In Proceedings of the AIPS-98 Workshop on Planning as Combinatorial Search. In Conjunction with AIPS-98, Pittsburgh 1998.

Abstract Local Search

James M. Crawford
i2 Technologies
909 East Las Colinas Boulevard
Irving, TX 75039
jc@i2.com

Mukesh Dalal
i2 Technologies
909 East Las Colinas Boulevard
Irving, TX 75039
mdalal@i2.com

Joachim P. Walser
Programming Systems Lab
Univ. des Saarlandes, Postfach 151150
66041 Saarbrücken, Germany
walser@ps.uni-sb.de

Introduction

Consider a simple problem of sequencing a set of tasks on a resource. Assume that each task has a deadline, and the objective is to schedule each task so that it ends by it's deadline. One way to view this problem is as a search in the space of start times. Under this view we have a simple constrained optimization problem in which the variables are the start times, the constraint is that no tasks can overlap, and the objective is to minimize lateness.

A different way to view the problem is to abstract away the start times, and consider just the ordering of the tasks on the resource. Given any total ordering on the tasks, an optimal schedule consistent with the ordering can be obtained in linear time by scheduling each task, in order, as early as possible. Since each globally optimal schedule can be created from its task ordering, the sequencing problem can then be solved by searching in the space of task orderings. This space is much smaller than the space of start times, since a large number of obviously sub-optimal schedules, exactly those with some gaps between some adjacent tasks, are not even represented.

This change of representation brings the underlying search and optimization problem more clearly into focus. There is one resource and all tasks are competing for it. The task ordering is essentially a priorization of the tasks. Tasks then draw from the resource in the priority order to generate a schedule.

The general principle here is that we can decouple an algorithm for solving combinatorial optimization problems into two parts: a priority generation algorithm and a greedy solution builder. For most combinatorial optimization algorithms one can write greedy algorithms that do a reasonable job much of the time. They usually fail because they are too greedy: e.g. they allow an early task to take a resource that turns out to be critical to a later task. Intuitively, if we had our priority order just right, a greedy solver would generate an optimal solution. The essential idea behind abstract local search (and other recent work such as that of (Clements *et al.* 1997)) is that we can iterate between using priorities as the input to a greedy solution builder, and using the proposed solution to intelligently update the priorities.

There is another, less obvious, advantage to the move to priority space: it is more suitable to a local search (Aarts & Lenstra 1997) than the space of start times. For example, a small change in the start time of a task can generate multiple hard constraint violations (that is, overlaps) that the local search then has to somehow weigh against soft constraint violations (that is, lateness). Since all hard constraints are automatically enforced in the optimal schedules corresponding to the task orderings, they can be simply evaluated by considering only the soft constraint violations.

Abstract local search (ALS) solves combinatorial optimization problems (Papadimitriou & Steiglitz 1982) by making local moves in the space of abstract solutions. An abstract solution (for example, a task ordering) is mapped to a concrete solution (for example, a schedule) by a greedy solution builder that, generally, enforces all hard constraints. The concrete solution is then evaluated to determine flaws, usually by measuring soft constraint violations. The flaws in the concrete solution are used to generate modifications (moves) in the abstract solution, that might reduce the flaws in the concrete solution.

A key idea in this architecture, which is illustrated in Figure 1, is that flaws are detected in concrete solutions but modifications are made in abstract solutions.

Detecting flaws in concrete solutions and using them to drive modifications has been shown to be effective in several local search applications (for example, GSAT for propositional satisfiability problems (Selman, Kautz, & Cohen 1993)). Minton *et al.*'s *informedness hypothesis* attributes the effectiveness of iterative repair to making use of important information about the current solution to guide the search (Minton *et al.* 1990). ALS operates in this spirit by using concrete solutions to make information apparent about concrete flaws to guide moves in the abstract space.

Modifications in abstract solutions are motivated by (i) the smaller size of the abstract search space, and (ii) its greater suitability for local search when the concrete solutions share an intricate structure that is difficult to maintain by local moves in concrete space. There are several requirements for abstract local search to work well:

Tractable builder: There should be a fast algorithm that maps any abstract solution to a feasible concrete solution (that is, a concrete solution with no hard constraint violations).

