

OPTIMAL SCHEDULING USING CONSTRAINT LOGIC PROGRAMMING

Ján Paralič, M.S.

Július Csontó, Ph.D.

Milan Schmotzer, M.S.

Department of Cybernetics and Artificial Intelligence

Technical University of Košice

Letná 9,041 20 Košice, Slovak Republic

E-mail: Jan.Paralic@tuke.sk

Abstract

In this paper a methodology to solve scheduling applications (e.g. job-shop) using constraint logic programming (CLP) is presented. Firstly, the CLP problem definition is briefly described. Secondly, a new strategy for finding optimal solution is presented. New strategy consists of three steps. (1) A heuristic capable to find an initial solution of good quality very quickly (upper bound). (2) A heuristic to find a good lower for optimisation. (3) An effective strategy for finding the optimal solution within a minimal number of steps. The results achieved using this methodology on a set of randomly generated job-shop problems are presented.

Keywords: constraint logic programming, optimisation, scheduling, job-shop.

1. Introduction

Constraint logic programming (Jaffar and Lassez, 1987) is a declarative programming paradigm derived from logic programming (PROLOG is the most important representative of the logic programming languages). CLP is particularly useful for solving constraint satisfaction problems, which are formulated as a set of variables and constraints between these variables. The goal is to find such an assignment of values to variables that none of the constraints is violated. In addition, there can be defined a cost function which has to be optimised in the final solution. Typical representative of this class of problems are scheduling applications (e.g. job-shop).

CLP serves powerful tools to handle this kind of problems. Constraints are used actively to prune the search space in an a priori way. Moreover, there are tools to define new constraints and way how the constraints are to be handled by the user. Optimisation is supported as well by predicates which implement the branch and bound strategy. By this strategy often a large number of iterations are needed to find an optimal solution. In this paper a better strategy is presented and the results achieved by solving randomly generated job-shop scheduling problems are shown.

The rest of this paper is organised as follows. In the next section, the CLP problem definition is briefly described on the job-shop scheduling example. Traditional optimisation methods used in CLP are described in section 3. New methods for finding optimal solution in CLP are presented in section 4.

The results achieved using this method to solve a set of randomly generated job-shop problems are presented and analysed in section 5. The paper is closed with a brief summary of the most important results (section 6).

2. Job-Shop Scheduling Using Constraint Logic Programming

2.1 Job-Shop Scheduling Problems

A scheduling problem is defined by a set of tasks and a set of resources. Tasks are constrained by precedence relationships, which bind some tasks to wait for other ones to complete before they can start. Tasks that share a resource are not interruptible (non-preemptive scheduling) and mutually exclusive (disjunctive versus cumulative scheduling). The goal is to find a schedule that performs all tasks in the minimum amount of time.

Job-shop is a special case where tasks are grouped into jobs. Let us suppose there are n **jobs** $\{J_1, J_2, \dots, J_n\}$. Each job is divided to a sequence of m **tasks** $[J_1^1, \dots, J_1^m]$ from which every task has to be processed on different machine. So there are m **machines** $\{M_1, M_2, \dots, M_m\}$ to be considered. Technological constraints demand that there is an ordering between the tasks in particular jobs (the order in the sequence is predefined). Each job has its own ordering of tasks. No two tasks from the same job can be processed simultaneously. The processing times (durations of tasks execution) are independent of the schedule. Such problems are called $n \times m$ job-shop problems.

The terminology of the job-shop scheduling theory arose in the processing and manufacturing industries. But the job-shop definition fits many scheduling problems arising in business, computing, government and the social services as well as those in industry.

2.2 CLP Program to Solve Job-Shop

Job-shop scheduling can be very easily represented as constraint satisfaction problem. Variables are starting times of tasks. Between these variables can be formulated precedence constraints (for tasks inside jobs) and constraints resulted from the shared resources (tasks to be processed on the same machine). Finally, optimisation is often required in form of the minimal finish time of the whole schedule. The core of the CLP program can be represented as follows (after the “%” sign follows commentary).

```
job_shop(Variables) :-
    input_data(List),           % input data - lists of task describing structures
    define_variables(List, Variables, End), % defining variables with their initial
                                     % domains; End represents the finish time
    state_constraints(Variables), % stating precedence and advanced disjunctive
                                     % constraints between Variables
    optimize(disjunction_as_choose(Variables), End). % traditional optimisation
```

First, input data in form of a list of task description data are unified with the variable **List**. Each task is represented by a structure $t(\text{TaskNr}, \text{JobNr}, \text{MachNr}, \text{Duration})$, where

TaskNr is number of this task (i)

JobNr is the number of the job to which this task belongs (j)

MachNr is the number of machine on which this task will be processed (m_{ij})

Duration is duration of this task (D_{ij})

In the next two steps the problem is declaratively defined using variables and constraints between them.

