# Preemptive Flowshop Scheduling Problem

Adimi Alaa Dania, Irmouli Maissa, Benazzough Nourelhouda, Rezkellah FatmaZohra, Hamzaoui Imane, Hamitidouche Thanina, and Bessedik Malika

*Higher National School of Computer Science ESI Algiers, Algeria*

April 9, 2024

**Abstract**

The scheduling of jobs in flowshop environments has become increasingly important in various industries. As the number of machines and jobs to be scheduled increases, the complexity of the problem also increases, which necessitates the need for efficient scheduling techniques. In this report, we will first introduce the flowshop problem then go through various methods for solving it, including exact methods, heuristics, metaheuristics, and hyperheuristics. We will compare the results obtained from each method and study their performance under different input parameters. Specifically, we will examine the makespan, which is an essential parameter for measuring the efficiency of scheduling. This report article aims to provide a comprehensive analysis of the different techniques used to solve the flowshop scheduling problem. This report is primarily aimed at individuals who are new to the concept of flowshop scheduling and are seeking to deepen their understanding of this problem domain.

## 1 Introduction

The flow-shop scheduling problem stands as one of the most critical challenges in industrial operations. Its resolution directly impacts production efficiency, resource allocation, and ultimately, the bottom line of businesses.

In this report, we will first go through the relevant background to understand the fundamentals related to the flowshop problem. Then, in later sections, we will explore various algorithms to address this problem and discuss the resultant findings acquired through their application.

## 2 Background

### 2.1 Scheduling problems

Scheduling is a crucial decision-making process used in a variety of manufacturing and service industries. It aims to optimize one or more objectives by allocating resources to tasks over specific periods. Resources and tasks can take various forms, such as machines in a workshop, runways at an airport, crew at a construction site, processing units in a computing environment, etc. Tasks may have specific characteristics, such as priority levels, earliest starting times, and due dates. The objectives can also vary, such as minimizing tasks completed after their respective due dates.

### 2.2 Classification of scheduling problem

Scheduling problems can be broadly categorized into three types: **Single Machine Scheduling, Flow Shop Scheduling, and Job Shop Scheduling**.

- **Single Machine Scheduling** - In this type of scheduling, jobs are arranged in a specific order on a given machine. To minimize the makespan, which the total time needed to complete all tasks, jobs with shorter in-process times are typically prioritized over those with longer in-process times.

- **Flowshop Scheduling** - The basic principle of this type of scheduling is that each machine takes the jobs in a particular sequence, which means each job has to get processed in every machine available in the flow shop environment. Also in each machine the in-process time for every job is distinct. Therefore while evaluating a flow shop problem, we carry out different possible sequences of carrying out the job, and then the best among those is chosen.
- **Jobshop Scheduling** - This differs from Flowshop Scheduling in that all jobs don't need to be processed on every machine available. Each job may be processed on a distinct number of machines, and the sequence of jobs processed on each machine is also unique. The best sequence is selected based on the requirements of the problem, after considering several possible sequences.

## 2.3 Parameters of a Scheduling Problem

The following parameters play a crucial role in optimizing a scheduling problem: Based on the following parameters we can achieve our objective of optimally scheduling a task:

1. Total completion time
2. Makespan
3. Lateness
4. Total flow time
5. Earliness
6. Number of jobs corresponding to number of machines
7. Due dates

## 2.4 Flowshop Problem

### 2.4.1 Problem Definition

Flowshop scheduling is a problem where a set of **n** jobs, each with **m** operations, must be processed on **m** machines. Each operation of a job must be executed on a specific machine, and no machine can perform more than one operation simultaneously. The objective is to determine the optimal arrangement of jobs that will result in the shortest possible total job execution time, also known as makespan. This is typically achieved by finding the optimal order in which to process the jobs on the machines.

The figure below taken from the paper "Promoting green internet computing throughout simulation-optimization scheduling algorithms" illustrates demonstrates the execution of three jobs across three machines within a flowshop setting.

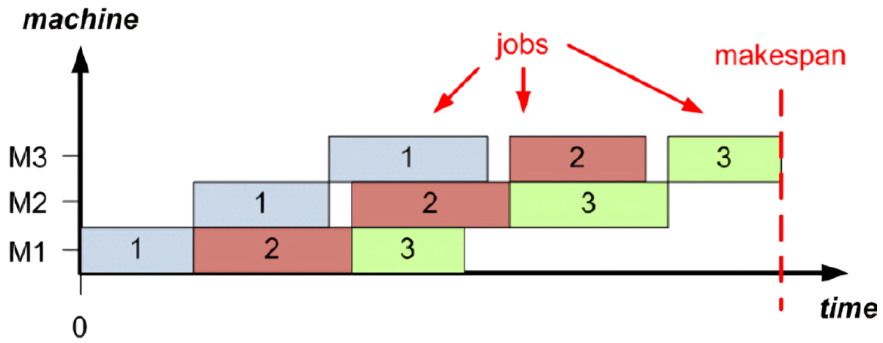


Figure 1: A simple flow-shop scheduling problem with 3 jobs and 3 machines

### 2.4.2 Complexity of Flowshop Scheduling

Flow-shop scheduling is a well-known combinatorial optimization problem that has been studied extensively in operations research and computer science. Its complexity is widely recognized, and it is classified as an ***NP-hard problem***, which means that finding an optimal solution for larger instances of the problem is impractical. The complexity arises from the large number of possible permutations of jobs and machines that need to be evaluated to find an optimal solution.

### 2.4.3 Makespan calculation

The makespan of a sequence in a flowshop scheduling problem is the time required to complete all jobs on all machines in the sequence. It can be calculated by starting with the first job on the first machine and adding the time required to process each job on each machine, while ensuring that the machines are not idle and the jobs are processed in the specified order.

Below is a Python function designed to compute the makespan for a provided sequence of tasks:

```
def evaluate_sequence(sequence, processing_times):
    _, num_machines = processing_times.shape
    num_jobs = len(sequence)
    completion_times = np.zeros((num_jobs, num_machines))

    # Calculate the completion times for the first machine
    completion_times[0][0] = processing_times[sequence[0]][0]
    for i in range(1, num_jobs):
        completion_times[i][0] = completion_times[i-1][0] + processing_times[sequence[
    i]][0]

    # Calculate the completion times for the remaining machines
    for j in range(1, num_machines):
        completion_times[0][j] = completion_times[0][j-1] + processing_times[sequence
    [0]][j]
        for i in range(1, num_jobs):
            completion_times[i][j] = max(completion_times[i-1][j], completion_times[i
    ][j-1]) + processing_times[sequence[i]][j]

    # Return the total completion time, which is the completion time of the last job
    in the last machine
    return completion_times[num_jobs-1][num_machines-1]
```

Listing 1: Code implementation of a function that calculates the makespan of a sequence.

### 2.4.4 Real World applications

Flowshop scheduling has many real-world applications in various industries such as manufacturing, transportation, logistics, and healthcare. In manufacturing, flowshop scheduling can be used to optimize production processes in industries such as food processing, automotive manufacturing, and electronics manufacturing. In transportation, flowshop scheduling can be used to optimize the use of resources such as vehicles and drivers for efficient delivery routes. In logistics, flowshop scheduling can be used to optimize warehouse operations and distribution processes. In healthcare, flowshop scheduling can be used to optimize patient flow and resource utilization in hospitals and clinics. Overall, flowshop scheduling can help organizations improve productivity, reduce costs, and enhance customer satisfaction by efficiently utilizing their resources.

