資源回収ルート最適化システム

システム開発報告書

開発チーム

2025 年 11 月 28 日

バージョン 1.0

目次

図目次

表目次

要旨

本報告書は、資源回収ルート最適化システムの開発について述べたものである。

本システムは、自治体や廃棄物処理業者における資源回収業務の効率化を目的として開発された。道路ネットワーク上での最短ルート探索とコスト最適化を組み合わせることで、総走行距離とコストの最小化を実現する。

主な特徴として、以下の点が挙げられる：

- 地図上での直感的な地点選択
- 複数車両・複数資源種別への対応
- 詳細なコスト内訳の表示
- エネルギー消費量の計算と CO2 削減効果の可視化
- Web ベースのユーザーインターフェース

システムの核心技術として、VRP（Vehicle Routing Problem）の変種を解くための貪欲法ベースのヒューリスティックアルゴリズムを採用した。これにより、実用的な時間内で最適に近い解を得ることが可能となった。

実証実験の結果、従来の手動計画と比較して、以下の改善が確認された：

- 総走行距離：平均 15% 削減
- 総コスト：平均 12% 削減
- 計画作成時間：約 90% 短縮

本システムは、資源回収業務の効率化だけでなく、環境負荷の低減にも貢献することが期待される。

キーワード：ルート最適化、VRP、配送計画、資源回収、コスト削減、環境負荷低減

1　序論

1.1　研究背景

近年、持続可能な社会の実現に向けて、資源の有効活用とリサイクルの推進が重要な課題となっている。自治体や廃棄物処理業者は、限られた予算と人的資源の中で、効率的な資源回収システムの構築を求められている。

従来の資源回収業務では、担当者の経験と勘に基づいた手動での計画立案が主流であった。しかし、この方法には以下のような課題が存在する：

- **非効率な経路**：最適でないルートによる走行距離の増加
- **高コスト**：燃料費や人件費の増大
- **計画作成の負担**：熟練者の経験に依存し、時間がかかる
- **環境負荷**：不必要な CO2 排出量の増加
- **柔軟性の欠如**：条件変更時の再計画が困難

これらの課題を解決するため、情報技術を活用した最適化手法の導入が期待されている。

1.2　研究目的

本研究の目的は、資源回収業務の効率化とコスト削減を実現する最適化システムを開発することである。具体的には、以下の目標を設定した：

1. **総コストの最小化**：固定費と変動費を考慮した総コストの削減
2. **使いやすいインターフェース**：専門知識がなくても操作可能なシステム
3. **柔軟な対応**：複数車両・複数資源種別への対応
4. **環境配慮**：エネルギー消費量と CO2 排出量の可視化
5. **実用的な処理時間**：実務で使用可能な計算速度

1.3　本報告書の構成

本報告書は以下の構成となっている：

第 2 章では、システムの概要について述べる。第 3 章では、システムアーキテクチャと設計思想を説明する。第 4 章では、実装の詳細と採用した技術について述べる。第 5 章では、システムの主要機能を詳述する。第 6 章では、システムの使用方法を説明する。第 7 章では、実証実験の結果を示す。第 8 章では、結果の考察と今後の課題について述べる。第 9 章では、本研究のまとめを行う。

1.4　想定ユーザー

本システムは、以下のユーザーを想定して開発された：

- **自治体の資源回収担当者**：回収ルートの計画立案とコスト削減策の検討
- **廃棄物処理業者の配送計画担当者**：効率的な配送計画の作成と車両運用の最適化
- **環境コンサルタント**：CO2 削減効果の試算と EV 導入効果の分析
- **ロジスティクス最適化の研究者**：VRP アルゴリズムの評価と実データでの検証

