NP 完全問題の近似解法アルゴリズムについて

目 次

1 はじめに **3**

2 基礎と用語 **3**

2.1 アルゴリズム 3

2.2 P $\neq$ NP 予想 4

3 NP 完全問題の例 **5**

3.1 充足可能性問題 (SAT) 5

3.2 k-彩色問題 5

3.3 ハミルトン閉路問題 5

3.4 巡回セールスマン問題 (TSP) 5

4 巡回セールスマン問題の近似解法 **6**

4.1 巡回セールスマン問題について 6

4.2 シンプルな方法での厳密解法 6

4.3 動的計画法での厳密解法 8

4.4 欲張り法での近似解法 9

4.5 Greedy 法での近似解法 11

4.6 最近挿入法での近似解法 15

4.7 最遠挿入法での近似解法 18

4.8 局所探索法でのさらなる近似 21

4.9 近似解法の精度向上 24

4.10 まとめ 28

5 おわりに **29**

6 参考文献 **29**

1　はじめに

昨今ではコンピュータの発達が著しく，実生活に欠かすことのできないツールとなっている．現代数学の未解決問題である最も重要な問題のひとつであり，コンピュータとも密接に関連している P $\neq$ NP 予想に着目し，その予想の成り立ちの考察と，特に NP 完全問題といわれるものに焦点をあてる．NP 完全問題を近似によって解法を得ることによって，コンピュータの計算アルゴリズムの改善を得ることができ，さらなるコンピュータの発展が見込まれる．

本論文では，NP 完全問題の近似アルゴリズムに焦点を当て，NP 完全問題である巡回セールスマン問題に対する，より良い近似解法を考察することを目的とする．

2　基礎と用語

本節では本論文で使われる主な用語について述べる．

2.1　アルゴリズム

アルゴリズムとは，問題を解くための手順を定式化した形で表現したものをいう．数学での具体例としては，2 つの整数の最大公約数を求めるユークリッドの互除法などが挙げられる．入力を 2 つの整数 n, m，出力を最大公約数 m とし，ユークリッドの互除法のアルゴリズムを記述すると次のようになる．

1. 入力を $m, n (m \geq n)$ とする．
2. $n = 0$ ならば，m を出力しアルゴリズムを終了する．
3. m を n で割った余りを新たに n とし，もとの n を新たに m とし，2. に戻る．

入力に対して，正しく出力することができれば，アルゴリズムとしては様々なものを考えることができる．その中で，良いアルゴリズムと言えるものにはどのようなものがあるか．この判断の材料のひとつとして，計算量（オーダー）とよばれる概念がある．これはアルゴリズムがどの程度の計算回数で終了することができるのか，その量を表現したものである．一般的に計算量は，ランダウの記号で表現される．

例えば，n 個の要素からなる集合 A から m と等しい要素が存在するかを判定するアルゴリズムは次のようになる．

1. $c = 1$ とする．
2. $c > n$ ならば, No と出力しアルゴリズムを終了する．
3. 集合 A の c 番目の要素 a が m と等しければ, Yes と出力しアルゴリズムを終了する．そうでなければ，次のステップに進む．
4. c に 1 を加える. 2.に戻る．

このアルゴリズムは要素が見つからない場合，合計ステップ数は最大となり，n の量が増えるごとに合計ステップ数が増える．この場合の計算量は，$O(n)$ と表現する．

もうひとつの例は，$n \times m$ の行列 X の中から，l と等しい成分が存在するかを判定するアルゴリズムは次のようになる．

1. $b = 1$ とする．

2. $b > m$ ならば, No と出力しアルゴリズムを終了する．

3. $a = 1$ とする．

4. $a > n$ ならば, 7.に進む．

5. 行列 X の (a, b) 成分が m と等しければ, Yes と出力しアルゴリズムを終了する．そうでなければ，次のステップに進む．

6. a に 1 を加える．4.に戻る．

7. b に 1 を加える．2.に戻る．

このアルゴリズムは，n と m の量が増えれば合計ステップ数が増える．しかしさきほどのアルゴリズムよりも増加する割合が大きくなっている．この場合の計算量は，$O(n^2)$ と表現する．

2.2 P $\neq$ NP 予想

P $\neq$ NP 予想とは次の命題である．

予想 1 P $\neq$ NP である．

この命題の P と NP は厳密には次のように定義される．P とは決定性チューリングマシンによって多項式時間（すなわち高々$O(n^c)$;c は定数）で解ける判定問題の集合，NP とは非決定性チューリングマシンによって多項式時間で解ける判定問題の集合である．

本論文ではチューリングマシンの厳密な定義を与えないので，次のように言い直すことにする．

P とは多項式時間で解が求められる問題の集合，NP とは解の候補が与えられたとき，その解が正しい解であるか多項式時間で判定できる問題の集合である．

このとき P の問題であるならば解が多項式時間で求められるので，解の候補が与えられたとき，その解が正しい解であるかを判定することは可能である．すなわち P $\subset$ NP である．しかし現時点では P $\supset$ NP であるか否かは証明されていない．大方の見方は P $\not\supset$ NP であるという予想である．

ここで，この予想を解く上で重要とされるものがあり，それは NP 完全とよばれる．NP 完全とは，NP の部分集合で，NP のすべての問題から多項式時間帰着可能な問題の集合である．すなわち NP 完全の問題のうち 1 つでも P 問題に帰着可能であれば，P = NP であることが証明できるというものである．

3　NP 完全問題の例

本節では NP 完全に属する問題の例について述べる．

3.1　充足可能性問題 (SAT)

充足可能性問題 (SAT) とは次のようなものである．

問題 1 ある和積形論理式が与えられたとき，それに含まれる変数の値を真または偽にうまく定めることによって，和積形論理式を真にすることは可能か．

例えば，$(x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor \neg x_2)$ という命題が与えられれば，$x_1 =$ 真，$x_2 =$ 偽 とすれば，命題は真になる．よって解答は Yes である．例えば $(x_1 \lor x_2) \land (\neg x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor \neg x_2)$ という命題が与えられれば，x_1, x_2 の値に関わらず，命題は偽となるため解答は No である．