# 3 Exact Resolution Methods for FSP

## 3.1 Overview

Exact methods are algorithms used to find optimal solutions to optimization problems, where the goal is to minimize or maximize a certain objective function subject to a set of constraints. In many real-world applications, such as logistics, scheduling, finance, and engineering, it is critical to find the best possible solution to these problems. Exact methods provide a way to obtain optimal solutions with a guaranteed level of optimality, but they can be computationally expensive, especially for large-scale

problems. One powerful exact method that has been widely used in solving a variety of optimization problems is Branch and Bound. Branch and Bound is a tree-based search algorithm that divides the search space into smaller subproblems, evaluates the subproblems, and prunes those that cannot lead to better solutions than the current best-known solution. Branch and Bound has proven to be a flexible and effective method for solving optimization problems, especially for mixed-integer programming problems, where some variables are restricted to take integer values.

The power of Branch and Bound lies in its ability to systematically eliminate large portions of the search space that are guaranteed to contain suboptimal solutions. This is achieved through the use of bounding techniques that enable the algorithm to quickly prune subproblems that cannot lead to better solutions than the current best-known solution.

One of the trickiest aspects of Branch and Bound is finding the lower bound for the problem, which is used to prune subproblems that cannot lead to a better solution. The lower bound is a value that provides an estimate of the minimum possible value of the objective function for a given subproblem. The lower bound is used to prune branches of the search tree that cannot lead to a better solution than the current best-known solution, thereby reducing the number of subproblems that need to be explored.

## 3.2 Branch and Bound for FSP

The Branch and Bound approach to solving flowshop problems involves building a search tree, where each node represents a partial schedule of the jobs on the machines.

Below is the node structure in our implementation:

```
# Define the Node structure of the search tree that we will be using
class Node:
    def __init__(self, jobs, remaining_jobs, parent=None, lower_bound=1e100):
        self.jobs = jobs
        self.remaining_jobs = remaining_jobs
        self.parent = parent
        self.lower_bound = lower_bound

    def __str__(self):
        return f"Node(jobs={self.jobs}, remaining_jobs={self.remaining_jobs},
    lower_bound={self.lower_bound})"
```

Listing 2: Code implementation of the node structure of the search tree.

The algorithm starts with an initial node, which represents the empty schedule, and generates child nodes by adding one job at a time to the schedule. The lower bound is then used to prune branches of the search tree that cannot lead to a better solution.

As the search tree is built, the Branch and Bound algorithm prunes branches that cannot possibly lead to an optimal solution, by using the lower bound to eliminate nodes that have a makespan greater than the current best solution. This allows the algorithm to focus on the most promising branches of the search tree, and ultimately find the optimal solution.

### 3.2.1 Implementation of Branch and Bound

In our implementation of the Branch and Bound algorithm, we incorporated a lower bound to estimate the minimum possible makespan for a given subproblem. This lower bound was obtained by partially scheduling some of the jobs and computing the completion time of the partial schedule. By incorporating this lower bound into our Branch and Bound algorithm, we were able to efficiently explore the solution space and find the optimal solution to the scheduling problem.

Below is the implementation of the Branch and Bound algorithm:

```
def branch_and_bound(processing_times, initial_solution, initial_cost):
    jobs, machines = processing_times.shape
    # Initialize the nodes list to the `root_node`
    root_node = Node([], set(range(jobs)))
    best_solution = initial_solution.copy()
    best_solution_cost = initial_cost
```

```
    nodes = [root_node]
    i = 1
    while nodes:
        node = nodes.pop()
        # Explore neighbors of the node ‘node‘
        for a job in node.remaining_jobs:
            child_jobs = node.jobs + [job]
            child_remaining_jobs = node.remaining_jobs - {job}
            child_lower_bound = evaluate_sequence(child_jobs, processing_times)
            # If the child node is a leaf node (i.e., all jobs have been assigned)
    then calculate its cost
            if not child_remaining_jobs:
                if child_lower_bound < best_solution_cost:
                    best_solution = child_jobs
                    best_solution_cost = child_lower_bound
                    continue
            # If the child node is not a leaf then calculate its lower bound and
    compare it with the current ‘best_solution_cost‘
            if child_lower_bound < best_solution_cost:
                child_node = Node(child_jobs, child_remaining_jobs, parent=node,
    lower_bound=child_lower_bound)
                nodes.append(child_node)
        i += 1
    return best_solution, best_solution_cost, i
```

Listing 3: Code implementation of Branch and Bound in Python.

### 3.2.2 Branch and Bound Pure

In this study, we implemented Branch and Bound pure which is a method that systematically explores all possible solutions by generating and examining every potential solution, without using any optimization techniques or heuristics to reduce the search space.

Below is the code implementation of Branch and Bound Pure:

```
def branch_and_bound_pure(processing_times,initial_solution, initial_cost):
    jobs, machines = processing_times.shape
    # Initialize the nodes list to the ‘root_node‘
    root_node = Node([], set(range(jobs)))
    best_solution = initial_solution.copy()
    best_solution_cost = initial_cost
    nodes = [root_node]
    i = 1
    while nodes:
        node = nodes.pop()
        # Explore neighbors of the node ‘node‘
        for a job in node.remaining_jobs:
            child_jobs = node.jobs + [job]
            child_remaining_jobs = node.remaining_jobs - {job}
            # If the child node is a leaf node (i.e., all jobs have been assigned)
    then calculate its cost
            if not child_remaining_jobs:
                child_lower_bound = evaluate_sequence(child_jobs, processing_times)
                if child_lower_bound < best_solution_cost:
                    best_solution = child_jobs
                    best_solution_cost = child_lower_bound
                    continue
            else:
                # If the child node is not a leaf then calculate its lower bound and
    compare it with current ‘best_solution_cost‘
                child_lower_bound = evaluate_sequence(child_jobs, processing_times)
                child_node = Node(child_jobs, child_remaining_jobs, parent=node,
    lower_bound=child_lower_bound)
                nodes.append(child_node)
        i += 1
    return best_solution, best_solution_cost, i
```

Listing 4: Code implementation of Branch and Bound Pure in Python.

We implemented this approach to compare its performance to Branch and Bound with lower bound. By comparing the two methods, we can evaluate the effectiveness of using lower bounds in reducing the search space and improving the efficiency of the search algorithm.

## 3.3 Tests

### 3.3.1 Random instance

We experimented to compare the performance of Branch and Bound pure, Branch and Bound with lower bound, and Brute Force algorithms on a randomly generated instance of 8 jobs and 5 machines. The processing times for each job on each machine were also randomly generated. We ran each algorithm and recorded the results. Here are the results:

```
Results of Branch & Bound:
Best sequence is [7, 5, 4, 1, 0, 2, 6, 3] with a makespan of 384.0.
No. Nodes visited is 26817.
Elapsed time of 8.128053426742554 seconds.
```

Listing 5: Results of Branch and Bound on an instance of 8 jobs and 5 machines.

```
Results of Branch & Bound pure:
Best sequence is [7, 5, 4, 1, 0, 2, 6, 3] with a makespan of 384.0.
No. Nodes visited is 69282.
Elapsed time of 15.603382110595703 seconds.
```

Listing 6: Results of Branch and Bound pure on an instance of 8 jobs and 5 machines.

```
Results of Brute Force:
Best sequence is [5, 4, 7, 1, 0, 2, 6, 3] with a makespan of 384.0.
No. of tested solutions 40320.
Elapsed time of 6.222885847091675 seconds.
```

Listing 7: Results of Brute force on an instance of 8 jobs and 5 machines.