Optimality-preserving abstraction: For any concrete solution S there is some abstract solution that maps to a

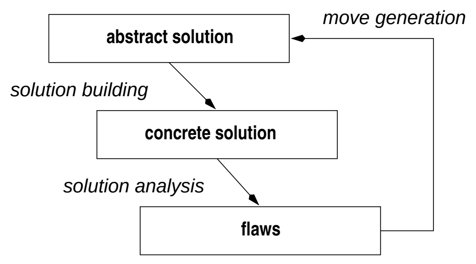

Figure 1: ALS architecture for combintorial optimization

concrete solution that is at least as good as S. Without this property, it is clear that abstract local search can not reach optimality.

Tractable analysis: There should be a fast algorithm that identifies flaws in the concrete solution, and maps them to possible modifications in the abstract solution.

The reasons for these requirements should be clear. However, as we will see, they may be relaxed in practical applications.

Background

Combinatorial optimization problems (Papadimitriou & Steiglitz 1982) generally consist of a set of decisions that must be made subject to a collection of constraints, and a goal function that evaluates candidate solutions for their "quality." For example, we might have a set of tasks whose start times are unknown (the decisions to be made) but which must satisfy the constraint that certain tasks must precede others (e.g. you have to sand the table before you paint it). The goal function might be to minimize the total cost of the proposed schedule.

Goal functions are sometimes expressed as *soft constraints*, that is, constraints that do not necessarily need to be enforced, but if they are violated then some penalty is incurred. In this kind of encoding, the optimization criterion is generally to minimize the sum of the penalties.

Many important combinatorial optimization problems are NP-hard(Garey & Johnson 1979). Intuitively, this is because all known methods for guaranteeing optimality are asymptotically equivalent in the worst case to an exhaustive enumeration of the space of all possible sets of decisions.

Local Search

Local search has been successfully used in solving many difficult combinatorial optimization problems, including satisfiability, planning, and scheduling (Aarts & Lenstra 1997). The essential idea is to start with some initial state and iteratively modify the current state by making a promising local move, until a final solution is obtained. A local move makes only a small change in the current state, for example, by flipping the truth-value of a single variable, or by offloading a task from one resource to another. An internal evaluation criterion based on feasibility and the external optimization criterion is used to determine the best among several possible moves. An analysis of some important flaw, that is, sub-optimality or infeasibility, in the current state is used to generate moves that might rectify the flaw. Some diversification technique, for example, heating in simulated annealing, is generally used to avoid getting trapped in local optima, for example, by allowing low probability moves that lead to less optimal states. Finally, most local search implementations restart several times to further reduce the effect of local optima.

Scheduling

The scheduling problem consists of a set of tasks $1, \ldots, n$ to be scheduled subject to a collection of constraints. A solution is a schedule giving the start time for each task (Garey & Johnson 1979).

Each task is associated with a processing time indicating the duration of the task. The constraints are usually *sequencing restrictions*, *resource capacity constraints*, and *ready times and deadlines* (Slowinski & Weglarz 1989). A sequencing restriction might state that task i must complete before j can begin. A resource capacity constraint states that tasks i and j conflict (usually because both require the same resource) and thus cannot be scheduled in parallel. A ready time might state the earliest time at which task i can start. A deadline is the time by which task i should be completed.

Depending on the application, resource capacities, ready time, and deadlines, can each be either hard or soft constraints. However, the most common case, and the case we focus on here, is where capacities and ready times are hard constraints and the deadline is a soft constraint.

Abstract Local Search for Scheduling

Scheduling is, in a sense, a generalization of the simple sequencing problem discussed in the introduction. Tasks must be assigned start times subject to capacity and ordering constraints. Since the objective is to minimize lateness, the natural thing to do is to order the tasks, and schedule each task, in order, as early as possible subject to the hard constraints. Both of the abstract local search (ALS) algorithms discussed below are variants on this basic idea.

While discussing ALS for scheduling problems, we will often use terms like abstract schedule, concrete schedule, and schedule builder, where "schedule" replaces the more general "solution." Further, we will often omit "concrete" from concrete schedule.

ALS Using Priority Vectors

A priority vector p maps each task i to an integer $p(i)$ that intuitively represents the global "importance" of the task. Any

```
while some task remains unscheduled do
    $t =$ highest priority enabled but unscheduled task;
    schedule $t$ as early as possible subject
            to hard constraints;
end
```

Figure 2: SB(PV): Priority-vector based schedule builder

such priority vector can be mapped to a schedule using the simple schedule builder SB(PV) given in Figure 2, where a task is considered *enabled* if and only if all of its predecessors have been scheduled.

