New effective neighborhoods for the permutation flow shop problem

Laurent Deroussi, Michel Gourgand, Sylvie Norre

New effective neighborhoods for the permutation flow shop problem

Laurent Deroussi[1] **Michel Gourgand Sylvie Norre**

Research Report LIMOS/RR-06-09

29 Novembre 2006

[1] deroussi@moniut.univ-bpclermont.fr

Abstract

We propose an extension of the Taillard's implementation, which allows to remove efficiently the less well inserted jobs in a permutation. We describe then six new neighborhoods for the permutation flow shop problem. Computational results show clearly that at least three of them are better than the insertion move. Their application into a simple metaheuristic is very effective, since a new upper bound has been found for a hard Taillard's instance.

Keywords: Flow-Shop Problem, Metaheuristic, Neighborhood.

Résumé

Dans ce papier, nous proposons une extension de l'implémentation de Taillard, qui permet de déterminer efficacement quelle est la pièce la moins bien insérée dans un ordonnancement. Nous décrivons alors six nouveaux systèmes de voisinage pour le problème du flow-shop de permutation. Les résultats expérimentaux obtenus montrent clairement qu'au moins trois d'entre eux sont plus performants que le mouvement d'insertion classiquement utilisé dans la littérature. Leur application dans une métaheuristique simple (une recherche locale itérée) confirme l'efficacité de notre approche, puisqu'une nouvelle borne supérieure a été trouvée sur une des instances de Taillard.

Mots-clés: Flow-Shop, Métaheuristiques, Systèmes de voisinage

New effective neighborhoods for the permutation flow shop problem

Abstract. We propose an extension of the Taillard's implementation, which allows to remove efficiently the less well inserted jobs in a permutation. We describe then six new neighborhoods for the permutation flow shop problem. Computational results show clearly that at least three of them are better than the insertion move. Their application into a simple metaheuristic is very effective, since a new upper bound has been found for a hard Taillard's instance.
Keywords: Flow-Shop Problem, Metaheuristic, Neighborhood.

1. Introduction

The permutation flow shop problem (PFSP) is one of the most studied scheduling problems. A set $J=\{1,...,n\}$ of n independent jobs has to be processed on a set $M=\{1,...,m\}$ of m machines in the order given by the indexation of the machines. The processing time for job i on machine j is denoted p_{ij}. The objective is to find the job permutation $\pi=\{\pi_1,\pi_2,...,\pi_n\}$ (in the PFSP, the processing sequence of the jobs is the same for all machines) which minimizes a given criterion, for example the makespan (C_{max}). More formally, the studied problem is classified as a $F\,|\,prmu\,|\,C_{max}$ problem according to the $\alpha\,|\,\beta\,|\,\gamma$–notation introduced by Graham *et al.* (1979).
If t_{ij} is the completion time for job i on machine j, the makespan can be computed as follows:

$$t_{\pi_1 1} = d_{\pi_1 1}$$
$$t_{\pi_i 1} = t_{\pi_{i-1} 1} + d_{\pi_i 1}, \text{ for } i = 2,...,n$$
$$t_{\pi_1 j} = t_{\pi_1, j-1} + d_{\pi_1 j}, \text{ for } j = 2,...,m$$
$$t_{\pi_i j} = max\left(t_{\pi_i, j-1}, t_{\pi_{i-1}, j}\right) + d_{\pi_i j}, \text{ for } i = 2,...,n; \text{ for } j = 2,...,m$$
$$C_{max} = t_{\pi_n m}$$

This problem is known to be NP-hard in general (Rinnooy Kan, 1976). Only small instances can be exactly solved. Researchers have mainly focused their energies towards heuristic approaches. Constructive methods provide a good solution in a short time. Among them (Campbell *et al.*, 1970; Dannenbring, 1977), it is commonly recognized that the best of them is the NEH heuristic (Nawaz *et al.*, 1983). These solutions could be improved by a metaheuristic approach such as simulated annealing (Osman and Potts, 1989; Ogbu and Smith, 1990), tabu search (Widmer and Hertz, 1989; Taillard, 1990; Reeves, 1993; Nowicki and Smutniki, 1996; Grabowski and Wodecki, 2004), iterated local search (Stützle, 1998; Ruiz and Stützle, 2006) and genetic algorithm (Reeves, 1995; Reeves and Yamada, 1998).
All these metaheuristics are based on the notion of neighborhood. A neighborhood is generally defined by a basic move allowing to slightly modify a solution. Three types of moves are considered in the literature for the flow shop problem: (i) swap two consecutive jobs at position i and $i+1$ (swap move), (ii) exchange jobs at positions i and j (exchange

