Một số thuật toán trên đồ thị sử dụng cấu trúc Heap (C++ priority_queue)

1. Thuật toán Dijkstra tìm đường đi ngắn nhất trên đồ thị có trọng số không âm

1.1. Bài toán

Cho đồ thị G có hướng, có trọng số không âm, gồm n đỉnh và m cung. Tìm đường đi ngắn nhất từ đỉnh s tới đỉnh t.

Input: vào từ tệp văn bản *Dijkstra.inp*

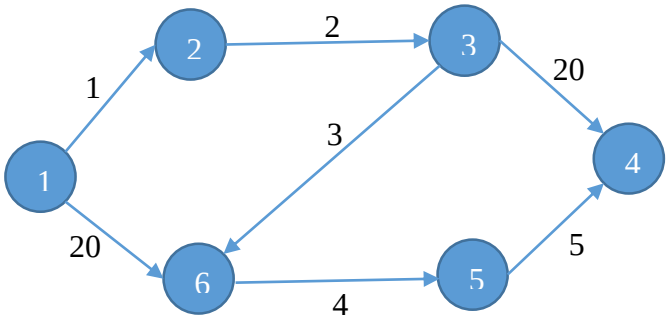
- Dòng 1: Chứa số đỉnh n (≤ 1000), số cung m của đồ thị, đỉnh xuất phát s, đỉnh đích t. Các giá trị cách nhau bởi dấu cách.

- m dòng tiếp theo, mỗi dòng chứa ba số u, v, c cách nhau bởi dấu cách, cho biết cung (u, v) có trọng số là c (c là số nguyên có giá trị tuyệt đối ≤ 1000)

Output: ghi ra tệp file văn bản *Dijsktra.out* độ dài đường đi ngắn nhất từ s tới đỉnh t và dãy đỉnh mô tả đường đi ngắn nhất (xem ví dụ mẫu).

Ví dụ:

Input	Output
6 7 1 4 1 2 1 1 6 20 2 3 2 3 6 3 3 4 20 5 4 5 6 5 4	Khoang cach 1 den 4 = 15 1->2->3->6->5->4



1.2. Thuật toán Dijkstra

Gồm các bước cơ bản sau:

B0: Coi đỉnh xuất phát s là đỉnh đã tối ưu (đỉnh tối ưu có nghĩa là đã tìm được đường đi ngắn nhất đến đỉnh đó), các đỉnh còn lại chưa tối ưu (khoảng cách từ s đến các đỉnh này là INFINITY).

B1: Tìm đỉnh gần đỉnh s nhất trong số các đỉnh chưa tối ưu và coi đỉnh đó là tối ưu. Nếu đỉnh tìm được là đỉnh t thì kết thúc thuật toán.

B2: Cập nhật lại khoảng cách từ các đỉnh chưa tối ưu đến s dựa trên đỉnh vừa tìm được ở B1. Quay lại B1.

1.3. Nhận xét

Thuật toán trên có độ phức tạp thời gian $O(n^2)$. Tại B1, bạn phải đi tìm đỉnh có khoảng cách đến s là bé nhất mất O(n).

Nếu có cách nào đó mà bạn có thể tìm được đỉnh tối ưu với thời gian gian nhỏ hơn thì chi phí thời gian sẽ giảm xuống.

Có một ý tưởng là, bạn cần một cấu trúc dữ liệu mà luôn cho bạn tìm được đỉnh tối ưu trong thời gian < O(n) (ví dụ O(1)).

Như vậy, bạn sẽ nghĩ đến một danh sách các đỉnh đã được sắp xếp tăng dần theo khoảng cách đến s và bạn chỉ việc lấy đỉnh đầu tiên trong danh sách đó. Nhưng lại nảy sinh vấn đề khác, danh sách mà bạn có thể luôn thay đổi do việc thực hiện cập nhật tại B2 → sau khi thực hiện B2, danh sách đó có thể không còn trật tự như đã sắp xếp.

Cách giải quyết ở đây là bạn đưa danh sách này vào một cấu trúc Heap (dạng Min Heap). Và đỉnh tối ưu cần tìm luôn ở đỉnh Heap, nó cho phép bạn chọn được đỉnh tối ưu trong O(1) và khi thực hiện cập nhật khoảng cách cách đỉnh tại B2 bạn lại "*nạp các khoảng cách mới cập nhật vào heap*". Việc nạp phần tử vào heap, sẽ bao gồm việc vun lại heap để bảo toàn tính chất của heap sẽ mất thời gian O(logn). Như vậy chi phí tổng thể của Dijkstra - Heap là O(nlogn).

