4
Competitive Deterministic Heuristics

4.1 Introduction

This work focuses on the PFS with the makespan criterion, which was firstly denoted as $F_m|prmu|C_{max}$ by Graham et al. [24]. Its main objective is to introduce new deterministic and polynomial time heuristics which are partially based on the NEH-T heuristic of Taillard [63], extended by an improvement phase. The heuristics construct k solutions and perform l iterations in order to improve them (see Section 4.3). These extensions lead to a total time complexity of $O(n^2 \cdot k \cdot l \cdot (m + \log k))$.

This work is organized as follows. A literature review of heuristics for the PFS is provided in Section 4.1(a). In Section 4.2, the original NEH of Nawaz et al. [42] and the NEH-T heuristics are considered. Section 4.3 discusses the effect of initial job ordering and tie breaking rules on the quality of the solutions generated by the NEH heuristic, suggesting new approaches to explore the solution space of the PFS. The construction and improvement phases of the new heuristics proposed in this work, named NEH-Delta and NEH-Alpha, are presented in Sections 4.4 e 4.5, respectively. In Section 4.6, experiments on the benchmark instances of Taillard [64] are taken, showing that these new deterministic heuristics are competitive with the metaheuristics currently standing among the most effective ones for the PFS. Final conclusions are drawn in Section 4.7.

(a) Previous work review

Since the PFS was proved to be *Strongly NP-hard*, fairly much research was attracted to find good heuristic algorithms for this problem. The algorithms range from deterministic and polynomial time constructive methods to meta-heuristics based on non-deterministic approaches.

PUC-Rio - Certificação Digital Nº 0521491/CA

PUC-Rio - Certificação Digital Nº 0521491/CA

Deterministic approaches appeared already four decades ago. The pioneer work of Johnson [34] was followed by the pairing algorithm of Page [47], Palmer's algorithm [48]'s, the CDS algorithm [7], the algorithms of Gupta [25, 26] and Bonney [5], Dannenbring's effective Rapid Access heuristic (RA) [10] and some further variations based on it, the famous NEH heuristic of Nawaz et al. [42], the algorithm of Hundall [32] and the heuristic of Koulamas [37]. In 1990, Taillard [63] showed how to bring the original runtime of NEH from $O(n^3 \cdot m)$ down to $O(n^2 \cdot m)$ (see Section 4.2). As the NEH is still considered one of the best heuristics for the `PFS`, as demonstrated in recent works such as [54] and [35], there can be found many NEH based works, extending or modifying the original algorithm. Some methods [15, 35, 11] work with different initial orders of the jobs and special tie breaking rules to improve the performance of the original heuristic.

Non-deterministic approaches include metaheuristics such as Tabu Search, Simulated Annealing, Local Search, Genetic Algorithms and Ant Colonies. The Tabu Search from Taillard [63] was one of the first non-deterministic approaches for the `PFS`. One of the today's most competitive metaheuristics is the Iterated Greedy Algorithm of Ruiz and Stützle [56]. They review many of the most effective metaheuristics and, given a specified time limit, outperform all of them in the computational experiments. Other references to metaheuristics for the `PFS` can be found in numerous surveys and reviews such as [14, 54, 31] that appeared due to the large number of algorithms for this problem.

Most of the surveys concerning heuristics for the `PFS` tried to address different scopes instead of giving just an up-to-date review of past works. Framinan et al [14] introduced a new framework to classify deterministic as well as non-deterministic approaches into three phases: index development, solution construction and solution improvement. Ruiz and Maroto [54] made a comprehensive evaluation of deterministic heuristics and metaheuristics for the `PFS`, presenting vast computational experiments regarding its comparison. Hejazi and Saghafian [31] extended this effort to exact methods as well.

4.2 NEH and NEH-T

The NEH algorithm of Nawaz et al. [42] can be described by the following steps:

1. For each job j, compute the sum S_j of its processing times on all

machines:

$$S_j = \sum_{i=1}^{m} t_{i,j}, \ \forall j \in J$$

2. Sort all jobs in descending order of S_j to form the sequence of jobs $j_1, .., j_n$.

3. Take the first two jobs j_1 and j_2 and order them so that the makespan of the sequence given by the ordering of these two jobs is minimized.

4. Do for each of the jobs $j_3, ..., j_n$, successively: Insert the next job into the sequence at the position that results in the smallest partial makespan among all possible insertion positions.

The first step needs $O(n \cdot m)$ time to calculate the sum of processing times for each job and $O(n \log n)$ to sort the jobs. In the following insertions for the n jobs, the original NEH evaluates each insertion position by determining the new makespan in $O(n \cdot m)$. As there are j insertion positions for the *j-th* job to be inserted, it takes $O(n^2 \cdot m)$ to evaluate all insertion positions for one job. Hence, the original NEH has a total time complexity of $O(n^3 \cdot m)$. Taillard [63] shows that the j insertion positions can be evaluated in $O(n \cdot m)$ time using dynamic programming and thus reduced the total running time to $O(n^2 \cdot m)$. This improved version is referred to the NEH-T heuristic.

Due to its exceptional results in fairly short running time, the NEH is up-to-date one of the most discussed and analyzed PFS heuristics. Comprehensive computational experiments [54] argue that, considering its low computation costs, the NEH can still be considered the best heuristic among all deterministic ones.

4.3 Extending NEH

The outstanding practical performance of the NEH heuristic is mainly related to two factors: first, the job grouping mechanism, based on the operation of inserting a new job at the best partial position, can be computed in $O(m)$ time. Second, the order in which jobs are taken for insertion (decreasing sum of processing times) has a considerable impact if compared with other ordering mechanisms. Framinan et al. [15] showed that the initial order of the original NEH is the best among roughly 140 evaluated starting sequences. Recently, Kalczynski and J. Kamburowski [35] presented a new initial order with special tie breaking rules which yields better results than the original NEH order.

