計算モデル論演習 II　課題 2

理学部物理情報科学科 3 年　長嶋 城 (20-2202-030-8)
連絡先:a030deu@yamaguchi-u.ac.jp

1　はじめに

この章では, 最適化問題及び巡回セールスマン問題 (TSP) の定義及び定式化を行う.

1.1　最適化問題

最適化問題 (**数理計画問題**)[1][2] とは, 目的達成度を表す指標となる**目的関数** f と, 目的関数で用いられる変数である**決定変数** $\boldsymbol{x}$=$(x_1, x_2, ..., x_n)$ に課された様々な条件を表す**制約条件**を用いて以下のように定式化することができる.

最適化問題の定式化

$$\text{目的関数:} f(\boldsymbol{x}) \rightarrow \text{最小 (あるいは最大)} \tag{1}$$

$$\text{制約条件:} \boldsymbol{x} \in \boldsymbol{S} \tag{2}$$

ただし, 制約条件によって制限された $\boldsymbol{x}$ のとりうる領域を $\boldsymbol{S} \subseteq \mathbb{R}^n$($\mathbb{R}$ は実数の集合) とする. このとき,$\boldsymbol{S}$ を**実行可能領域**といい, 各々の決定変数の値 $\boldsymbol{x} \in \boldsymbol{S}$ を**実行可能解**という.

つまり, 最適化問題とは, 決定変数 $\boldsymbol{x}$ のうち与えられた制約条件をすべて満たし, かつ目的関数 $f(\boldsymbol{x})$ の値が最小または最大になるような $\boldsymbol{x}$ の値をみつける問題といえる.

最適化問題は, 決定変数のとりうる値によって分類することができ, 決定変数が連続値を持つ場合, **連続最適化問題**といい, 一方で決定変数が離散値を持つ場合, **組合せ最適化問題**という. また, 組合せ最適化問題は, 連続最適化問題と区別するために**離散最適化問題**とも呼ばれる.

1.2　巡回セールスマン問題 (TSP)

代表的な組合せ最適化問題の中に, **巡回セールスマン問題** (**TSP**) がある. 巡回セールスマン問題 [1] とは, 都市の集合と 2 つの都市間の移動距離が与えられている時, すべての都市をそれぞれ 1 回のみ通り最初の都市に戻る経路のうち, 総距離が最小になる経路を求める問題のことをいう. つまり, 総距離が最短となる**ハミルトングラフ・ハミルトン閉路** [3] を求める問題と言い換えることもできる. 最適化問題と同様に目的関数 f と制約条件を用いて定式化を行うと以下のようになる.

巡回セールスマン問題 (TSP) の定式化 [1]

都市のグラフ $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ において, 辺 $e_{ij} = (i, j) \in \boldsymbol{E}$ に対する移動距離を d_{ij} とする. また,x_{ij} を e_{ij} が解に含まれるとき 1, そうでないとき 0 の値をとるような $0-1$ 変数とする.

$$\text{目的関数: } \sum_{e_{ij} \in \boldsymbol{E}} d_{ij} x_{ij} \to \text{最小} \tag{3}$$

$$\text{制約条件: } \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \boldsymbol{V} \tag{4}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in \boldsymbol{V} \tag{5}$$

$$\sum_{e_{ij} \in \delta(\boldsymbol{S})} x_{ij} \geq 1 \quad \forall \boldsymbol{S} \in \boldsymbol{V}, \boldsymbol{S} \neq \boldsymbol{0}, \boldsymbol{S} \neq \boldsymbol{V} \tag{6}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in \boldsymbol{V}, i \neq j \tag{7}$$

ただし,$\delta(\boldsymbol{S})$ は $\boldsymbol{S} \in \boldsymbol{V}$ に一方の端点,$\boldsymbol{V} \setminus \boldsymbol{S}$ にもう一方の端点をもつ辺全体の集合とする.

目的関数式 (3) は, 巡回路の総距離を最小にする目的を表している. 制約条件式 (4)(5) は, それぞれ点 i からでる辺は 1 本, 点 j からでる辺は 1 本であるということを表しており, 各都市 1 回のみ訪問するという条件を表している. 制約条件式 (6) は, グラフが連結となるような条件を表しており, 制約条件式 (4)(5)(6) 全体でハミルトングラフの制約条件を表している. 制約条件式 (7) は,x_{ij} が $0-1$ 変数であるということを表している.

TSP では, 全ての都市の数を n としたとき, 考えうるハミルトングラフの合計数は $(n-1)!/2$ となる. したがって, 都市数が増えるにつれて. ハミルトングラフの合計数は膨大な数となってしまい, 全ての解を調べて最適解を得るには非常に時間がかかってしまう. そこで,TSP を現実的に解くには, できるだけ少ない時間でかつ効率的に解を求めるような工夫が必要となってくる.

2　従来法

この章では,TSP の解法の一部である最近近傍法と 2-opt 法について取り上げ, これらの欠点を述べる.

2.1　最近近傍法

最近近傍法 [1][4] とは,TSP の解法の一つであり, **近似解法** (**ヒューリスティクス**) と呼ばれる比較的短時間でそれなりに良い近似解を求める解法に分類される. 具体的には, 以下のような手順で行われる.

最近近傍法の概要

【ステップ 1】 始点となる都市を選ぶ.
【ステップ 2】 未訪問の都市を探す. この時点で未訪問の都市が無ければ, 始点に戻り終了する.
【ステップ 3】 未訪問の都市の内, 現在の都市から最も近い都市を選び移動する.
【ステップ 4】 ステップ 2 に戻る.

つまり, 最近近傍法とは, 未訪問である都市を常にたどる距離がその時点での最小となるように選び続ける方法である. 一見良い解法のように見えるが, 図 1 のように, 探索の最後の方で遠い都市が残されてしまう傾向があり, これが欠点とされている.

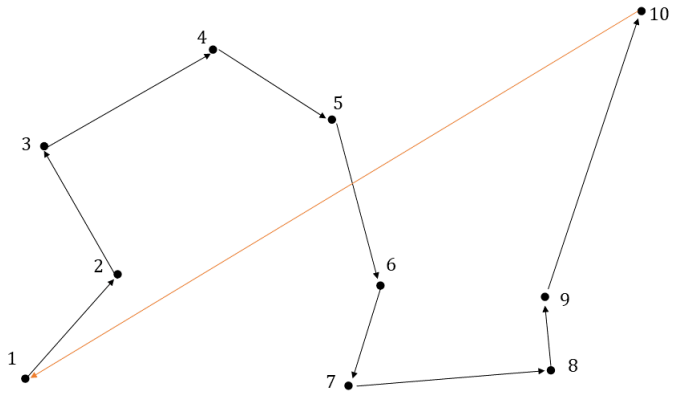

図 1　最近近傍法の欠点

2.2 2-opt 法

2-opt 法 [1][4] とは,TSP の解法の一つであり, 最近近傍法と同様に**近似解法**に分類されるものである. 具体的には, 以下のような手順で行われる.

2-opt 法の概要

訪問する都市の順番を表す配列を p として,t 番目に訪問する都市の番号が $p[t]$ の要素に含まれているとする.
【ステップ 1】 初期解 (経路) を選ぶ. つまり, 訪問するノードが既に格納されたある配列 p を用意する.
【ステップ 2】 i, j 番目に訪問する都市 $p[i], p[j]$ をランダムに選択する. なお隣り合うノードはとらないようにする, すなわち $|i-j|>1$ を条件とする.
【ステップ 3】 もし, 現在の経路 $(p[i], p[i+1])$,$(p[j], p[j+1])$ をそれぞれ経路 $(p[i], p[j])$,$(p[i+1], p[j+1])$ に変更したとき、距離が小さくなるならば、後者の経路を通るように $i+1$ 番目から j 番目までの訪問順番を逆順に変更する. ただし、$i<j$ とする.
【ステップ 4】 一定回数ループする.

【ステップ 3】では, 今現状の経路である i 番目の都市から $i+1$ 番目の都市ではなく,j 番目の都市へ経路を変更することを考えている. もし $i<j$ ならば,i 番目の都市から j 番目の都市へ移った後, 繋がっている $j-1$ 番目の都市へ移りその後は,$j-2, j-3, ..., i+1$ 番目の都市へと移る. つまり元の $i+1$ 番目から j 番目までの訪問順番を逆にたどっていることがわかる.$i+1$ 番目の都市に移ったあとは, 本来 j 番目から移るはずであった $j+1$ 番目の都市に移れば経路変更を終えることができる. 以上から, 経路 $(p[i], p[i+1])$,$(p[j], p[j+1])$ をそれぞれ経路 $(p[i], p[j])$,$(p[i+1], p[j+1])$ に変更するには,$i+1$ 番目から j 番目までの訪問順番を逆順にすればよいということがわかる.

実行可能解 $\boldsymbol{x} \in \boldsymbol{S}$($\boldsymbol{S}$ は実行可能領域) に少しの変形を加えて得られる実行可能解の集合 $\boldsymbol{N}(\boldsymbol{x}) \subset \boldsymbol{S}$ を $\boldsymbol{x}$ の**近傍** [1] と呼ぶ. また,k 個の要素を変化させる近傍のことを $\boldsymbol{k-opt}$ **近傍**と呼び, 今回の 2-opt 法では 2 つの経路が変化した 2-opt 近傍を取り扱っている.

目的関数を最小化する最適化問題において, 以下の条件を満たす実行可能解 $\boldsymbol{x} \in \boldsymbol{S}$ を**大域的最適解** [1] と呼ぶ.

$$f(\boldsymbol{x}) \leq f(\boldsymbol{x}'), \forall \boldsymbol{x}' \in \boldsymbol{S} \tag{8}$$

同様に, 以下の条件を満たす実行可能解 $\boldsymbol{x} \in \boldsymbol{N}$ を**局所的最適解** [1] と呼ぶ.

$$f(\boldsymbol{x}) \leq f(\boldsymbol{x}'), \forall \boldsymbol{x}' \in \boldsymbol{N}(\boldsymbol{x}) \tag{9}$$

2-opt 法は,2-opt 近傍を利用した局所的最適解を得る解法となっており, **局所探索法** [4] とも呼ばれる. したがって,2-opt 法では必ずしも大域的最適解を得られるとは限らないという欠点がある. より精度の高い解を得るためには, 局所的最適解から抜け出し, 大域的最適解を見つける必要がある. 図 2 に横軸を実行可能解 $\boldsymbol{x}$, 縦軸を目的関数の値 $f(\boldsymbol{x})$ とし,2-opt 法の欠点を示す.

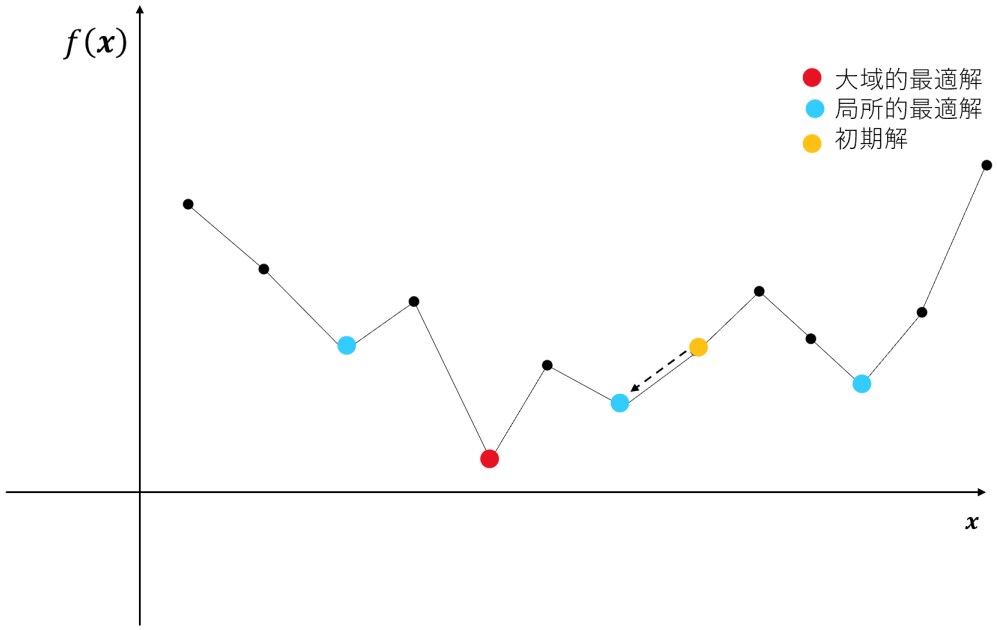

図 2　2-opt 法の欠点

3　用いた手法

従来法では, 比較的短時間でそれなりに良い近似解を求める近似解法取り上げたが, 大域的最適解を得ることが難しいといった欠点があった. そこで, 多少時間をかけてでも, より精度の高い解を求める解法の一つとして, 本実験ではシミュレーテッドアニーリング (SA 法) を使用した. この章では SA 法の説明を行う.

