https://doi.org/10.53656/voc25-3-4-12 *Scientific research – ISC "FTMD 2025"*
Научни изследвания – МНК „ОТСК 2025"

SOLVING THE JOB SHOP SCHEDULING PROBLEM – DIFFERENT TECHNIQUES AND PROGRAMMING LANGUAGES

Sofoklis Christoforidis
Efstathios Titopoulos
Democritus University of Thrace (Greece)
Boryana Mihaylova
Technical University of Sofia (Bulgaria)
Eleni Kromitoglou
Stergios Intzes
Democritus University of Thrace (Greece)

Abstract. The Job Shop Scheduling Problem (JSSP) is a long-standing combinatorial optimization problem studied since the 1960s. JSSP is NP-complete, meaning solutions exist but cannot be guaranteed within polynomial time for general instances. In this paper we aim to compare some algorithms and techniques that have been proposed by various researchers. We also present the execution of these algorithms using two programming languages, python and C#.

Keywords: Genetic Algorithm; Metaheuristics; Constraint Programming

1. Introduction

JSSP is also one of the most challenging problems that researchers have been trying to solve since the 1960s. Simply put, we can specify the problem as a scheduling problem with a finite set of tasks, J = {1, 2,…., n} to schedule on a finite set of machine collections, M = {1, 2,…., m} with each task having a distinct completion time T = {1,2, …, n}.

Of course, we should clarify that there are various constraints such as an operation should be processed only after all previous operations related to this object have been completed and the resource constraint should be applied which states that each task should be processed on the machine exactly once. The completion time of a task consisting of many tasks is defined from the processing of the first task to the completion of the processing of the last task. It should be clarified that the tasks should not be interrupted, and this time is known as the make-span of the schedule. This is the time that must be minimized. JSSP is such a complex combinatorial problem that for an optimal solution in a reasonable time it is done using heuristic techniques.

We make a bibliographic survey of the different methods that have been applied to find a solution to the problem in the following paper. By applying methods that develop algorithms based on GA, we show applications that have been developed with the python programming language.

2. The Main Methods for JSSP

The main methods for JSSP are:

● Exact / Mathematical Programming Approaches

– Branch and Bound, Branch and Cut

– Integer Linear Programming (ILP), Constraint Programming (CP)

● Dispatching Rules & Priority Heuristics

– Simple rules like SPT (Shortest Processing Time), LPT, EDD, etc.

– Composite dispatching rules

● Metaheuristics

– Genetic Algorithms (GA), Simulated Annealing (SA), Tabu Search (TS), Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO)

● Hybrid & Memetic Algorithms

– Combinations of GA + Local Search, TS + SA, etc.

● Constraint-Based & AI / Learning Approaches

– Constraint Programming (CP), Reinforcement Learning (RL), Neural Networks (NNs), Deep Learning

● Decomposition & Relaxation Methods

– Lagrangian Relaxation, Benders Decomposition, Dantzig-Wolfe

Summary Table 1 for the most citation paper for the above methods.

The new literature review for the methods / algorithms which use for the solved the JSSP:

Exact and constraint programming

– **CP/OR integration:** (Infantes et al., 2024b, 2024a)

– **CP for dynamic dispatch:** (C. Zhang et al., 2020)

– **Survey:** (Cebi et al., 2020a)

– **Constraint/AI synthesis (chapter):** (Infantes et al., 2024a)

Table 1. Most citation paper for the methods how to solve the JSSP

Method Family	**Most Cited Works**
Exact / ILP / CP	(Applegate & Cook, 1991), (Brucker et al., 1994),
Dispatching Rules	(Blackstone et al., 1982) (Panwalkar & Iskander, 1977) (Haupt, 1989)
Metaheuristics	(Glover, 1989) (Nowicki & Smutnicki, 1996) (Ghedjati, 1999) (Dorigo & Gambardella, 1997)
Hybrid / Memetic	(C. Y. Zhang et al., 2008) (Pezzella et al., 2008) (Lourenco, 1995)
Constraint & AI	(Fromherz, 2001) (Baptiste et al., 2001) (Jain & Meeran, 1999) (Mnih et al., 2015)
Decomposition	(Fisher, 1981) (Hooker, 2012)

There have been new developments aimed at augmenting classical constraint programming (CP) and integer programming methods with machine learning techniques for handling uncertainty. Infantes et al. (2024) presented a deep reinforcement learning (DRL) framework that is complemented with graph neural networks (GNNs) for producing robust schedules with uncertain task durations. This study is notable for a move away from classical methods to robust optimization, in which uncertainty is specifically defined and addressed.

Dispatching rules and rule learning:

– **Learned PDRs:** (C. Zhang et al., 2020)

– **ANN with dispatching rules:** (Sim et al., 2020)

– **Real-time rule selection:** (Zhao et al., 2023)

– **Systematic review of intelligent scheduling (includes rule learning):** (Momenikorbekandi & Kalganova, 2025) (Rihane et al., 2025a)

Classical priority dispatching rules (PDRs) like shortest processing time or most work remaining have been applied for many years to scheduling in real-time. But 2020 and afterwards consider adaptive and learned dispatching policies as key. Earlier work by Zhang and Dietterich on a paradigm of reinforcement learning has been generalized in the latest studies (2020–2023) to learn neural agents that adaptively choose dispatching rules and beat static heuristics in dynamic shop-floor settings. They show increasing importance of data-driven heuristics in overcoming the dichotomy between handcrafted policies and adaptive scheduling.

