Chương 1. Cây khung nhỏ nhất

Cho $G = (V, E, w)$ là đồ thị vô hướng liên thông có trọng số. Cây khung nhỏ nhất (*Minimum Spanning Tree – MST*) của đồ thị là một tập các cạnh kết nối tất cả các đỉnh, không tạo ra chu trình và có tổng trọng số nhỏ nhất có thể.

Với một cây khung T của G, ký hiệu $w(T)$ là tổng trọng số các cạnh trong T, gọi là trọng số của cây khung T. Bài toán đặt ra là trong số các cây khung của G, chỉ ra cây khung có trọng số nhỏ nhất, cây khung như vậy được gọi là cây khung nhỏ nhất (*Minimum Spanning Tree - MST*) của đồ thị.

Bài toán cây khung nhỏ nhất có nhiều ứng dụng trong thực tế: chẳng hạn khi các công ty viễn thông muốn đặt các đường dây kết nối tất cả các trạm thu phát trong thành phố với ràng buộc là các đường dây phải đặt dọc theo các đường phố có sẵn. Khi đó ta có một mô hình đồ thị với các đỉnh là các trạm thu phát và các cạnh là những tuyến đường kết nối các trạm thu phát. Việc đi dây dọc một tuyến đường có thể rẻ hoặc đắt tùy theo độ dài đường dây và chi phí lắp đặt theo địa hình tuyến đường đó. Trong thực tế, chi phí này không bao giờ âm và sơ đồ đặt các đường dây với chi phí nhỏ nhất chính là cây khung nhỏ nhất của đồ thị.

Ta nhắc lại định nghĩa và một số tính chất của cây khung (cây bao trùm):

Với $G = (V, E)$ là một đồ thị vô hướng liên thông gồm n đỉnh, một đồ thị con T chứa tất cả các đỉnh, liên thông và không có chu trình được gọi là cây khung của G. Khi ấy T có các tính chất sau đây tương đương với định nghĩa (tức là một đồ thị con T chứa tất cả các đỉnh chỉ cần thỏa mãn một trong những tính chất này thì nó là cây khung của G):

- T không chứa chu trình đơn và có $n - 1$ cạnh
- T liên thông và mỗi cạnh của nó đều là cầu
- Giữa hai đỉnh bất kỳ của T đều tồn tại đúng một đường đi đơn
- T không chứa chu trình đơn nhưng hễ cứ thêm vào một cạnh ta thu được một chu trình đơn.
- T liên thông và có $n - 1$ cạnh

Một đồ thị có thể có nhiều cây khung, thậm chí có nhiều cây khung nhỏ nhất. Sau đây ta sẽ xét hai thuật toán thông dụng để tìm ra một cây khung nhỏ nhất của đơn đồ thị vô hướng có trọng số, cả hai thuật toán này đều là thuật toán tham lam.

1.1. Phương pháp chung

Xét đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Cả hai thuật toán để tìm cây khung ngắn nhất đều dựa trên một phương pháp chung: Nở dần cây khung: Quản lý một tập các cạnh $A \subseteq E$ và cố gắng bổ sung các cạnh vào A sao cho A luôn nằm trong tập cạnh của một cây khung nhỏ nhất (tính bất biến).

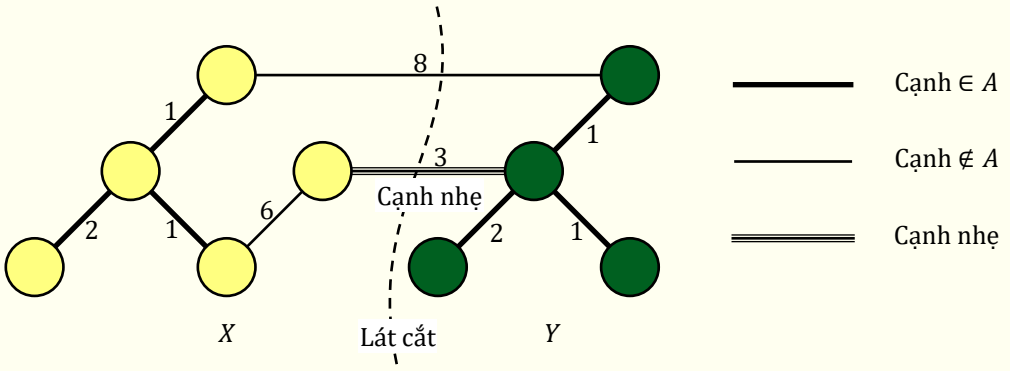
Tại mỗi bước lặp, thuật toán tìm một cạnh e để thêm vào tập A sao cho duy trì được tính bất biến, tức là $A \cup \{e\}$ phải nằm trong tập cạnh của một cây khung nhỏ nhất. Ta nói những cạnh e như vậy là *an toàn* (*safe*) đối với tập A.

```
Input → G = (V, E, w);
A = Ø;
while («A chưa phải cây khung»)
{
    «Tìm cạnh an toàn e đối với A»;
    A = A ∪ {e};
}
Output ← A;
```

Vấn đề còn lại là tìm thuật toán hiệu quả để xác định cạnh an toàn đối với tập A. Chúng ta cần một số khái niệm để giải thích tính đúng đắn của những thuật toán sau này.

Một lát cắt (*cut*) trên đồ thị $G = (V, E)$ là một cách phân hoạch tập đỉnh V thành hai tập rời nhau X, Y sao cho: $X \cup Y = V; X \cap Y = \emptyset$. Ký hiệu lát cắt này là (X, Y).

Ta nói một lát cắt (X, Y) tương thích với tập A nếu không có cạnh nào của A nối giữa một đỉnh thuộc X và một đỉnh thuộc Y. Trong những cạnh nối X với Y, ta gọi (những) cạnh có trọng số nhỏ nhất là cạnh nhẹ (*light edge*) của lát cắt (X, Y) (Hình 1-1).



Hình 1-1. Lát cắt và cạnh nhẹ

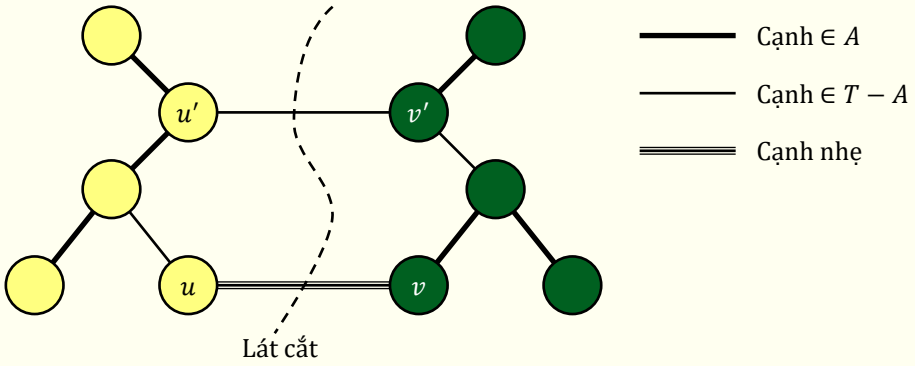
Với đồ thị vô hướng $G = (V, E)$, A là tập con của tập cạnh E, ta ký hiệu $G_A = (V, A)$ là đồ thị con của đồ thị G với tập đỉnh vẫn là V còn tập cạnh là A. Nếu đồ thị G có trọng số thì trọng số đó được giữ nguyên trên các cạnh $\in A$ của đồ thị G_A.

Bổ đề 1-1

Cho đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Gọi A là một tập con của tập cạnh một cây khung nhỏ nhất và (X, Y) là một lát cắt tương thích với A. Khi đó mỗi cạnh nhẹ của lát cắt (X, Y) đều là cạnh an toàn đối với A.

