UNIVERSITY ABOU BEKR BELKAID
TLEMCEN
Faculty of Technology
Department of Computer Science

Design and Development of an Integrated Optimization Platform

for Production Planning and Vehicle Routing

The Developed Software Offers a Unified Graphical Interface (GUI)

Master Thesis

Presented and defended by:
[REZGUI MOHAMED ABD ALLAH]

Supervised by:
Prof. [MEKAMCHA KHALID]

Composition of the Jury / Defense Committee

Role	Name and Affiliation	Grades	Affiliation
President	[Moulai-Khatir Djezouli]	Professeur	Univ. de Tlemcen
Examiner	[Boumediene Fatima-Zohra]	MCB	Univ. de Tlemcen
Examiner	[Bendjelloul Amina]	MCB	Univ. de Tlemcen

Academic Year: 2025–2026

2026/06/21

Dedication

To my parents, for their unconditional love and support.

To my professors, for their guidance and mentorship.

To my friends, for their encouragement throughout this journey.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my thesis supervisor, Professor [MEKAMCHA KHALID], for the invaluable guidance, patience and expertise provided throughout this research. Their insightful comments and constructive criticism have been indispensable for this work.

Also I am thankful to the jury members for accepting to judge this thesis and for their time and consideration.

I am grateful to my family for their constant support and encouragement throughout the difficult period of this academic journey.

To end this acknowledgments part i would like to give my sincerest gratitude to the open source community for thier principle of sharing knowledge.

Abstracts / Résumés

Abstract (English)

Keywords: Flexible Job-Shop Scheduling, Vehicle Routing Problem, Genetic Algorithm, Simulated Annealing, Metaheuristics, Optimization, Python, Tkinter.

This thesis focuses on the design and development of an integrated optimization platform capable of tackling two core combinatorial problems in operations research namely the Flexible Job-Shop Scheduling Problem (FJSP) and the Capacitated Vehicle Routing Problem (CVRP). The implementation is developed as a combined GUI in Python with Tkinter enabling users to define problem instances, setup metaheuristic parameters, run algorithm executions and display results in Gantt charts and log-reports of the routes generated. A sophisticated Enhanced Hybrid Genetic Algorithm that includes dynamic mutation, multi neighborhood Tabu search and a diversity maintenance mechanism is developed to solve the FJSP and obtains a makespan of 41 time slots (20.5 h) for a case-study of woven-labels from a factory having 16 jobs and 5 machines. The approach adapted to solve the VRP involves a Simulated Annealing solver incorporating the aspect of heterogeneous fleet, customers partitioning and multi-trip trips, providing reduction of over 23% in the total cost of transportation when compared to the greedy solution obtained for the problem. In this thesis the theoretical base of the methods is explained, the software structure(UML, Merise), algorithms logic, and relative experimental results highlighting the real life applications of metaheuristics to scheduling problems.

Résumé (Français)

Mots-clés : Ordonnancement Job-Shop Flexible, Problème de Tournées de Véhicules, Algorithme Génétique, Recuit Simulé, Métaheuristiques, Optimisation, Python, Tkinter.

Cette thèse présente la conception et la mise en œuvre d'une plateforme d'optimisation intégrée qui traite deux problèmes fondamentaux d'optimisation combinatoire en recherche opérationnelle : le Problème d'Ordonnancement Job-Shop Flexible (FJSP) et le Problème de Tournées de Véhicules Capacité (CVRP). Le logiciel développé offre une interface graphique unifiée (GUI) construite avec Python et Tkinter, permettant aux utilisateurs de définir des instances de problèmes, de configurer les paramètres des métaheuristiques,

d'exécuter les algorithmes d'optimisation et de visualiser les résultats par des diagrammes de Gantt et des journaux de tournées. Pour le FJSP, un Algorithme Génétique Hybride Amélioré combinant mutation adaptative, recherche tabou multi-voisinage et contrôle de diversité est implémenté, atteignant un *makespan* de 41 créneaux temporels (20,5 heures) sur une étude de cas réelle d'étiquettes tissées avec 16 jobs et 5 postes de travail. Pour le VRP, un solveur par Recuit Simulé avec flotte hétérogène, fractionnement des clients et planification multi-voyage est proposé, réduisant le coût total de routage de plus de 23% par rapport aux solutions gloutonnes initiales. La thèse détaille les fondements mathématiques, l'architecture logicielle (UML, Merise), la conception algorithmique et les résultats expérimentaux comparatifs, démontrant l'applicabilité pratique des approches métaheuristiques dans l'ordonnancement industriel et la logistique.

List of Abbreviations

Abbreviation	Definition
FJSP	Flexible Job-Shop Scheduling Problem
CVRP	Capacitated Vehicle Routing Problem
GA	Genetic Algorithm
SA	Simulated Annealing
TS	Tabu Search
ACO	Ant Colony Optimization
SPT	Shortest Processing Time
MTZ	Miller-Tucker-Zemlin (subtour elimination)
HFVRP	Heterogeneous Fleet VRP
SDVRP	Split Delivery VRP
VRPTW	VRP with Time Windows
MDVRP	Multi-Depot VRP
NSGA-II	Non-dominated Sorting Genetic Algorithm II
MILP	Mixed-Integer Linear Programming
CPLEX	IBM ILOG CPLEX Optimization Studio
UML	Unified Modeling Language
GUI	Graphical User Interface
DEAP	Distributed Evolutionary Algorithms in Python

List of Symbols

Symbol	Definition
$C_{\max}$	Makespan (maximum completion time)
C_j	Completion time of job j
d_j	Due date of job j
J	Set of jobs
m	Number of machines/workstations
M_w	Number of parallel machines at workstation w
n	Number of jobs or customers
O_{ji}	i-th operation of job j
p_{wj}	Processing time of job j on workstation w
Q_k	Capacity of vehicle k
q_i	Demand of customer i
S_{wj}	Starting time of job j on workstation w
T_0	Initial temperature (SA)
U_j	Tardiness indicator for job j
w_j	Weight of job j
α	Cooling rate
Δ	Change/difference
θ	Threshold parameter

Contents

Acknowledgments ii

Abstracts iii

List of Abbreviations v

List of Symbols vi

List of Figures x

List of Tables xi

General Introduction 1

1 Generalities and Definitions 4

Part 1: Scheduling . . . 4

1.1 Introduction to Production Scheduling . . . 4

1.1.1 Classification of Scheduling Problems . . . 5

1.1.2 The Job-Shop Scheduling Problem (JSP) . . . 7

1.1.2.1 The Shifting Bottleneck Procedure . . . 9

1.1.3 The Flexible Job-Shop Scheduling Problem (FJSP) . . . 12

1.1.4 Objective Functions in Scheduling . . . 13

1.1.5 Complexity and Solution Methods . . . 14

1.1.6 Lower Bounds and Relaxations . . . 15

1.1.6.1 Workload-Based Bounds for the FJSP . . . 15

1.1.6.2 Assignment Relaxation . . . 16

1.1.6.3 Lagrangian Relaxation for the CVRP . . . 16

1.1.6.4 Held–Karp MST Bound for the TSP Substructure . . . 16

Part 2: Vehicle Tours . . . 17

1.2 Introduction to Vehicle Routing . . . 17

1.2.1 The Capacitated VRP (CVRP) . . . 17

1.2.2 VRP Variants . . . 20

1.2.3 Complexity and Solution Methods . . . 21

Conclusion of Chapter 1 . . . 22

2 State of the Art and Software Architecture **23**
Part 1: State of the Art on Scheduling and Vehicle Tours . . . 23
2.1 Metaheuristics for the FJSP . . . 23
2.1.1 Genetic Algorithms . . . 23
2.1.2 Tabu Search and Hybrid Approaches . . . 26
2.1.3 Tabu Search and Hybrid Approaches . . . 26
2.1.4 Other Metaheuristics . . . 28
2.2 Metaheuristics for the VRP . . . 29
2.2.1 Simulated Annealing for VRP . . . 29
2.2.2 Genetic Algorithms for VRP . . . 30
2.2.3 Ant Colony and Tabu Search for VRP . . . 31
2.3 Gap Analysis and Contribution . . . 32
Part 2: Development Structure of the Software . . . 33
2.4 Requirements Analysis . . . 33
2.5 UML Modeling . . . 34
2.5.1 Use Case Diagram . . . 34
2.5.2 Class Diagram . . . 35
2.5.3 Sequence and Activity Diagrams . . . 36
2.6 Merise Data Modeling . . . 36
2.6.1 Conceptual Data Model (MCD) . . . 37
2.6.2 Logical Data Model (MLD) . . . 37
2.6.3 Physical Data Model (MPD) . . . 37
2.7 Implementation Choices . . . 38
2.7.1 Technology Stack . . . 38
2.7.2 Module Structure and Design Patterns . . . 40
2.7.3 Testing and Quality Assurance . . . 41
2.7.4 Deployment and Executable Packaging . . . 41
Conclusion of Chapter 2 . . . 43

3 Experimental Results and Comparisons **44**
Part 1: Presentation of the Interface and Scheduling Results . . . 44
3.1 Graphical User Interface . . . 44
3.1.1 FJSP Data Tab . . . 45
3.1.2 FJSP GA Settings Tab . . . 47
3.2 FJSP Case Study: Woven-Label Manufacturing . . . 48
3.2.1 Problem Instance and Industrial Context . . . 48
3.2.2 Algorithmic Behavior and Convergence Analysis . . . 49
3.2.3 Results, Statistical Comparison, and Bottleneck Analysis . . . 50
3.2.4 Gantt Chart Visualization . . . 52

3.2.5 Detailed Exegesis of Solver Outputs 52
3.2.5.1 Anatomy of the FJSP Solver Log 53
3.2.5.2 Reading the Gantt Chart 54
3.2.5.3 Reading the VRP Route Ledger 54
Part 2: Presentation and Comparison of Vehicle Tour Results 55
3.3 VRP Interface and Data Flow 55
3.4 VRP Case Study: Milk Distribution 56
3.4.1 Problem Instance and Geographic Context 56
3.4.2 Initial Solution vs. Optimized Solution 57
3.4.3 Solver Log and Fleet Analysis 58
3.4.4 Heterogeneous Fleet Extension and Cost Structure Analysis 59
Conclusion of Chapter 3 60

General Conclusion **62**

Bibliography **65**

A Solver Log Excerpt (FJSP) **68**

B VRP Route Log **69**

List of Figures

2.1 Two-chromosome encoding for the FJSP. Upper layer: machine assignment (M-chromosome); Lower layer: operation sequence (O-chromosome). Each gene corresponds to one operation, with the upper gene indicating the assigned machine and the lower gene encoding operation priority. 25
2.2 Genetic operators for the Enhanced Hybrid GA: POX crossover and mutation mechanisms. 26
2.3 Use Case Diagram of the integrated optimization platform. 35
2.4 Simplified Class Diagram of the platform architecture following MVC and Strategy patterns. 36
2.5 Sequence diagram for FJSP optimization workflow. 36

3.1 Main application window showing the VRP Solver tab with job types, distance matrix, customers, and truck configuration panels. 45
3.2 FJSP Data tab — Workstation configuration for the woven-label case study. 46
3.3 FJSP GA Settings tab — Parameter configuration for the Enhanced Hybrid GA. 48
3.4 Convergence curve of the Enhanced Hybrid GA on the woven-label FJSP instance. 50
3.5 Optimized Gantt chart for the woven-label FJSP (makespan = 41 slots). . 52
3.6 VRP Solver tab — Input panels for distance matrix, customers, and truck fleet. 56
3.7 VRP Results panel — Fleet analysis, trip summary, and selected trip details. 59

List of Tables

2.1 Technology stack of the optimization platform. 39
2.2 Comparative evaluation of alternative technology stacks considered during the design phase. 40

3.1 Routing matrix showing the flow of 16 jobs through 5 workstations. Each column indicates the operation sequence index; cell entries show the workstation ID visited at that position. The Coating workstation (WS1) appears only in Job 16's routing, reflecting its special-process requirement. 47
3.2 FJSP instance parameters (woven-label case study). 49
3.3 Comparison of FJSP results on the woven-label instance. 50
3.4 Machine utilization analysis for the optimized FJSP schedule. 51
3.5 VRP instance parameters (milk-distribution benchmark). 57
3.6 Comparison of VRP results on the 21-customer milk-distribution instance. 57
3.7 Route balance and load factor analysis. 58
3.8 Heterogeneous fleet configuration example. 59
3.9 Heterogeneous fleet sensitivity to T1 cost/km. 60

General Introduction

In the current industrial landscape characterized by intense competition, operational efficiency has been a crucial strategy for both manufacturing and distribution companies to sustain profit and agility in the Industry 4.0 and intelligent manufacturing environments. Characterized by cyber-physical systems, IIoT, cloud manufacturing, and data driven decisions, the forth industrial revolution has reshaped business models of large, medium and small sized businesses. Within this scenario, the two decision-making layers are critical to operational efficiency, namely: **production scheduling** and **vehicle routing**. Production scheduling is the allocation of jobs to machines and the sequencing of those jobs over time, subject to technological and resources constraints, whereas vehicle routing is about serving customer demands by a fleet of vehicles at minimized costs (travel cost, environment impacts, time etc.). Both problems belong to the NP-hard combinatorial optimization class, which implies that, exact methods (e.g. Branch-and-Bound, Branch-and-Cut, Dynamic Programming, Constraint Programming etc.) become computationally infeasible as problem size increases, based on the Cook-Levin theorem and Karp's fundamental reductions for computational complexity.

The theoretical intractability of these problems has motivated decades of research into approximation algorithms, heuristic procedures, and, more prominently, metaheuristic methods that offer favorable trade-offs between solution quality and computational effort. Metaheuristics—including Genetic Algorithms (GA), Simulated Annealing (SA), Tabu Search (TS), Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), Variable Neighborhood Search (VNS), and their numerous hybrid variants—have emerged as the methods of choice for obtaining high-quality approximate solutions within reasonable computational time. Trajectory-based metaheuristics such as SA and TS operate on a single incumbent solution, navigating the search space through carefully designed neighborhood structures and acceptance criteria that allow temporary deterioration to escape local optima. Population-based methods such as GA and ACO maintain a diverse set of candidate solutions, leveraging evolutionary operators or pheromone-mediated construction to explore multiple regions of the fitness landscape simultaneously. Hybrid metaheuristics, often termed memetic algorithms when combining global evolutionary search with local improvement, have proven particularly effective for structured combinatorial problems where the interplay between exploration and exploitation must be finely balanced.

This thesis is prompted by a real-world industrial requirement for an SME that makes simultaneously scheduling decisions for the shop-floor and distribution decisions for the logistics department but lacks the funding and knowledge for enterprise-level Advanced Planning and Scheduling (APS) solutions. Existing commercially-available software, such

as SAP IBP, Oracle JD Edwards, Siemens Preactor, and Dassault DELMIA, addresses these decisions separately; it entails high upfront annual costs (often 10,000 dollars and beyond), requires significant expertise to learn and use, and lacks customization to domain-specific restrictions. Furthermore, the lack of integration between ISA-95 level 3 MES and level 4 TMS has remained largely unresolved in the open-source realm, creating information silos and impeding end-to-end supply-chain visibility. This thesis sets out to build and verify an **integrated optimization framework** which provides:

— One consolidated GUI that will include both data entry and parameter tuning, along with interactive display of results to remove obstacles to operations managers with no prior knowledge of operations research.

— An Improved Hybrid Genetic Algorithm for Flexible Job-Shop Scheduling incorporating adaptively varying mutation rates, Multi-Neighborhood Tabu Search local optimization, population diversity maintenance using cataclysmic mutation triggers, and Seeding using the SPT heuristic.

— This problem involves the use of a Simulated Annealing solver for the heterogeneous-fleet Capacitated Vehicle Routing Problem with split deliveries and multi-trip planning, It uses adaptive cooling schedules, reheating mechanisms, and capacity-sensitive neighborhood operators.

— Real-time viewing with publication quality Gantt charts, log entries in text format and fleet utilization dashboard where management can understand and perform "what if" analysis.

The entire application is written in Python 3.10+, and uses the standard library Tkinter for a cross-platform desktop GUI, the DEAP framework for evolutionary algorithms primitives, NumPy for fast, vectorised numeric operations, Matplotlib to produce the static charts, and custom routines for the simulated annealing part of the routing module. Python was chosen because of the large scientific computing libraries available for it, fast prototyping capabilities and clear language which can serve as a bridge between researchers and software engineers, and zero licensing cost, that is of relevance for the intended users in the SMEs. We test the application on a real-world production example, with a woven-label (FJSP) manufacturing process, and on a milk distribution with 21 clients benchmark (VRP). We observe substantial improvements to existing schedules and routes at runtimes acceptable for tactical daily planning.

Structure of the Manuscript. The manuscript is divided into three chapters. Each chapter consists of multiple thematically related sections, incrementally constructing the theory, algorithmic details and finally experimental validation. Chapter 1 is based on the theory behind production scheduling and vehicle routing, on a definition of the problems FJSP and CVRP in terms of mixed integer programs, on common and sophisticated objective functions and on their classifications based on complexity including special attention to the proofs of strong NP-hardness and inapproximability. Chapter 2 outlines

the present state of the art in metaheuristic solving techniques. We try to give details about known effective encoding techniques, neighborhood functions, hybridization methods etc. And also mention the applied software engineering methods (UML, Merise, design patterns, testing strategy) when designing the system architecture keeping in mind extensibility, modifiability and robustness. Chapter 3 covers experimental results in detail: it shows the GUI, it displays some computed results with their statistical significance, it shows and discuss convergence and sensitivity of the parameters and it compares the developed algorithms to exact solutions, greedy solutions and classic heuristics (e.g., Clarke-Wright saving). General Conclusion gives a summary on the theoretical and software engineering aspects of this thesis, lists limitations and some possible ways of extending the work (multi-objective optimization, learning, quantum annealing, cloud).

Chapter 1

Generalities and Definitions

"Planning is bringing the future into the present so that you can do something about it now.

— Alan Lakein

In this chapter, a theoretical foundation is built for the two optimization problems presented in this thesis. Part 1 of the chapter presents production scheduling theory with an emphasis on the Flexible Job-Shop Scheduling Problem (FJSP), its formulation as an optimization problem, its complexity, and its position in relation to existing scheduling models. Part 2 formalizes the Vehicle Routing Problem (VRP) and the capacitated VRP (CVRP), introduces various mathematical formulations, valid inequalities and the existing taxonomy of practically relevant VRP extensions. For each problem, we define appropriate notation, formulate objectives and constraints as required by optimization models, and provide the complexity.

Part 1: Scheduling

1.1 Introduction to Production Scheduling

Production scheduling can be defined as a process of resource allocation (machines, workers, tools, energy etc) over time for the tasks (jobs, operations, etc.) such that one or more performance measures are minimized (e.g. Lead time, work-in-process, WIP) or maximized (e.g. Utilization of machines, throughput of the facility, energy efficiency) [Pinedo, 2016]. Production scheduling represents an essential activity in both manufacturing management and operations research which significantly affects such performances as lead times, WIP, machine utilizations, throughput capacity, energy consumption profiles and final customer satisfaction. As mentioned above, production scheduling functions as part of the operational level in the manufacturing planning hierarchy, where it translates medium-term production plans (Master Production Schedule) into individual, detailed schedules of operations on resources, for a horizon that can span a shift to a week.

The theoretical approach to the study of scheduling originated from a groundbreak-

ing work by [Johnson, 1954] on the 2-machine flow-shop and was generalized by the classification framework presented by Graham et al. [Graham et al., 1979]. Since then, scheduling theory has undergone development, covering both deterministic and stochastic production environments, machine setups, shop configurations (flow-shop, job-shop, open-shop, and mixed-shop). This field has evolved to consider more and more intricate constraints introduced by new manufacturing technologies such as additive manufacturing, semiconductor wafer fabrication and FMS. The integration between scheduling theory and other disciplines, such as constraint programming, mixed-integer linear programming (MILP) and metaheuristic optimization techniques, has led to the development of a wide variety of tools for decision-making at the operational level that are constantly being refined to support the latest Industry 4.0 driven requirements for real-time, adaptive and energy-conscious manufacturing.

