# Getting to the real problem: experience with BNR Prolog in OR

W.J. Older* G.M. Swinkels† M.H. van Emden‡

**Abstract**

Although job-shop scheduling is a much studied problem in OR, it is based on an unrealistic restriction, which is needed to make the problem computationally more tractable. In this paper we drop the restriction. As a result we encounter a type of cardinality constraint for which we have to develop a new method: translation to a search among alternative sets of inequalities between reals. Our solution method depends on logic programming: we run a specification and rely on the underlying interval constraint-solving machine of BNR Prolog to reduce the search space to a feasible size. In this way, by making the programming task trivial, it is possible to tackle the real problem rather than a related one for which code happens to be already written.

## 1 Introduction

To buy off the shelf, or to commission a custom program? This is a decision that often needs to be taken and is sometimes a difficult one. Even in easy-to-program applications such as bookkeeping or inventory control, this decision often comes out in favour of deploying an existing package because of the high cost of developing custom software. Especially in Operations Research (OR), where the algorithms are much more sophisticated, there is great reluctance to embark on a custom programming project. As a result there is great temptation to twist the actual problem to be solved to a different one for which a package happens to be available.

Logic programming is promising in such a situation: what demands considerable programming effort in Fortran or C can be easy in Prolog. However, when an OR problem is thus quickly and elegantly programmed in Prolog, performance tends to be abysmal. In response to this challenge, the designers of CHIP [7] have shown that it is possible to run logic programs for certain combinatorial OR problems sufficiently fast to be of practical

*Computing Research Laboratory, Bell Northern Research, Ottawa, Canada

†Department of Computer Science, Simon Fraser University, Canada

‡Department of Computer Science, University of Victoria, P.O. Box 3050, Victoria, B.C., Canada. Internet: vanemden@csr.uvic.ca. Voice: (604) 721-7225. Fax: (604) 721-7292.

interest. Van Hentenryck [11] has acknowledged that such programs tend to be slower than custom C code. But they are much faster to program. It is not often that one can choose between hiring a programmer to write custom C code or buying a workstation to absorb the extra computational load imposed by doing one's OR applications in CHIP.

CHIP derives its advantage over Prolog from the use of the consistency method based on finite domains [15, 14]. This gives good results for many combinatorial OR problems. Other variants of Prolog, such as BNR Prolog [2], are based instead on intervals of numbers. In this paper we present an OR problem where this other approach yields advantages similar to those demonstrated for CHIP and its successors.

Our starting point is a situation that is typical in practice: the actual problem does not fit one of the standard OR methods, at least not in a computationally feasible way. Instead of distorting the problem into a temptingly close relative that does occur in the OR repertoire, we make use of the high level of Prolog to tackle the problem itself. We find that the general-purpose interval narrowing algorithm based on consistency is sufficient to achieve an impressive reduction of the search space.

Thus the situation in general terms. In the remainder of this paper we describe the specific OR problem that we address, the mathematical analysis that makes it amenable to interval constraints, and finally we show the complete code of a short program for which interval constraints give a promising reduction in search space.

# 2 From trucking to the job-shop

Our starting point was how to schedule logging trucks. Logs are harvested and collected at the most remote sites that are accessible. They are then moved by truck to a depot where they are sorted in preparation for further transportation. The depot services around ten collection sites. Each truck departs empty from the depot, drives to a collection site, is loaded full, returns to the depot, is unloaded completely and then is ready to start another cycle, possibly servicing a different collection site, but returning to the same depot. Each collection site is characterized by the rate at which logs are harvested and by the maximum quantity of timber that can be stored at the site.

The penalty for poor scheduling is that trucks have to wait till a full load is available at their destination site, or that harvesting has to be interrupted because of the limit on storage capacity at the collection site. Similar harvesting situations occur elsewhere: wheat, sugar beets, and wherever agriculture is mechanized on a sufficiently large scale.

Thus we consider abstractly the *harvest collection scheduling problem*, where there are a number of *sites* at which *produce* is collected by some kind of harvesting machine. All sites are serviced by a single *depot*. A fixed number of trucks is available to drive empty from the depot to a site, get fully loaded there, and return to the depot to be unloaded.

This general formulation also covers the dual situation, where the depot supplies to the sites instead of collecting from them. Each of the sites is then again characterised by its distance from the depot and by the rates at which it consumes. But we do not consider *vehicle routing*, where other routes are considered than from depot to a site and back.

