Belgian Journal of Operations Research, Statistics and Computer Science Vol. 33 (1,2)

A job shop scheduling with set–up time

Ph. Fortemps

Faculté Polytechnique de Mons
21, Rue de Houdain
B-7000 Mons, Belgium

Abstract

Job shop scheduling is a well–known NP–hard problem. In this paper we consider such a problem with class–dependent set–up time. We use several heuristics to solve this problem : genetic algorithms, tabu search and a learning classifier system.

Keywords : Job shop scheduling, heuristics, genetic algorithms, tabu search, learning classifier systems

1 Introduction

There are lots of applications of Machine Scheduling e.g. in manufacturing but also in other fields like computer architecture. A number of recent surveys on the topic are [2], [4], [14] & [1]. One important point is the *'solvability'* of a problem, i.e. its status as P or NP-hard according to the theory of intractability [9]. A lot of scheduling problems are classified as NP-hard; thus, the time needed to solve them in an exact way grows exponentially (or even more rapidly) with their size. The challenge is to solve them more quickly than these exact algorithms, and yet obtain an acceptably good solution. That's the reason why we use *'approximation algorithms'* or *'heuristics'*, which try to find a sub-optimal solution in a reasonable time.

Further in this section we'll introduce a three-field notation for describing scheduling problems. In section II, we present the problem we want to solve. It is a real-world industrial job shop scheduling problem. A first feasibility study has been done in [3], on test sets of data. Our main contribution is to take into account the setup times of the different processors; our algorithms are tested on real industrial data.

In section III, we present the approach we followed to the scheduling problem: a basic algorithm called 'scheduler' is designed to produce a schedule provided a series of conflicts between tasks are solved by an external rule. Such rules can be the priority lists described in section IV and used with genetic algorithms [12, 10] and tabu search [11] heuristics in section V. Another approach (section VI) consists in letting a learning classifier system [13] manage and optimize a set of rules.

Finally, section VII contains some remarks on the results and the comparison of the three different methods.

1.1 Scheduling Problems

A set of tasks $\Theta = \{T_1, \ldots, T_n\}$ have to be processed on a set of processors $\Pi = \{P_1, \ldots, P_m\}$, in accordance with two constraints: first, a task T_i may not be processed by more than one processor at a time; then, each processor P_j is able to processing at most one task at a time.

If we assume that each processor is specialized for some kinds of operation, we may have three types of systems: *Flow Shop, Open Shop and Job Shop* . We'll present them briefly.

Each task T_i consists in a set of operations $\{O_{i1}, \ldots, O_{ik_i}\}$ and each operation may need a different processor, according to the abilities of the latter.

In the case of a *Flow Shop*, the tasks have the same number of operations, which is equal to m, the number of processors. Moreover, the operations of a task are ordered in a sequence. In that sequence, for each task i, the operation j has always to be processed on the processor P_j and O_{ij} may not begin before $O_{i(j-1)}$ is completed.

An *Open Shop* system is like a flow shop, but the order of processing operations is not specified. O_{ij} (on processor P_j) may be processed before $O_{i(j-1)}$.

In a *Job Shop* problem, all the characteristics are arbitrary but must be given in advance: the number of operations of task i (i.e. k_i), their assignment to the unique possible processor and the order of their processing may differ from those of task $(i+1)$ but have to be given. The Job

Shop problem is proved to be NP-hard [9]. Figure 1 gives an example of Job Shop Schedule for the following three tasks example:

Task A : 2 hours on P1, 1 hour on P2, 1 hour on P3
Task B : 5 hours on P1, 2 hours on P2
Task C : 1 hour on P3, 2 hours on P1