PUC-Rio - Certificação Digital Nº 0521491/CA

Considering that the makespan of a partial permutation schedule is an effective criterion to decide whether this partial solution must be explored or discarded, two important points, not directly treated by the NEH, come to mind. First, how to select the partial permutation schedule to be explored in the case of ties, when two or more solutions have the same makespan. Nowicki and Smutnicki [45] constructed a special family of instances for which the NEH algorithm have an approximation factor of $\Omega(\sqrt{m})$ due to ties on the selection of job insertion positions. Second, what is the effect of exploring not only one partial solution with minimum makespan but a set of partial solutions with low makespans.

The effect of initial job ordering and tie breaking rules. The following example illustrates how the initial job ordering and arbitrary tie breaking for the NEH have effect on the quality of the generated solutions. Consider an instance of the PFS composed of $n = 3$ jobs and $m = 9$ machines in which the processing times matrix is defined in Table 4.1.

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9
J_1	$1+2\epsilon$	0	0	0	0	1	0	1	0
J_2	0	$1+\epsilon$	0	0	1	0	1	0	0
J_3	0	0	1	1	0	0	0	0	1

Table 4.1: Example of a processing times matrix

PUC-Rio - Certificação Digital Nº 0521491/CA

Some considerations are necessary at this point. First, zero processing times on the matrix represent extremely small values, once by definition all processing times are strictly positive on the PFS. More precisely, the value represented by 0 is less than $\frac{1}{n+m-1}$. Consider also that $0 \leq \epsilon < 1/2$. The application of the NEH heuristic to such instance leads to an initial insertion ordering (J_1, J_2, J_3) calculated by the sums of jobs processing times. On the insertion phase, J_2 is positioned right before J_1, minimizing the partial makespan of these two first jobs. As consequence, any insertion point for J_3 will result in a final makespan of $5+\epsilon$. However, the optimal solution $\pi = (J_3, J_1, J_2)$ generates a makespan of $4+3\epsilon$. Making ϵ also very small, leads to a gap[1] of about 25% between the optimal solution and the one found by the NEH. This simple example illustrates how the selection of the second best partial solution (insertion of J_2 after J_1) can result in a better choice. In particular, when $\epsilon = 0$ ties can occur

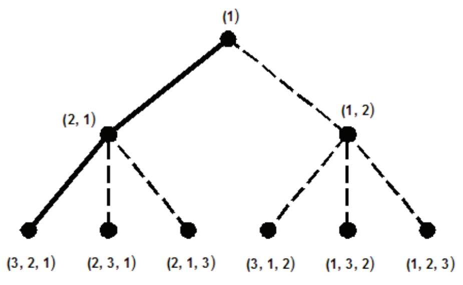
[1]We define the gap of a solution's makespan, denoted by $makespan_{sol}$, as its deviation from the optimal makespan for this instance, denoted by $makespan_{opt}$, i.e. $gap = \frac{makespan_{sol} - makespan_{opt}}{makespan_{opt}}$.

on the initial job ordering phase and on the job insertion phase. Similarly, tie breaking decisions can affect the quality of the generated solutions.

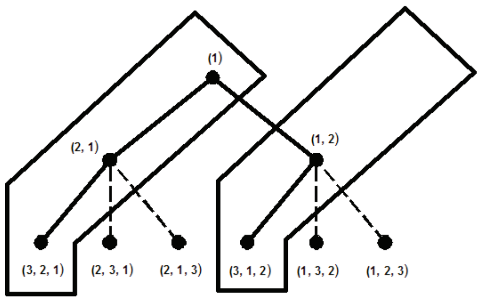
Many works such as Framinan et al. [15] and Kalczynski et al. [36, 35] confirmed the strong influence of the tie breaking decisions and showed that both the initial order and the tie breaking decisions are crucial to the good results of the NEH. Based on such studies, Kalczynski et al [35] presented a new initial order and a simple tie breaking method which outperform the solutions obtained by the original NEH method.

NEH and Enumeration Trees. Consider that the construction mechanism for the enumeration tree of the PFS selects, at the level j, the position $p \in \{1, \cdots, j\}$ in which the j-th job should be inserted in the partial permutation consisting of the first $j-1$ jobs. Once position p is selected, all jobs previously allocated at positions $p' \geq p$ are shifted to positions $p'+1$. Clearly, all possible permutation schedules can be generated from such enumeration process. As consequence, the partial permutation schedule represented by a node is the path from the root of the enumeration tree to it. The NEH heuristic defines a strategy that limits the exploration of every branch of a node on this enumeration tree. Following the insertion phase of NEH, only the node with minimum cost is explored at level j. All other nodes and its corresponding subtrees are pruned. The cost of a node in the enumeration tree is the makespan of the partial permutation schedule that it represents. Figure 4.1(a) illustrates such explicit enumeration process for the previous example with three jobs. The final permutation $\pi = (J_3, J_2, J_1)$ found by NEH is represented by the node $(3, 2, 1)$ on the enumeration tree, the path from root to this node denotes the job insertions carried out by NEH. Dashed edges represent pruned nodes on the enumeration tree.

PUC-Rio - Certificação Digital Nº 0521491/CA



4.1(a): Original NEH



4.1(b): Extended NEH

Figure 4.1: NEH and Enumeration Trees

New approaches to explore the PFS solution space. Following the principles introduced by NEH, a natural extended strategy to explore the solution space consists of considering, at each level of the enumeration tree, not only the node with the minimum cost (makespan of its corresponding partial permutation) but a set of k nodes with costs close to the lowest one on that level. The previously introduced example illustrates the benefits that can be obtained by following such approach, as shown in Figure 4.1(b). In such example, considering that $k = 2$, both nodes $(1, 2)$ and $(2, 1)$ are kept on the second level of the enumeration tree. Consequently, the two solutions of lowest costs are selected at the third level, one of them is the optimal solution. Finally, fixing parameter k or defining it as a polynomial function on input size lead to algorithms that explore the `PFS` solution space in polynomial time.

