## [Thuật toán] Tìm đường đi ngắn nhất Dijkstra, Floyd

**Update 25/05/2014: Do một số góp ý của các bạn nên mình đã viết thêm 1 chương trình của thuật toán Dijkstra theo cấu trúc hàm và cũng nhân tiện chỉnh lại chút code cho sáng sủa và chính xác hơn ^^.**

**Update 27/09/2014: bổ xung code pascal của thuật toán tại đây:**[**http://ideone.com/c7J0dq**](http://ideone.com/c7J0dq)

**Nội dung**  
[Thuật toán Dijkstra](http://www.nguyenvanquan7826.com/2013/10/13/thuat-toan-tim-duong-di-ngan-nhat-dijkstra-floyd/" \l "dijkstra)  
[Thuật toán Floyd](http://www.nguyenvanquan7826.com/2013/10/13/thuat-toan-tim-duong-di-ngan-nhat-dijkstra-floyd/" \l "floyd)  
[Code nâng cao cho cả 2 thuật toán](http://www.nguyenvanquan7826.com/2013/10/13/thuat-toan-tim-duong-di-ngan-nhat-dijkstra-floyd/" \l "advance)

[**Update 14/06/2014: Chương trình mô phỏng thuật toán Dijkstra**](http://nguyenvanquan7826.com/2014/06/14/java-thuat-toan-mo-phong-thuat-toan-dijkstra-tim-duong-di-ngan-nhat/)

Trong bài viết này chỉ đề cập tới các thuật toán tìm đường đi ngắn nhất Dijkstra và Floyd, một số thuật ngử liên quan mình sẽ không giải thích hay định nghĩa, các bạn tự tìm hiểu trong sách hoặc trên mạng.

Bài toán đường đi ngắn nhất nguồn đơn là bài toán tìm một đường đi giữa hai đỉnh sao cho tổng các trọng số của các cạnh tạo nên đường đi đó là nhỏ nhất. Hay nói một cách toán học là:  
Cho đơn đồ thị liên thông, có trọng số G=(V,E). Tìm khoảng cách d(a,b) từ một đỉnh a cho trước đến một đỉnh b bất kỳ của G và tìm đường đi ngắn nhất từ a đến b.

Như tiêu đề bài viết, chúng ta sẽ tìm hiểu 2 thuật toán để giải quyết bằng cách sử dụng mà trận kề cảu đồ thị(chú ý ta xét trọng số của đồ thị là không âm).

Ma trận kề của đồ thị có n đỉnh là ma trận vuông G có số hàng số cột là n. G[i][j] là độ dài đường đi từ đỉnh i tới đỉnh j. Nếu xét đồ thị vô hướng thì G[i][j] = G[j][i]. Độ dài từ một đỉnh tới chính nó luôn là 0 (G[i][i] = 0). Nếu giữa 2 cạnh i và j của đồ thị không tồn tại đường đi thì G[i][j] = vô cùng (∞). Tuy nhiên khi biểu diễn trong máy tính thì giá trị ∞ được đặt là 1 hằng số rất lớn hoặc là tổng các giá trị trong ma trận (tổng độ dài các cạnh).

## 1. Thuật toán Dijkstra

Về thuật toán Dijkstra có 2 loại là tìm đường đi ngắn nhất từ 1 đỉnh nguồn tới 1 đỉnh đích và tìm đường đi ngắn nhất từ 1 đỉnh nguồn tới các đỉnh còn lại của đồ thị, và ở đây mình sẽ nói về loại thứ 1. (loại thứ hai bạn có thể tìm trên mạng hoặc chỉ cần thay đổi dòng **while (s[b] == 0) (dòng 43 của code 1 & dòng 76 của code 2) thành vòng for duyệt từ 0 đến n-1 là sẽ tìm được tất cả các đỉnh).**

– Dùng 1 mảng Len[] – Len[i] là khoảng cách ngắn nhất từ đỉnh a tới đỉnh i.

– Dùng 1 mảng S đánh dấu các đỉnh i đặc biệt (các đỉnh i mà thời điểm hiện tại thì đường đi từ a tới i là ngắn nhất).

– Dùng mảng P[] đánh dấu đường đi. P[j] = i nếu i là đỉnh đi trước j trong đường đi ngắn nhất.

– Đặt lại giá trị vô cùng cho các cặp đỉnh không có đường đi.

– Khởi tạo tất cả các đường đi từ a đên các đỉnh khác bằng vô cùng.

– Khởi tạo đường đi từ a đến chính a = 0.

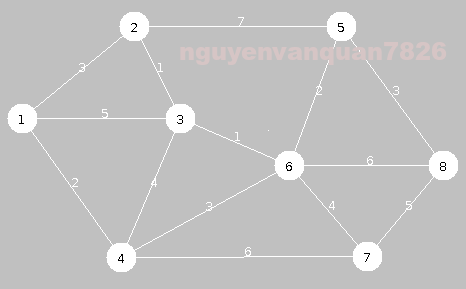
– Duyệt hết các đỉnh V của đồ thị

+ Tìm đỉnh i chưa nằm trong S mà đường đi từ a tới i là ngắn nhất để đưa vào S. Nếu không tìm được đỉnh nào nghĩa là đã duyệt hết các đỉnh có thể đi mà vẫn chưa thấy đỉnh đích => không thể đi được.

+ Nếu tìm được đỉnh i thì duyệt tất cả các đỉnh j chưa nằm trong S. Nếu Len[i] + G[i][j] < Len[j] (trong đó G[i][j] là khoảng cách từ đỉnh i tới đỉnh j) thì gán Len[j] = Len[i] + G[i][j]; và đánh dấu đường đi P[j] = i.