We observed that Branch and Bound performed better than Branch and Bound pure. Specifically, Branch and Bound visited significantly fewer nodes than Branch and Bound pure. This result indicates that the lower bound used in Branch and Bound was effective in pruning branches of the search tree which could not lead to a better solution.

The implementation of Branch and Bound required slightly more time than Brute force because of the computational cost of calculating the lower bound. Although the lower bound provides a way to prune branches of the search tree, the complexity of its calculation may offset some of the time savings.

### 3.3.2 Random instance with NEH Initialization

If we initialize the solution using a heuristic instead of a random initial solution, the Branch and Bound algorithm is likely to take less time to converge to the optimal solution. This is because the heuristic can provide a good starting point, reducing the size of the search space and the number of nodes that need to be visited. In the above experiment, we used a random initial solution.

```
Results of Branch & Bound (with NEH Initialization):
Best sequence is [5, 7, 4, 1, 0, 2, 6, 3] with a makespan of 384.0.
No. Nodes visited is 26444.
Elapsed time of 7.4751317501068115 seconds.
```

Listing 8: Results of Branch and Bound with NEH initialization on an instance of 8 jobs and 5 machines.

After initializing the solution with the NEH heuristic, we observed a significant improvement in the performance of both Branch and Bound and Branch and Bound Pure. The number of nodes visited decreased even further, and the running time of the algorithms was reduced. This shows that using a good initial solution can greatly improve the performance of the Branch and Bound algorithm.

### 3.3.3 Common instance

A random common was provided for us to test our branch and bound algorithm on, and we were also able to compare our results with other teams in the class. The instance consisted of 10 jobs and 5 machines, and we ran our algorithm multiple times to ensure accuracy. The results are summarized below.

```
Results of Branch & Bound:
Best sequence is [3, 2, 7, 6, 8, 0, 1, 4, 5, 9] with a makespan of 1102.0.
No. Nodes visited is 6235278.
Elapsed time of 1796.516449213028 seconds.
```

Listing 9: Results of Branch and Bound on an instance of 10 jobs and 5 machines.

### 3.3.4 Taillard instance

We tested our branch and bound algorithm on the first instance of the first Taillard benchmark (20 jobs and 5 machines) for 30mn then we stopped it. The results are summarized below.

```
Results of Branch & Bound:
Best sequence is [18, 2, 10, 16, 14, 15, 7, 13, 0, 3, 6, 4, 5, 9, 17, 1, 19, 11, 12]
    with a makespan of 1459.0.
No. Nodes visited is 874390.
Elapsed time of 1800.00000001 seconds.
```

Listing 10: Results of Branch and Bound on the first instance of the 20 jobs and 5 machines Taillard benchmark.

```
Results of Branch & Bound (With NEH initialization):
Best sequence is [8, 6, 15, 10, 7, 1, 16, 2, 14, 13, 17, 3, 9, 11, 0, 18, 5, 4, 12,
    19] with a makespan of 1334.0.
No. Nodes visited is 1476390.
Elapsed time of 1800.0000002 seconds.
```

Listing 11: Results of Branch and Bound with NEH initialization on the first instance of the 20 jobs and 5 machines Taillard benchmark.

## 3.4 Discussion And Analysis

After implementing the Branch & Bound algorithm and conducting several tests, we have found that this algorithm is significantly more efficient than Brute Force algorithms. Branch & Bound has the potential to yield highly promising results, especially when an initial solution is already relatively good. This is because such solutions can lead to more efficient pruning, resulting in reduced execution time.

Another advantage of Branch & Bound algorithms is that they can be parallelized. By leveraging high-performance computing capabilities, we can allocate the search space among multiple workers. This approach can lead to significantly faster attainment of the optimal solution.

Branch & Bound algorithms are particularly effective when the search space is not excessively large. However, when the search space becomes too large, other methods can be employed. We will explore these methods in the upcoming sections.

# 4 Heuristics for FSP resolution

## 4.1 Overview

Exact methods, such as branch and bound and brute force, guarantee optimality, but they can be computationally expensive and may require a significant amount of time to find an optimal solution, especially for large problem instances. Therefore, for many practical problems, exact methods are not always feasible. Heuristics, on the other hand, are approximate algorithms that provide good-quality solutions in a reasonable amount of time. They are designed to quickly generate feasible solutions that are close to optimal, without the guarantee of finding the global optimum. The flowshop problem has been widely studied in the literature and various heuristics have been proposed for solving it. Below are some of the most known heuristics for solving the problem.

## 4.2 Heuristics for FSP

1. **NEH Heuristic** NEH (Nawaz, Enscore, Ham) is a well-known insertion-based heuristic algorithm for the flowshop problem. The algorithm starts by calculating the total processing time for each job and then sorting the jobs in decreasing order of processing times. It then takes the first two jobs and selects the combination with the minimum makespan. The algorithm then inserts the third job and generates all possible combinations of the first three jobs, calculates the makespan for each combination, and selects the combination with the minimum makespan. This process continues until all jobs have been inserted.NEH (Nawaz, Enscore, Ham) is a well-known insertion-based heuristic algorithm for the flowshop problem. The algorithm starts by calculating the total processing time for each job and then sorting the jobs in decreasing order of processing times. It then takes the first two jobs and selects the combination with the minimum makespan. The algorithm then inserts the third job and generates all possible combinations of the first three jobs, calculates the makespan for each combination, and selects the combination with the minimum makespan. This process continues until all jobs have been inserted.

2. **Greedy NEH** The Greedy NEH algorithm is a constructive heuristic approach used to solve permutation flow shop scheduling problems. At each step, instead of selecting the best partial solution as dictated by the original NEH method, a modification is introduced: rather than choosing the absolute best partial solution, the algorithm randomly selects one out of the best five partial solutions with equal probabilities and keeps it for the next steps of the solution construction. When we have a complete solution, we reiterate the construction process, having at each iteration a high-quality individual that is different from its predecessor. We keep track of the best-found solution, and at the end of the execution, it's the one Greedy NEH returns. While it does not guarantee an optimal solution, the Greedy NEH algorithm provides a quick and effective method for solving permutation flowshop scheduling problems.

3. **Johnson's heuristic** Johnson's algorithm is a simple and effective method used to find the optimal sequence for a two-machine flowshop problem, and can also be applied to problems with more than two machines under certain conditions. The algorithm assigns processing times for each job to both machines and selects the job with the smallest processing time from the list. If the minimum processing time is on the first machine, the job is placed at the beginning of the sequence, and if it is on the second machine, it is placed at the end of the sequence. The selected job is then removed from the list, and the process is repeated until all jobs are sequenced. It ensures that the job with the smallest processing time is scheduled first, reducing idle time and maximizing machine utilization.