move), (iii) remove job at position i and insert it at position j (insertion move). Local search based on swap move is very fast, but yields local minima with a low quality. Exchange moves and insertion moves give comparable solutions, but it is possible to speed up the exploration of the insertion neighborhood by using the data structure proposed by (Taillard, 1990). So, the whole insertion neighborhood of a solution can be evaluated in $O(n^2m)$, as fast as for the swap neighborhood. Further, (Nowicki and Smutnicki, 1996) define the critical path concept, which allows to reduce the size of the insertion neighborhood, and to speed up yet its evaluation. Nevertheless, according to (Reeves and Yamada, 1998) and (Ruiz and Stützle, 2006), the use of critical paths adds a significant degree of complexity.
As a result of these neighborhood considerations, the insertion move is regarded as the best choice for the PFSP. Despite its efficiency, it remains a very simple move when compared with the sophisticated LK move (Lin and Kernigham, 1973) for the traveling salesman problem (TSP). Moreover, it is showed that for TSP, LK moves are much more efficient than simple moves (swap, exchange, insertion or 2-opt moves). Our main motivation lies in this fact. In order to reach a state-of-the-art level, researchers have developed for the PFSP complex techniques for the reduction of the insertion neighborhood size. Cannot we propose more complex (and more efficient) moves for this problem?
To answer this question, we draw our inspiration from the LK move, and we adapt it for the PFSP. Basically, the LK move can be seen as a recursive move, in which, at each step, we remove an edge of the current tour, and we add another edge (according to given criteria). This idea could be adapted to the PSFP, by designing a recursive move, in which a step consists in removing a job, for reinserting it elsewhere. Owing to the Taillard's implementation, we know how to insert efficiently a job in the permutation. In order to apply this idea, we also need to remove efficiently a job from a permutation.
This paper is organised as follows. In section 2, we propose to extend the Taillard's implementation for the case of job remove. In section 3, we describe the three classical moves for the PFSP and we propose six new moves. Section 4 gives a description of the iterated local search metaheuristic. In section 5, we present a complete experimental comparison of the proposed moves. We conclude in section 6 by some remarks and guidelines for further works.

2. Extension of the Taillard's implementation

2.1. Recall for efficient job insertion

We will first recall in algorithm 1 the Taillard's implementation, which allows to know the best position for inserting a given job (indexed k) in a partial permutation of $k-1$ jobs. More precisely, it allows to compute the k makespans obtained by inserting job k at the i^{th} position ($1 \le i \le k$) in $O(km)$ (as for the makespan calculation).

2.2. Proposition for efficient job remove

Once the earliest completion times and the tails are determined, it is easy to compute all makespans obtained by removing any one of the jobs. We present in algorithm 2 the extension of the Taillard's implementation in the case of a job remove.
We can note that the two first steps are the same, just excepted for the bound of the variable i (the partial permutation contains $k-1$ jobs in algorithm 1, and k jobs in algorithm 2).

Obviously, the calculation of M_i' has a complexity of $O(km)$. Consequently, it is possible to compute the k makespans of step 3 in $O(km)$.

Algorithm 1. The Taillard's implementation for fast job insertion.

1. Compute the earliest completion time e_{ij} of the i^{th} job on the j^{th} machine; the starting time of the first job on the first machine is 0.
   $e_{0j} = 0$; $e_{i0} = 0$; $e_{ij} = \max\left(e_{i,j-1}, e_{i-1,j}\right) + d_{ij}$ for $i = 1,\ldots,k-1$, $j = 1,\ldots,m$
2. Compute the tail q_{ij}, i.e. the duration between the starting time of the i^{th} job on the j^{th} machine and the end of the operations.
   $q_{kj} = 0$; $q_{i,m+1} = 0$; $q_{ij} = \max\left(q_{i,j+1}, q_{i+1,j}\right) + d_{ij}$ for $i = k-1,\ldots,1$, $j = m,\ldots,1$
3. Compute the earliest relative completion time f_{ij} on the j^{th} machine of job k inserted at the i^{th} position.
   $f_{i0} = 0$; $f_{ij} = \max\left(f_{i,j-1}, e_{i-1,j}\right) + d_{kj}$ for $i = 1,\ldots,k$, $j = 1,\ldots,m$
4. The value of the partial makespan M_i when adding job k at the i^{th} position is:
   $M_i = \max_{j=1,\ldots,m}\left(f_{ij} + q_{ij}\right)$ for $i = 1,\ldots,k$

Algorithm 2. Proposed implementation for fast job remove.

1. Compute the earliest completion time e_{ij} of the i^{th} job on the j^{th} machine; the starting time of the first job on the first machine is 0.
   $e_{0j} = 0$; $e_{i0} = 0$; $e_{ij} = \max\left(e_{i,j-1}, e_{i-1,j}\right) + d_{ij}$ for $i = 1,\ldots,k$, $j = 1,\ldots,m$
2. Compute the tail q_{ij}, i.e. the duration between the starting time of the i^{th} job on the j^{th} machine and the end of the operations.
   $q_{k+1,j} = 0$; $q_{i,m+1} = 0$; $q_{ij} = \max\left(q_{i,j+1}, q_{i+1,j}\right) + d_{ij}$ for $i = k,\ldots,1$, $j = m,\ldots,1$
3. The value of the partial makespan M_i' when removing the job at the i^{th} position is:
   $M_i' = \max_{j=1,\ldots,m}\left(e_{i-1,j} + q_{i+1,j}\right)$ for $i = 1,\ldots,k$

2.3. Criteria for the choice of the removed job

In the case of job insertion, the best possible choice of the insertion position is obviously the position p that minimizes the makespan, i.e. the position p such that $M_p = \min_{i=1..k}(M_i)$. The same criterion can also be chosen in the case of job remove (we suppress the job at the p^{th} position, such that $M_p = \min_{i=1..k}(M_i')$; we denote this criterion the absolute remove). But doing that, we may choose in priority jobs with high sum of processing times to the detriment of others. For this reason, we propose also a relative remove, which consists in maximizing the gain of makespan relatively to the sum of processing times of the removed job. The relative remove can be stated as follows: remove the job at the p^{th} position, such that

$M_p = \max_{i=1..k} \left(\frac{M - M'_i}{\sum_{j=1..m} p_{ij}} \right)$, where M is the makespan of the permutation before removing the job. Preliminary results have shown the superiority of the relative remove. So, we will only consider the latter in the following of this paper.
In the next section, we use Taillard's implementation and its extension for describing existing moves (swap, exchange and insertion) and for proposing new moves.

