Genetic Branch-and-Bound or Exact Genetic Algorithm?

C. Pessan[1,2], J.-L. Bouquard[1], and E. Néron[1]

[1] LI, Université François Rabelais Tours, 64 av. Jean Portalis, 37200 Tours, France
cedric.pessan@univ-tours.fr
[2] SKF France SA, Industrial division, MDGBB Factory, 204, boulevard Charles de Gaulle 37542 Saint-Cyr-sur-Loire CEDEX, France

Abstract. Production resettings is a vital element of production flexibility and optimizing the setup tasks scheduling within a production channel is required to improve production rate. This paper deals with a NP-Hard production resetting optimization problem based on an industrial case. In this paper we present how to hybrid a Branch-and-Bound method for this problem with a genetic algorithm. The idea is to use the genetic algorithm to improve the upper bound and thus speeding up the Branch-and-Bound while the genetic algorithm uses the content of the Branch-and-Bound stack to reduce its search space. Both methods are running in parallel and are therefore collaborating together.

1 Introduction

Improving production flexibility is one of the main problems encountered in the industry as it is closely linked with customer service improvement and the length of delays between customers orders and delivery of products. It is thus important to reduce resetting times between batches. A production resetting consists in operations made on each machine of a production channel made by operators. These operations are required to setup the machines for the new batch. One way to improve resetting times is to work on the global organization of the different setup tasks according to industrial constraints, e.g., the skills of the operators, their availability periods. The study is based on a real industrial case found in SKF MDGBB (Medium Deep Groove Ball Bearings) factories.

This problem can be identified as an unrelated parallel machine scheduling problem, for which we have developed a Branch-and-Bound method in Pessan et. al. [2006b]. The main drawback of this method is the upper bound that is so far away of the optimal solution in our experiments that the method can not prune any node at the beginning of the resolution. On the other hand, we have also developed in Pessan et. al. [2006a] a heuristic on a more general problem (serial-parallel production channel) based on a genetic algorithm hybridized with a local search that proved to work well for this problem. The idea of this paper is to hybrid the Branch-and-Bound with a genetic algorithm in order to get the

N. Monmarché et al. (Eds.): EA 2007, LNCS 4926, pp. 136–147, 2008.

best of both methods: fast convergence to good solutions and exact resolution. The genetic algorithm is not only run at the root node but in parallel with the Branch-and-Bound. Moreover, the encoding method is based on the content of the Branch-and-Bound stack: it means that while the Branch-and-Bound progresses, it reduces the search space of the genetic algorithm, and when the genetic algorithm improves the best known solution, it helps pruning more nodes. So, both methods are really collaborating together during the whole execution.

It is natural to use genetic algorithms to find a good upper bound of the optimal solution quickly either on the root node or regularly during the execution of the Branch-and-Bound. Such attempts have been made in several papers. Portman et. al. ([1998]) use a genetic algorithm on the root node of a Branch-and-Bound in order to provide a good initial upper bound to the Branch-and-Bound. Jouglet et. al. ([2005]) propose a similar approach but they use the genetic algorithm to provide an initial upper bound to a constraint programming method. In Basseur et. al. ([2005]) a biobjective unrelated parallel machine problem is tackled with a genetic algorithm that provides an initial pareto front to a 2 phases Branch-and-Bound. Branch-and-Bound are also commonly hybridized with other meta-heuristics like in Rocha et. al. ([2004]): the GRASP meta-heuristic is used to provide an upper bound to a Branch-and-Bound method. Cotta et. al. ([1995]) show some preliminary results on various combination of Branch-and-Bound and genetic algorithms: they have tried using Branch-and-Bound like methods as local search operator of a genetic algorithm leading to a heuristic hybrid method. On the other hand, they propose an hybrid method that run in parallel a Branch-and-Bound and a genetic algorithm but they mention some difficulties in handling diversification of the genetic algorithm population and the slow convergence of genetic algorithms for the traveling salesman problem they are working on. French et. al. ([2001]) present also such a hybrid algorithm, they use the Branch-and-Bound to find promising nodes and thus generate the initial genetic algorithm population and then use the genetic algorithm results to give hints to the Branch-and-Bound on where there can potentially be interesting solutions. They switch back and forth between the two methods. Their results seem promising. Puchinger ([2005]) in its survey on these hybrid methods distinguishes between integrative combinations that tend to use one method as an operator of the other and collaborative methods. This second category is also categorized between sequential execution and parallel execution. It is also mentioned that parallel execution algorithms have not been extensively tried but with the emergence of mainstream multi core processors that can execute in parallel several algorithms with fast access to a common memory area that ease data sharing between the algorithms, it may be time to give importance to such algorithms.

