A Hybrid Constrained Programming with Genetic Algorithm for the Job Shop Scheduling Problem

Alessandro Lorenzi
University of Trento
Trento, Italy
alessandro.lorenzi-1@studenti.unitn.it

Stefano Genetti
University of Trento
Trento, Italy
stefano.genetti@unitn.it

Chiara C. Rambaldi Migliore
University of Trento
Trento, Italy
cc.rambaldimigliore@unitn.it

Marco Roveri
University of Trento
Trento, Italy
marco.roveri@unitn.it

Giovanni Iacca
University of Trento
Trento, Italy
giovanni.iacca@unitn.it

Abstract

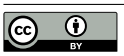
The Job Shop Scheduling Problem (JSSP) is a widely studied NP-hard optimization problem with significant academic and industrial relevance, particularly in the context of Industry 4.0, where efficient scheduling algorithms are crucial for improving decision-making in increasingly automated production systems. Despite extensive theoretical advancements, a gap remains between academic research and real-world implementation, as most studies either focus on theoretical aspects or emphasize numerical advantages while neglecting practical deployment challenges, including those related to computational constraints. To fill this gap, we propose a hybrid optimization approach, HCPGA, which integrates a state-of-the-art Constraint Programming (CP) solver, CP-SAT, with a custom Genetic Algorithm (GA). The CP solver generates feasible solutions, which are then used to initialize the GA's population. The GA further optimizes the schedule, minimizing the makespan. Our experimental evaluation on 74 JSSP benchmark instances of varying sizes demonstrates that, while standalone CP-SAT and HCPGA achieve comparable makespan results, the latter significantly reduces the time and memory required to find a good solution. This makes our approach highly valuable for industrial applications. To our knowledge, this is the first attempt to combine an evolutionary approach with an exact solver for solving the JSSP, specifically addressing the need for computational efficiency.

CCS Concepts

• **Theory of computation → Evolutionary algorithms**; **Scheduling algorithms**; • **Mathematics of computing → Combinatorial optimization**; • **Applied computing → Industry and manufacturing**.

Keywords

Job Shop Scheduling, Hybrid Algorithms, Constraint Programming, Genetic Algorithm, Industry 4.0



GECCO '25 Companion, July 14–18, 2025, Malaga, Spain

ACM ISBN 979-8-4007-1464-1/2025/07
https://doi.org/10.1145/3712255.3734284

ACM Reference Format:
Alessandro Lorenzi, Stefano Genetti, Chiara C. Rambaldi Migliore, Marco Roveri, and Giovanni Iacca. 2025. A Hybrid Constrained Programming with Genetic Algorithm for the Job Shop Scheduling Problem. In *Genetic and Evolutionary Computation Conference (GECCO '25 Companion), July 14–18, 2025, Malaga, Spain.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3712255.3734284

1 Introduction

Planning and resource allocation are fundamental challenges across all industries, making scheduling a combinatorial problem of central importance to practitioners [4, 7, 39]. In particular, as the *fourth industrial revolution*, commonly referred to as Industry 4.0, continues to drive higher levels of automation in cyber-physical ecosystems, efficient scheduling is becoming increasingly critical [31, 34]. As industries increasingly adopt advanced technologies to optimize processes, the ability to effectively schedule tasks plays a crucial role in improving key performance metrics and maintaining a competitive edge in the market [21].

Despite the relevance of scheduling in industrial domains and the large number of research studies on this topic, there is a significant gap between theory and practice. In fact, only a small percentage of the solutions proposed in the academic literature are effectively deployed in real-world scenarios. Research usually prioritizes theoretical analyses or emphasizes numerical advantages, for instance, in terms of performance metrics such as makespan, total tardiness, and flow time, but often neglects the practical feasibility of these algorithms in industrial applications. However, research on the trade-offs between solution quality, execution time, and memory consumption in state-of-the-art solvers remains limited, despite their crucial role in real-world deployment. Given the NP-hardness of most scheduling problems, evaluating execution time and memory consumption is, however, essential for real-world adoption, where computational resources are often limited, and efficient decision-making is required.

In this work, we specifically address these challenges. In particular, we focus on one common case of scheduling problems, namely the Job Shop Scheduling Problem (JSSP) [36], for which we introduce HCPGA, a hybrid solver combining a state-of-the-art Constraint Programming (CP) [41] solver, CP-SAT, with a custom Genetic Algorithm (GA), one of the most widely used metaheuristics for scheduling problems. Although CP-SAT is highly effective on

benchmark instances, its computational cost in terms of time and memory grows significantly with the size of the problem, making it impractical for large-dimensional instances, which are common in industrial applications [16]. Our research aims to evaluate whether the integration of CP-SAT with GA can enhance efficiency by reducing execution time and memory consumption while maintaining solution quality. The proposed hybrid approach takes advantage of the complementary strengths of both methods: CP-SAT is used to generate an initial population of high-quality solutions, which are then further refined by the GA.

To evaluate the performance of our proposed HCPGA approach, we tested it on 74 well-known JSSP instances of varying sizes, comparing it against standalone CP-SAT and GA to assess the trade-offs between makespan, computational time, and memory consumption.

The remainder of the paper is structured as follows. The next section introduces the JSSP formulation. Then, Section 3 briefly reviews the main related works. Section 4 describes the proposed approach. Section 5 details the experimental setup, while Section 6 presents and discusses the experimental results. Finally, Section 7 concludes this work by highlighting the main findings, underlining limitations, and providing possible future enhancements to this research.

2 Preliminaries

JSSP consists of scheduling a set $J = \{j_1, \ldots, j_{|J|}\}$ of activities, referred to as *jobs*, on a set M of *machines*, which serve as the resources required to carry out these activities. Each job $j \in J$ is made up of a predefined sequence of tasks $T_j := \langle t_0, \ldots, t_j \rangle$, where each task t_i has a predecessor t_{i-1} and a successor t_{i+1}. These tasks must be executed on specific machines, determined by their task type. Each task t is assigned to a machine $m_t \in M$ and has a fixed processing duration $d_t \in \mathbb{N}$. Tasks are subject to a *precedence constraint*, which requires that a task only begins once all its predecessors have been completed. Furthermore, machines adhere to a *non-overlapping constraint*, meaning that they can process at most one task at a time. Additionally, there is no preemption: once a task starts on a given machine, it must run to completion without interruption. For each task t, the start processing time is denoted as s_t, and the finish processing time is given by $f_t = s_t + d_t$. A feasible *schedule* for the JSSP is a valid assignment of start times for each task in every job. The most common optimization objective for the JSSP is the minimization of the *makespan* $C_{\max}$, also known as the total completion time, which represents the completion time of the last task completed in the schedule. Mathematically, this combinatorial optimization problem is expressed as follows:

Minimize: $C_{\max}$
Subject to:

(1) **Precedence constraint -** The execution order of tasks within each job must be maintained:

$$s_{t_{i+1}} \geq s_{t_i} + d_{t_i}, \quad \forall t_i \in T_j, \forall j \in J.$$

(2) **Non-overlapping constraint -** Any two tasks assigned to the same machine must not overlap in time:

$$s_{t_i} < s_{t_k} \implies s_{t_k} \geq s_{t_i} + d_{t_i}, \quad \forall t_i, t_k : m_{t_i} = m_{t_k}.$$

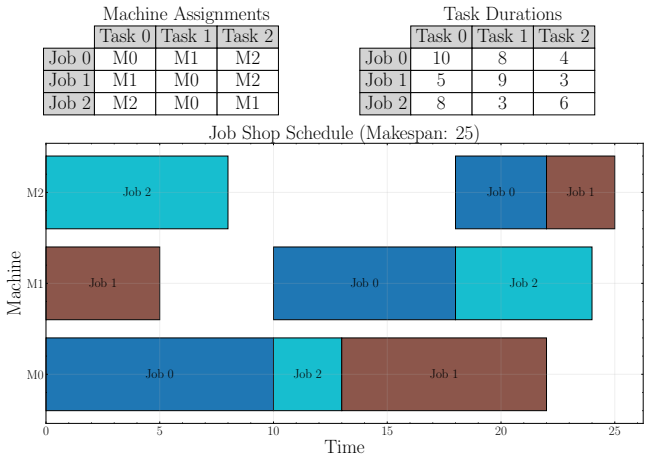


Figure 1: Example of JSSP instance and the corresponding optimal schedule with Gantt chart.

