P-III-4 Asfaltistán

Úlohu budeme řešit pomocí Dijkstrova algoritmu pro nalezení nejkratší cesty, jehož podrobný popis lze nalézt například v Programátorské kuchařce KSP (viz *http://ksp.mff.cuni.cz/tasks/20/cook4.html*). Vstupem Dijkstrova algoritmu je n měst propojených m silnicemi různých délek. Algoritmus v čase $\mathcal{O}((n+m)\log n)$ nalezne nejkratší cestu mezi dvěma zadanými městy, tj. takovou, že součet délek silnic této cesty je nejmenší možný. Poznamenejme, že existuje implementace tohoto algoritmu v čase $\mathcal{O}(m+n\log n)$, která využívá tzv. Fibonacciho haldy. Ve vzorovém řešení nám však bude stačit jednodušší implementace za použití standardní haldy.

V našem případě budou políčka čtvercové sítě představovat města a dvě sousední políčka spojíme „silnicí“ délky c_S, pokud se jejich nadmořské výšky liší nejvýše o 1 obří sáh. Dále musíme přidat silnice, které reprezentují mosty a tunely. Jednou z možností by bylo se z každého políčka vydat všemi čtyřmi možnými směry, dokud nenarazíme na políčko stejné výšky nebo okraj čtvercové sítě, a v prvním případě přidat silnici odpovídající mostu nebo tunelu. Takovéto řešení by vyžadovalo čas $\mathcal{O}(RS(R+S))$ na určení všech silnic. My si ukážeme, jak lze všechny silnice určit v čase $\mathcal{O}(RS)$.

Zamysleme se, jak lze určit možné mosty v jednom řádku čtvercové sítě. Řádek budeme procházet zleva doprava a budeme si v poli pamatovat ta políčka, která jsou od aktuálního políčka vlevo a všechna políčka mezi tímto políčkem a aktuálním políčkem mají menší výšku. Například pokud $S = 10$ a výšky políček v řádku jsou $10, 12, 6, 7, 8, 4, 3, 1, 9, 5$, pak po zpracování sedmi políček budeme mít v poli uloženo druhé, páté, šesté a sedmé políčko. Povšimněte si, že výšky políček v pořadí, v jakém jsou uloženy v poli, tvoří vždy klesající posloupnost.

Popišme si, jak lze hodnoty v poli udržovat a jak je využít k určení silnic, které představují mosty. Dokud je výška aktuálního políčka větší než výška posledního políčka v poli, políčka z pole budeme odebírat. Pokud se nyní výšky shodují, nalezli jsme dvojici políček, která lze spojit mostem. Tento most přidáme jako silnici mezi odpovídajícími dvěma políčky (její délka bude $c_S + \ell c_M$, kde ℓ je počet přemostěných políček) a poslední políčko v poli nahradíme aktuálním políčkem. Pokud je výška posledního políčka v poli vyšší, aktuální políčko do pole přidáme.

Je zřejmé, že výše uvedeným postupem nalezneme všechny možné mosty mezi políčky v jednom řádku. Protože každé políčko do pole jen jednou vložíme a jednou vyjmeme (a po porovnání buď políčka z pole vyjímáme, do pole přidáváme nebo měníme aktuální políčko), na zpracování jednoho řádku spotřebujeme čas $\mathcal{O}(S)$. Stejným způsobem pak můžeme určit silnice odpovídající tunelům.

Celkově tak na určení všech těchto silnic pro všechny řádky a sloupce je potřeba čas $\mathcal{O}(RS)$. Protože z každého políčka vede nejvýše osm silnic (čtyři na sousední políčka a čtyři tunely nebo mosty), je počet silnic lineární s počtem políček. Časová slo-

žitost Dijkstrova algoritmu aplikovaného na tento problém pak bude $\mathcal{O}(RS \log RS)$, a to je také časová složitost celého námi navrženého algoritmu. Paměťová složitost činí $\mathcal{O}(RS)$.