In this paper we present in section 2 the model of the problem we are studying and the existing Branch-and-Bound method. Then, in the section 3 we describe the hybrid method. Finally, experimental results are presented in section 4.

2 Existing Methods

2.1 Problem Description

Let n be the number of machines of a production channel, it is also the number of tasks to schedule. For each machine (or task) $M_i, i \in \{1, \ldots, n\}$, we know its release date r_i. It is the minimum duration needed for the last ball bearing of the previous batch to go from M_1 to M_i. We also know its tail q_i, the minimum duration needed for the first ball bearing of the next batch to go from M_i to M_n. When a machine M_i is restarted, it can not have any effect on production rate that is measured at the end of the production line before q_i time unit. t_i and C_i denote respectively the beginning and the completion time of setup task on machine M_i.

In the production unit, there are λ operators. Each operator $O_h, h \in \{1, \ldots, \lambda\}$, depending on his own experience, needs a different time to set up a machine: this time is denoted $p_{i,h}$. If an operator O_h does not have the skill for a machine M_i, we set, without loss of generality, $p_{i,h} = +\infty$. Moreover each operator is only available during a time interval $[R_h, D_h]$.

In this paper, we consider serial channels: it means that the production can only restart when all machines have been setup. Therefore, we have to optimize the maximum completion time of the setup tasks, also known as $C_{max} = max_{i=1,\ldots,n} C_i + q_i$ in standard scheduling notations.

According to classical scheduling problem classification, this problem can be identified as an unrelated multipurpose parallel machines problem with release dates and tails. In our problem, resources are the operators, operations are the setup tasks of each machine. This problem is denoted $R, MPM|r_i, q_i|C_{max}$. Figure 1 presents an instance made up of 4 machines and 2 operators.

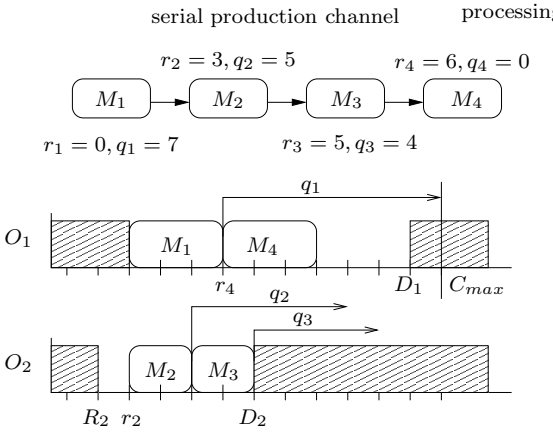
Moreover as explained previously, r_i and q_i can be seen as distances in time from machine M_i to respectively the beginning and the end of the production channel. It means that for a machine M_i, the farther it is from the beginning of the channel, the closer it is to the end of the channel. Then the non decreasing r_i order is the same as the non increasing q_i order. This proposition (prop. 1) is illustrated on the figure 2.

Proposition 1. *In a serial channel, r_i and q_i are such that: $\forall i \in \{1, \ldots, n\}, r_i \leq r_{i+1}$ and $q_i \geq q_{i+1}$.*

Corollary 1. *On each operator, scheduling tasks in non decreasing release date order is optimal*

Using proposition 1, it is easy to deduce the corollary 1 as shown in Pessan et. al. ([2006b]). Therefore, the problem can be seen as an assignment problem: once tasks are assigned to operators, their order and then their starting time are deduced using corollary 1.

