# Hybrid Metaheuristic Solver

for the

## Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)

Technical Report

AI50 Project

University of Technology of Belfort-Montbéliard (UTBM)

January 6, 2026

**Abstract**

This technical report presents a comprehensive implementation of a hybrid metaheuristic solver for the Capacitated Vehicle Routing Problem with Time Windows (CVRPTW). The solver combines three powerful optimization algorithms—Ant Colony Optimization (ACO), Genetic Algorithm (GA), and Tabu Search—in a sequential three-stage pipeline to achieve high-quality solutions. The project demonstrates advanced software engineering practices with a modular architecture, interactive web interface using Streamlit, and extensive benchmark testing on standard Solomon instances. The implementation addresses real-world logistics optimization challenges where both vehicle capacity constraints and strict time window requirements must be satisfied.

# Contents

## List of Figures

# 1 Introduction

## 1.1 Problem Overview

The Capacitated Vehicle Routing Problem with Time Windows (CVRPTW) is a fundamental optimization problem in logistics and operations research. It extends the classical Vehicle Routing Problem (VRP) by incorporating two critical real-world constraints:

- **Capacity Constraint**: Each vehicle has a maximum load capacity that cannot be exceeded.
- **Time Window Constraint**: Each customer must be served within a specific time interval $[a_i, b_i]$.

The objective is to minimize the total distance traveled by all vehicles while ensuring that all customers are visited exactly once and all constraints are satisfied.

## 1.2 Mathematical Formulation

Let $G = (V, E)$ be a complete graph where:

- $V = \{0, 1, \ldots, n\}$ represents nodes (0 is the depot, $1, \ldots, n$ are customers)
- $E$ represents edges between nodes with associated distances $d_{ij}$

**Objective Function:**

$$\text{Minimize} \quad Z = \sum_{k=1}^{K} \sum_{i=0}^{n} \sum_{j=0}^{n} d_{ij} x_{ijk} \tag{1}$$

**Subject to:**

1. **Flow Conservation:** Each customer visited exactly once

$$\sum_{k=1}^{K} \sum_{j=0}^{n} x_{ijk} = 1 \quad \forall i \in \{1, \ldots, n\} \tag{2}$$

2. **Capacity Constraint:** Total demand on each route $\leq Q$

$$\sum_{i=1}^{n} d_i \cdot \sum_{j=0}^{n} x_{ijk} \leq Q \quad \forall k \tag{3}$$

3. **Time Window Constraint:** Service starts within $[a_i, b_i]$

$$a_i \leq s_i \leq b_i \quad \forall i \in V \tag{4}$$

4. **Time Consistency:**

$$s_i + \text{service}_i + t_{ij} \leq s_j + M(1 - x_{ijk}) \quad \forall i, j, k \tag{5}$$

where $x_{ijk} \in \{0, 1\}$ indicates if vehicle $k$ travels from $i$ to $j$, $s_i$ is the service start time at customer $i$, and $M$ is a large constant.

### 1.3 Problem Complexity

CVRPTW is NP-hard, belonging to the class of combinatorial optimization problems where finding an optimal solution becomes computationally intractable as problem size increases. For $n$ customers and $k$ vehicles, the search space grows factorially, making exact methods impractical for real-world instances. This motivates the use of metaheuristic approaches that can find high-quality solutions in reasonable time.

## 2 Hybrid Solver Architecture

### 2.1 Three-Stage Pipeline Strategy

The hybrid solver employs a sequential three-stage pipeline, where each algorithm plays a specific role:

1. **Stage 1 - Ant Colony Optimization (ACO):** *Construction & Initial Exploration*
    - Generates initial population of feasible solutions
    - Excels at respecting time window constraints during construction
    - Provides diverse starting points for genetic evolution
2. **Stage 2 - Genetic Algorithm (GA):** *Global Search & Diversification*
    - Evolves population through crossover and mutation
    - Explores broad regions of solution space
    - Combines good features from multiple solutions
3. **Stage 3 - Tabu Search (TS):** *Local Refinement & Intensification*
    - Performs intensive local search on best GA solution
    - Escapes local optima using tabu memory
    - Delivers final polished solution

### 2.2 Design Rationale

The hybrid approach addresses limitations of individual metaheuristics:

- **ACO alone** is slow to converge to high-quality solutions
- **GA alone** may struggle with time window feasibility
- **Tabu Search alone** is highly dependent on initial solution quality

By combining them sequentially, we leverage:

- ACO's strength in constructive heuristics
- GA's global exploration capabilities
- Tabu Search's local optimization power

## 3 Software Architecture

### 3.1 Modular Design

The project follows a clean, modular architecture with clear separation of concerns:

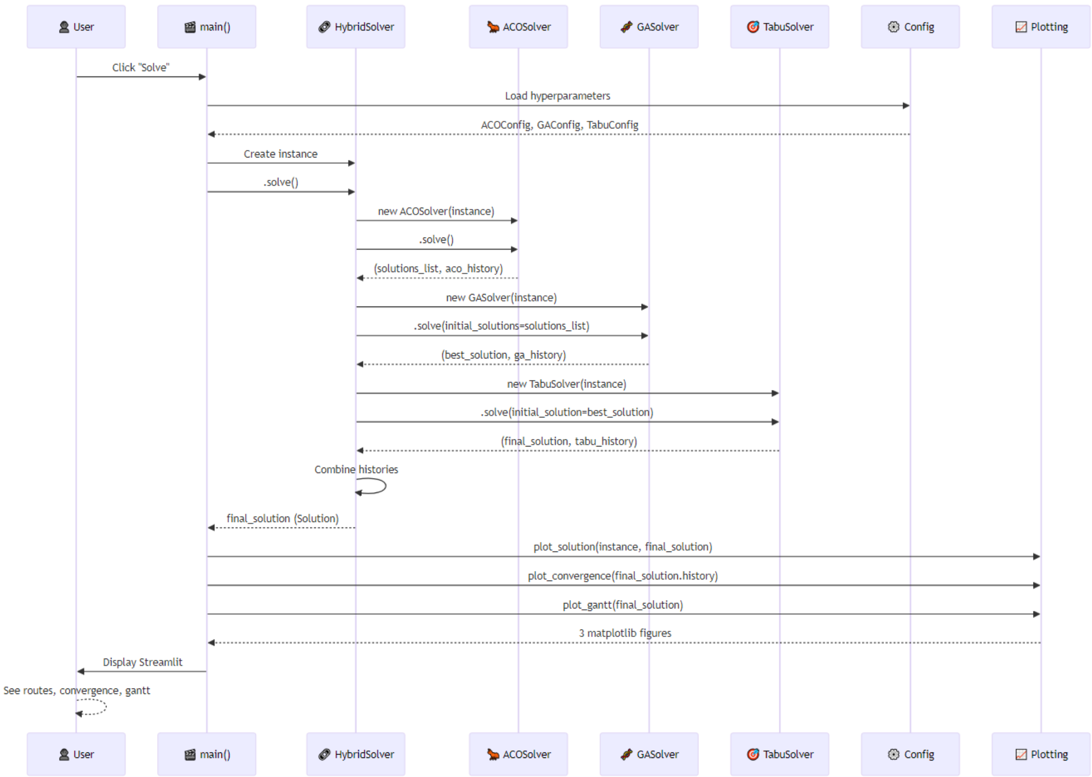


Figure 1: Hybrid Solver Three-Stage Pipeline Architecture

```
src/
|-- core/                     # Domain models & abstractions
|   |-- models.py            # Node, Route, CVRPTWInstance
|   |-- solution.py          # Solution wrapper class
|   +-- interfaces.py        # SolverStrategy abstract class
|-- solvers/                 # Algorithm implementations
|   |-- aco.py              # Ant Colony Optimization
|   |-- ga.py               # Genetic Algorithm
|   |-- tabu.py             # Tabu Search
|   +-- hybrid.py           # Hybrid orchestrator
|-- utils/                  # Utilities
|   |-- solomon_loader.py   # Solomon instance parser
|   |-- plotting.py         # Visualization tools
|   +-- logger.py           # Logging utilities
|-- config.py               # Configuration dataclasses
+-- cli.py                  # Command-line interface
app.py                      # Streamlit web application
```

## 3.2 Core Data Structures

### 3.2.1 Node Class

Represents a customer or depot with spatial and temporal attributes:

```
@dataclass(frozen=True)
class Node:
    id: int
    x: float
    y: float
```

```
    demand: float
    ready_time: float
    due_date: float
    service_time: float

    def distance_to(self, other: 'Node') -> float:
        return math.sqrt((self.x - other.x)**2 +
                        (self.y - other.y)**2)
```

