**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА**

**Лабораторна робота**

**Алгоритм Беллмана—Форда**

Підготував:

студент 4-го курсу

групи МІ-4

Василів Євген

2015

# Алгоритм Беллмана—Форда

|  |  |
| --- | --- |
| **Алгоритм Беллмана—Форда** | |
| **Клас** | Задача про найкоротший шлях (для зважених орієнтованих графів) |
| **Структура даних** | Граф |
| **Найгірша швидкодія** | O (|V| |E|) |
| **Просторова складність у найгіршому випадку** | O (|V|) |

**Алгоритм Беллмана—Форда** - алгоритм пошуку найкоротшого шляху в зваженому графі. Знаходить найкоротші шляхи від однієї вершини графа до всіх інших. На відміну від алгоритму Дейкстри, алгоритм Беллмана—Форда допускає ребра з негативною вагою. Запропоновано незалежно Річардом Беллманом і Лестером Фордом.

## Історія

Алгоритм носить ім'я двох американських вчених: Річарда Беллмана (Richard Bellman) і Лестера Форда (Lester Ford). Форд фактично винайшов цей алгоритм в 1956 році при вивченні іншої математичної задачі, підзадача якої звелася до пошуку найкоротшого шляху в графі, і Форд зробив начерк остаточного вирішення завдання цього алгоритма. Беллман в 1958 р. опублікував статтю, присвячену конкретно завданню знаходження найкоротшого шляху, і в цій статті він чітко сформулював алгоритм у тому вигляді, в якому він відомий нам зараз.

Алгоритм маршрутизації RIP (алгоритм Беллмана - Форда) був вперше розроблений в 1969 році, як основний для мережі ARPANET.

## Формулювання задачі

Дано орієнтований або неорієнтований граф  G(V, E) з ваговою функцією w:E\to \mathbb{R}. Вагою w(p) шляху p=\langle v_0, v_1, \dots, v_k \rangle назвемо суму ваг ребер, що входять в цей шлях: w(p)=\sum_{i=1}^k w(v_{i-1}, v_i).

Вхідними даними для алгоритму є граф G, вагова функція w, та стартова вершина s. Потрібно знайти найкоротші шляхи від вершини s до всіх вершин графа. Алгоритм Беллмана-Форда повертає логічне значення, яке вказує на те, чи міститься в графі цикл з негативною вагою, досяжний з витоку. Якщо такий цикл існує у графі G, алгоритм повідомляє, що найкоротших шляхів не існує. Якщо ж таких циклів немає, алгоритм видає найкоротші шляхи і їх вагу.

Сам алгоритм Форда-Беллмана представляє з себе кілька фаз. На кожній фазі проглядаються всі ребра графа, і алгоритм намагається справити релаксацію (relax, ослаблення) уздовж кожного ребра (u,v) ваги w(u,v). Релаксація вздовж ребра — це спроба поліпшити значенняv.d значенням v.u+w(u,v) . Фактично це означає, що ми намагаємося поліпшити значення для вершини v, користуючись ребром (u,v) і поточним значенням для вершини u . Стверджується, що достатньо |G.V|-1 фази алгоритму, щоб коректно порахувати довжини всіх найкоротших шляхів у графі (цикли негативної ваги відсутні). Для недосяжних вершин відстань v.d залишиться нескінченністю.

### Псевдокод

// Ініціалізація

**для** кожної вершини v \in G.V

v.d = \infty

v.\pi = NIL

s.d = 0

// Основна частина

**для** i = 1 **до** |G.V| - 1

**для** кожного ребра (u,v) \in G.E

**якщо** v.d > u.d + w(u, v)

v.d = u.d + w(u, v)

w.\pi = u // зберігаємо попередню вершину

// Перевірка на наявність циклів з від'ємною вагою

**для** кожного ребра (u,v) \in G.E

**якщо** v.d > u.d + w(u, v)

**повернути** ХИБА

**повернути** ІСТИНА

## Оцінка складності

Якщо граф заданий списком ребер: ініціалізація потребує  O(V) часу, кожен з  |V| - 1  проходів потребує  O(E) часу, прохід по усім ребрам для перевірки наявності негативного циклу займає  O(E) часу. Отже алгоритм працює за  O(VE) часу.

Якщо граф заданий матрицею суміжності, то алгоритм буде виконуватись за  O(E^3) часу.

Виконання лабораторної роботи

Лабораторну роботу виконував в середовищі Microsoft Visual C++ 2013, на двух-ядерному процесорі Intel(R) Core(TM) i5 CPU M450 1.7 GHz. Паралельну програму запускав на 4-ьох потоках.

**PARCS**

**Запуск на маєму комп'ютері для 2000 вершин**

|  |  |
| --- | --- |
| Кількість процесів | Час роботи(сек.) |
| 1 | 5.330 |
| 2 | 3.271 |
| 4 | 2.886 |

**Запуск в комп’ютерному класі для 2000 вершин**

|  |  |
| --- | --- |
| Кількість процесів | Час роботи(сек.) |
| 1 | 19.431 |
| 2 | 18.548 |
| 4 | 16.538 |

Код програми:

Graph.java

**package** Parallel;  
  
**import** java.io.Serializable;  
**import** java.util.Random;  
  
*/\*\*  
 \* Created by Yevgen on 16.05.2015.  
 \*/***public class** Graph **implements** Serializable {  
 **public static final int *INF*** = 1000000000;  
 **public int n**, **m**;  
 **public int**[] **from**, **to**, **w**;  
  