また同様の問題で 3-SAT 問題というものも存在する．これも NP 完全であることが知られている．ここで，リテラルとは変数，または $\neg$ 変数のことで，リテラルを $\lor$ で結んだものをクローズとよぶ．また，クローズを並べて $\land$ で結んだ論理式を CNF とよぶ．

問題 2 すべてのクローズが 3 つのリテラルからなる 3-CNF を充足することは可能か．

3.2　k-彩色問題

k - 彩色問題とは次のようなものである．

問題 3 あるグラフが与えられたとき，隣り合う頂点が同じ色にならないように頂点をある決まった $k(\geq 3)$ 色で塗ることは可能か．

3.3　ハミルトン閉路問題

ハミルトン閉路問題とは次のようなものである．

問題 4 あるグラフが与えられたとき，すべての頂点を一度だけ通る閉路が存在するか．

3.4　巡回セールスマン問題 (TSP)

巡回セールスマン問題 (TSP) とは次のようなものである．

問題 5 n 点間の距離が与えられ，全点を与えられた総距離以内の閉路でつなげることは可能か．

巡回セールスマン問題は，複数の都市があり，各都市間に移動コスト（距離など）が与えられたとき，すべての都市を一度づつ訪問する巡回路をコストが最小になるよう求める問題である．本論文ではこの巡回セールスマン問題を取り上げる．

4 巡回セールスマン問題の近似解法

本節では巡回セールスマン問題の近似解法について述べる．

4.1 巡回セールスマン問題について

巡回セールスマン問題とは，前節で紹介したような各頂点を最短コストで巡回路を求める問題である．シンプルな方法は，各巡回路を総当りで求める方法であるが，n 頂点を巡回する総数は $(n-1)!$ 個となり，n が増大するにつれ，現実的な時間で解くことはできない．本節では，解法を求める時間をより短くすることを目的とする．

4.2 シンプルな方法での厳密解法

巡回セールスマン問題を解くためのシンプルな方法は，巡回路を総当りで検索することである．オーダーは，$O((n-1)!)$ となる．以下，シンプルな解法のアルゴリズムを提示する．

```
/* 巡回路を示す頂点番号配列 */
int tour[N];
/* 最短巡回路を示す頂点番号配列 */
int tour_ans[N];
/* 最短距離 */
int length = INT_MAX;

/* ２点間の距離を返す関数 */
int Dis( int vertex1 , int vertex2 );

/* 現在の巡回路の合計コストを返す関数 */
int cost_sum()
{
    int i,sum;
    sum = Dis( N-1 , tour[0] );
    for( i=0 ; i<N-1 ; ++i )
       sum += Dis( tour[i] , tour[i+1] );
    return sum;
}

/* 各巡回路の長さを列挙する（再帰関数） */
void perm( int i )
{
    int j , cost , tmp;
    if( i < N-2 )
    {
        /* 現在の巡回路の長さを求める */
        perm( i+1 );

        /* 順番を入れ替える */
        for( j=i+1 ; j<N-1 ; ++j )
        {
            tmp = tour[i];
            tour[i] = tour[j];
            tour[j] = tmp;

            /* 順番を入れ替えた結果の巡回路の長さを求める */
```

```
                perm( i+1 );

                tmp = tour[i];
                tour[i] = tour[j];
                tour[j] = tmp;
            }
        }
        else
        {
            cost = cost_sum();
            if( cost < length )
            {
                length = cost;
                for( j=0 ; j<N-1 ; ++j )
                    tour_ans[j] = tour[j];
            }
        }
    }

    /* 実際に最短巡回路を求める */
    void enumerate()
    {
        int i;
        for( i=0 ; i<N-1 ; ++i )
            tour[i] = i;

        perm(0);

        printf( "%d" , length );
    }
```

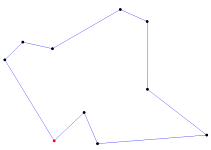

図 1: 頂点数 10 での結果

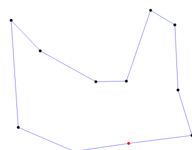

図 2: 頂点数 11 での結果

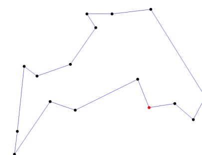

図 3: 頂点数 16 での結果

計測結果は次のとおりである.(試行回数 3 回あたりの平均値)

	頂点数 10	頂点数 11	頂点数 12	頂点数 13
計測時間(秒)	0.697494	16.134175	177.649482	1549.384358

4.3　動的計画法での厳密解法

厳密でシンプルな解法では，現実的な時間で解くことができない．そこで厳密な解法をさらに最適化した形で求めることを考える．ここで動的計画法を用いることで，巡回セールスマン問題の厳密解法をシンプルな解法よりも最適化された形で求めることができる．

頂点全体の集合を V とし，ある始点 $s \in V$ から出発し，頂点の部分集合 $S \subset V$ をすべて経由し，点 $i \in S$ にいたる最短距離を返す関数を $f(i, S)$ とする．$f(i, \{i\}) =$ 距離 d_{si} および $f(j, S \cup \{j\}) = \min_{i \in S} [f(i, S) + d_{ij}]$ と再帰的に定義することにより，$f(s, V)$ が求める最短距離となる．オーダーは，$O(n^2 2^n)$ である．

以下，動的計画法でのアルゴリズムを提示する．

```
void DP()
{
    int i,j,S,tmp;
    int f[n][SMAX];
    for( S=1 ; S<SMAX ; ++S )
        for( i=0 ; i<n ; ++i )
            f[i][S] = 9999;
    for( i=0 ; i<n-1 ; ++i )
        f[i][1<<i] = Dis( n-1 , i );
    for( S=1 ; S<SMAX ; ++s )
    {
        for( i=0 ; i<n ; ++i )
        {
            if( !( (1<<i) & S ) )
                continue;
            for( j=0 ; j<n ; ++j )
            {
                if( (1<<j) & S )
                    continue;
                tmp = f[i][S] + Dis( i , j );
                if( tmp < f[j][(S|(1<<j))] )
                    f[j][(S|(1<<j))] = tmp;
            }
        }
    }
    printf( "%d" , f[n-1][SMAX-1] );
}
```

