成品油配送问题两阶段启发式算法求解报告

1. 问题分析

1.1 问题背景

本实验研究的是一个复杂的成品油配送问题，涉及2个油库（A、B）和25个加油站的配送优化。问题的核心是在满足各项约束条件下，为3种油品（92号汽油、95号汽油、0号柴油）制定最优配送计划，以最小化总配送成本。

1.2 问题特征分析

1.2.1 网络结构特征

- **节点类型**：2个油库节点 + 25个加油站节点
- **网络性质**：非完全连通图（部分加油站之间不可达）
- **距离特征**：库站距离和站站距离已知，单位为千米

1.2.2 需求特征

- **需求不确定性**：每个加油站对每种油品的需求量以区间形式给出（下限、最可能值、上限）
- **需求可调性**：在区间范围内可根据运输能力适当调整配送量
- **时间敏感性**：需求随时间消耗，存在时间窗约束

1.2.3 车辆特征

- **车队规模**：油库A拥有40辆油罐车，油库B无配送车辆
- **车辆结构**：每辆车有两个容积相等的储油仓，隔板分离
- **载重约束**：每个仓对应一种油品，不能混装
- **行程限制**：每辆车每天最多配送两趟

1.2.4 时间约束

- **工作时间**：8:00-17:00（9小时）
- **时间窗**：每个配送任务有不容纳时间和断油时间限制
- **同时服务**：每个加油站同一时间只能由一辆车服务

1.3 问题复杂性分析

1.3.1 NP-难题特征

本问题属于车辆路径问题（VRP）的扩展，具有以下复杂性：

- **多约束**：容量、时间窗、同时服务等多重约束
- **多目标**：成本最小化与服务质量平衡
- **多阶段**：需求确定、车辆分配、路径规划的多阶段决策

1.3.2 求解挑战

- **约束耦合**：各约束间存在强耦合关系
- **解空间巨大**：40辆车×25个站点×3种油品的组合爆炸
- **实时性要求**：需要在合理时间内给出高质量解

2. 问题优化建模

2.1 决策变量定义

2.1.1 基本决策变量

- x_{ijk}^t：二进制变量，车辆i在第t趟是否从节点j直接行驶到节点k
- y_{ips}^t：连续变量，车辆i在第t趟为加油站p配送油品s的数量
- z_{ip}^t：二进制变量，车辆i在第t趟是否访问加油站p
- T_{ip}^t：连续变量，车辆i在第t趟到达加油站p的时间

2.1.2 辅助变量

- S_i^t：车辆i第t趟的出发时间
- E_i^t：车辆i第t趟的结束时间
- D_{ip}^t：车辆i在第t趟在加油站p的停留时间

2.2 目标函数

$\min Z = \sum_{i=1}^{|V|} \sum_{t=1}^{2} \sum_{j \in N} \sum_{k \in N} c_{jk} \cdot d_{jk} \cdot x_{ijk}^t$

其中：

- $|V|$：车辆总数（40辆）
- c_{jk}：车辆从节点j到节点k的单位距离成本
- d_{jk}：节点j到节点k的距离
- N：所有节点集合（油库+加油站）

2.3 约束条件

2.3.1 流平衡约束

$\sum_{j \in N} x_{ijk}^t = \sum_{j \in N} x_{ikj}^t = z_{ik}^t, \quad \forall i, k, t$

2.3.2 车辆容量约束

$\sum_{p \in P} \sum_{s \in S} y_{ips}^t \leq Q_i, \quad \forall i, t$

2.3.3 "一仓一罐"约束

$\sum_{p\in P}\sum_{s\in S}\delta_{ips}^{t}\leq 2,\quad \forall i,t$

其中 $\delta_{ips}^{t}=1$ 当且仅当 $y_{ips}^{t}>0$

2.3.4 需求量区间约束

$L_{ps}\leq\sum_{i=1}^{|V|}\sum_{t=1}^{2}y_{ips}^{t}\leq U_{ps},\quad \forall p,s$

其中 L_{ps} 和 U_{ps} 分别为加油站p对油品s的需求下限和上限。

2.3.5 时间窗约束

$ET_{ps}\leq T_{ip}^{t}\leq LT_{ps},\quad \forall i,p,s,t \text{ if } y_{ips}^{t}>0$