Figure 1 illustrates an instance of the JSSP along with an optimal schedule, visualized using a Gantt chart [13].

Even in its simplest form, where jobs are treated as atomic and not subdivided into tasks, the JSSP is NP-hard [4, 22, 33].

3 Related works

Due to its NP-hard nature, the JSSP has been the subject of extensive research, leading to the development of various solution approaches. These can be broadly categorized into (1) exact methods, (2) metaheuristics, and (3) hybrid approaches.

Exact methods. The first Mixed-Integer Linear Programming (MILP) formulations for the JSSP date back to the late 50s/early 60s [9, 36, 57]. Notably, recent studies have demonstrated that these formulations are still effective when combined with state-of-the-art solvers [30]. Lately, given the inherent complexity of the problem, branch-and-bound algorithms have also been explored to provide exact solutions, leveraging lower bounds to prove optimality [10–12]. Another exact approach that has been investigated is CP [27, 46], with research focusing on efficient variable and value ordering heuristics to minimize the search space [47]. As a combinatorial optimization problem, JSSP has also been addressed using Answer Set Programming (ASP), specifically in the context of industrial scheduling applications [5]. However, ASP-based approaches are known to introduce large computational overheads due to the need to translate the problem into propositional logic (grounding) [15]. More recently, formulations based on Satisfiability Modulo Theories (SMT) have also been developed, offering another promising approach for tackling JSSP [45].

Metaheuristics. While exact methods excel on small instances, they struggle with scalability [22]. As a result, heuristic and metaheuristic methods have gained prominence for efficiently generating high-quality solutions by balancing exploration and exploitation. Among the earliest metaheuristics applied to the JSSP are Tabu Search [52], Simulated Annealing [56], and Genetic Algorithms (GAs) [40]. Over the years, GAs have been refined with specialized

operators to preserve solution feasibility [28]. In addition, other bioinspired algorithms, such as Ant Colony Optimization (ACO) [55] and Particle Swarm Optimization (PSO) [35], have been explored to solve the JSSP. As described by [53], recent advancements emphasize hybridization strategies, integrating metaheuristics with each other, as well as with Machine Learning and data mining techniques, or with exact methods (as we do in this work) to improve performance and scalability.

Hybrid methods. Early hybrid approaches focused on combining different metaheuristics, often yielding better results than standalone methods. Various combinations have been explored to solve the JSSP. For example, GAs have been hybridized with Simulated Annealing [59], Local Search [25], and Tabu Search [8]. Other combinations include Evolution Strategies with Simulated Annealing [29] and Simulated Annealing with Tabu Search [58].

With the rapid advancement in Deep Learning and Reinforcement Learning, alternative hybrid approaches have also emerged, combining, e.g., CP with Reinforcement Learning [54], Supervised Learning [51], and Deep Learning [17]. An interesting direction has been taken by [48], where a SAT solver has been combined with a Machine Learning model providing a warm start.

Regarding the hybridization of exact methods, specifically CP, with metaheuristics, which is the main focus of this paper, several studies in the scheduling domain have experimented with different approaches. However, to the best of our knowledge, none have specifically addressed the JSSP. We therefore examine existing work that integrates metaheuristics with CP for scheduling problems in general. For dynamic scheduling, CP has been combined with GA, where the GA partitions the search space into smaller sub-problems that CP then solves more efficiently [19]. In [37], CP is used jointly with GA and Variable Neighborhood Search (VNS) to solve Distributed Flexible JSSP (DFJSP), with GA-VNS refining solutions before CP further optimizes them. CP has also been paired with Large Neighborhood Search (LNS), where difficult constraints are initially relaxed, penalized, and then progressively reduced by LNS [6]. This concept is further extended in [18], which integrates ASP into a self-adaptive LNS framework. Beyond metaheuristics, CP has also been combined with other heuristic methods. For example, [26] adopts constructive heuristics to enhance CP's efficiency in solving a parallel machine scheduling problem. Similarly, [2] explores a general CP-metaheuristic framework for multi-project scheduling, demonstrating how CP can be embedded within a broader heuristic search process. In [24], CP is incorporated into the Very Large Neighborhood Search (VLNS) to solve the Resource-Constrained Project Scheduling Problem (RCPSP), where CP is applied to generate feasible solutions to initialize the VLNS optimization. CP has also been applied to Casual Employee Scheduling (CES), generating initial feasible solutions later optimized using ACO and Variable Neighborhood Descent (VND) [20].

4 Methodology

The proposed hybrid approach, HCPGA, combines the strengths of CP and GAs to solve complex JSSP instances. SAT-based CP solvers, such as the OR-Tools' CP-SAT [43, 44], leverage constraint propagation and conflict-driven clause learning to efficiently prune the search space. This often allows for finding feasible solutions relatively quickly, although the scalability of these solvers in terms of computational cost (memory and time to reach optimality) when the size of the problem grows remains an issue [14, 16, 42].

To address this limitation, we integrate a custom GA to refine feasible solutions found by CP-SAT, hence enabling a more efficient exploration of the solution space while preserving the advantages of exact constraint-based methods. Specifically, our proposed HCPGA approach operates in two main steps:

(1) **CP-SAT step**: the CP solver runs for a predefined portion of the total time budget, generating one or more feasible schedule solutions for the problem considered.
(2) **GA step**: these feasible solutions serve as the initial population for the GA, which further optimizes the schedule within the remaining time budget.

If the CP-SAT step finds an optimal solution, the second step is skipped, as no further refinement is needed, and the obtained schedule is returned. Conversely, if the first step fails to produce a feasible solution, the process ends, since GA requires at least one feasible solution to initialize its population.

The pseudocode of the proposed HCPGA approach is provided in Algorithm 1. In the following, we describe the two steps as well as their integration in detail.

Algorithm 1 Proposed HCPGA algorithm

```
procedure runHCPGA(JSSProblem, timeBudget)
CPSat ← runCPSAT(JSSProblem, 30% × timeBudget)
if not foundFeasible(CPSat.solutions) then
    return None                          ▷ No feasible solution found
else if foundOptimum(CPSat.solutions) then
    return getOptimum(CPSat.solutions)   ▷ CP-SAT solution
else
    chromosomes ← convert(CPSat.solutions)
    pop ← createPopulation(chromosomes)
    GA ← runGA(JSSProblem, pop, 70% × timeBudget)
end if
return getOptimum(GA.solutions)          ▷ HCPGA solution
end procedure
```

4.1 CP-SAT step

The CP solver is implemented using the CP-SAT solver from Google OR-Tools (v9.11.4210) [43, 44], with Python 3.8.10. To prevent an unbounded scheduling problem, we define a scheduling horizon H, which serves as an upper bound for the integer variables:

$$H = \sum_{j \in J} \sum_{t \in T_j} d_t. \tag{1}$$

Following the definition given in Section 2, H represents the sum of all task durations. For each task t_j in job j, we define three integer variables namely: (1) the start time $s_{t_j} \in [0, H]$; (2) the finish time $f_{t_j} \in [0, H]$; and (3) the interval between start and finish time $I_{t_j} \in [s_{t_j}, f_{t_j}]$.

Given those variables, we enforce the **precedence constraint** by adding these Boolean constraints to the model:

$$f_t < s_{t+1} \quad \forall j \in J, \forall t \in T_j \setminus \{t_{|T_j|}\}. \tag{2}$$

Whereas, the **non-overlapping constraint** is enforced by using disjoint intervals on the same machine:

$$\forall I_{t_j}, I_{t'_{j'}} \text{ assigned to } m, \quad I_{t_j} \cap I_{t'_{j'}} = \emptyset \text{ if } (t_j) \neq (t'_{j'}) \tag{3}$$

meaning that, for each machine m, the set of intervals corresponding to tasks scheduled on that machine must not overlap in time:

$$\text{NoOverlap}(\{I_{t_j} \mid m_{t_j} = m\}), \quad \forall m \in M. \tag{4}$$

Finally, the objective is to minimize the makespan C_{max}, which is defined as:

$$C_{max} = \max_{j \in J} f_{t_{|T_j|}}. \tag{5}$$

4.2 GA step

The GA step is implemented using the *inspyred* Python library [23]. The GA employs a job-based representation for the chromosomes, where each individual corresponds to a potential complete schedule for the given problem instance. Specifically, as illustrated in Figure 2, the schedule representation is encoded as a sequence of job IDs, which defines the processing order of tasks across all the jobs. This linear representation enables the GA to efficiently explore solution sequences while simplifying the implementation of specific evolutionary operators with low computational costs.