計測結果は次のとおりである.（試行回数 3 回あたりの平均値）

	頂点数 12	頂点数 13	頂点数 20	頂点数 21	頂点数 22
総当り法/時間（秒）	177.649482	1549.384358	-	-	-
動的計画法/時間（秒）	0.015855	0.024714	4.871877	11.431168	24.097777

4.4　欲張り法での近似解法

ここでは最適でなくとも，ある程度の精度をもった近似解法を求めることを考える．実際に実験によって計測し，時間と精度を比較していく．

もっともシンプルな近似解法として欲張り法が挙げられる．アルゴリズムは以下のとおりである．

1. 適当な都市を選択する．
2. まだ訪問していない最も近い都市へ移動する．
3. 全ての街を訪問したら出発地へ戻る．

この方法のオーダーは $O(n^2)$ となる．

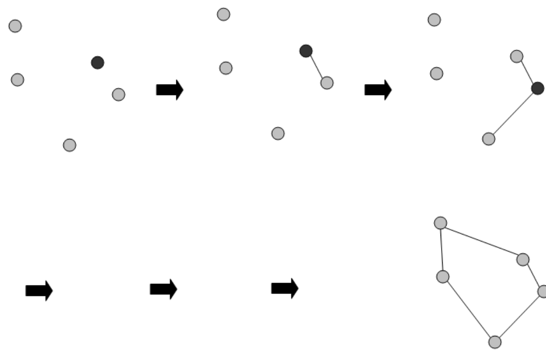

図 4: 欲張り法の生成手順

```
/* 頂点を巡回したかどうかを示すフラグ */
int tour_flag[N];
/* 初期頂点 */
int init;
/* 現在の頂点 */
int now;

void enumerate()
{
    int i,j,small_length,small_tour,len;
    for( i=0 ; i<N-1 ; ++i )
        tour_flag[N] = 0;

    length = 0;
    /* 適当な都市を選択する． */
    init = random() % N;
    now = init;
    tour_flag[init] = 1;
    tour_ans[0] = init;

    for( j=1 ; j<N-1 ; ++j )
```

```
        {
        small_length = INT_MAX;
            for( i=0 ; i<N-1 ; ++i )
            {
                if( tour_flag[i] == 0 )
                {
                    len = Dis( now , i );
                    if( len < small_length )
                    {
                        /* まだ訪問していない最も近い都市へ移動する.  */
                        small_length = len;
                        small_tour = i;
                    }
                }
            }
            tour_ans[j] = small_tour;
            now = small_tour;
            tour_flag[now] = 1;
            length += small_length;
        }

        /* 全ての街を訪問したら出発地へ戻る.  */
        length += Dis( now , init );

        printf( "%d" , length );
    }
```

欲張り法の注意点としては，はじめに選択する都市によって結果が変わる点にある．以下の図は，同じ頂点で別の都市を初期都市とした場合の結果例である．これによって，欲張り法では結果に誤差が出るため，精度として多少の問題を含んでいる．

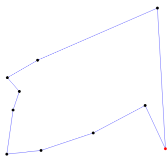

図 5: 欲張り法の結果 1

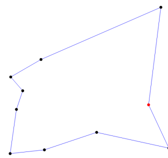

図 6: 欲張り法の結果 2

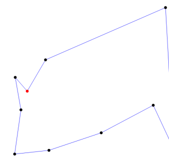

図 7: 欲張り法の結果 3

計測結果は次のとおりである．(試行回数 100 回あたりの平均値/距離計測は同頂点配置による計測)

	頂点数 15	頂点数 16	頂点数 20	頂点数 21	頂点数 22	頂点数 100	頂点数 1000
動的計画法/時間 (秒)	0.087193	0.185583	4.871877	11.431168	24097777	-	-
欲張り法/時間 (秒)	0.008694	0.008677	0.008830	0.009006	0.009150	0.010794	0.039235
動的計画法/総距離誤差	1.0	1.0	1.0	1.0	1.0	-	-
欲張り法/総距離誤差	1.152327	1.142614	1.161720	1.133257	1.084239	-	-

4.5 Greedy 法での近似解法

もうひとつのシンプルな近似解法として Greedy 法が挙げられる．アルゴリズムは以下のとおりである．

1. 各都市間の辺を長さの小さい順にソートする．
2. 辺の長さが小さい順に，[各都市の次数が 2 を超えない][すべての都市を訪問しない巡回路を作らない] 条件で，辺を加えていく．