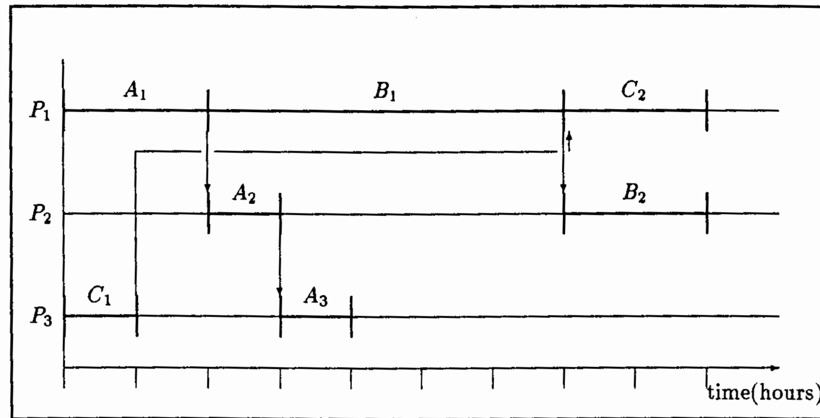

Figure 1: An example of Job Shop Schedule

1.2 About a Task

A task T_i consists in a set of successive operations $\{O_{i1}, \dots, O_{ik_i}\}$. Each operation O_{ij} requires a specific processor for a given period of time. Often a task T_i is also characterized by two dates: the first is the ready date (r_i), before which the task may not be started. Usually, r_i is determined by the availability of the raw materials. The second date is the due date (d_i), at which T_i has to be completed.

1.3 Formulation of the aim

One possible and most popular aim of scheduling is to have all the tasks processed in the shortest time interval. A lot of criteria (or objective functions) were used to characterize the quality of a solution, according to the aim. The most commonly used is the makespan: the maximum of the completion time (c_i) of all the tasks. Another one is the maximum lateness, according to the due dates. To solve a scheduling problem exactly, we have to find the schedule with the best value of the chosen criterion.

Here, for comparison purposes, we adopt the criterion used by Falkenauer and Bouffouix in [7]. The philosophy is to encourage jobs to be finished before their due dates and to penalize in

a more severe fashion all the delays. For each task T_i, we compute the lateness $l_i = (d_i - c_i)$, i.e. the due-date minus the completion date c_i. If the lateness is negative, we measure the quality of the schedule, according to T_i, by $-l_i^2$; else, the quality is equal to l_i. Thus, we have

$$\begin{cases} \text{if } c_i < d_i & q_i = (d_i - c_i) \quad \text{(positive)} \\ \text{else} & q_i = -(d_i - c_i)^2 \quad \text{(negative)} \end{cases}$$

and the objective function to maximize is

$$\mathcal{F} = \sum_{i=1}^{n} q_i$$

Such a criterion reflects the disadvantage of a lateness, for the total quality of the solution; it gives a linear bonus to the jobs which are finished before the due date.

1.4 Notation $\alpha|\beta|\gamma$

To facilitate the discussion about scheduling problems, a three-field notation is in general used. In the notation $\alpha|\beta|\gamma$, the first field α characterizes the processors environment ($\alpha = J$ for job shop, F for flow shop ...); β describes the tasks characteristics (such as due date d_i, ready date r_i and so on) and γ refers to the criterion or objective function.

2 An Industrial Problem

We have to cope with a scheduling problem submitted by a factory manufacturing transportation vehicle mechanical parts. It may be specified by the following notation

$$J \mid r_i, d_i, \text{class-dependent setup times } s_{i_j} \mid \mathcal{F}$$

It's a job shop problem, where tasks have ready date r_i and due date d_i and the criterion to optimize is $\mathcal{F}$. Moreover, we have to cope with operation setup time of a particular type which will be described below (and is similar to what can be found in [14]). Such a setup time results from the load of a new set of tools. As already mentioned, the same problem without the setup time characteristic has received a first treatment in [3].

