# Job Shop Scheduling

Marc Friedman
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
friedman@cs.washington.edu

October 6, 1997

## 1 Introduction

Shop scheduling is a combinatorial problem motivated by cost-effective management of expensive resources in manufacturing. For example, a shipyard builds only a few large boats, and would like to get every order completed with low latency and idle time. Management knows what orders they have to fill, what steps each boat requires, and how they are partially ordered. They know what people they have on each shift. They know what machines and drydocks they have, and what their maintenance schedules are. And they know when those resources are in use on other projects. They can certainly find a legal schedule which poorly utilizes their resources. What the computer program can offer them is a way to shave off a significant portion of their latency and cost by finding a better ordering of operations.

The job shop scheduling problem is a simplification of the problem sketched above. Jobs are totally ordered sequences of operations. Operations have fixed durations, and require fixed resources for that entire duration. A schedule is an assignment of start times to operations. One can imagine other shop scheduling problems, with stochastic durations, durations flexible subject to staffing, machine retool delays between operations, or penalties for failing to finish some jobs. These are not considered here.

This paper spans three major approaches to job shop scheduling. Various communities have approached the problem differently: with approximation algorithms, as an integer programming problem, or as a constraint satisfaction problem (CSP).

Shmoys, Stein, & Wein give randomized and deterministic algorithms for the currently best-known approximations of optimal job shop schedules. The randomized algorithm finds a schedule that has, with high probability, length at most $O(log^2 N/log\,log N)$ times the optimal. The deterministic algorithm derived from it has approximation ratio slightly more, $O(log^2 N)$. No lower bounds on approximations are known.

Applegate & Cook study consistency tests, heuristics, and algorithms for job-shop scheduling. The consistency tests they chose are cutting planes tests gleaned from the integer programming

literature. Rather than formulate the problem as integer programming, as has been done in the past, Applegate & Cook use the most promising results of that work (Balas 1985; Dyer & Wolsey 1990) integrated into a combinatorial search method (either branch and bound, or local search). They then cobble together three methods – local search, global search, and bottleneck analysis heuristics – into a very successful solution to optimal job-shop scheduling.

Sadeh & Fox use a CSP formulation with backtracking depth-first search through the space of operation start-time assignments. They study the effects of both domain-independent and scheduling-inspired heuristics on solution times. Simple depth-first search alone, with their heuristics, is one of the best performing algorithms known. Moreover, their heuristics are justified by the structure of realistic scheduling problems.

Crawford uses a local search method on a unique search space with an intuitively appealing neighborhood structure. His strategy for seeding the local search is also novel. Smith & Cheng apply a single new heuristic to get a very simple global search algorithm with performance as good as Sadeh & Fox.

In Section 2, I will define the problem in greater detail. In Section 3 I introduce the various auxiliary tools that help visualize and/or reformulate the problem. Section 4 summarizes the algorithms used by the authors in this study. Section 5 presents the approximation algorithm. Sections 6, 7, 8 break down the optimal and satisficing algorithms into their search, heuristic, and consistency components of note. Section 9 summarizes the experimental performance of those algorithms, and the analysis of the authors. Section 10 contains some further thoughts.

# 2 The Shop Scheduling Problem Space

Shop scheduling is a family of problems motivated by choosing start times for jobs on machines in a factory. All but the most restrictive forms of the problem (like some two-machine variants) are NP-complete in the strong sense (Garey & Johnson 1979). The job shop scheduling problem is a fairly simple problem in the shop scheduling family. Nevertheless, a famous ten-job, ten-machine instance of the problem proposed in 1963 (Muth & Thompson 1963) remained unsolved for 23 years.

**Definition 1** *A* **job shop scheduling (JSS) problem** *has $N$ operations, $M$ machines, and $J$ jobs. Jobs are disjoint sequences of operations. Each operation $i$ executes on a single machine $m_i$, for a specified duration $d_i$.*[1] *There are three possible objectives: to produce a decent, optimal, or satisficing valid schedule for the operations.*

**Definition 2** *A* **schedule** *is a mapping from operations to integral start times. There are three kinds of constraints on schedules:*

1. **Precedence:** *An operation begins only once its predecessors are complete.*

2. **Resources:** *A machine is assigned to at most one operation for that operation's entire duration.*

[1] We also require that no two operations in the *same* job use the same machine. Furthermore, all numeric quantities in the problem and solution are integers.

3. **Deadline:** *All operations start after the start time and end before a given deadline d (the satisficing objective).*

*A* **feasable schedule** *satisfies the precedence constraints. A* **valid schedule** *satisfies resource constraints as well, but not necessarily the deadline.*

The three constraints are qualitatively different. Precedence constraints form a partial ordering on all the operations in a problem, which can easily be satisfied with a topological sort.[2] Resource constraints, on the other hand, are disjunctive in nature and defy fast algorithms. To enforce mutual exclusion, a schedule must commit to an ordering decision for each set of operations contending for the same resource. In general any permutation is possible. All of the search in job-shop scheduling arises from this ordering decision.

## 2.1 Instead of Deadlines

I will also consider two variants of the deadline constraints to also be job shop scheduling: time windows and schedule length minimization. Some authors replace the deadline by target non-relaxable time windows for each operation. Time windows may be periodic. Time windows replace deadlines if a start and end time is given for each operation. This is more realistic, but not substantively harder, than the deadline problem. It better models the rescheduling application, where one set of jobs in a factory can be rescheduled without affecting other activities occurring at that time. It also better models workforce shifts and maintenance downtime.

Time windows make the problem no harder, despite the fact that in isolation, time windows are more powerful than deadlines. If there is one operation per job, and all operations use the same machine, the problem (with time windows) becomes the strongly NP-complete problem known as *sequencing with intervals* (Garey & Johnson 1979). This problem with a deadline rather than time windows is polynomial, since any ordering is equally good or bad. However, you can always reduce time windows to ordering constraints, by adding precedence constraints to imaginary operations that constrain any operation into a time window. So precedence constraints overwhelm any difference in difficulty introduced by time windows. Any algorithms we have seen work for both forms of the problem.

The optimization problem replaces the deadline by an objective function: find the earliest end time you can. The algorithms for the decision and optimization problems do not resemble each other, but there is a fundamental symmetry. Algorithms for the optimization problem can yield a decision, just by generating a schedule and testing whether it suits the deadline. Algorithms for the decision problem can progressively relax, or tighten, the deadline, until they yield a suitable schedule.

The shop scheduling family is an ideal candidate for approximation algorithms. Valid schedules are of immediate practical value, and are easy to find greedily. This is not the case for many NP-complete problems, for instance SAT. Incremental algorithms (such as Applegate & Cook's) or approximation algorithms, therefore, make good dollars and sense for scheduling.

[2]Precedence constraints with deadlines are NP-complete, however.

## 2.2 Major Variants

Some of the algorithms surveyed here work on more general classes of shop scheduling problems than JSS. The three important extensions are:

1. Partial orders: Replace the *sequences* of operations with binary precedence constraints (*dag shop scheduling*).

2. Multiple utilization: An operation may require any number of machines.

3. Multiple capacity: Machines may do $n$ operations simultaneously. Equivalently, there are classes of identical machines, and operations specify the class of machine they require.