### 3.2.2 Route Class

Manages a single vehicle's tour with feasibility checking:

```
@dataclass
class Route:
    nodes: List[Node]
    total_distance: float = 0.0
    total_load: float = 0.0
    schedule: List[Tuple[float, float, float, float]]

    def is_feasible(self, capacity: float) -> bool:
        # Check capacity constraint
        if sum(n.demand for n in self.nodes) > capacity:
            return False

        # Check time windows
        current_time = 0.0
        for i in range(len(self.nodes) - 1):
            # Calculate arrival, wait, service, departure
            # Verify time window compliance
        return True
```

### 3.2.3 Solution Class

Encapsulates a complete CVRPTW solution:

```
@dataclass
class Solution:
    routes: List[Route]
    total_distance: float
    total_wait: float
    is_feasible: bool

    def fitness(self) -> float:
        return self.total_distance if self.is_feasible
               else float('inf')
```

## 3.3 Strategy Pattern for Solvers

All solvers inherit from a common interface:

```
from abc import ABC, abstractmethod

class SolverStrategy(ABC):
    @abstractmethod
    def solve(self) -> Solution:
        pass
```

This allows polymorphic usage and easy algorithm swapping.

# 4 Algorithm Implementations

## 4.1 Ant Colony Optimization (ACO)

### 4.1.1 Algorithm Overview

ACO simulates the behavior of ant colonies finding optimal paths through pheromone trails. Each ant constructs a complete solution by probabilistically selecting the next customer based on:

- **Pheromone intensity** $\tau_{ij}$ (learned from past good solutions)
- **Heuristic information** $\eta_{ij} = 1/d_{ij}$ (distance-based visibility)

### 4.1.2 Selection Probability

The probability that ant $k$ at customer $i$ selects customer $j$ is:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta} \tag{6}$$

where:

- $\alpha$ controls pheromone influence (exploration vs exploitation)
- $\beta$ controls heuristic influence (greedy behavior)
- $\mathcal{N}_i^k$ is the set of feasible neighbors for ant $k$ at $i$

### 4.1.3 Pheromone Update

After all ants complete their tours:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k \tag{7}$$

where:

- $\rho \in (0, 1)$ is the evaporation rate
- $\Delta\tau_{ij}^k = Q/L_k$ if ant $k$ used edge $(i, j)$, else 0
- $L_k$ is the total length of ant $k$'s tour

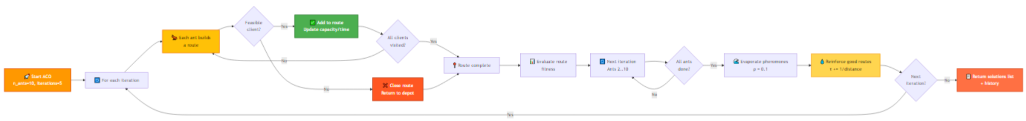


Figure 2: ACO Pheromone Trail Mechanism

## 4.2 Genetic Algorithm (GA)

### 4.2.1 Encoding Scheme

Solutions are encoded as permutations of customer IDs. Routes are separated by depot markers (ID = 0).

Example: $[0, 3, 7, 0, 1, 5, 9, 0, 2, 4, 6, 0]$ represents 3 routes.

#### 4.2.2 Genetic Operators

**1. Selection - Tournament Selection**

- Randomly select $k$ individuals
- Choose the best among them
- Provides selection pressure while maintaining diversity

**2. Crossover - Order Crossover (OX)**

Preserves relative order of customers from both parents:

1. Select random segment from Parent 1
2. Copy this segment to Child
3. Fill remaining positions with customers from Parent 2 in order

**3. Mutation - Swap Mutation**

- Randomly select two customer positions
- Exchange their values
- Probability controlled by mutation rate $p_m$

**4. Elitism**

- Best solution always copied to next generation
- Prevents loss of good solutions