In our problem, the tasks are grouped into classes, according to the shape or the physical characteristics of the parts involved. Tasks belong to the same class when they are similar enough not to need important setup times between them, i.e. they require the same set of tools when processed on the same machine. There is a setup time only between operations from different classes. For example, let tasks T_1 & T_2 be of the same class, meanwhile task T_3 is from another class. If, on a given processor, the last operation processed before operation x of task T_2 (O_{2x}) is an operation w of task T_1 (O_{1w}), then there is no need for a setup time. But, if an operation y of task T_3 (O_{3y}) is scheduled on the processor after x, there will be a setup time. The **presence of setup time** depends on the class of the two operations: the current and the previous one (O_{3y} and O_{2x}). The setup time period is represented on figure 2 by a oblique line.

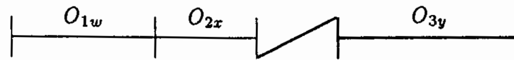

Figure 2: Setup Time Example

In our case, the **duration of the setup time** only depends of the class of the current operation (O_{3_y}) and of the processor. The setup times can be specified in a table with two entries, which are the operation class and the processor. All the operations of the same class and on the same processor have the same setup time duration, meanwhile another class or another processor may lead to a different duration. On the other hand, on the same processor and inside a class, each operation has its own duration, due to different number of parts to manufacture.

3 The notion of Conflict – The Scheduler

We illustrate the notions introduced using the example of figure 1.

We can view the problem of job shop scheduling as a global struggle, where each operation tries to access the needed processor, in agreement with the constraints on the processors and on the tasks (see section 1.). The task constraint says: 'only one operation of a task may be processed by at most one processor at a time'. Besides, the processor constraint is 'only one task at a time can be performed on any given processor'.

A *conflict* is the collision of several operations that want to access the same processor at the same time. Thus, for instance, in figure 1, the following operations are conflicting for accessing processor P_1 after the first unit of time: A1, B1 & C2.

Our scheduler [8] is an algorithm that retrieves progressively the conflicts and tries to solve them in agreement with the constraints. The scheduler routine makes decisions successively at all the moments when an operation is completed on a processor. At such a moment, it lists all operations that are ready to be performed on this processor (i.e. for which all operations of the same task to be performed before have been completed). All the operations (and the corresponding tasks) in the list are candidates for accessing the processor. If the candidate operation is unique, then it is decided to start it immediately. If there are several candidates, the conflict between them must be solved.

For instance in the example, at the initial time, the scheduler is able to set C1 on the third processor without any problem. At the same time, on the first processor, there are apparently only two operations, A1 and B1, in conflict, but because C1 operation time is really short, C2 can be ready to be performed before either A1 or B1 is ended, thus C2 has to be taken into account. Suppose it is decided to perform A1 first. By the time A1 is completed, B1 and C2 are again in conflict for P_1. If B1 is chosen first, we finally get the illustrated schedule without further conflict resolution.

But, how are the conflicts solved? First, a conflict that involves more than two operations

can be divided in several conflicts of two operations (elementary conflicts). The scheduler is not equipped for solving the elementary conflicts by itself. It needs external help. This help will be given by an external resolution rule. And the aim of scheduling is now to find the optimal conflict resolution rule, given the scheduler and the optimizing criterion.

It should be noted that this procedure does not guarantee overal optimality as conflict resolution is done on a sequential way determined by the end of processing times of the previously scheduled operations.

4 Priority Lists as Conflict Resolution Rules

4.1 Priority List

The simplest conflict resolution rule is a priority relation between all the operations. Such a relation can be described by a priority list (PL). Here is an example of PL for the schedule illustrated in figure 1, (the preference decreases from left to right).

$$PL \; : \; A1, B1, A2, B2, C2, A3, C1$$

As only operations on the same processor can possibly come in conflict, we use one priority list per processor. Then, a schedule is given by a set of m PLs, where m is the number of processors. For the figure 1, we can take as PLs

$$\begin{aligned} PL1 &: A1, B1, C2 \\ PL2 &: B2, A2 \\ PL3 &: A3, C1 \end{aligned}$$

When the scheduler is facing the conflict between A1 and B1 on the first processor, it looks in PL1, to find one of these operations. The first found operation (i.e. A1) is allowed to access the processor. The other one (B1) has to wait at least until the end of A1.