4. **Ham heuristic** Ham heuristic consists of dividing the execution time matrix into two sub-matrices, and based on that, it will generate two solutions, of which the best is retained. Each sub-matrix will contain $m/2$ columns, where $m$ is the number of machines, such that the first sub-matrix will contain the first $m/2$ machines, such that the first sub-matrix will contain the first machines and the second sub-matrix will contain the last $m/2$ machines. For each sub-matrix, the sum of execution times for each job $i$ is calculated, and we will have these two sums:

$$P_{i1} = \sum_{j=1}^{m/2} t_{ij}, \tag{1}$$

$$P_{i2} = \sum_{j=m/2+1}^{m} t_{ij}, \tag{2}$$

   Then we calculate $P_{i2} - P_{i1}$ for each job $i$.

   - The first solution is the list of jobs ordered in decreasing order according to $P_{i2} - P_{i1}$.
   - The second solution consists of reordering the list of jobs in increasing order according to $P_{i1}$ for jobs that have $P_{i2} - P_{i1} > 0$ and in decreasing order according to $P_{i2}$for jobs that have $P_{i2} - P_{i1} < 0$. The resulting solution will be the concatenation of these two lists.

5. **Palmer's heuristic** The Palmer heuristic involves assigning weights to each machine and then calculating the weighted sum of each job. To start the process, the problem of scheduling $n$ jobs on $m$ machines with a processing time matrix noted $t$ is considered. The weight of each job is evaluated is evaluated using the formula below:

$$f(i) = \sum_{j=1}^{m} (m - 2j + 1)t_{ij} \tag{3}$$

   The jobs are then sorted in ascending order of their weights$f(i)$, and a sequence is formed based on this sorting.

6. **CDS heuristic** The CDS (Constructive Dispatching Sequence) heuristic is a constructive approach. It works by constructing a sequence of jobs that is expected to have a relatively low makespan. The first step of the CDS heuristic is to select the job that has the smallest processing time on the first machine. Then, for each subsequent machine, the CDS heuristic selects the job that has the shortest processing time among the remaining jobs. This process continues until all jobs have been assigned to a machine. The resulting sequence of jobs is then evaluated to determine the makespan.

7. **Gupta heuristic (1971)** Gupta's is another heuristic that involves assigning weights to each job using the following formula:

$$f(i) = \frac{\text{sign}(t(i_0)) - t(i(m-1)))}{\text{min-gupta}(i,t)} \tag{4}$$

   The jobs are then sorted in ascending order of their weights, and a sequence is formed based on this sorting.

8. **PRSKE heuristic** PRSKE is another heuristic that orders job assigning weights to each job using the following formula:

$$AVG_i + STD_i + |SKE_i| \tag{5}$$

   Where $AVG_i$ is the average processing times of job $i$,$STD_i$ stands for the standard deviation of job $i$, and $SKE_i$ as the absolute value of the skewness of job $i$.

9. **Artificial heuristic** This heuristic algorithm works by iteratively reducing the number of machines considered in the problem until we are left with a single machine. At each iteration, we compute a weight matrix that assigns a weight to each operation based on its position in the processing sequence. We then use Johnson's algorithm, which is a well-known heuristic for the flow shop scheduling problem, to compute a feasible sequence of operations for the reduced problem. We evaluate the quality of the sequence using a cost function that computes the completion time of the jobs on the reduced set of machines. We repeat the process with the reduced problem until we are left with a single machine. In the end, we return the best solution found.

### 4.3 Tests

We conducted tests for each implemented heuristic on the first and seventh instances of the following Taillard benchmarks:

1. 20 jobs and 5 machines.
2. 50 jobs and 10 machines.
3. 100 jobs and 10 machines.

The quality of the results for each heuristic was evaluated using the deviation metric, which measures the difference between the makespan of the heuristic solution and the best known solution for a given instance. It is represented as:

$$\text{quality} = \frac{C_{\max} - \text{Upper Bound}}{\text{Upper Bound}} \tag{6}$$

### 4.3.1 First instance

The table provided below presents a summary of the test results for the initial instance of each benchmark. The test results offer a comprehensive overview of the performance and outcomes obtained from evaluating the algorithms on these specific instances.

| **Heuristic** | **20-5 (Deviation)** | **20-5 (Time)** | **50-10 (Deviation)** | **50-10 (Time)** | **100-10 (Deviation)** | **100-10 (Time)** |
|---|---|---|---|---|---|---|
| NEH | 4.382 | 0.030s | 6.744 | 0.730s | 5.061 | 6.391s |
| Greedy NEH | 3.365 | 0.332s | 7.405 | 10.708s | 2.149 | 157.971s |
| Ham | 10.876 | 0.000s | 20.562 | 0.003s | 8.406 | 0.004s |
| Gupta | 9.233 | 0.000s | 19.901 | 0.003s | 12.634 | 0.012s |
| Palmer | 8.249 | 0.000s | 12.893 | 0.006s | 4.471 | 0.007s |
| CDS | 8.746 | 0.004s | 13.091 | 0.021s | 7.608 | 0.046s |
| PRSKE | 24.648 | 0.000s | 28.926 | 0.006s | 22.929 | 0.011s |
| Artificial Heuristic | 6.964 | 0.000s | 14.413 | 0.025s | 7.574 | 0.068s |

Table 1: Performance comparison on the first instances from 3 Taillard benchmarks for different heuristics.

Here's a graphical depiction summarizing the results obtained above:

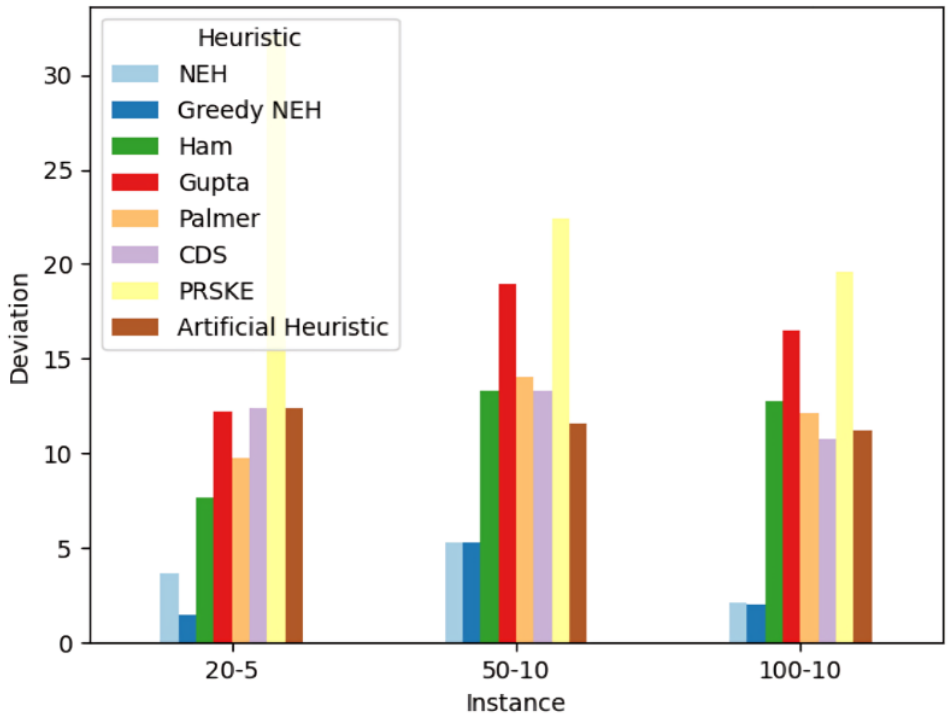


Figure 2: Makespan comparison for different first instances from 3 Taillard benchmarks using different heuristics.