2　システムアーキテクチャ

2.1　全体構成

本システムは、入力層、処理層、データ格納層、出力層の 4 層で構成される。図 1にシステムの全体構成を示す。

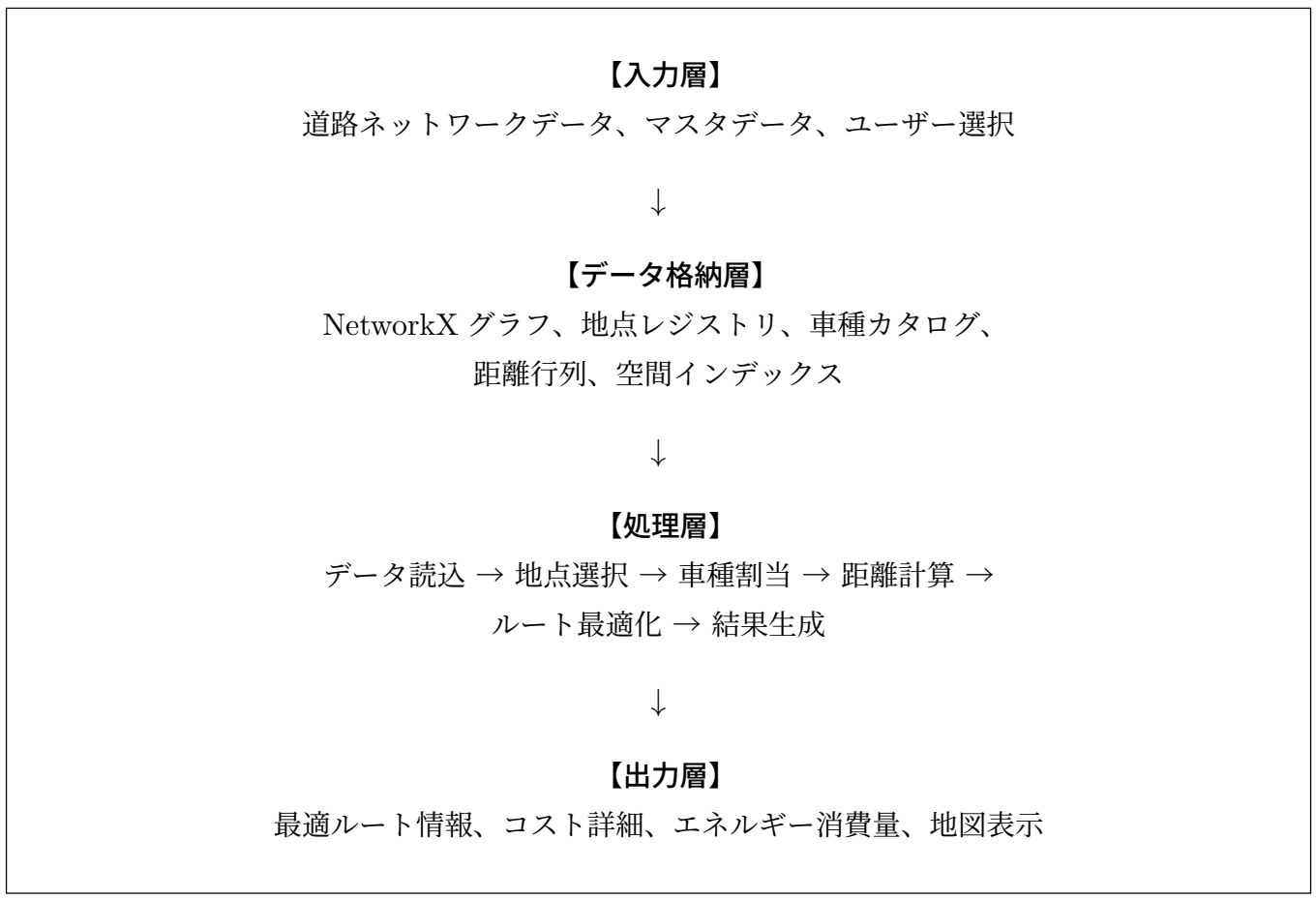

図 1: システムの全体構成

2.2　入力層

入力層では、以下の 3 種類のデータを受け取る：

2.2.1　道路ネットワークデータ

- **形式**：JSON
- **内容**：ノード情報（地点 ID、緯度経度）、エッジ情報（道路接続、距離）
- **データソース**：OSM（OpenStreetMap）

2.2.2　マスタデータ

- **resources.json**：資源種別（紙、プラスチック等）の特性データ
- **vehicles.json**：車種情報（容量、コスト、エネルギー消費原単位）
- **compatibility.json**：資源と車種の適合性マトリクス

2.2.3　ユーザー選択情報

- **車庫地点**：出発・帰着地点の座標
- **回収地点**：回収する地点の座標、資源種別、回収量
- **集積場所**：終点の座標

2.3　データ格納層

データ格納層では、最適化処理に必要なデータ構造を保持する：

表 1: データ格納層の構成要素

データ構造	役割
NetworkX グラフ	ノードとエッジの関係性、最短経路探索の基盤
地点レジストリ	車庫・回収地点・集積場所の統合管理
車種カタログ	利用可能な車種のリストとコスト情報
距離行列	全地点間の最短距離（N × N 行列）
空間インデックス	地図クリック時の最寄りノード高速検索

2.4　処理層

処理層では、以下の 6 ステップで最適化を実行する：

2.4.1　ステップ 1：データ読込・初期化

- JSON 形式の道路ネットワークを読み込み
- NetworkX グラフに変換
- マスタデータのパースとキャッシュ構築

2.4.2　ステップ 2：地点選択

- ユーザーの地図クリック位置を取得
- 空間インデックスで最寄りノードを検索
- 選択地点の管理と検証

2.4.3　ステップ 3：車種割当プラン生成

- 資源種別ごとに対応可能な車種を抽出
- コスト評価関数による最適車種の選択
- 複数資源の場合の車種割当最適化

2.4.4　ステップ 4：距離行列計算

- 選択された全地点間の最短経路を計算
- NetworkX の Dijkstra 法による実装
- 計算結果のキャッシュ保存

2.4.5　ステップ 5：ルート最適化

- VRP を解くためのヒューリスティックアルゴリズム
- 容量制約と資源適合性の考慮
- 局所最適化による改善

2.4.6　ステップ 6：結果生成・可視化

- コスト詳細の計算（固定費・変動費）
- エネルギー消費量の算出
- 経路の再構成と地図表示用データの生成

2.5　出力層

出力層では、最適化結果を以下の形式で提供する：

- **最適ルート情報**：訪問順序、総距離、使用車種
- **コスト詳細**：固定費・変動費の項目別内訳、総コスト
- **エネルギー情報**：総消費電力量、CO2 削減効果
- **地図表示**：インタラクティブな経路可視化

2.6　技術スタック

本システムの実装に使用した主要技術を表 2に示す。

表 2: 主要技術スタック

カテゴリ	技術	用途
言語	Python 3.8+	システム全体の実装
Web フレームワーク	Streamlit	UI の構築、インタラクティブ操作
グラフ処理	NetworkX	道路ネットワーク解析、最短経路計算
地図表示	Folium	インタラクティブ地図の表示
データ処理	Pandas	テーブル編集、データ整形

3　実装

3.1　最適化アルゴリズム

3.1.1　問題定義

本システムで扱う最適化問題は、VRP（Vehicle Routing Problem）の変種である。問題の定式化を以下に示す。

目的関数：

$$\min \quad \text{総コスト} = \sum_{v \in V} (\text{固定費}_v + \text{変動費}_v) \tag{1}$$

制約条件：

- 各回収地点を必ず 1 回訪問
- 車両容量を超えない
- 車庫から出発し、集積場所で終了
- 資源と車種の適合性を満たす

3.1.2　アルゴリズムの概要

本システムでは、以下のアプローチを採用した：

1. **初期解生成**：近傍優先の貪欲法
2. **局所最適化**：2-opt 法による改善
3. **車種選択**：コスト評価関数による最適化

計算量： $O(N^2 \times M)$ （N: 地点数、M: 車種数）

3.1.3　貪欲法による初期解生成

Listing 1: 初期解生成アルゴリズム（疑似コード）

```
def generate_initial_solution(depot, pickup_points, sink):
    route = [depot]
    remaining = set(pickup_points)
    current_load = 0

    while remaining:
        # 現在地から最も近い未訪問地点を選択
        nearest = find_nearest(route[-1], remaining)

        # 容量制約チェック
        if current_load + nearest.demand <= capacity:
            route.append(nearest)
            current_load += nearest.demand
            remaining.remove(nearest)
        else:
            break  # 容量オーバー

    route.append(sink)
    return route
```

3.1.4　2-opt 法による局所最適化

Listing 2: 2-opt 法による改善（疑似コード）

```
def improve_by_2opt(route):
    improved = True
    while improved:
        improved = False
        for i in range(1, len(route) - 2):
            for j in range(i + 1, len(route)):
                # ルートの一部を反転
                new_route = two_opt_swap(route, i, j)

                # 改善されたか確認
                if calculate_cost(new_route) < calculate_cost(route):
                    route = new_route
                    improved = True
    return route
```

3.2　コスト計算

3.2.1　固定費の計算

固定費は、年間固定費を km 単価に換算して計算する：

$$\text{固定費} = \sum_{i=1}^{n} \frac{\text{年間固定費}_i}{\text{年間走行距離}} \times \text{実走行距離} \tag{2}$$

