Politecnico di Torino

Corso di laurea in Ingegneria Gestionale classe L-8
A.a. 2023/2024
Sessione di Laurea luglio 2024

A heuristic for the SDST flowshop sequencing problem

Relatore:
Prof. Salassa Fabio Mario Guido

Candidato:
Racca Elia

1. Introduction

The problem of scheduling N jobs on M machines is one of the classical problems in flowshop scheduling that has interested researchers for many years, this problem revolves around two key elements: a production system of M machines and a set of N jobs to be processed on these machines. Due to their similarity, all N jobs follow the same order of processing on the M machines. The focus of this problem is to sequence or order the N jobs through the M-machine production system so that some measure of production cost is minimized.

Despite an incredible amount of research exists on this flowshop problem, over the years relatively little work has attempted to look into the 'N jobs on M machines' flowshop problem where the set-up times are sequence dependent ($N \times M$ SDST flowshop sequencing problem); such problem occur, especially in process industry operations, whenever processing on a facility has to be shutdown after a particular job is completed, in order to bring the facility to a desired state to process the next job at hand. This changeover delay is typically called 'set-up' time, and its magnitude depends on the similarity in technological processing requirements for the two jobs.
In the literature these problems had been studied starting from the single stage flowshop problems with sequence dependency, due to its similarity to the M-travelling salesman problem. However, the 'N-job, M-machine sequence dependent setup time flowshop problem' has started to be considerated after the publishment of Gupta[1] in 1982, when he proposed a branch and bound procedure to address an M-stage generalized jobshop problem with sequence dependent set-up times. Afterwards, in 1986, Srikar and Ghosh[2] published an article where they discussed the first Mixed Integer Linear Program (MILP) model; it was later corrected by Stafford and Tseng[3] in 1990, when they derived a condition under which the corrected Srikar and Ghosh model can optimally solve SDST flowshop problems even when the triangular inequality of set-up times is violated.

[1] GUPTA, S. K. (1982) 'n jobs and m machines job-shop problems with sequence-dependent set-up times', *International Journal of Production Research*, DOI: 10.1080/00207548208947793
[2] BELLUR N. SRIKAR & SOUMEN GHOSH (1986) 'A MILP model for the n-job, M-stage flowshop with sequence dependent set-up times', *International Journal of Production Research*, DOI: 10.1080/00207548608919815
[3] Jr, E. F. STAFFORD and TSENG, F. T. (1990) 'On the Srikar-Ghosh MILP model for the iVx M SDST flowshop problem', *International Journal of Production Research*, DOI: 10.1080/00207549008942836

Based on those researches and on the Rios-Mercado's work[4] [5] in 1998, Stafford and Tseng proposed two new MILP models for the $N \times M$ SDST flowshop sequencing problem[6] in 2001. After their publishment the literature increased with metaheuristics such as tabu search, simulated annealing and genetic algorithms by Ruiz and Maroto in 2005 and in the following years some advanced MILP models with multi-criteria objectives had been developed by Pan and Li[7] in 2016.
From 2015 to the present, there has been growing interest in hybrid solutions that combine multi-criteria MILP models with machine learning and artificial intelligence techniques. These contemporary approaches aim to enhance the efficiency and effectiveness of solving the $N \times M$ SDST flowshop sequencing problem for many practical cases, reflecting the evolving landscape of industrial optimization.

The heuristic proposed later in this paper by the author has been compared with one of the models formulated by Stafford and Tseng in 2001, which is one of the latest 'single objective function' models and in the next section of this paper it will be reported and briefly discussed in order to better understand the results of this work.

[4] RIOS-MERCADO, R. Z., and BARD, J. F. (1998) 'Computational experience with a branch-andcut algorithm for flowshop scheduling with setups', *Computers and Operations Research*, DOI: 10.1016/S0305-0548(97)00079-8
[5] RIOS-MERCADO, R. Z., and BARD, J. F. (1998) 'Heuristics for the flow line problem with setup costs', *European Journal of Operational Research*, DOI: 10.1016/S0377-2217(97)00213-0
[6] FAN T. TSENG and EDWARD F. STAFFORD Jr (2001) 'Two MILP models for the N × M SDST flowshop sequencing problem', *International Journal of Production Research*, DOI: 10.1080/00207540010029433
[7] KAI ZHOU GAO, PONNUTHURAI NAGARATNAM SUGANTHAN, QUAN KE PAN, TAY JIN CHUA, CHIN SOON CHONG, TIAN XIANG CAI (2016) 'An improved artificial bee colony algorithm for flexible job-shop scheduling problem with fuzzy processing time', *Expert Systems with Applications*, DOI: 10.1016/j.eswa.2016.07.046

2. Mixed Integer Linear Program models

The starting point of this work was to find a Mixed Integer Linear Programming (MILP) model able to solve the $N \times M$ Sequence Dependent Setup Time (SDST) flowshop sequencing problem in the shortest possible time and implement it on python language using the Python-MIP[8] package. This would provide a solid benchmark for comparing the results of the new heuristic presented.
During the research the author found many of the articles previously cited in the introduction, based on this review, the heuristic has been compared to the Stafford and Tseng models proposed in 2001[9].

2.1. The TS1 Model

The first of these models, hereafter referred to as the TS1 model, has been implemented as follows:

Instance parameters

N number of jobs to be processed	$N \in \mathbb{N}$;
M number of machines (or processing steps);	$M \in \mathbb{N}$;
T_{ri} processing time of job i on machine r;	$T_{ri} \in [M \times N]$;
S_{rik} sequence-dependent set-up time for job k on machine r when job i precedes job k in the sequence.	$S_{rik} \in [M \times N \times N]$

Decision variables

$$Z_{ij} = \begin{cases} 1 & if\ job\ i\ is\ in\ position\ j\ in\ the\ sequence \\ 0 & otherwise \end{cases} \qquad Z_{ij} \in \{0,1\}; \quad (1)$$

$$W_{ijk} = \begin{cases} 1\ if\ job\ i\ is\ in\ position\ j\ in\ the\ sequence\ and\ is \\ \ immediately\ followed\ by\ job\ k,\ \ with\ i \neq k \\ 0\ otherwise \end{cases} \qquad W_{ijk} \in \{0,1\}; \quad (2)$$

[8] Python-MIP is a collection of Python tools for the modeling and solution of Mixed-Integer Linear Programs (MILP), '*Python-MIP eases the development of high-performance MIP based solvers for custom applications by providing a tight integration with the branch-and-cut algorithms of the supported solvers…'* https://www.python-mip.com/
[9] FAN T. TSENG and EDWARD F. STAFFORD Jr (2001) 'Two MILP models for the N × M SDST flowshop sequencing problem', *International Journal of Production Research*, DOI: 10.1080/00207540010029433

$X_{rj} = idle\ time\ of\ machine\ r\ before\ start\ of\ job\ in\ position\ j\ in\ the\ sequence \quad X_{rj} \in \mathbb{N};$ (3)

$Y_{rj} = idle\ time\ of\ job\ in\ position\ j\ of\ the\ sequence\ after\ finishing\ processing\ on\ machine\ r \quad Y_{rj} \in \mathbb{N};$ (4)

$C_{rj} = completion\ time\ of\ the\ job\ in\ position\ j\ of\ the\ sequence\ on\ machine\ r \quad C_{rj} \in \mathbb{N};$ (5)

$i \in \{1, \dots, N\};\ j \in \{1, \dots, N\};\ k \in \{1, \dots, N\};\ r \in \{1, \dots, M\}$

Constraint sets

$$\sum_{j=1}^{N} Z_{ij} = 1 \quad \forall i = 1, \dots, N \tag{6}$$

$$\sum_{i=1}^{N} Z_{ij} = 1 \quad \forall j = 1, \dots, N \tag{7}$$

$$\sum_{k=1}^{N} W_{ijk} = Z_{ij} \quad \forall i = 1, \dots, N;\ \forall j = 1, \dots, N \tag{8}$$

$$\sum_{k=1}^{N} W_{k,j-1,i} = Z_{ij} \quad \forall i = 1, \dots, N;\ \forall j = 2, \dots, N \tag{9}$$

$$\sum_{k=1}^{N} W_{kNi} = Z_{i1} \quad \forall i = 1, \dots, N \tag{10}$$

$$\sum_{i=1}^{N} \sum_{k=1}^{N} (S_{rik} - S_{r+1,i,k}) W_{ijk} + \left(\sum_{i=1}^{N} T_{ri} Z_{i,j+1} - \sum_{i=1}^{N} T_{r+1,j} Z_{ij}\right) + \left(X_{r,j+1} - X_{r+1,j+1}\right) + + \left(Y_{r,j+1} - Y_{rj}\right) = 0 \quad \forall r = 1, \dots, M-1;\ \forall j = 1, \dots, N-1 \tag{11}$$

$$\sum_{i=1}^{N} T_{ri} Z_{i1} + \left(X_{r1} - X_{r+1,1}\right) + Y_{r1} = 0 \quad \forall r = 1, \dots, M-1 \tag{12}$$

$$\sum_{p=1}^{j} \sum_{i=1}^{N} T_{Mi} Z_{ip} + \sum_{p=2}^{j} \sum_{i=1}^{N} \sum_{k=1}^{N} S_{Mik} W_{i,p-1,k} + \sum_{p=1}^{j} X_{Mp} = C_{Mj} \quad \forall j = 1, \dots, N \tag{13}$$

Objective function

$$minimize\colon \sum_{j=1}^{N} C_{Mj} \tag{14}$$

The instance parameters consist of an $M \times N$ matrix T_{ri} representing the processing time of the job *i* on the machine *r* and an $M \times N \times N$ matrix S_{rik} which contains the known setup time of a typical job *k* on machine *r* when job *i* precedes job *k* in the processing sequence of jobs.