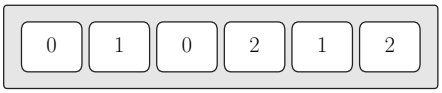


Figure 2: Chromosome representation: sequence of job IDs defining the schedule. In this example, the problem is composed of 3 jobs, each one with 2 operations, and the order encoded by the chromosome is the following: the first task of job 0 is processed first, followed by the first task of job 1, then the second task of job 0, and so on.

To efficiently explore the search space, we employ two specialized operators: (1) A **multi-point crossover operator**, which randomly selects up to three cutting points from two parent solutions and combines the corresponding segments to create new offspring. This operator promotes diversity by mixing the characteristics of different parent solutions, thus enhancing the algorithm's ability to explore a wide range of scheduling sequences. (2) A **multi-swap mutation operator**, which randomly swaps up to five job IDs within the chromosome. This operation not only introduces further diversity, but also offers several benefits depending on the number of swaps applied. When multiple swaps occur, the operator encourages broader exploration and helps prevent stagnation. Conversely, with fewer swaps, the perturbation remains smaller, allowing for finer adjustments to a promising solution. This dynamic fosters a good exploration-exploitation balance.

Since these operators can generate invalid solutions—such as chromosomes with too many or too few tasks assigned to a specific job—a **repair mechanism** is employed. The mechanism prioritizes the preservation of existing job assignments, the elimination of excess tasks, and the addition of missing tasks at the end of the chromosome to restore the correct task count. This process ensures feasibility and maintains a valid job-shop schedule in every chromosome.

To calculate the makespan for each solution, the chromosome is decoded into its corresponding schedule by processing the tasks in the order specified by the chromosome. The earliest start time s_{t_i} of each task t_i is calculated as the maximum of (1) the finish time $f_{t_{i-1}}$ of its predecessor t_{i-1} in the same job, and (2) the time a_{m_t} at which the machine assigned to task t becomes available:

$$s_{t_i} = \max\left(f_{t_{i-1}}, a_{m_t}\right). \tag{6}$$

This decoding procedure allows for a straightforward calculation of the makespan.

4.3 Proposed hybrid approach

In our proposed approach, the feasible solutions generated during the CP-SAT step serve as the initial population for the GA.

Solution conversion. The conversion of CP-SAT solutions into GA chromosomes is implemented as follows: for each CP-SAT feasible solution, the tasks in the schedule are sorted by their start times in ascending order, to maintain the temporal relationships. Then, a chromosome is constructed by extracting the job IDs from this ordered list, producing a sequence that inherently encodes both the job and task order on each machine as defined by the exact method. This transformation preserves the structural information from the CP-SAT step while transitioning between the two representations: interval variables with time assignments in CP-SAT, and permutation-based encodings in the GA.

Initial population. The construction of the initial population for the GA is a critical factor in the effectiveness of the hybrid approach. Relying too heavily on solutions from the first step can lead to stagnation and premature convergence to local optima. In contrast, if these schedules are not properly incorporated, the valuable guidance provided by CP-SAT is lost, eliminating the potential advantages of the hybrid approach. Striking the right balance is essential to ensure effective exploration and exploitation of the solution space. HCPGA employs a solution collector of all the valid schedules discovered in the CP step within the time budget limit. These outputs are used to generate the initial population for the GA. The diversity of the starting population is controlled through three parameters that determine: (1) the number of exact copies of each CP-SAT solution to include in the initial population, (2) the number of randomly generated chromosomes (which are ensured to be valid thanks to the repair mechanism discussed above), and (3) the number of individuals resulting from the multi-swap mutation applied to the CP-SAT solutions.

Time budget allocation. A parameter is used to control the ratio of the time budget allocated to the first and second steps. This parameter should be set in such a way as to provide an effective trade-off on the budget allocated to the two steps. In particular, it should be set so that enough time is allocated to the CP step to find and collect feasible, good-quality solutions without consuming too much time and memory, while also allowing GA to effectively refine the schedule provided by CP and further explore the search space.

5 Experimental setup

We evaluated the performance of our proposed HCPGA approach in comparison to (1) standalone CP-SAT and (2) standalone GA. To ensure statistical significance, each of the three algorithms was repeated 5 times with different seeds in each problem instance.

The standalone CP-SAT solver was executed with a maximum run-time limit of 3 hours due to resource constraints. All other configurations were set to their default settings provided by Google OR-Tools.

The standalone GA was executed with a population size of 100 randomly generated individuals. We ensured the feasibility of these starting solutions by applying the repair mechanism explained in Section 4.2. We set the mutation probability to 0.7, and the crossover probability to 0.6. These choices were based on empirical observations indicating that higher mutation rates helped maintain diversity in the population, which is crucial to avoid premature convergence to a local optimum. Based on observed performance, we employed a generational replacement strategy with 10 elites, ensuring that the best individuals from previous generations survive, preserving high-quality solutions. Concerning the parent selection, we used a tournament selection with a size of 5. The time limit for both the standalone GA and HCPGA was set to match the one allocated to the standalone CP-SAT solver for finding the optimal schedule. Moreover, we introduced a second stopping criterion based on experimental results, to avoid unnecessary computations, which is that if no improvement in the makespan was observed for 300 consecutive generations, the evolutionary process was halted.

As for HCPGA, the configuration of the two steps was set as in the corresponding standalone executions. In order to create the initial population for the GA starting from the CP step output, one copy of each CP solution was cloned into the initial set, 30 individuals were created randomly (and repaired whenever needed), and the remaining individuals were generated by applying multi-swap mutation to the chromosome(s) obtained from the CP solution(s). The choices were made empirically to balance the use of CP-SAT information with the need to avoid excessive reliance on it, which could lead to stagnation in local optima. Regarding the allocation of the time budget for the two steps, in HCPGA we set the time allocated to the first step (CP-SAT) as 30% of the time required by the standalone CP-SAT solver to achieve the optimal solution; whereas the remaining time was allocated to the second step (GA). This choice was made empirically based on various preliminary tests, which are not reported here for brevity.

Evaluation metrics. To comprehensively evaluate the performance of the various algorithms, we analyzed the following metrics:

- **Makespan**: Measured as the finish time of the last task.
- **Execution time** (in seconds): Measured as the time required by each algorithm until its conclusion.
- **Peak memory usage** (in MB): Measured as the peak memory use from the start of the optimization process to its conclusion.

To measure memory consumption accurately, specialized monitoring tools were employed for each of the two steps. For CP-SAT, which is a native C++ library accessed via Python bindings, we used the *psutil* package to track system-level memory allocation, as its usage occurs outside Python's memory management. For the GA step, which is implemented entirely in Python, we invoked *tracemalloc* to monitor memory allocations within the interpreter, providing detailed insights into its behavior. In the hybrid approach, both steps were monitored simultaneously, and we reported the sum of their peak memory usage.

Datasets. We conducted our experiments on the Classic Benchmark dataset described in [16], which comprises 74 benchmark instances widely used in the constraint programming literature. These instances were selected based on their prevalence in MiniZinc Challenges [41, 50], ensuring a diverse and representative set of problem types and allowing for a broader evaluation of solver performance across different problem domains. This benchmark includes samples derived from the following works:

- 3 instances from Fisher and Thompson (ft) [39]
- 40 instances from Lawrence (la) [32]
- 5 instances from Adams/Balas/Zawack (abz) [1]
- 10 instances from Applegate and Cook (orb) [3]
- 14 instances from Storer et al. (swv) [49]
- 1 instance from Yamada/Nakano (yn) [60]
- 1 additional instance (3 × 3) from [38]

The size of these problem instances ranges from 10 to 50 jobs and up to 20 machines, as shown in Figure 3.

