uc3m | Universidad Carlos III de Madrid

Master in Connected Industry 4.0
2024-2025

Master's Thesis

Reinforcement Learning in Smart Manufacturing systems: A case study in Flexible Job Shop Scheduling

Richard Daniel Cardenas Rondan

Tutor

Mario Muñoz Organero

Madrid, September 2024

ABSTRACT

The Flexible Job Shop Scheduling Problem (FJSSP) is a central challenge in smart manufacturing within Industry 4.0, where flexibility and scale make exact methods impractical on realistic instances. This thesis studies a Reinforcement Learning Genetic Algorithm (RLGA) that uses Q-learning to adapt genetic operators during the search, improving guidance without extensive manual tuning. We compare RLGA with three established metaheuristics: Genetic Algorithm (GA), Simulated Annealing (SA), and Tabu Search (TS) on standard FJSSP benchmarks. Across small, medium, and large instances, RLGA achieves top or near top makespan values, matching strong baselines on easier cases and improving median solution quality on the hardest sets. It maintains consistent performance across heterogeneous configurations, indicating superior scalability and robustness relative to the other methods.

Keywords: Flexible Job Shop Scheduling, Smart Manufacturing, Reinforcement Learning, Q-learning, Genetic Algorithm, Simulated Annealing, Tabu Search, RLGA.

INDEX OF CONTENT

INDEX OF FIGURES

INDEX OF TABLES

ABBREVIATIONS AND ACRONYMS

FJSSP	Flexible Job Shop Scheduling Problem
FJSP	Flexible Job Shop Problem
RL	Reinforcement Learning
GA	Genetic Algorithm
SA	Simulated Annealing
TS	Tabu Search
RLGA	Reinforcement Learning Genetic Algorithm
MDP	Markov Decision Process
SARSA	State-action-reward-state-action
DRL	Deep Reinforcement Learning
$\boldsymbol{C_{max}}$	Makespan
SR	Success Rate
GAP	Solution quality gap
K	Patience stop (no-improvement stop counter)
E	Evaluation stop (hit evaluation budget)
EV	evaluations
CPU (ms)	Computational time (milliseconds)
Lit. Best	Literature best value
Med.	Median

1. INTRODUCTION

Smart manufacturing, a key part of Industry 4.0, is a highly connected and adaptable way of production. It uses technologies like cyber-physical systems, Industrial Internet of Things and cloud computing to link machines, sensors and information systems. This allows monitoring conditions, data sharing, and coordinate actions in real-time. All of this interconnectivity leads to production faster response times, more customized products, and constant improvement through the use of digital models and data analysis [1].

Production scheduling in smart manufacturing is the process of assigning operations to eligible machines and sequencing them over time so that a connected factory meets cost, quality and delivery goals in an efficient manner [2].

1.1. Problem Context

Scheduling in production systems is a field with well-defined problem classes. Classical problem families include the single-machine problem, where all operations compete for one resource, the flow shop, where jobs visit machines in the same order, and the job shop, where each job follows its own operation sequence over shared machines.

The Flexible Job Shop Scheduling Problem (FJSSP) extends the job shop by allowing each operation to be processed on one machine selected from an eligible set. This flexibility better reflects modern cells with parallel or substitutable equipment [2], [3].

In the context of smart factories machines can handle many types of machining operation with different processing times, or have multiple machines with multiple configurations. There is no method that can always find an optimal schedule with solution candidates that grow polynomially with instance size, which means that FJSSP is NP-hard[3].

When problem is NP-hard exact methods scale poorly on realistic instances. Therefore, high-performing solvers are preferred such as Simulated Annealing (SA), Tabu Search (TS), and Genetic Algorithm (GA) that balance exploration and exploitation under determined computational budgets[4], [5].

Reinforcement learning (RL) provides a complementary capability by learning policies that guide the search. In a hybrid approach, Q-Learning can estimate the value of scheduling decisions or choose operators and parameters for a metaheuristic. This learning-to-search view aims to improve solution quality and consistency, especially under varying instance characteristics and limited time[6].

Recent work like in [7], [8] shows that reinforcement learning can act as an online controller for metaheuristics, yielding smarter search dynamics in hard combinatorial settings. In particular, a genetic algorithm based on reinforcement learning can use Q-learning to self-learn crossover probabilities, adapting operator usage to what is most effective at each stage of the search. Onwards, this work focuses on testing RLGA on small, medium, and large scheduling instances, evaluating performance and growth potential.

1.2. Objective

The main goal of this thesis is to test Reinforcement Learning in combination with a specific metaheuristic algorithm, Genetic Algorithm, and test its performance in flexible job shop scheduling problem instances. To achieve this, the work will focus on the following specific goals:

- Define a set of FJSSP benchmark instances and describe their key features like size, machines, jobs, flexibility.
- Implement classic Genetic Algorithm (GA), Simulated Annealing (SA), and Tabu Search (TS) solvers for the defined instances.
- Implement a Reinforcement Learning Genetic Algorithm (RLGA) with capabilities to perform in the same environment as metaheuristic algorithms.
- Create one fair test setup for all methods with the same stopping rules and metrics.
- Measure how well RLGA keeps quality and speed across small, medium, and large cases in contrast to other metaheuristic algorithms. Compare RLGA with classic GA performance as size of instances grow.

1.3. Scope

The following study uses simulated FJSSP data for all experiments, no live factory data or system integration. The RL environment focuses in online evaluations only, not pretrained Q-learning controller.

In terms of scheduling dataset to be used, it is a deterministic problem setting, disturbances are out of this scope. Finally, this data to be studied provides enough information so the FJSSP objective focuses on optimization of makespan.

1.4. Thesis Structure

This thesis is organized into seven chapters as described below:

- Chapter 1 – Introduction: Presents the motivation, problem, main objectives, scope, and overall context of the work.
- Chapter 2 – Literature Review: Provides an overview of scheduling, introduces Flexible Job Shop Scheduling Problems (FJSSP), traditional scheduling techniques, and how reinforcement learning is applied to scheduling.
- Chapter 3 – Problem Definition: Defines the structure of the FJSSP addressed in this work, including the dataset, scheduling constraints, objective function, and assumptions made in the modeling.
- Chapter 4 – Methodology: Describes the approach used to solve the scheduling problem, detailing classic heuristic methods to be implemented and Reinforcement learning method. Evaluation metrics are defined
- Chapter 5 – Results and Discussion: Presents the experimental results, analyzes the performance of all models. Contrast RLGA with other algorithms.
- Chapter 6 – Conclusions and Future Work: Summarizes the findings of the research, discusses its contributions, and suggests potential directions for future studies.

2. LITERATURE REVIEW

2.1. Scheduling

For manufacturing and production purposes, scheduling aims to optimally assign jobs to a single or groups of machines while following manufacturing rules as well as job specifications[9]. Three definitions define scheduling:

- Machine – A resource capable of processing operations, each machine can process only one operation at a time.
- Job – A complete manufacturing task or order that consists of a sequence of operations. Each job has its own processing route, which defines the order in which its operations must be completed.
- Operation – A single processing step within a job. Each operation must be executed on one eligible machine for a specific processing time and must respect precedence constraints from other operations in the same job.

Scheduling problems come in different machine and operation environments, some rules are added or removed in each environment, which are described in Table 1[2], [10].

TABLE 1. SCHEDULLING MACHINE ENVIRONMENTS

Environment	Machines available	Job routing constraint	Operation–machine eligibility
Single Machine	One machine	All jobs compete for the same machine	Every operation must run on the single machine
Flow Shop	Multiple machines in fixed stage order	All jobs visit machines in the same order	Each stage has one machine; no choice per operation
Flexible Flow Shop	Multiple stages, each with parallel machines	All jobs visit stages in the same order	At each stage, choose one machine from that stage's pool
Job Shop	Multiple machines	Each job has its own fixed route across machines	Each operation is assigned to a specific machine only
Flexible Job Shop	Multiple machines	Each job has its own route; order is fixed per job	Each operation has an eligible set; choose one machine
Open Shop	Multiple machines	Each job must be processed on all machines; any order	One operation per machine; order is decided by the scheduler

2.2. Flexible Job Shop Scheduling

Starting with the basics, a Job Shop as described in [2] is a multi-operation production model consisting of multiple machines, each dedicated to a specific type of operation. The system is capable of handling multiple jobs simultaneously, where each job follows a fixed and predefined sequence of machines.

A Flexible Job Shop is generalized version of the classical job shop model [2]. In this model, each operation of a job can be assigned to one of several parallel machines, meaning that different machines are capable of performing the same type of operation. While the sequence of operations for each job remains fixed, the availability of parallel machine options allows dynamic routing decisions.

Furthermore, there is are two situations when evaluating Flexible Job shop, which is the degree of flexibility, sometimes denoted as partial flexibility and total flexibility [11]. On one hand, partial flexibility implies that an operation within a job can be performed in more than one machine, on the total flexibility case is assumed that any operation of any job can be held at any machine.

For instance, in the following Table 2, a scheduling problem that involves 3 machines, 3 jobs and 3 operations are presented for a Job shop problem, a partial flexibility FJSSP, and a total Flexibility FJSSP, the intersection between machine and operation is the processing time of the operation in the designated machine, all values presented are random.

TABLE 2. COMPLEXITY OF FLEXIBLE JOB SHOP SCHEDULLING

ENVIRONMENT	**JOB**	**OPERATION**	**MACHINES**		
			M1	M2	M3
JOB SHOP	JOB 1	OP1-1	3	∞	∞
		OP1-2	∞	7	∞
		OP1-3	∞	∞	9
	JOB 2	OP2-1	∞	1	∞
		OP2-2	4	∞	∞
		OP2-3	∞	∞	1
	JOB 2	OP3-1	∞	∞	1
		OP3-2	3	∞	∞
		OP3-3	∞	2	∞
FLEXIBLE JOB SHOP (PARTIAL FLEXIBILITY)	JOB 1	OP1-1	3	2	∞
		OP1-2	4	7	5
		OP1-3	∞	∞	9
	JOB 2	OP2-1	∞	1	∞
		OP2-2	4	∞	∞
		OP2-3	∞	2	1
	JOB 2	OP3-1	∞	3	1
		OP3-2	3	∞	∞
		OP3-3	2	2	∞
FLEXIBLE JOB SHOP (TOTAL FLEXIBILITY)	JOB 1	OP1-1	3	2	3
		OP1-2	4	7	5
		OP1-3	8	7	9
	JOB 2	OP2-1	2	1	2
		OP2-2	4	4	2
		OP2-3	3	2	1
	JOB 2	OP3-1	2	3	1
		OP3-2	3	3	3
		OP3-3	2	2	1

2.3. Heuristic solution approaches

As reviewed in [2], [12], heuristics are practical strategies that aim to find good solutions in a reasonable amount of time, but not guaranteeing the optimal one. The most basic form of heuristics used in scheduling are dispatching rules, these “rules” are particularly useful when the goal is to construct a reasonably effective schedule with respect to a single performance objective, such as minimizing the makespan, the total completion time, or the maximum lateness.

Based on literature found in [2], table 3 presents a set of widely used dispatching rules for job shop scheduling are presented. Each rule provides a simple decision-making strategy for selecting the next job or operation to schedule based on specific criteria, such as processing time, due dates, machine availability, or flexibility.

TABLE 3. COMMON DISPATCHING RULES USED IN JOB SHOP SCHEDULING

Rule	Full Name	Description
SIRO	Simple Random Order	Selects a job randomly among all available ones. Used when no prioritization is applied.
ERD	Earliest Release Date	Selects the job with the earliest release time, giving priority to the job that becomes available first.
EDD	Earliest Due Date	Prioritizes the job with the earliest due date, aiming to reduce lateness or due date violations.
MS	Minimum Slack	Chooses the job with the least slack time, calculated as due date minus processing time.
SPT	Shortest Processing Time	Selects the job or operation with the shortest processing time to improve machine utilization and reduce queues.
WSPT	Weighted Shortest Processing Time	Selects the job with the lowest weighted processing time ratio.
LPT	Longest Processing Time	Prioritizes the job with the longest processing time, useful in load-balancing scenarios.
CP	Critical Path	Prioritizes operations that lie on the longest path of the job graph to avoid schedule delays.
LNS	Least Number of Successors	Selects the job with the fewest remaining operations to complete.
SST	Shortest Setup Time	Chooses the job that requires the least setup time on the current machine.
LFJ	Least Flexible Job	Prioritizes the job that can be processed by the fewest machines, preserving flexibility for other jobs.

2.4. Metaheuristic solution approaches

Researchers have explored ways to improve the basic heuristics methods stated before. Metaheuristic search methods search smartly instead of checking every option. They work with practical limits like machine rules and due dates. Metaheuristic algorithms can be stochastic, which means the use random decisions to escape local traps, while some can be deterministic and pick a choice based on certain results[6], [10].

A Genetic Algorithm (GA) treats each schedule as an individual in a population, this algorithm is stochastic. It starts with a random population, uses random crossover points, and applies random mutations. These random steps add variety and help explore many schedule patterns.[2], [8], [13], [14].

Simulated Annealing (SA) is an stochastic algorithm that starts with one individual, the schedule solution candidate for this study, and makes small moves, like swapping machines or operation order. It always accepts better moves and sometimes accepts worse ones to escape traps. The chance of accepting worse moves goes down as the temperature cools. Careful cooling and stopping rules matter. [2], [3], [4], [5].

Tabu Search (TS) is often compared to Simulated Annealing because both make small local moves, but TS is usually deterministic: from a fixed start and fixed tie-breaking, the tabu list and aspiration rules define a single path. It picks the best allowed move at each step, even if it is temporarily worse, and can override the tabu if the move beats the best-so-far. Memory rules drive focused search around good areas and broader exploration elsewhere. Randomness is optional and help TS escape patterns and explore new regions, while keeping a fixed seed makes the run repeatable. [2], [5], [15], [16].

The following Table 4 summarize the characteristic of each algorithm. One main characteristic to highlight is the step for each algorithm. In a step of the algorithm, SA only picks one candidate solution and evaluates it. GA evaluates a population and that is a generation for them, TS evaluates a full neighborhood each iteration. In conclusion each algorithm has its own pace and methodology but they all search for an optimum solution.

