A Novel Generative Hyperheuristic Approach Integrating Genetic Algorithm and Tabu Search with RL for Optimizing PFSP Problem

CHOUIKRAT Sabrina[1], DJELLAL Meriem[1], KARIM Meryem Batoul[1], GUERROUACHE Hiba[1], DJEBLAHI Maria[1], and BESSEDIK Malika[1]

[1]**Ecole Nationale Superieure d'informatique (ESI) , Algiers, Algeria**

January 10, 2026

Abstract

This paper presents a novel hyperheuristic approach, termed Genetic Tabu Search (GTS), for solving the Permutable flowshop scheduling problem (PFSP). The PFSP problem, known for its complexity and combinatorial nature, poses significant challenges in manufacturing and production environments. Our approach integrates the adaptive search capabilities of Genetic Algorithms (GA) with the memory-based strategic search of Tabu Search (TS), enhanced by reinforcement learning (RL) to dynamically adjust heuristic parameters and guide the search process. The hybrid methodology leverages GA's global search efficiency, TS's local search intensification, and RL's adaptive learning to enhance solution quality . We conducted extensive computational experiments on a set of benchmark PFSP instances to evaluate the performance of GTS. The results demonstrate that GTS rivals traditional GA and TS algorithms, as well as several existing hybrid approaches and gives the best known results for several benchmark instances. The proposed method exhibits robust performance across diverse problem instances, indicating its potential as a powerful tool for complex scheduling problems. This research contributes to the field by providing an effective hyperheuristic framework, paving the way for further advancements in scheduling optimization.

Keywords: Flow Shop Permuation Problem, Genetic Algorithm, Tabu Search, Reinforcement Learning, Hyper-heuristics, NP-hard.

1 Introduction

The Permutable Flowshop Scheduling Problem (PFSP) stands as a fundamental conundrum in production and manufacturing optimization, requiring the efficient sequencing of a set of jobs through a series of machines while keeping the same order for the execution of each job. Its importance lies in its direct impact on resource utilization, production throughput, and overall operational efficiency in various industries.

Efficient scheduling in manufacturing environments is paramount for meeting production deadlines, optimizing resource allocation, and enhancing competitiveness. However, the PFSP's inherent complexity, characterized by numerous possible job permutations and number of machines constraints, poses a formidable computational challenge. Traditional optimization techniques, such as exact algorithms and metaheuristics, often struggle to strike a balance between exploration and exploitation in the vast solution space of the PFSP.

In recent years, hyperheuristics have emerged as a promising avenue for tackling the shortcomings of conventional optimization methods. By operating at a higher level of abstraction and leveraging a diverse set of heuristics, hyperheuristics offer the flexibility and adaptability needed to navigate complex optimization landscapes effectively.

In this paper, we propose a novel hyperheuristic approach, Genetic Tabu Search (GTS), tailored specifically for addressing the PFSP. GTS amalgamates the adaptive search capabilities of Genetic Algorithms (GA) with the memory-based strategic search of Tabu Search (TS), complemented by reinforcement learning (RL) to dynamically adjust heuristic parameters. Through this integration, GTS aims

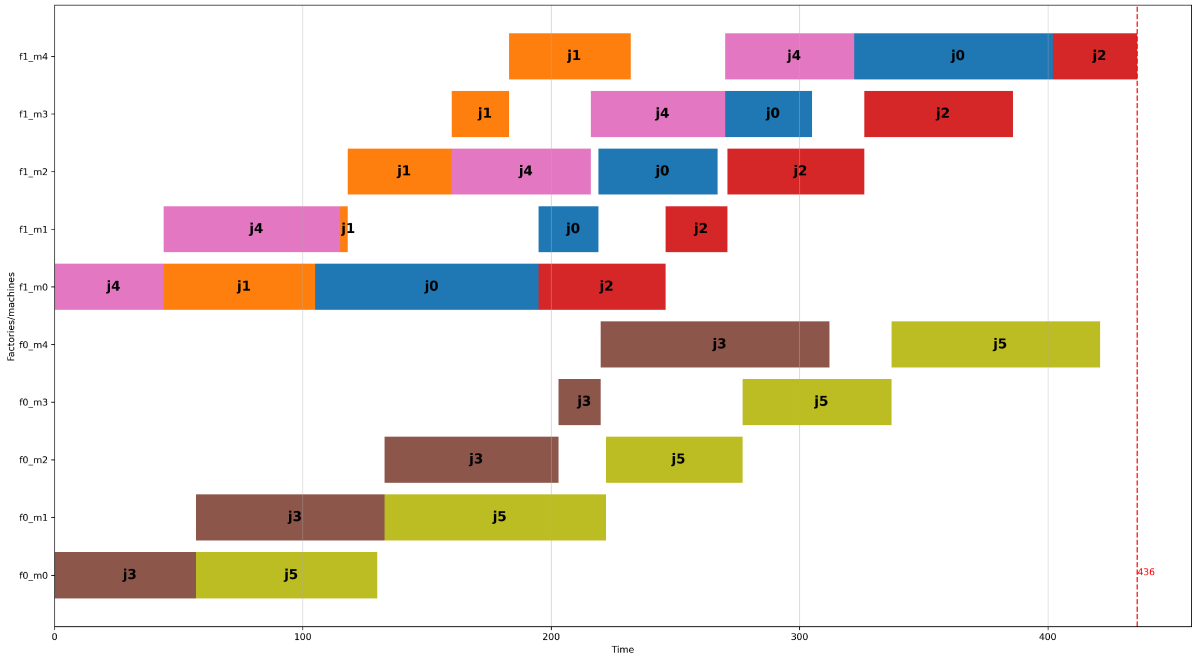

Figure 1: Representation of the permutable flowshop scheduling problem

to enhance solution quality and convergence speed for PFSP instances. The remainder of this paper is structured as follows: Section 2 provides a comprehensive review of related literature concerning the PFSP and hyperheuristic methodologies. Section 3 defines the problem in a more mathematical manner, Section 4 delineates the methodology and framework of the proposed GTS algorithm. Section 5 presents the results of experiments and performance analysis comparing GTS to existing methods. Section 6 discusses the implications of our findings and avenues for future research. By shedding light on the efficacy of hyperheuristic approaches in addressing complex manufacturing challenges, this study endeavors to contribute to the advancement of scheduling optimization techniques.

2 Related Work

The Permutable Flowshop Scheduling Problem (PFSP) has been extensively studied due to its critical importance in manufacturing and production optimization. Traditional approaches, such as exact algorithms and heuristic methods, have laid the foundation for solving PFSP, but they often struggle with reaching optimal and best-known results.

2.1 Exact Algorithms

Early research on PFSP emphasized exact algorithms to address its complexity. Techniques such as branch and bound, dynamic programming, and integer programming were explored to find optimal solutions . Branch and bound algorithms [22] decompose the problem into smaller subproblems, systematically exploring the solution space while pruning branches that cannot lead to optimal solutions. However, these methods often face challenges in scalability due to their exponential time complexity.

2.2 Heuristic Methods

Heuristic methods have played a significant role in addressing the complexities of flowshop scheduling problems. Among the early heuristic approaches, Johnson's rule [12] stands out as a seminal contribution, particularly for two-machine flowshops, offering optimal solutions under specific conditions. Johnson's rule prioritizes jobs based on their processing times on two machines, aiming to minimize the makespan.

The NEH (Nawaz, Enscore, and Ham) heuristic, proposed by Nawaz et al. [19], addresses flowshop scheduling problems by iteratively inserting jobs into the schedule based on a priority rule derived from

the total processing time. Despite its simplicity, NEH has demonstrated competitive performance in generating near-optimal solutions for moderate-sized instances of flowshop scheduling problems.

Other heuristics specific to flowshop scheduling problems include Palmer's heuristic [21], CDS (Constructive Dispatching Sequence) [7], and Regret Heuristic. Palmer's heuristic prioritizes jobs based on their earliest due dates, aiming to minimize tardiness. CDS constructs a feasible dispatching sequence by prioritizing jobs with the shortest total processing times. Regret heuristic selects jobs based on the difference in processing times between the two machines, aiming to minimize the regret in scheduling decisions.

These heuristic methods, tailored to the characteristics of flowshop scheduling problems, offer efficient approaches for generating feasible schedules and have been widely adopted in practice due to their simplicity and effectiveness, however for certain problem instances they fail to reach optimal results and are more susceptible to get stuck in local optimums.

2.3 Metaheuristic Approaches

Metaheuristic algorithms offer robust optimization techniques for addressing complex combinatorial optimization problems like the flowshop scheduling problem. These methods provide efficient strategies for exploring the solution space, often without requiring full knowledge of the problem domain. Metaheuristics can be broadly classified into population-based algorithms and single-solution-based algorithms.

2.3.1 Population-Based Metaheuristics

Population-based metaheuristics maintain and evolve a population of candidate solutions throughout the optimization process. They explore the solution space by iteratively updating and exchanging information among multiple solutions.

- Genetic Algorithms (GA) [9] proposed by David.E Inspired by the principles of natural evolution, genetic algorithms use techniques such as selection, crossover, and mutation to iteratively evolve a population of candidate solutions towards optimal or near-optimal solutions .
- Ant Colony Optimization (ACO)[4]:proposed by Marco and Thomas. ACO is based on the foraging behavior of ants and employs a population of artificial ants to explore the solution space. These ants deposit pheromones on the edges of a solution graph, guiding the search towards promising regions .
- Genetic Programming (GP)[15]:proposed by John R Similar to genetic algorithms, genetic programming evolves populations of computer programs or solutions. It represents solutions as trees and applies genetic operators to create new program variations .
- Particle Swarm Optimization (PSO)[13]:proposed by James and Russel ,PSO simulates the social behavior of bird flocking or fish schooling. It maintains a population of particles moving in the solution space, adjusting their positions based on their own best-known position and the global best-known position found by the swarm .

2.3.2 Single-Solution-Based Metaheuristics

Single-solution-based metaheuristics focus on iteratively improving a single candidate solution without maintaining a population.