其中：

- ET_{ps}：不容纳时间（Earliest Time）
- LT_{ps}：断油时间（Latest Time）

2.3.6 工作时间约束

$E_{i}^{2}-S_{i}^{1}\leq H_{max},\quad \forall i$

其中 $H_{max}=9$ 小时

2.3.7 同时服务约束

$\sum_{i=1}^{|V|}\sum_{t=1}^{2}\alpha_{ipt}^{t'}\leq 1,\quad \forall p,t'$

其中 $\alpha_{ipt}^{t'}$ 表示车辆i在时刻t'是否正在为加油站p服务。

2.3.8 最终返回约束

$\sum_{j\in N}x_{ij1}^{2}=1,\quad \forall i$ 使用车辆 i

其中节点1代表油库A，确保所有车辆最终返回油库A。

2.4 时间窗计算模型

2.4.1 库存消耗模型

设加油站p对油品s的初始库存为 I_{ps}^{0}，日消耗率为 r_{ps}，则：

不容纳时间（ET）：
$ET_{ps}=\max\left(0,\frac{y_{ps}-(C_{ps}-I_{ps}^{0})}{r_{ps}^{min}}\right)$

断油时间（LT）：
$LT_{ps}=\min\left(H_{max},\frac{I_{ps}^{0}-SS_{ps}}{r_{ps}^{max}}\right)$

其中：

- C_{ps}：油罐容量
- SS_{ps}：安全库存

- $r_{ps}^{min}, r_{ps}^{max}$：最小和最大消耗率

3. 算法设计

3.1 总体框架：两阶段启发式算法

考虑到问题的NP-难性质和实际应用需求，设计了一个两阶段启发式算法：

```
阶段一：库存控制与任务生成
├── 输入：需求区间、库存状态、消耗率
├── 输出：具体配送任务列表、时间窗
└── 方法：基于库存理论的确定性任务生成

阶段二：路径优化
├── 输入：配送任务列表、时间窗
├── 输出：车辆路径方案
└── 方法：基于遗传算法的启发式优化
```

3.2 阶段一：库存控制与任务生成算法

3.2.1 算法思想

基于库存管理理论，将不确定的需求区间转化为确定的配送任务。

3.2.2 详细步骤

```
算法1：任务生成算法
输入：station_inventory, demands, depot_inventory
输出：delivery_tasks, time_windows

1. 初始化 tasks = [], depot_allocations = {}
2. 设置库存管理参数：
   safety_stock_hours = 4.0
   planning_horizon_hours = 24.0
   target_stock_level_ratio = 0.9

3. FOR 每个加油站 p IN stations:
4.     FOR 每种油品 s IN products:
5.         FOR 每个油罐 tank IN tanks[p][s]:
6.             获取需求区间 [min_demand, most_likely, max_demand]
7.             计算消耗率区间 [min_rate, avg_rate, max_rate]
8.
9.             计算补货决策点：
10.                safety_stock = max_rate * safety_stock_hours
11.                reorder_point = safety_stock + avg_rate * planning_horizon_hours
12.
13.            IF current_inventory < reorder_point:
14.                计算配送量：
15.                    target_level = tank_capacity * target_stock_level_ratio
16.                    base_quantity = target_level - current_inventory
17.
18.                调整配送量以满足油库库存约束
```

```
19.
20.             计算时间窗：
21.                 ET = 计算不容纳时间
22.                 LT = 计算断油时间
23.
24.             创建配送任务并添加到tasks
25.             更新depot_allocations
26.
27. RETURN tasks, time_windows
```

3.2.3 时间窗计算细节

不容纳时间计算：

```
def calculate_earliest_time(quantity, current_inventory, tank_capacity,
min_consumption_rate):
    space_needed = quantity - (tank_capacity - current_inventory)
    if space_needed > 0:
        return space_needed / min_consumption_rate
    return 0
```

断油时间计算：

```
def calculate_latest_time(current_inventory, safety_stock, max_consumption_rate):
    if current_inventory > safety_stock:
        return (current_inventory - safety_stock) / max_consumption_rate
    return 0
```

3.3 阶段二：基于遗传算法的路径优化

3.3.1 遗传算法框架

```
算法2：遗传算法主框架
输入：delivery_tasks, time_windows
输出：best_solution

1. 初始化参数：
   population_size = 80
   generations = 300
   crossover_rate = 0.8
   mutation_rate = 0.3

2. 初始化种群 population = initialize_population()

3. FOR generation = 1 TO generations:
4.    FOR individual IN population:
5.       individual.fitness = calculate_fitness(individual)
6.
7.    排序并选择精英：elites = top_k(population, elite_size)
8.
9.    生成新种群：
```

```
        new_population = elites
        WHILE len(new_population) < population_size:
            parent1, parent2 = selection(population)
            child1, child2 = crossover(parent1, parent2, crossover_rate)
            mutate(child1, mutation_rate)
            mutate(child2, mutation_rate)
            new_population.extend([child1, child2])

    population = new_population

RETURN best_individual.solution
```