*Lưu ý: Do trong C, mảng bắt đầu từ 0. Do vậy các đỉnh khi tính toán thì sẽ tính từ đỉnh 0 đến đỉnh n-1. Tuy nhiên khi hiển thị ra thì vẫn phải là từ đỉnh 1 đến n và trong file input.inp thì đỉnh đầu và đỉnh cuối cũng sẽ được tính từ 1 đến n. Do đó trong code trước khi tính toán ta cần giảm đỉnh đầu và đỉnh cuối đi 1 đơn vị. Sau khi tính toán xong thì khi xuất kết quả lại cần tăng các đỉnh trong đường đi tìm được lên 1 đơn vị để hiển thị đúng (VD ta muốn tính đường đi từ đỉnh 4 đến đỉnh 8, thì đỉnh 4 tương ứng với vị trí thứ 3 trong mảng, đỉnh 8 ứng với vị trí thứ 7 nên ta cần giảm 4 xuống 3, 8 xuống 7 để tính toán. Khi tìm được đường đi, giả sử là 3 -> 5 -> 4 -> 7 thì phải in ra là 4 -> 6 -> 5 -> 8).*

Chúng ta sẽ đi thực hành với đồ thị sau theo 2 code là làm ngay trong main và thực hiện theo các hàm:



file **input.inp**: hàng đầu tiên thể hiện có 8 điểm, đi từ điểm 4 đến điểm 8. ma trận 8×8 ở dưới là ma trận kề của đồ thị.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 8 4 8  0 3 5 2 0 0 0 0  3 0 1 0 7 0 0 0  5 1 0 4 0 1 0 0  2 0 4 0 0 3 6 0  0 7 0 0 0 2 0 3  0 0 1 3 2 0 4 6  0 0 0 6 0 4 0 5  0 0 0 0 3 6 5 0 |

**\* Code ngay trong main**  
Code này đã được sửa và khắc phục một số lỗi từ [code ngày trước](http://ideone.com/zZmmog) (hoặc [link dự phòng](http://pastebin.com/GXWGGgx4)).

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 | #include <stdio.h>  #include <stdlib.h>  #define INP "input.inp"  #define OUT "output.out"    **int** main() {  **FILE** \*fi = **fopen**(INP, "r");  **FILE** \*fo = **fopen**(OUT, "w");  **int** n, a, b, i, sum = 0;        // nhap du lieu tu file input  **fscanf**(fi, "%d%d%d", &n, &a, &b);  **int** G[n][n];  **int** S[n], Len[n], P[n];        // nhap ma tran va tinh gia tri vo cung (sum)  **for** (i = 0; i < n; i++)  **for** (**int** j = 0; j < n; j++) {  **fscanf**(fi, "%d", &G[i][j]);              sum += G[i][j];          }      // dat vo cung cho tat ca cap canh khong noi voi nhau  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < n; j++) {  **if** (i != j && G[i][j] == 0)                  G[i][j] = sum;          }      }        /\* Do mang tinh tu G[0][0] nen can giam vi tri       di 1 don vi de tinh toan cho phu hop\*/      a--;      b--;    **for** (**int** i = 0; i < n; i++) {          Len[i] = sum;                   // khoi tao do dai tu a toi moi dinh la vo cung          S[i] = 0;                       // danh sach cac diem da xet          P[i] = a;                       // dat diem bat dau cua moi diem la a      }        Len[a] = 0;                         // dat do dai tu a -> a la 0        // tim duong di ngan nhat tu 1 dinh den moi dinh khac thi thay bang vong for:      //for (int k = 0; k < n; k++)  **while** (S[b] == 0) {                 // trong khi diem cuoi chua duoc xet  **for** (i = 0; i < n; i++)          // tim 1 vi tri ma khong phai la vo cung  **if** (!S[i] && Len[i] < sum)  **break**;            // i >=n tuc la duyet het cac dinh ma khong the tim thay dinh b -> thoat  **if** (i >= n) {  **printf**("done dijkstra\n");  **break**;          }    **for** (**int** j = 0; j < n; j++) {    // tim diem co vi tri ma do dai la min  **if** (!S[j] && Len[i] > Len[j]) {                  i = j;              }          }            S[i] = 1;                       // cho i vao danh sach xet roi    **for** (**int** j = 0; j < n; j++) {    // tinh lai do dai cua cac diem chua xet  **if** (!S[j] && Len[i] + G[i][j] < Len[j]) {                  Len[j] = Len[i] + G[i][j];      // thay doi len                  P[j] = i;                       // danh dau diem truoc no              }          }      }    **printf**("done dijkstra\n");        /\* Do ta dang tinh toan tu dinh 0 nen       muon hien thi tu dinh 1 thi can dung i + 1 de phu hop \*/    **printf**("start find path\n");    **if** (Len[b] > 0 && Len[b] < sum) {  **fprintf**(fo, "Length of %d to %d is %d\n", a + 1, b + 1, Len[b]);            // truy vet  **while** (i != a) {  **fprintf**(fo, "%d <-- ", i + 1);              i = P[i];          }  **fprintf**(fo, "%d", a + 1);      } **else** {  **fprintf**(fo, "khong co duong di tu %d den %d\n", a + 1, b + 1);      }    **printf**("done find path\n");    **fclose**(fi);  **fclose**(fo);    **printf**("done - open file output to see result\n");  **return** 0;  } |

l[ink dự phòng](http://pastebin.com/kN4MJZe4)

