Constraint-Based Optimization and Approximation for Job-Shop Scheduling*

Philippe Baptiste and Claude Le Pape and Wim Nuijten

ILOG S.A., 2 Avenue Gallieni, BP 85, F-94253 Gentilly Cedex, France
Email: {baptiste, lepape, nuijten}@ilog.fr
Url: http://www.ilog.fr

Abstract

We present constraint-based optimization algorithms and a constraint-based approximation algorithm for the job-shop scheduling problem. An empirical performance analysis shows that both the optimization algorithms and the approximation algorithm perform well. Especially the approximation algorithm is among the best algorithms known to date. We, furthermore, show that we can improve the performance of the optimization algorithms by combining them with the approximation algorithm. In particular, this combination allows us to solve an instance that was not yet reported to be solved.

1 Introduction

An important issue in a manufacturing environment is the improvement of resource utilization. A classical way of achieving improved resource utilization is by using scheduling algorithms. As defined by Baker [1974], scheduling is concerned with the problem of allocating scarce resources to activities over time. In this paper we focus on one of the best known scheduling problems, viz., the *Job Shop Scheduling Problem* (JSSP). In the JSSP one is given a set of jobs and a set of resources. Each job consists of a set of activities that must be processed in a given order. Furthermore, each activity is given an integer processing time and a resource on which it has to be processed. Once an activity is started, it is processed without interruption and a resource can process at most one

*In: Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI-95, Montreal, Canada, 1995.

activity at a time. A schedule specifies a start time for each activity. In the optimization variant of the JSSP one is asked to find a schedule that minimizes the *makespan*, i.e., the maximum completion time of the activities.

Optimization algorithms for the JSSP proceed by branch-and-bound. Most lower bound calculations on the makespan are based on the relaxation of the capacity constraints of all resources except one, resulting in a single-machine scheduling problem with release times and deadlines. This single-machine bound has been strengthened by a number of people among which Carlier & Pinson [1989], Brucker, Jurisch & Sievers [1992], Nuijten [1994], and Caseau & Laburthe [1994]. Until recently, approximation algorithms for the JSSP used *dispatch rules*, which schedule the activities according to some priority function; for an overview see [Panwalkar & Iskander, 1977] and [Haupt, 1989]. In the last decade, many variants of local search have been proposed for the solution of the JSSP. Iterative improvement, simulated annealing, and threshold accepting were used by Aarts, Van Laarhoven, Lenstra & Ulder [1994], Matsuo, Suh & Sullivan [1988], and Van Laarhoven, Aarts & Lenstra [1992]. Tabu search was used by Taillard [1989], Barnes & Chambers [1991], Dell'Amico & Trubian [1993], and Nowicki & Smutnicki [1993]. Genetic local search was used by Aarts, Van Laarhoven, Lenstra & Ulder [1994] and Dorndorf & Pesch [1992]. Adams, Balas & Zawack [1988] developed an approach that is a combination of a constructive approach and iterative improvement. Variants of this approach are the *bottle-t* and *shuffle* algorithms proposed by Applegate & Cook [1991].

In constraint-based scheduling the usual way of solving problems is by using systematic tree search. In this way constraint-based scheduling stays close to optimization algorithms. In this paper we present both constraint-based optimization algorithms and a constraint-based *approximation* algorithm. This approximation algorithm uses two important ideas from [Applegate & Cook, 1991] and [Nuijten, 1994]. All algorithms are built on top of ILOG SCHEDULE [Le Pape, 1994], an add-on to ILOG SOLVER, a C++ library for constraint programming [Puget, 1994]. The crucial component of ILOG SCHEDULE for this approach is its extensive propagation for disjunctive constraints as discussed by [Baptiste & Le Pape, 1995].

2 Some Optimization Algorithms

In ILOG SCHEDULE, the propagation of temporal constraints, i.e., precedence relations with or without minimal and maximal delays between activities, is *complete* [Baptiste, Le Pape & Nuijten, 1995]. This means that propagating the constraints is sufficient to determine whether a set of temporal constraints is consistent. The propagation also determines earliest and latest start and end times for activities that are globally consistent with all the temporal constraints.