Algorithms and positive theoretical results achieved on any of these extensions apply to the job shop scheduling problem as a special case. Unfortunately, the converse is not true, and we'll see some methods that do not generalize. I follow Crawford in calling the combination of these three extentions the *resource constrained project scheduling (RCPS) problem.* Sadeh & Fox solve RCPS as well, but restricted to tree-like precedence constraints. Shmoys, Stein, & Wein and Smith & Cheng tackle the dag shop scheduling problem, which encompasses its easier cousin, the *flow shop scheduling* problem, in which the project plan for every job is the same. For NP-completeness results on the shop scheduling family, see Garey & Johnson.

One NP-complete shop scheduling problem seems to be easier than the others. *Open shop scheduling* is flow shop scheduling minus the precedence constraints. The operations in a job must still be processed one at a time, though. Open shop scheduling is easily approximable (see Section 5).

# 3 Alternative Representations

Job shop scheduling as stated seeks start times for operations. Figure 1 shows an example problem with three jobs and three machines. The natural representation for the problem is as a choice of consistent values for the start time variables. Several alternative representations simplify the problem, or aid in its solution. Graphs, demand profiles, sets of linear inequalities, and polyhedra all contribute valuable intuitions.

## 3.1 The Scheduling Graph

Figure 2 shows the scheduling graph (Balas 1985) for the example scheduling problem in Figure 1. The nodes are the operations and a source and sink. There are both directed edges and undirected edges. Directed edges represent precedence constraints.[3] Undirected edges (dashed lines) link any two operations that use the same resource. Node weights (not shown) correspond to durations of the operations. We assume in the rest of the discussion that the set of operations is not empty.

**Definition 3** *A* selection $S$ *is a choice of direction for unordered edges in a graph. The new graph* $G_S$ *replaces unordered edges in* $G$ *with ordered edges in* $S$*.*

[3]Extra directed edges enforce that the source and sink are the extreme points.

Jobs: A,B,C
Operations: $1-8$
Machines: $X, Y, Z$
Process Plans:
$A \rightarrow (1,2,3)$
$B \rightarrow (4,5,6)$
$C \rightarrow (7,8)$
Utilization:
$X \rightarrow \{1,5,8\}$
$Y \rightarrow \{2,6,7\}$
$Z \rightarrow \{3,4\}$
Duration:

| | |
|---|---|
| 1 | 1 hour |
| 2 | 7 |
| 3 | 8 |
| 4 | 9 |
| 5 | 7 |
| 6 | 2 |
| 7 | 3 |
| 8 | 3 |

Figure 1: Example Job Shop Scheduling Problem

**Lemma 1** *A valid schedule for $G$ defines an acyclic selection $S$ on all unordered edges of $G$.*

**Lemma 2** *A selection $S$ for* **all** *unordered edges of the graph defines an infinite set of valid schedules, one of which has minimal length.*

**Proof:** Construct a valid schedule from $S$ as follows. Since there are no cycles, traverse the graph topologically. Assign to each operation any start time after all of the predecessors of that operation complete. The shortest such schedule is just the one where the minimum time is always chosen. □

**Theorem 3 (Correspondence)** *Let the length of a path through $G_S$ be the sum of the labels on the nodes. The scheduling optimization problem is isomorphic to the problem of finding $S$ to minimize the length of the maximum length path through $G_S$.*

The scheduling graph is a great way to visualize the scheduling problem. Each resource induces a clique on the operations which use it. There are three such cliques in Figure 2, connected by dashed lines. An acyclic selection on edges in the clique yields a temporal ordering for the resource. This ordering is, in turn, consistent with other orderings in the graph if and only if no cycle exists anywhere.[4]

There are two drawbacks of the scheduling graph:

[4]Dave Smith claims the typical scheduling problem's scheduling graph has lots of disjoint cliques, all small compared to the entire graph, loosely connected to each other.

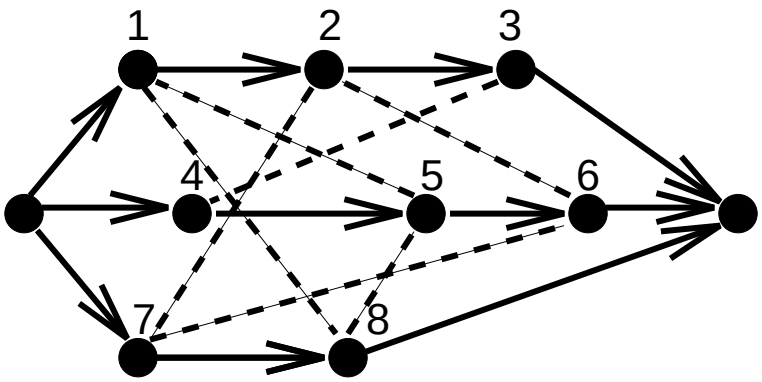


Figure 2: Scheduling Graph

- $t_j \geq t_i + p_i$ whenever $i$ precedes $j$
- $t_j \geq t_i + p_i - U\delta_{ji}$ whenever $i$ and $j$ use the same resource
- $t_j \geq 0$ for all operations $j$
- $\delta_{ij} + \delta_{ji} = 1$ for all $i < j$
- $\delta_{ij} + \delta_{jk} + \delta_{ki} \leq 2$ for all $i \neq j \neq k$
- $t_i \in \mathcal{Z}_+$
- $\delta_{ij} \in \{0, 1\}$

Figure 3: Example Integer Formulation adapted from Dyer & Wolsey. $t$ is a vector composed of components $t_i$, the start time of operation $i$. $\delta_{ij}$ is true if $i$ is before $j$. $U$ is a large constant.

1. The scheduling graph is not helpful for multiple capacity JSS problems. A machine with capacity $n$ can perform at most $n$ operations at once. Thus a valid schedule is one in which the clique associated with an operation must be covered by at most $n$ disjoint paths. This does not require a choice of direction for each undirected edge, and is more difficult to check than acyclicity.

2. The scheduling graph overcounts the number of decisions that must be made. Selection of orientations for an acyclic clique of size $k$ requires chosing from among the $k!$ permutations of nodes. But the space of possible selections is immensely larger: $2^{k(k-1)/2}$. Most of these are cyclic, and therefore invalid. The larger space may indeed have a convenient neighborhood structure; however, this counting argument does suggest that a tighter mapping to graphs might exist.

## 3.2 Linear Mixed Integer Programs

Another view of the scheduling problem treats the constraints on start time variables as a set of inequalities (a linear program) on those and other variables. The problem is then one of finding values of the variables that satisfy the equations, subject to the objective function that

the maximum end time be minimized. The hope of linear programming is to use generic linear programming solvers to find a concise set of linear constraints which must be true of a satisficing schedule. A simple example of such a program is given in Figure 3.

Unfortunately, the integrality constraints (the last two) make the problem hard. This particular formulation illustrates some of the problems with using linear programming here. The $t$ variables are integers, so after a linear solver has found a compact representation of the linear constraints, the valid part of the space must still be searched for integral solutions. But far more discouraging is the fact that binary zero-one variables have been introduced to represent disjunctive ordering decisions. There are $O(N^2)$ such variables,[5] and $O(N^3)$ equations on them. The large constant $U$ destroys any neighborhood properties that linear programming would leverage.