The solution to the problem is represented in the $N \times N$ matrix of binary variables, Z_{ij}, that are used to allocate each job to one specific position in the sequence.
In order to link adjacent jobs in the sequence, an additional set of binary variables is included: the $M \times N \times N$ matrix W_{ijk}. There are some conventions needed for this variables, in fact when job *i* is the last job in the sequence, and job *k* is the first in the sequence we let $W_{iNk} = 0$, so that the last job "precedes" the first one. Also when $i = k$ we set $W_{ijk} = 0$, so the diagonal of the matrix is null.
X_{rj} is an $M \times N$ matrix of integer variables whose purpose is to show machine idle times in terms of the relative positions of the jobs in the production sequence. Similarly the Y_{rj} set of variables, also an $M \times N$ matrix, represents job idle times in terms of the relative positions of the jobs in the sequence.
Lastly, the $M \times N$ matrix C_{rj} is used to evaluate the completion time of the job *j* on the machine (or stage) *r*. This set of variables is included in the model to accommodate the objective function, which will be explained later in the paper. If a different objective function were used, this set might not be included.

The constraint sets are briefly[10] discussed in order of appearence in the model illustrated above.
Equations (6) and (7) represent 2N constraints that are identical in form and purpose to the constraints of the classical assignment problem. Due to the binary nature of the Z_{ij} variables, in each of these 2N constraints only one of the decision variables will be non-zero. The first equation ensure that each job is assigned to only one position in the sequence, while the second is used to guarantee that every position is occupied by only one job.
Equations (8) and (9) establish the links between adjacent jobs, in fact in a sequence each job precedes exactly one job and follows only one job, and these two jobs are distinct. The first relation, where a job i is in position j if and only if job i precedes the job in position j+1, is represented by the 2N set of constraints (8); while the (9) equation ensure that job i in position j follows only the job in position j-1. There is a special case that could not be covered with the (9) set of constraints, when j=1, we have $W_{k0i} = W_{kNi}$, this convention is represented by the N constraints that follow the (10) equation.

[10] For a full description of the constraint sets the autor suggests to check the original discussion: FAN T. TSENG and EDWARD F. STAFFORD Jr (2001) 'Two MILP models for the N × M SDST flowshop sequencing problem', *International Journal of Production Research*, DOI: 10.1080/00207540010029433, pp. 6-15

Equation (11) represents $(M-1)\times(N-1)$ relationships of equal time slices on adjacent machines for all jobs in the sequence. In particular this set of constraints links the setup time, processing time and idle time of two adjacent jobs in the sequence between two consecutive machines (or production stages).
Equation (12) ensures the starting time of first job in the sequence on all the machines, in fact the first job has to begin the production procedure on a subsequent machine only after it has completed the process, and eventually after an idle time, on the previous machine.
Finally equation (13) represents a set of N constraints that are necessary to measure the N jobs completion times. The completion time of the job in position j is the sum of the times for all activities that preceded the processing of this job on the last machine, M, plus the processing time for this job on machine M.

The objective function selected from the five proposed by Srikar and Ghosh[11] is minimize the mean flow time, in the model it is represented by the equation (14), which aims to reduce the value of the total flow time. The mean flow time is $\bar{C} = \frac{\sum_{j=1}^{N} C_{Mj}}{N}$, minimizing $\sum_{j=1}^{N} C_{Mj}$ automatically returns the minimum of $\bar{C}$ because the denominator of the fraction is a constant and it cannot be reduced.

[11] FAN T. TSENG and EDWARD F. STAFFORD Jr (2001) 'Two MILP models for the N × M SDST flowshop sequencing problem', *International Journal of Production Research*, DOI: 10.1080/00207540010029433, pp. 15-16

3. Heuristic

3.1. Heuristic introduction

The present a new heuristic for the flowshop scheduling problem with sequence dependent setup times (SDST) and the minimization of the mean flow time of the production line. In this section the author explains and analyzes the main processes that led to the development of this heuristic, while also discussing the pseudocode of the algorithm.

3.2. The starting idea

During the development and testing of the model, the author attempted to solve smaller problems in order to verify the optimality of the model's best solution for the proposed instances. The instances tested with complete enumeration of all the feasible solutions for the $N \times M$ SDST flowshop sequencing problem had size of $N = 3{,}5$ and $M = 2$. These sizes were chosen so they could be solved manually on paper within a reasonable time.

While evaluating the first instances, the author observed that the sequences with the lowest mean flow time were those in which the setup procedure for the jobs on the next machine began before the completion of the processing time of that job on the previous machine. Ideally, this arrangement allows a job to be processed on the following machine immediately after finishing its process on the previous machine, without waiting for setup time on the adjacent machine because it has been completed while it was being processed on the previous machine.
Based on these observations, the author elaborated an equation that could find those differences between two jobs on adjacent machines. The parameters used in this equation, as well as in other formulations in this section, are defined using the following notation:

P_{ri}	the processing time of job *i* on machine *r*
S_{rik}	the setup time from job *i* to job *k* on machine *r*

The initial equation formulated and applied on the small size instances by the author was:

$$\Delta = \left(P_{r+1,i} + S_{r+1,i,k}\right) - \left(P_{rk} + S_{rik}\right) \tag{15}$$

The idea behind the heuristic was to calculate the differences between all possible job combinations and select the sequence with the minimum gaps between jobs. To complete the total enumeration of feasible solutions, the author chose sequences with the lowest difference values that had not been used in previous solutions. This initial approach to navigating the solution space was developed manually by the author, solving the first instances on paper.
In the time the author was using the previous equation and heuristical approach, he discovered that the key factor in achieving the best solutions was not the anticipation of the setup procedure on the next machine.
Statistically[12], the factor that led to a lower mean flow time was having a shorter gap in between the end of the processing time on a machine and the start on the next one. This reduces the idleness of both jobs and machines, resulting in smaller completion times for many jobs and, on average, it minimizes the mean flow time.
Based on this conclusion, the author formulated the equation that defines the initial solution and the neighborhood:

$$\Delta = \left|\left(P_{r+1,i} + S_{r+1,i,k}\right) - \left(P_{rk} + S_{rik}\right)\right| \tag{16}$$

The simple addition of the absolute value dramatically improves the effectiveness of the heuristic and the underlying theory. Without the absolute value constraint in equation (15), the differences could be negative, and sorting by the minimum or maximum value of Δ would result in sequences with extremely large gaps. This approach is ineffective for reducing the mean flow time of a production line as the completion times of the jobs increase with the idleness of both machines and jobs.

In contrast, formulation (16) evaluates the magnitude of the gap without considering the sign. Sorting all possible job pairs in ascending order of the Δ value provides an ordered list in which the first elements are the job couples likely to be adjacent in a sequence, minimizing the gap between them, thereby reducing also the second machine's idle time

[12] 25 out of 30 top three sequences, on 10 instances, had been calculated first by following the equation (16) during the manual enumeration of the solutions of the small size SDST problems

or the first job's idleness. This approach, based on the observations above, leads to a lower mean flow time on the production line.

3.3. The definition of the heuristic

After the formulation of the starting idea behind the heuristic, the author needed to select an heuristic method as a model for the development of the heuristic. Following the classification presented by L. De Giovanni in "*Metodi euristici di ottimizzazione combinatoria"*[13] the heuristic proposed in this paper is based on the metaheuristic procedure known as 'neighborhood search'.

The metaheuristics represent general algorithmic schemes designed to provide near-optimal solutions for optimization problems. Unlike traditional optimization methods, metaheuristics are versatile and applicable to a wide range of problems without requiring substantial changes. These methods define components and their interaction, for any application those components are specific for the distinct problem.
Some examples of metaheuristic are *neighborhood search, Simulated Annealing, Tabu Search, Genetic Algorithms, etc…*

The 'neighborhood search' fundamental concept is to define an initial solution (current solution) and attempt to improve it by exploring a neighborhood (appropriately defined) of this solution. If the optimization within the neighborhood of the current solution yields an improved solution, the procedure is repeated, starting with the newly determined solution as the current solution. The algorithm usually ends when it is found a solution with the same value of a predetermined bound, or when it completes a number of iterations, or even when it reaches a time limit. The heuristic proposed in this paper was tested with a time limit of five minutes.

