P-III-4 Výlet do Švýcarska

Vzhledem k omezení na počet N měst uvedenému v zadání úlohy, není možné si vstup reprezentovat maticí velikosti $N \times N$. Uložíme si tedy raději vstup tak, že pro každé z měst vytvoříme seznam nejvýše 100 měst, do kterých z tohoto města vedou trasy. Pro jednoduchost budeme nazývat města, mezi kterými vede trasa, *sousední*.

Nejjednodušším řešením úlohy je zvolit každé z N měst jako počátek Tiborovy trasy, pak zvolit každé z nejvýše 100 sousedních měst jako druhé město trasy, každé z nejvýše 100 sousedních měst druhého města jako třetí město trasy a nakonec každé ze 100 sousedních měst třetího města jako čtvrté město trasy a pak ověřit, že mezi prvním a čtvrtým městem vede trasa. Toto řešení má sice lineární časovou složitost $\mathcal{O}(N)$, ale konstanta schovaná do „velkého $\mathcal{O}$“ je 100^4 (vyzkoušení 100^3 možností volby druhého, třetího a čtvrtého města a kontrola, že první a čtvrté město jsou sousední), takže toto řešení nemůže splnit časový limit pro tuto úlohu.

Rychlejší řešení bude založeno na následující myšlence. Pokud $ABCDA$ je čtyřdenní plán výletu, pak ABC a ADC jsou dvě dvojice na sebe navazujících tras. Jestliže je navíc plán $ABCDA$ nejdelší možný, musí ABC a ADC tvořit nejdelší a druhou nejdelší ze všech dvojic tras, které začínají v A a končí v C. Kdyby tomu tak nebylo, bylo by možné plán výletu ještě zlepšit.

Naše řešení tedy bude vypadat následovně. Nejdříve zafixujeme město A. Poté pro každé z měst sousedících s A projdeme všechny jeho sousedy C a pokud nalezená dvojice (na sebe navazujících tras) z A do C je největší nebo druhá největší dosud nalezená, uložíme její hodnotu do pomocného pole. V průběhu výpočtu navíc kontrolujeme, zda jsme nenalezli dvě dvojice na sebe navazujících tras z A do C, jejichž součet ohodnocení by byl lepší než součet ohodnocení dosud nejlepšího nalezeného plánu pro výlet. Pokud tomu tak je, nahradíme plán výletu těmito trasami.

Toto řešení vyžaduje pro každé z N měst prozkoumání nejvýše 100^2 dvojic (na sebe navazujících) tras, které z něj vedou. Je tedy rychlejší než první námi navrhované řešení. Na závěr si ještě vysvětleme, jak se vyhnout inicializaci pomocného pole pro ukládání dvojic tras mezi A a C (toto pole má velikost $\mathcal{O}(N)$) – spolu s hodnotou dvojic na sebe navazujících tras si do pole též uložíme číslo města A a pak snadno poznáme, zda hodnoty uložené v poli odpovídají právě zafixovanému městu A nebo městu A z některé z předchozích iterací.

Časová složitost námi navrženého řešení je $\mathcal{O}(D^2N)$, kde D je maximální počet sousedů města. Paměťová složitost je pak $\mathcal{O}(DN)$.