The priority-vector approach to scheduling satisfies all the conditions discussed in the introduction. The schedule builder SB(PV) is clearly tractable, since it builds a feasible schedule in $O(n^2)$ time, where n in the number of tasks (we conjecture that the expected run time of SB(PV) would be more like $O(n\log(n))$ but a formal proof of this is beyond the scope of this paper).

Theorem 1 shows that the priority vector abstraction is an optimization preserving abstraction.

Theorem 1: For any scheduling problem, and any schedule s, there is some priority vector such that SB(PV) produces a schedule with total lateness less than or equal to that of s.

Finally, analysis of a schedule can be done in a variety of ways (Pinson, Prins, & Rullier 1994). For our implementation, we used a technique we refer to as *general critical path analysis*. If a task is late then its priority is increased by an amount b that is calculated based on how late the task is. Whenever we increase the priority on a task, if that task could not be scheduled any earlier because of a precedence relationship then the priority of that predecessor is increased by b. If the task could not be scheduled earlier because of a resource contention, then we increase the priority of all tasks using that resource at that time by $b/2$. Both of these rules are applied recursively (until the priority increment becomes negligible). The analysis calls the function *assignBlame*(t, b), shown in Figure 3, for each task t that is late by b days.

The schedule-build-analyze cycle is shown in Figure 4. This is basically a vanilla iterative improvement local search enriched by an intensification strategy (Glover & Laguna 1993). Note that we assign blame for all due-date violations. The thinking behind this is that running the schedule builder on large problems is relatively expensive – at least compared to flips in SAT problems – and we want to leverage the analysis as much as possible. An interesting variant of this (and more in the spirit of GSAT) would be to assign blame for just one due-date violation at a time and then rebuild the schedule. A steepest-descent variant (perhaps computationally too expensive for very large problems) would be to consider several ways to resolve each due-date violation, and evaluate each.

```
proc assignBlame(task $t$, int $b$)
  increase priority of $t$ by $b$ units;
  if $t$ was late because of a precedence relation
    with task $t_2$ then assignBlame$(t_2, b)$;
  else   // $t$ was late because of a resource contention
    foreach task $t_2$ using the same resource as $t$:
      assignBlame$(t_2, b/2)$;
  end
end
```

Figure 3: Priority vector based schedule analysis

```
for $i = 1$ to max-restarts do
  initialize priorities;
  for $i = 1$ to max-iterations do
    build schedule for current set of priorities;
    if this gives a new best schedule then save it;
      with prob. $p$ return to best schedule ever seen;
      foreach late task $t$: assignBlame$(t$, days late$)$;
```

Figure 4: ALS using priority vectors, top-level control loop

The final piece of the puzzle is the initial assignment of priorities. On large scheduling problems it turns out to be worth the trouble to make sure that we start from a reasonably good initial priority vector in order to make the most of our relatively limited computational resources. In the examples we have looked at to date, each task can be uniquely associated with a "delivery" task that has a deadline (e.g. We never have a situation in which $t1$ has deadline $d1$ and must preceded $t2$ that has deadline $d2$. We also never have $t1$ preceding $t2$ and $t3$ each of which has a deadline.) We thus take the initial priority of each task to be the arithmetic inverse of the due date of the corresponding delivery task (i.e. tasks with early deadlines are given the highest priorities). This turns out to be a surprisingly good first-cut heuristic.

This approach to scheduling has been implemented and run on problems involving over 30.000 tasks. We discuss the results in a separate section.

ALS using Priority Graphs

A priority graph is a directed acyclic graph whose nodes represent tasks and arcs represent priorities: an arc from A to B indicates that task A has higher priority than task B, that is, the schedule builder should schedule task A before scheduling task B (unless sequencing restrictions require that task B must complete before task A).

Priority graphs represent a somewhat different kind of information than priority vectors. While the numbers in vectors can encode relative strengths in priorities, the information in graphs is purely relational, e.g. task A has higher priority than task B. (Although it is possible to label the arcs by numbers indicating their relative strengths, this extension of priority graphs is beyond the scope of this paper.) Another difference is that while vectors force a decision on relative priorities between all pairs of tasks, graphs do not have to commit to these extraneous priorities that are not motivated by the analysis. Thus, ALS using priority graphs has the flexibility of allowing more decisions to be made by the schedule builder. This could lead to more effective use of sophisticated schedule builders that would otherwise be unnecessarily constrained by priority vectors.