- Simulated Annealing (SA)[14]:Proposed by S Kirkptrick, Inspired by the annealing process in metallurgy, simulated annealing probabilistically accepts worse solutions during the search, allowing it to escape local optima. It gradually decreases the probability of accepting worse solutions as the search progresses .
- Tabu Search (TS)[8]:Proposed by Fred Glover, Tabu search maintains a single current solution and explores the neighborhood by iteratively moving to neighboring solutions while avoiding previously visited solutions based on a tabu list. It aims to escape local optima by diversifying the search .
- GRASP (Greedy Randomized Adaptive Search Procedure)[6]:Proposed by Ta Feo, GRASP combines greedy construction with randomization and iterative improvement. It constructs solutions incrementally and applies local search to refine them. The randomization component allows GRASP to explore diverse solution spaces .

- Iterated Local Search (ILS)[17]:Proposed by Hr Lourenço, ILS iteratively applies local search to improve a candidate solution and perturbs the solution to explore new regions of the solution space. It combines intensification and diversification strategies .

These metaheuristic approaches offer flexible and efficient optimization strategies for a wide range of combinatorial optimization problems, including flowshop scheduling. The choice of algorithm depends on factors such as problem characteristics, computational resources, and desired solution quality.

2.4 Hyperheuristics

Several researches in the area of hyperheuristics have contributed significantly to the field, starting with pioneering work by Burke et al. (2003). Hyperheuristics can generally be categorized into two main types: selection hyperheuristics and generation hyperheuristics.

2.4.1 Selection Hyperheuristics

Selection hyperheuristics focus on selecting the most appropriate heuristic from a predefined set of heuristics at each decision point during the problem-solving process.

- **Burke et al. (2003)**: The foundational work by Burke et al. [2] emphasized the importance of adaptive heuristic selection in solving diverse computational problems. They introduced a framework that dynamically chooses between heuristics to enhance the problem-solving process.
- **Gupta and Palmer (2016)**: Gupta and Palmer [20] proposed a selection hyperheuristic framework based on genetic algorithms for scheduling problems in the textile industry to manipulate Gupta and palmer heuristics in low level.
- **Bandit-based approaches**: Droste et al. (2021) [5] investigated using bandit algorithms for online selection of hyper-heuristics in resource-constrained environments

2.4.2 Generation Hyperheuristics

Generation hyperheuristics aim to generate new heuristics or heuristic components by exploring the space of possible heuristics.

- **Kumar et al. (2010)**: Kumar et al. [16] developed a generation hyper-heuristic that utilizes particle swarm optimization to evolve composite heuristics for job scheduling problems.
- **Bacha et al. (2017)**: Bacha et al. [1] introduced an innovative generation hyper-heuristic that employs a high-level genetic algorithm to effectively manipulate low-level genetic algorithm components.
- **Mizraqi et al. (2018)**: Mizraqi et al. [18] introduced a generation hyper-heuristic approach that employs ant colony optimization to evolve heuristic rules for the Knapsack Problem.

2.5 Our Contribution

Building on these foundations, our proposed approach involves the design of a high-level genetic algorithm that manipulates the low-level algorithm of tabu search. This manipulation occurs through a hyperchromosome that contains parameters of the low-level tabu search. The objective is to guide the adaptation of these parameters effectively to reach the most optimal possible solution. Integrating this concept with a reinforcement learning approach for population initialization enhances the adaptability and efficiency of the algorithm.

In summary, while significant advancements have been made in solving PFSP, challenges remain in achieving both optimality and computational efficiency for large-scale instances. Our work contributes to this ongoing effort by introducing an adaptive hyperheuristic framework that leverages the strengths of genetic algorithms, tabu search, and reinforcement learning to try and address these challenges effectively.

3 Problem Definition

The Permutable Flowshop Scheduling Problem (PFSP) is a well-known combinatorial optimization problem in the field of production and scheduling. In the PFSP, a set of n jobs must be processed on a series of m machines, with each job requiring processing on each machine in the same order. The objective is to determine the optimal sequence in which to process the jobs on the machines, minimizing a certain performance criterion such as makespan, total completion time, or total flow time.

Formally, let $J = \{J_1, J_2, \ldots, J_n\}$ denote the set of n jobs and $M = \{M_1, M_2, \ldots, M_m\}$ denote the set of m machines. Each job J_i consists of m operations, denoted as O_{ij}, where $1 \leq i \leq n$ and $1 \leq j \leq m$. The processing time of operation O_{ij} on machine M_k is denoted as p_{ijk}.

The goal of the PFSP is to find a permutation π of the jobs such that when the jobs are processed in the order specified by π, the performance criterion is minimized. The performance criterion varies depending on the specific objective function chosen. Common objective functions include:

- **Makespan**: Minimize the maximum completion time of any job, i.e., the time at which all jobs are completed.
- **Total Completion Time**: Minimize the sum of completion times of all jobs.
- **Total Flow Time**: Minimize the sum of flow times of all jobs, where the flow time of a job is the difference between its completion time and its release time.

The PFSP is subject to the following constraints:

- **Machine Order Constraint**: Each job must be processed on the machines in the same order. That is, for any job J_i, it must be processed on M_1, then M_2, and so on up to M_m.
- **Job Execution Constraint**: Each machine can process only one job at a time. That is, no two jobs can be processed simultaneously on the same machine.
- **Job Precedence Constraint**: Once a job J_i starts processing on a machine M_j, it must complete processing on that machine before moving on to the next machine M_{j+1}.
- **No Preemption Constraint**: Jobs cannot be interrupted once they start processing on a machine.
- **Initial Condition Constraint**: All jobs are available for processing at time zero.

The PFSP is known to be NP-hard, and its complexity increases with the number of jobs and machines. Thus, finding optimal solutions for large instances of the PFSP is computationally challenging, necessitating the development of efficient heuristic and metaheuristic algorithms for practical implementation.

4 Proposed Approach

We propose a new generation hyper-heuristic based on genetic algorithms and Tabu Search to solve the Permutational Flowshop Problem (PFSP). Our challenge is to tailor a suitable Tabu Search algorithm for a given problem instance through a genetic algorithm. Our hyper-heuristic consists of three layers as it is illustrated in the following figure and explained forth:

- **Reinforcement Learning Layer**: Used to initialize the population of the high-level algorithm.
- **High-Level Layer**: A genetic algorithm that evolves low-level algorithm configurations to find the optimal combination.
- **Low-Level Layer**: Consists of Tabu Search instances dedicated to solving the PFSP.

In this section, we first present the strategy and design of the low-level Tabu Search, followed by the high-level genetic algorithm through their respective encoding schemes and operators. Then, we move on to presenting the reinforcement learning approach used for the initialization of the genetic algorithm.

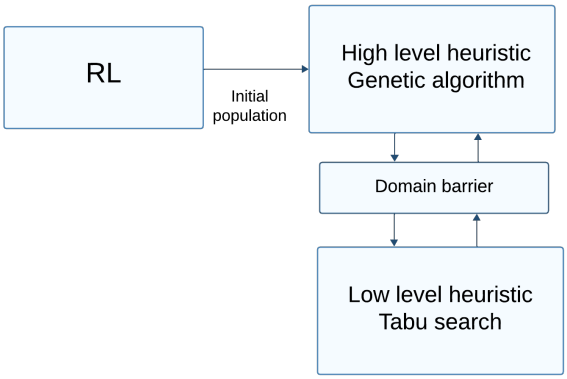

Figure 2: General architecture of our proposed hyperheuristic.

4.1 Low-Level Layer

As described above, we use Tabu Search as the low-level heuristic. This layer comprises TS operators and parameters such as the size of the Tabu list, initialization function, maximum iterations in stagnation, and neighbor generation functions.

In the following subsections, we present the encoding scheme and objective function. Then, we detail the different Tabu Search operators and initialization methods.

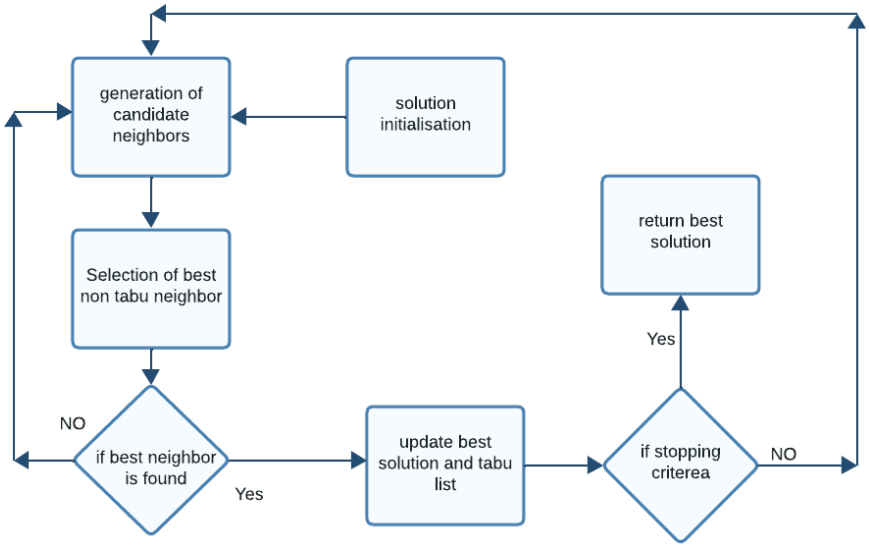

Figure 3: General architecture of a tabu search algorithm.

4.1.1 Encoding Scheme and Objective Function

Since the Tabu Search algorithm is used as a low-level algorithm for the hyper-heuristic, the problem it tackles is an instance of the PFSP. Therefore, the solution will be represented as a sequence of jobs, and the objective function will be the makespan or the necessary time for all the jobs to be executed on all the machines.

4.1.2 Initialization

We use two strategies for the generation of the initial solution for the Tabu Search:

- **Random Generation**: The initial solution is generated randomly.

- **Heuristic-Based Generation**: A problem-specific heuristic is used to generate the initial solution. We have used six well-known PFSP heuristics: NEH [19], CDS [7], Palmer [21], Gupta [11], PRSKE [23], and HAM [3].

4.1.3 Generating the Neighborhood of a Solution

For each solution, we can generate its neighborhood using four methods:

- **Swap**: The neighborhood is generated by swapping two jobs at two random positions for each neighbor.
- **Element Insertion**: A random job is selected from the sequence and is then inserted into a random position in the sequence to generate each neighbor.
- **Block Insertion**: A random block of random length of jobs is selected and inserted into a random position in the sequence to generate each neighbor.
- **Random Mix**: Each neighbor is generated using one of the three previous techniques randomly.