In fact, PL1 is a powerful rule able to solve all the potential conflicts on P1. It's equivalent to the conjunction of three simple rules, each solving one conflict.

$$PL1 \; : \; A1, B1, C2$$

$$\begin{aligned} PL1 &= A1, B1 \\ &\;\& \; A1, C2 \\ &\;\& \; B1, C2 \end{aligned}$$

4.2 The Canonical PL

Specifying a PL for each processor and using the scheduler together with the PLs determine a unique schedule. But several PLs can lead to the same schedule. The set of PLs is an indirect representation of the scheduling. As a matter of fact, in our example, because the PLs are used only in case of conflicts and as there is no conflict between A2 and B2, the priority relation given by PL2 has no effect on the schedule.

Hence the priority lists

$$\begin{aligned} PL1 &: A1, B1, C2 \\ PL2' &: A2, B2 \\ PL3 &: A3, C1 \end{aligned}$$

also yield the schedule in figure 1 through the use of the scheduler. In general, for each schedule, there is a class of priority lists which yield it. In particular, we call *canonical* lists associated to a given scheduler, a set of lists, one for each processor, each list containing the operations to be processed in the order defined by the schedule. In our example, the priority lists just given above are canonical. Such canonical lists can be described as the 'fixed points' of the scheduler in the sense that in the schedule obtained from them through the scheduler, the operations are exactly processed in the order specified by the priority lists.

5 Priority Lists and Heuristics

The PL approach was used in conjunction with two different heuristics: Genetic Algorithms (GA) and Tabu Search (TS).

5.1 Genetic Algorithms

5.1.1 Principle

Holland, was inspired by the 'struggle for life' in designing his 'Genetic Algorithms' (GA) [12]. A complete discussion of the GA can be found in [10] (see also [8]). Here is a summary of these general heuristics for optimization.

A set of solutions, called *a population of individuals*, are evaluated: the fitness of each individual (i.e. the performance of each solution in view of the objective function) is computed. Like with the natural selection, some individuals are selected to be the 'parents' of the next generation. This selection is made at random, but by giving to the individuals a probability to be chosen proportional to their fitness. In this way, good solutions have a higher chance of having 'children'.

After having selected a set of 'good solutions' (the 'parents'), we create a new population of – possibly – 'better solutions' (the 'children'). Here, the GA uses the so-called *genetic operators*. Information is exchanged between pairs of selected individuals yielding new pairs of individuals (cross-over operation). Some other individuals are submitted to the 'mutation' (a random modification of the solution).

The new population so obtained is evaluated. And, the whole process iterates.

5.1.2 Implementation

When implementing a GA, three elements have to be tailored to the problem: the coding of the population, the evaluation and the genetic operators.

The population in the GA is a set of individuals; each of them describes a scheduling and consists, here, in a set of m PLs (one for each processor). Each individual has to be evaluated: we use our scheduler to compute the schedule and the value of the objective function $\mathcal{F}$.

Then we have to apply genetic operators to the selected individuals. Here, applying a genetic operator on a given individual will be done by applying the genetic operator to each PL (crossover) or a randomly generated subset of PLs (mutation) of the individual. Moreover, because of the representation condition (i.e. each operation on P_j must be one and only one time in PL_j), we used operators designed for the Traveling Salesman Problem: i.e. permutation, inversion and PMX.

For the operator description, we'll use the following two PLs, which aren't relevant for the example of figure 1.

$$\begin{aligned} PL1 &: A,B,C,D,E,F,G,H,I,J \\ PL2 &: A,C,E,G,I,B,D,F,H,J \end{aligned}$$

Permutation This operator consists in the swapping of 2 operations of one PL. For example, starting with PL1 and swapping A and F, we get

$$PL1' : F,B,C,D,E,A,G,H,I,J$$

Permutation is the elementary operator, according to the representation condition. In [8], Inversion and PMX are shown to be composed of 'permutations'.