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

typedef unsigned long long ull; // 64-bitový typ

#define MAXRS 1000              // Maximální rozměr
#define MAXP (MAXRS*MAXRS+1)    // Maximální počet políček
#define MAXH 1000000            // Maximální výška
#define INFTY (1ULL<<62)        // Pro naše účely nekonečno

int R, S;                       // Parametry vstupu
ull cS, cM, cT;
int h[MAXRS][MAXRS];            // Vstup
int hint[MAXRS][MAXRS][4];      // Předpočítané mosty/tunely pro směr
                                // vlevo, vpravo, nahoru, dolů

// Přepočet mezi souřadnicemi a pořadovým číslem políčka
#define E(i,j) ((i)*MAXRS + (j))
#define D(i,j,c) i = (c)/MAXRS, j = (c)%MAXRS

ull best[MAXP];                 // Nejlepší zatím známé ohodnocení políčka
int prev[MAXP];                 // Předchůdce na nejkratší cestě

/** Halda pro Dijkstrův algoritmus **/

int heap[MAXP];                 // Halda sama (obsahuje čísla políček)
int hcnt;                       // Počet políček v haldě
int back[MAXP];                 // Pro každé políčko pozice v haldě
                                // (0=není tam)

// Prohození dvou prvků v haldě
static inline void swap(int i, int j)
{
  int z = heap[i];
  heap[i] = heap[j];
  heap[j] = z;
  back[heap[i]] = i;
  back[heap[j]] = j;
}

// Úprava (nastavení či snížení) ohodnocení políčka
static void heap_set(int i, ull new)
{
  best[i] = new;
  if (back[i])                  // Už je v haldě?
    i = back[i];
  else                          // Ne, musíme ho přidat
    {
      heap[++hcnt] = i;
      back[i] = hcnt;
      i = hcnt;
    }
  while (i > 1 && best[heap[i/2]] > best[heap[i]])
    {                           // Vybubláme nahoru
      swap(i, i/2);
```

```
      i /= 2;
    }
}

// Nalezení a smazání minima z haldy
static int heap_delmin(void)
{
  if (!hcnt)
    return -1;
  int old = heap[1];
  swap(1, hcnt);
  hcnt--;

  int i = 1;                    // Zabubláme dolů
  while (2*i <= hcnt)
    {
      int j = 2*i;
      if (j+1 <= hcnt && best[heap[j+1]] < best[heap[j]])
        j++;
      if (best[heap[i]] < best[heap[j]])
        break;
      swap(i, j);
      i = j;
    }

  back[old] = 0;
  return old;
}

/** Předvýpočet mostů a tunelů **/

// Pomocný zásobník
struct pt { int i, j, h; };
static struct pt stack[MAXRS+1];

static void precalc2(int d, int i0, int j0, int ri, int rj, int si, int sj)
{
  /*
   * Předpočítá hint[*][*][d].
   * Vyrazí z počátečního políčka (i0,j0),
   * (ri,rj) je "řádkový" směr, (si,sj) "sloupcový" směr.
   */
  while (i0 >= 0 && i0 < R && j0 >= 0 && j0 < S)
    {
      for (int s=-1; s<=1; s+=2)        // 1=tunely, -1=mosty
        {
          int i=i0, j=j0;
          int sp = 0;
          stack[0] = (struct pt) { 0, 0, MAXH+1 };      // Zarážka
          while (i >= 0 && i < R && j >= 0 && j < S)
            {
              int ht = s*h[i][j];
              while (ht > stack[sp].h)
                sp--;
              if (ht == stack[sp].h)
                {
                  int dist = abs(stack[sp].i - i) + abs(stack[sp].j - j);
                  if (dist > 1)
```

```
                hint[i][j][d] = dist;
              sp--;
            }
          stack[++sp] = (struct pt) { i, j, ht };
          i += si, j += sj;
        }
      }
    i0 += ri, j0 += rj;
  }
}

static void precalc(void)
{
  precalc2(0, 0, 0, 1, 0, 0, 1);
  precalc2(1, 0, S-1, 1, 0, 0, -1);
  precalc2(2, 0, 0, 0, 1, 1, 0);
  precalc2(3, R-1, 0, 0, 1, -1, 0);
}

// Obsloužení hrany z políčka x do (i,j) s cenou cost
static void try(int x, int i, int j, ull cost)
{
  // Nevede do autu?
  if (i < 0 || i >= R || j < 0 || j >= S)
    return;

  // Má povolený rozdíl výšek?
  int ii, jj;
  D(ii, jj, x);
  if (h[ii][jj] > h[i][j]+1 || h[ii][jj] < h[i][j]-1)
    return;

  // Dává lepší ohodnocení než to stávající?
  cost += best[x];
  int y = E(i,j);
  if (cost < best[y])
    {
      heap_set(y, cost);
      prev[y] = x;
    }
}

int main(void)
{
  // Čtení vstupu
  scanf("%d%d%lld%lld%lld", &R, &S, &cS, &cM, &cT);
  for (int i=0; i<R; i++)
    for (int j=0; j<S; j++)
      {
        scanf("%d", &h[i][j]);
        best[E(i,j)] = INFTY;
      }
  precalc();