Priority graphs are similar to disjunctive graphs (Balas 1969) which have been applied for several complete and local re-optimization strategies for scheduling. In contrast with previous work, changes in the priority graph are interlaced with domain-specific greedy scheduling.

Greedy Scheduling. The schedule builder starts with an empty schedule and keeps scheduling tasks one at a time until the schedule is complete. The next task to be scheduled is selected from the enabled tasks, using a customizable *task dispatching criterion* that uses a prioritized sequence of heuristics to filter all available tasks. The selected task is scheduled using a customizable *task scheduling criterion* that uses a prioritized sequence of constraints to guarantee that hard constraints are satisfied.

A variety of dispatching criteria have been reported in (Lawrence 1984). We have implemented the EST/EFT combination of criteria for task dispatching: (i) select the task that can start the earliest, and (ii) among those, select the task that can finish the earliest.

We have implemented the following task scheduling criterion: schedule the selected task on the first least-constrained resource among those that can start at the earliest possible time, without violating any capacity constraints. This provides an efficient $O(n^2)$ time schedule builder.

Schedule Analysis. The schedule analyzer selects the most late task for flaw analysis. It determines all possible direct causes for the delay, and suggests changes in the priority graph to offset those causes. It constructs a *lateness DAG* (directed acyclic graph) whose nodes consist of the late task as well as all other tasks that could have contributed to the lateness. ALS randomly selects a set of moves and evaluates each move by constructing new abstract and concrete schedules. The most promising pair of schedules is used to start the next iteration of local search. Meta-heuristic techniques like tabu memory (Glover & Laguna 1993), are used in a straightforward manner to improve the search.

Experimental Results

This section reports on experimental results on a class of scheduling problems which arise in the domain of supply chain planning. The problem under consideration is an extension of *Resource Constrained Project Scheduling* (RCPS) (Slowinski & Weglarz 1989). The problem is parameterized as follows. Given are n tasks and r renewable resources. Resource k has constant capacity R_k, task t_i has a duration of p_i during which r_{ik} units of resource k are occupied (no preemption is allowed). Additionally, sequencing restrictions (precedences) between tasks must be obeyed. The main extension with respect to the standard RCPS class are ready-times and due-dates for tasks. As a consequence, the objective is to minimize a measure of the overall lateness rather than *makespan* (the length of time from the start of the first task to the completion of the last task). The goal is to assign each task a start time s_i such that all precedence and capacity constraints are met and the total lateness is minimized.

Problem instances of this extended RCPS class are typically much larger than classical scheduling instances (tens of thousands of tasks instead of on the order of one hundred tasks reported in the RCPS literature). This is a challenge as algorithms for scheduling (e.g. job shop scheduling) often fail to scale up to this problem size.

We are limited by concerns for company propriety in what we can say about the applications, but we can give limited results for one real-world problem in semiconductor manufacturing. This problem involves over 30.000 tasks that must be scheduled on more than 20 resources.

Algorithm	Iterations	Restarts	Lateness	Runtime
DBO	NA	NA	4710 d	9 s
SB(PV)	NA	NA	3930 d	6 s
ALS^1	200	0	3663 d	93 s
ALS^2	200	50	3600 d	4600 s

Table 1: Experimental results. Reported runtimes of double-back optimization DBO, greedy scheduling SB(PV), and ALS. The table reports number of iterations and restarts; total lateness (in days) and runtimes in seconds CPU time, averaged over 50 runs for ALS^1, and 2 runs for ALS^2.

Experimental results are given in Table 1. "DBO" refers to the double-back optimization (Crawford 1996). Double-back optimization works well on resource-constrained project scheduling – where the objective is to minimize makespan. However, DBO does not seem to work as well on problems with multiple deadlines where the objective is to minimize total lateness. The reason for this is not yet clear. SB(PV) is the result of running just the scheduler builder with the initial heuristics SB(PV) of Figure 2. 'ALS' is the version of ALS with priority vectors. There are two key tuning parameters for this version of ALS: the number of restarts, and the probability that we accept moves that do not improve the quality of the schedule. The number of iterations and the number of restarts were varied as shown in the table. An intensification strategy (Glover & Laguna 1993) was performed by flipping a coin after each move and returning to the best state ever seen with probability $1/2$. Additional experiments have been performed with different restart frequencies, and different noise levels, but the results were not found to be qualitatively different.

Discussion

As discussed in the introduction, ALS works best in problem domains having three attributes: a tractable solutions builder, optimality-preserving abstractions, and tractable analysis routines. Although these restrictions are important, each of them can be weakened.