The $R, MPM|r_i, q_i|C_{max}$ problem is $\mathcal{NP}-Hard$ even in our case of serial channel since the particular case $P, MPM||C_{max}$ is known to be $\mathcal{NP}-Hard$ as shown by Garey and Johnson ([1978]).



processing times related to skills of operators

$p_{i,h}$	O_1	O_2
M_1	3	2
M_2	2	2
M_3	∞	2
M_4	3	∞

	O_1	O_2
R_h	3	7
D_h	2	12

Fig. 1. A 2 operators 4 machines instance

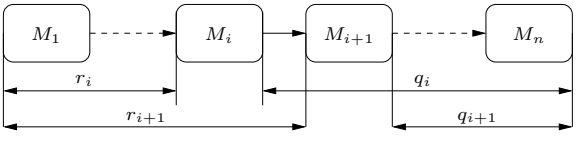


Fig. 2. Property on r_i and q_i

Unlike the $P|r_i, q_i|C_{max}$ problem studied in Carlier ([1987]) and Gharbi and Haouari ([2002]) and unlike the multipurpose parallel machines problem studied in Jurisch ([**?**]), the $R, MPM|r_i, q_i|C_{max}$ problem has not been extensively studied in the literature. We can mention for instance Gharbi and Haouari ([2005]) but the corollary 1 on our specific problem allows us to implement more efficient algorithms (cf. section 2.2).

2.2 Branch-and-Bound Method

In this section we describe briefly the Branch-and-Bound method presented in Pessan et al. ([2006b]). This Branch-and-Bound is used in the hybrid method.

Generalities: A Branch-and-Bound method is a classical way of implicitly enumerating all solutions of a search space to find the optimal one. In a Branch-and-Bound method the search space is assimilated with a tree stored using a data structure, e.g. stack, containing not yet explored nodes. Each node represents a partial solution and a sub-domain of the search space. Moreover, going from one level to the next one means making a decision and reducing the domain of at least one variable. On a leaf node, all variables are fixed. The figure 3 shows the relationships between the search space representation, the tree and the stack. The tree is a representation of the search space and the stack contains the frontier of the unexplored parts of the search space.

At each iteration, a node (N) is extracted from the stack and a lower bound $lb(N)$ of all the solutions in its corresponding sub-space is computed. Then, this

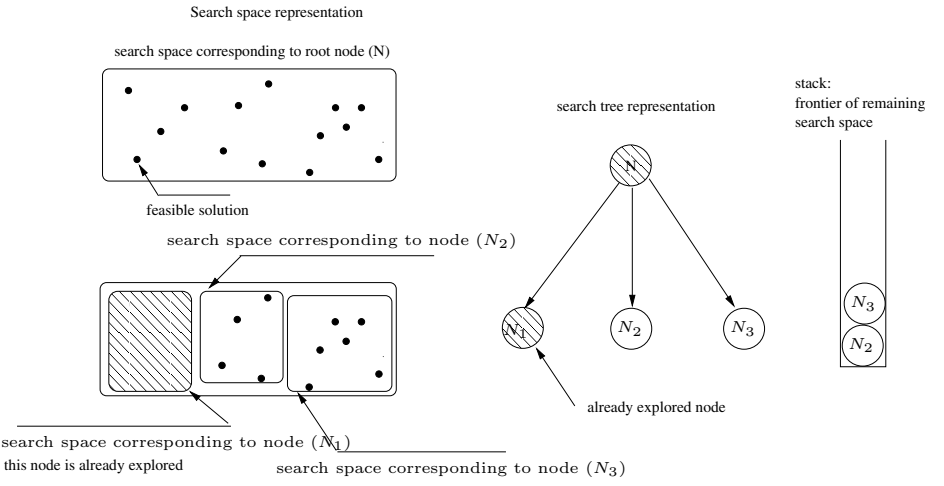


Fig. 3. Relationship between search space, stack and tree

lower bound is compared to the best known solution ub also known as the upper bound of the optimal solution. If (N) is a leaf node and if its criterion is better than the upper bound, the upper bound is updated. Otherwise, if the lower bound is greater that the upper bound, i.e. $lb(N) > ub$, the node is discarded (we also say that the node is pruned) and if it is lower, child nodes are generated and pushed on the stack. A cutting rule can be seen as an extension of the use of a lower bound to prune the search tree. A cutting rule is a procedure that takes one parameter D and returns a boolean. Answer 'no', corresponds to the fact that no promising solution, i.e., solution having a makespan lower than or equal to D, could be found in the subspace corresponding to the current node. If the cutting condition answers 'no' with D= $ub - 1$ then the node can be pruned.