3.1　シミュレーテッドアニーリング (SA 法) の概要

シミュレーテッドアニーリング (SA 法)[1][4] とは,TSP の解法の一つであり, **メタヒューリスティクス**と呼ばれる時間をかけてでも精度の高い解を追求する方法に分類される. 現在の解 $\boldsymbol{x}$ の近傍 $\boldsymbol{N}(\boldsymbol{x})$ 内の各解 $\boldsymbol{x}'$ に, 解のよさに応じた遷移確率 (良い解ほど移行しやすい) を設定し, それに従って次の解を選ぶ. 改悪解でも遷移する確率を与えることによって, 局所的最適解から脱出を図るものである. 遷移確率は, 物理現象の焼きなましからヒントを得ており, 温度と呼ばれるパラメータ $temperature$ により調整される (このことから別名**焼きなまし法**とも呼ばれる). 図 3 に横軸を実行可能解 $\boldsymbol{x}$, 縦軸を目的関数の値 $f(\boldsymbol{x})$ とし,SA 法の探索の様子を示す.

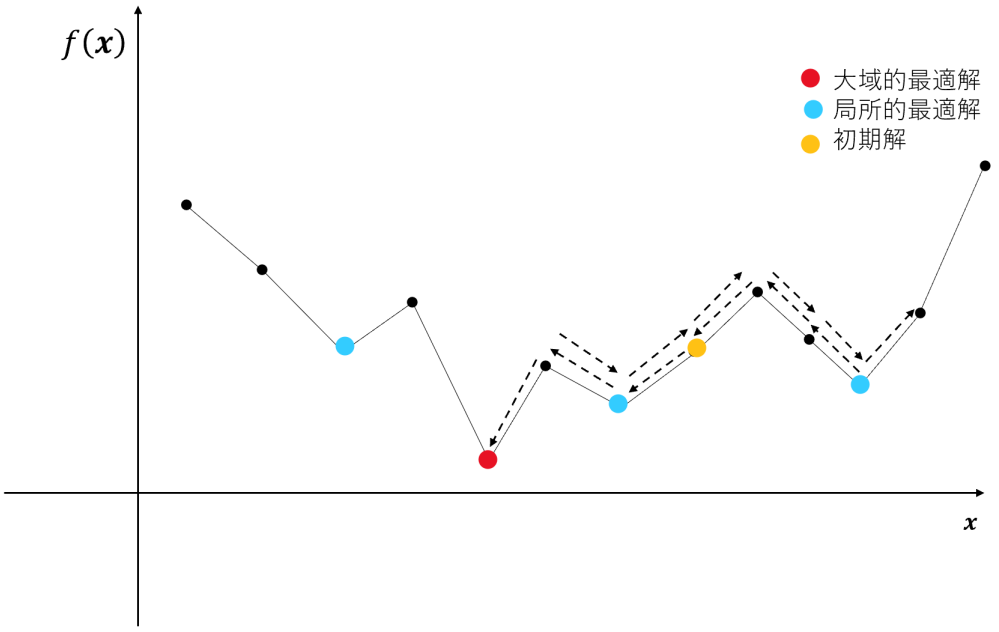

図 3　SA 法の探索の様子

SA 法のアルゴリズム及びフローチャートを以下に示す.

SA 法のアルゴリズム [4]

【ステップ 1】 初期解 $\boldsymbol{x}$ を生成する. また, 初期温度 $temperature$ 及び終了温度 $last$ を定める.
【ステップ 2】 以下の a,b 及び c を, ループの終了条件が満たされるまで反復する.
(a) $\boldsymbol{N}(\boldsymbol{x})$ 内の解をランダムに一つ選び $\boldsymbol{x}'$ とする.
(b) $\Delta := f(\boldsymbol{x}') - f(\boldsymbol{x})$ を計算する ($\boldsymbol{x}'$ が改悪解ならば $\Delta > 0$).
(c) $\Delta \leq 0$ ならば確率 1, そうでなければ確率 $e^{-\Delta/temperature}$ で $\boldsymbol{x} := \boldsymbol{x}'$(解 $\boldsymbol{x}'$ を受理) とする.
【ステップ 3】 温度 $temperature$ を更新する.$temperature < last$ ならば暫定解を出力して探索を終了する. そうでなければ, ステップ 2 に戻る.

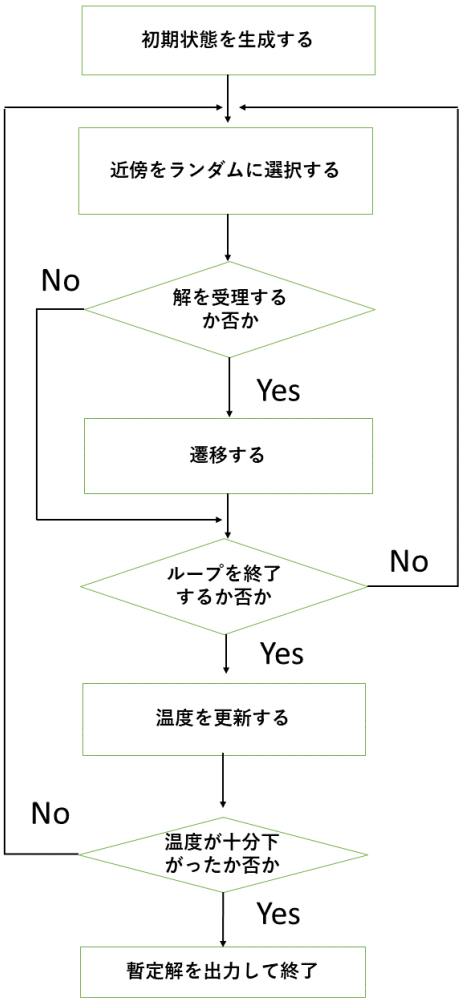

図 4　SA 法のフローチャート [5]

次節では, 各ステップにおける内容を詳しく述べる.

3.2 ステップ 1

【ステップ 1】では, 初期状態を生成する. まず, 初期解 $\boldsymbol{x}$ を生成する. 本実験では初期解として, 最近近傍法で得られた解を一律に採用する. 次に, 初期温度 $temperature$ 及び終了温度 $last$ を決定する. 初期温度及び終了温度のパラメータの決定方法は, 特定の方法が確立されているわけではない [5]. 小西らは次のような設定方法を用いている [6].

- 初期温度：最大の改悪となる推移を 50%の確率で受理する温度とする.
- 終了温度：最小の改悪となる推移がクーリング周期内で最低 1 回は受理される温度とする.

本実験では, 簡単のため, この方法をもとに以下のように温度を設定した.

- 初期温度：最大の改悪となる推移をある確率で受理する温度とする.
- 終了温度：最小の改悪となる推移をある確率で受理する温度とする.

各確率については,5. 予備実験にて決定するものとする.

3.3 ステップ 2

【ステップ 2】では, ループの終了条件が満たされるまで, ある一定温度で解の探索を行う.

まず,$\boldsymbol{N(x)}$ 内の解をランダムに一つ選び $\boldsymbol{x}'$ とする. 本実験では近傍 $\boldsymbol{N}$ として 2-opt 近傍を用いることとする.

次に,$\Delta := \tilde{f}(\boldsymbol{x}') - \tilde{f}(\boldsymbol{x})$ を計算する. ただし, 目的関数 $\tilde{f}(\boldsymbol{x})$ の値は, 都市間の距離が平均距離で正規化された巡回路の総距離とする. つまり, 都市間の経路 (i, j) の距離を d_{ij}, 平均距離を $\bar{d}$ としたとき, 以下のように正規化された距離を用いて巡回路の総距離を求める.

$$\tilde{d_{ij}} := \frac{d_{ij}}{\bar{d}} \tag{10}$$

たとえば, 図 5 のような横軸を実行可能解 $\boldsymbol{x}$, 縦軸を都市間の距離をそのまま使用した巡回路の総距離を表す目的関数の値 $f(\boldsymbol{x})$ とする解の探索空間を考える. この探索空間では凹凸が多く, 局所最適解の数や, これらから脱出するために乗り越えるべき山が多くなってしまい, このような場合には SA 法は適していない [7]. したがって, このようなことを防ぐために探索空間の平滑化 [4] が必要であり, 正規化された距離を用いることで, 図 6 のように探索空間を平滑化することができると考えられる. また,Δ は $\boldsymbol{N}$ に 2-opt 近傍を用いているため, 経路 $(p[i], p[j])$,$(p[i+1], p[j+1])$ の距離の和と, 経路 $(p[i], p[i+1])$,$(p[j], p[j+1])$ の距離の和との差で計算することができる. ここで, 経路の距離は前述通り正規化されたものを使用していることに注意せよ.

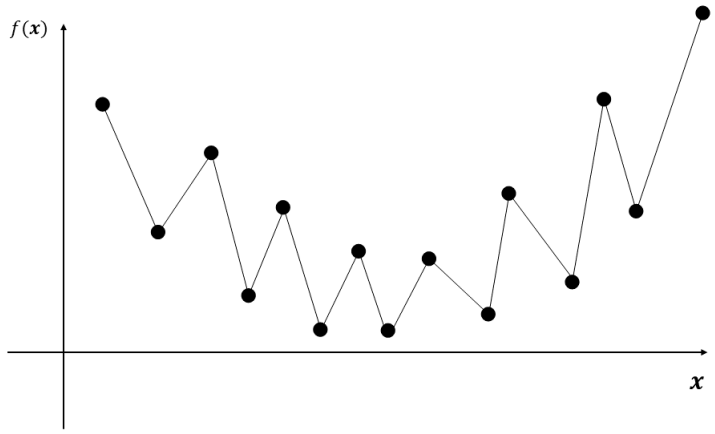

図 5　SA 法に適さない解の探索空間

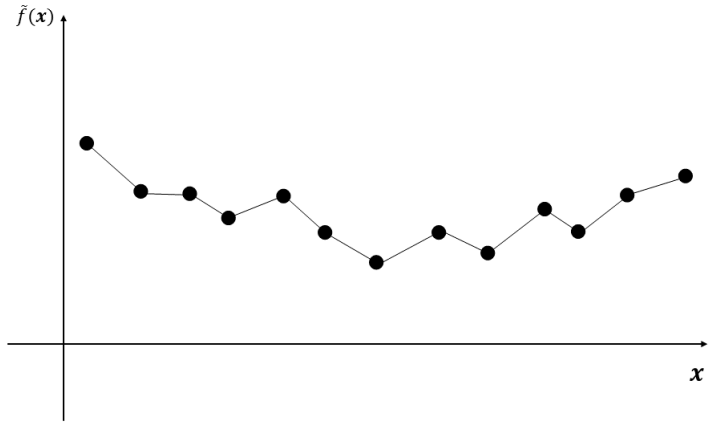

図 6　SA 法に適している解の探索空間

続いて, 解を受理するかどうかを決める. もし,$\Delta \leq 0$ ならば改良解が得られたということなので, 確率 1 で解を受理し, 遷移する. もし,$\Delta > 0$ ならば改悪解ではあるが,$e^{-\Delta/temperature}$ の確率で解を受理し, 遷移する.$temperature$ が大きいほど改悪解を受理する確率が高くなるため, 最初は温度を高くしておき, 徐々に温度を下げていくことで大域的最適解への到達を目指す.
最後にループの終了判定を行う. ループの終了条件の簡単なものとして,$\alpha(> 0)$ をパラメータとして与え, ループを $\alpha \times |(N(x))|$ 回反復したら終了するというものがある [4]. 本実験ではこの終了条件

を用いることとする. なお, 近傍の数 $|(N(x))|$ は, 今回 2-opt 近傍を用いるので, 全都市数を n とすると, 以下のようにして計算できる.

$$|\boldsymbol{N(x)}| = \frac{(n-1)*(n-2)}{2} \tag{11}$$

3.4 ステップ 3

【ステップ 3】では, 温度を更新し, 温度が終了温度よりも低くなれば探索を終了し, 暫定解を出力する. 温度の更新方法の中で, 簡単でかつ実用的なものとして, **幾何冷却法** [4] がある. 幾何冷却法とは, パラメータ $\beta(0 < \beta < 1)$ を用意し, 以下の式で温度の更新を行うものである.

$$temperature := \beta * temperature \tag{12}$$

最適解への収束を保障するためには, 温度の冷却は慎重に行う必要がある [6]. したがって, パラメータ β の値を大きくする必要がある. しかし, 逆に大きくしすぎると, 計算に膨大な時間がかかる. 最適解への収束可能性を犠牲にするか, 時間を犠牲にするかをその都度考え, パラメータ β を調整する必要がある.
もし温度 $temperature$ を更新したとき, 終了温度 $last$ を下回っていれば, 十分に温度が下がったと判断し, 暫定解を出力して終了する.

