Problém čínského listonoše

Zpráva

Jan Hinterholzinger

Programátorské strategie
Fakulta aplikovaných věd
30. listopadu 2020

1 Definice

Problém čínského listonoše (The Chinese Postman Problem) se zaobírá nalezením nejkratší cesty v grafu takovou, která prochází všemi přímými hranami. Konec cesty může být definován dvěma způsoby. Tak že konečný vrchol musí být stejný jako počáteční vrchol, nebo tak, že cesta nemusí být ukončena v počátečním vrcholu.

2 Motivační příklad

Představme si zasněžené město. Silnice jsou zasněžené je třeba sníh odhrnout. Silničáři mají za úkol sníh ze všech silnic odhrnout. Aby se ovšem zbytečně neplatilo za spotřebovaném palivo, je dobré zjistit nejkratší možnou cestu, s kterou je možné splnit úkol.

3 Analýza problému

Tento problém je variací na Eulerovský tah (Eulerian problem), který popisuje situaci, kde každou hranou se projde právě jednou. Náš problém avšak dovoluje, aby se hranami procházelo víckrát, ale hledáme takový tah, který bude „nejlevnější“. Jako výstup tedy požadujeme cenu celkové cesty a postupný seznam cest, který reprezentuje cestu od počátečního bodu do koncového bodu.

4 Existující metody

4.1 Uzavřený Eulerovský tah

Graf obsahuje uzavřený Eulerovský tah pokud splňuje jednu z následujících podmínek:

- Neorientovaný graf obsahuje uzavřený Eulerovský tah (Eulerovský cyklus) pokud je souvislý a pokud všechny vrcholy jsou sudého stupně.
- Orientovaný graf obsahuje uzavřený Eulerovský tah (Eulerovský cyklus) pokud je silně souvislý a pro všechny vrcholy grafu platí, že počet vstupujících hran do vrcholu je stejný jako počet vystupujících hran z vrcholu.

Tyto podmínky lze zkontrolovat například prohledáváním grafu do hloubky (DFS). Pokud tyto podmínky splňuje vstupní graf, vystačíme si s algoritmem pro hledání Eulerovského cyklu.

[1] Algoritmus

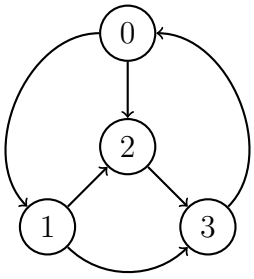
1. Zkontrolujeme, zda graf obsahuje uzavřený Eulerovský tah (cyklus).
2. Vybereme kterýkoli vrchol. Tento vrchol bude naším aktuálním vrcholem.
3. Dále opakujeme následující kroky dokud aktuální vrchol nemá žádné další sousedy a zároveň zásobník je prázdný.
    - Pokud aktuální vrchol nemá zádné další sousedy → přidáme jej do obvodu, odebereme vrchol zásobníku a nastavíme jej jako aktuální. Jinak (pokud má sousedy) → přidáme aktuální vrchol do zásobníku, zvolíme si souseda tohoto vrcholu, odstraníme hranu mezi aktuálním vrcholem a tímto sousedem a souseda nastavíme jako aktuální vrchol.

Z definice Eulerovského tahu je jasné, že se bude procházet každou hranou právě jednou a proto je cena celé trasy součtem ohodnocení všech hran (v případě neohodnoceného grafu součtem všech hran). Tímto se tedy jedná o „nejlevnější“ variantu problému. Jelikož je ale nepravděpodobné, že zadaný graf bude splňovat podmínky pro Eulerovský tah je proto nutné řešení poupravit, aby splňovalo zadaný problém.

5 [2]Zvolené řešení

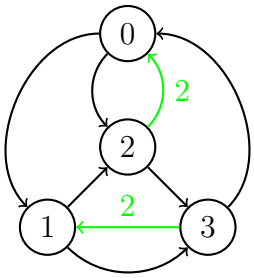
Jako vybrané řešení jsem vybral algoritmus zmíněný v [3] knize, který jsem nalezl na [4] stránce autora profesora Harolda Thimblebyho (odkaz v [3] knize nefungoval). Rád bych také algoritmus popisoval vedle příkladu. Řekněme, že máme graf se čtyřmi vrcholy propojené následujícími hranami: {(0,1), (0,2), (1,2), (1,3), (2,3), (3,0)} (viz. Obrázek 1)