Notice that generally lower bounds are computed once node is extracted from the stack, but in the case that the exploration strategy is based on the lower bounds, e.g, best-first strategy (see section 3.2), the lower bound is computed before new nodes are pushed on the stack.

Branching Scheme: According to corollary 1 the only decision we have to make is the assignment of the tasks to operators. Moreover without loss of generality, we assume that the tasks are already sorted so that $\forall i \in \{1, \ldots, n-1\}, r_i \leq r_{i+1}$. At each level (i) of the search tree, the branching scheme tries to assign task M_i to each operator that is able to perform the task. It means that there are n levels in the tree and a maximum of λ branches per node depending on the number of operators who master the skill corresponding to the task.

Upper Bound: The upper bound is computed using a simple greedy algorithm that uses the Earliest Completion Time (ECT) priority rule. It is assigning tasks to the operators who can complete it first. In our experimental results (Pessan et. al. [2006a]), this upper bound was the main problem of the method as it was too far away from the optimal to prune nodes. Moreover, solutions built using ECT may not be feasible, regarding to $[R_h, D_h]$.

Cutting Rule: The cutting rule is based on relaxation of non-preemption constraint : it means a task can be interrupted and restarted later either on the same operator or on another one. We define deadlines for each task as $\tilde{d}_i = ub - 1 - q_i$. The lower bound checks if it is possible to achieve all the tasks within the allowed time-windows (within $[r_i, \tilde{d}_i]$). If it is not possible, then the node can be pruned. Checking this can be done polynomially using a linear program as shown by Lawler and Labetoulle ([1978]).

3 Hybrid Method

3.1 Generalities

We have seen that a method such as Branch-and-Bound is enumerating implicitly all feasible solutions. The search space can be reduced whenever a node (N) can be pruned, that is when $lb(N) \geq ub$ or when the cutting condition return 'no', thus, the upper bound is an important part: the upper bound is improved as the method is discovering better solutions but as long as the upper bound is not close enough of the optimal, it is usually hard to prune nodes as the condition $lb(N) \geq ub$ is rarely satisfied. So it is important to find a good upper bound as fast as possible. Moreover, if the search is stopped before reaching the optimal solution, it is interesting to have a good solution that can be given to the decision maker.

Here we describe how a Branch-and-Bound can be improved using an efficient method to compute an upper bound, namely, genetic algorithm. The genetic algorithm can be used at the root node or it can be used at any time to search a better upper bound. The idea, is that the genetic algorithm process should focus on improving the best known solution by searching within the remaining search space, i.e, whose frontier is still in the stack of the Branch-and-Bound, while the Branch-and-Bound should eliminate as large parts of the search space as possible. Basically there are two ways to modify the search space. First, each time child nodes are created in the Branch-and-Bound, the search space of the parent node is subdivided into disjoint sub-spaces of the newly created nodes. Second, when a node is pruned in the Branch-and-Bound, a part of the search space is eliminated. So the genetic algorithm should be implemented in a way such that it only searches within the unexplored areas and a convenient way to do this is to use the nodes contained in the stack.

3.2 Hybrid Exact Genetic Algorithm

Genetic algorithm is a meta-heuristic inspired by biological evolution that has been introduced by Holland ([1975]). The method uses a population of solutions that are encoded using a carefully chosen encoding function: these encoded solutions are called individuals or chromosoms. During each iteration, the algorithm follows these steps:

- **selection:** pairs of individuals are selected
- **crossover:** a crossover operator is applied to the selected individuals pairs.

- **mutation:** each generated individuals may be mutated by slightly changing them. The probability of mutating an individual should be low
- **replacement:** individuals that will survive through next generation should be selected in order to keep a constant population size

The idea is that good individuals should propagate their characteristics. This kind of algorithms has proved to work well on a more general case of the problem (serial-parallel production channel, see Pessan et. al. ([2006a])), that is why we have tried to use a genetic algorithm to improve the Branch-and-Bound.

Encoding Function: As mentioned above, the corollary 1 means that our problem can be reduced to the assignment problem of tasks to operators. So, the encoding function of the genetic algorithm should only contain assignments of operators to the tasks. The decoding function of the genetic algorithm only has to order tasks in non decreasing r_i order, and then computing starting time of tasks.