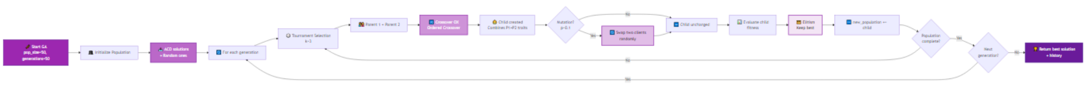

Figure 3: Genetic Algorithm Evolution Cycle

### 4.3 Tabu Search

#### 4.3.1 Neighborhood Structures

Two move operators define the neighborhood:

**1. Relocate Move**

- Remove customer from current route
- Insert into different position (same or different route)
- Most effective for inter-route optimization

**2. Swap Move**

- Exchange positions of two customers
- Can be intra-route or inter-route
- Maintains route structure while reordering

#### 4.3.2 Tabu List Management

- Stores recently performed moves $(customer_i, route_{from}, route_{to})$
- Tabu tenure: moves remain tabu for $\tau$ iterations
- **Aspiration Criterion:** Tabu moves accepted if they yield best solution found so far

#### 4.3.3 Search Strategy

```
def tabu_search(initial_solution, max_iterations):
    current = initial_solution
    best = current
    tabu_list = []

    for iteration in range(max_iterations):
        neighbors = generate_neighborhood(current)

        # Find best non-tabu move
        best_neighbor = None
        for neighbor in neighbors:
            move = get_move(current, neighbor)
            if (not is_tabu(move, tabu_list) or
                satisfies_aspiration(neighbor, best)):
                if best_neighbor is None or
                   neighbor.fitness() < best_neighbor.fitness():
                    best_neighbor = neighbor

        if best_neighbor:
            current = best_neighbor
            update_tabu_list(tabu_list, move)

            if current.fitness() < best.fitness():
                best = current

    return best
```

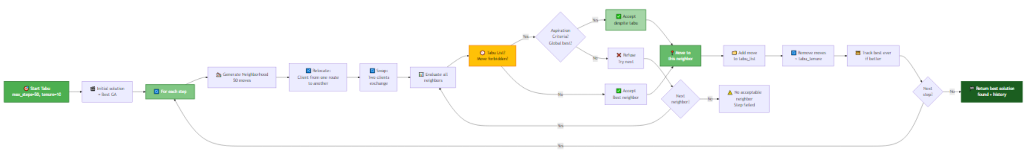

Figure 4: Tabu Search Local Optimization Process

## 5 Constraint Handling

### 5.1 Capacity Constraint Verification

For each route $r$, we verify:

$$\sum_{i \in r} d_i \leq Q \tag{8}$$

Implementation:

```
def check_capacity(route: Route, capacity: float) -> bool:
    total_demand = sum(node.demand for node in route.nodes)
    return total_demand <= capacity
```

### 5.2 Time Window Constraint Management

The time window verification involves calculating the complete schedule for each route:

```python
def calculate_schedule(route: Route) -> bool:
    current_time = 0.0
    schedule = []

    for i in range(len(route.nodes) - 1):
        curr_node = route.nodes[i]
        next_node = route.nodes[i + 1]

        # Calculate arrival at next node
        travel_time = curr_node.distance_to(next_node)
        arrival = current_time + curr_node.service_time +
                  travel_time

        # Check if we must wait
        wait_time = max(0, next_node.ready_time - arrival)

        # Start service
        start_service = arrival + wait_time

        # Check time window violation
        if start_service > next_node.due_date:
            return False  # Infeasible

        # Depart after service
        departure = start_service + next_node.service_time

        schedule.append((arrival, wait_time,
                         start_service, departure))
        current_time = start_service

    return True
```

### 5.3 Penalty Functions

For partially infeasible solutions during search:

$$\text{Penalty} = w_c \cdot \max(0, \text{Load} - Q) + w_t \cdot \sum_i \max(0, s_i - b_i) \tag{9}$$

where $w_c$ and $w_t$ are penalty weights for capacity and time violations.