Chứng minh

Với mỗi cạnh nhẹ (u, v) ta cần chỉ ra sự tồn tại của một cây khung nhỏ nhất chứa A và cả cạnh (u, v). Thật vậy, gọi T là một cây khung nhỏ nhất chứa tất cả các cạnh của A. Nếu T chứa cạnh (u, v), ta có điều phải chứng minh. Nếu T không chứa cạnh (u, v), do $u \in X$ và $v \in Y$ nên đường đi từ u tới v trên T sẽ chứa ít nhất một cạnh nối X với Y, gọi cạnh đó là (u', v'). Do lát cắt $X \cup Y$ tương thích với A nên (u', v') và (u, v) đều không thuộc A.



Hình 1-2. Thay cạnh trên cây bởi cạnh nhẹ

Theo giả thiết (u, v) là cạnh nhẹ nên $w(u, v) \leq w(u', v')$. Cắt bỏ cạnh (u', v') khỏi cây T, cây sẽ bị tách rời làm hai thành phần liên thông, sau đó thêm cạnh (u, v) vào cây nối lại hai thành phần liên thông đó để được cây T' (Hình 1-2). Ta có:

$$\begin{aligned} w(T') &= w(T) - w(u', v') + w(u, v) \\ &\leq w(T) \end{aligned} \tag{1.1}$$

Do T là cây khung nhỏ nhất ta suy ra $w(T') = w(T)$ tức là T' cũng phải là cây khung nhỏ nhất. Ngoài ra cây T' chứa cạnh (u, v) và tất cả các cạnh của A. Ta có ĐPCM■

Hệ quả

Cho đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Gọi A là một tập con của tập cạnh của một cây khung nhỏ nhất. Gọi C là tập các đỉnh của một thành phần liên thông trên đồ thị $G_A = (V, A)$. Khi đó nếu $(u, v) \in E$ là cạnh trọng số nhỏ nhất nối từ C tới một thành phần liên thông khác thì (u, v) là cạnh an toàn đối với A.

Chứng minh

Xét lát cắt $(C, V - C)$, lát cắt này tương thích với A và cạnh (u, v) là cạnh nhẹ của lát cắt này. Theo Bổ đề 1-1, (u, v) an toàn đối với A■

Chúng ta sẽ trình bày hai thuật toán tìm cây khung nhỏ nhất trên đơn đồ thị vô hướng và cài đặt chương trình với khuôn dạng Input/Output như sau:

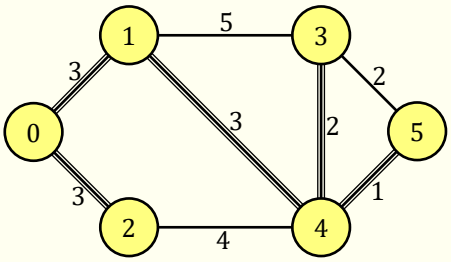
Input

- Dòng 1 chứa số đỉnh $n \leq 10^5$ và số cạnh $m \leq 10^5$ của đồ thị
- m dòng tiếp theo, mỗi dòng chứa chỉ số hai đỉnh đầu mút và trọng số của một cạnh. Trọng số cạnh là số nguyên có giá trị tuyệt đối không quá 10^4.

Output

Cây khung nhỏ nhất của đồ thị

Sample Input	Sample Output
6 8 0 1 3 0 2 3 1 3 5 1 4 3 2 4 4 3 4 2 3 5 2 4 5 1	Minimum Spanning Tree: Edge 0: (0, 1); Weight = 3 Edge 1: (0, 2); Weight = 3 Edge 5: (3, 4); Weight = 2 Edge 3: (1, 4); Weight = 3 Edge 7: (4, 5); Weight = 1 Tree Weight: 12



1.2. Thuật toán Kruskal

Thuật toán Kruskal [1] tìm cây khung ngắn nhất của đồ thị $G = (V, E, w)$:

Khởi tạo tập cạnh A ban đầu là $\emptyset$. Duyệt danh sách cạnh của đồ thị từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn, mỗi khi xét tới một cạnh và việc thêm cạnh đó vào A không tạo thành chu trình đơn trong đồ thị G_A thì kết nạp thêm cạnh đó vào A... Cứ làm như vậy cho tới khi:

- Hoặc đã kết nạp được $|V| - 1$ cạnh vào trong A thì ta được $G_A = (V, A)$ chính là cây khung nhỏ nhất,
- Hoặc khi duyệt hết danh sách cạnh mà vẫn chưa kết nạp đủ $|V| - 1$ cạnh vào A. Trong trường hợp này đồ thị G đã cho là không liên thông, việc tìm kiếm cây khung thất bại.

Như vậy cần xử lý hai vấn đề kỹ thuật khi cài đặt thuật toán Kruskal:

- Làm thế nào để duyệt qua danh sách các cạnh từ cạnh có trọng số nhỏ tới cạnh có trọng số lớn.
- Làm thế nào kiểm tra xem việc thêm một cạnh vào A có tạo thành chu trình đơn trong G_A hay không.

1.2.1. Duyệt danh sách cạnh

Để duyệt danh sách các cạnh từ cạnh có trọng số nhỏ tới cạnh có trọng số lớn, cách đơn giản và phổ biến nhất là sắp xếp lại danh sách các cạnh.

Một cải tiến có thể làm giảm thời gian thực thi là khéo léo lồng thuật toán Kruskal vào trong thuật toán sắp xếp sao cho danh sách cạnh được sắp xếp tới đâu sẽ được thuật toán Kruskal xử lý luôn đến đó. Thuật toán sắp xếp có thể dừng ngay khi đã có được cây khung chứ không cần chờ tới lúc sắp xếp xong toàn bộ danh sách cạnh. QuickSort và Heap Sort là những thuật toán thích hợp nhất cho kỹ thuật này.

1.2.2. Kết nạp cạnh và hợp cây

Trong quá trình xây dựng cây khung, đồ thị G_A không có chu trình đơn, mỗi thành phần liên thông của G_A cũng chính là cây khung của thành phần liên thông đó. Muốn thêm một cạnh (u, v) vào A mà không tạo thành chu trình đơn trên G_A thì (u, v) phải nối hai thành phần liên thông khác nhau của G_A.

Thuật toán quản lý các tập đỉnh, mỗi tập đỉnh ứng với một thành phần liên thông của đồ thị G_A. Ban đầu tập cạnh $A = \emptyset$ thì ta có n tập đỉnh ứng với n thành phần liên thông, mỗi tập đỉnh gồm đúng một đỉnh duy nhất.

Khi xét tới cạnh (u, v), nếu u và v thuộc hai tập khác nhau thì thuật toán bổ sung cạnh (u, v) vào A. Tập đỉnh chứa u và tập đỉnh chứa v sau đó được hợp nhất lại thành một, điều này tương ứng với việc hợp nhất hai miền liên thông chứa u và chứa v.