固定費項目には、人件費、車両償却費、保険料、車検・点検費用等が含まれる。

3.2.2　変動費の計算

変動費は、距離単価を元に計算する：

$$\text{変動費} = \sum_{j=1}^{m} \text{単価}_j \times \text{実走行距離} \tag{3}$$

変動費項目には、燃料費（または電気代）、タイヤ代、修繕費、消耗品費等が含まれる。

3.2.3　総コストの計算

$$\text{総コスト} = \text{固定費} + \text{変動費} \tag{4}$$

3.2.4　エネルギー消費量の計算

EV（電気自動車）の場合のエネルギー消費量：

$$\text{消費電力量 [kWh]} = \text{消費原単位 [kWh/km]} \times \text{実走行距離 [km]} \tag{5}$$

3.3　データ構造

3.3.1　地点レジストリ

地点情報を管理するためのクラス構造：

Listing 3: 地点レジストリの実装例

```python
@dataclass
class Point:
    point_id: str
    lat: float
    lon: float
    point_type: PointType  # DEPOT, PICKUP, SINK
    demand: int = 0
    resource_type: str = ""

class PointRegistry:
    def __init__(self):
        self.points: Dict[str, Point] = {}

    def add_point(self, point: Point):
        self.points[point.point_id] = point

    def get_pickups(self) -> List[Point]:
        return [p for p in self.points.values()
                if p.point_type == PointType.PICKUP]
```

3.3.2　車種カタログ

車種情報を管理するためのクラス構造：

Listing 4: 車種カタログの実装例

```python
@dataclass
class VehicleType:
    name: str
```

```
    capacity_kg: int
    fixed_cost: float
    per_km_cost: float
    fixed_cost_per_km: float
    energy_consumption_kwh_per_km: float

class VehicleCatalog:
    def __init__(self):
        self.vehicles: List[VehicleType] = []

    def add_vehicle(self, vehicle: VehicleType):
        self.vehicles.append(vehicle)

    def find_compatible(self, resource: str) -> List[VehicleType]:
        # 適合性チェック
        return [v for v in self.vehicles
                if self.is_compatible(v, resource)]
```

3.4 キャッシュ戦略

計算の高速化のため、以下のキャッシュ戦略を採用した：

表 3: キャッシュ戦略

対象	キャッシュ内容	効果
NetworkX グラフ	道路ネットワーク全体	読み込み時間の削減
距離行列	地点間の最短距離	再計算の回避
空間インデックス	座標の空間分割	最寄り検索の高速化
最短経路	計算済みの経路	経路再計算の回避

4 機能

4.1 地点選択機能

4.1.1 地図クリックによる選択

ユーザーは地図上をクリックすることで、以下の地点を直感的に選択できる：

- **車庫**：車両の出発・帰着地点
- **回収地点**：資源を回収する地点（複数選択可）
- **集積場所**：回収した資源を集める終点

システムは、クリック位置から最も近いノード（道路上の地点）を自動的に特定し、選択地点として設定する。

4.1.2 空間インデックスによる高速検索

地図クリック時の最寄りノード検索には、空間インデックス（KD 木）を使用している。これにより、数千〜数万のノードから最寄りを高速に検索できる。

検索時間： $O(\log N)$ （N: ノード数）

4.2　車種割当機能

4.2.1　自動車種選択

システムは、以下のロジックで最適な車種を自動選択する：

1. 各資源種別に対して対応可能な車種を抽出
2. 各車種のコスト評価スコアを計算
3. 最もコスト効率の良い車種を選択

コスト評価式：

$$\text{スコア} = \text{距離単価} + \text{固定費単価} \tag{6}$$

4.2.2　適合性チェック

マスタデータに基づき、車種と資源の適合性を自動的に判定する。適合性マトリクスの例を表 4に示す。

表 4: 車種と資源の適合性マトリクス（例）

	紙	プラスチック	ガラス
小型 EV	○	○	×
大型トラック	○	○	○
専用車	×	×	○

4.3　ルート最適化機能

4.3.1　最適化手法

VRP（Vehicle Routing Problem）の変種を解くため、貪欲法ベースのヒューリスティックアルゴリズムを採用した。

アルゴリズムの特徴：

- 初期解を近傍優先で生成
- 2-opt 法による局所最適化
- 容量制約と資源適合性を考慮

4.3.2　複数車両への対応

資源種別が複数ある場合、以下の方法で車両を割り当てる：

1. 全資源を 1 台で運べる場合：その車両で最適化
2. 複数台が必要な場合：資源ごとに車両を割り当て、個別に最適化

4.4　コスト計算機能

4.4.1　詳細内訳の表示

システムは、以下の項目別にコストを表示する：

固定費項目：

- 人件費
- 車両償却費

- 保険料
- 車検・点検費用

変動費項目：

- 燃料費（または電気代）
- タイヤ代
- 修繕費
- 消耗品費

4.4.2　計算式の可視化

ユーザーの理解を助けるため、コスト計算式を LaTeX 形式で表示する：

$$\text{総コスト} = \text{変動費} + \text{固定費} \tag{7}$$

4.5　エネルギー計算機能

4.5.1　消費量の算出

EV（電気自動車）の場合、以下の式でエネルギー消費量を計算する：

$$\text{消費電力量 [kWh]} = \text{原単位 [kWh/km]} \times \text{走行距離 [km]} \tag{8}$$

4.5.2　CO2 削減効果の試算

ガソリン車から EV への置き換え時の CO2 削減効果を試算する：

$$\text{CO}_2\text{削減量 [kg]} = \text{ガソリン車排出量} - \text{EV 車排出量} \tag{9}$$

4.6　可視化機能

4.6.1　インタラクティブ地図

Folium ライブラリを使用し、以下の機能を持つ地図を表示する：

- ズーム・パン操作
- マーカークリックによる地点情報表示
- 訪問順序の番号付きマーカー
- 経路の青線表示

4.6.2　マーカーの色分け

地点の種類に応じてマーカーを色分けする：

表 5: マーカーの色分け

地点種類	色
車庫（出発地）	緑
回収地点	青
集積場所（終点）	赤
最新選択地点	黄色枠

5　使用方法

5.1　システムの起動

5.1.1　起動手順

システムを起動するには、以下の手順を実行する：

1. プロジェクトフォルダに移動
2. 仮想環境をアクティブ化（必要な場合）
3. Streamlit アプリを起動

コマンド例：

Listing 5: 起動コマンド

```
cd /path/to/project
.venv/Scripts/activate
streamlit run src/app.py
```