Jelikož obecně se již nemusí jednat o Eulerovský graf, tak je třeba vyřešit problém, se přes různé hrany půjde několikrát. Označme si D^+ jako soubor vrcholů, kde převažují vycházející hrany a D^-, kde naopak převládají vstupní hrany. Rovnou si označíme i hodnotu $\delta(v) = d^+(v) - d^-(v)$, kde d^+ je počet



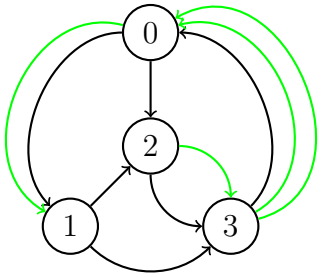
Obrázek 1: Vstupní graf

vycházejících hran a d^- počet vstupujících hran. Pro optimální řešení se musí nalézt mezi vrcholy D^- a D^+ další cesty takové, aby měly co nejnižší cenu. Takovou cestu si můžeme označit notací $i \rightsquigarrow j$, kde takový zápis označuje cestu s nejnižšími náklady z vrcholu i do j a nechť c_{ij} je její cena. Může se samozřejmě stát, že se mezi vrcholy vyskytne více cest. V takovém případě samozřejmě použijeme tu „nejlevnější“. V ukázkovém grafu existují dvě možnosti jak vybrat soubor nových cest. Jeden soubor jsou cesty $2 \rightsquigarrow 0$ a $3 \rightsquigarrow 1$ a druhá sada cest je $2 \rightsquigarrow 1$ a $3 \rightsquigarrow 0$. Z propočítání jejich cen víme, že jsou si rovny $(c_{20} + c_{31} = 2 + 2, c_{21} + c_{30} = 3 + 1)$, proto mohou oba soubory cest být použity pro optimální řešení. Nyní si můžeme ukázat upravený graf rozšířený o dvě nové cesty (viz Obrázek 2)



Obrázek 2: Upravený graf

Lze nyní vidět, že jsme graf doplnili o hrany, aby nyní splňoval vlastnost Eulerovského cyklu. Respektive, že orientovaný graf je eulerovský právě tehdy, je-li souvislý a každý jeho vrchol má vstupní stupeň rovný výstupnímu. Je potřeba si uvědomit, že nové cesty stále prochází původními vrcholy (cesta $3 \rightsquigarrow 1$ je vlastně stále cesta $3 \to 0 \to 1$). Původní cesty se pouze zdvojily a tím tedy „nebudujeme nové silnice“. Správně by tedy graf měl vypadat následovně (viz. Obrázek 3)



Obrázek 3: Zvojené cesty

Při sečtení hodnocení všech hran (v našem případě počet hran) získáme cenu cesty problému čínského listonoše. Nyní jen stačí jen graf projít algoritmem na uzavřený Eulerovský tah (cyklus) a nalézt konkrétní sekvenci vrcholů (nebo hran) udávající cestu.

Podrobnější popis algoritmu Nejprve si ukážeme jak vlastně bychom chtěli připravovanou třídu ovládat.

```
public static void main(...) {
CPP graf = new CPP(pocet_vrcholu)

graf.pridatHranu(nazev_hrany, poc_vrchol, konc_vrchol, cena_cesty);
...
...

graf.zpracovat();
graf.vypisCestu(pocatecni_vrchol);
vypsat("Cena: " + graf.cena())
}
```

Pokud budeme chtít přidat neorientovanou hranu, tak stačí přidat dvě orientované hrany s tím, že jedna z nich bude mít opačné pořadí vrcholů.

Výpis si představujeme přibližně následovně:

```
CPP z˜vrcholu 0
0 -> 1 přes a
1 -> 2 přes c
2 -> 3 před d
```

```
...
2 -> 0 přes x
Cena: 20
```

Nyní se přesuneme na samotnou třídu řešící tento problém. Nejprve se podíváme na seznam atributů, které budeme v průběhu výpočtu potřebovat.

```
int n; // počet vrcholů
int delta[]; // delta vrcholů
int neg[], pos[]; // soubory nevyvážených vrcholů
int arcs[][]; // matice sousednosti grafu
Vector label[][]; // vektory popisků hran
int f[][]; // opakované hrany
float c[][]; // ceny nejlevnějších hran nebo cest
String cheapestLabel[][]; // popisky nejlevnějších hran
boolean defined[][]; // zda je mezi vrcholy definovaná cena cesty (cesta)
int path[][]; // kostra grafu
float basicCost; // celkové náklady při průchodu každé hrany jednou
```