4.4 Construction Phase

This work proposes two extensions over the NEH heuristic, named NEH-Delta and NEH-Alpha. The common framework governing these extensions is related to the definition of a set of nodes to be explored at a level of the enumeration tree. In the classical NEH heuristic, this set has only one element. The extensions proposed in this work consider sets of nodes with larger cardinality but always limited to a polynomial function on the input size so that the final heuristic can be executed in polynomial time. Throughout this work, we refer the execution of the algorithms NEH-Delta and NEH-Alpha as the *construction phase*.

(a) NEH-Delta

Given a positive integer k defined as a parameter, this heuristic makes use of a greedy strategy to select at most k nodes to be explored from a level of the enumeration tree. All other nodes at this level and its corresponding subtrees are pruned. The k nodes selected at a level are those of minimum cost that are *valid*. A node is called *valid* if none of its ancestral on the enumeration tree was pruned. A node u is said an ancestral of a node v on the enumeration tree if and only if u is on the unique path from the root of the enumeration tree to v. Each level of the enumeration tree corresponds to the insertion of a job, selected on the initialization phase of NEH that sorts the jobs by decreasing sum of its processing times. At the end of the execution, k solutions are obtained from which that of minimum makespan is returned. The complete pseudo-code for the NEH-Delta is presented in Algorithm 4.

PUC-Rio - Certificação Digital Nº 0521491/CA

Algorithm 4: NEH-Delta

```
Input  : Set $J$ of $n$ jobs,
         Set $M$ of $m$ machines,
         Integer $k$,
         Processing times matrix $T \in \Re^{+}_{M \times J}$.
Output: permutation function $\pi : \{1, \ldots, n\} \mapsto J$.
begin
    Sort the jobs in set $J$ in decreasing order of its processing time
    sums resulting in order $(j_1, \ldots, j_n)$ ;
    $\Delta \longleftarrow \{j_1\}$;
    for each job $j_i$, $i = 2 \ldots n$ do
        $\Delta_{next} \longleftarrow \emptyset$ ;
        for each partial permutation $\delta \in \Delta$ do
            for each position $p \in \{1, \ldots, i\}$ do
                Insert job $j_i$ at the position $p$ of partial permutation $\delta$
                generating partial permutation $\delta'$ ;
                if $|\Delta_{next}| < k$ then
                    $\Delta_{next} \longleftarrow \Delta_{next} \cup \{\delta'\}$ ;
                else if $makespan(\delta') < \max_{\underline{\delta'} \in \Delta_{next}} \{makespan(\underline{\delta'})\}$
                then
                    $\Delta_{next} \longleftarrow \Delta_{next} - \{\underline{\delta'}\}$ ;
                    $\Delta_{next} \longleftarrow \Delta_{next} \cup \{\delta'\}$ ;
        $\Delta \longleftarrow \Delta_{next}$ ;
    $\pi \longleftarrow$ permutation of $\Delta$ with minimum makespan;
    return $\pi$ ;
end
```

Implementation details and time complexity analysis. The NEH-Delta implementation implicitly constructs the pruning mechanism on the enumeration tree. Every partial permutation schedule δ is represented by a linked list so that, given a partial permutation δ, a new job j_i can be inserted at position p, generating partial permutation δ', in $O(1)$ amortized time. An important point concerning this time complexity for the construction of δ' is that the elements of δ do not necessarily have to be stored in δ'. In fact, it is possible to store in δ' only a reference (pointer) to δ and the position p in which job j_i should be inserted. However, at the end of an iteration, when the elements of Δ_{next} are copied to Δ, every permutation $\delta' \in \Delta_{next}$ must copy the elements from its originating permutation δ in $O(k \cdot i)$ time. The partial permutation sets Δ and Δ_{next} are implemented as binary heaps using the permutations' makespan as key. Thus, the permutation $\underline{\delta'}$ of maximum makespan in Δ_{next} can be found in $O(\log k)$ time. Similarly, permutations can be inserted or removed from these sets in $O(\log k)$ time.

In the case of makespan ties, the solution that was firstly inserted into the heap

PUC-Rio - Certificação Digital Nº 0521491/CA

always has the priority of being maintained inside the heap. This criterion is used throughout this work as a tie breaking rule. The job sorting initial operation can be accomplished in $O(n \cdot m + n \log n)$ time. Making use of the dynamic programming algorithm of Taillard [63], the makespan of all partial solutions δ' obtained by inserting job j_i in every position of a previous partial solution δ can be generated and calculated in $O(i \cdot m)$. Consequently, NEH-Delta can be implemented in $O(k \cdot n^2 \cdot (m + \log k))$ time.

(b) NEH-Alpha

Though the NEH-Delta tends to yield better results increasing k, this is not a rule. In some rare cases the increase of k leads to a worse final makespan[2], as the better solution is pruned by other partial solutions during the construction phase. This behavior is directly linked to the fact that only the k partial solutions with best makespan are further explored. In order to increase the diversity of the explored solutions, the NEH-Alpha heuristic determines, at each level of the enumeration tree, not an unique set of nodes to be explored but n disjoint sets of nodes. Each set A^p represents the nodes to be explored at level i of the enumeration tree so that the i^{th} job (following the job ordering) is inserted at position p. Furthermore, given a positive integer k defined as a parameter, all sets $A^1, A^2, \cdots, A^n$ must have its cardinalities limited to $\lfloor \frac{k}{n} \rfloor$. Making use from the same greedy criterion introduced for the NEH-Delta heuristic, the $\lfloor \frac{k}{n} \rfloor$ nodes selected to compose the set A^p at a level of the enumeration tree are those of minimum cost that are *valid*. The main difference from this strategy to the one employed by NEH-Delta is that, in this approach, nodes only compete with nodes at the same level of the enumeration tree in which the current job is inserted at the same position. The heuristic obtains up to k solutions (in fact, we have a total of $n \cdot \lfloor \frac{k}{n} \rfloor$ solutions) at the end of the heuristic's execution. The one with minimum makespan is the final solution. The complete pseudo-code for the NEH-Alpha is presented in Algorithm 5.