No IP method has kept pace with other combinatorial search methods for the scheduling problem. Research continues in finding more effective IP formulations, nonetheless. In a good formulation, the space would be smooth. The number of variables should stay within reason. The number of constraints, however, may not be very important. This particular formulation of scheduling as an integer program mirrors the scheduling graph formulation above. This formulation has a poor neighborhood structure, since the large constant $U$ is multiplied by a binary variable, making for a jumpy, rough space. This makes it hard to search for an integer solution. It may not be possible, however, to bring the number of variables down. A supposedly better formulation by Dyer & Wolsey increases the number of variables and constraints. The basic tradeoff made is an increase in variables *vs.* an improvement in neighborhood structure. They introduce $O(n^2)$ slack variables and $O(\left( \begin{array}{c} n \\ s \end{array} \right))$ variables representing initial sequences of $s$ operations. See Appendix A for the details. The number of variables and equations skyrockets with an increase in $s$. It appears that to smooth out the mixed integer problem, so that a linear solver can handle it, you suffer a (tunable) exponential increase in size, that is controlled by choice of $s$.

## 3.3 Constraint Satisfaction Problems (CSPs)

A CSP problem is a problem of assigning values from finite domains to a set of variables to satisfy some set of constraints on those variables. In their most general form a constraint may be any binary relation of two of the variables, specifying which pairs of assignments of values to those variables are legal. For the scheduling problem, one CSP representation (close to that of Sadeh & Fox) is:

- **variables:** start times
- **values:** any time between 0 and a predetermined maximum
- **constraints:** precedence and resource constraints expressed on those variables, *e.g.*:

$$start_i + duration_i < start_j \bigvee start_j + duration_j < start_i$$

  which is true whenever $i$ and $j$ use the same resource. This constraint could be expanded into a table of pairs of legal values for $start_i$ and $start_j$, given finite domains for these variables.

[5] The same overcounting shows up here as in the case of orienting graphs.

The CSP framework is very general, and does not really provide any help in visualizing the scheduling problem. Any scheduling problem with finite domains, and in fact any satisficing NP-complete problem over finite domains, can be stated as a CSP. The value of the CSP framework is that heuristics and consistency methods have been studied thoroughly and can be leveraged for the scheduling problem (Haralick & Elliott 1980; Prosser 1993; Kondrak & van Beek 1997; Bacchus & van Run 1997).

## 3.4 Demand Profiles

Demand profiles rapidly identify the hard part of a scheduling problem. Define the **demand function** of an operation on a machine $D_m^o(t)$, as the probability that operation $o$ uses machine $m$ at time $t$. Demand profiles are demand functions shown over time. Once an end time for the scheduling problem has been chosen, these probabilities can be estimated with a uniform prior: any consistent starting time for $o$ is equally likely. If part of the schedule has been chosen already, only consider the remaining start times that are consistent with previous decisions.

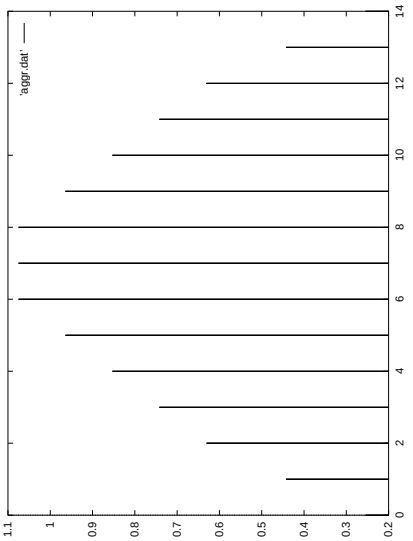


Figure 4: Aggregate Demand Profile for Machine X

**Aggregate demand profiles** give a comparative measure of the estimated *criticality* of machines. The sum of demand profiles of all operations requiring a given machine, $D_m^{aggr}$ gives a numeric estimate of the contention on that machine through time. Machines with high demand aggregates are probably more difficult to schedule. Figure 4 shows a demand profile for machine X.

Profiles have the peculiar advantage of working for the multiple capacity and multiple resource shop scheduling problems. They also can be defined on partial schedules, in which some of the operation start times have been chosen, or some constraints have been added.

### 3.5 The Scheduling Polyhedron

It is a small leap to view the start time variables $t_i$ as a vector in $\Re^n$. Now the precedence and deadline inequalities induce a *convex polyhedron* $P$ in $\Re^n$ which containts all the solution vectors. The challenge then is to find valid inequalities (*cuts*) that further shrink the search space. As we will see, these will come from the *facets* of the polyhedra for subgraphs of $G$.

Being inside all such polyhedra is a necessary conditions for valid schedules. But polyhedra only add conjunctive linear constraints. The disjunctive or integral constraints cannot be modeled in this fashion, though various authors cleverly tease out non-intuitive necessary conditions. Using a few of these extra constraints turns out to be very valuable, but there appears to be a scheduling problem that will requires exponential search for any polynomial polyhedron preprocessing scheme.

## 4 Algorithm Roundup

Six algorithms from five authors are surveyed in this paper. We name them for their first author, except for CUTS and SHUFFLE, both of which are from Applegate & Cook.

Below the differences are highlighted at a glance. SHMOYS is a fast approximation algorithm with the best known approximation ratio (of schedule elapsed time lengths). If $C$ is the elapsed time of the schedule returned, and $C*$ is the optimal, the approximation ratio is

$$\frac{C}{C*} = O(\tfrac{\log^2 N}{\log \log N}).$$

The other methods are search-based, since they seek either a satisficing or an optimal solution, NP-complete problems for which no polynomial algorithm is known. SADEH uses a CSP formulation on the space of operation start time assignments. The algorithm includes adaptations from modern CSP heuristics, consistency, and backtracking, and some new heuristics tailored to scheduling (ORR and FSS).

SMITH searches the space of ordering decisions, relying on a single simple heuristic that works well in that space. CBA (constraint-based analysis) is a method by Erschler, Roubellat, & Vernhes for pruning inconsistent states. CUTS uses a variety of cuts for heuristic guidance and consistency checking, while searching the space of ordering choices in a branch-and-bound fashion. SHUFFLE is Applegate & Cook's local search algorithm, which does well in some instances when global search does not. CRAWFORD uses a radically different search routine called **limited discrepancy search (LDS)**, to seed a local hillclimbing algorithm called doubleback optimization which is similar to putting the schedule on a see-saw.

| Name | Problem | Top level routine | Successor Fn | Consistency |
|---|---|---|---|---|
| SHMOYS | $C/C*$ | greedy | none | N/A |
| SADEH | satisficing | depth-first search | ORR-FSS heuristics | forward-checking *et al.* |
| SMITH | satisficing | depth-first search | min-slack heuristic | CBA |
| CUTS | optimal | branch and bound | order pair of ops | cutting planes |
| SHUFFLE | optimal | hillclimb | EDGEFINDER | cutting planes |
| CRAWFORD | optimal | LDS | doubleback opt. | none |

SADEH is unique in searching the space of start time assignments to operations, a distinction that is not represented in the table. Another distinction not mentioned is that only SADEH

and CRAWFORD generalize to the multiple capacity shop scheduling problem, and only SADEH generalizes to the multiple utilization problem.