As shown on figure 4, the chosen encoding function generates individuals compound of three parts: a node, the node level and an assignment array. A node structure contains a partial solution and its level in the search tree: a node at level k containt k assignments. So, the node of the individual defines the first assignments. The remaining assignments should be encoded in the third part of the individual.

node	node level	assignments of remaining tasks
⑦	2	1/3/4/4

Fig. 4. An example of an encoded solution for a problem with $n = 6$ and $m = 4$

The individual of the example presented in figure 4 is using the node 7 that is at level 2 of the search tree: the 2 first assignments are taken from the node. The node contains only a partial solution with 2 assignments, so the rest of the individual requires $n - node\_level = n - 2 = 4$ additional assignments needed to build a solution.

The advantage of this function is that the hybrid method is using the stack to generate individuals and keeping the node in the encoded individual eases the synchronization of the population with the stack content as explained below.

Crossover Operator: The crossover operator is a classical one point crossover that generates two individuals from two parents. It selects randomly a number p between 0 and n. To generate the first child, the operator copies, p assignments from the first parent and $n - p$ assignments from the second parent. The node of the child is the node of first parent. The second child is created by exchanging the roles of parents. An advantage of this operator is that child always contains existing nodes of the stack and valid assignments and thus belong to the remaining search space whose frontier are the nodes of the Branch-and-Bound stack.

On the example of the figure 5, p is set to 3. So, 3 assignments are copied from the first individual to the first child: the 2 assignments of the node and

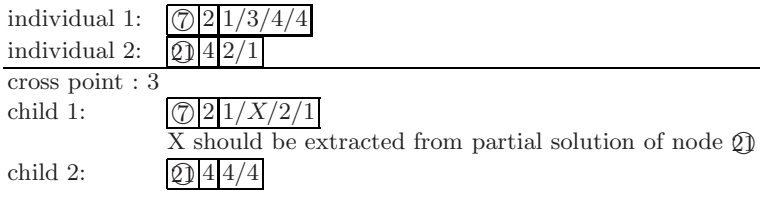


Fig. 5. Crossover example

one additional assignment. Then, other assignments are taken from the second individual: one that comes from the node 21 and the rest from the third field of the individual.

Mutation Operator: There are two mutation operators in this method. The first one is randomly changing a gene within the third field of an individual, that is, within the assignments that complete the partial solution of the node.

The second mutation operator is randomly switching to a new node present in the stack. If the new node is at a lower, it extracts missing assignements from the old node. On the example of figure 6, node 7 that is at level 2 is replaced by node 21 that is at level 4: the 4 first assignements of the mutated individual are extracted from node 21 and the two remaining assignments come from the original individual.

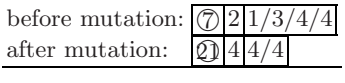


Fig. 6. Second mutation operator example

The probability $pm1$ of mutating an individual should be set to a high value mainly because of the chosen encoding function and crossover operator: as long as there are no mutations, no new assignments are introduced in the population. But it is also required because of the need to explore quickly a subspace in this hybrid method. If there is a mutation, the probability $pm2$ of using the second operator should be low because this operator change many assignments in an individual and using it too much would lead to too much randomness.

Synchronization Operator: it is an operator that is specific to our hybrid method. It is there to check that the genetic algorithm is only searching within the unexplored area: the part of the search space whose frontier is in the stack. This operator requires non negligible amount of time to be executed and should not be executed at each iteration. Moreover if it is called too many times, it may be hard for the genetic algorithm to evolve correctly as it would eventually invalidate solutions as soon as they are created. So we have chosen to call it every $maxIt$ ($maxIt = 2000$) iterations of the genetic algorithm.

The operator is simply checking that all individuals of the genetic algorithm has a node that is still in the stack. If a node that is not anymore in the stack is

discovered the second mutation operator (that changes the node of an individual) is called.

Moreover, if there is no improvement of the best solution using the genetic algorithm for quite a while and if we are running the method on a mono-processor computer, the genetic algorithm can be paused a few seconds in order to give the Branch-and-Bound more cpu time: this is done because in many cases it can takes time to prove that we have found the optimal solution and then trying to improve the best known solution is not relevant.