Na této myšlence jsou založeny následující dva programy: první v C++, druhý v Pascalu.

```
#include <stdio.h>
#include <vector>
using namespace std;

struct hrana {                          // hrana
    int to, v;                          // cílový vrchol a délka hrany
};

struct vrchol {                         // mezivýsledky pro jeden vrchol
    // do P[c] si zapamatujeme dvojice tras z vrcholu P[c].a do c
    int a;
    int b, abc;                         // nejlepší dvojice tras
    int d, adc;                         // druhá nejlepší
    vrchol() { b = abc = d = adc = -1000; }
};
vrchol P[100000];

int main(){
    FILE *f = fopen("swiss.in", "r"), *fo = fopen("swiss.out", "w");
    int m, n;
    vector <vector <hrana>> G;

    fscanf(f, "%i %i", &n, &m); n++;    // načteme vstup
    for (int i = 0; i < n; i++) G.push_back(vector <hrana>());
    for (int i = 0; i < m; i++) {
        int from, to, val;
        fscanf(f, "%i%i%i", &from, &to, &val);
        hrana e; e.v = val;
        e.to = from; G[to].push_back(e);
        e.to = to;   G[from].push_back(e);
    }

    int besta, bestb, bestc, bestd, bestsum=-1;
    for (int a = 0; a < n; a++) {
        int b, c, ab, bc;
        for (int j = 0; j < G[a].size(); j++) {
            b = G[a][j].to; ab = G[a][j].v;
            for (int k = 0; k < G[b].size(); k++) {
                c = G[b][k].to; bc = G[b][k].v;
                if (a == c) continue;
                if (a != P[c].a) {
                    P[c] = vrchol();
                    P[c].a = a;
                }
                if (ab+bc > P[c].abc) {
                    P[c].adc = P[c].abc; P[c].d = P[c].b;
                    P[c].abc = ab+bc; P[c].b = b;
                } else if (ab+bc > P[c].adc) {
                    P[c].adc = ab+bc; P[c].d = b;
                }
                if (P[c].abc+P[c].adc > bestsum) {
                    bestsum = P[c].abc + P[c].adc;
                    besta = a; bestb = P[c].b; bestc = c; bestd = P[c].d;
                }
            }
```

```
        }
    }
    if (bestsum < 0)
        fprintf(fo, "NEEXISTUJE\n");
    else
        fprintf(fo, "%d\n%d %d %d %d %d\n",
            bestsum, besta, bestb, bestc, bestd, besta);
    fclose(fo);
    return 0;
}
```

```
program swiss;
const
   MAXN = 10000;                        { maximální počet měst }
   MAXM = 1000000;                      { ... a tras }
type
   edge  = record                       { trasa: odkud, kam, délka }
              first, second, value : longint;
           end;
   dedge = record                       { cíl trasy: kam, délka }
              second, value : longint;
           end;
   path  = record                       { nalezené trasy }
              start          : longint; { počáteční město }
              first, second  : longint; { nejdelší trasa }
              fvalue, svalue : longint; { druhá nejdelší }
           end;
var
   m, n  : longint;                     { počet tras a měst }
   P     : array [1..MAXN] of path;     { nalezené trasy }
   edges : array [1..MAXM] of edge;     { trasy v pořadí ze vstupu }
   ec    : longint;
   pos   : array [0..MAXN] of longint;  { trasy z města i jsou uloženy ... }
   out   : array [1..MAXM] of dedge;    { ... v out[pos[i]..pos[i+1]-1] }
   deg   : array [1..MAXN] of longint;  { počet tras z každého města }
   first, second, value : longint;      { pomocné proměnné }
   i, v, w, x, e, f     : longint;
   fin, fout            : text;         { vstupní a výstupní soubor }
   best, a, b, c, d     : longint;  { nejlepší řešení je a-b-c-d-a s hodnotou best }

procedure SetEdge(v, w, value : longint);  { přidá do pole out další trasu }
begin
   out[pos[v]-deg[v]+1].second := w;
   out[pos[v]-deg[v]+1].value := value;
   dec(deg[v]);
end;

begin
   assign(fin,'swiss.in'); reset(fin);          { načítáme vstup }
   assign(fout,'swiss.out'); rewrite(fout);
   readln(fin, n, m);

   for i := 1 to n do deg[i] := 0;              { čteme trasy a počítáme deg[] }
   for i := 1 to m do begin
      readln(fin, edges[i].first, edges[i].second, edges[i].value);
```

```
      inc(deg[edges[i].first]);
      inc(deg[edges[i].second]);
      end;

   pos[0] := 0;  { naplánujeme si, kde v out[] budou trasy z kterého města }
   for v := 1 to n do
      pos[v] := pos[v-1] + deg[v];

   for i := 1 to m do begin                    { uložíme trasy do out[] }
      SetEdge(edges[i].first, edges[i].second, edges[i].value);
      SetEdge(edges[i].second, edges[i].first, edges[i].value);
      end;

   best := 0;
   for v := 1 to n do begin                    { projdeme všechna města v }
      for e := pos[v-1]+1 to pos[v] do begin    { a z každého všechny trasy vw }
         w := out[e].second;
         for f := pos[w-1]+1 to pos[w] do begin { a trasy wx navazující na vw }
            x := out[f].second;
            if x = v then continue;             { vrátili jsme se zpět }

            if v <> P[x].start then begin       { inicializace }
               P[x].start := v;
               P[x].fvalue := 0;
               P[x].svalue := 0;
               end;

            if out[e].value + out[f].value > P[x].fvalue then begin
               P[x].second := P[x].first;       { nejlepší dvojice }
               P[x].svalue := P[x].fvalue;
               P[x].first := w;
               P[x].fvalue := out[e].value + out[f].value;
               end
            else if out[e].value + out[f].value > P[x].svalue then begin
               P[x].second := w;                { druhá nejlepší dvojice }
               P[x].svalue := out[e].value + out[f].value;
               end;

            if (P[x].svalue > 0) and (P[x].fvalue + P[x].svalue > best) then begin
               best := P[x].fvalue + P[x].svalue;       { nejlepší čtveřice }
               a := v; b := P[x].first;
               c := x; d := P[x].second;
               end;
            end;
         end;
      end;

   if best = 0 then                            { zapíšeme výstup }
      writeln(fout, 'NEEXISTUJE')
   else begin
      writeln(fout, best);
      writeln(fout, a, ' ', b, ' ', c, ' ', d, ' ', a);
      end;
   close(fout);
end.
```

