimal LP solution is called the subproblem or pricing problem. Note that the column generation subproblem inherits only some of the difficulty of the full problem, making it easier to solve than the master problem.

Given an optimal solution to the current LP restricted master problem, the *dual price* for each job  $j$  determines how expensive it is for the current solution to cover job  $j$ . The

idea of column generation is to use these dual prices to determine which jobs should be included in “good” new columns for the master problem. For each line, we need to solve a different subproblem. The subproblems differ because the lines are compatible with different subsets of the jobs, and because the time to complete a job depends on which line it runs on.

When solving subproblems, we must evaluate or “price” candidate columns. Only candidate columns corresponding to feasible line schedules need to be evaluated, i.e. we require that all jobs included can run on that line. If we find a column whose cost  $c_{lm}$  is smaller than the sum of the dual prices of the jobs covered by that column, it is a candidate to enter the master problem. We say that such a column has a *negative reduced cost*. If no such column exists for any of the lines, the current solution to the master problem is optimal.

If we could solve the subproblems to optimality quickly, we would then have a fast LP solver which could then be embedded into a branch-and-bound algorithm for solving the entire scheduling problem. Unfortunately, the subproblems, while being easier than the original problem, are still NP-hard and a very large number of them need to be solved. Therefore we will solve the subproblems heuristically and stop generating columns when our heuristic terminates. This implies that the best solution to the LP master problem may not give a true lower bound. Thus, by using this approximate lower bound in the branch-and-bound phase of the algorithm we cannot guarantee that an optimal solution will be found to the entire scheduling problem.

## 4 Experimental results

We have several sets of test data, ranging in size from 40 to 297 tasks. In each problem there are 13 production lines. We compare the following solution methods:

**TABU** Uses TABU search, a local search algorithm in which moves that increase the cost are permitted to avoid getting trapped at a local optimum. To avoid cycling, when an “uphill” move is made, it is not allowed to be immediately undone.

**SWO** Applies the SWO architecture to the entire problem, running for a fixed number of iterations and returning the best schedule it finds.

**H-OPT** Uses the best schedules generated by SWO as the set of initial columns in the IP formulation described previously. Note that in these preliminary experiments no additional columns were generated during the solution process.

On the 297-task problem, our implementation of TABU was much less effective than either SWO or H-OPT, failing to find a feasible schedule after running for over 24 hours. On the smaller problems, TABU was able to find solutions, but both SWO and H-OPT outperformed TABU by a substantial margin.

Table 1 presents results for SWO and H-OPT on test sets with the number of jobs ranging from 40 to 297. In each case, ten trials were run and the results averaged. The second column of the table shows the best objective function value we have ever observed on each problem. The next two columns show the average objective function value, and the average time required for SWO. For H-OPT, the last three columns in the table show the average objective function value, the average time required by MINTO to

Data Set	Best Obj	SWO		H-OPT		
		Avg Obj	Avg Time	Avg Obj	Extra Time	% Impr.
40	1890	1890.0	162	1890.0	3	0.000
50	3101	3128.8	201	3127.2	3	0.051
60	2580	2580.6	242	2580.0	9	0.023
70	2713	2714.2	282	2713.0	5	0.044
148	8869	8951.7	604	8874.7	31	0.858
297	17511	17806.8	1209	17556.1	111	1.406

Table 1: Experimental results

	Time allowed for SWO (seconds)					
	60	300	600	900	1200	1800
SWO	17979.4	17881.9	17818.1	17858.0	17806.8	17777.0
H-OPT	17715.0	17581.5	17553.8	17547.4	17556.1	17543.1
% impr.	1.460	1.678	1.482	1.738	1.406	1.315
Avg. MINTO time	11	49	125	121	111	110

Table 2: Experimental results (297 task problem)

optimize over the set of columns in the schedules generated by SWO, and the percentage improvement in the objective function. All times are in user processor seconds. These experiments were run on a Sparcstation 10 Model 50.