TABLE 4. CHARACTERISTIC OF METAHEURISTIC ALGORITHMS

Aspect	Genetic Algorithm (GA)	Simulated Annealing (SA)	Tabu Search (TS)
Search unit per algorithm step	Population of many solutions	Single solution	Single solution, but evaluates a neighborhood each iteration
Per-iteration evaluation set	All individuals in the population	One random neighbor	Full neighborhood
Control step	Generation	Iteration	Iteration
Randomness	High: random init, crossover points, mutations	High: random neighbor choice and probabilistic acceptance	Low: deterministic given start and tie breaks; optional randomness in start
Acceptance rule	Selection keeps fitter individuals; elitism may preserve best	Always accept better; sometimes accept worse by $\exp(-\Delta/T)$	Choose best admissible move; tabu forbids recent moves; aspiration can override if it improves best-so-far
Cost per step	Higher per generation due to many fitness evaluations	Low per iteration	Moderate to high if scanning full neighborhood; lower with candidate list
Key parameters	Population size, crossover rate, mutation rate, selection pressure, elitism	Initial temperature, cooling rate, iterations per temperature, schedule	Tabu tenure, neighborhood size, aspiration rule

2.5. Reinforcement Learning in Scheduling

Reinforcement learning (RL) is a learning approach in which an agent interacts with an environment, selects actions, and learns a policy that maximizes cumulative reward in a Markov decision process (MDP). For scheduling, RL can learn dispatching policies directly from experience or simulation, adapt online to changes, and reduce hand-crafted rule design; deep RL helps when state spaces are large. Reported benefits include better global coordination across work centers, fast re-training for new objectives or product mixes, and improved solution quality or effort when RL guides metaheuristics such as Tabu Search.[6]

Deep RL is a variant that uses neural networks to approximate value or policy functions, which helps with large state spaces common in fabs and job shops, and it can be pre-trained from data then fine-tuned online in simulation.[10]

Learning workflows can be run offline, by training on similar instances and then deploying, or online, by interleaving data collection, training, and application within the same instance. This project takes an online perspective that opens the possibility to work with instances while learning in process.[17]

Q-learning is a reinforcement learning algorithm that, in the setting of a Markov decision process, learns the optimal action-value function from trial-and-error interaction without a model of the transitions. It evaluates greedy actions while exploring with a different behavior, which is useful for guiding search. In scheduling, Q-learning or its deep variants can pick dispatching rules, select neighborhoods or operators, and steer metaheuristics such by ranking moves, which reduces hand-tuned heuristics and improves adaptation.[16], [18]

Figure 1 shows online Q-learning coupled with a metaheuristic for the FJSSP inside an MDP[6], [16].

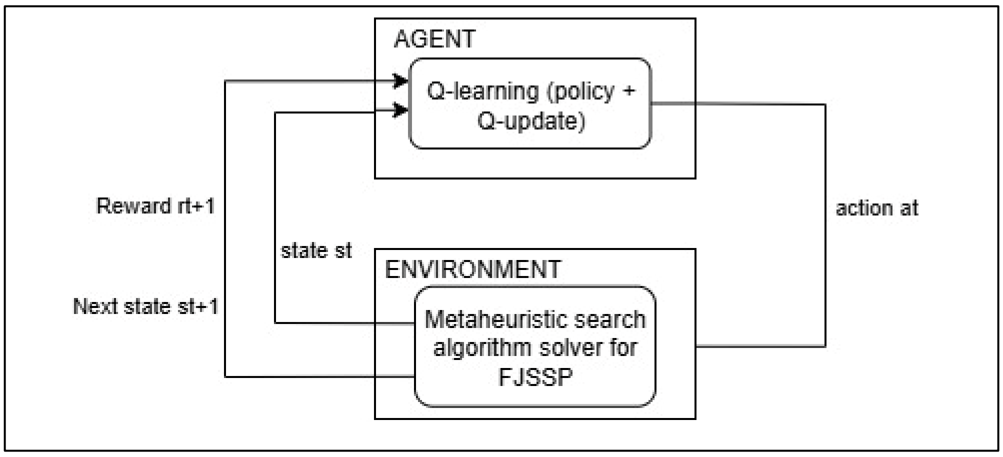

AGENT
Q-learning (policy + Q-update)
Reward rt+1
state st
action at
ENVIRONMENT
Next state st+1
Metaheuristic search algorithm solver for FJSSP

Figure 1. Q-Learning and Metaheuristic algorithms as part of a MDP

- The agent is the Q-learning controller that stores action values and picks an action with epsilon-greedy.
- Actions tell the metaheuristic what to do, such as select a neighborhood, choose a move, or adjust a search parameter.
- The environment is the FJSSP simulator plus the metaheuristic executor, from the current state it applies the action, builds a new schedule, and returns the next state and a reward.
- The reward reflects immediate progress, for example a drop in makespan or tardiness, with penalties for infeasible steps.

After each step the agent updates its values and repeats. This is online training because experience is generated and learned from during the running search. Q-learning is off-policy, so it estimates the value of greedy decisions while it still explores[8], [16], [18].

3. PROBLEM DEFINITION

In this chapter, the dataset used to model the FJSSP is introduced and described in detail. The structure of the dataset is explained, including the description of jobs, the number and order of operations for each job, the set of alternative machines capable of performing each operation, and the corresponding processing times. The dataset obtained from [4], and discussed in [4], [11], will be used in this and following chapters.

3.1. Description of Flexible Job Shop Scheduling Problem

The study case for this project is meant to be smart manufacturing environment simulation instances, therefore constraint of the problem will aim to simulate some smart factories characteristics. However, due to computational limitations and project scope some constraints won't reflect a real-time working environment.

The assumptions of the selected test problems are as follows:

- Machines are configurable, this is connected to the ideal that cyber-physical systems in Industry 4.0 enable machines adaptability.
- Jobs are independent, there is no prerequisite for a job to start or finish before any other. This is considered a limitation because in real factories, some job or products should be finished before the start of other or be done at same time.
- Operations must follow a sequence for job competition, this is a common rule in manufacturing where operations must be made following an exact order. Scheduling problems that do not follow this rule fall in "open shop problems".
- Processing times are provided when available, thus the objective for scheduling solvers is to obtain a feasible schedule with the lowest makespan possible. The makespan is the finish time of the last completed operation

Limitations, that differ this study over papers that study scheduling in general, with definitions found in [3], are discussed below:

- Time lag constraint, which is a waiting time between operation, occurs in real factories due to controlled or arbitrary factors. This waiting time is not considered due to its stochastic behavior not considered for this project.
- Availability constraint, when machines can randomly fail and become unavailable, not considered due to its stochastic behavior.
- Setup time constraints, defined as the time it takes a machine to be prepared to take the next operation, is a common characteristic in smart manufacturing, however to simplify this study, it is assumed to be included in the processing time.

3.2. Optimization Objective

The main optimization objective for this study is makespan. Makespan is the total time the schedule takes, from time zero until the very last operation of all job finishes. The variable used to define makespan is C_{max}. The objective is to minimize C_{max} but there is no exact algorithm to do so, that is why FJSSP are considered NP-hard.

Research have found various equations to define makespan in terms of machines, jobs and operations, these are stablished in [4], [7]. However, transferring these equations and comparisons to computer programming is more practical.

To make sure a candidate schedule solution is valid the following rules are followed:

- Each machine needs to be assigned to a valid operation of a valid job.
- The order of operations needs to follow a job precedence order.

After this schedule candidate demonstrates to be a valid solution, makespan calculation follow this step:

- Get the operation time for each Operation at designated machine.
- Because the order of operations received are valid, start moving operations to machines running sequence.
- Once a machine starts an operation, the next available operation for that machine waits for the previous to end. Each machine
- After all operations have been correctly assigned to their respective machine, search for machine with the highest duration of operation. That duration is C_{max}.

As for Flexible job shop scheduling problems, global minimum C_{max} is very difficult to reach and prove is the best solution, for that reason metaheuristics algorithms can help in searching for solutions.

3.3. Structure and Variables in the Dataset

For this FJSSP project, dataset from simulated environments is used as a reference, the following resource [19] is chosen for the project, it groups various datasets of different sizes from other authors.

The test problems selected for this study consist of 30 instances, divided in 3 groups of 10, each instance groups have completely different dimensions, the size and definition of each instance is defined in table 5.

- The first 10 instances consist of small scheduling problems with known optimum makespan.
- The second group of instances consist in 10 medium scheduling problem instances with not known optimum makespan but a range of possible outcomes have been evaluated by the dataset developer in [4] using a computer solver and a literature best value found in study papers.
- Last group of instances consist in 10 large size problems from another paper [20]. This has the characteristics of having more operations and jobs than previous instances. Known lower bound and optimum solution have been obtained from [11].

TABLE 5. DATASET INSTANCES FOR FJSSP STUDY

Problem no.	jobs	machines	operations (max)	FJSSP Flexibility	C_{max}
SFJS1	2	2	2	Total	66
SFJS2	2	2	2	Partial	107
SFJS3	3	2	2	Partial	221
SFJS4	3	2	2	Partial	355
SFJS5	3	2	2	Total	119
SFJS6	3	2	3	Partial	320
SFJS7	3	5	3	Total	397
SFJS8	3	4	3	Total	253
SFJS9	3	3	3	Total	210
SFJS10	4	5	3	Partial	516
MFJS1	5	6	3	Partial	(396, 470) 468
MFJS2	5	7	3	Partial	(396, 484) 446
MFJS3	6	7	3	Partial	(396, 564) 466
MFJS4	7	7	3	Partial	(496, 684) 554
MFJS5	7	7	3	Partial	(414, 696) 514
MFJS6	8	7	3	Partial	(469, 786) 635
MFJS7	8	7	4	Partial	(619, 1433) 879
MFJS8	9	8	4	Partial	(619, 1914) 884
MFJS9	11	8	4	Partial	(764, 2908) 1088
MFJS10	12	8	4	Partial	(944, 4960) 1267
MK01	10	6	7	Partial	40
MK02	10	6	7	Partial	(24, N/A) 27
MK03	15	8	10	Partial	204
MK04	15	8	10	Partial	60
MK05	15	4	10	Partial	(168, N/A) 174
MK06	10	15	15	Partial	(33, N/A) 59
MK07	20	55	5	Partial	(133, N/A) 143
MK08	20	10	15	Partial	523
MK09	20	10	15	Partial	307
MK10	20	15	15	Partial	(165, N/A) 214

In the following table 6 the instance “SFJ01” is shown, Highlighting the best possible operations to take in order to reach minimum makespan. in the figure 2 a Gant chart of the same instance presents the optimal makespan schedule distribution.

Furthermore, in table 7 the instance “SFJ10” is shown, Highlighting the best possible operations to take in order to reach minimum makespan. in the figure 3 a Gant chart of the same instance presents the optimal makespan schedule distribution.

TABLE 6. SFJ01 INSTANCE OF A TOTAL FLEXIBILITY FJSSP

		M1	M2
J1	**O1,1**	25	**37**
	O1,2	32	**24**
J2	**O2,1**	**45**	65
	O2,2	**21**	65

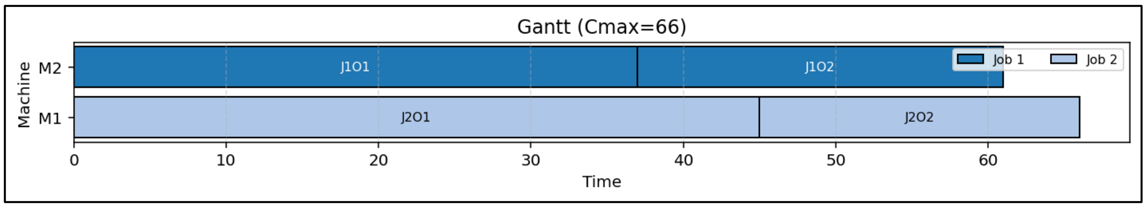


Figure 2. “SFJ01” Gantt chart of minimum makespan

TABLE 7. SFJ01 INSTANCE OF A TOTAL FLEXIBILITY FJSSP

		M1	M2	M3	M4	M5
	O1,1	**147**	-	-	-	-
J1	O1,2	-	130	-	**140**	-
	O1,3	-	-	-	**150**	160
	O2,1	214	-	**150**	-	-
J2	O2,2	-	**66**	87	-	-
	O2,3	-	-	-	-	**178**
	O3,1	87	**62**	-	-	-
J3	O3,2	-	-	**180**	-	-
	O3,3	-	-	-	190	**100**
	O4,1	87	**65**	-	-	-
J4	O4,2	-	-	-	-	**173**
	O4,3	-	-	**136**	145	-

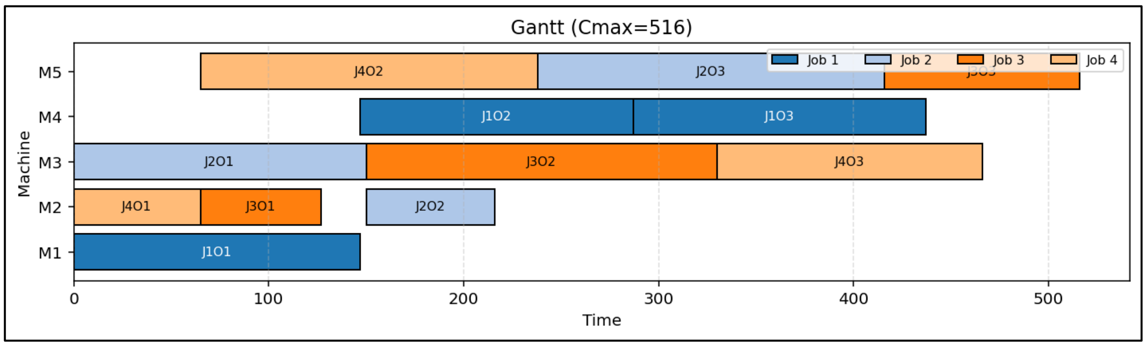


Figure 3. “SFJ10” Gantt chart of minimum makespan

Finally, a schedule solution of “mk10” instance is presented in figure 4. As it can be noted, this schedule that consist of 20 jobs and 13 is an optimum solution candidate. However, this C_{max} value of 278 is high in contrast to best value found literature of 214.