1.2.3. Cài đặt bằng rừng các tập rời nhau

Thuật toán Kruskal cài đặt thích hợp nhất với rừng các tập rời nhau (*Disjoint-set forest*), đây là cấu trúc dữ liệu cho phép thực hiện hiệu quả ba thao tác:

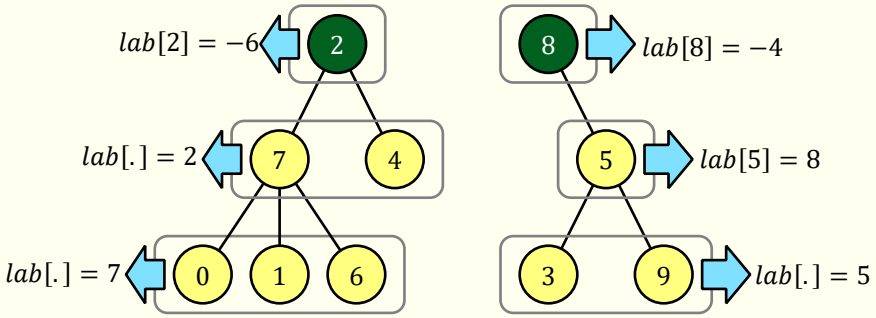
- MakeSet(u): Tạo tập $\{u\}$
- FindSet(u): Tìm tập chứa u
- Unite(r, s): Hợp hai tập rời nhau r và s lại thành một.

Cài đặt cấu trúc dữ liệu trong trường hợp tổng quát tương đối dài. Trong trường hợp cụ thể này, do các phần tử của tập hợp là số nguyên từ 0 tới $n - 1$, ta sẽ trình bày một mẹo cài đặt để chương trình được ngắn gọn, chạy nhanh và tiết kiệm bộ nhớ.

Mỗi tập đỉnh được biểu diễn bởi một cây (chú ý rằng cây này là cấu trúc dữ liệu với quan hệ cha-con chứ không liên quan tới đồ thị dạng cây). Mỗi nút của cây chứa một đỉnh và được đồng nhất với đỉnh đó. Mỗi nút chứa con trỏ tới nút cha: $lab[v]$ là nút cha của nút v. Trong trường hợp v là nút gốc của cây, ta đặt:

$$lab[v] = -\text{số nút trong cây}$$

Hình 1-3 là ví dụ về cách biểu diễn hai tập rời nhau: {0,1,2,4,6,7} và {3,5,8,9}.

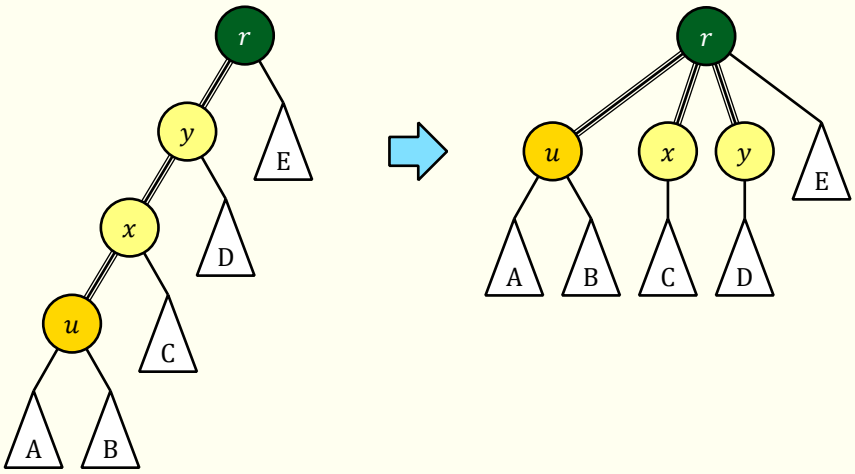


Hình 1-3. Rừng các tập rời nhau

Ban đầu mỗi tập chỉ gồm một đỉnh, nên cấu trúc dữ liệu này được khởi tạo với các nhãn $lab[0 \ldots n) = -1$ tương ứng với rừng gồm các cây chỉ có 1 nút.

Để xác định hai đỉnh u, v có thuộc 2 tập khác nhau hay không, ta chỉ cần xác định xem gốc của cây chứa u và gốc của cây chứa v có khác nhau hay không. Việc xác định gốc của cây chứa u được thực hiện bởi hàm FindSet(u): Đi từ u lên nút cha, đến khi gặp nút gốc (nút r có $lab[r] < 0$) thì dừng lại. Đi kèm với hàm $FindSet(u)$ là phép *nén đường* (*path compression*): Dọc trên đường đi từ u tới nút gốc r, đi qua đỉnh nào ta cho luôn đỉnh đó làm con của r (Hình 1-4):

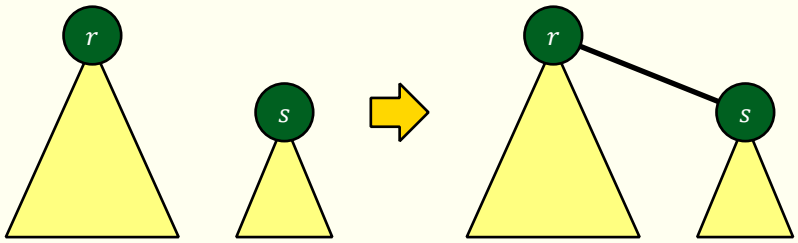
```
1  int FindSet(int u) //Xác định tập chứa u
2  {
3      return lab[u] < 0 ? u : lab[u] = FindSet(lab[u]);
4  }
```



Hình 1-4. Phép tìm gốc kết hợp với nén đường

Để hợp nhất hai tập, ta xây dựng một cây chứa tất cả các đỉnh trong hai cây ban đầu. Điều này thực hiện đơn giản bằng cách cho một gốc cây làm con của gốc cây còn lại. Phép *hợp theo hạng* (*union-by-rank*) được sử dụng ở đây nhằm tăng tốc độ của giải thuật: Gốc cây có số nút ít hơn phải làm con của gốc cây có số nút nhiều hơn:

```
void Unite(int r, int s) //Hợp nhất cây gốc r và và cây gốc s
{
    if (lab[r] > lab[s]) //Cây r có số nút ít hơn,
        std::swap(r, s);      //→ đảo vai trò r và s
    lab[r] += lab[s]; //Số nút của cây hợp thành = tổng số nút hai cây thành phần
    lab[s] = r;       //Cho s làm con của r
}
```



Hình 1-5. Hợp tập theo hạng

Vì dữ liệu cạnh có cấu trúc lớn (hai đầu mút, trọng số, trường đánh dấu thuộc cây), ta sẽ sử dụng con trỏ tới cạnh làm đại diện cho cạnh trong các thao tác sắp xếp, xử lý cạnh. Điều này không những giúp thao tác xử lý được nhanh hơn mà còn giữ nguyên được danh sách cạnh, phục vụ cho việc in kết quả.

KRUSKAL.cpp Thuật toán Kruskal

```
#include <iostream>
#include <algorithm>
using namespace std;
const int maxN = 1e5;
const int maxM = 1e5;