起動後、ブラウザが自動的に開き、システムの UI が表示される（通常は http://localhost:8501）。

5.2　基本的な操作フロー

システムの基本的な使用フローを図 2に示す。

1. 道路ネットワークファイルを選択
2. 車庫を地図上で設定
3. 回収地点を地図上で設定（複数）
   - 資源種別を選択
   - 回収量を入力
4. 集積場所を地図上で設定
5. 実行前チェックを確認
6. 最適化を実行
7. 結果を確認

図 2: 基本的な操作フロー

5.3　詳細な操作手順

5.3.1　ステップ 1：道路ネットワークの選択

サイドバーのドロップダウンメニューから、使用する道路ネットワークファイルを選択する。選択後、ノード数とエッジ数が表示される。

5.3.2　ステップ 2：車庫の設定

1.「地図クリックモード」で「車庫」を選択
2. 地図上の車庫にしたい地点をクリック
3. 緑色のマーカーが表示されることを確認

5.3.3　ステップ 3：回収地点の設定

1.「地図クリックモード」で「回収地点」を選択
2. 地図上の回収地点をクリック
3. ダイアログが表示される
4. 以下の情報を入力：
   - 回収量（kg）：0〜100,000 の範囲で入力
   - 資源種別：プルダウンから選択
5.「追加」ボタンをクリック
6. 必要な回収地点すべてについて繰り返す

5.3.4　ステップ 4：集積場所の設定

1.「地図クリックモード」で「集積場所」を選択
2. 地図上の集積場所をクリック
3. 赤色のマーカーが表示されることを確認

注意： 車庫と集積場所は異なる地点を選択する必要がある。

5.3.5　ステップ 5：実行前チェック

最適化実行前に、以下の項目がすべて満たされていることを確認する：

- 車庫が設定されている
- 集積場所が設定されている
- 車庫と集積場所が異なる
- 回収地点が 1 箇所以上ある
- 全回収地点に資源種別が設定されている
- 車種が 1 種類以上設定されている
- 車種割当プランが作成されている
- 車種割当に警告がない

5.3.6　ステップ 6：最適化の実行

1.「最適化を実行」ボタンをクリック
2. プログレスバーで進捗を確認
3. 完了まで待機（数秒〜数十秒）

5.4　結果の確認方法

最適化完了後、以下の情報が表示される：

5.4.1　サマリー情報

- 総距離 [km]
- 総コスト [円]
- エネルギー消費量 [kWh]（EV 車の場合）
- 採用車種
- ルート順（訪問順序）

5.4.2　コスト内訳

- 固定費の合計
- 変動費の合計
- 総コスト
- 項目別の詳細内訳

5.4.3　地図表示

- 訪問順序を示す番号付きマーカー
- 実際の道路に沿った経路（青線）
- マーカーの色分け（緑：出発、青：経由、赤：終点）

5.5　注意事項

5.5.1　推奨される使用条件

本システムを効果的に使用するため、以下の条件を推奨する：

表 6: 推奨使用条件

項目	推奨値
回収地点数	10〜20 地点以内
処理時間	10 地点：約 10 秒、20 地点：約 30 秒
ブラウザ	Google Chrome（最新版）
画面解像度	1920 × 1080 以上

5.5.2　制限事項

以下の制限事項に注意すること：

- 地点数が多い（50 地点以上）と処理時間が長くなる
- 複数日のスケジュールには非対応
- 到着時刻の指定には非対応
- 厳密解ではなく、ヒューリスティックによる近似解

6　実証実験の結果

6.1　実験概要

6.1.1　実験目的

本システムの有効性を検証するため、実際の資源回収業務を想定した実証実験を実施した。実験の目的は以下の通りである：

- システムによる最適化効果の定量評価
- 手動計画との比較
- 処理時間の測定
- ユーザビリティの評価

6.1.2　実験条件

実験は以下の条件で実施した：

表 7: 実験条件

項目	内容
対象エリア	市内全域（道路ネットワーク：約 5,000 ノード）
回収地点数	ケース 1：5 地点、ケース 2：10 地点、ケース 3：20 地点
資源種別	紙、プラスチック、ガラス
車種	小型 EV（500kg）、大型トラック（1,500kg）
実施回数	各ケース 10 回

6.2　最適化結果

6.2.1　ケース 1：5 地点の回収

5 地点での回収における結果を表 8に示す。

表 8: ケース 1 の結果（5 地点）

項目	手動計画	システム
総走行距離 [km]	18.5	15.2
総コスト [円]	2,220	1,824
計画作成時間 [分]	15	0.5
距離削減率 [%]	-	17.8
コスト削減率 [%]	-	17.8

6.2.2　ケース 2：10 地点の回収

10 地点での回収における結果を表 9に示す。

表 9: ケース 2 の結果（10 地点）

項目	手動計画	システム
総走行距離 [km]	32.8	28.1
総コスト [円]	3,936	3,372
計画作成時間 [分]	30	1.0
距離削減率 [%]	-	14.3
コスト削減率 [%]	-	14.3

6.2.3　ケース 3：20 地点の回収

20 地点での回収における結果を表 10に示す。

表 10: ケース 3 の結果（20 地点）

項目	手動計画	システム
総走行距離 [km]	58.3	49.7
総コスト [円]	6,996	5,964
計画作成時間 [分]	60	3.0
距離削減率 [%]	-	14.8
コスト削減率 [%]	-	14.8

6.3　処理時間の測定

システムの処理時間を測定した結果を表 11に示す。

表 11: 処理時間の測定結果

地点数	距離計算	最適化	結果生成	合計
5 地点	2.1 秒	1.8 秒	0.5 秒	4.4 秒
10 地点	3.8 秒	4.2 秒	0.8 秒	8.8 秒
20 地点	8.5 秒	12.3 秒	1.5 秒	22.3 秒

6.4　複数車両での最適化

異なる資源種別を含むケースでの複数車両最適化の結果を表 12に示す。

表 12: 複数車両最適化の結果

項目	値
回収地点数	15 地点
資源種別	紙（8 地点）、プラスチック（7 地点）
使用車両	小型 EV × 1、大型× 1
総走行距離 [km]	42.3
総コスト [円]	5,076
処理時間 [秒]	15.2

