メタヒューリスティクスと巡回セールスマン問題

> *Everyone knows that the traveling salesman problem is a metaphor or a mith. So you take cleaner formulations, study them as closely as possible, go deeply into their structures, and hope that results will transfer over to the real problems.*
> — Richard Karp (Turing Award Interview, 1985)

1　巡回セールスマン問題の定義と種類

難問中の難問として知られる巡回セールス問題だが，問題を理解することはいたって易しく，厳密な定義も単純明快．以下に 2 つの定義を示すが，表現は異なるものの求めていることは全く同じである．

グラフを用いた定義:　グラフ $G = (V, E)$，枝上の距離 (重み，費用) 関数 $d : E \to \mathbf{R}$ が与えられたとき，V の頂点すべてをちょうど 1 度ずつ通る巡回路 (Hamilton 閉路) で，その枝上の距離の総和が最小となるものを求めよ．

順列を用いた定義:　$n \times n$ の行列 $D = [d_{ij}]$ が与えられたとき，$V = \{1, 2, \ldots, n\}$ から V への 1 対 1 写像 (順列)ρ で

$$\sum_{i=1}^{n-1} d_{\rho(i),\rho(i+1)} + d_{\rho(n),\rho(1)}$$

が最小となるものを求めよ．

与えられたグラフ G が無向グラフで，任意の頂点対 (i, j) の間の距離が行きも帰りも同じ $(d_{ij} = d_{ji})$ 問題を**対称巡回セールスマン問題**とよぶ．また，G が有向グラフで，$d_{ij} \neq d_{ji}$ となる頂点対 (i, j) が存在する問題を**非対称巡回セールスマン問題**とよぶ．ランダムに作られた問題では，一般に非対称巡回セールスマン問題の方が対称巡回セールスマン問題よりもはるかに解きやすい．

このほかに応用重要な特殊例がいくつか存在する．例えば，現実の問題では同じ点を何度も経由できることもあるが，この場合，あらかじめすべての 2 頂点間の最短距離を計算しておくことで，**三角不等式**:

$$d_{ij} \leq d_{ik} + d_{kj}$$

がすべての i, j, k に対して満たされる巡回セールスマン問題に帰着させることができる．さらにこの特殊例として，頂点がすべて 2 次元平面上にある問題を**ユークリッド巡回セールスマン問題と**よぶ．2 頂点間の距離はその (x, y) 座標から計算されるが，ユークリッド距離は無理数となることもあるので，巡回セールスマン問題集および解答集 **TSPLIB** などでは次の関数 Dis() が用いられている:

```
int Dis(i, j)
int i, j;
{
  float xd, yd;
  double sqrt();

  xd = x[i] - x[j]; yd = y[i] - y[j];
  return((int) sqrt(xd * xd + yd * yd) + .5);
}
```

なお，TSPLIB は現在，以下で管理されている:

www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

2 単純な厳密算法

巡回セールスマン問題の最適解は，原理的にはすべての解 (V の順列) を列挙することで求めることができる．試しにユークリッド巡回セールスマン問題を解くための列挙法のプログラムを書いてみよう．

まず，巡回路を表す配列を tour[] が与えられたとき，その巡回路の長さを返す関数を記述しよう:

```
int cost_evaluate()
{
  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);
}
```

次にコアとなる順列の生成だが，tour の i 番目以降の順列を与える関数 perm(i) を用いる．これを再帰的に呼び出すことで，

- i 番目の要素を固定したまま，$i+1$ 番目以降の順列を生成し，
- i 番目の要素を j $(> i)$ 番目の要素と交換したのち，$i+1$ 番目以降の順列を生成する．

要素が $n-1$ に達したら，巡回路の長さを cost_evaluate() で評価し，今までに見つかった最良巡回路長 length よりも短ければその値を更新する．

```
perm(i)
int i;
{
  int j, tmp, cost;

  if (i < n - 2) {
    perm(i + 1);
    for (j = i + 1; j < n - 1; j++) {
      tmp = tour[i]; tour[i] = tour[j]; tour[j] = tmp;
      perm(i + 1);
      tmp = tour[i]; tour[i] = tour[j]; tour[j] = tmp;
    }
  } else {
    cost = cost_evaluate();
    if (cost < length) length = cost;
  }
}
```

最後に呼び出しルーチンを作ろう．初期順列として $0, 1, \ldots, n-1$ を tour に入れたのち，0 番目以降の順列を perm() で生成する．perm() の再帰が終了した時点での length が最適な巡回路長である．最良巡回路長を更新したときに，その巡回路も保存しておけば最適巡回路も得られるが，ここでは省略する．

```
enumerate()
{
  int i;

  for (i = 0; i < n; i++)
```

```
    tour[i] = i;
  perm(0);
  printf("%d", length);
}
```