The heuristic presented for the SDST flowshop sequencing problem will be explained following the algorithmic scheme of the 'neighborhood search' exposed by L. De Giovanni[14] and hereafter reported:

[13] L. DE GIOVANNI (n.d.) 'Metodi euristici di ottimizzazione combinatoria', *Metodi e Modelli per l'Ottimizzazione Combinatoria*
[14] L. DE GIOVANNI (n.d.) 'Metodi euristici di ottimizzazione combinatoria', *Metodi e Modelli per l'Ottimizzazione Combinatoria, pp.10-20*

- Rappresentation of the solutions
- Definition of the initial solution
- Definition of the neighborhood
- Evaluation of a solution
- Strategies of exploration in the neighborhood
- Trajectory method (neighborhood search)

The definitions of the single components for the SDST flowshop sequencing heuristic will be followed by a pseudocode that represents the application of the theoretical steps in order to complete the algorithm of the heuristic.

3.3.1. Rappresentation of the solutions

The rappresentation of solutions shows the characteristics of the themselves and provides the foundations for the processes that allow the exploration of the solution space.

The rappresentation of solutions of this heuristic does not follow the classical rappresentation of a sequence traditionally used in the SDST flowshop sequencing instances. Where the solution can be presented as a list of jobs where the position in the list represents also the position in the sequence, or an $N \times N$ matrix of binary variables where if the value at the coordinates (x, y) is non-zero it means that the job x is in the position y in the sequence.
Instead, it focuses on the differences between the end of the processing time of a job on a machine and the previous job on the next machine. This approach ensures, as mentioned before in section 3.1, that the gap between finishing on one stage and starting on the next is as small as possible.

Each sequence in the SDST flowshop sequencing heuristic is treated as an ordered list, where the items are pairs of jobs sorted by the difference between their processing times on adjacent machines, arranged in ascending order. This approach simplifies the creation of a list in which jobs with shorter gaps between them are likely to be close to each other. The following procedure outlines an algorithm to implement this method:

```
algorithm calculateGaps is

for job i in range (0, N) do
    for job k in range (0, N) do
        if i = k do
            skip
        else
            $gap_{ik} \leftarrow \sum_{r\in\{0,M-1\}} \left| (P_{r+1,i}+S_{r+1,i,k}) - (P_{rk}+S_{rik}) \right|$
            listGap ← listGap + ($gap_{ik}$ and jobs associated)

return listGap
```

gap_{ik} is the gap explained above between job i and k;
listGap is the data structure containing the gaps associated to the pairs of jobs.

In the heuristic code, the sorting of the list of differences is performed just before evaluating the initial solution. To ensure that the pseudocode accurately reflects the function developed for the heuristic, the sorting process will be included in the initial solution subsection.

To visualize the sequence, the solution represented by the array of job pairs can be transformed into a more practical and intuitive format, such as a list of jobs where the position in the array indicates the position in the sequence. This transformation also facilitates the calculation of the mean flow time, which will be discussed later in section 3.2.4. The following pseudocode illustrates the conversion from a set of differences and pairs of jobs to a list of jobs:

```
algorithm convertSequence is

for each (job i, job k) in listGap do
    if none of the two jobs in the gap is in finalSequence do
        finalSequence ← finalSequence + job i + job k
    else if job i in finalSequence and job k not in finalSequence
        insert job k in finalSequence after job i and its successor
    else if job i not in finalSequence and job k in finalSequence
        insert job i in finalSequence before job k and its predecessor

return finalSequence
```

```
listGap represents the list that contains only the pairs of job
ordered from the lower gap to the highest.
finalSequence is the listGap converted in a sequence of jobs.
```

3.2.2. Initial solution

Typically, the initial solution in heuristic methods can be generated randomly or by simpler heuristics. However, for the heuristic proposed in this paper, the starting sequence is determined by sorting the list of gaps identified in the previous step. Consequently, the procedure for finding the initial solution is heavily linked to the procedure for ordering the list. This algorithm is significantly influenced by the chosen programming language and the data structures used to store the gap values, as discussed in subsection 3.2.1.

The heuristic discussed in this paper was developed by the author using Python version 3.8.2. In the first phase, a dictionary data structure was used to store and link the job pairs. This dictionary was then sorted into a list of tuples representing the initial solution. The following pseudocode outlines how this approach was implemented in the Python version of the heuristic:

```
algorithm firstSolution is

for each ((job i,job k),gap_ik) in dictGap do
     reverseDict ← reverseDict + (gap_ik,(job i,job k))
sortedReverseDict ← sort reverseDict (by ascending order of gap_ik)
for each (gap_ik,(job i,job k)) in sortedReverseDict do
     listGap ← listGap + (job i,job k)

return listGap

dictGap is the dictionary with the pairs of job associated to the
gaps;
reverseDict is a copy of dictGap dictionary with tuples inverted
sortedReverseDict is the reverseDict dictionary sorted by the value
of gap_ik;
listGap in this pseudocode represents the list that contains only
the pairs of job ordered from the lower gap to the highest.
```

This approach could have been implemented more simply in Python by using the `key` parameter of the sorting function in combination with the `lambda` operator. However, at the time of code development, the author was not yet familiar with these advanced methods.

3.2.3. The neighborhood

The next step for the completion of the heuristic is the definition of the neighborhood. Each solution, referred to as the 'center of the neighborhood,' can be transformed into another potential solution for the specific problem instance by applying certain moves that change some of its characteristics. The new solution formed through this process is called a 'neighbor of the center solution,' and the entire set of neighbors constitutes the 'neighborhood' of a given solution.

In the heuristic described in this work, the moves that define the neighborhood involve removing the first pair of jobs from the solution list and appending it to the end of the list. This approach allows job pairs to gradually gain relevance and become more likely to appear in the final sequence. The following pseudocode represents the algorithm that inverts the position of the first item of a list:

```
algorithm findNeighbors is

for each gap in listGap do
     listGap ← listGap + first couple in listGap
     remove first couple in listGap
```

`listGap represents the list that contains only the pairs of job ordered from the lower gap to the highest.`

In the heuristic this algorithm is implemented in the neighbors search through a recursion. A recursive definition involves the term defined appearing within its own definition. In programming a method is considered recursive when inside its definition it calls itself. In computer science, recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem. The recursive function to explore the neighborhood will be explained more in detail in the specific subsection 3.2.5.

3.2.4. Evaluation of a solution

The objective of the heuristic is to minimize the mean flow time and compete with the MILP model developed in the section 2, in order to lower this objective function it is needed to define and evaluate the mean flow time of a solution.
The mean flow time represents the average time spent by the jobs in the production line. Minimizing the mean flow time reduces the time that jobs spend in the production system, thereby directly increasing the production rate.
The mean flow time is calculated using the following equation:

$$\bar{C} = \frac{\sum_{j=1}^{N} C_{Mj}}{N} \tag{17}$$

Considering equation (17), evaluating a solution involves calculating the completion times of each job. The algorithm that computes the completion times of every job on each machine and uses these values to determine the mean flow time of a given sequence is proposed in the following pseudocode:

```
algorithm evaluateSolution is

$C_{00} \leftarrow P_{00}$
for r in range (1, M) do
    $C_{r0} \leftarrow C_{r-1,0} + P_{r0}$
tft ← tft + $C_{M0}$

for i in range (1, N) do
    $C_{0i} \leftarrow C_{0,i-1} + S_{r,i-1,i} + P_{ri}$
    for r in range (1, M) do
        $C_{ri} \leftarrow \max\{C_{r-1,i} ; C_{r,i-1} + S_{r,i-1,i}\} + P_{ri}$
    tft ← tft + $C_{Mi}$

return tft/N
```

C_{ri} is an N×N matrix representing the completion time of job i on machine r;
tft is the value of the total flow time, by dividing per N the algorithm returns the mean flow time.

The first three lines of the algorithm calculates the completion times of the first job in the sequence. Subsequently, it evaluates the completion times of the remaining jobs in the sequence and adds the total completion time of each job to the total flow time.

3.2.5. Exploration of the neighborhood

The exploration of the neighborhood consists in the transitions from a solution to another within the neighborhood based on predetermined moves. Traditionally, in the local search methods, the transition between two solutions is made when the second is evaluated to be better than the first in terms of the weight or the cost of the solution. The strategy adopted for making this transition is crucial and significantly impacts the effectiveness of the search.

For example the two typical strategies shown in the literature are:

- *first improvement*: the exploration of the neighborhood ends as soon as a better solution than the starting one it is found. This new solution is set as the new 'center of the neighborhood' and a new search begins from this point, or it is retained as best solution.
- *best improvement* (or *steepest descent*): the exploration of the neighborhood does not end when it is found an improving solution. Instead the search continues until the entire neighborhood is explored, or until a given limit is reached (such as a time limit, or a maximum number of iteractions). At the end of the process, the best solution found is set as the new 'center of the neighborhood' and a new neighborhood begins to be explored, or the heuristic return this solution as the best.