3.3.2 染色体编码设计

采用双染色体编码方案：

车辆染色体：

- 表示：$[v_1, v_2, \ldots, v_{40}]$
- 含义：车辆的优先级排列
- 长度：40（可用车辆数）

任务染色体：

- 表示：$[t_1, t_2, \ldots, t_n]$
- 含义：任务的分配顺序
- 长度：n（总任务数）

3.3.3 解码算法

```
算法3：构造式解码算法
输入：individual（包含车辆染色体和任务染色体）
输出：solution（路径集合）

初始化：
    routes = []
    assigned_tasks = [False] * num_tasks
    vehicle_available_time = {v_id: 8.0 for v_id in vehicles}
    station_schedules = {}

FOR vehicle_id IN individual.vehicle_chromosome:
    FOR trip_number IN [1, 2]:
        current_tasks = []
        start_time = vehicle_available_time[vehicle_id]

        确定起点油库：
            IF trip_number == 1: start_depot = 油库A
            ELSE: start_depot = 上一趟的终点

        FOR task_idx IN individual.task_chromosome:
            IF NOT assigned_tasks[task_idx]:
                temp_route = create_route(vehicle_id, current_tasks + [task_idx])
```

```
            IF 满足容量约束 AND 满足时间约束:
                current_tasks.append(task_idx)
                assigned_tasks[task_idx] = True

        IF current_tasks NOT empty:
            确定终点油库:
                IF trip_number == max_trips: end_depot = 油库A
                ELSE: end_depot = 就近油库选择

            route = create_final_route(vehicle_id, trip_number,
                                       start_depot, end_depot, current_tasks)

            优化路径:
                route = local_search_2opt(route)
                route = optimize_loading(route)
                route = adaptive_quantity_adjustment(route)

            routes.append(route)
            更新车辆可用时间和站点日程表

    RETURN Solution(routes)
```

3.3.4 局部搜索优化

2-opt改进:

```
算法4：2-opt局部搜索
输入：route
输出：improved_route

提取站点访问序列 stations = extract_stations(route)
best_route = route
improved = True

WHILE improved:
    improved = False
    FOR i = 1 TO len(stations)-1:
        FOR j = i+1 TO len(stations):
            new_sequence = reverse_segment(stations, i, j)
            new_route = reconstruct_route(new_sequence)

            IF 满足约束 AND cost(new_route) < cost(best_route):
                best_route = new_route
                stations = new_sequence
                improved = True
                BREAK
        IF improved: BREAK

RETURN best_route
```

3.3.5 适应度函数设计

```
def calculate_fitness(individual):
    solution = decode(individual)

    # 计算未完成任务数
    unserved_tasks = total_tasks - tasks_in_solution

    # 主要惩罚：未完成任务
    unserved_penalty = unserved_tasks * 1_000_000

    # 次要惩罚：约束违反
    violation_penalty = 0 if solution.feasible else 500_000

    total_penalty = unserved_penalty + violation_penalty

    return 1.0 / (1.0 + solution.cost + total_penalty)
```

3.4 算法创新点

3.4.1 动态时间窗管理

- 基于库存消耗率的精确时间窗计算
- 考虑需求区间不确定性的鲁棒性设计

3.4.2 多约束集成处理

- 同时处理容量、时间窗、同时服务等多重约束
- 在解码过程中动态维护全局约束一致性

3.4.3 自适应配送量调整

- 根据车辆装载率和时间利用率动态调整配送量
- 在需求区间范围内优化资源利用效率

4. 实现细节

4.1 系统架构

```
成品油配送求解系统
├── data_loader.py          # 数据加载模块
├── problem_model.py        # 问题建模模块
├── solver.py               # 求解器模块
├── result_analyzer.py      # 结果分析模块
└── main.py                 # 主程序入口
```