この方法のオーダーは $O(n^2 \log n)$ となる．先ほどの欲張り法と違って，初期条件が統一されるため総距離数の誤差に偏りがでることはない．

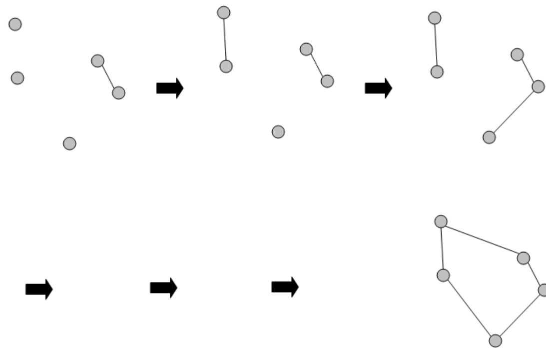

図 8: Greedy 法の生成手順

```
// 辺構造体
class TourLine
{
    public int index1 = 0;
    public int index2 = 0;
    public float length = 0;
}
class TourLineComparer : System.Collections.IComparer
{
    public int Compare(object x, object y)
    {
        if (x == null && y == null)
            return 0;
        if (x == null)
            return -1;
        if (y == null)
            return 1;

        TourLine a = (TourLine)x;
        TourLine b = (TourLine)y;

        if (a.length == b.length)
```

```
            return 0;
        if (a.length < b.length)
            return -1;
        return 1;
    }
}

void enumerate()
{
    System.Collections.ArrayList lines = new System.Collections.ArrayList();
    System.Collections.ArrayList result = new System.Collections.ArrayList();

    // 辺配列を作成
    for (int y = 0; y < tour_point.Length; ++y)
    {
        for (int x = 0; x < tour_point.Length; ++x)
        {
            if (x >= y)
                continue;

            TourLine line = new TourLine();
            line.index1 = x;
            line.index2 = y;
            line.length = tour_point[x].GetLength(tour_point[y]);
            lines.Add(line);
        }
    }

    // ソート
    lines.Sort(new TourLineComparer());

    // 小さい辺から処理する
    foreach (TourLine line in lines)
    {
        // 加える辺でグラフが次数が２以下であるかチェック
        int deg1 = 1;
        int deg2 = 1;
        foreach (TourLine res_line in result)
        {
            if (line.index1 == res_line.index1 || line.index1 == res_line.index2)
                ++deg1;
            if (line.index2 == res_line.index1 || line.index2 == res_line.index2)
                ++deg2;
        }
        if (deg1 > 2 || deg2 > 2)
            continue;

        // 部分閉路ができているかチェック
        // 全ての頂点を通っているなら追加して終了
        int seek_now = line.index1;
        int seek_prev = -1;
        bool loop = false;
        bool all = false;

        int[] flag = new int[tour_point.Length];
        for (int i = 0; i < flag.Length; ++i)
            flag[i] = 0;
```

```
for (; ; )
{
    flag[seek_now]++;
    bool find = false;
    foreach (TourLine res_line in result)
    {
        // つながっている辺を探す
        if (seek_now == res_line.index1 && seek_prev != res_line.index2)
        {
            if (res_line.index2 == line.index2)
            {
                flag[res_line.index2]++;
                loop = true;
            }

            seek_prev = seek_now;
            seek_now = res_line.index2;
            find = true;
            break;
        }
        if (seek_now == res_line.index2 && seek_prev != res_line.index1)
        {
            if (res_line.index1 == line.index2)
            {
                flag[res_line.index1]++;
                loop = true;
            }

            seek_prev = seek_now;
            seek_now = res_line.index1;
            find = true;
            break;
        }
    }
    if (!find)
        break;
    if (loop)
    {
        // 全て通ったかチェック
        bool deg_all_2 = true;
        for (int i = 0; i < flag.Length; ++i)
        {
            if (flag[i] != 1)
            {
                deg_all_2 = false;
                break;
            }
        }
        if (deg_all_2)
        {
            all = true;
        }

        break;
    }
}

if (loop && !all)
```

```
                continue;

            result.Add(line);

            if (all)
                break;
        }

        // 結果の順路をめぐる
        {
            int seek = 0;
            int seek_now = 0;
            int seek_prev = -1;

            for (; ; )
            {
                bool find = false;
                foreach (TourLine res_line in result)
                {
                    // つながっている辺を探す
                    if (seek_now == res_line.index1 && seek_prev != res_line.index2)
                    {
                        tour_root[seek++] = seek_now;
                        seek_prev = seek_now;
                        seek_now = res_line.index2;
                        find = true;
                        break;
                    }
                    if (seek_now == res_line.index2 && seek_prev != res_line.index1)
                    {
                        tour_root[seek++] = seek_now;
                        seek_prev = seek_now;
                        seek_now = res_line.index1;
                        find = true;
                        break;
                    }
                }
                if (seek >= tour_root.Length)
                    break;
                if (!find)
                    break;
            }
        }
    }
```

計測結果は次のとおりである.（試行回数 100 回あたりの平均値/距離計測は同頂点配置による計測）

	頂点数 15	頂点数 16	頂点数 20	頂点数 21	頂点数 22	頂点数 100	頂点数 1000
動的計画法/時間 (秒)	0.087193	0.185583	4.871877	11.431168	24.097777	-	-
Greedy 法/時間 (秒)	0.009960	0.010025	0.011521	0.008942	0.009011	0.023044	8.091780
動的計画法/総距離誤差	1.0	1.0	1.0	1.0	1.0	-	-
Greedy 法/総距離誤差	1.130186	1.127949	1.142909	1.150388	1.084239	-	-

4.6 最近挿入法での近似解法

ここで少しアプローチの仕方を変えてみる．巡回セールスマン問題の解の巡回路が，おおよそ外側を通る大きな巡回路となることに着目し，アルゴリズムを考える．ひとつ目は最近挿入法とよばれるアルゴリズムである．これは小さな巡回路から徐々に広げていく手法である．

アルゴリズムは以下のとおりである．

1. 適当な都市を選択する．
2. まだ訪問していない最も近い都市と連結し，部分巡回路 T を作る．
3. 部分巡回路 T からの距離が最小となる都市 i を選択する．
4. i との距離が最小の都市 j と隣接している都市 k と l を考え，距離 $\mathrm{d}(i,k)+\mathrm{d}(i,j)-\mathrm{d}(k,j)$ と距離 $\mathrm{d}(i,l)+\mathrm{d}(i,j)-\mathrm{d}(l,j)$ を比較し，距離が小さい都市と都市 i を接続し，都市 j との接続を外す．
5. まだ訪問していない都市が存在しなくなるまで繰り返す．