### 4.3.2 Seventh instance

The table provided below presents a summary of the test results for the seventh instance of each benchmark. The test results offer a comprehensive overview of the performance and outcomes obtained from evaluating the algorithms in these specific instances.

Here's a graphical depiction summarizing the results obtained above:

| Heuristic | 20-5 (Deviation) | 20-5 (Time) | 50-10 (Deviation) | 50-10 (Time) | 100-10 (Deviation) | 100-10 (Time) |
|---|---|---|---|---|---|---|
| NEH | 3.632 | 0.034s | 5.278 | 1.735s | 2.143 | 12.799s |
| Greedy NEH | 1.453 | 1.892s | 5.343 | 21.663s | 1.982 | 296.247s |
| Ham | 7.667 | 0.006s | 13.293 | 0.008s | 12.788 | 0.000s |
| Gupta | 12.187 | 0.002s | 18.957 | 0.007s | 16.521 | 0.003s |
| Palmer | 9.766 | 0.004s | 14.033 | 0.012s | 12.163 | 0.004s |
| CDS | 12.429 | 0.005s | 13.293 | 0.048s | 10.752 | 0.025s |
| PRSKE | 32.042 | 0.001s | 22.433 | 0.010s | 19.593 | 0.000s |
| Artificial Heuristic | 12.429 | 0.004s | 11.587 | 0.047s | 11.252 | 0.037s |

Table 2: Performance comparison on the seventh instances from 3 Taillard benchmarks for different heuristics.

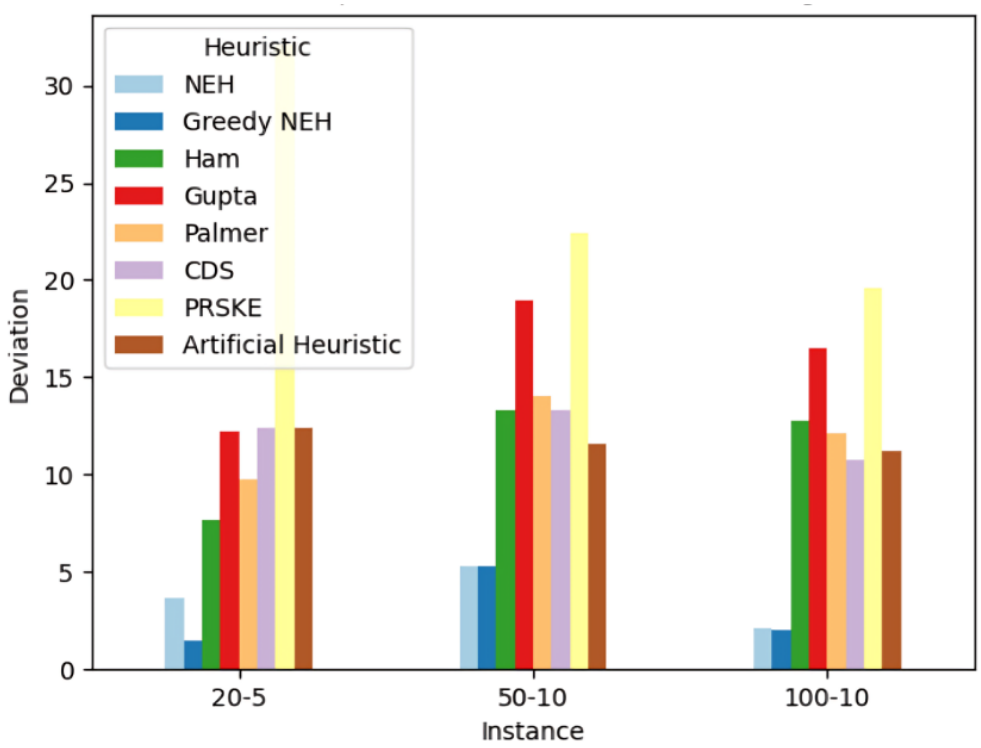


Figure 3: Makespan comparison for different seventh instances from 3 Taillard benchmarks using different heuristics

## 4.4 Discussion And Analysis

After conducting experiments with all of the heuristics mentioned above, we have determined that heuristics can provide solutions of reasonable quality within a reasonable amount of time. Additionally, heuristics can serve as a useful initialization step for subsequent use with Branch & Bound or metaheuristic algorithms, as we will explore in the next chapter.

Heuristics are often used in real-world applications due to their speed and practicality, even though they may not always yield optimal solutions. Additionally, the choice of which heuristic to use can depend on the specific problem and its constraints, as different heuristics may perform better in different scenarios.

# 5 Local search-based Metaheuristics for FSP resolution

## 5.1 Overview

Local search metaheuristics are a class of optimization algorithms that use local search methods to explore the search space and find good solutions to optimization problems. These algorithms start with an initial solution and then iteratively improve it by making small changes and evaluating the new solution. The goal is to find the best solution within a given time or iteration limit. These algorithms are often used for solving combinatorial optimization problems.

Compared to heuristics, local search metaheuristics are more effective for solving the flowshop problem because they can escape from local optima and find better solutions. Heuristics are simpler algorithms that make use of domain-specific knowledge to generate solutions. While they can be effective in some cases, they often get stuck in suboptimal solutions and cannot explore the search space as thoroughly as local search metaheuristics.

## 5.2 Local search metaheuristics for FSP

### 5.2.1 Random walk

Random walk metaheuristic is very simple and easy to implement and can be applied to a wide range of optimization problems. The algorithm begins by initializing a starting solution $S_0$ and then generates random neighboring solutions. It continues iterating until a certain condition is met, at which point it stops.

### 5.2.2 Hill climbing

Hill climbing is another simple metaheuristic optimization algorithm that iteratively improves a candidate solution. The algorithm starts with an initial solution $S_0$ and then repeatedly evaluates neighboring solutions and moves to the best neighboring solution that improves the objective function until a local optimum is reached. There are several types of hill climbing algorithms, including:

- **Simple hill climbing:** This is the most basic type of hill climbing algorithm. It chooses the first neighbor solution that is better than the current solution. The search stops either when no further improvements can be made or when the stop condition is met.
- **Steepest-ascending hill climbing:** This variant of hill climbing generates all neighboring solutions and selects the best one that improves the objective function the most. It is more computationally expensive than simple hill climbing but can converge to better solutions.
- **Stochastic hill climbing:** To add variation in the search process, this variant of hill climbing selects a random neighbor from all the neighbors that outperform the current solution, rather than choosing the best one. This randomness aids in avoiding local optima and promoting a more comprehensive search of the solution space.

### 5.2.3 Simulated annealing

Simulated annealing is a metaheuristic optimization algorithm inspired by the process of annealing in metallurgy. The algorithm starts with an initial solution and a high temperature, and then iteratively generates a new solution by making a small perturbation to the current solution. The acceptance of the new solution is determined by the probability function, which depends on the difference in the objective function value and the current temperature. The temperature is gradually reduced over time according to a cooling schedule, which determines the rate at which the temperature decreases. The algorithm terminates when the final temperature is reached.

The effectiveness of the algorithm depends on the choice of the cooling schedule and the initial temperature, as well as the perturbation strategy used to generate new solutions.

