情報工学実験 1 レポート

巡回セールスマン問題

【第八班】

AJG23055 牧野唯希

2025/06/03

A. 目的

知能と情報に関わる様々な分野ではシステムの最適化やアルゴリズムを用いたシミュレーションに基づく問題解決が試みられる場面が多い。本実験を通して、能や情報との関連が深い問題に対して最適化手法や種々のアルゴリズムを用いてシステムの解析やシミュレーションを行うことにより、最適化アルゴリズムに関する理解を深める。

B. 解説

私の班では「巡回セールスマン問題」について取り組んだ。そのため、「巡回セールスマン問題」の概要について解説する。

巡回セールスマン問題とは、セールスマンの顧客がいくつかの都市に住んでいるとして、ある都市から出発し、他の都市を一回ずつ順にすべて訪問して、最初の年に戻ってくる場合に、総移動距離が最小になる巡回路を探索する問題のことである。

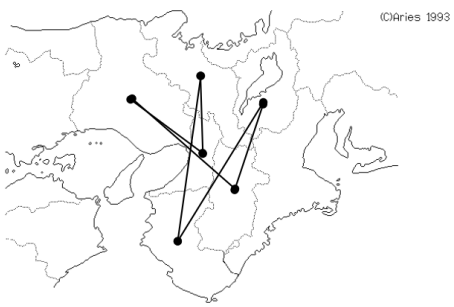

図 1

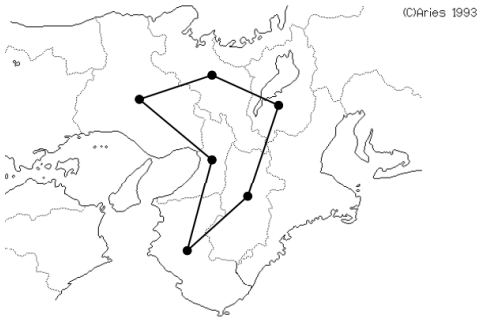

図 2

例えば、近畿地方の各都市を巡回する場合、図 1 のように巡回するのではなく、図 2 のように巡回する方が効率的であることは明らかである。このような経路最小化問題は実世界にも存在し、例えば、コンピュータなどに使われているプリント基板への穴あけや素子は一といった工程の効率化と密接な関係がある。この最小経路の探索は一見単純に思われるが、実際には計算困難な組み合わせ最適化問題として有名である。都市の数が n 個であ

るとすると、巡回路の総数は$\frac{1}{2}(n-1)!$となるが、仮に n=30 の時には経路数は 4.42×10^{30} にもおよび、そのすべてを調べることは容易でなくなる。このように問題の大きさに関する多項式に比例した計算時間では、厳密な最適解を求めることが困難な問題は NP 困難問題と呼ばれ、種々の工夫を施しながら、なるべく良い解を許容時間内に得ようとするアプローチが必要になる。

本実験では、

(1) 簡便な計算に基づく準回路の構築法
(2) 巡回路の改善を繰り返す局所最適解の探索法
(3) 局所解からの脱出を試みるメタ戦略
(4) 大域的最適解の探索法

の 4 種類を用いてシミュレーションを行うこととする。

また、検証した結果について、プレゼンテーションを行う。

C. 使用機器

本実験では、以下の条件で行った。

C P U：	AMD Ryzen 7 5825U
R A M：	16.0GB
使用エディター：	Visual Studio Code
使 用 言 語：	Python

D. 実験方法

先ほど述べた、シミュレーションを行う前に前提として、都市がランダムに配置された空間を用意する必要がある。そこでまず、都市の生成を行った。

(0) 都市の生成（担当：湯村、山本）

空間の条件としては、200×200 の 2 次元平面上にランダムかつ重なりがないように年を設定した。各都市の位置は座標として設定し、三平方の定理で各都市間の距離を計測できるようにした。

```
# 距離を計算する関数（ユークリッド距離）
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
```

生成した空間の描画には matplotlib を用いた。

(1) 簡便な巡回路の構築法（担当：牧野、浅野）

実行可能な巡回路を直感的に構築する発見的な手法であり、計算が簡便で実装が容易なアルゴリズムが種々、考案されている。本実験では、その中の一つの「最近隣法」を使用した。