**\* Code theo từng hàm**  
Trong code theo hàm có hàm:  
– **readData** thực hiện đọc thông tin từ file input.  
– **dijkstra** thực hiện thuật toán  
– **back** thực hiện trả về chuỗi là đường đi tìm được  
– **outResult** thực hiện in ra file output kết quả

|  |  |
| --- | --- |
| 001  002  003  004  005  006  007  008  009  010  011  012  013  014  015  016  017  018  019  020  021  022  023  024  025  026  027  028  029  030  031  032  033  034  035  036  037  038  039  040  041  042  043  044  045  046  047  048  049  050  051  052  053  054  055  056  057  058  059  060  061  062  063  064  065  066  067  068  069  070  071  072  073  074  075  076  077  078  079  080  081  082  083  084  085  086  087  088  089  090  091  092  093  094  095  096  097  098  099  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181 | #include <stdio.h>  #include <stdlib.h>  #include <cstring>    #define INP "input.inp"  #define OUT "output.out"    // read data in file input  **int** readData(**int** \*\*\*G, **int** \*n, **int** \*a, **int** \*b) {  **FILE** \*fi = **fopen**(INP, "r");  **if** (fi == NULL) {  **printf**("file input not found!\n");  **return** 0;      }  **printf**("start read file\n");    **fscanf**(fi, "%d %d %d", n, a, b);        \*G = (**int** \*\*) **malloc**((\*n) \* **sizeof**(**int**));  **for** (**int** i = 0; i < \*n; i++) {          (\*G)[i] = (**int** \*) **malloc**((\*n) \* **sizeof**(**int**));  **for** (**int** j = 0; j < \*n; j++) {  **int** x;  **fscanf**(fi, "%d", &x);              (\*G)[i][j] = x;          }      }    **fclose**(fi);  **printf**("done read file\n");  **return** 1;  }    // thuat toan dijkstra  **int** dijkstra(**int** \*\*G, **int** n, **int** a, **int** b, **int** P[]) {        /\* Do mang tinh tu G[0][0] nen can giam vi tri       di 1 don vi de tinh toan cho phu hop\*/      a--;      b--;    **printf**("start dijkstra\n");    **int**\* Len = (**int** \*) **malloc**(n \* **sizeof**(**int**));  **int**\* S = (**int** \*) **malloc**(n \* **sizeof**(**int**));    **int** sum = 0;            // gia tri vo cung        // tinh gia tri vo cung (sum)  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < n; j++) {              sum += G[i][j];          }      }        // dat vo cung cho tat ca cap canh khong noi voi nhau  **for** (**int** i = 0; i < n; i++) {  **for** (**int** j = 0; j < n; j++) {  **if** (i != j && G[i][j] == 0)                  G[i][j] = sum;          }      }    **for** (**int** i = 0; i < n; i++) {          Len[i] = sum;       // khoi tao do dai tu a toi moi dinh la vo cung          S[i] = 0;           // danh sach cac diem da xet          P[i] = a;           // dat diem bat dau cua moi diem la a      }        Len[a] = 0;             // dat do dai tu a -> a la 0    **int** i;        // tim duong di ngan nhat tu 1 dinh den moi dinh khac thi thay bang vong for:      //for (int k = 0; k < n; k++)  **while** (S[b] == 0) {                 // trong khi diem cuoi chua duoc xet  **for** (i = 0; i < n; i++)          // tim 1 vi tri ma khong phai la vo cung  **if** (!S[i] && Len[i] < sum)  **break**;            // i >=n tuc la duyet het cac dinh ma khong the tim thay dinh b -> thoat  **if** (i >= n) {  **printf**("done dijkstra\n");  **return** 0;          }    **for** (**int** j = 0; j < n; j++) {    // tim diem co vi tri ma do dai la min  **if** (!S[j] && Len[i] > Len[j])                  i = j;          }            S[i] = 1;                       // cho i vao danh sach xet roi    **for** (**int** j = 0; j < n; j++) {    // tinh lai do dai cua cac diem chua xet  **if** (!S[j] && Len[i] + G[i][j] < Len[j]) {                  Len[j] = Len[i] + G[i][j];      // thay doi len                  P[j] = i;                       // danh dau diem truoc no              }          }      }  **printf**("done dijkstra\n");  **return** Len[b];  }    //  truy vet duong di  **void** back(**int** a, **int** b, **int** \*P, **int** n, **char** \*path) {        //char \*path = (char \*) malloc((n \* 10) \* sizeof(char));        /\* Do mang tinh tu G[0][0] nen can giam vi tri       di 1 don vi de tinh toan cho phu hop\*/      a--;      b--;    **printf**("start find path\n");    **int** i = b;  **int** point[n];   // danh sach cac dinh cua duong di  **int** count = 0;        /\* Do ta dang tinh toan tu dinh 0 nen       muon hien thi tu dinh 1 thi can dung i + 1 de phu hop \*/        point[count++] = i + 1;  **while** (i != a) {          i = P[i];          point[count++] = i + 1;      }    **strcpy**(path, "");  **char** temp[10];  **for** (i = count - 1; i >= 0; i--) {  **sprintf**(temp, "%d", point[i]);  **strcat**(path, temp);    **if** (i > 0) {  **sprintf**(temp, " --> ");  **strcat**(path, temp);          }      }    **printf**("done find path\n");  }    **void** outResult(**int** len, **char**\* path) {  **FILE** \*fo = **fopen**(OUT, "w");    **if** (len > 0) {  **fprintf**(fo, "\nLength of %c to %c is %d\n", path[0],                  path[**strlen**(path) - 1], len);      }    **fprintf**(fo, "path: %s\n", path);    **fclose**(fo);  }    **int** main() {  **int** \*\*G, n, a, b, len;    **if** (readData(&G, &n, &a, &b) == 0) {  **return** 0;      }    **char** \*path = (**char** \*) **malloc**((10 \* n) \* **sizeof**(**char**));  **int** P[n];        len = dijkstra(G, n, a, b, P);    **if** (len > 0) {          back(a, b, P, n, path);          outResult(len, path);      } **else** {  **char** \*path = (**char** \*) **malloc**((n \* 10) \* **sizeof**(**char**));  **sprintf**(path, "khong co duong di tu %d den %d\n", a, b);          outResult(len, path);      }    **printf**("done - open file output to see result\n");  **return** 0;  } |