Inversion We simply invert a portion of PL1 (the limits of the 'inversion section' are randomly chosen) to obtain

$$PL1'' : A,B,C, \mid H,G,F,E,D, \mid I,J$$

The first two operators belong to the class of 'mutation operators'. The next one is a 'crossover'.

PMX This operator has been introduced by Goldberg ([10]), to solve the Traveling Salesman Problem. To obtain the crossed-over individuals PL1* and PL2*, we define an 'equalization area' at random, for instance between the fifth and eighth positions. The 'equalization area' is represented in the example by two vertical lines. Starting with PL1, we operate by well-chosen permutations to get this area of PL1* equal to the one of PL2. In the example below, we swap successively E and I, F and B, G and D, H and F and obtain

$$\begin{aligned} PL1 &: A,B,C,D, \mid E,F,G,H, \mid I,J \\ PL2 &: A,C,E,G, \mid I,B,D,F, \mid H,J \\ \\ PL1^* &: A,H,C,G, \mid I,B,D,F, \mid E,J \\ PL2^* &: A,C,I,D, \mid E,F,G,H, \mid B,J \end{aligned}$$

Some Characteristics In our study, we tried to use permutation and inversion as the mutation operator; permutation appeared at least as good as inversion. But, to get 'good' solutions, our program had to work with a relatively high mutation-rate (i.e. 10 % of the population individuals are submitted to the mutation), while the crossover rate was 'normal' (30 % of the pairs of parents were crossed-over). We used different population sizes ranging from 30 to 100 individuals without substantial improvement.

5.2 Tabu Search

5.2.1 Algorithm

Tabu Search is an improvement of the Steepest Ascent local search heuristics. A more extensive presentation can be found in [11].

With Steepest Ascent, we start from an initial solution s. The next solution is the one maximising the objective function) in the neighbourhood $\mathcal{N}(s)$ of the current solution, if better than the current one. This iterative procedure ensures to reach, at least, a local maximum. But, this method remains trapped on the 'mountain' on which it starts.

Tabu Search allows not only to climb along the gradient but also to go down in order to reach others 'mountains'. In many implementations of TS, only a portion $\mathcal{N}^*(s)$ of the neighbourhood $\mathcal{N}(s)$ of the current solution s is evaluated. The next solution is the best one in $\mathcal{N}^*(s)$, except for the current solution.

A Tabu List (TL) prevents the algorithm from cycling indefinitely between the same solutions, alternatively climbing and descending. Typically, the Tabu List contains the few last solutions reached (or a property thereof), that are said 'tabu' for a while. The new definition of $\mathcal{N}^*(s)$ is *a portion of* $\mathcal{N}(s)$ *minus* $\{s\}$ *and minus the eventual intersection with the TL.* Hence when exploring $\mathcal{N}(s)$, the best non tabu solution in $\mathcal{N}^*(s)$ is selected.

This method allows to visit several peaks and test several maxima, in order to reach a global (or sub-global) maximum.

5.2.2 Implementation

As we used the standard Tabu Search algorithm just described, we have only to specify the neighbourhood and the tabu list.

The neighbourhood of the current solution $\mathcal{N}(s)$ (where s is the current set of PLs) is defined as 'all the solutions reachable by at most one permutation (see section 5.1.1) in each PL'.

In TS, only a randomly chosen portion of $\mathcal{N}(s)$ is considered. Thus, we draw at random a set of m* processors and we compute k* solutions by applying, for each of the k* solutions, the permutation operator on the PL of those m* processors. Then, we get to the best of these k* solutions.