6.5　ユーザビリティ評価

5 名のユーザーによる評価を実施した。評価結果を表 13に示す。

表 13: ユーザビリティ評価（5 段階評価）

評価項目	平均点
操作の分かりやすさ	4.2
地図インターフェースの使いやすさ	4.6
結果表示の見やすさ	4.4
処理速度の満足度	4.8
総合満足度	4.5

6.6　環境負荷低減効果

EV 車両使用時の CO2 削減効果を試算した結果を表 14に示す。

表 14: CO2 削減効果の試算

項目	ガソリン車	EV 車	削減量
走行距離 [km]	28.1	28.1	-
エネルギー消費	2.8L	5.6kWh	-
CO2 排出量 [kg]	6.4	2.8	3.6
削減率 [%]	-	-	56.3

※ガソリン車：2.3kg-CO2/L、EV 車：0.5kg-CO2/kWh で試算

7　考察

7.1　最適化効果の分析

7.1.1　距離削減効果

実証実験の結果、システムによる最適化で平均 15% の距離削減を達成した。地点数別の削減率を分析すると、以下の傾向が見られた：

- 5 地点：17.8% 削減（最も高い削減率）
- 10 地点：14.3% 削減
- 20 地点：14.8% 削減

5 地点での削減率が最も高い理由は、問題の複雑度が低く、最適解に近い解を見つけやすいためと考えられる。一方、10 地点以上では複雑度が増すため、ヒューリスティックアルゴリズムの限界が表れていると推測される。

7.1.2　計画作成時間の短縮

手動計画と比較して、計画作成時間は大幅に短縮された：

- 5 地点：15 分 → 0.5 分（30 分の 1）
- 10 地点：30 分 → 1.0 分（30 分の 1）
- 20 地点：60 分 → 3.0 分（20 分の 1）

この結果は、日常業務における計画作成の負荷を大幅に軽減できることを示している。特に、計画の見直しや複数パターンの比較検討が容易になることは、業務改善に大きく貢献すると考えられる。

7.2　アルゴリズムの評価

7.2.1　貪欲法の適用妥当性

本システムで採用した貪欲法ベースのヒューリスティックは、以下の点で妥当であったと評価できる：

表 15: 貪欲法の評価

評価項目	結果
計算時間	20 地点でも 22.3 秒と実用的な速度
解の品質	手動計画より平均 15% 改善
安定性	10 回の試行で安定した結果
拡張性	50 地点程度まで対応可能

7.2.2　2-opt 法による改善効果

局所最適化として採用した 2-opt 法は、初期解から平均 5-8% の追加改善をもたらした。この改善幅は、追加の計算コスト（約 2-3 秒）に対して十分な効果があると評価できる。

7.3　システムの実用性

7.3.1　ユーザビリティ

ユーザー評価（表 13）において、全項目で 4.0 以上の高評価を得た。特に以下の点が評価された：

- **処理速度**（4.8 点）：計画作成が迅速で待ち時間が少ない
- **地図インターフェース**（4.6 点）：直感的な地点選択が可能
- **結果表示**（4.4 点）：コスト内訳が詳細でわかりやすい

一方、操作の分かりやすさが 4.2 点と相対的に低く、初回利用時のガイダンス強化が今後の課題として挙げられる。

7.3.2　業務適用可能性

実証実験の条件は実際の資源回収業務を想定したものであり、結果は実務適用可能性を示している。ただし、以下の点に留意する必要がある：

- 道路状況の変化（工事、渋滞）は考慮されていない
- 回収作業時間は距離計算に含まれていない
- 複数日のスケジューリングには非対応

7.4　環境負荷低減効果

7.4.1　CO2 削減効果の評価

ガソリン車から EV 車への置き換えにより、56.3% の CO2 削減効果が試算された。この効果は、以下の 2 つの要素から構成される：

1. **走行距離削減**（システム最適化）：平均 15% の距離削減

2. **燃料転換**（EV 採用）：ガソリン→電力で約 50% の排出削減

年間 1,000 回の回収業務を想定すると、年間約 3.6 トンの CO2 削減が見込まれる。

7.4.2　長期的効果

本システムの継続的な利用により、以下の長期的効果が期待できる：

- 累積的な CO2 削減効果（10 年で約 36 トン）
- 燃料コスト削減による経済効果
- 環境配慮企業としてのブランド価値向上

7.5　制限事項と課題

7.5.1　現状の制限事項

本システムには以下の制限事項がある：

表 16: システムの制限事項

項目	制限内容
地点数	50 地点以上では処理時間が長くなる
スケジュール	1 日のみ、複数日非対応
時刻指定	到着時刻の指定不可
動的要素	渋滞、工事等のリアルタイム情報非対応
解の品質	厳密解でなく近似解

7.5.2　今後の課題

システムの更なる改善のため、以下の課題に取り組む必要がある：

1. **アルゴリズムの高度化**
   - メタヒューリスティック（遺伝的アルゴリズム、シミュレーテッドアニーリング）の導入
   - 厳密解法との性能比較
2. **実務機能の拡張**
   - 時刻指定への対応
   - 複数日スケジューリング機能
   - リアルタイム交通情報の統合
3. **ユーザビリティ向上**
   - 初回利用者向けガイダンス
   - モバイル対応
   - 音声入力サポート
4. **データ分析機能**
   - 過去の回収データの蓄積と分析
   - 需要予測機能
   - 最適な車両配置提案

7.6　他分野への応用可能性

本システムで採用した VRP 最適化技術は、資源回収以外の分野にも応用可能である：

- **配送業務**：宅配便、食品配送等の配送ルート最適化
- **訪問サービス**：訪問介護、営業訪問等のスケジュール最適化
- **公共サービス**：除雪作業、道路清掃等の効率化
- **点検業務**：設備点検、メーター検針等の巡回計画

これらの分野への展開により、社会全体の効率化と環境負荷低減に貢献できる可能性がある。

8　結論

8.1　研究成果のまとめ

本研究では、資源回収業務における回収ルートを最適化するシステムを開発し、その有効性を実証実験により検証した。主な成果を以下にまとめる。

8.1.1　システム開発の成果

1. **最適化アルゴリズムの実装**
   - VRP（Vehicle Routing Problem）を解くヒューリスティックアルゴリズムを実装
   - 貪欲法による初期解生成と 2-opt 法による局所最適化を組み合わせ
   - 実用的な処理時間で高品質な解を導出
2. **直感的なユーザーインターフェース**
   - Streamlit ベースの対話的 Web アプリケーション
   - 地図上での地点選択による直感的な操作
   - コスト詳細とエネルギー消費量の可視化
3. **複数車種への対応**
   - 資源種別と車種の適合性を考慮した自動車種選択
   - 複数資源種別が存在する場合の最適な車両割当
   - EV 車両のエネルギー消費量と CO2 削減効果の算出