最近隣法は、任意の都市から出発し、まだ訪問していない都市の中で現在位置から最も近い都市へ移動する方法である。これを繰り返し、すべての都市を訪問したら出発点に戻ることで初期経路を作成した。

```
# 最近隣法による経路探索（デコレータ適用）
@measure_time
def nearest_neighbor(cities):
    n = len(cities)
    visited = [False] * n
    path = [0]  # スタートは都市 0
    visited[0] = True
    current = 0
    for _ in range(n - 1):
        nearest_city = None
        nearest_dist = float('inf')
        for j in range(n):
            if not visited[j]:
                d = distance(cities[current], cities[j])
                if d < nearest_dist:
                    nearest_dist = d
                    nearest_city = j
        path.append(nearest_city)
        visited[nearest_city] = True
        current = nearest_city
    return path
```

(2) 巡回路の改善を繰り返す局所最適解の探索法（担当：牧野、浅野）

最近隣法により構築された巡回路を部分的にしゅうせいすることにより改善するアプローチであり、それ以上の改善が不可能になる解（局所的最適解）が得られるまで修正を繰り返す。「初期の巡回路」や「部分的な修正の方法」によっては、必ずしも全準回路で最も良い解（大域的な最適解）に到達できるとは限らないが、比較的に短時間の操作で最近隣法の回を改善できる点で有効である。「部分的な修正」を行う際に改善の候補を探索する「近傍」の設定方法により、いくつかの手法があるが、今回は 2-opt 法を用いて探索を行った。

2-opt 法は、構築済みの巡回路に対して、任意の 2 本の枝を付け替えることで得られる巡回路を改善の候補とし、候補の中で移動距離の短縮が最も大きいものを次の解とする。そして、更新後の解について同様に改善の候補を作成して、次の候補を探索した。改善が可能な限りこの操作を続け、候補の中に改善される解が無くなれば、終了

することで、最適化探索を行うものである。

```
# 2-opt 法による経路改善
@measure_time
def two_opt(path, cities):
    improved = True
    iteration = 0
    best_distance = total_distance(cities, path)

    while improved:
        improved = False
        iteration += 1
        for i in range(1, len(path) - 1):
            for j in range(i + 1, len(path)):
                if j - i == 1:  # 隣接辺はスキップ
                    continue
                # 経路の一部を反転
                new_path = path[:i] + path[i:j+1][::-1] + path[j+1:]
                new_distance = total_distance(cities, new_path)

                if new_distance < best_distance:
                    path = new_path
                    best_distance = new_distance
                    improved = True
                    break
            if improved:
                break

        if iteration % 10 == 0:
            print(f"2-opt 反復回数: {iteration}, 現在の距離: {best_distance:.2f}")

    print(f"2-opt 法の総反復回数: {iteration}")
    return path
```

(3) 局所解からの脱出を試みるメタ戦略（担当：牧野、浅野）

(2)の 2-opt 法では、初期巡回路の取り方によっては大域的な最適解に到達できないため、場合によっては制度の悪い解となっている可能性がある。これは解の探索が狭い範囲に集中していることに起因する。そこで、曲初回からの脱出を試みる工夫を施すことで、なるべく広い範囲を探索しながら最適解に到達しようとするメタ戦略に取り組んだ。本実験ではその中の一つである多出発局所探索を用いた。