As a result, solutions to the JSSP can be obtained by sequencing all the activities that require a common resource. Given an upper bound for the makespan, the following algorithm either finds a solution to the JSSP or proves that no solution exists:

1. Select a resource among the resources required by unordered activities.
2. Select the activity to execute first (or last) among the unordered activities that require the chosen resource. Post and propagate the corresponding precedence constraints. If an inconsistency is detected, a backtrack occurs. Keep the other activities as alternatives to be tried upon backtracking.
3. Iterate step 2 until all the activities that require the chosen resource are ordered.
4. Iterate steps 1 to 3 until all the activities that require a common resource are ordered.

To minimize makespan, this search algorithm merely needs to be encapsulated in a `CtMinimize` SOLVER statement. `CtMinimize` is a SOLVER function which receives as its arguments a search algorithm and a criterion to minimize. As long as the search algorithm succeeds in generating a solution to the problem, `CtMinimize` adds a new constraint stating that the value of the criterion must be strictly smaller than the value of the best solution found so far. When the search algorithm fails, i.e., reports that there is no better solution, it is known that the best solution found is optimal.

An alternative to `CtMinimize` consists in performing a binary search that dichotomizes the interval where the possible values of the makespan occur. Given a lower bound lb and an upper bound ub for the optimal makespan, we try to solve the problem with a new constraint stating that the makespan must be at most $(lb + ub)/2$. If this succeeds, ub becomes the makespan of the solution that was found; if this fails, lb becomes $((lb + ub)/2) + 1$. When lb and ub are equal, the optimal makespan has been found.

Coming back to the algorithm described above; heuristics are used to select the resource (step 1) and the activity to execute either first or last among the unordered activities that require the resource (step 2). We have implemented the following heuristics.

Resource Selection. The *slack* of each resource is computed. The slack is defined as the minimal difference between *supply* and *demand* over each time interval $[EST, LET)$, where EST and LET denote the minimal earliest start time and the maximal latest end time of any of the unordered activities. The resource supply over $[EST, LET)$ is $LET - EST$, which reflects the fact that the resource can perform only one activity at a time. The demand over the interval $[EST, LET)$ is the sum of the durations of the activities that must execute between EST and LET. The resource with the smallest slack is selected.

Activity Selection. There are two things to consider in this step. First, we need to decide whether we select an activity to execute first or an activity to execute last among the unordered activities, and second which activity is indeed selected. Several alternatives may be considered: (a) always selecting an activity to execute first; (b) always selecting an activity to execute last; (c) deciding between first and last with respect to a dynamic criterion like the number of activities that can be scheduled first and the number of activities that can be scheduled last. Indeed, the earliest and latest start and end times computed by constraint propagation can be used to determine that some activities cannot be first and that some activities cannot be last [Carlier & Pinson, 1989] [Baptiste & Le Pape, 1995]. When the number of activities that can be first is smaller than the number of activities that can be last, we may rather select one of the possible firsts to avoid branching on a high number of possibilities. Similarly, when the number of activities that can be last is smaller than the number of activities that can be first, we shall rather select one of the possible lasts. This may be particularly important when there are very few solutions: it is less likely to make a mistake when a choice is made between few alternatives. However, this argument may be balanced by the fact that we noticed that scheduling always from the same end may allow better exploitation of the constraint propagation that takes place. As the choice was unclear, we tried all three alternatives (a), (b), and (c).

The activity to schedule first is selected according to the following EST-LST rule: the activity with the smallest earliest start time is chosen; in case of ties, the activity with the smallest latest start time is chosen. A symmetric heuristic, LET-EET, applies when selecting an activity to execute last. When the number of possible firsts equals the number of possible lasts and strategy when (c) is used, the tie is broken by looking at the difference between the best activity F_1 and the second best activity F_2 according to the EST-LST rule, and at the difference between the best activity L_1 and the second best activity L_2 according to the LET-EET rule. If the difference between F_1 and F_2 (according to EST-LST) is greater than the difference between L_1 and L_2, F_1 is selected to execute first; otherwise, L_1 is selected to execute last.