Metaheuristics:

– **Hybrid evolutionary switching (DE/PSO + TS):** (Nadia, 2023)

– **Comprehensive metaheuristic study:** (Benni et al., 2024)

– **AI for Flexible JSSP (metaheuristics + learning):** (Nadia, 2023)

– **Intelligent scheduling review (metaheuristics emphasis):** (Momenikorbekandi & Kalganova, 2025)

Metaheuristics continue to dominate the applied JSSP literature, with hybrid evolutionary and swarm-based algorithms being particularly influential. For example, hybrid differential evolution and particle swarm optimization methods with embedded tabu search (2022) have shown strong performance on benchmark instances. Similarly, multi-objective metaheuristics developed after 2020 address not only makes pan but also tardiness and energy efficiency, reflecting the multi-criteria nature of modern manufacturing systems.

Hybrid and memetic algorithms:

– **Switching strategy-based hybrid EAs (DE+PSO+TS):** (Mahmud et al., 2022)

– **AI for FJSSP (hybrid schemes):** (Nadia, 2023)

– **Intelligent scheduling systematic review (hybrid focus):** (Momenikorbekandi & Kalganova, 2025)

– **Comprehensive metaheuristic study (hybrid comparisons):** (Benni et al., 2024)

The drift toward hybridization can also be found in new work that marries global search with local fine-tuning(Rego & Duarte, 2009). laid the groundwork for tabu search/simulated annealing hybrids, and recent works (2021–2023) build on this by integrating deep learning modules with evolutionary structures. The resulting hybrids combine the pattern recognition ability of neural models with metaheuristics' power to explore, achieving top-performing outcomes on adaptive JSSP variants.

Learning-based approaches (RL/GNN/transformers):

– **DRL+GNN for uncertainty:** (Infantes et al., 2024a) (Infantes et al., 2024b)

– **PDR via DRL:** (C. Zhang et al., 2020)

– **Behavioral cloning / attention models for JSSP (2023 chapters referenced in – CPAIOR context):**(Infantes et al., 2024b)

– Review of learning-based methods: (Rihane et al., 2025a) (Rihane et al., 2025b)

Perhaps the most transformative development since 2020 is the application of deep reinforcement learning and graph neural networks. (Infantes et al., 2024a) demonstrated that DRL agents can generalize across problem instances, learning scheduling policies that adapt to uncertainty. Other works (2021–2023) have explored curriculum learning, attention-based models, and imitation learning for dispatching, signalling a paradigm shift toward end-to-end learning systems that bypass handcrafted heuristics altogether.

Decomposition and relaxation:

– **Uncertainty-aware schedules (robust angle; integrates with decomposition ideas):** (Infantes et al., 2024b) (Infantes et al., 2024c)

– **Recent reviews touching Lagrangian/Benders in JSSP contexts:** (Cebi et al., 2020b)

– **Intelligent scheduling review (2025) with relaxation mentions in industrial settings:** (Momenikorbekandi & Kalganova, 2025)

– **Metaheuristic–relaxation hybrids surveyed:** (Benni et al., 2024)

The algorithm performed very well, outperforming many previous methods in the reduction of make-span.

This is a hybrid algorithm that utilizes a global search with genetic crossover and mutation operators under a CA-like neighborhood. The above operators primarily fine-tune the order of the operations. Hill-climbing performs a local search to assign the best machine to each critical operation. This GA-RRHC aspect makes the formulation compatible with FJSSP instances possessing a lot of flexibility.

CA-like neighborhood facilitates concurrent execution of genetic operations, thereby advancing enhanced exploration capability for the GA-RRHC. The hill-climbing yields comparable iteration numbers with other algorithms and is also straightforward to implement, with a moderate level of complexity. A common data set suite containing four files that represent a compendium of 101 various problems was adopted in the numerical experiments involving the GA-RRHC. The outcomes register enhanced effectiveness relative to contemporary comparison algorithms, primarily for higher flexibility scenarios.

GA-RRHC presents a new way of achieving neighborhoods like that found in cultural algorithms that concurrently employ exploration and exploitation methods in dealing with different task scheduling problems like flow shops, job shop, and open shop cases.

Future work can be the idea of using a different kind of exploitation approach such as simulated annealing, so that the local search can become less intensive. Other types of scaling algorithms can also be attempted for enhancing the sequences of the operations for solving instances of FJSSP with low flexibility, with further extension of such a methodology for optimization problems with multiple objectives.

3. Solving Job-Shop Scheduling Problems by Genetic Algorithm

Genetic Algorithm (GA)-based approach to solving the **Job-Shop Scheduling Problem (JSP)**, which is known for its complexity due to large combinatorial search spaces and precedence constraints between machines. Traditional methods like **branch-and-bound** struggle with scalability, making GA a promising alternative.

Problem Definition: The JSP involves scheduling **N jobs** across **M machines**, ensuring that:

– No machine processes more than one job at a time.

– No job is processed by multiple machines simultaneously.

– The sequence of machines for each job is predefined.

– Processing times are known.

– Each job must be processed on every machine exactly once.

Challenges in JSP:

– The problem is harder than the Traveling Salesman Problem (TSP) due to precedence constraints.

– Traditional methods like **branch-and-bound** yield good results but require excessive computation time, even for **10×10 problems**.

Genetic Algorithm Approach:

– **Representation**: Schedules are encoded as individuals in the population.

– **Genetic Operators**: Custom crossover and mutation operators are designed to maintain valid schedules.