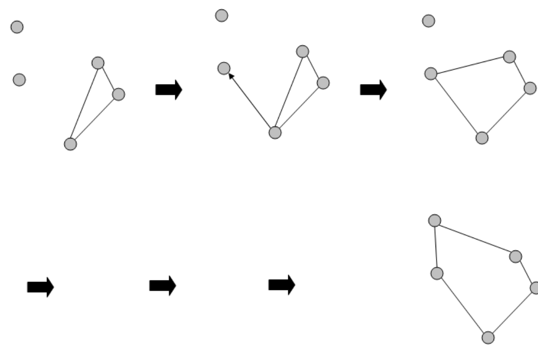

図 9: 最近挿入法の生成手順

```
// 初期頂点を選択する
int init = rand.Next(0, tour_point.Length - 1);

// 初期巡回路
List<int> loop_list = new List<int>();
// 残りの頂点
List<int> rem_list = new List<int>();

loop_list.Add(init);
for (int i = 0; i < tour_point.Length; ++i)
{
    if (i != init)
        rem_list.Add(i);
}
```

```
// 巡回路を拡張する
while (loop_list.Count < tour_root.Length)
{
    // 部分巡回路から最短距離の点を探す
    float min_length = float.MaxValue;
    int min_vertex_from = -1;   // 巡回路の頂点
    int min_vertex_to = -1;     // 残りの頂点
    for (int i = 0; i < loop_list.Count; ++i)
    {
        for (int j = 0; j < rem_list.Count; ++j)
        {
            float len = tour_point[loop_list[i]].GetLength(
                                tour_point[rem_list[j]]);
            if (len < min_length)
            {
                min_vertex_from = i;
                min_vertex_to = j;
                min_length = len;
            }
        }
    }

    // 巡回路に頂点を加え、再度巡回路を生成
    if (loop_list.Count == 1)
    {
        loop_list.Add(rem_list[min_vertex_to]);
        rem_list.RemoveAt(min_vertex_to);
    }
    else
    {
        int index1 = min_vertex_from - 1;
        int index2 = min_vertex_from + 1;
        if (index1 < 0)
            index1 = loop_list.Count - 1;
        if (index2 >= loop_list.Count)
            index2 = 0;

        if (tour_point[loop_list[index1]].GetLength(
                tour_point[rem_list[min_vertex_to]])
            < tour_point[loop_list[index1]].GetLength(
                tour_point[rem_list[min_vertex_to]]))
        {
            loop_list.Insert(index1,rem_list[min_vertex_to]);
        }
        else
        {
            loop_list.Insert(index2, rem_list[min_vertex_to]);
        }

        rem_list.RemoveAt(min_vertex_to);
    }
}

// 結果を挿入する
for (int i = 0; i < loop_list.Count; ++i)
    tour_root[i] = loop_list[i];
```

計測結果は次のとおりである.（試行回数 100 回あたりの平均値/距離計測は同頂点配置による計測）

	頂点数 15	頂点数 16	頂点数 20	頂点数 21	頂点数 22	頂点数 100	頂点数 1000
動的計画法/時間 (秒)	0.087193	0.185583	4.871877	11.431168	24.097777	-	-
最近挿入法/時間 (秒)	0.010182	0.010590	0.011145	0.008803	0.012261	0.016766	3.886422
動的計画法/総距離誤差	1.0	1.0	1.0	1.0	1.0	-	-
最近挿入法/総距離誤差	1.277413	1.256310	1.290854	1.263411	1.266207	-	-

4.7 最遠挿入法での近似解法

先ほどのアルゴリズムを少し改変してみる．最近挿入法で，部分巡回路からまだ訪問していない都市を選択する方法を変更し，最も遠い都市を選択するというものである．これは最遠挿入法とよばれるアルゴリズムである．

アルゴリズムは以下のとおりである．

1. 適当な都市を選択する．
2. まだ訪問していない最も遠い都市と連結し，部分巡回路 T を作る．
3. 部分巡回路 T からの距離が最小となる都市 i を選択する．
4. i との距離が最小の都市 j と隣接している都市 k と l を考え，距離 $\mathrm{d}(i,k)+\mathrm{d}(i,j)-\mathrm{d}(k,j)$ と距離 $\mathrm{d}(i,l)+\mathrm{d}(i,j)-\mathrm{d}(l,j)$ を比較し，距離が小さい都市と都市 i を接続し，都市 j との接続を外す．
5. まだ訪問していない都市が存在しなくなるまで繰り返す．

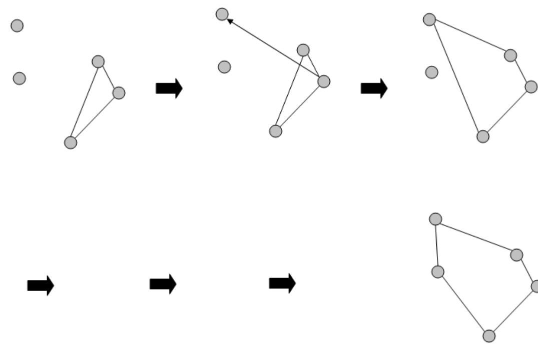

図 10: 最遠挿入法の生成手順

```
// 初期頂点を選択する
int init = rand.Next(0, tour_point.Length - 1);

// 初期巡回路
List<int> loop_list = new List<int>();
// 残りの頂点
List<int> rem_list = new List<int>();

loop_list.Add(init);
for (int i = 0; i < tour_point.Length; ++i)
{
    if (i != init)
        rem_list.Add(i);
}
```

```
// 巡回路を拡張する
while (loop_list.Count < tour_root.Length)
{
    // 部分巡回路から最長距離の点を探す
    float max_length = float.MinValue;
    int max_vertex_from = -1;   // 巡回路の頂点
    int max_vertex_to = -1;     // 残りの頂点
    for (int i = 0; i < loop_list.Count; ++i)
    {
        for (int j = 0; j < rem_list.Count; ++j)
        {
            float len = tour_point[loop_list[i]].GetLength(
                tour_point[rem_list[j]]);
            if (len > max_length)
            {
                max_vertex_from = i;
                max_vertex_to = j;
                max_length = len;
            }
        }
    }

    // 巡回路に頂点を加え、再度巡回路を生成
    if (loop_list.Count == 1)
    {
        loop_list.Add(rem_list[min_vertex_to]);
        rem_list.RemoveAt(min_vertex_to);
    }
    else
    {
        int index1 = min_vertex_from - 1;
        int index2 = min_vertex_from + 1;
        if (index1 < 0)
            index1 = loop_list.Count - 1;
        if (index2 >= loop_list.Count)
            index2 = 0;

        if (tour_point[loop_list[index1]].GetLength(
                tour_point[rem_list[min_vertex_to]])
            < tour_point[loop_list[index1]].GetLength(
                tour_point[rem_list[min_vertex_to]]))
        {
            loop_list.Insert(index1,rem_list[min_vertex_to]);
        }
        else
        {
            loop_list.Insert(index2, rem_list[min_vertex_to]);
        }

        rem_list.RemoveAt(min_vertex_to);
    }
}

// 結果を挿入する
for (int i = 0; i < loop_list.Count; ++i)
    tour_root[i] = loop_list[i];
```