# 5 Approximation Algorithms

Shmoys, Stein, & Wein present the first deterministic polylog approximations to job, dag, and flow shop scheduling. They also briefly survey the state of knowledge for approximation algorithms. There are no known negative results for shop problems, except that they are strongly NP-complete.

## 5.1 Some Useful Quantities

Several quantities bound $C*$, the length of the optimal schedule, from below. Any valid schedule must take at least $P_j$, the duration of job $j$. Likewise, each machine must work on each operation assigned to it one at a time. The sum of the durations of operations using machine $m$ will be denoted $\Pi_m$.

$$C* \geq \ max(P_{max}, \Pi_{max}) \geq \ (P_{max} \ + \Pi_{max})/2$$

Another quantity that plays a role in the approximation analysis is $d_{max}$, the longest duration of an operation.

All approximations for shop scheduling use the measure of schedule length to evaluate the quality of the schedule. In particular, the approximation ratio used is $C/C*$. The approximation ratios so far achieved are as follows:

| Problem | Ratio |
|---|---|
| open shop | 2 |
| all other shop (randomized) | $O(\frac{\log^2 N}{\log \log N})$ |
| all other shop (deterministic) | $O(log^2 N)$ |

## 5.2 Open Shop Scheduling

Open shop scheduling may be easier to approximate than the other shop scheduling problems. A simple greedy algorithm for open shop scheduling has approximation ratio 2. Consider the algorithm that schedules an idle machine immediately on any job that is not being processed. It turns out (due to Anna Racsmany) that

$$C_{greedy} \leq \ P_{max} \ + \Pi_{max} \ \leq 2C*$$

Consider the single machine $m$ that finishes last. It is processing at most $\Pi_{max}$ time, but its idle time is bounded by $P_{max}$ since the longest job was always processing while $m$ was idle (otherwise, $m$ would greedily process it).

There seems to be a gap between open shop and other problems, which may be harder (polylog at best). Shmoys, Stein, & Wein give a randomized polylog algorithm in two parts. They first build a feasable schedule, and then spread it until it is valid. The algorithm has high probability of

generating a feasable schedule with few (less than $O(\frac{\log N}{\log\log N})$) jobs contending for any machine at a given time. Those schedules need to be made valid by spreading out the contending operations. They show how to spread schedules with at most $c$ operations contending for a machine into valid schedules at most $2c\log d_{max}$ times as long. When $d_{max}$ is bounded above by a polynomial in $N$, the net result is a schedule $O(\frac{\log^2 N}{\log\log N})$ times the lower bound on $C*$.

In the rest of this section I will give the polylog approximation ratio proof. There are three parts: generating feasable schedules with contending operations at any given time at most $t$, spreading the schedules to get rid of contention, and derandomization.

## 5.3 Generating Feasable Schedules with Bounded Contention

First generate a feasable schedule by scheduling each job independently. Then to complete the algorithm, just add to each job an integer delay selected uniformly at random from $[0, \Pi_{max}]$.

**Lemma 4** *The probability that more than $k\frac{\log N}{\log\log N}$ jobs will be in contention is at most $p < N^{-(k-1)}$.*

**Proof:** The proof from (Shmoys, Stein, & Wein 1994) is sketched here. The probability of $\delta$ jobs being in contention at a fixed time $t$ is

$$\left(\begin{array}{c}\Pi_{max}\\ \delta\end{array}\right)\frac{1}{\Pi_{max}}^{\delta} \leq \frac{e}{\delta}^{\delta}$$

which is $\leq (N)^{-(k-1)}$ when you set $\delta = k\frac{\log N}{\log\log N}$.[6] The probability of contention at any time on any machine is bounded by this quantity, times $M$ times $\Pi_{max} + P_{max}$. Assuming both are polynomial in $N$, then by choosing $k$ large enough, we can make the probability very small. □

## 5.4 Spreading Schedules

Suppose an invalid schedule has at most $c$ jobs contending for a machine at any time. If the schedule is *synchronized*, *i.e.*, has all overlapping operations starting in lock step, then you can produce a valid schedule with spreading factor $c$, as in Figure 5, where $c = 3$. Just take each processing period, delimited by start times, and spread it into up to $c$ such periods, with contending operations assigned to different periods.

Unfortunately, the various operations are not synchronized. Let's make them so by sorting them by size as follows. Break up the schedule into time blocks of length $d_{max}/2$. Consider an obvious property of operations that begin in a block:

**Lemma 5** *Any operation starting in the block which is at least as long as the block ends after the block.*

So it is legal to spread the schedule by $d_{max}$, and move those long operations to a new period of size $d_{max}$ immediately after the block. Now divide the block into two. Move the long operations in each subblock to a new period of length $d_{max}/2$ after each of the new, smaller blocks, *etc.* (Figure 6). The schedule will ultimately spread by a factor of the number of phases, $\log d_{max}$. Each newly created period will be synchronized.[7] This completes the randomized algorithm. □

[6] They neglect to say that $k$ is chosen after $N$ is known, which is the only way this is possible.

[7] This analysis is not quite finished, since $d_{max}$ may not be $O(N)$. But a simple reduction generalizes to the case

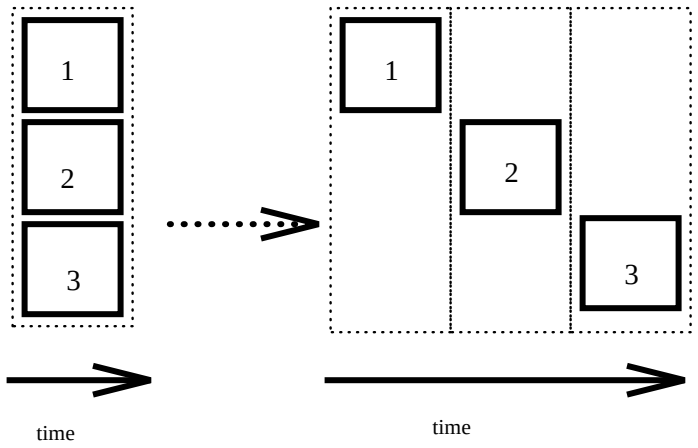


Figure 5: Spreading operations contending for the same resource

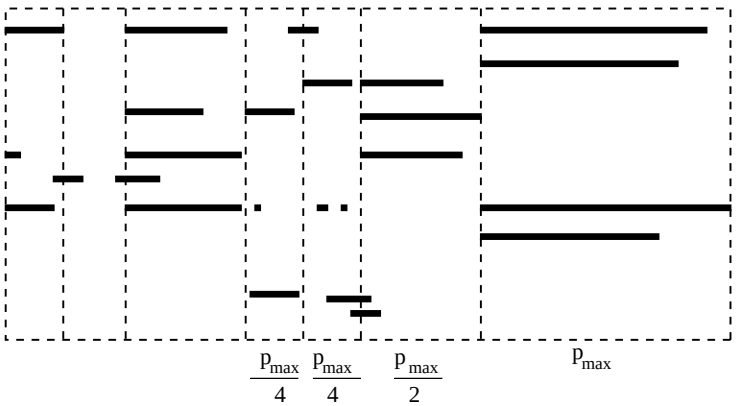


Figure 6: The synchronization algorithm after two phases. The boxes of size $>= d_{max}/2$ are synchronized.

