An Approach to Resource Constrained Project Scheduling

James M. Crawford
CIRL
1269 University of Oregon
Eugene, OR 97403-1269
jc@cirl.uoregon.edu

Abstract

This paper gives an overview of a new approach to Resource Constrained Project Scheduling. The approach is based on the combination of a novel optimization technique with limited discrepancy search, and generates the best known solutions to benchmark problems of realistic size and character.

Introduction

Historically there has often been a mismatch between the types of scheduling problems that have been studied academically and the needs of the manufacturing community. The most obvious difference is that many real problems are much larger than common academic benchmarks. A second, and equally important, difference is that real problems generally involve constraints that have a more complex structure than can be expressed within a limited framework like job shop scheduling.

In this paper we overview ongoing work on resource constrained project scheduling (RCPS). RCPS is a generalization of job shop scheduling in which tasks can use multiple resources, and resources can have a capacity greater than one. RCPS is thus a good model for problems, like aircraft assembly, that cannot be expressed as job shop problems. In fact, if we take arguably the most widely used commercial scheduling program, Microsoft Project, RCPS seems to capture exactly the optimization problem that the "resource leveler" in Project solves.

It turns out that the algorithms that have been developed for job shop scheduling do not work particularly well for RCPS. In this paper we overview an approach to RCPS that is based on the combination of limited discrepancy search (LDS) with a novel optimization technique. The resulting system produces the best known schedules for problems of realistic size and character.

Resource Constrained Project Scheduling

A Resource Constrained Project Scheduling (RCPS) problem consists of a set of tasks, and a set of finite capacity resources. Each task puts some demand on the resources. For example, changing the oil might require one workman and one car lift. A partial ordering on the tasks is also given specifying that some tasks must preceded others (*e.g.*, you have to sand the board before you can paint it). Generally the goal is to minimize makespan without violating the precedence constraints or over-utilizing the resources.

RCPS is more general than job shop because resources can have capacity greater than one, and because tasks can use a collection of resources. This allows resources to be taken to be anything from scarce tools, to specialized workmen, to work zones (such as the cockpit of an airplane). RCPS problems arise in applications ranging from aircraft assembly to chemical refining.

Formally, Resource Constrained Project Scheduling is the following:

Given: A set of tasks T, a set of resources R, a capacity function $C : R \rightarrow \mathbb{N}$, a duration function $D : T \rightarrow N$, a utilization function $U : T \times R \rightarrow \mathbb{N}$, a partial order P on T, and a deadline d.

Find: An assignment of start times $S : T \rightarrow \mathbb{N}$, satisfying the following:

1. Precedence constraints: if t_1 precedes t_2 in the partial order P, then $S(t_1) + D(t_1) \leq S(t_2)$.

2. Resource constraints: For any time x, let $running(x) = \{t | S(t) \leq x < S(t) + D(t)\}$ Then for all times x, and all $r \in R$, $\sum_{t \in running(x)} U(t,r) \leq C(r)$.

3. Deadline: For all tasks $t : S(t) \geq 0$ and $S(t)+D(t) < d$.

A Benchmark Problem

Our experiments have been run on a series of problems made available on the WWW at:

`http://www.neosoft.com/~benchmrx`

by Barry Fox of McDonnell Douglas and Mark Ringer of Honeywell, serving as Benchmarks Secretary in the AAAI SIGMAN and in the AIAA AITC, respectively. These problems have 575 tasks and 17 resources. Some of the resources represent zone (geometric) constraints, and some represent labor constraints. Labor availability varies by shift. This is a synthetic problem that has been generated from experience with multiple large scale assembly problems. It is comparable to real problems is size and character, but simpler in the complexity of the constraints.

The results that have been posted to the WWW to date are shown in figure 1. Mark Ringer's results were found using a simple, first fit, interval based algorithm with no optimization. They were posted to encourage other contributions, rather than to generate the best solutions possible.

Who	Problem			Note
	2	3	4	
Mark Ringer	45	57	56	Honeywell
Nitin Agarwal	40	47	57	SAS
Colin Bell	39	-	-	Univ. of Iowa
Barry Fox	38	45	42	McDonnell Douglas
Crawford *et. al.*	38	43	41	CIRL
Crawford *et. al.*	38	39	38	(Lower Bound)

Figure 1: Results

Solution Methods

The two most important methods used in our scheduler are doubleback optimization and limited-discrepancy search. We discuss each in turn and then discuss how they work together in the scheduler.