Bằng việc sử dụng thư viện C++ STL, bạn sẽ lợi dụng được các phép toán nội tại của heap mà không phải lo về việc viết mã tạo, vun lại heap.

1.4. Cài đặt

Với cấu trúc C++ priority_queue sẵn có, bạn có thể viết chương trình đó như sau:

```
#include <fstream>
#include <queue>
#include <vector>

using namespace std;
const int MAXN = 10000;
const int INF = 1 << 30;

typedef pair<int, int> pii;

struct cmp{
    bool operator()(pii x, pii y){return x.second > y.second;}
};

vector< vector<pii> > dske;
vector<int> dist, prev;
vector<bool> toiuu;
priority_queue<pii, vector<pii>, cmp > Heap ;
int n, m, s, t;

ofstream fou("graph.out");
ifstream fin("graph.inp");

void get_data() {
    fin >> n >> m >> s >> t;
    dske.resize(n+1);
    int u, v, w;
    for (int i = 1; i <= m; ++i) {
        fin >> u >> v >> w;
        dske[u].push_back(pii(v,w));
    //  dske[v].push_back(pii(u,w));  do thi vo huong thi them canh nay
    }
}

void printPaths(int u)
{
    if (u == s) fou << u ;
    else {
```

```
39.         printPaths(prev[u]);
40.         fou << "->" << u;
41.     }
42. }
43.
44. int Dijkstra(int s, int t) {
45.     dist.resize(n+1);   // dist[i] = khoang cach tu i den s
46.     toiuu.resize(n+1);  // toiuu[i] = dinh i da toi uu
47.     prev.resize(n+1);   // prev[v] = u, u la dinh lien truoc dinh v trong duong di
48.     int i, u, v, w, sz;
49.     // khoi tao ban dau
50.     for (i = 1; i <= n; i++) {
51.         dist[i] = INF;
52.         prev[i] = s;
53.         toiuu[i] = false;
54.     }
55.     dist[s] = 0;
56.     Heap.push(pii(s,0)); // nap dinh s vao Heap
57.     while (!Heap.empty()) {
58.         u = Heap.top().first; //lay dinh tot nhat trong Heap
59.         if (u == t) break;
60.         if(toiuu[u]) continue;
61.         Heap.pop(); // loai bo dinh u khoi Heap
62.         sz = dske[u].size();
63.         for (int i = 0; i < sz; ++i){
64.             v = dske[u][i].first;
65.             w = dske[u][i].second;
66.             if (toiuu[v]) continue;
67.             if (dist[v] > dist[u] + w) {
68.                 dist[v] = dist[u] + w;
69.                 prev[v] = u;
70.                 Heap.push(pii(v,dist[v]));
71.             }
72.         }
73.         toiuu[u] = true;
74.     }
75.     return dist[t];
76. }
77.
78. int main() {
79.     get_data();
80.     fou <<"Khoang cach "<< s <<" den "<< t <<" = " << Dijkstra(s,t) << endl;
81.     printPaths(t);
82.     return 0;
83. }
```

Trong chương trình trên có một vài kiểu biến, hàm lạ cần giải thích để các em rõ hơn:

- Dòng 9: `typedef pair<int, int> pii;`

Câu lệnh này khai báo một kiểu dữ liệu tên là pii đại diện cho kiểu dữ liệu gồm một cặp phần tử kiểu int, trong đó phần tử đứng trước trong cặp là *first*, phần tử đứng sau là *second*.

Trong bài toán này, ta lợi dụng kiểu pair<int, int> để biểu diễn dữ liệu của danh sách kề khai báo ở dòng 15. Mỗi phần tử của danh sách kề dske[u][i] (đỉnh thứ i kề với đỉnh u) là một cặp phần tử, trong đó: first là số hiệu đỉnh kề với u, second là độ dài cung từ u đến đỉnh này.