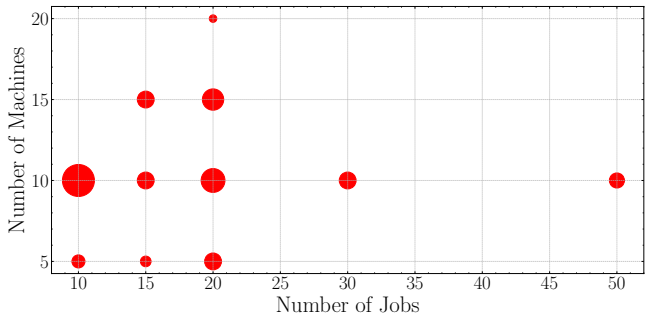


Figure 3: Instance sizes for the Classic Benchmark from [16]. Bubble areas are proportional to the size frequencies.

Computational setup. All experiments were conducted on a Linux workstation equipped with an Intel Core i9-7940X CPU @ 3.10GHz and 64 GB of RAM. To efficiently process multiple instances, we implemented a parallel execution framework using 7 concurrent workers for the Classic Benchmark, distributing the workload evenly across the available CPU cores while ensuring each instance had consistent computational resources for fair comparison. We make our code publicly available on GitHub[1].

6 Results

In the following, we present the results obtained on the 5 available runs of each algorithm (see Table 1 for the detailed numerical results). Note that Figures 4-6 report on the x-axis the various tested problem instances, ordered by label category.

[1]https://github.com/lorenzialessandro/hcpga

Makespan. Figure 4 illustrates the makespan results. These results reveal that the hybrid approach yields results closely aligned with those obtained by the standalone CP-SAT, indicating that the GA step effectively refines the initial solutions found by CP-SAT, without compromising the overall solution quality. Moreover, for the most challenging instances (e.g., *swv01*, *swv14*), HCPGA maintains solution quality comparable to CP-SAT, whereas the standalone GA exhibits significant performance degradation, with makespan values deteriorating by up to 40% worse. This highlights the effectiveness of the hybrid strategy in preserving high-quality solutions for complex problems. Conversely, for smaller instances (*la02–la40*), all approaches produce similar solutions, with makespan differences generally remaining below 5%. This suggests that for less complex problems, the choice of the algorithm may have a marginal impact, only considering the solution quality.

Execution time. Figure 5 presents the computation time required by each algorithm. The y-axis is shown on a logarithmic scale to account for significant variations in computational effort. The time limit of 3h is also highlighted. CP-SAT exhibits a highly variable computational profile, with several instances (e.g., *abz8*, *abz9*, *ft10*, *la27*, and most *swv* instances) requiring the full-time budget and reaching the 3h limit. This confirms the well-documented scalability challenges of exact methods when applied to larger problem instances. In contrast, the GA approach demonstrates more consistent execution time across instances, with a gradual increase for larger problems—an expected characteristic of population-based methods—which allows effective execution also in the presence of time constraints. The hybrid approach achieves a balanced computational profile, inheriting CP-SAT's efficiency on instances where optimality is reached quickly while leveraging the available computational time in more complex cases to refine solutions. This results in significantly reduced computation times compared to standalone CP-SAT, while maintaining good solutions.

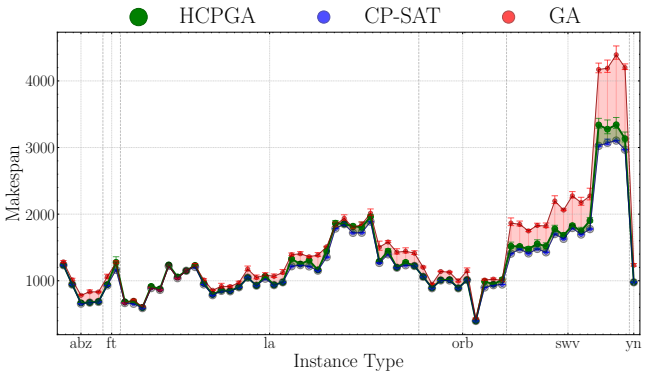


Figure 4: Makespan comparison for the different algorithms (HCPGA, CP-SAT, and GA). Results are shown as mean ± std. dev. across 5 independent runs.

Peak memory usage. Figure 6 compares the memory consumption of each approach. The y-axis, displayed on a logarithmic scale, represents the memory usage in megabytes. CP-SAT consistently exhibits the highest memory consumption, particularly for complex

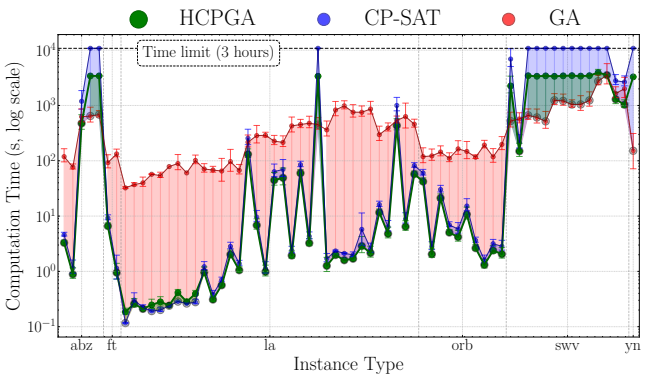


Figure 5: Execution time comparison (log scale) for the different algorithms (HCPGA, CP-SAT, and GA). Results are shown as mean ± std. dev. across 5 independent runs.

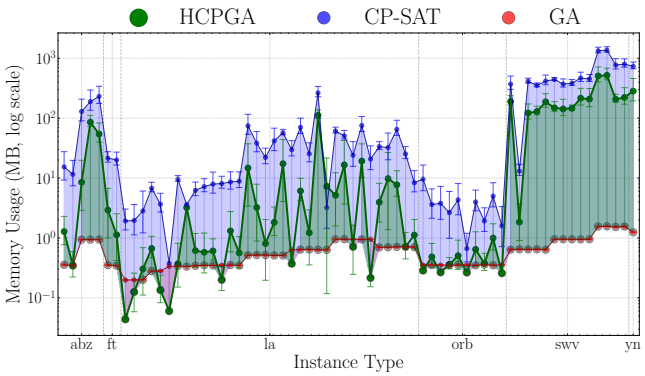


Figure 6: Peak memory usage comparison (log scale) for the different algorithms (HCPGA, CP-SAT, and GA). Results are shown as mean ± std. dev. across 5 independent runs.

instances, with peaks exceeding 1 GB for certain *swv* instances. This aligns with the well-known memory demands of exact methods, which require extensive search state storage in order to maintain large search trees and constraint propagation states. In contrast, the standalone GA approach demonstrates exceptional memory efficiency, maintaining a minimal footprint across all instances and rarely exceeding 10 MB, confirming the space efficiency of evolutionary algorithms. HCPGA approach exhibits intermediate memory usage, typically ranging between 100 and 400 MB for most instances, marking a substantial reduction compared to the standalone CP-SAT. This reduction compared to CP-SAT can be attributed to the delegation of global exploration to the GA component, which limits the depth and breadth of the search space explored by CP-SAT. Moreover, by focusing the exact solver on a narrowed solution space defined by GA candidates, the hybrid method mitigates the exponential state explosion typically observed in exact methods. Additionally, while memory consumption increases with problem size, the hybrid approach demonstrates a more controlled scaling pattern, highlighting its advantage in managing computational resources more efficiently.