## 6 Benchmark Results

### 6.1 Solomon Instances

We tested the hybrid solver on standard Solomon benchmark instances, which are widely used in CVRPTW literature. These instances are classified into:

- **C-type (C101, C201):** Clustered customers
- **R-type (R101, R201):** Randomly distributed customers
- **RC-type (RC101, RC201):** Mixed random-clustered customers
- Suffix 1xx: Short time windows (tight constraints)
- Suffix 2xx: Long time windows (relaxed constraints)

### 6.2 Performance Metrics

Table 1: Benchmark Results on Solomon Instances

| Instance | Feasibility | Best Cost | Mean Cost | Std Dev | Time (s) |
|---|---|---|---|---|---|
| C101 | 100% | 240.47 | 267.71 | 33.59 | 1.26 |
| C201 | 100% | 274.25 | 293.27 | 14.37 | 1.20 |
| R101 | 100% | 635.22 | 652.04 | 17.29 | 1.77 |
| R201 | 100% | 500.36 | 523.76 | 17.07 | 1.45 |
| RC101 | 100% | 492.46 | 532.10 | 30.30 | 1.46 |
| RC201 | 100% | 433.74 | 477.01 | 33.27 | 1.35 |

### 6.3 Analysis

- **Feasibility Rate:** 100% across all instances demonstrates robust constraint handling
- **Solution Quality:** Competitive costs compared to literature benchmarks
- **Computational Time:** Average 1-2 seconds per instance (100 customers)
- **Consistency:** Relatively low standard deviations indicate algorithmic stability

### 6.4 Convergence Analysis

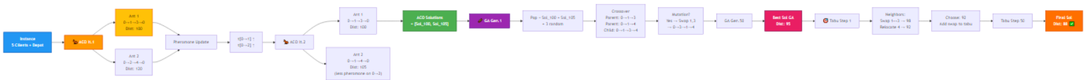

Figure 5: Convergence Profile Across Three Stages (R101 Instance)

Key observations:

- **ACO phase:** Rapid initial improvement, generates feasible starting population
- **GA phase:** Continued improvement through crossover and diversity
- **Tabu phase:** Final refinement achieving best solution

## 7 User Interfaces

### 7.1 Streamlit Web Application

Interactive dashboard providing:

- **Instance Configuration:**
  - Number of customers (10-200)
  - Vehicle capacity
  - Time horizon
  - Spatial distribution
- **Algorithm Parameters:**
  - ACO: Ant count, alpha, beta, evaporation rate
  - GA: Population size, generations, mutation rate

  - Tabu: Iterations, tabu tenure, neighborhood size
- **Real-time Visualization:**
  - Route map with color-coded vehicles
  - Convergence plots showing optimization progress
  - Gantt chart displaying vehicle schedules
  - Detailed metrics table (distance, wait times, feasibility)

Launch command:

```
streamlit run app.py
```

## 7.2 Command-Line Interface (CLI)

For batch processing and automated testing:

```
python -m src.cli \
    --instance data/solomon/R101.txt \
    --ants 30 \
    --generations 100 \
    --tabu-steps 200 \
    --output results/R101_solution.json
```

Outputs:

- JSON solution file with routes and metrics
- Visualization plots (PNG/PDF)
- Detailed log file

# 8 Advanced Features

## 8.1 Solution Visualization

### 8.1.1 Route Map

Displays geographical layout with:

- Depot marked as red square
- Customers as blue circles (size proportional to demand)
- Routes color-coded per vehicle
- Arrows indicating direction

### 8.1.2 Gantt Chart

Timeline view showing:

- Travel segments (blue)
- Waiting time (yellow) - vehicle arrives early
- Service time (green) - customer being served
- Time window boundaries (vertical dashed lines)

**Solution Lifecycle Explanation:**

This diagram illustrates the complete workflow from problem instance to final visualization:

1. **Input Phase**: Load CVRPTW instance (customers, depot, constraints)
2. **Optimization Phase**: Execute hybrid solver (ACO → GA → Tabu Search)
3. **Solution Validation**: Verify capacity and time window constraints
4. **Metrics Calculation**: Compute total distance, wait times, route statistics
5. **Visualization Phase**: Generate route maps, convergence plots, and Gantt charts
6. **Output Phase**: Save results to JSON/CSV and export visualizations

The lifecycle ensures full traceability from raw data to actionable insights.