In OR a problem is rarely new. So we looked around for logically similar, though possibly superficially different scheduling problems. A promising candidate seemed to be that of job shop scheduling of which we next give a brief description.

**The job-shop scheduling problem.** Even if a job is as simple as making a hole in a piece of metal, it may be necessary to drill, tap, counterbore, ream, and countersink, all in a particular order. Each activity requires a machine that cannot do anything else. When many jobs have to be done, each requiring the same machines in a different order, it is difficult to find an optimal schedule.

Many scheduling problems have similar characteristics. They have been abstracted as follows to the *job-shop scheduling problem*: a number of *jobs*, each consisting of a sequence of *activities*, have to be completed. There are available *machines*, one type of machine for each activity. Each machine can only be engaged in one activity at any moment in time. The sequence of the activities in each job is given, being determined by technological constraints. Two problems are considered in such situation:

1. the *minimization variant*, which is to minimize the criterion of interest. A common example of such a criterion is the *makespan*, the elapsed time between the start time of the first activity and the completion time of the last activity.

2. the *decision variant*, which is to decide whether a feasible schedule exists with a given makespan

**From harvest collection to the job-shop.** Harvest collection can be regarded as an instance of the job shop: getting the day's produce from a site can be regarded as a job. The activities of the job are to get the successive truck loads of produce to the depot. As in the job shop, each activity requires a machine, a truck in this case. Viewed in this way, the sequence of activities is not given explicitly beforehand, but is generated by a nondeterministic algorithm that takes the production rate and the storage capacity of the site as inputs. Thus it might seem we have a special case of the job-shop scheduling problem.

However, in the harvest collection problem it is the rule that trucks are interchangeable, though exceptions to this rule exist. In job shop scheduling it is the other way around: for each activity there is exactly one machine on which this activity can be performed. At first it was not clear how important this distinction is.

What is clear, is that in practice we are not alone in requiring such a generalization of the job-shop scheduling problem: whenever one machine consistently is found to be the bottleneck, another functionally equivalent copy of it will usually be acquired. Thus job-shops where the numbers of copies of each machine roughly reflect the relative demand for each type of activity are the rule rather than the exception.

The question then arises, how important is the restriction to one copy of each machine type? The history of an important benchmark, the 10 machine, 10 job problem, helps to answer this question. Originally proposed by Fischer and Thompson in 1963 [8], various

researchers kept getting better solutions for this problem for twenty years, until finally Lageweg posted a record that still stands. In fact a few years later it was proved by Carlier and Pinson [4] that the record was indeed the optimum. For a recent survey of the problem, see [17].

This proof was done by reducing the search space so far that complete traversal was possible. To achieve this reduction essential use was made of the restriction of one copy per machine type. Hence the temptation is great to pretend to have a job-shop scheduling problem when one in fact does not.

The harvest collection problem thus leads to an important issue in OR: to formulate and solve a suitable generalization of job-shop scheduling. In this paper we leave the harvest collection problem at this point. From now on we are only concerned with the generalization of job-shop scheduling to a practically more relevant form.

**Cardinality constraints, cumulative constraints, and the generalized job-shop scheduling problem.** In the case of one copy of a machine type, the constraint that a machine can only do one thing at at time is easily expressed as a *disjunctive constraint* [12]. For example, activities 1 and 2 with start times $S_1$ and $S_2$ and durations $D_1$ and $D_2$ on the same machine give rise to the disjunctive constraint

$$S_1 + D_1 \leq S_2 \text{ or } S_2 + D_2 \leq S_1.$$

The problem with disjunctive constraints is that they do not generalize to the case of more than one copy of a machine: then it is no longer the case that of two activities on the same machine type one is before the other or the other way around.