[link dự phòng](http://pastebin.com/B7iUfbnD)

Nhìn code có vẻ hơi dài nhưng khi đọc hiểu rồi thì chả dài tẹo nào @@ =)).

## 2. Thuật toán Floyd

Thuật toán này cho phép chúng ta tìm đường đi ngắn nhất giữa mọi cặp đỉnh.

Nếu đỉnh k nằm trên đường đi ngắn nhất từ đỉnh i tới đỉnh j thì đoạn đường từ i tới k và từ k tới j phải là đường đi ngắn nhất từ i tới k và từ k tới j tương ứng. Do đó ta sử dụng ma trận A để lưu độ dài đường đi ngắn nhất giữa mọi cặp đỉnh.  
– Ban đầu ta đặt A[i,j] = C[i,j], tức là ban đầu A chứa độ dài đường đi trực tiếp (không đi qua đỉnh nào cả).  
– Sau đó thực hiện n lần lặp, sau lần lặp thứ k, ma trận A sẽ chứa độ dài đường đi ngắn nhất giữa mọi cặp đỉnh chỉ đi qua các đỉnh thuộc tập {1,2,..,k}. Như vậy, sau n lần lặp ta nhận được ma trận A chứa độ dài các đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị.

– Ký hiệu Ak là ma trận A sau lần lặp thứ k, tức là Ak[i,j] là độ dài đường đi ngắn nhất từ i đến j chỉ đi qua các đỉnh thuộc {1, 2,.., k}. Ak[i,j] được tính theo công thức như sau:    Ak[i,j] = min {Ak -1[i,j], Ak-1[i,k] + Ak-1[k,j]}.

– Trong quá trình lặp ta phải lưu lại vết đường đi, tức là đường đi ngắn nhất đi qua các đỉnh nào. Khi đó ta sử dụng mảng phụ P[nxn], trong đó P[i,j] lưu đỉnh k nếu đường đi ngắn nhất từ i đến j đi qua đỉnh k.  Ban đầu P[i,j]=0 với mọi i,j, vì lúc đó đường đi ngắn nhất là đường đi trực tiếp, không đi qua đỉnh nào cả.

Code thuật toán:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **void** Floyd (**int** a, **int** b)  {  **int** max = tongthiethai();  **for** (**int** i=0; i<n; i++)  **for** (**int** j=0; j<n; j++)          {  **if** (G[i][j])                  A[i][j] = G[i][j];  **else** A[i][j] = max;              P[i][j] = -1;          }    **for** (**int** k=0; k<n; k++)   // lap n lan      {  **for** (**int** i=0; i<n; i++)   // thay doi do dai duong di cua cac dinh  **for** (**int** j=0; j<n; j++)  **if** (A[i][j] > A[i][k] + A[k][j])                  {                      A[i][j] = A[i][k] + A[k][j];                      P[i][j] = k ;                  }      }  } |

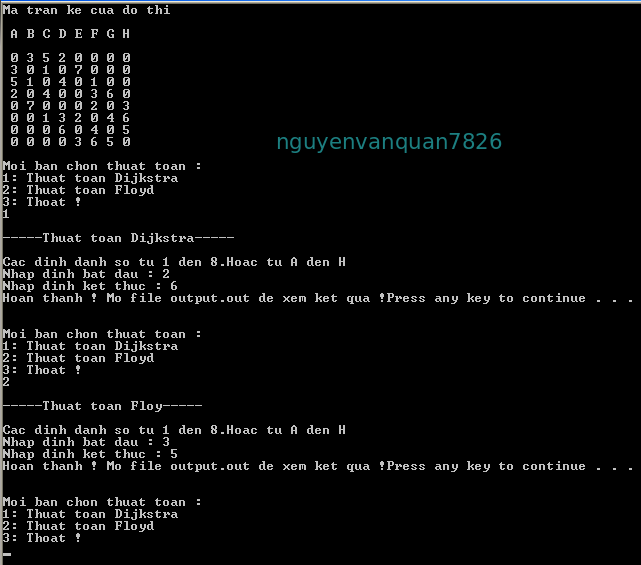
Cách xây dựng chương trình hoàn chỉnh hoàn toàn giống với thuật toán Dijkstra.