Implementation details and time complexity analysis. Data structures and implementation strategies used for NEH-Alpha are quite similar to the ones of NEH-Delta. The main difference is that partial permutations sets $A^1 \cup \ldots \cup A^n$ are implemented as n independent binary heaps so that permutations can be inserted or removed from the permutation sets in $O(\log\lfloor \frac{k}{n} \rfloor)$ time. Consequently, NEH-Alpha can be implemented in $O(k \cdot n^2 \cdot (m + \log\lfloor \frac{k}{n} \rfloor))$ time.

[2]As example, we refer to Taillard's benchmark instance tai20_5. The NEH-Delta with $k = 1$ leads to a makespan of 1223, whereas $k = 5$ yields a final solution of makespan 1229.

PUC-Rio - Certificação Digital Nº 0521491/CA

Algorithm 5: NEH-Alpha

```
Input  : Set $J$ of $n$ jobs,
         Set $M$ of $m$ machines,
         Integer $k$,
         Processing times matrix $T \in \Re^{+}_{M \times J}$.
Output: permutation function $\pi : \{1, \ldots, n\} \mapsto J$.
begin
    Sort the jobs in set $J$ in decreasing order of its processing time
    sums resulting in order $(j_1, \ldots, j_n)$ ;
    $A^1 \longleftarrow \{j_1\}$ ;
    for $p \in \{2, \ldots, n\}$ do
        $A^p \longleftarrow \emptyset$ ;
    for each job $j_i$, $i = 2 \ldots n$ do
        for $p \in \{1, \ldots, i\}$ do
            $A^p_{next} \longleftarrow \emptyset$ ;
        for each partial permutation $\alpha \in A^1 \cup \ldots \cup A^i$ do
            for each position $p \in \{1, \ldots, i\}$ do
                Insert job $j_i$ at the position $p$ of partial permutation $\alpha$
                generating partial permutation $\alpha'$ ;
                if $|A^p_{next}| < k$ then
                    $A^p_{next} \longleftarrow A^p_{next} \cup \{\alpha'\}$ ;
                else if $makespan(\alpha') < \max_{\underline{\alpha} \in A^p_{next}} \{makespan(\underline{\alpha})\}$
                then
                    $A^p_{next} \longleftarrow A^p_{next} - \{\underline{\alpha}\}$ ;
                    $A^p_{next} \longleftarrow A^p_{next} \cup \{\alpha'\}$ ;
        for $p \in \{1, \ldots, i\}$ do
            $A^p \longleftarrow A^p_{next}$ ;
    $\pi \longleftarrow$ permutation of $A^1 \cup \ldots \cup A^n$ with minimum makespan;
    return $\pi$ ;
end
```

(c) Computational experiments

All computational experiments in this work were performed using the well known benchmark instances of Taillard [64]. For all instances that already have been solved to optimality, the optimum value is used to compute the gap. For all other instances, the upper bounds shown in Table 4.2 are used. The algorithms were implemented in C++ and compiled under the standard configuration of Visual Studio 2005 Version 8. The experiments were carried out on a Notebook Sony VAIO (VGN-FZ21M), Dual Core 2.0GHz, 3Gb memory, on Windows Vista Home.

PUC-Rio - Certificação Digital Nº 0521491/CA

PUC-Rio - Certificação Digital Nº 0521491/CA

Figure 4.2 shows the average gaps over all instances of Taillard's benchmark for the NEH-Delta and NEH-Alpha for varying k. The results for the NEH-Alpha are only given for $k \geq 500$, as Taillard's benchmark contains instances with up to 500 jobs.

An interesting fact is that the NEH-Alpha performs better than the NEH-Delta for low values of k, but is then outperformed by the NEH-Delta for higher k. Figure 4.3 illustrates the execution times of the algorithms. The linear increase confirms the linear influence of the parameter k in the asymptotic time complexity of the algorithm. Though the execution time grows linearly in k, the gap improvement decreases in both methods.

Instance	lb	ub	Instance	lb	ub	Instance	lb	ub
ta051	3771	3847	ta081	6106	6202	ta101	11152	11181
ta052	3668	3704	ta083	6252	6271	ta102	11143	11203
ta053	3591	3640	ta084	6254	6269	ta107	11337	11360
ta054	3635	3719	ta085	6262	6314	ta108	11301	11334
ta055	3553	3610	ta086	6302	6364	ta109	11145	11192
ta057	3672	3704	ta087	6184	6268	ta110	11284	11288
ta058	3627	3691	ta088	6315	6401	ta111	26040	26059
ta059	3645	3741	ta089	6204	6275	ta112	26500	26520
ta060	3696	3756	ta090	6404	6434	ta116	26469	26477

Table 4.2: Upper and lower bounds for Taillard's benchmark instances (http://mistic.heig-vd.ch/taillard/, May 2008) that have not been solved to optimality yet

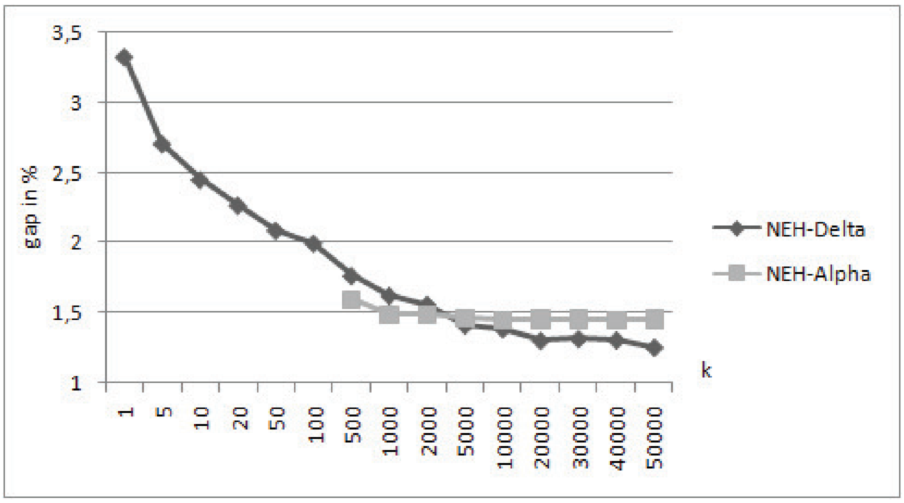


Figure 4.2: Average gap over Taillard's benchmark instances for the NEH-Delta and NEH-Alpha construction phase using varying k