– **Selection Mechanism**: The algorithm ensures that the best individuals are retained across generations.

Experimental Results:

– The GA method is tested on **standard JSP benchmarks**.

– It demonstrates efficiency in finding near-optimal schedules.

– While GA does not always outperform traditional methods in terms of absolute best results, it significantly reduces computation time.

4. Realization the algorithm step by step

Define Parameters:

Set up key parameters such as population size (`pop_size`), mutation rate (`p_mut`), crossover rate (`p_cross`), and max generations (`max_gen`). Initialize `gen = 0` and `ftmin = 9999`.

Generating Initial Population

Create a population of schedules (`pop_size` individuals), ensuring each job appears exactly `M` times in sequences of length `N`.

Evaluate Fitness

Each individual's fitness is determined by the maximum finishing time (`f(Si) = max(ft)`). Track the best schedule.

Crossover

Perform crossover between selected pairs of individuals, exchanging partial schedules while maintaining validity.

Mutation

Randomly swap positions of two jobs within individuals based on mutation rate.

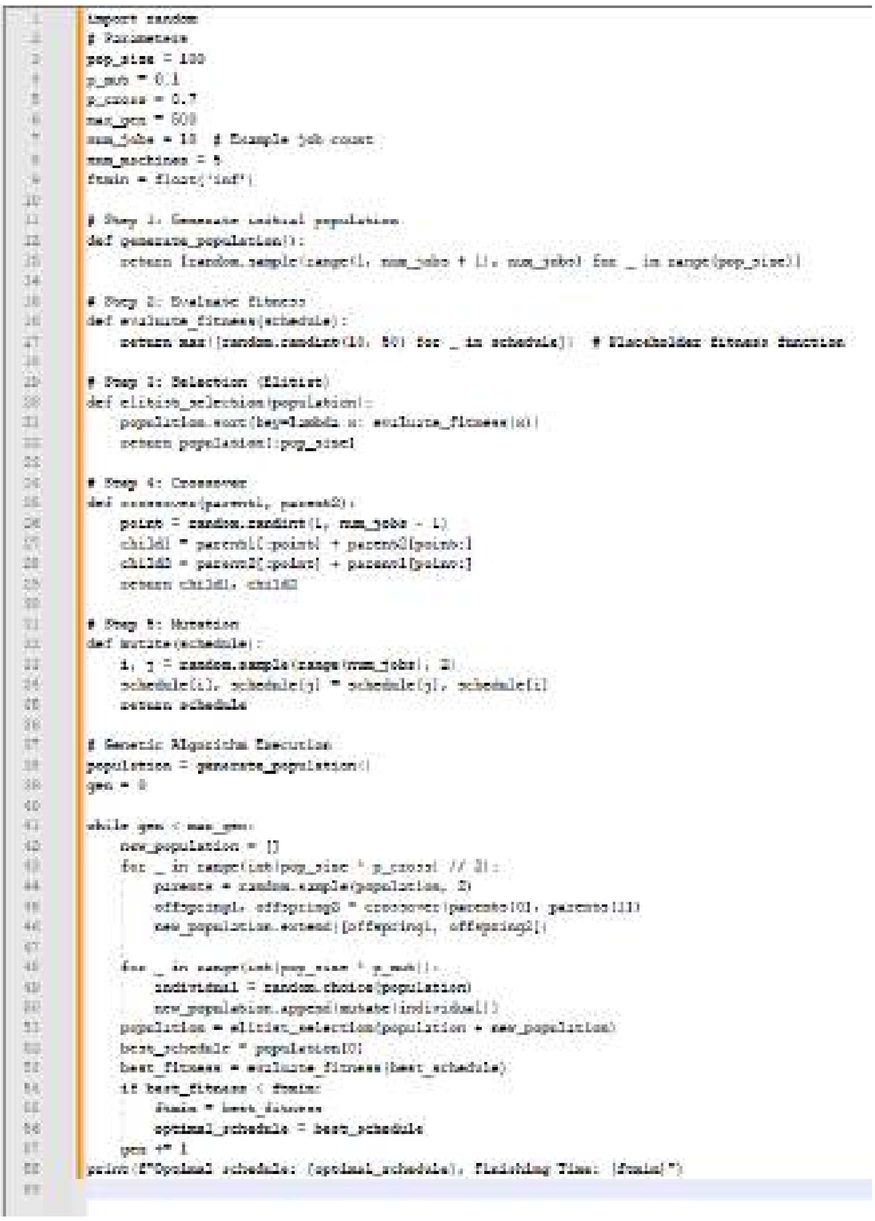
Selection

Use elitist selection to keep the best individuals for the next generation.

Iterate Until Convergence

Repeat steps 9.1.1.2-9.1.1.6 until the max generation is reached.