Computational results. Tables 1, 2, and 3 give the results obtained on the ten 10x10 JSSP instances used in the computational study of Applegate & Cook [1991]. Table 1 gives the results obtained by always selecting an activity to execute first. Table 2 gives the results obtained by always selecting an activity to execute last. Table 3 gives the results obtained by dynamically deciding between first and last. In both cases, a binary search that dichotomizes on the possible values of the makespan is used. The columns "BT" and "CPU" give the total number of backtracks and CPU time needed to find an optimal solution and prove its optimality. Columns "BT(pr)" and "CPU(pr)" give the number of backtracks and CPU time needed for the proof of optimality. Column "CPU(1)" gives the CPU time needed to find the first solution, including the time needed for stating the problem and performing initial constraint propagation. All CPU times in this paper are given in seconds on a RS6000 workstation. We remark that none of the three strategies dominates the others. Over the ten instances, the total number of backtracks is 579711 for strategy (a), 416971 for strategy (b) and 501174 for strategy (c). We remark that for each of the three strategies, the difference between different instances is very important.

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)
MT10	.4	69758	1076.4	7792	126.8
ABZ5	.3	17636	218.1	5145	62.6
ABZ6	.3	898	15.2	291	4.7
LA19	.3	21910	293.3	5618	75.8
LA20	.3	74452	845.9	22567	249.2
ORB1	.4	13944	222.0	5382	84.7
ORB2	.3	114715	1917.2	30519	500.9
ORB3	.4	190117	3193.8	25809	449.0
ORB4	.3	64652	1131.2	22443	395.2
ORB5	.3	11629	172.8	3755	55.0

Table 1: Results on ten 10x10 instances of the JSSP with strategy (a)

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)
MT10	.4	14076	199.1	4181	58.6
ABZ5	.3	14747	205.5	3562	51.2
ABZ6	.3	2186	28.7	822	10.4
LA19	.3	25906	350.9	7935	109.9
LA20	.3	53773	612.3	18044	193.5
ORB1	.3	95802	1569.6	23107	385.8
ORB2	.3	33557	525.7	12385	194.2
ORB3	.4	151136	2451.8	35945	571.5
ORB4	.3	14524	174.7	1841	22.4
ORB5	.3	11264	152.5	2522	34.8

Table 2: Results on ten 10x10 instances of the JSSP with strategy (b)

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)
MT10	.3	12844	188.1	4735	66.9
ABZ5	.3	17992	247.5	4519	62.2
ABZ6	.3	1116	16.4	312	4.7
LA19	.3	22235	312.7	6561	92.0
LA20	.3	95611	1126.1	20626	231.1
ORB1	.3	24749	423.9	6261	109.6
ORB2	.3	37982	647.7	14123	234.4
ORB3	.3	269697	4160.9	22138	351.6
ORB4	.3	9770	125.5	1916	24.1
ORB5	.3	9178	130.1	2658	37.4

Table 3: Results on ten 10x10 instances of the JSSP with strategy (c)

3 An Approximation Algorithm

The results obtained by the optimization algorithms are quite good. However, the overall CPU time varies a lot from one instance to the other. By examining a number of runs more closely, we found two explanations for that. First, the dichotomizing strategy may lead to proving several times the quasi-optimality of a solution. Second, and more surprisingly, we found that a significant portion of the CPU time is spent finding solutions relatively far from the optimal value. For example, on the MT10 instance, strategy (a) spends nearly half of the total CPU time finding a schedule of makespan 968, while the optimal value is 930. Our interpretation of this phenomenon is that when the upper bound is much higher than the optimal makespan, there are many solutions, making it easy to find one. Furthermore, when the upper bound is very close to the optimal makespan, constraint propagation becomes very effective in pruning the search space. Then, although it is not *easy* to find a solution, constraint propagation provides reliable guidance. However, for intermediate values of the upper bound, it is fairly difficult to find a solution and constraint propagation does not provide much guidance. This increases the probability of taking a wrong decision and as a systematic search strategy is used, it may take long to recover from such a mistake.