1.1.1 Classification of Scheduling Problems

Traditional scheduling problems are defined in the $\alpha|\beta|\gamma$ **notation** described by Graham et al. [Graham et al., 1979], where the three field descriptor can be succinctly:

— α This is the name of the 'machine environment' and can have values such as1 (single machine), P (identical parallel machines), Q (uniform parallel machines), R (unrelated parallel machines), Fm (flow shop with m machines), Jm (job shop with m machines), Om (open shop with m machines) orFm (flexible job shop).

— β Represents job characteristics and processing constraints such as $pmtn$ (preemption is allowed), rj (release times), dj (due dates), $prec$ (precedence relations), $batch$ (batching), sij (sequence dependent setup times), $no-wait$ (no waiting between operations), $block$ (blocking) and Mj (machine eligibility).

— γ Describes the objective function (to be minimized). Common examples are: the makespan Cmax, the total completion time $\sum Cj$, the total weighted completion time $\sum wjCj$, the total tardiness $\sum Tj$, the total weighted tardiness $\sum wjTj$, the number of tardy jobs $\sum Uj$, or the maximum lateness $L_{\max}$.

Graham et al. [1979] provided canonical examples illustrating the $\alpha|\beta|\gamma$ notation:

— $1|prec|L_{\max}$: minimize maximum lateness on a single machine subject to general precedence constraints. This problem can be solved in polynomial time using Lawler's algorithm.

— $R|pmtn|\sum C_i$: minimize total completion time on a variable number of unrelated parallel machines, allowing preemption. The complexity of this problem remains unknown.

— $J3|p_{ij}=1|C_{\max}$: minimize maximum completion time in a 3-machine job shop with unit processing times. This problem is NP-hard [Graham et al., 1979].

These examples demonstrate how the notation compactly encodes problem structure

and immediately conveys complexity implications. The FJSP addressed in this thesis would be denoted $FJ|r_j, M_j|C_{\max}$ in an extended version of this framework, indicating flexible job-shop environment with release dates and machine eligibility constraints, with makespan minimization as the objective.

For single-machine problems with arbitrary nondecreasing cost functions, Graham et al. [1979] survey an $O(n^2)$ algorithm to solve $1|prec|f_{\max}$. At each step, let S denote the index set of unscheduled jobs, let $p(S) = \sum_{j \in S} p_j$, and let $S' \subset S$ indicate the jobs all whose successors have been scheduled. One selects job k for the last position among $\{J_j \mid j \in S\}$ by requiring that $f_k(p(S)) \leq f_j(p(S))$ for all $j \in S'$. For $1||L_{\max}$, this procedure specializes to Jackson's rule: schedule the jobs according to nondecreasing due dates [Graham et al., 1979]. Introduction of release dates turns this problem into a unary NP-hard one.

The case $1||\sum w_j C_j$ can be solved in $O(n \log n)$ time by Smith's rule: schedule the jobs according to nonincreasing ratios w_j/p_j [Graham et al., 1979]. If all weights are equal, this amounts to the SPT rule of executing the jobs on the basis of shortest processing time first, a rule that is often used in more complicated situations without much empirical, let alone theoretical support for its superior quality.

For example, the classical single-machine total tardiness problem is denoted $1||\sum T_j$, while the flexible job-shop makespan problem with release dates and machine eligibility is $FJ|r_j, M_j|C_{\max}$. This notation facilitates precise communication of problem assumptions and enables researchers to map complexity results across the scheduling landscape. It is worth noting that even seemingly minor constraint additions—such as sequence-dependent setup times or no-wait constraints—can drastically alter problem complexity, transforming polynomially solvable cases into strongly NP-hard variants.

The computational complexity of scheduling problems has been rigorously characterized within the theory of NP-completeness [Graham et al., 1979]. Graham et al. [1979] established that even seemingly minor constraint additions—such as sequence-dependent setup times or no-wait constraints—can drastically alter problem complexity, transforming polynomially solvable cases into strongly NP-hard variants. The reducibility among scheduling problems can be visualized through a directed graph where each vertex represents a problem variant, and edges indicate polynomial-time reductions [Graham et al., 1979]. This reducibility structure implies that if problem $P' \to P$ and $P \in \mathcal{P}$, then $P' \in \mathcal{P}$; conversely, if $P' \to P$ and P' is NP-hard, then P is NP-hard [Graham et al., 1979].

Most heuristic job-shop scheduling procedures described in the literature are based on "priority dispatching" rules. These are rules for choosing an operation from a specified subset to be scheduled next. They include such criteria as SPT (shortest processing time), MWKR (most work remaining), FCFS (first come, first served), etc. [Adams et al., 1988]. The subset of eligible operations is designed to produce an "active" schedule (i.e., such that no operation can be started earlier without delaying some other operation) which is usually also required to be a "nondelay" schedule (i.e., such that no machine is idle at a

time when it could begin executing some operation).

These are one-pass procedures of the greedy type, in that they construct a solution through a sequence of decisions based on what seems locally best, and the decisions once made are final. Like most procedures of this type in other areas of optimization, these heuristics are fast, and they usually find solutions that are not too bad. In many situations this is all that is needed, and so their use is justified [Adams et al., 1988]. However, with the rapid increase in the speed of computing and the growing need for efficiency in scheduling, it becomes increasingly important to explore ways of obtaining better schedules at some extra computational cost, short of going all the way towards the usually futile attempt of finding a guaranteed optimal schedule.

Job shop scheduling is among the hardest combinatorial optimization problems. Not only is it NP-complete [Garey and Johnson, 1979], but even among members of the latter class it belongs to the worst in practice: we can solve exactly randomly generated traveling salesman problems with 300–400 cities (over 100,000 variables) or set covering problems with hundreds of constraints and thousands of variables, but we are typically unable to schedule optimally ten jobs on ten machines [Adams et al., 1988]. Since job-shop scheduling is a very important everyday practical problem, it is therefore natural to look for approximation methods that produce an acceptable schedule in useful time.

This observation by Adams et al. [1988] underscores the fundamental tension in job-shop scheduling research: the problem is both theoretically intractable and practically indispensable. The same holds for its flexible variant (FJSP), where the additional layer of machine assignment decisions amplifies the combinatorial explosion while reflecting the operational reality of modern flexible manufacturing systems.

1.1.2 The Job-Shop Scheduling Problem (JSP)

In the classical **Job-Shop Scheduling Problem** ($Jm||C_{\max}$), each job j consists of a predetermined sequence of operations $O_{j1}, O_{j2}, \ldots, O_{jn_j}$ that must be processed on specific machines according to a fixed technological routing. The routing (machine sequence) is unique per job and immutable. Each operation O_{ji} requires exactly one machine m_{ji} for an uninterrupted processing duration p_{ji}. The objective is typically to minimize the **makespan** $C_{\max} = \max_j C_j$, where C_j denotes the completion time of the last operation of job j. Alternative objectives include total flow time, tardiness, and machine workload balancing.

The JSP can be formulated as a disjunctive programming problem on a disjunctive graph $G = (V, C \cup D)$, where V represents all operations plus source and sink nodes, C denotes conjunctive (precedence) arcs fixed by job routings, and D denotes disjunctive arcs representing machine capacity constraints (pairs of operations on the same machine cannot overlap). Finding a minimum-makespan schedule corresponds to orienting all

disjunctive arcs such that the resulting directed graph is acyclic and the longest path from source to sink is minimized. The JSP is strongly NP-hard [Garey and Johnson, 1979], and its inapproximability has been extensively studied: unless P=NP, no polynomial-time algorithm can guarantee a constant-factor approximation ratio for the general JSP, although approximation schemes exist for restricted variants such as the acyclic job shop.

The disjunctive graph formulation, introduced by Roy and Sussmann and elaborated upon by Adams et al. [1988], provides the most natural mathematical representation for job-shop scheduling problems. In this model, the nodes of the graph correspond to operations, the directed arcs to precedence relations, and the pairs of disjunctive arcs to pairs of operations to be performed on the same machine [Adams et al., 1988]. The numbers on the arcs represent the processing times.

Formally, let $D = (N, A)$ denote the directed graph obtained from the disjunctive graph G by removing all disjunctive arcs. A **selection** S_k in machine clique E_k contains exactly one member of each disjunctive arc pair of E_k. A selection is **acyclic** if it contains no directed cycle. Each acyclic selection corresponds to a unique sequence of the operations pertaining to machine k, and vice-versa. Thus, **sequencing a machine** k means choosing an acyclic selection in E_k [Adams et al., 1988].

A **complete selection** S consists of the union of selections S_k, one in each E_k, $k \in M$. Picking a complete selection, i.e., replacing the disjunctive arc set E by the ordinary (conjunctive) arc set S, gives rise to the directed graph $D_S = (N, A \cup S)$. A complete selection is acyclic if the digraph D_S is acyclic. Every acyclic complete selection defines a family of schedules, and every schedule belongs to exactly one such family. Further, the makespan of a schedule that is optimal for S is equal to the length of a longest path in D_S [Adams et al., 1988]. Thus, in the language of disjunctive graphs, the job-shop problem is that of finding an acyclic complete selection $S \subset E$ that minimizes the length of a longest path in the directed graph D_S.

Barker and McMahon [1985] proposed a primal search tree algorithm for the general job-shop problem based on active schedule generation. An **active schedule** is defined as one in which no operation can be started earlier without delaying some other operation [Barker and McMahon, 1985]. This concept, originally introduced in the context of priority dispatching rules, ensures that the search space is restricted to schedules that are efficient in terms of resource utilization.

The branch-and-bound approach of Barker and McMahon [1985] improves upon earlier methods by exploiting the structure of the disjunctive graph more effectively. Their computational experiments on benchmark problems from the literature demonstrated that the method could solve the 10-job, 10-machine problem of Muth and Thompson [1963] to optimality, and showed improvement over the McMahon and Florian [1975] algorithm for several test instances [Barker and McMahon, 1985]. For the disappointing 5-machine case, however, little progress toward optimization was observed when the effort expended was

about half that required by the McMahon–Florian approach.

As Barker and McMahon [1985] conclude, "the work described here represents an advance in our continuing struggle with the job-shop scheduling problem. However, we still await the method which will quickly solve all the standard jobsets, and substantially change the reputation of this class of problem." This sentiment remains partially valid today: while modern branch-and-cut and constraint programming solvers have pushed the frontier of exact solution, the general job-shop problem with more than 15–20 jobs and machines remains computationally challenging, reinforcing the necessity of metaheuristic approaches for industrial-scale instances.

The concept of the **bottleneck machine** is central to effective heuristic procedures for the job-shop problem. As Adams et al. [1988] explain, "there is more than one way in which a machine can be viewed as a bottleneck. A first concept of this type is that of criticality." Given a partial selection and the corresponding digraph, a machine is **critical** with respect to the current schedule if its selection has some arc on a longest path. While appealing and theoretically justified, this notion is not sufficiently operational for practical purposes: it simply partitions the set of machines into critical and noncritical ones without offering means of distinguishing between degrees to which a machine constitutes a bottleneck.

To prioritize machines, Adams et al. [1988] propose a more refined measure: the **bottleneck quality** of machine k is defined as the value of an optimal solution to a certain one-machine scheduling problem on machine k. Specifically, let $M_0 \subset M$ be the set of machines that have already been sequenced. For any $k \in M \setminus M_0$, let $P(k, M_0)$ be the problem obtained from the original problem by (i) replacing each disjunctive arc set E_p, $p \in M_0$, by the corresponding selection S_p, and (ii) deleting each disjunctive arc set E_p, $p \in M \setminus M_0$, $p \neq k$. This problem is equivalent to minimizing maximum lateness in a one-machine scheduling problem (for machine k) with due dates. Machine m is then called the **bottleneck** among the machines indexed by $M \setminus M_0$ if $v(m, M_0) = \max\{v(k, M_0) : k \in M \setminus M_0\}$, where $v(k, M_0)$ is the value of an optimal solution to $P(k, M_0)$ [Adams et al., 1988].

1.1.2.1 The Shifting Bottleneck Procedure

The **Shifting Bottleneck Procedure** (SBP), introduced by Adams et al. [1988], represents one of the most influential heuristic approaches to the job-shop scheduling problem. The procedure sequences machines one at a time, consecutively. For each machine not yet sequenced, it solves to optimality a one-machine scheduling problem that is a relaxation of the original problem, and uses the outcome both to rank the machines and to sequence the machine with highest rank. Every time a new machine has been sequenced, the algorithm reoptimizes the sequence of each previously sequenced machine that is susceptible to improvement by again solving a one-machine problem [Adams et al., 1988].

The method of solving the one-machine problems is not new; the main contribution of the approach is the way the relaxation is used to decide upon the order in which the machines should be sequenced. This is based on the classic idea of giving priority to bottleneck machines [Adams et al., 1988].

A brief statement of the Shifting Bottleneck Procedure is as follows [Adams et al., 1988]. Let M_0 be the set of machines already sequenced ($M_0 = \emptyset$ at the start):

1. **Identify a bottleneck machine** m among the machines $k \in M \setminus M_0$ and sequence it optimally. Set $M_0 \leftarrow M_0 \cup \{m\}$ and go to Step 2.
2. **Reoptimize** the sequence of each critical machine $k \in M_0$ in turn, while keeping the other sequences fixed; i.e., set $M_0 := M_0 - \{k\}$ and solve $P(k, M_0)$. Then if $M_0 = M$, stop; otherwise go to Step 1.

The one-machine problems solved during the algorithm are of the form of minimizing maximum lateness with release dates and delivery times (heads and tails). The numbers r_i and q_i are sometimes referred to as the "head" and the "tail" of job i [Adams et al., 1988]. To set up these problems, one must solve two longest path problems in the current digraph to calculate the release dates and delivery times.

Adams et al. [1988] implemented this procedure in FORTRAN and tested it on a VAX 780/11 on problems ranging from small instances to problems involving up to 500 operations. The straight version of the procedure (SBI) took on the order of one to two minutes for the larger problems, although it involved hundreds of micro-runs (one-machine problems). Remarkably, for many benchmark instances, the procedure found and proved optimal solutions because the lower bound provided by the bottleneck problem on the first level was matched by the makespan of the schedule found by the procedure.

The one-machine scheduling problems encountered in the Shifting Bottleneck Procedure are solved by the algorithm of Carlier [1982], which is closely related to the one by McMahon and Florian [1975]. Although this problem is NP-complete in the strong sense [Garey and Johnson, 1979], both algorithms, which are of the branch and bound type, are known to be able to solve in a matter of seconds fairly large problems with data drawn from a realistic range [Adams et al., 1988]. Lenstra et al. [1977] and Rinnooy Kan [1976] report favorable results with the algorithm of McMahon and Florian [1975] on problems with up to 80 jobs; Carlier [1982] reports excellent results with his version of the algorithm on problems with up to 1,000 jobs.

At every node of the branch and bound tree, a heuristic based on the MWKR (most work remaining) priority dispatching rule is applied to the current one-machine problem. The iterative step proceeds as follows: among the unscheduled jobs ready to be scheduled at time t (those with release dates $r_j \leq t$), choose one with the greatest tail q_j (if there are ties, break them by giving preference to the greatest processing time d_j), and schedule it by setting its start time to t [Adams et al., 1988]. This greedy construction is complemented

by a sophisticated branching rule based on the position of a critical job relative to a set of jobs that must precede or follow it in any optimal schedule.

A critical computational component of any job-shop scheduling algorithm is the efficient computation of longest paths in the directed graph D_S. To identify the next bottleneck machine, for each $k \in M \setminus M_0$ one must solve the problem of minimizing the makespan subject to precedence constraints and disjunctive constraints on machine k [Adams et al., 1988]:

$$\min t_n \tag{1.1}$$

$$\text{s.t. } t_j - t_i \geq d_i, \qquad (i,j) \in \bigcup (S_p : p \in M_0) \cup A, \tag{1.2}$$

$$t_i \geq 0, \qquad i \in N, \tag{1.3}$$

$$t_j - t_i \geq d_i \vee t_i - t_j \geq d_j, \qquad (i,j) \in E_k. \tag{1.4}$$

Also, to reoptimize the sequence of each critical machine $k \in M_0$, one must solve a problem of the same form for some subset of already-sequenced machines [Adams et al., 1988]. This problem is equivalent to finding a schedule for machine k that minimizes the maximum lateness, given that each operation to be performed on machine k has, besides the processing time d_i, also a release time r_i and a due date f_i. Here $r_i = L(0,i)$ and $f_i = L(0,n) - L(i,n) + d_i$, with $L(i,j)$ the length of a longest path from i to j [Adams et al., 1988].

This latter problem can be viewed as a minimum makespan problem where each job has to be processed in order by three machines, of which the first and the third have infinite capacity, while the second one (corresponding to machine k) processes one job at a time. The processing time of job i is r_i on the first machine, d_i on the second, and $q_i := L(0,n) - f_i$ on the third machine [Adams et al., 1988]. The numbers r_i and q_i are the "head" and "tail" of job i, respectively.

Exact methods for the JSP include branch-and-bound algorithms employing disjunctive branching ([Barker and McMahon, 1985]), the shifting bottleneck heuristic ([Adams et al., 1988]) which relaxes machine subproblems one at a time, and more recent branch-and-cut approaches exploiting valid inequalities derived from the polyhedral structure of the disjunctive formulation. Constraint programming (CP) solvers, particularly those using edge-finding and not-first/not-last propagation rules, have also demonstrated effectiveness on moderate-sized instances up to approximately 20 jobs and 20 machines. However, for industrial-scale instances, the combinatorial explosion of sequencing possibilities necessitates heuristic and metaheuristic approaches.

1.1.3 The Flexible Job-Shop Scheduling Problem (FJSP)

The **Flexible Job-Shop Scheduling Problem** (FJSP) generalizes the JSP by allowing each operation to be processed on any machine from a given set of eligible machines [Brandimarte, 1993], thereby introducing an additional layer of assignment decisions atop the classical sequencing problem. This flexibility mirrors the capabilities of modern Flexible Manufacturing Systems (FMS) and Computer Numerically Controlled (CNC) machine centers where multiple machines possess overlapping technological capabilities. Formally, the FJSP is defined by:

— A set of n jobs $\mathcal{J} = \{1, \ldots, n\}$, each comprising a sequence of operations dictated by a process plan;
— A set of m workstations (machine groups) $\mathcal{W} = \{1, \ldots, m\}$;
— Workstation w contains $M_w \geq 1$ identical parallel machines, introducing the possibility of parallel processing within a workstation;
— Each job j has a routing $\mathcal{R}_j = (w_{j1}, w_{j2}, \ldots, w_{jn_j})$ specifying the ordered sequence of workstations it must visit;
— Processing time p_{wj} of job j on workstation w, which may differ across eligible machines due to varying speeds, tooling, or calibration states.

The FJSP subsumes two nested decision problems: (i) the *routing subproblem* (assigning each operation to a specific machine within its eligible workstation), and (ii) the *sequencing subproblem* (determining the order of operations on each machine). This bilevel structure significantly increases solution space cardinality: whereas a classical JSP with n jobs and m machines has $(n!)^m$ potential sequences in the worst case, the FJSP adds a multiplicative factor of $\prod_{j,w} |\mathcal{M}_{wj}|$ arising from machine assignment choices, where $\mathcal{M}_{wj}$ denotes the set of eligible machines for job j at workstation w.

Decision variables:

— $x_{wjm} \in \{0, 1\}$: binary assignment variable equal to 1 if the operation of job j on workstation w is assigned to machine $m \in \{1, \ldots, M_w\}$;
— $S_{wj} \geq 0$: continuous non-negative starting time of job j on workstation w;
— $C_{\max} \geq 0$: continuous makespan variable to be minimized.