1. Variables

For each task i a variable T_{ij} is defined representing the starting time of the respective task. Initial domain of each variables is an interval of integers $\langle 0, Max_end \rangle$ where Max_end is the sum of durations of all tasks. This corresponds to the worst possible final schedule, if no parallelisation would be possible. There is one more variable (End) for the finishing time of the whole schedule. For each task it holds:

$$End \#>= T_{ij} + D_{ij}$$

2. Constraints

There are two kinds of constraints - *precedence* and *disjunctive* constraints.

- Within a job its tasks must be processed in given order. This means in general that for two tasks k, l of the job j with starting times T_{kj}, T_{lj} and durations D_{kj}, D_{lj} in case that k must be processed before l , it can be represented in CLP language ECLiPSe as numeric constraint:

$$T_{lj} \#>= T_{kj} + D_{kj}$$

- On one machine, tasks from different jobs have to be processed, but at each time there can be processed only one of them. That means for two tasks, i.e. T_{ij} (with duration D_{ij}) and T_{im} (with duration D_{im}) sharing the same machine only one of the two disjoint alternatives will be true:

```
disjunction_as_choice(Tij, Dij, Tim, Dim) :-  
    Tij #>= Tim + Dim.  
disjunction_as_choice(Tij, Dij, Tim, Dim) :-  
    Tim #>= Tij + Dij.
```

These alternatives are tried on backtracking within the search phase inside optimisation. In addition, more redundant constraints can be added in order to improve the constraint propagation and reduce in such a way the search space in advance (see next section for more details).

3. Optimisation

Optimisation is realised in CLP (see next section for a detailed description) using a higher order built-in predicate based on the branch and bound method (`optimize/2` in the program above). This predicate has at least two arguments: a goal, which has to be optimised, and an evaluation function, which has to be minimised. In our case:

- The goal is to find an order of tasks (schedule), i.e. the right selections among disjunctions of task pairs. These are represented by the `disjunction_as_choice/2` predicate.
- Criterion, which is very often used for as optimisation by the job-shop scheduling, is the minimal total length of the schedule. Here it means to find a schedule, where the variable `End` has the smallest possible value (second argument of the `optimize/2` predicate).

3. Traditional Optimisation Methods in CLP

In the CLP optimisation is usually achieved using an higher order predicate incorporating a method known from mathematical programming, namely branch and bound. Predicates with this functionality can be found in CLP languages like *CHIP*, *ECLⁱPS^e*, *cc(FD)* and others.

The basic idea is that branch and bound searches for a solution to the problem and after finding a new solution adds a further constraint that any new solution must be better than the current best one with respect to the evaluation function. This strategy fits also very well with the standard backtracking search of most sequential logic programming systems.

There are essentially two strategies (implemented first in *CHIP* and are also available in *ECLⁱPS^e*) as presented in (Mudambi and Preswitch, 1995):

MIN_MAX Starting with known upper and lower bounds (C^{max} and C^{min} respectively) for the evaluation function C first finds some solution by standard backtracking search, using the initial upper and lower bound as a constraint $C^{min} \#<= C \#<= C^{max}$ to prune the search space. Whenever a new solution is found with cost C_n (an integer number), the search halts and restarts using the tighter constraint $C^{min} \#<= C \#<= C_n - 1$ to further prune the search space. If no further solutions were found or $C_n = C^{min}$, then the last found solution is optimal.

MINIMIZE In this case the process is the same, but after finding a solution procedure does not restart the search continuing further in the search space. On the one hand this approach brings an advantage in comparison with previous approach avoiding wasted effort in many cases because it does not re-traverse the initial empty part of the search tree. On the other hand another problem may occur, called “trashing”. Trashing occurs when many solutions with the same cost are topologically close in the search space, resulting in a lot of wasted search.

Both strategies can be generalised to search for a solution which is optimal within some tolerance E (a number between 0 and 1). This approach can partially prevent trashing and it is in fact often much quicker since it avoids finding many solutions which are only marginally better than the currently best solution. This is usually done so that after finding a new solution with cost C_n , a new upper bound for the evaluation function will be $C_n(E-1)$ (instead of C_n-1). If no better solutions have been found then the optimal solution must have cost within the interval $\langle C_n, C_n E \rangle$.