An alternative to an optimization algorithm is an approximation algorithm. As discussed in Section 1, many types of approximation algorithms have been applied to the JSSP. We used two important ideas from [Applegate & Cook, 1991] and [Nuijten, 1994] to develop our own approximation algorithm.

Applegate & Cook [1991] use the *shuffle* procedure to improve solutions to the JSSP. The basic idea is to fix a number of decisions, based on the best solution found so far, and search for an optimal schedule among those that respect those decisions. This procedure gives very good results, especially when effective constraint propagation techniques are used. Indeed, the propagation of the imposed decisions results in a drastic reduction of the search space and thus enables a fast resolution of the remaining subproblems.

Nuijten [1994] uses a randomized procedure to solve a wide range of scheduling problems. Of this procedure we use the way to escape from dead ends. The employed approach consists of two parts. First, chronological backtracking is used in order to solve the instance at hand. However, if this does not lead to a solution after a reasonable number of backtracks, the search is restarted. A problem then is to direct the search along a path different from the ones followed previously. Therefore, the restarting of the search is combined with randomized selection strategies. In this way, the probability of following the same search path more than once is very small since the number of possible paths is usually very large.

Our algorithm combines the two ideas as follows. At each step, a number of ordering decisions are kept. As in [Applegate & Cook, 1991], we search for an optimal schedule among those that respect those decisions. However, the decisions that are kept are randomly selected and the search is stopped after a given number of backtracks. More precisely, our algorithm works as follows.

1. Generate an initial solution by solving the constraint satisfaction problem

in which the makespan is not constrained. The propagation of temporal constraints is complete and thus a first solution is obtained quickly and without backtracking.

2. For at most N iterations without improvement, the following subroutine is applied:
   - Constrain the makespan to be lower than the best known makespan.
   - Keep randomly some ordering decisions from the previous schedule. For each pair of activities (A B) consecutively scheduled on the same resource, the ordering decision "A before B" is kept with a probability p.
   - Search for a solution using a limited number of backtracks NB: a `CtMinimize` is launched on a problem-solving algorithm equivalent to the one described in Section 2. However, the `CtMinimize` is bound to return after NB backtracks. If a new solution is found then it is recorded as the new best solution, and the algorithm restarts step 2. Otherwise, a new iteration is attempted.
3. Step 2 terminates when the solution has not been improved for N iterations. Then the probability p is decreased. If the probability p is greater than a given threshold p_0, proceed to step 2. Otherwise return the best solution found so far.

We further refined the algorithm by introducing four different activity selection heuristics. Each of these heuristics is tried in turn in step 2 of the algorithm. Step 2 terminates when N iterations of each of the heuristics have failed to produce a solution better than the best available solution. The four activity selection heuristics are the following.

- The first one is the strategy referred as (c) in Section 2. This strategy dynamically determines whether to select an activity to schedule first or last. The activity to schedule first is chosen according to the EST-LST rule. The activity to schedule last is chosen according to the LET-EET rule. Ties are broken as explained in Section 2.
- The second strategy also dynamically determines whether to select an activity to schedule first or last. However, the activity to schedule first is chosen according to another rule being the EDD-EST rule. With this rule the activity with the smallest latest end time is chosen, and in case of ties, preference is given to the activity with the smallest earliest start

time. The symmetric rule, LRD-LET, is used for determining the activity to schedule last.

- The third strategy always selects an activity to execute first. This activity is selected randomly among the activities that can be scheduled first.
- The fourth strategy always selects an activity to execute last. This activity is selected randomly among the activities that can be scheduled last.