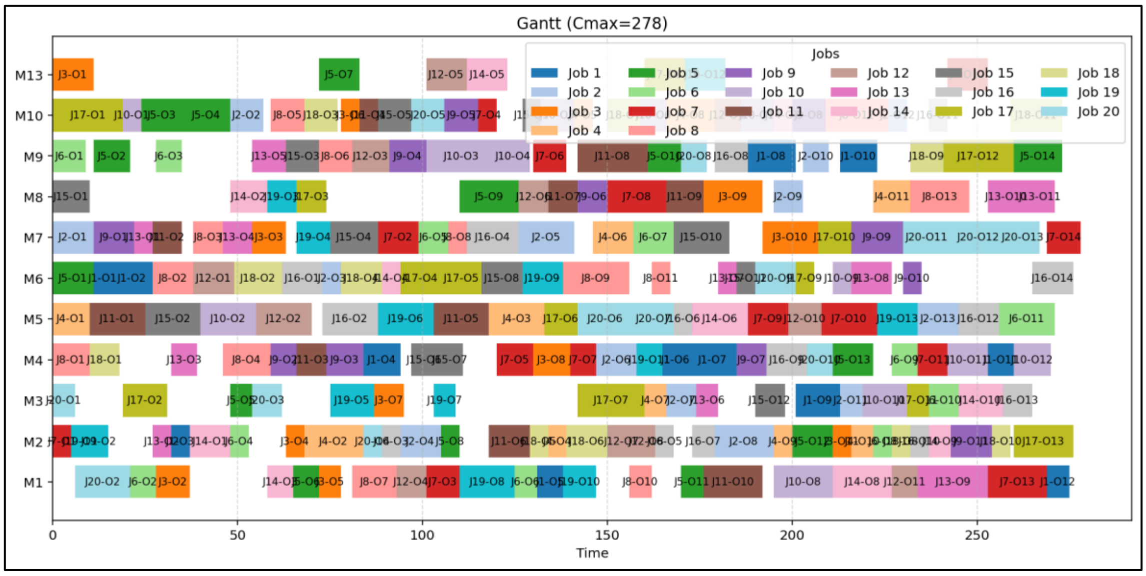


Figure 4. “SFJ10” Gantt chart of minimum makespan

4. METHODOLOGY

This chapter describes the methodology and references used to implement three metaheuristics, GA, SA and TS algorithms. A RLGA metaheuristic is defined used previous works from authors. Finally, some evaluation and metrics are presented to fairly compare these four search algorithms.

4.1. Genetic Algorithm baseline

As reviewed in the literature in chapter 2 and on the following papers [2], [5], [8], GA tries to mimic the evolution and mutation of chromosomes. The following Table 8 defines a general pseudocode commonly stablished for GA and based to the descriptions proposed in [8]. Genetic Algorithm special behavior for scheduling instances is de definition of a chromosome as an array of "schedule arrangement solution" which changes over generations with a probability of improvement.

TABLE 8. GENETIC ALGORITHM PSEUDOCODE

Genetic algorithm pseudocode

```
Input:
Jobshop Instance
GA parameters [pop_size], [gens], [tournament_k], [p_crossover], [p_mut_seq], [p_mut_assign], [elitism]
Begin
  Generation t = 0
  Initialize population P(t) of size [pop_size] by encoding two vector chromosome
  Two vector chromosome = operation_sequence + machine_sequence
  Decode and evaluate all individuals in P(t), compute makespan
  While generation t < [gens]
    While New_population < [pop_size]
      For a Total number according to [elitism]
        Get best individual from a tournament pool of [tournament_k]
      End For
      With probability [p_crossover]
        POX crossover method for best individuals
      Else:
        Keep best individuals
      With probability [p_mut_seq]
        Swap a gene of chromosome vector1 with vector2
      With probability [p_mut_assign]
        Mutate each gene of chromosome if possible (Job or machine restriction)
      Add best individuals to New_population
      Calculate best maskespan in new_population
    End While
    Next generation t = t+1
  End while
End
Output:
best_cmax
```

The tunning parameters used for this algorithm are defined in table 9. As it can be observed the main parameters that define the computational time cost of the algorithm is the population size and the total generations to be evaluated.

TABLE 9. TUNNING PARAMETERS FOR GA

Parameter	Definition and effects	Selected value
pop_size	Number of individuals in the population. Larger values increase diversity and stability but require more time per generation.	60
gens	Number of generations to run. More generations allow more improvement but increase runtime.	300
tournament_k	Tournament size for parent selection. Higher k increases selection pressure and speeds convergence but raises risk of premature convergence. Lower k preserves diversity.	3
p_crossover	Probability of applying crossover to a selected pair. Drives recombination of sequences and machine assignments. Very high values can disrupt good building blocks if mutation is low.	0.9
p_mut_seq	Sequence mutation rate (swap two positions in the operation sequence). Increases exploration of job orderings; too high can add noise.	0.10
p_mut_assign	Assignment mutation rate (per operation gene, reassign to another capable machine). Explores machine choices; too high can destabilize good assignments.	0.05
elitism	Number of top individuals copied unchanged into the next generation. Protects the best found so far; too large reduces diversity.	2
show_plots	If true, display convergence and Gantt charts after the run. No effect on search, only on output.	True

4.2. Simulated Annealing baseline

Simulated Annealing algorithm pseudocode for FJSSP problem is presented in Table 10 based on method explored in papers [2], [4], [5], [15]. The algorithm is observed to take many random decisions depending on a “temperature” parameter factor which decays over iterations.

TABLE 10. SIMULATED ANNEALING ALGORITHM PSEUDOCODE

Simulated Annealing pseudocode

```
Input:
Jobshop Instance
SA parameters [iterations], [T_start], [alpha]
Begin
  Generate random_solution, get makespan
  Assign makespan as best_cmax
  Assign random_solution as neighbour_solution
  set temperature as [T_start]
  For [iterations]
    Generate a new_neighbour_solution based on neighbour_solution
      Random probability of machine reassignment between operations
      Random probability of swapping opperations
    Evaluate new_neighbour_solution, get new_makespan
    if new_makespan < best_cmax
      update new_neightbour_solution to neighbour_solution
      update new_makespan to best_cmax
    if new_makespan > best_cmax
      with probability criteria (based on temperature)
        accept worse_neighbour
        update new_neightbour_solution to neighbour_solution
    reduce temperature based on [alpha]
  End for
End
Output:
best_cmax
```

In table 11 the tunning parameters for SA algorithm are detailed. In comparison to previous GA algorithm that uses population and generations to explore many solutions in batches, SA method uses only iterations, evaluating one candidate per iteration. High temperature opens the possibility to explore unexpected solutions.

TABLE 11. TUNNING PARAMETERS FOR SA

Parameter	Definition and effects	Selected value
max_iter	Total SA iterations. More iterations allow more exploration and refinement but increase runtime.	2000
T_start	Initial temperature. Higher values accept worse moves more often at the beginning, increasing exploration.	300.0
alpha	Geometric cooling factor per iteration. Temperature update: T ← T * alpha. Values closer to 1.0 cool more slowly and explore longer.	0.995

4.3. Tabu Search baseline

While SA algorithm previously defined is based on probabilities, Tabu search (TS) aims to be deterministic [2]. Pseudocode for TS algorithm that works on FJSSP environments is presented in Table 12 based on papers [4], [15]. This method also works with iterations like SA, but for each iteration it evaluates a group of candidates, called neighbors, with similar characteristics, and creates a tabu list to prevent the algorithm return backwards in exploration.

TABLE 12. TABU SEARCH ALGORITHM PSEUDOCODE

Simulated Annealing pseudocode

```
Input:
Jobshop Instance
TS parameters [iteration], [tenure], [neighbor_samples],[swap_prob]
Begin
  Generate random_solution, get makespan
  Assign makespan as best_cmax
  Create tabu_list with length of [tenure]
  set random_solution as actual_solution
  For [iterations]
    Generate neighbour_candidates from actual_solution
      Random probability of machine reassignment between operations
      Random probability of swapping opperations
    Evaluate neighbour_candidates, get candidates_makespan
    Select best_cantidate from neighbor_candidates
    if best_candidates in tabu_list
      Search for another candidate
    if best_candidate_cmax < best_cmax
      update best_cmax
    accept best_candidate as actual_solution
    update tabu_list with actual_solution
  End for
End
Output: best_cmax
```

In table 13 the tunning parameters for TS algorithm are detailed. The algorithm works by iterations, but each iterations evaluate a number of neighbor samples, which best candidate goes for next iteration.

TABLE 13. TUNNING PARAMETERS FOR TS

Parameter	Definition and effects	Selected values
iters	Maximum number of TS iterations. A higher value allows more search steps; runtime grows linearly	1000
tenure	Tabu tenure in iterations. The selected move’s key is stored with expiry it + tenure. Larger tenure increases diversification; too large can block progress when aspiration does not trigger.	10
neighbor_samples	Number of neighbors evaluated per iteration. More samples improve the chance to find a better move, but increase computation per iteration.	60

4.4. Reinforcement Learning Genetic Algorithm

A RLGA based on a project in [8] is defined, this RLGA treats GA as an environment and inserts a lightweight RL controller that adapts the algorithm search behavior online while the population evolves. At each generation, the RL agent observes a compact summary of the population state, chooses an action that sets the genetic operators' intensities, and receives a reward based on how the population improves. In figure 5 the architecture of RLGA is presented based on research in [8].

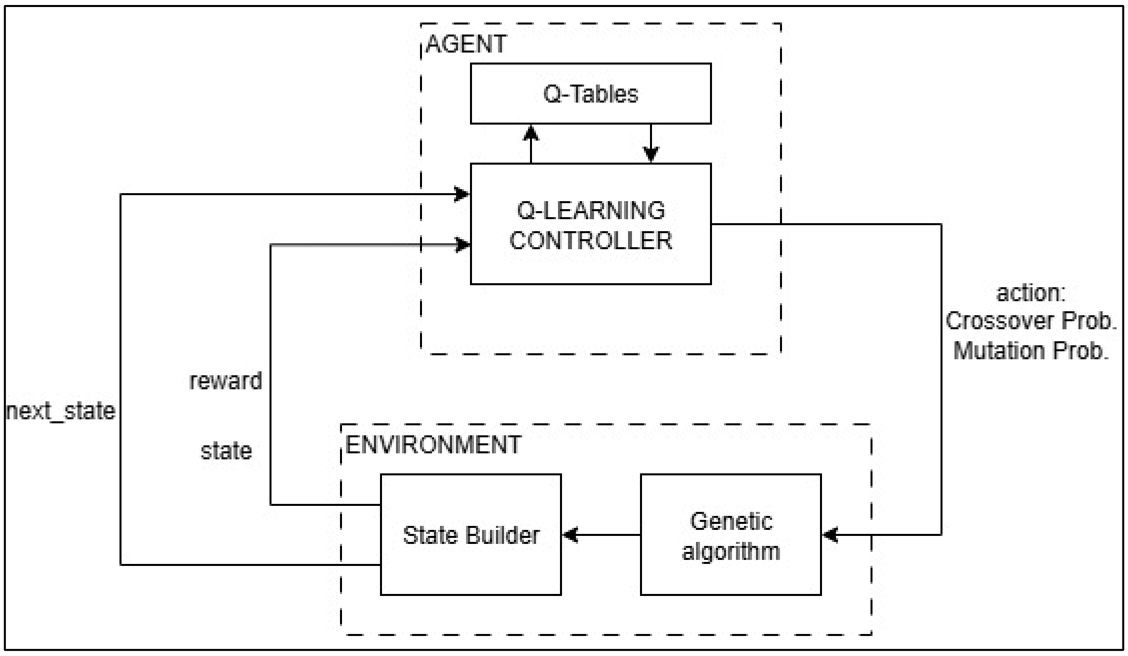


Figure 5. RLGA architecture

Based on figure 5, a description of each component is described below:

- Agent: The learning module that holds the Q-table and applies RL to tune GA parameters. It uses SARSA at start, a RL algorithm that provides a fast start search, but mainly focuses on Q-learning, updating Q with predefined rules.
- Environment: The genetic algorithm is the environment. When the agent sets crossover probability (Pc) and mutation probability (Pm), GA runs one generation and transitions from one state to next state, producing the feedback used by the agent.
- State: Carries information like fitness of population, population diversity, and fitness of best individual, being fitness equal to makespan in this evaluation.
- Action: Adjust the GA operator rates. The action set has 10 possible outcomes that map to ranges. “Pc” in [0.40, 0.90] with step 0.05, and “Pm” in [0.01, 0.21] with step 0.02. Action outcome multiplies the “Pc” and “Pm” in that range.
- Reward: Reward is based on the best fitness of the generation and its average the fitness compared to the previous generation.

In summary, based on the diagram presented in figure 5 and algorithm behavior detailed, the parameters needed to tune the RLTS algorithm are shown in table 14. As it can be observed, similar to TS, this RLTS can also work with iterations, but because of extra steps the computational time is slower.

TABLE 14. TUNNING PARAMETERS FOR RLGA

Parameter	Definition and effects	Selected values
pop_size	Number of individuals in the population. Larger values increase diversity and stability but require more time per generation.	50
gens	Number of generations to run. More generations allow more improvement but increase runtime.	60 (default)
tournament_k	Tournament size for parent selection. Higher k increases selection pressure and speeds convergence but raises risk of premature convergence. Lower k preserves diversity.	3
elitism	Number of top individuals copied unchanged into the next generation. Protects the best found so far; too large reduces diversity.	1
alpha	[RL tunning] Q-learning learning rate for the agent. Higher reacts faster to new evidence; too high can be noisy.	0.75
gamma	[RL tunning] Q-learning discount factor. Closer to 1.0 values future gains more, encouraging long-term strategies.	0.5
epsilon	[RL tunning] ε-greedy exploration rate. With probability ε the agent picks a random action	0.85

4.5. Evaluation Metrics

As algorithm perform in different places, GA and RLGA in populations, SA in iterations, and TS in neighbor iterations, trying to compare directly the algorithms could not led to a fair evaluation, so a group of tests are going to be stablish accordingly.

- For 10 small FJSSP with known optimal solution a quality test, two types of tests are set, limited evaluation criteria and limited time criteria,
- For 10 medium FJSSP with unknown optimal solution, a literature best found value will be set. Therefore, similar limited evaluations criteria and limited time criteria are going to be set.
- For 10 large FJSSP, only limited time criteria is going to be tested.

The evaluation test and expectations from results are defined in the rest of this chapter

4.5.1. Metrics for the small FJSSP

Because an optimum $C_{max}$for these instances is known, the algorithms are expected to be fast and converge efficiently, for that reason a stopping criteria modification is going to be programmed, so that after 500 evaluations or 100 evaluations with no improvements in searching for C_{max}, then the algorithm stops. The following metrics are going to be measured for a total of 1000 program runs.

- Success rate: defined as the number of times solution is achieved, formula in (1).

$$SR\% = \frac{\#runs\ \{Obtained_{Cmax} = Optimun_{Cmax}\}}{\#runs} x100\% \qquad (1)$$

- Solution quality (Gap): indicates how far was the obtained solution from optimum, formula set in (3).

$$Gap = 100 * \frac{Obtained_{Cmax} - Optimun_{Cmax}}{Optimun_{Cmax}} \qquad (3)$$

- Computational time of positive runs: the average time per run it takes to reach the optimum value.

Another evaluation is going to be made is the behavior of the search algorithm in a reasonable long time stopping time criteria. 1000 runs of 0.1 seconds are going to be given to each search algorithm to run freely, the following metrics are going to be capture:

- Average total evaluation per run: defined as the total schedule candidates evaluated by each algorithm in 0.1s, this indicates the speed of the algorithm but is hardly affected by the hardware where it is performed.
- Success Rate (SR): previously defined in (1).
- Solution Quality (Gap): previously defined in (3).
- Average time optimum solution found: In the given 0.1s, measure the average time it takes the algorithm to reach the optimum solution.