## 5.5 Derandomization

Shmoys, Stein, & Wein provide a deterministic algorithm very similar to the randomized one above. The only step that is different is the selection of an initial delay for each job in the feasable schedule. Instead of choosing randomly, find an assignment that makes no more than $O(logMN)$ jobs contend for a resource at any given time. It turns out that this is a polynomial process, and they give an integer programming algorithm to solve it.

Let $L$ be the maximum length of a schedule, $P_{max} + d_{max}$. Job $j$'s machine utilization given an initial delay $d$ can be represented with a matrix $A_{jd}$ of size $L \times M$, where $A_{jd}[t, m]$ represents whether job $j$ uses machine $m$ at time $t$.

We form the integer program as follows. $x_{jd}$ indicates whether job $j$ is delayed by $d$. Then we attempt to minimize $W$, with the constraints:

1. For each job $j$,

---

of large $d_{max}$.

$$\sum_{d=1}^{d_{max}} x_{jd} = 1$$

2. For each position in the matrix $\mathbf{i} = (t, j)$.

$$\sum_{j=1}^{J} \sum_{d=1}^{d_{max}} x_{jd} A_{jd}[\mathbf{i}] \leq W$$

From (Raghavan 1988), we know that there is a polynomial time algorithm for finding a satisfying assignment when $W = W_{OPT} + \log NM$.[8] □

# 6 Search Routines

Four different styles of high-level search routines are applied in the algorithms we study: depth-first backtracking search, branch and bound, and two hybrids of local search and global search.

## 6.1 Global Search Routines

Global refinement search routines choose a commitment to make next, and branch on all the different ways to make it (Kambhampati, Knoblock, & Yang 1995). Commitments we have seen include choosing an order for a pair of operations which contend for a resource, or choosing a start time for an operation. *Depth-first search* methods, arguably the simplest of such methods, continue to build a single working partial solution by adding commitments until a point is reached where that partial solution becomes unworkable. They then backtrack, by removing decisions in reverse order, until the inconsistency is no longer provable, and continue adding different decisions, down another path.

The parameters of interest in depth-first search through CSPs are the space searched, the method of choosing variables to assign, and the method of preferring values to assign to those variables. Sadeh, Sycara, & Xiong have in fact studied the effects of more sophisticated backtracking techniques on the scheduling problem, but Sadeh & Fox's algorithms are presented here as though they use simple depth-first search. Smith & Cheng do, in fact, use simple depth-first search.

*Branch and bound* is like depth-first search, but it focuses refinement search on areas of the graph where a good solution may exist. A monotonic lower bound, which never gets lower as refinements are added, is used to prefer which decision to try next, and to remove regions of the graph entirely if they can not possibly have a good enough solution in them. Applegate & Cook's pruning bound is the length of the shortest valid schedule currently known, which is found by their SHUFFLEalgorithm. If a partial schedule represented by a search node can not be shorter than that, they throw it away.

[8] Raghavan calls this the vector selection problem.

One advantage of branch-and-bound is that it produces a lower bound even if the search is terminated before a solution schedule is reached. The lower bound value which is lowest among the current candidates is a valid lower bound on schedule length for the problem.

The monotonic function used as a lower bound is in this case the *preemptive lower bound* consistent with the constraints already posted. The preemptive lower bound is the maximum end time of any machine achieved by scheduling all the remaining operations on that machine as early as possible. You are allowed (in contrast to the JSS problem itself) to interrupt operations, in order to process them quickly. So it is an easy (polynomial) relaxation of JSS. The unvisited node with the smallest lower bound is greedily visited and refined by choosing an ordering constraint. When the lower bound for a node is a longer schedule than the best known valid schedule, the node can be pruned.

## 6.2 Local/Global Search Hybrids

Local search methods wander through a search space orthogonal to that of refinement search. That space may be poorer in hierarchical structure, so a smaller space will suffice: for instance, the space of fully specified schedules, rather than partially-constrained ones. Applegate & Cook use an algorithm SHUFFLE that sets some of the ordering constraints, and then uses a global search algorithm to optimally complete it. SHUFFLE relies on this vague performance principle for exponential search:

> If you reduce the global search depth by a little, you reduce the search by a huge amount.

In particular, SHUFFLE starts with a valid (presumably bad) solution. Then it retains $t$ randomly chosen ordering decisions from that solution. Then it extends that assignment to an optimal valid schedule using Carlier & Pinson's complete backtracking EDGEFINDER algorithm. This searches a much smaller space, so it can be done repeatedly. It restarts, fixing a set of $t$ operations disjoint from the previous set. Whenever a better schedule is found than the one before, it replaces the currently best solution. So it is a local search at the top level, with global search as a subroutine.

The local search phase is then followed by another phase of branch-and-bound search for a better solution, this time using the BOTTLE heuristic of Adams, Balas, & Zawack, which resembles Sadeh & Fox's ORR heuristic.

Crawford's algorithm is totally opposite: at the inner loop, it uses *doubleback optimization* to move from one schedule to a neighboring complete schedule. Doubleback optimization works like this. Take a schedule, and fix a maximum time limit. Slide each task, in turn, to the right (positive time axis) as far as it will legally go, considering the time bound, the time windows, the precedence constraints, and resource conflicts with already shifted schedules. Once you shift them all right, shift them all back toward the left. Now check to see if the schedule got better.[9]

The global search routine, limited discrepancy search, is used to seed the optimizer. The refinement search space is much like that used by Smith & Cheng and Applegate & Cook, but the order in which it is searched is unique.

[9]This is hillclimbing: if it fails to get better, stop.

Limited discrepancy search takes the view that backtracking with heuristics is a flawed way to search. Backtracking relies on a notion of neighborhood that artificially places a higher value on earlier decisions: earlier decisions will not be repealed until *all* later decisions are. Limited discrepancy uses the heuristic until the first backtrack. Then it searches the space of states with one decision different from the heuristic. Then two discrepancies, etc. This would be exactly the way to do it if, as they suggest, the heuristic is 'good' in the sense that it is right most of the time.

## 6.3 Discussion

What both of the hybrid methods offer is a way to blend the two (or three) kinds of search, using the advantages and alleviating the disadvantages of each. Local search is apt to get stuck in local minima, and possibly miss a globally optimal solution, without help from a global search routine to help shore up its coverage of the search space. Local search has the advantage of finding a pretty good solution pretty fast, however. Global search may take forever, and since it has to check every search node, it may not find a pretty good solution for a long while. It is the only way, though, to guarantee finding the optimal solution.

SHUFFLE has a particularly nice property: you can tune the number of operations with fixed order, thereby shifting the boundary between local and global search. Each iteration, the search depth of EDGEFINDER is $t$ less than for the entire search space, which reduces search by an exponential factor. Furthermore, SHUFFLE can leverage future results in global search by using any refinement search algorithm as a subroutine.

But there is no free lunch. $t$ divides up the incomplete and complete parts of the search. As subproblem size increases, the incomplete heuristic search subroutine will consider smaller and smaller fractions of the available solutions, and therefore find poorer and poorer suboptimal solutions.[10] On the other hand, a complete search subroutine would experience exponential blowup as long as $t$ is polynomial in problem size. If $t$ is kept small, however, the number of restarts required for equal coverage of the search space goes up exponentially. Applegate & Cook do not give any intuition into how to set $t$. They seem to use a value around the square root of the total number of operations.