### 5.2.4 Tabu Search

Tabu search is another popular metaheuristic optimization algorithm that is used to find high-quality solutions to combinatorial optimization problems. The algorithm is based on the idea of using a "tabu list" to keep track of previously visited solutions and prevent the algorithm from revisiting them, to encourage exploration of new parts of the search space. The tabu list is a memory structure that stores information about recently visited solutions, such as the moves that were made to reach them. These moves are "tabu" or forbidden for a certain number of iterations, to prevent the algorithm from revisiting solutions that have already been explored. This encourages the algorithm to explore new parts of the search space and avoid getting stuck in local optima.

### 5.2.5 VNS

Variable Neighborhood Search is another local search algorithm for solving combinatorial optimization problems. It starts with an initial solution and iteratively explores the search space by generating neighboring solutions and evaluating their quality. Unlike other local-search metaheuristics, VNS uses multiple search neighborhoods of increasing size and diversity to escape from local optima. At each iteration, the algorithm selects a random neighborhood and applies a perturbation to escape the current local minimum. The search then continues in the new neighborhood until a better solution is found or a stopping criterion is met.

## 5.3 Tests

The tests for local search metaheuristics were divided into two parts, each focusing on generating the initial solution using different methods. The purpose of this division was to compare the performance of the metaheuristic algorithms based on the initial solutions obtained.

In the first part of the tests, the initial solution was randomly generated whereas in the second part, the initial solution was generated using the NEH heuristic.

### 5.3.1 Random initialization

The following table summarizes the test results on the first and seventh instances of the first benchmark. The initial solution was generated randomly.

| **Metaheuristic** | **20-5-1 (Makespan)** | **20-5-1 (Time)** | **20-5-7 (Makespan)** | **20-5-7 (Time)** |
|---|---|---|---|---|
| Random Walk | 1485.000 | 0.207s | 1509.000 | 0.179s |
| Simple Hill climbing | 1323.000 | 0.265s | 1262.000 | 0.177s |
| Steepest Ascent Hill climbing | 1385.000 | 27.874s | 1392.000 | 28.596s |
| Stochastic Hill climbing | 1305.000 | 0.331s | 1251.000 | 0.430s |
| Simulated annealing | 1297.000 | 26.561s | 1251.000 | 27.610s |
| Tabu Search | 1297.000 | 26.501s | 1285.000 | 26.214s |
| VNS | 1297.000 | 0.133s | 1252.000 | 0.12 |

Table 3: Performance comparison on the first and seventh instances of the first Taillard benchmark for different local search algorithms using a random initialization.

### 5.3.2 NEH initialization

The following table summarizes the test results on the first and seventh instances of the first benchmark. The initial solution was generated by NEH heuristic.

| Metaheuristic | 20-5-1 (Makespan) | 20-5-1 (Time) | 20-5-7 (Makespan) | 20-5-7 (Time) |
|---|---|---|---|---|
| Random Walk | 1422.000 | 0.212s | 1476.000 | 0.210s |
| Simple Hill climbing | 1305.000 | 0.087s | 1251.000 | 0.129s |
| Steepest Ascent Hill climbing | 1331.000 | 28.402s | 1280.000 | 28.898s |
| Stochastic Hill climbing | 1305.000 | 0.098s | 1251.000 | 0.208s |
| Simulated annealing | 1305.000 | 27.971s | 1251.000 | 25.217s |
| Tabu Search | 1305.000 | 27.079s | 1251.000 | 27.072s |
| VNS | 1305.000 | 0.135s | 1259.000 | 0.131s |

Table 4: Performance comparison on the first and seventh instances of the first Taillard benchmark for different local search algorithms using the NEH initialization.

Here is a visual representation that illustrates the makespans achieved by different methods, considering both random initialization and NEH initialization:

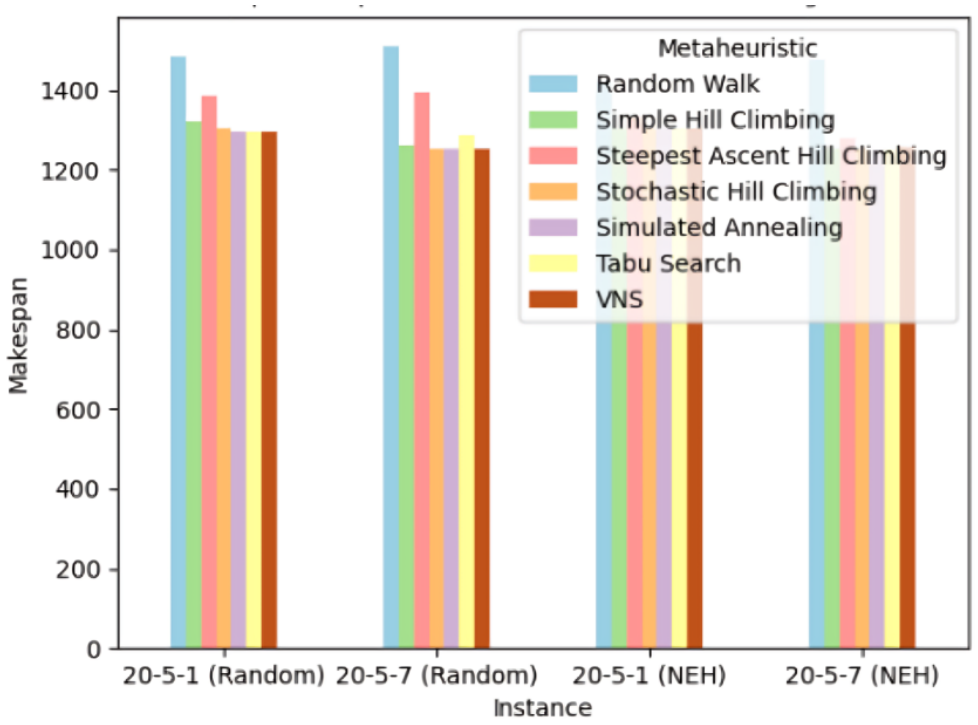


Figure 4: Makespan comparison for different instances from using different metaheuristics

### 5.3.3 Hyperparameters tuning

Hyperparameters play a critical role in the performance of local search metaheuristics, as they can significantly impact the resulting solution quality and computation time. Selecting appropriate values for hyperparameters, such as the initial temperature and cooling rate for Simulated Annealing, can be crucial in achieving optimal results. Additionally, tuning hyperparameters can involve a trade-off between solution quality and computation time, as choosing certain hyperparameters may lead to better solutions but also result in longer execution times. Below are some tests to elaborate on this point.

**Simulated Annealing** Our implementation of simulated annealing has four hyperparameters: initial temperature, final temperature, *alpha* and neighboring method. Below, we tried to change the initial temperature to see Its effect on the result and time. Here are the results obtained using an initial temperature of 100, a final temperature of 1, an *alpha* value of 0.1, and random insertion as the neighboring method.

```
Initial solution: [9, 7, 2, 5, 8, 4, 1, 6, 0, 3]
Makespan: 1515.0
Generated solution: [0, 2, 4, 9, 5, 3, 7, 1, 8, 6]
Makespan: 1312.0
Elapsed time: 0.47452425956726074 seconds
```

Listing 12: Results of Simulated Annealing with initial temperature of 100 a final temperature of 1 an alpha of 0.1 on a random sequence of 10 jobs and 5 machines.