```

```cpp
int n, m, k;
struct TEdge //Cấu trúc cạnh
{
    int u, v;      //Hai đỉnh đầu mút
    int w;         //Trọng số
    bool selected; //Có chọn vào cây khung hay không
};
using PEdge = TEdge*; //Kiểu con trỏ tới cạnh
TEdge e[maxM]; //Danh sách các cạnh
PEdge p[maxM]; //Danh sách các con trỏ tới cạnh
int lab[maxN]; //Nhãn biểu diễn rừng các tập rời nhau

void ReadInput() //Nhập dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
    {
        cin >> e[i].u >> e[i].v >> e[i].w;
        e[i].selected = false; //Chưa cạnh nào được chọn vào A
        p[i] = &e[i]; //Cho p[i] trỏ tới e[i]
    }
}

void Init()
{
    sort(p, p + m, [](PEdge e1, PEdge e2) //Xếp danh sách p
    {
        return e1->w < e2->w; //Con trỏ tới cạnh nhỏ hơn được xếp trước
    });
    fill(lab, lab + n, -1); //Quản lý n tập, mỗi tập chứa duy nhất một đỉnh
}

int FindSet(int u) //Tìm tập chứa u, nén đường
{
    return lab[u] < 0 ? u : lab[u] = FindSet(lab[u]);
}

void Unite(int r, int s) //Hợp 2 tập ứng với cây gốc r và gốc s, hợp theo hạng
{
    if (lab[r] > lab[s]) //Cây r nhỏ hơn cây s
        swap(r, s);      //→ Đảo vai trò cây r và cây s
    lab[r] += lab[s]; //Cây r sẽ chứa thêm toàn bộ nút của cây s
    lab[s] = r; //Cho gốc s làm con gốc r
}

void Kruskal() //Thuật toán Kruskal
{
    k = 0; //Số cạnh đã được kết nạp
    for (int i = 0; i < m; ++i)
    {  //Xét cạnh *p[i]
        int r = FindSet(p[i]->u), s = FindSet(p[i]->v);
        if (r != s) //Hai đầu mút thuộc 2 tập khác nhau
        {
            p[i]->selected = true; //Chọn cạnh này vào A
            Unite(r, s);           //Hợp 2 tập chứa 2 đầu mút
            ++k;                   //Kết nạp thêm được 1 cạnh
            if (k == n - 1) //Nếu đã đủ n - 1 cạnh
                break;      //→ Dừng thuật toán
        }
    }
}

```

```
void Print() //In kết quả
{
    if (k < n - 1) //Kết nạp không đủ n - 1 cạnh,
        cout << "Graph is not connected!"; //→ đồ thị không liên thông
    else //In ra các cạnh của cây khung nhỏ nhất
    {
        cout << "Minimum Spanning Tree:\n";
        int TreeWeight = 0;
        for (int i = 0; i < m; ++i)
            if (e[i].selected) //Cạnh e[i] được chọn vào cây khung nhỏ nhất
            {
                cout << "Edge " << i << ": ("
                     << e[i].u << ", " << e[i].v << "); "
                     << "Weight = " << e[i].w << '\n';
                TreeWeight += e[i].w;
            }
        cout << "Tree Weight: " << TreeWeight << '\n';
    }
}

int main()
{
    ReadInput();
    Init();
    Kruskal();
    Print();
}
```

1.2.4. Tính đúng đắn và thời gian thực hiện giải thuật

Mỗi khi một cạnh (u, v) được xét đến, nó sẽ chỉ được kết nạp vào A nếu như u và v thuộc hai tập đỉnh khác nhau ứng với hai thành phần liên thông S_u, S_v khác nhau của G_A, hay nói cách khác cạnh (u, v) là cạnh trọng số nhỏ nhất nối từ thành phần liên thông S_u tới một thành phần liên thông khác. Theo hệ quả của Bổ đề 1-1, cạnh (u, v) là cạnh an toàn với A.

Khi thuật toán Kruskal kết thúc, ta thu được tập A, khi đó mọi cạnh $(u, v) \in E$ sẽ không thể nối hai thành phần liên thông khác nhau của G_A, bởi nếu không thì khi thuật toán xét đến cạnh (u, v), nó đã kết nạp luôn cạnh đó vào A rồi. Vậy thì:

- Nếu đồ thị $G = (V, E, w)$ liên thông, thuật toán sẽ thu được A là tập cạnh của một cây khung. Mặt khác, vì A luôn nằm trong tập cạnh của một cây khung nhỏ nhất, ta suy ra A chính là tập cạnh của một cây khung nhỏ nhất.
- Nếu đồ thị $G = (V, E, w)$ không liên thông, thuật toán sẽ thu được A là rừng các cây khung của các thành phần liên thông trong G.

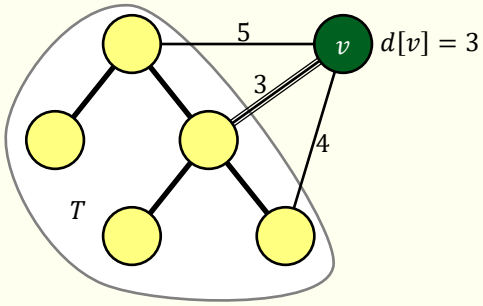
Xét về độ phức tạp tính toán, người ta đã chứng minh rằng một dãy m thao tác $FindSet$ và $Union$ trên n phần tử có thời gian thực hiện $O(m \times \alpha(n))$ nếu ta sử dụng các mẹo giải nén đường và hợp theo hạng, tức là thuật toán Kruskal có thể thực hiện trong thời gian $O(m \times \alpha(n))$ trên danh sách cạnh đã sắp xếp tăng dần theo trọng số. Tuy nhiên nếu phải thực hiện sắp xếp danh sách cạnh, chúng ta cần cộng thêm thời gian thực hiện giải thuật sắp xếp nữa.

1.3. Thuật toán Prim

Trong trường hợp đồ thị dày (có nhiều cạnh), có một thuật toán hiệu quả hơn để tìm cây khung ngắn nhất là thuật toán Prim [2]: với một cây khung T và một đỉnh $v \notin T$, ta gọi

khoảng cách từ v tới T, ký hiệu $d[v]$, là trọng số nhỏ nhất của một cạnh nối v với một đỉnh nằm trong T (Hình 1-6):

$$d[v] = \min_{u \in T}\{w(u, v)\}$$



Hình 1-6. Khoảng cách từ một đỉnh ngoài cây tới cây

Tư tưởng của thuật toán: Ban đầu khởi tạo một cây T chỉ gồm 1 đỉnh bất kỳ của đồ thị, sau đó ta cứ tìm đỉnh gần T nhất (có khoảng cách tới T ngắn nhất) kết nạp vào T và kết nạp luôn cạnh tạo ra khoảng cách gần nhất đó, cứ làm như vậy cho tới khi:

- Hoặc đã kết nạp đủ n đỉnh vào T, ta có một cây khung ngắn nhất.
- Hoặc chưa kết nạp đủ n đỉnh nhưng không còn cạnh nào nối một đỉnh trong T với một đỉnh ngoài T. Ta kết luận đồ thị không liên thông và không thể tồn tại cây khung.

Như vậy cần xử lý hai vấn đề kỹ thuật khi cài đặt thuật toán Prim

- Làm thế nào để đo được khoảng cách từ một đỉnh ngoài cây tới cây cũng như cập nhật lại các khoảng cách đó khi cây được kết nạp thêm một đỉnh
- Làm thế nào để xác định đỉnh ngoài cây nằm gần cây nhất.

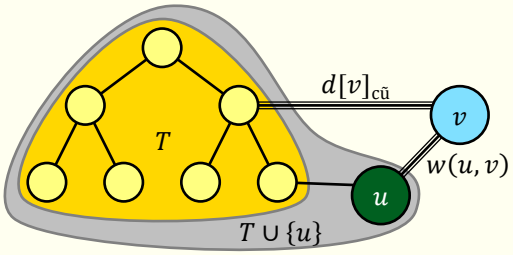
1.3.1. Nhãn khoảng cách

Thuật toán Prim sử dụng các nhãn khoảng cách $d[v]$ để lưu khoảng cách từ v tới T tại mỗi bước. Nếu v không có cạnh nào nối với một đỉnh trong T thì coi như $d[v] = +\infty$.

Mỗi khi cây T bổ sung thêm một đỉnh u, ta tính lại các nhãn khoảng cách theo công thức:

$$d[v]_{\text{mới}} = \min\{d[v]_{\text{cũ}}, w(u, v)\}$$

Tính đúng đắn của công thức có thể hình dung như sau: $d[v]$ là khoảng cách từ v tới cây T, theo định nghĩa là trọng số nhỏ nhất trong số các cạnh nối v với một đỉnh nằm trong T. Khi cây T "nở" ra thêm đỉnh u nữa mà đỉnh u này lại gần v hơn tất cả các đỉnh khác trong T, ta ghi nhận khoảng cách mới $d[v]$ là trọng số cạnh (u, v), nếu không ta vẫn giữ khoảng cách cũ (Hình 1-7).