Instance	HCPGA			CP-SAT			GA		
	Makespan	Time (s)	Memory (MB)	Makespan	Time (s)	Memory (MB)	Makespan	Time (s)	Memory (MB)
abz5	1236.00 ± 2.00	**3.29 ± 0.37**	1309.80 ± 1036.28	**1234.00 ± 0.00**	4.60 ± 0.49	15729.60 ± 12508.22	1283.20 ± 19.73	118.81 ± 46.42	**364.11 ± 9.54**
abz6	945.80 ± 1.79	**0.89 ± 0.27**	**354.10 ± 252.04**	**943.00 ± 0.00**	1.15 ± 0.45	11813.60 ± 8420.78	1013.00 ± 24.67	76.56 ± 11.10	356.70 ± 1.33
abz7	667.00 ± 2.00	**474.78 ± 207.72**	8734.85 ± 11555.38	**656.00 ± 0.00**	1191.39 ± 669.16	132488.00 ± 80457.85	781.60 ± 13.05	625.86 ± 241.16	**960.73 ± 13.94**
abz8	673.40 ± 4.16	3415.04 ± 18.47	87747.77 ± 26481.43	**671.80 ± 2.59**	10801.36 ± 0.28	193233.60 ± 108538.83	834.80 ± 27.95	**644.94 ± 286.61**	**959.86 ± 17.08**
abz9	686.20 ± 5.17	3427.59 ± 26.46	55678.78 ± 29568.31	**682.20 ± 4.02**	10803.29 ± 2.02	236731.20 ± 112534.77	831.40 ± 11.26	**683.13 ± 124.33**	**961.57 ± 7.40**
ft06	**55.00 ± 0.00**	0.15 ± 0.00	2420.00 ± 0.00	55.00 ± 0.00	**0.15 ± 0.00**	3808.00 ± 0.00	58.00 ± 0.00	31.39 ± 0.00	**212.93 ± 0.00**
ft10	942.00 ± 9.92	**6.60 ± 0.90**	2991.22 ± 3922.15	**930.00 ± 0.00**	9.24 ± 1.31	21996.80 ± 6207.22	1060.00 ± 38.64	92.50 ± 36.16	**360.21 ± 7.14**
ft20	1277.20 ± 83.65	**0.95 ± 0.45**	1153.27 ± 1445.21	**1165.00 ± 0.00**	1.14 ± 0.82	20527.20 ± 7256.36	1271.60 ± 27.50	132.93 ± 29.61	**353.46 ± 5.52**
la01	688.75 ± 10.21	0.18 ± 0.03	**78.04 ± 66.43**	**666.00 ± 0.00**	**0.11 ± 0.01**	1540.00 ± 1266.40	672.00 ± 12.00	31.88 ± 1.58	203.18 ± 0.56
la02	690.00 ± 16.99	**0.26 ± 0.06**	**128.44 ± 136.81**	**655.00 ± 0.00**	0.30 ± 0.11	2002.00 ± 1696.48	707.00 ± 15.68	36.98 ± 3.28	204.43 ± 1.48
la04	596.25 ± 7.32	**0.21 ± 0.01**	311.04 ± 394.08	**590.00 ± 0.00**	0.22 ± 0.03	2902.00 ± 3316.19	611.00 ± 14.63	39.86 ± 10.87	**205.06 ± 3.55**
la06	**926.00 ± 0.00**	**0.19 ± 0.00**	520.00 ± 0.00	926.00 ± 0.00	0.19 ± 0.00	2772.00 ± 0.00	926.00 ± 0.00	42.56 ± 0.00	**285.18 ± 0.00**
la07	910.25 ± 21.69	0.26 ± 0.06	574.72 ± 397.95	**890.00 ± 0.00**	**0.19 ± 0.01**	6233.00 ± 2031.29	897.25 ± 8.46	54.28 ± 5.55	**287.17 ± 1.04**
la08	874.25 ± 17.58	0.30 ± 0.07	1231.59 ± 2188.66	**863.00 ± 0.00**	**0.20 ± 0.02**	4640.00 ± 2390.45	863.00 ± 0.00	51.10 ± 8.80	**288.07 ± 3.45**
la09	**951.00 ± 0.00**	**0.17 ± 0.02**	764.00 ± 718.42	951.00 ± 0.00	0.17 ± 0.01	3436.00 ± 1244.51	951.00 ± 0.00	42.78 ± 1.42	**284.56 ± 0.14**
la10	**958.00 ± 0.00**	0.17 ± 0.00	**260.00 ± 0.00**	958.00 ± 0.00	**0.17 ± 0.00**	4912.00 ± 0.00	958.00 ± 0.00	43.41 ± 0.00	285.29 ± 0.00
la11	1230.00 ± 11.31	**0.24 ± 0.00**	**148.82 ± 123.29**	**1222.00 ± 0.00**	0.26 ± 0.03	4888.00 ± 6369.62	1222.00 ± 0.00	68.71 ± 13.92	342.65 ± 1.85
la12	1058.00 ± 11.31	0.41 ± 0.06	381.40 ± 452.74	**1039.00 ± 0.00**	**0.28 ± 0.01**	9784.00 ± 1465.13	1039.00 ± 0.00	89.24 ± 40.49	**346.30 ± 8.22**
la13	**1150.00 ± 0.00**	0.28 ± 0.00	3292.00 ± 0.00	1150.00 ± 0.00	**0.26 ± 0.00**	3692.00 ± 0.00	1150.00 ± 0.00	60.34 ± 0.00	**340.59 ± 0.00**
la14	**1292.00 ± 0.00**	**0.21 ± 0.00**	548.00 ± 0.00	1292.00 ± 0.00	0.27 ± 0.00	5148.00 ± 0.00	1292.00 ± 0.00	59.19 ± 0.00	**339.40 ± 0.00**
la15	1226.00 ± 12.88	0.36 ± 0.08	488.37 ± 556.22	**1207.00 ± 0.00**	**0.27 ± 0.03**	5620.00 ± 2482.86	1228.75 ± 22.43	99.20 ± 21.26	**350.42 ± 7.99**
la16	961.40 ± 20.91	**0.96 ± 0.24**	590.53 ± 647.12	**945.00 ± 0.00**	1.22 ± 0.29	7364.80 ± 2630.46	1011.60 ± 18.72	71.07 ± 17.62	**358.23 ± 5.34**
la17	794.00 ± 2.65	**0.31 ± 0.05**	622.05 ± 370.40	**784.00 ± 0.00**	0.38 ± 0.07	8114.67 ± 4768.65	857.33 ± 4.73	67.94 ± 11.40	**355.43 ± 3.43**
la18	856.80 ± 3.49	**0.56 ± 0.04**	**204.71 ± 137.68**	**848.00 ± 0.00**	0.70 ± 0.08	8286.40 ± 2687.15	916.20 ± 45.44	65.27 ± 45.12	357.98 ± 7.96
la19	847.20 ± 3.70	**2.03 ± 0.40**	1343.31 ± 1484.53	**842.00 ± 0.00**	2.83 ± 0.53	8812.00 ± 4213.97	913.00 ± 17.46	96.52 ± 64.12	**361.95 ± 14.85**
la20	905.80 ± 3.56	**1.05 ± 0.16**	575.22 ± 507.26	**902.00 ± 0.00**	1.39 ± 0.20	9075.20 ± 3935.16	969.60 ± 24.53	67.13 ± 18.69	**354.84 ± 2.82**
la21	**1046.00 ± 0.00**	**131.13 ± 33.07**	15096.03 ± 23218.29	1046.00 ± 0.00	254.40 ± 117.53	76360.00 ± 42663.55	1175.00 ± 46.20	216.30 ± 76.24	**524.26 ± 8.32**
la22	928.00 ± 2.24	**6.83 ± 2.07**	3301.75 ± 4787.01	**927.00 ± 0.00**	9.57 ± 3.07	39152.00 ± 22005.30	1047.80 ± 39.02	285.16 ± 150.03	**531.21 ± 17.07**
la23	1067.75 ± 20.24	**0.98 ± 0.31**	826.23 ± 1250.16	**1032.00 ± 0.00**	1.12 ± 0.38	22655.00 ± 9668.64	1085.25 ± 36.35	288.90 ± 39.33	**529.59 ± 8.27**
la24	937.60 ± 1.52	**44.59 ± 16.66**	1868.90 ± 1496.06	**935.00 ± 0.00**	63.47 ± 23.82	42874.40 ± 26826.27	1066.80 ± 35.83	226.59 ± 60.99	**523.02 ± 6.98**
la25	**977.00 ± 0.00**	**49.13 ± 24.91**	17794.57 ± 27239.29	977.00 ± 0.00	69.84 ± 35.57	58335.20 ± 8270.90	1124.40 ± 44.99	214.80 ± 67.58	**522.02 ± 4.31**
la26	1327.80 ± 19.33	**1.94 ± 0.40**	**381.22 ± 69.13**	**1218.00 ± 0.00**	2.39 ± 0.45	30399.20 ± 12739.22	1390.00 ± 27.64	427.21 ± 190.86	649.79 ± 10.49
la27	1251.40 ± 7.33	**59.83 ± 9.19**	6243.85 ± 3920.95	**1235.00 ± 0.00**	84.90 ± 13.20	72181.60 ± 30272.25	1405.80 ± 31.00	451.88 ± 108.59	**655.03 ± 7.91**
la28	1306.60 ± 12.26	**3.25 ± 0.67**	1249.23 ± 1772.37	**1216.00 ± 0.00**	4.30 ± 0.87	25995.20 ± 13735.41	1360.20 ± 13.39	476.94 ± 161.97	**656.82 ± 11.44**
la29	1161.60 ± 9.15	3377.93 ± 8.33	114063.02 ± 26333.93	**1152.40 ± 0.55**	10802.89 ± 1.91	270831.20 ± 75563.84	1385.20 ± 38.00	**450.20 ± 155.83**	**648.61 ± 8.37**
la30	1445.25 ± 62.91	**1.26 ± 0.53**	7479.40 ± 14717.70	**1355.00 ± 0.00**	1.70 ± 0.41	3275.00 ± 3584.07	1510.00 ± 9.27	361.61 ± 162.29	**647.31 ± 11.86**