We will also be using a list of candidates instead of generating the whole neighborhood. The neighborhood will only be generated completely if there are no better solutions within the list of candidates.

Algorithm 1 low level Tabu Search

```
Input: processing_times, s_0, tabu_size, max_it, max_it_stagn, neighbor_generation
Output: best_solution, best_cmax
Initialize best_cmax ← evaluate_sequence(s_0, processing_times)
Initialize best_solution ← s_0
Initialize TabuList with size tabu_size
Initialize stagnation_count ← 0
for i from 1 to max_it do
    Generate candidate_list(s_0, neighbor_generation)
    best_neighbor ← select_best_neighbor(processing_times, candidate_list, s_0, TabuList, best_cmax)
    if best_neighbor is None then
        stagnation_count ← stagnation_count + 1
        if stagnation_count ≥ max_it_stagn then
            break
        end if
    else
        stagnation_count ← 0
        s_0 ← best_neighbor
        current_makespan ← evaluate_sequence(s_0, processing_times)
        if current_makespan < best_cmax then
            best_solution ← s_0
            best_cmax ← current_makespan
        end if
        TabuList.add(s_0)
    end if
end for
return best_solution, best_cmax
```

4.2 High-Level Layer

The effectiveness of a Tabu Search algorithm greatly depends on the correct choice of parameters and operators. Calibrating the Tabu Search is necessary to get an algorithm that is more effective than all other configurations. We consider the following possibilities for the different configurations of the low-level Tabu Search instances:

- **Tabu List Size:** 9 values [3-11].

- **Initial Solution Generation Methods:** 7 methods (random, NEH, CDS, Palmer, Gupta, PRSKE, and HAM).
- **Maximum Number of Iterations:** 20 values between 100 and 2000 iterations.
- **Maximum Number of Iterations in Stagnation:** 21 values between 50 and 2000, ensuring it is always inferior or equal to the maximum number of iterations.
- **Neighbor Generation Functions:** 4 methods (swap, element insertion, block insertion, random mix).

All these factors result in a total of 105840 different combinations, and thus, just as many search algorithms for each problem instance. Therefore, the objective of the higher-level genetic algorithm is to select the best value for each parameter, and this is done for each instance of the problem. Therefore, the higher-level GA will be manipulating a population composed of Tabu Search instances, each encoded as a hyper-chromosome that is composed of genotypes as follows:

tabu_size	max_it	max_it_stagn	init_funct	neighbor_generation

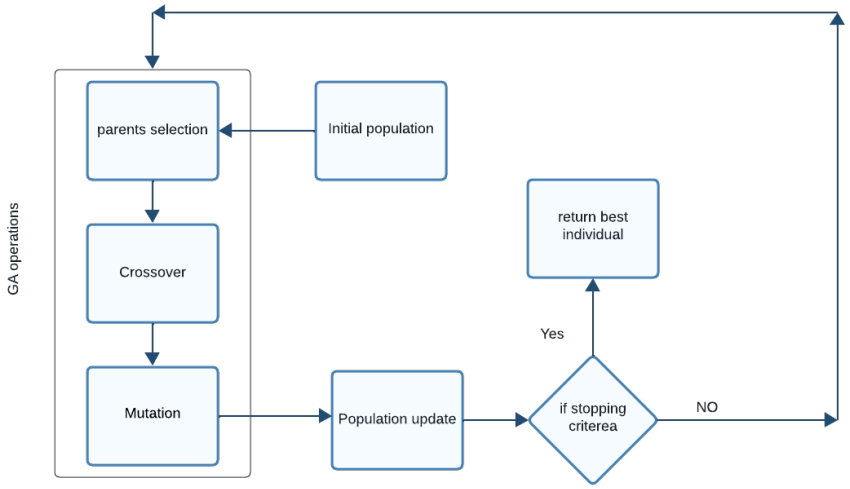

Figure 4: General architecture of a genetic algorithm.

4.2.1 Genetic Algorithm Parameters

The GA also is composed of multiple parameters that allow us to select the right Tabu Search algorithm for the problem instance, which are:

- **Initial Population Size:** Between 10 to 20.
- **Number of Generations:** Between 10 to 20.
- **Selection Method:** We compared four selection methods: roulette selection, rank-based selection, elitist selection, and tournament selection.
- **Probability of Crossover (Pc):** Typically set at 0.9.
- **Probability of Mutation (Pm):** Typically set at 0.1.
- **Population Update Strategy:** Either select the best two candidates out of the parents and child, select one parent and one child, or a percentage of the best solutions and a percentage of the remaining population.

The initial population will be generated using the reinforcement learning logic that will be explained in detail in the upcoming section. Here, we will describe the key components and operators of our high level GA.

4.2.2 Selection Methods

The GA employs various selection methods [10] to ensure diversity and intensification of good solutions. We consider the following selection methods:

- **Roulette Wheel Selection:** Selects individuals based on their fitness proportion, providing a higher chance for better individuals to be selected.
- **Rank-Based Selection:** Ranks individuals and assigns selection probabilities based on their ranks, ensuring a more uniform selection pressure.
- **Elitist Selection:** Always selects the top-performing individuals, ensuring the best solutions are carried forward to the next generation.
- **Tournament Selection:** Randomly selects a subset of individuals and chooses the best among them.

We will be using the tournament selection in our algorithm since it is the one that provides the most balance between exploration and exploitation and has shown the most promising results.

4.2.3 Crossover and Mutation

Crossover and mutation operators are main components of the GA, maintaining genetic diversity and exploring the search space:

- **Crossover:** which is an operation that combines the parameters of two parent individuals to produce offspring. A single-point crossover[9] is the one we will be using, where a random crossover point is selected, and the genotypes before and after the point are exchanged between parents as shown below:

```
Parent 1: tabu_size1, max_it1, max_it_stagn1, init_funct1, neighbor_generation1
Parent 2: tabu_size2, max_it2, max_it_stagn2, init_funct2, neighbor_generation2

Offspring1: tabu_size1, max_it1, max_it_stagn2, init_funct2, neighbor_generation2
Offspring2: tabu_size2, max_it2, max_it_stagn1, init_funct1, neighbor_generation1
```

- **Mutation:** Introduces random changes to an individual's parameters to explore new regions of the search space. The mutation operator randomly selects one genotype and changes its value within the predefined range of the genotype. For example, if init_funct is selected and its value was random, the next value selected will be one of the heuristics used to initialize the Tabu Search.

The mutation and crossover each have a probability of being applied equal to Pm and Pc respectively.

4.2.4 Population Update Strategy

The population update strategy determines how the next generation is formed using the previous generations and the hyper-chromosomes obtained from the crossovers and mutations. We have decided to adopt a combined replacement strategy to ensure that we keep a certain level of diversity in the population while ensuring that the best individuals are always retained. This strategy combines the resulting offspring from the crossovers and mutations with the old population and then selects a percentage of the best individuals. The remaining percentage is randomly selected from the remaining individuals.

4.3 Reinforcement Learning Layer

In this section, we introduce the Q-learning algorithm, which we use as a type of reinforcement learning, and explain how it is implemented in our approach. We first provide an overview of the standard Q-learning algorithm, followed by a detailed description of its implementation within our method.

Algorithm 2 TGA(Npop, generations, num_elites, processing_times,pc,pm,p,k=2)

pop ← initialization(Npop) {Initialize the population}
for each generation in 1 to *generations* **do**
  parents ← tournament_selection(pop, processing_times, Npop // 2, k, p,pc,pm) {Select parents for crossover}
  new_pop ← []
  for each (*parent*1_*idx*, *parent*2_*idx*) in *parents* **do**
    *parent*1 ← *pop*[*parent*1_*idx*]
    *parent*2 ← *pop*[*parent*2_*idx*]
    if random.random() < pc **then**
      *child*1 ← crossover(parent1, parent2) {Perform crossover with 90% probability}
      *child*2 ← crossover(parent2, parent1)
    end if
    if random.random() < pm **then**
      new_pop.append(mutation(child1)) {Perform mutation with 50% probability}
      new_pop.append(mutation(child2))
    end if
  end for
  pop ← PopUpdate(pop, new_pop, num_elites, processing_times) {Update population using elitist strategy}
end for
best_sol ← findBestSolution(pop, processing_times) {Find the best solution in the final population}
return *best_sol* {Return the best individual and its objective value}

4.3.1 Standard Q-Learning Algorithm

The Q-Learning algorithm falls within the realm of reinforcement learning, specifically tailored to enable agents to make optimal decisions within controlled Markov domains [26] .

Fundamentally, Q-Learning operates under the framework of the Markov decision process, where the agent utilizes a value function linked to the Bellman equation. This approach defines a sequential decision-making problem in mathematics, accommodating dynamic environments where states (i.e., transitions or conditions) may change randomly after action execution. Through policies, Q-Learning establishes guidelines for action selection in current states.

Q-Learning has been proven to converge towards the optimal policy under certain conditions within Markov Decision Processes [25]. It continually assesses outcomes based on rewards received during interactions with the environment. This capability, expressed through Q-values, quantifies the effectiveness of specific actions in particular states. Refer to Table 1 for detailed notations and descriptions within the Q-Learning model. By iteratively exploring actions across all states, the agent determines the most advantageous course of action in the current environment, adjusting its strategy as the environment evolves. Figure 5 illustrates the operational dynamics of the standard Q-Learning model. At each step, the agent perceives the current state $s \in S$ of the environment and evaluates possible actions $a \in A(s)$. Based on Q-values stored in the Q-table, the agent selects an action a that maximizes future rewards. After executing action a, the environment transitions to a new state s'.

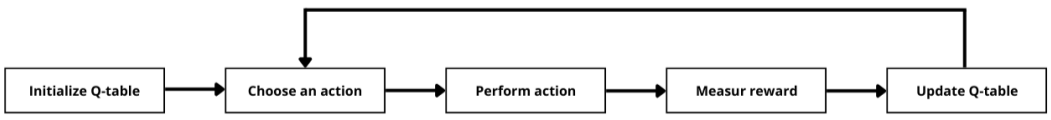

Figure 5: Q-learning Algorithm Workflow.

The Q-value for the state-action pair (s, a) is updated using the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Here, α (learning rate) determines the extent of Q-value updates, ranging from 0 to 1. The parameter γ (discount factor) influences the importance of future rewards, also ranging from 0 to 1. A higher γ value emphasizes long-term rewards, while a lower value prioritizes immediate rewards.