4.2 核心类设计

4.2.1 数据结构定义

```
@dataclass
class DeliveryTask:
    """配送任务"""
    station_id: int          # 加油站ID
    product_id: int          # 油品ID
    quantity: float          # 配送量
    tank_id: Optional[int]   # 油罐ID

@dataclass
class Route:
    """车辆路径"""
    vehicle_id: int          # 车辆ID
    trip_number: int         # 趟次号(1或2)
    start_depot: int         # 起点油库
    end_depot: int           # 终点油库
    tasks: List[DeliveryTask] # 配送任务列表
    start_time: float        # 出发时间

@dataclass
class Solution:
    """完整解决方案"""
    routes: List[Route]      # 所有路径
    total_cost: float        # 总成本
    is_feasible: bool        # 可行性
    violations: List[str]    # 约束违反列表
```

4.2.2 约束检查实现

```
class ProblemModel:
    def check_capacity_constraint(self, route: Route) -> Tuple[bool, str]:
        """检查容量约束"""
        # 按油品分组统计配送量
        product_quantities = {}
        for task in route.tasks:
            product_quantities[task.product_id] = \
                product_quantities.get(task.product_id, 0) + task.quantity

        # 检查油品种类限制(最多2种)
        if len(product_quantities) > 2:
            return False, "超过2种油品限制"

        # 检查车仓容量分配
        compartment1_cap = self.vehicles[route.vehicle_id]['compartment1_capacity']
        compartment2_cap = self.vehicles[route.vehicle_id]['compartment2_capacity']

        quantities = list(product_quantities.values())
        if len(quantities) == 2:
            q1, q2 = quantities
```

```
            if not ((q1 <= compartment1_cap and q2 <= compartment2_cap) or
                    (q1 <= compartment2_cap and q2 <= compartment1_cap)):
                return False, "无法满足车仓容量分配"

        return True, ""

    def check_time_constraint(self, route: Route, time_windows: Dict,
                              existing_schedules: Dict = None) -> Tuple[bool, str]:
        """检查时间约束"""
        route_time = self.calculate_route_time(route)
        if route_time > self.working_hours:
            return False, f"路径时间{route_time:.2f}h超过工作时间限制"

        # 逐站点检查时间窗和冲突
        current_time = route.start_time
        current_location = route.start_depot

        for task in route.tasks:
            # 计算到达时间
            distance = self.get_distance(current_location, task.station_id)
            travel_time = distance / self.vehicles[route.vehicle_id]['speed']
            arrival_time = current_time + travel_time

            # 检查时间窗
            task_key = (task.station_id, task.product_id, task.tank_id)
            tw = time_windows[task_key]
            service_start = max(arrival_time, tw['earliest'])

            if service_start > tw['latest']:
                return False, f"超过时间窗限制"

            # 检查与现有服务的冲突
            if existing_schedules:
                service_end = service_start + self.stations[task.station_id]['unload_time']
                if self._has_schedule_conflict(task.station_id, service_start,
                                               service_end, existing_schedules):
                    return False, "服务时间冲突"

            current_time = service_end
            current_location = task.station_id

        return True, ""
```

4.3 关键算法实现

4.3.1 贪心初始化

```
def _create_greedy_individual(self) -> GAIndividual:
    """贪心构造高质量初始解"""
    solution = Solution(routes=[])
    assigned_tasks = set()
```

```
    # 按任务优先级排序(时间窗紧急度)
    task_priorities = []
    for i, req in enumerate(self.requirements):
        urgency = req.latest_time - req.earliest_time  # 时间窗宽度
        task_priorities.append((urgency, i))
    task_priorities.sort()  # 优先处理时间窗紧的任务

    for _, task_idx in task_priorities:
        if task_idx in assigned_tasks:
            continue

        req = self.requirements[task_idx]
        task = DeliveryTask(req.station_id, req.product_id, req.quantity, req.tank_id)

        best_insertion = None
        min_cost_increase = float('inf')

        # 尝试插入现有路径
        for route_idx, route in enumerate(solution.routes):
            for pos in range(len(route.tasks) + 1):
                new_tasks = route.tasks[:pos] + [task] + route.tasks[pos:]
                temp_route = Route(route.vehicle_id, route.trip_number,
                                   route.start_depot, route.end_depot, new_tasks)

                if self._is_route_feasible(temp_route):
                    cost_increase = (self.model.calculate_route_cost(temp_route) -
                                     self.model.calculate_route_cost(route))
                    if cost_increase < min_cost_increase:
                        min_cost_increase = cost_increase
                        best_insertion = (route_idx, pos, temp_route)

        # 如果无法插入，创建新路径
        if best_insertion is None:
            new_route = self._create_new_route_for_task(task)
            if new_route:
                solution.routes.append(new_route)
                assigned_tasks.add(task_idx)
        else:
            route_idx, pos, new_route = best_insertion
            solution.routes[route_idx] = new_route
            assigned_tasks.add(task_idx)

    return self._solution_to_individual(solution)
```