3.5 考えうるパラメータとその役割

SA 法で考えうるパラメータとして, 以下の 4 つがある.
α: ある一つの温度で反復回数を決めるときに用いられるパラメータである.α を大きくすると単純に探索回数が増えるので, より良い解を得られる可能性が高まるが, 計算時間が増える.
β: 温度の冷却をどれくらいの速さで行うかを決めるパラメータである.β を大きくすると最適解への収束可能性が高まるが, 計算時間が増える.
$temperature$(初期温度):SA 法の初期温度を表すパラメータである. 初期温度を高く設定しておくと, 改悪解を受理する可能性が高まり, 大域的最適解に辿り着く可能性が高まるが, 高すぎると最初の内はランダムな移動が生じやすくなってしまい [4], 無駄な計算時間が増える. 逆に低く設定すると, 改悪解を受理する可能性が低くなってしまい, 本来の目的である局所的最適解から脱出する可能性が低くなってしまう.
$last$(終了温度):SA 法の終了条件となるパラメータである. 最適解を得るためには, 改悪解を受理しないように十分小さい値をとる必要があるが, 低すぎるとこれ以上暫定解を更新することがないにもかかわらず,SA 法を実行してしまい, 無駄な計算時間が増えてしまう.
これらのパラメータの設定は 5. 予備実験にて行う.

4　実験準備

この章では, 先ほどの TSP の解法をプログラム上でどのように構成したかを述べる.

4.1　使用する基本的な関数及び構造体

この節では, プログラムで使用する基本的な関数及び構造体の説明を行う.

4.1.1　都市座標を表す構造体配列

Location

これは, 都市座標の x 座標を変数 double x,y 座標を変数 double y で表した構造体配列である. 図 7 にプログラムを示す.

```
1    /*構造体(座標の表現)*/
2    typedef struct{
3       double x;     /*x 座標*/
4       double y;     /*y 座標*/
5    }Location;
```

図 7　構造体　Location

4.1.2　都市座標の読み込みを行う関数

int __ load __ location()

まず, 関数の引数を示す.
char *fname: 実際に都市座標が書かれているファイル名に対応する. このファイルの都市座標の読み込みを行う.
Location loc[NMAX]: 都市座標を表す構造体配列に対応する. ファイルに書かれた都市座標を構造体配列 loc[NMAX] に保存を行う. ただし,NMAX は関数内で定義されておらず, マクロ定義されている変数であり, 配列の要素数を表すものである. したがって,NMAX を超える数の都市座標の読み込みは行えないことに注意せよ.

次に, 関数で使用する変数を示す.

FILE *fp: ファイルのポインタ変数に対応する.
int n=0: 全体のノード数に対応する. 0 で初期化をしておき,1 つのノードの都市座標を読み込む際にインクリメントすることでノード数も同時に数える.
int i: ファイルの読み込みで使用する変数に対応する.
double x, y: ファイルにある都市座標の x 座標と y 座標の値に対応する.

最後に, 関数の流れについて図 8 を用いて説明を行う.
6 行目から 9 行目でファイルが開けるかどうかの確認を行い, ファイルが開けた場合,11 行目から 15 行目で都市座標の読み込みを行っていく. 前述通り一つのノードの都市座標を読み込む際にノード数 n を計測しておく.16 行目でファイルを閉じ,17 行目で計測したノード数を返り値として出力して終了する.

```
1   /* 都市座標の読み込み(実際のデータを扱う場合)*/
2   int load_location( char *fname, Location loc[] ) {
3     FILE *fp;
4     int   n=0, i;
5     double x, y;
6     if ( (fp=fopen(fname,"r")) == NULL ) {
7       printf("%s not open¥n", fname);
8       exit(0);
9     }
10
11    while(fscanf(fp, "%d%lf%lf", &i, &x, &y)==3) {
12      loc[n].x = x;
13      loc[n].y = y;
14      n++;
15    }
16    fclose(fp);
17    return n;    /*ノード数を返す*/
18  }
```

図 8　int __ load __ location()

4.1.3　ノード間の距離を計算する関数

void calc __ distance()

まず, 関数の引数を示す.
Location loc[]: 既に都市座標が格納された構造体配列に対応する.
double **d: ノード間の距離を保存するポインタ変数に対応する.
int n: 計算する全体のノード数に対応する.

次に, 関数で使用する変数を示す.
int i:i 番目のノードに対応する.
int j:j 番目のノードに対応する.
double INF=10000000: 十分に大きな値である数値に対応する. これは, 関数内では定義されておらず, マクロ定義されている変数である.

最後に, 関数の流れについて図 9 を用いて説明を行う.
6 行目から 13 行目にかけてノード間の距離計算を行っている.10,11 行目では, 対角成分つまり i と j が一致した場合について考えており, 本来であれば 0 となるが, 最小距離を扱う TSP において移動先となる次のノードを選択できなくなるといった不都合が生じるため, 十分に大きな値である INF の代入を行っている.8 行目,9 行目では, それ以外つまり i と j が不一致の場合について考えており, 以下の式 (13) で計算を行っている.

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{13}$$

```
1   /* ノード間の距離を計算する関数 */
2   void calc_distance(Location loc[],double **d,int n){    /*d はポインタ変数*/
3     int i;   /*i 番目のノード*/
4     int j;   /*j 番目のノード*/
5
6     for(i=0;i<n;i++){
7       for(j=0;j<n;j++){
8           if (i!=j)    /*i と j が異なるとき計算式*/
9               d[i][j]=sqrt((loc[i].x-loc[j].x)*(loc[i].x-loc[j].x)+(loc[i].y-loc[j].y)*(loc[i].y-loc[j].y));
10          else
11              d[i][j]=INF;    /*対角成分は十分大きな値とする*/
12      }
13    }
14  }
```

図 9　void calc __ distance()

4.1.4　訪問チェック関数

int is __ int __ path()

まず, 関数の引数を示す.
int p[]: 訪問順配列 p[NMAX] に対応する. 訪問順配列とは, 要素数が NMAX であり,t 番目に訪問したノード p[t] の要素となっている配列である. つまり既に訪問済みであるノードは, 訪問順配列の要素に含まれている.
int s: 訪問済みかどうかを確認したい s 番目のノードに対応する.
int n: 全体のノード数に対応する.

次に, 関数で使用する変数を示す.
int t: 訪問順番を表す変数に対応する.t 番目に訪問したノードは p[t] で表される.
int r=0: 訪問済みか示す変数に対応する. 未訪問であれば 0, 訪問済みであれば 1 をとる.0 で初期化しておき, 訪問済みであった場合のみ 1 に変更する.

最後に, 関数の流れについて図 10 を用いて説明を行う.
5 行目から 10 行目にかけて,s 番目のノードが訪問済みかどうかの確認を行っており, もし訪問済みであれば必ず訪問順配列の要素に含まれるので r=1 として訪問済みと判定し,11 行目で r を返り値として終了する. また, もし訪問順配列の要素に s が含まれていなければ, 未訪問であるので r=0 のまま 11 行目で r を返り値として終了する.

```
1   /*訪問チェック関数*/
2   int is_in_path(int p[],int s,int n){
3       int t;    /*t 番目に訪問する*/
4       int r=0;  /*返却値(0 は未訪問)*/
5       for(t=0;t<n;t++){
6           if (p[t]==s){    /*一致したら 1 を返す(訪問済み)*/
7               r=1;
8               break;
9           }
10      }
11      return r;
12  }
```

図 10　int is __ int __ path()

4.1.5　実行結果後の経路長を表示する関数 (結果表示)

double show __ path()

まず, 関数の引数を示す.
int p[]: 既に解が求まった訪問順配列 p[NMAX] に対応する.
double **d: 既にノード間の距離が求まった距離ポインタ変数に対応する.
int n: 全体のノード数に対応する.

次に, 関数で使用する変数を示す.
int t: 訪問順番を表す変数に対応する.t 番目に訪問したノードは p[t] で表される.
double length=0.0: 暫定の総距離を表す変数に対応する.length=0.0 で初期化しておき, 経路を辿るたびに距離を加算していくことで総距離を求める.

最後に, 関数の流れについて図 11 を用いて説明を行う.
7 行目から 9 行目において,t 番目に訪問したノード p[t] と,t+1 番目に訪問したノード p[t+1] の距離を計算し,length に加算していく. 経路すべての距離を計算し終えれば, 総距離が求まるので,11 行目で最終的な length つまり総距離を出力し, さらに 12 行目で length を返り値として終了する.

```
1   /*実行結果後の経路長を表示する関数(結果表示)*/
2   double show_path(int p[],double **d,int n)
3   {
4     int t;
5     double length=0.0;
6
7     for(t=0;t<n;t++){              /*経路長計算(表示は長くなるのでしない)*/
8       length+=d[p[t]][p[t+1]];     /*p[n]=0 で初期化されているので可*/
9     }
10
11    printf("path length=%f¥n",length);    /*最終的な距離表示*/
12    return length;
13  }
```

図 11　double show __ path()

4.1.6　訪問順をファイルに書き出す関数 (結果保存)

void save __ location()

まず, 関数の引数を示す.
char *fname: 訪問順の結果を保存するファイル名に対応する.
Location loc[]: 既に座標が格納された loc[NMAX] に対応する.
int p[]: 既に求まった訪問順配列 p[NMAX] に対応する.
int n: 全体のノード数に対応する.

次に, 関数で使用する変数を示す.
FILE *fp: ファイルのポインタ変数に対応する.
int t: 訪問順番を表す変数に対応する.t 番目に訪問したノードは p[t] で表される.

最後に, 関数の流れについて図 12 を用いて説明を行う.
5 行目から 8 行目において, ファイルが開けるかどうかの確認を行い, ファイルが開けた場合,9 行目から 12 行目において結果の保存を行う. 結果の保存は,TSP の解が保存された訪問順配列の順番に基づいてノードの都市座標をファイルに書き出していき, 最後に終点つまり始点と同じノードの都市座標を書き出し終了する.

```
1   /*訪問順をファイルに書き出す関数(結果保存)*/
2   void save_location( char *fname, Location loc[], int p[], int n) {
3     FILE *fp;
4     int t;
5     if ( (fp=fopen(fname,"w")) == NULL ) {
6       printf("%s not open¥n", fname);
7       exit(0);
8     }
9     for(t=0; t<n ; t++ )
10      fprintf(fp, "%f %f¥n", loc[p[t]].x, loc[p[t]].y);
11    fprintf(fp, "%f %f¥n", loc[p[0]].x, loc[p[0]].y);
12    fclose(fp);
13  }
```

図 12　void save __ location()

4.1.7　経路を保存 (コンテスト提出用)

void save __ path()

まず, 関数の引数を示す.
char *fname: 訪問順の結果を保存するファイル名に対応する.
Location loc[]: 既に座標が格納された loc[NMAX] に対応する.
int p[]: 既に求まった訪問順配列 p[NMAX] に対応する.

次に, 関数で使用する変数を示す.
FILE *fp: ファイルのポインタ変数に対応する.
int t: 訪問順番を表す変数に対応する.t 番目に訪問したノードは p[t] で表される.

最後に, 関数の流れについて図 13 を用いて説明を行う.
5 行目から 8 行目において, ファイルが開けるかどうかの確認を行い, ファイルが開けた場合,9 行目から 11 行目において結果の保存を行う. 結果の保存は,TSP の解が保存された訪問順配列の順番に基づいてノードの番号をファイルに書き出していき, 最後に終点つまり始点と同じノードの番号を書き出し終了する.

```
1   /* 経路を保存(コンテスト提出用) */
2   void save_path( char *fname, int p[], int n) {
3     FILE *fp;
4     int t;
5     if ( (fp=fopen(fname,"w")) == NULL ) {
6       printf("%s not open¥n", fname);
7       exit(0);
8     }
9     for(t=0; t<n ; t++ )
10      fprintf(fp, "%d ", p[t]);
11    fprintf(fp, "%d¥n", p[0]);
12    fclose(fp);
13  }
```

図 13　void save __ path()

4.2　従来法の実行関数

この節では, 従来法である最近近傍法と 2-opt 法の実行関数の説明を行う. なお,2 章従来法の内容と対応付けを行いながら説明を行う.

