Program Promenade

連載

どこで会える？

寺田　実（電気通信大学情報通信工学科）
terada@ice.uec.ac.jp

■問題

今回取り上げるのは，2001 年 11 月にはこだて未来大学で行われた，アジア地区予選函館大会の問題の F，"Young, Poor and Busy" である（http://www.fun.ac.jp/icpc/ から入手可能）．

若くて，お金がなくて，しかも忙しい 2 人が函館と東京に住んでいて，最も安く会うためのプランを考えるのが目標である．

入力データとして，駅を結ぶ列車の時刻表が与えられる．列車は始発駅と終着駅とを直接結んでおり，途中に停車駅はない．料金は列車ごとに決まっている．

それらの列車を利用して，函館駅からは Ken が，東京駅からは Keiko がそれぞれ出発し，どこかの駅で落ち合い，またそれぞれの出発地に戻る（落ち合う駅は函館や東京でもよい）．

制約事項は，

- 出発は午前 8 時以降，
- 帰着は午後 6 時以前，
- 30 分以上会っていられること，

であり，問題として要求されているのはその条件を満たす最も安いプランの総費用である．もし制約条件を満たす解が存在しない場合には 0 を結果とすることになっている．

入力データはデータの組の並びであり，ひと組のデータは，列車の総数を表す行と，そのあとに各列車の発駅，発時刻，着駅，着時刻，料金を示す行が続く．データ全体の終了は 0 だけからなる行で表す．

問題文に含まれている入力データの組の具体例を見よう．

```
5
Hakodate 08:15 Morioka 12:30 2500
Morioka 14:05 Hakodate 17:30 2500
Morioka 15:30 Hakodate 18:00 3000
Morioka 14:30 Tokyo 17:50 3000
Tokyo 08:30 Morioka 13:35 3000
```

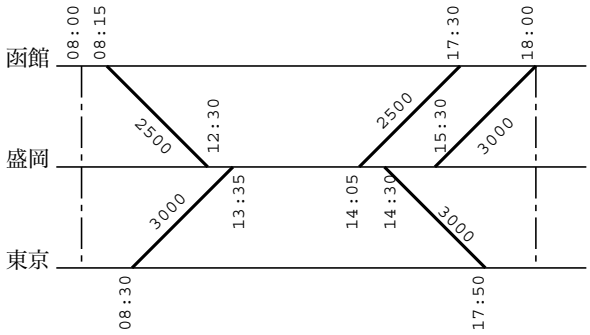

図 -1　ダイヤグラムによる表現

列車は 5 本ある．同じ情報をダイヤグラムとして表現したものが図 -1 である．縦軸に路線の距離，横軸に時刻をとり，列車は斜めの線として表される（通常の列車ダイヤには料金は書いてないが，ここでは線に沿った斜めの数字として料金を示す．また，起点の駅を上にするのが普通だが，コンテストの開催地に敬意を表して函館を起点としよう）．

この例では，明らかに函館や東京で会うことはできない．そこで，唯一の可能性である盛岡を考えると，東京からの往復は 1 通りに定まるが，函館からは帰りの列車の選び方が 2 通りあり，そのいずれでも 2 人は 30 分以上一緒にいられるので制約条件を満たす．総費用は，早い列車（14:05 発）で函館に帰る場合には 2500 + 2500 + 3000 + 3000 = 11000 円であるのに対して，遅い方の列車（15:30 発）では 500 円だけ高くなってしまうので，結局求める総費用は 11000 円ということになる．

なお，規模に関する条件としては，
- 駅の数の上限は 100
- 列車の数の上限は 2000

となっている．

■考え方

この問題を解くための中心となる値は，たとえば

函館を午前 8 時以降に出発して盛岡に向かうとして，ある時刻に到着するためにはいくら払う必要があるか

である（その時刻までに到着する列車がない場合には無限大の費用がかかるとする）．この値の変化を，到着時刻を変数とみて考えると，値に変化があるのはその駅に列車が到着したときに限られ，また，その変化は減少だけに限られる．なぜなら，仮にその列車での到着がそれまでよりも費用がかかるものであったとしたら，その列車を利用しなければよいだけの話だからである．つまり，この値は時刻変化に対して単調減少の階段状関数となる．上記の例でいえば，08:00 から 12:29 までは無限大であるが，12:30 以降は 2500 円である．