ここで注意しておくが，いきなり大きな巡回セールスマン問題に対して上のプログラムを実行しないこと．例えば $n = 30$ の問題を作成して enumerate() を実行しても，何十年待っても終了しない．せいぜい $n = 10$ 程度の問題に止めておくのが無難である．

3 動的計画法による厳密算法

それでは $n = 10$ を越える巡回セールスマン問題を厳密に解くことは絶望的かと言えば，計算の工夫次第ではもう少し何とかなる．ここで，**動的計画法**による算法を紹介しよう．

ある始点 s から出発し，$S \subset V$ の頂点すべてを巡って頂点 $j \in S$ にいたる最短路長を $f(j, S)$ と書くことにしよう．この値 $f(j, S)$ を与える道で頂点 j の直前に訪れる頂点 i へは，始点 s から $S \setminus \{j\}$ の頂点すべてを巡って i にいたる最短路を経由しているはずである．したがって，次の関係の成り立つことがわかる:

$$f(j, S) = \min_{i \in S \setminus \{j\}} \{f(i, S \setminus \{j\}) + d_{ij}\}. \tag{1}$$

一方，

$$f(i, \{i\}) = d_{si}, \quad i \in V, \tag{2}$$

は明らかなので，(2) から初めて (1) を使い，逐次 S を大きいくして $f(s, V)$ まで計算すれば，それが最適な巡回路長となる．

列挙法のときと同じように頂点を $0, \ldots, n-1$ で表し，始点 s を $n-1$ 番目の頂点としよう．頂点集合は整数で表し，頂点が集合に含まれているかどうかを 2 進表記のビットで表すことにする．つまり V の部分集合 S は，0 ($S = \emptyset$) から $2^n - 1$ ($S = V$) までの整数となる．整数 2^n は 1 を n 回左にビットシフトすることで得られるので，定数 SMAX を 1 << n としておく．

はじめに，$f(i, S)$ の値を示す f[i][S] に大きな値を初期設定し，次に (2) の条件を代入する．次に (1) を計算するが，頂点 i が集合 S に含まれているかどうかは 1<<i と S とのビットごとの論理積をとることで判定する．また，$S \cup \{j\}$ は 1<<j と S とのビットごとの論理和をとる．

```
dynamic()
{
  int i, j, S, tmp, 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]);
}
```

このプログラムを使えば，$n = 20$ 程度の巡回セールスマン問題も最近のパソコンを使えば解けるかもしれない．これよりも大きな問題 (数千都市！) に対しては分枝限定法や分枝カット法で厳密な最適解を求めることもできるが，その詳細については別の機会に説明しよう．

4 ヒューリスティクス概観

前節のプログラムではとても無理だが，計算の工夫次第では数千都市の巡回セールスマン問題を厳密に解くことも可能で，実際 1995 年には $n = 7397$ の問題も解かれている．そうなると最早，近似算法など必要ないようにも思えるが，現実には数十万，数百万都市の問題を高速に解くことが求められることも少なくない．

近似算法には大きく分けて 2 種類ある．

構築法: 何もないとろから実行可能な解を構築する．

改善法:　何らか方法で得られた実行可能解を改善していく.

後者については次節以降に述べることにして，以下では対称巡回セールスマン問題に絞って構築法をいくつか紹介しよう.

最近隣接法:　適当な頂点から出発して，まだ訪れていない頂点で現在の頂点から最も近い頂点へ移動する．すべての頂点を訪問したら出発点へ戻る.

貪欲法:　枝を短い順に選んで頂点をすべて通る閉路を構築する．すべての頂点を通るまえに閉路ができたり，接続している枝の本数が 2 をこえてしまう頂点ができる場合には，その枝を選ばない.

最遠挿入法:　まず，適当な頂点を選び，そこから出発して他の頂点をいっさい通らずに戻る自己閉路を作り，これを初期巡回路とする．まだ巡回路に含まれていない頂点の中で現在の巡回路からもっとも遠い頂点を選び，それを巡回路の隣接した 2 つの頂点の間に，距離の増加が最も小さくなるように挿入する．この操作を，すべての頂点を通る巡回路が得られるまで繰り返す.

以上の方法は自然ではあるが，生成される解が最適である保証がないばかりか，最適解との誤差の保証すらない．このように何の保証もない算法は，実は近似算法と呼ばれず，**ヒューリスティクス**と総称される．近似算法と呼ぶことができるのは，最悪の場合でも最適値との誤差が事前に保証される実行可能解を生成するものだけである．これについては，また別の機会に説明しよう.