3. Neighborhoods for the PFSP

The description of standard and new moves uses the following subroutines:
$insert(\pi, p, i)$: this subroutine inserts the job i at the position p in the permutation π. We obtain a new permutation $\pi' = (\pi_1, \ldots, \pi_{p-1}, i, \pi_p, \ldots, \pi_n)$ where n is the number of jobs in the permutation π, and π_α represents the α^{th} job in the permutation π.
$p \leftarrow best\_insert(\pi, i, \{constraints\})$: this subroutine inserts the job i at the best possible position p in the permutation π. The position, which is returned by the subroutine, is determined by the makespans computed by algorithm 1, and must satisfy a given set of constraints.
$i \leftarrow remove(\pi, p)$: this subroutine removes the job i, which is returned, at the p^{th} position in the permutation π. We obtain a new permutation $\pi' = (\pi_1, \ldots, \pi_{p-1}, \pi_{p+1}, \ldots, \pi_n)$.
$i, p \leftarrow best\_remove(\pi, \{constraints\})$: this subroutine removes the less well inserted job i that satisfies a given set of constraints in the permutation π. The position p, which is determined by the makespans computed by algorithm 2, and the job i are returned by the subroutine.
We propose first to describe the standard moves using these subroutines.

3.1. Description of the standard neighborhoods

As we mention above, we find mainly three moves in the literature: the swap moves, the exchange moves and the insertion moves.

3.1.1. The Swap moves (SW-moves)

We give a formulation of swap moves in algorithm 3. Local search algorithms based on these moves don't allow to reach good quality solutions. The size of the neighborhood is $n-1$, and it can be examined in $O(n^2 m)$.

Algorithm 3. Swap moves.

```
// Exchange two consecutive jobs at positions p and p+1, p = 1,...,n-1
i ← remove(π, p)
insert(π, p+1, i)
```

3.1.2. The exchange moves (E-moves)

The E-move consists in exchanging two jobs at position p_1 and p_2 with $p_1 < p_2$. Its formulation is given in algorithm 4. The size of the neighborhood is $\frac{n(n-1)}{2}$. Local search algorithms based on E-moves provide good quality local minima, but the neighborhood exploration is in $O(n^3 m)$.

Algorithm 4. Exchange moves.

```
// Exchange two jobs at positions p1 and p2 (p1 < p2)
i1 ← remove(π, p1)
insert(π, p2 − 1, i1)
i2 ← remove(π, p2)
insert(π, p1, i2)
```

3.1.3. The Insertion moves (I-moves)

This move consists in removing the job at position p_1, and inserting it at position p_2 ($p_1, p_2 \in 1,\ldots n$, $p_2 \notin \{p_1 - 1, p_1\}$ (Nowicki and Smutnicki, 1996)). I-moves give at least the same quality solution than E-moves. The size of the neighborhood is $(n-1)^2$ but it can be evaluated in $O(n^2 m)$ using the Taillard's implementation (Taillard, 1990). Consequently, this move is considered by most of the authors afterwards. Its description is given in algorithm 5.

Algorithm 5. Insertion moves.

```
// Insert the job at position p at the best possible position
i ← remove(π, p)
best_insert(π, i, ∅)
```

3.2. Proposition of new neighborhoods

We present in this section new moves for the PFSP.

3.2.1. The Best Exchange moves (BE-moves)

We can note that I-moves can be obtained from SW-moves by replacing the insertion at the position $p+1$ by the best possible insertion. We can apply this method to E-moves and we obtain the Best Exchange moves (BE-moves) (algorithm 6).

We don't have any more the symmetry between p_1 and p_2. This leads to consider two cases depending on whether $p_1 < p_2$ or not. The size of neighbourhood is now $n(n-1)$. It can be explored in $O(n^3 m)$. BE-moves have the same drawback that E-moves; namely the exploration of the whole neighborhood requires an important CPU time. In return, it is expected that local search algorithms obtain good quality solutions.

Algorithm 6. Best Exchange moves.

// BE move between the positions p_1 and p_2
$i_1 \leftarrow remove(\pi, p_1)$
if $(p_2 > p_1)$ then
    $insert(\pi, p_2 - 1, i_1)$
else
    $insert(\pi, p_2, i_1)$
endif
if $(p_2 > p_1)$ then
    $i_2 \leftarrow remove(\pi, p_2)$
else
    $i_2 \leftarrow remove(\pi, p_2 + 1)$
endif
$best\_insert(\pi, i_2, \varnothing)$

3.2.2. The k-Exchange moves (k-E-moves)

The E-moves consists in removing a job i_1, and to insert it instead of another job i_2. The latter is then inserted at the position of the first job. In BE-moves, i_2 is inserted at the best possible position. If the new permutation thus obtained is better than the previous one, that's okay. If not, we can continue. i_2 is inserted at the best possible position, instead of a third job i_3, and so on while we don't find a better solution, or while we don't satisfy a stop criterion. So it is possible to define a recursive move; the *k*-Exchange move, where *k* is dynamically evaluated (algorithm 7). The constraint $\{p_i \neq p\}$ ensures that job i is inserted at a different position in the permutation, and a new permutation is effectively built. The move is stopped is no improvement has been found in the $kmax$ first steps. The size of the neighbourhood is n, but the evaluation of a neighbour is in $O(kmax \times nm)$ at the worst case. We propose to take $kmax = \lfloor \sqrt{n} \rfloor$, which realises a good compromise between the depth of the search and the increase of the complexity. We can note that k-E move with $kmax = 1$ corresponds to I-move.