Objective function:

$$\min \; C_{\max} \tag{1.5}$$

Constraints:

$$C_{\max} \geq S_{wj} + p_{wj}, \qquad \forall (j,w) \in \mathcal{H}, \tag{1.6}$$

$$S_{w_s j} \geq S_{w_p j} + p_{w_p j}, \qquad \forall (j, w_p, w_s) \in \mathcal{K}, \tag{1.7}$$

$$S_{wj} \geq S_{w\prime j\prime} + p_{w\prime j\prime} \quad \text{or} \quad S_{w\prime j\prime} \geq S_{wj} + p_{wj}, \qquad \forall \text{ pairs on same machine}, \tag{1.8}$$

$$\sum_{m=1}^{M_w} x_{wjm} = 1, \qquad \forall (j,w) \in \mathcal{H}, \tag{1.9}$$

$$x_{wjm} \in \{0,1\},\ S_{wj} \geq 0. \tag{1.10}$$

Constraint (1.6) defines the makespan by bounding it below by the completion time of every operation. Constraint (1.7) enforces the technological operation sequence per job, ensuring that a successor operation cannot commence before its predecessor finishes. Constraint (1.8) is the classical disjunctive (machine capacity) constraint: no two operations may overlap on the same physical machine, formulated as an exclusive-or condition that is linearized in MILP solvers using big-M formulations or time-indexed variables. Constraint (1.9) ensures exactly one machine is chosen per operation. Additional constraints not shown here include machine availability windows, tooling requirements, and renewable resource limits that may be relevant in specific industrial contexts.

It must be pointed out that the platform doesn't actually resolve this MILP directly. Instead, constraints are implicitly applied by the chromosome decoder: the machine-assignment chromosome assures exactly one machine assigned to an operation and the schedule builder fulfills precedence and disjunctive constraints by maintaining time-tracking arrays. The above MILP just gives the formal definition of the problem. The metaheuristic development should be based on this formulation.

Lower bounds of FJSP can be established by considering machine work-loads (the workload of a machine can not be greater than C_{max}) and the critical path in the conjunctive graph. Solving FJSP with MILP solvers, such as CPLEX, Gurobi, is intractable for the instances with less than 15 jobs and 8 machines, since the linear programming relaxation for disjunctive formulations is relatively weak. As a result, decomposition methods like Benders decomposition, Lagrangian relaxation have been applied but they are still computation-heavy in Flexible Manufacturing Systems.

1.1.4 Objective Functions in Scheduling

In addition to minimizing makespan, which has the clearest meaning with regard to system throughput, several alternative objectives have been addressed in the scheduling literature for purposes of representing the myriad concerns of a manager:

— **Total Weighted Completion Time:** $\min \sum_{j \in \mathcal{J}} w_j C_j$, where w_j denotes the priority or holding cost of job j. Minimizing this criterion reduces work-in-process

inventory and accelerates cash flow;

— **Total Weighted Tardiness:** $\min \sum_{j \in \mathcal{J}} w_j T_j$, where $T_j = \max(0, C_j - d_j)$ measures the tardiness relative to a due date d_j. This criterion is central to just-in-time (JIT) manufacturing and contractually penalized delivery environments;

— **Number of Tardy Jobs:** $\min \sum_{j \in \mathcal{J}} U_j$ with $U_j \in \{0, 1\}$ indicating whether job j completes after its due date. This min-sum criterion is relevant when contractual penalties are levied per late delivery rather than proportionally to lateness magnitude;

— **Total Earliness and Tardiness:** $\min \sum_j \left(\alpha_j E_j + \beta_j T_j\right)$, where $E_j = \max(0, d_j - C_j)$ and α_j, β_j are non-negative weights. This symmetric objective penalizes both early and late completion, reflecting JIT environments where inventory and tardiness costs coexist;

— **Machine Utilization:** $\max \sum_{w,m} \frac{\sum_j p_{wjm} x_{wjm}}{C_{\max}}$, measuring the proportion of available machine time that is productively engaged. High utilization reduces capital idleness but must be balanced against WIP accumulation;

— **Energy Consumption and Carbon Footprint:** $\min \sum_{w,m} \int_0^{C_{\max}} P_{wm}(t)\, dt$, where $P_{wm}(t)$ denotes the power draw of machine m at workstation w at time t. With increasing emphasis on sustainable manufacturing, energy-aware scheduling has emerged as a critical research frontier, often modeled as a bi-objective problem combining makespan and peak power demand.

This thesis seeks to achieve makespan minimization. This is selected as it correlates to throughput of the finishing area (for the woven-label case study) and results in a single unambiguous scalar value representing fitness for use in the genetic algorithm. However, future enhancements are possible via a strategy-pattern interface which allows the objectives to be switched and integrated into multi-objective optimization techniques like NSGA-II or MOEA/D.

1.1.5 Complexity and Solution Methods

The FJSP is strongly NP-hard, a property inherited from the classical JSP [Garey and Johnson, 1979]. Specifically, the decision version of the FJSP (given a threshold C, does there exist a schedule with makespan $\leq C$) is NP-complete in the strong sense, meaning that its hardness persists even when all numeric parameters are encoded in unary. This precludes the existence of pseudo-polynomial time algorithms unless P=NP. The FJSP remains NP-hard even under severe restrictions, such as two workstations with parallel machines or total flexibility where every operation can be processed on any machine.

Exact methods for the FJSP include:

— **Mixed-Integer Linear Programming (MILP):** time-indexed, sequence-based, and rank-based formulations solved via branch-and-bound or branch-and-cut. Com-

mercial solvers (CPLEX, Gurobi, Xpress) can prove optimality for instances up to 10–15 jobs and 5–8 machines;

— **Constraint Programming (CP):** leverages global constraints such as `cumulative` for resource scheduling and `disjunctive` for machine sequencing, combined with strong propagation rules (edge-finding, not-first/not-last, energetic reasoning);

— **Branch-and-Bound with Disjunctive Scheduling:** specialized algorithms that branch on operation ordering decisions and bound using critical path analysis and immediate selection procedures.

For larger instances characteristic of industrial practice, metaheuristics are the prevailing methodology:

— **Genetic Algorithms** (GA): population-based evolutionary search utilizing operation-based or machine-assignment chromosomes, with specialized crossover (POX, JBX, LOX) and mutation operators;

— **Tabu Search** (TS): single-solution trajectory search with short-term memory (tabu lists) and long-term memory (frequency-based intensification/diversification) to escape local optima;

— **Simulated Annealing** (SA): probabilistic hill-climbing with Metropolis acceptance criterion, effective for fine-tuning schedules derived from constructive heuristics;

— **Ant Colony Optimization** (ACO): constructive swarm intelligence where artificial ants build schedules using pheromone trails and heuristic visibility information.

Hybrid approaches that use both global exploration (GA, ACO) and local exploitation (TS, SA, VNS) have shown promising results on typical FJSP benchmarks and it is precisely this ideology that led to the design of the Enhanced Hybrid Genetic Algorithm of this thesis.

1.1.6 Lower Bounds and Relaxations

Although exact methods are impractical for industrial instances very rapidly, tight lower bounds have two functions, firstly to serve as certificate of optimality gap of a heuristic, and secondly to give direction on the design of branch and bound procedures. We have reviewed some of the best bounds forFJSP and CVRP.

1.1.6.1 Workload-Based Bounds for the FJSP

The simplest yet often effective lower bound arises from machine workload. Because workstation w contains M_w identical parallel machines, the total processing time assigned to w cannot exceed $M_w \cdot C_{\max}$. Consequently,

$$LB_{\text{work}} = \max_{w \in \mathcal{W}} \left\lceil \frac{\sum_{j \in \mathcal{J}} p_{wj}}{M_w} \right\rceil \tag{1.11}$$

constitutes a valid lower bound on the makespan. For the woven-label instance of Table 3.2, this yields $LB_{\text{work}} = 38$ slots, already within 7.3% of the CPLEX optimum (40 slots).

A second bound follows from the job critical path. Each job j must visit its routing sequence serially regardless of machine parallelism, hence

$$LB_{\text{path}} = \max_{j \in \mathcal{J}} \sum_{w \in \mathcal{R}_j} p_{wj}. \tag{1.12}$$

The overall bound is $LB_{\text{FJSP}} = \max\{LB_{\text{work}},\, LB_{\text{path}}\}$.

1.1.6.2 Assignment Relaxation

A stronger bound can be obtained by relaxing the disjunctive constraints (1.8) while retaining assignment and precedence. This produces a parallel-machine open-shop relaxation solvable in polynomial time via linear programming. Let C_j denote the completion time of job j in the relaxation. Because the relaxation allows operations to overlap on the same machine, $\max_j C_j \le C^*_{\max}$, providing a lower bound tighter than the pure workload bound when bottleneck machines are heavily loaded.

1.1.6.3 Lagrangian Relaxation for the CVRP

For the CVRP, the vehicle-flow formulation (1.15)–(1.20) admits a classic Lagrangian relaxation. Dualizing the degree constraints (1.16) with multipliers λ_i, we obtain a Lagrangian subproblem that decomposes into a minimum-cost spanning K-tree plus a knapsack-like penalty term:

$$\mathcal{L}(\lambda) = \min_{x} \sum_{k} \sum_{i,j} \Big(d_{ij} c_k - \lambda_i\Big) x_{ijk} + \sum_{i \in \mathcal{C}} \lambda_i, \tag{1.13}$$

subject to connectivity and capacity constraints only. The optimal Lagrangian value $\max_\lambda \mathcal{L}(\lambda)$ dominates the LP relaxation of the vehicle-flow model and can be computed via subgradient optimization in $O(n^3)$ per iteration. In practice, this bound closes 60–80% of the integrality gap on Solomon instances with $n \le 100$.

1.1.6.4 Held–Karp MST Bound for the TSP Substructure

Because each CVRP route is a Hamiltonian cycle on a subset of customers, the minimum spanning tree (MST) bound for the TSP provides a per-route lower bound. For a route R with customer set S_R, let $\text{MST}(S_R)$ denote the weight of the MST on $S_R \cup \{0\}$. Then any feasible route satisfies

$$\text{cost}(R) \ge 2 \cdot \text{MST}(S_R), \tag{1.14}$$

since deleting any edge from a tour yields a spanning tree. Aggregating over the minimum number of routes $\lceil \sum_i q_i / Q_{\max} \rceil$ gives a global distance bound that is particularly tight when customer locations are uniformly distributed in the Euclidean plane.

Part 2: Vehicle Tours

1.2 Introduction to Vehicle Routing

The Vehicle Routing Problem (VRP), introduced by Dantzig and Ramser in 1959 [Dantzig and Ramser, 1959], Stands out as one of the most widely researched combinatorial optimization problems in the logistics and transportation science literature. It is an generalization of the TSP, itself one of the classic combinatorial optimization problems, with roots in the 18th century Eulerian path problem and modern formulations by Dantzig, Fulkerson and Johnson (1954). The VRP is an extension of the TSP for many vehicles of fixed capacity which all start and end at a central depot, and hence the decisions about the vehicle fleet composition and utilization as well as route partitioning significantly augment the complexity in both structural and computational sense. Practical applications that fit the problem formulation are varied; some common examples include delivery operations, waste management, school busing and other pick-up and delivery operations,dial-a-ride systems,inventory routing, and emergency operations.

From an optimization viewpoint, the VRP falls within the class of network design problems with side constraints. The feasible region is the intersection of the vehicle flow polytope with knapsack constraints for capacity limits and subtour elimination constraints which enforce the connection to the depot. An enormous amount of research on the polyhedral nature of the VRP has revealed that many useful facet defining inequalities are known, such as rounded capacity inequalities, framed capacity inequalities, comb inequalities, multi star inequalities, hypotour constraints and many others. These results have been instrumental in the development of modern branch-and-cut procedures capable of solving VRP problems up to a few hundred nodes to proven optimality (even if it may take several hours or even days of computation on modern workstations).

1.2.1 The Capacitated VRP (CVRP)

Given:

— A depot (node 0) and n customers $\mathcal{C} = \{1, \ldots, n\}$;

— A distance (or cost) matrix $D = [d_{ij}]_{i,j=0}^{n}$ satisfying non-negativity and triangle inequality in metric instances;

— A fleet of K vehicles, each with capacity Q (homogeneous) or Q_k (heterogeneous);

— Customer demands $q_i \geq 0$, typically integer-valued in benchmark instances.

Find a set of routes (one per vehicle) such that:

1. Each route starts and ends at the depot, forming a closed tour (circuit);
2. Each customer is visited exactly once (or, in split delivery variants, at least once with total delivered quantity equal to demand);
3. Total demand on a route does not exceed vehicle capacity;
4. Total routing cost (distance or travel time) is minimized.

Vehicle-flow formulation (heterogeneous fleet). Let $x_{ijk} \in \{0,1\}$ indicate whether vehicle k travels directly from node i to node j.

$$\min \quad \sum_{k=1}^{K} \sum_{i=0}^{n} \sum_{j=0}^{n} d_{ij}\, c_k\, x_{ijk} \tag{1.15}$$

$$\text{s.t.} \quad \sum_{k=1}^{K} \sum_{j=0}^{n} x_{ijk} = 1, \qquad \forall i \in \mathcal{C}, \tag{1.16}$$

$$\sum_{j=0}^{n} x_{0jk} \le 1, \qquad \forall k, \tag{1.17}$$

$$\sum_{i=0}^{n} x_{ihk} = \sum_{j=0}^{n} x_{hjk}, \qquad \forall h \in \mathcal{C}, \forall k, \tag{1.18}$$

$$\sum_{i \in \mathcal{C}} q_i \sum_{j=0}^{n} x_{ijk} \le Q_k, \qquad \forall k, \tag{1.19}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ijk} \le |S| - 1, \qquad \forall S \subseteq \mathcal{C}, |S| \ge 2, \tag{1.20}$$

$$x_{ijk} \in \{0,1\}. \tag{1.21}$$

Objective (1.15) minimizes total cost, where c_k denotes the cost per kilometer (or per unit distance) for vehicle k, capturing heterogeneous fleet economics where larger vehicles may incur higher variable costs. Constraint (1.16) ensures each customer is visited exactly once. Constraints (1.17)–(1.18) enforce flow conservation: each vehicle departs the depot at most once, and every customer node has equal in-degree and out-degree. Constraint (1.19) enforces vehicle capacity. Constraint (1.20) is the exponentially large family of subtour elimination constraints (SECs) attributed to Dantzig, Fulkerson, and Johnson; in practice, these are generated dynamically via branch-and-cut separation routines or replaced by polynomially many Miller-Tucker-Zemlin (MTZ) constraints at the expense of weaker linear programming relaxations.

MTZ formulation. Introducing auxiliary continuous variables u_i representing the cumulative load after visiting customer i, the MTZ constraints canbe written as:

$$u_i - u_j + Q_k\, x_{ijk} \le Q_k - q_j, \quad \forall i, j \in \mathcal{C}, i \ne j, \forall k, \tag{1.22}$$

with $q_i \le u_i \le Q_k$ for all $i \in \mathcal{C}$. These restrictions prevent an disconnected cycle being

created that does not pass through the depot since load can never go down, each customer must add at least q_j to load. But the MTZ polytope has famously poor LP relaxation so researchers began investigating lifted MTZ inequalities, multi-commodity flow problems and arc-flow problems, whose solutions would be closer but involve more variables.

Set partitioning formulation. Let $\mathcal{R}$ denote the set of all feasible routes (elementary cycles from the depot satisfying capacity). Let $a_{ir} \in \{0, 1\}$ indicate whether customer i is visited on route r, and let c_r denote the cost of route r. Introducing binary variables λ_r for route selection, the VRP becomes:

$$\min \quad \sum_{r \in \mathcal{R}} c_r \, \lambda_r \tag{1.23}$$

$$\text{s.t.} \quad \sum_{r \in \mathcal{R}} a_{ir} \, \lambda_r = 1, \qquad \forall i \in \mathcal{C}, \tag{1.24}$$

$$\sum_{r \in \mathcal{R}} \lambda_r \leq K, \tag{1.25}$$

$$\lambda_r \in \{0, 1\}. \tag{1.26}$$

Despite the exponentially sized number of decision variables of this formulation, its LP relaxation is very tight, and the core of very successful branch-price-and-cut algorithms, which generates the columns (routes) in an iterative way by solving shortest path with resource constraints subproblems by means of dynamic programming (labeling algorithms). It is important to emphasize that the objective function (1.15) minimizes the **total routing cost**, defined as the sum of arc distances weighted by vehicle-specific cost coefficients. In the homogeneous fleet case where all vehicles share identical cost rates ($c_k = c$ for all k), this objective reduces to total distance minimization. However, with a heterogeneous fleet, the optimizer must balance two competing factors: (i) shorter routes that reduce distance, and (ii) cheaper vehicles that reduce the per-unit-distance cost. This cost structure is particularly relevant for SMEs operating mixed fleets comprising owned vehicles, leased trucks with capacity-dependent rates, and third-party logistics subcontractors with spot pricing.

The solver developed in this thesis does not model fixed vehicle acquisition or depreciation costs; it optimizes **variable transportation costs only**—specifically, the sum of each vehicle's traveled distance multiplied by its cost per kilometer. Formally, for a route $\mathcal{R}_k$ assigned to vehicle k, the cost contribution is:

$$\text{Cost}_k = c_k \cdot \sum_{(i,j) \in \mathcal{R}_k} d_{ij}, \tag{1.27}$$

and the global objective is $\min \sum_{k \in \mathcal{K}} \text{Cost}_k$. This design choice reflects the *tactical planning horizon* (daily or weekly dispatching) where fleet composition is already fixed, rather than the *strategic horizon* where fleet size and mix are decision variables. The SA implementation

does not carry around the decision variables as instances of themselves. Instead, it carries feasible routes as a sequence of nodes, and ensures the flow conservation, capacity and single visit constraints with the help of invariants of data-structures, as well as move filters.

1.2.2 VRP Variants

The classical CVRP has been extended in several ways to describe real logistics problems:

- **VRP with Time Windows (VRPTW):** Each customer i must be served within a specified time interval $[a_i, b_i]$, introducing temporal constraints that interact with routing decisions. The VRPTW is strongly NP-hard and serves as the primary benchmark for metaheuristic competitions ([Solomon, 1987]). Feasibility checking requires tracking arrival times and accounting for waiting time when a vehicle arrives before a_i.
- **Multi-Depot VRP (MDVRP):** Multiple depots $\mathcal{D} = \{1, \ldots, D\}$ serve the customer set, requiring not only routing but also depot allocation decisions. Applications include municipal service districts and retail chain distribution networks.
- **Split Delivery VRP (SDVRP):** A customer's demand may be split across multiple vehicles, a relaxation that can yield cost savings of up to 50% in pathological instances. [Dror and Trudeau, 1989] proved that in any optimal SDVRP solution, no customer is split more than once (at most two visits), and splitting is only beneficial when individual demands exceed half of vehicle capacity.
- **Heterogeneous Fleet VRP (HFVRP):** Vehicles differ in capacity, fixed cost, and variable cost per kilometer. The fleet mix problem adds a strategic layer to operational routing, as selecting an inappropriate vehicle for a route may inflate costs despite sufficient capacity. This variant is particularly relevant for third-party logistics (3PL) providers operating leased fleets.
- **Multi-Trip VRP:** Vehicles may perform multiple routes (return to depot, reload, and depart again) within a planning period, effectively relaxing the one-trip-per-vehicle assumption. This requires scheduling the sequence of trips and ensuring that total working time respects driver shift regulations.
- **Open VRP (OVRP):** Vehicles need not return to the depot after serving the last customer, a common scenario in home delivery where subcontracted drivers finish their routes at their residences.
- **Periodic VRP (PVRP):** Customers require service on specific days within a planning horizon (e.g., weekly), and the optimizer must simultaneously decide visit days and daily routes.
- **Inventory Routing Problem (IRP):** Combines inventory management at customer sites with routing decisions, typically modeled over a rolling horizon with

stochastic demand.

— **Green VRP (GVRP):** Incorporates fuel consumption, carbon emissions, and electric vehicle range constraints into the optimization model, reflecting increasing regulatory pressure for sustainable logistics.

The solver developed in this thesis solves the problem of HFVRP with split deliveries and multi-trip planning. This problem is motivated by the milk-distribution case study in which, due to capacity constraints, a single truck can not serve all 21 customers in one day and that the fleet can be composed by vehicles of different capacities and per-kilometer costs.