la31	1864.40 ± 40.37	**1.98 ± 0.14**	5282.39 ± 7319.96	**1784.00 ± 0.00**	2.34 ± 0.08	62660.00 ± 9841.65	1834.80 ± 45.52	828.82 ± 353.06	**971.24 ± 16.57**
la32	1854.00 ± 2.24	**1.59 ± 0.07**	16981.70 ± 26422.77	**1850.00 ± 0.00**	2.11 ± 0.08	52032.00 ± 11871.39	1937.00 ± 53.36	983.93 ± 232.56	**979.45 ± 11.61**
la33	1818.80 ± 28.65	**1.70 ± 0.19**	**728.76 ± 951.59**	**1719.00 ± 0.00**	1.99 ± 0.26	24557.60 ± 16914.49	1796.20 ± 27.34	755.63 ± 279.44	969.24 ± 11.54
la34	1803.00 ± 78.29	**2.88 ± 1.39**	19707.97 ± 18456.67	**1721.00 ± 0.00**	5.75 ± 5.63	77277.00 ± 32006.10	1845.00 ± 38.43	753.25 ± 227.32	**969.86 ± 12.30**
la35	1963.00 ± 33.29	**2.16 ± 0.69**	**220.54 ± 127.27**	**1888.00 ± 0.00**	2.56 ± 0.69	21224.80 ± 14111.11	2013.80 ± 64.13	862.58 ± 323.17	974.02 ± 11.32
la36	1293.20 ± 12.76	**11.63 ± 1.76**	4041.68 ± 4323.29	**1268.00 ± 0.00**	16.16 ± 2.45	34222.40 ± 7309.97	1503.40 ± 91.06	296.08 ± 126.49	**711.24 ± 7.96**
la37	1444.60 ± 36.87	**4.79 ± 1.52**	10029.19 ± 17252.48	**1397.00 ± 0.00**	6.36 ± 1.95	33087.20 ± 24801.70	1582.00 ± 21.01	391.55 ± 92.23	**717.56 ± 3.92**
la38	1198.60 ± 5.81	**433.60 ± 142.72**	7905.39 ± 5684.65	**1196.00 ± 0.00**	1000.14 ± 399.43	66698.40 ± 27823.30	1428.20 ± 52.64	536.20 ± 107.63	**725.54 ± 10.77**
la39	1276.60 ± 25.24	**6.44 ± 1.80**	**728.40 ± 538.68**	**1233.00 ± 0.00**	8.83 ± 2.68	25851.20 ± 8470.64	1443.00 ± 51.61	628.47 ± 474.51	730.94 ± 24.30
la40	1223.80 ± 1.10	**57.45 ± 7.52**	1144.07 ± 939.76	**1222.00 ± 0.00**	81.79 ± 10.71	8549.60 ± 5057.44	1413.60 ± 37.33	458.42 ± 106.14	**721.90 ± 8.17**
orb01	1060.00 ± 2.24	**42.18 ± 6.86**	**292.77 ± 49.58**	**1059.00 ± 0.00**	60.14 ± 9.76	9839.20 ± 7082.17	1202.40 ± 17.53	118.78 ± 52.83	360.69 ± 8.22
orb02	892.60 ± 7.54	**2.04 ± 0.51**	495.35 ± 333.74	**888.00 ± 0.00**	2.80 ± 0.64	3708.00 ± 3043.36	948.80 ± 10.57	122.43 ± 40.88	**360.49 ± 4.77**
orb03	1008.80 ± 8.50	**21.21 ± 3.98**	**273.33 ± 16.47**	**1005.00 ± 0.00**	30.19 ± 5.75	3879.20 ± 3568.26	1139.20 ± 6.69	143.99 ± 42.08	361.49 ± 5.39
orb04	1015.50 ± 4.12	**5.04 ± 0.64**	370.19 ± 111.58	**1005.00 ± 0.00**	6.93 ± 0.97	2713.00 ± 3405.00	1128.00 ± 13.11	110.75 ± 28.70	**356.82 ± 3.27**
orb05	888.50 ± 1.91	**4.18 ± 1.33**	517.89 ± 425.22	**887.00 ± 0.00**	5.82 ± 1.81	4428.00 ± 3864.32	997.00 ± 27.77	163.99 ± 66.76	**363.52 ± 10.19**
orb06	1013.40 ± 2.88	**10.72 ± 3.79**	**271.56 ± 28.66**	**1010.00 ± 0.00**	15.19 ± 5.45	680.00 ± 545.42	1148.40 ± 44.72	147.37 ± 76.06	362.66 ± 11.56
orb07	397.75 ± 1.50	**2.64 ± 0.41**	661.06 ± 638.77	**397.00 ± 0.00**	3.63 ± 0.53	4091.00 ± 2363.03	431.00 ± 13.86	116.01 ± 12.33	**357.21 ± 2.18**
orb08	989.00 ± 5.66	**1.31 ± 0.32**	384.65 ± 183.66	**899.00 ± 0.00**	1.61 ± 0.36	1966.00 ± 1298.25	1011.00 ± 15.56	192.02 ± 42.13	**368.36 ± 8.20**
orb09	948.50 ± 8.02	**2.39 ± 0.53**	1017.15 ± 750.13	**934.00 ± 0.00**	3.16 ± 0.62	5110.00 ± 3449.58	1024.50 ± 17.02	117.95 ± 48.09	**358.12 ± 6.88**
orb10	1015.25 ± 29.52	**2.05 ± 0.63**	**264.03 ± 13.33**	**944.00 ± 0.00**	2.84 ± 0.81	1627.00 ± 1787.82	1002.50 ± 22.93	197.43 ± 76.44	363.13 ± 4.61
swv01	1519.40 ± 60.62	2255.79 ± 1132.38	193819.93 ± 48790.67	**1407.00 ± 0.00**	6914.42 ± 3713.45	378844.00 ± 137978.65	1866.20 ± 77.07	**523.51 ± 231.42**	**653.02 ± 14.83**
swv02	1515.40 ± 21.63	**148.96 ± 58.78**	1881.61 ± 1902.38	**1475.00 ± 0.00**	212.25 ± 84.03	13329.60 ± 4028.76	1846.80 ± 49.36	569.35 ± 168.81	**661.50 ± 12.19**
swv03	1476.00 ± 38.20	3451.53 ± 119.34	124914.46 ± 52165.96	**1410.60 ± 7.09**	10802.74 ± 0.99	422694.40 ± 53387.69	1748.20 ± 21.16	**663.17 ± 203.88**	**657.53 ± 9.17**
swv04	1555.40 ± 60.19	3373.38 ± 10.38	131971.43 ± 30827.90	**1478.20 ± 5.40**	10802.47 ± 0.70	360424.80 ± 36353.10	1830.60 ± 33.47	**608.54 ± 253.71**	**661.64 ± 16.36**
swv05	1518.60 ± 51.52	3429.92 ± 116.34	192068.85 ± 70773.02	**1427.20 ± 3.27**	10804.34 ± 4.16	426636.80 ± 110474.29	1820.80 ± 46.49	**517.14 ± 262.43**	**651.47 ± 16.73**
swv06	1780.60 ± 52.44	3364.66 ± 19.53	152235.62 ± 41133.81	**1701.40 ± 6.43**	10802.24 ± 0.59	453061.60 ± 33580.14	2197.40 ± 78.54	**1238.95 ± 398.97**	**974.66 ± 16.42**
swv07	1683.00 ± 30.95	3435.79 ± 28.25	145997.24 ± 66233.11	**1628.80 ± 11.17**	10802.08 ± 0.84	380436.80 ± 83428.28	2061.60 ± 17.36	**1208.88 ± 398.08**	**974.61 ± 13.52**
swv08	1827.60 ± 26.45	3464.71 ± 65.46	149343.50 ± 20809.35	**1792.00 ± 8.00**	10802.52 ± 0.98	391724.80 ± 62068.55	2274.40 ± 63.71	**1040.41 ± 226.39**	**973.27 ± 12.01**
swv09	1746.40 ± 56.01	3396.68 ± 47.65	220519.08 ± 102122.39	**1694.20 ± 4.97**	10802.44 ± 0.70	479194.40 ± 120983.64	2173.20 ± 79.76	**1044.18 ± 459.92**	**969.98 ± 13.51**
swv10	1903.00 ± 47.27	3457.85 ± 53.81	213005.65 ± 107469.09	**1775.60 ± 10.04**	10802.47 ± 1.10	462436.80 ± 86949.54	2272.40 ± 116.74	**1234.71 ± 528.94**	**974.87 ± 20.12**
swv11	3339.20 ± 98.76	3940.30 ± 556.60	520086.77 ± 213963.81	**3024.80 ± 22.40**	10803.06 ± 1.19	1363312.80 ± 203738.72	4170.80 ± 96.64	**2750.69 ± 770.31**	**1588.87 ± 12.63**
swv12	3275.80 ± 138.01	3546.85 ± 135.10	531455.46 ± 168663.94	**3070.00 ± 57.14**	10802.82 ± 1.29	1399128.80 ± 199889.92	4190.00 ± 123.03	**3539.60 ± 2168.39**	**1600.74 ± 22.54**
swv13	3341.80 ± 109.09	**1296.16 ± 431.03**	211409.95 ± 45091.99	**3104.00 ± 0.00**	2801.41 ± 819.17	789079.20 ± 218909.44	4392.00 ± 132.87	1649.53 ± 529.26	**1562.19 ± 9.02**
swv14	3133.40 ± 99.54	**1033.40 ± 231.85**	227333.74 ± 103655.96	**2968.00 ± 0.00**	2654.10 ± 773.06	818642.40 ± 183348.16	4199.00 ± 56.54	2003.49 ± 1220.90	**1586.60 ± 25.00**
yn4	**976.80 ± 4.87**	3287.41 ± 56.44	289560.27 ± 181802.36	977.20 ± 5.63	10800.87 ± 0.84	751854.40 ± 134873.25	1232.20 ± 26.32	**151.61 ± 159.82**	**1275.42 ± 6.95**