The Tabu List (TL) is, in our implementation, a list of forbidden processors. These processors may not be chosen for being in the set of the m* processors lists on which permutations are done. At each TS iteration, we get the oldest processor from the TL and set it 'choseable', meanwhile, one of the m* chosen processors is set 'unchoseable' and enters the TL. This protects us from

getting completely back on our steps, because some of the latest modified PLs cannot be turned to their former state.

We get good results by setting m* to $\lfloor(1+m/10)\rfloor$. Besides, k* was computed as the total number of operations on the m* processor lists.

6 Classifier System as Rule Set Optimizer

6.1 A Rule System

Generally, the computerized treatment of a problem consists in the automatization of the human (by-hand) way of doing. Rather than constructing a Priority List (PL), a human scheduler compares conflicting operations and decides which one has the priority. His decision is mainly based on the operations characteristics. For instance, SPT (Shortest Processing Time operation first) and LST (Least Slack Time operation first) are two well-known rules which are commonly used.

We can create a program able to solve a scheduling problem, in such a way, providing a decision system with a set of rules and a natural hierarchy between them (e.g. first rule first). A simplified example of such a rule system is given in figure 3.

Rules			
Condition		⇒	Action
O1 shorter than O2	O1 more urgent than O2	⇒	O1 has priority
O1 longer than O2	O1 more urgent than O2	⇒	O1 has priority
O1 shorter than O2	O1 less urgent than O2	⇒	O1 has priority
O1 longer than O2	O1 less urgent than O2	⇒	O2 has priority

Figure 3: Classical Rule System

Such a system can sometimes give an acceptable solution. But, more generally, it yields a poor performance schedule, which needs to be modified. Indeed, this rule system has only local views on the problem; it's only able to cope with conflicts on the basis of the conflicting operations characteristics alone and has no idea about the global effects of its decision on the performances measured by the criterion.

Besides, the previous methods, Genetic Algorithms and Tabu Search implemented with Priority Lists, have such a global view. They cope with conflicts but have a feedback of the value of the criterion. This allows them to optimize their way of solving conflict, to optimize their set(s) of PL. The following section is concerned with the construction of a rule system, which adapts according with its preformance. In [13], Holland suggests the use of a so-called 'Classifier System'.

6.2 A Classifier System – A Rule Set Optimizer

The previous Rule System (RS) has a natural structured hierarchy inside its set of rules: when a conflict has to be solved, the rules are searched from the first one to the last one in order to find a rule of which condition holds. This rule can thus solve the conflict. The idea of the Classifier System (CS) is to give the system more rules and the ability to modify the rules hierarchy by itself.

We construct a new set of rules, where rules may be contradictory and we give each rule a hierarchy number (called 'quality'). Such a rule is called a '*classifier*'. We'll describe the concept of Classifier System (CS) in the framework of our application. Figure 4 gives a representation of the CS.

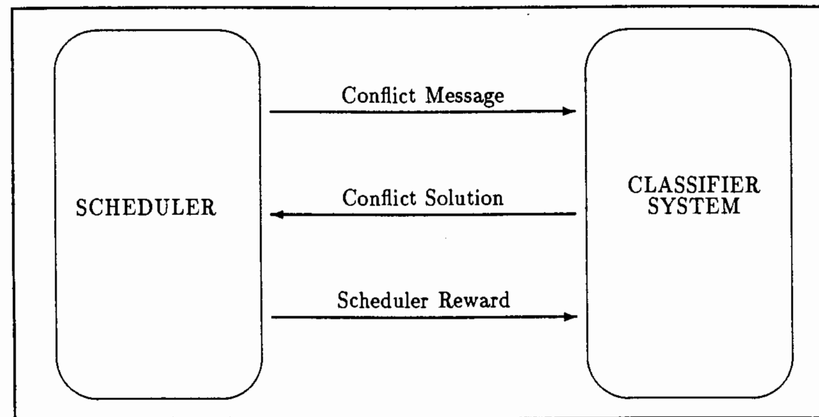

Figure 4: Scheduler & Classifier System

When a conflict occurs, the scheduler sends a 'message' to the CS together with some informations about the conflict, like the duration of the operations in conflict, the slack time of the tasks owning the operations in conflict,... This message is coded in a binary string. Each rule in the CS has a condition coded in a ternary alphabet: the binary one plus a "don't care" character ('*'). In figure 5, we give an example of a message and a set of classifiers.