この値を $fr\_h(st, t)$ と書くことにしよう．h は函館起点を表し，st は目的の駅（ここでは盛岡），t は到着時刻である．もちろん函館の代わりに東京を起点とする同様の値 $fr\_h(st, t)$ もある．

また，それとは逆に

盛岡を出発して函館に午後 6 時までに帰着するとして，ある時刻まで盛岡にいるためにはいくら払う必要があるか

という値も定義できる．こちらは出発時刻に対して単調増加の階段状関数である．上記の例では，08:00 から 14:05 までは 2500 円，15:30 までは 3000 円，それ以降は無限大となる．こちらは $to\_h(st, t)$, $to\_t(st, t)$ としよう．

これが求まったとすると，駅 st において 2 人が時刻 t から 30 分会うための総費用 $cost(st, t)$ は，

$$\begin{aligned} cost(st,t) = \; & fr\_h(st, t)+ \\ & fr\_t(st, t)+ \\ & to\_h(st, t+30)+ \\ & to\_t(st, t+30) \end{aligned}$$

となる．最終結果はこれをすべての駅と時刻で計算した最小値であるから，

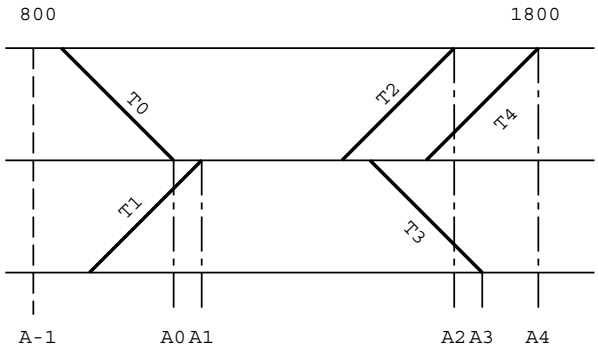

図 -2　時刻の離散化

$$\min_{st} \min_{8:00 \le t \le 17:30} cost(st, t)$$

となる．t の上限の 17:30 は，目的地での 30 分以上の滞在を考えに入れてのことである．

■*fr_h* を求める

以下で，前章で用いた 4 つの値の代表として，$fr\_h(st, t)$ の求め方を考える．

まず，時刻 t を離散化する．函館からある駅までの到達費用に変化が生じるのは，その駅にどこかから列車が着いたときだけである．そこで，すべての列車を**到着時刻順**に並べて，$T_0, T_1, ..., T_{nt-1}$ とする．ここで，nt は列車の総数を表す．列車 T_i の発駅を FR_i, 発車時刻を D_i, 着駅を TO_i, 到着時刻を A_i, 料金を $FARE_i$, とする．また，8:00 を A_{-1} とする（図 -2）．これで，時刻 t のかわりに，列車到着事象の番号（つまり A_i の添字 i）を用いることができる．それによって，値 $fr\_h(st, t)$ は離散的な添字を持つ 2 次元の配列とすることができ，これを順に埋めていくことが可能になる．

まず，8:00 における値 $fr\_h(st, -1)$ を考えよう．この時点では列車がどこかに着いていることは不可能であるから，函館を除くすべての駅で無限大の値をとり，函館についてはすでにそこにいるのだから 0 円である．

次に，時刻を 1 つ進めて A_0 を考える．これは列車 T_0 が駅 FR_0 から駅 TO_0 に到着した時刻である．前出のデータ例でいえば，函館 8:15 発の列車が盛岡に 12:30 に到着した時点である．このとき，fr_h（盛岡，t）は無限大から 2500 円へと変化する．その他の駅については変化はない．