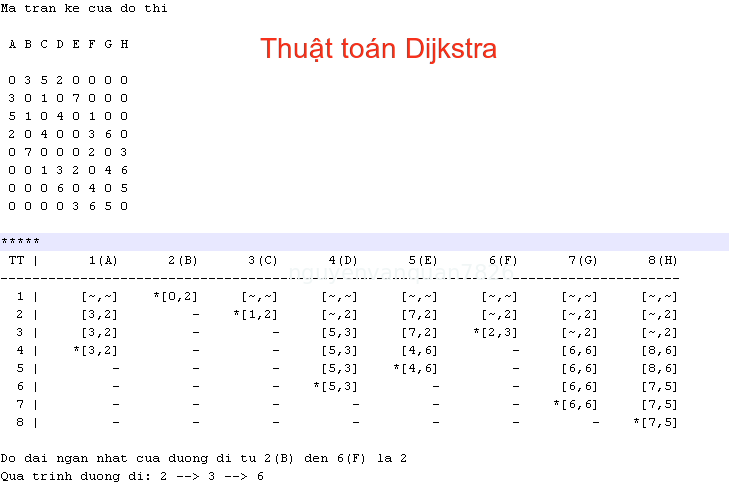
## Code nâng cao

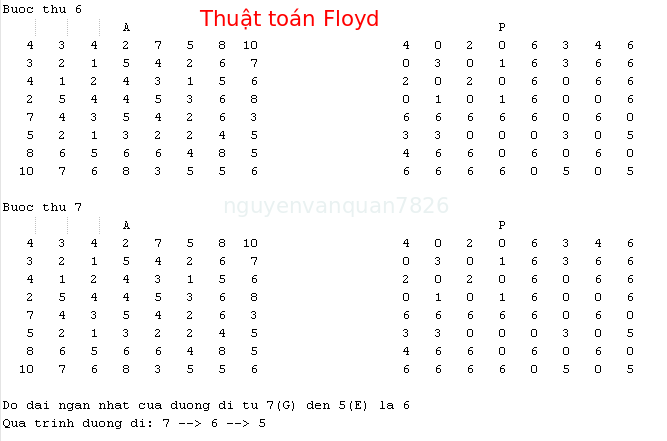
[**Đây là code cho phép chọn 1 trong 2 thuật toán trên và xuất ra file đúng theo quá trình làm như các kết quả trong hình bên dưới**](http://ideone.com/DpaGPW)[**hoặc link dự phòng**](http://pastebin.com/FiZzb3UH)

file input.inp:

*8  
A B C D E F G H  
0 3 5 2 0 0 0 0  
3 0 1 0 7 0 0 0  
5 1 0 4 0 1 0 0  
2 0 4 0 0 3 6 0  
0 7 0 0 0 2 0 3  
0 0 1 3 2 0 4 6  
0 0 0 6 0 4 0 5  
0 0 0 0 3 6 5 0*

**Menu console**  


**Output Dijkstra**  


**Output Floyd**  


Thuật toán Dijkstra tìm đường đi ngắn nhất từ một đỉnh-- Dijkstra Algorithm

January 12, 2016 in [Uncategorized](http://www.giaithuatlaptrinh.com/?cat=1) | [No comments](http://www.giaithuatlaptrinh.com/?p=764" \l "respond)

Trong bài này chúng ta sẽ tìm hiểu một thuật toán kinh điển tìm đường đi ngắn nhất từ một đỉnh tới mọi đỉnh khác trong đồ thị có hướng, có trọng số. Thuật toán này cũng áp dụng được cho đồ thị vô hướng nếu như ta coi mỗi cạnh vô hướng uvuvlà 2 cung có hướng ngược chiều nhau u→vu→v và v→uv→u với cùng trọng số của cạnh uvuv. Do đó, ta chỉ phát biểu thuật toán trong bài này cho đồ thị có hướng mà thôi.

Các khái niệm cơ bản về cạnh, đường đi, chu trình, etc., mình đã viết ở [bài trước](http://www.giaithuatlaptrinh.com/?p=553)nên ở đây mình không nhắc lại nữa. Ta gọi w:E⃗ →R+w:E→→R+ là một *hàm trọng số* gán cho mỗi cung (u→v)(u→v) một trọng số w(u→v)w(u→v) không âm. Đôi khi ta cũng có thể gọi trọng số của một cung là chiều dài của cung đó. *Chiều dài của một đường đi* P(u,v)P(u,v) giữa hai điểm được định nghĩa là tổng chiều dài của các cạnh nằm trên đường đi đó và ta kí hiệu là w(P(u,v))w(P(u,v)). Bài toán mà chúng ta sẽ tìm hiểu trong bài này như sau:

**Problem 1:**Cho một đồ thị có hướng G(V,E⃗ )G(V,E→), một hàm trọng số w:E⃗ →R+w:E→→R+ và một đỉnh ss. Tìm đường đi ngắn nhất từ ss tới mọi đỉnh trong GG.

Để đơn giản, ta sẽ dùng VV và EE lần lượt để chỉ số đỉnh và số cạnh của đồ thị. Năm 1959, Dijkstra[1] công bố *thuật toán Dijkstra* giải bài toán này trong thời gian O(V2)O(V2). Thuật toán Dijkstra là tối ưu với đồ thị dày (E=O(V2)E=O(V2)) nhưng với đồ thị thưa thì thuật toán này khá chậm. Tuy nhiên, ý tưởng của Dijkstra sau này đã được cải tiến [3] và có thời gian chạy O(E+VlogV)O(E+Vlog⁡V) . Trong bài này, chúng ta sẽ tìm hiểu một phiên bản thực thi thuật toán đó sử dụng [Heap nhị phân](http://www.giaithuatlaptrinh.com/?p=736) trong thời gian O((V+E)logV)O((V+E)log⁡V). Cuối bài, mình sẽ thảo luận ngắn gọn thuật toán O(E+VlogV)O(E+Vlog⁡V) với [Fibonacci Heap](https://en.wikipedia.org/wiki/Fibonacci_heap) của Fredman và Tarjan.

**Theorem 1:**Thuật toán Dijkstra tìm đường đi ngắn nhất từ một đỉnh tới mọi đỉnh khác trong đồ thị với thời gian O((V+E)logV)O((V+E)log⁡V).