- Dòng 11-13: `struct cmp{ bool operator()(pii x, pii y){return x.second > y.second;}};`

Đây là một cách định nghĩa hàm so sánh *cmp* dùng để khai báo Heap: vì cấu trúc Heap của C++ có tính năng tự vun Heap, theo mặc định Heap sẽ sử dụng phép so sánh Max (phần tử lớn nhất sẽ ở đỉnh Heap) tuy nhiên do nhu cầu sử dụng của chúng ta trong mỗi bài toán khác nhau. Cụ thể trong bài này, mỗi phần tử của Heap là một cặp số nguyên pair<int, int>, số nguyên thứ 2 *second* lưu giữ khoảng cách từ đỉnh lưu trong *first* đến *s* và yêu cầu Heap phải vun đống sao cho phần tử nào có giá trị lưu trong *second* nhỏ nhất thì phải đưa lên đỉnh Heap, nên ta phải khai báo hàm như trên để Heap dựa vào đó mà vun đống cho đúng. Đây là một cú pháp sử dụng trong lập trình hướng đối tượng của C++ (các em chưa học - lên ĐH nhé) cho phép ta khai báo hàm *cmp* như một hàm thành viên của đối tượng *pii*. Hàm cho phép ta so sánh hai phần tử *pii* theo ý muốn.

2. Thuật toán Kruskal tìm cây khung nhỏ nhất

2.1. Bài toán

Cho G = (V, E) là đồ thị vô hướng liên thông có trọng số, với một cây khung T của G, ta gọi trọng số của cây T là tổng trọng số các cạnh trong T.

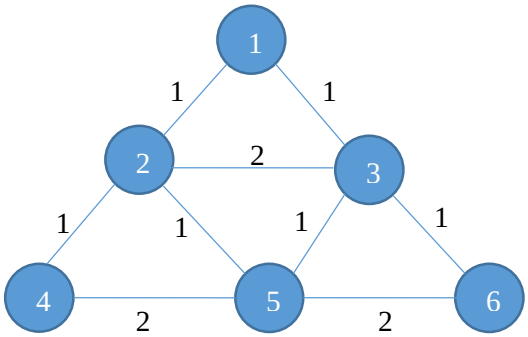
Bài toán đặt ra là tìm trong số các cây khung của G, cây khung có trọng số nhỏ nhất (minimum spanning tree - MST) và bài toán này gọi là bài toán cây khung nhỏ nhất.

Input: vào từ tệp văn bản MST.INP

- Dòng 1: Ghi hai số số đỉnh n (≤ 1000) và số cạnh m của đồ thị cách nhau ít nhất 1 dấu cách
- m dòng tiếp theo, mỗi dòng có dạng 3 số u, v, c[u, v] cách nhau ít nhất 1 dấu cách thể hiện đồ thị có cạnh (u, v) và trọng số cạnh đó là c[u, v]. (c[u, v] là số nguyên có giá trị tuyệt đối không quá 1000).

Output: ghi ra tệp văn bản MST.OUT các cạnh thuộc cây khung và trọng số của cây khung (xem mẫu trong ví dụ)

Input	Output
6 9 1 2 1 1 3 1 2 4 1 2 3 2 2 5 1 3 5 1 3 6 1 4 5 2 5 6 2	(1,2) = 1 (2,4) = 1 (3,6) = 1 (3,5) = 1 (1,3) = 1 5



2.2. Thuật toán

Ý tưởng của thuật toán Kruskal có thể mô tả ngắn gọn như sau, giả sử G=(V,E) là đơn đồ thị vô hướng có N đỉnh và M cạnh. T là cây khung nhỏ nhất cần tìm, bắt đầu T rỗng:

B1: Nạp tất cả các cạnh của đồ thị vào Heap (Heap lưu cạnh có trọng số nhỏ nhất ở đỉnh)

B2: Thực hiện lặp cho đến khi Heap rỗng các bước sau:

B2.1: Lấy một cạnh e =(u, v) từ đỉnh Heap, loại bỏ cạnh e khỏi Heap

B2.2: tìm r1= gốc của u và r2 = gốc của v.

B2.3: Nếu r1 = r2 thì quay lại B2.1

B2.4: Nạp e vào cây T, nếu |T| = n-1 thì kết thúc thuật toán.

B2.5: Hợp nhất hai cây gốc r1 và r2 thành một cây (điều chỉnh gốc của cây mới, cây nào ít con hơn làm con cây kia), quay về B2.1

B3: Nếu |T| < n-1 thì G không liên thông, không có cây khung, ngược lại thì in ra cây T.