4.2.1　最近近傍法の実行関数

void solve __ neighborhoods()

まず, 関数の引数を示す.
int p[]: 訪問順配列 p[NMAX] に対応する. ただし, 引数で受け取る際事前に全て p[NMAX]=0 で初期化されていることとする. したがって, 始点及び終点は 0 番目のノードとなり, これが最近近傍法の【ステップ 1】に対応する. また, 訪問順配列の各要素は, 最近近傍法実行中に順に格納されていくこととなる.
double **d：既に計算済みの距離ポインタ変数に対応する.
int n: 全体のノード数に対応する.

次に, 関数で使用する変数を示す.
int i: 訪問順番を表す変数に対応する.i 番目に訪問したノードは p[i] で表される. このとき, 訪問順番 i 番目までは訪問済みであるとする.
int j:j 番目のノードの番号に対応する.i+1 番目に訪問するノードの候補で, 始点と終点以外のすべてのノードの番号を取りうる.
int k:k 番目のノードの番号に対応する.i+1 番目に訪問するノードの暫定版である. 未訪問で,i 番目に訪問するノードから距離が最小となるようなノードの番号をとる.
double min=INF: 暫定の最小距離に対応する.i 番目から i+1 番目に移る際の最小距離を表す. 最小距離は更新しつつ求めるため,min の初期値はある十分に大きな数 INF としている.

最後に, 関数の流れについて図 14 を用いて説明を行う.
7 行目から 19 行目までが, 主に最近近傍法の実行部分となっている.7 行目で現時点の訪問順番を指定しており, この変数が i である.9 行目で i+1 番目に訪問する候補となる j 番目のノードの内, 現時点で未訪問のノードを探しており, 未訪問の都市を特定している. これが最近近傍法の【ステップ 2】に対応する.10 行目から 12 行目で未訪問のノードの内, 現在の都市から最も近い都市となる k 番目のノードを選んでおり, 確定後 17 行目にて i+1 番目に訪問するノードとして k を格納する. これが最近近傍の【ステップ 3】に対応する. これを繰り返し行うことで最近近傍法の解を得る. これが最近近傍法の【ステップ 4】に対応する.

```
1   /*近傍法*/
2   void solve_neighborhoods(int p[],double **d,int n){
3       int i;    /*i 番目に訪問する*/
4       int j;    /*j 番目のノード(すべてのノード,更に言うと i+1 番目に訪問するノードの候補)*/
5       int k;    /*k 番目(i+1 番目に訪問するノードの確定版)*/
6       double min=INF;    /*暫定の最小距離(後々調整必要)*/
7       for(i=0;i<n-1;i++){    /*n 番目に訪問するノードは既に決まっている(始点)ことに注意*/
8           for(j=1;j<n;j++){
9               if(is_in_path(p,j,n)==0){    /*未訪問*/
10                  if(d[p[i]][j]<min){          /*最短距離更新判定*/
11                      min=d[p[i]][j];    /*暫定の最短距離*/
12                      k=j;    /*暫定の最短距離となるノード*/
13                  }
14              }
15          }
16          /*最短距離確定*/
17          p[i+1]=k;    /*i+1 番目に訪問するノードを格納*/
18          min=INF;    /*次のループに向けて初期化*/
19      }
20  }
```

図 14　void solve __ neighborhoods()

4.2.2　2-opt 法の実行関数

void solve __ two __ opt()

まず, 関数の引数を示す.
int p[]: 訪問順配列 p[NMAX] に対応する. 今回は最近近傍法で求めたものを使用する. これが 2-opt 法の【ステップ 1】に対応し, 初期解として最近近傍法で得られたものを一律に用いる.
double **d: 既に計算済みの距離ポインタ変数に対応する.
int n: 全体のノード数に対応する.

次に, 関数で使用する変数を示す.
int i: 訪問順番を表す変数に対応する.i 番目に訪問したノードは p[i] で表される.
int j: 訪問順番を表す変数に対応する.j 番目に訪問したノードは p[j] で表される.
int t: 現在の 2-opt 法反復回数に対応する. ただし, 本実験では反復回数を一律で 2-opt 法の反復回数を 10000000 としたため,t<10000000 の値を取りうる.
int save: 一時保存用の変数に対応する.j<i 時の i,j 入れ替えに用いる.
int temp: 一時保存用の変数に対応する. 訪問順の入れ替えに用いる.

int k: 訪問順を逆にするのにかかる回数に対応する.
int l: 訪問順を逆にする際に使用する走査用の変数に対応する.

最後に, 関数の流れについて図 15 を用いて説明を行う.
11 行目から 15 行目から i,j 番目に訪問する都市を乱数を用いて $|i-j|>0$ となるようにランダムに選択している. これは 2-opt 法の【ステップ 2】に対応している.17 行目から 21 行目では,【ステップ 3】にむけて,$i<j$ となるように調整を行っている.23 行目では現在の経路 $(p[i],p[i+1])$,$(p[j],p[j+1])$ をそれぞれ経路 $(p[i],p[j])$,$(p[i+1],p[j+1])$ に変更したとき, 距離が小さくなるかどうかを判別している. もし, 経路が小さくなれば,24 行目から 31 行目にかけて i+1 番目から j 番目までの訪問順番を逆順にする. これが 2-opt 法の【ステップ 3】に対応している.32 行目で現在の反復回数 t を計算し,33 行目で反復回数が t=10000000 になったら 2-opt 法を終了する. これが 2-opt 法の【ステップ 4】に対応している.

```
1   /*2-opt 法*/
2   void solve_two_opt(int p[],double **d,int n){
3     int i;    /*i 番目に訪問する*/
4     int j;    /*j 番目に訪問する*/
5     int t=0;    /*現在の 2-opt 法反復回数 t*/
6     int save;    /*一時保存用の変数(j<i 時の i,j 入れ替え)*/
7     int temp;    /*一時保存用の変数(訪問順入れ替え)*/
8     int k;    /*訪問順を逆にするのにかかる反復回数*/
9     int l;    /*走査用*/
10
11    do{
12      do{          /*i と j の決定*/
13        i=rand()%(n-1)+1;
14        j=rand()%(n-1)+1;
15      }while((abs(i-j))<=1);
16
17      if(j<i){    /*i と j の入れ替え(j<i)*/
18        save=i;
19        i=j;
20        j=save;
21      }
22
23      if((d[p[i]][p[j]]+d[p[i+1]][p[j+1]])<(d[p[i]][p[i+1]]+d[p[j]][p[j+1]])){    /*訪問順入れ替え*/
24        k=(j-i+1)/2;    /*k=j-(i+1)+1 に調整用の+1 を加え 2 で割る*/
25        /*i+1 と j の表示をやめる*/
26        for(l=0;l<k;l++){    /*p[i+1]から p[j]までを逆順にする*/
27          temp=p[i+1+l];
28          p[i+l+1]=p[j-l];
29          p[j-l]=temp;
30        }
31      }
32      t++;
33    }while(t<10000000);
34  }
```

図 15　void solve __ two __ opt()

4.3　SA 法で使用する関数

この節では,SA 法を実行する際に必要となる関数の説明を行う. なお,3 章用いた手法の内容と対応付けを行いながら説明を行う.

4.3.1 距離行列を正規化する関数

void calc __ distance2()

まず, 関数の引数を示す.
Location loc[]: 既に都市座標が格納された構造体配列に対応する.
double **d2: ノード間の距離を保存するポインタ変数に対応する.
int n: 計算する全体のノード数に対応する.

次に, 関数で使用する変数を示す.
int i:i 番目のノードに対応する.
int j:j 番目のノードに対応する.
double INF=10000000: 十分に大きな値である数値に対応する. これは, 関数内では定義されておらず, マクロ定義されている変数である.
double sum: 異なるノード間の距離の合計に対応する.
int a=0: 異なるノード間の距離の要素数に対応する.
double ave=0.0: 異なるノード間の距離の平均に対応する.

最後に, 関数の流れについて図 16 を用いて説明を行う.
9 行目から 11 行目は異なるノード間の距離の要素数を計算している. 対角成分は十分に大きな値となっているため無視し,$d_{ij} = d_{ji}$ の関係性を使うと, 要素数 a は以下のように計算できる.

$$a = (n-1)! \tag{14}$$

たとえば, ノード数 n=5 については, 図 17 のように要素数 a は $a = (5-1)!$ で計算できる. 13 行目から 25 行目はノード間の距離を計算する関数 calc __ distance() と同様にノード間の距離の計算を行っているが, 計算の際, 異なるノード間の距離の合計を変数 sum を用いて計算を行っている.26 行目で異なるノード間の距離の平均を計算し,28 行目から 32 行目でその平均距離で正規化を行っている.SA 法でこの距離を使用することで, 解の探索空間の平滑化し, 良い解を得る可能性を高めることができる.

```
1  /*距離行列を正規化する関数*/
2  void calc_distance2(Location loc[],double **d2,int n){    /*d2 はポインタ変数*/
3    int i;   /*i 番目のノード*/
4    int j;   /*j 番目のノード*/
5    double sum=0.0;   /*合計*/
6    int a=0;              /*要素数*/
7    double ave=0.0;   /*平均*/
8
9    for(i=1;i<=n-1;i++){
10     a+=i;
11   }
12
13   for(i=0;i<n;i++){
14     for(j=0;j<n;j++){
15       if (i!=j) {   /*i と j が異なるとき計算式*/
16         d2[i][j]=sqrt((loc[i].x-loc[j].x)*(loc[i].x-loc[j].x)+(loc[i].y-loc[j].y)*(loc[i].y-loc[j].y));
17         if(i<j){
18           sum+=d2[i][j];
19         }
20       }
21       else{
22         d2[i][j]=INF;    /*対角成分は十分大きな値とする*/
23       }
24     }
25   }
26   ave=(double)sum/a;    /*平均計算*/
27
28   for(i=0;i<n;i++){    /*正規化*/
29     for(j=0;j<n;j++){
30       d2[i][j]=d2[i][j]/ave;
31     }
32   }
33 }
```

図 16　void calc __ distance2()

d_{ij}	i=1	i=2	i=3	i=4	i=5
j=1	INF				
j=2		INF			
j=3			INF		
j=4				INF	
j=5					INF

図 17　異なるノード間の距離の要素数

4.3.2　初期温度決定関数 (焼きなまし法で使用)

double calc __ first __ temperature()

まず, 関数の引数を示す.
int p[]: 訪問順配列 p[NMAX] に対応する. 今回は最近近傍法で求めたものを使用する. これが SA 法の【ステップ 1】の初期解 $\boldsymbol{x}$ の生成に対応し, 初期解として最近近傍法で得られたものを一律に用いる.
double **d2: 既にノード間の距離が求まった距離ポインタ変数に対応する. ただし,d2 は関数 void calc __ distance2() で正規化された距離である.
int n: 全体のノード数に対応する.
double **cost: 現在の経路 $(p[i], p[i+1])$,$(p[j], p[j+1])$ をそれぞれ経路 $(p[i], p[j])$,$(p[i+1], p[j+1])$ に変更したときの距離の変化量を表すポインタ変数に対応する. この値が正の場合, 改悪解となることを示しており, さらに値が最大となれば, 最大の改悪解となる.

次に, 関数で使用する変数を示す.
int i: 訪問順番を表す変数に対応する.i 番目に訪問したノードは p[i] で表される.
int j: 訪問順番を表す変数に対応する.j 番目に訪問したノードは p[j] で表される.
double max=0.0: 最大の改悪となるときの距離の変化量に対応する.
double ans=0.0:SA 法で使用する初期温度に対応する.

最後に, 関数の流れについて図 18 を用いて説明を行う.
8 行目から 17 行目は,2-opt 近傍で訪問順番 i,j を選んだ時の解の変化量を計算し, 改悪となりかつ暫定の改悪解よりも悪くなる場合,max の更新を行う. 最大の改悪となるときの距離の変化量 max が求まった後,18 行目で初期温度 ans の計算を行う. 初期温度 ans の計算は, たとえば初期温度を最大

の改悪となる推移を 50 %で受理する温度とすると, 以下のように計算できる.

$$e^{-\frac{max}{ans}} = 0.50 \tag{15}$$

$$-\frac{max}{ans} = log0.50 \tag{16}$$

$$ans = -\frac{max}{log0.50} \tag{17}$$

なお, この初期温度は SA 法の【ステップ 1】の初期温度 $temperature$ に対応する.

```
1   /*初期温度決定関数(焼きなまし法で使用)*/
2   double calc_first_temperature(int p[],double **d2,int n,double **cost){
3       int i;
4       int j;
5       double max=0.0;    /*最大の改悪*/
6       double ans=0.0;    /*使用する初期温度*/
7
8       for(i=1;i<=n-1;i++){    /*2-opt 近傍で i,j を選んだ時の距離の変化配列*/
9           for(j=1;j<=n-1;j++){
10              if((abs(i-j))>1){
11                  cost[i][j]=(d2[p[i]][p[j]]+d2[p[i+1]][p[j+1]])-(d2[p[i]][p[i+1]]+d2[p[j]][p[j+1]]);
12              }
13              if(max<cost[i][j]){
14                  max=cost[i][j];
15              }
16          }
17      }
18      ans=max/0.6931471806;    /*0.6931...は log2 である*/
19
20      return ans;
21  }
```

図 18　calc __ first __ temperature()

4.3.3　終了温度決定関数 (焼きなまし法で使用)

calc __ last __ temperature()

まず, 関数の引数を示す.

int p[]: 訪問順配列 p[NMAX] に対応する. 今回は最近近傍法で求めたものを使用する. これが SA 法の【ステップ 1】の初期解 $\boldsymbol{x}$ の生成に対応し, 初期解として最近近傍法で得られたものを一律に用いる.
double **d2: 既にノード間の距離が求まったポインタ変数 d2 に対応する. ただし,d2 は関数 void calc __ distance2() で正規化された距離である.
int n: 全体のノード数に対応する.
double **cost: 現在の経路 $(p[i], p[i+1])$,$(p[j], p[j+1])$ をそれぞれ経路 $(p[i], p[j])$,$(p[i+1], p[j+1])$ に変更したときの距離の変化量を表すポインタ変数に対応する. この値が正の場合, 改悪解となることを示しており, さらにこのとき最小値となれば, 最小の改悪解となる.