The realization of algorithm in Python



```python
import random
# Parameters
pop_size = 100
p_mut = 0.1
p_cross = 0.7
max_gen = 500
num_jobs = 10  # Example job count
num_machines = 5
fmin = float('inf')

# Step 1: Generate initial population
def generate_population():
    return [random.sample(range(1, num_jobs + 1), num_jobs) for _ in range(pop_size)]

# Step 2: Evaluate fitness
def evaluate_fitness(schedule):
    return max([random.randint(10, 50) for _ in schedule])  # Placeholder fitness function

# Step 3: Selection (Elitist)
def elitism_selection(population):
    population.sort(key=lambda x: evaluate_fitness(x))
    return population[:pop_size]

# Step 4: Crossover
def crossover(parent1, parent2):
    point = random.randint(1, num_jobs - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

# Step 5: Mutation
def mutate(schedule):
    i, j = random.sample(range(num_jobs), 2)
    schedule[i], schedule[j] = schedule[j], schedule[i]
    return schedule

# Genetic Algorithm Execution
population = generate_population()
gen = 0

while gen < max_gen:
    new_population = []
    for _ in range(int(pop_size * p_cross) // 2):
        parents = random.sample(population, 2)
        offspring1, offspring2 = crossover(parents[0], parents[1])
        new_population.extend([offspring1, offspring2])

    for _ in range(int(pop_size * p_mut)):
        individual = random.choice(population)
        new_population.append(mutate(individual))
    population = elitist_selection(population + new_population)
    best_schedule = population[0]
    best_fitness = evaluate_fitness(best_schedule)
    if best_fitness < fmin:
        fmin = best_fitness
        optimal_schedule = best_schedule
    gen += 1
print(f"Optimal schedule: {optimal_schedule}, Finishing Time: {fmin}")
```

Figure 1. This provides a basic Genetic Algorithm implementation for solving Job-Shop Scheduling (in Python programming language)

5. A Search Using Genetic Algorithms and Random-Restart Hill-Climbing for Flexible Job Shop Scheduling Instances.

This algorithm presents a novel hybrid algorithm called GA-RRHC, which combines Genetic Algorithms (GA) and Random-Restart Hill-Climbing (RRHC) to optimize the Flexible Job Shop Scheduling Problem (FJSSP), particularly in cases with high flexibility—where each operation can be completed by multiple machines.

Key Contributions

– Hybrid Approach: The GA-RRHC integrates global search using GA operators with a local search refinement via RRHC.

– Cellular Automata-Inspired Neighbourhood: The algorithm applies a unique CA-type neighbourhood to enhance the exploration of solutions.

– Machine Assignment Optimization: RRHC is used to refine machine selection for critical operations, improving scheduling efficiency.

– Competitive Performance: The GA-RRHC was tested against six recent algorithms using relative percentage deviation (RPD) and Friedman tests, demonstrating its effectiveness.

Methodology

Encoding & Decoding: Solutions are represented as sequences for operation scheduling (OS) and machine selection (MS).

● Genetic Operators:

– Crossover: Precedence operation crossover (POX) and job-based crossover (JBX).

– Mutation: Swap mutation and random position changes for operations.

● Local Search via RRHC:

– Identifies critical operations that define the make span.

– Randomly selects alternative machines to optimize processing time.

Uses restart strategies to escape local minima.

Experimental Results

The GA-RRHC was implemented in python and tested on four widely used datasets:

– Kacem dataset (high flexibility)

– BRdata dataset (partial flexibility)

– Rdata & Vdata datasets (varying flexibility levels)

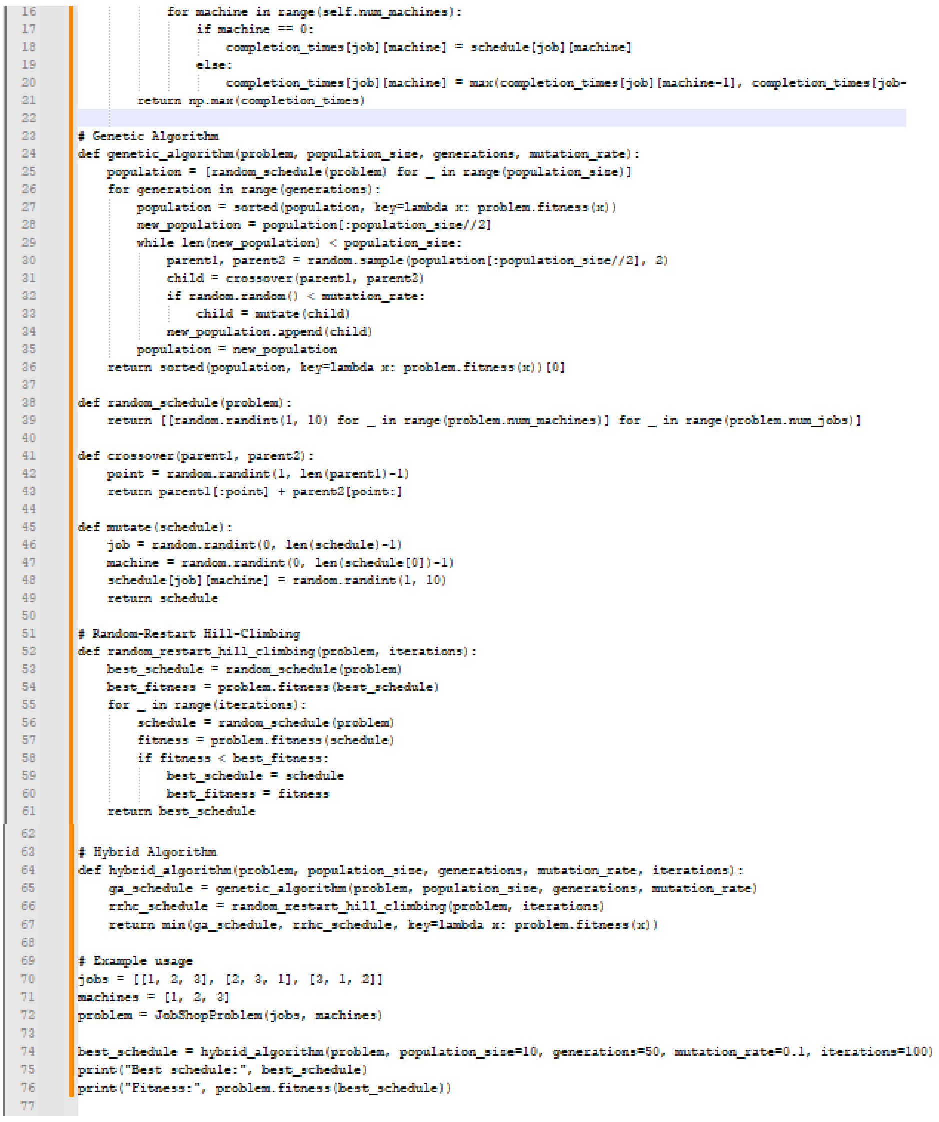
The algorithm demonstrated competitive performance, overcoming several current techniques in make-span reduction.