Počáteční inicializace bude provedena v konstruktoru:

```
public CPP(int pocetVrcholu) {
this.n = pocetVrcholu;

delta = new int[n];
defined = new boolean[n][n];
label = new Vector[n][n];
c = new float[n][n];
f = new int[n][n];
arcs = new int[n][n];
cheapestLabel = new String[n][n];
path = new int[n][n];
basicCost = 0;
}
```

Všechny prvky všech polí máme díky vlastnostem Java (ve které implementuji) nulové a nemusíme je tedy díky tomu zvlášť inicializovat. Nyní si ukážeme jak bude probíhat procedura přidání hrany.

```
void pridejHranu(String popis, int start, int cil, float cena) {
if(!defined[start][cil]) {
label[start][cil] = new Vector();
}
```

```
label[start][cil].addElement(popis); //ulozeni popisku hrany
basicCost += cena;
if(!defined[start][cil] || c[start][cil] > cena) {
c[start][cil] = cena;
cheapestLabel[start][cil] = popis;
defined[start][cil] = true;
path[start][cil] = cil;
}

arcs[start][cil]++;
delta[start]++;
delta[cil]--;
}
```

Celý algoritmus se rozloží na několik částí, které voláme v metodě colve().

```
void solve() {
nejmensiCenaCest();
zkontrolujValidnost();
najdiNevyvazenost();
najdiRealizovatelnost();
while (vylepseni())
;
}
```

Nyní použijeme Floyd-Warshallův algoritmus, který umí efektivně vyhledávat a zaznamenávat nejkratší cesty.

```
void nejmensiCenaCest() {
for( int k~= 0; k~< n; k++ )
for( int i = 0; i < n; i++ )
if( defined[i][k] )
for( int j = 0; j < n; j++ )
if( defined[k][j] && (!defined[i][j]
|| c[i][j] > c[i][k]+c[k][j]) ) { path[i][j] = path[i][k];
c[i][j] = c[i][k]+c[k][j];
defined[i][j] = true;
if( i == j && c[i][j] < 0) return;
// zastavení při nalezení negativního cyklu
}
}
```

Potřebujeme také kontrolovat, zda je graf silně souvislý s žádnými zápornými vahami cyklu. To je zajištěno kontrolování cen nejkratších cest. Pokud je totiš nějaká cena c nedefinována, tak graf není souvislý.

```
void zkontrolujValidnost() {
for( int i = 0; i < n; i++ ) {
for( int j = 0; j < n; j++ )
if( !defined[i][j] ) throw new Error("Graf není silně souvislý");
if( c[i][i] < 0) throw new Error("Graf obsahuje negativní cyklus");
}
}
```

Nyní nalezneme nevyvážené soubory vrcholů. Tyto soubory jsme si označili jako D^+ a D^- a jsou určovány pomocí hodnoty δ, kterou jsme si také dříve nadefinovali.

```
void najdiNevyvazenost() {
int nn = 0 , np = 0 ; // počet vrcholů se zápornou/kladnou deltou
for(int i = 0; i < n; i++)
if(delta[i] < 0) nn++;
else if(delta[i] > 0) np++;
neg = new int[nn];
pos = new int[np];
nn = np = 0 ;
for(int i = 0; i < n; i++) // inicializace
souborů
if(delta[i] < 0) neg[nn++] = i;
else if(delta[i] > 0) pos[np++] = i;
}
```