3.2.3. The Best Removed Exchange moves (BRE-moves) and fast BRE-moves

We have proposed two new moves, but without using the fast job remove. This could be done by replacing the second remove in the definition of BE-moves by a best remove. We obtain the move described in algorithm 8. The constraints defined in the subroutine *best_remove* forbid to remove the job we have just inserted. BRE-moves have the same complexity that BE-moves. We can define a fast neighbourhood evaluation by replacing the first *insert* subroutine by *best_insert* (one must verify that no improvement is found after the first insertion). We obtain the *fast BRE*-moves (algorithm 9) for which the neighbourhood can be evaluated in $O(n^2 m)$. These moves can be generalised by the recursive *k*-Insert moves defined in the next section.

Algorithm 7. k-Exchange moves.

// k-E move build from the initial position p
$found \leftarrow false$
$k \leftarrow 0$
While $(not\ found)$ and $(k < kmax)$ do
    $M \leftarrow makespan(\pi)$
    $i \leftarrow remove(\pi, p)$
    $best\_insert(\pi, i, \{p_i \neq p\})$
    $M' \leftarrow makespan(\pi)$
    if $M' < M$ then
        $found \leftarrow true$
    else
        $k \leftarrow k+1$
    endif
end

Algorithm 8. Best Remove Exchange moves.

// BRE move between the positions p_1 and p_2
$i_1 \leftarrow remove(\pi, p_1)$
if $(p_2 > p_1)$ then
    $insert(\pi, p_2 - 1, i_1)$
else
    $insert(\pi, p_2, i_1)$
endif
if $(p_2 > p_1)$ then
    $i_2, p_2' \leftarrow best\_remove(\pi, \{p_2' \neq p_2 - 1\})$
else
    $i_2, p_2' \leftarrow best\_remove(\pi, \{p_2' \neq p_2\})$
endif
$best\_insert(\pi, i_2, \varnothing)$

3.2.4. k-insertion moves (k-I moves) and fast k-insertion moves

There are several manners to see k-I moves. It can result from the fast BRE-move in which we reiterate the instruction *if…then…endif*. It can also be seen as k-E moves in which we replace the subroutine *remove* by *best_remove*. We obtain the move described in algorithm 10. In order to prevent (or at least to reduce) cycles, a removed job becomes tabu and it cannot be chosen again in the next iterations. Starting from a current solution, we can generate n neighbours which can be evaluated in $O(kmax \times nm)$. As for k-E moves, we take $kmax = \lfloor \sqrt{n} \rfloor$.

Fast k-I moves are obtained simply by fixing as initial position the position obtained when removing a job with algorithm 2. Fast k-I moves define a very peculiar neighbourhood, in the

sense that each solution of the search space has at most one neighbour (0 if the while loop ends with the value $found = false$; 1 if it ends with the value $found = true$). Nevertheless, this kind of neighbourhood can be attractive for very large scale instances, or if a solution must be returned in a short CPU time. (Stützle, 1998) mentions that "for large FSP instances the computation time for the local search still grows fast".

Algorithm 9. Fast BRE-moves.

```
// fast BRE move build from the initial position p_1
M ← makespan(π)
i_1 ← remove(π, p_1)
p_1' ← best_insert(π, i_1, {p_1' ≠ p_1})
M' ← makespan(π)
if M' > M then
        i_2, p_2 ← best_remove(π, {p_2 ≠ p_1'})
        p_2' ← best_insert(π, i_2, {p_2' ≠ p_2})
endif
```

Algorithm 10. k-Insertion moves.

```
// k-I move build from the initial position p
found ← false
k ← 0
M ← makespan(π)
i ← remove(π, p)
TabuJobs ← {i}
While (not found) and (k < kmax) do
        best_insert(π, i, {p_i ≠ p})
        M' ← makespan(π)
        if M' < M then
                found ← true
        else
                k ← k + 1
                i, p ← best_remove(π, p, {i ∉ TabuJobs})
                TabuJobs ← TabuJobs ∪ {i}
        endif
end
```

4. Iterated local search

To compare these moves, we apply a simple but effective metaheuristic called Iterated Local Search (ILS). Let us briefly recall the principle algorithm (algorithm 11) as it is presented by (Lourenço *et al.*, 2003).

Algorithm 11. Principle algorithm of Iterated Local Search metaheuristic.

$s_0 \leftarrow$ GenerateInitialSolution()
$s^* \leftarrow$ LocalSearch(s_0)
While stopping criterion is not met Do
    $s' \leftarrow$ Perturbation($s^*, history$)
    $s^{*'} \leftarrow$ LocalSearch(s')
    $s^* \leftarrow$ ApplyAcceptanceCriterion($s^*, s^{*'}, history$)
End While

Iterated local search is based on a local search subroutine, which modify a current solution s into a local minimum s^*. The perturbation of a local minimum must be strong enough to leave the current local minimum, but weak enough to keep memory of the current local minimum. The acceptance criterion is used to decide from which local minimum the search is continued. A new local minimum can be always accepted (*random walk* strategy), accepted only if it is better than the current local minimum (*best walk* strategy) or any compromise between these two strategies.
We have chosen ILS mainly for its simplicity. As ILS does not contain any complex mechanisms, it is in position to correctly restore the neighborhood ability of obtaining good solutions.