SHUFFLE is crucial for the performance of the subsequent branch-and-bound search. Note that the approximation ratios between optimal solutions and the upper bounds supplied by SHUFFLE are all under 2% in their study. That may be the most valuable positive result of Applegate & Cook's study.

Doubleback optimization seems to be a winning local search strategy as well, being the fastest method known on the problems Crawford has run it on (which are mutually exclusive from the studies in the other papers). Whether LDS is better than depth-first search, on the other hand, is more of a religious issue. Crawford argues that LDS avoids falling into local minima, because it provides different seeds of steadily degrading quality at each iteration. But depth-first search provides seeds that are more different (in terms of more decisions different from the heuristic), but less good (by the same measure), than LDS. But it is the degree of differentness, rather than the degree of goodness, of the seeds, that determines whether local minima are avoided. Perhaps they should use a random seed if differentness is indeed what they are after. Moreover, the goodness

[10] But the rate of degradation may be vary critically, depending on the nature of the quality metric (in this case schedule length) and the smoothness of the space.

measure Crawford uses (being near the heuristically generated answer) may be no more realistic than a measure that validates backtracking search: one in which early decisions really do dominate later decisions in importance.

# 7 Heuristics

The goal of the heuristics we have seen is always the same: to find the most constrained variable, and assign it the least constraining value. This chooses the assignment with the highest apparent probability (given proper independence assumptions) of participating in a good solution.

Heuristics for CSPs work when they can solve the hard part of the problem first, and then the rest falls out. The assumption here (somewhat contrary to that of Crawford) is that there is a hard subproblem, and an easy remainder. If instead the difficulty were evenly spread among the variables or assignments, then no ordering heuristic would matter. Fortunately, this assumption matches well with the intuition that scheduling problems will have hard parts, or bottlenecks.

## 7.1 Heuristics for Start Time Assignments

Sadeh & Fox start from the realization that once a schedule length is imposed, finding any valid assignment of start times to operations is a CSP. They leverage generic CSP consistency methods like forward checking (Haralick & Elliott 1980) that are well-understood and extremely helpful in basic backtracking search. They try generic domain-independent heuristics, but find them lacking. So they develop new heuristics tailored to scheduling to choose a good refinement.

There are two decision points in global search for start time assignments: what operation to schedule next (variable ordering), and what time to schedule it (value ordering).

### 7.1.1 Variable Ordering Heuristics

The object of variable ordering heuristics is to work out the difficult part of the scheduling problem first, and hopefully extend that to a complete schedule with little difficulty. The new heuristics of Sadeh & Fox grew out of heuristics from the CSP literature, many of which they tried. The simplest of those is **dynamic search rearrangement (DSR)**, which chooses the remaining operation with the smallest number of possible start times. DSR's weakness is its failure to account for *bottlenecks* in the unscheduled portion of the constraint graph.

Bottlenecks are sets of resources whose capacity is expected to be severely taxed. Some methods that take bottlenecks into consideration give higher priority to highly connected subgraphs. **minimum width** orders the nodes last to first by removing the node with lowest degree from the constraint graph. **minimum degree** does the same thing without deleting the nodes. **maximum cardinality** picks the first variable at random, but then the variable most highly connected to those already chosen is picked next. Which of these three best discovers the bottlenecks will depend on the shape of the graph. But bottlenecks are not the end of the story. These heuristics do not take into account the varying tightness of the constraints in the graph. Moreover, they treat precedence and resource constraints uniformly, even though precedence constraints are easy to satisfy.

Heuristics can take into account the strength of constraints by using probablistic notions of the likelihood of backtracking. In particular, Keng & Yun invented a heuristic for scheduling called the survivability measure, which takes into account varying tightness of constraints. They assign to each start time assignment an estimated **survivability measure**, and then assign the variable with the lowest overall survivability. This survivability measure can be derived from the aggregate demand profiles (see Section 3.4). The probability that an assignment will survive contention is the product of the probability that each resource needed by an operation is not taken by some other unscheduled operation. This is a crude estimate, since it depends on subjective probabilities, and then assumes independence among them. But Keng & Yun's measure, used for variable ordering, proved superior to all the heuristics mentioned above.

Sadeh & Fox have a more complex variable ordering heuristic that they call **operation resource reliance (ORR)** which works even better. ORR looks for the highest point on any aggregate demand profile of some machine. ORR defines the most critical operation as the one "with the highest contribution to the demand for the most contended-for resource/time interval" (p. 21).

### 7.1.2 Value Ordering Heuristics

The object of value ordering heuristics is to choose a start time for an operation that is in more complete solutions than any other. These are the least constraining values. Actually finding such a value would require solving the complete scheduling problem, so heuristics rely on various simplifying assumptions to gauge the likelihood that a value is in many complete solutions.

Dechter & Pearl provide the point of departure: a heuristic method called **advised backtracking (ABT)**. They realized that if a constraint graph is acyclic, the number of complete solutions that a value assignment participates in can be computed fast, in $O(nk^2)$ time, where $n$ is the number of variables and $k$ is the size of the largest domain of these variables. Tree-like relaxations can be obtained from the constraint graph by finding spanning trees. If a 'good' tree is desired, each edge can be labelled with the number of pairs of values satisfying that constraint, and the **minimal spanning tree (MST)** can be found easily. The MST should contain the most constraining constraints.

ABT has problems with cost and accuracy. With dynamic variable ordering, a new MST may be needed at each search node, potentially an $n^2$ prospect. But far more damning is the fact that a tree-like relaxation throws away exactly the information most needed to solve the hard subproblems: the highly connected subgraphs. Since a clique of size $m$ will have $\begin{pmatrix} m \\ 2 \end{pmatrix}$ nodes, the vast majoriy will be lost in a tree-like relaxation.

Keng & Yun do better, by using measures of contention on a single machine, to estimate which values are least constraining. Sadeh & Fox's heuristic, **filtered survivable schedules (FSS)** combines the influence of contention with tree-like relaxaions. It is pretty complicated (see Appendix B), but it works more or less as follows:

- Consider the operation $o$ to be scheduled, and the tree of precedence constraints containing it. This is a tree-like relaxation.
- Enumerate the solutions to this relaxation that each start time participates in, using Dechter & Pearl's $nk^2$ algorithm.

- Weight each solution with its survivability estimate, drawn from the demand profiles. This factors in the highly connected subgraphs.
- Sum the weighted solution counts, and choose the start time value with the highest score.

## 7.2 Min-slack: A Heuristic for Ordering Assignments

Smith & Cheng have found a simple method that does as well as Sadeh & Fox's ORR-FSS on Sadeh & Fox's problem suite without all the complications of survivability measures and demand profiles.

They consider the single-machine, single-utilization job shop scheduling problem with non-relaxable time windows. They search the space of pairwise operation ordering decisions, rather than the space of start time assignments. The variables they assign are the binary variables that determine whether one operation precedes another. They consider this a major selling point of their approach, but note that it is really an old idea taken from the work on scheduling graphs and linear programs (Section 3).