 **private void** InitArrays() {  
 **from** = **new int**[**m**];  
 **to** = **new int**[**m**];  
 **w** = **new int**[**m**];  
 }  
  
 **public void** RandInit() {  
 **n** = 2000;  
 **m** = **n** \* **n** / 2;  
 InitArrays();  
 Random r = **new** Random();  
 **for** (**int** i = 0; i < **m**; ++i) {  
 **from**[i] = r.nextInt(**n**);  
 **to**[i] = r.nextInt(**n**);  
 **w**[i] = r.nextInt(***INF***);  
 }  
 }  
  
 **public** Graph Filtered(**int** threads, **int** id) {  
 Graph res = **new** Graph();  
 res.**n** = **n**;  
 res.**m** = **m** / threads;  
 **if** (id < **m** % threads) { res.**m**++; }  
 res.InitArrays();  
 **for** (**int** i = id; i < **m**; i += threads) {  
 res.**from**[i / threads] = **from**[i];  
 res.**to**[i / threads] = **to**[i];  
 res.**w**[i / threads] = **w**[i];  
 }  
 **return** res;  
 }  
}

Distances.java

**package** Parallel;  
  
**import** java.io.Serializable;  
  
*/\*\*  
 \* Created by Yevgen on 16.05.2015.  
 \*/***public class** Distances **implements** Serializable {  
 **public int n**;  
 **public long d**[];  
 **void** Init(**int** N) {  
 **n** = N;  
 **d** = **new long**[**n**];  
 **d**[0] = 0;  
 **for** (**int** i = 1; i < **n**; ++i) { **d**[i] = (**long**)Graph.***INF*** \* (**long**)Graph.***INF***; }  
 }  
}

BellmanFord.java

**package** Parallel;  
  
**import** parcs.\*;  
  
**import** java.util.ArrayList;  
  
**public class** BellmanFord **implements** AM {  
 **public static void** main() {  
 task curtask = **new** task();  
 curtask.addJarFile(**"PARCS.jar"**);  
 (**new** BellmanFord()).run(**new** AMInfo(curtask, (channel) **null**));  
 curtask.end();  
 }  
  
 **public void** run(AMInfo info) {  
 **long** start = System.*currentTimeMillis*();  
  
 **int** num\_of\_threads = 1;  
 point[] P = **new** point[num\_of\_threads];  
 channel[] C = **new** channel[num\_of\_threads];  
 System.***out***.println(**"Random initialization..."**);  
 Graph g = **new** Graph();  
 g.RandInit();  
 **for** (**int** i = 0; i < num\_of\_threads; ++i) {  
 P[i] = info.createPoint();  
 C[i] = P[i].createChannel();  
 P[i].execute(OneStep.**class**.getName());  
 Graph cur\_g = g.Filtered(num\_of\_threads, i);  
 C[i].write(cur\_g);  
 }  
 Distances d = **new** Distances();  
 d.Init(g.**n**);  
 Distances cur\_d;  
 System.***out***.println(**"Waiting for result..."**);  
 **for** (**int** step = 0; step < g.**n** - 1; ++step) {  
 *//System.out.println("Step " + step);* **for** (**int** i = 0; i < num\_of\_threads; ++i) {  
 C[i].write(d);  
 }  
 **for** (**int** i = 0; i < num\_of\_threads; ++i) {  
 cur\_d = (Distances) C[i].readObject();  
 **for** (**int** j = 0; j < g.**n**; ++j) {  
 d.**d**[j] = Math.*min*(d.**d**[j], cur\_d.**d**[j]);  
 }  
 }  
 }  
 System.***out***.println(**"Result found."**);  
 System.***out***.println(d.**n**);  
 **for** (**int** i = 0; i < d.**n**; ++i) { System.***out***.print(d.**d**[i] + **" "**); }  
 System.***out***.println();  
 **long** timeSpent = System.*currentTimeMillis*() - start;  
 System.***out***.println(**"timeSpent = "** + timeSpent);  
 }  
}

OneStep.java

**package** Parallel;  
  
**import** parcs.AM;  
**import** parcs.AMInfo;  
  
**public class** OneStep **implements** AM {  
 **public void** run(AMInfo info) {  
 Graph g = (Graph)info.**parent**.readObject();  
 System.***out***.println(**"Graph read."**);  
 Distances d;  
 **for** (**int** step = 0; step < g.**n** - 1; ++step) {  
 d = (Distances)info.**parent**.readObject();  
 *//System.out.println("Distances read. Step " + step);* **for** (**int** i = 0; i < g.**m**; ++i) {  
 **int** v = g.**from**[i];  
 **int** to = g.**to**[i];  
 **int** w = g.**w**[i];  
 d.**d**[to] = Math.*min*(d.**d**[to], d.**d**[v] + w);  
 }  
 info.**parent**.write(d);  
 *//System.out.println("Distances wrote. Step " + step);* }  
 }  
}

Parallel.java

**package** Parallel;  
  
**public class** Parallel {  
 */\*\*  
 \** ***@param args*** *\*/* **public static void** main(String[] args) {  
 BellmanFord.*main*();  
 }  
}

**Висновок:** Як видно з результатів, при збільшенні кількості процесорів на яких виконується програма час зменшується (не залежно чи ми запускаємо на власному компрютері чи в комп’ютерному класі). Можна помітити, що час виконання в комп’ютерному класі значно більший за час виконання на персональному комп’ютері. Це повязано з тим, що для виконання алгоритму в комп’ютерному класі нам потрібно пересилати багато даних по мережі (яка є повільною порівнняно з пересиланням даних на персональному комп’ютері).