**Tính chất của đường đi ngắn nhất**

Gọi δ(s,v)δ(s,v) là khoảng cách ngắn nhất từ ss tới vv trong GG và SP(s,v)SP(s,v) là một đường đi ngắn nhất bất kì từ ss tới vv. Ta có:

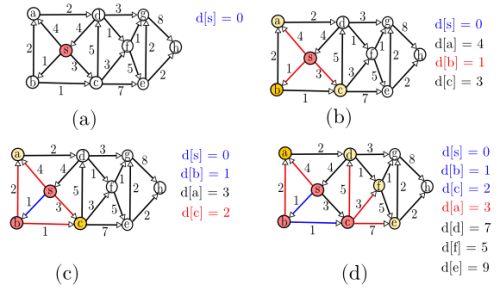
**Lemma 1:**Đường đi con của SP(s,v)SP(s,v) từ ss tới một đỉnh u∈SP(s,v)u∈SP(s,v) cũng là đường đi ngắn nhất từ ss tới uu.

**Chứng minh:** Gọi P(s,u)P(s,u) là đường đi con của SP(s,v)SP(s,v) từ ss tới uu. Giả sử tồn tại một đường đi khác P′(s,u)P′(s,u) sao cho w(P′(s,u))w(P′(s,u)) < w(P(s,u))w(P(s,u)). Thay thế đường đi P(s,u)P(s,u) trong SP(s,v)SP(s,v) bằng đường đi P′(s,u)P′(s,u) , ta sẽ thu được đường đi khác từ ss tới vv với chiều dài nhỏ hơn, trái ngược với định nghĩa SP(s,v)SP(s,v) là đường đi ngắn nhất.

**Thuật toán Dijkstra**

**Experimental Thought** Trước hết ta xét ss và các hàng xóm của nó, gọi là tập N(s)N(s). Gọi uu là một hàng xóm với độ dài cạnh w(s→u)w(s→u) nhỏ nhất. Khoảng cách ngắn nhất từ ss tới uu là bao nhiêu? Câu trả lời chính là w(s→u)w(s→u), vì nếu không, đường đi ngắn nhất từ ss tới uu đó phải đi qua một hàng xóm khác của ss, và do đó, sẽ dài hơn w(s→u)w(s→u). Từ đó ta suy ra δ(s,u)=w(s→u)δ(s,u)=w(s→u). Xem hình (b) ở hình dưới đây. Đỉnh được tô màu vàng đậm (đỉnh bb) là đỉnh có trọng số cạnh từ ss nhỏ nhất.

Tiếp theo ta xét tập các hàng xóm của cả ss và uu (uu chính là bb trong hình minh họa (c)), gọi là tập N(s,u)N(s,u). Với mỗi v∈N(s,u)v∈N(s,u), ta gọi d[v]=min(w(s→v),δ(s,u)+w(u→v))d[v]=min(w(s→v),δ(s,u)+w(u→v)). Dễ thấy nếu d[v]=w(s→v)d[v]=w(s→v), độ dài cạnh w(s→v)w(s→v) sẽ nhỏ hơn độ dài đường đi s→u→vs→u→v, và ngược lại. Trong số các đỉnh của tập N(s,u)N(s,u), gọi vv là đỉnh có d[v]d[v] nhỏ nhất, i.e, v=argminx∈N(s,u)(d[x])v=argminx∈N(s,u)(d[x]). Câu hỏi cũ vẫn là khoảng cách ngắn nhất từ ss tới vv là bao nhiêu? Dễ dàng thấy rằng δ(s,v)=d[v]δ(s,v)=d[v] với cùng lí do như trên. Xem hình (c) trong hình dưới đây. Đỉnhh được tô màu hồng đậm là các đỉnh đã tìm được đường đi ngắn nhất. Đỉnh cc là đỉnh có d[.]d[.] nhỏ nhất, do đó, d=δ(s,c)d=δ(s,c).

Tiếp theo ta lại xét N(s,u,v)N(s,u,v), blah. blah...(Xem hình (d)(d)) Cứ tiếp tục lặp lại như vậy, ta sẽ tìm ra được đường đi ngắn nhất (chính xác phải là độ dài của đường đi ngắn nhất) từ ss tới mọi đỉnh trong đồ thị. Đó chính là tư tưởng của thuật toán Dijkstra.  


Giả mã "thô" của thủ tục trên như sau:

Dijkstra(G(V,E⃗ ),w,sG(V,E→),w,s):   
    **for** v←1v←1 to VV  
        d[v]←+∞d[v]←+∞  
    d[s]←0d[s]←0  
    **for** every neighbor vv of ss  
        d[v]←w(s→v)d[v]←w(s→v)  
    B←V∖{s}B←V∖{s}         ≪B≪B*contains all vertices except*s≫s≫  
    **repeat**  
        u←argminx∈Bd[x]u←argminx∈Bd[x]  
        B←B∖{u}B←B∖{u}         ≪d[u]=δ(s,u)≫≪d[u]=δ(s,u)≫  
        **for** every neighbor vv of uu  
            d[v]←min(d[v],d[u]+w(u→v))d[v]←min(d[v],d[u]+w(u→v))  
    **until** B=∅B=∅

Ta có:

**Theorem 2:**Với mỗi đỉnh uu, giá trị d[u]d[u] sau khi thực hiên Dijkstra(G(V,E⃗ ),w,sG(V,E→),w,s) là đường đi ngắn nhất từ ss tới uu.