  // Dijkstrův algoritmus
  heap_set(E(R-1,S-1), cS);
  int x;
  while ((x = heap_delmin()) >= 0)
```

```
    {
      int i, j;
      D(i, j, x);
      // Posun na sousední políčko
      try(x, i, j-1, cS);
      try(x, i, j+1, cS);
      try(x, i-1, j, cS);
      try(x, i+1, j, cS);
      // Stavíme tunel nebo most
      if (hint[i][j][0])                // Doleva
        try(x, i, j - hint[i][j][0],
            ((h[i][j-1] < h[i][j]) ? cM : cT) * (hint[i][j][0]-1) + cS);
      if (hint[i][j][1])                // Doprava
        try(x, i, j + hint[i][j][1],
            ((h[i][j+1] < h[i][j]) ? cM : cT) * (hint[i][j][1]-1) + cS);
      if (hint[i][j][2])                // Nahoru
        try(x, i - hint[i][j][2], j,
            ((h[i-1][j] < h[i][j]) ? cM : cT) * (hint[i][j][2]-1) + cS);
      if (hint[i][j][3])                // Dolů
        try(x, i + hint[i][j][3], j,
            ((h[i+1][j] < h[i][j]) ? cM : cT) * (hint[i][j][3]-1) + cS);
    }

  // Výpis cesty
  if (best[E(0,0)] < INFTY)
    {
      printf("%lld\n", best[E(0,0)]);
      int pos = 0;
      do
        {
          int i, j;
          D(i, j, pos);
          printf("%d %d\n", i, j);
          pos = prev[pos];
        }
      while (pos != E(R-1,S-1));
      printf("%d %d\n", R-1, S-1);
    }
  else
    puts("NEEXISTUJE");