Doubleback Optimization

The optimizer starts with any schedule satisfying the precedence constraints and generates a legal (and often a shorter) schedule. It works in two steps: a right shift and then a left shift (a very similar technique, *schedule packing* was independently invented previously by Barry Fox [1996]).

We first establish the right hand end point. Recall that the availability of some labor resources varies by shift. Because of this it turns out that it matters where in the daily cycle the endpoint is set.[1] The best heuristic seems to be to set it at the same point in the daily cycle as the end point of the current schedule. Another approach is to select the right hand end point randomly. This tends to shake things up a bit and makes iterating the optimizer more effective.

[1] This was first observed by Joe Pemberton.

Once the right hand endpoint is selected we right shift the schedule. In the right shift we take the tasks in order of decreasing finish times (i.e., from the right hand end). We shift each task as far right as it will go. In doing this shift we consider only precedence constraints and resource constraints with previously shifted tasks (one way to think about this is to envision the new right hand endpoint as being at positive infinity: the tasks that have not yet been shifted do not interfere with the construction of the new schedule).

Once the right shift is completed, we left shift back to time zero, starting from the beginning of the right shifted schedule. As before, we left shift as far as possible subject to precedence constraints, and resource conflicts with previously left shifted tasks.

This sequence can be iterated. At some points this produces longer schedules (possibly then followed by shorter schedules after additional iterations). At present we have no theoretical method to predict the optimal number of iterations: we simply iterate ten times and keep the best schedule produced.

To see the optimization works, consider the example shown in figure 2.

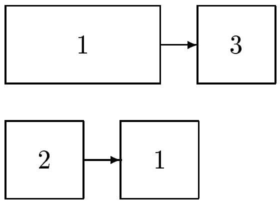

Figure 2: A simple scheduling problem.

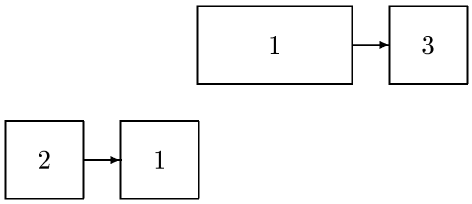

Figure 3: A bad schedule.

Here the boxes represent tasks and the arrows represent precedence relations. The numbers in the boxes are the resources the tasks need. For this example all resources have capacity one.

In order to break the resource conflict between the two tasks using resource one, we have to establish an

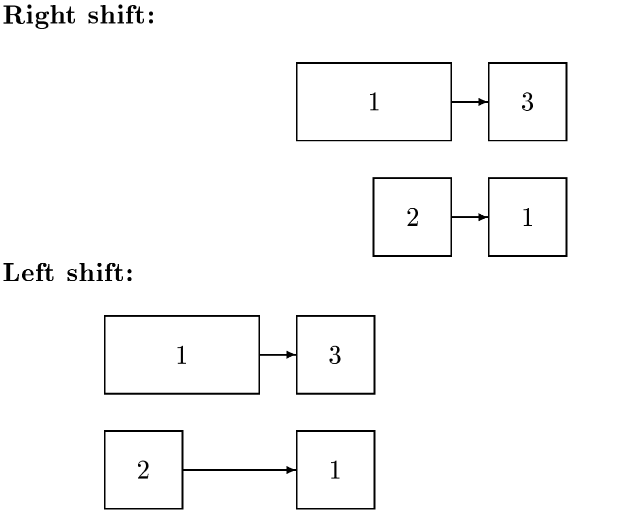

Figure 4: The effect of a right and then left shift.

ordering between the tasks. Assume that we do this non-optimally, generating the schedule shown in figure 3.[2]

Now consider what happens when we apply the optimizer. The right-shifted, and then left-shifted, schedules are shown in figure 4. The key thing to notice is that in the right shift the bottom task falls to the end of the schedule (because it has no successors), while the top task is forced to the beginning of the schedule. Thus the left shift schedules the top task first, generating the optimal schedule.