P-III-5 Záchranná akce

V popisu řešení bude L vždy značit délku posloupnosti příkazů, S šířku a V výšku mapy skladiště. *Stavem* robota budeme označovat čtveřici $\langle X, Y, P, Směr\rangle$, kde X, Y je robotova pozice ve skladišti, $1 \leq P \leq L$ je pozice následujícího příkazu v zadané sekvenci a *Směr* je směr natočení robota.

Počet stavů, ve kterých se robot může nacházet, je nanejvýš $4SVL$. Všimněte si, že jeho poloha ve skladišti, pozice v sekvenci příkazů a natočení jednoznačně určují, co bude robot dělat dál. Jakmile se robot ocitne v některém stavu podruhé, je jisté, že se zachová stejně jako po poslední návštěvě, a tedy se „zacyklí". Ještě si všimněte, že robot se může dostat i do cyklu, který neobsahuje jeho výchozí stav, a že to, že se robot vrátil do stejné *pozice ve skladišti* při jiné pozici v sekvenci příkazů, ještě nemusí znamenat zacyklení.

Jistě vás napadlo, že nejjednodušší by bylo pro každé volné pole plánu simulovat chování robota začínajícího na tomto poli až k cíli nebo do zacyklení. Zacyklení je možné detekovat několika způsoby: poznamenáváním prošlých stavů cesty, simulováním $4SVL$ příkazů, či simulováním dvou robotů různých rychlostí. Takovéto programy ale mají složitost $\mathcal{O}(S^2V^2L)$, protože nevylučují možnost, že robot pro velkou část startovních polí projde téměř každé pole řádově L-krát.* S chytře napsaným zjišťováním cyklů může ale i toto řešení získat část bodů i za některé velké vstupy (jako například náhodně a hustě rozmístěné překážky).

Snadná pomoc, řeknete si, stačí si pro každý ze $4SVL$ stavů pamatovat, jak dlouho trvá dostat se z něj mimo mapu, či zda se z něj robot zacyklí. Ať už si tabulku hodnot stavů budeme počítat „od konce" od stavů těsně před opuštěním mapy nebo rekurzivní funkcí z počátečních stavů, platí, že celkově na každý ze stavů provedeme pouze jeden výpočet. To nám dá algoritmy s časovou složitostí $\mathcal{O}(SVL)$, ale stejně vysoká bude i paměťová složitost – tabulka mezivýpočtů bude mít velikost až $4SVL$, což je pro daná omezení až $5 \cdot 10^8$ položek.