Hình 1-7. Cơ chế cập nhật nhãn khoảng cách

Tại mỗi bước, đỉnh ngoài cây có nhãn khoảng cách nhỏ nhất sẽ được kết nạp vào cây, sau đó các nhãn khoảng cách được cập nhật và lặp lại. Mô hình cài đặt dưới đây sử dụng các biến với vai trò như sau:

- T: Cây khung đang xây dựng
- s: Một đỉnh bất kỳ, cây khung nhỏ nhất được "nở ra" từ đỉnh s
- $d[v]$: Khoảng cách từ đỉnh v tới cây.
- $trace[v]$: Cạnh nối tới v trên cây (v được "hút" vào cây theo cạnh này)

```
Input → G = (V, E, w);
s = «Một đỉnh bất kỳ»;
T = Ø; //Tập đỉnh trong cây được khởi tạo là rỗng
for (∀v ∈ V) d[v] = +∞; //Các đỉnh có nhãn khoảng cách d[.] = +∞
d[s] = 0;               //ngoại trừ d[s] = 0
for (n lần)
{
    u = «Đỉnh ∉ T có d[u] nhỏ nhất»;
    if (d[u] == +∞)
        break; //Không tồn tại cây khung do đồ thị không liên thông
    T = T U {u}; //Đưa u vào cây
    for (∀e = (u, v) ∈ E) //Xét các cạnh liên thuộc u
        if (v ∉ T && d[v] > w(u, v))
        {
            d[v] = w(u, v); //Cực tiểu hóa nhãn khoảng cách d[v] theo w(u, v)
            trace[v] = e; //Lưu vết: cạnh nhỏ nhất nối từ cây tới v là cạnh e
        }
}
if («Đồ thị liên thông»)
    for (∀v ∈ V - {s})
        Output ← Cạnh (trace[v], v);
else
    Output ← "Đồ thị không liên thông";
```

Để việc cài đặt được ngắn gọn, mô hình này có hơi khác so với tư tưởng ban đầu của thuật toán. Ban đầu ta không khởi tạo cây khung với 1 đỉnh bất kỳ mà khởi tạo cây khung rỗng, các nhãn $d[.]$ được đặt bằng $+\infty$ ngoại trừ $d[s] = 0$. Điều này làm cho ở lượt lặp đầu tiên, thuật toán sẽ đưa s vào cây khung, sau đó sửa nhãn các $d[v]$ thành $w(s, v)$ ($\forall v \neq s$). Từ lượt lặp thứ hai trở đi, thuật toán hoạt động chính xác như trình bày ban đầu.

1.3.2. Cài đặt trên đồ thị dày

Khi đồ thị có ít đỉnh và nhiều cạnh, thuật toán Prim cần nhiều lần cập nhật các nhãn khoảng cách $d[.]$. Vì vậy ta cố gắng giữ cho phép cập nhật nhãn $d[.]$ thực hiện trong thời gian $O(1)$ cho dù phải trả giá thời gian $O(n)$ cho việc tìm kiếm đỉnh ngoài cây có nhãn $d[.]$ nhỏ nhất.

Một mảng động (`std::vector<int>`) `Q` được dùng để chứa các đỉnh $\notin T$ có nhãn $d[.] < +\infty$. Ngoài ra có mảng `InTree` để đánh dấu, ở đây `InTree[v] == true` cho biết đỉnh v đã nằm trong cây T.

Tại mỗi bước lặp của thuật toán Prim, đầu tiên mảng `Q` được quét tuần tự để tìm đỉnh `u` ngoài cây có khoảng cách tới cây nhỏ nhất. Sau đó `u` được kết nạp vào T (`InTree[u] = true`) và loại bỏ khỏi `Q`: đưa phần tử cuối mảng `Q` vào thế chỗ `u`, và thực hiện phép hủy phần tử cuối mảng `Q` (`Q.pop_back()`). Khi sửa nhãn khoảng cách của một đỉnh, ta đẩy đỉnh đó vào Q nếu nó chưa có trong Q.

Vì cạnh có cấu trúc lớn nên ta sẽ dùng con trỏ để đại diện cho cạnh trong các phép xử lý. Đồ thị được biểu diễn bằng danh sách liên thuộc: mỗi đỉnh u cho ứng với một danh sách $adj[u]$ các cạnh liên thuộc với u. Như vậy với mỗi cạnh (u, v) thì con trỏ tới cạnh đó nằm trong cả danh sách $adj[u]$ và $adj[v]$.

PRIM_DENSE.cpp Thuật toán Prim trên đồ thị dày

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;
const int maxN = 1e5;
const int maxM = 1e5;
const int infty = 1e4 + 1; //Hằng số +∞ cần đủ lớn hơn mọi trọng số cạnh

struct TEdge  //Cấu trúc cạnh
{
    int u, v; //Hai đỉnh đầu mút
    int w;    //Trọng số
};
using PEdge = TEdge*; //Kiểu con trỏ tới cạnh

int n, m;      //Số đỉnh và số cạnh
TEdge e[maxM]; //Danh sách các cạnh
vector<PEdge> adj[maxN]; //adj[u] chứa các con trỏ tới các cạnh liên thuộc u
int d[maxN]; //Nhãn khoảng cách
bool InTree[maxN]; //Đánh dấu đỉnh đang ∈ cây
PEdge trace[maxN]; //trace[u]: con trỏ tới cạnh hút u vào cây

void ReadInput() //Nhập dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
        cin >> e[i].u >> e[i].v >> e[i].w;
    for (int i = 0; i < m; ++i)
    {   //Đưa &e[i] vào danh sách liên thuộc của 2 đầu mút
        adj[e[i].u].push_back(&e[i]);
        adj[e[i].v].push_back(&e[i]);
    }
}

void Init() //Khởi tạo
{
    fill(d, d + n, infty); //Các đỉnh có nhãn d[.] = +∞
    d[0] = 0;              //Ngoại trừ d[0] = 0;
    fill(InTree, InTree + n, false); //Cây được khởi tạo không có đỉnh nào
}

```

```cpp
void Prim() //Thuật toán Prim
{
    vector<int> Q(1, 0); //Q = {0}: Tập các đỉnh ngoài cây có nhãn d[.] < +∞
    while (!Q.empty()) //Lặp chừng nào Q ≠ Ø
    {
        auto it = min_element(Q.begin(), Q.end(), [](int i, int j)
        {
            return d[i] < d[j];
        }); //it = iterator tới đỉnh có d[.] nhỏ nhất trong Q
        int u = *it; //u: Đỉnh ∈ Q có d[.] nhỏ nhất
        *it = Q.back(); //Đưa phần tử cuối Q vào thế chỗ u
        Q.pop_back();   //Hủy phần tử cuối Q
        InTree[u] = true; //Đánh dấu u thuộc cây
        for (PEdge p: adj[u]) //Xét các cạnh *p = (u, v) liên thuộc u
        {
            int v = p->u + p->v - u; //Cách nhanh hơn: v = p->u ^ p->v ^ u;
            if (!InTree[v] && d[v] > p->w) //v ∉ cây và d[v] cần cập nhật lại
            {
                bool NotInQ = d[v] == infty; //d[v] = +∞ tức là v ∉ Q
                d[v] = p->w; //Cập nhật d[v] mới theo trọng số cạnh *p;
                trace[v] = p; //Vết: Cạnh nhỏ nhất nối v với cây là cạnh *p
                if (NotInQ) //v đang ∉ Q
                    Q.push_back(v);
            }
        }
    }
}

