[ICPC2009 国内予選問題 D]　離散的速度

- 最短時間を求める問題であり、基本的には最短経路問題に対するダイクストラ法(dynamic programming の一種）が使える。通常の最短経路問題では、都市を決めると、出発地からその都市までの最短経路長とその都市から目的地までの最短経路長が一意的に決まる。しかしながら、この問題では、(直前にいた都市、今いる都市、今いる都市に到着したときの速度）の三つ組に対して、出発からの最短時間や目的地に速度１で到着するまでの最短時間が決まると考えられる。したがって、この三つ組を通常の最短経路問題における都市と考える必要がある。
- 以下のプログラム例では、出発地からの探索が進み現時点で到達している状態（上記の三つ組）のうち最短時間が求まっていない状態をヒープで管理している。また、ゴール状態（目的地へ速度 1 で到着）までの最短時間に影響しないことが判明した状態は探索に含めない（ヒープに入れない）ようにしている。
- なお、回答には 0.001 以下の誤差は許されるので、回答のチェックにはこの誤差を考慮して比較するプログラム check を作成し、それを用いている。

プログラム例

```
/* ACM-ICPC2009 国内予選 Problem D */
// http://www.waseda.jp/assoc-icpc2009/preliminary/contest/all_ja.html#section_D
// filename = pd.c
// コンパイル： cc -O2 pd.c
// 実行方法：　./a.out < D0 > D0.result 等
// 確認方法：　./check D0.ans D0.result 等
//　　（check は最大誤差 0.001 を考慮して比較）
// アルゴリズム：　基本的にはダイクストラ法だが、
//　　都市の状態として、どの都市から来たか、その時の速度を含めて考える。
//　　状態集合の管理にヒープを用いる。
//　　現時点で未到達の状態はヒープに入れない。
//　　目的地に速度 1 で到着する現時点での最短時間を超える状態は
//　　ヒープに加えない
//　　ヒープ内の最短時間の状態を取り出し、その隣接都市（状態）への
//　　到着時間を計算する。
//　　隣接都市の状態に初めて到達した場合はその状態をヒープに加え
//　　既にヒープ内にある場合は、到着時間が短ければ到着時間を更新する。
//　　ヒープ内のデータがなくなれば終了。

#include <stdio.h>
#include <stdlib.h>

#define MAX_N 30                // 都市数の最大値
#define MAX_D 100               // 都市間距離の最大値
#define MAX_SL 30               // 制限速度の最大値
#define INF (MAX_N * MAX_D)     // 到達可能なら再短時間はこれ以下
```

```
int n;  // 都市数 2 <= n <= 30
int m;  // 都市間道路の本数
int s;  // 出発地の都市番号
int g;  // 目的地の都市番号  g != s

int d[MAX_N+1][MAX_N+1];        // 都市間道路の距離 1 以上 100 以下
int c[MAX_N+1][MAX_N+1];        // 都市間道路の制限速度 1 以上 30 以下

struct std {
  double st;    // 現時点での最短時間
  int from;     // どの都市から来たか
  int city;     // 今いる都市
  int speed;    // この町に来たときの速度
  int hp;       // ヒープ内の位置
};

// t[a][b][c] 都市 a から都市 b に速度 c で来たときの状態データ
struct std t[MAX_N+1][MAX_N+1][MAX_SL+1];

double tmin;            // 求める再短時間
struct a_list {
  int city;
  struct a_list *next;
};
struct a_list *al[MAX_N+1];     // al[i] = 都市 i の隣接都市のリスト

struct std *h[MAX_N*MAX_N*MAX_SL+1];    // ヒープデータ
int nh;         // ヒープに格納しているデータ数

// 親方向に向かってヒープを再構成 (h[i]->st により小さな値を代入したときに
// 呼び出す)
void upheap(int i)
{
  int oya;
  struct std *tmp;
  while(i>0){
    oya = (i-1)/2;
    if(h[oya]->st <= h[i]->st) break;
    tmp = h[oya];       // 親節点と交換
    h[oya] = h[i];
    h[i] = tmp;
    h[i]->hp = i;       // 状態データのヒープ内位置の値を更新
    h[oya]->hp = oya;
    i = oya;            // 親を i として繰り返す
  }
}

// 都市 from から都市 to に速度 speed で到達したときの時間 nt を
// 必要に応じてヒープに挿入
void ins(int from, int to, int speed, double nt)
{
  struct std *stp;
  int i, oya,ls,rs;
```

```
  if(to == g && speed == 1){    // 目的地に速度 1 で到着した場合
    if(nt < tmin){              // より短時間で到着したら
      tmin = nt;                // 最短時間を更新
      t[from][to][speed].st = nt;
    }
    return;                     // このケースをヒープに加える必要は無い
  }
  stp = &t[from][to][speed];
  if(stp->st == -1){            // 初めてこの状態に達した場合
    stp->st = nt;               // 到着時刻の情報をセットし
    i = stp->hp = nh++;         // ヒープの最後に追加し
    h[i] = stp;
    upheap(i);                  // upheap によりヒープを再構築