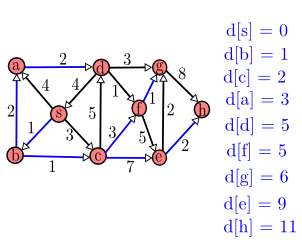
**Chứng minh** Nhận thấy mỗi bước trong vòng lặp, ta sẽ lấy ra đúng một đỉnh khỏi tập BB, do đó vòng lặp này thực hiện n−1n−1 lần, trong đó n=Vn=V là số đỉnh. Gọi u1,u2,…,un−1u1,u2,…,un−1 là thứ tự các đỉnh được lấy ra khỏi BB theo thuật toán trên. Theo thảo luận ở phần experimental thought, d[u1]=δ(s,u1)d[u1]=δ(s,u1) do u1u1 là đỉnh có w(s→u1)w(s→u1) nhỏ nhất trong số các hàng xóm của ss.

Gọi di[v]di[v] là giá trị d[v]d[v] ngay trước vòng lặp ii của thuật toán. Ta có di[v]di[v] chính là chiều dài của đường đi ngắn nhất từ ss tới vv đi qua một hoặc nhiều (hoặc 0) điểm trong tập {u1,u2,…,ui−1}{u1,u2,…,ui−1}. Vì uiui là đỉnh được lấy ra khỏi BB ở bước thứ ii, ta có:

di[ui]≤di[uj]di[ui]≤di[uj] với mọi i+1≤j≤n−1(1)i+1≤j≤n−1(1)

Giả sử di[u]≠δ(s,ui)di[u]≠δ(s,ui), đường đi ngắn nhất từ ss tới uiui, SP(s,ui)SP(s,ui), sẽ phải đi qua một đỉnh nào đó không thuộc {u1,…,ui−1}{u1,…,ui−1}. Gọi xx là điểm đầu tiên trên đường đi từ ss tới uu thỏa mãn tính chất đó. Theo Lemma 1, đường đi con của SP(s,ui)SP(s,ui)từ ss tới xx sẽ là đường đi ngắn nhất từ ss tới xx đi qua các điểm {u1,…,ui−1}{u1,…,ui−1}. Do đó di[x]=δ(s,x)di[x]=δ(s,x). Từ đó ta suy ra di[x]<di[u]di[x]<di[u] vì δ(s,x)≤w(SP(s,ui))<di[u]δ(s,x)≤w(SP(s,ui))<di[u]. Điều này trái với bất đẳng thức (1)(1).

Áp dụng thuật toán Dijkstra với hình trên, ta thu được (cây) đường đi ngắn nhất trong hình sau:



Các cung màu xanh là các cung của cây đường đi ngắn nhất. thứ tự các đinh được lấy ra từ BB sẽ có chiều tăng dần của khoảng cách δ(s,.)δ(s,.).

**Thực thi thuật toán Dijkstra**

Bây giờ ta sẽ thực thi thuật toán trong giả mã ở trên. Như đã phân tích, vòng lặp repeat/until thực hiện V−1V−1 lần lặp. Hai thao tác tốn kém nhất của mỗi lần lặp là:

1. Lấy đỉnh uu có giá trị d[u]d[u] nhỏ nhất ra khỏi BB
2. Cập nhật nhãn d[v]d[v] của các hàng xóm vv của uu.

Ta có thể thực thi tập hợp BB mảng. Do đó, thao tác đầu tiên có thể thực hiện trong thời gian O(V)O(V) còn thao tác thứ 2 có thể thực hiện trong thi gian O(deg(u))O(deg⁡(u)) trong đó deg(u)deg⁡(u) là bậc của đỉnh uu trong GG (nếu sử dụng cấu trúc danh sách kề). Như vậy, ta có thể thực thi thuật toán trong thời gian:

O(V2+∑u∈Gdeg(u))=O(V2+E)=O(V2)(2)O(V2+∑u∈Gdeg⁡(u))=O(V2+E)=O(V2)(2)

Tuy nhiên, nếu biểu diễn BB bằng một [Heap nhị phân](http://www.giaithuatlaptrinh.com/?p=736), thao tác đầu tiên chính là **ExtractMin** và thao tác thứ 2 là **DecreaseKey**. Do mỗi thao tác **ExtractMin** và**DecreaseKey** mất thời gian O(logn)O(log⁡n), thời gian để thực hiện thuật toán Dijkstra với Heap nhị phân là:

O(VlogV+∑u∈Glog(V)deg(u))=O((V+E)logV)(3)O(Vlog⁡V+∑u∈Glog⁡(V)deg⁡(u))=O((V+E)log⁡V)(3)

Từ đó ta suy ra Theorem 1.

Giả mã của thuật toán:

DijkstraHeapA(G(V,E⃗ ),w,sG(V,E→),w,s):   
    **for** v←1v←1 to VV  
        d[v]←+∞d[v]←+∞  
    d[s]←0d[s]←0  
    **for** every neighbor vv of ss  
        d[v]←w(s→v)d[v]←w(s→v)  
    B←B← BuildHeap(V∖{s})(V∖{s})         ≪B≪B*contains all vertices except*s≫s≫  
    **repeat**  
        u←u← ExtractMin(B)(B)         ≪d[u]=δ(s,u)≫≪d[u]=δ(s,u)≫  
        **for** every neighbor vv of uu  
            d[v]←min(d[v],d[u]+w(u→v))d[v]←min(d[v],d[u]+w(u→v))  
            DecreaseKey(B,v,d[v])(B,v,d[v])  
    **until** B=∅B=∅