We can, of course, generate test cases in which the optimizer fails to find the shortest schedule, and we currently cannot offer any theoretical guarantees on the optimizer's performance. The strongest statement we can make is that on the benchmark examples, the optimizer is experimentally the single most effective scheduling technique we are aware of.

Why the Optimizer Works

Experimentally doubleback optimization is quite effective. Starting with the schedule that starts each task as early as possible subject to only the precedence constraints, the optimizer is able to produce 43 day schedules for problem 4. The obvious question is why such a simple technique works so well.

In a sense the key decision to be made in scheduling is the ordering of tasks that compete for a resource. In fact the search portion of our approach (see below) essentially searches all possible ways to break resource contention by establishing an ordering between competing tasks.

The challenge of breaking resource contention is that we do not know which task will turn out to be the most important. In some cases it is obvious which tasks are most critical. For example, if a task must be followed by a long series of tasks, then clearly we want to give it a high priority – otherwise it will be postponed and the "tail" of tasks that follow it is likely to exceed the deadline. Unfortunately it is not generally this simple (or else scheduling would be tractable). In essence what goes wrong is that we do not know how hard it will be to schedule the sets of tasks that must follow the conflicting tasks. However, if we start from a "seed" schedule, then we can decide with reference to the seed, how hard the subsequent tasks will be, and use this information to make better decisions about task priorities.

It turns out that this is exactly what the optimizer does. The right shift pushes all tasks as late as possible. So, if a task is near the beginning of the right shifted schedule, it is there because it must be followed by a large number of tasks. So it should be given high priority. This is exactly what the left shift does: the left shifted schedule is formed by first schedule the tasks that are near the beginning of the right shifted schedule.

Limited Discrepancy Search

The results returned by the optimizer are sensitive to the "seed" schedule given to the optimizer. One can construct examples of "bad" schedules that the optimizer cannot correct. In a sense the optimizer is walking down to a kind of local minimum, and the quality of the final schedule depends on where the walk starts.

Our implementation uses LDS (Harvey & Ginsberg 1995) to produce a series of seed schedules that are then passed to the optimizer. Here we give a brief overview of LDS. Details can be found in Harvey and Ginsberg [1995].

Imagine that we have a schedule that satisfies the precedence constraints, but not the resource constraints. The natural way to produce a legal schedule is to iteratively pick a resource conflict, delay one or more tasks long enough to break the conflict, and then propagate these delays through the precedence constraints. This is, in fact, how our current implementation works (starting from the left shifted schedule satisfying only the precedence constraints).

Each conflict that is broken creates a choice point, and breaking a series of conflicts produces a search tree. In the case of the benchmark problems this produces a search tree with a branching factor of about three,

[2] Such a mistake is unlikely in such a small problem, but our ability to avoid analogous mistakes in larger problems is, in a sense, the entire source of the intractability of scheduling.

and a depth of about 1000.

The traditional approach to searching such a tree is to use a depth-first search. In a depth-first search, we make a series of decisions until we reach a leaf node in the tree (in this case a leaf node is a schedule satisfying both precedence and resource constraints). We then back up to the last choice point and take the other branch, and follow it to a leaf node. We then back up again, this time to the latest branch point that still has unexplored children. Repeating this we eventually search the entire tree, and are thus guaranteed to find the optimal schedule.

Unfortunately, if the search tree is 1000 nodes deep then a depth-first search will examine only a tiny fraction of the entire search tree (backing up perhaps 10 or 20 nodes). Further, it is reasonable to expect that the choices that will be reconsidered are exactly the choice for which the heuristic is most likely to have made the right decision. To see why this is so, notice that near the top of the search tree there are still many resource conflicts, so the heuristics are working from a "schedule" that is far from legal, so the heuristics are having to guess at how the resolution of these other conflicts will interact with the current conflict. Near the bottom of the tree, however, the schedule is in nearly its final form so the heuristics have good information on which to base their decisions.

As a result, traditional depth-first search is relatively little help on scheduling problems. This has lead many practitioners to either use no search (just following the heuristic and returning the first schedule produced) or to use a local search (which can reconsider any decision at any point).