Finally, the synchronization operator gives the list of solutions with criterion equals to ub to the Branch-and-Bound. The Branch-and-Bound can then quickly explore the corresponding nodes. This has two advantages:

- it removes these solutions of the stack and forces the genetic algorithm to search in other areas of the search space
- creating child nodes of all the nodes that are in the path that lead to these solutions introduce in the stack some nodes in the neighborhood of the best genetic algorithm solutions that can potentially lead either the Branch-and-Bound or the genetic algorithm to better solutions.

Exploration Strategy: We have tried to use the depth first search. In classical Branch-and-Bound, this strategy has the advantage of finding feasible solutions quickly and to keep a reasonable stack size. But for our method, it may not be the best strategy because all the nodes of the stack are within the same area of the search space and it does not help the genetic algorithm much.

So, we have implemented another strategy: the best first strategy that takes the node that has smallest lower bound and within the nodes that have equals lower bound, it takes the deepest one. This has the advantage of keeping a large

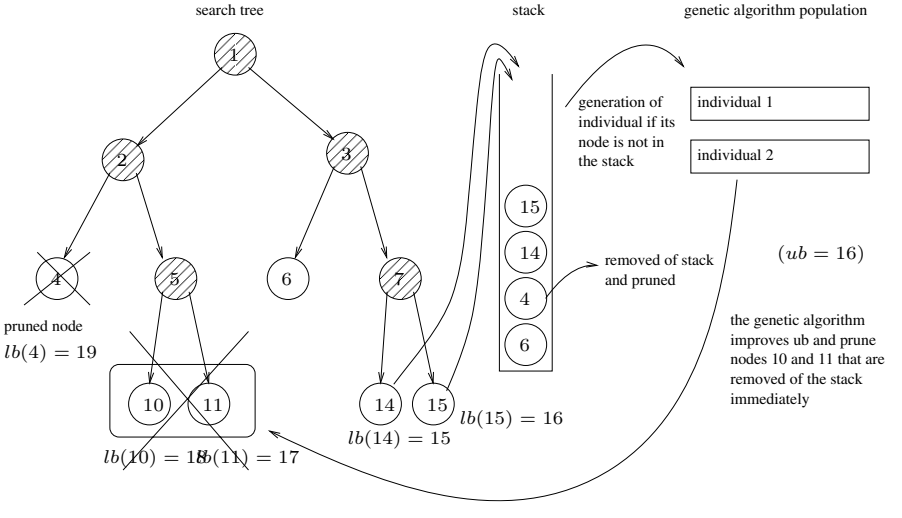


Fig. 7. Synchronization of the stack with the genetic algorithm and effect of the genetic algorithm on the Branch-and-Bound

enough stack that contains nodes that can be within all areas of the search space: it gives a large choice for the genetic algorithm that can be more efficient. Another advantage of this strategy is that it can improve the lower bound of the overall problem and it is an information that can be given to the decision maker to let him decide if it is worth continuing to search the optimal for very long to solve problem. The idea behind this strategy is to use the Branch-and-Bound to cut as many nodes as possible and to finally prove that we have found the optimal while the task of searching good solutions is done by the genetic algorithm.

Notice that we do not have explicit lower bound but a lower bound can be computed using cutting condition: let us consider D^* the smallest value of D for which the cutting condition answers 'no', then $D^* + 1$ is a valid lower bound. Then to compute a valid lower bound at node N different values of D are tested starting from the lower bound of the parent nodes of N. At the root node lb is computed using binary search on D.

Moreover when the synchronization operator is called, it sends to the method that handles nodes priority all the individuals that have a criterion equals to the best solution found until now. The Branch-and-Bound will then immediately explore these nodes. This way, they are removed from the stack and the genetic algorithm will be forced in the next synchronization to search other solutions than these best ones.

4 Experimental Results

The method is coded using Java language and was testing on a monoprocessor machine equipped with a 1.7GHz centrino and 1Gb of RAM. The two algorithms run in their own thread meaning that the program would immediately benefit of a second processor on a multicore or multiprocessor machine.