The Q-values are stored in a Q-table and updated iteratively (as illustrated in Table 2). The agent's decision-making process incorporates an ϵ-greedy policy, balancing exploration and exploitation of actions. Initially, the agent explores the environment randomly to gather experience. The parameter ϵ controls the degree of exploration; for example, setting $\epsilon = 0.75$ means the agent follows the optimal strategy 75% of the time and explores randomly 25% of the time.

Table 1: The Q-Learning model notations

Notation	Description
S	A set of environmental states
A	A set of actions
R	A set of scalar rewards
s	A state in S
s'	A next state in S
a	An action in A
r	A reward in R
T	A transition function

As the agent gains knowledge, ϵ can be adjusted to favor exploitation over exploration. In our approach, ϵ remains fixed during decision-making. The specific steps of the standard Q-Learning algorithm are outlined in algorithm 3.

State	Action 1	Action 2	Action 3	Action 4
State 1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$Q(s_1, a_3)$	$Q(s_1, a_4)$
State 2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$Q(s_2, a_3)$	$Q(s_2, a_4)$
State 3	$Q(s_3, a_1)$	$Q(s_3, a_2)$	$Q(s_3, a_3)$	$Q(s_3, a_4)$
State 4	$Q(s_4, a_1)$	$Q(s_4, a_2)$	$Q(s_4, a_3)$	$Q(s_4, a_4)$

Table 2: Sample Q-table.

Algorithm 3 Q-Learning Algorithm

Initialization: Initialize the Q-table arbitrarily.
Set Parameters: Define the learning rate α and discount factor γ.
for each episode **do**
  for each step within the episode **do**
    Select an action a using the ϵ-greedy policy.
    Execute action a, observe the resulting state s' and reward r.
    Update the Q-value for (s, a):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right]$$

    Transition to the new state s'.
  end for
end for

4.3.2 The implementation of Q-Learning

In our genetic algorithm (GA), effective population initialization is crucial for achieving optimal solutions efficiently. Here, we integrate Q-Learning to guide the search towards promising regions early in the optimization process, serving a dual purpose. Q-Learning not only enhances exploration by starting with configurations that are evaluated favorably but also supports exploitation by refining these initial setups over time. This approach allows the algorithm to leverage learned policies when determining

the initial configurations of individuals (hyperchromosomes) within our population, ensuring informed decisions about where exploration should begin.

States and Actions:

- **States:** Represent different combinations ofhyper chromosomes.
- **Actions:** Correspond to modifications applied to hyperchromosomes based on Q-Learning predictions.

Reward Mechanism: The reward for each action is derived from the evaluation function ($c_{\max}$) returned by the tabu search execution using the parameters specified by the hyperchromosome.

After obtaining the final Q-table through Q-Learning training, we utilize it to initialize the population. Each chromosome's initial configuration is determined based on the Q-values stored in the Q-table. Specifically, the Q-table guides the selection of actions (modifications to chromosomes) by prioritizing those actions that have historically led to higher rewards . This ensures that the genetic algorithm starts with configurations that are likely to be more promising, thus accelerating the optimization process.The details of the algorithm are described in Algorithm 4.

Algorithm 4 Q-Learning Initialization for Genetic Algorithm

Input: Trained Q-table
Output: Initialized population for Genetic Algorithm
Initialize an empty population P
$n \leftarrow$ population size
$iterations \leftarrow$ number of iterations for improvement
Find a subset of chromosomes with high Q-values from the Q-table
Let C_{subset} be the set of selected chromosomes
for $i \leftarrow 1$ to n **do**
    Randomly select a chromosome C_i from C_{subset}
    Remove C_i from C_{subset}
    Determine the current state S_i representing the chromosome C_i
    Initialize a counter $iter \leftarrow 0$
    while $iter < iterations$ **do**
        Determine the possible actions from the Q-table for state S_i
        Select action A_i with the highest Q-value for state S_i
        Apply action A_i to modify the chromosome C_i
        Update state S_i to reflect the changes in C_i
        $iter \leftarrow iter + 1$
    end while
    Add C_i to population P
end for
return P

5 Empirical Study

This section presents the results of a series of computational experiments conducted to evaluate the efficiency of the newly developed Hyperheuristic Approach Integrating Genetic Algorithm and Tabu Search with RL (TGA) tailored for Permutation Flow Shop Scheduling Problem (PFSP) instances. The TGA implementation was done in Python using Jupyter Notebook. The experiments were conducted on a computer running Windows 10 Professional, equipped with 8 GB RAM and an Intel Core i5 processor.

All experiments were performed using the well-known flowshop benchmark dataset introduced by Taillard [24].This dataset consists of 12 subsets, each with dimensions n × m. The variable n can be one of the following values: 20, 50, 100, 200, or 500. The variable m can be one of these values: 5, 10, or 20. Therefore, there are a total of 120 instances in the dataset, with 10 instances for each unique combination of n and m.

As metrics we used :

1. **Makespan**: The makespan is the total time required to complete the entire set of jobs. It is a key measure of scheduling efficiency.

2. **Deviation**: The deviation refers to the difference between the result obtained by our hyperheuristic and the best-known solution (upper bound) for each instance.

$$\text{Deviation} = \left(\frac{C_{\max} - \text{UP}}{\text{UP}} \right) \times 100$$

3. **CPU Execution Time**: The CPU execution time is the time required for the algorithm to find the most optimal solution possible.

5.1 Instance Characteristics

In our testing framework for the Permutaional Flow Shop Scheduling Problem (PFSP), we selected instances from the well-known Taillard benchmark[24]. This benchmark is widely recognized in the literature for evaluating the performance of scheduling algorithms, particularly for permutation flowshop problems.

Number of Jobs (n):

- The instances include a variety of job sizes to test the scalability and efficiency of algorithms.
- The specific job sizes are: 20, 50, 100, 200, and 500 jobs.

Number of Machines (m):

- The instances also vary in the number of machines to represent different levels of complexity.
- The specific machine sizes are: 5, 10, and 20 machines.

Instance Categories:

- Small Instances:
  - Job sizes: $n = \{20\}$
  - Machine sizes: $m = \{5, 10, 20\}$
- Medium Instances:
  - Job sizes: $n = \{50\}$
  - Machine sizes: $m = \{5, 10, 20\}$
- Large Instances:
  - Job sizes: $n = \{100, 200, 500\}$
  - Machine sizes: $m = \{5, 10, 20\}$

Processing Time Matrix:

	M_1	M_2	$\cdots$	M_m
J_1	t_{11}	t_{12}	$\cdots$	t_{1m}
J_2	t_{21}	t_{22}	$\cdots$	t_{2m}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
J_n	t_{n1}	t_{n2}	$\cdots$	t_{nm}

The instances chosen for our tests include sizes 20×5, 20×10, 50×5 and 100×5. These specific sizes were selected to cover a range of problem complexities, from smaller instances like 20×5 to larger ones like 100×5, allowing us to assess algorithm performance across different problem scales. This selection strategy ensures that our evaluation encompasses diverse scenarios, enabling a comprehensive analysis of algorithm effectiveness and scalability across various FSP instances.

5.2 Parameter Calibration

In calibrating the parameters for our optimization process, we conducted experiments across various settings. We begin by adjusting the hyperparameters of the tabu algorithm (low level) in order to minimize the search space of the genetic algorithm (high level) then we moved to the Genetic algorithm (high level) calibration.

Phase 1: Tuning Tabu Search HyperParameters (Low Level)

The parameters included the size of the tabu list, the maximum number of iterations allowed, the number of iterations before stagnation. Our calibration process involved systematically adjusting these parameters and observing their effects on the optimization results. We varied the tabu list size from, max iterations , and stagnation iterations as well as the different initialization functions. By carefully tuning these parameters, we aimed to strike a balance between exploration and exploitation within the optimization algorithm, ultimately aiming for improved convergence and solution quality across different problem instances represented by the Taillard benchmark.

- To refine the Tabu Search parameters, we systematically tested various combinations. For instance, to determine the optimal Tabu List Size, we varied this parameter while also adjusting the maximum number of iteration and the number of iterations before stagnation and have done the same for the other parameters and this is how we obtained the different values we worked with in the GA. Some results of these experiments are illustrated in the figures below.

In the figure, the green bar represents the makespan from the Tabu Search, while the blue bar represents the makespan from NEH. The label above each bar indicates the number of maximum iterations.