Here are the results obtained using an initial temperature of 300, a final temperature of 1, an *alpha* value of 0.1, and random insertion as the neighboring method.

```
Initial solution: [9, 7, 2, 5, 8, 4, 1, 6, 0, 3]
Makespan: 1515.0
Generated solution: [2, 5, 4, 1, 9, 3, 0, 4, 7, 8, 6]
Makespan: 1306.0
Elapsed time: 1.59752425447726023 seconds
```

Listing 13: Results of Simulated Annealing with initial temperature of 100 a final temperature of 1 an alpha of 0.1 on a random sequence of 10 jobs and 5 machines.

**VNS** Our implementation of VNS has two main hyperparameters: max number of iterations, and $k_max$. Below, we tried to change the $k_max$ hyperparameter to see Its effect on the result and time. Here are the results obtained using a maximum number of iterations equals 400, and $k_max$ equals 5.

```
Initial solution: [9, 7, 2, 5, 8, 4, 1, 6, 0, 3]
Makespan: 1515.0
Generated solution: [5, 2, 0, 4, 3, 7, 1, 6, 9, 8]
Makespan: 1324.0
Elapsed time: 0.5181682109832764 seconds
```

Listing 14: Results of VNS with a maximum number of iterations equals to 400 and k max equals to 5.

Here are the results obtained using a maximum number of iterations equal to 400, and k max equal to 6.

```
Initial solution: [9, 7, 2, 5, 8, 4, 1, 6, 0, 3]
Makespan: 1515.0
Generated solution: [2, 4, 0, 5, 1, 7, 8, 6, 3, 9]
Makespan: 1293.0
Elapsed time: 1.4143846035003662 seconds
```

Listing 15: Results of VNS with a maximum number of iterations equals to 400 and k max equals to 6.

## 5.4 Discussion And Analysis

After conducting experiments with all of the local search metaheuristics listed above, we have found that these methods can yield promising results. However, the effectiveness of these methods is dependent on several factors, including the initial solution and the chosen hyperparameters. It is worth noting that certain methods, such as Simple Hill Climbing and Steepest Ascent Hill Climbing, are prone to getting stuck in local optima.

# 6 Population based Metaheuristics

## 6.1 Overview

Population-based metaheuristics, such as genetic algorithms, differential evolution, particle swarm optimization, and ant colony optimization, are a class of optimization algorithms that can be applied to a wide range of problems. These algorithms generate a set of candidate solutions and iteratively update the population by applying a set of operators.

One key characteristic that sets population-based metaheuristics apart from local search methods is their ability to explore a larger portion of the search space. Local search algorithms start from a single solution and iteratively improve upon it by searching the neighborhood of the current solution. However, these methods can easily get stuck in a local optimum, preventing the algorithm from finding a better solution that may be located in a different region of the search space.

On the other hand, population-based metaheuristics maintain a diverse set of solutions and explore different regions of the search space by using various operators to create new solutions. However, they tend to require more computation time compared to local search algorithms. This is because they need to maintain and update a population of candidate solutions and perform operations such as crossover and mutation, which can be computationally expensive.

## 6.2 Genetic algorithm

One of the most well-known population-based metaheuristics is the genetic algorithm (GA) which simulates the process of natural selection and evolution to search for the optimal solution. At each iteration, a population of candidate solutions is evaluated based on a fitness function, and then the most fit solutions are selected for reproduction. The reproduction is performed by combining the selected solutions through crossover and mutation operators to create new solutions. These new solutions replace the least fit solutions in the population, and the process is repeated until a termination criterion is met. One advantage of GA over other population-based metaheuristics is its ability to handle multiple objectives simultaneously, known as multi-objective optimization. GA can search for a set of solutions that optimize multiple objectives, rather than a single optimal solution. Additionally, GA can handle various types of decision variables, such as binary, integer, and real-valued variables.

## 6.3 Ant colony optimization

ACO is a metaheuristic approach adapted for the Flow Shop Scheduling Problem (FSP). Ants construct schedules by selecting tasks based on pheromone levels and machine attractiveness. Pheromone levels are updated based on solution quality, reinforcing paths to better schedules. ACO explores the solution space using collective intelligence and can handle varying problem instances. Enhancements like local search and hybridization further improve its performance for complex FSP instances. By leveraging the collective intelligence of the ant population, ACO can effectively explore the solution space of the FSP. It can potentially discover high-quality schedules that minimize the makespan or other objective criteria.

## 6.4 Hybrid GA with VNS

The research paper titled "Minimizing makespan in permutation flow shop scheduling problems using a hybrid metaheuristic algorithm" proposes a hybrid metaheuristic algorithm that integrates Genetic Algorithm and Variable Neighborhood Search to solve the permutation flow shop scheduling problem and minimize the makespan. It combines global exploration with local search to find near-optimal solutions, improving the efficiency of flow shop scheduling. Below is a schematic representation illustrating the hybridization process:

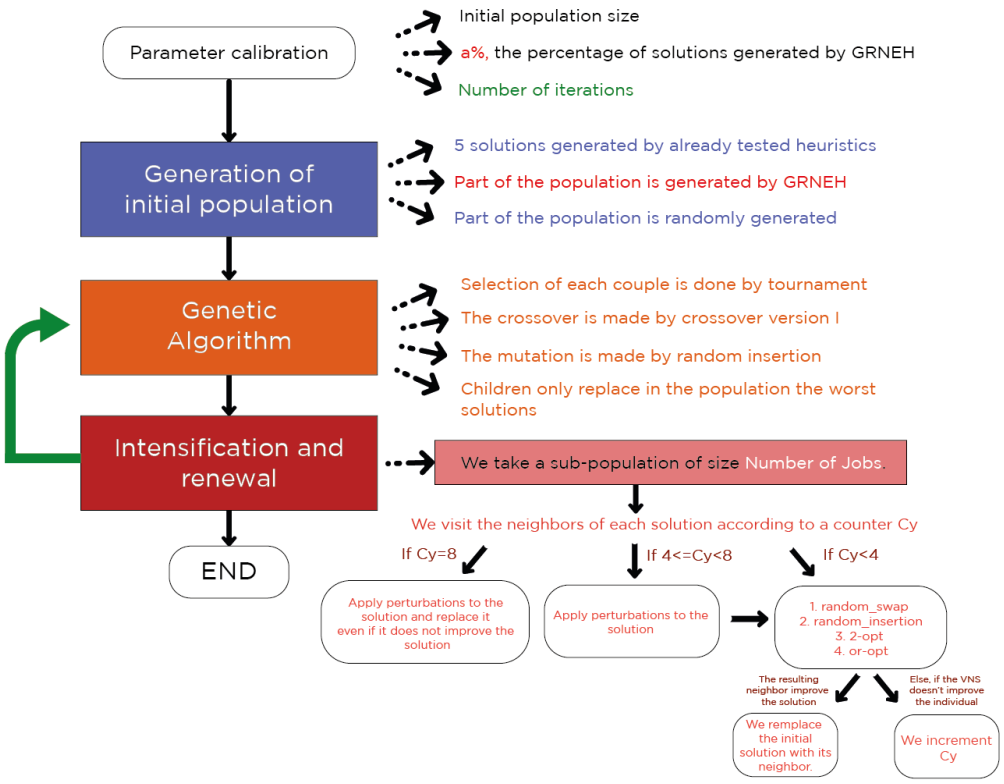


Figure 5: An illustration summarizing the hybridization process.