次に, 関数で使用する変数を示す.
int i: 訪問順番を表す変数に対応する.i 番目に訪問したノードは p[i] で表される.
int j: 訪問順番を表す変数に対応する.j 番目に訪問したノードは p[j] で表される.
double min=INF: 最小の改悪となるときの距離の変化量に対応する.
double ans=0.0:SA 法で使用する終了温度に対応する.

最後に, 関数の流れについて図 19 を用いて説明を行う.
8 行目から 17 行目は,2-opt 近傍で訪問順番 i,j を選んだ時の解の変化量を計算し, 改悪となりかつ現在の改悪解よりも良くなる場合,min の更新を行う. 最小の改悪となるときの距離の変化量 min が求まった後,18 行目で終了温度 ans の計算を行う. 終了温度 ans の計算は, たとえば終了温度を最小の改悪となる推移を 1 %で受理する温度とすると, 以下のように計算できる.

$$e^{-\frac{min}{ans}} = 0.01 \tag{18}$$

$$-\frac{min}{ans} = log0.01 \tag{19}$$

$$ans = -\frac{min}{log0.01} \tag{20}$$

なお, この初期温度は SA 法の【ステップ 1】の終了温度 $last$ に対応する.

```
/*終了温度決定関数(焼きなまし法で使用)*/
double calc_last_temperature(int p[],double **d2,int n,double **cost){
    int i;
    int j;
    double min=INF;    /*最小の改悪*/
    double ans=0.0;    /*使用する終了温度*/

    for(i=1;i<=n-1;i++){    /*2-opt 近傍で i,j を選んだ時の距離の変化配列*/
        for(j=1;j<=n-1;j++){
            if((abs(i-j))>1){
                cost[i][j]=(d2[p[i]][p[j]]+d2[p[i+1]][p[j+1]])-(d2[p[i]][p[i+1]]+d2[p[j]][p[j+1]]);
            }
            if(cost[i][j]>0 && min>cost[i][j]){    /*最小の改悪について*/
                min=cost[i][j];
            }
        }
    }
    ans=min/6.214608098;    /*6.214...は log0.002 である*/
    if(ans<0.0001){        /*contest で調整必要。低すぎると時間がかかるため制限*/
        ans=0.0001;
    }

    return ans;
}
```

図 19　calc __ last __ temperature()

4.4　SA 法の実行関数

この節では, 本実験で使用した SA 法の実行関数の説明を行う. なお,3 章用いた手法の内容と対応付けを行いながら説明を行う.

void solve __ SA()

まず, 関数の引数を示す.

int p[]: 訪問順配列 p[NMAX] に対応する. 今回は最近近傍法で求めたものを使用する. これが SA 法の【ステップ 1】の初期解 $\boldsymbol{x}$ の生成に対応し, 初期解として最近近傍法で得られたものを一律に用

いる.
double **d2: 既にノード間の距離が求まった距離ポインタ変数に対応する. ただし,d2 は関数 void calc __ distance2() で正規化された距離である.
int n: 全体のノード数に対応する.
double **cost: 現在の経路 $(p[i], p[i+1])$,$(p[j], p[j+1])$ をそれぞれ経路 $(p[i], p[j])$,$(p[i+1], p[j+1])$ に変更したときの距離の変化量となるポインタ変数に対応する.

次に, 関数で使用する変数を示す.
int i: 訪問順番を表す変数に対応する.i 番目に訪問したノードは p[i] で表される.
int j: 訪問順番を表す変数に対応する.j 番目に訪問したノードは p[j] で表される.
int t: ある温度における現在の SA 法の反復回数に対応する.
double T: ある温度における最大の反復回数に対応する. これは, 関数内では定義されておらず, マクロ定義されている変数である. また, この値はパラメータ α と 2-opt 近傍の数 $|\boldsymbol{N}(\boldsymbol{x})|$ の積で計算されたものである.
int save: 一時保存用の変数に対応する.j<i 時の i,j 入れ替えに用いる.
int temp: 一時保存用の変数に対応する. 訪問順の入れ替えに用いる.
int k: 訪問順を逆にするのにかかる回数に対応する.
int l: 走査用の変数に対応する.
double delta=0:2-opt 近傍と現在の解の距離の差を表す変数に対応する.
double z=0:0 から 1 の数値をとる変数 (乱数) に対応する.
double temperature: 現在の温度を表す変数に対応する. 初期値は初期温度決定関数で得られたものとする.
double best __ d=0.0: 暫定の最適な総距離を表す変数に対応する.
double best __ p[NMAX]=0.0: 暫定の最適な訪問順配列に対応する.
double current __ d=0.0: 現在の総距離を表す変数に対応する.
double last: 終了温度を表す変数に対応する. 初期値は終了温度決定関数で得られたものとする.
double BETA: 温度の更新で用いる式 (12) のパラメータ β に対応する. これは, 関数内では定義されておらず, マクロ定義されている変数である.

最後に, 関数の流れについて図 20〜22 を用いて説明を行う.
18,19 行目は初期温度 *temperature* と終了温度 *last* の決定を行っている. これが SA 法の【ステップ 1】の初期温度と終了温度の決定に対応している.22 行目から 24 行目は暫定解の総距離及び訪問順配列を, 最近近傍法の解としている.27 行目から 39 行目では 2-opt 近傍の解をランダムに選んでおり, これが SA 法の【ステップ 2】(a) に対応している.40 行目では 2-opt 近傍の解と現在の解の距離の差を計算しており, このときの変数 *delta* が SA 法の【ステップ 2】(b) の Δ に対応している.43 行目から 62 行目は変数 *delta* が 0 以下つまり解が良くなった場合についての処理となっており,44 行目から 50 行目で 2-opt 近傍に移るために i+1 番目から j 番目までの訪問順番を逆順にする. これが

SA 法の【ステップ 2】(c) の $\Delta \leq 0$ の時の処理で,2-opt 近傍の解を受理している.51 行目から 54 行目では 2-opt 近傍の解を受理した後の現在の距離を計算し,55 行目から 60 行目で現在の解が暫定解よりも良ければ更新を行う.64 行目から 75 行目は変数 $delta$ が 0 よりも大きいつまり解が悪くなった場合についての処理となっており,64,65 行目で乱数となる変数 z を用いて確率 $e^{-\Delta/temperature}$ で 2-opt 近傍の解を受理するように処理している. これが SA 法の【ステップ 2】(c) の $\Delta > 0$ の時の処理に対応している.76 行目である温度での反復回数を更新し,77 行目である温度での反復回数を T 回に制限している. これが SA 法の【ステップ 2】のループの終了条件に対応する.79 行目では幾何冷却法に基づき温度 $temperature$ を更新し,80 行目で $temperature < last$ つまり温度が十分に下がったか判定し, 条件を満たせば探索を終了する. これが SA 法【ステップ 3】に対応している.

```
1   /*焼きなまし法*/
2   void solve_SA(int p[],double **d2,int n,double **cost){
3     int i;    /*i 番目に訪問する*/
4     int j;    /*j 番目に訪問する*/
5     int t=0;    /*現在の SA 法反復回数 t(ある温度における)*/
6     int save;    /*一時保存用の変数(j<i 時の i,j 入れ替え)*/
7     int temp;    /*一時保存用の変数(訪問順入れ替え)*/
8     int k;    /*訪問順を逆にするのにかかる反復回数*/
9     int l;    /*走査用変数*/
10    double delta=0;    /*変数 Δ*/
11    double z=0;    /*0〜1 をとる乱数*/
12    double temperature;    /*現在の温度*/
13    double best_d=0.0;    /*ベスト総距離*/
14    double best_p[NMAX]={0.0};    /*ベスト訪問順*/
15    double current_d=0.0;    /*現在の総距離*/
16    double last;    /*終了温度*/
17
18    temperature=calc_first_temperature(p,d2,n,cost);    /*初期温度決定*/
19    last=calc_last_temperature(p,d2,n,cost);    /*終了温度決定*/
20    printf("初期温度=%f,終了温度=%f ¥n",temperature,last);    /*温度表示*/
21
22     for(l=0;l<n;l++){                /*初期解を基準とする*/
23       best_d+=d2[p[l]][p[l+1]];
24       best_p[l]=p[l];
25    }
26
27    do{
28       t=0;    /*各温度において反復回数を初期化*/
29       do{
30          do{            /*i と j の決定*/
31                i=rand()%(n-1)+1;
32                j=rand()%(n-1)+1;
33          }while((abs(i-j))<=1);
34
35          if(j<i){    /*i と j の入れ替え(j<i)*/
36                   save=i;
```

図 20　void solve __ SA() 1

```
37                  i=j;
38                  j=save;
39          }
40
41          delta=(d2[p[i]][p[j]]+d2[p[i+1]][p[j+1]])-(d2[p[i]][p[i+1]]+d2[p[j]][p[j+1]]);    /*Δ の計算*/
42
43          if(delta<=0){    /*delta が負ならば確率 1 で解を受理*/
44                  k=(j-i+1)/2;    /*k=j-(i+1)+1 に調整用の+1 を加え 2 で割る*/
45                  /*i+1 と j の表示をやめる*/
46                  for(l=0;l<k;l++){    /*p[i+1]から p[j]までを逆順にする*/
47                      temp=p[i+1+l];
48                      p[i+l+1]=p[j-l];
49                      p[j-l]=temp;
50                  }
51            current_d=0.0;    /*初期化*/
52            for(l=0;l<n;l++){
53                  current_d+=d2[p[l]][p[l+1]];    /*現在の距離*/
54            }
55            if(current_d<best_d){
56                  for(l=0;l<n;l++){
57                      best_p[l]=p[l];    /*ベスト訪問順配列更新*/
58                  }
59                  best_d=current_d;    /*ベスト距離更新*/
60                  printf("!");
61            }
62          }
63
64          else{    /*delta が正ならば確率 exp(-delta/t)で解を受理*/
65                  z=(double)rand()/RAND_MAX;
66                  if(z<=exp(-delta/temperature)){
67                   k=(j-i+1)/2;    /*k=j-(i+1)+1 に調整用の+1 を加え 2 で割る*/
68                    /*i+1 と j の表示をやめる*/
69                   for(l=0;l<k;l++){    /*p[i+1]から p[j]までを逆順にする*/
70                      temp=p[i+1+l];
71                      p[i+l+1]=p[j-l];
72                      p[j-l]=temp;
```

図 21　void solve__SA() 2

```
73                      }
74                    }
75                }
76                t++;
77            }while(t<T);    /*各温度において T 回反復*/
78            printf("%f¥n",temperature);    /*現在の温度表示*/
79            temperature*=BETA;    /*温度を下げる*/
80        }while(temperature>last);    /*十分に温度が下がったら止める(処理が遅ければ調整)*/
81
82        for(l=0;l<n;l++){    /*全ての探索の中で最善の p をとることができる！*/
83            p[l]=best_p[l];
84        }
85    }
```

図 22　void solve __ SA() 3

4.5　メイン関数

この節では, 本実験で使用したメイン関数の説明を行う.

int main(void)

まず, 関数で使用する変数を示す.

int n=0: 全体のノード数に対応する.

int NMAX: 訪問順配列の最大の要素数に対応する. これは, 関数内では定義されておらず, マクロ定義されている変数である.