5. Experimental results

We suggest comparing all these moves in the following way: we first measure the intrinsic performance of each neighborhood (both in term of solution quality and in term of CPU time). We consider then a simple metaheuristic base on each neighborhood, and we make short runs and long runs. For the comparisons we use the standard benchmark set of Taillard (1993) and we focus on hard instances of size (50x20), (100x20) and (200x20). Many of these instances still remain unsolved.

Table 1. Best known lower and upper bounds for Taillard's instances as of April 2005.

(50x20) instances		
Name	**LB**	**UB**
TA051	3771	3850
TA052	3668	3704
TA053	3591	3640
TA054	3635	3723
TA055	3553	3611
TA056	3667	3681
TA057	3672	3704
TA058	3627	3691
TA059	3645	3743
TA060	3696	3756

(100x20) instances		
Instance	**LB**	**UB**
TA081	6106	6202
TA082	6183	6183
TA083	6252	6271
TA084	6254	6269
TA085	6262	6314
TA086	6302	6364
TA087	6184	6268
TA088	6315	6401
TA089	6204	6275
TA080	6404	6434

(200x20) instances		
Instance	**LB**	**UB**
TA101	11152	11195
TA102	11143	11203
TA103	11281	11281
TA104	11275	11275
TA105	11259	11259
TA106	11176	11176
TA107	11337	11360
TA108	11301	11334
TA109	11145	11192
TA110	11284	11288

We give in Table 1 the best known lower bounds (obtained by branch-and-bound methods) and upper bounds (the best known solutions) for each instance. The performance measure used is the percentage increase over the best known upper bound ($100\times\frac{x-UB^*}{UB^*}$ where x is the makespan of the obtained solution and UB^* the best known upper bound).
All the programs are written in C and run on a Pentium IV 3.4 GHz.

5.1. Comparison of local minima

We consider local search based on all the presented moves. We can say that a move is efficient if it finds quickly some good solutions. So, it is a compromise between the quality of the obtained solution and the time spent to reach it. 1000 independent local searches were completed for each move on the instances of size (50x20). The initial solution is randomly generated (of course, we can start local searches from the solution obtained with the NEH heuristic. We obtain thus better results, but the difference between moves is less marked). The results are summarized in table 2. The row time gives the approximate CPU time in seconds.

Table 2. Comparison of moves. Average of 1000 independent local search starting from a random initial solution.

	Initial solution: randomly generated								
	SW	**E**	**I**	**BE**	***k*-E**	**BRE**	**fast BRE**	***k*-I**	**fast *k*-I**
TA051	19.91	7.63	4.88	3.47	4.00	3.20	3.92	3.04	6.50
TA052	20.59	8.83	6.09	4.29	4.98	3.51	4.74	3.38	7.65
TA053	21.69	8.33	6.17	4.40	5.04	3.81	4.71	3.56	7.85
TA054	19.73	7.77	5.30	3.52	4.20	3.16	4.09	3.00	7.27
TA055	23.31	9.08	6.10	4.34	5.04	3.83	4.80	3.59	8.69
TA056	21.81	8.90	5.63	3.89	4.59	3.44	4.47	3.37	7.33
TA057	21.91	9.20	5.84	4.17	4.80	3.60	4.59	3.50	7.91
TA058	21.93	8.92	6.30	4.46	5.10	3.88	4.87	3.68	8.06
TA059	20.33	8.30	5.91	4.14	4.75	3.52	4.53	3.41	7.50
TA060	21.54	8.97	5.13	3.67	4.18	3.36	4.11	3.30	7.10
Average	**21.27**	**8.59**	**5.73**	**4.03**	**4.67**	**3.53**	**4.48**	**3.38**	**7.59**
Time (s)	3	100	16	500	80	1000	35	100	4

We can see that SW-moves and E-moves obtain effectively poor results. BE-moves and BRE-moves provides good quality results (compared with I-moves) but they are slow and they are dominated by *k*-I moves (the latter is both better and faster). Fast BRE-moves and *k*-I-moves obtain promising results because they improve significantly the results of I-moves with a moderate increase of CPU time. *K*-E is dominated by fast BRE. Finally fast *k*-I moves give relatively poor quality results, but are as fast as SW-move. They may be interesting for very large instances.

5.2. Comparison of neighborhoods on short runs

In this section, we investigate the results of short runs for the iterated local search (ILS) metaheuristic. Let us give first some details about our ILS implementation.
The initial solution is given by the NEH heuristic. The local search procedure is based successively on the nine moves. The perturbation is composed by three random exchange moves. The acceptance criterion is based on a simulated annealing type. The parameter T that

simulates the temperature is a geometric sequence of common ratio T_f. T_f is computed such that $T_f = \sqrt[iter]{\frac{T_{end}}{T_0}}$ where

T_0 is the initial temperature. We fix $T_0 = 5$ (We have then a probability of 0.5 for accepting a deterioration of the current solution of $T_0 \times \ln(2)$ at the beginning of the cooling schedule). T_{end} is the expected final temperature (arbitrarily fixed at 10^{-2}).

iter represents the number of local searches (iterations of ILS) that are accomplished during the given time. This parameter is evaluated by running iterated local search during the given running time with the best walk acceptance criterion (which is equivalent to the simulated annealing with null temperature).