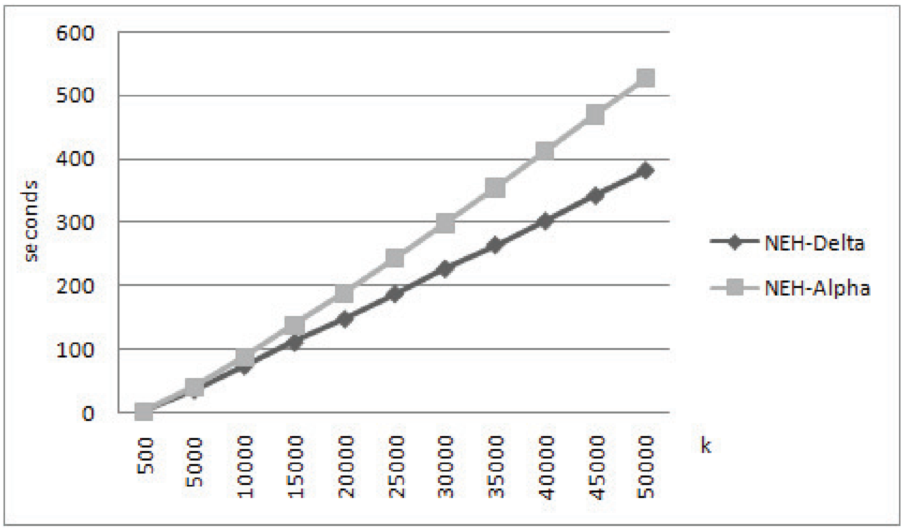


Figure 4.3: Average execution time (in seconds) over Taillard's benchmark instances for the NEH-Delta and NEH-Alpha construction phase using varying k

4.5 Improvement Phase

After keeping active k distinct nodes at each level of the enumeration tree, the application of the algorithms NEH-Alpha and NEH-Delta leads to a final set of k distinct solutions. From this set, the permutation schedule of minimum makespan is selected as the final solution. The central point that motivates these new algorithms relies on the expectation of obtaining better final solutions when increasing the value of parameter k. However, increasing the value of k beyond large values (for example 50.000) seems not to pay-off in relation to the computational efforts, as it can be observed in Figure 4.2. Clearly, it is always possible to achieve optimal solutions by increasing the values of k up to $n!$. These results induced the development of new strategies to improve the permutation schedules generated by NEH-Delta and NEH-Alpha, using the knowledge of such threshold values for k to benefit better from the computational effort and focus on new techniques to enhance the quality of the solutions obtained by the construction phase.

(a) Breadth Search Improvement Phase

As result of the construction phase, after applying either the NEH-Delta or the NEH-Alpha heuristic, a final permutation is selected from a set $S = \{s_1, s_2, \ldots, s_k\}$ of candidate solutions. These candidate solutions are constructed in the last iteration of NEH-Delta or NEH-Alpha, after inserting the last job on the permutation schedules. The objective of the Breadth Search Impro-

PUC-Rio - Certificação Digital Nº 0521491/CA

PUC-Rio - Certificação Digital Nº 0521491/CA

vement Phase (BSI) is to improve the solutions obtained from the construction phase by working over the whole set S. The strategy adopted by the BSI is composed of a sequence of l improvement iterations. Each of such improvement iterations is constituted of n *basic steps*. A basic step removes and reinserts the same job from every permutation in S. After removing a job, all n feasible positions in which it can be reinserted are considered. Hence, applying a basic step of a BSI improvement iteration to a solution $s_i \in S$ creates n possible solutions, including one identical to s_i. However, not all n possible solutions are considered. In fact, from the set of all $n \cdot k$ (possibly new) solutions obtained by removing and reinserting the same job in every permutation of S only k of them with minimum makespan values are selected at the end of a basic step. All other solutions obtained are discarded. The job selected to be removed and reinserted on the i^{th} basic step of a BSI improvement iteration is the i^{th} job in the initial ordering of the classical NEH heuristic. Algorithm 6 outlines the pseudo-code of the method.

Implementation details and time complexity analysis. Each of the l iterations of the BSI is composed of a set of n basic steps. A basic step comprises the deletion of a job from the permutation set, its reinsertion in all n possible positions of every partial permutation and the selection of k solutions to be considered. Hence, the execution time of a basic step is dominated by its n operations of reinsertion and makespan calculations. The deletion of a job can be executed in $O(n)$ time. A job can also be reinserted on a permutation's position p in $O(1)$ amortized time if reinsertions are always taken on subsequent positions. The makespan calculation after reinserting a job can be achieved in $O(m)$ time using the dynamic programming algorithm of Taillard. The set S' can be maintained as a binary heap, bringing a cost of $O(\log k)$ to delete or insert a permutation in S'. As consequence, a basic step can be executed in $O(m + \log k)$ and the total execution time of the BSI is therefore $O(n^2 \cdot k \cdot l \cdot (m + \log k))$. In fact, this work applies two slightly distinct implementation strategies for the BSI improvement phase. The decision of which strategy should be applied is related to the choice of the algorithm used on the construction phase. In the case that NEH-Delta was applied, the BSI implementation strategy is identical to Algorithm 6. However, when NEH-Alpha is used there is a slight modification on the BSI method. Instead of having an unique set S of solutions for each iteration, k disjoint sets A^p of solutions are considered, following exactly the same idea as applied to the construction phase of NEH-Alpha.