Provedeme doporučenou optimalizaci, kterou autor doporučuje.

```
void najdiRealizovatelnost() {
int delta[] = new int[n];
for (int i = 0; i < n; i++)
delta[i] = this.delta[i];
for (int u~= 0; u~< neg.length; u++) {
int i = neg[u];
for (int v~= 0; v~< pos.length; v++) {
int j = pos[v];
f[i][j] = -delta[i] < delta[j] ? -delta[i] : delta[j];
delta[i] += f[i][j];
delta[j] -= f[i][j];
```

```
}
}
}

boolean vylepseni() {
CPP residual = new CPP(n);
for (int u~= 0; u~< neg.length; u++) {
int i = neg[u];
for (int v~= 0; v~< pos.length; v++) {
int j = pos[v];
residual.pridejHranu(null, i, j, c[i][j]);
if (f[i][j] != 0)
residual.pridejHranu(null, j, i, -c[i][j]);
}
}
residual.nejmensiCenaCest();
for (int i = 0; i < n; i++)
if (residual.c[i][i] < 0) {
int k~= 0, u, v;
boolean kunset = true;
u~= i;
do {
v~= residual.path[u][i];
if (residual.c[u][v] < 0 && (kunset || k~> f[v][u])) {
k~= f[v][u];
kunset = false;
}
} while ((u~= v) != i);
u~= i;
do {
v~= residual.path[u][i];
if (residual.c[u][v] < 0)
f[v][u] -= k;
else
f[u][v] += k;
} while ((u~= v) != i);
return true;
}
return false;
}
```

Cenu nalezené optimální cesty získáme součtem všech ohodnocení původního grafu *basicCost* a celkovým ohodnocením všech přidaných cest *phi*().

```
float cena() {
return basicCost+phi();
}

float phi() {
float phi = 0;
for( int i = 0; i < n; i++ )
for( int j = 0; j < n; j++ )
phi += c[i][j]*f[i][j];
return phi;
}
```

Nyní nám nezbývá nic jiného než vypsat si průběh trasy. Tento kód si vytvoří kopie originálních dat, takže po výpisu nejsou smazána.

```
int findPath(int from, int f[][]) { // najde cestu mezi nevyváženými vrcholy
for (int i = 0; i < n; i++)
if (f[from][i] > 0)
return i;
return -1; // vrací -1 pokud cesta nebyla nalezena
}

void printCPT(int startVertex) {
int v~= startVertex;
int arcs[][] = new int[n][n];
int f[][] = new int[n][n];
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++) {
arcs[i][j] = this.arcs[i][j];
f[i][j] = this.f[i][j];
}
while (true) {
int u~= v;
if ((v~= findPath(u, f)) != -1) {
f[u][v]--; //smazani cesty
for (int p; u~!= v; u~= p)
{
p = path[u][v];
System.out.println(u + " -> " + v~+ " pres "
```

```
+ cheapestLabel[u][p]);
}
} else {
int bridgeVertex = path[u][startVertex];
if (arcs[u][bridgeVertex] == 0)
break; // finished if bridge already used
v~= bridgeVertex;
for (int i = 0; i < n; i++) // nalezení nepouzite hrany
if (i != bridgeVertex && arcs[u][i] > 0) {
v~= i;
break;
}
arcs[u][v]--; // snizit pocet "dvojitych" hran
System.out.println(u + " -> " + v~+ " pres "
+ label[u][v].elementAt(arcs[u][v]));
}
}
}
```

6 Experimenty a výsledky

Implementace proběhla v jazyce Java. Byla ve značné části použita implementace autora. Při testování testovacích grafů byly vždy správné. Vypsaná data grafu příkladu (Obrázek 1 na str. 3) jsou zaznamenána v tabulce 1 (počátek ve vrcholu 0). Výsledná data se zdají být správné.

7 Závěr

Algoritmus je funkční a rozhodně dosahuje optimálního řešení problému. Algoritmus má polynomiální složitost. Jedná se o velmi užitečný algoritmus v mnoho aplikacích. Při jeho použití je potřeba znalosti několika dalších známých algoritmů.

Přechod	z	do	přes
1.	0	2	b
2.	2	0	e
3.	3	0	f
4.	0	1	a
5.	1	2	c
6.	2	3	e
7.	3	1	f
8.	0	1	a
9.	1	3	d
10.	3	0	f
Celková cena	10		

Tabulka 1: Tabulka výstupních dat ze vstupního příkladu 1

Reference

[1] Ciubatii Dumitru. Graph magisc. http://www.graph-magics.com/articles/euler.php.

[2] Harold Thimbleby. The directed chinese postman. http://harold.thimbleby.net/cpp/SPAEcpp.pdf.

[3] S. Skiena. *The Alhorithm Design Manual*. Springer, New York, USA, 2008.

[4] Harold Thimbleby. The chinese postman. http://harold.thimbleby.net/cpp/index.html.