多出発局所探索とは、初期巡回路を種々に変えながら局所探索を多数回実行し、その中でも最も総移動距離が短いものを解として、採用する手法である。本実験では、初期巡回路の設定のために二つの方法を用いた。一つ目は、最近隣法の初期都市を他の都市にすることで、もう一つは完全にランダムな巡回路を設定することである。これらによって生成された初期巡回路について、2-opt 法で最適化探索を行い、得られた結果通しを比較し最も短いものを採用した。

```
# 多出発局所探索（Multi-start Local Search）
@measure_time
def multi_start_local_search(cities, num_starts=20, max_iterations=1000):
    n = len(cities)
    best_path = None
    best_distance = float('inf')
    all_results = []

    print(f"\n 多出発局所探索を実行中... (開始点数: {num_starts})")

    # 戦略 1: 全都市を開始点とした最近隣法
    for start in range(min(n, num_starts // 2)):
        print(f"都市 {start} から始める最近隣法 + 2-opt...")
        initial_path = nearest_neighbor_from(cities, start)
        improved_path, improved_dist, iters = two_opt(initial_path, cities,
max_iterations)
        all_results.append((improved_path, improved_dist, f"NN from city {start}",
iters))

        if improved_dist < best_distance:
            best_distance = improved_dist
            best_path = improved_path.copy()
            print(f"新たな最良解: {improved_dist:.2f} (都市{start}からの最近隣法)")

    # 戦略 2: ランダム経路
    remaining_starts = num_starts - min(n, num_starts // 2)
    for i in range(remaining_starts):
        print(f"ランダム初期解 {i+1}/{remaining_starts} + 2-opt...")
        initial_path = random_path(n)
        improved_path, improved_dist, iters = two_opt(initial_path, cities,
max_iterations)
        all_results.append((improved_path, improved_dist, f"Random path {i+1}",
iters))

        if improved_dist < best_distance:
            best_distance = improved_dist
            best_path = improved_path.copy()
            print(f"新たな最良解: {improved_dist:.2f} (ランダム経路{i+1})")

    # 結果をソートして表示
    all_results.sort(key=lambda x: x[1])
    print("\n--- 多出発局所探索の結果一覧 ---")
    for i, (path, dist, method, iters) in enumerate(all_results):
        print(f"{i+1}. {method}: {dist:.2f} (反復: {iters}回)")

    return best_path, best_distance
```

(4) 大域的最適解の探索法（担当：高井、山本）

(3)の多出発局所探索などのメタ戦略では、なるべく広い範囲を探索することで良い解

を得ようとするが、必ずしも大域的最適解（全体の最適解）に到達する保証はない。したがって、真の最適解を得たい場合には、すべての巡回路を列挙して、その中で最適なものを見つける（列挙法という）か、もしくは、緩和問題などを利用して全探索を簡略化する解法（厳密解法）を用いる必要がある。その中でも今回は「Held-Karp 法」と「分枝限定法」の 2 通りの手法を用いて大域的探索解の探索法を行った。

1. Held-Karp 法（担当：高井）

   Held-Karp 法は、与えられた都市の集合に対して、動的計画法を用いて、すべての都市を一度だけ訪問して最短の巡回路を求める方法である。まず、都市の部分集合ごとに最短経路を再帰的に計算し、最後に全都市を訪問する最短経路を組み合わせて求める。計算量は指数的に増加するが、最適解を提供することが出来る。

```python
# --- Held-Karp 法で TSP を解く ---
INF = float('inf')
dp = [[INF] * n for _ in range(1 << n)]
prev = [[-1] * n for _ in range(1 << n)]
dp[1][0] = 0  # 開始は City 0

# 状態遷移
for mask in range(1 << n):
    for u in range(n):
        if not (mask & (1 << u)):
            continue
        for v in range(n):
            if mask & (1 << v):
                continue
            next_mask = mask | (1 << v)
            cost = dp[mask][u] + dist[u][v]
            if cost < dp[next_mask][v]:
                dp[next_mask][v] = cost
                prev[next_mask][v] = u

# ゴールに戻る
final_mask = (1 << n) - 1
min_cost = INF
last_city = -1
for i in range(1, n):
    cost = dp[final_mask][i] + dist[i][0]
    if cost < min_cost:
        min_cost = cost
        last_city = i

# 経路復元
path = [0]
mask = final_mask
current = last_city
while current != 0:
    path.append(current)
    current, mask = prev[mask][current], mask ^ (1 << current)
path.append(0)
```

```
path.reverse()

# 出力
print("\n巡回順:")
for i, city in enumerate(path):
    print(f"{i + 1}: City {city}")

print("\n最短経路のコスト:", round(min_cost, 2))

elapsed_time = time.time() - start_time
print("計算時間: {:.4f} 秒".format(elapsed_time))
```