As mentioned in the subsection 3.2.3, the exploration of the neighbors in the heuristic proposed has been implemented using a recursive procedure based on the *best improvement*. In fact, the recursion is often used in computer science to simplify a problem by using the concept of 'divide et impera'. This approach consist in splitting a bigger problem in various smaller parts, then solve the sub-problems and combining the partial solutions into the overall solution of the original problem. The recursive procedures follows always a scheme hereafter illustrated:

```
algorithm recursion is

instructions always executed
if terminalCondition do
    doSomething
    stop
else
    for each smallProblem in allSmallProblems do
        computePartial
        if filtro do
            recursion with the smallProblem solved
        backtracking
```

As described above, the recursive approach can be outlined through distinct blocks of instructions. It begins with code that must be executed at every iteration, regardless of whether it is the initial or the final iteration. Following this, the algorithm checks the terminal condition, a critical step to prevent the recursive method from falling into an infinite loop, which would result in endlessly stacking calls without resolution. If the terminal condition is not triggered, the algorithm proceeds to solve the partial task and verifies the validity of the partial solution. If the partial solution is valid, it is passed to a recursive call of the algorithm, which then addresses the subsequent small problem. The final instruction involves backtracking, a step that is used to discard the move previously done in order to allow the exploration of the feasible solutions starting from a new move instead of the first.

In the heuristic the starting problem is to explore the entire space of neighbors of the initial solution. This challenge is dealt with the recursive method, which divides this issue into many small tasks. Each task involves exploring one neighbor of the 'center of the neighborhood' solution, then visiting the neighbor of the subsequent sequence, followed by the neighbor of that sequence, and so forth. This recursive approach allows for a systematic and accurate examination of the neighborhood space.
The following pseudocode outlines how this procedure can be implemented in a program:

```
algorithm neighborExploration is

if time of recursion > 60 seconds do
     stop
if lenght of sequence = N and sequence is better than the best
     best ← sequence
     if lenght of listGap > 80% lenght of bestListGap
          bestListGap ← listGap
     stop
else
     if (recursion explored all the N jobs for this iteration or
     listGap is empty) do
          stop
     else
          remove first couple in listGap
          for each gap in listGap do
               listGap ← listGap + first couple in listGap
               execute neighborExploration with the new listGap
               if listGap is not empty
                    remove first couple in listGap

return best
```

```
best is the sequence with the lower mean flow time until the
iteration;
bestListGap is the list that contains the pairs of jobs ordered with
the lower mean flow time until the iteration;
listGap represents the list that contains only the pairs of jobs
ordered from the lower gap to the highest;
sequence is the listGap converted in a sequence of jobs.
```

The terminal condition of the neighbor exploration is multiple in the heuristic proposed, firstly the algorithm stops if the exploration exceeds one minute, this time constraint is due to the fact that the heuristic has been compared with the Stafford and Tseng models on a five minutes time limit, with this contraint the author wanted to test at least the exploration of five neighborhoods. Secondly when the algorithm identifies a sequence with a lower mean flow time[15] it updates the best solution and if the size of the

[15] The pseudocode that checks if the mean flow time of a sequence is better than the best solution is a simple evaluation of the mean flow time of the given sequence with the `evaluateSolution` algorithm, then if this value is lower than the better found until that iteration, it is updated and returns `True`, otherwise `False`

neighborhood centered in the given better solution is sufficiently large to potentially contain even better solutions, the algorithm changes also the center of the neighborhood for the next exloration[16]. Lastly if the recursion has executed all the possible changes in the given list of differences, or if the list has been emptied by the backtracking, the procedure stops because there are no moves available for exploration.

If none of the terminal conditions are triggered, the procedure removes the first pair of jobs from the list of differences and enters a cycle in which it swaps the positions of job pairs as described in subsection 3.2.3. The recursion is optimal because it allows the heuristic to evaluate every combination for each pair of jobs in first position, unless the time exceeds the limit.

The backtraking is implemented in this algorithm by removing of the first pair of jobs after the call to the recursive method. This ensures that while the calls to the recursion are stacked, the procedure evaluates every combination with one pair of jobs in first position. Once the exploration for the current first pair is completed, the algorithm moves to the next pair and repeats the process, thus systematically exploring all potential solutions.

3.2.6. Exploration of the solutions space

The final interaction in the heuristic involves the exploration of feasible solutions within the neighborhood. The steps described above represents a 'local search' heuristic. However, a significant limitation of local search heuristics is that they typically explore only a narrow set of solutions, which may lead the algorithm to converge on a local minimum rather than a global one, except for some particular cases.

To overcome this issue, many methods that can be implemented to explore the space over the neighborhood. The application of these methods has led to the development of more advanced approaches, such as 'neighborhood search' or 'trajectory methods,' which enhance the efficiency of the heuristic.
Some of the best known approaches are briefly[17] described as follows:

[16] The exploration of the space of solutions is explained in the subsection 3.2.6. However, the procedure that changes the center of the neighborhood begins in the recursive method.
[17] The description has been reported from L. DE GIOVANNI (n.d.) 'Metodi euristici di ottimizzazione combinatoria', *Metodi e Modelli per l'Ottimizzazione Combinatoria, p. 21*

- *random multistart*: consists in computing the local search with various initial solutions, randomly generated or found with simpler heuristics;
- *dinamic change of the neighborhood*: changes the neighborhood under some conditions. On this approach are based some advanced techniques such as 'Variable Neighborhood Descent' or 'Variable Neighborhood Search';
- *randomization of the research strategy*: choosing randomly one of the n improving solutions;
- *Backtracking based on memory*: saves some alternative moves that can be considerated in a following moment.

The neighborhood search in the heuristic proposed is primarily based on the 'random multistart'. Despite this, it dinamically adjusts the center of the neighborhood when an improving solution with a sufficiently large neighborhood is found. Specifically, when the local search is completed, the algorithm moves the center of the neighborhood to the best solution found and randomizes the new starting list of differences.
The following pseudocode outlines the algorithm for the neighborhood search explained:

```
algorithm solutionSpaceExploration do

randomically shuffle bestListGap
while 20% of the pairs in bestListGap is not inverted do
    swap the order of the jobs in a random pair of bestListGap

return bestListGap

bestListGap is the list that contains the pairs of jobs ordered with
the lower mean flow time.
```

This algorithm, with the addition of the dynamic change of the neighborhood center illustrated above in the previous subsection, define the exploration of the solution space in the heuristic proposed.

3.2.7. The complete heuristic

The heuristic presented in this paper integrates the components and interactions defined in the previous subsections. The pseudocode outlined below represents the complete algorithm:

```
algorithm SDSTFlowshopHeuristic is
input: N×M matrix P with process time,
       N×N×M matrix S with setup times.
output: sequence $S$ with the minimal mean flow time
for job i in range (0, N) do
    for job k in range (0, N) do
        if i = k do
            skip
        else
            $gap_{ik} \leftarrow \sum_{r\in\{0,M-1\}} |(P_{r+1,i}+S_{r+1,i,k}) - (P_{rk}+S_{rik})|$
            listGap ← listGap + ($gap_{ik}$ and jobs associated)
for each ((job i, job k), $gap_{ik}$) in dictGap do
    reverseDict ← reverseDict + ($gap_{ik}$, (job i, job k))
sortedReverseDict ← sort reverseDict (by ascending order of $gap_{ik}$)
for each ($gap_{ik}$, (job i, job k)) in sortedReverseDict do
    listGap ← listGap + (job i, job k)
if time of recursion > 60 seconds do
    stop
if lenght of sequence = N and sequence is better than the best
    $S$ ← sequence
    if lenght of listGap > 80% lenght of bestListGap
        bestListGap ← listGap
    stop
else
    if (recursion explored all the N jobs for this iteration or
    listGap is empty) do
        stop
    else
        remove first couple in listGap
        for each gap in listGap do
            listGap ← listGap + first couple in listGap
            execute neighborExploration with the new listGap
            if listGap is not empty
                remove first couple in listGap
```

```
while time limit is not reached
    randomically shuffle bestListGap
    while 20% of the pairs in bestListGap is not inverted do
        swap the order of the jobs in a random pair of bestListGap
    if time of recursion > 60 seconds do
        stop
    if lenght of sequence = N and sequence is better than the best
        $S$ ← sequence
        if lenght of listGap > 80% lenght of bestListGap
            bestListGap ← listGap
        stop
    else
        if (recursion explored all the N jobs for this iteration
        or listGap is empty) do
            stop
        else
            remove first couple in listGap
            for each gap in listGap do
                listGap ← listGap + first couple in listGap
                execute neighborExploration with the new listGap
                if listGap is not empty
                    remove first couple in listGap
    return $S$
```

The variables used for this pseudocode are the same defined in the various blocks discussed in the previous subsections. Hereafter the author proposes a the pseudocode that represents the same entire heuristic integrating the algorithms, this approach allows the reader to gain a better perception of the heuristic:

```
algorithm SDSTFlowshopHeuristic is
input: N×M matrix P with process time,
       N×N×M matrix S with setup times.
output: sequence $S$ with the minimal mean flow time

algorithm calculateGaps
algorithm firstSolution
algorithm neighborExploration
while time limit is not reached
       algorithm solutionSpaceExploration
       algorithm neighborExploration
return $S$
```

4. Comparison and evaluation of the Heuristic

4.1. Description of the tests

The second purpose of this work is to compare the heuristic formulated with the MILP model implemented and discussed in the section 2 of this paper[18].
Both heuristic and TS1 model were implemented and tested on Python version 8.3.2, on an Intel® Core™ i5-5200U CPU at 2.20GHz processor and under the same conditions to ensure consistency in between the results.