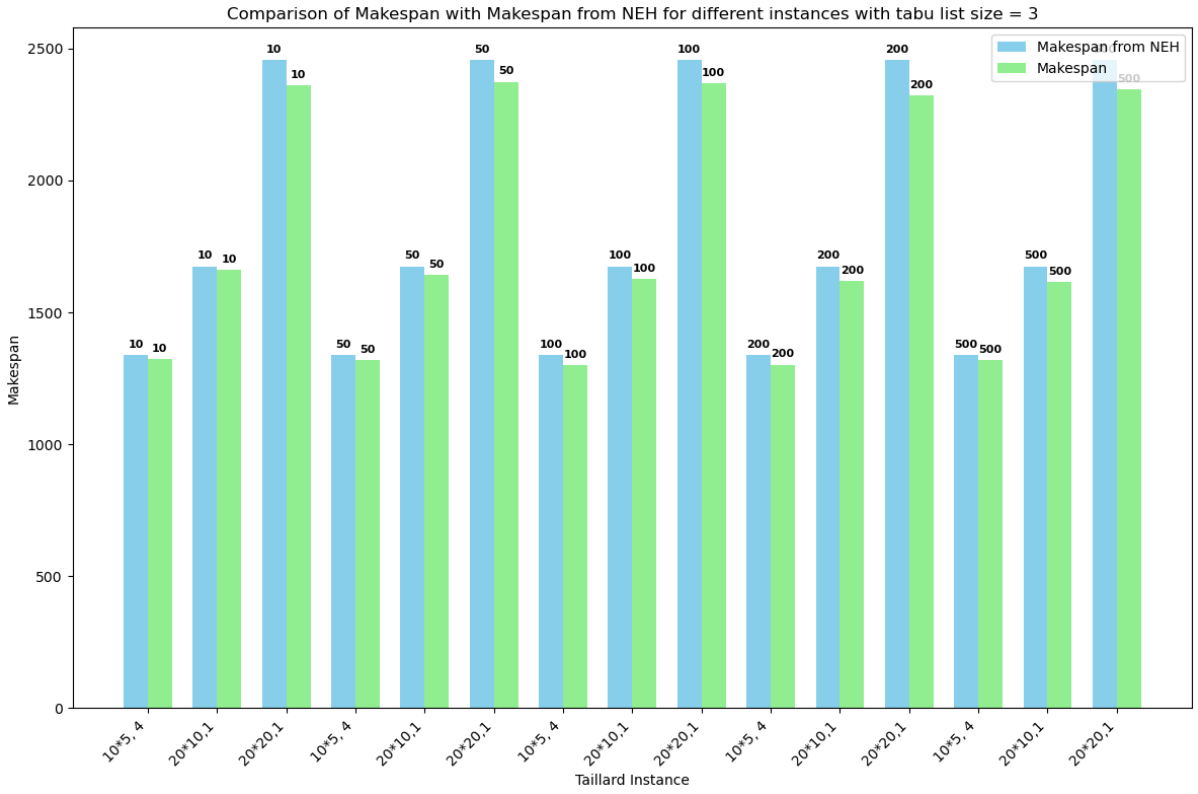

Figure 6: Makespan for Tabu List Size = 3

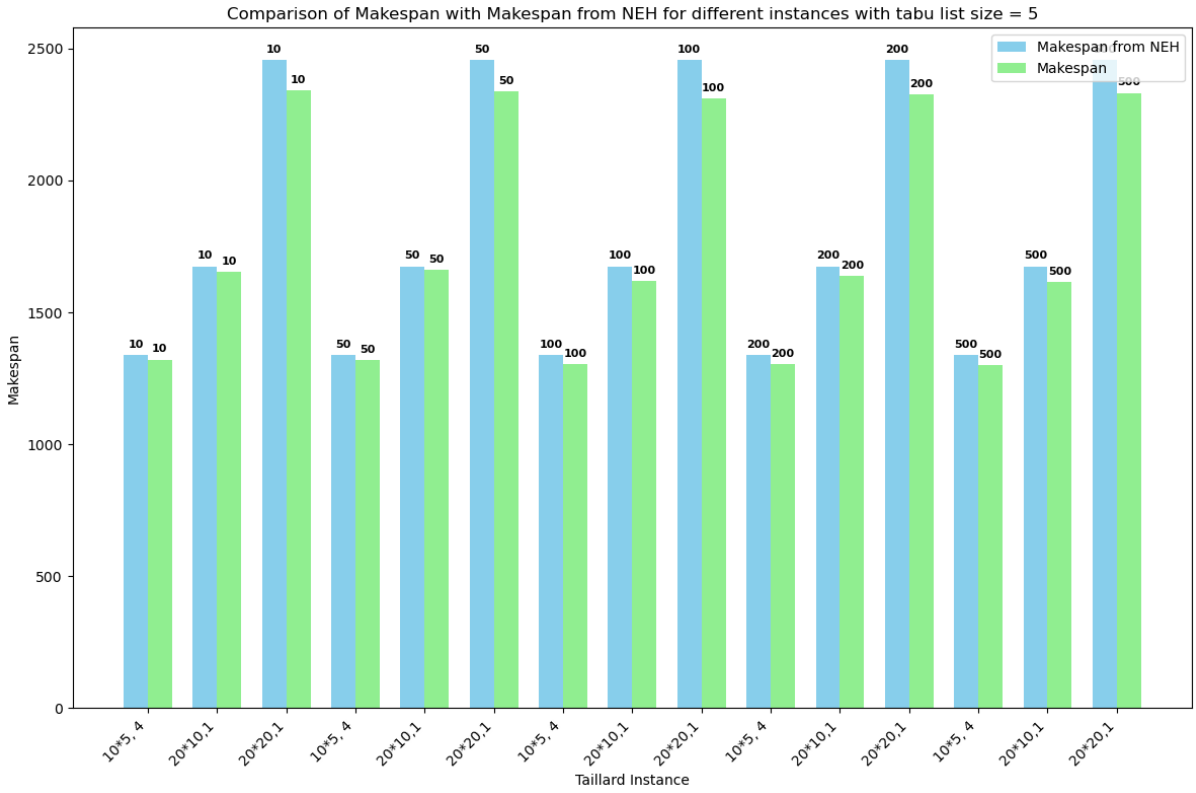

Figure 7: Makespan for Tabu List Size = 5

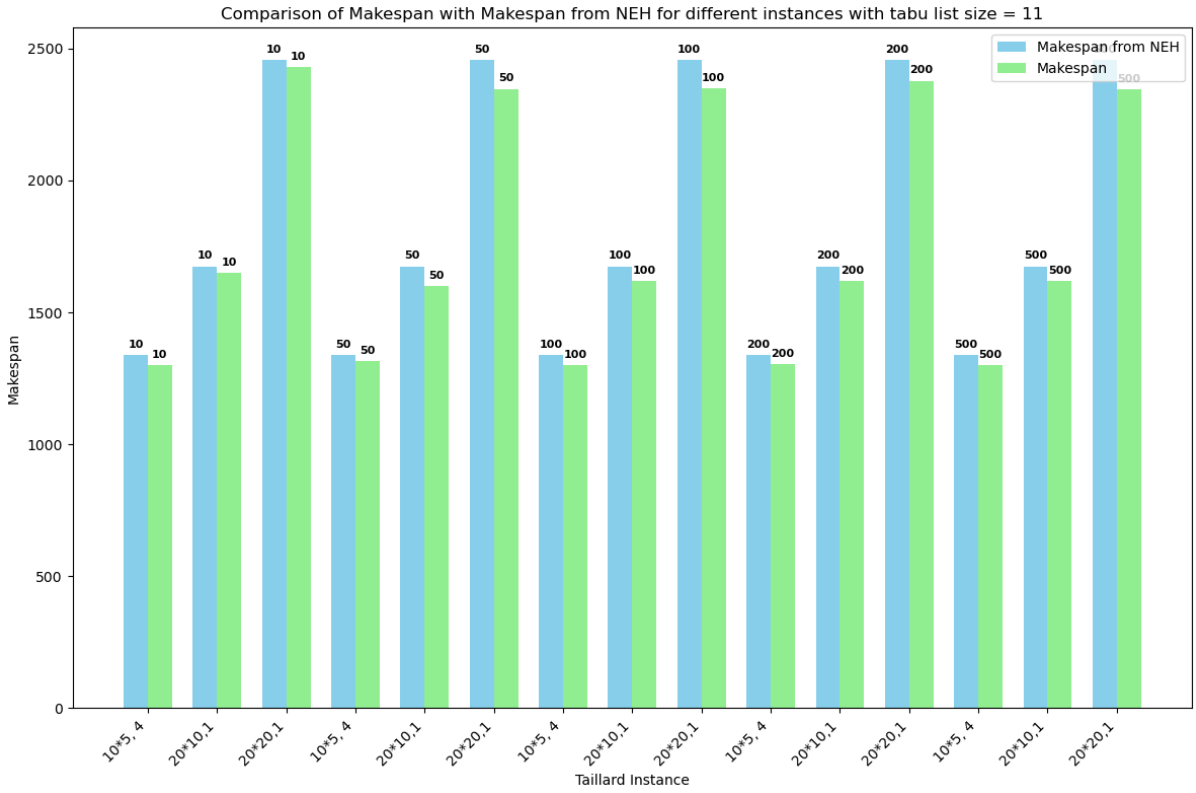

Figure 8: Makespan for Tabu List Size = 11

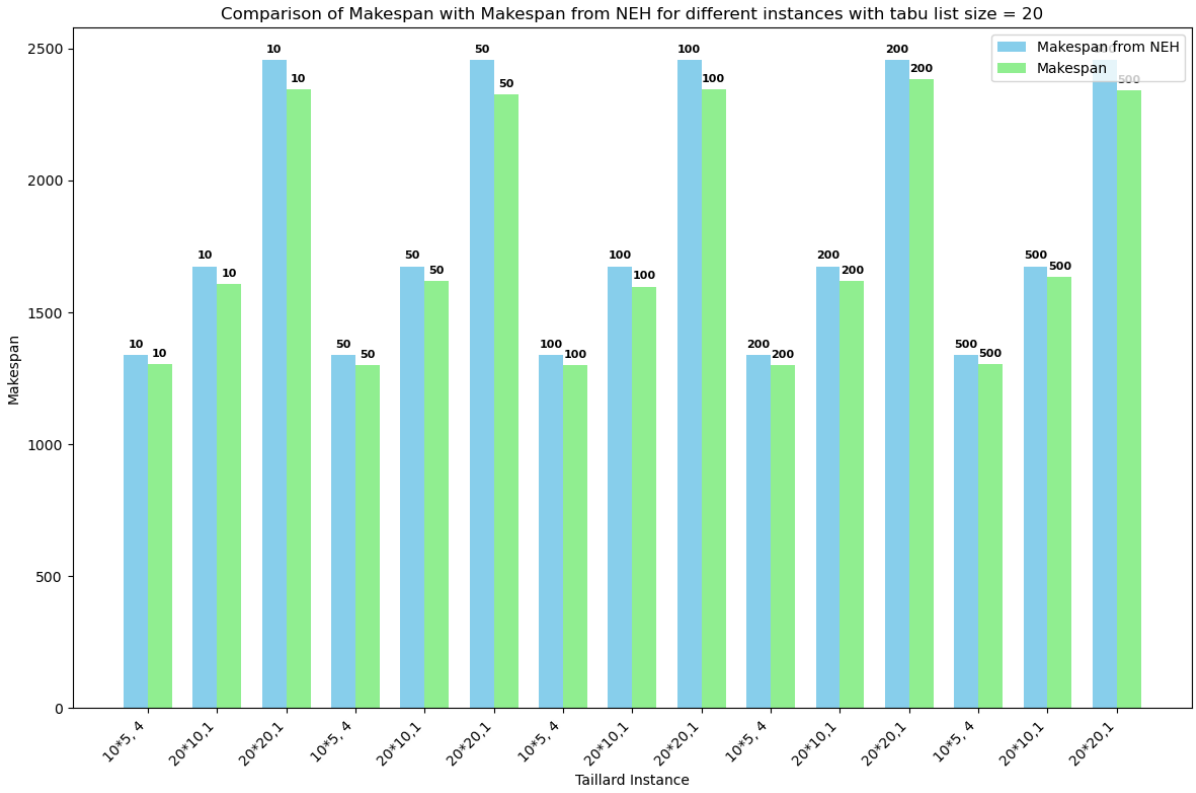

Figure 9: Makespan for Tabu List Size = 20

- We proceeded similarly to determine the values of the other parameters of tabu search algorithm, and the results obtained will be used in the Genetic Algorithm.