2.3. Nhận xét

Như đã nói trong phần thuật toán ở trên, ta sử dụng heapsort để sắp xếp danh sách cạnh. Trong C++, ta dùng luôn cấu trúc priority_queue để nhập danh sách cạnh từ input (mỗi cạnh gồm số hiệu hai đầu mút (u, v) và trọng số cạnh w. Quá trình nhập dữ liệu Heap sẽ tự được vun đống.

Ta sử dùng thêm một mảng *Tree* để chứa danh sách cây khung T. Mỗi lần có một cạnh được nạp vào cây khung ta sẽ nạp nó vào vector *Tree*.

2.4. Cài đặt

Dưới đây là một vài bản cài đặt thuật toán với ý tưởng nêu trên:

```
#include <fstream>
#include <queue>
#include <vector>

using namespace std;
ifstream fi("mst.inp");
ofstream fo("mst.out");

struct edge { int u, v; double w; };

bool operator < (const edge x, const edge y){return x.w > y.w;}

priority_queue<edge> heap;
vector<int> pa;
vector<edge> tree;
int n, m;

void get_data(){
    fi >> n >> m;
    edge c;
    for (int i = 0; i < m ; i++){ fi >> c.u >> c.v >> c.w; heap.push(c); }
}

void init(){
    pa.resize(n+1);
    for(int i = 1; i <= n; i++) pa[i] = -1;
}

int GetRoot(int r){
   while(pa[r] > 0) r = pa[r];
   return r;
}

void Union(int r1, int r2){
    int x = pa[r1] + pa[r2];
    if(pa[r1] > pa[r2]){
        pa[r1] = pa[r2];
        pa[r2] = x;
    }
```

```
    else{
        pa[r2] = pa[r1];
        pa[r1] = x;
    }
}

bool kruskal(){
    int r1, r2;
    edge e;
    while(!heap.empty()){
        e = heap.top(); heap.pop();
        r1 = GetRoot(e.u);
        r2 = GetRoot(e.v);
        if (r1 != r2){
            tree.push_back(e);
            if(tree.size() == n-1) return true;
            Union(r1, r2);
        }
    }
    return false;
}

void print(){
    int weight = 0;
    for(int i = 0; i < tree.size(); i++){
        fo <<'('<< tree[i].u <<','<<tree[i].v<<") = "<<tree[i].w << endl;
        weight += tree[i].w;
    }
    fo << weight <<endl;
}

int main()
{
    get_data();
    init();
    if(kruskal())print(); else fo≪ "The graph isn't connected!";
    return 0;
}
```

Bản cài đặt sau đây sử dụng 1 đối tượng *C++ map* để lưu nút gốc của các cây con trong quá trình hợp nhất:

```
#include <fstream>
#include <vector>
#include <queue>
#include <map>

using namespace std;
typedef pair<int,pair<int,int> > node;
typedef priority_queue<node, vector<node>, greater<node> > min_heap;

ifstream fin("mst.inp");
ofstream fou("mst.out");

vector<node> MST_KRUSHAL(int N, min_heap Q) {
    vector<node> mintree;
    map<int,int> parent;
    for (int i = 1; i <= N; ++i) parent[i] = i;
    while(!Q.empty())
    {
        int u = Q.top().second.first, v = Q.top().second.second, w = Q.top().first;
```

```
20.         int r1 = parent[u], r2 = parent[v];
21.         if(r1 != r2) {
22.             mintree.push_back(make_pair(w, make_pair(u, v)));
23.             If (mintree.size() == n-1) return(mintree);
24.             // hop nhat cac cay goc r2 thanh cay goc r1
25.             for (int i = 1; i <= N; ++i) if(parent[i] == r2) parent[i] = r1;
26.         }
27.         Q.pop();
28.     }
29.     return mintree;
30. }
31.
32. int main()
33. {
34.     int N, m;
35.     fin >>N >>m;
36.     int u, v, w;
37.     min_heap Q;
38.     for (int i = 0; i < m; ++i) {
39.         fin >> u >> v >> w;
40.         Q.push(make_pair(w, make_pair(u, v)));
41.     }
42.     vector<node> mintree(MST_KRUSHAL(N, Q));
43.     if(mintree.size() == n-1){
44.     for (int i = 0; i < mintree.size(); ++i)
45.        fou << A.second.first <<" -> " <<A.second.second <<" : " <<A.first <<"\n";
46.     } else fo≪ "The graph isn't connected!";
47.     return 0;
48. }
```