void Print() //In kết quả
{
    if (find(InTree, InTree + n, false) != InTree + n) //Có đỉnh ngoài cây
        cout << "Graph is not connected!"; //Đồ thị không liên thông
    else
    {
        cout << "Minimum Spanning Tree:\n";
        int TreeWeight = 0;
        for (int v = 1; v < n; ++v) //Xét mọi đỉnh v ngoại trừ đỉnh 0
        {
            PEdge p = trace[v]; //Cạnh *p được chọn vào cây khung
            cout << "Edge " << p - e << ": ("
                 << p->u << ", " << p->v << "); "
                 << "Weight = " << p->w << '\n';
            TreeWeight += p->w;
        }
        cout << "Tree Weight: " << TreeWeight << '\n';
    }
}

int main()
{
    ReadInput();
    Init();
    Prim();
    Print();
}
```

1.3.3. Cài đặt trên đồ thị thưa

Trong trường hợp đồ thị thưa (có nhiều đỉnh, ít cung), những cấu trúc dữ liệu biểu diễn hàng đợi ưu tiên thường được sử dụng để biểu diễn tập Q gồm các đỉnh ngoài cây có nhãn khoảng cách $d[.] < +\infty$, mục đích là để dễ dàng xác định đỉnh ngoài cây gần với cây nhất. Cài đặt dưới đây sử dụng lớp mẫu `std::priority_queue` của C++ để biểu diễn tập Q.

Cấu trúc dữ liệu Q sẽ chứa các bản ghi (struct) gồm hai trường $\{vertex, dlab\}$: $vertex$ là số hiệu đỉnh và $dlab$ là nhãn khoảng cách $d[vertex]$ của đỉnh đó vào thời điểm nó được cập nhật. Bản ghi nào có $dlab$ càng nhỏ thì càng được ưu tiên hơn. Ban đầu Q chỉ chứa một bản ghi có $vertex = s$ và $dlab = 0$.

Mỗi khi một nhãn khoảng cách $d[v]$ được cập nhật mới, ta đẩy bản ghi $\{v, d[v]\}$ vào Q, tức là mỗi đỉnh v có thể được đẩy vào Q nhiều lần cùng với nhãn $d[v]$ tương ứng tại nhiều thời điểm khác nhau, lần đẩy vào sau vô hiệu hóa lần đẩy vào trước. Một bản ghi $\{vertex, dlab\}$ được gọi là còn hiệu lực nếu $vertex \notin T$ và $d[vertex] = dlab$, ngược lại nó được coi là vô hiệu.

Tại mỗi bước, thuật toán sẽ lấy ra một bản ghi $\{vertex, dlab\}$ có $dlab$ nhỏ nhất từ Q. Nếu bản ghi này vô hiệu lực, ta bỏ qua và lặp lại để lấy một bản ghi khác ra xử lý… cho tới khi gặp bản ghi $\{vertex, dlab\}$ có hiệu lực, khi đó đỉnh $vertex$ sẽ được hút vào cây, các đỉnh v ngoài cây kề với $vertex$ sẽ được duyệt để cập nhật lại nhãn $d[v]$, khi một nhãn $d[v]$ thay đổi, bản ghi $\{v, d[v]\}$ sẽ được đẩy vào Q chờ xử lý…

PRIM_SPARSE.cpp Thuật toán Prim trên đồ thị thưa

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;
const int maxN = 1e5;
const int maxM = 1e5;
const int infty = 1e4 + 1; //Hằng số +∞ phải đủ lớn hơn mọi trọng số cạnh

struct TEdge //Cấu trúc cạnh
{
    int u, v;      //Hai đỉnh đầu mút
    int w;         //Trọng số
    bool selected; //Có chọn vào cây khung hay không
};
using PEdge = TEdge*; //Kiểu con trỏ tới cạnh

int n, m;      //n: Số đỉnh, m: Số cạnh
TEdge e[maxM]; //Danh sách các cạnh
vector<PEdge> adj[maxN]; //adj[u] chứa các con trỏ tới các cạnh liên thuộc u
int d[maxN]; //Nhãn khoảng cách
PEdge trace[maxN]; //trace[u]: cạnh hút u vào cây
bool InTree[maxN]; //Đánh dấu đỉnh trong cây

struct TPQItem
{
    int vertex, dlab; //Số hiệu đỉnh và nhãn khoảng cách
    bool operator < (const TPQItem& other) const //Ưu tiên other hơn nếu…
    {
        return dlab > other.dlab;
    }
    bool Valid() const //Bản ghi có hiệu lực không
    {
        return !InTree[vertex] && dlab == d[vertex];
    }
};

```

```cpp
void ReadInput() //Nhập dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < m; ++i)
    {
        cin >> e[i].u >> e[i].v >> e[i].w;
        e[i].selected = false; //Chưa cạnh nào được chọn vào cây khung
    }
    for (int i = 0; i < m; ++i)
    {   //Đưa &e[i] vào danh sách liên thuộc của 2 đầu mút
        adj[e[i].u].push_back(&e[i]);
        adj[e[i].v].push_back(&e[i]);
    }
}

void Init() //Khởi tạo
{
    fill(d, d + n, infty); //Các đỉnh có nhãn d[.] = +∞
    d[0] = 0;              //Ngoại trừ d[0] = 0;
    fill(InTree, InTree + n, false); //Cây đang rỗng
}

void Prim() //Thuật toán Prim
{
    priority_queue<TPQItem> Q;
    Q.push({0, 0});
    while (!Q.empty()) //Lặp chừng nào X ≠ Ø
    {
        TPQItem item = Q.top(); //item: bản ghi có .dlab nhỏ nhất trong Q
        Q.pop(); //Bỏ item ra khỏi Q
        if (!item.Valid()) //item vô hiệu lực
            continue;      //→ bỏ qua
        int u = item.vertex; //u: đỉnh ngoài cây có nhãn d[u] nhỏ nhất
        InTree[u] = true; //Đánh dấu u thuộc cây
        for (PEdge p: adj[u]) //Xét các cạnh *p = (u, v) liên thuộc u
        {
            int v = p->u + p->v - u; //Cách nhanh hơn: v = p->u ^ p->v ^ u;
            if (!InTree[v] && d[v] > p->w) //v ∉ cây và d[v] cần cập nhật lại
            {
                d[v] = p->w; //Cập nhật d[v] mới theo trọng số cạnh *p;
                trace[v] = p; //Vết: Cạnh nhỏ nhất nối v với cây là cạnh *p
                Q.push({v, d[v]}); //Đẩy vào Q một bản ghi {v, d[v]}
            }
        }
    }
}

```

```
85  void Print() //In kết quả
86  {
87      if (find(InTree, InTree + n, false) != InTree + n) //Còn đỉnh ngoài cây
88          cout << "Graph is not connected!"; //Đồ thị không liên thông
89      else
90      {
91          cout << "Minimum Spanning Tree:\n";
92          int TreeWeight = 0;
93          for (int v = 1; v < n; ++v) //Xét mọi đỉnh v ngoại trừ đỉnh 0
94          {
95              PEdge p = trace[v]; //Cạnh *p được chọn vào cây khung
96              cout << "Edge " << p - e << ": ("
97                   << p->u << ", " << p->v << "); "
98                   << "Weight = " << p->w << '\n';
99              TreeWeight += p->w;
100         }
101         cout << "Tree Weight: " << TreeWeight << '\n';
102     }
103 }
104
105 int main()
106 {
107     ReadInput();
108     Init();
109     Prim();
110     Print();
111 }
```

1.3.4. Tính đúng đắn và thời gian thực hiện giải thuật

Tính đúng đắn của thuật toán Prim cũng dễ dàng suy ra được từ hệ quả của Bổ đề 1-1: Gọi A là tập cạnh của cây T tại mỗi bước, xét lát cắt tách tập đỉnh V làm 2 tập rời nhau, một tập gồm các đỉnh $\in T$ và tập còn lại gồm các đỉnh $\notin T$. Đỉnh v được kết nạp vào cây T tại mỗi bước được "hút" vào cây theo cạnh nhỏ nhất nối một đỉnh trong T với một đỉnh ngoài T nên cạnh này là cạnh nhẹ của lát cắt, tức là cạnh an toàn với tập A, việc bổ sung cạnh này vào A vẫn đảm bảo A là tập con của tập cạnh một cây khung ngắn nhất.