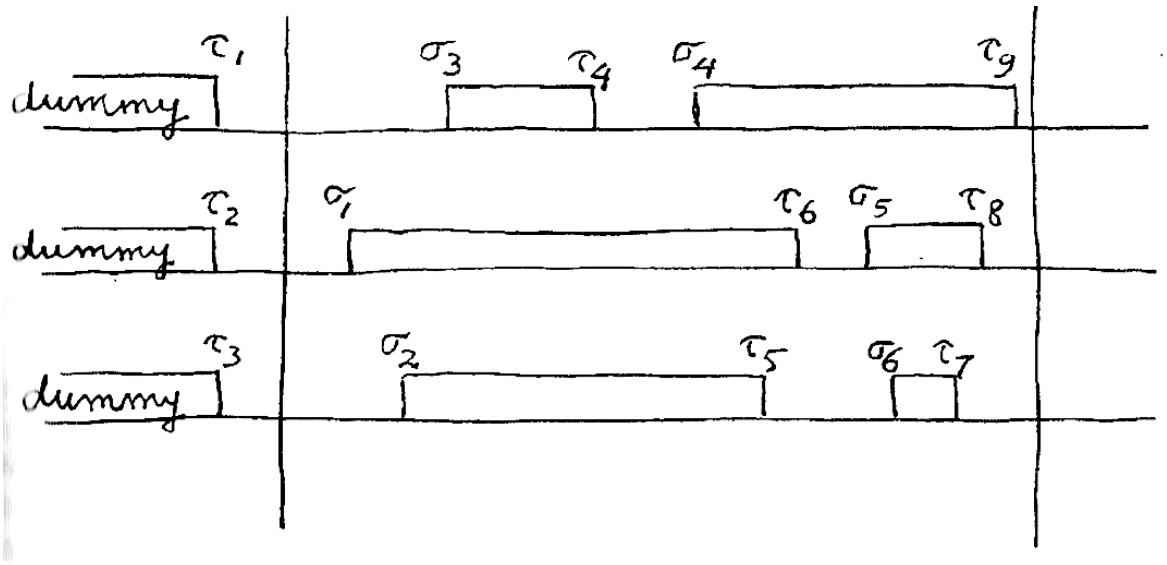
We will refer as JSS to the usual version of the job-shop scheduling problem [3, 9], and as GJSS to the generalized, and practically more common, situation where it is possible that more than one functionally equivalent copy of a machine is available. In JSS "machine" is unambiguous. To avoid confusion in GJSS, we will henceforth avoid the use of "machine" and instead speak either of "machine type" or of "machine copy".

Instead of the usual disjunctive constraint, we have a form of *cardinality constraint*. This case study shows how to solve this type of cardinality constraint by a translation to the type of interval constraints that is built into a logic programming language such a BNR Prolog.

For practically useful solution of scheduling problems one needs both powerful primitives in the programming language to generate constraint systems as well as methods adequate for solving these systems. The cardinality operator of Van Hentenryck and DeVille [13] is a primitive that makes it easy to express cardinality constraints. The work reported here does not help in expressing the problem, but is a solution method.

# 3 Translating the cardinality constraints of GJSS to inequality constraints

**An example.** Figure 1 shows the Gantt chart of a schedule with three functionally




```
j    = 1  2  3  4  5  6
f(j) = 2  3  1  4  6  5
```

Figure 1: Gantt chart for three functionally equivalent copies of a machine type.

equivalent copies of a machine type. There is a horizontal time line for each machine. On it the scheduled activities are shown as strips covering the time periods during which the machine is engaged in the activity.

One way to express the constraint that three copies of the machine type are available and that each copy can be engaged in at most one activity is to choose a finite sequence of time instants and to define for each instant a 0-1 variable for each activity on the machine type, indicating whether a machine copy is engaged in that activity at that time instant. For each time instant the sum of the 0-1 variables should not exceed the number of available machine copies. This leads to an integer linear programming problem. If one wants a reasonably fine time resolution, than it leads to a *large* integer linear programming problem.

In general, integer linear programming problems are NP complete. When such a problem is treated as a continuous linear programming problem, it may happen that the solution comes out in integer values. Then a solution to the integer linear programming problem has been found in a much more efficient manner. Gebotys and Elmasry [10] have successfully followed this approach, using the criterion of unimodality. It may well be that this applies in our case, but we have not investigated this matter.

In our situation a resource is either not used at all, or it is used to the full. Hence the 0-1 variable. In other applications there is a quantitative degree to which a resource is used. Then the 0-1 variables need to be changed to continuous ones, whose sum is constrained not to exceed a given maximum. This leads to the cumulative constraints of Aggoun and Beldiceanu [1]. Note that there are two aspects to cumulative constraints: the language

primitive that makes the problem easy to express and the method for solving the resulting constraint system. We only address the latter aspect, giving a solution method for the special case of cumulative constraints when the resource demand is either 0 or 1.