Their heuristics are based on the notion of *slack*, also found in Appendix A:

$$\begin{aligned} slack(i \rightarrow j) = & \text{ Latest end time of } j \\ & - \text{ Earliest start time of } i \\ & - \text{ combined processing time of } i \text{ and } j \end{aligned}$$

The most constraining variable, which they will assign next, is the variable ordering pair $i, j$ with either $slack(i \rightarrow j)$ or $slack(j \rightarrow i)$ being the lowest slack value of any possible ordering. This almost works, but the perceptive reader will note that this unfortunately ignores the larger of the pair of slack values. The least constraining *value*, however, is obvious: choose the ordering with the larger slack value. Combined with a consistency maintenence method due to Erschler, Roubellat, & Vernhes, which they do not discuss in depth, their method almost bests ORR-FSS.

In order to make it work as well, they use a similarity measure between the pair of slack values $slack(i \rightarrow j)$ and $slack(j \rightarrow i)$. Holding the smaller value fixed, the larger value is more constraining when it is small, *i.e.*, the spread between the two values is small. Since there are two important quantities to combine in a heuristic, they take a parametrized function of the two quantities and experimentally found good parameters. So good, in fact, that it does as well as ORR-FSS.

# 8 Cuts

Applegate & Cook blend integer programming work with a branch-and-bound algorithm by using true inequalities as **cuts**. Cuts are necessary conditions. They are used as consistency measures in search, which have two uses: to restrict the choices explicit in a search node, and to prune the node. Their search routines track the lower bounds of start times and end times, and the cuts help to shrink these intervals. If such an interval becomes empty, the node can be pruned.

In general, some polynomial-time constraint propagation and consistency checking occurs at each node in the search tree. During refinement, some linear inequalities accrue that are necessary

conditions for a solution to be in the part of the solution space represented by the node. In the case of clique cuts (see below), the system is required to solve a linear program to apply a cut. Otherwise, the check is very fast.

Consistency checks make a tradeoff: Fewer nodes to search versus more work at each node. Cuts are helpful for the simple reason that extra constraints can only make global search trees get smaller, by pruning higher up. Empirical tests can determine how much consistency checking seems to be enough.

## 8.1 Basic Cuts

Basic cuts can be derived by algebraic manipulation. Basic cuts express the following simple relationship between start times of all operations using a certain machine, and the durations of the other operations on that machine:

> Each operation waits for all of those before it to complete.

Consider a set $S$ of operations that use the same resource. Then in any valid solution, the set of operations must be scheduled non-overlapping in time. $t_i$ will be the start time of operation $i$, $d_i$ its duration. Likewise, $d_S$ is the total duration of the set $S$, and $t_S$ is its first start time. The following is therefore true of any valid start times:

$$\sum_i d_i t_i \geq \; d_S t_S + \sum_{i \neq j \in S} d_i d_j.$$

**Proof:** Suppose we process the operations $o_1 \ldots o_N$ in lexicographic order. We will derive the inequality above, which is symmetric in the operations, so it will be true for any ordering imposed. Keeping in mind that the operations may not overlap in time, we define new delay variables $l_i$ for each i, indicating how much later $o_i$ starts than $o_{i-1}$.

$$\begin{aligned}
\textstyle\sum_i d_i t_i &= d_0 t_0 + d_1 t_1 + \ldots + d_n t_n \\
&= d_0 t_0 + d_1(t_0 + l_1) + \ldots + d_n(t_0 + l_1 + \ldots + l_n) \\
&\geq d_0 t_0 + d_1(t_0 + d_0) + \ldots + d_n(t_0 + d_0 + \ldots + d_{n-1}) \\
&= d_S t_0 + \textstyle\sum_{i \neq j \in S} d_i d_j \\
&= d_S t_S + \textstyle\sum_{i \neq j \in S} d_i d_j \;\; \square
\end{aligned}$$

The inequality is tight (equality) exactly when the operations are scheduled back to back, without any waiting time.[11] There is an equivalent cut that is the reverse of this one involving the end times and durations.

[11] Applegate & Cook attribute the basic cuts to Dyer & Wolsey, but I do not see how they are derived from that work.

| Permutation | Earliest Start Time (3) | Earliest Start Time (4) |
|---|---|---|
| $(3,4)$ | 8 | 16 |
| $(4,3)$ | 9 | 0 |

Figure 7: Clique for machine X in Figure 1. The earliest start times are given by the precedence constraints, within each job. Operation 3, for instance, is preceded by operation 2, which could finish at time 8. But if it must wait for operation 4, it cannot start until time 9.

## 8.2 Clique Cuts

By far the most interesting and promising class of cuts are the clique cuts. A few definitions are in order first. Say $z$ is the size of set $K$.

**Definition 4** *$P(K)$, the* **scheduling polygon for clique** *$K$, is the set of all partial schedules $\vec{t} \in \Re^z$ for the operations in $K$, consistent with the precedence constraints. The schedule is represented as a vector of start times of operations.*

**Definition 5** *An inequality $\alpha$ induces a* **k-dimensional face** *of a polyhedron $P$ if the inequality is true of all points in $P$, and is tight for $k+1$ affinely independent points of $P$.*

**Definition 6** *A* **facet** *of a polyhedron $P$ is a face of dimension one less than the dimension of $P$. So a facet of $P(K)$ is a face of dimension $z-1$.*

The clique cuts are the facets of the scheduling polyhedron for sets of operations that use the same resource. A facet supplies a single inequality that is a necessary condition for the vector $\vec{t}$ to be a valid schedule. Facets with a bounded number of variables (Applegate & Cook limit themselves to 5) are not hard to compute. Each one is a solution to a set of linear equations in at most, say, 5 variables.

**Definition 7** *The* **polar** *of $P$ is the set of non-negative vectors*

$\{\vec{u} \mid \ \vec{u}\vec{t} \leq 1$ *for all* $\vec{t} \in P.\}$

*The* **reverse polar** *$P^{\#}$ is similar, but with the less than sign reversed.*

Notice that any point $\alpha$ in $P^{\#}$ defines a necessary condition for $\vec{t}$ to be in $P$, and thus a necessary condition for $\vec{t}$ to be a valid schedule. The strongest such constraints, the facets of $P(K)$, are the vertices of $P(K)^{\#}$. In fact, the converse is true as well: the facets of $P(K)^{\#}$ are the vertices of $P(K)$.[12]

I will now sketch how to find clique cuts for $P(K)$:

1. Find the vertices of $P(K)$. The vertices of $P(K)$ are a subset of the extreme points of $P(K)$, which we can enumerate easily. An extreme point of $P$ is a point in $P$ that is not an affine transformation of two other points in $P$. In the case of cliques, these are exactly the rows of the table in Figure 7. The vertices are just the rows that are not dominated by any other row. [13]

[12] This is not true of all polyhedra, but scheduling polyhedra have certain benevolent properties, like being displaced cones, and being nonnegative, which make this true.

[13] One row $\alpha$ dominates another row $\beta$ when all entries of $\alpha$ are smaller than the corresponding entries of $\beta$.

2. Find vertices $\alpha$ of $P(K)^{\#}$. This amounts to solving a system of linear inequalities, for an extreme value. The system of linear inequalities is $\{\vec{u} \mid A\vec{u} \geq \mathbf{1}\}$ where $A$ is the table from Figure 7 and $\mathbf{1}$ is the vector whose entries are all 1. This is polynomial in the size of $A$, but $A$ is exponential in the size of the cliques.