計測結果は次のとおりである.（試行回数 100 回あたりの平均値/距離計測は同頂点配置による計測）

	頂点数 15	頂点数 16	頂点数 20	頂点数 21	頂点数 22	頂点数 100	頂点数 1000
動的計画法/時間 (秒)	0.087193	0.185583	4.871877	11.431168	24.097777	-	-
最遠挿入法/時間 (秒)	0.011351	0.010980	0.012624	0.010454	0.012143	0.018683	3.893259
動的計画法/総距離誤差	1.0	1.0	1.0	1.0	1.0	-	-
最遠挿入法/総距離誤差	2.046981	2.080959	2.165671	2.123031	1.928646	-	-

4.8 局所探索法でのさらなる近似

ここでは，今まで見てきた近似解法の改良を考える．様々な方法で，近似の解を求めてきたわけだが，その解に若干の修正を加えることを考える．解をひとつの要素として考え，解集合を探索していく．これは局所探索法とよばれる．具体的には次のアルゴリズムで考える．

1. 初期解 x を生成する．
2. 解 x の近傍解を生成し，解 x よりも改善されている解を選択する．
3. 近傍内に改善解が存在しなくなるまで繰り返す．

ここで，近傍とは，解 x に微小な変更を加えて得られる解集合 $N(x)$ である．

この方法は,「よい解同士は似通った構造を持っている」ということを念頭においた手法で，与えられた解を少しずつ改善することができる．

さて近傍解を生成するための手法だが，ここでは 2-opt 法とよばれるアルゴリズムで改善解を探す．具体的には次のアルゴリズムである．

1. 適当な都市 a を選び，巡回路で a の次の都市を b とする．
2. 都市 a から最も近い都市で，b 以外の都市を c とし，巡回路で c の次の都市を d とする．
3. 距離 $\mathrm{d}(a,b)+\mathrm{d}(c,d) > \mathrm{d}(a,c)+\mathrm{d}(b,d)$ であれば，辺 ab，cd を ac，bd に入れ替えて改善する．

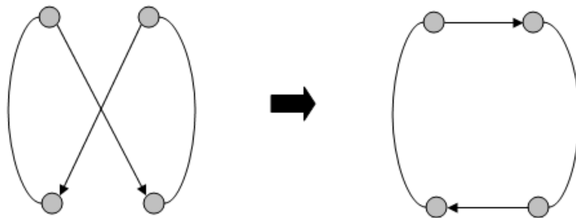

図 11: 2-opt 法の最適手順

2-opt 法を用いて，局所探索法の初期解を今まで見てきたアルゴリズムを用いて考察する．

```
// 2-opt 法
for (int z=0; z<tour_point.Length*10 ;++z )
{
    bool find = false;

    for (int i = 0; i < tour_root.Length+1; ++i)
    {
        // 適当な都市を決める
        int a=0, b=0, c=0, d=0;
        a = i % tour_root.Length;
        b = (i+1) % tour_root.Length;
```

```
        // 近傍都市を求める
        float min_len = float.MaxValue;
        for (int j = 0; j < tour_root.Length; ++j)
        {
            if (a == j || b == j)
                continue;
            float len = tour_point[ tour_root[a] ].GetLength(
                     tour_point[ tour_root[j] ] );
            if( len < min_len )
            {
                c = j;
                min_len = len;
            }
        }
        d = (c+1) % tour_root.Length;

        // 距離を判定する
        int a_idx = tour_root[a];
        int b_idx = tour_root[b];
        int c_idx = tour_root[c];
        int d_idx = tour_root[d];
        if (tour_point[a_idx].GetLength(tour_point[c_idx])
             + tour_point[b_idx].GetLength(tour_point[d_idx])
            < tour_point[a_idx].GetLength(tour_point[b_idx])
                 + tour_point[c_idx].GetLength(tour_point[d_idx]))
        {
            // 置き換える
            find = true;

            float prev_length = GetLength();
            int[] prev_root = new int[tour_root.Length];
            for (int n = 0; n < tour_root.Length; ++n)
                prev_root[n] = tour_root[n];

            tour_root[b] = c_idx;
            tour_root[c] = b_idx;

            float now_length = GetLength();

            if (prev_length < now_length)
            {
                // 改悪ならば戻す
                for (int n = 0; n < tour_root.Length; ++n)
                    tour_root[n] = prev_root[n];
            }
        }
    }
```

計測結果は次のとおりである.（試行回数 100 回あたりの平均値/距離計測は同頂点配置による計測/カッコ内の数値は総距離数の平均を示す）

	頂点数 15	頂点数 16	頂点数 20	頂点数 21	頂点数 22	頂点数 100	頂点数 1000
動的計画法/時間 (秒)	0.087193	0.185583	4.871877	11.431168	24.097777	-	-
（ノーマル）							
欲張り法/時間 (秒)	0.008694	0.008677	0.008830	0.009006	0.009150	0.010794	0.039235
Greedy 法/時間 (秒)	0.009960	0.010025	0.011521	0.008942	0.009011	0.023044	8.091780
最近挿入法/時間 (秒)	0.010182	0.010590	0.011145	0.008803	0.012261	0.016766	3.886422
最遠挿入法/時間 (秒)	0.011351	0.010980	0.012624	0.010454	0.012143	0.018683	3.893259
（局所探索法）							
欲張り法/時間 (秒)	0.010216	0.011504	0.013194	0.011098	0.010728	0.258100	240.658416
Greedy 法/時間 (秒)	0.010978	0.010104	0.012391	0.010050	0.010357	0.269266	236.259804
最近挿入法/時間 (秒)	0.011783	0.012176	0.011521	0.010242	0.009146	0.254373	241.466804
最遠挿入法/時間 (秒)	0.011612	0.012444	0.014705	0.012149	0.011447	0.304086	295.540890
動的計画法/総距離誤差	1.0	1.0	1.0	1.0	1.0	-	-
（ノーマル）							
欲張り法/総距離誤差	1.152327	1.142614	1.161720	1.133257	1.084239	(6390.634000)	-
Greedy 法/総距離誤差	1.130186	1.127949	1.142909	1.150388	1.084239	(6026.531000)	-
最近挿入法/総距離誤差	1.277413	1.256310	1.290854	1.263411	1.266207	(7065.230000)	-
最遠挿入法/総距離誤差	2.046981	2.080959	2.165671	2.123031	1.928646	(14589.070000)	-
（局所探索法）							
欲張り法/総距離誤差	1.142420	1.128432	1.150661	1.107664	1.075397	(6347.113000)	-
Greedy 法/総距離誤差	1.124755	1.121975	1.137547	1.140051	1.073425	(6005.053000)	-
最近挿入法/総距離誤差	1.250376	1.232117	1.258755	1.242894	1.238407	(6954.663000)	-
最遠挿入法/総距離誤差	1.526811	1.549562	1.687559	1.690709	1.675337	(11385.360000)	-