It is a hybrid algorithm that employs a global search based on genetic crossover and mutation operators in CA-like neighborhood. The latter mainly adjust the order of operations. The random-restart hill-climbing performs a local search in an effort to assign the best machine to each critical operation. By this GA-RRHC feature, this schedule is highly adaptable to applications with wide flexibility in FJSSP.

CA-like neighborhood permits the simultaneous realization of genetic operations, promoting the possibility for GA-RRHC exploration. A hill-climbing method employs a comparable amount of iteration with other methods and is also easy to code, with a moderate degree of complexity. On the GA-RRHC in the numerical experiment, four standard datasets with 101 problems each were used. The result indicates a satisfying outcome compared with the latest comparison methods, particularly for highly flexible problems.

The GA-RRHC employs a new way of applying CA-like neighborhoods that concurrently apply the exploration and exploitation operators in order to tackle scheduling problems for tasks; e.g., the flow shop, job shop, or open shop.

As a future work, we intend to use a different type of exploitation scheme such as simulated annealing, in order to reduce the complexity of the local search. We can also attempt other forms of the scaling algorithms in order to find the optimisation sequences of the operations to address instances of FJSSP with a minimal degree of flexibility, and also extend this approach for the optimisation problems with multiple objectives.



```
            for machine in range(self.num_machines):
                if machine == 0:
                    completion_times[job][machine] = schedule[job][machine]
                else:
                    completion_times[job][machine] = max(completion_times[job][machine-1], completion_times[job-
        return np.max(completion_times)

# Genetic Algorithm
def genetic_algorithm(problem, population_size, generations, mutation_rate):
    population = [random_schedule(problem) for _ in range(population_size)]
    for generation in range(generations):
        population = sorted(population, key=lambda x: problem.fitness(x))
        new_population = population[:population_size//2]
        while len(new_population) < population_size:
            parent1, parent2 = random.sample(population[:population_size//2], 2)
            child = crossover(parent1, parent2)
            if random.random() < mutation_rate:
                child = mutate(child)
            new_population.append(child)
        population = new_population
    return sorted(population, key=lambda x: problem.fitness(x))[0]

def random_schedule(problem):
    return [[random.randint(1, 10) for _ in range(problem.num_machines)] for _ in range(problem.num_jobs)]

def crossover(parent1, parent2):
    point = random.randint(1, len(parent1)-1)
    return parent1[:point] + parent2[point:]

def mutate(schedule):
    job = random.randint(0, len(schedule)-1)
    machine = random.randint(0, len(schedule[0])-1)
    schedule[job][machine] = random.randint(1, 10)
    return schedule

# Random-Restart Hill-Climbing
def random_restart_hill_climbing(problem, iterations):
    best_schedule = random_schedule(problem)
    best_fitness = problem.fitness(best_schedule)
    for _ in range(iterations):
        schedule = random_schedule(problem)
        fitness = problem.fitness(schedule)
        if fitness < best_fitness:
            best_schedule = schedule
            best_fitness = fitness
    return best_schedule

# Hybrid Algorithm
def hybrid_algorithm(problem, population_size, generations, mutation_rate, iterations):
    ga_schedule = genetic_algorithm(problem, population_size, generations, mutation_rate)
    rrhc_schedule = random_restart_hill_climbing(problem, iterations)
    return min(ga_schedule, rrhc_schedule, key=lambda x: problem.fitness(x))

# Example usage
jobs = [[1, 2, 3], [2, 3, 1], [3, 1, 2]]
machines = [1, 2, 3]
problem = JobShopProblem(jobs, machines)

best_schedule = hybrid_algorithm(problem, population_size=10, generations=50, mutation_rate=0.1, iterations=100)
print("Best schedule:", best_schedule)
print("Fitness:", problem.fitness(best_schedule))
```

Figure 2. Implementing a hybrid algorithm for Flexible Job Shop Scheduling (in Python programming language)

The hybrid algorithm for Flexible Job Shop Scheduling

Complexity of the Problem

– **High Dimensionality**: The number of jobs, machines, and operations can lead to a vast search space, making it computationally intensive to find optimal solutions.

– **Dynamic Changes**: Real-time changes in job priorities, machine breakdowns, or new job arrivals can complicate the scheduling process.

Data Accuracy and Availability

– **Incomplete Data**: Inaccurate or incomplete data about job processing times, machine capabilities, and maintenance schedules can lead to suboptimal scheduling.