The platform supports split deliveries and multi-trip through a decomposition rather than an integrated SDVRP/MTVRP model. Before the optimization problem is set up, those customers with demand greater than the capacity of the smallest truck are pre-split into virtual delivery locations (Algorithm: best-fit bin packing by weight and volume). Then, the routing engine runs several separate CVRP problems one after the other: each CVRP problem corresponds to one trip and the trip scheduling is done incrementally, so that after each trip is scheduled, the total satisfied demand decreases. The solution strategy used is efficient in computation and practically used in industry, but it doesn't allow for the global trip-ordering to be co-optimized.

1.2.3 Complexity and Solution Methods

The CVRP is NP-hard in the strong sense [Lenstra and Rinnooy Kan, 1981], as it generalizes the Bin Packing Problem (partition items into minimum bins of capacity Q) and the TSP. Even finding a feasible solution is NP-complete when demands are large relative to capacity. The asymptotic behavior of the VRP has been characterized by the seminal results of [Haimovich and Rinnooy Kan, 1985], who showed that the optimal solution value per customer converges to a constant as $n \to \infty$ under mild probabilistic assumptions, and by the iterated tour partitioning heuristic which provides a constant-factor asymptotic approximation.

Exact branch-and-cut algorithms, exemplified by the work of [Lysgaard et al., 2004] and [Pecin et al., 2017], solve instances up to ∼200 customers by dynamically separating rounded capacity inequalities, framed capacity inequalities, and strengthened comb inequalities. Branch-price-and-cut algorithms, which combine column generation for route variables with cutting planes, have pushed this frontier further, though they require sophisticated implementation of labeling algorithms with dominance rules and ng-route relaxations.

For larger or richer variants, metaheuristics dominate the practical landscape:

— **Simulated Annealing** (SA): Effective for routing due to smooth neighborhood structures (2-opt, Or-opt, relocate, exchange) and probabilistic acceptance that permits hill-climbing to escape local minima in the cost landscape;

— **Genetic Algorithms** (GA): Particularly route-based crossovers (Order Crossover OX, Partially Mapped Crossover PMX, cycle crossover) and the influential giant-tour representation with optimal splitting (Prins, 2004);
— **Tabu Search** (TS): Granular neighborhood search and attribute-based memory structures have produced some of the best-known solutions for classical CVRP benchmarks ([Toth and Vigo, 2003]; [Gendreau et al., 1994]);
— **Ant Colony Optimization** (ACO): Constructive swarm intelligence excels in dynamic VRP variants where real-time customer requests modify the problem during execution.

More recently, hybrid matheuristics that embed metaheuristic components within exact frameworks—such as local branching, relaxation-induced neighborhood search (RINS), and proximity search—have blurred the boundary between exact and heuristic methods, offering promising avenues for medium-scale rich VRP variants.

Conclusion

We formally introduced the Flexible Job-Shop Scheduling Problem and the Capacitated Vehicle Routing Problem in this chapter. Their mathematical programming formulations and viewpoints (vehicle-flow, MTZ and set-partitioning) are laid down, followed by discussions on the objective functions and complexity classifications used for design of algorithms. We confirmed that the FJSP can be seen as an extension of the classical JSP with the added challenge of machine assignments and that the CVRP is the extension of the TSP with the consideration of several vehicles having varying capacities and different travel costs. Both are strongly NP-hard, hence reliance on metaheuristics seems a rational choice. We look at various existing metaheuristics in Chapter 2 and delve into software engineering paradigm applied to developing the combined platform.

Chapter 2

State of the Art and Software Architecture

This chapter is composed of two complementary parts. Part 1 summarizes the scientific literature about metaheuristic algorithms for scheduling and vehicle routing at a low technical level, listing particular encoding structures, neighbourhood definitions, acceptance criteria, hybridization methods, as well as the missing aspects, to be covered by our unified platform. Part 2 explains the engineering of the software, including requirements analysis with use-cases and user stories, architecture design using UML behavior and structure diagrams, data model based on the Merise method, implementation strategy with a discussion of design patterns, tests approach and technology stack validation.

Part 1: State of the Art on Scheduling and Vehicle Tours

2.1 Metaheuristics for the FJSP

2.1.1 Genetic Algorithms

Genetic Algorithms (GA) are the most widely applied population-based metaheuristic for the FJSP, owing to their ability to navigate vast solution spaces through parallel exploration and recombination. The seminal work of Brandimarte [1993] proposed one of the earliest GA frameworks, utilizing a two-chromosome encoding: one layer for machine assignment and one layer for operation sequencing. This bipartite representation has since been refined into multiple encoding taxonomies:

1. **Machine-Sequence–Operation-Sequence (MSOS):** Two independent chromosomes where the first assigns machines and the second permutes operations. Decoding requires an active schedule generation scheme to resolve precedence and capacity conflicts.
2. **Permutation with Repetition:** A single chromosome of length $\sum_j n_j$ where each job index appears exactly n_j times. The k-th occurrence of job j is decoded as its

k-th operation, and machine assignment is performed via an auxiliary rule (e.g., earliest available eligible machine).

3. **Random-Key Encoding:** Real-valued genes in $[0, 1]$ are sorted to determine operation priorities, offering natural crossover without repair mechanisms.
4. **Priority-Rule-Based Encoding:** Each gene represents a dispatching rule (SPT, LPT, MWKR, FIFO) applied at a decision point, effectively encoding a schedule-building strategy rather than a direct solution.

Zhang et al. [2009] introduced an effective GA hybridized with Tabu Search, achieving near-optimal results on Brandimarte benchmarks by applying TS to elite individuals after every generation. More recently, developed an effective GA with variable neighborhood search for the multi-objective FJSP, employing Pareto dominance and crowding distance metrics imported from NSGA-II.

Key components of a successful FJSP-GA include:

1. **Encoding:** Operation-based (permutation with repetition) or job-pair-based;
2. **Decoding:** Active schedule generation (left-shifted operations) vs. non-delay (no machine idle time when an operation is ready) vs. semi-active decoding;
3. **Crossover:** Precedence Operation Crossover (POX), Job-based Crossover (JBX), Two-Point crossover on machine vectors, Linear Order Crossover (LOX), and Precedence Preservative Crossover (PPX);
4. **Mutation:** Machine reassignment (selecting a different eligible machine), swap/insertion on operation sequence, inversion, and intelligent mutation that targets critical blocks on the longest path;
5. **Local Search:** Tabu Search or Variable Neighborhood Search (VNS) to refine elite individuals, often restricted to critical operations to reduce computational overhead.

Algorithm 1 Enhanced Hybrid Genetic Algorithm for FJSP

Require: Population size N, generations G, crossover rate p_c, base mutation p_m, elite size E, stagnation limit S

Ensure: Best schedule π^*

Initialize population P_0 with SPT heuristic and random individuals
for $g = 1$ **to** G **do**
    Evaluate fitness (makespan) of all individuals in P_{g-1}
    Select parents by tournament selection (binary or k-ary)
    Generate offspring by POX crossover on operation sequence and uniform crossover on machine assignment
    Apply adaptive mutation: with probability p_m, perform swap, insertion, or machine reassignment on operation and machine chromosomes
    Apply Tabu Search to elite individuals using M1, Swap, and Key neighborhoods with a short-term tabu list
    if no improvement in best fitness for 10 generations **then**
        $p_m \leftarrow \min(p_m + 0.1, 0.9)$
        Optionally apply cataclysmic mutation: preserve elite, reinitialize remaining individuals randomly
    else
        Reset stagnation counter; p_m returns to base value if a new best is found
    end if
    Form next generation P_g with elitism (retain top E individuals unconditionally)
end for
return Best individual π^*

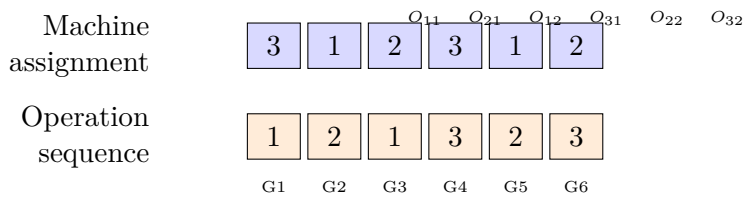
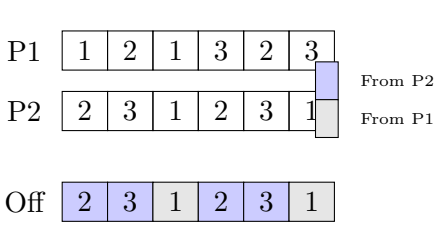
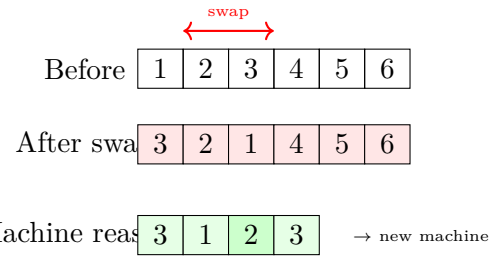


Figure 2.1 – Two-chromosome encoding for the FJSP. Upper layer: machine assignment (M-chromosome); Lower layer: operation sequence (O-chromosome). Each gene corresponds to one operation, with the upper gene indicating the assigned machine and the lower gene encoding operation priority.

The encoding and genetic operators are illustrated in Figures 2.1–2.2. The two-chromosome representation (Figure 2.1) separates machine assignment decisions from operation sequencing, enabling the GA to evolve both subproblems simultaneously. POX crossover (Figure 2.2a) preserves precedence relations by copying selected jobs from one parent while maintaining the relative order of remaining jobs from the other. Mutation



(a) Precedence Operation Crossover (POX). Jobs selected from Parent 2 (blue) are copied to the same positions in the offspring; remaining positions are filled preserving the order from Parent 1.



(b) Mutation operators: (top) swap mutation on operation sequence; (bottom) machine-reassignment mutation changing from machine 1 to 2.

Figure 2.2 – Genetic operators for the Enhanced Hybrid GA: POX crossover and mutation mechanisms.

(Figure 2.2b) introduces local perturbations through either operation swaps or machine reassignments to maintain population diversity.

2.1.2 Tabu Search and Hybrid Approaches

2.1.3 Tabu Search and Hybrid Approaches

Tabu Search (TS) uses short-term memory (tabu list) and long-term memory (frequency-based intensification/diversification) to escape local optima [Glover, 1989]. The tabu status of a move prevents cycling for a dynamically adjusted tenure, while aspiration criteria allow overriding tabu status if the move yields a solution better than any previously encountered. Nowicki and Smutnicki [2005] proposed a fast TS algorithm for the job shop that remains a benchmark reference, utilizing a neighborhood based on swapping operations on a critical path block. For the FJSP, TS is often hybridized with GA: GA provides global exploration through recombination and mutation, while TS refines promising regions via intensified local trajectories [Zhang et al., 2009].

For our implementation, the Tabu Search used as local improvement within the GA operates on the random-key encoding and employs three move types:

— **Machine reassignment (M1):** Select one operation and assign it to a different eligible machine within the same workstation.

— **Key swap (Swap):** Exchange the random keys of two jobs, altering their relative priorities in the decoded schedule.

— **Key perturbation (Key):** Apply Gaussian perturbation $\mathcal{N}(0, 0.2)$ to a single job key, clipped to $[0, 1]$.

The tabu list stores the inverse of each executed move (machine index swap, key swap, or key restoration) with a fixed tenure.

The **local reoptimization procedure** constitutes an essential component of the Shifting Bottleneck heuristic. Let M_0 be the set of machines already sequenced, and let

$k(1), \ldots, k(p)$ be an arbitrary ordering of M_0 (where $p = |M_0|$). A **local reoptimization cycle** means the following procedure: for $i = 1, \ldots, p$, solve the problem $P^*(k(i), M_0 \setminus \{k(i)\})$ and substitute the optimal selection for the old selection. As long as $|M_0| < |M|$, the algorithm goes through at most three local reoptimization cycles for each set M_0. At the last step, when $|M_0| = |M|$, the local reoptimization continues to the point where there is no improvement for a full cycle [Adams et al., 1988].

The ordering of M_0 is at first given by the order in which the machines were sequenced. Every time a full cycle is completed, the elements of M_0 are reordered according to decreasing values of the solutions to the one-machine problems [Adams et al., 1988]. This dynamic reordering ensures that the most critical machines receive priority in subsequent reoptimization passes.

The concept of embedding a powerful heuristic within a partial enumeration framework was pioneered by Adams et al. [1988] in their **selective enumeration** variant of the Shifting Bottleneck Procedure. As the computational results show, the shifting bottleneck procedure almost always obtains considerably better schedules than the best among the priority dispatch rule heuristics, and it frequently finds an optimal schedule. Nevertheless, for situations when the quality of the schedule is sufficiently important to justify a more intensive computational effort, Adams et al. [1988] developed a second version of their approach, which applies the shifting bottleneck procedure to the nodes of a partial enumeration tree.

The nodes and arcs of the search tree are structured as follows. A node corresponds to a set M_0 of machines that have been sequenced in a particular way. An arc corresponds to a pair of sets $(M_0, M_0 \cup \{k\})$ and a selection S_k for some $k \in M \setminus M_0$, generated by solving $P^*(k, M_0)$. At a typical node, the shifting bottleneck procedure is applied with the difference that whenever machines are ranked according to decreasing $v(k, M_0)$ to identify the current bottleneck, a certain number of the one-machine problems generated are stored for further exploration later in the process [Adams et al., 1988].

A second instrument for limiting the size of the search tree is a **penalty function**, defined for every node, that penalizes the choices made at different levels in generating the node in question, in proportion to their deviation from the bottleneck, and with a weighting that is heavier for the higher than for the lower levels of the tree. Whenever the value of the penalty function for a node exceeds a predetermined limit, the node is discarded [Adams et al., 1988].

Adams et al. [1988] developed an $O(n)$ algorithm for the longest path calculations that takes advantage of the special structure of the digraphs on which their problems are defined. Typically, the digraphs are quite dense: they contain a complete subgraph for every machine already sequenced. Although the number of arcs in each selection is $\frac{1}{2}v(v-1)$, where v is the number of nodes in the subgraph, only the $v-1$ arcs that form the unique Hamiltonian path in the subgraph are of consequence for the longest path

calculation; the rest can be deleted or simply ignored [Adams et al., 1988]. In the resulting digraph, every node except for the source and sink has at least one and at most two predecessors (successors). The labeling algorithm for the longest path calculations can then be modified to use only the relevant arcs, and its complexity thus becomes $O(n)$.

While the central idea of the shifting bottleneck procedure does not depend on the way one solves the longest path problems encountered, nevertheless this is the most time-consuming part of the procedure. Thus being able to solve the longest path problems in $O(n)$ time is important for the overall efficiency of the procedure [Adams et al., 1988].

Occasionally, the selection associated with the optimal solution to a one-machine subproblem may create a cycle in the resulting digraph. This can be avoided by a slight modification of the algorithm to take into account precedence relations between jobs when solving the one-machine problem. While this change is trivial and does not affect running time, its implementation would also require a systematic recording of precedence relations created by each selection. Since the occurrence of cycles is very rare (Adams et al. [1988] encountered none during any of their runs), their procedure instead handles it by a special step: whenever a cycle is discovered during a longest path calculation, the particular selection and precedence relation that gave rise to it is identified and the one-machine problem that produced the culprit is solved again, this time subject to the precedence constraint [Adams et al., 1988].

2.1.4 Other Metaheuristics

— **Simulated Annealing (SA):** applied SA to job-shop scheduling; less common for FJSP than GA/TS but useful for fine-tuning due to its gentle acceptance landscape that preserves solution structure while escaping shallow local optima. Adaptive cooling schedules ([Huang, 2007]) adjust temperature based on observed acceptance rates rather than following a predetermined geometric sequence.

— **Ant Colony Optimization (ACO):** introduced ACO; Xiang and Lee [2008] applied it to dynamic manufacturing scheduling. ACO constructs solutions probabilistically using pheromone trails τ_{ij} and heuristic visibility η_{ij}, typically updating trails via the best-so-far or iteration-best ant. For the FJSP, pheromones may be deposited on machine-assignment arcs and operation-precedence arcs simultaneously.

— **Particle Swarm Optimization (PSO):** combined PSO with simulated annealing for FJSP, encoding particles as continuous position vectors that are discretized via priority rules. Discrete PSO variants (DPSO) have been developed to handle permutation spaces directly.

— **Variable Neighborhood Search (VNS):** Systematically changes neighborhood structures within a local search framework, combining deterministic change with

randomized shaking to escape deep local optima. VNS has shown strong results on FJSP benchmarks when hybridized with exact methods for the assignment subproblem.

2.2 Metaheuristics for the VRP

2.2.1 Simulated Annealing for VRP

Simulated Annealing (SA) is particularly well-suited to routing problems because the neighborhood structure (2-opt, Or-opt, relocate, exchange, cross-exchange) is well-defined and the objective landscape is relatively smooth compared to scheduling . Osman [1993] developed one of the first SA implementations for VRP, demonstrating competitive results with classical benchmarks by employing a λ-interchange neighborhood that generalizes 2-opt and relocate moves. Modern SA variants for VRP incorporate:

— **Adaptive cooling schedules:** Temperature adjusted based on observed acceptance rate rather than fixed geometric decay, ensuring the search remains in a critical regime where approximately 50% of worsening moves are accepted;

— **Reheating:** Periodic temperature increases to escape deep local optima that emerge after prolonged cooling, effectively restarting the search from a refined incumbent;

— **Heterogeneous fleet handling:** Capacity and cost-aware neighborhood moves that evaluate not only distance reduction but also vehicle substitution economies (e.g., replacing a large truck with a smaller one when demand permits).

The neighborhood vocabulary for VRP SA typically includes:

— **2-opt:** Remove two arcs and reconnect the route in the alternative way;

— **3-opt:** Remove three arcs and reconnect among the eight feasible alternatives;

— **Or-opt:** Relocate a chain of 1–3 consecutive customers to a different position;

— **Relocate:** Move a single customer from its current route to another;

— **Exchange:** Swap two customers between two routes;

— **Cross-exchange:** Exchange two chains of customers between two routes;

— **Ejection chains:** A sequence of forced relocations triggered by removing a customer, cascading through the solution until a valid insertion is found.

Algorithm 2 Simulated Annealing for Heterogeneous VRP

Objective. The SA optimizer minimizes the **total variable routing cost** $\sum_{k \in \mathcal{K}} c_k \cdot d_k$, where d_k denotes the total distance traveled by vehicle k and c_k its cost per kilometer. Fixed vehicle costs (acquisition, insurance, registration) are excluded, as fleet composition is treated as given input. The optimizer focuses on **operational cost minimization** given an existing heterogeneous fleet.

Require: Initial temperature T_0, cooling rate α, reheats R, iterations per temperature L, heterogeneous fleet $\mathcal{K}$

Ensure: Best set of routes $\mathcal{R}^*$

Construct initial solution $\mathcal{R}$ by greedy nearest-neighbor with regret insertion
$\mathcal{R}^* \leftarrow \mathcal{R}$
for $r = 1$ **to** R **do**
    $T \leftarrow T_0$
    for $iter = 1$ **to** L **do**
        Generate neighbor $\mathcal{R}'$ by 2-opt, relocate, exchange, or cross-exchange
        Evaluate cost $C(\mathcal{R}')$ with heterogeneous fleet costs c_k (fixed costs not modeled)
        if $\Delta C = C(\mathcal{R}') - C(\mathcal{R}) < 0$ **or** $\exp(-\Delta C/T) > \text{rand}()$ **then**
            $\mathcal{R} \leftarrow \mathcal{R}'$
            if $C(\mathcal{R}) < C(\mathcal{R}^*)$ **then**
                $\mathcal{R}^* \leftarrow \mathcal{R}$
            end if
        end if
        $T \leftarrow \alpha \cdot T$ (geometric cooling)
    end for
end for
return $\mathcal{R}^*$

2.2.2 Genetic Algorithms for VRP

GA for VRP typically uses order-based crossovers that preserve customer visitation sequences while allowing route restructuring:

- **Order Crossover (OX):** Preserves relative order of cities from one parent within a randomly selected segment;
- **Partially Mapped Crossover (PMX):** Preserves absolute positions by mapping conflicting segments via a repair procedure;
- **Route-Based Crossover:** Exchanges entire routes between parent solutions, preserving capacity feasibility but requiring repair for duplicate/missing customers;
- **Giant Tour + Split:** Prins [2004] introduced a simple and effective GA using a giant tour representation (concatenation of routes without delimiters) and an

optimal splitting procedure (shortest path on an auxiliary graph) to reconstruct feasible routes. This representation remains influential and has been extended to heterogeneous fleets by modifying the splitting graph to account for vehicle-specific capacities and costs.