In LDS we fix a bound on the number of times we will diverge from the heuristic. If that bound is zero then we just produce the single schedule given by always following the heuristics. If the bound is one then we produce a set of schedules generated by ignoring the heuristic exactly once.

The difference between LDS and depth-first search is illustrated in figures 5 and 6. In both search trees the branch preferred by the heuristic is always drawn on the left. In figure 5 the leaves are numbered according to the order in which depth-first search will visit them. Notice that if the heuristic makes a mistake high in the tree, for example, at the first choice point, then depth-first search will have to search half of the search tree before correcting the mistake. In the LDS search tree (figure 6), leaf nodes are labeled according to how many times the path from the root to the leaf diverges from the heuristic (*i.e.*, how many right turns are necessary to reach the leaf). LDS searches the leaves by first searching the leaf marked 0, then all the leaves marked 1, and so on.

If the heuristic is generally correct, but sometimes makes mistakes (as if generally the case with heuristics) then we can reasonably expect to find good quality schedules by searching the nodes for which the number of divergences is low. If the height of the tree is h, then the complexity of visiting each node with d divergences is h^d. In practice we usually set d to 1 or 2. This produces much better results than a depth-first search examining the same number of nodes. Further, unlike a local search, LDS is systematic: if we continue to raise d we are guaranteed to eventually find the optimal schedule.

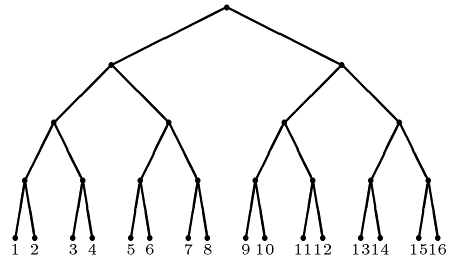

Figure 5: Backtracking search tree.

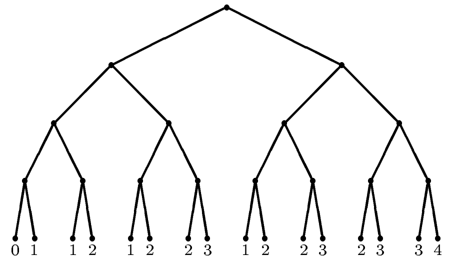

Figure 6: LDS search tree.

Finally we should node that LDS and the optimizer work well together. The optimizer is sensitive to the nature of the schedule it gets as input. Since LDS produces a series of reasonably good (but different) seed schedules, we can optimize each one, and in the end produce a schedule that is significantly shorter than we get by just optimizing the schedule given by following the heuristics exactly.

We can take this one step further and design the heuristic to avoid the kind of mistakes that the optimizer cannot fix.[3] This goes beyond the scope of the current paper, but it turns out that we can identify certain kinds of task-ordering mistakes that a simple right-left shift is unable to untangle. We can then generate heuristics that will generally avoid these mistakes. This may cause us to find worse unoptimized

[3]This idea came out of discussions with Matt Ginsberg.

schedules, but better schedules after optimization.

Conclusion

We have outlined an approach to RCPS problems that is based on using LDS to generate a series of "seed" schedules that are passed to an optimizer that can be seen as doing a kind of scheduling-specific local search. The results are currently the best known on problems of realistic size and character. Work continues on transitioning this technology to various application areas, and increasing the complexity of the constraints we can represent and effective optimize under.

Acknowledgment

This work has been supported by the Air Force Office of Scientific Research under grant number F49620-92-J-0384, by ARPA/Rome Labs under grant numbers F30602-93-C-00031 and F30602-95-1-0023, and by the National Science Foundation under grant number IRI-94 12205. The work was done at the Computational Intelligence Research Laboratory, and owes much to discussions with all the members of the lab, particularly Matt Ginsberg, Joe Pemberton, and Ari Jonsson. Finally, we should add that this work could not have been done without the effort Barry Fox and Mark Ringer have put in to make realistic benchmark problems available on the WWW.

References

Fox, B. 1996. An algorithm for scheduling improvement by scheduling shifting. Technical Report 96.5.1, McDonnell Douglas Aerospace - Houston. McDonnell Douglas has applied for a patent on this work.

Harvey, W. D., and Ginsberg, M. L. 1995. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, 607–613.