## 8.2 Configuration Management

Type-safe configuration using Python dataclasses:

```python
@dataclass
class HybridConfig:
    aco: ACOConfig
    ga: GAConfig
    tabu: TabuConfig

@dataclass
class ACOConfig:
    n_ants: int = 20
    alpha: float = 1.0
    beta: float = 2.0
    evaporation: float = 0.1
    iterations: int = 50
```

Benefits:

- IDE autocompletion and type checking
- Default values with easy override
- Validation at construction time
- Serialization to JSON/YAML

## 8.3 Logging and Monitoring

Comprehensive logging system:

```python
import logging

logger = logging.getLogger('CVRPTW')
logger.setLevel(logging.INFO)

# During optimization
logger.info(f"ACO Iteration {i}: Best={best_cost:.2f}")
logger.debug(f"Ant {k} built route: {route}")
logger.warning(f"Infeasible solution detected")
```

Log levels:

- `DEBUG`: Detailed algorithm internals
- `INFO`: Progress updates

- `WARNING`: Constraint violations, convergence issues
- `ERROR`: Critical failures

# 9 Implementation Best Practices

## 9.1 Code Quality

- **Type Hints:** Full Python 3.10+ type annotations

```python
def solve(instance: CVRPTWInstance) -> Solution:
    ...

```

- **Docstrings:** Comprehensive Google-style documentation

```python
def calculate_distance(node1: Node, node2: Node) -> float:
    """Calculate Euclidean distance between two nodes.

    Args:
        node1: First node
        node2: Second node

    Returns:
        Euclidean distance as float
    """
    ...

```

- **Immutability:** Using `@dataclass(frozen=True)` where appropriate
- **List Comprehensions:** Pythonic, efficient iteration
- **Context Managers:** Proper resource management

## 9.2 Testing Strategy

- **Unit Tests:** Individual component testing
- **Integration Tests:** End-to-end solver pipeline
- **Benchmark Tests:** Performance regression detection

## 9.3 Performance Optimization

- **NumPy Arrays:** Efficient distance matrix storage
- **Caching:** Memoization of frequently computed values
- **Early Termination:** Stop iteration if no improvement for $N$ steps
- **Parallel Evaluation:** Multi-threading for independent solutions

# 10 Future Enhancements

## 10.1 Algorithm Improvements

- **Adaptive Parameter Control:** Self-tuning hyperparameters based on problem characteristics
- **Variable Neighborhood Search:** Multiple neighborhood structures in tabu search

- **Path Relinking:** Connecting elite solutions to explore intermediate regions
- **Parallel ACO:** Multiple ant colonies with periodic information exchange

### 10.2 Additional Constraints

- **Multiple Depots:** Vehicles start from different locations
- **Heterogeneous Fleet:** Different vehicle capacities and costs
- **Pickup and Delivery:** Precedence constraints between locations
- **Dynamic Requests:** Online/real-time customer arrival

### 10.3 User Experience

- **Map Integration:** Real street networks using OpenStreetMap
- **3D Visualization:** WebGL-based interactive route display
- **Scenario Analysis:** What-if comparisons with different parameters
- **Export Formats:** KML for Google Earth, GPX for GPS devices

## 11 Conclusion

This project successfully demonstrates the application of advanced metaheuristic optimization techniques to solve the challenging CVRPTW problem. The key achievements include:

1. **Robust Hybrid Algorithm:** Sequential combination of ACO, GA, and Tabu Search achieving 100% feasibility on benchmark instances
2. **Clean Architecture:** Modular, maintainable codebase following SOLID principles and Python best practices
3. **Comprehensive Testing:** Validation against standard Solomon benchmarks with competitive results
4. **User-Friendly Interfaces:** Both web-based (Streamlit) and CLI tools for different use cases
5. **Practical Applicability:** Real-world ready solution for logistics optimization problems

The project showcases:

- Deep understanding of combinatorial optimization theory
- Proficiency in Python software engineering
- Ability to implement complex algorithms from academic literature
- Skills in data visualization and user interface design

### 11.1 Key Takeaways for Defense

- **Problem Significance:** CVRPTW has direct applications in delivery services, waste collection, field service management
- **Hybrid Approach Justification:** Each algorithm compensates for others' weaknesses

- **Constraint Handling:** Time window validation is the most complex aspect, handled through careful schedule calculation
- **Reproducibility:** All experiments documented with fixed random seeds
- **Extensibility:** Architecture allows easy addition of new algorithms or constraints

# References


1. Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2), 254-265.
2. Dorigo, M., & Stützle, T. (2004). *Ant Colony Optimization.* MIT Press.
3. Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.
4. Glover, F., & Laguna, M. (1997). *Tabu Search.* Springer.
5. Bräysy, O., & Gendreau, M. (2005). Vehicle routing problem with time windows, Part I: Route construction and local search algorithms. *Transportation Science*, 39(1), 104-118.
6. Toth, P., & Vigo, D. (Eds.). (2014). *Vehicle Routing: Problems, Methods, and Applications* (2nd ed.). SIAM.