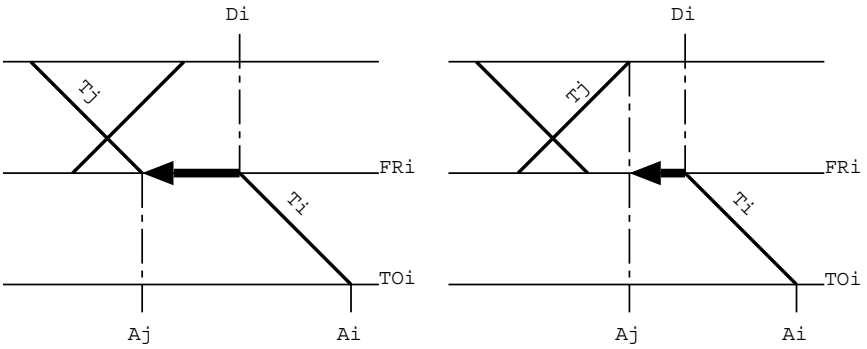

図 -3　*change(i)* の決定

$$fr\_h(st, i) = \begin{cases} 0 & i = -1,\ st = \text{函館} \\ \infty & i = -1,\ st \neq \text{函館} \\ fr\_h(st, i-1) & i \geq 0,\ st \neq TO_i \\ \min[fr\_h(st, i-1),\ FARE_i + fr\_h(FR_i,\ change(i))] & otherwise \end{cases}$$

図 -4　*fr_h* の再帰的定義

これを一般的にいえば，$fr\_h(st, i)$ は，

- T_i の着駅 TO_i が st でなければ，$fr\_h(st, i-1)$ から変化なし
- TO_i が st と一致していれば，以下の小さい方の値
  - $fr\_h(st, i-1)$
  - $FARE_i + fr\_h(FR_i,\ change(i))$

    ここで $change(i)$ は，列車 T_i に乗り継げる最も近い列車の着時刻 A_j の添字 j である（図 -3 左）（言い替えると，D_i 以前の最も近くに FR_i に到着する列車の着時刻）．もし，そのような列車がない場合には，-1 とする．

として，順次求めていくことができる．

ここで，$change(i)$ の定義について補足．いまの説明では，「乗り継ぎ」を条件として，駅 FR_i への列車の到着時刻としたが，実は必ずしも駅 FR_i への到着である必要はなく，D_i 以前の着時刻であればどこの駅への到着でもよい（図 -3 右）．この違いは，本章のように表を順次埋めていくアプローチでは実行効率に何の違いもないが，次章で述べる再帰的な方法では，FR_i への到着まで再帰的に遡る必要が生じ，呼出の深さの点で大きな負担となる（図 -4 の 3 行目の再帰呼出）．

データ例について *fr_h* を求めると，以下のようになる．

i	時刻	函館	東京	盛岡
−1	8:00	0	∞	∞
0	12:30	0	∞	2500
1	13:35	0	∞	2500
2	17:30	0	∞	2500
3	17:50	0	5500	2500
4	18:00	0	5500	2500

東京を起点とした *fr_t* についてもまったく同様に求めることができる．

■ *fr_h* の再帰的定義

前章のように表を順次埋めていくのは，この問題についてはごく自然な発想であると思う（筆者はこの方式を先に思いついた）．しかし，表を順次埋めるというのは動的計画法の典型的なパターンであり，その元になる再帰的な定義が存在するはずで，それは前章の漸化式から簡単に求めることができる（図 -4）．人によっては先にこちらの再帰的定義を思いつき，それに動的計画法を適用する，という手順を踏むかもしれないが．

いずれにせよ，実行効率を考えると，再帰的定義をそのまま計算するのは列車数の指数オーダになる可能性があり，現実的な方法ではない．問題文中に示して

ある例（駅数 4，列車数 18）程度では解けても，審判団のテストデータ（これも Web ページで公開されている）には駅数 25，列車数 1123 のものがあり，これはまったく望みがない．