  }
  else if(stp->st > nt){        // ヒープ内の状態により短時間で到達した場合
    stp->st = nt;               // 到着時刻の情報を更新し
    i = stp->hp;
    upheap(i);                  // upheap によりヒープを再構築
  }
}

// ヒープの根節点（最小値）を取り出す。
struct std *delmin()
{
  int i,ls,rs;
  struct std *rval, *tmp;
  if(nh == 0) return NULL;
  rval = h[0];                  // 根節点
  rval->hp = -1;
  nh--;                         // ヒープ内のデータ数を更新
  if(nh==0) return rval;
  h[0] = h[nh];                 // ヒープの最後のデータを根節点へ移動
  i = 0;                        // 根節点から downheap 操作でヒープを再構築
  while(1){
    ls = 2*i+1;                 // 左の子の位置
    if(ls >= nh) break;         // 左の子が無ければ再構築終了
    rs = ls+1;                  // 右の子の位置
    if(rs >= nh){               // 右の子は無い（左の子は有る）
      if(h[i]->st <= h[ls]->st) break;  // 親の値が左の子以下なら再構築終了
      tmp = h[i];               // 左の子と親を交換
      h[i] = h[ls];
      h[ls] = tmp;
      h[i]->hp = i;
      h[ls]->hp = ls;
      i = ls;                   // 元の左の子の位置から downheap を繰り返す
    }
    else {                      // 左右の子がある場合
      // 親の値が両方の子の値以下なら再構築終了
      if(h[i]->st <= h[ls]->st && h[i]->st <= h[rs]->st) break;
      if(h[ls]->st < h[rs]->st){        // 左の子の値の方が右の子の値より小さい
        tmp = h[i];             // 親と左の子を交換
        h[i] = h[ls];
        h[ls] = tmp;
```

```
        h[i]->hp = i;
        h[ls]->hp = ls;
        i = ls;                 // 元の左の子の位置から downheap を繰り返す
      }
      else {                    // 右の子の値が左の子の値以下
        tmp = h[i];             // 親と右の子を交換
        h[i] = h[rs];
        h[rs] = tmp;
        h[i]->hp = i;
        h[rs]->hp = rs;
        i = rs;                 // 元の右の子の位置から downheap を繰り返す
      }
    }
  }
  return rval;  // ヒープの再構築が終了したので、最初の根節点（最小値）を返す
}

// 都市 x の隣接都市に y を加え、都市 y の隣接都市に x を加える
void gen_al(int x, int y){
  struct a_list *pt;
  // 都市 x の隣接都市リストに y を追加
  pt = (struct a_list *)malloc(sizeof(struct a_list));
  pt->city = y;
  pt->next = al[x];
  al[x] = pt;
  // 都市 y の隣接都市リストに x を追加
  pt = (struct a_list *)malloc(sizeof(struct a_list));
  pt->city = x;
  pt->next = al[y];
  al[y] = pt;
}

// 最短時間の探索（基本的にはダイクストラ法と同じ）
void search()
{
  struct std *stp, *nst;
  struct a_list *alp;
  int ns;
  double nt;
  while(nh){            // ヒープにデータがある間以下を繰り返す
    stp = delmin();     // 最小の値を持つデータをヒープから取り出す
    alp = al[stp->city];        // その都市の隣接都市リスト
    while(alp){         // 各隣接都市に対して以下を繰り返す
      if(alp->city != stp->from){       // Ｕターン禁止
        for(ns = stp->speed - 1; ns <= stp->speed + 1; ns++){   // 速度変更
          if(ns == 0 || ns > c[stp->city][alp->city]) continue; // 不当な速度
          // 隣接都市への到着時間を計算
          nt = stp->st + (double)d[stp->city][alp->city]/(double)ns;
          if(nt >= tmin) continue;    // 現時点での目的地到着時間を超えている場合
                                      // 何もしない
          ins(stp->city, alp->city, ns, nt); // ヒープに追加/ヒープ内データの更新
        }
      }
      alp = alp->next;          // 次の隣接都市を調べる
```

```
    }
  }
}

int main()
{
  int i,j,k,l;
  int xi, yi, di,ci;
  struct a_list *alp, *aln;