4.5.2. Metrics for the medium FJSSP

These 10 FJSSP instances do not have a known optimum minimum, but other literature papers offer a solution that could be taken as a reference. A stop criterion of 500 evaluations or 100 evaluations with no improvements is set to search for C_{max}. The following metrics are going to be measured for a total of 1000 runs.

- Best solution so far: Measure if, after 1000 runs the algorithm manages to reach a value close to literature best.
- Median of best results: The median value of 1000 runs; the median is selected over the average due to potential noise of randomness in the start of algorithms.
- Median Gap: to define the algorithm search quality, formula set in (4) is going to be calculated.

$$Median\_Gap = \frac{Obtained_{Cmax} - Literaute\_Best}{Literature\_Best} \tag{4}$$

- Evaluation stops (E): following the rule set in (2) the sum of, patience stops, times a run stop by no improvement, and evaluation stop, times a run stop by finishing the evaluation, is 1. In "E" is measured.

$$(Patience\ Stop) + (Evaluations\ stop) = 1 \tag{2}$$

All algorithm has different behavior in their processing time for each loop, as a consequence short evaluation or limiting by no improvement stopping criteria may led to not showing its performance. This evaluation gives a total of 0.1 seconds to run freely. A total of 1000 runs per algorithm are tested and the following measurements.

- Average total evaluation per run.
- Best C_{max}obtained in all runs.
- Median C_{max}obtained of all best C_{max}in runs.
- Median Gap: formula presented in (4)

4.5.3. Metrics for the Large FJSSP instances

These large instances will be test only for long term performance, 100 runs of 5 seconds for each algorithm. The following metrics are measured:

- Average total evaluation per run.
- Best C_{max}obtained in all runs.
- Median C_{max}obtained of all best C_{max}in runs.
- Median Gap: formula presented in (4)

5. EXPERIMENTAL RESULTS

The following chapter performs the tests set for previous chapter evaluation metrics. The evaluation has two stages, one for small FJSSP instances and one for medium FJSSP Instances. The evaluation setup is defined and the tests are performed with a evaluation of each result. Finally, a summary of finding is presented.

5.1. Evaluation setup

Some considerations that may affect the speed and variance of the results are detailed. All experiments run on a laptop with an 11th Gen Intel Core i7-1165G7 @ 2.80 GHz (4 cores, 8 threads). Computation is CPU-only, no discrete GPU is used. Operating system is windows 11.

Experiments are implemented in a Python stable version (3.13) and executed from Spyder IDE in version 6.0.7, both programming language and IDE being open-source technologies.

The execution parameters for all runs of all iteration use the parameters previously detailed, with the only exception of stopping conditions for the algorithm, which is determined for the evaluation criteria. The data instances are found in [19], and the scripts elaborated for this project can be found in [21] and its usage detailed in Annex A.

5.2. Small FJSSP instances test with limited evaluation

In this evaluation all four algorithms consider a stop criterion of 500 evaluation or 100 evaluations with no improvement of C_{max}. Table 15 shows the results for the 10 instances showing the three metrics previously defined as "Solution Quality, "Success Rate", "Patience stop" and "Computational time".

TABLE 15. SMALL FJSSP INTANCES UNDER LIMITED EVALUATIONS

INSTANCE	**GA**			**SA**			**TS**			**RLGA**		
	SR%	GAP%	CPU (ms)	SR%	GAP%	CPU (ms)	SR%	GAP%	CPU (ms)	SR%	GAP%	CPU (ms)
SFJSP-01	99,8	0,06	3	96,5	1,07	5	100	0	2	100	0	1
SFJSP-02	100	0	2	100	0	5	100	0	2	100	0	1
SFJSP-03	93,9	0,48	3	86	1,24	8	64,4	4,96	4	89,1	0,92	1
SFJSP-04	94,3	0,23	3	95,5	0,17	8	0	11,55	N/A	96,6	0,11	1
SFJSP-05	100	0	2	49,4	4,1	8	64,4	2,83	3	57,2	3,38	1
SFJSP-06	75	1,44	5	57,6	2,6	12	38,1	7,2	6	63,4	1,93	1
SFJSP-07	100	0	4	46,9	2,03	13	88,4	0,6	5	60,6	0,99	1
SFJSP-08	18,4	5,02	7	6,2	9,31	15	15,8	8,3	9	11,4	7,43	2
SFJSP-09	10,7	5,61	8	4,2	9,31	14	3,3	7,63	13	6,8	7,63	2
SFJSP-10	72,7	2,41	6	18,7	10,39	15	23,4	23,7	10	26	7,73	3

The following figure 6 shows three graphs representing the success, quality and speed of the results over ten instances.

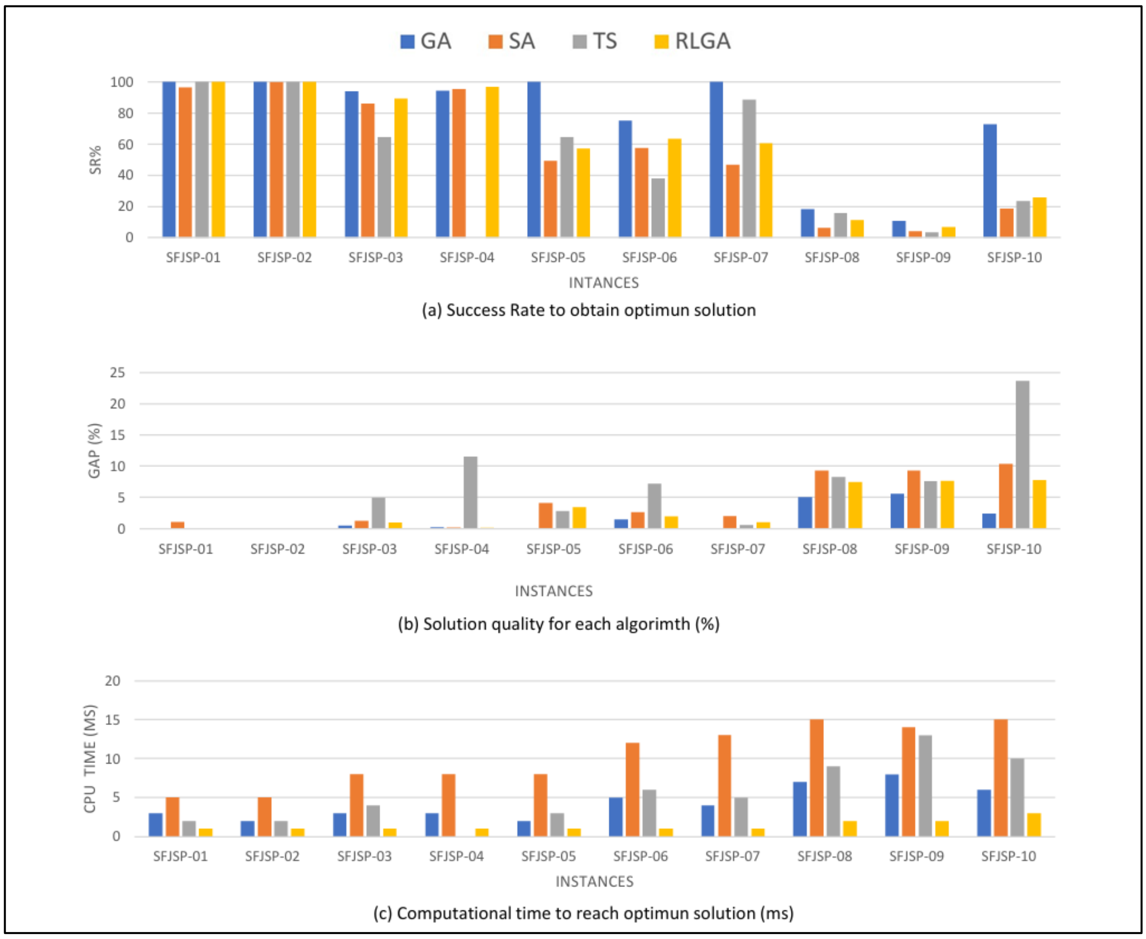


Figure 6. Small FJSSP instances evaluated with limited iterations

Figure X (a) shows all algorithms perform well in the first small instances, but they have a special trouble in dealing with instances 8 and 9, classic GA algorithm shows increased success rate than rest of algorithms.

Figure X (b) measures Gap (%) which refers to the average solution of 1000 runs in comparison to optimum value, therefore less is better. GA shows as the best followed by RLGA and SA, but TS being the one underperforming.

In figure X (c), the computational time results show that the fastest algorithm to reach the results is the RLGA, followed by GA, TS and finally SA. Considering all run under same hardware speed conditions, it can be concluded that RL improved GA algorithm to be faster. Also, SA random search takes more tries to found a solution.

This test, of limited evaluations and no improvement, evaluates the algorithm fast convergence in a small pool of candidates in contrast for next environments. In terms of computational speed and success solution found, GA performs better than other schedules, followed by RLGA that may need more action steps to learn. TS seems faster than SA to reach solutions, but on average SA has more accuracy than TS.

5.3. Small FJSSP instances test with time limit

In this test all algorithms will face the same 10 small FJSSP instances for 1000 runs, instead of stopping by a number of evaluations or no improvement, the algorithms are going to run indefinitely for 1s (100ms), this is a reasonable time considering in previous test the optimum solution is found in 1 to 15ms. The following table 16 show the results from this test.

TABLE 16. SMALL FJSSP INTANCES UNDER LIMITED TIME TEST

INSTANCE	**GA**				**SA**			
	AVG EV	SR%	GAP%	CPU (ms)	AVG EV	SR%	GAP%	CPU (ms)
SFJSP-01	2699,8	100	0	1	3723,2	100	0	1
SFJSP-02	3679,6	100	0	1	3738,1	100	0	1
SFJSP-03	3576,2	97,4	0,18	2	1579,1	100	0	5
SFJSP-04	4355,2	100	0	1	1623,6	100	0	3
SFJSP-05	2380,9	100	0	1	1639,2	99	0,08	11
SFJSP-06	2356,3	100	0	6	2195,2	99	0,02	10
SFJSP-07	3411,9	100	0	1	2111,9	100	0	13
SFJSP-08	3108,6	68	0,46	16	1421,6	57	1,17	43
SFJSP-09	2967,1	86	0,73	21	2346,4	40	2,69	19
SFJSP-10	2360,9	99,9	0,1	6	2127,2	89	0,75	21

(CONTINUE) SMALL FJSSP INTANCES UNDER LIMITED TIME TEST

INSTANCE	**TS**				**RLGA**			
	AVG EV	SR%	GAP%	CPU (ms)	AVG EV	SR%	GAP%	CPU (ms)
SFJSP-01	5227,3	100	0	1	5755,1	100	0	1
SFJSP-02	5104,7	100	0	1	5638	100	0	1
SFJSP-03	3823,7	100	0	4	5025,8	100	0	1
SFJSP-04	4252	100	0	13	4194,1	100	0	1
SFJSP-05	4221,6	98	0,12	10	5233,8	94	0,45	5
SFJSP-06	2819,6	94	0,17	10	4264,5	91	0,3	5
SFJSP-07	3068,8	100	0	2	3715,1	100	0	4
SFJSP-08	2856,6	35	1,81	18	3806,7	53	1,81	21
SFJSP-09	2819,4	64	1,58	28	3964,2	48	3,07	21
SFJSP-10	2368,9	91	1,13	29	3468,3	82	1,47	15

To notice, table 16 shows the average of evaluation each algorithm does in average is the set time of 0.1 seconds. In general, RLGA algorithm is faster than the rest, followed by TS, GA and last SA. This metric alone only proves the algorithm efficiency.

The following figure 7 shows three graphs representing the success, quality and speed of the results over ten instances.

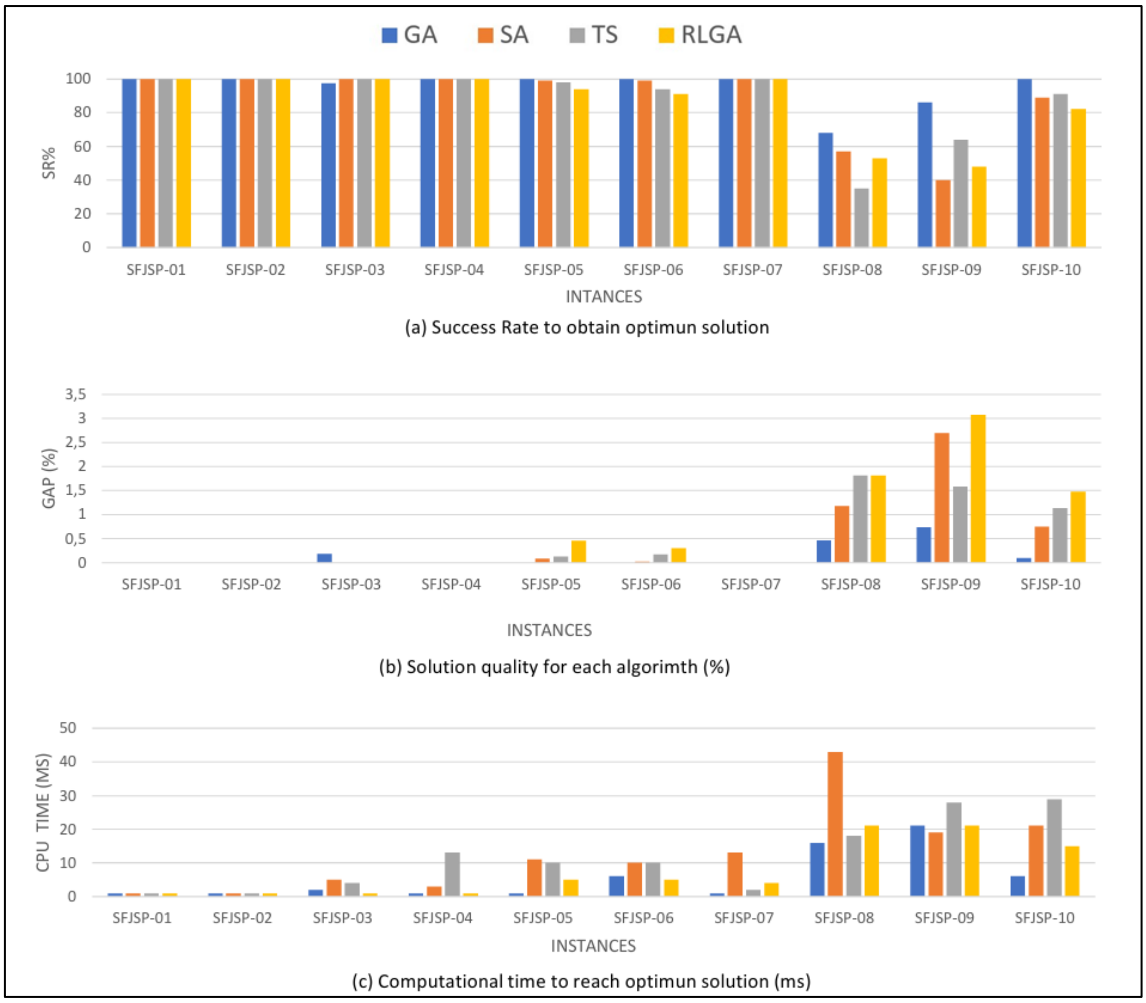


Figure 7. Small FJSSP instances evaluated with limited iterations

Figure 7 (a) SR graph shows that in the long term all algorithms perform well, but still have trouble in dealing with instances 8 and 9, classic GA is slightly better than TS and SA, but reinforcement learning does not outperform others noticeably.