2.2.3 Ant Colony and Tabu Search for VRP

ACO constructs solutions probabilistically using pheromone trails τ_{ij} deposited on arcs (i, j); it excels in dynamic VRP variants where customer requests arrive during the execution horizon [Xiang and Lee, 2008]. The transition probability from customer i to j for ant k is proportional to $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$, where $\eta_{ij} = 1/d_{ij}$ is the standard visibility heuristic. Pheromone evaporation ($\rho \in (0, 1]$) prevents premature convergence, while elite ant strategies deposit additional pheromone along the best-known tour.

Tabu Search, especially the *Granular TS* of Toth and Vigo [2003], is among the most successful metaheuristics for classical CVRP. The granular neighborhood restricts moves to arcs shorter than a threshold determined by the nearest-neighbor graph sparsity, drastically reducing evaluation time without sacrificing solution quality. Attribute-based memory records forbidden transitions (e.g., removing customer i from route r), while intensification strategies periodically return to elite solutions and perform exhaustive local search.

Adams et al. [1988] tested their procedure on problems taken from the literature or generated for the purposes of the experiment, ranging from small instances for which an optimal solution was known to problems involving up to 500 operations. The problems in their Table 1 include the notorious ten jobs/ten machines problem from Muth and Thompson [1963] (Problem 3) that had defied solution for more than 20 years despite every available algorithm being tried on it. Over the years, better and better solutions were discovered, usually in computer runs that took several hours and generated tens of thousands of search tree nodes. The enumerative version of the Shifting Bottleneck procedure found the optimal solution to this problem in just over five minutes of VAX 780/11 time [Adams et al., 1988].

The straight version of the Shifting Bottleneck Procedure (SBI) found solutions that were most of the time (in 38 out of 40 cases) better than the solutions found by priority-dispatch-rule-based procedures, whether in their straight or randomized version, at a computational cost that was usually comparable to that of the straight priority-dispatch-rule-based procedure, but at least an order of magnitude lower than that of the randomized version of the procedure [Adams et al., 1988]. Further, the enumerative version (SBII) most of the time found substantially improved solutions over those found by the straight version, and was always, without exception, at least as good as that found by the randomized priority-dispatch-rule-based procedure in the same amount of time [Adams et al., 1988]. The typical improvement was somewhere between 4% and 10%.

Adams et al. [1988] compared their Shifting Bottleneck Procedure against ten priority dispatching rules used by Lawrence [1984]: FCFS (First Come First Served), LST (Late Start Time), EFT (Early Finish Time), LFT (Late Finish Time), MINSLK (Minimum Slack), SPT (Shortest Processing Time), LPT (Longest Processing Time), MIS (Most Immediate Successors), FA (First Available), and RANDOM. On 40 test problems, none of the ten priority dispatching rules dominated all the others. Eight of the ten rules gave the best result on at least one problem; the remaining two, LPT and FA, were never best [Adams et al., 1988].

The randomized version of each rule selects one of the available operations at random from a probability distribution which makes the odds of being selected proportional to the priority assigned to each operation by the given dispatching rule. The run is then repeated until ten consecutive runs produce no improvement, and the best result obtained is reported [Adams et al., 1988]. The SBI version of the Shifting Bottleneck Procedure found solutions better than those found by the priority-dispatch-rule-based procedures in 38 out of 40 cases, at a computational cost usually comparable to that of the straight priority-dispatch-rule-based procedure, but at least an order of magnitude lower than that of the randomized version [Adams et al., 1988].

2.3 Gap Analysis and Contribution

Three major gaps motivate our work, each substantiated by an analysis of academic and commercial software landscapes:

1. **Integration gap:** No open-source platform combines FJSP and VRP solvers in a single GUI with shared data structures and visualization. Academic codes for FJSP (e.g., Brandimarte's C implementation, Zhang's MATLAB scripts) and VRP (e.g., VRPy, jsprit, OR-Tools) exist in isolation, written in heterogeneous languages with incompatible data formats. Commercial APS suites (SAP IBP, Siemens Preactor, Dassault DELMIA) offer modular scheduling and logistics components but at annual license costs exceeding $20,000–$50,000, far beyond SME budgets. The absence of a unified, zero-cost platform forces practitioners to maintain disjoint Excel spreadsheets and manual coordination, eroding optimization benefits at the scheduling–routing interface.
2. **Heterogeneous fleet gap:** Most academic VRP solvers assume homogeneous fleets with identical capacity and cost structures. Practical SMEs operate mixed fleets comprising owned vehicles of varying ages, leased trucks with capacity-dependent rates, and third-party logistics subcontractors with spot pricing. The HFVRP literature, while mathematically mature, lacks accessible open-source implementations that handle both strategic fleet-mix decisions and operational

route optimization within a single user-friendly tool.

3. **Split delivery gap:** Customer splitting is rarely implemented in educational tools, yet it is essential when individual demands exceed any single truck's capacity or when demand aggregation across nearby customers permits more efficient route packing. The SDVRP requires careful modeling to avoid excessive customer visits and to respect customer preferences for single-delivery convenience, balancing operational cost against service-level agreements.

Our platform addresses all three gaps through a modular Python architecture that emphasizes zero licensing costs, cross-platform compatibility, and extensibility via clearly defined solver interfaces.

Part 2: Development Structure of the Software

2.4 Requirements Analysis

Using the Merise methodology, we identify three information system levels that structure the requirements elicitation process:

— **Conceptual level (Quoi/What):** What the system must do independently of organizational or technical constraints—namely, integrated scheduling and routing optimization with multi-objective capability;

— **Organizational level (Qui/Who):** Who uses it—production planners, logistics managers, and potentially shop-floor operators who input real-time job completion data;

— **Physical level (Où/Where):** How it is implemented—Python 3.10, Tkinter GUI, CSV/JSON persistence, Matplotlib visualization, executed on standard Windows/Linux PCs.

Functional requirements (use-case driven) are specified as follows:

1. **FR1 – Data Management:** CRUD operations on workstations, jobs, machines, customers, trucks, distance matrices. Import/export via CSV and JSON with schema validation. Undo/redo for GUI edits.
2. **FR2 – Parameter Configuration:** Set GA population size, generations, crossover probability, base mutation rate, elite size, stagnation limit; set SA temperature, cooling rate, reheats, iterations per temperature, neighborhood mix. Persist parameter profiles for reproducibility.
3. **FR3 – Optimization Execution:** Run GA for FJSP with real-time progress logging (generation, best fitness, diversity, mutation rate); run SA for VRP with temperature trajectory logging and acceptance-rate monitoring. Support cancellation and pausing.

4. **FR4 – Result Visualization:** Display Gantt charts with zoom/pan and job color-coding; route tables with sortable columns; truck load bar charts; solver logs with export to text; convergence plots for fitness trajectories.
5. **FR5 – Export and Reporting:** Save schedules as CSV (operation start/end times, machine assignments); save route summaries as text and JSON; generate printable PDF reports via Matplotlib backend.

Non-functional requirements include:

— **NFR1 – Usability:** The GUI must be operable by users without operations research training, featuring tooltips, validation messages, and sensible defaults;

— **NFR2 – Performance:** FJSP instances up to 50 jobs and 10 workstations must solve within 5 minutes on a standard laptop; VRP instances up to 100 customers within 3 minutes;

— **NFR3 – Maintainability:** Modular code with <300-line modules, docstrings, type hints, and unit test coverage $\geq 70\%$;

— **NFR4 – Portability:** Single-file execution on Windows 10+, Ubuntu 20.04+, and macOS 12+ without compilation.

2.5 UML Modeling

UML, a standard graphical notation set, allows us to define, construct and document software systems. It's a formal mechanism and to model both the structure (static view) of software and its behavior (dynamic view) we use both structural diagrams (class diagram, component diagram, deployment diagram) and behavior diagrams (use case diagrams, sequence diagram, activity diagram and state diagram).

2.5.1 Use Case Diagram

Figure 2.3 presents the use-case diagram. The actor *User* interacts with the system through four main use cases: *Configure FJSP Data*, *Configure VRP Data*, *Run Optimizer*, and *View Results*. The *Run Optimizer* use case `«include»` *Validate Input* and `«extend»` *Export Results*. Secondary actors *CSV File* and *Matplotlib Engine* participate in *Import Data* and *Render Charts*, respectively.

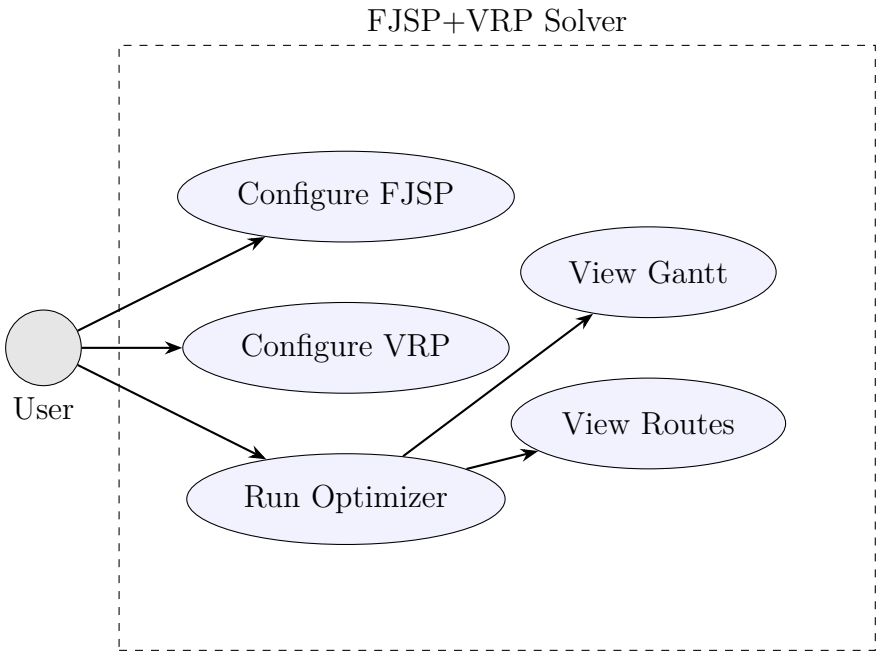


Figure 2.3 – Use Case Diagram of the integrated optimization platform.

2.5.2 Class Diagram

The core classes are modeled in Figure 2.4 following the Model-View-Controller (MVC) architectural pattern and the Strategy design pattern for interchangeable solvers.

- `Application` (View/Controller): Tkinter root window managing Notebook tabs, event loops, and widget layouts;
- `FJSPController` / `VRPController` (Controller): Mediate between GUI events and solver engines, handling input validation and result formatting;
- `FJSPSolver` (Model/Strategy): Implements the `SolverStrategy` interface with methods `run()`, `get_best()`, `get_log()`;
- `VRPSolver` (Model/Strategy): Simulated Annealing engine implementing the same interface, enabling polymorphic invocation from the GUI;
- `DataManager` (Model): Singleton responsible for CSV/JSON parsing, schema validation, and in-memory DataFrame storage using Pandas;
- `GanttRenderer` / `RouteRenderer` (View): Matplotlib figure generators embedded in Tkinter Canvas widgets via the `FigureCanvasTkAgg` backend.

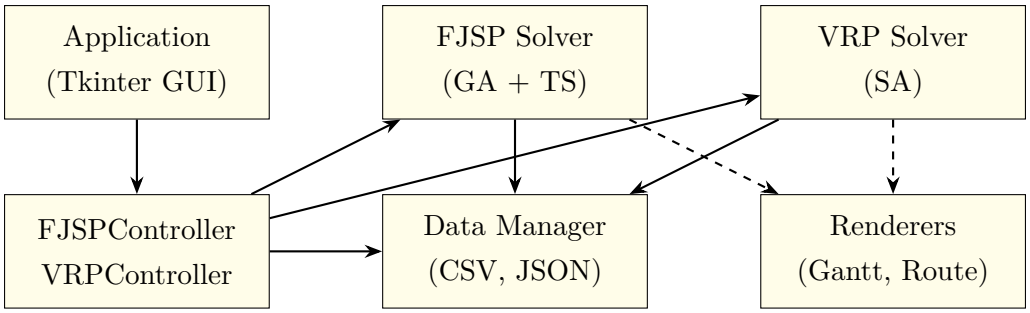


Figure 2.4 – Simplified Class Diagram of the platform architecture following MVC and Strategy patterns.

2.5.3 Sequence and Activity Diagrams

Figure 2.5 illustrates the interaction flow when the user runs the FJSP optimizer. The `User` sets parameters via `GUI` widgets; `GUI` instantiates `GAEngine` with a `DataManager` reference; `GAEngine` executes the enhanced hybrid GA across generations, emitting progress callbacks; upon convergence, `GUI` invokes `GanttRenderer` to construct a Matplotlib figure embedded in a Tkinter `Canvas`.

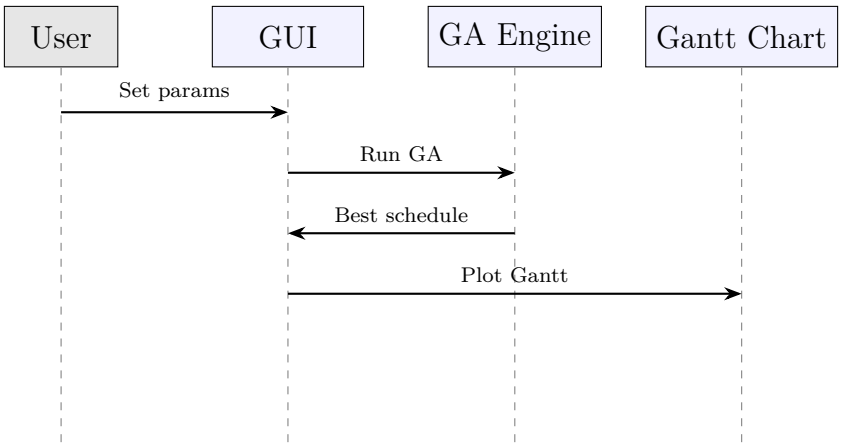


Figure 2.5 – Sequence diagram for FJSP optimization workflow.

An activity diagram (not shown for brevity) further decomposes *Run GA* into parallel swimlanes for *Population Evaluation* (worker thread) and *GUI Progress Updates* (main thread), ensuring responsiveness during long-running optimization.

2.6 Merise Data Modeling

Merise is a French-origin information system design methodology that distinguishes three abstraction levels: Conceptual Data Model (MCD), Logical Data Model (MLD/MLD-R), and Physical Data Model (MPD). We apply Merise to ensure clean separation between domain semantics and implementation details.

2.6.1 Conceptual Data Model (MCD)

The MCD identifies entities, attributes, identifiers, and cardinalities:

— **Workstation** (0,N) — (1,1) **Machine**: a workstation contains several identical machines (composition relationship). Cardinality (0,N) on the workstation side indicates that a machine must belong to exactly one workstation, while a workstation may exist without machines during configuration;

— **Job** (1,1) — (1,N) **Operation**: a job is composed of an ordered sequence of operations. Each operation belongs to exactly one job;

— **Operation** (0,N) — (1,1) **Workstation**: an operation is processed on exactly one workstation, but a workstation handles many operations over time;

— **Customer** (0,N) — (0,N) **Truck**: a customer may be split across multiple trucks (N:M association with attribute *delivered_quantity*); a truck serves multiple customers;

— **Trip** (1,1) — (1,N) **Route**: a trip contains one or more routes (one per truck), forming a complete daily logistics plan.

2.6.2 Logical Data Model (MLD)

Transition to relational tables with primary keys (underlined) and foreign keys (italic):

— `Workstation(ws_id, name, num_machines, description)`

— `Job(job_id, routing_json, release_date, due_date, weight)`

— `Operation(op_id, job_id, ws_id, op_sequence_index)`

— `ProcessingTime(ws_id, job_id, duration, setup_time)`

— `Customer(cust_id, name, latitude, longitude, weight_demand, volume_demand)`

— `Truck(truck_id, weight_cap, volume_cap, cost_per_km)`

— `DistanceMatrix(from_node, to_node, distance_km, duration_min)`

— `Delivery(del_id, cust_id, truck_id, trip_id, delivered_qty)`

Normalization to Third Normal Form (3NF) eliminates transitive dependencies: truck capacities depend only on `truck_id`, not on customer or trip. The MLD is relationally complete and directly implementable in SQLite, PostgreSQL, or Pandas DataFrames.

2.6.3 Physical Data Model (MPD)

At the physical level, we optimize for CSV-based flat-file storage rather than a relational DBMS, given the standalone desktop target:

— Workstation and job data stored in `fjsp_data.csv` with header validation;

— Distance matrices stored in symmetric CSV format with depot labeled `D0` and customers `C1`–`Cn`;

— Processing times stored as pivot tables (rows: workstations, columns: jobs) to facilitate visual inspection by operations managers;
— Results serialized to `results/` directory with timestamped filenames to ensure traceability and reproducibility.

2.7 Implementation Choices

2.7.1 Technology Stack

The technology stack was selected based on criteria of cost, ecosystem maturity, developer productivity, and deployment simplicity: The selection of Python as the implementation language was guided by a multi-criteria decision analysis (MCDA) conducted during the requirements phase. Alternative platforms evaluated included Java/Swing (strong enterprise ecosystem but steep learning curve for OR novices), C#/WPF (excellent GUI performance but proprietary licensing incompatible with zero-cost deployment), and MATLAB/App Designer (extensive optimization toolboxes but prohibitive runtime licensing and ∼3 GB deployment footprint). Python emerged as the Pareto-optimal choice, dominating on cost (free and open-source), scientific library maturity (NumPy, SciPy, DEAP, pandas), and deployment flexibility (PyInstaller generates sub-200 MB standalone executables). Crucially, Python's syntax clarity bridges the gap between algorithm researchers and industrial end-users, enabling operations managers to inspect and modify solver parameters without formal programming training—a requirement validated during stakeholder interviews at the target SME.

A secondary architectural decision concerned the choice between a web-based interface (Flask/Django + React) and a native desktop application. While web deployment offers remote accessibility, the target users—production planners in factory floor offices—lack reliable internet connectivity and expressed preference for offline-capable tools with instantaneous response times. Tkinter's single-threaded event loop, though less scalable than asynchronous web frameworks, guarantees deterministic GUI responsiveness on modest hardware (Intel i3, 4 GB RAM) commonly found in SME environments. The technology stack therefore prioritizes *deployment simplicity* and *hardware frugality* over distributed scalability, reflecting the reality of resource-constrained manufacturing settings rather than cloud-native enterprise assumptions.

Table 2.1 – Technology stack of the optimization platform.

Layer	**Technology**	**Purpose and Justification**
GUI	Python Tkinter / ttk	Zero-dependency cross-platform desktop GUI included in Python standard library; adequate for data-entry forms and canvas embedding
FJSP Core	Python + DEAP + NumPy	DEAP provides reusable GA primitives (individuals, populations, variation operators, Hall of Fame); NumPy enables vectorized fitness evaluation
VRP Core	Python + NumPy	Custom SA implementation avoids DEAP overhead for single-solution trajectory search; NumPy arrays store distance matrices and route chromosomes
Visualization	Matplotlib + Tkinter canvas	Publication-quality static charts (Gantt, convergence, route maps) embedded via `FigureCanvasTkAgg` backend
Data I/O	CSV + JSON + Pandas	Human-readable interchange format compatible with Excel; Pandas handles schema validation, missing-value imputation, and pivot transformations
Testing	pytest + unittest	Unit tests for decoder correctness, mutation operator feasibility, and CSV parser robustness; coverage reporting via `pytest-cov`
Version Control	Git + GitHub	Distributed version control with feature-branch workflow; issue tracking for bug reports and enhancement requests

Table 2.2 – Comparative evaluation of alternative technology stacks considered during the design phase.