Because inequality constraints between reals are easier to solve, we consider instead the start times and completion times of the activities. Let $\sigma_1, \ldots, \sigma_6$ be the start times of the activities for the given machine type in nondecreasing order of magnitude. For reasons to be explained later, the completion times of the activities are indexed as $\tau_4, \ldots, \tau_9$ in nondecreasing order of magnitude. Note that because of the differing durations of the activities, the order among the start times gives a different order to the activities than the order among the completion times.

The schedule determines for each activity two things: the machine copy on which it is to be performed and the rank among the activities on this machine copy. For activities that are not first on their machine copy, both determinations can be made by stating the activity immediately preceding it. We prefer to use this method for *all* activities, first activities included. This is accomplished by assuming a dummy activity on each machine copy that completes before the first activity starts. Including these dummy activities we have as completion times $\tau_1, \ldots, \tau_9$ in nondecreasing order.

The completion times can now be treated as *machine availabilities*. Each completion time indicates that a machine is available from that time onwards until the next activity is started on that same machine copy. A schedule can now be described by specifying for each activity $j$ that it is the first to start after a particular element of the sequence of machine availabilities $\tau_1, \ldots, \tau_6$. Let us express this by saying that for each $j$, activity $j$ is the first to start after $\tau_{f(j)}$. The mapping $f$ is such that $j \neq k$ implies that $f(j) \neq f(k)$. In other words, each machine availability can only be used up by one activity. See Figure 1 for the $f$ in the example.

Given $f$ it is now easy to express the machine constraints: for each activity $j$, $\sigma_j \geq \tau_{f(j)}$. Notice also in Figure 1 that we also have the simpler set of inequalities $\sigma_j \geq \tau_j$ for each activity $j$. We next show that these latter inequalities are not a special property of this example, but hold true in general.

**The general case.** Let there be $n$ activities to be scheduled on $k$ copies of the same machine type. The start times of the activities are $\sigma_1, \ldots, \sigma_n$ in nondecreasing order. Including the $k$ completion times of dummy activities before $\sigma_1$, the completion times are $\tau_1, \ldots, \tau_{n+k}$ in nondecreasing order. A schedule is a function $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that such that $j \neq k$ implies that $f(j) \neq f(k)$. A schedule is *feasible* iff for all $j = 1, \ldots, n$, we have that $\sigma_j \geq \tau_{f(j)}$.

**Theorem 1.** A schedule is feasible iff for all $j = 1, \ldots, n$, we have that $\sigma_j \geq \tau_j$.

**Proof.** (If) We assume $\sigma_j \geq \tau_j$. Hence at least $j$ availabilities occurred at or before $\sigma_j$. Because the $\sigma$'s are nondecreasing, $j - 1$ starts occurred before that time. It follows that there is at least one machine copy available at time $\sigma_j$. Hence the schedule is feasible.

(Only if). We assume the schedule to be feasible. As by the assumption on $f$, all of $f(1), \ldots, f(j)$ are different integers not less than one, at least one of these must be greater than or equal to $j$. Hence at least one of $\tau_{f(1)}, \ldots, \tau_{f(j)}$ must be, by the ordering of the $\tau$'s, greater than or equal to $\tau_j$. By the assumption of feasibility,

$$\sigma_1 \geq \tau_{f(1)}, \ldots, \sigma_j \geq \tau_{f(j)},$$

so that at least one of $\sigma_1, \ldots, \sigma_j$ is at least $\tau_j$. As $\sigma_1, \ldots, \sigma_j$ are in nondecreasing order, $\sigma_j$ is such a one. Hence $\sigma_j \geq \tau_j$.

The theorem allows us to translate the integer formulation of the cardinality constraint to an equivalent set of inequalities between reals. This is more powerful for two reasons.

1. The Simplex algorithm for continuous linear programming, though in the worst case also exponential, is in practice vastly more efficient than integer linear programming; on average linear in the number of constraints and logarithmic in the number of variables [5]. And the more recent interior point algorithms for continuous linear programming of Khachian and of Karmaker are polynomial in the worst case and are beginning to be competitive with Simplex in the typical average case [16]. For the simple type of inequality encountered here, interval narrowing may well be faster still, at least in the incremental version required of both methods.