Location loc[NMAX]: 都市座標を表す構造体配列に対応する.

int p0[NMAX]=0: 最近近傍法の訪問順配列に対応する.

int p1[NMAX]=0:2-opt 法の訪問順配列に対応する.

int p2[NMAX]=0:SA 法の訪問順配列に対応する.

time __ t tp: 時刻を表す変数に対応する. 乱数を用いるときの種として使用し, 図 23 の 11 行目で現在の時刻を取得し,12 行目で乱数の初期化を行う.

int i: 走査用の変数に対応する.

int j: 走査用の変数に対応する.

double current __ 2 __ opt=0.0:2-opt 法で得られた現在の解の総距離に対応する.

double best __ 2 __ opt=INF:2-opt 法で得られた一番良い解の総距離に対応する.

double current __ SA=0.0:SA 法で得られた現在の解の総距離に対応する.

double best __ SA=INF:SA 法で得られた一番良い解の総距離に対応する.

double **d: ノード間の距離を表すポインタ変数に対応する.

double **d2: 平均値で正規化されたノード間の距離を表すポインタ変数に対応する.
double **cost:2-opt 近傍の解と現在の解の総距離の差を表すポインタ変数に対応する.SA 法の温度決定に使用する.
int TRIAL:2-opt 法及び SA 法の実行関数の呼び出し回数を表す変数に対応する. これは, 関数内では定義されておらず, マクロ定義されている変数である.

最後に, 関数の流れについて図 23〜25 を用いて説明を行う.
20 行目で都市座標の読み込みを行い, 計測した全体のノード数を代入する.21 行目から 37 行目でポインタ変数を用いているので, メモリを動的に確保する.39,40 行目でノード間の距離を計算する.42 行目から 47 行目で最近近傍法を実行し, 結果の表示および結果の保存を行う.49 行目から 63 行目で 2-opt 法を最近近傍法で得られた解を初期解とし,TRIAL 回実行し結果の表示を行う.TRIAL 回実行した中で一番良い結果を保存する.65 行目から 79 行目で SA 法を最近近傍法で得られた解を初期解とし,TRIAL 回実行し結果の表示を行う.TRIAL 回実行した中で一番良い結果を保存する.81,82 行目で 2-opt 法および SA 法で得られた一番良い解をそれぞれ改めて表示する.84 行目から 91 行目で動的に確保したポインタ変数のメモリを解放して終了する.

```
1  /*メイン関数*/
2  int main(void)
3  {
4    /*変数生成及び初期化*/
5    int n=0;    /*座標数*/
6    Location loc[NMAX];    /*構造体の配列*/
7    int p0[NMAX]={0};    /*近傍法の訪問順配列*/
8    int p1[NMAX]={0};    /*2-opt 法の訪問順配列*/
9    int p2[NMAX]={0};    /*SA 法の訪問順配列*/
10   time_t tp;    /*時刻の変数*/
11   time(&tp);    /*現在時刻を取得*/
12   srand(tp);    /*乱数の初期化*/
13   int i;    /*走査用の変数*/
14   int j;    /*走査用の変数*/
15   double current_2_opt=0.0;
16   double best_2_opt=INF;
17   double current_SA=0.0;
18   double best_SA=INF;
19
20   n=load_location("city91.loc.txt",loc);    /*contest で調整必要。ファイル名を書いて読み込み*/
21   double **d;    /*距離の二重ポインタ変数*/
22   d=(double**)malloc(sizeof(double*)*n);    /*メモリを動的に確保*/
23   for(i=0;i<n;i++){
24     d[i]=(double*)malloc(sizeof(double)*n);
25   }
26
27   double **d2;    /*正規化した距離*/
28   d2=(double**)malloc(sizeof(double*)*n);    /*メモリを動的に確保*/
29   for(i=0;i<n;i++){
30     d2[i]=(double*)malloc(sizeof(double)*n);
31   }
32
33   double **cost;    /*初期温度決定において使う*/
34   cost=(double**)malloc(sizeof(double*)*n);    /*メモリを動的に確保*/
35   for(i=0;i<n;i++){
36     cost[i]=(double*)malloc(sizeof(double)*n);
```

図 23　int main(void) 1

```
37      }
38
39      calc_distance(loc,d,n);    /*距離計算*/
40      calc_distance2(loc,d2,n);
41
42      printf("neighborhoods¥n");    /*近傍法*/
43      solve_neighborhoods(p0,d,n);
44      show_path(p0,d,n);
45      save_location("neigh.txt",loc,p0,n);
46      save_path("neigh_path.txt",p0,n);
47      putchar('¥n');
48
49      for(i=0;i<TRIAL;i++){    /*2-opt を TRIAL 回実行*/
50        for(j=0;j<NMAX;j++){
51            p1[j]=p0[j];
52        }
53        printf("2-opt¥n");    /*2-opt 法*/
54        solve_two_opt(p1,d,n);
55        current_2_opt=show_path(p1,d,n);
56
57        if(current_2_opt<best_2_opt){    /*2-opt 法でベスト更新*/
58          best_2_opt=current_2_opt;
59          save_location("2-opt_best.txt",loc,p1,n);
60          save_path("2-opt_pathbest.txt",p1,n);
61        }
62        putchar('¥n');
63      }
64
65      for(i=0;i<TRIAL;i++){    /*焼きなましを TRIAL 回実行*/
66        for(j=0;j<NMAX;j++){
67            p2[j]=p0[j];
68        }
69        printf("SA 法¥n");    /*焼きなまし法*/
70        solve_SA(p2,d2,n,cost);
71        current_SA=show_path(p2,d,n);
72
```

図 24　int main(void) 2

```
73          if(current_SA<best_SA){    /*焼きなまし法でベスト更新*/
74            best_SA=current_SA;
75            save_location("SA_best.txt",loc,p2,n);
76            save_path("SA_pathbest.txt",p2,n);
77          }
78          putchar('¥n');
79        }
80
81        printf("2-opt:best_path_length=%f¥n",best_2_opt);    /*ベスト総距離表示*/
82        printf("SA:best_path_length=%f¥n",best_SA);
83
84        for(i=0;i<n;i++){    /*メモリの解放*/
85          free(d[i]);
86          free(d2[i]);
87          free(cost[i]);
88        }
89        free(d);
90        free(d2);
91        free(cost);
92
93        return 0;
94      }
```

図 25　int main(void) 3

5　予備実験

3.5 節では,SA 法で考えうるパラメータについて述べた. これらのパラメータに対してコンテスト本番でどのように数値を与えれば良いか予備実験を行い決定した. この章ではその予備実験の概要について述べる.

5.1　実験の目的

コンテスト本番で使用する SA 法のパラメータである α,β,$temperature$,$last$ を決定する.

5.2　実験の手法

まず, デフォルトのパラメータとして $\alpha = 5$,$\beta = 0.95$,$temperature$ を最大の改悪となる推移を 50 %で受理する温度,$last$ を最小の改悪となる推移を 1 %で受理する温度とした.α や β は一般的に大きな値をとることで長く探索を行うことができ, 良い解を得られる可能性が高まる. また $temperature$ は高く設定することで, 最初の探索で無駄な計算時間が増えてしまうかもしれないが, 局所的最適解から脱出する可能性は高まる. さらに $last$ も低く設定することで, 終盤の探索で改悪解を受理しないようにすることができる. デフォルトのパラメータの数値は以上のことを考慮しつつ決定した.

予備実験では, デフォルトのパラメータの一部を変えて, 対照実験を行った. 対照実験は各パラメータについて行ったため, 大きく 4 つに分類できる.1 つ目は,α に対する実験で,α=1,2,3,4,5 についてそれぞれ 10 個の解を得て比較を行った (それ以外のパラメータ β,$temperature$,$last$ はデフォルトのままである).2 つ目は,β に対する実験で,β=0.75,0.80,0.85,0.90,0.95 についてそれぞれ 10 個の解を得て比較を行った.3 つ目は,$temperature$ に対する実験で,$temperature$ が最大の改悪となる推移を 1,25,50,75 %で受理する温度であるときについてそれぞれ 10 個の解を得て比較を行った.4 つ目は,$last$ に対する実験で,$last$ が最小の改悪となる推移を 1,25,50,75 %で受理する温度であるときについてそれぞれ 10 個の解を得て比較を行った. これらの実験結果を箱ひげ図を用いて示す. ただし, ここでいう解は総距離の値とする. なお, 比較のため 2-opt 法の解も 10 個求め, 同時に結果を表示する.

予備実験で使用した都市座標は, 本番を見据えて小規模都市として city91(課題番号 91) を使用した. 中規模都市, 大規模都市としては TSPLIB[8] からそれぞれ lin318,rat575 を使用した. 以下に結果を示す.

5.3　実験結果

まず, 小規模都市の結果を示す. 縦軸を総距離の値 length として箱ひげ図を用いると,α,β,$temperature$,$last$ の対照実験の結果はそれぞれ図 26〜29 となった (横軸は箱ひげ図なので存在しない).

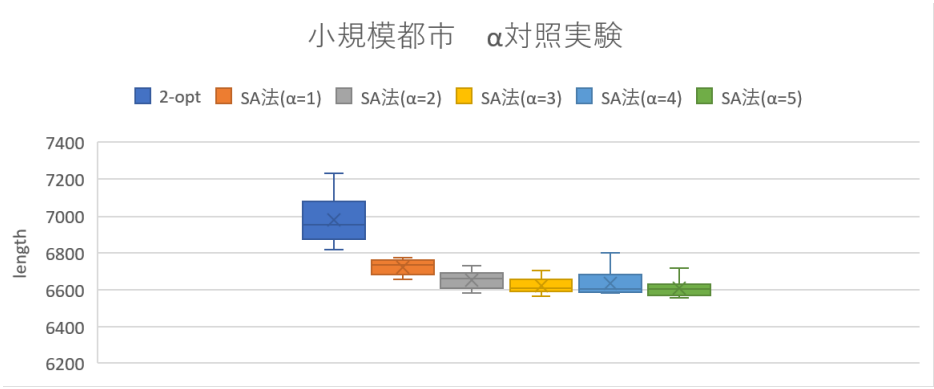

図 26　小規模都市　α 対照実験

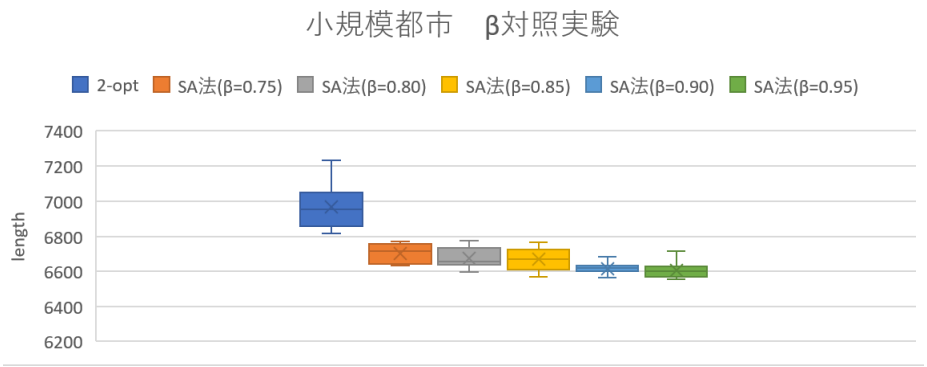

図 27　小規模都市　β 対照実験

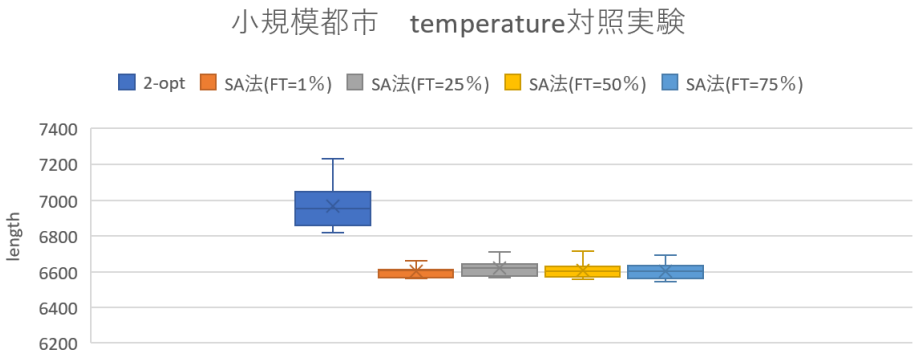

図 28　小規模都市　$temperature$ 対照実験

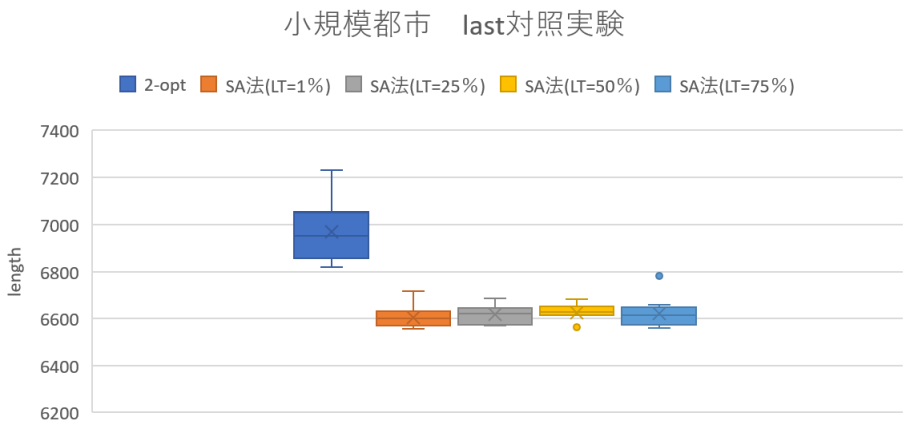

図 29　小規模都市　*last* 対照実験

小規模都市では, 図 26 から α の値が大きくなるにつれて平均値や最小値が低くなる傾向があり, 良い結果が得られることがわかる. 図 27 から β の値が大きくなるにつれて平均値や最小値が低くなる傾向があり, 良い結果が得られることがわかる. 図 28 から 2-opt 法よりも SA 法の解が良くなることはわかるが,*temperature* の値による違いはあまり見られなかった. 図 29 から *last* が最小の改悪となる推移をより低い確率で受理する温度にすると, 平均値や最小値が低くなる傾向があり, 良い結果が得られることがわかる.