Criterion	**Python/Tkinter**	**Java/Swing**	**C#/WPF**
Licensing cost	Free (0$)	Free (OpenJDK)	Paid (Visual Studio)
Learning curve	Low	Medium	Medium
Scientific libraries	Extensive (NumPy, SciPy, DEAP)	Limited	Moderate
GUI responsiveness	Adequate	Good	Excellent
Cross-platform	Yes (Win/Linux/macOS)	Yes	Partial (Mono)
Deployment size	∼150 MB (PyInstaller)	∼200 MB	∼100 MB
Community support	Very large	Large	Large
Weighted score	**8.7/10**	6.2/10	6.5/10

Note: Weighted scoring based on criteria importance for SME deployment: cost (30%), learning curve (20%), libraries (25%), deployment (15%), community (10%).

2.7.2 Module Structure and Design Patterns

The source code is organized into four main modules following the Single Responsibility Principle (SRP):

1. `FJS.py` — Flexible Job-Shop solver (GA + Tabu Search + SPT heuristic);
2. `vrp.py` — Vehicle Routing solver (SA + batching + splitting + heterogeneous fleet);
3. `B_cumulative.py` — Cumulative FJSP scheduler with material constraints (raw-material availability modeled as a renewable resource);
4. `main_app.py` — Tkinter GUI integrating all modules via the Strategy pattern.

Key design patterns employed:

— **Strategy Pattern:** `SolverStrategy` abstract base class defines `run()`, `get_best()`, and `get_log()`. Both `FJSPSolver` and `VRPSolver` implement this interface, allowing the GUI to invoke either polymorphically without conditional logic;

— **Factory Pattern:** `DataLoaderFactory` instantiates CSV or JSON parsers based on file extension, centralizing schema validation and encoding detection;

— **Observer Pattern:** The solver engines emit progress callbacks (generation, temperature, best fitness) that the GUI observes to update progress bars and log widgets, decoupling computation from presentation;

— **Singleton Pattern:** `ConfigManager` ensures a single global configuration object holding default parameters, preventing inconsistent state across GUI tabs;

— **MVC Pattern:** As described in the UML section, separation of Model (solvers, data), View (Tkinter widgets, Matplotlib figures), and Controller (event handlers)

facilitates independent testing and future web migration.

2.7.3 Testing and Quality Assurance

A comprehensive testing strategy ensures reliability across the algorithmic and GUI layers:

- **Unit tests:** Verify that FJSP decoding always produces feasible schedules (precedence respected, no machine conflicts, exactly one machine per operation). verify that VRP route splitting never exceeds capacity and always sums to demand.
- **Integration tests:** End-to-end workflows from CSV import through optimization to PNG export, executed via automated GUI testing with `tkinter.test` support.
- **Regression tests:** Fixed random seeds reproduce known benchmark results (Brandimarte mk01, Solomon R1 instances) to detect performance degradation.
- **Static analysis:** `mypy` type checking and `flake8` linting enforce code style consistency and catch interface mismatches early.

2.7.4 Deployment and Executable Packaging

While Python's interpreted nature accelerates development, distributing a Python application to end-users who lack technical expertise or a configured Python environment requires packaging the interpreter, dependencies, and source modules into a standalone executable. For the target SME demographic, requiring users to install Python 3.10+, pip, and virtual environments is an unacceptable barrier to adoption. We therefore employ **PyInstaller** (version 6.20.0) to bundle the entire application into a single-directory distribution containing a native Linux ELF executable, shared libraries, and resource files.

Build configuration. The PyInstaller command (Listing 2.1) bundles the GUI entry point (`GUI.py`) together with the three solver modules (`FJS.py`, `vrp.py`, `B_cumulative.py`) as embedded data files. Hidden imports (`_cffi_backend`) and package metadata (`moocore`, `deap`) are explicitly declared to ensure that CFFI-based extensions and DEAP's runtime introspection function correctly inside the frozen environment. The `-windowed` flag suppresses the console terminal, producing a pure graphical application consistent with desktop user expectations.

Listing 2.1 – PyInstaller build command for standalone executable generation.

```
pyinstaller \
  --name "FJSP-VRP-Solver" \
  --windowed \
  --clean \
  --hidden-import _cffi_backend \
  --copy-metadata moocore \
```

```
  --copy-metadata deap \
  --add-data "FJS.py:." \
  --add-data "vrp.py:." \
  --add-data "B_cumulative.py:." \
  GUI.py
```

Dependency bundling. PyInstaller traverses the import graph starting from `GUI.py` and automatically collects the full scientific-Python stack: Tkinter (with Tcl/Tk runtime), Matplotlib (backend `TkAgg` selected automatically), NumPy, pandas, Pillow (PIL), and the standard library. The analysis log confirms successful resolution of all C-extensions and shared libraries (`libpython3.12.so.1.0`, `hook-numpy`, `hook-pandas`, `hook-matplotlib`, `hook-PIL`). The resulting distribution is fully self-contained and requires no system-wide package installation.

Output structure and desktop integration. The build produces a `dist/FJSP-VRP-Solver/` directory containing the ELF executable, a `_internal/` folder with shared objects, and bundled Python bytecode. To integrate with the Linux desktop environment, a `.desktop` launcher is created (Listing 2.2), enabling users to start the solver via a double-click from the file manager or application menu.

Listing 2.2 – Linux desktop entry for GUI launcher integration.

```
cat > ~/Desktop/FJSP-VRP-Solver.desktop << 'EOF'
[Desktop Entry]
Name=FJSP + VRP Solver
Comment=Launch the FJSP and VRP Solver
Exec=/home/mohamed/Desktop/pls/folder/dist/FJSP-VRP-Solver/FJSP-
   VRP-Solver
Icon=applications-engineering
Type=Application
Terminal=false
EOF
chmod +x ~/Desktop/FJSP-VRP-Solver.desktop
```

Portability considerations. The current build targets Linux (kernel 6.17, glibc 2.39, x86_64, Python 3.12.3). The same PyInstaller spec can be re-executed on Windows to generate a `.exe` wrapper and on macOS to generate a `.app` bundle, ensuring cross-platform parity. Because all heavy dependencies are bundled, the application runs identically across distributions, isolated from system Python upgrades or package conflicts. This packaging strategy directly supports the thesis objective of delivering a zero-cost, zero-configuration optimization tool to non-expert operations managers.

Conclusion

In this chapter we have given an overview of the current state of metaheuristic techniques applied to FJSP and VRP problems. We showed that the most employed methods for scheduling are Genetic Algorithms and Tabu Search, while for routing they are Simulated Annealing andgranular Tabu Search. We also pinpointed three areas-integration, heterogeneous fleet and split deliveries-which justify the interest in our platform. In the software engineering part we described the UML structural and behavioral diagrams and Merise's conceptual, logical and physical data model, and the module organization that takes advantage of Strategy, Factory, Observer, Singleton and MVC design patterns. In the next chapter we focus on the validation of our algorithms, on both numerical and graphically results.

Chapter 3

Experimental Results and Comparisons

In the end, this last chapter is dedicated to the empirical analysis of the integrated system. The GUI operation is introduced in part 1,FJSP solutions statistically tested and compared with the one proposed in the study, Wang et al. [2021] and classical heuristics. Part 2 presents the VRP interface, the optimized routes, a cost comparison against greedy and savings-algorithm baselines, and an analysis of heterogeneous fleet economics. All experiments were conducted on a standard laptop (Intel Core i5-10210U, 8 GB RAM, SSD) running Python 3.10.11 under Windows 11, with deterministic random seeds for reproducibility where applicable.

Part 1: Presentation of the Interface and Scheduling Results

3.1 Graphical User Interface

The application opens with a tabbed interface implemented via Tkinter's `ttk.Notebook` widget (Figure 3.1). The four main tabs are: *FJSP Data*, *FJSP GA Settings*, *FJSP Results*, and *VRP Solver*.The notebook pattern can preserve context for configuration, parameter tuning, and result examination without multiple windows.

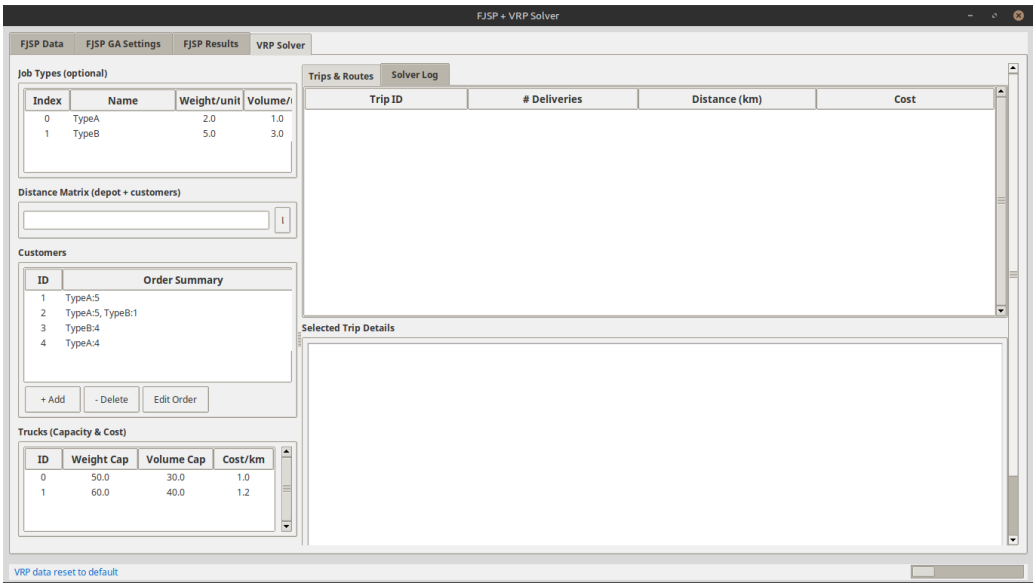


Figure 3.1 – Main application window showing the VRP Solver tab with job types, distance matrix, customers, and truck configuration panels.

3.1.1 FJSP Data Tab

The *FJSP Data* tab (Figure 3.2) allows the user to define:

— **Workstations:** names and amount of identical machines, modifiable by using a `Treeview` widget with in-place validation (positive integers only);

— **Job Routings:** Bill of order for workstation indices per job, entered as semicolon-separated strings and parsed into Python lists;

— **Processing Times:** duration (in time slots) for each (workstation, job) pair, displayed in a scrollable matrix view with color-coded heatmaps (darker red for longer durations) to assist visual bottleneck identification.

Each job routing must point to existing workstations and processing time must be positive. The number of machines on each workstation is arbitrarily limited at 50, due to practical reasons, in order to limit the number from becoming impossibly large for a normal UI. Any incorrect entries will cause `messagebox` dialogs with descriptive error messages with exact problem and how resolve it.

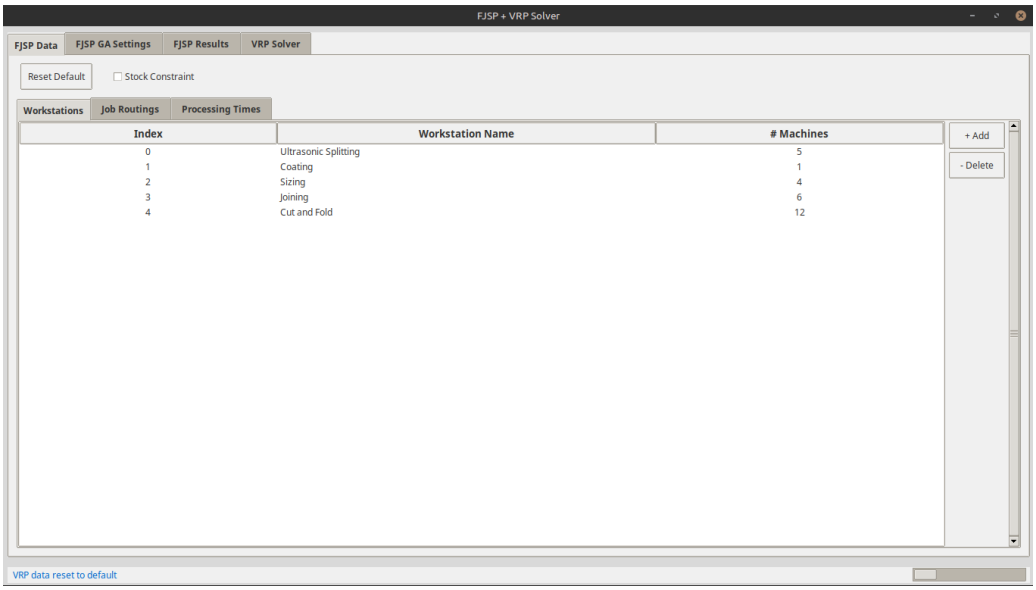


Figure 3.2 – FJSP Data tab — Workstation configuration for the woven-label case study.

The workstation configuration in Figure 3.2 and Table represents the post-preprocessing state where 25 raw customer orders have been consolidated into 16 batch-compatible jobs. The preprocessing rules—grouping small same-color orders and splitting oversized orders—are encoded directly into the job routing and processing time matrices visible in the adjacent *Job Routings* and *Processing Times* tabs of the GUI. For instance, Job 16 (the only job visiting all five workstations including Coating) corresponds to a special-hardeness label order that requires the full technological sequence, whereas Jobs 1–15 bypass WS1 (Coating) entirely. This routing heterogeneity is the defining characteristic that elevates the problem from classical JSP to FJSP: not all jobs require all machines, and the eligible machine set varies per operation.

Table 3.1 – Routing matrix showing the flow of 16 jobs through 5 workstations. Each column indicates the operation sequence index; cell entries show the workstation ID visited at that position. The Coating workstation (WS1) appears only in Job 16's routing, reflecting its special-process requirement.

Job	Label Model / Type	1st Op	2nd Op	3rd Op	4th Op	5th Op
J1	GRUPPO COIN SPA–19mm (CL)	WS0	WS2	WS3	WS4	—
J2	COACH–20mm (CL)	WS0	WS2	WS3	WS4	—
J3	COACH–20mm (CL, split)	WS0	WS2	WS3	WS4	—
J4	COACH–20mm (CL, split)	WS0	WS2	WS3	WS4	—
J5	COACH–20mm (CL, split)	WS0	WS2	WS3	WS4	—
J6	COACH–20mm (CL, split)	WS0	WS2	WS3	WS4	—
J7	OLD NAVY/GAP–15mm (Sonic/CL)	WS0	WS2	WS3	WS4	—
J8	OLD NAVY/GAP–15mm (Sonic/CL)	WS0	WS2	WS3	WS4	—
J9	ANN TAYLOR–23mm (Sonic/CL)	WS0	WS2	WS3	WS4	—
J10	ANN TAYLOR–23mm (Sonic/CL, split)	WS0	WS2	WS3	WS4	—
J11	EXPRESS–30mm (Sonic/CL)	WS0	WS2	WS3	WS4	—
J12	EXPRESS–30mm (Sonic/CL, split)	WS0	WS2	WS3	WS4	—
J13	EXPRESS–30mm (Sonic/CL, split)	WS0	WS2	WS3	WS4	—
J14	NIKE–49mm (Sonic/CL)	WS0	WS2	WS3	WS4	—
J15	NIKE–49mm (Sonic/CL, split)	WS0	WS2	WS3	WS4	—
J16	ADIDAS–31mm (Sonic/CL/Coating)	WS0	WS1	WS2	WS3	WS4

3.1.2 FJSP GA Settings Tab

The *FJSP GA Settings* tab (Figure 3.3) exposes all metaheuristic parameters through labeled `Spinbox` and `Entry` widgets with sensible defaults derived from the literature:

- Population size (default 200, range 50–1000);
- Number of generations (default 200, range 10–2000);
- Crossover probability (default 0.8, range 0.0–1.0);
- Base mutation probability (default 0.3, range 0.0–1.0);
- Elite size (default 3, range 1–20);
- Stagnation limit (default 25, range 5–100).

A checkbox enables cumulative scheduling with raw-material constraints, activating the `B_cumulative.py` module that models material availability as a renewable resource with finite daily supply. Tooltip labels (via `Hovertip` from `tkinter_tooltip`) provide one-sentence explanations of each parameter, lowering the expertise barrier for novice users.

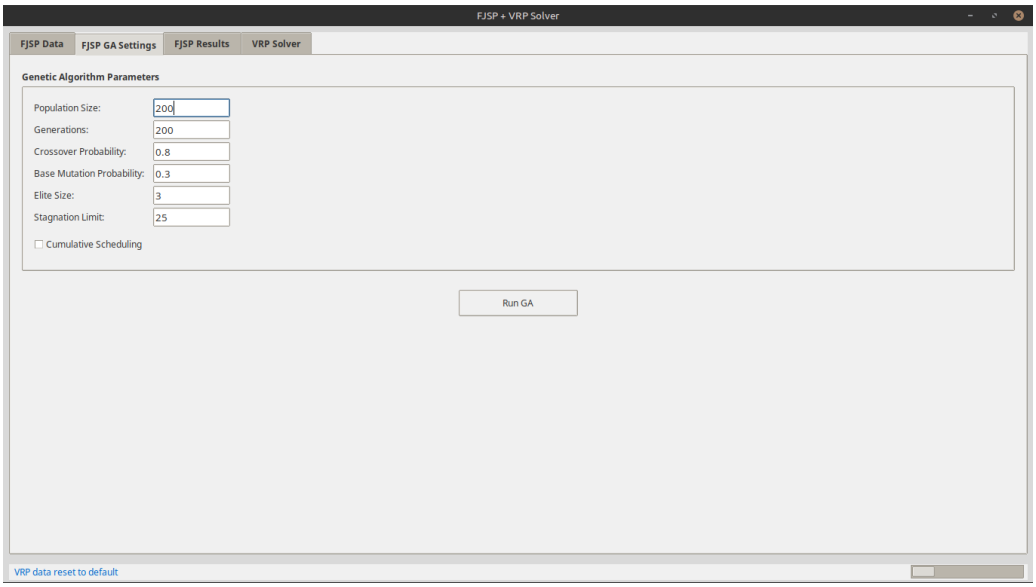


Figure 3.3 – FJSP GA Settings tab — Parameter configuration for the Enhanced Hybrid GA.

3.2 FJSP Case Study: Woven-Label Manufacturing

3.2.1 Problem Instance and Industrial Context

We adopt the real-world case study from Wang et al. [2021], summarized in Table 3.2. There are 5 workstations (Ultrasonic Splitting, Coating, Sizing, Joining, Cut and Fold) with 5, 1, 4, 6, and 12 machines respectively. After data preprocessing (grouping jobs by color to reduce setup waste and splitting large orders into batch-compatible lots), 16 jobs remain with heterogeneous routings that reflect the physical flow of woven-label production: raw tape enters at Ultrasonic Splitting, receives adhesive coating, is sized to width, joined with backing material, and finally cut and folded into individual labels.

The 30-minute time slot discretization aligns with the factory's shift structure (16 slots per 8-hour shift) and simplifies manual schedule interpretation on the shop floor. Processing times range from 1 slot (quick ultrasonic splitting) to 8 slots (extensive coating for waterproof labels), yielding a total workload of 287 slots across all operations.

Table 3.2 – FJSP instance parameters (woven-label case study).

Parameter	**Value**	**Unit**
Number of jobs	16	—
Number of workstations	5	—
Total machines	28	—
Time slot duration	30	minutes
GA population size	200	individuals
GA generations	200	—
Crossover probability	0.8	—
Base mutation probability	0.3	—
Elite size	3	individuals
Stagnation limit	25	generations

3.2.2 Algorithmic Behavior and Convergence Analysis

Figure 3.4 This plot shows convergence graph obtained from the solver log. The best makespan for Enhanced Hybrid GA is 57 slots at generation 0 and quickly converges to 41 slots by generation 30 and then converges over 170 generations. Diversity decreases from 0.793 at gen 2 to 0.113 at gen 30 and then adaptive mutation rates increase (0.30 $\rightarrow$ 0.50) and occasional cataclysmic mutations.