We have improved these methods in order to minimise the number of iterations as much as possible. The new methods are described in the following section.

4. New Optimisation Methods

The basic idea behind our improved methods is not to decrease only the upper bound of the evaluation function, but take the whole interval $\langle C^{min}, C^{max} \rangle$ and split it in the middle trying this value as new upper bound of the evaluation function. If a solution is found, the upper bound decreases to the value of its evaluation function. If no solution exists, the lower bound increases. In both cases the interval will decrease in each step on the half of it.

Both traditional methods (MIN_MAX and MINIMIZE) can be improved in this way. We call the resulting methods LOGARITHMIC MIN_MAX and LOGARITHMIC MINIMIZE respectively. The algorithm of the first one will be described in the following.

4.1 LOGARITHMIC MIN_MAX

1. Find heuristically a good initial solution (see 4.2) and take *its_cost* as upper bound (*Max*).
2. Determine the lower bound (*Min*).
3. Let $Limit = Max - fix\left(\frac{Max - Min}{2}\right)$.
4. If $Limit = Max$, then let $Limit = Max - 1$.
5. Can it be proven that there exists a solution with cost less than *Limit*? If yes, then let $Max = its_cost$, if not, let $Min = Limit + 1$.
6. Let $Deviation = 100 * \frac{Max - Min}{Min}$.
7. If $Max = Min$ or $Deviation \leq AllowedDeviaton$, find a solution with cost equal or less than *Max* (we know that there exists a solution with such limitation) and stop, otherwise go to step 3.

For a good performance of the algorithm quality of initial upper and lower bounds is very important. Stating upper bound means to find an initial solution using a deterministic heuristic algorithm, i.e. quickly and as good as possible. Stating lower bound means to estimate the distance of a solution to the optimal one (under this bound cannot be the optimal solution). Algorithms we used to get these initial bounds are briefly described in 4.2.

Proof of the solution existence is done by setting a constraint that the cost of the solution cannot be greater than *Limit* and then by finding a solution limited by this constraint (condition). It can be saved some time in the step 5 if the solution found in the step 3 can be stored.

The presented algorithm is called LOGARITHMIC MIN_MAX. In every iteration it divides the interval to half size owing to it needs at about $\log_2(Max - Min)$ proofs of solution existence. The classic (old) versions of MIN_MAX needed $Max - Min - 1$ proofs of solution existence (in the worst case).

Similar works the LOGARITHMIC MINIMIZE algorithm with the same difference as by the MINIMIZE when compared with MIN_MX, i.e. after finding a new - better solution the search continues and does not restart as it does by the LOGARITHMIC MIN_MAX. Both methods were implemented in CLP language ECLPS^e (Schmotzer, 1997). The results we achieved on a number of randomly generated job-shop scheduling problems are presented in section 5.

4.2 Finding Initial Bounds

A classical method for obtaining starting solution is to use priority-dispatching rules (Caseau and Laburthe, 1995): the schedule is constructed chronologically, tasks are selected one after the other and performed as soon as possible. The algorithm works as follows: At each step, a set of "selectable" tasks is kept. In the beginning, this set initialised to the set of all tasks that are first in a job (i.e. tasks that do not require any other task to be performed before them). One of the tasks in this set is selected and scheduled as soon as possible on its machine. It is

then removed from the set of selectable tasks and replaced in this set by its direct successor in the job. This process is repeated until no more tasks are to be selected.

The whole algorithm depends on the selection rule. We tested eight different priority rules and derived a new one with the best average behavior (Paraliè, 1997). It is a combination of EST (select the task with the earliest starting time) and in case when more tasks are equal with respect to this criterion, select the one with the MWKR (most work remaining - the task which has the longest sum of durations of tasks to be processed in the job after it).

For the lower bound one needs to compute the sum of durations of all tasks within the same job and for all tasks to be processed on the same machine. From these sums one can choose the greatest one as lower bound. We further increased this sum adding the smallest possible starting time from the respective tasks within the chosen job (machine).

5. Results

We tested our methods on a number of randomly generated job-shop scheduling problems of different size. In Table 1 the results of twenty such problems are presented. Ten problems are of size 8 x 8 (8 jobs, 8 machines) and ten of size 10 x 10. For each problem (numbered from 0 to 10) the resulted time (in seconds on PC AMD5x86 133MHz running under Linux operating system) and the number of backtracks necessary for finding the optimal solution to the problem is given. Number of backtracks (i.e. number of generated failures, or bad schedules) is a good measure to see how large was the searched part of the search space. The lower is this number the better method (the more straightforward looks for an optimal solution).