Algorithm 6: Breadth Search Improvement Phase

```
Input  : Set $S = \{s_1, s_2, ..., s_k\}$ of solutions obtained from
         construction phase,
         Integer $l$,
         Set $J$ of $n$ jobs,
         Set $M$ of $m$ machines,
         Processing times matrix $T \in \Re^{+}_{M \times J}$.
Output: permutation function $\pi : \{1, \ldots, n\} \mapsto J$.
begin
    Sort the jobs in set $J$ in decreasing order of its processing time
    sums resulting in order $(j_1, \ldots, j_n)$ ;
    $S^1 \longleftarrow S$ ;
    for each iteration $t = 1 \ldots l$ do
        $S' \longleftarrow \emptyset$ ;
        for each job position $q \in \{1, \ldots, n\}$ do
            for each solution $s \in S^t$ do
                $s_{rem} \longleftarrow$ remove job $j_q$ from solution $s$ ;
                for each position $p \in \{1, \ldots, n\}$ do
                    $s_{new} \longleftarrow$ insert job $j_q$ at position $p$ in $s_{rem}$ ;
                    if $|S'| < k$ then
                        $S' \longleftarrow S' \cup \{s_{new}\}$ ;
                    else if $makespan(s_{new}) < \max_{s' \in S'} \{makespan(s')\}$
                    then
                        $S' \longleftarrow S' - \{s'\}$ ;
                        $S' \longleftarrow S' \cup \{s_{new}\}$ ;
        $S^{t+1} \longleftarrow S'$ ;
    $\pi \longleftarrow$ permutation of $S^{l+1}$ with minimum makespan;
    return $\pi$ ;
end
```

PUC-Rio - Certificação Digital Nº 0521491/CA

(b) Depth Search Improvement Phase

This improvement phase aims to improve separately each of the solutions given by the construction phase. As the previously presented improvement phase, this method receives an integer l, indicating the number of times (iterations) that all jobs will be removed and reinserted following the ordering given by the original NEH. In contrast to the BSI phase, the Depth Search Improvement Phase (DSI) inserts each job exclusively at the position that leads to the best makespan and uses the generated solution as starting point for the next job reinsertion. The procedure applied to each of the given starting solutions can be seen as a Local Search with a finite number of steps and a fixed order of the jobs whose neighborhood is examined. Algorithm 7 outlines the pseudo-code of the method.

Algorithm 7: Depth Search Improvement Phase

```
Input  : Set $S = \{s_1, s_2, ..., s_k\}$ of solutions obtained from
         construction phase,
         Integer $l$,
         Set $J$ of $n$ jobs,
         Set $M$ of $m$ machines,
         Processing times matrix $T \in \Re^{+}_{M \times J}$.
Output: permutation function $\pi : \{1, \ldots, n\} \mapsto J$.
begin
    Sort the jobs in set $J$ in decreasing order of its processing time
    sums resulting in order $(j_1, \ldots, j_n)$ ;
    for each solution $s_i \in S$ do
        for each iteration $t = 1 \ldots l$ do
            for each job $j_i$, $i = 2 \ldots n$ do
                $s_{rem} \longleftarrow$ remove job $j_i$ from solution $s_i$ ;
                $s_i \longleftarrow$ insert job $j_i$ at its best position in $s_{rem}$ ;
    $\pi \longleftarrow$ permutation of $S$ with minimum makespan;
    return $\pi$ ;
end
```

Implementation details and time complexity analysis. For each of the k solutions from the set S given in the input data, the algorithm performs l iterations, i.e. the deletion and reinsertion of all n jobs will be performed l times. Using Taillard's dynamic programming algorithm to calculate the makespans for all possible insertion positions, the deletion and reinsertion of a job at the best position costs $O(n \cdot m)$. This leads to a total asymptotic runtime of $O(n^2 \cdot m \cdot k \cdot l)$ for this improvement method. In practice, we can skip further evaluation of a solution if it does not improve during one whole iteration, i.e. after removing and reinserting all n jobs. We also can prune further iterations $t', t'+1, \ldots, l$ on a solution, if the solution's permutation is the same as one of the already examined ones from other solutions generated after the first $t'-1$ iterations. An implementation of such redundancy verifications may change the overall time complexity. However, in our experiments, the use of such pruning turned out to be very effective and significantly reduced the execution time on all instances.

(c) Computational experiments

Figure 4.4 illustrates the influence of the number of iterations during the improvement phase. Both NEH-Delta and NEH-Alpha (with a fixed k = 500) are followed by the Depth and Breadth Search Improvement phases. Two observations can be made. First, the gap improves similarly for each

PUC-Rio - Certificação Digital Nº 0521491/CA

configuration of construction and improvement phase so that a ranking can be established. Second, the average gap seems to converge, i.e. one may limit the number of iterations without a strong influence on the final gap.

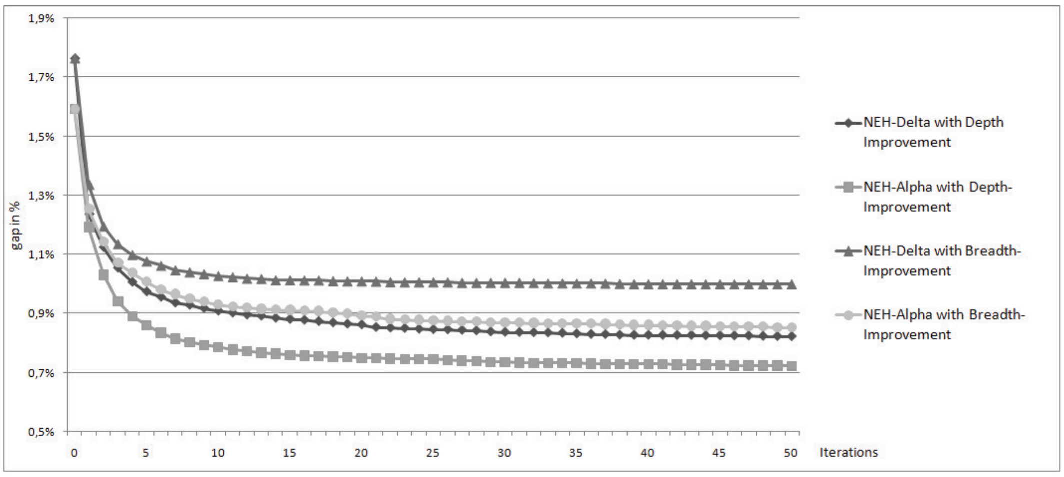


Figure 4.4: Gap improvement during 50 iterations of the two improvement phases combined with the NEH-Delta and NEH-Alpha for k=500 (average gaps over Taillard's benchmark instances)