Table 1: Detailed results of the different algorithms (HCPGA, CP-SAT, and GA) across multiple metrics (makespan, computational time, and memory usage). Results are shown as mean ± std. dev. across 5 independent runs. The boldface indicates the best result per metric and instance. Highlighted cells indicate the cases in which HCPGA performs best.

7 Conclusions

Despite the vast literature on the JSSP, the gap between theoretical advancements and practical industry applications remains significant. In this paper, we presented HCPGA, a hybrid optimization algorithm that integrates Google's state-of-the-art open-source CP solver CP-SAT with a custom GA to tackle the JSSP. By leveraging CP-SAT for initial solution construction and transitioning to GA for further optimization, HCPGA aims to significantly reduce both computational time and memory consumption. To the best of our knowledge, this work represents the first attempt to combine an evolutionary algorithm with a state-of-the-art exact method for the

JSSP to improve time efficiency and memory usage, both of which are crucial for feasible industrial implementation.

We evaluated our method on 74 benchmark instances of varying size, selected from the most commonly used datasets in the literature. Our experimental analysis demonstrates that HCPGA achieves competitive makespan performance compared to the standalone CP-SAT while outperforming the GA baseline. More importantly, the hybrid approach leads to an average 30.79% reduction in computation time and a 77.29% reduction in memory usage compared to CP-SAT, showcasing its potential for scheduling in industrial contexts characterized by computational constraints. This efficiency gain is largely due to the hybrid design, which limits the scope of exact search through guided candidate solutions from the genetic algorithm, thereby reducing the need for extensive search state storage.

Future research should extend this work by evaluating HCPGA on larger-scale benchmark datasets, such as the `Large-TA` and `known-optima` benchmarks introduced in [16], aligning with problem dimensions that better reflect large-scale real-world industrial scenarios. Additionally, alternative methods could be explored for creating the initial population for the GA. Moreover, other kinds of hybrid algorithm configurations could be investigated.

References

[1] Joseph Adams, Egon Balas, and Daniel Zawack. 1988. The shifting bottleneck procedure for job shop scheduling. *Management science* 34, 3 (1988), 391–401.

[2] Arben Ahmeti and Nysret Musliu. 2025. Hybridizing constraint programming and meta-heuristics for multi-mode resource-constrained multiple projects scheduling Problem. *Journal of Heuristics* 31, 1 (2025), 1–37.

[3] David Applegate and William Cook. 1991. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing* 3, 2 (1991), 149–156.

[4] Kenneth R Baker and Dan Trietsch. 2018. *Principles of sequencing and scheduling*. John Wiley & Sons, Oxford, UK.

[5] Marcello Balduccini. 2011. Industrial-size scheduling with ASP+CP. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, Berlin, Heidelberg, 284–296.

[6] Gustav Björdal, Pierre Flener, Justin Pearson, Peter J Stuckey, and Guido Tack. 2020. Solving satisfaction problems using large-neighbourhood search. In *International Conference on Principles and Practice of Constraint Programming*. Springer, Cham, 55–71.

[7] Jacek Blazewicz, Klaus Ecker, Erwin Pesch, Günter Schmidt, and J Weglarz. 2019. *Handbook on scheduling*. Springer, Cham, Switzerland.

[8] Mohammed Boukedroun, David Duvivier, Abdessamad Ait-el Cadi, Vincent Poirriez, and Moncef Abbas. 2023. A hybrid genetic algorithm for stochastic job-shop scheduling problems. *RAIRO-Operations Research* 57, 4 (2023), 1617–1645.

[9] Edward H Bowman. 1959. The schedule-sequencing problem. *Operations Research* 7, 5 (1959), 621–624.

[10] Peter Brucker and Bernd Jurisch. 1993. A new lower bound for the job-shop scheduling problem. *European Journal of Operational Research* 64, 2 (1993), 156–167.

[11] Jacques Carlier. 1982. The one-machine sequencing problem. *European Journal of Operational Research* 11, 1 (1982), 42–47.

[12] Jacques Carlier and Éric Pinson. 1989. An algorithm for solving the job-shop problem. *Management Science* 35, 2 (1989), 164–176.

[13] Wallace Clark. 1922. *The Gantt chart: A working tool of management*. Ronald Press Company, London, UK.

[14] Giacomo Da Col and Erich Teppan. 2019. Google vs IBM: A Constraint Solving Challenge on the Job-Shop Scheduling Problem. *Electronic Proceedings in Theoretical Computer Science* 306 (2019), 259—-265.

[15] Giacomo Da Col and Erich C Teppan. 2016. Declarative decomposition and dispatching for large-scale job-shop scheduling. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer, Cham, Switzerland, 134–140.

[16] Giacomo Da Col and Erich C. Teppan. 2022. Industrial-size job shop scheduling with constraint programming. *Operations Research Perspectives* 9 (2022), 100249.

[17] Imanol Echeverria, Maialen Murua, and Roberto Santana. 2025. Leveraging constraint programming in a deep learning approach for dynamically solving the flexible job-shop scheduling problem. *Expert Systems with Applications* 265 (2025), 125895.