**Remark** Nếu ta biểu diễn BB bằng [Fibonacci Heap](https://en.wikipedia.org/wiki/Fibonacci_heap), thao tác **ExtractMin** có thời gian O(logn)O(log⁡n) (khấu trừ) và thao tác **DecreaseKey** có thời gian O(1)O(1). Do đó, tổng thời gian của thuật toán Dijkstra với Fibonacci Heap là:

O(VlogV+∑u∈Gdeg(u))=O((VlogV+E)(4)O(Vlog⁡V+∑u∈Gdeg⁡(u))=O((Vlog⁡V+E)(4)

Để in ra đường đi ngắn nhất, mỗi khi cập nhật lại giá trị d[v]←min(d[v],d[u]+w(u→v))d[v]←min(d[v],d[u]+w(u→v)) trong thuật toán trên, ta sẽ đánh dấu uu là hàng xóm làm thay đổi nhãn d[v]d[v] của vv. Ta sẽ dùng một mảng PP và đánh dấu P[v]=uP[v]=u. Như vậy, thuật toán cuối cùng như sau:

DijkstraHeapB(G(V,E⃗ ),w,sG(V,E→),w,s):   
    **for** v←1v←1 to VV  
        d[v]←+∞d[v]←+∞  
    d[s]←0d[s]←0  
    P[s]←sP[s]←s  
    **for** every neighbor vv of ss  
        d[v]←w(s→v)d[v]←w(s→v)  
        P[v]←sP[v]←s  
    B←B← BuildHeap(V∖{s})(V∖{s})         ≪B≪B*contains all vertices except*s≫s≫  
    **repeat**  
        u←u← ExtractMin(B)(B)         ≪d[u]=δ(s,u)≫≪d[u]=δ(s,u)≫  
        **for** every neighbor vv of uu  
            **if** d[v]d[v] > d[u]+w(u→v))d[u]+w(u→v))  
                d[v]←d[u]+w(u→v)d[v]←d[u]+w(u→v)  
                P[v]←uP[v]←u  
            DecreaseKey(B,v,d[v])(B,v,d[v])  
    **until** B=∅B=∅

FindReverseShortestPath(G(V,E⃗ ),w,s,tG(V,E→),w,s,t):   
    DijkstraHeapB(G,w,s)(G,w,s)  
    print tt  
    **while** P[t]≠tP[t]≠t  
        t←P[t]t←P[t]  
        print tt

Code C với biểu diễn ma trận kề:

[+ expand source](http://www.giaithuatlaptrinh.com/?p=764)

**Nhắc lại Heap nhị phân**

Chúng ta sẽ phải sửa đổi lại một chút mã của [Heap nhị phân](http://www.giaithuatlaptrinh.com/?p=736) trong bài trước để áp dụng vào thuật toán Dijkstra. Trong bài trước, chúng ta sử dụng mảng H[1,2,…,n]H[1,2,…,n] để thực thi Heap, trong đó phần tử H[i]H[i] lưu **giá trị khóa** tương ứng với nút thứ ii. Do đó, khi cập nhật một giá trị khóa sử dụng **DecreaseKey** ta cần phải có địa chỉ của nút cần cập nhật và giá trị khóa mới của nút đó.

Với thuật toán Dijkstra, ta vẫn sử dụng mảng H[1,2,…,n]H[1,2,…,n] để thực thi Heap. Tuy nhiên, mỗi phần tử H[i]H[i] sẽ lưu một đỉnh của đồ thị (chứ không phải khóa) còn khóa sẽ được lưu trong mảng dd. Như vậy, khóa của nút thứ ii của Heap là d[H[i]]d[H[i]]. Để thực hiện **DecreaseKey**, như đã nói ở đoạn văn trước, ta sẽ lưu một mảng pos[1,2,…,V]pos[1,2,…,V] trong đó pos[v]pos[v] sẽ lưu vị trí của đỉnh v∈Gv∈G trong Heap HH. Hay nói cách khác,

pos[v]=ipos[v]=i khi và chỉ khi H[i]=v(4)H[i]=v(4)

Giả mã của Heap:

DecreaseKey(Heap HH, Vertex uu, Key KK):   
    D[u]←KD[u]←K  
    UpHeapify(pos[u])(pos[u])

ExtractMin(Heap HH):   
    n←n← length of the heap HH  
    tmp←H[1]tmp←H[1]  
    H[1]←H[n]H[1]←H[n]  
    pos[H[n]]←1pos[H[n]]←1         ≪≪*update the position of the vertex*H[n]≫H[n]≫  
    n←n−1n←n−1  
    DownHeapify(1)(1)  
    return tmptmp

DownHeapify( Heap Node uu):   
    m←2∗um←2∗u         ≪v≪v*is the left child of*uu≫≫  
    **if** m≤nm≤n         ≪u≪u*is not a leaf*≫≫  
        **if** d[H[m]]d[H[m]] > d[H[2∗u+1]]d[H[2∗u+1]]  
            m←2∗u+1m←2∗u+1  
        **if** d[H[u]]d[H[u]] > d[H[m]]d[H[m]]  
            swap H[u]←H[m]H[u]←H[m]  
            update pos H[u]H[u] and H[m]H[m] after swap  
            DownHeapify(mm)

Parent( uu):   
    **if** uu is even  
        return u/2u/2  
    **else**  
        return (u−1)/2(u−1)/2

BuildHeap(V∖{s}V∖{s}):   
    **for** i←1i←1 to s−1s−1  
        H[i]←iH[i]←i  
        pos[i]←ipos[i]←i  
    **for** i←s+1i←s+1 to VV  
        H[i−1]←iH[i−1]←i  
        pos[i]←i−1pos[i]←i−1  
    **for** i←V−1i←V−1 down to 11  
        DownHeapify(i)(i)