Parameter	**Options**
Function for generating neighbors	Swap, insertion of element, insertion of block, or random
Maximum iterations in stagnation	10 values between 100 and 2000 iterations
Tabu List Size	21 values between 50 and 2000 iterations
Initialization function	Random, NEH, CDS, Palmer, Gupta, PRSKE, and HAM

Table 3: Parameters of the Tabu Search

Phase 2: Tuning Genetic Algorithm Hyperparameters (High Level)

To refine the Genetic algorithm parameters, we systematically tested various combinations. For instance, to determine the optimal selection method, we varied this parameter and calculate the CPU Execution Time and The derivation. The results of these experiments are illustrated in the figures below.

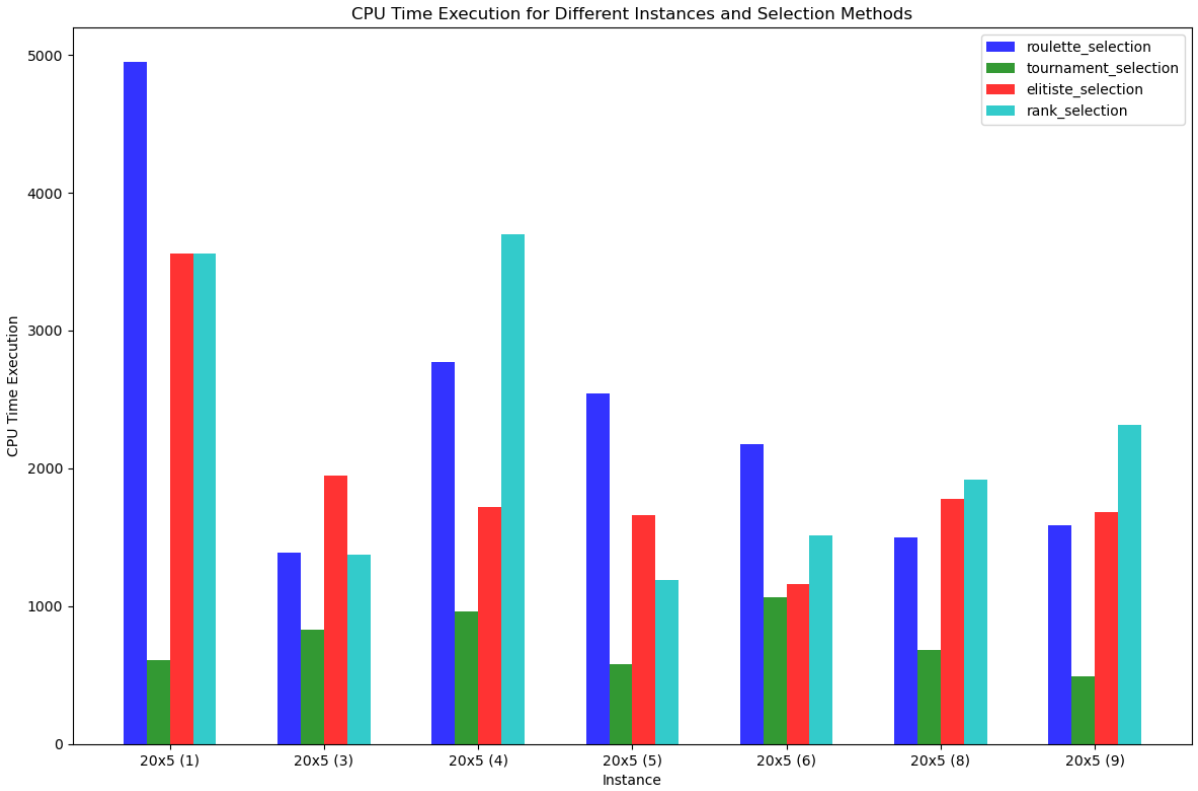

Figure 10: CPU Time Execution with the variation of Selection Methods for different Instances

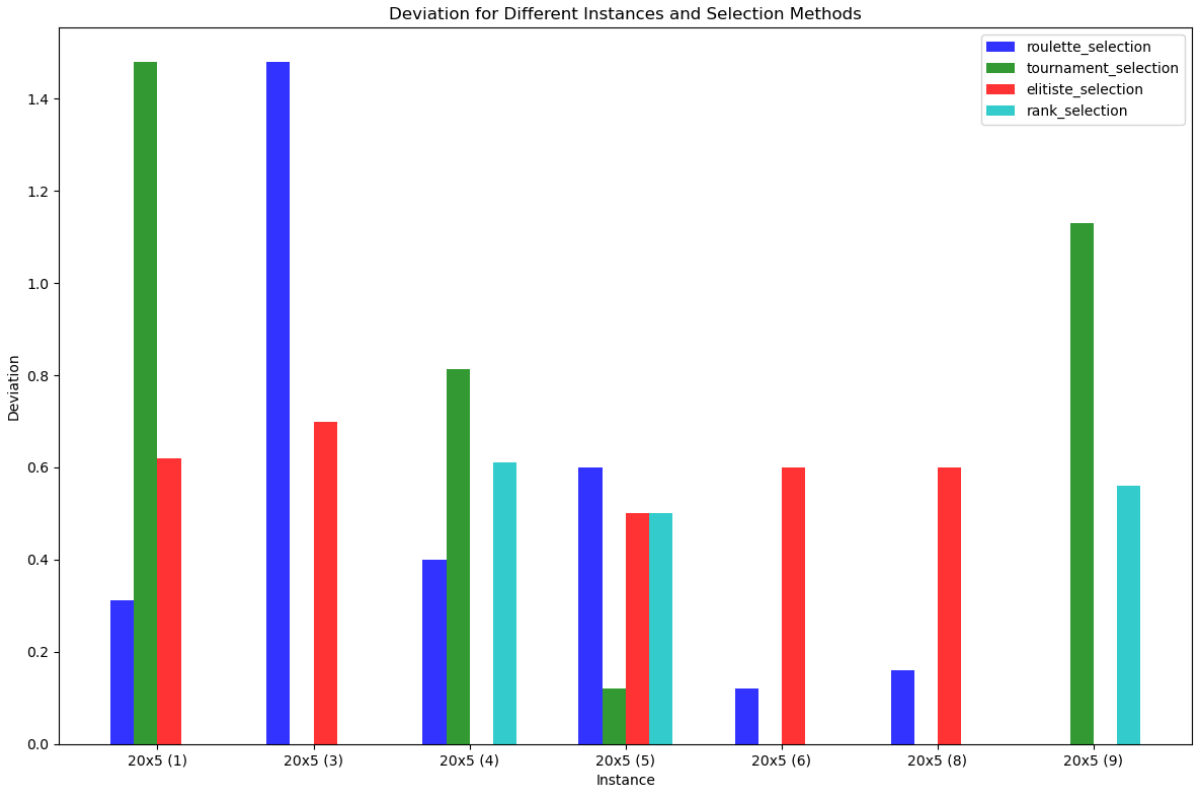

Figure 11: Deviation with the variation of Selection Methods for different Instances

We have chosen the tournament selection method because it consistently provides the best results

compared to other selection methods. Additionally, it achieves these results in a shorter time frame, making it more efficient for our purposes. - We proceeded similarly to determine the values of the other parameters of Genetic algorithm, and the results are represented in the table below.

Parameter	**Values**
Population size (Npop)	10-20
Number of generations	10-20
Selection	Tournament
Crossover	Unimodal
Mutation	Parameter perturbation
Crossover Rate (PC)	0.7-0.9
Mutation Rate (PM)	0.5
Replacement	Combined replacement

Table 4: Parameters of the high-level genetic algorithm

5.3 Performance Study and Result Analysis

In this section, we present the results of our computational experiments designed to evaluate the efficiency and effectiveness of TGA for solving the Permutation Flow Shop Scheduling Problem (PFSP). We compare our approach to other state-of-the-art methods using the well-known Taillard benchmark instances and we also compare the results of our approach in different instances. Our analysis focuses on these primary metrics: deviation from the best-known solutions and CPU execution time.

Performance Analysis of TGA per Instance and Benchmark

Using the parameter values derived from our calibration process, we tested instances from various benchmark sizes, including small, medium, and large instances. Our tests applied the TGA on these diverse instances, and some of these results are summarized in the following table. The detailed performance metrics, including makespan and deviation from the best-known solutions, showcase the effectiveness and scalability of our approach across different problem scales. We have categorized the instances as follows:

- **Small Instances**: $n = 20$ jobs with $m = 5, 10$ machines.
- **Medium Instances**: $n = 50$ jobs with $m = 5$ machines.
- **Large Instances**: $n = 100$ jobs with $m = 5$ machines.