5　実用的なヒューリスティクス

巡回セールスマン問題に対する改善法の代表的なものは**局所探索法**である．その原理は，暗い夜道を懐中電灯頼りに山登りするのと全く同じである．自分が今いる場所の周りを懐中電灯で照らし，今より高い所が見つかればそっちの方向へ移動する．照らせる範囲に今より高いところがなければ，そこがその夜に到達できる最も高い場所である．もちろん，朝になって今いる場所よりもずっと高い場所が見つかるかもしれないが，とりあえずこの方法を使えば局所的に最適な解には到達できる.

今いる場所を $\mathbf{x}$，その標高を $c(\mathbf{x})$，懐中電灯で照らせる範囲を $N(\mathbf{x})$ で

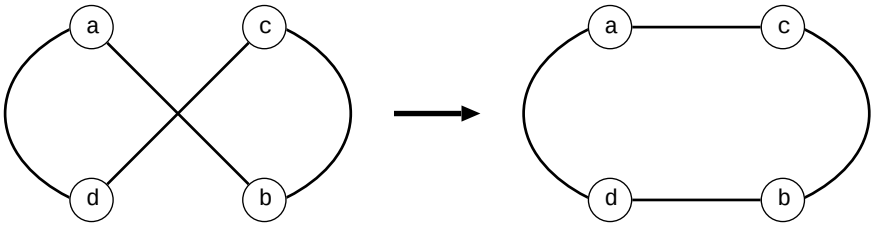

図 1: 2-opt

表すことにしよう．関数

$$\text{improve}(\mathbf{x}) = \begin{cases} \text{any } \mathbf{x}' \in N(\mathbf{x}) \text{ with } c(\mathbf{x}') > c(\mathbf{x}) & \text{if such an } \mathbf{x}' \text{ exists} \\ \emptyset & \text{otherwise} \end{cases}$$

を用いれば，山登りの局所探索法は次のような疑似アルゴルで記述できる:

algorithm local_search
begin
  x := any initial feasible solution;
  while improve(**x**) $\neq \emptyset$ **do**
    x := improve(**x**);
  return x;
end;

巡回セールスマン問題に対して局所探索法を用いる場合，懐中電灯で照らせる範囲に相当する**近傍**の定義によっていくつかのバリエーションを考えることができるが，最も簡単なのは次の **2-opt** と呼ばれる方法である．図 1 は 2-opt における近傍の考え方を表している．この図のように巡回路の 2 つの枝を繋ぎ変えることで得られる別の巡回路は，元の巡回路の近傍であると定義される．したがって，左の巡回路が「今いる場所」とすれば，その近傍である右の巡回路長の方が短いとき，そちらへ「移動」する．この操作を巡回路長を減少させる近傍がみつからなくなるまで続ける．

5.1 データ構造と必要な関数

ユークリッド巡回セールスマン問題では各頂点対の間に枝が存在するが，巡回路にはそれほど長い枝は含まれないはずである．そこで事前に，各頂点ごとに近い順に 10 個程度の頂点との間にだけ枝があるものとして残りはすべて消去しておく．その結果得られるグラフで，どの点とどの点が隣接しているかという情報は計算途中で変化しないので，**Forward-Star** と呼ばれるグラフの表現法を用いることにする．

まず，adj[]と head[] の 2 つの配列を用意する．頂点 i に隣接する頂点は配列 adj[] の head[i] から head[i+1] - 1 番目に，頂点 i に近い順に保存する．巡回路を表すデータ構造は，前と同じように 1 次元配列 tour[] を使う．tour[i] には i 番目に通過した頂点の番号を保存する．このほか，2-opt 法には以下を行う関数が必要となる．

Next(i)： 頂点の番号 i を入力すると，現在の tour[] 上における次の頂点を返す．

Flip(a, b, c, d)： 4 つの頂点 a, b, c, d で b = Next(a), d = Next(c) を満たすものを入力すると，現在の巡回路にける枝 ab, cd を ac, bd に置き換える (図 1 を見よ)．さらに，tour[] 上の頂点 b, c 間のすべての頂点を逆順にするか，頂点 d, a 間のすべての頂点を逆順にする．

5.2 プログラムと考察

入力として適当な構築法で求めた初期巡回路 tour[] と，その長さ length を与える．プログラムは 2 つの基本ループの中で解の改善の可能性をチェックし，改善されれば Flip() を呼んで解を更新する．

```
tow_opt()
{
  int i, j, a, b, c, d, tmp;

  tow_opt_start;
  for (i = 0; i < n; i++) {
    a = tour[i];
    b = Next(a);
    for (j = head[a]; j < head[a+1]; j++) {
      c = adj[j];
      if (b == c) break;
      d = next(c);
      tmp = Dis(a, b) + Dis(c, d) - Dis(a, c) - Dis(b, d);
      if (tmp > 0) {
        length -= tmp;
        Flip(a, b, c, d);
        goto two_opt_start;
      }
    }
```