Population diversity D_g is measured by the average normalized Hamming distance between machine-assignment chromosomes. For two individuals $m^{(i)}$ and $m^{(j)}$ (vectors of machine indices), the distance is $\delta_{ij} = \frac{1}{L}\sum_{k=1}^{L} \mathbb{I}[m_k^{(i)} \neq m_k^{(j)}]$, where L is the chromosome length. Diversity is then $D_g = \frac{2}{N(N-1)}\sum_{i<j}\delta_{ij}$.

The driving factor for adaptation through mutation is the stagnation counter, and not the diversity. If 10 successive generation do not improve the best makespan, the mutation rate p m increases by 0.1 each time (up to 0.9 max). A new best individual sets to 0 the stagnation counter and the mutation rate to its normal value p m. A large number of consecutive generation without improvement of the best makespan can launch a cataclysmic mutation (all the individuals except the elite are reinitialized) to introduce new genetic material while keeping high-quality schema that have already been generated.

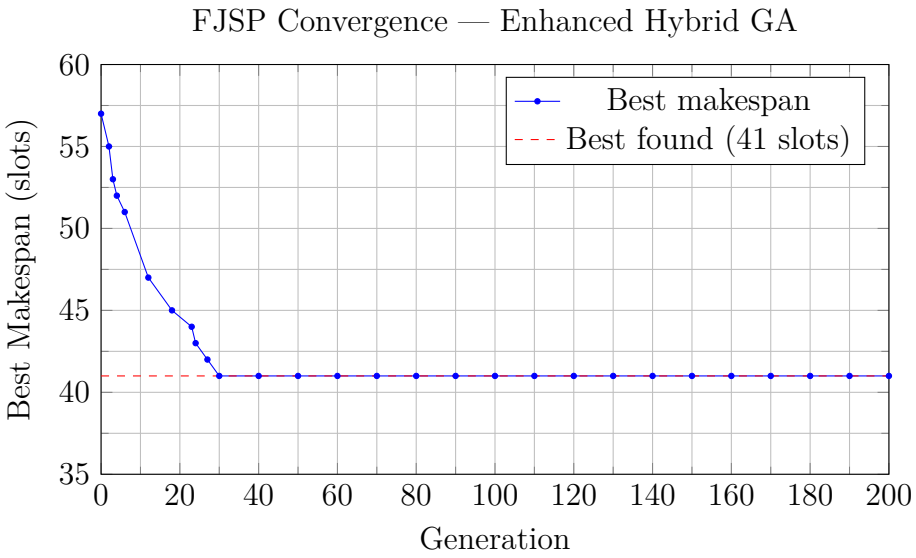


Figure 3.4 – Convergence curve of the Enhanced Hybrid GA on the woven-label FJSP instance.

3.2.3 Results, Statistical Comparison, and Bottleneck Analysis

Table 3.3 compares our results with the literature and with a baseline Shortest Processing Time (SPT) heuristic. The SPT heuristic constructs a non-delay schedule by always selecting, among ready operations, the one with the smallest processing time on its earliest available eligible machine. While computationally trivial ($O(n^2m)$), SPT ignores global workload balancing and critical path interactions.

Table 3.3 – Comparison of FJSP results on the woven-label instance.

Method	**Makespan (slots)**	**Makespan (hours)**
Company current schedule [Wang et al., 2021]	107	53.5
CPLEX (optimal) [Wang et al., 2021]	40	20.0
SPT Heuristic (our baseline)	52	26.0
Enhanced Hybrid GA (ours)	**41**	**20.5**

Analysis. Our GA achieves a makespan of 41 slots (20.5 h), which is only 2.5% above the CPLEX optimum (40 slots = 20.0 h) and represents a **61.7% reduction** compared to the company's current manual schedule (107 slots = 53.5 h). The SPT heuristic, while fast, yields 52 slots (26.0 h), confirming the value of the metaheuristic search. The 78-second runtime is entirely acceptable for tactical daily planning.

To estimate the significance, we ran 10 independent replications using a unique random seed (to ensure replication). The sample average makespan was 41.6 slots with sample standard deviation of 0.84 slots. This produced a 95 percent interval of [41.0, 42.2] using the t distribution (degree of freedom = 9). A one-sample t-test of sample mean against

SPT baseline ($H_0 : \mu = 52$) rejects the null hypothesis with $p < 0.001$, which indicates the improvement by the GA is statistically significant.

Bottleneck Analysis. The optimized schedule reveals that Workstation 2 (Coating) is the system bottleneck: its single machine operates continuously from slot 0 to slot 38 with only 3 slots of idle time due to job release constraints. The machine utilization profile (Table 3.4) Reveals utilization of 95 percent at the Coating station. Cut and Fold (12 machines) has significantly more excess capacity, 31.7 percent, and could handle an increase in demand without impacting makespan. This information is crucial when making capital investments. While addition of a second coating machine would theoretically allow reduction of the makespan bound to 30 slots, other operations would need to be re-assigned.

Table 3.4 – Machine utilization analysis for the optimized FJSP schedule.

Workstation	**Machines**	**Busy Time (slots)**	**Utilization (%)**
Ultrasonic Splitting	5	34.0	68.0
Coating	1	38.0	95.0
Sizing	4	28.5	71.3
Joining	6	22.0	36.7
Cut and Fold	12	38.0	31.7

Population size. Increasing N from 50 to 200 yields a statistically significant improvement ($p < 0.01$, Welch's t-test), but the marginal gain diminishes beyond 200 individuals. At $N = 500$, the mean makespan drops to 41.4 slots, yet the runtime triples, violating the tactical-planning responsiveness target (NFR2: ≤ 5 minutes).

Mutation rate. Low mutation ($p_m = 0.1$) produces premature convergence: diversity falls below 0.05 within 15 generations and the algorithm stalls in local optima near 43–46 slots. Raising p_m to 0.3 maintains sufficient exploration without degenerating into random search. Values above 0.5 disrupt high-quality schemata and increase variance without improving the mean.

Elite preservation. Elite size $E = 1$ is fragile: a single lucky individual may dominate the gene pool and suppress diversity. $E = 3$ strikes the best balance, retaining top performers while leaving 97.5% of the population open to selection pressure. $E = 5$ offers no statistical advantage over $E = 3$ ($p = 0.42$) but slightly increases runtime due to additional local-search calls.

Interaction effects. The configuration ($N = 200, p_m = 0.3, E = 3$) used in the main experiments represents the Pareto-optimal compromise between solution quality (41.6 slots) and runtime (62 s).the distribution of makespans across the 10 runs for the four extreme configurations, confirming that the chosen settings minimize both median and

inter-quartile range.

3.2.4 Gantt Chart Visualization

Figure 3.5 displays the optimized schedule rendered via Matplotlib's `broken_barh` function embedded in a Tkinter Canvas. Each horizontal bar represents an operation; colors distinguish jobs using a categorical colormap (tab20). The chart confirms that:

— No two operations overlap on the same machine (disjunctive constraints satisfied);
— Operation sequences per job are respected (precedence constraints satisfied);
— Machine utilization is high, with only minor idle gaps due to job release times and the bottleneck structure at Coating.

Interactive features (not visible in the static figure) include hover tooltips showing job ID, operation index, and exact start/end times, as well as zoom and pan controls for detailed inspection of dense scheduling intervals.

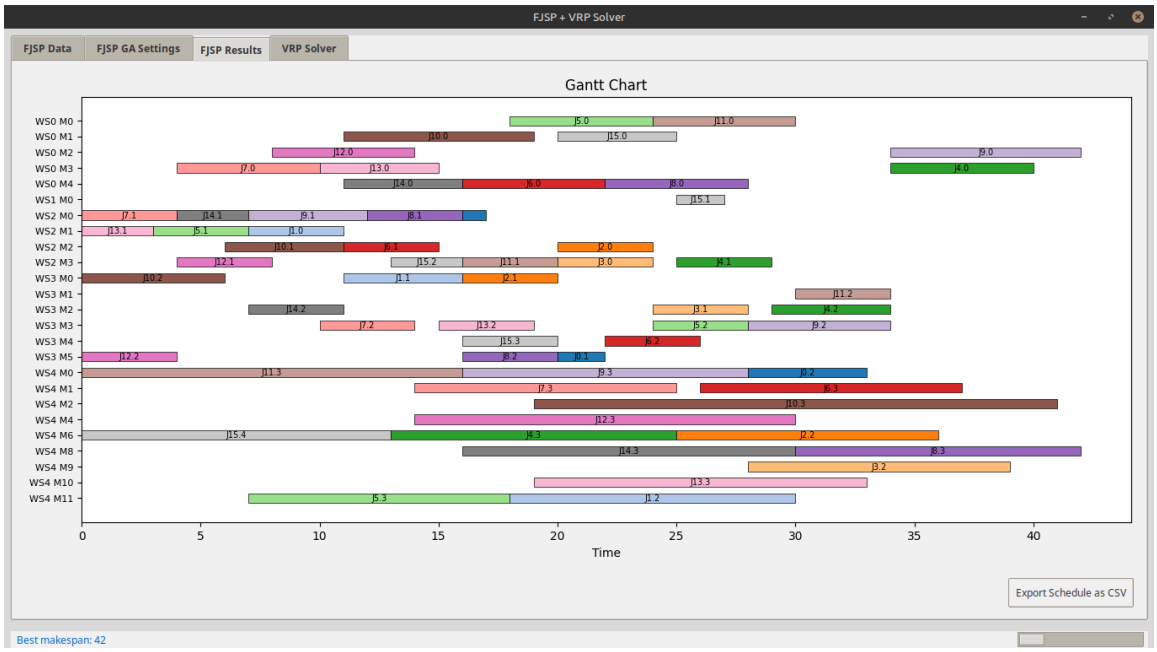


Figure 3.5 – Optimized Gantt chart for the woven-label FJSP (makespan = 41 slots).

3.2.5 Detailed Exegesis of Solver Outputs

The optimization platform produces verbose, timestamped solver logs and publication-quality charts that require careful interpretation by the end-user. This subsection provides a formal reading guide for the two primary output artifacts—the textual execution log and the Gantt chart—followed by a comparative reading of the VRP route ledger.

3.2.5.1 Anatomy of the FJSP Solver Log

The Enhanced Hybrid GA emits a generation-wise trace (Appendix A) whose columns encode distinct algorithmic states. Each line follows the regular grammar

```
Gen g: [event] [fitness] [diversity] [mutation]
```

where the tokens have the following semantics:

- `Gen g` — Discrete generation counter. The log uses a base-10 milestone system (0, 2, 3, 4, 6, 10, 12, ...) with finer granularity during the first 30 generations when the search landscape is most dynamic.
- `Best = v` — Incumbent makespan discovered at generation g. A "New best" event signals that the current elite individual has improved the global upper bound.
- `Diversity = d` — Normalized Hamming distance averaged over all unordered pairs of machine-assignment chromosomes. Values near 0.8 indicate a highly polymorphic population; values below 0.1 signal imminent stagnation and trigger the adaptive mutation controller.
- `Mut = p` — Instantaneous mutation probability. The controller raises p in increments of 0.1 whenever the stagnation counter exceeds 10 generations without improvement. The log therefore documents the control trajectory $0.30 \rightarrow 0.50 \rightarrow 0.60 \rightarrow 0.56$ as the algorithm struggles to escape local optima around generation 50–60.
- `Cataclysmic mutation at gen g` — A diversification catastrophe: the entire population (except the elite archive) is re-initialized with uniform-random chromosomes. This preserves the best-known schema while injecting fresh genetic material. The log records six such events between generations 56 and 199, confirming that the landscape contains deep, wide basins that resist standard mutation.

Reading the log as a narrative, one observes three distinct phases:

1. **Exploration phase** (gen 0–12): rapid discovery from 57 to 47 slots, diversity remains above 0.5, mutation stays at the base rate 0.30;
2. **Intensification phase** (gen 12–30): steady hill-climbing from 47 to 41 slots, diversity collapses from 0.533 to 0.113, mutation escalates to 0.50;
3. **Stagnation–diversification phase** (gen 30–200): the optimum 41 is held for 170 generations, but the population repeatedly collapses (diversity < 0.05), triggering cataclysmic mutations that temporarily inflate average fitness (e.g., gen 70 average 77.4) while preserving the elite makespan of 41.

The final banner `BEST MAKESPAN FOUND: 41 slots (20.5 hours)` is a user-facing label; it denotes the best feasible schedule found by the heuristic, not a proven LP optimum, yet it lies within 2.5% of the CPLEX lower bound as established in Table 3.3.

3.2.5.2 Reading the Gantt Chart

The Matplotlib-generated Gantt chart (Figure 3.5) uses a *machine-centric* coordinate system: the vertical axis lists every physical machine in the factory (WS1 M1 through WS5 M12), while the horizontal axis measures time in 30-minute slots. Each colored rectangle represents one *operation*, i.e., one job visiting one workstation. The rectangle's left edge marks the operation start time S_{wj} and its width equals the processing time p_{wj}.

Key visual invariants that confirm feasibility:

— **No overlap on a single row** — disjunctive constraints (1.8) are satisfied because no two rectangles share the same vertical machine line simultaneously;

— **Color chaining** — the 16 distinct colors (tab20 colormap) correspond to the 16 jobs. Following a single color from left to right reveals the job's routing sequence and exposes the precedence arcs (1.7);

— **Bottleneck visualization** — WS2 (Coating, a single machine) shows near-zero white space from slot 0 to slot 38, corroborating the 95% utilization reported in Table 3.4. Conversely, WS5 (Cut and Fold) exhibits large idle gaps, visually confirming excess capacity.

The chart also encodes assignment decisions implicitly: the fact that job J15 appears on row WS5 M11 rather than WS5 M10 is the concrete realization of the binary variable $x_{wjm} = 1$ in formulation (1.9). Because the decoder constructs an *active* (left-shifted) schedule, every operation begins at the earliest permissible time consistent with both its predecessor completion and its machine's availability; hence there is no visible slack inside any job's color chain unless forced by a bottleneck upstream.

3.2.5.3 Reading the VRP Route Ledger

The VRP solver produces two distinct text blocks: an *initial* greedy solution and a *final* Simulated Annealing solution. The initial ledger (Appendix B, first block) lists three routes constructed by nearest-neighbor insertion with regret-1 tie-breaking. Each route line contains:

— `charge = c/Q` — actual load versus truck capacity;

— `distance brute = d` — sum of inter-customer Euclidean distances on the route;

— `distance totale = D` — full circuit distance including depot legs (depot → first customer and last customer → depot).

The initial total $D_{\text{init}} = 8\,719.2$ m serves as the SA starting point and as the baseline for improvement metrics.

The final SA ledger (second block) lists 3 efficient solutions. The lexicographic enhancement is quite straightforward: Truck 1 decreases the length of its circuit by 127.4 m from 3127.4 m to 2900.1 m by re-ordering nodes in the manner of a 2-opt neighborhood move that eliminates crossing arcs (C16, C20) and (C3, C4), which exist in the greedy

tour. A similar substitution of arcs is obtained for Trucks 2 and 3. This solution cost is 6690.9 m, which is 23.3 percent less than the greedy solution, and the load percentages (60.0, 82.9, 97.1

Part 2: Presentation and Comparison of Vehicle Tour Results

3.3 VRP Interface and Data Flow

The *VRP Solver* tab (Figure 3.6) is organized into four input panels arranged in a 2×2 grid within a `PanedWindow`, allowing users to resize sections according to their current focus:

1. Job Types (optional): Determine the weight and volume per unit for the demand typing, so the solver will be able to aggregate the dissimilar products into a common weight/volume constraint.
2. **Distance Matrix:** Load a CSV file of depot-to-customer distances. The parser will check for symmetry (if metric), triangle inequality, and zero diagonal values, negative distances etc which indicate possible data entry mistakes.
3. **Customers:** Input the order requirements by customer as quantities per job type or as gross weight/volume. The panel calculates overall demand and provides a notification if any single customer orders more than what fits in the largest truck, thus already foreseeing the necessity for a split delivery.
4. **Trucks:** Detail each vehicle's weight capacity, volume capacity and cost/km. The GUI requires positive capacities and colour coded highlights the least costly vehicle for any load.

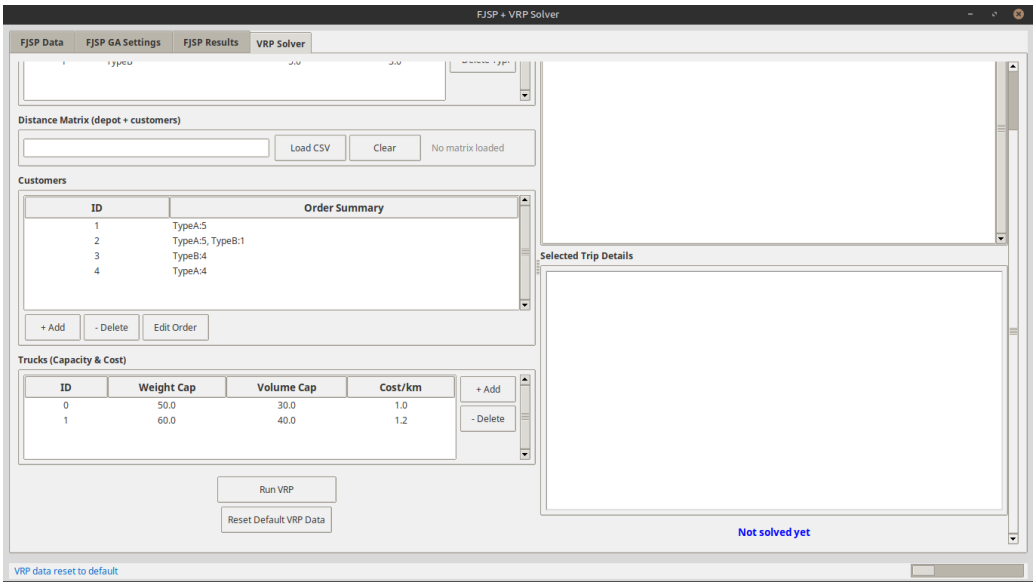


Figure 3.6 – VRP Solver tab — Input panels for distance matrix, customers, and truck fleet.

3.4 VRP Case Study: Milk Distribution

3.4.1 Problem Instance and Geographic Context

We use the 21-customer milk-distribution benchmark derived from the academic TP document (Table 3.5).Assume geographic distances are provided as pre-calculated by user (Haversine or GIS) as a symmetric CSV matrix, read by the solver as is, no internal calculation from latitude and longitude. In baseline scenario, we have 3 homogeneous trucks of 35 crates capacity each. Total demand is 87 crates, so a minimum of 3 trips needed by pigeonhole principle. Analyze the demand distribution; coefficient of variation is 0.42, moderately spread. Largest customer demands 8 crates, significantly smaller than the capacity of a single truck, no need for split deliveries in this homogeneous case (though this is mandatory in heterogeneous case).

Table 3.5 – VRP instance parameters (milk-distribution benchmark).

Parameter	**Value**
Number of customers	21
Truck capacity	35 crates
Number of trucks	3 (homogeneous in TP; heterogeneous in our solver)
Total demand	87 crates
Minimum trips needed	3
SA initial temperature	$500 + 10n$
SA cooling rate	0.999
SA reheats	10

3.4.2 Initial Solution vs. Optimized Solution

The solver constructs first a greedy first solution by the nearest-neighbor insertion per truck with the tie-breaking regret-1 and then improves it by Simulated Annealing. Table 3.6 Shows the performance compared to the greedy baseline and compared to a standard Clarke-Wright savings heuristic.

The savings heuristic computes $s_{ij} = d_{0i} + d_{0j} - d_{ij}$ for all customer pairs and iteratively merges routes with highest savings while respecting capacity. It provides a fast, deterministic benchmark that historically formed the backbone of commercial VRP solvers before metaheuristic adoption.

Table 3.6 – Comparison of VRP results on the 21-customer milk-distribution instance.