Figure 7 (b) Gap (%) in long term shows that GA has the lowest values among all. TS has the second-best performance, being SA third and RLGA last. This combined with results in (a) heavily define RLGA being underperforming in contrast to other metaheuristics.

In figure 7 (c), the computational time results show that the fastest algorithm to reach the results is the RLGA, followed by GA, TS and finally SA. All algorithms run in same conditions, it can be concluded that GA and RLGA methods find the optimum solution faster. SA and TS take more time to find the best solution.

The conclusion of this test is that, overall, GA performs good in the long term, giving consistent better results and faster to obtain. TS and SA are slightly slower to found solutions but its performance to reach the best solution is high. RLGA is faster than SA and TS, slower than GA, but its success rate, which transfers to accuracy, and quality which transfers to precision, is lower in contrast to other metaheuristic algorithms.

5.4. Medium FJSSP instances test with limited evaluation

In comparison to the previous tests, these instances are bigger which means that more schedule candidates are possible. Also, there is no known optimum C_{max}, for that reason literature best findings are going to be used to make comparison. The following table 17 show metrics previously detailed in chapter 4.5.2 over 100 runs with 500 evaluations limit or a no improvement of 100 evaluations.

TABLE 17. MEDIUM FJSSP INTANCES UNDER LIMITED EVALUATIONS

INSTANCE	Lit. Best	GA					SA				
		Best	Med. best	Med. GAP	E-stop %	CPU	Best	Med. best	Med. GAP	E-stop %	CPU
MFJSP-01	468	469	572	0,22	0	11	477	570	0,22	1	5
MFJSP-02	446	459	566	0,27	0	12	457	564	0,26	1	5
MFJSP-03	466	491	680	0,46	0	12	529	681	0,46	1	6
MFJSP-04	554	630	814	0,47	1	14	619	813	0,47	1,3	6
MFJSP-05	514	599	790	0,54	2	13	592	786	0,53	1	7
MFJSP-06	635	752	932	0,47	2	17	733	932	0,47	2	7
MFJSP-07	879	1082	1359	0,55	2	18	1053	1362	0,55	1,9	10
MFJSP-08	884	1097	1383	0,56	2	21	1075	1383	0,56	3,5	11
MFJSP-09	1088	1317	1698	0,56	4	23	1353	1705	0,57	5	14
MFJSP-10	1267	1522	1960	0,55	2	32	1533	1974	0,56	5	15

(CONTINUE) MEDIUM FJSSP INTANCES UNDER LIMITED EVALUATIONS

INSTANCE	Lit. Best	TS					RLGA				
		Best	Med. best	Med. GAP	E-stop %	CPU	Best	Med. best	Med. GAP	E-stop %	CPU
MFJSP-01	468	468	530	0,13	5	20	468	555	0,19	0,6	6
MFJSP-02	446	446	511	0,15	8	20	457	545	0,22	0,4	7
MFJSP-03	466	479	616	0,32	21	23	507	652	0,4	1,8	7
MFJSP-04	554	575	764	0,38	18	28	584	791	0,43	1,7	9
MFJSP-05	514	554	803	0,56	10,5	25	594	759	0,48	1,6	9
MFJSP-06	635	700	862	0,36	20,8	35	697	901	0,42	1,9	12
MFJSP-07	879	993	1210	0,38	12,7	43	1090	1332	0,52	3,1	13
MFJSP-08	884	1022	1272	0,44	19,2	54	1078	1366	0,55	3,5	16
MFJSP-09	1088	1324	1590	0,46	28,2	82	1351	1680	0,54	3,3	20
MFJSP-10	1267	1486	1916	0,51	16	84	1539	1964	0,55	4,8	21

The following figure 8 show 4 graphs, median best value vs literature, quality using gap formula, the percentage of times algorithm reach evaluations limit, and algorithm speed.

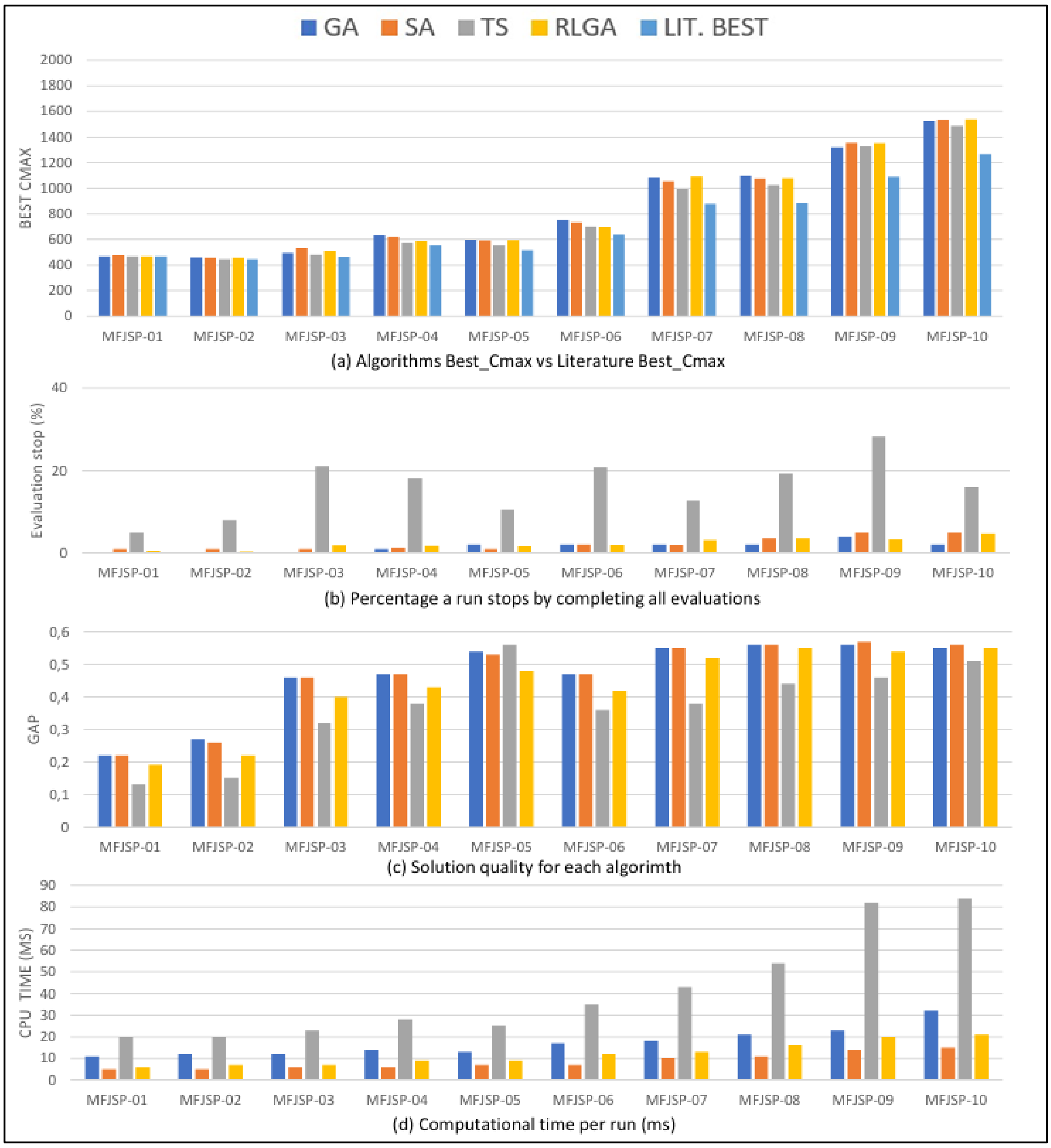


Figure 8. Medium FJSSP instances evaluated with limited iterations

Figure 8 (a) shows that all algorithms in a certain moment performs well, however TS consistently finds better solutions in all instances. Graph (b) shows that usually only TS finishes the 500 evaluations with no improvement, which means it escapes more usually from local minima, this implies TS converges better than other algorithms.

In Graph (c) it is also observed that TS has better median gap than other, being RLGA the second best. Therefore, if we consider best C_{max} obtained as accuracy and lowest Gap value as precision, TS performs better in this test, being RLGA the second best.

Finally in Graph (d) the computing time for all algorithms, TS shows to be the slowest among all, this can be attributed to performing more evaluations till the very end. However, compared to RLGA, second best after SA, being four times faster gives as good results as TS.

5.5. Medium FJSSP instances test with time limit

In this test all algorithms are tested for 0.1 second for a total of 1000 runs. The objective of this test is to observe the long-term performance of each algorithm and get who obtain better results and faster performance. The following table 18 show the result of this test.

TABLE 18. MEDIUM FJSSP INTANCES UNDER LIMITED TIME TEST

INSTANCE	**Lit. Best**	**GA**				**SA**			
		AVG EV	BEST CMAX	MED CMAX	GAP	AVG EV	BEST CMAX	MED CMAX	GAP
MFJSP-01	468	1152,2	468	477	0,02	4097,6	468	471	0,01
MFJSP-02	446	1138,1	448	492	0,1	4073,6	446	468	0,05
MFJSP-03	466	963,7	466	576	0,24	3734,4	466	508	0,09
MFJSP-04	554	993,4	575	662	0,19	3099,2	554	599	0,08
MFJSP-05	514	887,5	527	632	0,23	3201,1	514	569	0,11
MFJSP-06	635	863,9	694	814	0,28	2930,8	634	690	0,09
MFJSP-07	879	975,9	954	1122	0,28	2079,2	888	1023	0,16
MFJSP-08	884	954,6	1014	1197	0,35	1848,3	914	1034	0,17
MFJSP-09	1088	818,3	1285	1533	0,41	1791,3	1145	1274	0,14
MFJSP-10	1267	799,2	1525	1695	0,34	1526,3	1320	1482	0,17

(CONTINUE) MEDIUM FJSSP INTANCES UNDER LIMITED TIME TEST

INSTANCE	**Lit. Best**	**TS**				**RLGA**			
		AVG EV	BEST CMAX	MED CMAX	GAP	AVG EV	BEST CMAX	MED CMAX	GAP
MFJSP-01	468	1648,4	468	491	0,05	2682,2	468	498	0,06
MFJSP-02	446	1813,3	446	468	0,05	2933,5	448	470	0,05
MFJSP-03	466	1498,2	466	527	0,13	2483,4	466	533	0,14
MFJSP-04	554	1123,6	564	676	0,22	2193,6	564	652	0,18
MFJSP-05	514	1249,3	514	650	0,26	2021,2	518	624	0,21
MFJSP-06	635	1102,2	649	770	0,21	2151,2	639	743	0,17
MFJSP-07	879	692,7	975	1133	0,29	1729	954	1133	0,25
MFJSP-08	884	577,2	1028	1207	0,37	1318,5	999	1136	0,29
MFJSP-09	1088	449,8	1316	1564	0,44	1332,2	1214	1398	0,28
MFJSP-10	1267	398,4	1547	1862	0,47	1175,3	1470	1632	0,29

In figure 9 3 graphs are presented, (a) the best C_{max} value obtained per algorithm in all 1000 runs, (b) the gap value using median value of C_{max} obtained in all runs, finally in (c) the average number of evaluations per run of each algorithm.

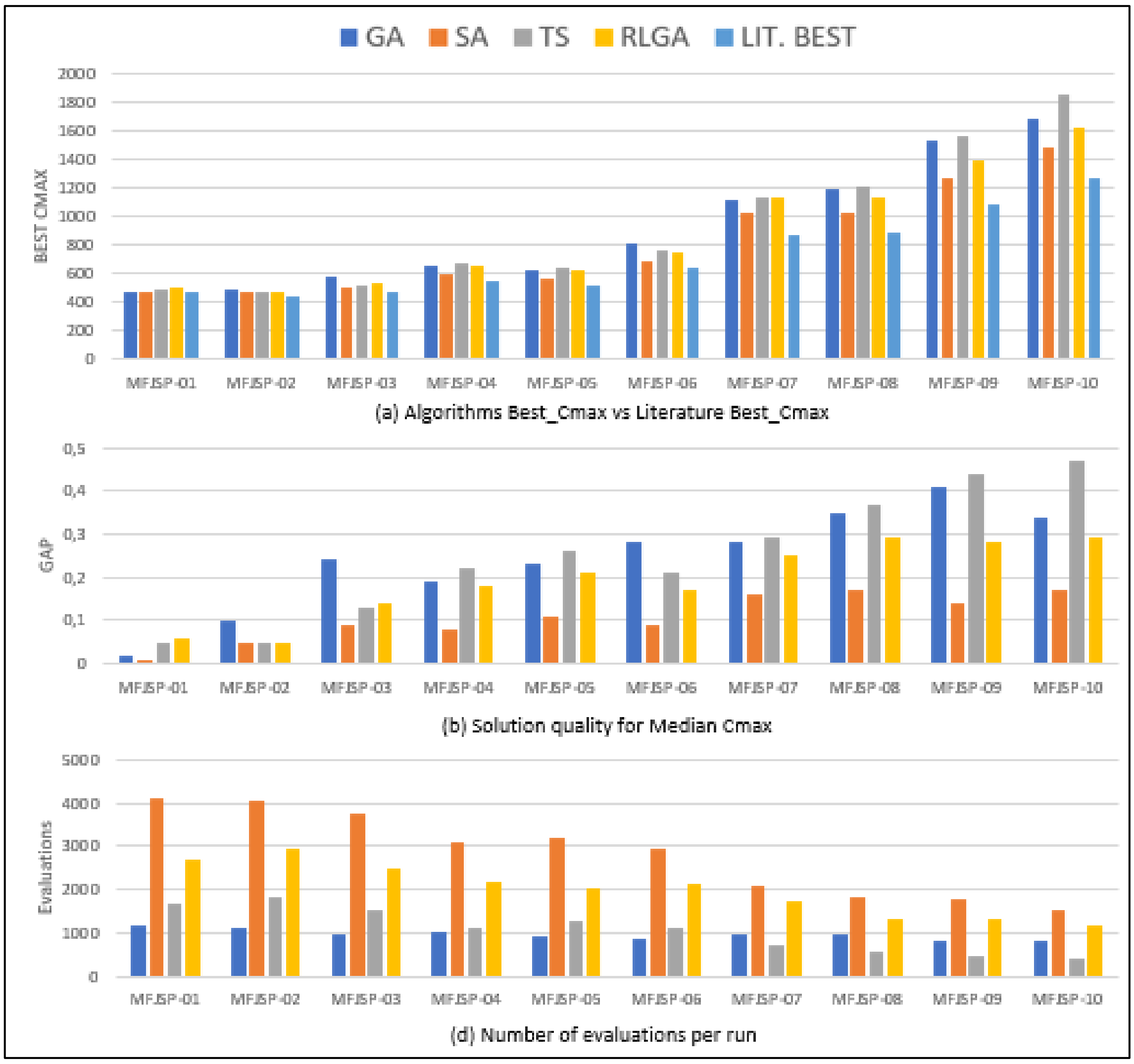


Figure 9. Medium FJSSP instances evaluated with in a limited period of time

As it can be observed in figure X graph (a) in contrast to the limited evaluations test presented before, in this graph is observed more slightly differences in accuracy to find the lowest C_{max}. SA being the algorithm that performs best followed by RLGA. TS and GA achieve higher Cmax values which means worse solutions were found in 1000 runs.