In flowshop sequencing research, the standard approach for evaluating a new problem solving technique, whether it is a heuristic or an optimization model, is to generate a set of problems of differing sizes, and then to solve this common set of problems with the new technique and with one other proven technique designed for the same flowshop problem. In general, the quality of a technique's solutions is measured in at least two dimensions. Firstly, how close the solution comes to the optimal solution, if it can be measured. Secondly, how much computer time is required to solve problems of a given size.

The heuristic was tested using the same procedure previously described. The SDST flowshop instances were generated and initially solved by the TS1 model, then after developing the heuristic, the same instances were run through it. The efficiency of the heuristic has been evaluated depending on how close the mean flow time of the solution comes to the TS1 solution, or if it finds a better sequence than the model's one within the same time limit, which is set at five minutes.

For testing, 80 SDST flowshop sequencing instances were generated and solved by the TS1 model. The size of the problems ranged between $N = \{10, 20, 30, 40\}$ and $M = \{2, 5, 10, 20\}$, which were considered a challenge for both the model and the heuristic. The instances were grouped in cells, each with a dimention of five instances for an $N \times M$ problem, so that the evaluation of the mean value registered for every cell of instances of a specific problem was reliable for the comparison.

[18] In the appendix the reader can also find a comparison with the second model of Stafford and Tseng, attached to a brief discussion of the model.

The generator of instances was programmed on Python version 3.8.2 and the psudocode for both generation of the job process and setup times is hereafter illustrated:

```
algorithm generateProcessTimes is
input:    N as the number of jobs
          M as the number of machines
Output:   $M \times N$ matrix $P_{ri}$ with process time of job i on machine r

for r in range (0, M) do
     for i in range (0, N) do
          $P_{ri}$ ← random number between 10 and 99
return $P_{ri}$

algorithm generateSetupTimes is
input:    N as the number of jobs
          M as the number of machines
Output:   $M \times N \times N$ matrix $S_{rik}$ with the setup time of job k preceeded
          by job i on machine r

for r in range (0, M) do
     for i in range (0, N) do
          for k in range (0, N) do
               if i = k do
                    $S_{rik}$ ← 0
               else
                    $S_{rik}$ ← random number between 0 and 10
return $S_{rik}$
```

For the process times the author chose to stick to the values of the experiments conducted by Stafford and Tseng in 2001[19]. However, for the setup times, the values range between 0 and 10. The lower limit indicates no setup time between two jobs on one machine, for example if one job requires the same process of the previous one on the same machine it is correct to assume that between them there is no setup process. The maximum setup time can eventually last the same as the minimum process time, this could be possible in practical case if an inconvenient sequence of jobs is used on the production line.

[19] FAN T. TSENG and EDWARD F. STAFFORD Jr (2001) 'Two MILP models for the N × M SDST flowshop sequencing problem', *International Journal of Production Research*, DOI: 10.1080/00207540010029433, pp. 21-30

The convention used on the setup times generator for the cases where job i and k are the same is to set the setup time as zero. This is a purely conventional choice, because either the model and the heuristic include a constraint that avoid a job to be followed or preceeded by itself.

4.2. Results of the TS1 Model test

After developing the instance generators, the instances were executed on the TS1 model, implemented in Python using the Python-MIP package, following the scheme described in the previous section. The results of this test are illustrated in the Table 1, which can be consulted on Appendix B of this paper for a more detailed overview of the experiment.

The results from Table 1 can also be visualized in a graph, where the horizontal axis represents the number of jobs, and the vertical axis represents the percentage value. The different lines in the plot illustrate the TS1 performance against the sets of machines in the instances blocks.

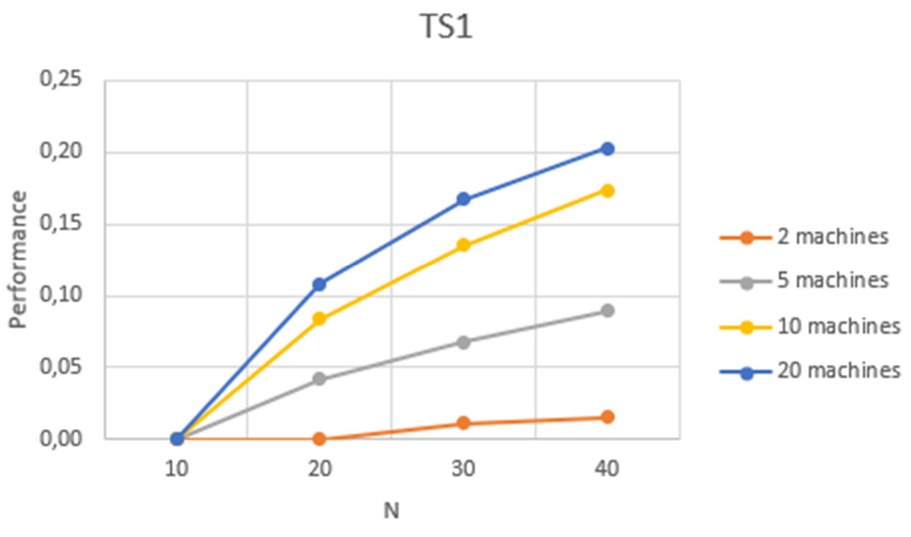

Figure 1. Computation results of the test of the instances on the TS1 Model

As shown in both the Table 1 and the Figure 1, the instance sets with two machines did not challenge the TS1 model. Consequently, the author decided to exclude these instance sets from heuristic testing. The purpose of developing the heuristic is to provide a good solution when techniques that guarantee optimal solutions are too slow. For example, with two machines or ten jobs, the MILP Model TS1 can return optimal solutions in a short time, so the heuristic is not useful for these particular problem sets.

4.3 Results of the SDST Flowshop Heuristic test

The heuristic proposed in section 3 has been tested under the same conditions of the TS1 Model, reported in subsection 4.1. Due to the ability in the resolution of the instances of the MILP model, the evaluation focused on 45 out of the 80 SDST flowshop sequencing instances previously executed on the TS1 Model. These problems were grouped in cells of the following sizes: $N = \{20, 30, 40\}$ and $M = \{5, 10, 20\}$ and each cell contained 5 instances of the problem.
The comparison between TS1 and heuristic are illustrated in the Table 2, which can be consulted on Appendix B of this paper for a more detailed overview of the experiment.

The results from Table 2 can also be visualized in histograms, where the columns represent the mean value of the solutions evaluated by each approach. Additionally, a column shows the difference between the two methods. Each set of machines can be reduced to an histogram and the figures that follows represent the machines sets presented in Table 2.

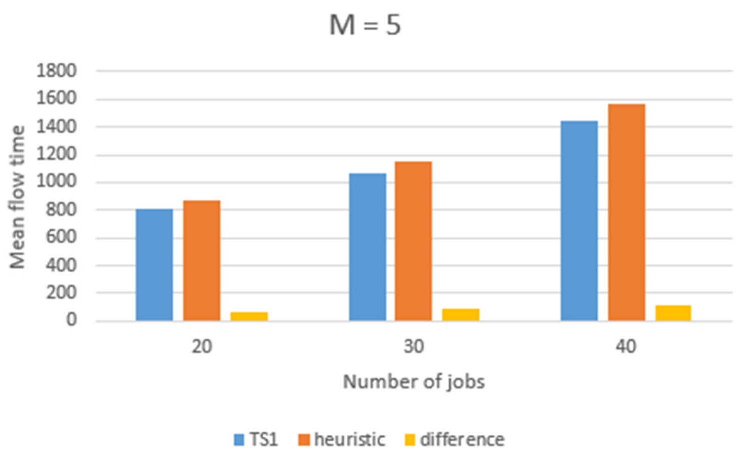

Figure 2. Comparison between the solutions provided by TS1 Model and Heuristic on 5 machines

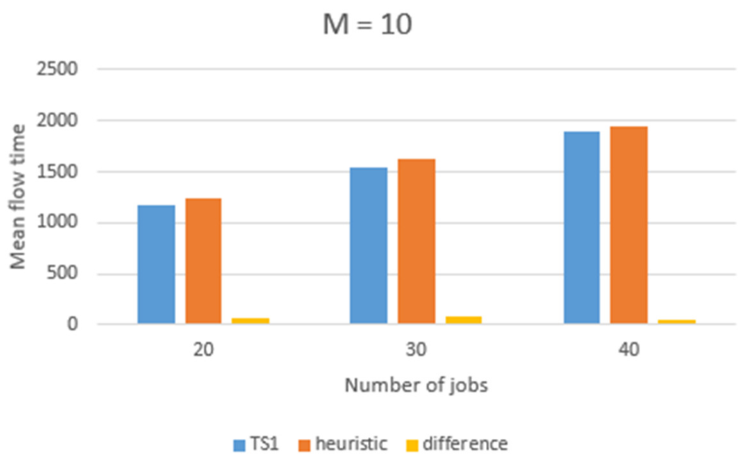

Figure 3. Comparison between the solutions provided by TS1 Model and Heuristic on 10 machines

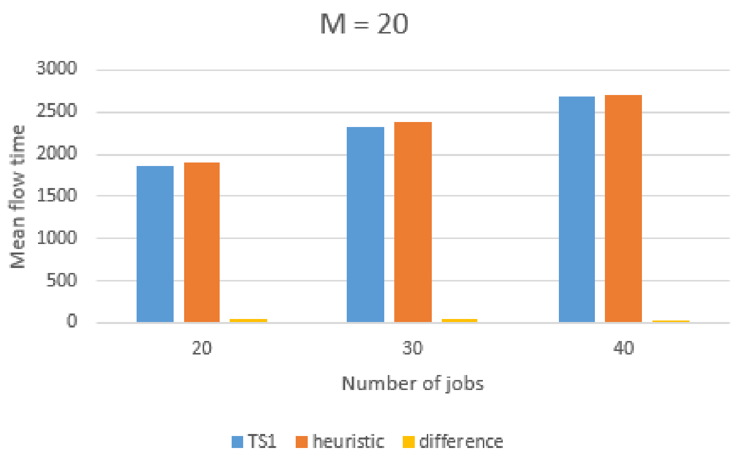

Figure 4. Comparison between the solutions provided by TS1 Model and Heuristic on 20 machines

As shown by the results of the testing, the heuristic proposed cannot overtake the TS1 Model in the solving process of the SDST flowshop sequencing problem. However, the trend observed in the difference between the two approaches suggests that the heuristic could potentially become a valid alternative of the MILP model when the M and N parameters increase.