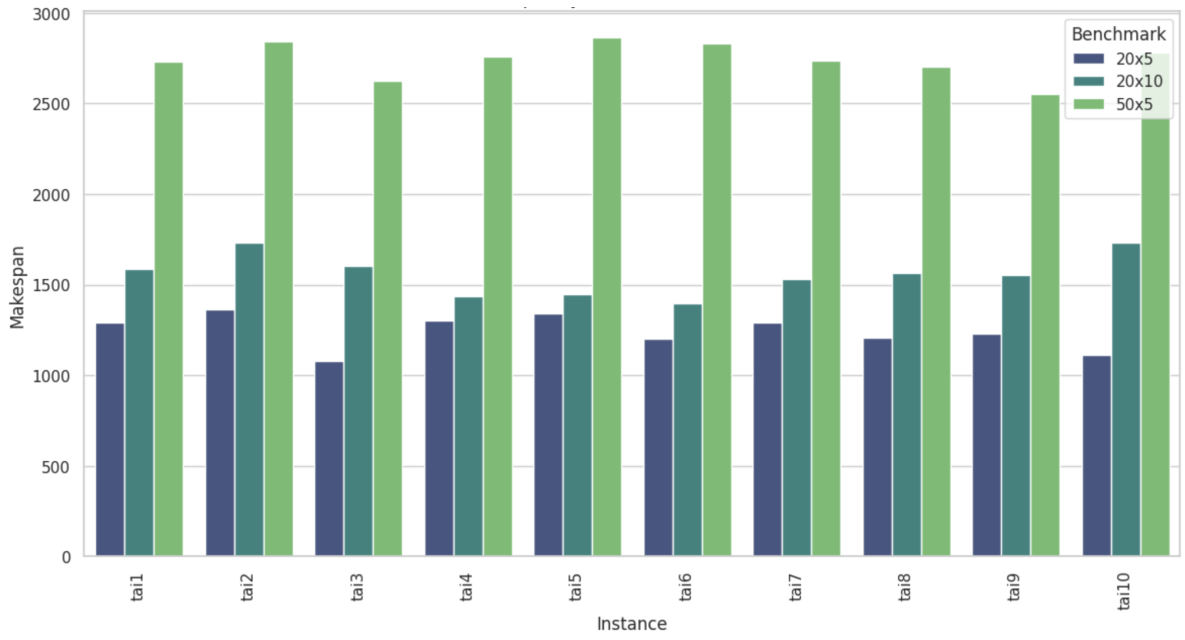

Figure 12: Makespan Value for different instances with the proposed Generative hyperheuristic

Benchmark	Instance	Makespan	Execution Time (s)	Deviation
20x5	(1)	1290	498.0	0.93
20x5	(2)	1366	454.0	0.51
20x5	(3)	1081	454.0	0.0
20x5	(4)	1300	659.0	0.54
20x5	(5)	1342.0	430.0	0.08
20x5	(6)	1201	734.0	0.5
20x5	(7)	1293	642.0	0.0
20x5	(8)	1206	401.0	0.0
20x5	(9)	1232	633.0	0.16
20x5	(10)	1110	523.0	0.18
20x10	(1)	1586	3146	0.25
20x10	(5)	1448	1453	2.041
20x10	(6)	1398	2874	0.071
20x10	(8)	1567	2231	1.88
20x10	(9)	1553	1700	0.06
50x5	(1)	2729.0	9752.26	0.18
50x5	(2)	2843.0	8525.58	0.31
50x5	(3)	2622.0	8543.34	0.038
50x5	(4)	2759.0	7852.64	0.29
50x5	(5)	2864.0	8522.15	0.03
50x5	(6)	2833.0	8542.13	0.14
50x5	(7)	2734	8541.03	0.33
50x5	(8)	2705.0	7521.15	0.11
50x5	(9)	2554.0	6452.77	0.078
50x5	(10)	2782.0	8450.22	0.0
100x5	(2)	5403.0	28865.35	0.02

Table 5: Benchmark Instances with Makespan, Execution Time, and Deviation with the Novel generative hyperheuristic

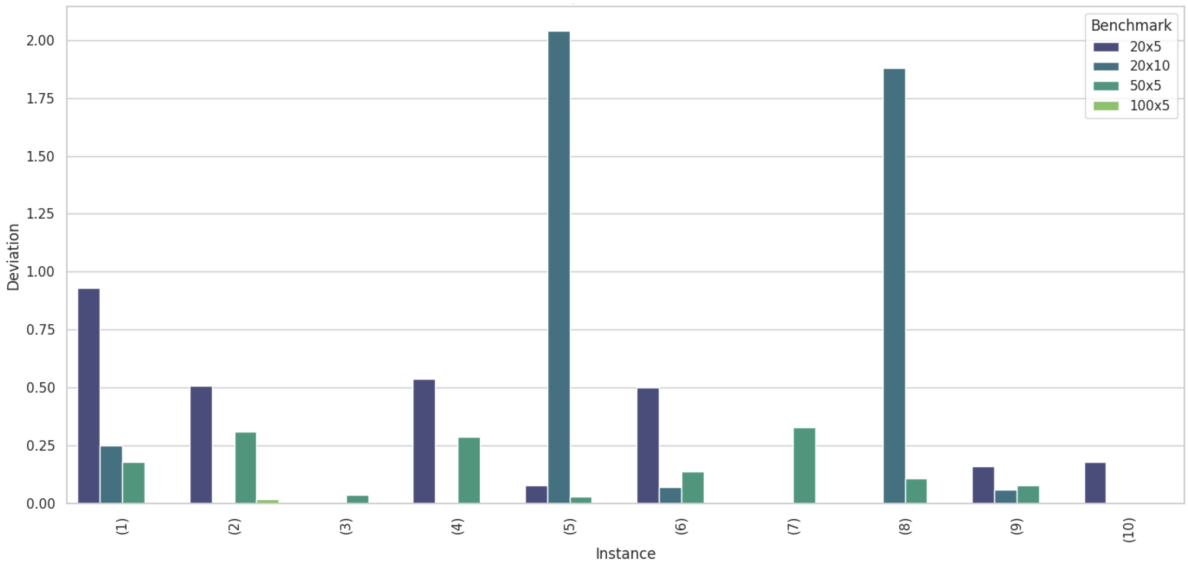

Figure 13: Deviation Value for different instances with the proposed Generative hyperheuristic

Based on the execution table, we observe that out of the 9 instances analyzed, we achieved the exact best known value or values extremely close to it in these 3 instances dor example: 20x5_3, 50x5_10, and 20x10_6. For the remaining instances, although we obtained values that are acceptably close to the best known value of $C_{\max}$, there are a few instances where the deviation is higher. Notably, instance 20x10_5 stands out with a deviation of 2.04, which is the highest among all instances analyzed however

this does not prove that the TGA is not efficient but maybe it has only reached the second best solution for this instance which is a little bit from the best known solution.

This highlights both successes and areas for improvement in our optimization approach. Further analysis of instances with higher deviations could uncover insights and patterns related to how the algorithm operates.

We can also observe that the execution times align reasonably with expectations, given the nature of running multiple tabu search algorithms to evaluate and select the best solutions for each instance. Naturally, this process consumes time. However, it becomes evident that execution times increase significantly as the instances grow larger in size. This trend underscores the computational complexity associated with tackling larger-scale instances, where the search space expands, necessitating more extensive computational resources and time to explore and optimize solutions effectively and this could be done for example through including parallelism in our solution and optimizing for example the execution of the different tabu instances when eveluated.

We can conclude that our algorithm demonstrates relatively good performance on several instances. However, its performance could potentially improve on other instances through multiple executions, leveraging the algorithm's inherent randomness. This approach allows for exploring diverse solutions and potentially discovering better outcomes across different runs, thereby enhancing overall performance and robustness in solving the problem instances.

Performance Analysis of TGA comparing to other state of the art methods

Our proposed hyper-heuristic approach has been rigorously compared with several state-of-the-art methods, including the Branch and Bound algorithm, Tabu Search, Genetic Algorithm, and GRASP algorithm. These comparisons were conducted to evaluate the efficiency, scalability, and overall performance of our approach across various instances of the Permutational Flowshop Problem (PFSP) and we have compiled these results in the various figures below:

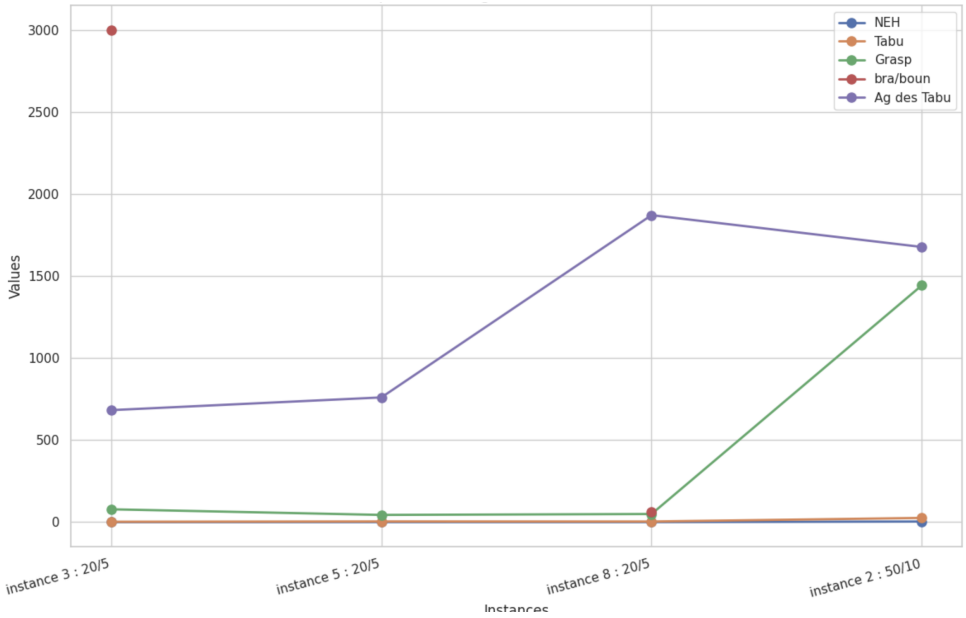

Figure 14: comparison of the total execution time of the TGA algorithm with other state of the art algorithms.

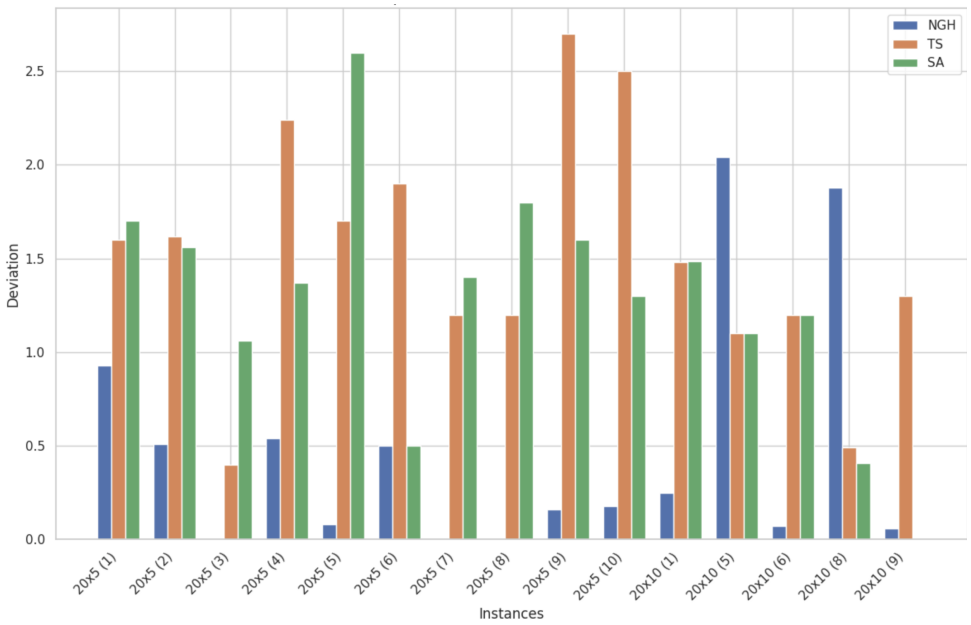

Figure 15: comparison of deviation of the TGA algorithm with TS and SA on multiple instances.