V právě popsaném řešení jsme si pamatovali hodnoty pro každý stav – což takhle ušetřit a pamatovat si pouze některé stavy? Zkusme se omezit na stavy, ve kterých je robot před provedením prvního příkazu programu (tedy stavy $\langle X, Y, P = 1, Směr\rangle$). Vybraných stavů je nanejvýš $4SV$ a následující vybraný stav lze z aktuálního vybraného stavu zjistit odsimulováním L příkazů.

Nejprve si popíšeme řešení pomocí rekurzivní funkce, která bude simulovat procházku ze zadaného stavu až k opuštění mapy či zacyklení, a zjištěné hodnoty pro vybrané stavy si zapamatuje. V tabulce $Kroku[X, Y, Směr]$ bude pro každý vybraný stav uloženo, zda je ještě neznámý, zda jsme jej již navštívili během aktuální procházky, zda se z něj robot zacyklí, nebo po kolika krocích z něj robot opustí mapu.

Funkce $KolikKroků(X, Y, Směr)$ vrátí buď počet kroků nutný k opuštění mapy, nebo -1, pokud zjistí cyklus. Po zavolání funkce se podíváme do tabulky *Kroků*

* Autoři znají konstrukci mapy $N \times N$ a posloupnosti příkazů délky N, kde výše uvedené programy běží v čase řádově N^4.

a pokud je hodnota již známa, vrátíme ji. Pokud je v tabulce uvedeno, že jsme tento stav již navštívili, vrátíme informaci o tom, že tato procházka vede do cyklu. Pokud není hodnota pro stav známa, poznamenáme do *Kroků*, že jsme skrz tento stav na této procházce šli, odsimulujeme L příkazů k dalšímu vybranému stavu $\langle X', Y', P' = 1, Směr' \rangle$ (nebo k okamžiku opuštění mapy) a zavoláme $k := KolikKroků(X', Y', Směr')$. Je-li vrácené k počet příkazů, uložíme součet $k + L$ do $Kroků[X, Y, Směr]$ a vrátíme tuto hodnotu; je-li to informace o zacyklení, uložíme ji a vrátíme.

První zavolání funkce *KolikKroků* na konkrétní stav spotřebuje čas nanejvýš $\mathcal{O}(L)$ a paměť $\mathcal{O}(1)$. Každé další zavolání stojí čas pouze $\mathcal{O}(1)$ a již nic nevolá, takže tento čas můžeme „naúčtovat“ při analýze časové složitosti volající funkci. Vzhledem k tomu, že prvních volání *KolikKroků* bude nanejvýš tolik, kolik je vybraných stavů, máme algoritmus s časovou složitostí $\mathcal{O}(SVL)$ a paměťovou složitostí $\mathcal{O}(SV)$. Jeho implementace v Pascalu je uvedena níže.