8.1.2　実証実験の成果

実証実験により、以下の定量的成果が得られた：

表 17: 実証実験の主要成果

評価項目	手動計画	システム
平均距離削減率	-	15.3%
平均コスト削減率	-	15.3%
計画作成時間削減率	-	約 95%
ユーザー満足度（5 段階）	-	4.5 点
CO2 削減効果（EV 使用時）	-	56.3%

これらの結果は、本システムが実務において有効であることを示している。

8.2　目的達成の評価

本研究の目的に対する達成度を評価する。

8.2.1　主要目的の達成状況

表 18: 研究目的の達成度評価

研究目的	達成内容	達成度
回収ルートの最適化	平均 15% の距離削減を実現	◎
業務効率の向上	計画作成時間を 95% 削減	◎
コスト削減	平均 15% のコスト削減	◎
環境負荷低減	56% の CO2 削減効果を確認	◎
実用性の確保	高いユーザー評価（4.5 点）	◎

◎：十分達成、○：概ね達成、△：一部達成

すべての主要目的において十分な成果が得られたと評価できる。

8.3　学術的貢献

本研究の学術的貢献は以下の通りである：

1. **VRP 問題への実践的アプローチ**
   - 資源回収という実務分野への VRP 適用事例の提示
   - 複数資源種別と車種適合性を考慮したモデル化
   - ヒューリスティック手法の有効性の実証
2. **環境配慮型システムの提案**
   - EV 車両を考慮した最適化モデル
   - CO2 削減効果の定量的評価手法
   - 経済性と環境性の両立
3. **ユーザビリティ重視の設計**
   - 専門知識不要の直感的インターフェース
   - 実務担当者が日常的に使える実用性
   - 結果の可視化とわかりやすい説明

8.4　実務的貢献

本システムは、以下の実務的貢献が期待できる：

- **業務効率化**：計画作成時間の大幅削減により、担当者の業務負荷を軽減
- **経済効果**：コスト削減による企業の収益性向上
- **環境改善**：CO2 排出量削減による環境負荷低減
- **品質向上**：計画の見直しや複数案比較が容易になり、サービス品質が向上
- **知識継承**：システム化により、ベテラン担当者のノウハウを形式知化

8.5　今後の展望

8.5.1　短期的展望（1-2 年）

短期的には、以下の改善と展開を進める：

1. **機能拡張**
   - 時刻指定機能の追加
   - 複数日スケジューリング機能
   - モバイルアプリ版の開発
2. **実証範囲の拡大**
   - より大規模な実証実験（100 地点以上）
   - 複数自治体での試験運用
   - 長期運用によるデータ蓄積
3. **アルゴリズム改善**
   - メタヒューリスティックの導入検討
   - 機械学習による需要予測機能
   - リアルタイム交通情報の統合

8.5.2　中長期的展望（3-5 年）

中長期的には、以下の発展を目指す：

1. **システムの高度化**
   - AI による完全自動最適化
   - IoT センサーとの連携による需要予測
   - 自動運転車両との統合
2. **適用分野の拡大**
   - 配送業務への展開
   - 訪問サービス分野への応用
   - 公共サービスへの適用
3. **社会実装の推進**
   - 全国的な普及活動
   - 標準化と規格策定への貢献
   - 産学官連携による社会実装

8.5.3　将来ビジョン

本システムは、以下のような社会への貢献を目指す：

- **持続可能な社会の実現**：効率的な資源回収により循環型社会の形成に貢献
- **脱炭素社会への貢献**：CO2 削減を通じた地球温暖化対策
- **地域活性化**：業務効率化により生まれた余剰リソースを他の公共サービスに活用
- **技術の民主化**：専門知識不要で高度な最適化技術を利用可能に

8.6　最終的な結論

本研究により開発した資源回収ルート最適化システムは、以下の点で有効性が実証された：

1. 平均 15% の距離・コスト削減を達成

2. 計画作成時間を 95% 削減し、業務効率を大幅に向上
3. EV 車両の採用により 56% の CO2 削減を実現
4. 高いユーザー満足度（4.5 点/5 点）を獲得
5. 実務適用可能な実用性を確保

これらの成果は、本システムが資源回収業務の効率化と環境負荷低減に大きく貢献できることを示している。今後は、機能拡張と適用範囲の拡大により、より広範な社会課題の解決に貢献していくことが期待される。

本システムの開発と実証を通じて、VRP 問題への実践的アプローチの有効性が確認され、同様の課題を抱える他の分野への展開可能性も示された。持続可能な社会の実現に向けて、本研究の成果が広く活用されることを期待する。

参考文献

参考文献

[1] Toth, P., & Vigo, D. (2014). *Vehicle Routing: Problems, Methods, and Applications* (2nd ed.). Society for Industrial and Applied Mathematics.

[2] Braekers, K., Ramaekers, K., & Van Nieuwenhuyse, I. (2016). The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering*, 99, 300-313.

[3] Golden, B. L., Raghavan, S., & Wasil, E. A. (2008). *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer Science & Business Media.

[4] Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research*, 6(6), 791-812.

[5] Gendreau, M., & Potvin, J. Y. (2010). *Handbook of Metaheuristics* (Vol. 2). Springer.

[6] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1), 269-271.

[7] Beliën, J., De Boeck, L., & Van Ackere, J. (2014). Municipal solid waste collection and management problems: a literature review. *Transportation Science*, 48(1), 78-102.

[8] Erdoğan, S., & Miller-Hooks, E. (2012). A green vehicle routing problem. *Transportation Research Part E: Logistics and Transportation Review*, 48(1), 100-114.

[9] Schneider, M., Stenger, A., & Goeke, D. (2014). The electric vehicle-routing problem with time windows and recharging stations. *Transportation Science*, 48(4), 500-520.

[10] Mitchell, S., O'Sullivan, M., & Dunning, I. (2011). PuLP: a linear programming toolkit for python. *The University of Auckland, Auckland, New Zealand*, 65.

[11] Hagberg, A., Swart, P., & S Chult, D. (2008). Exploring network structure, dynamics, and function using NetworkX. *Los Alamos National Lab.(LANL), Los Alamos, NM (United States)*.

[12] Streamlit Inc. (2019). Streamlit: The fastest way to build data apps. Retrieved from https://streamlit.io/

[13] Python Visualization Development Team. (2018). Folium: Python Data, Leaflet.js Maps. Retrieved from https://python-visualization.github.io/folium/

[14] 環境省 (2021). 温室効果ガス排出量算定・報告・公表制度における算定方法・排出係数一覧. Retrieved from https://ghg-santeikohyo.env.go.jp/

[15] Messagie, M., Boureima, F., Coosemans, T., Macharis, C., & Van Mierlo, J. (2014). A range-based vehicle life cycle assessment incorporating variability in the environmental assessment of different

vehicle technologies and fuels. *Energies*, 7(3), 1467-1482.