Nếu đồ thị G liên thông, thuật toán Prim sẽ tìm ra cây khung nhỏ nhất. Nếu đồ thị G không liên thông, thuật toán Prim sẽ tìm ra cây khung nhỏ nhất của thành phần liên thông chứa đỉnh khởi tạo cây.

Cách cài đặt trên đồ thị dày thực hiện trong thời gian $O(n^2 + m)$ còn cách cài đặt trên đồ thị thưa thực hiện trong thời gian $O(m \log m)$. Thuật toán trên đồ thị thưa có thể thực hiện trong thời gian $O(n \log n + m)$ nếu sử dụng Fibonacci Heap để biểu diễn tập Q. Thuật toán Prim có cách thức thực hiện khá giống với thuật toán Dijkstra, chỉ khác nhau ở định nghĩa và cách thức cập nhật nhãn khoảng cách.

1.4. Chọn lựa thuật toán

Ta đã khảo sát hai thuật toán để tìm cây khung nhỏ nhất trên đồ thị, câu hỏi đặt ra là thuật toán nào tốt hơn?

Nhìn vào đánh giá độ phức tạp tính toán, thuật toán Kruskal tỏ ra tốt hơn trên đồ thị thưa còn thuật toán Prim lại tốt hơn trên đồ thị dày. Phiên bản trên đồ thị thưa của thuật toán Prim sẽ tốt hơn nếu dùng Fibonacci Heap còn thuật toán Kruskal sẽ tốt hơn nếu có thể sắp xếp được danh sách cạnh trong thời gian $O(m)$ (chẳng hạn dùng các thuật toán sắp xếp cơ số hoặc đếm phân phối). Tuy nhiên những cải tiến đó khá phức tạp và khó so sánh bằng

các bộ dữ liệu thực tế. Nói cách khác, chênh lệch về tốc độ giữa hai thuật toán là không đáng kể khi cài đặt chương trình chạy với dữ liệu thực tế.

Việc lựa chọn thuật toán Kruskal hay Prim đôi khi phụ thuộc vào việc ta sử dụng kết quả của thuật toán để làm việc gì tiếp theo. Thuật toán Kruskal có lợi khi ta có thể tìm được rừng cây khung của tất cả các thành phần liên thông, trong khi đó thuật toán Prim có lợi hơn khi ta muốn dò đường đi trên cây khung, hoặc thiết lập quan hệ cha con trên cây khung tìm được.

1.5. Một số thuật toán khác

Thuật toán tìm cây khung nhỏ nhất đầu tiên được Otakar Borůvka đề xuất năm 1926 [3] với mục đích thiết kế mạng lưới điện cho vùng Moravia. Ban đầu với $T = \emptyset$, thuật toán thực hiện qua từng giai đoạn: Mỗi giai đoạn, gọi là bước Borůvka, thuật toán duyệt tất cả các cạnh (u, v) và bổ sung vào T nếu nó là cạnh trọng số nhỏ nhất liên thuộc với u hoặc là cạnh trọng số nhỏ nhất liên thuộc với v, điều này đảm bảo các cạnh trong T không tạo ra chu trình. Tiếp theo, các thành phần liên thông với tập cạnh T được chập lại thành một đỉnh để chuẩn bị cho bước tiếp theo. Thuật toán kết thúc khi đồ thị chỉ còn đúng 1 đỉnh.

Mỗi bước Borůvka thực hiện trong thời gian $O(|E|)$ và số bước Borůvka phải thực hiện là $O(\log|V|)$ vì sau mỗi bước thì số đỉnh của đồ thị giảm đi ít nhất một nửa. Thuật toán Borůvka thực hiện trong thời gian $O(|E| \log|V|)$

Một thuật toán khác ít khi được sử dụng để tìm cây khung nhỏ nhất là thuật toán xóa ngược (reverse-delete): Xét các cạnh từ trọng số lớn tới trọng số nhỏ và xóa luôn cạnh đó nếu việc này không làm mất tính liên thông của đồ thị. Thuật toán có thời gian thực hiện $O(|E| \log|V| (\log\log|V|)^3$ và cần sử dụng thêm những cấu trúc dữ liệu phức tạp.

Một số nghiên cứu khác cũng đã cho ra đời những thuật toán tốt hơn. Thuật toán ngẫu nhiên của Karger, Klein và Tarjan [4] có thể tìm được cây khung nhỏ nhất trong thời gian $O(|V| + |E|)$ với xác suất rất cao. Thuật toán nhanh nhất được biết hiện nay do Chazelle đề xuất [5] có độ phức tạp $O\big(|E|\alpha(|V|, |E|)\big)$ dựa trên cấu trúc dữ liệu SoftHeap [6]

1.6. Một số vấn đề liên quan

Cây Steiner (*Steiner tree*): Cây trọng số nhỏ nhất chứa toàn bộ một tập đỉnh nằm trong đồ thị cho trước. Bài toán tìm cây Steiner được chứng mình là bài toán NP-đầy đủ.

Cây khung k đỉnh (*k-minimum spanning tree*): Cây trọng số nhỏ nhất chứa đúng k đỉnh nằm trong đồ thị cho trước. Bài toán tìm cây khung k đỉnh cũng là bài toán NP-đầy đủ.

Cây khung nhỏ nhất với khoảng cách Euclid (*euclidean minimum spanning tree*) và khoảng cách Manhattan(*rectilinear minimum spanning tree*): có vài nghiên cứu những trường hợp đặt biệt này của hàm trọng số để đề xuất những thuật toán hiệu quả hơn trong trường hợp hai chiều hoặc nhiều chiều.

Cây khung nhỏ nhất với ràng buộc sức chứa (*Capacitated minimum spanning tree*): Cho đồ thị với một đỉnh đặc biệt r và một hằng số ràng buộc sức chứa c, bài toán đặt ra là tìm cây khung trọng số nhỏ nhất sao cho nếu bỏ đi đỉnh r thì các cây con tách ra có số nút không quá c. Đây là bài toán NP-khó.

Cây khung nhỏ nhất với bậc bị chặn (*degree-constrained minimum spanning tree*): Tìm cây khung trọng số nhỏ nhất với ràng buộc bậc của mỗi đỉnh trên cây không được vượt quá một hằng số cho trước. Đây cũng là bài toán NP-khó.

Cây khung nhỏ nhất trong đồ thị động (*Dynamic Minimum Spanning Tree*): Với đồ thị mà trọng số cạnh liên tục thay đổi cùng với truy vấn tìm cây khung nhỏ nhất, có vài thuật toán hiệu quả để cập nhật cây khung nhỏ nhất sau mỗi sự thay đổi thay vì phải tìm từ đầu. Những thuật toán này thường tích hợp một số cấu trúc dữ liệu phức tạp.