  return 0;
}
```

P-III-5 Básník Honzík II

Pro zjednodušení nejdříve zkusíme vynechat požadavek, aby básnička začínala na stejný rým, kterým končí. Protože je pořadí slok pevně dané, lze takto zjednodušené zadání řešit pomocí dynamického programování. Můžeme totiž využít toho, že pokud máme množinu všech slok R_i, na které by báseň mohla končit, kdyby měla být dlouhá pouze i slok, pak jsme schopní snadno najít množinu všech slok R_{i+1}, na které může končit báseň dlouhá $i + 1$ slok.

Postup je jednoduchý – ze všech variant sloky $i + 1$, které označme jako V_{i+1}, vybereme ty, které začínají na stejný rým, kterým končí nějaká sloka v množině R_i.

Množina takto vybraných slok pak tvoří množinu R_{i+1}. Pokud se na sloky budeme dívat jako na dvojice přirozených čísel, pak lze tento fakt zapsat matematicky například takto:

$$R_{i+1} = \{(a, b) \mid (a, b) \in V_i \wedge \exists x : (x, a) \in R_i\}.$$

Z této myšlenky lze snadno odvodit řešení zjednodušené úlohy. Množinu R_1 získáme přímo jako množinu všech variant první sloky. Báseň pak postupně prodlužujeme, až dostaneme množinu R_N. Protože cílem úlohy je i vypsat vybrané varianty, musíme si zároveň s každou slokou s v R_{i+1} pamatovat, díky které variantě i-té sloky byla sloka s vybrána do množiny R_{i+1}. Pokud je takových variant více, lze vybrat libovolnou z nich. S pomocí této dodatečné informace můžeme snadno odzadu zrekonstruovat některou z hledaných posloupností.

Požadavek na cykličnost rýmů nám situaci mírně zkomplikuje. Nestačí totiž najít sloku v množině R_N takovou, že se rýmuje s některou první slokou. Jako příklad uvažme, že varianty první sloky jsou (1, 2) a (2, 3) a jediná varianta druhé sloky je (3, 1). V množině R_N pak bude jediná sloka (3, 1), ale jako první sloku nemůžeme vzít sloku (1, 2), neboť pak báseň nebude cyklická.

Můžeme si ale např. s každou slokou s z množiny R_i navíc pamatovat, na které sloky báseň musí začínat, aby mohla končit slokou s. Jinou možností je pustit výpočet zvlášť pro všechny varianty první sloky a pokud některá sloka z R_N končí na rým, kterým začíná aktuální první sloka, tak jsme našli řešení. Oba postupy mají stejnou časovou složitost. Liší se ale složitostí paměťovou, neboť v prvním případě si budeme muset ukládat až S_1-krát více informací pro rekonstrukci básničky.

Výsledný algoritmus je nyní již snadný. Pro každou variantu první sloky vyrobíme množinu R_1, což je jednoprvková množina obsahující vybranou první sloku. Použijeme výše popsaný algoritmus pro necyklickou báseň. Pokud se jímž v množině R_N nachází sloka, která končí na rým, jímž začíná aktuální první sloka, tak jsme nalezli řešení, které snadno zrekonstruujeme, vypíšeme a skončíme. V opačném případě pokračujeme další variantou první sloky.

Jak algoritmus efektivně implementovat? Množinu R_i si budeme reprezentovat přímočaře booleovským polem. To bude pro každou variantu i-té sloky určovat, zda se nachází v množině R_i, nebo ne. Pro výpočet R_{i+1} si pro každou variantu V_{i+1} zjistíme, zda existuje sloka v R_i, se kterou se rýmuje. Pokud ano, vložíme variantu do R_{i+1}. Časová složitost toho postupu je $S_i \cdot S_{i+1}$ a výsledná složitost celé úlohy je tedy $\mathcal{O}(S_1 \cdot N \cdot (\max S_i)^2)$. Zkusme ji ale ještě trochu zlepšit.

První vylepšení spočívá v tom, že zkusíme jeden krok algoritmu zrychlit na $\mathcal{O}(S_i + S_{i+1})$. To uděláme tak, že si v rámci předvýpočtu seskupíme sloky z V_i do skupin tak, aby v každé skupině byly pouze sloky končící na stejný rým. Nyní každou sloku s z V_{i+1} propojíme se skupinou, která odpovídá rýmu, na který s začíná, případně si zapamatujeme, že taková skupina neexistuje.

Potom můžeme jedním průchodem přes V_i pro každou skupinu zjistit, zda se v ní nachází aspoň jedna sloka, která patří do R_i. Druhým průchodem přes V_{i+1} pak

snadno zjistíme, které sloky patří do R_{i+1} a které ne. K tomu stačí podívat se na výsledek odpovídající skupiny z předchozího průchodu.

Předvýpočet provedeme tak, že si sloky z V_i setřídíme podle druhého rýmu a sloky z V_{i+1} podle prvního rýmu. Poté lineárním průchodem očíslujeme skupiny čísly 0, 1, ..., *počet skupin* a pro každou množinu V_i si např. v obyčejném poli zapamatujeme, do které skupiny jednotlivé varianty patří. Druhým lineárním průchodem pak pro sloky z V_{i+1} určíme, se kterou skupinou jsou propojené.

Časová složitost se změní na $\mathcal{O}(N \cdot \max S_i \cdot \log \max S_i)$ na předzpracování vstupu a $\mathcal{O}(S_1 \cdot N \cdot \max S_i)$ pro samotný výpočet.

Druhé urychlení (které ale pro dosažení plného počtu bodů nebylo nutné implementovat), spočívá v myšlence, že ve skutečnosti nezáleží na tom, kterou slokou začínáme. Pokud tedy začneme hledat báseň od sloky, která má nejméně variant, snížíme časovou složitost výpočtu na $\mathcal{O}(\min S_i \cdot N \cdot \max S_i)$.