Preliminary tests have shown that the following parameters give nice results: a population size of 500 individuals, $pm = 30\%$, $pm2 = 5\%$ and $maxIt = 2000$. The method is limited to 10 minutes. The tests have been done on 2000 generated instances with a number of tasks between 10 and 45 and a number of operators between 2 and 10. These have been generated such that they have a similar skill repartition and similar values than what is found in industrial instances. In our industrial case, instances usually have between 30 and 40 tasks. In the results table, we have used the following abbreviation: **bb** means Branch-and-Bound only, **df** means hybrid method with depth-first search and **bf** means hybrid method with best-first search. When tested alone, the Branch-and-Bound is using a depth first search because using a best first search leads to stack size explosion due to the poor upper bound.

We can see on the table 1 that the hybrid method with depth first search is nearly always as good or better than the Branch-and-Bound alone despite the fact that in the hybrid method less nodes can be explored as the processor is shared by both method. Between the two hybrid methods, it looks like the best first search is better for large instances with higher differences starting from

Table 1. Percentage of instances solved to optimality

	n=10			n=15			n=20			n=25			n=30			n=35			n=40			n=45		
m	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf
2	100	100	**100**	100	100	**100**	100	100	**100**	100	100	**100**	100	100	97	97	97	**97**	97	97	**97**	83	83	**83**
3	100	100	**100**	100	100	**100**	100	100	**100**	100	100	97	97	100	97	80	87	**93**	70	70	**77**	50	47	**77**
4	100	100	**100**	100	100	**100**	100	100	97	80	87	**90**	73	73	**80**	50	50	**57**	43	43	**47**	27	33	**50**
5	100	100	**100**	100	100	**100**	83	83	**83**	77	80	77	47	47	**63**	40	43	**57**	23	20	**27**	3	7	**37**
6	100	100	**100**	100	100	**100**	90	93	**93**	60	63	**67**	23	23	**37**	20	23	**27**	3	3	**7**	6	7	**7**
7	100	100	**100**	97	100	97	60	70	60	60	60	57	13	13	**20**	3	13	**20**	3	7	3	0	7	**7**
8	100	100	**100**	93	93	83	47	50	**60**	33	33	30	17	20	**23**	13	13	**13**	3	7	3	6	7	**7**
9	100	100	**100**	90	93	90	53	53	37	23	27	**30**	12	3	0	3	7	**7**	0	3	**7**	3	3	**3**
10	100	100	**100**	90	93	90	13	17	13	10	13	**13**	7	7	3	3	3	0	0	3	**3**	3	3	**3**

Table 2. Average gap between lower and upper bound

	n=10			n=15			n=20			n=25			n=30			n=35			n=40			n=45		
m	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf	bb	df	bf
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.03	.01	.01	.01	.05	.05	.05	.3	.3	.3
3	0	0	0	0	0	0	0	0	0	0	0	.01	.2	.1	.02	.5	.5	.09	.7	.7	.4	1.7	1.7	.2
4	0	0	0	0	0	0	0	0	.02	.6	.6	.1	.9	.9	.2	1.9	1.9	.6	2.7	2.7	.5	2.9	2.9	.6
5	0	0	0	0	0	0	.9	.9	.5	.9	.8	.3	2.7	2.7	.4	3.1	3.1	.6	3	3	1	5.5	5.5	1.6
6	0	0	0	0	0	0	.4	.4	.1	2	2	.7	3.7	3.7	1.1	4.7	4.7	1.4	7.7	7.7	2.3	6.3	6.3	2.1
7	0	0	0	3	3	.03	.6	5.6	1.6	3	3	2	6	6	2.9	6.4	6.4	2.1	7	7	3.4	8	8	3.7
8	0	0	0	5	5	2	11	11	6.3	5.1	5.1	2.5	10	10	3.5	7.9	7.9	3.3	8	8	4.2	9.1	9.2	4.2
9	0	0	0	4	3	1.5	13	13	8.2	15	15	11	9.4	9.4	4.8	8.8	8.8	5.2	9.6	9.6	4.5	10.5	10.5	5.5
10	0	0	0	4	3	1.5	25	25	13	28	28	14	14	14	5.9	9	9	5.5	11	11	5.3	10.4	10.4	6.1

n=30. This corresponds to the size of the instances found in our industrial case.