4.9　近似解法の精度向上

ここでは今までの手法をもとに，さらに精度をあげることを考える．そのために，新たなアルゴリズムを導入する．ここで導入するアルゴリズムは，遺伝的アルゴリズム（Genetic Algorithm：GA）とよばれるものである．このアルゴリズムは，1975 年に Holland によって初めて導入されたもので，生物進化から着想を得た汎用的な探索手法である．

現実世界では，環境下にたくさんの個体が存在し，その中には環境に適した個体，適さない個体が存在する．環境に適応できないものは自然淘汰され，環境に適した個体は生き残り，子供を産む．子供は親の環境に適した遺伝子を受け継ぎ，より環境に適した個体となる可能性がある．中には，突然変異を起こす子供もいる．このような過程を経て，世代を重ねるごとに，より環境に適した個体が生き残るということになる．

遺伝的アルゴリズムでは，このことをアルゴリズムとして形式化したもので,「選択（自然淘汰）」「交叉（子供を産む）」「突然変異」に着目し，解集合からより良いものを厳選するものである．

具体的には次のアルゴリズムである．

1. 初期個体集団を生成し，現世代とする．
2. ランダムに環境に適した個体を選択する．
3. 選択した個体を交叉させ，次世代とする．
4. ある一定の確率で突然変異を起こす．
5. 次世代を現世代とし，繰り返す．

巡回セールスマン問題に遺伝的アルゴリズムを適用するためには，以下の処理が必要である．

1. 解の個体表現
   解（巡回路）の個体表現には，パス表現を用いる．これは，頂点に名前 $a, b, c, d, ...$ とつけ，巡回順を並べたものを個体にするというものである．
   例えば，$a \to c \to d \to b \to e$ という巡回路があるならば，パス表現は (a, c, d, b, e) となる．
2. 選択
   環境に適しているものを選択するための判断基準として，ここでは巡回路の総距離数を用いる．総距離数が短いものであればあるほど，環境に適しているという判断である．
3. 交叉
   パス表現による交叉は，例えば次のようになる．
   $(a, b, c, d), (d, c, b, a) \to$ (前半 2 つと後半 2 つを入れ替える交叉）$\to (a, b, b, a), (d, c, c, d)$
   ここですぐにわかることは，このような交叉では結果が巡回路になっていない場合があるということである．
   そこでパス表現から順序表現に変換することを考える．順序表現とは，次に巡回する都市が残りの都市の中で何番目にあるかの番号を要素とするものである．例えば次のようになる．
   $(a, d, e, c, b) \to (1, 3, 3, 2, 1)$
   a は 1 番目の都市であるので，最初の要素は 1，次に d は a を除いた都市から 3 番目の都市であるので，次の要素は 3，というように求めていく．
   順序表現にし，交叉を行うと必ず巡回路となる．

4. 突然変異

   突然変異は，パス表現でランダムに選んだ２つを入れ替えることで行う．

   $(a, b, c, d) \to (a, d, c, b)$

以上を踏まえて，巡回セールスマン問題に遺伝的アルゴリズムを適用する．ここで，初期個体集団を今まで見てきた手法の欲張り法と Greedy 法によってある程度，厳密解に近いものになるようにする．

```
int kotai_num = 100;    // 世代の個体数
int sedai_num = 2000;    // 世代数

List<int[]> now_sedai = new List<int[]>();  // 現世代
List<int[]> next_sedai = new List<int[]>(); // 次世代

// 初期配置を現世代とする
for (int i = 0; i < kotai_num * 3; ++i)
{
    // 局所探索法・欲張り法
    do7();
    int[] root = new int[tour_root.Length];
    for (int n = 0; n < root.Length; ++n)
        root[n] = tour_root[n];
    now_sedai.Add(root);
}
for (int i = 0; i < 5; ++i)
{
    // 局所探索法・Greedy 法
    do8();
    int[] root = new int[tour_root.Length];
    for (int n = 0; n < root.Length; ++n)
        root[n] = tour_root[n];
    now_sedai.Add(root);
}

// 距離順にソートする
now_sedai.Sort(new RootComparer(this));


for (int i = 0; i < sedai_num; ++i)
{
    next_sedai = new List<int[]>();

    // 選択を行い、交叉を行う
    for (int n = 0; n < kotai_num / 2; ++n)
    {
        int[] kotai1 = now_sedai[rand.Next(0, now_sedai.Count - 1)];
        int[] kotai2 = now_sedai[rand.Next(0, now_sedai.Count - 1)];
        int[] child1 = new int[tour_root.Length];
        int[] child2 = new int[tour_root.Length];

        // 個体選択
        for (int a = 0; a < now_sedai.Count; ++a)
        {
            if (rand.Next(0, 6) == 0)
            {
                kotai1 = now_sedai[a];
```

```
                break;
            }
        }
        for (int a = 0; a < now_sedai.Count; ++a)
        {
            if (rand.Next(0, 8) == 0)
            {
                kotai2 = now_sedai[a];
                break;
            }
        }

        // 交叉
        Kousa(kotai1, kotai2, child1, child2);

        // 子供を次世代に
        next_sedai.Add(child1);
        next_sedai.Add(child2);
    }

    // 世代を交代する
    now_sedai = next_sedai;

    // 突然変異させる
    for (int n = 0; n < now_sedai.Count; ++n)
    {
        // 1/10000 の確率で突然変異させる
        for (int a = 0; a < 100; ++a)
        {
            if (rand.Next(0, 10000) == 1)
                Henni(now_sedai[n]);
        }
    }

    // 距離順にソートする
    now_sedai.Sort(new RootComparer(this));
}

{
    // 一番優秀な個体をルートとする
    int[] great = now_sedai[0];
    for (int n = 0; n < tour_root.Length; ++n)
        tour_root[n] = great[n];
}
```