2. The formulation allowed by Theorem 1 is exact. The integer linear programming formulation, even with many variables, is still an approximation that can be improved by yet further increasing the number of variables. The continuous formulation is exact and involves a number of variables that is small compared with even rough approximations to time in the integer linear programming formulation.

Figure 2 shows in diagram form the inequalities constraining the start times and completion times of the activities on three copies of a machine type. The horizontal arrows determine the order in time of the start and completion times. Note that all activities, over all three machines copies, are shown here. Thus a horizontal arrow typically connects start times of activities on *different* machine copies.

# 4 A constraint logic program for GJSS

Let us review what the constraints are when our translation is used.

**Theorem-1 constraints.** We need to schedule all activities for a machine class, subject to the constraint that each machine copy is engaged in at most one activity and that no more than, say, $k$ machine copies are ever engaged at the same time. We saw that according to theorem 1 this is equivalent to the constraint that the $j$-th termination occurs before the $j$-th start of an activity on this machine class. Thus if there are $n$ activities for a given machine class, then there are $n - k$ inequality constraints arising from the limited availability of machines.

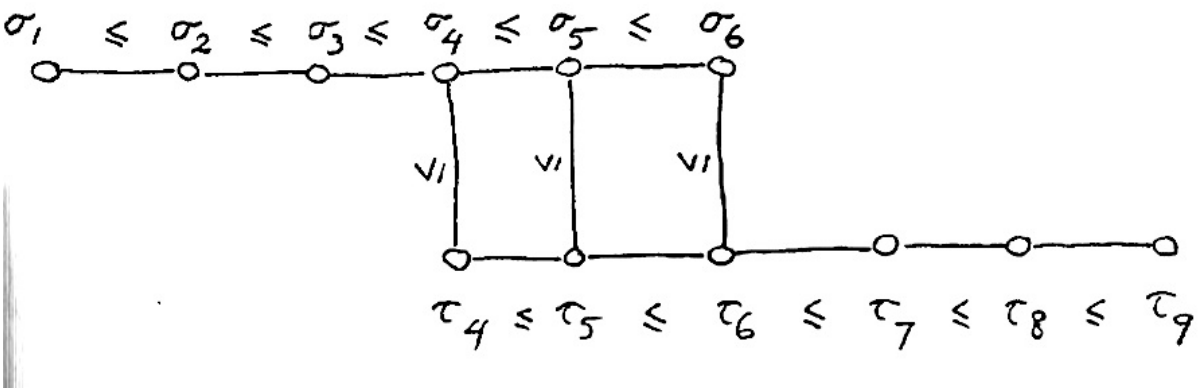



Figure 2: Inequality constraints resulting from translation of cardinality constraint. $\tau_1$,$\tau_2$, and $\tau_3$ are completion times of dummy activities completing before the beginning of the scheduling period.

**The duration constraints.** As our translation forces us to include a variable for the completion time as well as for the start time of each activity, we need to introduce the equality constraint stating that these times differ by the given duration of the activity.

**The job-precedence constraints.** For any two adjacent activities in a job, there is the constraint that the completion time of the earlier one is not greater than the start time of the later one.

**The makespan constraint.** No start time can be earlier than a given time. No completion time can be later than a given time.

**The sortedness constraint.** The variables for the start and completion times that occur in the Theorem-1 constraints refer to these times in chronological order. The variables for these same times that occur in the duration and the job precedence constraints are determined by the identity of the activity, as given in the data of the problem. The order in which the activities occur in the data bear no relation to the order in which the activities start or complete in any of the schedules that are considered when searching for an optimum schedule. Thus we have to have two variables for the start time of each activity and two variables for each completion time.

Suppose now that the variables for the start times for the activities on all machines of the same type occur in a list $M$. To make it possible to use this list to express the Theorem-1 constraints, we have to assume it is in nondecreasing numerical order. Suppose that the variables for the same start times used in the other constraints also occur in a list, say $J$. Then in addition to the constraints described above, we also have the constraint that for each machine type, $M$ is the sorted version of $J$.