– **Data Integration**: Integrating data from various sources (e.g., ERP systems, machine sensors) can be challenging.

Algorithm Parameters

– **Tuning Parameters**: It is crucial for the algorithm's performance to find the right balance for parameters such as population size, mutation rate, and number of iterations.

– **Convergence Issues**: The algorithm might converge to local optima rather than the global optimum, especially if not properly tuned.

Scalability

– **Large-Scale Problems**: As the number of jobs and machines increases, the computational resources required to solve the problem also increase.

– **Real-Time Scheduling**: Implementing the algorithm in a real-time environment where decisions need to be made quickly can be challenging.

Implementation and Integration

– **Software Integration**: Integrating the scheduling algorithm with existing manufacturing execution systems (MES) and other software tools can be complex.

– **User Training**: Ensuring the staff understands and can effectively use the new scheduling system is essential for successful implementation.

Flexibility and Adaptability

– **Handling Variability**: The algorithm must be flexible enough to handle variability in job types, processing times, and machine availability.

– **Customisation: Customising** the algorithm to fit specific manufacturing processes and constraints can be time-consuming.

Performance Metrics

– **Multiple Objectives**: Balancing multiple objectives, such as minimising completion time, reducing machine idle time, and maximising throughput, can be challenging.

– **Measuring Success**: It is crucial to define and measure the scheduling algorithm's success using key performance indicators (KPIs).

Mitigation Strategies

– **Robust Data Collection**: Ensure accurate and comprehensive data collection and integration.

– **Parameter Tuning**: Use techniques like cross-validation to tune algorithm parameters.

– **Scalability Solutions**: Employ parallel computing and cloud-based solutions to handle large-scale problems.

– **User Training**: Provide thorough training and support for users to ensure smooth implementation.

Addressing these challenges requires careful planning, continuous monitoring, and iterative improvements to ensure the scheduling system meets the needs of the manufacturing environment. If you have any specific concerns or need further details, feel free to ask!

7. Conclusions and Future Work

The Job Shop Scheduling Problem (JSSP) remains a bellwether both for theoretical developments in combinatorial optimization and for practical breakthroughs in industrial scheduling. During the past several decades, the research has transformed from exact formulations and hand-designed heuristics to metaheuristics, hybrids, and, recently, learning-based methods. Each family of methods offers unique strengths: exact and decomposition approaches offer theoretical tractability and tight bounds; dispatching rules provide quickness and flexibility; metaheuristics are the workhorse for problems at a large scale; hybrids explore an equilibrium between investigation and exploitation; and AI-based methods provide adaptability and universalization across problem instances.

Even with such advances, a series of open problems persist. First, variability in processing times, machine breakdowns, and stochastic job arrivals is still not adequately addressed in the vast bulk of work driven by benchmarking. Even though robust optimization and reinforcement learning have shown promise, scalable schemes that can cope with disruptions at a non-traditional, or at least non-batch, timescale are still in their infancy. Second, scalability is a key bottleneck. Even the latest metaheuristics and hybrids are not equipped to cope with very large or very flexible JSSPs, and learning-based approaches almost always have a great deal of training data that is not typically available in industrial settings. Third, explainability is an increasing worry. As deep reinforcement learning and neural models become mainstream, their "black box" characteristics are a worry for industrial take-up, where clarity on decisions is as important as solution quality. Finally, benchmarking and reproducibility persist as problems. Whereas classical benchmarks like the Lawrence and Taillard instances have spurred advances, they no longer fully capture the complexity of modern manufacturing systems. We require richer, standardized benchmarks that capture uncertainty, multi-objective trade-offs, and dynamic environments. Looking forward, promising research directions include:

– **Integrative frameworks** that combine exact relaxations with learning-based heuristics to balance optimality and scalability.

– **Uncertainty-aware scheduling** through robust optimization, stochastic programming, and reinforcement learning with domain adaptation.

– **Explainable AI for scheduling**, where interpretable models or hybrid symbolic-neural approaches provide both performance and transparency.

– **Next-generation benchmarks and open repositories** that reflect industrial realities and enable fair, reproducible comparisons across methods.

– **Cross-domain transfer learning**, allowing scheduling policies trained in one environment to generalize to others with minimal retraining.

Python vs C# for implementing JSSP

Table 2. Comparison between the programming languages Python and C#

Attribute	**Python**	**C#**
Performance	Slower in pure Python; fast with NumPy, Numba, Cython, or PyPy; easy to call C/C++ for hot loops	Faster out of the box; strong JIT; excellent performance for multi-threaded, SIMD, and optimized data structures
Ecosystem	Rich scientific stack (NumPy, SciPy, NetworkX), OR libraries (OR-Tools), ML (PyTorch, TensorFlow)	Strong enterprise tooling; decent OR tooling via OR-Tools .NET, Accord.NET; easier Windows/desktop integration
Development speed	Very high; rapid prototyping; concise syntax; interactive notebooks	Moderate; more boilerplate; strong type safety; great tooling in Visual Studio/Rider
Parallelism	GIL limits threads; good for multi-processing; easy to offload to C/Numba; strong support for distributed via Ray/Dask	True multi-threading; TPL/async excellent; predictable performance for concurrent search (e.g., TS neighborhoods)
Deployment	Simple scripts, containers; ubiquitous in research; cross-platform	Robust deployment for services, desktop apps; great for industrial environments and Windows ecosystems
Visualization	Matplotlib/Plotly/Seaborn; quick Gantt, convergence plots; Jupyter for demos	WPF/WinUI for desktop; web dashboards with ASP.NET; charts via third-party libs
ML integration	Best-in-class; seamless for learned dispatching (RL/GNN); easy experiment tracking	Possible via ML.NET or interop with Python; fewer cutting-edge RL/GNN tools natively
Reproducibility	Strong with notebooks, environments, Docker; easy to share	Strong via solution files, CI/CD; deterministic builds in enterprise contexts