Bài tập 1-1

Cho T là cây khung nhỏ nhất của đồ thị G và (u, v) là một cạnh trong T. Chứng minh rằng nếu ta trừ trọng số cạnh (u, v) đi một số dương thì T vẫn là cây khung nhỏ nhất của đồ thị G.

Bài tập 1-2

Cho G là một đồ thị vô hướng liên thông, C là một chu trình trên G và e là cạnh trọng số lớn nhất của C. Chứng minh rằng nếu ta loại bỏ cạnh e khỏi đồ thị thì không ảnh hưởng tới trọng số của cây khung nhỏ nhất.

Bài tập 1-3

Chứng minh rằng đồ thị có duy nhất một cây khung nhỏ nhất nếu với mọi lát cắt của đồ thị, có duy nhất một cạnh nhẹ nối hai tập của lát cắt. Cho một ví dụ để chỉ ra rằng điều ngược lại không đúng.

Bài tập 1-4 (Bottleneck Spanning tree)

Ta gọi mức "thắt" của cây khung là trọng số lớn nhất của một cạnh trên cây khung đó. Bài toán đặt ra là tìm cây khung có mức thắt cực tiểu.

Chứng minh rằng cây khung nhỏ nhất là một trong những cây khung có mức thắt cực tiểu.

Bài tập 1-5

Gọi T là một cây khung nhỏ nhất của đồ thị vô hướng liên thông G, ta giảm trọng số của một cạnh không nằm trong cây T, hãy tìm thuật toán để xây dựng cây khung nhỏ nhất của đồ thị mới.

Bài tập 1-6

Giả sử rằng đồ thị vô hướng liên thông G có cây khung nhỏ nhất T, người ta thêm vào đồ thị một đỉnh mới và một số cạnh liên thuộc với đỉnh đó. Tìm thuật toán xác định cây khung nhỏ nhất của đồ thị mới.

Bài tập 1-7

Giáo sư X đề xuất một thuật toán tìm cây khung ngắn nhất dựa trên ý tưởng chia để trị: Với đồ thị vô hướng liên thông $G = (V, E)$, phân hoạch tập đỉnh V làm hai tập rời nhau V_1, V_2 mà lực lượng của hai tập này hơn kém nhau không quá 1. Gọi E_1 là tập các cạnh chỉ liên thuộc với các đỉnh $\in V_1$ và E_2 là tập các cạnh chỉ liên thuộc với các đỉnh $\in V_2$. Tìm cây khung nhỏ nhất trên đồ thị $G_1 = (V_1, E_1)$ và $G_2 = (V_2, E_2)$ bằng thuật toán đệ quy, sau đó chọn cạnh trọng số nhỏ nhất nối V_1 với V_2 để nối hai cây khung tìm được thành một cây.

Chứng minh tính đúng đắn của thuật toán hoặc chỉ ra một phản ví dụ cho thấy thuật toán sai.

Bài tập 1-8 (Cây khung nhỏ thứ nhì)

Cho $G = (V, E, w)$ là đồ thị vô hướng liên thông có trọng số, giả sử rằng $|E| \geq |V|$ và các trọng số cạnh là hoàn toàn phân biệt. Gọi $\mathcal{T}$ là tập tất cả các cây khung của G và A là cây khung nhỏ nhất của G, khi đó cây khung nhỏ thứ nhì được định nghĩa là cây khung $B \in \mathcal{T}$ thỏa mãn:

$$w(B) = \min_{T \in \mathcal{T} - \{A\}} \{w(T)\}$$

- Chỉ ra rằng đồ thị G có duy nhất một cây khung nhỏ nhất là A, nhưng có thể có nhiều cây khung nhỏ thứ nhì.
- Chứng minh rằng luôn tồn tại một cạnh $(u, v) \in A$ và $(x, y) \notin A$ để nếu ta loại bỏ cạnh (u, v) khỏi A rồi thêm cạnh (x, y) vào A thì sẽ được cây khung nhỏ thứ nhì.
- Với $\forall u, v \in V$, gọi $f[u, v]$ là cạnh mang trọng số lớn nhất trên đường đi duy nhất từ u tới v trên cây A. Tìm thuật toán $O(|V|^2)$ để tính tất cả các $f[u, v]$, $\forall u, v \in V$.
- Tìm thuật toán hiệu quả để tìm cây khung nhỏ thứ nhì của đồ thị.

Bài tập 1-9

Cho s và t là hai đỉnh của một đồ thị vô hướng có trọng số $G = (V, E, w)$. Tìm một đường đi từ s tới t thỏa mãn: Trọng số cạnh lớn nhất đi qua trên đường đi là nhỏ nhất có thể.

Bài tập 1-10 (Euclidean Minimum Spanning Tree)

Trong trường hợp các đỉnh của đồ thị đầy đủ được đặt trên mặt phẳng trực chuẩn và trọng số cạnh nối giữa hai đỉnh chính là khoảng cách hình học giữa chúng. Người ta có một phép tiền xử lý để giảm bớt số cạnh của đồ thị bằng thuật toán tam giác phân Delaunay ($O(n \lg n)$), đồ thị sau phép tam giác phân Delaunay sẽ còn không quá $3n$ cạnh, do đó sẽ làm các thuật toán tìm cây khung nhỏ nhất hoạt động hiệu quả hơn. Hãy tự tìm hiểu về phép tam giác phân Delaunay và cài đặt chương trình để tìm cây khung nhỏ nhất.

Bài tập 1-11

Trên một nền phẳng với hệ toạ độ trực chuẩn đặt n máy tính, máy tính thứ i được đặt ở toạ độ (x_i, y_i). Đã có sẵn một số dây cáp mạng nối giữa một số cặp máy tính. Cho phép nối thêm các dây cáp mạng nối giữa từng cặp máy tính. Chi phí nối một dây cáp mạng tỉ lệ thuận với khoảng cách giữa hai máy cần nối. Hãy tìm cách nối thêm các dây cáp mạng để cho các máy tính trong toàn mạng là liên thông và chi phí nối mạng là nhỏ nhất.

Bài tập 1-12

Hệ thống điện trong thành phố được cho bởi n trạm biến thế và các đường dây điện nối giữa các cặp trạm biến thế. Mỗi đường dây điện e có độ an toàn là $p(e) \in (0,1]$. Độ an toàn của cả lưới điện là tích độ an toàn trên các đường dây. Hãy tìm cách bỏ đi một số dây điện để cho các trạm biến thế vẫn liên thông và độ an toàn của mạng là lớn nhất có thể.

Gợi ý: Bằng kỹ thuật lấy logarithm, độ an toàn trên lưới điện trở thành tổng độ an toàn trên các đường dây.

[1] J. B. Kruskal Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society,* vol. 7, no. 1, pp. 48-50, 1956.

[2] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal,* vol. 36, pp. 1389-1401, 1957.

[3] O. Borůvka, "O jistém problému minimálním (Czech) [About a certain minimal problem]," *Práce Moravské přírodovědecké společnosti,* pp. 37-58, 1926.

[4] D. R. Karger, P. N. Klein and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM,* vol. 42, no. 2, pp. 321-328, 1995.

[5] B. Chazelle, "A minimum spanning tree algorithm with inverse-Ackermann type complexity," *Journal of the ACM,* vol. 47, no. 6, p. 1028–1047, 2000.

[6] B. Chazelle, "The soft heap: an approximate priority queue with optimal error rate," *Journal of the ACM,* vol. 47, no. 6, pp. 1012-1027, 2000.