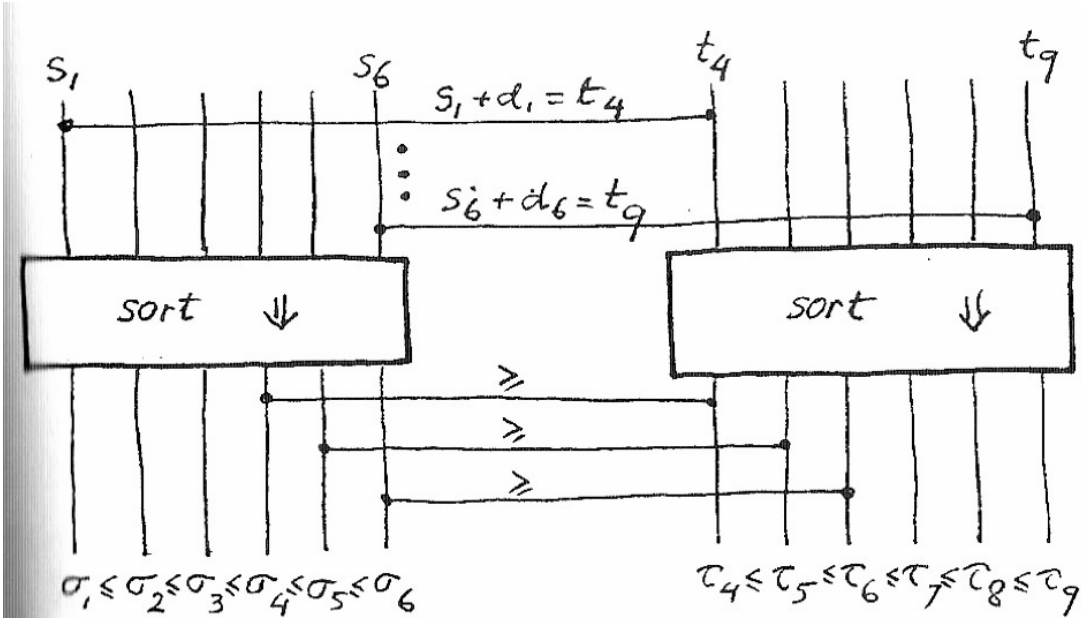



Figure 3: The five constraint types.

Figure 3 is a graphical summary of the five types of constraints resulting from our translation.

# 5 The algorithm

All constraints, except the sortedness constraint, are inequalities between reals. We use the following method to combine these with the sortedness constraint.

**Nondeterministic sorting.** The most natural way to express the sortedness constraint is as a goal in a logic program `sort(J,M)`, stating that list `M` is a sorted version of list `J`. In conventional programming and in standard Prolog this is not possible, as the elements of $J$ are the as yet unknown start times of the activities. We need a nondeterministic form of sorting that sorts with incomplete information on inputs.

Let us first consider the case where the elements of list $J$ are known. The goal `sort(J,M)` unfolds into a sequence of inequalities. Solving these inequalities then results in one permutation, which, when applied to $J$, results in $M$. In interval constraint logic programming, the elements of $J$ are allowed to be specified only by intervals, which can be large. As a result, the goal `sort(J,M)` unfolds into a tree where inequalities between elements of $J$ are the nodes. Each path along the tree specifies a permutation. Prolog's computation rule results in depth-first traversal of this tree.

A typical example of the power of constraint logic programming is that the nondeterministic sorting algorithm just sketched is automatically performed by any of the same

simple sorting algorithms that one finds, for example, in one of the earliest texts on Prolog [6], provided that the list elements are interval variables as allowed in BNR Prolog:

```
sort(X,Y) :- sort(X,Y,[]).
sort([],A,A).
sort([X|Xs],A,C) :-
part(X,Xs,Low,High),sort(Low,A,[X|B]),sort(High,B,C).

part(X,[],[],[]).
part(U,[X|Xs],[X|Low],High) :- X =< U, part(U,Xs,Low,High).
part(U,[X|Xs],Low,[X|High]) :- U  < X, part(U,Xs,Low,High).
```

When activated with the query

```
?- range(X,[1,3]),range(Y,[2,4]),range(Z,[5,7]),range(U,[6,8]),
   sort([X,Y,Z,U],ZZ).
```

the program gives four answers, each containing one of the permutations that is compatible with the initial intervals of `X`, `Y`, `Z`, and `U`. Moreover, in each of these the intervals are narrowed as required by that particular permutation.

```
ZZ = [[2,3],[2,3],[6,7],[6,7]]   permutation:  [Y,X,Z,U]
     [[2,3],[2,3],[5,7],[6,8]]                 [Y,X,U,Z]
     [[1,3],[2,4],[6,7],[6,7]]                 [X,Y,U,Z]
     [[1,3],[2,4],[5,7],[6,8]]                 [X,Y,Z,U]
```

A constraint logic programming language incrementally solves the constraints as they are encountered during the traversal of the tree. As soon as an inconsistent set of constraints has been accumulated when going down a path, backtracking ensues.