5. Conclusions and directions for the improvement

5.1. Starting conclusions

The purpose of this work was to present a heuristic for the Sequence Dependent Setup Times (SDST) Flowshop sequencing problem that could compete with a Mixed Integer Linear Program (MILP). Based on the development of the heuristic described above, and on the comparison with the first MILP model introduced by Stafford and Tseng in 2001 (TS1), the following conclusions were drawn.

Although the TS1 has been developed in early 2000s, it is still considerable as a reliable MILP Model for solving SDST flowshop sequencing problems without a multi-criteria objective.
As reported in the results of the comparison, the heuristic proposed by the author challenges the TS1. However, on average, it does not produce better solutions than the TS1 for instances with a decent size of $N \times M$ jobs per machines.
In the following section the author suggests three different approaches that could generate a better heuristic based on the starting idea discussed in section 3.2.

5.2. Directions for the improvement of the Heuristic

During the testing of the heuristic, the author observed that the algorithm evaluated approximately 90-95% of the best solutions proposed within the exploration of the first two neighborhoods. The author considered that this behaviour indicates a great implementation of the recursion procedure developed and a lack of efficiency in the exploration of the neighborhoods space. Consequently, the procedure that controls the randomization and the scanning of new neighborhoods represents a bottleneck in the overall efficiency of the heuristic. As a result, only a small portion of best solutions are found after the first or second exploration of neighborhood in the algorithm.

5.2.1. The 'Tabu Search' alternative

Based on this observation, to enhance the efficiency of the heuristic, a valid solution could potentially be the evaluation of a new method for the neighborhood search. This new procedure could better align the theory behind the 'Trajectory method', where in order to search new neighborhoods, the algorithm builds a trajectory in the space of the solutions.
These methods base their strategy on the acceptance of not improving solutions as transitioning through a worse solution can potentially lead the heuristic to discover an even better solution than the best. In order to avoid inifinite loops, these tecnhiques must implement algorithms based on randomization, which is alredy part of the heuristic, and on the memorization of previously calculated solutions.

The author thought that the implementation of one of these methods such as the 'Tabu Search', where a set of solutions, or moves, are saved in a 'Tabu List', can improve the efficiency of the heuristic.
The algorithm could potentially save the tuples randomly inverted, and in the subsequential iteractions, it would avoid inverting the 'Tabu pairs'. This approach ensures that the algorithm explores new neighborhoods without cycling through previously evaluated solutions.
However, saving the moves for the exploration of the solutions space might result in some solution being discarded from the exploration, even if they have never been evaluated. To solve this issue, it could be implemented an 'aspiration criteria', that in case it is met allows the algorithm to override the 'Tabu rule' and evaluate the solution.
An appropriate 'aspiration criteria' for the heuristic proposed in this work could be that if a pair of jobs appears in a certain number of best solutions evaluated by the algorithm, the tuple will not be included in the 'Tabu List'. This ensures that potentially better solutions are not overlooked in the exploration process.

5.2.2. The genetic heuristic alternative

In recent studies on the SDST flowshop sequencing problem, many researchers used hybrid methods to develop more efficient heuristics, but another type of heuristic that could potentially be useful on the flowshop problems are the population based heuristics.

The author suggests a genetic heuristic in spite of the characteristics of the family of flowshop problems. The algorithm could generate a set of 'parent solutions' based on the gaps between jobs earlier discussed in this paper at section 3.2.
These solutions will then be combined to generate 'offspring solutions' that will inherit the best characteristics from the parents. These characteristics can be visualized as subsequences of jobs with a mean flow time lower compared to other orders. Through this process, the genetic algorithm could gradually form a sequence made by the characteristics of prior solutions and converge to the optimal sequence more efficiently.

The 'parent solutions' do not have to be the sequences with the lowest gaps evaluated. The diversification is a key factor in the genetic heuristics because it allows to have a larger genetic heritage. This increased diversity can improve the probability of the 'offspring solutions' to access better characteristics that could not have been selected without the randomical factor or a more various parent set.

In the genetic procedures another key factor is the 'mutations', which involve the random change of a characteristic in a 'offsping solution'. The mutations are important to avoid the premature convergence of the heuristic, that could happen if the sequences are similar to each other and does not have a way to generate improving 'offspring solutions' for a couple of generations. This random variability ensures the evolution and improvement of solutions, enhancing the overall effectiveness of the genetic algorithm.

5.2.3. The Artificial Intelligence alternative

With the exponential development of the Artificial Intelligence (AI) in the recent history, the AI started to become a powerful tool capable of enhancing various aspects of research.
Its integration into the heuristic development, has improved the efficiency of the processes inside hybrid heuristics.
The author believes that implementing Artificial Intelligence in the SDST flowshop sequencing problem studies can lead to achieve superior results. This potential is mainly due to the nature of the AI to absorb enormous amount of datas and elaborate trends for the problems. This characteristic provides more effective representations and solutions for complex problems.

5.3. Conclusions

In conclusion, The proposed heuristic represents a compromise between efficiency and simplicity of the code. In a five minutes time comparison the heuristic does not produce sequences with a lower objective value than the MILP Model TS1. The main reason for this outcome is the lack of efficiency in the neighborhood search, which could potentially be enhanced with the different approaches discussed.
However, the potential of the local search can be observed in short time comparison with the MILP Model TS1, which the author believes can be overtaken in a comparison with a lower time constraint such as two minutes.

Politecnico di Torino

Appendix A

TS2 Model

After implementing and testing the TS1 model in Python[20], the author decided to run the instances also on the second model, hereafter referred to as the TS2, which was developed by Srikar and Ghosh in order to solve smaller $N \times M$ SDST flowshop sequencing problems more efficiently than the TS1 model. The TS2 Model has been implemented on the same enviroment of the TS1, Python version 3.8.2 using the Python-MIP packages. The following pages present the TS2 model and a brief[21] discussion of its characteristics:

Instance parameters

N number of jobs to be processed	$N \in \mathbb{N}$;
M number of machines (or processing steps);	$M \in \mathbb{N}$;
T_{ri} processing time of job i on machine r;	$T_{ri} \in [M \times N]$;
S_{rik} sequence-dependent set-up time for job k on machine r when job i precedes job k in the sequence.	$S_{rik} \in [M \times N \times N]$

Decision variables

$$Z_{ij} = \begin{cases} 1 & if\ job\ i\ is\ in\ position\ j\ in\ the\ sequence \\ 0 & otherwise \end{cases} \qquad Z_{ij} \in \{0,1\}; \quad (1)$$

$$W_{ijk} = \begin{cases} 1\ if\ job\ i\ is\ in\ position\ j\ in\ the\ sequence\ and\ is \\ \ \ immediately\ followed\ by\ job\ k,\ \ with\ i \neq k \\ 0\ otherwise \end{cases} \qquad W_{ijk} \in \{0,1\}; \quad (2)$$

$$C_{rj} = completion\ time\ of\ the\ job\ in\ position\ j\ of\ the\ sequence\ on\ machine\ r \qquad C_{rj} \in \mathbb{N}; \quad (5)$$

$i \in \{1, \dots, N\};\ j \in \{1, \dots, N\};\ k \in \{1, \dots, N\};\ r \in \{1, \dots, M\}$

[20] The TS1 Model is explained above in the section 2

[21] For a full description of the model the autor suggests the orginal explanation: FAN T. TSENG and EDWARD F. STAFFORD Jr (2001) 'Two MILP models for the N × M SDST flowshop sequencing problem', *International Journal of Production Research*, DOI: 10.1080/00207540010029433, pp. 16-21

Constraint sets

$$\sum_{j=1}^{N} Z_{ij} = 1 \quad \forall i = 1, \dots, N \tag{6}$$

$$\sum_{i=1}^{N} Z_{ij} = 1 \quad \forall j = 1, \dots, N \tag{7}$$

$$\sum_{k=1}^{N} W_{ijk} = Z_{ij} \quad \forall i = 1, \dots, N;\ \forall j = 1, \dots, N \tag{8}$$

$$\sum_{k=1}^{N} W_{k,j-1,i} = Z_{ij} \quad \forall i = 1, \dots, N;\ \forall j = 2, \dots, N \tag{9}$$

$$\sum_{k=1}^{N} W_{kNi} = Z_{i1} \quad \forall i = 1, \dots, N \tag{10}$$

$$C_{rj} + \sum_{i=1}^{N}\sum_{k=1}^{N} S_{rik} W_{ijk} + \sum_{i=1}^{N} T_{ri} Z_{i,j+1} \le C_{r,j+1} \quad \forall r = 1, \dots, M;\ \forall j = 1, \dots, N-1 \tag{11'}$$

$$C_{rj} + \sum_{i=1}^{N} T_{r+1,i} Z_{ij} \le C_{r+1,j} \quad \forall r = 1, \dots, M-1;\ \forall j = 1, \dots, N \tag{11''}$$

$$\sum_{i=1}^{N} T_{ri} Z_{i1} \le C_{r1} \quad \forall r = 1, \dots, M \tag{12'}$$

Objective function

$$minimize\text{: } \sum_{j=1}^{N} C_{Mj} \tag{14}$$

The instance parameters are identical to the ones in TS1 model, so their explanation is treated in the section 2.1.