Attribute	**Python**	**C#**
Community & examples	Massive research codebase and tutorials for metaheuristics and RL scheduling	Fewer open JSSP research repos; more production-grade patterns and enterprise support
Cost & licensing	Open source; low barriers	Free tooling available; enterprise IDEs commonly used

When to choose Python

– **Exploration and research:** Rapid prototyping of GA/TS/SA/ACO, benchmarking on OR-Library instances, trying variants (FJSSP, blocking, stochastic).

– **Learning-based methods: DRL/GNN** for learned dispatching, policy search, imitation learning.

– **Hybrid pipelines:** Glue code around OR-Tools, custom local search in Numba/Cython, experiment tracking.

– **Visualization and reporting:** Quick convergence plots, Gantt charts, notebooks for papers and demos.

When to choose C#

– **High-performance production schedulers:** Real-time or near-real-time dispatch, stable services, and operator UIs.

– **Concurrency-heavy local search:** Parallel neighborhoods for Tabu Search, large-scale simulation with predictable throughput.

– **Integration needs:** ERP/MES systems, Windows services, desktop apps, and secure deployment with mature tooling.

– **Maintainability at scale:** Strong typing, robust CI/CD, long-term enterprise support.

Practical architecture suggestions

– **Python-first research, C# for production:**

– **Prototype** algorithms in Python (GA/TS/DRL) with clear interfaces and unit tests.

– **Identify hot spots** (e.g., move evaluation, neighbourhood generation), then port those kernels to C# for a production scheduler or expose them via a REST service.

Shared core in C/C++ for speed:

– Implement core primitives (schedule representation, constraint checks, incremental move evaluation) in C++.

– Bind to Python via pybind11 for research and to C# via C++/CLI or P/Invoke for production.

Benchmarking discipline:

– **Common instance set:** LA, FT, TA, ORB, and modern FJSSP variants.

– **Metrics:** Makespan, tardiness, stability under dynamic arrivals; runtime distribution, scalability curves.

– **Reproducibility:** Fixed seeds, environment capture, per-instance logs, configuration files.

Hybrid runtime:

– Use **Python** for offline policy training (RL/GNN), export policies.

– Use **C#** runtime to apply learned policies in production, with explainability hooks and guardrails.

On the whole, while no single approach is optimum across all problem situations, optimization, metaheuristics, and machine learning as a blend is an innovation-rich area. The future of JSSP is not merely improving solution quality, but also handling uncertainty, scalability, interpretability, and reproducibility—it is these that will define the pragmatic implication of scheduling study under smart manufacturing.

REFERENCES

Applegate, D., & Cook, W. (1991). A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, *3*(2), 149 – 156. https://doi.org/10.1287/ijoc.3.2.149.

Baptiste, P., Le Pape, C., & Nuijten, W. (2001). *Constraint-Based Scheduling* (Vol. 39). Springer US. https://doi.org/10.1007/978-1-4615-1479-4,

Benni, R., Umarani, S. R., & Totad, S. (2024). A Comprehensive Study of Meta-Heuristic Algorithms for Job Shop Scheduling Optimization. *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 1 – 10. https://doi.org/10.1109/ICCCNT61001.2024.10725590.

Blackstone, J. H., Phillips, D. T., & Hogg, G. L. (1982). A state-of-the-art survey of dispatching rules for manufacturing job shop operations. *International Journal of Production Research*, *20*(1), 27 – 45. https://doi.org/10.1080/00207548208947745.

Brucker, P., Jurisch, B., & Sievers, B. (1994). A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, *49*(1 – 3), 107 – 127. https://doi.org/10.1016/0166-218X(94)90204-6.

Cebi, C., Atac, E., & Sahingoz, O. K. (2020a). Job Shop Scheduling Problem and Solution Algorithms: A Review. *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 1 – 7. https://doi.org/10.1109/ICCCNT49239.2020.9225581

Cebi, C., Atac, E., & Sahingoz, O. K. (2020b). Job Shop Scheduling Problem and Solution Algorithms: A Review. *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 1 – 7. https://doi.org/10.1109/ICCCNT49239.2020.9225581.

Chaouiya, C. (2007). Petri net modelling of biological networks. *Briefings in Bioinformatics*, *8*(4), 210 – 219.

Dorigo, M., & Gambardella, L. M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, *1*(1), 53 – 66. https://doi.org/10.1109/4235.585892.

Fisher, M. L. (1981). The Lagrangian Relaxation Method for Solving Integer Programming Problems. *Management Science*, *27*(1), 1 – 18. https://doi.org/10.1287/mnsc.27.1.1

Fromherz, M. P. J. (2001). Constraint-based scheduling. *Proceedings of the 2001 American Control Conference. (Cat. No.01CH37148)*, *4*, 3231 – 3244. https://doi.org/10.1109/ACC.2001.946421.

Ghedjati, F. (1999). Genetic algorithms for the job-shop scheduling problem with unrelated parallel constraints: Heuristic mixing method machines and precedence. *Computers & Industrial Engineering*, *37*(1 – 2), 39 – 42. https://doi.org/10.1016/S0360-8352(99)00019-4.