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Sloka {
    Sloka() {}
    Sloka(int prvni, int druhy, int poradi)
        : PrvniRym(prvni), DruhyRym(druhy), Poradi(poradi),
        Skupina(-1), Propojeni(-1), Predchozi(-1) {}
    int PrvniRym;        // číslo prvního rýmu sloky
    int DruhyRym;        // číslo druhého rýmu sloky
    short int Poradi;    // pořadí sloky mezi variantami
    short int Skupina;   // skupina, do které sloka patří
    short int Propojeni; // skupina, s níž je sloka spojená; -1 pokud s žádnou
    short int Predchozi; // předchozí sloka (pro rekonstrukci řešení)
};

// porovnání podle prvního rýmu
bool PodlePrvniho(const Sloka &leva, const Sloka &prava)
{
    return leva.PrvniRym < prava.PrvniRym;
}

// porovnání podle druhého rýmu
bool PodleDruheho(const Sloka &leva, const Sloka &prava)
{
    return leva.DruhyRym < prava.DruhyRym;
}

int main()
{
    int N;
    scanf("%d", &N);

    std::vector<std::vector<Sloka>> V(N); // seznam všech variant

    // načtení variant
    int maxS = 0, min = 0;
    for (int i=0; i<N; i++) {
        int S;
```

```
    scanf("%d", &S);
    for (int j=0; j<S; j++) {
        int a, b;
        scanf("%d%d", &a, &b);
        V[i].push_back(Sloka(a, b, j));
    }
    // průběžně hledáne číslo sloky s nejmenším počtem variant
    if (V[i].size() < V[min].size()) min = i;
    // a také maximální velikost sloky
    if (S > maxS) maxS = S;
}

// otočíme vstup tak, abychom začínali slokou s nejmenším počtem variant
std::rotate(V.begin(), V.begin() + min, V.end());

// předzpracování
for (int i=0; i<N-1; i++) {
    // setřídíme varianty
    std::sort(V[i].begin(), V[i].end(), PodleDruheho);
    std::sort(V[i+1].begin(), V[i+1].end(), PodlePrvniho);

    // nejdříve zařadíme sloky z V[i] do skupin
    int rym = -1, skupina = -1;
    for (int j=0; j<V[i].size(); j++) {
        if (V[i][j].DruhyRym != rym) {
            V[i][j].Skupina = ++skupina;
            rym = V[i][j].DruhyRym;
        } else {
            V[i][j].Skupina = skupina;
        }
    }

    // nyní spojíme sloky z V[i+1] s odpovídajícími skupinami
    int k = 0, l = 0;
    while (k < V[i].size() && l < V[i+1].size()) {
        if (V[i][k].DruhyRym == V[i+1][l].PrvniRym) {
            V[i+1][l].Propojeni = V[i][k].Skupina;
            ++l;
        } else if (V[i][k].DruhyRym < V[i+1][l].PrvniRym) {
            ++k;
        } else {
            ++l;
        }
    }
}

std::vector<short int> skupiny(maxS, -1); // výsledky pro jednotlivé skupiny
std::vector<bool> R(maxS, false);         // množina R

for (int prvni=0; prvni<V[0].size(); prvni++) {
    std::fill(R.begin(), R.begin() + V[0].size(), false);
    R[prvni] = true; // postupně zkoušíme jednotlivé varianty první sloky

    for (int i=0; i<V.size()-1; i++) {
        std::fill(skupiny.begin(), skupiny.begin()+V[i].size(), -1);

        // spočítáme, které skupiny obsahují aspoň jednu sloku,
        // která je v R, a rovnou si jednu z nich zapamatujeme
        for (int k=0; k<V[i].size(); k++) {
```

```
                if (R[k])
                    skupiny[V[i][k].Skupina] = k;
            }

            std::fill(R.begin(), R.begin()+V[i+1].size(), false);

            // do R dáme ty sloky, které jsou spojeny se skupinou,
            // která obsahuje aspoň jednu sloku z minulé verze R
            for (int k=0; k<V[i+1].size(); k++) {
                if (V[i+1][k].Propojeni >= 0 && skupiny[V[i+1][k].Propojeni] >= 0) {
                    V[i+1][k].Predchozi = skupiny[V[i+1][k].Propojeni];
                    R[k] = true;
                }
            }
        }

        // nyní máme R spočítanou a zjistíme, zda lze navázat na první sloku
        for (int i=0; i<V[N-1].size(); i++) {
            if (R[i] && V[N-1][i].DruhyRym == V[0][prvni].PrvniRym) {

                // našli jsme řešení, nyní ho zrekonstruujeme
                std::vector<short int> vysledek(N);
                short int aktualni = i;
                for (int k = N-1; k >= 0; --k) {
                    // kvůli třídění slok se původní pořadí mohlo změnit
                    vysledek[k] = V[k][aktualni].Poradi;
                    aktualni = V[k][aktualni].Predchozi;
                }

                // výslednou posloupnost musíme otočit zpátky tak,
                // aby začínala opět původní první slokou
                std::rotate(vysledek.begin(), vysledek.begin() +
                    ((vysledek.size() - min) % vysledek.size()), vysledek.end());

                for (int k = 0; k < N; k++)
                    printf("%d\n", vysledek[k] + 1);
                return 0; // stačilo najít libovolné řešení, takže můžeme skončit
            }
        }
    }

    printf("NEEXISTUJE\n"); // pokud se program dostal až sem, tak nenalezl řešení
    return 0;
}
```