続いて中規模都市の結果を示す. 先ほどと同様に縦軸を総距離の値 length として箱ひげ図を用いると,α,β,*temperature*,*last* の対照実験の結果はそれぞれ図 30〜33 となった.

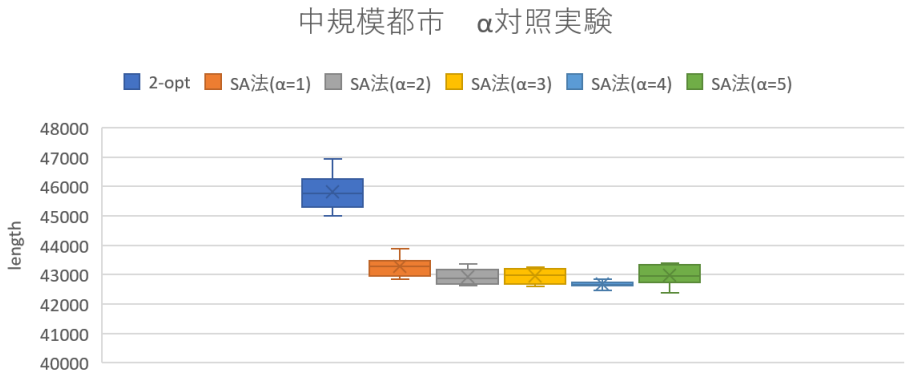

図 30　中規模都市　α 対照実験

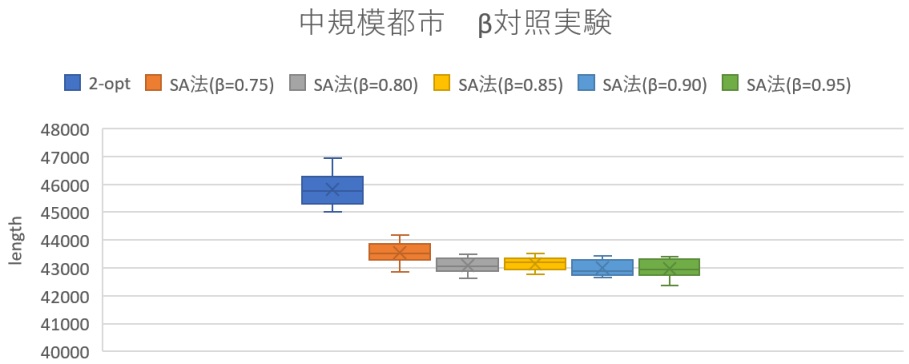

図 31　中規模都市　β 対照実験

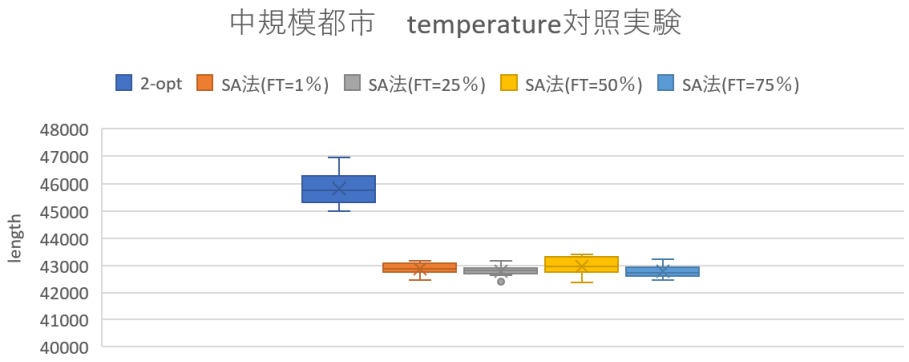

図 32　中規模都市　$temperature$ 対照実験

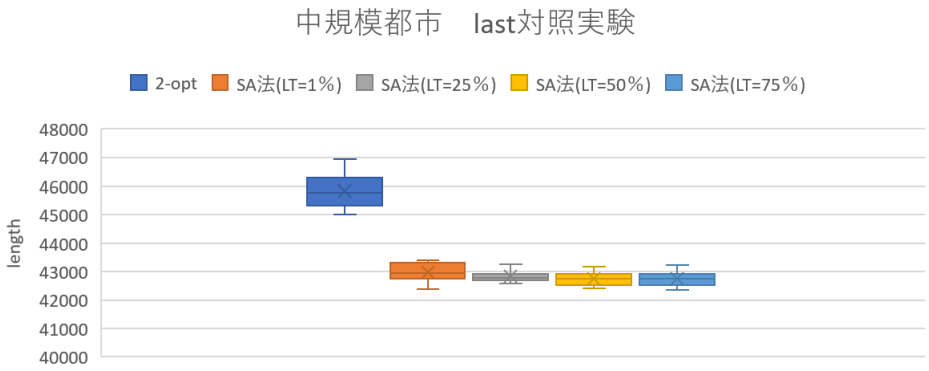

図 33　中規模都市　$last$ 対照実験

中規模都市では, 図 30 から α の値が大きくなるにつれて最小値が低くなる傾向があり, 良い結果が得られることがわかる. 図 31 から β の値が大きくなるにつれて平均値が低くなる傾向があり, 良い結果が得られることがわかる. 図 32 から 2-opt 法よりも SA 法の解が良くなることはわかる

が,$temperature$ の値による違いはあまり見られなかった. 図 33 から 2-opt 法よりも SA 法の解が良くなることはわかるが,$last$ の値による違いはあまり見られなかった.

続いて大規模都市の結果を示す. 先ほどと同様に縦軸を総距離の値 length として箱ひげ図を用いると,α,β,$temperature$,$last$ の対照実験の結果はそれぞれ図 34〜37 となった.

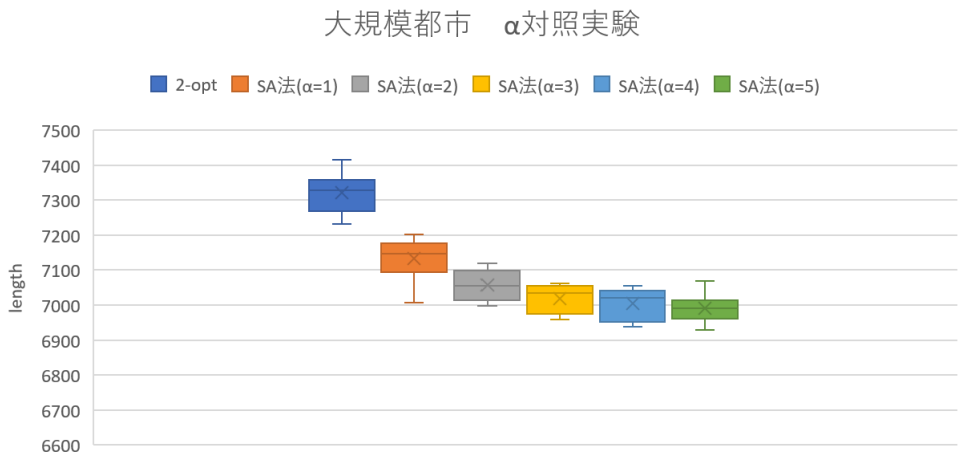

図 34　大規模都市　α 対照実験

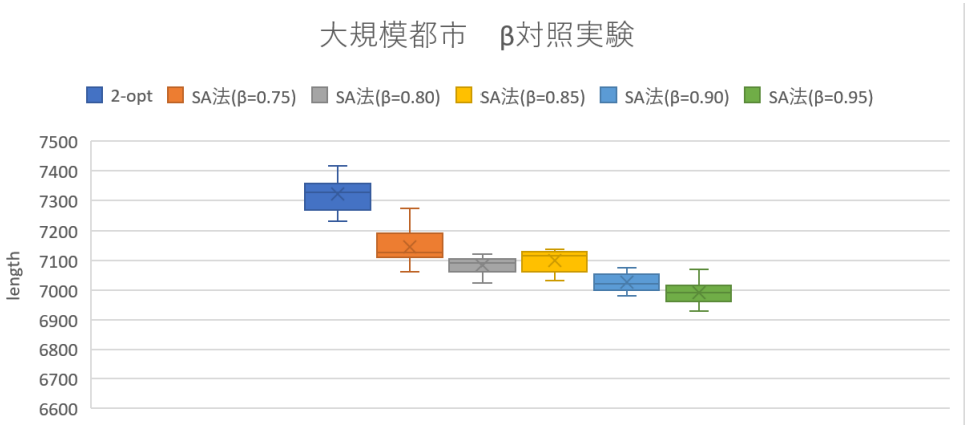

図 35　大規模都市　β 対照実験

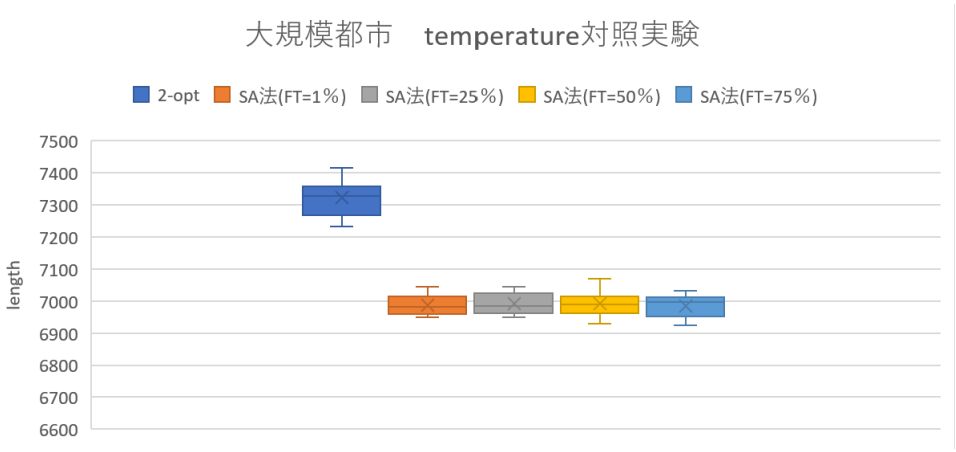

図 36　大規模都市　*temperature* 対照実験

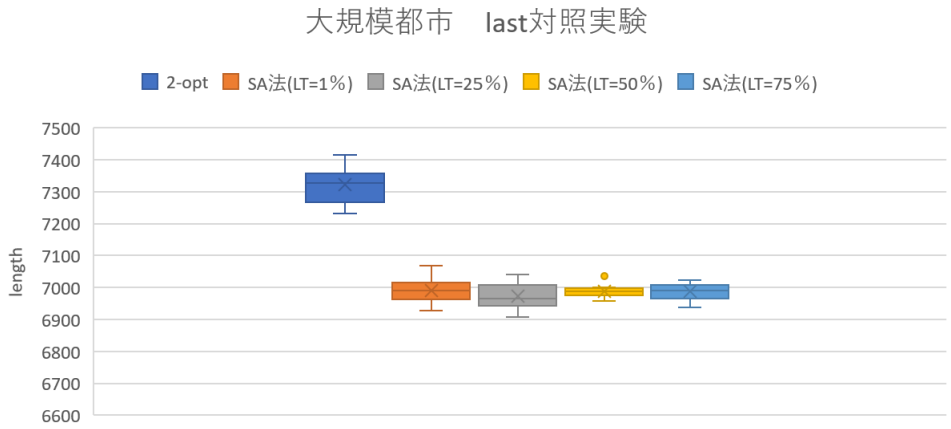

図 37　大規模都市　*last* 対照実験

大規模都市では, 図 34 から α の値が大きくなるにつれて平均値や最小値が低くなる傾向があり, 良い結果が得られることがわかる. 図 35 から β の値が大きくなるにつれて平均値や最小値が低くなる傾向があり, 良い結果が得られることがわかる. 図 36 から 2-opt 法よりも SA 法の解が良くなることはわかるが,*temperature* の値による違いはあまり見られなかった. 図 37 から 2-opt 法よりも SA 法の解が良くなることはわかるが,*last* の値による違いはあまり見られなかった.