PUC-Rio - Certificação Digital Nº 0521491/CA

4.6 Performance Comparison

In this section, the competitiveness of the algorithms presented in this work are evaluated by comparing them with the most effective heuristics known for the PFS. In particular, the following algorithms are considered: the **NEH-T** algorithm of Taillard [63], the **NEH-KK1** algorithm from Kalczynski and Kamburowski [35], the **NEH-D** method by Dong et al. [11], the genetic algorithms **GA_RMA** and **HGA_RMA** from Ruiz and Maroto [55], the simulated annealing algorithm **SA_OP** from Osman and Potts [46], the tabu search **SPIRIT** from Widmer and Hertz[66], the genetic algorithm **GA_CHEN** of Chen et al. [9], the genetic algorithm **GA_REEV** from Reeves [52], the hybrid genetic algorithm **GA_MIT** from Murata et al. [39], the genetic algorithm **GA_AA** from Aldowaisan and Allahverdi [1] and the ant colony algorithms **M-MMAS** and **PACO** from Chandrasekharan and Ziegler [8]. Furthermore, we compare our results with the Iterated Local Search **(ILS)** , the Iterated Greedy method **(IG_RS)** and Iterated Greedy with Local Search **(IG_RSLS)** of Ruiz and T. Stützle [56]. Our algorithms were configured as follows:

– NEH-Delta and NEH-Alpha with Depth Search Improvement phase, denoted by NEH-Delta/DSI and NEH-Alpha/DSI respectively. The values for k were chosen as follows: $k = 10000$ for the instances with 20 jobs, $k = 5000$ for instances with 50 jobs, $k = 2000$ for instances with 100 jobs and $k = 1000$ for instances with 200 or 500 jobs. The improvement phase was limited to 10 iterations.

– NEH-Delta and NEH-Alpha with Breadth Search Improvement phase, denoted by NEH-Delta/BSI and NEH-Alpha/BSI respectively. The values for k were chosen as follows: $k = 5000$ for the instances with 20 jobs, $k = 2500$ for instances with 50 jobs, $k = 1000$ for instances with 100 jobs and $k = 500$ for instances with 200 or 500 jobs. The improvement phase was limited to 20 iterations.

The execution time for all algorithms was strictly limited to $(n \cdot m \cdot 3/100)$ seconds, the same stopping criterion as used by Ruiz and Stützle [56] to compare their algorithm with other algorithms for the PFS. If this time limit was exceeded, the best makespan found so far was considered.

In order to compare the results of this work with the ones of others, one must consider two crucial factors that complicate direct comparison. First, the upper bounds of Taillard's benchmark instances were constantly updated in the last years. Hence, works based on this set of instances may have used different values of upper bounds to compute the gaps. Second, the performance of the computational resources may directly impact on the results when the experiments are time limited. Clearly, the fairest way to compare results of different works is to use the original implementation of the authors and execute the algorithms under the same conditions. If this is not the case, the above mentioned points must be considered. For algorithms whose results are not impacted by a time limit within their execution (i.e. the algorithms always terminate before the time limit is reached), the impact of computational resources does not have to be considered.

In our comparisons, these crucial factors were considered as best as possible. The NEH-T algorithm was implemented and executed in the same machine that our algorithms to guarantee a fair comparison. Table 4.3 shows the average gaps per instance group. All NEH-Delta and NEH-Alpha variations considerably improve the results when compared with the original NEH-T. Clearly, the computational effort for these algorithms are much higher.

We now compare with the other algorithms mentioned in the beginning of this section. Table 4.4 compares their average gaps. The original code of Ruiz and Stützle's Iterated Greedy Algorithm with Local Search (IG_RSLS) was

PUC-Rio - Certificação Digital Nº 0521491/CA

Instance	NEH-T	NEH-Delta /DSI	NEH-Alpha /DSI	NEH-Delta /BSI	NEH-Alpha /BSI
20x5	3.30	0.36	0.20	0.39	0.27
20x10	4.60	0.21	0.19	0.22	0.19
20x20	3.73	0.10	0.09	0.03	0.05
50x5	0.73	0.18	0.02	0.18	0.08
50x10	5.07	1.05	0.94	1.53	1.38
50x20	6.68	1.64	1.77	1.89	1.79
100x5	0.53	0.11	0.06	0.21	0.03
100x10	2.21	0.39	0.35	1.03	0.57
100x20	5.34	1.99	1.94	2.28	2.19
200x10	1.26	0.26	0.25	0.47	0.38
200x20	4.42	1.83	1.60	2.30	2.06
500x20	2.07	0.80	0.85	1.22	1.05
Average	3.33	0.74	0.69	0.98	0.84

Table 4.3: Comparison of the average gaps of the NEH-Delta and NEH-Alpha with the original NEH-T

PUC-Rio - Certificação Digital Nº 0521491/CA

executed 20 times, yielding an average gap of 0.417%. In order to compare with the results for the other eleven algorithms listed Ruiz and Stützle by [56], we estimated the difference within the sets of upper bounds and computational resources as explained in the following.