[18] Thomas Eiter, Tobias Geibinger, Nelson Higuera Ruiz, Nysret Musliu, Johannes Oetsch, Dave Pfliegler, and Daria Stepanova. 2024. Adaptive large-neighbourhood search for optimisation in answer-set programming. *Artificial Intelligence* 337 (2024), 104230.

[19] Abdallah Elkhyari and Adil Bellabdaoui. 2017. Combining constraint programming and genetic algorithm for dynamic scheduling problems. In *International Colloquium on Logistics and Supply Chain Management*. IEEE, New York, NY, US, 19–24.

[20] Nikolaus Frohner, Stephan Teuschl, and Günther R Raidl. 2019. Casual employee scheduling with constraint programming and metaheuristics. In *International Conference on Computer Aided Systems Theory*. Springer, Cham, Switzerland, 279–287.

[21] Helio Yochihiro Fuchigami and Socorro Rangel. 2018. A survey of case studies in production scheduling: Analysis and perspectives. *Journal of Computational Science* 25 (2018), 425–436.

[22] Michael R Garey, David S Johnson, and Ravi Sethi. 1976. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research* 1, 2 (1976), 117–129.

[23] Aaron Garrett. 2012. inspyred: Bio-inspired Algorithms in Python. https://pythonhosted.org/inspyred/

[24] Tobias Geibinger, Florian Mischek, and Nysret Musliu. 2024. Investigating constraint programming and hybrid methods for real world industrial test laboratory scheduling. *Journal of Scheduling* 27, 6 (2024), 607–622.

[25] José Fernando Gonçalves, Jorge José de Magalhães Mendes, and Mauricio GC Resende. 2005. A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research* 167, 1 (2005), 77–95.

[26] Vilém Heinz, Antonín Novák, Marek Vlk, and Zdeněk Hanzálek. 2022. Constraint programming and constructive heuristics for parallel machine scheduling with sequence-dependent setups and common servers. *Computers & Industrial Engineering* 172 (2022), 108586.

[27] Joxan Jaffar and J-L Lassez. 1987. Constraint logic programming. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, US, 111–119.

[28] Yoginath R Kalshetty, Amol C Adamuthe, and S Phani Kumar. 2020. Genetic algorithms with feasible operators for solving job shop scheduling problem. *Journal of Scientific Research* 64 (2020), 310–321.

[29] Bilal Khurshid and Shahid Maqsood. 2024. A hybrid evolution strategies-simulated annealing algorithm for job shop scheduling problems. *Engineering Applications of Artificial Intelligence* 133 (2024), 108016.

[30] Wen-Yang Ku and J Christopher Beck. 2016. Mixed integer programming models for job shop scheduling: A computational analysis. *Computers & Operations Research* 73 (2016), 165–173.

[31] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. 2014. Industry 4.0. *Business & Information Systems Engineering* 6 (2014), 239–242.

[32] SR Lawrence. 1984. An Experimental Investigation of Heuristic Scheduling Techniques.

[33] Jan Karel Lenstra and AHG Rinnooy Kan. 1979. Computational complexity of discrete optimization problems. *Annals of Discrete Mathematics* 4 (1979), 121–140.

[34] Matheus E Leusin, Enzo M Frazzon, Mauricio Uriona Maldonado, Mirko Kück, and Michael Freitag. 2018. Solving the job-shop scheduling problem in the Industry 4.0 era. *Technologies* 6, 4 (2018), 107.

[35] Zhixiong Liu. 2007. Investigation of particle swarm optimization for job shop scheduling problem. In *International Conference on Natural Computation*, Vol. 3. IEEE, New York, NY, US, 799–803.

[36] Alan S Manne. 1960. On the job-shop scheduling problem. *Operations Research* 8, 2 (1960), 219–223.

[37] Leilei Meng, Weiyao Cheng, Biao Zhang, Wenqiang Zou, and Peng Duan. 2024. A novel hybrid algorithm of genetic algorithm, variable neighborhood search and constraint programming for distributed flexible job shop scheduling problem. *International Journal Of Industrial Engineering Computations* 15 (2024), 813––832.

[38] MiniZinc. 2013. MiniZinc Benchmarks: Jobshop. https://github.com/MiniZinc/minizinc-benchmarks/tree/master/jobshop

[39] John F Muth, Gerald Luther Thompson, and Peter R Winters. 1963. *Industrial scheduling*. Prentice-Hall, Englewood Cliffs, NJ, US.

[40] Ryohei Nakano and Takeshi Yamada. 1991. Conventional genetic algorithm for job shop problems. In *International Conference on Genetic Algorithms*, Vol. 91. Morgan Kaufmann, San Francisco, CA, US, 474–479.

[41] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. 2007. MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*. Springer, Berlin Heidelberg, Germany, 529–543.

[42] Juan M. Novas. 2019. Production scheduling and lot streaming at flexible job-shops environments using constraint programming. *Computers & Industrial Engineering* 136 (2019), 252–264.

[43] Laurent Perron and Frédéric Didier. 2024. *CP-SAT*. Google. https://developers.google.com/optimization/cp/cp_solver/

[44] Laurent Perron and Vincent Furnon. 2024. *OR-Tools*. Google. https://developers.google.com/optimization/
[45] Sabino Francesco Roselli, Kristofer Bengtsson, and Knut Åkesson. 2018. SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation. In *International Conference on Automation Science and Engineering*. IEEE, New York, NY, US, 547–552.
[46] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of constraint programming*. Elsevier, Amsterdam, The Netherlands.
[47] Norman Sadeh and Mark S Fox. 1996. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence* 86, 1 (1996), 1–41.
[48] Mikhail Shirokikh, Ilya Shenbin, Anton Alekseev, and Sergey Nikolenko. 2023. Machine learning for SAT: Restricted heuristics and new graph representations. arXiv preprint arXiv:2307.09141.
[49] Robert H Storer, S David Wu, and Renzo Vaccari. 1992. New search spaces for sequencing problems with application to job shop scheduling. *Management Science* 38, 10 (1992), 1495–1509.
[50] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. 2014. The MiniZinc Challenge 2008–2013. *AI Magazine* 35, 2 (2014), 55–60.
[51] Yuan Sun, Su Nguyen, Dhananjay Thiruvady, Xiaodong Li, Andreas T Ernst, and Uwe Aickelin. 2024. Enhancing constraint programming via supervised learning for job shop scheduling. *Knowledge-Based Systems* 293 (2024), 111698.
[52] Eric D Taillard. 1994. Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing* 6, 2 (1994), 108–117.
[53] El-Ghazali Talbi. 2009. *Metaheuristics: from design to implementation*. John Wiley & Sons, Oxford, UK, Chapter 5, 385–459.
[54] Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. 2023. An end-to-end reinforcement learning approach for job-shop scheduling problems based on constraint programming. In *International Conference on Automated Planning and Scheduling*, Vol. 33. AAAI Press, Palo Alto, CA, US, 614–622.
[55] Cansin Turguner and Ozgur Koray Sahingoz. 2014. Solving job shop scheduling problem with Ant Colony Optimization. In *International Symposium on Computational Intelligence and Informatics*. IEEE, New York, NY, US, 385–389.
[56] Peter JM Van Laarhoven, Emile HL Aarts, and Jan Karel Lenstra. 1992. Job shop scheduling by simulated annealing. *Operations Research* 40, 1 (1992), 113–125.
[57] Harvey M Wagner. 1959. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly* 6, 2 (1959), 131–140.
[58] Bing Wang, Xiaozhi Wang, Fengming Lan, and Quanke Pan. 2018. A hybrid local-search algorithm for robust job-shop scheduling under scenarios. *Applied Soft Computing* 62 (2018), 259–271.
[59] Ling Wang and Da-Zhong Zheng. 2001. An effective hybrid optimization strategy for job-shop scheduling problems. *Computers & Operations Research* 28, 6 (2001), 585–596.
[60] Takeshi Yamada and Ryohei Nakano. 1992. A genetic algorithm applicable to large-scale job-shop problems. In *PPSN*, Vol. 2. Elsevier, Amsterdam, The Netherlands, 281–290.