Tractable builder: If the solution builder fails in some rare cases to generate a feasible solution then this is not necessarily fatal. One can give such priority vectors a very low "score" causing the search to avoid them. Whether this patch is workable in practice depends entirely on how common such priority vectors are.

Optimality-preserving abstraction: For large problems we can safely assume that we are not going to generate optimal solutions. Thus abstractions that are "nearly" optimality preserving may be sufficient. The utility of such abstractions can only be assessed by comparison of ALS against other techniques.

Tractable analysis: If there is no way to map from soft-constraint violations in the concrete solution to suggested changes in the abstract solution then local search essentially performs a weighted random walk. This may be acceptable, however, since undirected local search has been used successfully in various domains.

Conclusion

Abstract local search is a useful technique for solving constrained optimization problems. It is particularly suited to a problem domain where some fast deterministic algorithm can map a set of priorities into a solution that satisfies the hard constraints in the problem. ALS leverages off of several observations:

- The solution builder itself can encode a reasonable amount of domain knowledge, allowing the higher-level control (that is, the local search) to be domain-independent.
- The space of priority vectors offers a generic way to perform local search in complex domains. If there is an intricate solution structure that is easy to obtain constructively, but difficult to maintain by local repairs, priority space appears to be more suitable to local search than the space of concrete solutions.
- Optimal priorities cannot, in general, be determined a priori (if they can, and the abstraction is optimality preserving, then the optimization problem is tractable). However, they can often be improved by an analysis of concrete solutions. The general critical path analysis algorithm is an example of such an analysis.
- The solution builder can be efficient (at least in the domain of scheduling). Further, small changes to the priority vector can translate into large changes in the concrete solution. Together these facts allow ALS to be used productively on large problems.

Our experimental results demonstrate that ALS can perform meaningful optimization, compared to simple heuristic techniques, on large scheduling problems of high complexity. The utility of this approach is not limited to scheduling however. We see potential application domains for abstract local search in distribution planning, vehicle routing, or multi-level scheduling problems. Recent work (Clements *et al.* 1997) has examined superficially yet different problem domains such as graph coloring.

Although some forms of priorities have previously been used for representing schedules (Davis 1985; Syswerda 1991; Pinson, Prins, & Rullier 1994; Baar, Brucker, & Knust 1997), no general framework to integrate priorities into local search has been proposed. We conjecture that the basic loop of priorities feeding into a greedy solution builder and an analysis technique that updates the priorities will be applicable to a large class of constrained optimization problems, and will scale to problems of realistic size and complexity.

References

Aarts, E., and Lenstra, J. K., eds. 1997. *Local Search in Combinatorial Optimization*. John-Wiley & Sons Ltd.

Baar, T.; Brucker, P.; and Knust, S. 1997. Tabu-search algorithms for the resource-constrained project scheduling problem. Technical Report 192, Osnabrücker Schriften zur Mathematik.

Balas, E. 1969. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. *Operations Research* 17.

Clements, D.; Crawford, J.; Joslin, D.; Nemhauser, G.; Puttlitz, M.; and Savelsbergh, M. 1997. Heuristic optimization: A hybrid ai/or approach. In *Workshop on Industrial Constraint-Directed Scheduling (held in conjunction with CP'97, Schloss Hagenberg, Austria)*.

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

Davis, L. 1985. Job shop scheduling with genetic algorithms. In Grefenstette, J., ed., *Proc. of the 1st Intl. Conf. on Genetic Algorithms and their Applications*, 136–140. Pittsburgh, PA: Lawrence Erlbaum Associates.

Garey, M., and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. NY: Freeman, W. H.

Glover, F., and Laguna, M. 1993. Tabu search. In Reeves, C. R., ed., *Modern Heuristic Techniques for Combinatorial Problems*. Halsted Press. chapter 3, 70–150.

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.

Lawrence, S. 1984. Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques. Technical report, GSIA, Carnegie Mellon Univeristy.

Minton, S.; Johnston, M. D.; Philips, A. B.; and Laird, P. 1990. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. *Artificial Intelligence* 58:161–205.

Papadimitriou, C. H., and Steiglitz, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, New York.

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)*.

Slowinski, R., and Weglarz, J., eds. 1989. *Advances in Project Scheduling*. Elsevier Science.

Syswerda, G. 1991. Schedule optimization via genetic algorithms. In Davis, L., ed., *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.