**The program.** We post the sortedness constraint after the other constraints. Then, as the tree is traversed, the constraints determining the permutation are considered simultaneously with all other constraints.

The program is activated by the query:

```
?- extract(Begin,End,JL,ML), jobPrecedences(JL),
   machineConstraints(ML,Schedule).
```

We will not be concerned with the details of the first goal. The data for the problem appears in some format of facts and is processed into two lists: the job list `JL` specifying the precedences of activities within each job, and the machine list `ML` specifying the machine type for each activity. The arguments `Begin` and `End` specify the period in which all activities are scheduled. They are needed here because the initial interval for the variables are created here is `[Begin,End]`.

`JL` is a list of the form

```
[[...,[S,D],...],...,[...,[S,D],...]]
```

where each `[S,D]` is the `[start time,duration]` of an activity, and each `[...,[S,D],...]` is the list of activities of a job, in the order as they have to be executed.

`ML` is a list of the form

```
[[JAs,Ss,Ts],...,[JAs,Ss,Ts]],
```

with one element for each machine. `[JAs,Ss,Ts]` consists of three lists. The $i$-th elements of each of the three lists are `[J,A]`, `S`, and `T`, which are the [Job number, Activity number], start time, and completion time of the $i$-th activity assigned to the machine. The $i$-th activity is in the order as the activities are listed in the data. This order is typically not the schedule order.

The essence of the approach via interval constraints is that we create variables standing for the real numbers which are the start times and completion times, even though we don't know these. All these real-valued variables have intervals associated with them within which their true value is known to lie. Initially, these intervals are the entire scheduling period. As constraints are applied, the intervals become smaller.

We proceed with the definitions of the predicates called above.

```
jobPrecedences([]).
jobPrecedences([J|Js])
:- sorted(J),jobPrecedences(Js).

sorted([X]).
sorted([[X,Dx],[Y,Dy]|Zs]) :- X+Dx =< Y, sorted([[Y,Dy]|Zs]).

/* X+Dx is the completion time of the preceding activity and Y is
the start time of the following activity.
*/
```

The meat of the program is in the machine constraints. The main predicate here is `machineConstraints(ML,Schedule)` meaning that `Schedule` is the schedule determined by the data sorted in machine order, given in `ML`. Its definition defines the equivalent of Figure 3 for each machine type. In the following clause body you see the two sort boxes of Figure 3. The Theorem-1 constraints are generated by `postST`. The body does not generate the duration constraints at the top of Figure 3; as the data were prepared in the top-level goal `extract`, it was easiest to include them at that time.

```
machineConstraints([],[]).
machineConstraints([[JAs,Ss,Ts]|Ms],[Sigs|Z])
:- sort(Ss,Sigs),sort(Ts,Taus),
   postST(Sigs,Taus,3),
   machineConstraints(Ms,Z).
```

```
postST(Sigs,Taus,0) :- postST(Sigs,Taus).
postST([Sig|Sigs],Taus,K)
:- K > 0, K1 is K-1, postST(Sigs,Taus,K1).

postST([],_).
postST([Sig|Sigs],[Tau|Taus])
:- /* Theorem 1: */ Tau =< Sig,postST(Sigs,Taus).
```

Here the `sort` goals call a quicksort predicate as defined earlier.

# 6 Solving inequalities by interval constraints

BNR Prolog solves interval constraints by means of an arc-consistency algorithm that generalizes AC-3 [14] to allow relations of arbitrary arity. Such an algorithm can only guarantee that no solution exists outside the resulting intervals; one cannot conclude that a solution exists within. That is, if BNR Prolog terminates with YES, then a solution *may* exist and, if so, is guaranteed to lie in the final intervals. If BNR Prolog terminates with NO, then no solution exists.