On graph (b) using the median C_{max} obtained in each algorithm, the lowest Gap belongs to SA followed by RLGA, third place by GA and last TS being considerably worst than the rest in the majority of the cases.

The previous results are consistent with the measure of evaluations pers run. SA is by far the fastest in all instances, followed again by RLGA. TS is the slowest of all algorithms, Evaluating less candidates in instances 9 and 8 than in the previous test.

To conclude, even thought in evaluation limit test (Chapter 5.4) TS result best in performance but slow, in this test the slow performance of the algorithm makes it the worst performing. On the other hand, RLGA is the second best after SA in speed, accuracy and precision.

5.6. Large FJSSP instances test with time limit

This test is performed in the tan large FJSSP instances, which have a high number of machine and operation, this gives a high pool of schedule variants with a high difficulty in optimization. For this test 100 runs of 1 second are performed for each algorithm, results are presented in table 19.

TABLE 19. LARGE FJSSP INTANCES UNDER LIMITED TIME TEST

INSTANCE	**Lit. Best**	**GA**				**SA**			
		AVG EV	BEST CMAX	MED CMAX	GAP	AVG EVALS	BEST CMAX	MED CMAX	GAP
MK01	40	5346,7	42	42	0,05	10044,6	40	42	0,05
MK02	27	5296,2	31	34	0,26	11414,6	28	31	0,15
MK03	204	2670,2	223	241	0,18	4263,8	204	216	0,06
MK04	60	4726,2	71	75	0,25	7861,7	66	73	0,22
MK05	174	3780,9	184	192	0,1	7329,8	178	181	0,04
MK06	59	2664,1	101	110	0,86	3237,5	89	105	0,78
MK07	143	3607,8	167	180	0,26	7635,2	150	172	0,20
MK08	523	2212,2	537	564	0,08	2663,3	535	565	0,08
MK09	307	2008,7	419	444	0,44	2408,7	376	417	0,36
MK10	214	1887,1	340	367	0,71	2442,2	304	344	0,61

(CONTINUE) LARGE FJSSP INTANCES UNDER LIMITED TIME TEST

INSTANCE	**Lit. Best**	**TS**				**RLGA**			
		AVG EV	BEST CMAX	MED CMAX	GAP	AVG EVALS	BEST CMAX	MED CMAX	GAP
MK01	40	3408,4	42	47	0,17	10147,2	42	43	0,07
MK02	27	3246,6	31	37	0,37	9762,9	32	36	0,31
MK03	204	801,1	236	272	0,34	3563,3	219	234	0,15
MK04	60	1190,1	81	93	0,55	6338,7	69	75	0,25
MK05	174	1292,2	193	214	0,23	5178,6	180	188	0,08
MK06	59	865,9	95	104	0,76	4062,2	102	107	0,81
MK07	143	1171,7	171	192	0,34	5533,4	154	178	0,2
MK08	523	382,6	596	635	0,21	2237,7	530	558	0,07
MK09	307	399,4	427	460	0,5	2492,6	395	407	0,33
MK10	214	306,6	328	366	0,71	2502,6	284	321	0,5

The following figure 10 show 3 graphs for large FJSSP instances, (a) the best C_{max} value obtained per algorithm in 100 runs, (b) the gap value using median value of C_{max} obtained, and in (c) the average number of evaluations per run of each algorithm.

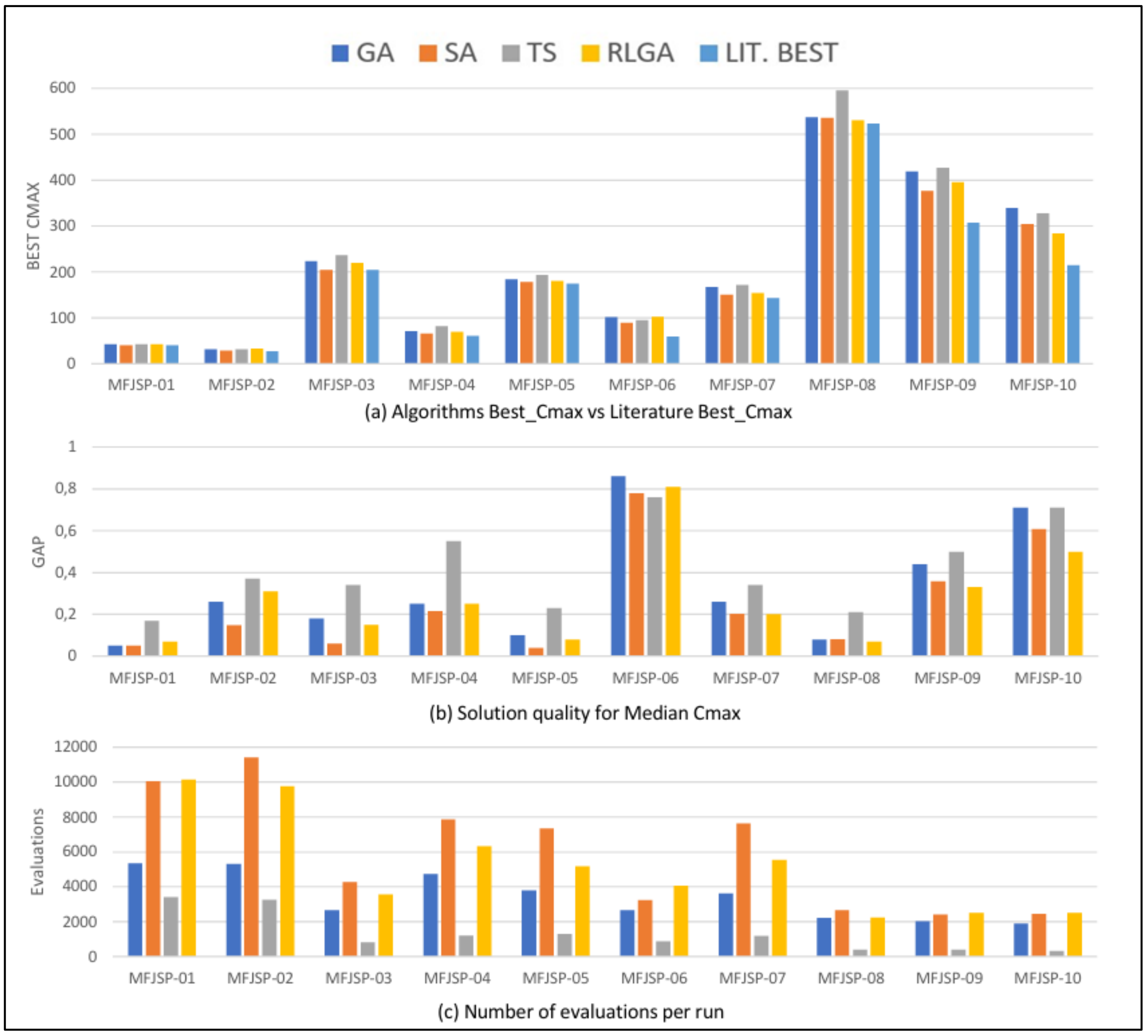


Figure 10. Large FJSSP instances evaluated with in a limited period of time

From figure 10 graph (a) and (b) show the best C_{max} obtained per algorithm and the gap between the median values with the literature, these measurements can be translated to accuracy of the algorithm and its precision over runs, respectively. RLGA do have similar performance than other algorithms ins the first instances, but its general performance is slightly better in the 3 last ones, where obtain similar results in minimum C_{max} with SA, but slightly better median gap, which means that across runs RLGA gave much better minimums.

In graph (c) the average evaluation of algorithms per run is measured. Overall, the fastest algorithm in terms of evaluations is SA, followed by RLGA. The speed of SA algorithm helps in finding better solution. However, for the largest, and in consequence harder to solve instances, RLGA speed and SA speed are about the same, being RLGA a better solution for harder FJSSP instances.

6. CONCLUSIONS AND FUTURE WORK

Across small, medium, and large Flexible Job Shop Scheduling benchmarks, the Reinforcement Learning Genetic Algorithm (RLGA) proved to be consistently reliable, robust to parameter choices, and scalable with problem size. Although RLGA was not the single best performer on every metric, it repeatedly delivered competitive solution quality while maintaining strong runtime characteristics. In small time-limited runs it achieved solutions rapidly, and in medium and large sets it remained among the two fastest approaches while preserving accuracy and precision within close range of the best metaheuristic per test. These patterns, observed over success rate, median gap, and evaluation throughput, support the claim that RLGA offers a balanced trade-off between speed and quality that generalizes across instance scales.

In relation to the comparative behavior of the classic Genetic Algorithm and RLGA, GA excelled on several small instances and short horizons but exhibited a decay in performance as problem size grew and when evaluation budgets favored longer searches. This decay is consistent with fixed operator schedules and static parameterization that can stagnate in local basins. RLGA outperformed GA in these regimes by using a learning signal to adapt operator selection, search intensity, and diversification on the fly. In practice, the RL layer acted as a controller for GA, reallocating effort toward promising generations and increasing exploration when improvements stalled, which improved median gaps and stabilized performance under stricter time budgets.

Limitations should be noted, solutions were drawn under CPU-only execution with fixed stop criterion and fairness constraints that map evaluations and patience to heterogeneous algorithmic loops. While this design enables controlled comparisons, it can underperform methods whose inner-loop costs differ. Similarly, evaluating in a short time constraint do not represent all algorithm best performance, if more time is given, like minutes or hours, algorithms like GA and RLGA can continue improving in theory, while TS and SA will eventually plateau due to its algorithm behavior.

In our experiments, GA, SA, and TS used fixed, user defined parameters, whereas RLGA adapted its settings on the run. This limits strict comparability because the baselines did not respond to instance scale or time budgets. It also highlights a practical advantage of RLGA, namely reduced manual tuning and more stable performance across cases.

In future works it could take a try replacing the current RL controller with Deep RL metaheuristics used in two modes: online and pretrained. Online DRL could be capable to read richer state features from the scheduler and adjust operators and intensification in real time. Pretrained DRL policies will be trained on many FJSSP instances and then reused on new ones, with brief fine tuning for size and routing changes. This aims to keep RLGA robustness while closing the remaining quality gaps and lowering run time.

BIBLIOGRAPHY

[1] J. Namjoshi and M. Rawat, “Role of smart manufacturing in industry 4.0,” *Mater Today Proc*, vol. 63, pp. 475–478, 2022, doi: https://doi.org/10.1016/j.matpr.2022.03.620.

[2] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*. Springer International Publishing, 2022. doi: 10.1007/978-3-031-05921-6.

[3] S. Dauzère-Pérès, J. Ding, L. Shen, and K. Tamssaouet, “The flexible job shop scheduling problem: A review,” *Eur J Oper Res*, vol. 314, no. 2, pp. 409–432, 2024, doi: https://doi.org/10.1016/j.ejor.2023.05.017.

[4] P. Fattahi, M. Saidi Mehrabad, and F. Jolai, “Mathematical modeling and heuristic approaches to flexible job shop scheduling problems,” *J Intell Manuf*, vol. 18, no. 3, pp. 331–342, 2007, doi: 10.1007/s10845-007-0026-8.

[5] D. T. Pham and D. Karaboga, *Intelligent optimisation techniques : genetic algorithms, tabu search, simulated annealing and neural networks*. London: : Springer, 2000.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[7] M. Jiang, H. Yu, and J. Chen, “Improved Self-Learning Genetic Algorithm for Solving Flexible Job Shop Scheduling,” *Mathematics*, vol. 11, no. 22, 2023, doi: 10.3390/math11224700.

[8] R. Chen, B. Yang, S. Li, and S. Wang, “A self-learning genetic algorithm based on reinforcement learning for flexible job-shop scheduling problem,” *Comput Ind Eng*, vol. 149, p. 106778, 2020, doi: https://doi.org/10.1016/j.cie.2020.106778.

[9] M. Khadivi *et al.*, “Deep reinforcement learning for machine scheduling: Methodology, the state-of-the-art, and future directions,” *Comput Ind Eng*, vol. 200, p. 110856, 2025, doi: https://doi.org/10.1016/j.cie.2025.110856.

[10] M. Khadivi *et al.*, “Deep reinforcement learning for machine scheduling: Methodology, the state-of-the-art, and future directions,” *Comput Ind Eng*, vol. 200, p. 110856, 2025, doi: https://doi.org/10.1016/j.cie.2025.110856.

[11] D. Behnke and M. J. Geiger, “Test instances for the flexible job shop scheduling problem with work centers,” 2012, *Helmut-Schmidt-Universität.*

[12] R. Ruiz and J. A. Vázquez-Rodríguez, “The hybrid flow shop scheduling problem,” *Eur J Oper Res*, vol. 205, no. 1, pp. 1–18, 2010, doi: https://doi.org/10.1016/j.ejor.2009.09.024.

[13] L. Song, C. Liu, H. Shi, and J. Zhu, “An Improved Immune Genetic Algorithm for Solving the Flexible Job Shop Scheduling Problem with Batch Processing,” *Wirel Commun Mob Comput*, vol. 2022, Jun. 2022, doi: 10.1155/2022/2856056.

[14] G. Zhang, L. Gao, and Y. Shi, “An effective genetic algorithm for the flexible job-shop scheduling problem,” *Expert Syst Appl*, vol. 38, no. 4, pp. 3563–3573, 2011, doi: https://doi.org/10.1016/j.eswa.2010.08.145.

[15] W. Ma, Y. Zuo, J. Zeng, S. Liang, and L. Jiao, “A memetic algorithm for solving flexible Job-Shop Scheduling Problems,” in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 66–73. doi: 10.1109/CEC.2014.6900332.