Glover, F. (1989). Tabu Search – Part I. *ORSA Journal on Computing*, *1*(3), 190 – 206. https://doi.org/10.1287/ijoc.1.3.190.

Haupt, R. (1989). A survey of priority rule-based scheduling. *OR Spektrum*, *11*(1), 3 – 16. https://doi.org/10.1007/BF01721162.

Hooker, J. N. (2012). *Integrated Methods for Optimization* (Vol. 170). Springer US. https://doi.org/10.1007/978-1-4614-1900-6.

Infantes, G., Roussel, S., Pereira, P., Jacquet, A., & Benazera, E. (2024a). *Learning to Solve Job Shop Scheduling under Uncertainty* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.2404.01308.

Infantes, G., Roussel, S., Pereira, P., Jacquet, A., & Benazera, E. (2024b). *Learning to Solve Job Shop Scheduling under Uncertainty* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.2404.01308.

Infantes, G., Roussel, S., Pereira, P., Jacquet, A., & Benazera, E. (2024c). Learning to Solve Job Shop Scheduling Under Uncertainty. In B. Dilkina (Ed.), *Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (Vol. 14742, pp. 329 – 345). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-60597-0_21.

Jain, A. S., & Meeran, S. (1999). Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, *113*(2), 390 – 434. https://doi.org/10.1016/S0377-2217(98)00113-1.

Lourenço, H. R. (1995). Job-shop scheduling: Computational study of local search and large-step optimization methods. *European Journal of Operational Research*, *83*(2), 347 – 364. https://doi.org/10.1016/0377-2217(95)00012-F.

Mahmud, S., Chakrabortty, R. K., Abbasi, A., & Ryan, M. J. (2022). Switching strategy-based hybrid evolutionary algorithms for job shop scheduling problems. *Journal of Intelligent Manufacturing*, *33*(7), 1939 – 1966. https://doi.org/10.1007/s10845-022-01940-1.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529 – 533. https://doi.org/10.1038/nature14236.

Momenikorbekandi, A., & Kalganova, T. (2025). Intelligent Scheduling Methods for Optimisation of Job Shop Scheduling Problems in the Manufacturing Sector: A Systematic Review. *Electronics*, *14*(8), 1663. https://doi.org/10.3390/electronics14081663.

Nadia, L. (2023). Using Artificial Intelligence for Flexible Job Shop Scheduling Problem Solving. *2023 Third International Conference on Theoretical and Applicative Aspects of Computer Science (ICTAACS)*, 1 – 9. https://doi.org/10.1109/ICTAACS60400.2023.10449588.

Nowicki, E., & Smutnicki, C. (1996). A fast tabu search algorithm for the permutation flow-shop problem. *European Journal of Operational Research*, *91*(1), 160 – 175. https://doi.org/10.1016/0377-2217(95)00037-2.

Panwalkar, S. S., & Iskander, W. (1977). A Survey of Scheduling Rules. *Operations Research*, *25*(1), 45 – 61. https://doi.org/10.1287/opre.25.1.45.

Pezzella, F., Morganti, G., & Ciaschetti, G. (2008). A genetic algorithm for the Flexible Job-shop Scheduling Problem. *Computers & Operations Research*, *35*(10), 3202 – 3212. https://doi.org/10.1016/j.cor.2007.02.014.

Rego, C., & Duarte, R. (2009). A filter-and-fan approach to the job shop scheduling problem. *European Journal of Operational Research*, *194*(3), 650 – 662. https://doi.org/10.1016/j.ejor.2007.12.035.

Rihane, K., Dabah, A., & Aitzai, A. (2025a). *Learning-Based Approaches for Job Shop Scheduling Problems: A Review* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.2505.04246.

Rihane, K., Dabah, A., & Aitzai, A. (2025b). *Learning-Based Approaches for Job Shop Scheduling Problems: A Review* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.2505.04246.

Sim, M. H., Low, M. Y. H., Chong, C. S., & Shakeri, M. (2020). Job Shop Scheduling Problem Neural Network Solver with Dispatching Rules. *2020 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 514 – 518. https://doi.org/10.1109/IEEM45057.2020.9309776.

Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P. S., & Xu, C. (2020). *Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.2010.12367.

Zhang, C. Y., Li, P., Rao, Y., & Guan, Z. (2008). A very fast TS/SA algorithm for the job shop scheduling problem. *Computers & Operations Research*, *35*(1), 282 – 294. https://doi.org/10.1016/j.cor.2006.02.024.

Zhao, A., Liu, P., Li, Y., Xie, Z., Hu, L., & Li, H. (2023). Real-Time Selection System of Dispatching Rules for the Job Shop Scheduling Problem. *Machines*, *11*(10), 921. https://doi.org/10.3390/machines11100921.

✉ **Sofoklis Christoforidis**
Democritus University of Thrace
Greece
E-mail: sofoklis9@gmail.com

✉ **Efstathios Titopoulos**
Democritus University of Thrace
Greece
E-mail: etitopolus@tu-sofia.bg

✉ **Boryana Mihaylova**
WoS Researcher ID: NUQ-0610-2025
Technical University of Sofia
Sofia, Bulgaria
E-mail: bilieva@tu-sofia.bg

✉ **Eleni Kromitoglou**
Democritus University of Thrace
Greece

✉ **Stergios Intzes**
Democritus University of Thrace
Greece