Computational results We first applied our approximation algorithm to the ten instances of [Applegate & Cook, 1991]. For each instance we did five runs using different seeds for the random number generator. We chose $N = 10$, $NB = 100$, $p = 0.6$, and had p decreased by steps of 0.6 down to $p_0 = 1/(2 \cdot m)$, where m is the number of resources. The first run we found an optimal solution for seven out of the ten instances. The second run found an optimal solution for two of the three remaining instances. The average mean relative error (MRE) over three runs was 0.1%. This means that on average, the algorithm provided a solution within 0.1% of the optimal solution. The CPU time for executing the complete algorithm varied between 90 and 180 seconds. On average, the best solution was found after 72.5 seconds of CPU time. This is of course much smaller than the CPU times obtained with the optimization algorithm.

Instance	LB/UB	AVG(5)	BEST(5)	CPU
MT10	930	930	930	187.6
LA02	655	655	655	4.1
LA19	842	843	842	173.8
LA21	1046	1061	1046	610.9
LA24	935	941	940	430.8
LA25	977	980	977	376.8
LA27	1235	1269	1254	1012.1
LA29	1130/1157	1214	1196	1320.7
LA36	1268	1269	1268	633.8
LA37	1397	1397	1397	233.8
LA38	1196	1216	1207	1071.9
LA39	1233	1235	1233	805.6
LA40	1222	1234	1229	947.2

Table 4: Results on thirteen instances of the JSSP

We also applied the algorithm to the thirteen instances used by Vaessens, Aarts & Lenstra. The parameters were unchanged except for p_0 that was set to

$1/m$ to accomodate instances with a high number of jobs in a reasonable amount of time. Table 4 gives the results obtained. In this table, the column "LB/UB" gives the best known lower and upper bound on the minimum makespan. If only a single number is given, both bounds coincide resulting in an optimal value of the makespan. Column "AVG(5)" provides the average makespan obtained over five runs of the algorithm. Column "BEST(5)" provides the best makespan obtained after the same five runs. Column "CPU" provides the average CPU time.

These results significantly improve on those reported in [Nuijten, 1994] for a similar approach. The average MRE over five runs is 1.20% and the MRE for the best of five runs is 0.72%. Even if we use the lower bounds used in [Vaessens, Aarts & Lenstra, 1994], we get an MRE of 0.92% for the best of five runs. Only one of the algorithms used in [Vaessens, Aarts & Lenstra, 1994] achieves an MRE smaller than 1 % percent, namely the tailored taboo search algorithm of Nowicki & Smutnicki [1993] for which the MRE is 0.54%.

4 Combining Approximation and Optimization

We finally combined the approximation and optimization algorithms as follows. When the approximation algorithm terminates, the optimization algorithm is launched using `CtMinimize` and the strategy referred to as (c) in Section 2. Table 5 gives the average results obtained over three runs of the ten 10x10 instances of Applegate & Cook. The total number of backtracks over the ten instances is 215256, which significantly improves on the results reported in Section 2.

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)
MT10	.3	13684	235.8	4735	67.3
ABZ5	.3	19303	282.1	4519	61.3
ABZ6	.3	6227	100.6	312	4.7
LA19	.4	18102	269.5	6561	91.0
LA20	.3	40597	496.7	20626	227.2
ORB1	.3	22725	407.3	6261	108.0
ORB2	.3	31490	507.1	14123	228.7
ORB3	.4	36729	606.1	22138	342.6
ORB4	.3	13751	213.7	1916	23.7
ORB5	.3	12648	210.9	2658	36.5

Table 5: Results on ten 10x10 instances of the JSSP

5 Conclusion

We showed that the algorithms we presented perform well. In particular, on a well studied set of instances one single approach known to date performs slightly better than our constraint-based approximation algorithm. We, furthermore, showed that by using this approximation algorithm, we can improve the performance of the optimization algorithms. We remark that this combination allowed us to solve the LA21 instance to optimality. To the best of our knowledge, the optimal makespan for this instance was not yet published anywhere, although we know that various people succeeded to obtain the same result. However, the number of backtracks that the proof of optimality took, being 3.955.036, has inspired us to do research in which we focus on improving the propagation algorithms that we use. We expect that the theoretical results we obtained in that respect, will lead to even better computational results fairly soon.