It can be shown that the special properties of the constraint network generated by our algorithm do guarantee the existence of a solution. That is, if BNR Prolog terminates with YES with the program shown, then a solution exists.

In general, we have to search for such a solution. That is, one can bisect one of the answer intervals and re-activate the constraint solver and repeat this for the other intervals. Whenever failure results, there is always a possibility to backtrack to another choice of bisected interval. For the constraint network generated by our algorithm this is guaranteed to yield a solution as precise as allowed by the floating-point number system. However, such a search can be very time-consuming.

It can also be shown that the network generated by our algorithm has the *sequential instantiation property* [2]. This guarantees that not only at least one solution exists within the answer intervals, but that there are even so many, that we can instantiate any variable to *any* point within its answer interval, reactivate the constraint solver, and repeat the process for each variable in turn in arbitrary order without incurring failure. Thus, we can extract a point solution from the final intervals without search.

# 7 Conclusions

We started with the problem of scheduling logging trucks. As the obvious formulation in terms of integer linear programming is intractable, we tried to find a translation to another OR problem for which a computationally feasible method exists. This pointed to the job-shop scheduling problem, although this has remained a very difficult problem in spite of much research literature devoted to it. However, we would have to distort our problem to

make it fit the job-shop scheduling problem. An essential feature of our problem translated to multiple copies of the same machine type in JSS. As this is common in many practical situations other than ours, we have formulated a suitable generalization of JSS, which we called GJSS.

This paper shows how the type of cardinality constraint that distinguishes GJSS from JSS can be solved by writing a logic program as simple (and as simple-minded) as a specification and running it under BNR Prolog.

JSS is a hard problem: as small an instance as ten machines, ten jobs and one hundred activities can require extremely sophisticated code that heavily relies on there being no more than one copy of each machine type. This does not prove that GJSS is harder, but it does lend some plausibility that it is so. We believe that the method described in this paper is the first to make any headway with it. The program shown was run on an instance of GJSS with six machines (three types of which two copies each), seven jobs and 23 activities total. It is interesting to note that our naive algorithm specifies in principle trying all permutations of four lists of eight elements and two lists of seven elements, independently of each other, that is, a search space of $(7!)^2(8!)^4 \approx 6.7 \times 10^{25}$ combinations of permutations. The constraints reduce it so far that what remains is traversed in less than a minute on a laptop computer.

The pure GJSS problem is too hard to solve in any practical size. But what we need to solve in practice is the logging truck problem, which is GJSS with a large number of additional constraints. In GJSS, as in JSS, there is no constraint on when an activity is scheduled other than that it obeys the precedences within its job. The corresponding entity in the logging truck problem is a trip by a truck, which is severely constrained in time: it must arrive neither too early nor too late at a site. This translates to GJSS with severe constraints in absolute time in addition to the job precedence constraints. In constraint logic programming, such a variant is easily accommodated: just add the extra constraints.

In deciding whether to buy off the shelf or to commission custom code, CHIP has tilted the balance in favour of the latter option, making it more likely that the real problem gets solved. Interval constraints add a further contribution in this direction. But of course the *really* real problem will remain elusive.

# References

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling problems. *Journal of Mathematical and Computer Modelling*, 17:57–73, 1993.

[2] Frédéric Benhamou and William J. Older. Programming with CLP(BNR): Examples on finite domains. To be published Journal of Logic Programming, 1993.

[3] J. Carlier and P. Chrétienne. *Problèmes d'Ordonnancement.* Masson, 1988.

[4] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35:164–176, 1989.

[5] V. Chvatal. *Linear Programming*. W.H. Freeman, 1983.

[6] W.L. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[7] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint programming language CHIP. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, 1988.

[8] H. Fischer and G.L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In J.F. Muth and G.L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice Hall, 1963.

[9] S. French. *Sequencing and Scheduling*. Ellis Horwood, 1982.

[10] C. Gebotys and M Elmasry. Global optimization approach for architectural synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12:1266–1278, 1993.

[11] P. Van Hentenryck. Personal communication. 1990.

[12] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[13] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logic connective for constraint logic programming. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming*, pages 383–403. MIT Press, 1993.

[14] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[15] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[16] G. Nemhauser and G. Wolsey. *Integer and Combinatorial Optimization*. John Wiley, 1988.

[17] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by local search. *Mathematical Programming B*. To appear.