  while(1){
    scanf("%d %d", &n, &m);      // 都市数 n, 道路数 m の入力
    if(n==0 && m==0) break;      // 両方とも 0 なら終了
    scanf("%d %d", &s, &g);      // 出発都市、目的都市の都市番号を入力
    l = 0;
    tmin = INF;                  // 速度 1 での目的都市到着時間を INF にセット
    for(i=1; i<=n; i++){
      al[i] = NULL;              // 各都市の隣接都市リストを初期化
      for(j=1; j<=n; j++){
        d[i][j] = c[i][j] = 0;   // 都市間道路の距離と制限速度を初期化
        for(k=1; k<=30; k++){
          t[i][j][k].st = t[i][j][k].hp = -1;   // 状態データの初期化
          t[i][j][k].from = i;
          t[i][j][k].city = j;
          t[i][j][k].speed = k;
          h[l++] = NULL;         // ヒープデータの初期化
        }
      }
    }

    for(i=0; i<m; i++){
      scanf("%d %d %d %d", &xi, &yi, &di, &ci); // 都市間道路の距離、
                                                // 制限速度の入力
      d[xi][yi] = d[yi][xi] = di;
      c[xi][yi] = c[yi][xi] = ci;
      gen_al(xi,yi);                            // 隣接都市リストを作成
    }

    nh = 0;              // ヒープのデータ数を 0 に初期化
    alp = al[s];         // 出発都市の隣接都市リスト
    while(alp){          // s から速度 1 で出発したときの各隣接都市の状態を
                         // ヒープに入れる
      ins(s, alp->city, 1, d[s][alp->city]);
      alp = alp->next;
    }
    search();            // 目的地に速度 1 で到着する最短時間を求める
    if(tmin == INF)      // 再短時間が INF のままなら到達不能
      printf("unreachable¥n");
    else
      printf("%lf¥n", tmin);     // 再短時間を出力
    for(i=1; i<=n; i++){         // 隣接都市リストに使用したメモリを解放
      alp = al[i];
      while(alp){
        aln = alp->next;
```

```
            free(alp);
            alp = aln;
          }
        }
      }
    }
```

答え合わせのプログラム例

```
/* result check program for ICPC2009 国内予選問題 D */
// filename = check.c
// Compile: cc -o check check.c -lm
// 使い方： ./check D0.ans D0.result  等

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

char buff1[256], buff2[256];
double gosa = 0.001;

void check(FILE *fd1, FILE*fd2)
{
  int flag=0;
  double d1, d2;
  int ret1, ret2;
  int line = 0;

  while(1){
    ret1 = fscanf(fd1, "%s", buff1);
    ret2 = fscanf(fd2, "%s", buff2);
    line++;
    if(ret1 == EOF || ret2 == EOF){
      if(ret1 != EOF || ret2 != EOF) flag = 1;
      if(flag)
        printf("not match at line %d¥n", line);
      else
        printf("OK¥n");
      break;
    }
    if(strcmp("unreachable", buff1)==0){
      if(strcmp("unreachable", buff2) != 0) flag = 1;
    }
    else {
      sscanf(buff1, "%lf", &d1);
      sscanf(buff2, "%lf", &d2);
      if(fabs(d1-d2) > gosa) flag = 1;
    }
    if(flag)
      printf("not match at line %d¥n", line);
  }
}

int main(int argc, char *argv[])
{
  FILE *fd1, *fd2;
  if(argc != 3){
    fprintf(stderr,"Usage: %s filename1 filename2¥n", argv[0]);
    exit(1);
  }
  fd1 = fopen(argv[1], "r");
```

```
  if(fd1 == NULL){
    fprintf(stderr, "Cannot open %s¥n", argv[1]);
    exit(2);
  }
  fd2 = fopen(argv[2], "r");
  if(fd2 == NULL){
    fprintf(stderr, "Cannot open %s¥n", argv[2]);
    fclose(fd1);
    exit(3);
  }
  check (fd1, fd2);
  exit(0);
}
```