[16] N. Niroumandrad, N. Lahrichi, and A. Lodi, “Learning tabu search algorithms: A scheduling application,” *Comput Oper Res*, vol. 170, p. 106751, 2024, doi: https://doi.org/10.1016/j.cor.2024.106751.

[17] N. Niroumandrad, N. Lahrichi, and A. Lodi, “Learning tabu search algorithms: A scheduling application,” *Comput Oper Res*, vol. 170, p. 106751, 2024, doi: https://doi.org/10.1016/j.cor.2024.106751.

[18] L. Wang, Z. Pan, and J. Wang, “A Review of Reinforcement Learning Based Intelligent Optimization for Manufacturing Scheduling,” *Complex System Modeling and Simulation*, vol. 1, no. 4, pp. 257–270, 2021, doi: 10.23919/CSMS.2021.0027.

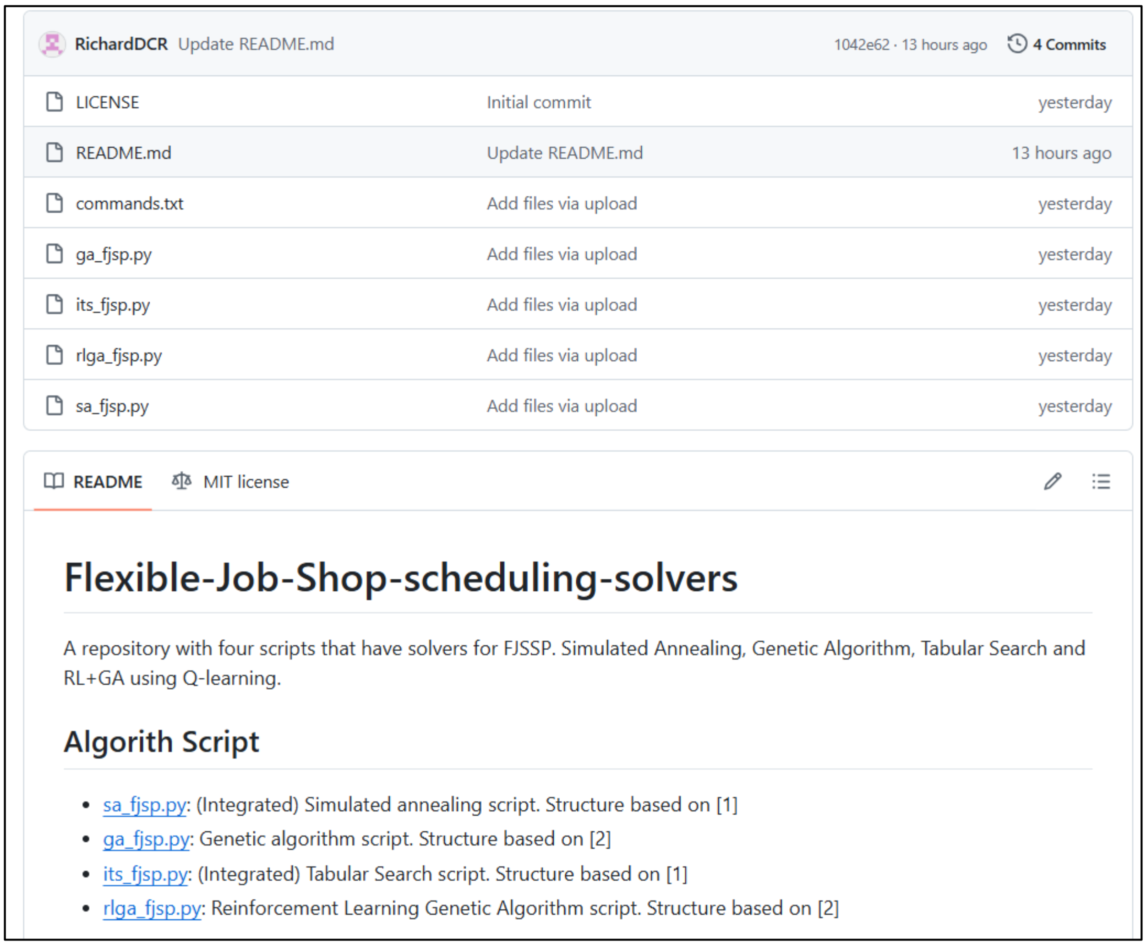
[19] SchedulingLab, “Flexible Job-shop Instances,” Jun. 2025.

[20] P. Brandimarte, “Routing and scheduling in a flexible job shop by tabu search,” *Ann Oper Res*, vol. 41, no. 3, pp. 157–183, 1993, doi: 10.1007/BF02023073.

[21] RichardDCR, “Flexible-Job-Shop-scheduling-solvers,” Sep. 2025.

ANNEX A: CODE USAGE AND EXPERIMENT REPRODUCTION

In the following reference [21], a GitHub repository with the scripts used in the project are published. The following figure A1 shows the repository.



RichardDCR Update README.md 1042e62 · 13 hours ago 4 Commits

LICENSE	Initial commit	yesterday
README.md	Update README.md	13 hours ago
commands.txt	Add files via upload	yesterday
ga_fjsp.py	Add files via upload	yesterday
its_fjsp.py	Add files via upload	yesterday
rlga_fjsp.py	Add files via upload	yesterday
sa_fjsp.py	Add files via upload	yesterday

Flexible-Job-Shop-scheduling-solvers

A repository with four scripts that have solvers for FJSSP. Simulated Annealing, Genetic Algorithm, Tabular Search and RL+GA using Q-learning.

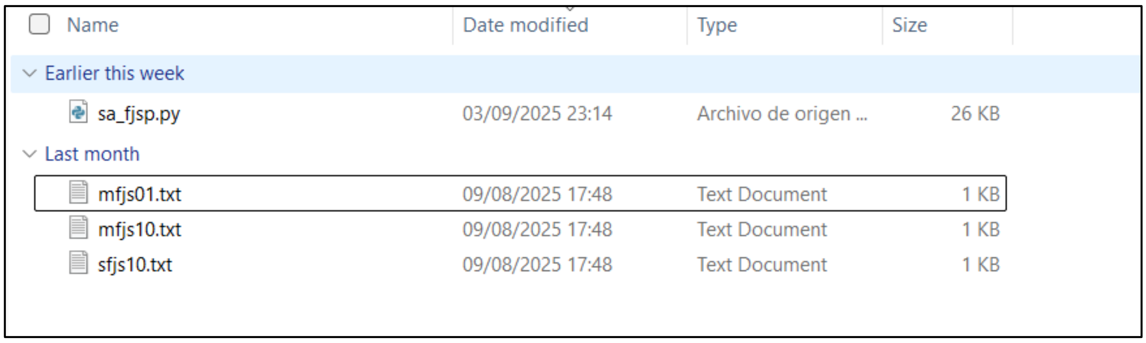
Algorith Script

- sa_fjsp.py: (Integrated) Simulated annealing script. Structure based on [1]
- ga_fjsp.py: Genetic algorithm script. Structure based on [2]
- its_fjsp.py: (Integrated) Tabular Search script. Structure based on [1]
- rlga_fjsp.py: Reinforcement Learning Genetic Algorithm script. Structure based on [2]

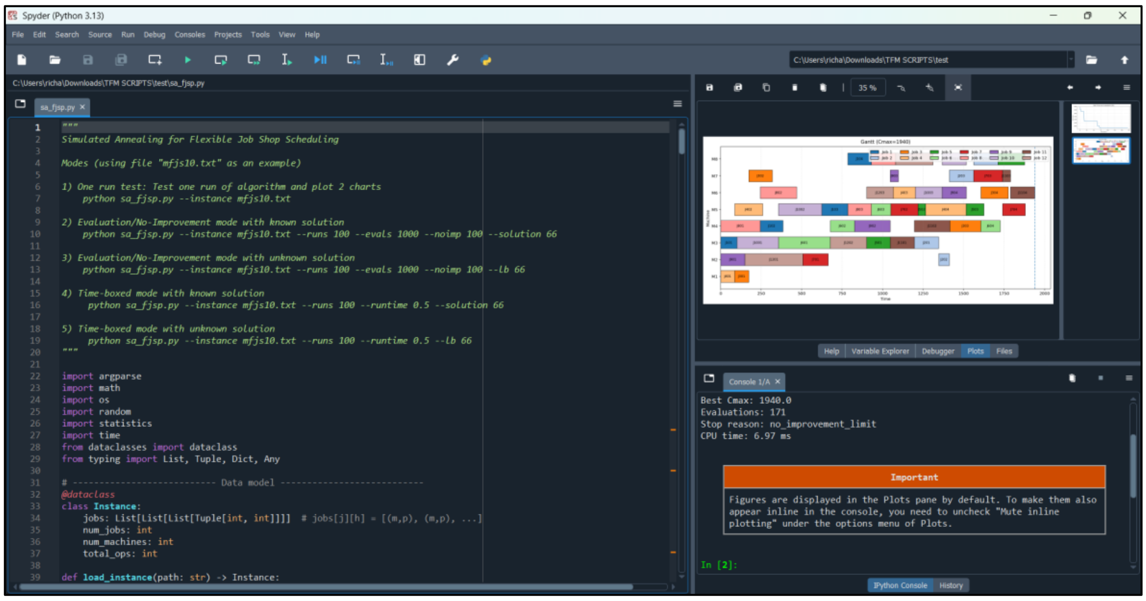
Figure A1. GitHub repository with FJSSP scripts

The repository has four scripts, one for each algorithm. As it is discussed in the project, the scripts are tested on Spyder IDE environment, but the scripts are not enough to get solutions because a FJSP instance is required for running. All scripts by default search for instance “mfjs10.txt”.

In figure A2, two figures are presented, in the first one is the files organization used to correctly manage a script, and in the second figure the representation of the default solution used when running the code in Spyder.



Name	Date modified	Type	Size
Earlier this week			
sa_fjsp.py	03/09/2025 23:14	Archivo de origen ...	26 KB
Last month			
mfjs01.txt	09/08/2025 17:48	Text Document	1 KB
mfjs10.txt	09/08/2025 17:48	Text Document	1 KB
sfjs10.txt	09/08/2025 17:48	Text Document	1 KB


Spyder (Python 3.13)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\richa\Downloads\TFM SCRIPTS\test
C:\Users\richa\Downloads\TFM SCRIPTS\test\sa_fjsp.py
sa_fjsp.py
Gantt (Cmax=1940)
Time
Help
Variable Explorer
Debugger
Plots
Files
Console 1/A
IPython Console
History

```python
"""
Simulated Annealing for Flexible Job Shop Scheduling

Modes (using file "mfjs10.txt" as an example)

1) One run test: Test one run of algorithm and plot 2 charts
    python sa_fjsp.py --instance mfjs10.txt

2) Evaluation/No-Improvement mode with known solution
    python sa_fjsp.py --instance mfjs10.txt --runs 100 --evals 1000 --noimp 100 --solution 66

3) Evaluation/No-Improvement mode with unknown solution
    python sa_fjsp.py --instance mfjs10.txt --runs 100 --evals 1000 --noimp 100 --lb 66

4) Time-boxed mode with known solution
     python sa_fjsp.py --instance mfjs10.txt --runs 100 --runtime 0.5 --solution 66

5) Time-boxed mode with unknown solution
     python sa_fjsp.py --instance mfjs10.txt --runs 100 --runtime 0.5 --lb 66
"""

import argparse
import math
import os
import random
import statistics
import time
from dataclasses import dataclass
from typing import List, Tuple, Dict, Any

# -------------------------- Data model --------------------------
@dataclass
class Instance:
    jobs: List[List[List[Tuple[int, int]]]]  # jobs[j][h] = [(m,p), (m,p), ...]
    num_jobs: int
    num_machines: int
    total_ops: int

def load_instance(path: str) -> Instance:
```

```
Best Cmax: 1940.0
Evaluations: 171
Stop reason: no_improvement_limit
CPU time: 6.97 ms

Important
Figures are displayed in the Plots pane by default. To make them also
appear inline in the console, you need to uncheck "Mute inline
plotting" under the options menu of Plots.

In [2]:
```

Figure A2. File organization and first run in Spyder IDE

All scripts have 5 different run modes which require different arguments to insert in spyder to perform different actions:

1. One run test: Test one run of algorithm and plot 2 charts

   python sa_fjsp.py --instance mfjs10.txt

2. Evaluation/No-Improvement mode with known solution

   python sa_fjsp.py --instance mfjs10.txt --runs 100 --evals 1000 --noimp 100 --solution 66

3. Evaluation/No-Improvement mode with unknown solution

   python sa_fjsp.py --instance mfjs10.txt --runs 100 --evals 1000 --noimp 100 --lb 66

4. Time-boxed mode with known solution

   python sa_fjsp.py --instance mfjs10.txt --runs 100 --runtime 0.5 --solution 66

5. Time-boxed mode with unknown solution

   python sa_fjsp.py --instance mfjs10.txt --runs 100 --runtime 0.5 --lb 66

The first, and default run mode, requires the script to be called and a instance to be called, this could be done in the Spyder console by pressing the “run” button or by running the following command.

```
%run sa_fjsp.py --instance mfjs10.txt
```

In addition, if not in Spyder IDE, is possible to call the command the way showed below, but the expected plots to be plotted are not going to appear because they are only available to be shown in Spyder IDE.

```
python sa_fjsp.py --instance mfjs10.txt
```

Figure A3 shows the expected result of this mode when being called in Spyder IDE. This mode uses the parameters set for each algorithm in Chapter 4.