全体を通して, 特に α,β による違いが大きく見られ, 両者とも大きく値を設定することで良い解が得られる可能性が高まることがわかった. 逆に,*temperature*,*last* による大きな違いは見られなかったが, 様々な都市配置を想定し,*temperature* は高く,*last* は低くしておく必要があると考えた. この予備実験の結果をもとに, 本番で使用するパラメータを決定した. 次章でそのパラメータについて述べる.

6　実験結果

コンテストで出題された都市に対して, 本番で実際に使用したパラメータで改めて実験を行った. この章ではその実験の概要について述べる.

6.1　実験の目的

コンテストで出題された都市に対して, 本番で実際に使用したパラメータを用いて解の分布を改めて確認する.

6.2　実験の手法

コンテスト本番では, 予備実験の結果をもとに, 以下のようにパラメータを決定した.
小規模都市 (課題番号 20):α=5,β=0.95,$temperature$ を最大の改悪となる推移を 50 %で受理する温度,$last$ を最小の改悪となる推移を 1 %で受理する温度として, ひとまず解を得る. その後,α のみを 10 に変更し, 時間をかけて解を得る.
中規模都市 (課題番号 31):α=5,β=0.95,$temperature$ を最大の改悪となる推移を 50 %で受理する温度,$last$ を最小の改悪となる推移を 1 %で受理する温度として, ひとまず解を得る. その後,α のみを 10 に変更し, 時間をかけて解を得る.
大規模都市 (課題都市 42):α=1,β=0.95,$temperature$ を最大の改悪となる推移を 50 %で受理する温度,$last$ を最小の改悪となる推移を 1 %で受理する温度として, ひとまず解を得る. その後,α のみを 5 に変更し, 時間をかけて解を得る.
予備実験では α を大きくするとより良い解を得られる可能性が高くなったが, 同時に計算時間が増えてしまった. コンテストでは時間制限があったため, ひとまず α を小さく設定してひとまず解を出し, その後より良い解を求めて α を大きく設定して解を出すという流れで行った.

これらのパラメータについて, 解の分布の確認を行う. 横軸を解の個数, 縦軸を総距離 length として折れ線グラフを用いて比較を行った. なお, 比較のため 2-opt 法の結果も同時に表示することとする. 解の個数は, 小規模都市は 50 個, 中規模都市は 30 個, 大規模都市は 10 個求めた.

6.3　実験の結果

まず, 小規模都市の結果を示す. 横軸を解の個数 solution, 縦軸を総距離 length とし,2-opt 法及び SA 法 (α=5,β=0.95,$temperature$=50 %,$last$=1 %),SA 法 (α=10,β=0.95,$temperature$=50 %,$last$=1 %) の結果を折れ線グラフに示すと以下のようになった.

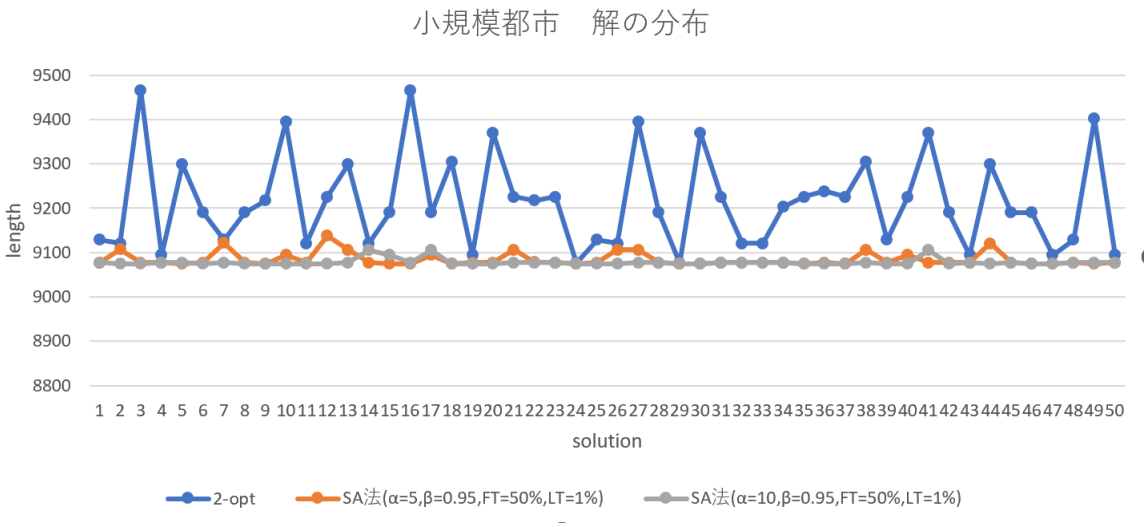

図 38　小規模都市　コンテストの解の分布

図 38 より, 小規模都市では 2-opt 法よりも SA 法の方が良い解を得られたことがわかる. また,SA 法 (α=10,β=0.95,$temperature$=50 %,$last$=1 %) のとき, 一番良い解を得る可能性が高いことがわかる. さらに, 最小値である 9074.148048 となることが複数回あり, これ以上解が良くなることはなかったため, これが大域的最適解だと考えられる.

続いて中規模都市の結果を示す. 先ほどと同様に横軸を解の個数 solution, 縦軸を総距離 length とし,2-opt 法及び SA 法 (α=5,β=0.95,$temperature$=50 %,$last$=1 %),SA 法 (α=10,β=0.95,$temperature$=50 %,$last$=1 %) の結果を折れ線グラフに示すと以下のようになった.

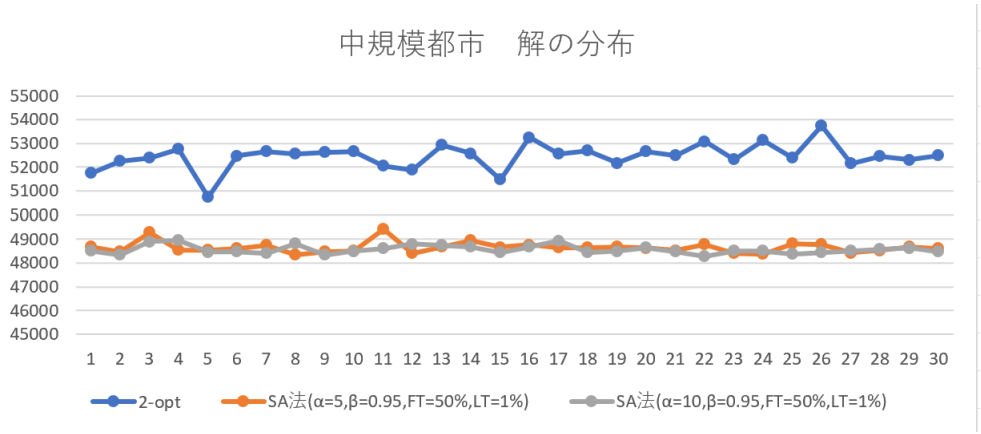

図 39　中規模都市　コンテストの解の分布

図 39 より, 中規模都市では 2-opt 法よりも SA 法の方が良い解を得られたことがわかる. また,2 つの SA 法の結果を見ると, あまり違いがないように見えるが, 実際にそれぞれの平均値および最小値を計算してみると,SA 法 (α=5,β=0.95,$temperature$=50 %,$last$=1 %) では平均値 48653.85248, 最小

値 48360.53911 となり,SA 法 (α=10,β=0.95,$temperature$=50 %,$last$=1 %) では平均値 48564.84193, 最小値 48282.25042 となり, 後者の方が良い結果が得られていることがわかる.

最後に大規模都市の結果を示す. 先ほどと同様に横軸を解の個数 solution, 縦軸を総距離 length とし,2-opt 法及び SA 法 (α=1,β=0.95,$temperature$=50 % ,$last$=1 %),SA 法 (α=5,β=0.95,$temperature$=50 %,$last$=1 %) の結果を折れ線グラフに示すと以下のようになった.

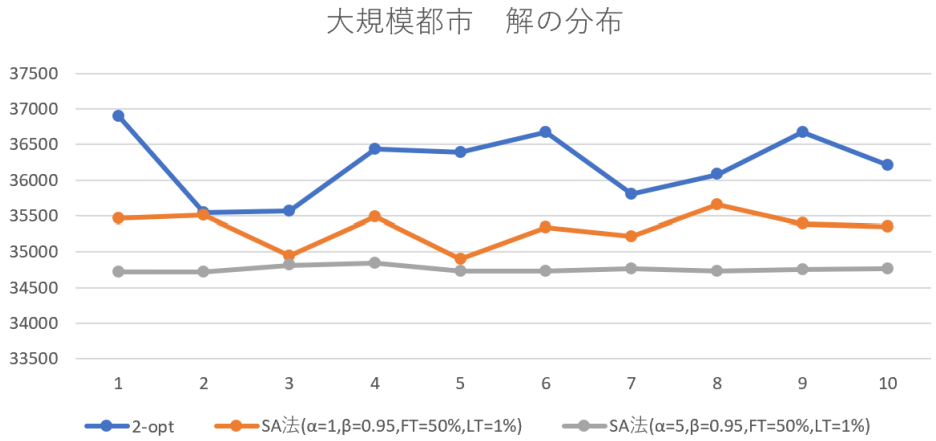

図 40　大規模都市　コンテストの解の分布

図 40 より, 大規模都市では 2-opt 法よりも SA 法の方が良い解を得られたことがわかる. また,SA 法 (α=5,β=0.95,$temperature$=50 %,$last$=1 %) のとき, 一番良い解を得る可能性が高いことがわかる. また, この時の最小値は 34719.09548 であった.

7　まとめ・考察

7.1　まとめ

コンテストの都市データに対して, 解の分布を求めた結果, 一番良い解はそれぞれ小規模都市で length=9074.148048, 中規模都市で length=48282.25042 , 大規模都市で length=34719.09548 となった. また一番良い解を得られたときの解法は全ての都市において SA 法でかつ α のパラメータの大きい方であった. さらに,2-opt 法で得られた解と SA 法で得られた解は大きく異なり, 平均値及び最小値は全て SA 法の方が良くなった.

7.2　考察

実験結果から, 従来法の最近近傍法や 2-opt 法よりも,SA 法の方が良い結果が得られたことがわかった. これは, ある確率で改悪解を許容することで, 局所的最適解から脱出し, 大域的最適解に辿り着く可能性を持たせたことによる結果だと考えられる.

また, パラメータについて, 初期温度 $temperature$ を局所的最適解とならないように高く設定し, かつ終了温度 $last$ を改悪解を受理しないように低く設定した条件下の元で, 探索時間を α や β を大きくすることで長くすると, 良い解が得られる可能性が高まると考えられる. ただし,SA 法ではノード数が大きくなるにつれて計算時間が大幅に増加してしまうため, 時間制約がある場合にはある程度妥協する必要がある. 改善案としては,$temperature$ を局所的最適解とならないギリギリまで低くし, かつ終了温度を $last$ を改悪解を受理しないギリギリまで高くすることで, 無駄な計算時間を削り, その分 α,β を大きくするといったことが考えられる.

参考文献

[1] 今日から使える！組合せ最適化離散問題ガイドブック, 穴井宏和 斎藤努, 講談社,2015

[2] 新版 数理計画入門, 福島雅夫, 朝倉書店,2011

[3] グラフ理論 原書第 4 版,R.J. ウィルソン, 近代科学社,2001

[4] 組合せ最適化 メタ戦略を中心として, 柳浦睦憲 茨木俊秀, 朝倉書店,2001 年

[5] 最良解を基準とする SA の適応的温度スケジュール, 輪湖純也 三木光範 廣安知之, 最適化シンポジウム講演論文集,2004 年

[6] 温度並列シミュレーテッド・アニーリング法とその評価, 小西健三 瀧和男木村宏一, 情報処理学会論文誌,1995 年

[7] シミュレーテッドアニーリング, 喜多一, 日本ファジィ学会誌,1997 年

[8] TSPLIB 「Symmetric traveling salesman problem (TSP)」(http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/)