3. Add the inequalities $\alpha\vec{t} \geq 1$ to the active constraints.

Consider the example above. We need to solve the following system of equations:

$$8u_1 + 16\ u_2 \geq 1$$
$$9u_1 + 0\quad u_2 \geq 1$$

There is only one extreme $\vec{u}$ for this equation: $(1/9, 1/144)$. So we get the clique cut

$$\frac{1}{9}t_1 + \frac{1}{144}t_2 \geq 1$$

.

### 8.3 Other Cuts

The other cuts are based on clique cuts. You can extend the clique cuts to two machines to get two-machine cuts, in which all orderings of each job are crossed to form a row of the matrix $A$. Applegate & Cook only consider the **two-operation, two-machine cuts**.

The remaining cuts they propose unfortunately require extra binary variables $Y_{ij}$, which indicate whether or not $i$ is before $j$, where $i$ and $j$ are operations using the same machine. A simple one involving only one machine says that for a set $S$ of operations using one machine, for some $i \in S$, $i$'s earliest start time is at least $S$'s earliest start time plus the duration of all operations in S scheduled before $i$ (**half cuts**). **Late job cuts** are a slight improvement on half cuts.

When Applegate & Cook compared the various cuts, they could not get reasonable running times out of the cuts involving $Y$ variables. One possible reason for this is that the methodology changes from branching on a decision to using linear programming, where $Y$ variables range over real values. This requires a different search algorithm, which they do not divulge. Their results make sense, however, as indicating that only very slight improvements are possible at great cost using these cuts.

## 9 Experiments, Claims, and Counterclaims

The scheduling community has been blessed with two prominent problem suites that have guided research: the famous Muth & Thompson ten-machine, ten-operation problem, and Sadeh & Fox's more recent randomly generated scheduling problems. Sadeh & Fox make the claim that their random problems are realistic, which is much more credible than any similar claim in the planning world, since there are, and have long been, scheduling programs used in industry. Applegate & Cook's study predates the widespread use of Sadeh & Fox's domains, so they used some others that are, to their minds and to their programs, harder than the zinger by Muth & Thompson.

Smith & Cheng compare their algorithm head-to-head against Sadeh & Fox on the latter's problem suite, and do as well. Crawford uses test problems unrelated to those of these other studies, but he hints that these are genuine problems new to circulation that other practitioners have tried and failed to solve as well.

One of the triumphant moments for the scheduling community came at a symposium in 1992 when three research groups (including Sadeh & Fox)announced three different methods that all solved the Sadeh & Fox suite.

Sadeh & Fox's experimental results lead them to advocate the use of ORR-FSS. ORR-FSS is a good algorithm, and has the advantage of generalizing to the multiple utilization, multiple capacity shop scheduling problem. However, it is scruffy and complex. Their formulas for approximating availability and survivability have no particular claim on correctness or optimality. The notions themselves are based on assumptions about the shape of typical scheduling problems. There are two other factors working against ORR-FSS: the fact that two other groups solved the Sadeh & Fox suite, and the fact that Smith & Cheng have more recently matched it with a very simple algorithm.

Smith & Cheng do as well with a simpler heuristic on a different search space: that of pairwise orderings. Their main conclusion is that Sadeh & Fox searched the wrong space. A further conclusion that one may be tempted to draw is that complex heuristics may be unnecessary, since there could be an equally good simple heuristic. From a practitioner's point of view, however, the cost of developing complex heuristics may not matter, if those heuristics work.

Smith & Cheng's experiments have one glaring weakness: they do not compare run times, just solved problems versus unsolved, after some time threshold. I do not believe this nullifies their conclusion, but it does weaken it a bit. Smith & Cheng make another important contribution: they configure their heuristic-combining parameters experimentally. Something like that could be very helpful for the hybrid algorithms of Applegate & Cook, who combine three search methods without a whiff of rigor.

Applegate & Cook feel that there is great value to each of the methods they combine–SHUFFLE, EDGEFINDER, and BOTTLE, and their positive results back this up. They also found negative results, since they could not solve seven of the ten new problems of their own design, possibly the hardest test suite around. They do not comment further on how to combine or configure their algorithms, or what intuitions they have gained from the process of devising this particular algorithm.

Applegate & Cook do not analyze their intriguing results very deeply. With regard to cuts, their commentary seems slightly odd. They point to the $\leq 7\%$ improvement in lower bounds found by basic cuts as an indication of the great value of cuts. They then go on to make excuses for their poor linear solver, which they blame for the poor run times of the complex cuts. They then drop the discussion of cuts for good, without answering the crucial question of what cuts one should use.

The cuts requiring a mixed-integer formulation look useless, since they rarely terminate, and raise the lower bounds by less than 1% when they do. The question that remains is whether to pay one order of magnitude for the basic cuts, or the three orders of magnitude for the even less astounding improvement of clique cuts. It appears by their silence that *they may not use cuts at all in their solver!*

Crawford feels that doubleback optimization is the reason why his scheduler is the best known

for the problems he has run it on. He is quiet about the contribution of LDS. Since he does not show any direct comparisons, other than his one algorithm versus other people's best solutions, there is little he can tell us about LDS's value.

# 10 Conclusions

The papers in this study paint an unhappy picture of a useless theory (that of linear and mixed-integer programming), and a scruffy science not really homing in on any unified framework.

The two promising efforts at using a well-founded theory have problems. Sadeh & Fox have integrated their heuristics efforts with a study of modern CSP techniques and how they work on shop scheduling (Sadeh, Sycara, & Xiong 1995). Unfortunetely, new algorithms have already outdated the ones they studied (Prosser 1993; Kondrak & van Beek 1997; Bacchus & van Run 1997). Furthermore, all their efforts *may* be wasted because they chose the inferior search space of operation start times. Smith & Cheng make the sound theoretical contribution of experimentally configuring their heuristic-combining parameters. Their work is very preliminary, however.

But on the bright side, practical problem instances are easy to identify, and two suites of them are widely studied, with good algorithms emerging. If I had to choose one, I would probably pick SHUFFLE, which found 2%-worse-than-optimal schedules. Of course, this is not a performane guarantee, and there probably will never be one for local search. However, since the test problems are probably realistic, there is hope that SHUFFLE would work pretty well on a real problem.

I outline some useful work that could improve the current state of affairs below.

- Try the best-known, up-to-date CSP algorithms (Kondrak & van Beek 1997; Bacchus & van Run 1997) on the JSS problem in both its start times and ordered operations formulations. See which one wins. Start adding heuristics back in.
- Run doubleback optimization with various search routines for seeds, including random seeds, to see if seeds actually matter. Try doubleback optimization on the well-studied problem suites.
- Use SHUFFLE, followed by a better global search routine than BOTTLE, such as Smith & Cheng's.
- Tune SHUFFLE to find a good value for $t$ (see Section 6.2). Try varying other parts of SHUFFLE, like its choice of which $t$ operations to retain.
- Try other combinations, such as SMITH with basic cuts for consistency.
- Try to apply SADEH's heuristics explicitly in SMITH's space.

# A A Superior Mixed Integer Program for Scheduling

Dyer & Wolsey give a more complicated linear mixed integer program for scheduling that is superior to the one in Figure 3. It only applies to the non-relaxable time windows formulation. The great advantage here is that the resource constraint is modeled without making the LP problem too