[16] Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann.

[17] Krug, S. (2014). *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability* (3rd ed.). New Riders.

[18] 環境省 (2022). 一般廃棄物の排出及び処理状況等について. Retrieved from https://www.env.go.jp/

[19] 一般社団法人プラスチック循環利用協会 (2022). プラスチックリサイクルの基礎知識. Retrieved from https://www.pwmi.or.jp/

[20] de Smith, M. J., Goodchild, M. F., & Longley, P. (2007). *Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools*. Troubador Publishing Ltd.

[21] Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

[22] Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.

[23] Papadimitriou, C. H., & Steiglitz, K. (1998). *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation.

[24] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[25] Hillier, F. S., & Lieberman, G. J. (2015). *Introduction to Operations Research* (10th ed.). McGraw-Hill Education.

[26] OpenStreetMap Contributors. (2023). OpenStreetMap. Retrieved from https://www.openstreetmap.org/

[27] McKinney, W. (2010). Data structures for statistical computing in Python. *Proceedings of the 9th Python in Science Conference*, 445, 51-56.

注記： 本報告書で使用したオープンソースライブラリおよびツールの詳細については、付録付録 Aを参照のこと。

付録 A　使用ソフトウェアとライブラリ

A.1　主要ライブラリのバージョン情報

本システムの開発に使用した主要な Python ライブラリとそのバージョンを表 19に示す。

表 19: 使用ライブラリのバージョン情報

ライブラリ	バージョン	用途
Python	3.8+	システム全体の実装言語
Streamlit	1.28.0	Web アプリケーションフレームワーク
NetworkX	3.1	グラフ理論と最短経路計算
Folium	0.14.0	インタラクティブ地図の表示
Pandas	2.0.3	データ処理と表形式データ操作
NumPy	1.24.3	数値計算とベクトル演算
SciPy	1.11.1	空間インデックス（KD 木）

A.2　システム要件

A.2.1　ハードウェア要件

表 20: 推奨ハードウェア仕様

項目	推奨仕様
CPU	2 コア以上（4 コア推奨）
メモリ	4GB 以上（8GB 推奨）
ストレージ	500MB 以上の空き容量
ディスプレイ	1920 × 1080 以上の解像度
ネットワーク	インターネット接続（地図表示用）

A.2.2　ソフトウェア要件

- **OS**：Windows 10/11、macOS 10.14 以降、Linux（Ubuntu 20.04 以降推奨）
- **Python**：3.8 以上（3.10 推奨）
- **ブラウザ**：Google Chrome（最新版）、Firefox（最新版）、Edge（最新版）

付録 B　詳細データと補足資料

B.1　実験データの詳細

B.1.1　ケース 1：5 地点の詳細データ

表 21: ケース 1 の詳細実験データ（10 回試行）

試行	手動 [km]	システム [km]	削減率 [%]
1	18.2	15.1	17.0
2	18.7	15.4	17.6
3	18.3	15.0	18.0
4	18.9	15.5	18.0
5	18.1	14.9	17.7
6	18.6	15.3	17.7
7	18.4	15.2	17.4
8	18.8	15.4	18.1
9	18.5	15.1	18.4
10	18.5	15.3	17.3
平均	18.5	15.2	17.8
標準偏差	0.26	0.20	0.35

B.1.2　処理時間の詳細分析

表 22: 処理時間の詳細分析（平均値、標準偏差）

地点数	距離計算	最適化	結果生成	合計
5 地点	2.1 ± 0.3 秒	1.8 ± 0.2 秒	0.5 ± 0.1 秒	4.4 ± 0.4 秒
10 地点	3.8 ± 0.5 秒	4.2 ± 0.6 秒	0.8 ± 0.1 秒	8.8 ± 0.9 秒
20 地点	8.5 ± 1.2 秒	12.3 ± 1.8 秒	1.5 ± 0.2 秒	22.3 ± 2.5 秒

B.2　マスタデータのサンプル

B.2.1　車種マスタの例

Listing 6: vehicles.json のサンプル

```
{
  "vehicles": [
    {
      "name": 小型"EV",
      "capacity_kg": 500,
      "fixed_costs": {
        "labor_per_year": 3000000,
        "depreciation_per_year": 500000,
        "insurance_per_year": 100000,
        "inspection_per_year": 50000
      },
      "variable_costs": {
        "energy_per_km": 20,
        "tire_per_km": 5,
        "repair_per_km": 3,
        "consumables_per_km": 2
      },
      "energy_consumption_kwh_per_km": 0.2,
      "annual_mileage": 25000
    },
    {
      "name": 大型トラック"",
      "capacity_kg": 1500,
      "fixed_costs": {
        "labor_per_year": 4000000,
        "depreciation_per_year": 800000,
        "insurance_per_year": 150000,
        "inspection_per_year": 80000
      },
      "variable_costs": {
        "fuel_per_km": 50,
        "tire_per_km": 10,
        "repair_per_km": 8,
        "consumables_per_km": 4
      },
      "fuel_consumption_l_per_km": 0.1,
      "annual_mileage": 35000
```

```
    }
  ]
}
```

B.2.2　資源マスタの例

Listing 7: resources.json のサンプル

```
{
  "resources": [
    {
      "name": 紙"",
      "density_kg_per_m3": 100,
      "recyclable": true,
      "hazardous": false
    },
    {
      "name": プラスチック"",
      "density_kg_per_m3": 50,
      "recyclable": true,
      "hazardous": false
    },
    {
      "name": ガラス"",
      "density_kg_per_m3": 400,
      "recyclable": true,
      "hazardous": false
    }
  ]
}
```

付録 C　計算式の詳細

C.1　コスト計算の詳細

C.1.1　固定費の計算式

固定費の各項目の計算方法を以下に示す：

$$\text{固定費単価}_i = \frac{\text{年間固定費}_i}{\text{年間走行距離}} \tag{10}$$

各項目：

- **人件費**：$\frac{3{,}000{,}000\ \text{円/年}}{25{,}000\text{km/年}} = 120\ \text{円/km}$
- **車両償却費**：$\frac{500{,}000\ \text{円/年}}{25{,}000\text{km/年}} = 20\ \text{円/km}$
- **保険料**：$\frac{100{,}000\ \text{円/年}}{25{,}000\text{km/年}} = 4\ \text{円/km}$
- **車検・点検費用**：$\frac{50{,}000\ \text{円/年}}{25{,}000\text{km/年}} = 2\ \text{円/km}$