3. Thuật toán Prim tìm cây khung nhỏ nhất

3.1. Bài toán

Mô tả bài toán tương tự như đã nêu trong phần nói về thuật toán Kruskal

3.2. Thuật toán

Ý tưởng của thuật toán Prim có thể mô tả ngắn gọn như sau, giả sử G=(V,E) là đơn đồ thị vô hướng có N đỉnh và M cạnh. T là cây khung nhỏ nhất cần tìm, bắt đầu T rỗng:

B1: Chọn 1 đỉnh s bất kỳ (s = 1..N), đánh dấu s là đã thăm (visited[s] = true)

B2: Đặt tất cả các cạnh nối với s và Heap (Heap được sắp xếp sao cho cạnh nhỏ nhất ở đỉnh)

B3: Lặp cho đến khi Heap rỗng các thao tác sau:

B3.1: Lấy ra một cạnh e ở đỉnh Heap, loại bỏ luôn e ra khỏi Heap

B3.2: Nếu cả hai đầu mút của e là đã thăm thì quay lại B3.1;

B3.3: Thêm e vào cây T; Nếu |T| = n-1 thì kết thúc thuật toán

B3.4: gọi v là đỉnh đầu mút của e và visited[v] = false, thêm tất cả các cạnh nối với v vào Heap, đánh dấu visited[v] = true, quay lại B3.1

B4: Nếu |T| < n-1 thì đồ thị không liên thông và không có cây khung. Ngược lại thì in T

3.3. Nhận xét

Theo ý tưởng ở trên, ta sử dụng một cấu trúc C++ priority_queue để lưu danh sách các cạnh tiềm năng cho việc xây dựng cây T.

3.4. Cài đặt

Dưới đây là một cài đặt theo thuật toán trong phần 3.2

```
#include <fstream>
#include <queue>
#include <vector>
using namespace std;
ifstream fi("mst.inp");
ofstream fo("mst.out");

struct dinhke{
    int id; double w;
    dinhke(){}
    dinhke(int iid, double dw): id(iid), w(dw){}
};
struct edge{ int u, v; double w; };
bool operator < (const edge x, const edge y){return x.w > y.w;}

priority_queue<edge> heap;
vector< vector<dinhke> > dske;
vector<bool> visited;
vector<edge> tree;
int n, m;

void get_data(){
    fi >> n >> m;
    int u, v;
    double w;
    dinhke x;
    dske.resize(n+1);
    for (int i = 0; i < m ; i++){
        fi >> u >> v >> w;
        dske[u].push_back(dinhke(v,w));
        dske[v].push_back(dinhke(u,w));
    }
}

bool prim(int u){
    visited.resize(n+1, false);
    visited[u] = true;
    // dat cac canh ke u vao heap, sap xep tang dan
    for(int i = 0; i < dske[u].size(); i++){
        edge x = {u,dske[u][i].id, dske[u][i].w};
        heap.push(x);
    }
    while(!heap.empty()){
        edge e = heap.top();
        heap.pop();
        if (visited[e.u] && visited[e.v]) continue;
        tree.push_back(e);
        if (tree.size() == n-1) return true;
        int v = (visited[e.u]?e.v:e.u);
        for(int i = 0; i < dske[v].size(); i++){
            int x = dske[v][i].id;
            double z = dske[v][i].w;
            if(!visited[x]){
                edge y = {v,x,z};
                heap.push(y);
            }
        }
```

```
58.         visited[v] = true;
59.     }
60.     return false;
61. }
62.
63. void print(){
64.     double w = 0.0;
65.     for(int i = 0; i < tree.size(); i++){
66.         fo <<"("<< tree[i].u <<","<<tree[i].v <<") = "<<tree[i].w << endl;
67.         w += tree[i].w;
68.     }
69.     fo << w << endl;
70. }
71. int main()
72. {
73.     get_data();
74.     if (prim(1)) print();
75.     else fo<<"The graph is not connected!";
76.     return 0;
77. }
```

(Còn nữa.....)