2. 分枝限定法（担当：山本）

分枝限定法は、分枝操作と限定操作の 2 つから成り立つ手法である。分子操作は問題をより小さな部分問題に分割する操作で、限定操作はある部分問題の上限値が現在の暫定解よりも悪い場合、その部分問題はそれ以上調べる価値がないと判断し「枝刈り」によって探索から除外する操作だ。このようにすることで、計算量を大幅に削減しながら厳密な最適解を求めることが出来る。

```
# ======== 分枝限定法 ========
@measure_time
def branch_and_bound_tsp(cities):
    n = len(cities)
    best_cost = float('inf')
    best_path = []

    queue = []
    heapq.heappush(queue, (0, 0, [0]))  # (bound, cost, path)

    while queue:
        bound, cost, path = heapq.heappop(queue)

        if bound >= best_cost:
            continue

        if len(path) == n:
            total = cost + distance(cities[path[-1]], cities[0])
            if total < best_cost:
                best_cost = total
                best_path = path[:]
            continue

        for next_city in range(n):
            if next_city in path:
                continue
            new_cost = cost + distance(cities[path[-1]], cities[next_city])
            remaining = set(range(n)) - set(path) - {next_city}
            estimate = new_cost
            if remaining:
                estimate += min(distance(cities[next_city], cities[r]) for r in
remaining)
            heapq.heappush(queue, (estimate, new_cost, path + [next_city]))
```

```
    return best_path, best_cost
```

E. 実験結果

(0) 都市の生成

都市を生成した結果を平面上にプロットしたのが以下の図 3 である。

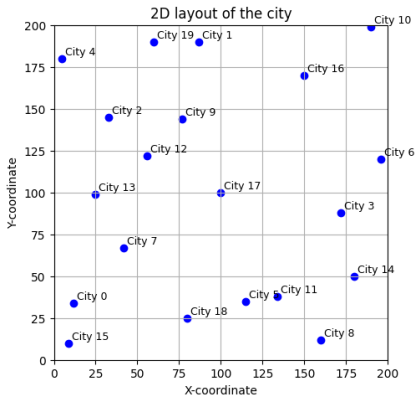

図 3 都市 20 の空間

(1) 簡便な巡回路の構築法

上で生成した都市に対して最近隣法を用いて簡便な巡回路を生成した結果が図 4 である。なお、計算時間は、0.000167 秒であり、総移動距離は 963.57 であった。

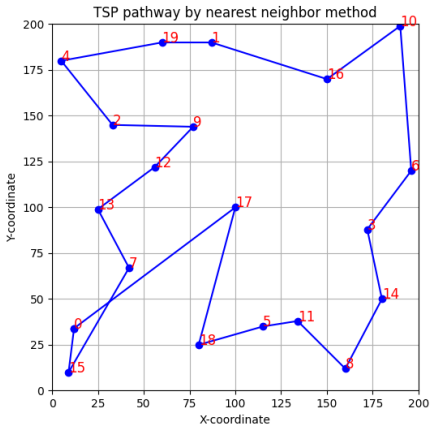

図 4 最近隣法

(2) 巡回路の改善を繰り返す局所最適解の探索法

最近隣法を用いて作成した巡回路を 2-opt 法を用いて改善させた結果が図 5 である。なお、計算時間は 0.007701 秒であり、総移動距離は 910.07 であった。

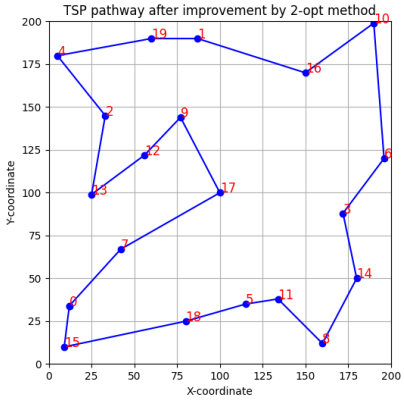

図 5 2-opt 法

(3) 局所解からの脱出を試みるメタ戦略

次に、多出発局所探索を用いて、局所解からの脱出を試み、大域的な最適解を求めようとした結果が図 6 である。なお、計算時間は 0.257616 秒で、総移動距離は 895.68 となった。