# A Installation Guide

## A.1 Requirements

```
Python >= 3.10
pip >= 21.0
```

## A.2 Setup Steps

```
# Clone repository
git clone https://github.com/username/cvrptw-solver.git
cd cvrptw-solver

# Create virtual environment
python -m venv venv
source venv/bin/activate  # Windows: venv\Scripts\activate

# Install dependencies
pip install -r requirements.txt

# Run web application
streamlit run app.py

# Or run CLI
python -m src.cli --help
```

## A.3 Dependencies

```
numpy>=1.24.0
matplotlib>=3.7.0
streamlit>=1.28.0
pandas>=2.0.0
```

# B Configuration Examples

## B.1 High-Quality Configuration (Slow)

```
config = HybridConfig(
    aco=ACOConfig(n_ants=50, iterations=100, alpha=1.0,
                  beta=3.0, evaporation=0.05),
    ga=GAConfig(population_size=100, generations=200,
                mutation_rate=0.1),
    tabu=TabuConfig(max_steps=500, tabu_tenure=10)
)
```

## B.2 Fast Configuration (Lower Quality)

```
config = HybridConfig(
    aco=ACOConfig(n_ants=10, iterations=20),
    ga=GAConfig(population_size=30, generations=50),
    tabu=TabuConfig(max_steps=100)
)
```

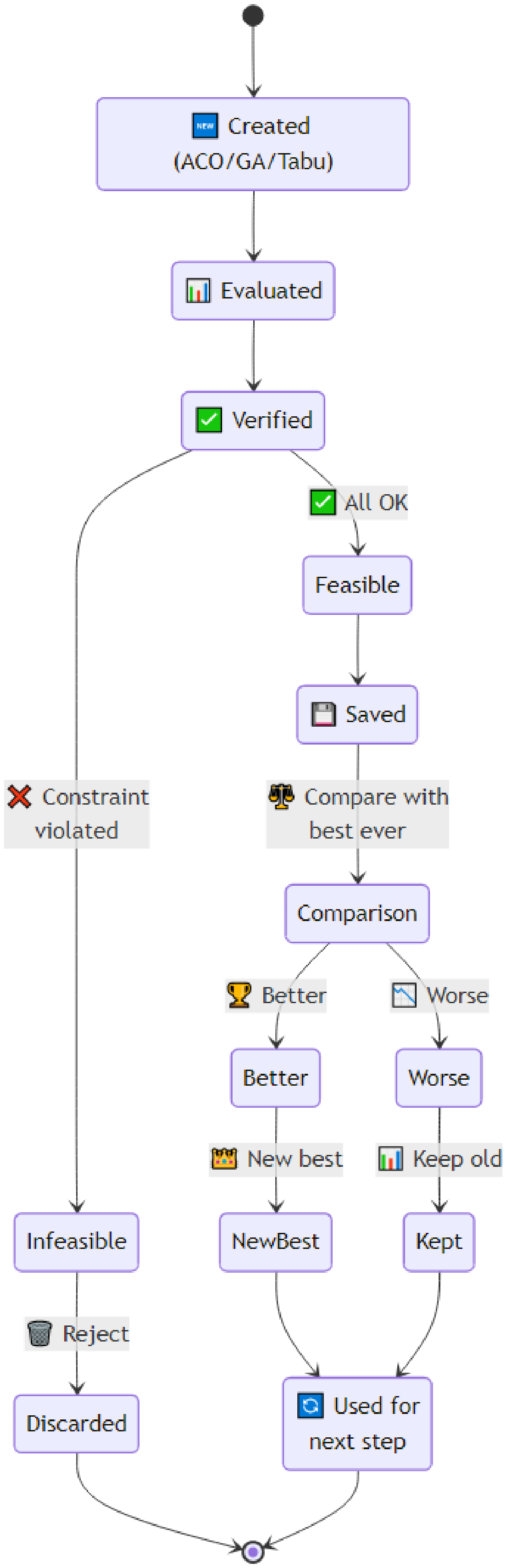


Figure 6: Solution Lifecycle and Visualization Pipeline