Each rule having its condition matched by the message can compete to solve the current conflict. One of those rules is drawn at random, with a probability depending on their qualities ('roulette wheel' algorithm); the rule with higher quality has higher probability to be selected (This allows the system to try unknown rules with poor quality number, that may appear very good later on). The conflict is solved by the action of the selected rule (e.g. action 0 means 'the first operation involved in the conflict has priority').

At each conflict, the rules pay taxes and the selected one pays a supplementary taxe. The rules have their quality decreased by the amount of the taxes. At the end of a complete schedule, each rule selected for solving a conflict receives a reward, according to the value of $\mathcal{F}$. This reward

Conflict Message					
010	101				
Rules					
Condition		⇒	Action	Quality	Matched ?
100	*01	⇒	1	4	0
11*	*00	⇒	0	2	0
010	1**	⇒	1	1	1
01*	*01	⇒	0	5	1
000	1**	⇒	0	3	0
0**	101	⇒	1	2	1

Figure 5: Matching & Non-Matching Rules

is added to its quality. This modification of the qualities modifies in a way the set of rules.

By solving several times the same job shop problem with the set of rules, starting each time with the quality numbers obtained at the previous iteration, 'good rules' emerge from the set, because the 'good' classifiers have their quality increased, while the quality of the other ones vanishes.

In order to let the system evolve to better and better sets of rules, one wants to create new rules from the best ones. As those best ones have emerged from the set, their quality measures their fitness to the problem. One applies now the GA on the set of rules considered as a population of individuals (selection of high-quality rules – crossing-over, i.e. exchange of some parts of the condition of the selected rules – mutation). Due to the GA, we get a new set of classifiers. This new set is evaluated in the same way as previously, tested several times to let the good rules emerge, and then used to create another new set and so on.

7 Results – Remarks and Conclusion

The following results are obtained from two different sets of data. The first file, TEST1, contains industrial data. There are 31 processors on which 40 tasks have to be completed. The total number of operations is 225. The second set, TEST2, was designed for testing purposes. But, it seems more difficult as the due dates are more demanding, while, in TEST1, the due dates are relatively far from the ready date.

We have conducted our experiments on many different systems, from a PC to a Silicon Graphics workstation. The following results were obtained on a DECstation 5000/120 (21.7 MIPS). Our programs were compiled by the RISC C compiler under the ULTRIX environment.

For both data sets and for each of the different methods considered, we have computed the mean of the best values of the objective function obtained during each of 50 runs, as well as the average of the running time.

TEST1 31 Processors, 40 Tasks, 225 Operations		
	Mean Fitness	*Running Time*
SPT	591,868	< 1 sec.
LST	519,335	< 1 sec.
Classifier System	596,786	18 min.
Genetic Algorithms	625,577	40 min.
GA with Canonical lists	637,439	43 min.
Tabu Search	641,199	63 min.
TS with Canonical lists	639,327	65 min.

TEST2 8 Processors, 8 Tasks, 53 Operations		
	Mean Fitness	*Running Time*
SPT	-24,400	< 1 sec.
LST	800	< 1 sec.
Classifier System	3,000	2.5 min.
Genetic Algorithms	3,200	1.5 min.
GA with Canonical lists	3,900	1.6 min.
Tabu Search	4,100	0.5 min.
TS with Canonical lists	3,900	0.6 min.