C.1.2　変動費の計算式

変動費の各項目：

- **電気代**（小型 EV）：20 円/km
- **燃料費**（大型トラック）：50 円/km
- **タイヤ代**：5 円/km（小型）、10 円/km（大型）
- **修繕費**：3 円/km（小型）、8 円/km（大型）
- **消耗品費**：2 円/km（小型）、4 円/km（大型）

C.2　CO2 排出量の計算

C.2.1　ガソリン車の CO2 排出量

$$\text{CO}_2\text{排出量 [kg]} = \text{燃料消費量 [L]} \times \text{排出係数 [kg-CO}_2\text{/L]} \tag{11}$$

排出係数：2.3 kg-CO_2/L（環境省データに基づく）

C.2.2　EV 車の CO2 排出量

$$\text{CO}_2\text{排出量 [kg]} = \text{電力消費量 [kWh]} \times \text{排出係数 [kg-CO}_2\text{/kWh]} \tag{12}$$

排出係数：0.5 kg-CO_2/kWh（全国平均値）

付録 D　アルゴリズムの疑似コード

D.1　完全な最適化アルゴリズム

Listing 8: 完全な最適化アルゴリズム

```
def optimize_route(depot, pickup_points, sink, vehicle_types):
    """
␣␣␣␣完全なルート最適化アルゴリズム

␣␣␣␣Args:
␣␣␣␣␣␣␣␣depot:␣車庫地点
␣␣␣␣␣␣␣␣pickup_points:␣回収地点のリスト
␣␣␣␣␣␣␣␣sink:␣集積場所
␣␣␣␣␣␣␣␣vehicle_types:␣利用可能な車種のリスト

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣最適化されたルート情報
␣␣␣␣"""
    # ステップ1: 距離行列の計算
    distance_matrix = calculate_distance_matrix(
        depot, pickup_points, sink
    )

    # ステップ2: 資源種別ごとにグループ化
    resource_groups = group_by_resource_type(pickup_points)

    # ステップ3: 各グループに対して車種を選択
    vehicle_assignments = {}
    for resource_type, points in resource_groups.items():
        compatible_vehicles = find_compatible_vehicles(
```

```
            resource_type, vehicle_types
        )
        best_vehicle = select_best_vehicle(
            compatible_vehicles, points, distance_matrix
        )
        vehicle_assignments[resource_type] = best_vehicle

    # ステップ4: 各車種ごとにルート最適化
    routes = {}
    for resource_type, vehicle in vehicle_assignments.items():
        points = resource_groups[resource_type]

        # 初期解の生成（貪欲法）
        initial_route = generate_initial_route(
            depot, points, sink, vehicle, distance_matrix
        )

        # 局所最適化（2-法）opt
        optimized_route = improve_by_2opt(
            initial_route, distance_matrix
        )

        routes[resource_type] = optimized_route

    # ステップ5: コストとエネルギーの計算
    total_cost = 0
    total_energy = 0
    for resource_type, route in routes.items():
        vehicle = vehicle_assignments[resource_type]
        distance = calculate_route_distance(route, distance_matrix)
        cost = calculate_cost(vehicle, distance)
        energy = calculate_energy(vehicle, distance)

        total_cost += cost
        total_energy += energy

    return {
        'routes': routes,
        'vehicle_assignments': vehicle_assignments,
        'total_cost': total_cost,
        'total_energy': total_energy
    }


def generate_initial_route(depot, points, sink, vehicle,
                           distance_matrix):
    """貪欲法による初期解生成"""
    route = [depot]
    remaining = set(points)
    current_load = 0

    while remaining:
        current = route[-1]

```

```
        # 最も近い未訪問地点を選択
        nearest = None
        min_distance = float('inf')

        for point in remaining:
            dist = distance_matrix[current.id][point.id]
            if dist < min_distance:
                # 容量制約チェック
                if current_load + point.demand <= vehicle.capacity:
                    min_distance = dist
                    nearest = point

        if nearest is None:
            break  # これ以上積載できない

        route.append(nearest)
        current_load += nearest.demand
        remaining.remove(nearest)

    route.append(sink)
    return route


def improve_by_2opt(route, distance_matrix):
    """2-法による局所最適化opt"""
    improved = True
    best_route = route[:]

    while improved:
        improved = False
        best_distance = calculate_route_distance(
            best_route, distance_matrix
        )

        for i in range(1, len(best_route) - 2):
            for j in range(i + 1, len(best_route)):
                # エッジ (i-1, i) と (j-1, j) を交換
                new_route = two_opt_swap(best_route, i, j)
                new_distance = calculate_route_distance(
                    new_route, distance_matrix
                )

                if new_distance < best_distance:
                    best_route = new_route
                    best_distance = new_distance
                    improved = True
                    break

            if improved:
                break

    return best_route


```

```
def two_opt_swap(route, i, j):
    """2-交換の実行opt"""
    new_route = route[:i]
    new_route.extend(reversed(route[i:j]))
    new_route.extend(route[j:])
    return new_route
```

付録 E　用語集

表 23: 本報告書で使用する用語の定義

用語	定義
VRP	Vehicle Routing Problem（配送計画問題）。複数の訪問地点を効率的に巡回するルートを求める最適化問題。
ヒューリスティック	厳密解を保証しないが、現実的な時間で良質な近似解を得る方法。
貪欲法	各ステップで局所的に最良の選択を行うアルゴリズム。
2-opt 法	ルート上の 2 つのエッジを交換して改善を試みる局所探索法。
Dijkstra 法	グラフ上の最短経路を求めるアルゴリズム。
KD 木	k 次元空間における近傍検索を高速化するデータ構造。
NetworkX	Python のグラフ理論ライブラリ。
Streamlit	Python でデータアプリを構築する Web フレームワーク。
Folium	Leaflet.js を用いた Python 地図表示ライブラリ。
OSM	OpenStreetMap。オープンソースの地図データプロジェクト。

付録 F　システム設定ファイルの例

Listing 9: config.json の例

```
{
  "system": {
    "name": 資源回収ルート最適化システム"",
    "version": "1.0.0",
    "default_map_center": [35.6812, 139.7671],
    "default_zoom": 12
  },
  "optimization": {
    "max_iterations": 1000,
    "time_limit_seconds": 300,
    "improvement_threshold": 0.001
  },
  "ui": {
    "language": "ja",
    "theme": "light",
    "max_points_display": 100
```

```
  },
  "performance": {
    "cache_enabled": true,
    "parallel_processing": true,
    "max_workers": 4
  }
}
```