The X_{rj} and Y_{rj} variables included in the TS1 model, are discarded from the TS2 model. However, in the second model, only three sets of decision variables are included: Z_{ij}, W_{ijk} and C_{rj}. The types of these sets are the same as those in the previous model, and the discussion of these variables has already been provided in the section 2.1.

The constraint sets (6), (7), (8), (9), and (10) are identical to those used in the TS1 model for assigning each job to its sequence position and linking adjacent jobs in the production sequence. Therefore, their purpose and use can be found in Section 2.1.

The equation (11) used in the TS1 model to represent the relation between two consecutive jobs on two adjacent machines is now replaced in this model by the two sets of job completion time inequality constraints (11′) and (11″). The $(N-1)\times M$ constraint set (11′) is related to the completion times of two adjacent jobs in the sequence processed on the same machine. While the $N\times(M-1)$ set derived from the inequality (11″) accounts for the completion times of the same job on two consecutive machines in the production flow line.
Due to their inequality nature the TS2 model supports the items and machines idle times, so the problem is not converted in the "no idle queue" (NIQ) variant of the SDST flowshop sequencing problem. However, if these inequalities were turned in equalities, the problem representation would no longer support idle times no more and it would become an NIQ SDST flowshop sequencing problem. Since the author wanted to ensure consistency in testing the models and the heuristic on the same instances without differences in the problem variant, and solutions, the TS2 model has been implemented with these sets of inequality constraints.
A consequence of (11′) and (11″) bonds is the inability to evaluate the idle times, in fact in the TS1 model the X_{ij} and Y_{ij} decision variables allow the evaluation of the idleness within the sequence. However in this second model the idleness is permitted, but impossible to quantify.
Finally, the TS2 model includes a set of inequality constraints (12′) to control the first job starting time on the M machines. Similar to the constraints in equation (12), the completion time of the first job on the machine r must be at least as great as the completion time of the process on the same machine.

Due to the inability to evaluate idle times for machines or jobs, only three objective functions are compatible with the TS2 model using the instance parameters explained in this paper. To ensure consistency in testing the models and the heuristic, the author decided to minimize the mean flow time, as was done with the TS1 model.

Results of the TS2 Model test

After the implementation on Python following the model presented above, the instances tested on the TS1 and heuristic have been run on the TS2 Model on the same environment as the other tests presented in this paper: Intel® Core™ i5-5200U CPU at 2.20GHz processor and in Python version 3.8.2.

The results of this set of instances run on TS2 Model are proposed in the table 3, which can be consulted at Appendix B, where the reader can find a more detailed overview of the experiment.

The model competes with the TS1 model in the initial groups of instances, however, when the parameters reach the $N = 40$ and $M = 10$ block, the TS2 model fails to find a feasible solution within the five minute time limit, stopping the algorithm during the lower bound search procedure of one out of five instances. However, the author evaluated the mean value of the block between the four instances that have reached a feasible solution for the instance proposed and reported this result in the Table 3.
The author generated three graphs in order to illustrate the difference between the TS1 and TS2 models in the solving performance of the SDST flowshop sequencing instances. These graphs can be seen in the appendix C.

Comparison with the SDST Heuristic

Despite this limitation, the model was also compared to the heuristic. However, the results of the comparison does not include the set of instances with $M = 20$, because the TS2 model was unable solve either N=30 and N=40 problems for that amount of machines.
The comparison with the heuristic has been provided in the two plots named Figure 5 and Figure 6.

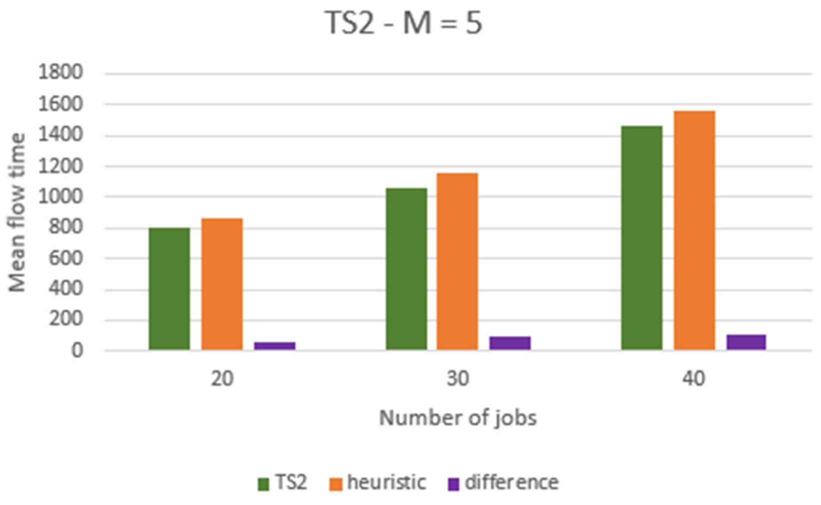

Figure 5. Comparison between the solutions provided by TS2 Model and Heuristic on 5 machines

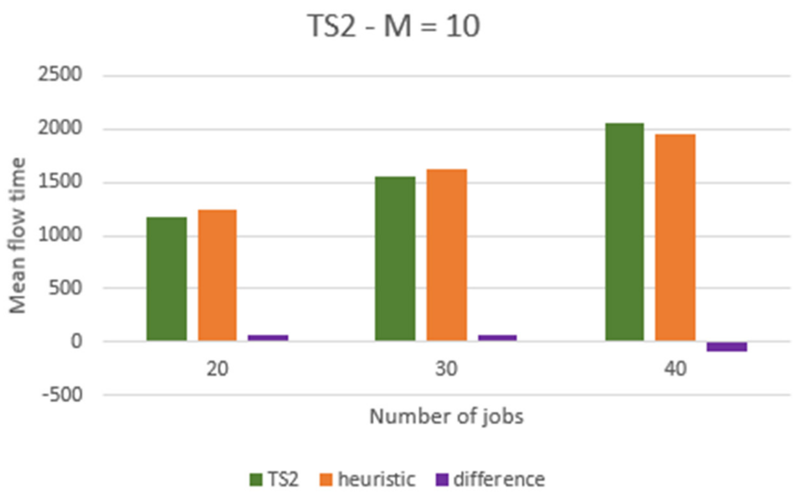

Figure 6. Comparison between the solutions provided by TS2 Model and Heuristic on 10 machines

Similarly to the results shown in Table 3, Figure 6 present the evaluation of the set of instances with $M = 10$ and $N = 40$ as the mean value derived from 4 out of the 5 instances tested on the TS2 model. This adjustment was necessary because of the inability of the model to reach a feasible solution for one instance of the cell.

As illustrated in Figure 5 and Figure 6, while the heuristic competes with the model in the initial blocks of instances and finally overtakes the TS2 model on higher values of the N and M parameters. This outcome highlights the significant potential of the developed heuristic. Unfortunately, the TS2 model could not be compared on the $M = 20$ group of instances due to its inability to solve the problems within the given constraints.

Appendix B

Tables representing the values of the test conducted

The following tables report the results of the experiments conducted by the author on the TS1, TS2 and heuristic developed. All the tests were run on the same processor: Intel® Core™ i5-5200U CPU at 2.20GHz, and in the same environment in terms of code language and processor threads.

Table 1. TS1 model

$N \downarrow$ $M \rightarrow$	2	5	10	20
10	0. 9521634 sec	4.074321 sec	11.48941 sec	22.61587 sec
20	25.05367 sec	4.16852%	8.3917%	10.80064%
30	1.14902%	6.78936%	13.50054%	16.7596%
40	1.5435%	8.91356%	17.3285%	20.29126%

Table 1. Computation results of the test of the instances on the TS1 Model

The values measured in seconds in the table represent the mean time taken by the TS1 model to solve the instances. When the values are in percentage (%), they indicate the gap between the lower bound found by the model and the best solution identified within the time limit.