The similarity between sets of upper bound values used in different works is measured by using the average gap of the NEH-T as a reference. Ruiz and Stützle as well as Kalczynski and Kamburowski reported a NEH-T average gap that is very close to the average gap found in the experiments of this work. Thus, it is assumed that the set of upper bounds used by the above authors is similar to the one used in our experiments. Dong et al. seem to have worked with an older set of upper bounds and reported an NEH-T average gap of 2.74%, which is 21.5% less than the average we found. Hence, the gap reported for their algorithm was multiplied by this factor.

Now, possible differences in the computational resources must be considered. Kalczynski and Kamburowski's NEH-KK1 algorithm does not exceed the given time limit, so it is valid to compare their results directly with the ones of our algorithms. Ruiz and Stützle report their results based on the above explained time limit. The average gap reported for their IG_RSLS algorithm is 0.44%, a difference of exactly 5.227% to the gap we found for this algorithm executing it on our machine. In order to have an estimated average gap for the other algorithms used in the comparisons of Ruiz and Stützle, we discount this 5.227% from the average gaps of these algorithms.

Algorithm	Avg gap
IG_RSLS	0.42
HGA_RMA	0.54
NEH-Alpha/DSI	**0.69**
PACO	0.71
IG_RS	0.74
NEH-Delta/DSI	**0.74**
M-MMAS	0.84
NEH-Alpha/BSI	**0.84**
NEH-Delta/BSI	**0.98**
ILS	1.01
GA_RMA	1.07
GA_REEV	1.52
GA_AA	2.16
SA_OP	2.24
GA_MIT	2.30
NEH-D	2.87
NEH-KK1	3.15
NEH-T	3.33
GA_CHEN	4.57
SPIRIT	4.83

Table 4.4: Comparison of the average gaps of the NEH-Delta and NEH-Alpha with other heuristics and metaheuristics by time limited execution

PUC-Rio - Certificação Digital Nº 0521491/CA

All algorithms in Table 4.4, except the NEH-T, NEH-KK1, NEH-D and our algorithms, are metaheuristics, i.e. are non-polynomial and non-deterministic. The metaheuristics of Ruiz and Stützle (IG_RSLS, IG_RS), Ruiz (HGA_RMA) and Chandrasekharan and Ziegler (PACO, M-MMAS) perform extremely well on the set of instances, leading to an average gap lower than one percent. While the NEH-T, NEH-KK1 and NEH-D reach average gaps around three percent, our NEH-Delta and NEH-Alpha approaches lead to gaps lower than one percent, competing well with the other leading metaheuristics.

Farahm et al. [13] presented five polynomial deterministic heuristics while Ruiz and Maroto [54] reviewed further 13 polynomial-time deterministic heuristics for the `PFS`. None of those algorithms reported an average gap smaller then the ones listed above for the heuristics introduced in this work, considering Taillard's benchmark instances.

4.7 Conclusions

This work introduces new deterministic heuristics for the `PFS` based on extensions of the classical NEH algorithm. The development of the new proposed methods was motivated by an analysis of the partial solutions generated during the NEH construction process, exploring the quality of partial solutions close to the ones selected by the NEH heuristic and its relation to the nodes belonging to the enumeration tree of the `PFS`. Following such analysis, the algorithm NEH-Delta was proposed with the objective of considering not only the best partial permutation schedules on a level of the enumeration tree, but a set with k promising solutions to be explored. The NEH-Alpha heuristic also preserves k partial solutions active, but, in order to increase the diversity of solutions, keeps it in n disjoint sets representing the position of the last inserted job. Computational experiments demonstrated a significant enhancement on the quality of the solutions generated by NEH-Delta and NEH-Alpha heuristics when compared to the results obtained from the classical NEH. However, it was observed that, for some threshold values of k, the increase of parameter k leads to little improvement on solutions in comparison with the increase of the total execution time. Inspired by this fact, there were proposed two strategies to comprise a phase of improvement of the solutions generated by NEH-Delta and NEH-Alpha, named Breadth Search Improvement (BSI) and Depth Search Improvement (DSI).

PUC-Rio - Certificação Digital Nº 0521491/CA

NEH-Alpha/DSI, NEH-Alpha/BSI, NEH-Delta/DSI and NEH-Delta/BSI are deterministic heuristics, which can be executed in polynomial time taking parameters k and l as a polynomial function in n or m. Following the methodology introduced by Ruiz and Stützle [56], computational experiments were carried out in order to compare the performance of the new stated methods with currently well performing heuristics for the `PFS`.

Regarding the heuristics here proposed, it is interesting to observe that the best results for the construction phase alone is not confirmed when the improvement phase is added. NEH-Delta beats NEH-Alpha for large values of k in the construction phase. One possible explanation for this phenomenon is that the diversity of the solutions can be enforced by augmenting the number k of partial solutions, not requiring an explicit mechanism for this as NEH-Alpha provides. The comparison of the four new NEH heuristics, improvement phase included, says this is not necessarily true. It allows two observations. The first one is that for both search approaches, BSI and DSI, the NEH-Alpha was dominant. This may be due to the larger diversity imposed by the explicit mechanism which now, with the improvement phase, makes a difference. The second one is that the DSI (depth search) seems to be consistently better. Although the purpose of the application of the breadth search was to provide a more diversified search, it seems that selecting a whole new set of k best solutions at each step may, in fact, lead to a less diversified one, when comparing to equivalent sets of the BSI. In other words, assuming that good heuristics are the ones capable of diversifying and intensifying the search, the dominance of NEH-Alpha/BSI was somewhat expected.

Finally, the experimental results attest that NEH-Alpha/DSI, NEH-Alpha/BSI, NEH-Delta/DSI and NEH-Delta/BSI stand among the most effective heuristics already proposed for the PFS. In particular, from the best of our knowledge, following the experimental methodology introduced by Ruiz and Stützle, no polynomial time deterministic heuristics proposed so far lead to experimental results close to the ones obtained in this work.

PUC-Rio - Certificação Digital Nº 0521491/CA