On the other hand, the table 2 shows the gap in percentage between the lower bound of all the nodes remaining in the stack and the upper bound. It shows that the hybrid method with depth first search rarely improves the gap but best first search improves it significantly. It is mainly due to two factors. The first factor is that the diversity of the stack content gives more chance to the genetic algorithm to improve its best solution than the local search done with depth first search. The second factor is that with the best first strategy, the lower bound of the remaining nodes is naturally improved over time.

5 Conclusion

We have presented how a genetic algorithm can be combined with a Branch-and-Bound method in order to improve both methods. From the point of view of the genetic algorithm, the Branch-and-Bound is there to reduce its search space. From the point of view of the Branch-and-Bound, the genetic algorithm helps to improve the upper bound and thus to prune more node earlier. The results are promising for this method especially when used with a best first search. Moreover, these tests have been done on a monoprocessor machine and meaning it is very promising in the case of multicore processors.

References

[2005] Basseur, M., Lemesre, J., Dhaenens, C., Talbi, E.G.: Cooperation between Branch and Bound and Evolutionary Approaches to solve a BiObjective Flow Shop Problem. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 72–86. Springer, Heidelberg (2005)

[1987] Carlier, J.: Scheduling jobs with release dates and tails on identical parallel machines to minimize the makespan. European Journal of Operational Research 127, 298–306 (1987)

[1995] Cotta, C., Aldana, J.F., Nebro, A.J., Troya, J.M.: Hybridizing genetic algorithms with Branch and Bound techniques for the resolution of the TSP. In: Pearson, D.W., Steele, N.C., Albrect, R.F. (eds.), Artificial Neural Nets and Genetic Algorithms. Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms, Ales, France, pp. 277–280 (1995)

[2001] French, A.P., Robinson, A.C., Wilson, J.M.: Using a Hybrid Genetic-Algorithm/Branch and Bound Approach to Solve Feasibility and Optimization Integer Programming Problems. Journal of Heuristics 7, 551–564 (2001)

[1978] Garey, M.R., Johnson, D.S.: Strong NP-Completeness results: motivations, examples, and implications. Journal of the ACM 5(3), 499–508 (1978)

[2002] Gharbi, A., Haouari, M.: Minimizing makespan on parallel machines subject to release dates and delivery times. Journal of Scheduling 5, 329–355 (2002)

[2005] Gharbi, A., Haouari, M.: Optimal parallel machines scheduling with availability constraints. Discrete Applied Mathematics 148, 63–87 (2005)

[1975] Holland, J.: Adaptation in natural and artificial systems. University of Michigan Press, Michigan (1975)

[2005] Jouglet, A., Sevaux, M., Oguz, C.: Flowshop hybride: de nouvelles perspectives en mêlant algorithme génétique et propagation de contraintes. In: ROADEF 2005, Congrés de la Société Française en Recherche Opérationnelle, Tours, France (2005)

[1978] Lawler, E.L., Labetoulle, J.: On preemptive scheduling of unrelated parallel processors by linear programming. Journal of the ACM 25(4), 612–619 (1978)

[2006a] Pessan, C., Néron, E., Bellenguez-Morineau, O.: Modélisation et planification des opérations de réglage de machines lors de changements de série. In: Gourgand, M., Riane, F. (eds.) MOSIM 2006 conference, vol. 2, pp. 1545–1554 (2006)

[2006b] Pessan, C., Bouquard, J.-L., Néron, E.: An unrelated parallel machines model for an industrial production resetting problem. European J. Industrial Engineering 2(2), 153–171 (2008)

[1998] Portman, M., Vignier, A., Dardilhac, D., Dezalay, D.: Branch and Bound crossed with GA to solve hybrid flowshops. Eur J Oper Res 107, 389–400 (1998)

[2005] Puchinger, J., Raidl, G.R.: Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In: Proceedings of the First International Work-Conference on the Interplay Between Natural and Artificial Computation, vol. 3562, pp. 41–53 (2005)

[2004] Rocha, P.L., Ravetti, M.G., Mateus, G.R.: The meta-heuristic grasp as an upper bound for a branch and bound algorithm in a scheduling problem with non-related parallel machines and sequence-dependent setup times. In: EU/ME Workshop, Nottingham, UK (2004)