Navržený rekurzivní algoritmus je správně, ale používá příliš velký zásobník. Ukázková implementace proto používá vlastní zásobník na vybrané stavy navštívené během procházky a celá procedura *KolikKroků* je nerekurzivní. V programu níže jsme si ještě ušetřili práci sjednocením informace o tom, že stav je už navštívený v aktuální procházce, s informací, že stav vede do cyklu. V situaci, kdy narazíme na cyklus nebo navštívený stav, by se program měl chovat stejně a takto ušetříme vybírání zásobníku.

```
program akce;
const
  { pohyb směry sever (0), východ (1), jih (2) a západ (4) }
  dx : array [0..3] of longint = (0, 1, 0, -1);
  dy : array [0..3] of longint = (-1, 0, 1, 0);
  { omezení vstupu }
  MaxS = 500;
  MaxV = 500;
  MaxL = 500;

var
  Seq : array [1..MaxL] of char;  { sekvence příkazů }
  L : longint;                    { počet příkazů }
  Mapa : array [1..MaxS, 1..MaxV] of boolean; { mapa }
  S, V : longint;                 { rozměry plánu }
  Uteku : longint;                { počet políček, z nichž robot unikne }
  Kroku : array [1..MaxS, 1..MaxV, 0..3] of longint;
  { 0 => neznámo, -1 => vede do smyčky nebo zkoumáno aktuální procházkou }
  { >0 => počet tahů k úniku }

  { zásobník stavů pro procházku }
  ZasX, ZasY : array [1..MaxS * MaxV * 4] of longint;
  ZasSmer : array [1..MaxS * MaxV * 4] of longint;
  ZasVel : longint;               { aktuální velikost zásobníku }

{ simuluj jeden příkaz ze zadaného stavu }
procedure Prikaz(var X, Y, P, Smer : longint);
var NX, NY : longint;
```

```
begin
  NX := X; NY := Y;
  case Seq[P] of
     'L' : Smer := (Smer + 3) mod 4;
     'R' : Smer := (Smer + 1) mod 4;
     '+' : begin NX := NX + dx[Smer]; NY := NY + dy[Smer]; end;
     '-' : begin NX := NX - dx[Smer]; NY := NY - dy[Smer]; end;
  end;
  if (NX < 1) or (NY < 1) or (NX > S) or (NY > V) or Mapa[NX, NY] then
     begin X := NX; Y := NY; end;
  P := (P mod L) + 1;
end;

procedure KolikKroku(X0, Y0 : longint);
var
  i, X, Y, P, Smer, k: longint;
begin
  ZasVel := 0;
  k := 0; { příkazů do cíle }
  P := 1; { pozice v programu }
  X := X0; Y := Y0; Smer := 0; { X0, Y0, sever }
  { dokud nedojdeme do cíle a prozkoumáváme nové stavy }
  while (k = 0) and (Kroku[X, Y, Smer] = 0) do
  begin
    Kroku [X, Y, Smer] := -1;           { navštívený stav }
    inc(ZasVel);                        { uložit na zásobník }
    ZasX[ZasVel] := X;
    ZasY[ZasVel] := Y;
    ZasSmer[ZasVel] := Smer;
    { provést nanejvýš L kroků }
    for i := 1 to L do
    begin
      Prikaz(X, Y, P, Smer);
      if (X < 1) or (Y < 1) or (X > S) or (Y > V) then
         begin k := i; break; end;      { po k krocích v cíli }
    end;
  end;
  { došli jsme po L krocích na známou hodnotu? }
  if (k = 0) and (Kroku [X, Y, Smer] > 0) then
    k := Kroku [X, Y, Smer] + L;
  { pokud jsme došli do smyčky, můžeme nechat Kroku[X, Y, Smer] = -1 }
  { pro všechny vybrané stavy procházky }
  if k = 0 then exit;
  { pokud dojdeme na známou hodnotu nebo do cíle, vybereme zásobník }
  while ZasVel > 0 do
  begin
    Kroku [ZasX[ZasVel], ZasY[ZasVel], ZasSmer[ZasVel]] := k;
    k := k + L; dec(ZasVel);
  end;
end;

var
  i, j : longint;
  z : char;
  f : text;
```

```
begin
  { načtení vstupu }
  assign(f, 'akce.in');
  reset(f);
  readln(f, L);
  for i := 1 to L do
  begin
    read(f, z);
    Seq[i] := z;
  end;
  readln(f, S, V);
  for j := 1 to V do
  begin
    for i := 1 to S do
    begin
      read(f, z);
      Mapa[i, j] := (z = '.');
    end;
    readln(f);
  end;
  close(f);

  { prozkoumáme všechna políčka }
  Uteku := 0;
  for i := 1 to S do
    for j := 1 to V do
    begin
      if Mapa[i, j] and (Kroku[i, j, 0] = 0) then
          KolikKroku(i, j);
      if Kroku[i, j, 0] > 0 then
          inc(Uteku);
    end;

  { výstup }
  assign(f, 'akce.out');
  rewrite(f);
  writeln(f, Uteku);
  for j := 1 to V do
  begin
    for i := 1 to S do
    begin
      if i > 1 then write(f, ' ');
      if Kroku[i, j, 0] > 0 then
        write(f, Kroku[i, j, 0])
      else
        write(f, 0);
    end;
    writeln(f);
  end;
  close(f);
end.
```