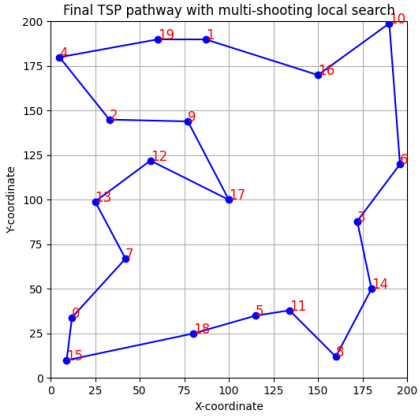

図 6 多出発局所探索

ただし、多出発局所探索では、試行によっては異なる解が得られる事象もあった。

(4) 大域的最適解の探索法

ここからは直接大域的最適解を探索した結果を記す。

まず Held-Karp 法を用いて探索した結果が図 7 である。計算時間は、62.2539 秒で、

総移動距離は 895.68 となった。

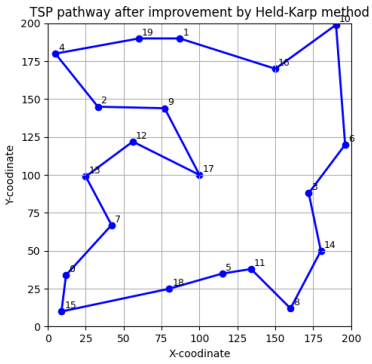

図 7 Held-Karp 法

次に分枝限定法を用いた探索を行ったが、都市数 20 の設定では、数分経過しても、処理されず、パソコンがうなり始めた。Replit 上でも実行してみたところ、時間経過で強制終了されてしまった。

なお、都市数を 10 に減らしてみたところ、時間はかかったものの問題なく処理することが出来た。

そのため、分枝限定法では処理不可能ということが分かった。

(5) 結果まとめ

各最適化法の計算時間と総移動距離をまとめたものは以下の図 8 の通りである。

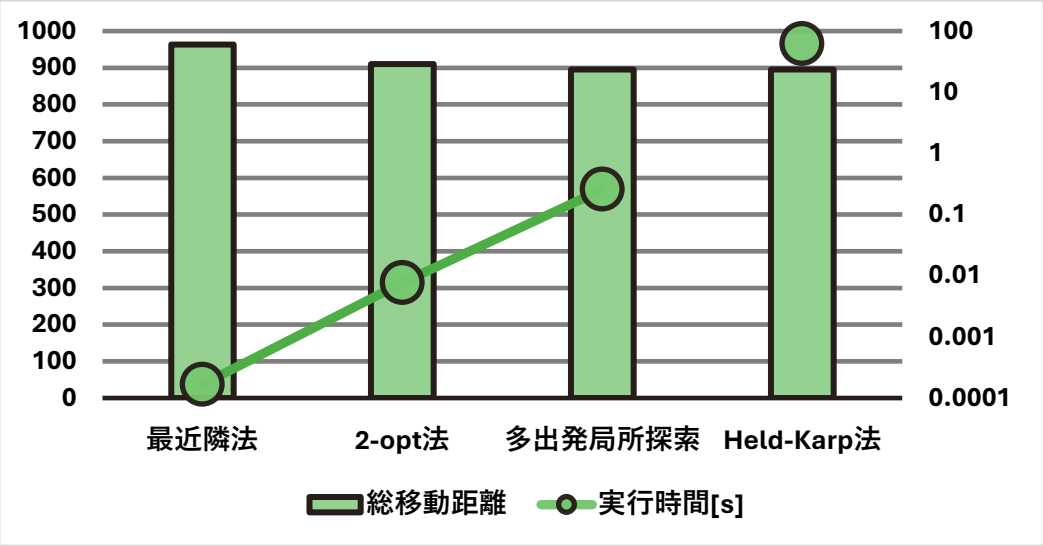

図 8 各最適化手法の総移動距離と実行時間

また、都市数を 25,30 に変えた場合は以下の図 9,10 の通りである。ただし、都市数 25 と 30 の場合においては Held-Karp 法は処理しきれずに測定不可能となった。