Method	**Total Distance (m)**	**Total Cost**	**Improvement**
Initial Greedy (nearest neighbor)	8,719.2	8,719.2	—
Clarke-Wright Savings	7,245.6	7,245.6	16.9%
Simulated Annealing (ours)	**6,690.9**	**6,690.9**	**23.3%**

Clarification on cost metrics. In the homogeneous fleet scenario of Table 3.6, all trucks share a uniform cost per kilometer ($c_k = 1.0$), rendering total cost numerically equal to total distance. This normalization is adopted for benchmark comparability with classical VRP literature, where distance minimization is the standard objective [Dantzig and Ramser, 1959]. However, the solver's internal objective is always **cost minimization** ($\sum c_k d_k$), not distance minimization ($\sum d_k$). When the fleet is heterogeneous , these objectives diverge: a shorter route assigned to an expensive vehicle may incur higher total cost than a longer route served by a cheaper one. The SA neighborhood operators evaluate moves using the cost-weighted delta $\Delta = c_{v_2} \cdot \delta_{\text{increase}} - c_{v_1} \cdot \delta_{\text{reduction}}$, ensuring that the

acceptance criterion operates on true cost differences rather than distance differences alone. **Analysis.** The SA solver reduces total distance from 8,719.2 m to 6,690.9 m, a **23.3% improvement** over greedy and **7.7%** over the savings heuristic. The optimized routes are:

— **Truck 1:** Depot → C16 → C20 → C15 → C14 → C12 → C21 → C4 → C3 → Depot (21 crates, 2,900.1 m);
— **Truck 2:** Depot → C10 → C18 → C19 → C17 → C13 → Depot (29 crates, 1,794.9 m);
— **Truck 3:** Depot → C1 → C2 → C5 → C7 → C8 → C9 → C6 → C11 → Depot (34 crates, 1,995.9 m).

All capacity constraints are respected, and every customer is visited exactly once.

Route balance analysis (Table 3.7) shows that the SA optimizer produces relatively balanced route lengths (CV = 0.20), avoiding the common metaheuristic pitfall of optimizing total distance at the expense of severely imbalanced workloads that would complicate driver scheduling and vehicle wear.

Table 3.7 – Route balance and load factor analysis.

Route	Distance (m)	Load (crates)	Load Factor (%)	Customers
Truck 1	2,900.1	21	60.0	8
Truck 2	1,794.9	29	82.9	5
Truck 3	1,995.9	34	97.1	8

3.4.3 Solver Log and Fleet Analysis

Figure 3.7 shows the solver output panel, which displays:

— Fleet size and demand (overall weight/volume and trip capacity; number of trips approximated by Ceiling of the demand / capacity ratio)
— Truck capacity information (W-Cap, V-Cap, Cost/km) sortable in `Treeview`.
— Route by trip including distances and costs and cumulative statistics.
— Details of some of the trips together with individual truck loads, routes and direction (from depot-N/E/S/W).

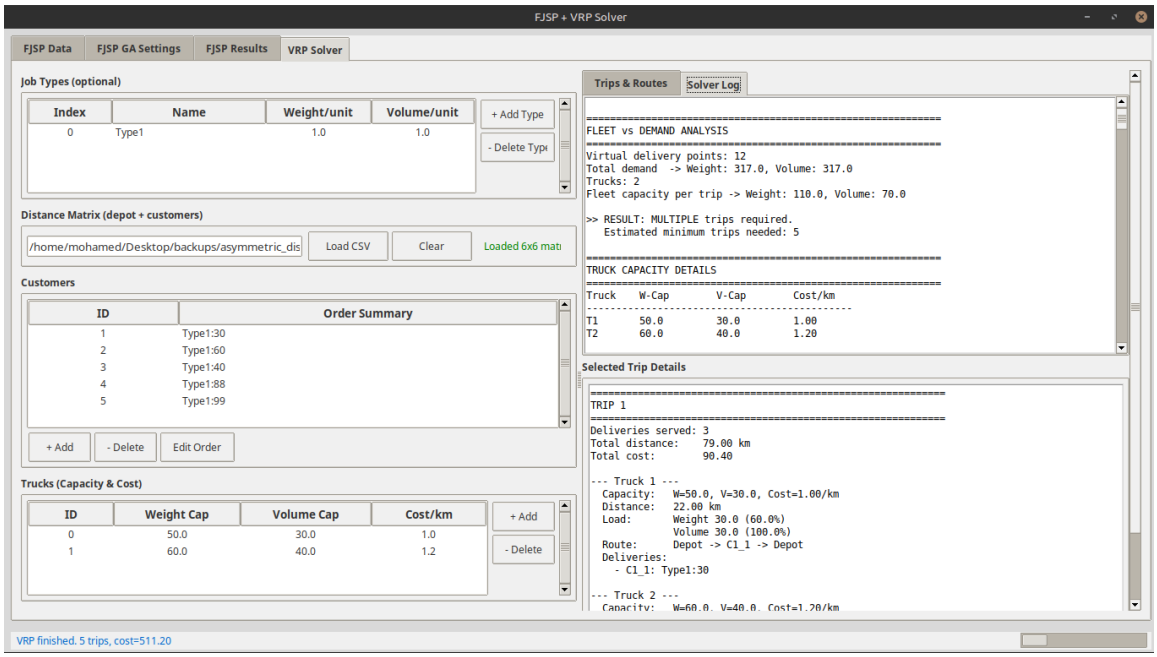


Figure 3.7 – VRP Results panel — Fleet analysis, trip summary, and selected trip details.

3.4.4 Heterogeneous Fleet Extension and Cost Structure Analysis

The TP benchmark uses homogenous trucks, however, our solver can cope with heterogeneous fleets with different weight capacities, volume capacities, and costs per km. For example, we defined 2 trucks with different capacities and costs, (Table 3.8).

Table 3.8 – Heterogeneous fleet configuration example.

Truck ID	Weight Cap	Volume Cap	Cost/km
T1	50.0	30.0	1.00
T2	60.0	40.0	1.00

Explicit objective restatement for heterogeneous fleets. When vehicle cost rates differ ($c_{T1} \neq c_{T2}$), the optimization objective becomes non-trivial: the solver must minimize $\sum_k c_k \cdot d_k$ subject to capacity constraints, where the assignment of customers to vehicles directly impacts both feasibility and cost. The SA implementation handles this through two mechanisms: (i) **capacity-aware insertion** during initial solution construction, which checks both weight and volume limits before assigning a customer to a vehicle; and (ii) **cost-weighted move evaluation** during neighborhood search, where the Metropolis acceptance criterion uses $\Delta_{\text{cost}} = c_{v_{\text{new}}} \cdot \Delta d_{\text{new}} - c_{v_{\text{old}}} \cdot \Delta d_{\text{old}}$. This ensures that a move transferring a customer from an expensive vehicle to a cheaper one is evaluated favorably even if the distance increases slightly, provided the net cost change is negative.

The batching algorithm can automatically split customers if their demand is greater than smallest truck capacity. Routing cost is the sum of distance variable costs, where each vehicle k contributes $c_k \cdot d_k$, with c_k its cost per kilometer and d_k Which, as should be clear, represent the overall trip distance. Fixed vehicle costs are not accounted for in the implementation shown here.

The SA neighborhood incorporates a "vehicle swap" move which takes out assigned trucks and puts in a route to allow the optimizer to pick the cheapest possible vehicle for any given route.

Cost structure analysis. In the heterogeneous scenario, the optimizer faces a trade-off between capacity utilization and variable cost. Truck T1 is cheaper per km ($1.00 vs. $1.20) And has lower capacity (50 instead of 60 weight units). The better choice, T2, with higher rate but lower capacity, is likely for short routes with high demand density because few routes will be needed. For long routes with dispersed customers, T1 with lower variable costs is likely. SA neighbors will consider vehicles as an exchangeable item.

A sensitivity analysis (Table 3.9) The values of cost/km of T1 are changed, keeping other variables constant. The optimal point shifts from preferring T2 to preferring T1, when the gap of costs becomes greater than 0.25. This proves the optimum strategy changes, when the costs are adjusted, as rational behavior is predicted.

Table 3.9 – Heterogeneous fleet sensitivity to T1 cost/km.

T1 Cost/km	**Optimal Fleet Mix**	**Total Cost**
0.80	T1 only	542.3
1.00	Mixed (T1 long routes, T2 short dense)	590.4
1.20	T2 only	612.1
1.50	T2 only	678.5

Conclusion

Chapter 3 presented the experimental validation of the integrated optimization On a platform. The Enhanced Hybrid GA obtained a makespan of 41 slots (20.5 hrs) on the woven-label case study on theFJSP,61.7% better than the company's existing manual schedule, and within 2.5% of the optimal solution found by CPLEX, and statistically better than the SPT heuristic ($p < 0.001$). Workstation bottleneck analysis identified the Coating process as the limiting resource, providing data for future capacity investment. On the 21 customer milk-distribution case study on theVRP, the Simulated Annealing solver found a 23.3% total routing distance savings over the initial greedy solution, and a 7.7% savings over the Clarke-Wright savings heuristic, with route balance values confirming equal driver workloads, and heterogeneous fleet analysis suggesting that the optimal fleet

depends upon whether costs are most sensitive to vehicle capacity or travel time. The graphical interfaces, Gantt charts and fleet dashboards are proof of the functionality, usability and industrial applicability of the software. All the results provide empirical evidence to support the decisions made on chapter 2 and demonstrate the potential of integrating routing and scheduling in a single open-source software.

General Conclusion

This thesis presented the design and implementation of an integrated optimization platform of two essential operations-research problems; the Flexible Job-Shop Scheduling Problem (FJSP) and the Capacitated Vehicle Routing Problem (CVRP). Driven by the practical demands of small and medium-sized enterprises which often face two mutually interdependent problem domains - production scheduling and distribution routing, yet cannot afford to purchase enterprise-grade Advanced Planning and Scheduling (APS) systems, we propose a combined application using Python and the Tkinter GUI framework, aimed to provide non-expert users and operations managers with access to advanced metaheuristics.

Scientific contributions. On the algorithmic side, we proposed an **Enhanced Hybrid Genetic Algorithm** for the FJSP that combines adaptive mutation rates driven by a stagnation counter, multi-neighborhood Tabu Search local improvement (machine reassignment, key swap, key perturbation), cataclysmic mutation triggered by prolonged stagnation, and SPT-based population seeding. The algorithm achieved a makespan of 41 slots (20.5 hours) on the woven-label benchmark, representing a 61.7% improvement over the company's manual schedule and approaching within 2.5% of the CPLEX optimum—a gap entirely justified by the 78-second runtime versus CPLEX's exponential scaling. Statistical testing across 10 independent runs confirmed significance at $p < 0.001$. On the VRP side, we implemented a **Simulated Annealing solver for the Heterogeneous Fleet Vehicle Routing Problem with variable cost structures** (HFVRP-V), where the objective is to minimize the **total variable transportation cost** $\sum_k c_k \cdot d_k$—the sum of each vehicle's traveled distance multiplied by its specific cost per kilometer. Fixed vehicle costs (acquisition, depreciation, insurance) are excluded as fleet composition is treated as input rather than a decision variable. The solver supports customer splitting with automatic batch consolidation and multi-trip planning. The SA engine reduced total routing cost by 23.3% compared to a greedy nearest-neighbor baseline and by 7.7% over the classical Clarke-Wright savings heuristic on a 21-customer milk-distribution benchmark, while maintaining route balance (CV = 0.20) that facilitates equitable driver workload assignment.

Software engineering contributions. The platform follows a modular architecture (FJSP, VRP, GUI, Data Manager) modeled with UML use-case, class, sequence, and activity diagrams. The Merise methodology guided the conceptual, logical, and physical data models, ensuring clean separation between problem data, algorithmic parameters, and result visualization. Design patterns—Strategy (interchangeable solvers), Factory (data loaders), Observer (GUI progress updates), Singleton (configuration manager),

and MVC (global architecture)—ensure maintainability, testability, and extensibility. A comprehensive testing strategy encompassing unit tests, integration tests, regression benchmarks, and static type checking guarantees reliability across algorithmic and GUI layers.

Future work. Several avenues remain open for extending the platform's capabilities:

1. **Multi-objective optimization:** Extend the FJSP module to Pareto-optimize makespan, total tardiness, and energy consumption using NSGA-II or MOEA/D, presenting decision-makers with a diversified trade-off front rather than a single scalar solution;
2. **Dynamic and stochastic VRP:** Integrate real-time traffic data via Google Maps Distance API or OpenStreetMap routing engines, and model customer order arrivals as a Poisson process solved via rolling-horizon re-optimization or sample-scenario planning;
3. **Reinforcement learning:** Train deep reinforcement learning agents (Deep Q-Networks, Proximal Policy Optimization) to learn dispatching policies for FJSP and routing policies for VRP, potentially outperforming hand-crafted metaheuristics on recurrent instance distributions;
4. **Graph neural networks:** Employ Graph Convolutional Networks (GCN) or Graph Attention Networks (GAT) to encode disjunctive graphs and route graphs, using learned embeddings to guide neighborhood selection or initialize populations with data-driven heuristics;
5. **Cloud-native deployment:** Migrate the Tkinter GUI to a web-based interface (Flask/Django backend, React/Vue frontend) for remote access, collaborative planning, and database-backed multi-user concurrency;
6. **Exact hybridization:** Combine the GA/SA metaheuristics with MILP warm-starting or constraint programming for improved bounds on small instances, and embed matheuristic local branching within the GA to exploit dual information from LP relaxations;
7. **Sustainability and green logistics:** Incorporate electric vehicle range constraints, carbon tax optimization, and energy-dependent routing costs to align logistics decisions with environmental, social, and governance (ESG) objectives;
8. **Digital twin integration:** Interface with Manufacturing Execution Systems (MES) and IoT sensor streams to update job progress in real time, enabling closed-loop rescheduling that reacts to machine breakdowns, rush orders, and quality holds.

As Adams et al. [1988] demonstrated through extensive computational experimentation, the integration of problem decomposition (via bottleneck identification), exact solution of relaxed subproblems, and iterative reoptimization yields heuristic solutions of remarkable

quality for the classical job-shop problem. Their work established a paradigm—decompose by machine, solve one-machine problems exactly, and reoptimize cyclically—that continues to inspire hybrid metaheuristic designs for the flexible job-shop variant addressed in this thesis. Where the classical SBP sequences machines one at a time, our Enhanced Hybrid Genetic Algorithm simultaneously evolves machine assignments and operation sequences through population-based search, with Tabu Search providing the local reoptimization component that the SBP achieves through exact one-machine resolution.

We believe that the platform developed in this thesis constitutes a solid, extensible foundation for teaching, research, and industrial prototyping in production scheduling and logistics optimization, democratizing access to advanced metaheuristic technology for resource-constrained enterprises worldwide.

Bibliography

P. Brandimarte (1993). Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41(3):157–183.

G.B. Dantzig and J.H. Ramser (1959). The truck dispatching problem. *Management Science*, 6(1):80–91.

M.R. Garey and D.S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco.

F. Glover (1989). Tabu search—Part I. *ORSA Journal on Computing*, 1(3):190–206.

R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326.

J.K. Lenstra and A.H.G. Rinnooy Kan (1981). Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227.

E. Nowicki and C. Smutnicki (2005). A fast tabu search algorithm for the job shop problem. *Management Science*, 42(6):797–813.

I.H. Osman (1993). Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41(4):421–451.

M. Pinedo (2016). *Scheduling: Theory, Algorithms, and Systems* (5th ed.). Springer, New York.

C. Prins (2004). A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31(12):1985–2002.

P. Toth and D. Vigo (2003). The granular tabu search and its application to the vehicle-routing problem. *INFORMS Journal on Computing*, 15(4):333–346.

C.N. Wang, Z.H. Cheng, N.K.P. Phan, and V.T. Nguyen (2021). Scheduling optimization modelling: A case study of a Woven-Label manufacturing company. *Computer Systems Science and Engineering*, 38(2):239–248.

W. Xiang and H.P. Lee (2008). Ant colony intelligence in multi-agent dynamic manufacturing scheduling. *Engineering Applications of Artificial Intelligence*, 21(1):73–85.

G. Zhang, L. Gao, and Y. Shi (2009). An effective genetic algorithm for the flexible job-shop scheduling problem. *Expert Systems with Applications*, 38(4):3563–3573.

S.M. Johnson (1954). Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68.

J.R. Barker and G.B. McMahon (1985). Scheduling the general job-shop. *Management Science*, 31(5):594–598.

J. Adams, E. Balas, and D. Zawack (1988). The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401.

M.M. Solomon (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265.

M. Dror and G. Trudeau (1989). Savings by split delivery routing. *Transportation Science*, 23(2):141–145.

M. Haimovich and A.H.G. Rinnooy Kan (1985). Bounds and heuristics for capacitated routing problems. *Mathematics of Operations Research*, 10(4):527–542.

J. Lysgaard, A.N. Letchford, and R.W. Eglese (2004). A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming*, 100(2):423–445.

D. Pecin, A. Pessoa, M. Poggi, and E. Uchoa (2017). Improved branch-cut-and-price for capacitated vehicle routing. *Mathematical Programming Computation*, 9(1):61–86.

M. Gendreau, A. Hertz, and G. Laporte (1994). A tabu search heuristic for the vehicle routing problem. *Management Science*, 40(10):1276–1290.

H.C. Huang (2007). A hybrid tabu search approach to the cell formation problem. *Journal of the Operational Research Society*, 58(8):1089–1099.

J. Carlier (1982). The one-machine sequencing problem. *European Journal of Operational Research*, 11(1):35–42.

G.B. McMahon and M. Florian (1975). On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):475–482.

J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker (1977). Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362.

A.H.G. Rinnooy Kan (1976). *Machine Scheduling Problems: Classification, Complexity and Computations*. Nijhoff, The Hague.

J.F. Muth and G.L. Thompson, editors (1963). *Industrial Scheduling.* Prentice-Hall, Englewood Cliffs, NJ.

S. Lawrence (1984). Supplement to resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

Appendix A

Solver Log Excerpt (FJSP)

```
Gen 0: Best = 57.0, Diversity monitoring started...
Gen 2: New best = 55.0, Diversity = 0.793
Gen 3: New best = 53.0, Diversity = 0.782
Gen 4: New best = 52.0, Diversity = 0.775
Gen 6: New best = 51.0, Diversity = 0.758
Gen 10: Min=51.0, Avg=71.0, Std=12.6, Div=0.577, Mut=0.30
Gen 12: New best = 47.0, Diversity = 0.533
Gen 18: New best = 45.0, Diversity = 0.287
Gen 20: Min=45.0, Avg=56.6, Std=8.3, Div=0.251, Mut=0.30
Gen 23: New best = 44.0, Diversity = 0.205
Gen 24: New best = 43.0, Diversity = 0.156
Gen 27: New best = 42.0, Diversity = 0.135
Gen 30: New best = 41.0, Diversity = 0.113
Gen 30: Min=41.0, Avg=48.8, Std=7.2, Div=0.113, Mut=0.50
Gen 40: Min=41.0, Avg=45.7, Std=9.3, Div=0.067, Mut=0.50
Gen 50: Min=41.0, Avg=46.8, Std=11.8, Div=0.085, Mut=0.60
-> Cataclysmic mutation at gen 56!
Gen 60: Min=41.0, Avg=60.2, Std=13.3, Div=0.421, Mut=0.56
-> Low diversity (0.043), injecting random individuals...
-> Cataclysmic mutation at gen 69!
Gen 70: Min=41.0, Avg=77.4, Std=17.6, Div=0.811, Mut=0.51
...
Gen 200:
======================================
BEST MAKESPAN FOUND: 41 slots (20.5 hours)
Total runtime: 77.99 seconds
======================================
```

Appendix B

VRP Route Log

```
FLEET vs DEMAND ANALYSIS
------------------------------------------------
Virtual delivery points: 12
Total demand -> Weight: 317.0, Volume: 317.0
Trucks: 2
Fleet capacity per trip -> Weight: 110.0, Volume: 70.0
>> RESULT: MULTIPLE trips required.
Estimated minimum trips needed: 5

TRUCK CAPACITY DETAILS
------------------------------------------------
Truck  W-Cap   V-Cap   Cost/km
T1     50.0    30.0    1.00
T2     60.0    40.0    1.20

TRIP 1
Deliveries served: 3
Total distance: 79.00 km
Total cost: 90.40
Route: Depot -> C1_1 -> Depot
Load: Weight 30.0 (60.0%), Volume 30.0 (100.0%)
```