From the traditional methods only the MIN_MAX could be used for these problems (MINIMIZE suffered from trashing). The results are compared with the two new optimisation methods. The best results for each problem are represented in the table using bold font. In all cases the same initial lower and upper bounds were used.

As we can see in most of the cases at least one of the two new methods was better than the traditional one in both merits, i.e. with respect to the computational time as well as the number of backtracks. More precisely LOGARITHMIC MIN_MAX achieved the best results with respect to the number of backtracks, namely by the 8 x 8 problems it was 11,2 times and by 10 x 10 problems 10,2 times better than MIN_MAX in average. LOGARITHMIC MINIMIZE had best results with respect to the computational time. It was in average 5,7 times faster by the 8 x 8 problems and 7,2 times faster by the 10 x 10 problems than traditional MIN_MAX.

There are some instances of the job-shop problem, where the proof of the optimality is very hard (proof of optimality is the search through the whole search space with the constraint that the value of the evaluation function must be lower than the one of the optimal solution - this is necessary to show, that it is in fact optimal). In these cases new methods need more backtracks due to more searches under the optimal value. We improved our strategy for this type of problems in such a way, that in case the current interval between the lower and upper bound is smaller than 3, the strategy continues like MIN_MAX decreasing the upper bound. In such a way we achieve better performance in all tested instances of the job-shop.

Table 1 A comparison of traditional optimisation method MIN_MAX with two new methods LOGARITHMIC MIN_MAX a LOGARITHMIC MINIMIZE.

problem	MIN_MAX		LOG_MIN_MAX		LOG_MINIMIZE	
	time (s)	BT	time (s)	BT	time (s)	BT
8_8_0	30.75	69	12.74	96	10.25	746
8_8_1	40.63	384	5.55	95	5.89	319
8_8_2	98.50	1331	48.71	774	32.65	1238
8_8_3	70.34	2019	7.90	79	9.99	744
8_8_4	68.70	3654	5.95	56	5.47	676
8_8_5	114.01	6534	24.26	467	21.51	873
8_8_6	31.13	50	3.80	6	3.41	452
8_8_7	42.06	356	9.55	104	11.01	755
8_8_8	29.75	73	12.21	173	12.50	604
8_8_9	109.03	1813	34.84	667	30.94	1073
10_10_0	808.42	13517	129.45	1245	108.28	2130
10_10_1	547.48	10287	51.69	288	32.65	883
10_10_2	1064.87	20233	499.71	8446	351.28	8037
10_10_3	1010.60	19271	324.59	3965	289.85	4851
10_10_4	2307.45	1117019	128.99	1907	156.26	4506
10_10_5	382.99	8450	73.73	830	56.30	3286
10_10_6	126.64	1425	19.92	148	18.68	1043
10_10_7	10647.00	116817	10239.50	151256	8404.28	150821
10_10_8	590.05	27063	99.97	1846	89.90	2577
10_10_9	6256.24	111350	2565.99	38861	1228.24	29234

6. Summary

In this paper methods to solve job-shop scheduling problems using constraint logic programming approach are described and analysed. The traditional methods MIN_MAX and MINIMIZE have been improved in order to minimise necessary iterations during the search phase. As a result the performance of the two new methods is much better with respect to the computational time (especially the LOGARITHMIC MINIMIZE) as well as with respect to the number of backtracks (especially the LOGARITHMIC MIN_MAX).

References

- Caseau, Y. and Laburthe, F. (1995), "Disjunctive Scheduling with Task Intervals", LIENS Technical Report 95-25, Laboratoire d'Informatique de l'Ecole Normale Supérieure, 1995.
- Jaffar, J. and Lassez, J. L. (1987), "Constraint Logic Programming", Proc. of the 14th Symp. POPL'87, Munich, Jan. 1987, pp. 111-119.
- Paraliè, J. (1997), "Solving Scheduling Problems Using Constraint Logic Programming" (in Slovak), Ph.D. thesis, Technical University of Košice, April 1997.
- Schmotzer, M. (1997), "Analysis and Design of New Methods to Solve Time Scheduling Using Constraint Logic Programming", Master's thesis, Technical University of Košice, April 1997.