The probability of accepting a worse transition is given by the Boltzmann factor $\exp\left(-\frac{\Delta E}{T}\right)$.

For each neighborhood and for each Taillard's instance of size (50x20), 5 independent short runs were performed under the same conditions but with different random number seeds. The stopping criterion is fixed at 60 seconds of CPU time. We indicate in Figure 1 the average performance (one curve represents the average result of 50 runs) as a function of the time. This figure shows clearly the superiority of three moves (fast-BRE, *k*-I and fast *k*-I) which obtain an average performance of 0.80% compared to I-move (1.20%).

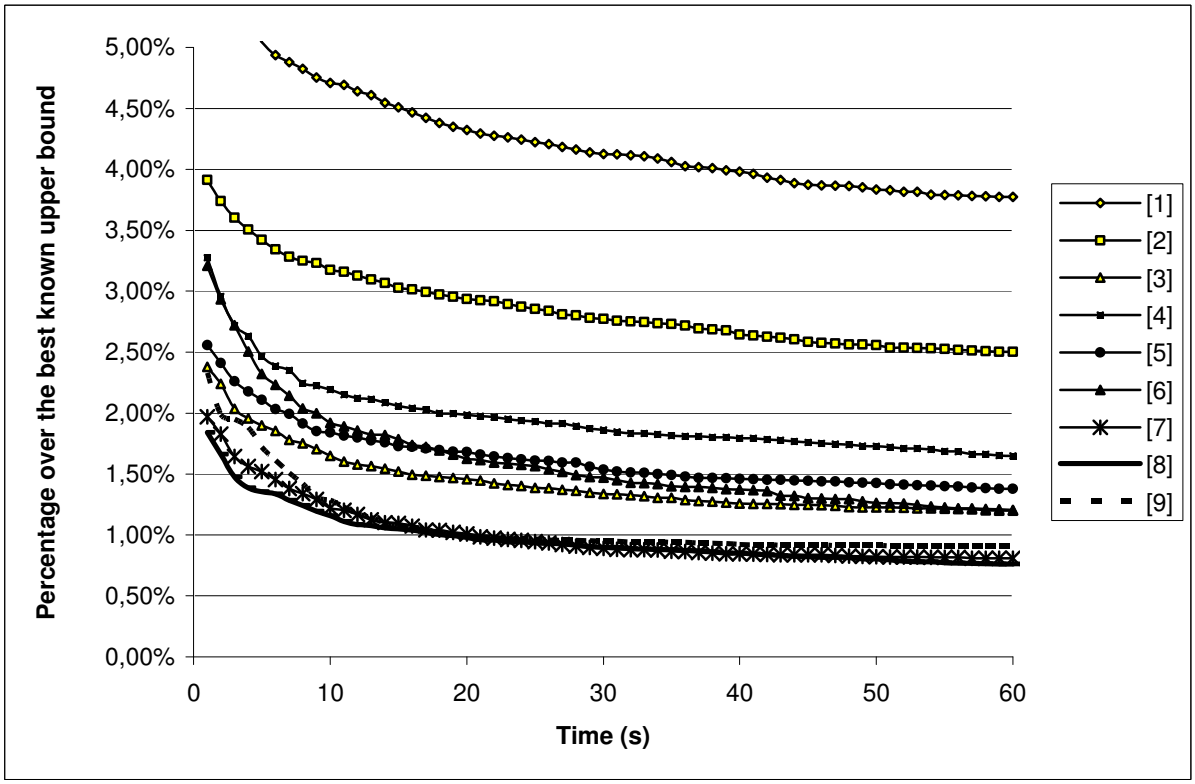

Figure 1. Average performance of each move as a function of time. The legend is:

[1] SW-moves	[2] E-moves	[3] I-moves
[4] BE-moves	[5] *k*-E-moves	[6] BRE-moves
[7] fast BRE-moves	[8] *k*-I-moves	[9] fast *k*-E-moves

We give in table 3 some detailed results for the reference I-moves, and for the three best moves (fast-BRE, *k*-I and fast *k*-I). 60 seconds of CPU time corresponds respectively to 6000, 2600, 800 and 35000 local search iterations. This gives a non-dependent machine running time. Results obtained for I-moves are similar to those published in (Stützle, 1998). The main difference between our ILS implementation and those of Stützle lies in the acceptance criterion. Nevertheless, the author notes that ILS performance is better if sometimes worse solutions are accepted, but most of the acceptance criteria that satisfie this condition give closed results. As can be seen, the moves fast-BRE, *k*-I and fast *k*-I are significantly more effective than I-move. For each of the instances of dimension (50x20), the three latter moves provide better results and they allow to find very good solutions in a reasonable CPU time.

Table 3. Summary of the results obtained with short runs (60 seconds of CPU time). The value in bold type indicates the best found solution.