For the SWO/H-OPT experiments, we allowed the heuristic solver (SWO) to generate solutions up to a time limit proportional to the number of jobs, with approximately 20 minutes (1200 seconds) allowed for the largest problem (297 tasks). The line schedules from these solutions then formed the initial set of columns of H-OPT. H-OPT searches for a better combination of those columns. In other words, if it finds an improvement, it is the result of using columns from different schedules generated by SWO.

On the smallest problems in our test set, SWO by itself finds solutions that are as good as the best solutions we have found by any method. (In fact, it finds them, on average, in under 10 seconds.) As the problem size increases, H-OPT is able to show greater improvements by re-combining columns from the schedules found by SWO, with an improvement of 1.4% over the performance of SWO alone on the 297-job problem.

To further characterize the performance of H-OPT, we ran the largest data set (297 jobs) with varying amounts of time allowed for SWO to generate the “seed” schedules. Each column in the table represents ten runs with a fixed amount of time allowed for seed generation. As shown in Table 2, MINTO required only a relatively small amount of time to improve upon the schedules produced by SWO. The degree of improvement ranged from 1.3% to 1.7%, corresponding to causing an additional 2 to 3 jobs to be completed by their due dates. (The actual improvement, of course, may be a combination of reducing lateness and reducing the total setup time.)

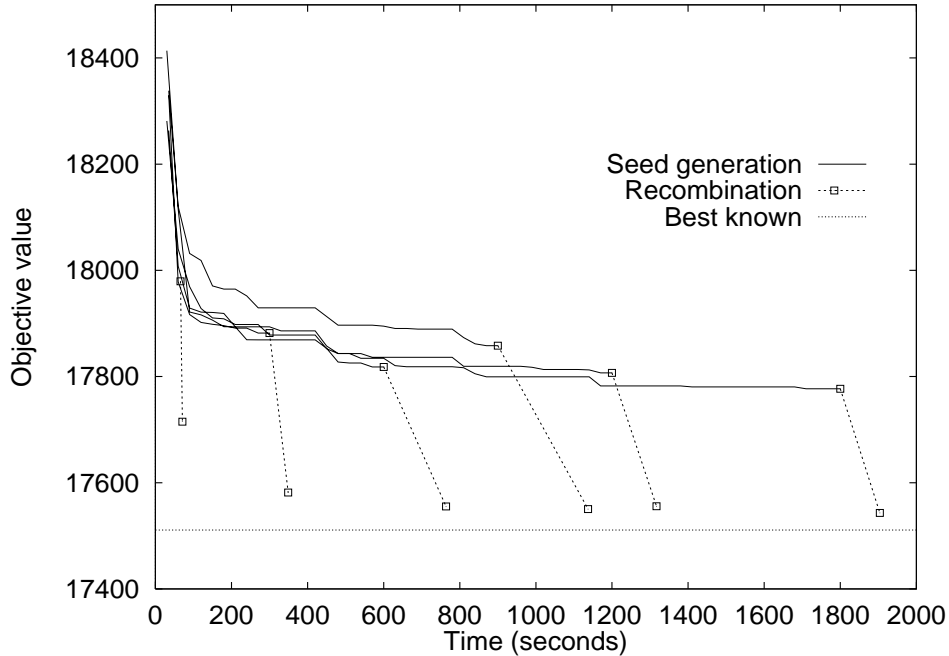


Figure 1: Time vs. objective function (297 task problem)

Figure 1 gives another view of the same data. For each of the six experiments shown in Table 2, the solid line shows the best schedule produced by SWO, on average, versus time. The dashed line segments show the results of taking the set of schedules generated by SWO up to some point (1 minute, 5 minutes, etc., up to 30 minutes) and allowing MINTO to optimize over that set of schedules. The horizontal line shows the best objective value we have ever observed, for reference.

As more time is allowed for SWO to search, better schedules are found, but with diminishing returns. On the other hand, until SWO has been allowed to run long enough to have produced a sufficient number of “good” schedules, MINTO does not have enough to work with. This suggests that H-OPT should take a dynamic approach to the boundary between SWO and MINTO. For example, SWO might be allowed to generate schedules, keeping a set of the best  $N$  schedules found, until  $Z$  consecutive iterations go by without any change to that set, for empirically determined  $N$  and  $Z$ . On the 297 task problem, H-OPT could have made the transition from SWO to MINTO as early as 300 seconds, with only a relatively small penalty.