## 6.5 Tests

We conducted tests on the shared instance (10 jobs 5 machines) as well as the first, second, fifth, and seventh instances of the first benchmark of Taillard instances (20 jobs and 5 machines) to show the performance of the different metaheuristics that we implemented. The results are implemented below, where (MS) stands for MakeSpan.

| **Algorithm** | Common (MS) | Common (Time) | 20-5-1 (MS) | 20-5-1 (Time) | 20-5-2 (MS) | 20-5-2 (Time) | 20-5-5 (MS) | 20-5-5 (Time) | 20-5-7 (MS) | 20-5-7 (Time) |
|---|---|---|---|---|---|---|---|---|---|---|
| Hybrid | 1104 | 0.504s | 1278 | 7.048s | 1359 | 9.155s | 1235 | 7.586s | 1239 | 9.022s |
| GA | 1107 | 0.547s | 1330 | 1.129s | 1393 | 1.193s | 1311 | 1.074s | 1288 | 1.074s |
| Ant colony | 1113 | 0.264s | 1380 | 0.289s | 1415 | 0.554s | 1355 | 0.519s | 1360 | 0.521s |
| NEH | 1105 | 0.004s | 1334 | 0.026s | 1367 | 0.035s | 1319 | 0.034s | 1284 | 0.030s |
| Artificial heuristic | 1110 | 0.002s | 1367 | 0.004s | 1432 | 0.003s | 1300 | 0.003s | 1393 | 0.004s |
| VNS | 1105 | 1.572s | 1297 | 16.344s | 1366 | 14.122s | 1250 | 11.958s | 1251 | 12.660s |

Table 5: Performance comparison for different instances and algorithms.

Here's a graphical depiction showcasing the makespans of various methods seen in this report, including all the metaheuristics :

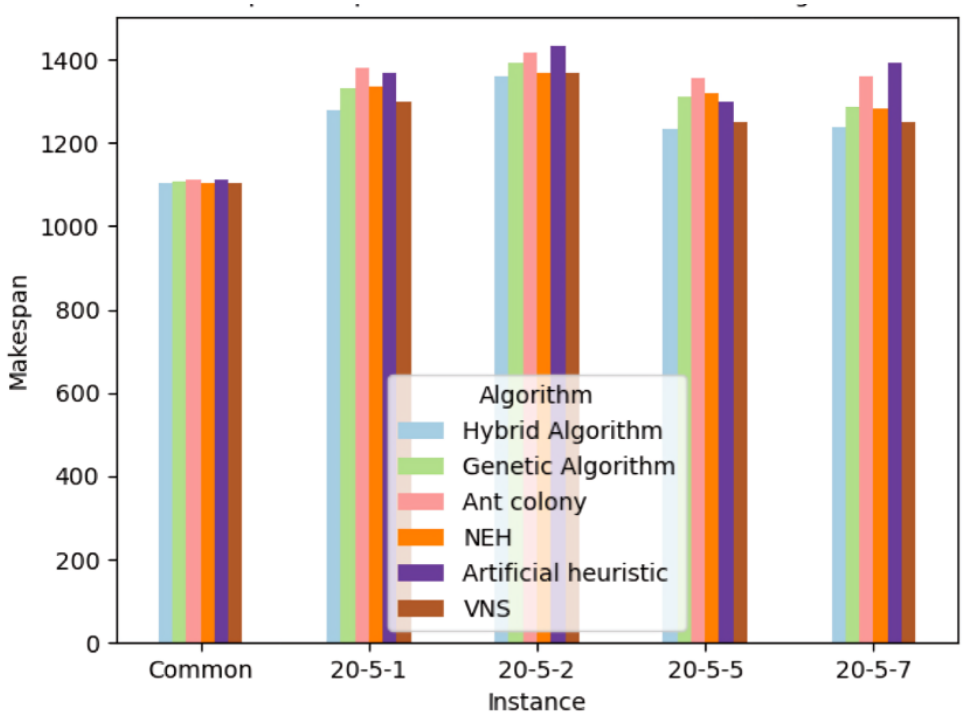


Figure 6: Makespan comparison for different instances using different algorithms.

### 6.6 Discussion And Analysis

After conducting experiments with all the population-based metaheuristics mentioned earlier, we have observed that these approaches can produce promising results. Among them, the hybrid algorithm stood out by achieving the best makespan outcomes. However, it is important to note that this algorithm required more computational time compared to other metaheuristics and heuristics. Additionally, it is worth mentioning that the effectiveness of these methods is also influenced by the selection and tuning of hyperparameters. For the hybrid algorithm, a good compromise would be to exclude the VNS from the iterations, keeping only the GA, but to pass the highest quality individual that the GA returns through a VNS.

## 7 Conclusion

In this report, we have explored various optimization techniques for the flowshop scheduling problem, ranging from exact algorithms such as Branch and Bound to heuristic methods like NEH, local search heuristics such as VNS, and population-based metaheuristics like GA. Each method has its strengths and weaknesses, and the choice of method depends on the specific problem instance and constraints.

While exact methods provide optimal solutions, they may be computationally infeasible for large problem instances. Heuristic methods provide good solutions within reasonable time limits but may get stuck in local optimum. Local search metaheuristics are more effective for solving the flowshop problem because they can escape from local optima and find better solutions. On the other hand, Population-based methods explore a larger portion of the search space and can provide better solutions but at the expense of higher computational time. Overall, it is important to carefully select and adapt optimization methods based on the specific problem requirements and constraints.

# References


1. Pérez-Rosés, Hebert Cabrera, Guillem Juan, Angel Marquès, Joan Faulin, Javier. (2013). Promoting green internet computing throughout simulation-optimization scheduling algorithms. Proceedings of the 2013 Winter Simulation Conference - Simulation: Making Decisions in a Complex World, WSC 2013. 10.1109/WSC.2013.6721571.
2. Das, M. K. (2014). Selected heuristic algorithms for solving job shop and flow shop scheduling problems (Doctoral dissertation).
3. Liu, W., Jin, Y., Price, M. (2017). A new improved NEH heuristic for permutation flowshop scheduling problems. International Journal of Production Economics, 193, 21-30.
4. Gupta, A., Chauhan, S. (2015). A heuristic algorithm for scheduling in a flow shop environment to minimize makespan. International Journal of Industrial Engineering Computations, 6(2), 173-184.
5. Hansen, P., Mladenović, N. (2001). Variable neighborhood search: Principles and applications. European journal of operational research, 130(3), 449-467.
6. Riyanto, O. A. W., Santosa, B. (2015). ACO-LS algorithm for solving no-wait flow shop scheduling problem. In Intelligence in the Era of Big Data: 4th International Conference on Soft Computing, Intelligent Systems, and Information Technology, ICSIIT 2015, Bali, Indonesia, March 11-14, 2015. Proceedings 4 (pp. 89-97). Springer Berlin Heidelberg.
7. Zobolas, G. I., Tarantilis, C. D., Ioannou, G. (2009). Minimizing makespan in permutation flow shop scheduling problems using a hybrid metaheuristic algorithm. Computers Operations Research, 36(4), 1249-1267.