■ *to_h* を求める

次に，函館に帰着する方を考えよう．本質的には函館を起点とする *fr_h* と同様であるが，逆の処理が必要となる．まず，時刻は発車時刻を用いて離散化し，列車は**発車時刻順**に並べておく必要がある．18:00 の時点から時刻を遡るかたちで表を埋めていく．これをその通りプログラムとするのに別に困難はないが，ほとんど同形であるにもかかわらず共用できないコードが 2 組できる結果になってしまう．

ここで，本連載の執筆者の 1 人の石畑氏に教示していただいた非常にうまい方法がある．これを用いれば，*to_h* は *fr_h* と同じコードで求めることができるのである．

基本的なアイディアは「フィルムの逆回し」である．午後 6 時から時間軸を逆転して見てみると，最後に到着した列車が最初に発車する列車となり，もともとは始発駅だった駅に向かって進行する．ダイヤグラムで説明すれば，時間軸を左右逆転することに相当する．こうすると，

> *盛岡をできるだけ遅く出発して函館に 18:00 までに到着する*

というプランが，

> *函館を 18:00 に出発して，できるだけ早く盛岡に到着する*

となり，これはまさに *to_h* を *fr_h* に変換していることになる．

この変換をプログラム上で実現するには，列車のデータに対して

- 始発駅 FR_i と終点 TO_i の交換
- 発時刻 D_i と着時刻 A_i の時間軸反転と交換

の処理が必要である．ここで，時間軸反転とは，元の時刻 t を $-t$ にするだけでよい（実際には，デバッグの都合を考えると適当なバイアスを加えて正数にしたほうが都合がよいだろう）．

以上でアルゴリズムの設計は完了し，以下でコードを示そう．

■データ構造

まず，駅を表現する方法である．入力データでは駅は文字列として登場するが，これは整数値に変換するのが定石である．そのためには駅名を表に登録する処理が必要であるが，これは省略．データを読み込む前に函館と東京を登録してしまうことによって，これらに 0 と 1 の番号をあらかじめ割り当てている．

次に，列車の表現である．前述の通り列車は複数の値の組になるので，構造体を使うのがよいだろう（フィールドごとに別の配列とする方法もあり，これもなかなかシンプルでよいのだが，今回は列車データ全体を着時刻をキーとしてソートする必要があるため，メモリ上でもまとまっていた方が扱いやすい）．

列車の本数は変数 nconn で保持する．これはプログラムの入力データとして与えられた列車データ総数から，8:00 以前に発車する列車や 18:00 以降に到着する列車を除いた数とする．列車データはオリジナルのもの（trains[]）のほかに，前章で述べた「時間軸反転」データも rtrains[] として保持する．

また，*fr_h*(*st*, *t*) なども 2 次元の整数配列として実装する．ただし，第 2 の添字（離散化した時刻）に −1 が現れるので，実装では 1 だけずらすことにした．

```
#define MAXCITY 100
/* 駅名の表 */
char city_name[MAXCITY][18];
int ncity;       /* 駅数 */

#define MAXCONN 2000+1

/* 列車の情報 */
struct train {
  int from, to; /* 駅番号 */
  int dpt, arv; /* 0:00 からの分単位 */
  int fare;     /* 料金 */
} trains[MAXCONN], rtrains[MAXCONN];

int nconn;       /* 列車数 */

/* 重要な駅番号 2 つ */
#define HAKODATE 0
#define TOKYO 1

#define INFINITE 99999999

int from_hakodate[MAXCITY][MAXCONN],
```

```
  from_tokyo[MAXCITY][MAXCONN],
  to_hakodate[MAXCITY][MAXCONN],
  to_tokyo[MAXCITY][MAXCONN];

#define BIAS ((18-8)*60)
```