全ての結果をまとめたものは次のとおりである.（試行回数 100 回あたりの平均値/距離計測は同頂点配置による計測/カッコ内の数値は総距離数の平均を示す）

	頂点数 15	頂点数 16	頂点数 20	頂点数 21	頂点数 22	頂点数 100	頂点数 1000
動的計画法/時間 (秒)	0.087193	0.185583	4.871877	11.431168	24.097777	-	-
（ノーマル）							
欲張り法/時間 (秒)	0.008694	0.008677	0.008830	0.009006	0.009150	0.010794	0.039235
Greedy 法/時間 (秒)	0.009960	0.010025	0.011521	0.008942	0.009011	0.023044	8.091780
最近挿入法/時間 (秒)	0.010182	0.010590	0.011145	0.008803	0.012261	0.016766	3.886422
最遠挿入法/時間 (秒)	0.011351	0.010980	0.012624	0.010454	0.012143	0.018683	3.893259
（局所探索法）							
欲張り法/時間 (秒)	0.010216	0.011504	0.013194	0.011098	0.010728	0.258100	240.658416
Greedy 法/時間 (秒)	0.010978	0.010104	0.012391	0.010050	0.010357	0.269266	236.259804
最近挿入法/時間 (秒)	0.011783	0.012176	0.011521	0.010242	0.009146	0.254373	241.466804
最遠挿入法/時間 (秒)	0.011612	0.012444	0.014705	0.012149	0.011447	0.304086	295.540890
（遺伝的アルゴリズム）							
GA/時間 (秒)	0.677608	0.705027	0.844544	0.872180	0.791028	3.097260	98.269290
GA+探索法/時間 (秒)	2.320956	2.939477	3.384881	3.501381	3.610852	84.511662	-
動的計画法/距離誤差	1.0	1.0	1.0	1.0	1.0	-	-
（ノーマル）							
欲張り法/距離誤差	1.152327	1.142614	1.161720	1.133257	1.084239	(6390.634000)	-
Greedy 法/距離誤差	1.130186	1.127949	1.142909	1.150388	1.084239	(6026.531000)	-
最近挿入法/距離誤差	1.277413	1.256310	1.290854	1.263411	1.266207	(7065.230000)	-
最遠挿入法/距離誤差	2.046981	2.080959	2.165671	2.123031	1.928646	(14589.070000)	-
（局所探索法）							
欲張り法/距離誤差	1.142420	1.128432	1.150661	1.107664	1.075397	(6347.113000)	-
Greedy 法/距離誤差	1.124755	1.121975	1.137547	1.140051	1.073425	(6005.053000)	-
最近挿入法/距離誤差	1.250376	1.232117	1.258755	1.242894	1.238407	(6954.663000)	-
最遠挿入法/距離誤差	1.526811	1.549562	1.687559	1.690709	1.675337	(11385.360000)	-
（遺伝的アルゴリズム）							
GA/距離誤差	1.022320	1.025435	1.031562	1.023722	1.018169	(5768.786000)	-
GA+探索法/距離誤差	1.020653	1.024034	1.026482	1.009456	1.006975	(5699.077000)	-

4.10　まとめ

前ページにて，全ての結果をまとめた表を示した．まず厳密解法である動的計画法による手法では，筆者の環境で頂点数が２２を超えると現実的な時間では解けないことがわかった．NP 完全問題がいかに難問であるかが，ここで見て取ることができる．

そこで，近似解法によって，現実的な時間で解けるものを考えてきた.「欲張り法」「Greedy 法」「最近挿入法」「最遠挿入法」の４つを比較すると，計測時間は，欲張り法が一番優秀で，その他の方法についても現実的な時間で解けていることがわかる．距離の誤差率を見ると,「欲張り法」「Greedy 法」の２つが優秀であることがわかる．よって，この４つを比較したとき，近似解法としては「欲張り法」「Greedy 法」が時間・精度ともに優秀なアルゴリズムであることがわかった．

次に「局所探索法」によって，精度をあげることを考えた．計測時間はあがるものの，精度はすべてのアルゴリズムに対しても上がっており，特に，精度が悪かった「最遠挿入法」の精度改善が著しいことがわかる．

さらに「遺伝的アルゴリズム（GA）」を用いて，精度をあげることを考えた．初期個体集団として，精度が優秀だった「欲張り法」「Greedy 法」を用いて，求めた．結果は，計測時間が若干上昇しているが，その分，精度はかなり改善されていることがわかる.「局所探索法」を初期個体集団に適用した場合，頂点数 1000 の場合，現実的な時間で求めることができず,「局所探索法」を適用しない場合と，精度の違いは微差である．

以上より，時間と精度のバランスを考えると,「欲張り法」「Greedy 法」を適用した「遺伝的アルゴリズム」での近似解法が現実的なアルゴリズムであることが結論づけられる．

5　おわりに

本論文では，巡回セールスマン問題を題材に NP 完全問題の解法を見てきた．現在知られている厳密解法では，多項式時間で解を見つけることが困難であり，近似解法によって解を見つける時間を短縮することが可能であった．近似解法には様々な手法があり，それぞれに特徴があることを見てきた．さらに近似解法の正確性を上げるための手法も見てきた．遺伝的アルゴリズムは，巡回セールスマン問題だけでなく他に応用のきくものである．

今後さらなる近似解法での精度の高い多項式時間アルゴリズムを考察することによって，$P \neq NP$ 予想へのアプローチのひとつになればと思う．

6　参考文献

本論文で出てきたテーマに関して，さらに詳しく知りたい方は以下の文献が参考になる．

参考文献

[1] 結城浩 著,『数学ガール 乱択アルゴリズム』，SoftBank Creative，2011 年

[2] B. コルテ/J. フィーゲン 著 浅野孝夫/浅野泰仁/小野孝男/平田富夫 訳,『組合せ最適化 第 2 版 理論とアルゴリズム』，シュプリンガー・ジャパン，2009 年

[3] J. ホロムコヴィッチ 著 和田幸一/増澤利光/元木光雄 訳,『計算困難問題に対するアルゴリズム理論』，シュプリンガー・フェアラーク東京，2005 年