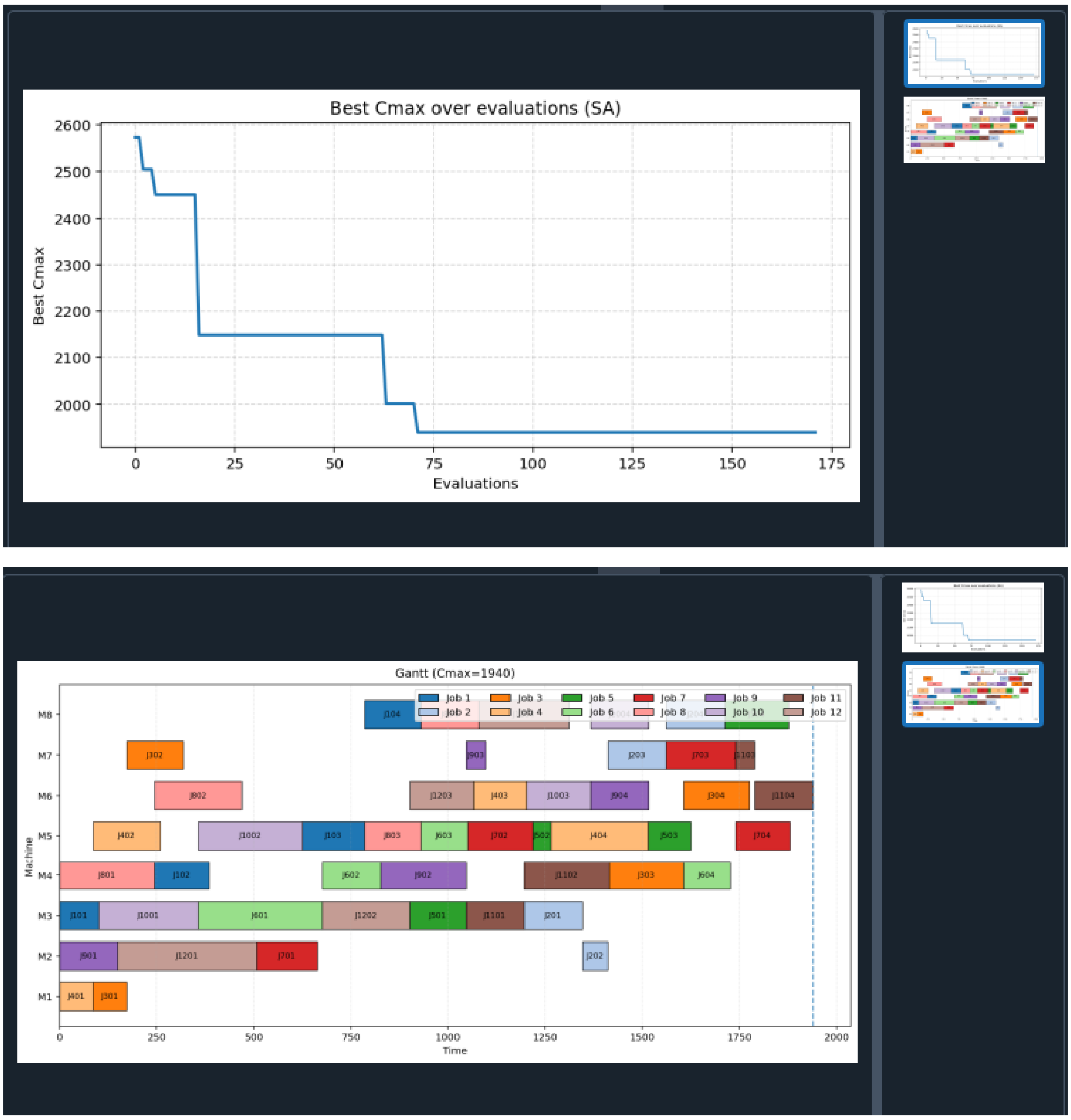


Figure A3. Results from first run mode

The second run mode requires the script to be called, the instance, the number of runs you want to perform, the total number of evaluations per run, the number of evaluations without improvement limitation as a stopping criteria and the best solution if known. The following scripts are valid for different script situations.

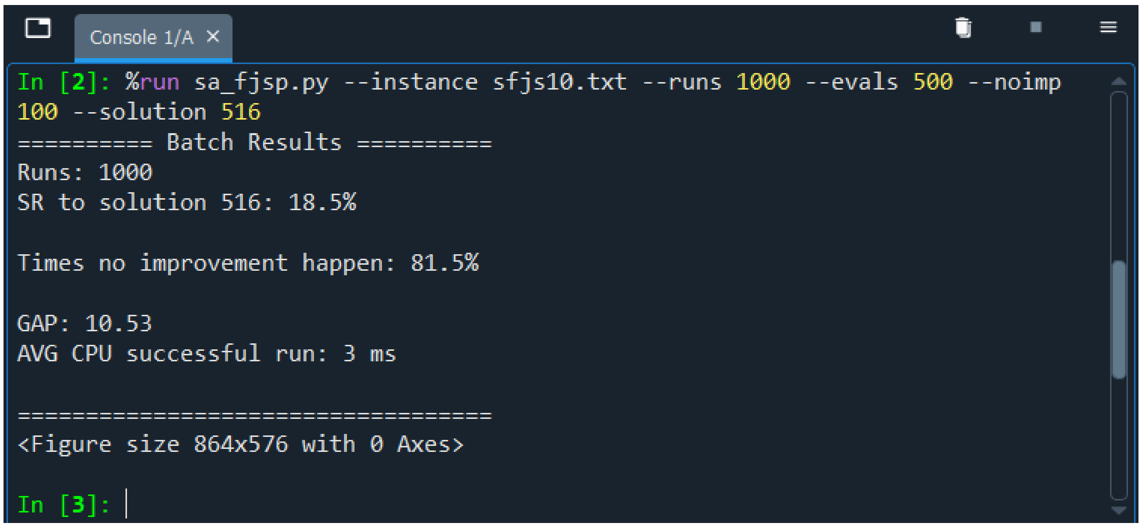
%run sa_fjsp.py --instance sfjs01.txt --runs 1000 --evals 500 --noimp 100 --solution 66

%run ga_fjsp.py --instance sfjs08.txt --runs 1000 --evals 500 --noimp 100 --solution 253

%run its_fjsp.py --instance sfjs10.txt --runs 1000 --evals 500 --noimp 100 --solution 516

%run rlga_fjsp.py --instance sfjs03.txt --runs 1000 --evals 500 --noimp 100 --solution 221

The following figure A4 shows the expected result plotted in the Spyder IDE console.

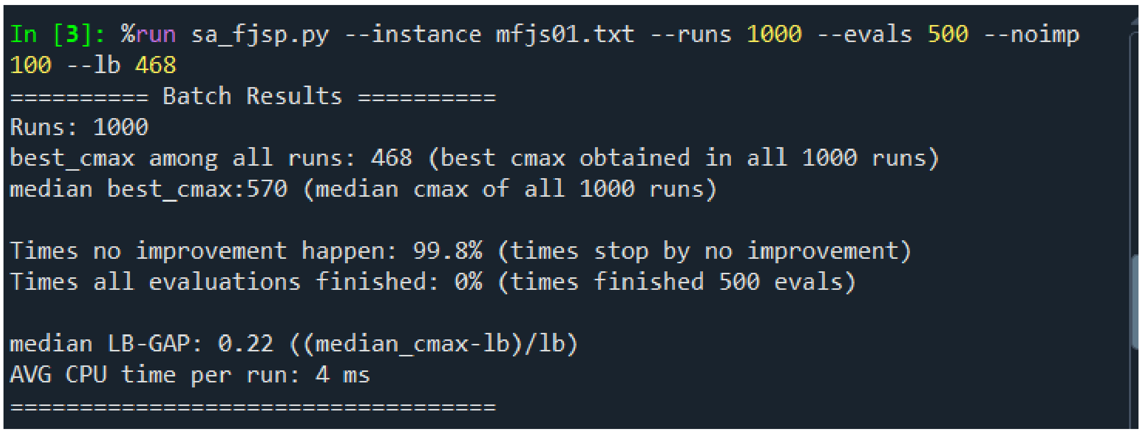


```
In [2]: %run sa_fjsp.py --instance sfjs10.txt --runs 1000 --evals 500 --noimp
100 --solution 516
========== Batch Results ==========
Runs: 1000
SR to solution 516: 18.5%

Times no improvement happen: 81.5%

GAP: 10.53
AVG CPU successful run: 3 ms

===================================
<Figure size 864x576 with 0 Axes>

In [3]:
```

Figure A4. Results from second run mode

The third run mode is similar to second, but instead of declaring a known solution, it is required a lower bound, the metrics are different because lower bound is rarely reached. Figure A5 shows the expected results.

%run sa_fjsp.py --instance mfjs01.txt --runs 1000 --evals 500 --noimp 100 --lb 468

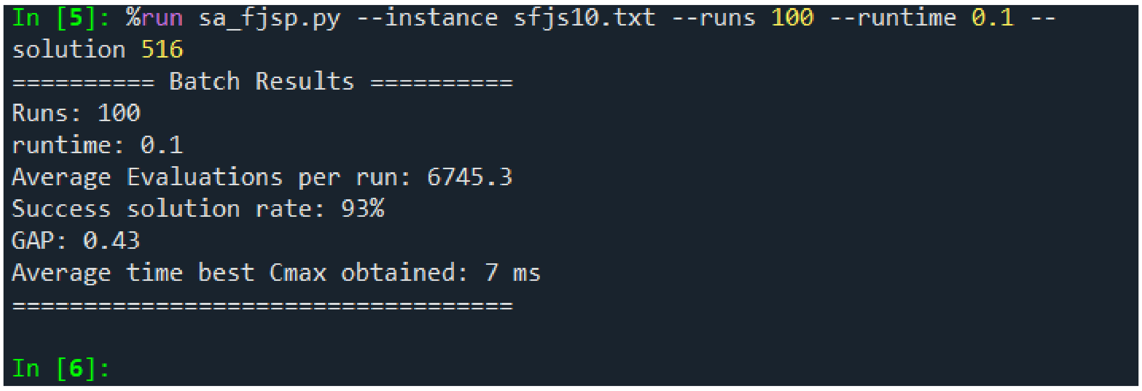


```
In [3]: %run sa_fjsp.py --instance mfjs01.txt --runs 1000 --evals 500 --noimp
100 --lb 468
========== Batch Results ==========
Runs: 1000
best_cmax among all runs: 468 (best cmax obtained in all 1000 runs)
median best_cmax:570 (median cmax of all 1000 runs)

Times no improvement happen: 99.8% (times stop by no improvement)
Times all evaluations finished: 0% (times finished 500 evals)

median LB-GAP: 0.22 ((median_cmax-lb)/lb)
AVG CPU time per run: 4 ms
===================================
```

Figure A5. Results from third run mode

The fourth and fifth mode consist in running the algorithm for a determined amount of time, one for known solutions and the other for unknown solutions. The structure and results are presented in figure A6 and A7.

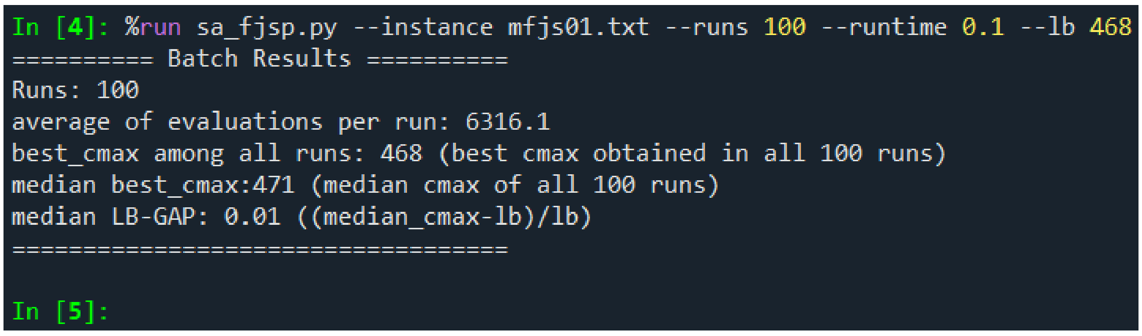
%run sa_fjsp.py --instance sfjs10.txt --runs 100 --runtime 0.1 --solution 516



```
In [5]: %run sa_fjsp.py --instance sfjs10.txt --runs 100 --runtime 0.1 --
solution 516
========== Batch Results ==========
Runs: 100
runtime: 0.1
Average Evaluations per run: 6745.3
Success solution rate: 93%
GAP: 0.43
Average time best Cmax obtained: 7 ms
===================================

In [6]:
```

Figure A6. Results from fourth run mode

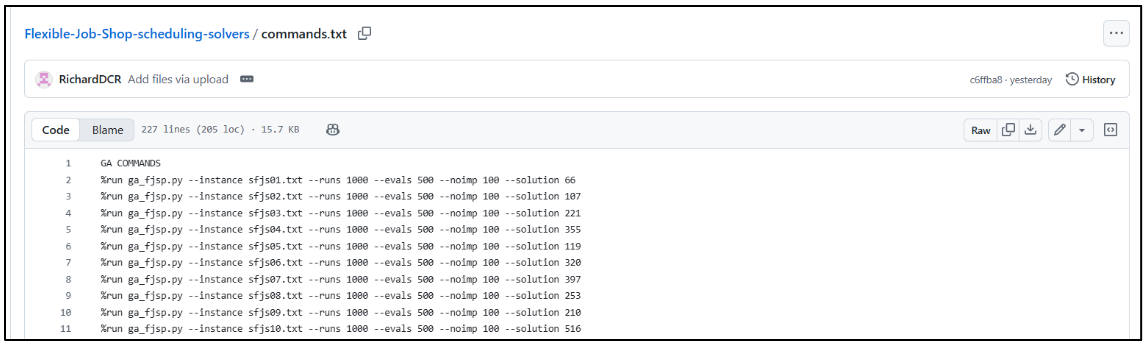
%run sa_fjsp.py --instance mfjs01.txt --runs 100 --runtime 0.1 --lb 468



```
In [4]: %run sa_fjsp.py --instance mfjs01.txt --runs 100 --runtime 0.1 --lb 468
========== Batch Results ==========
Runs: 100
average of evaluations per run: 6316.1
best_cmax among all runs: 468 (best cmax obtained in all 100 runs)
median best_cmax:471 (median cmax of all 100 runs)
median LB-GAP: 0.01 ((median_cmax-lb)/lb)
===================================

In [5]:
```

Figure A7. Results from fifth run mode

In the GitHub repository the commands used for the project can be found in the file “commands.txt”, figure A8 shows the example on the web.


Flexible-Job-Shop-scheduling-solvers / commands.txt
RichardDCR Add files via upload
c6ffba8 · yesterday
History
Code
Blame
227 lines (205 loc) · 15.7 KB
Raw

```
GA COMMANDS
%run ga_fjsp.py --instance sfjs01.txt --runs 1000 --evals 500 --noimp 100 --solution 66
%run ga_fjsp.py --instance sfjs02.txt --runs 1000 --evals 500 --noimp 100 --solution 107
%run ga_fjsp.py --instance sfjs03.txt --runs 1000 --evals 500 --noimp 100 --solution 221
%run ga_fjsp.py --instance sfjs04.txt --runs 1000 --evals 500 --noimp 100 --solution 355
%run ga_fjsp.py --instance sfjs05.txt --runs 1000 --evals 500 --noimp 100 --solution 119
%run ga_fjsp.py --instance sfjs06.txt --runs 1000 --evals 500 --noimp 100 --solution 320
%run ga_fjsp.py --instance sfjs07.txt --runs 1000 --evals 500 --noimp 100 --solution 397
%run ga_fjsp.py --instance sfjs08.txt --runs 1000 --evals 500 --noimp 100 --solution 253
%run ga_fjsp.py --instance sfjs09.txt --runs 1000 --evals 500 --noimp 100 --solution 210
%run ga_fjsp.py --instance sfjs10.txt --runs 1000 --evals 500 --noimp 100 --solution 516
```

Figure A8. Commands used on the project