		I		**fast BRE**		***k*-I**		**fast *k*-I**	
Instance	**UB**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**
TA051	3850	3893	3899.2	3876	3893.0	***3875***	3881.8	***3875***	3888.6
TA052	3704	3721	3733.8	3715	3718.4	***3708***	3718.0	3714	3719.2
TA053	3640	3688	3703.4	***3658***	3675.2	3666	3669.4	3673	3677.2
TA054	3723	3758	3763.2	3748	3754.0	3743	3748.2	***3742***	3760.4
TA055	3611	3640	3651.0	***3626***	3636.2	3632	3644.2	3637	3643.2
TA056	3681	3709	3721.8	***3696***	3707.6	3704	3709.4	3699	3714.6
TA057	3704	3737	3747.2	3729	3732.6	***3727***	3732.0	***3727***	3740.0
TA058	3691	3732	3746.4	3729	3733.8	3716	3723.0	***3715***	3728.4
TA059	3743	3779	3790.8	***3765***	3773.4	***3765***	3777.4	3768	3776.4
TA060	3756	3783	3791.8	***3768***	3777.8	3772	3782.4	3784	3792.0
Average		0.91%	1.20%	0.56%	0.81%	0.55%	0.76%	0.62%	0.91%

5.3. Results of longer runs

The previous results are promising, but they can be still improved by giving more time to each run. So, we decide to make 5 independent longer runs of 3600 seconds for each of the instances of size (50x20), (100x20) and (200x20). The results are provided in table 4.
As can be seen, we obtain very good results for the (50x20) instances. A new upper bound was even found for the TA055 instance (this solution is given in the appendix). However, it seems that the effectiveness of ILS decreases with size instances. The same remark is noticed by (Ruiz and Stützle, 2006). Anyway, the neighborhoods fast BRE, *k*-I and fast *k*-I *always* still obtain better results than I-move. An average run for the (50x20) instances is at 0.25% to the upper bound for k-I moves again 0.55% for I-moves. fast BRE and *k*-I provide equivalent results, while fast *k*-I seems less effective.
These results show clearly the interest of the proposed neighborhoods. They can be used instead of I-move into the literature methods using local search, so improving their efficiency.
To conclude this section, we want to remind that our objective is to compare the efficiency of the neighborhoods. For that reason, the comparison with other methods of the literature has no sense. Nevertheless, we can say that our simple metaheuristic is competitive, especially for the (50x20) instances.

Table 4. Summary of the results obtained with long runs (3600 seconds of CPU time). The value in bold type indicates the best found solution. The symbol "*" means that the obtained solution is a new upper bound.

		I		**fast BRE**		***k*-I**		**fast *k*-I**	
Instance (50x20)	**UB**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**
TA051	3850	3865	3880.2	3858	3863.8	3863	3865.8	**3857**	3867.0
TA052	3704	**3708**	3712.2	3714	3714.2	**3708**	3711.4	3713	3715.4
TA053	3640	3656	3662.2	3651	3656.6	**3641**	3651.6	3653	3657.0
TA054	3723	3733	3740.8	3730	3737.6	**3724**	3732.6	3737	3739.0
TA055	3611	3619	3629.8	3616	3623.2	***3610****	3615.2	3614	3619.2
TA056	3681	3688	3695.4	3690	3694.0	**3687**	3691.2	3690	3693.0
TA057	3704	3717	3725.2	**3710**	3715.6	3711	3712.8	3711	3715.0
TA058	3691	3713	3720.4	**3692**	3700.4	3699	3705.4	3702	3710.2
TA059	3743	3763	3767.2	3754	3759.2	**3747**	3754.2	**3747**	3758.6
TA060	3756	**3767**	3774.2	**3767**	3768.0	**3767**	3768.0	**3767**	3769.6
Average		***0.34%***	***0.55%***	***0.21%***	***0.35%***	***0.14%***	***0.28%***	***0.24%***	***0.38%***
Instance (100x20)	**UB**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**
TA081	6202	6247	6267.0	**6245**	6256.0	6251	6254.8	6254	6262.6
TA082	6183	6234	6245.8	6226	6234.8	**6208**	6219.6	6230	6240.8
TA083	6271	6310	6322.0	**6300**	6309.0	**6300**	6304.4	**6300**	6311.0
TA084	6269	**6303**	6321.0	**6303**	6311.0	**6303**	6303.0	6320	6335.8
TA085	6314	6378	6392.2	6350	6364.8	**6344**	6349.2	6350	6364.0
TA086	6364	6437	6474.4	6403	6417.6	**6384**	6404.8	6417	6443.0
TA087	6268	6315	6322.8	6298	6304.8	**6296**	6305.0	6302	6308.6
TA088	6401	6434	6456.6	**6428**	6441.0	6434	6442.6	6440	6448.6
TA089	6275	6321	6338.8	6310	6328.2	**6303**	6314.2	6321	6330.6
TA090	6434	6478	6481.6	6463	6475.0	**6444**	6464.6	6477	6482.0
Average		***0.76%***	***1.02%***	***0.55%***	***0.73%***	***0.46%***	***0.61%***	***0.68%***	***0.87%***
Instance (200x20)	**UB**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**
TA101	11195	11286	11325.4	**11261**	11270.0	11274	11281.2	11268	11292.2
TA102	11203	11320	11338.8	11314	11320.8	**11300**	11311.4	**11300**	11321.2
TA103	11281	11416	11443.0	11398	11422.4	**11390**	11404.8	11410	11421.0
TA104	11275	11354	11377.2	11349	11369.8	**11340**	11350.8	11351	11368.6
TA105	11259	11315	11326.6	**11290**	11307.2	11301	11304.6	11316	11332.0
TA106	11176	11311	11328.2	**11255**	11277.2	11272	11288.8	11273	11304.4
TA107	11360	11437	11455.4	11438	11444.6	**11419**	11442.4	11446	11457.8
TA108	11334	11426	11434.6	**11392**	11426.8	11414	11424.2	11433	11439.8
TA109	11192	11303	11353.8	**11267**	11319.6	11289	11304.6	11312	11341.4
TA110	11288	11375	11402.8	11355	11367.6	11350	11367.4	**11345**	11380.4
Average		***0.87%***	***1.09%***	***0.67%***	***0.86%***	***0.70%***	***0.82%***	***0.79%***	***0.97%***