The obtained results show a quality order between our implementations of the described methods. The less good (or the worst...) systems are one-rule-systems, like SPT and LST. Then, comes the Classifier System, with a fixed number of rules. Finally, the best one involves Priority Lists. The quality order results from a rule-number order. The more the system has rules, the better solutions it gets. Indeed, the best one, PL, has complex rules giving a preference order between all the operations; thus, because a complex rule is the conjunction of simple rules, its number of simple rules is equal to the number of the possible conflicts, while the other systems have fixed number of rules.

Besides, the use of canonical lists appears in favour of the GAs, and not of TS. Indeed, the use of canonical lists allows to crossover individuals, according to the time axis, by providing a time order to the initial preference order. Thus, the canonical list is important in GAs. But, for Tabu Search, the canonical lists appear like an unnecessary perturbation in the set of PLs.

The last result of our study is that the best program is the TS one. It gives the best schedules, in the smallest time (or hardly larger) than the other ones. Moreover, as far as the implementation is concerned, the TS program is the easiest one.

Acknowledgments

It is a pleasure for us to acknowledge our debt to Jacques Teghem and Marc Pirlot for helping us in this work and assailling us for the writing of this paper. We are also grateful to Emmanuel Falkenauer for introducing us to GA and CS, suggesting the problem and stimulating us with constructive discussions.

References

[1] BLAZEWICZ J., DROR M. & WEGLARZ J. (1991). Mathematical Programming Formulations for Machine Scheduling: a Survey. *European Journal of Operations Research* 51, pp. 282-300.

[2] BLAZEWICZ J., FINKE G., HAUPT R & SCHMIDT G. (1988). New Trends in Machine Scheduling. *European Journal of Operational Research* 37, pp. 303-317.

[3] BOUFFOUIX S. (1990). *Contribution à l'Ordonnancement de Production: Apport des Algorithmes Génétiques et des Systèmes à Apprentissage'.* Graduate thesis, Université Libre de Bruxelles.

[4] CHENG T.C.E. & SIN C.C.S. (1990). A State-of-the-art Review of Parallel-Machine Scheduling Research'. *European Journal of Operational Research* 47, pp. 271-292, 1990.

[5] DAVIS L. (1985). Job Shop Scheduling with Genetic Algorithms. *Proceedings of the first International Conference on Genetic Algorihtms and their Applications*, ed. Grefenstette J.J., L. Erlbaum Publishers, New Jersey.

[6] DELLA CROCE F., TADEI R. & VOLTA G. (1992). *A Genetic Algorithm for the Job Shop Problem.* Working paper, Politecnico di Torino (Italy)

[7] FALKENAUER E. & BOUFFOUIX S. (1991). A Genetic Algorithm for Job Shop. *Proceedings of the 1991 IEEE International Conference on Robotics and Automation.*

[8] FORTEMPS Ph. (1992). *Les Systèmes à Classifieurs dans l'Ordonnancement de Production avec Temps de Réglage. Comparaison avec d'autres Méta-Heuristiques.* Graduate thesis, Faculté Polytechnique de Mons.

[9] GAREY M.R. & JOHNSON D.S. (1979). *Computers and Intractability, a Guide to the Theory of NP-Completeness.* Freeman and Co, San Francisco.

[10] GOLDBERG D.E. (1989). *Genetic Algorithms in Search, Optimisation and Machine Learning.* Addison-Wesley Publishing Company.

[11] HERTZ A. & de WERRA D.(1988). The Tabu Search Metaheuristic: How We Used It. *Annals of Mathematics and Artificial Intelligence.*

[12] HOLLAND J.H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor.

[13] HOLLAND J.H. (1985). Properties of the Bucket Brigade Algorithm'. *Proceedings of the first International Conference on Genetic Algorihtms and their Applications*, ed. Grefenstette J.J., L. Erlbaum Publishers, New Jersey.

[14] KUT C. (1990). Some Heuristics for Scheduling Jobs on Parallel Machines with Setup. *Management Science* 36, 4 pp. 467-475.