References

Aarts, E.H.L., P.J.M. van Laarhoven, J.K. Lenstra, and N.J.L. Ulder [1994], A computational study of local search algorithms for job shop scheduling, *ORSA Journal on Computing* **6**, forthcoming.

Adams, J., E. Balas, and D. Zawack [1988], The shifting bottleneck procedure for job shop scheduling, *Management Science* **34**, 391–401.

Applegate, D., and W. Cook [1991], A computational study of the job-shop scheduling problem, *ORSA Journal on Computing* **3**, 149–156.

Baker, K.R. [1974], *Introduction to Sequencing and Scheduling*, Wiley & Sons.

Baptiste, P., and C. Le Pape [1995], A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling, *Proc. 14th International Joint Conference on Artificial Intelligence.*

Baptiste, P., C. Le Pape, and W. Nuijten [1995], Incorporating efficient operations research algorithms in constraint-based scheduling, *Proc. 1st International Joint Workshop on Artificial Intelligence and Operations Research.*

Barnes, J.W., and J.B. Chambers [1991], *Solving the Job Shop Scheduling Problem using Tabu Search*, Technical Report ORP91-06, University of Texas, Austin.

Brucker, P., B. Jurisch, and B. Sievers [1992], *A Branch & Bound Algorithm for the Job-Shop Scheduling Problem*, Working document, University of Osnabrück, Germany.

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

Caseau, Y., and F. Laburthe [1994], Improved CLP scheduling with task intervals, *Proc. 11th International Conference on Logic Programming.*

Dell'Amico, M., and M. Trubian [1993], Applying tabu-search to the job-shop scheduling problem, *Annals of Operations Research* **41**, 231–252.

Dorndorf, U., and E. Pesch [1992], *Evolution Based Learning in a Job Shop Environment*, Working document, University of Limburg, the Netherlands.

Haupt, R. [1989], A survey of priority rule-based scheduling, *OR Spektrum* **11**, 3–16.

Laarhoven, P.J.M. van, E.H.L. Aarts, and J.K. Lenstra [1992], Job shop scheduling by simulated annealing, *Operations Research* **40**, 113–125.

Le Pape, C. [1994], Implementation of resource constraints in ilog schedule: A library for the development of constraint-based scheduling systems, *Intelligent Systems Engineering* **3**, 55–66.

Matsuo, H., C.J. Suh, and R.S. Sullivan [1988], *A Controlled Search Simulated Annealing Method for the General Job Shop Scheduling Problem*, Working Paper 03-04-88, Graduate School of Business, University of Texas at Austin, Austin, USA.

Nowicki, E., and C. Smutnicki [1993], *A Fast Taboo Search Algorithm for the Job Shop Problem*, Preprinty nr. 8/93, Instytut Cybernetyki Technicznej, Politechnicki Wroclawskiej, Poland.

Nuijten, W.P.M. [1994], *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*, Ph.D. thesis, Eindhoven University of Technology.

Panwalkar, S.S., and W. Iskander [1977], A survey of scheduling rules, *Operations Research* **25**, 45–61.

Puget, J.-F. [1994], *A C++ Implementation of CLP*, Technical Report 94-01, ILOG S.A., Gentilly, France.

Taillard, E. [1989], *Parallel Taboo Search Technique for the Job Shop Scheduling Problem*, Internal Report ORWP 89/11, Département de Mathématique, Ecole Polytechnique Fédérale de Lausanne.

Vaessens, R.J.M., E.H.L. Aarts, and J.K. Lenstra [1994], *Job Shop Scheduling by Local Search*, COSOR Memorandum 94-05, Eindhoven University of Technology.