6. Conclusion

In this paper, we have first proposed an extension of the Taillard's implementation (Taillard, 1990). It allows to compute efficiency all the possible partial permutations obtained by removing a job. We propose then six new neighborhoods, and the results show that three of them outperform the insertion move, which is commonly considered as the most effective move for the permutation flow shop problem. Moreover, all of the neighborhoods are easy to implement, on the condition of using the data structures proposed by Taillard.

We have implemented a simple iterated local search metaheuristic. The obtained results, both with short runs and with long runs, show the relevance of our approach. A new upper bound has been found for a hard instance from the benchmark test of Taillard (1993).

These works give promising perspectives:
At first, the neighborhoods we have proposed may allow to improve the methods of the literature, at least those that use local search based on the movement of insertion (Stützle, 98, Ruiz and Stützle, 2006). The notion of critical path (Nowicki and Smutnicki, 1996; Reeves and Yamoto, 1998; Grabowski and Wodecki, 2004)), which speeds up yet the exploration of the insertion neighborhood, may also be investigated.
Furthermore, if we look attentively at the results reported by our neighborhoods, it is difficult to differentiate them. Each of them seems to have its qualities and its drawbacks. It may be interesting to use them all three simultaneously, for example by using a variable neighborhood search (Mladenovic and Hansen, 1998; Hansen and Mladenovic, 2003).
Lastly, the use of more elaborated metaheuristic, in particular population based metaheuristic such as genetic local search or particle swarm optimization may improve the results of iterated local search.

7. References

Campbell, H.G., R.A. Dudek and M.L. Smith. (1970). A heuristic algorithm for the *n* job, *m* machine sequencing problem. *Management Science*, 16, 630-637.
Dannenbring, D.G. (1977). An evaluation of flow shop sequencing heuristics. *Management Science*, 23, 1174-1182.
Grabowski, J. and M. Wodecki. (2004). A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. *Computers & Operations Research*, 31, 1891-1909.
Graham, R.L., E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan. (1979). Optimisation and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5, 236-287.
Hansen, P., and N. Mladenovic. (2003). Variable Neighborhood search. In F. Glover and G. Kochenberger (eds.), *Handbook of Metaheuristics*, Kluwers Academic Publishers, 145-184.
Lin, S. and B.W. Kernigham. (1973). An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21, 498-516.
Lourenço, H.R., O.C. Martin, and T. Stützle. (2003). Iterated local search. In F. Glover and G. Kochenberger (eds.), *Handbook of Metaheuristics*, Kluwers Academic Publishers, 321-353.
Mladenovic, N. and P. Hansen. (1997). Variable Neighborhood Search. *Computers & Operations Research*, 24, 1097-1100.
Nawaz, M., E.E. Enscore Jr and I. Ham. (1983). A heuristic algorithm for the *m*-machine, *n*-job flow-shop sequencing problem. *OMEGA, The International Journal of Management Science*, 11, 91-95.
Nowicki, E. and C. Smutniki. (1996), A fast tabu search algorithm for the permutation flow shop problem. *European Journal of Operational Research*, 91, 160-175.
Ogbu, F.A. and D.K. Smith. (1990). The application of the simulated annealing algorithm to the solution of *n/m/Cmax* flow-shop problem. *Computers & Operations Research*, 17, 243-253.
Osman, I.H. and C.N. Potts. (1989). Simulated annealing for permutation flow shop scheduling. *OMEGA, The International Journal of Management Science*, 17, 551-557.
Reeves, C.R. (1993). Improving the efficiency of tabu search for machine scheduling problems. *Journal of Operational Research Society*,44, 375-382.
Reeves, C.R. (1995). A genetic algorithm for flowshop sequencing. *Computers & Operations Research*, 22, 5-13.
Reeves, C.R. and T. Yamada. (1998). Genetic Algorithms, path relinking, and the flowshop sequencing problem. *Evolutionary Computation*, 6, 45-60.

Rinnooy Kan, A.H.G. (1976). *Machine Scheduling Problems: Classification, Complexity and Computations*. Martinus Nijhoff, The Hague, The Netherlands.
Ruiz, R. and T. Stützle. (2006). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, to appear.
Stützle, T. (1998). Applying iterated local search to the permutation flow shop problem. *Technical Report*, AIDA-98-04, FG Intellecktik, TU Darmstadt.
Taillard, E. (1990). Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47, 67-74.
Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64, 278-285.
Widmer, M. and A. Hertz (1989), A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research*, 41, 186-193.

8. Appendix

new best solution for ta055 (50x20)
40 48 4 2 19 31 50 28 20 49 34 5 23 21 32 25 43 45 44 18 26 36 33 42 27 16 41 14 8 47 39 38 10 6 22 17 30 12 13 3 37 9 7 1 46 24 15 29 35 11
Cost 3610