Table 2. Comparison between TS1 and heuristic

$M \rightarrow$	5			10			20		
$N \rightarrow$	20	30	40	20	30	40	20	30	40
TS1	807.3	1060.4	1444.2	1177.1	1543.5	1901.0	1861.3	2325.4	2687.9
Heuristic	868.3	1154.1	1561.7	1235.3	1625.1	1951.6	1909.9	2374.7	2715.1
Difference	61.0	93.7	117.5	58.2	81.6	50.6	48.59	49.3	27.1

Table 2. Comparison between the solutions provided by TS1 Model and Heuristic

The comparison between the results of the model and the heuristic was based on the mean flow time of the sequences generated within the time limit by both approaches. The values presented in Table 2 represent the mean objective values of the solutions obtained from the five instances executed for each respective $N \times M$ problem cell.

Table 3. TS2 model

$N \downarrow$ $\quad$ $M \rightarrow$	2	5	10
10	1.292083 sec	3.171954 sec	9.148072 sec
20	42.7515 sec	4.6949%	8.01246%
30	1.99352%	6.99644%	14.72938%
40	2.49845%	10.0248%	23.0385%*

Table 3. Computation results of the test of the instances on the TS2 Model
*this result is the mean value between 4 out of 5 instances tested

The values measured in seconds in the table represent the mean time taken by the TS2 model to solve the instances. Meanwhile the values in percentage (%) represent the gap between the lower bound found by the model and the best solution identified within the time limit.

The tests conducted on this model underscore the purpose of its development by Stafford and Tseng, which was to ensure efficiency at lower values of the N and M parameters. The model competes with the TS1 model in the initial groups of instances, however, when the parameters reach the $N = 40$ and $M = 10$ block, the TS2 model fails to find a feasible solution within the five minute time limit, stopping the algorithm during the lower bound search procedure of one out of five instances. However, the author evaluated the mean value of the block between the four instances that have reached a feasible solution for the instance proposed and reported this result in the Table 3.

Appendix C

Comparison between TS1 and TS2 models

The following three graphs in Figure 7, 8 and 9 represent the comparison between the efficiency of TS1 and TS2 models in solving SDST flowshop sequencing problems with different values of N and M parameters.
Each figure compares the two MILP models on an equal set of machines and with the same instances run onto the Intel® Core™ i5-5200U CPU at 2.20GHz processor.

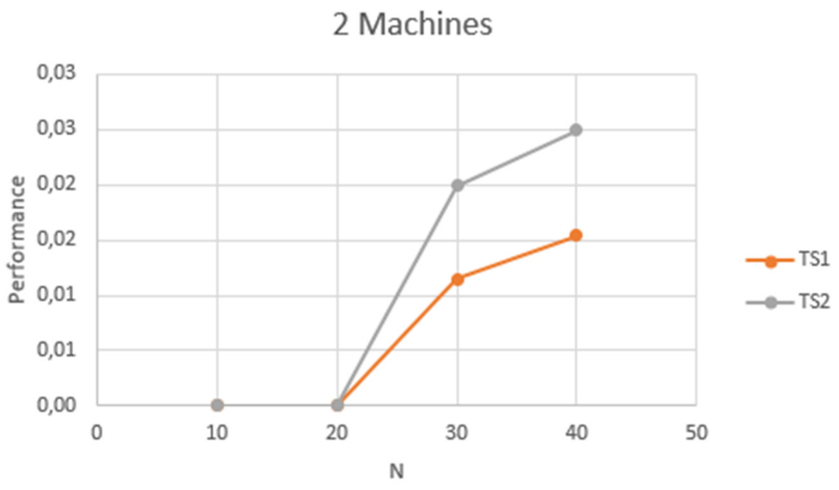

Figure 7. Comparison between the solutions provided by TS1 Model and TS2 Model on 2 machines

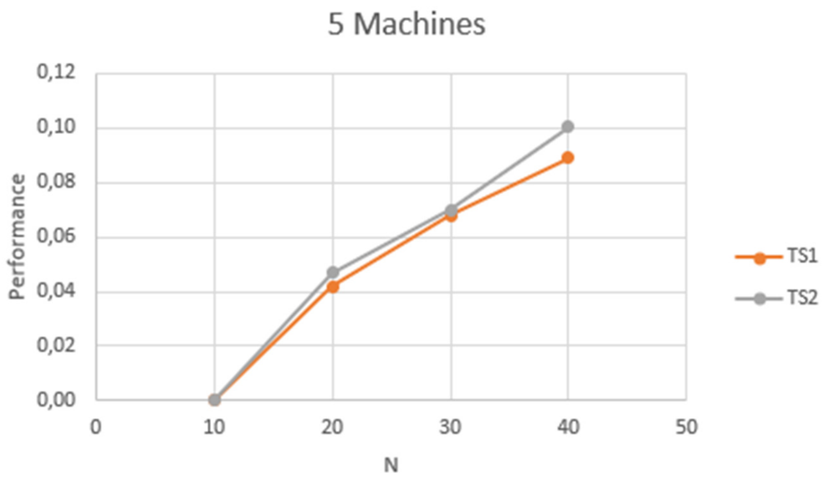

Figure 8. Comparison between the solutions provided by TS1 Model and TS2 Model on 5 machines

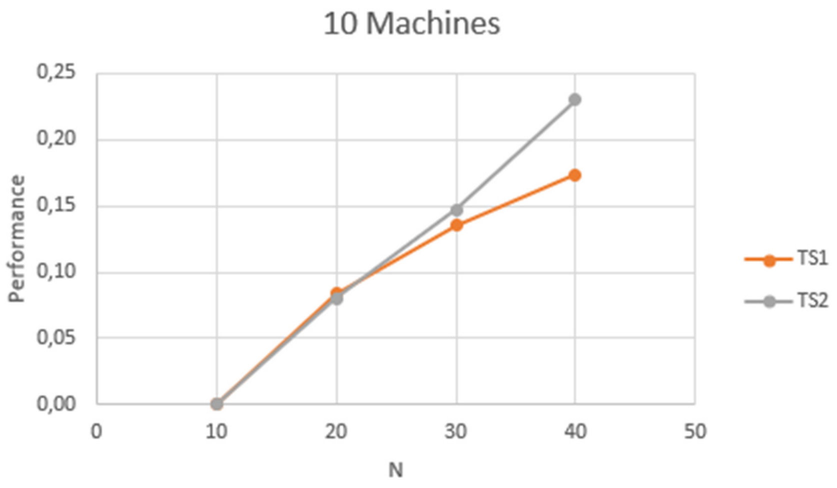

Figure 9. Comparison between the solutions provided by TS1 Model and TS2 Model on 10 machines

The comparison presented underscore the inefficiency of the TS2 model at the increasing value of N, where the model cannot compete with the first one and the gap between the lower bound found by the model and the best feasible solution explored exponentially increases.

Similarly to the results shown in Table 3, in Figure 9 the value of the set of instances with $M = 10$ and $N = 40$ represents the mean value derived from 4 out of the 5 instances tested on the TS2 model. This adjustment was necessary because of the inability of the model to reach a feasible solution for one instance of the cell.

References

For clarity, all references cited in this paper are listed below in the order of their appearance.

GUPTA, S. K. (1982) ‘n jobs and m machines job-shop problems with sequence-dependent set-up times’, *International Journal of Production Research*, DOI: 10.1080/00207548208947793

BELLUR N. SRIKAR & SOUMEN GHOSH (1986) ‘A MILP model for the n-job, M-stage flowshop with sequence dependent set-up times’, *International Journal of Production Research*, DOI: 10.1080/00207548608919815

Jr, E. F. STAFFORD and TSENG, F. T. (1990) ‘On the Srikar-Ghosh MILP model for the iVx M SDST flowshop problem’, *International Journal of Production Research*, DOI: 10.1080/00207549008942836

RIOS-MERCADO, R. Z., and BARD, J. F. (1998) ‘Computational experience with a branch-andcut algorithm for flowshop scheduling with setups’, *Computers and Operations Research,* DOI: 10.1016/S0305-0548(97)00079-8

RIOS-MERCADO, R. Z., and BARD, J. F. (1998) ‘Heuristics for the flow line problem with setup costs’, *European Journal of Operational Research*, DOI: 10.1016/S0377-2217(97)00213-0

FAN T. TSENG and EDWARD F. STAFFORD Jr (2001) ‘Two MILP models for the N × M SDST flowshop sequencing problem’, *International Journal of Production Research*, DOI: 10.1080/00207540010029433

KAI ZHOU GAO, PONNUTHURAI NAGARATNAM SUGANTHAN, QUAN KE PAN, TAY JIN CHUA, CHIN SOON CHONG, TIAN XIANG CAI (2016) ‘An improved artificial bee colony algorithm for flexible job-shop scheduling problem with fuzzy processing time’, *Expert Systems with Applications*, DOI: 10.1016/j.eswa.2016.07.046

Python-MIP packages and documentation, https://www.python-mip.com/

L. DE GIOVANNI (n.d.) ‘Metodi euristici di ottimizzazione combinatoria’, Metodi e Modelli per l'Ottimizzazione Combinatoria