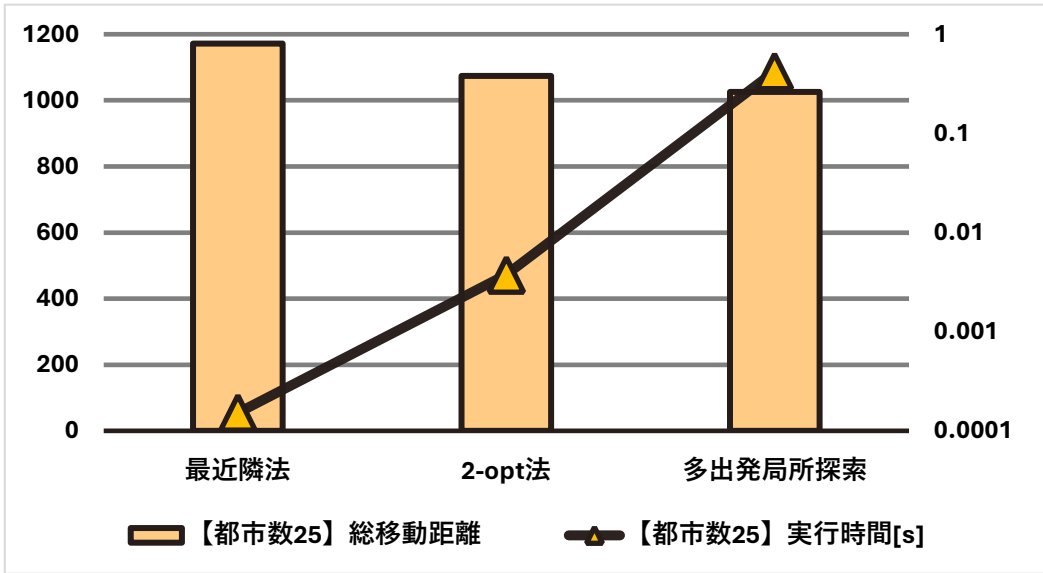

図 9 各最適化手法の総移動距離と実行時間（都市数 25）

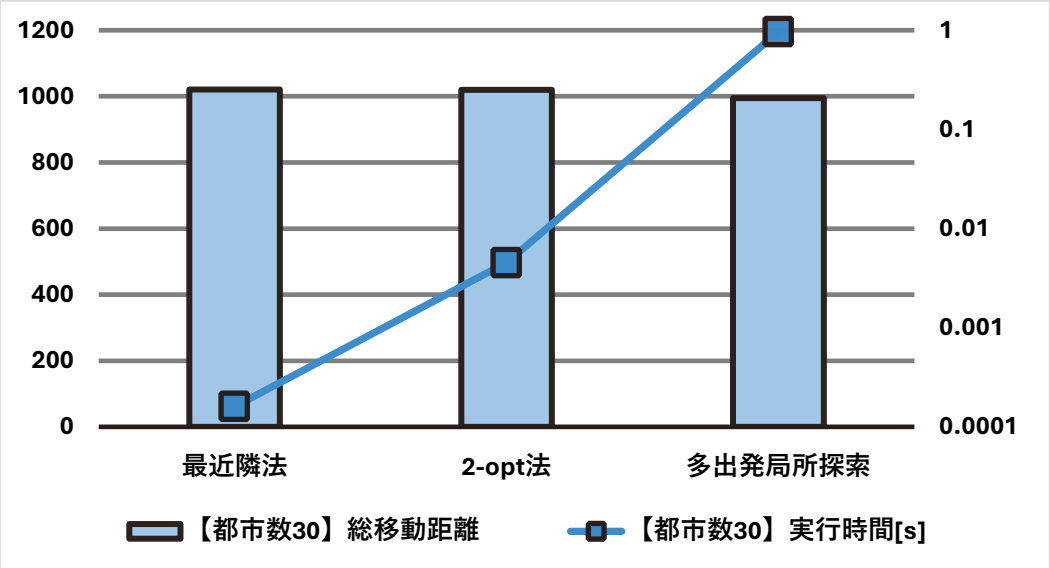

図 10 各最適化手法の総移動距離と実行時間（都市数 30）

F. 検討考察

得られた結果から考察をする。まず、(1)から(3)に書けて、図で見てもわかるように経路が最適化されていることが分かる。一方で、実行に必要な計算時間は指数関数的に増大していることが分かる。

また、多出発局所探索により得られた解と、Held-Karp 法により得られた解が一致したことから、多出発局所探索により大域的最適解を求めることが出来たことが分かった。一方で、分枝限定法による解法は Held-Karp 法以上に計算時間が必要だということが分かった。

都市数が増加した場合においてもほとんど同じ傾向がみられたが、都市数 25 の場合と 30 の場合で最近隣法において都市数 30 の方が総移動距離が短いという事象が起こった。これについては、都市数 30 のとき偶然最近隣法によりほぼ最適な解が求まっていたため、都市数 25 の場合に比べて総移動距離が短い解が求められたのだと考える。その根拠として、都市数 30 の場合、2-opt 法や多出発局所探索を適応しても大きな改善がみられていないことが挙げられる。

今回の実験では都市数 20,25,30 の場合のみ検証したが、より細かく検証をおこないどのような処理時間の増え方をするかを調べてみても面白そうだと感じた。