■列車データの準備

列車データを読み込んだ後(プログラムは省略),「時間軸反転」データを作り，オリジナルデータとそれを到着時間順にソートする．ソーティングには，C の標準ライブラリ関数 qsort を使っている．第 3 引数として，要素の大小を判定する関数を渡す．

```
int cmp_arv(struct train *t1,
             struct train *t2)
{ return (t1->arv - t2->arv); }

void prepare_data(void)
{
  int i;

  for(i=0; i<nconn; i++){
    rtrains[i].from = trains[i].to;
    rtrains[i].to = trains[i].from;
    rtrains[i].dpt = BIAS-trains[i].arv;
    rtrains[i].arv = BIAS-trains[i].dpt;
    rtrains[i].fare = trains[i].fare;
  }

  qsort(trains, nconn,
        sizeof(struct train), cmp_arv);
  qsort(rtrains, nconn,
        sizeof(struct train), cmp_arv);
}
```

■表の作成

動的計画法を利用して，*fr_h*(*st*, *t*) をはじめとする 4 つの表を埋める処理である．関数 make_table は，埋めるべき表，起点／終点，列車の時刻データを引数として受け取る．

```
int change(struct train tv[], int p,
           int st, int dpttime)
{
  while(p >= 0){
    if((tv[p].to == st) &&
       (tv[p].arv <= dpttime)) break;
    p--;
  }
  return p;
}

void make_table(int v[MAXCITY][MAXCONN],
                int org,
                struct train tv[])
{
  int ti;
  int i;
  int a;

  for(i = 0; i < ncity; i++)
    v[i][0] = INFINITE;
  v[org][0] = 0;

  for(ti=0; ti<nconn; ti++){
    for(i = 0; i < ncity; i++){
      v[i][ti+1] = v[i][ti];
    }
    a = change(tv, ti-1, tv[ti].from,
               tv[ti].dpt);
    v[tv[ti].to][ti+1] =
        min(v[tv[ti].to][ti],
            tv[ti].fare
            + v[tv[ti].from][a+1]);
  }
}
```

■解を求める

作成した表をもとに，解を求める処理である．関数 calc_cost はまず駅を固定し，(列車到着で離散化した) 時刻ごとに，30 分以上滞在する場合の費用の最小値を求める．それには，その駅への到着時刻を表す a と，発車時刻を表す d を，30 分以上で最短の間隔を開けながら並行して動かしていく．関数 solve は，それをすべての駅に対して行うことで全体の最小値を求めている．

```
int calc_cost(int city)
{
  int a=0, d=nconn-1;
  int stay;
  int c, min_c=INFINITE;

  while(1){
    stay = (BIAS - rtrains[d].arv)
           - trains[a].arv;
    if(stay < 30){
```

```
      d --;
      if(d < 0) break;
      continue;
    }
    c = from_hakodate[city][a+1] +
      from_tokyo[city][a+1] +
      to_hakodate[city][d+1] +
      to_tokyo[city][d+1];
    if(c < min_c)
      min_c = c;
    a ++;
    if(a >= nconn) break;
  }
  return min_c;
}

int solve(void)
{
  int c, a_i;
  int cost, min_cost;

  prepare_data();

  make_table(from_hakodate, HAKODATE,
             trains);
  make_table(from_tokyo, TOKYO, trains);

  make_table(to_hakodate, HAKODATE,
             rtrains);
  make_table(to_tokyo, TOKYO, rtrains);

  min_cost = INFINITE;
  for(c = 0; c<ncity; c++){
    cost = calc_cost(c);
    if(cost < min_cost)
      min_cost = cost;
  }

  if(min_cost >= INFINITE)
    min_cost = 0;
  return min_cost;
}
```

必要となる計算の手間は，表の作成，解を求める処理とも，（駅の数）×（列車の数）であり，問題の規模の制約のもとで，十分実行可能である．

ただ，このプログラムは単純化のために，不必要な計算を行っているのも事実である．具体的には，*fr_h*(*st*, *t*) の表をすべての（離散化）到着時刻に対して構成している点が問題で，本当に必要なのはその駅に到着する時刻においてだけである．そうするなら表のサイズは，全体として列車の数でおさえられ，全体の処理時間も列車数でおさえられる．

（平成 15 年 1 月 8 日受付）