4.3.2 自适应配送量优化

```
def _adaptive_quantity_adjustment(self, route: Route, time_windows: Dict) -> Route:
    """自适应调整配送量"""
    route_time = self.model.calculate_route_time(route)

    if route_time > self.model.working_hours:
        # 时间超限，减少配送量
```

```
            return self._reduce_quantities_for_time(route, time_windows)
        elif route_time < self.model.working_hours * 0.8:
            # 时间充裕，增加配送量提高效率
            return self._increase_quantities_for_efficiency(route, time_windows)
        else:
            return route

    def _increase_quantities_for_efficiency(self, route: Route, time_windows: Dict) -> Route:
        """在时间允许的情况下增加配送量"""
        vehicle = self.data.vehicles[route.vehicle_id]
        total_capacity = vehicle['total_capacity']
        current_load = sum(task.quantity for task in route.tasks)
        available_capacity = total_capacity - current_load

        if available_capacity <= 0:
            return route

        adjusted_tasks = []
        remaining_capacity = available_capacity

        for task in route.tasks:
            task_key = (task.station_id, task.product_id, task.tank_id)
            if task_key in time_windows:
                tw_info = time_windows[task_key]
                max_qty = tw_info.get('max_quantity', task.quantity)

                # 计算可增加的量
                possible_increase = min(max_qty - task.quantity, remaining_capacity)

                if possible_increase > 0:
                    new_quantity = task.quantity + possible_increase
                    remaining_capacity -= possible_increase

                    adjusted_task = DeliveryTask(
                        task.station_id, task.product_id,
                        new_quantity, task.tank_id
                    )
                    adjusted_tasks.append(adjusted_task)
                else:
                    adjusted_tasks.append(task)
            else:
                adjusted_tasks.append(task)

        return Route(route.vehicle_id, route.trip_number,
                     route.start_depot, route.end_depot,
                     adjusted_tasks, route.start_time)
```

4.4 性能优化策略

4.4.1 约束检查优化

- **预计算距离矩阵**：避免重复计算节点间距离
- **增量约束检查**：只检查变化部分的约束违反
- **早期剪枝**：一旦发现约束违反立即停止后续检查

4.4.2 内存管理优化

- **对象池模式**：重用Route和Task对象
- **惰性计算**：只在需要时计算成本和时间
- **垃圾回收优化**：及时释放不需要的中间解

4.4.3 算法参数调优

```
# 遗传算法参数
GA_PARAMS = {
    'population_size': 80,       # 种群规模
    'generations': 300,          # 迭代次数
    'crossover_rate': 0.8,       # 交叉概率
    'mutation_rate': 0.3,        # 变异概率
    'elite_size': 3              # 精英数量
}

# 库存管理参数
INVENTORY_PARAMS = {
    'safety_stock_hours': 4.0,      # 安全库存时间
    'planning_horizon': 24.0,       # 计划时间范围
    'target_stock_ratio': 0.9,      # 目标库存比例
    'min_delivery_hours': 12.0      # 最短配送间隔
}
```

5. 实验结果与分析

5.1 实验环境

- **硬件配置**：Intel i5-13500H CPU，32GB RAM
- **软件环境**：Python 3.11，numpy 1.21.0，pandas 1.3.0
- **数据规模**：25个加油站，40辆车，3种油品，75个配送任务

5.2 算法性能评估

5.2.1 求解质量

```
最优解质量指标：
├── 总配送成本：16,268.49元
├── 路径数量：46条
├── 平均装载率：87.3%
```

```
├── 需求满足率：100%（所有任务完成）
└── 约束违反：0项
```

5.2.2 算法收敛性

- **收敛代数**：约150代后趋于稳定
- **最优解稳定性**：连续50代无改进后停止
- **解的多样性**：种群熵值保持在合理范围

5.2.3 计算效率

```
运行时间分析：
├── 阶段一(任务生成)：2.3秒
├── 阶段二(路径优化)：45.7秒
├── 结果分析：1.2秒
└── 总耗时：49.2秒
```

详细结果见`solution_report.txt`。