Although the improvements achieved by MINTO are relatively small, on the order of 1.5%, MINTO achieves this improvement quickly, and SWO is unable to achieve the same degree of optimization even when given substantially more time. In cases where it is important to optimize as much as possible, and to do so quickly, the H-OPT combination of local search and global IP optimization can be highly effective. Although this is only a small experiment, we believe that these results clearly indicate that the approach is promising and should be explored further by both enhancing the techniques and the

scope of applications.

## 5 Conclusions and future work

Although our results so far are limited in scope, they are very encouraging. As might be expected based on other successful applications, our local search approach is capable of generating high-quality schedules very quickly. Allowing additional time does not improve those initial results dramatically. However, a combined exact and heuristic optimization approach allows a large number of good (and not so good) schedules to be “taken apart and recombined” in a way that quickly results in a higher quality schedule.

This hybrid approach takes advantage of the relative strengths of each part: local search is able to find good schedules, but tends to get stuck in local optima, and IP techniques provide a kind of global optimization that has no counterpart in local search. In a given solution, the local search approach may get some things right, and some things wrong, but the parts that are handled badly in one solution may be handled well in another. In a sense, global optimization allows the best parts of the different solutions to be combined.

The SWO architecture is itself innovative, and there is still a great deal of room for improvement. We are looking at more sophisticated methods of analysis and construction, and also looking at other domains that would require us to further generalize the approach taken here.

The use of randomness in integer programming algorithms has received little attention, and we believe the randomness introduced by SWO (in random tie-breaking during the construction of solutions) is partially responsible for the success of H-OPT. In future experiments, we hope to explore further the incorporation of randomization techniques. Within SWO, we can allow randomization to occur in any of the three main modules, and experiments are currently underway to try to understand the impact of randomization on the various parts of this architecture. In future work we will also experiment with a branch-and-price algorithm [1] in which randomized heuristics are used to generate new columns throughout the search tree.

Applying local search techniques for column generation may improve performance still further. The intuition behind this is that the dual values provided by the LP solver may provide valuable feedback to the local search engine about which tasks are “most critical.” This can provide a bias toward different (and hopefully useful) areas of the search space. We have tried to use the dual values provided by MINTO to assist in the prioritization of jobs for SWO, but our results so far have been mixed. We hope in the future to better understand this mode of interaction between the two parts of H-OPT, and to show that the feedback that MINTO can provide can be very useful to SWO.

**Acknowledgements.** The authors wish to thank Robert Stubbs of Lucent Technologies for providing us with data to use for our experiments.

This effort was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-96-1-0335 (CIRL and Georgia Tech); 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 (CIRL); and by the National Science Foundation under grant numbers

CDA-9625755 (CIRL) and DMI-9700285 (Georgia Tech).

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, the Air Force Office of Scientific Research, the National Science Foundation, or the U.S. Government.

## References

- [1] C. Barnhart, E. Johnson, G. Nemhauser, M. Savelsbergh, and P. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 1996.
- [2] CPLEX Optimization, Inc. Using the CPLEX callable library and CPLEX mixed integer library, version 4.0, 1996.
- [3] J. M. Crawford. An approach to resource constrained project scheduling. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, 1996.
- [4] G. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.
- [5] T. A. Feo and M. G. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.
- [6] B. R. Fox and M. Ringer. Planning and scheduling benchmarks, 1995. <http://www.NeoSoft.com/benchmrx/>.
- [7] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman, 1979.
- [8] F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.
- [9] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, Portland, OR, 1996.
- [10] G. Nemhauser, M. Savelsbergh, and G. Sigismondi. Minto, a mixed integer optimizer. *Operations Research Letters*, 15:47–58, 1994.
- [11] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Second DIMACS challenge workshop on cliques, coloring, and satisfiability*, Rutgers University, October 1993.
- [12] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 440–446, 1992.
- [13] G. P. Syswerda. Generation of schedules using a genetic procedure, 1994. U.S. Patent number 5,319,781.