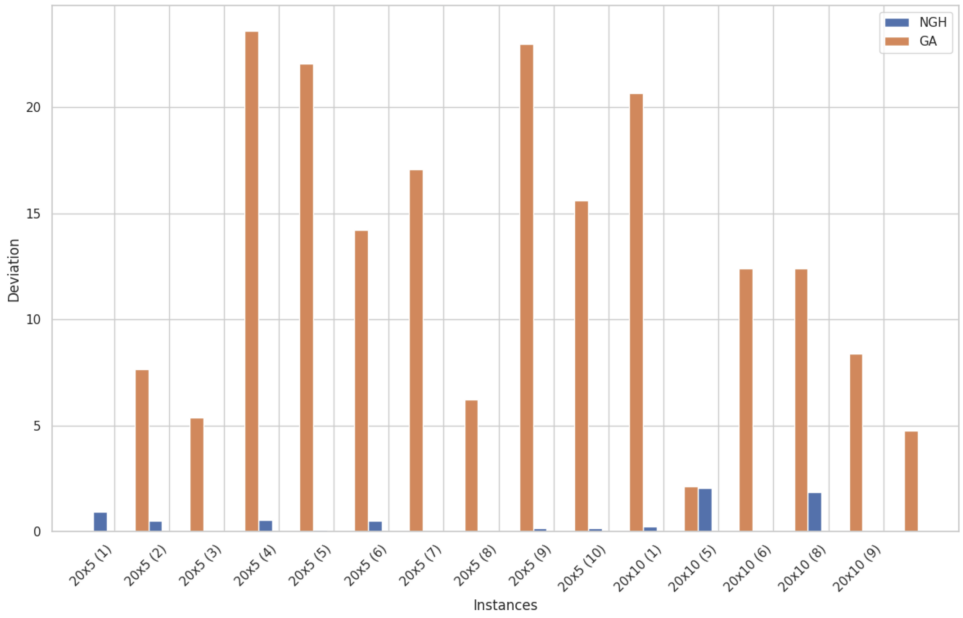

Figure 16: comparison of deviation of the TGA algorithm with AG on multiple instances.

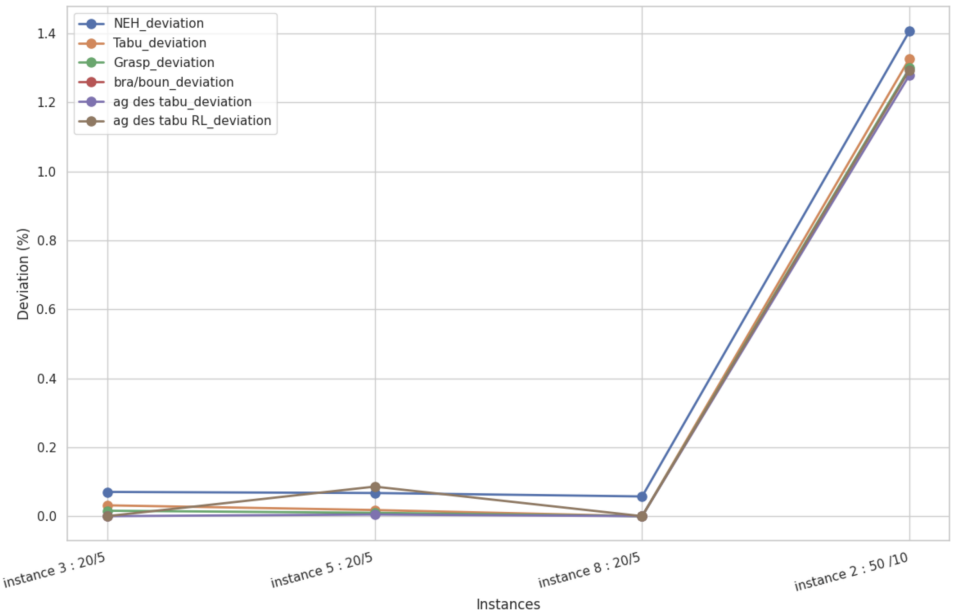

Figure 17: comparison of deviation of the TGA algorithm with other basic methodes on multiple instances.

From the results obtained, our algorithm (NGH/TGA) demonstrates superior performance across multiple instances compared to Tabu Search (TS) and Simulated Annealing(SA) but gives far better results than the GA algorithm in all of the instances that we have used above and therefore our algorihtm can compete with state of the art algorithms in terms of effectiveness.

6 Conclusion

In this paper, we have presented a novel generative hyper-heuristic approach (TGA) that integrates Genetic Algorithm (GA) with Tabu Search (TS) and Reinforcement Learning (RL) to effectively solve the Permutational Flowshop Problem (PFSP). Our three-layered architecture leverages the strengths of each component to provide a robust and adaptive solution framework. The Reinforcement Learning Layer initializes the population of the high-level genetic algorithm, which evolves low-level algorithm configurations to optimize performance. The low-level layer, comprising various Tabu Search instances, is tailored to address specific problem instances through carefully calibrated parameters and operators.

Our empirical study, conducted on benchmark datasets introduced by Taillard, demonstrates the effectiveness of our hyper-heuristic in finding the best known objective function values for multiple instances. This confirms the robustness and precision of our approach in solving PFSP instances of varying complexities. Furthermore, our proposed method has proven itself to rival other state-of-the-art algorithms, such as simple GA, standalone Tabu Search, and other advanced heuristic methods. This highlights the competitive edge and reliability of our integrated approach.

However, despite its effectiveness, the approach faces challenges in terms of computational efficiency. Execution times expand significantly, especially for larger instances, even with the reinforcement learning-based initialization. Addressing the computational scalability remains a critical challenge. Future work should focus on parallelizing the execution and optimizing the algorithm to enhance its computational efficiency. Introducing parallelization could dramatically improve the scalability of the algorithm, making it feasible for larger and more complex instances.

References

[1] Sarra Zohra Ahmed Bacha, Mohamed Walid Belahdji, Karima Benatchba, and Fatima Benbouzid-Si Tayeb. A new hyper-heuristic to generate effective instance ga for the permutation flow shop problem. *Laboratoire LMCS, Ecole nationale Sup erieure d'Informatique (ESI), BP 68M, Oued Smar , 16000 Alger, Alg erie*, 2017.

[2] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches. In *Handbook of metaheuristics*, pages 449–468. Springer, Boston, MA, 2010.

[3] D. G. Dannenbring. An evaluation of flow shop sequencing heuristics. *Management Science*, 23(11):1174–1182, 1977.

[4] Marco Dorigo and Thomas Stützle. *Ant colony optimization*. MIT press, 2006.

[5] Stefan Droste and Others. Bandit-based hyper-heuristic selection for resource-constrained environments. *Applied Soft Computing*, 2021.

[6] Ta Feo and Mg Resende. A greedy randomized adaptive search procedure for the quadratic assignment problem. *Journal of global optimization*, 6(2):109–133, 1995.

[7] George Finke. An algorithm for the n-job, m-machine sequencing problem. *Operations Research*, 24(6):1141–1154, 1976.

[8] Fred Glover. Future paths for integer programming and links to artificial intelligence. *computers & operations research*, 13(5):533–549, 1986.

[9] David E Goldberg. Genetic algorithms in search, optimization, and machine learning. *Addison-Wesley*, 1989.

[10] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, volume 1, pages 69–93. Elsevier, 1991.

[11] J. N. D. Gupta. A functional heuristic algorithm for the flowshop scheduling problem. *Operational Research Quarterly*, 22:39–47, 1971.

[12] S. M. Johnson. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly*, 1(1):61–68, 1954.

[13] James Kennedy and Russell C Eberhart. *Particle swarm optimization*. IEEE, 1995.

[14] S Kirkpatrick, CD Gelatt Jr, and MP Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[15] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992.

[16] Kumar. Particle swarm optimization-based generation hyper-heuristic for job scheduling problems. *Journal of Scheduling*, 2010.

[17] Hr Lourenço, O Martin, and T Stützle. An iterated greedy algorithm for the linear ordering problem. *INFORMS Journal on Computing*, 14(3):281–292, 2002.

[18] Mizraqi. Ant colony optimization-based generation hyper-heuristic for the knapsack problem. *Journal of Optimization Theory and Applications*, 2018.

[19] Muhammad Nawaz, Eugene Emory Enscore Jr, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *OMEGA*, 11(1):91–95, 1983.

[20] C. E. Nugraheni and L. Abednego. A new hyper-heuristic to generate effective instance ga for the permutation flow shop problem. *International Journal of Modeling and Optimization*, 6(5), 2016.

[21] D Palmer. Sequencing jobs through a multi-stage process in the minimum total time—a quick method of obtaining a near optimum. *Operations research*, 13(1):93–95, 1965.

[22] Quan-Ke Pan and Yong-Mei Liu. A new branch-and-bound algorithm for the permutation flowshop scheduling problem. *Management Science*, 35(3):1067–1076, 1989.

[23] C. Rajendran and D. Chaudhuri. An efficient heuristic approach to the scheduling of jobs in a flowshop. *European Journal of Operational Research*, 61:318–325, 1992.

[24] Éric D. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

[25] J.N. Tsitsiklis. Asynchronous stochastic approximation and q-learning. In *Proceedings of 32nd IEEE Conference on Decision and Control*, pages 395–400. IEEE, 1993.

[26] C.J.C.H. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.