

Bài toán Luồng cực đại trên mạng

Bài toán Luồng cực đại trên mạng

Tác giả:

- ▶ Nguyễn Đức Kiên, Trường Đại học Công nghệ, ĐHQGHN.

Reviewer:

- ▶ Phạm Công Minh - THPT chuyên Khoa học Tự Nhiên, ĐHQGHN
- ▶ Đặng Đoàn Đức Trung - UT Austin
- ▶ Nguyễn Minh Nhật - Trường THPT chuyên Khoa học Tự nhiên, ĐHQGHN

Luồng cực đại (Maximum Flow) và Lát cắt cực tiểu/hẹp nhất (Minimum Cut) là những bài toán quan trọng trong lớp các bài toán về đồ thị. Bài viết sau đây sẽ giới thiệu một vài nội dung cơ bản về bài toán luồng cực đại và các thuật toán liên quan.

Một số khái niệm sử dụng trong bài viết

Để hiểu hơn về phần này, bạn đọc nên có sẵn những kiến thức cơ bản về đồ thị, cũng như biểu diễn và duyệt (BFS, DFS, ...) chúng.

Bài viết sẽ không nêu lại các khái niệm cơ bản về đồ thị.

- ▶ **Ký hiệu đồ thị** có $G(V, E)$: Đồ thị tập các đỉnh là V và tập các cạnh là E
- ▶ **Cạnh đi vào đỉnh** u : Các cạnh có dạng (v, u) , với v là đỉnh bất kỳ của đồ thị.
- ▶ **Cạnh đi ra khỏi đỉnh** u : Các cạnh có dạng (u, v) , với v là đỉnh bất kỳ của đồ thị.
- ▶ **Đường đi đơn từ s tới t** : Dãy các đỉnh $s, u_1, u_2, \dots, u_k, t$ sao cho giữa hai đỉnh liên tiếp trong dãy tồn tại một cạnh nối chúng theo đúng chiều như trên.

Bài toán Luồng cực đại

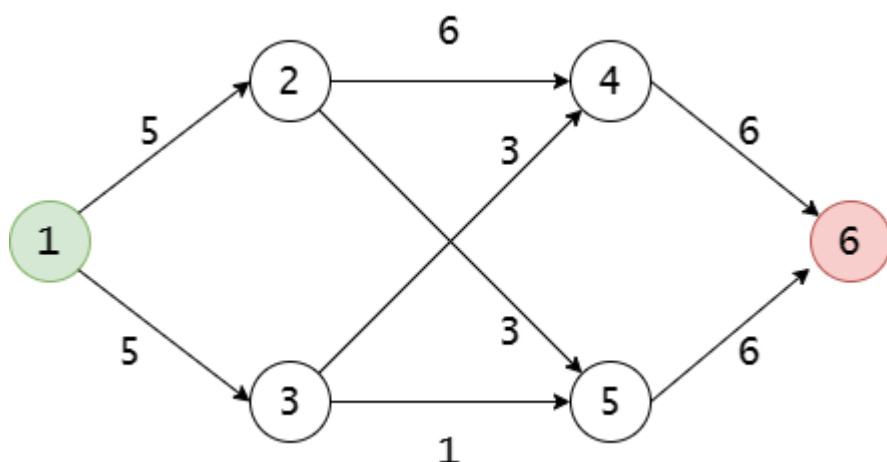
Các định nghĩa

Có rất nhiều hình ảnh thực tế để miêu tả một mạng và luồng trên mạng đó, như một mạng điện, một mạng kết nối dữ liệu giữa các máy, hay phổ biến hơn là một hệ thống ống nước.

Một đồ thị $G(V, E)$ được gọi là **mạng** (network) nếu nó là đồ thị **có hướng**, trong đó:

- ▶ Tồn tại một đỉnh s không có cạnh đi vào, gọi là **đỉnh phát/nguồn** (source)

- ▶ Tồn tại một đỉnh t không có cạnh đi ra, gọi là **đỉnh thu/đích** (sink)
- ▶ Mỗi cạnh (u, v) được gán một trọng số $c(u, v)$, gọi là **khả năng thông qua/dung lượng** (capacity) của cạnh.

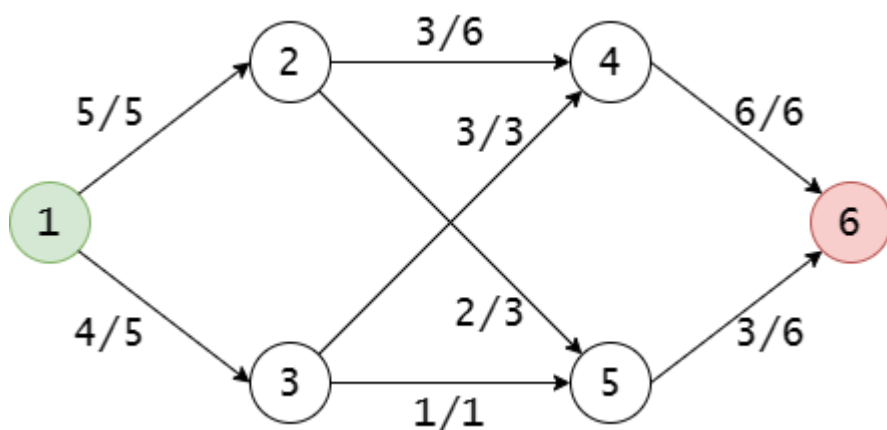


Một mạng hợp lệ. Đỉnh phát và đỉnh thu được đánh dấu bằng hai màu khác.

Một **luồng** (flow) trên mạng $G(V, E)$ là một phép gán cho mỗi cạnh (u, v) một số thực $f(u, v)$ thỏa mãn:

- ▶ Luồng trên mỗi cạnh có giá trị không vượt quá khả năng thông qua của cạnh đó:
 $0 \leq f(u, v) \leq c(u, v), \forall u, v \in V$
- ▶ Với mọi đỉnh v không trùng với đỉnh phát s và đỉnh thu t , tổng luồng trên các cạnh đi vào v bằng tổng luồng trên các cạnh đi ra v . Tính chất này tương đối giống với định luật I Kirchoff của dòng điện.

$$\sum_{v \in V, \exists (v, u) \in E} f(v, u) = \sum_{w \in V, \exists (u, w) \in E} f(u, w)$$
- ▶ Giá trị $f(u, v)$ được gọi là **luồng trên cạnh** (u, v)
- ▶ **Giá trị của luồng** là tổng luồng trên các cạnh đi ra khỏi đỉnh phát, cũng chính là tổng luồng trên các cạnh đi vào đỉnh thu.

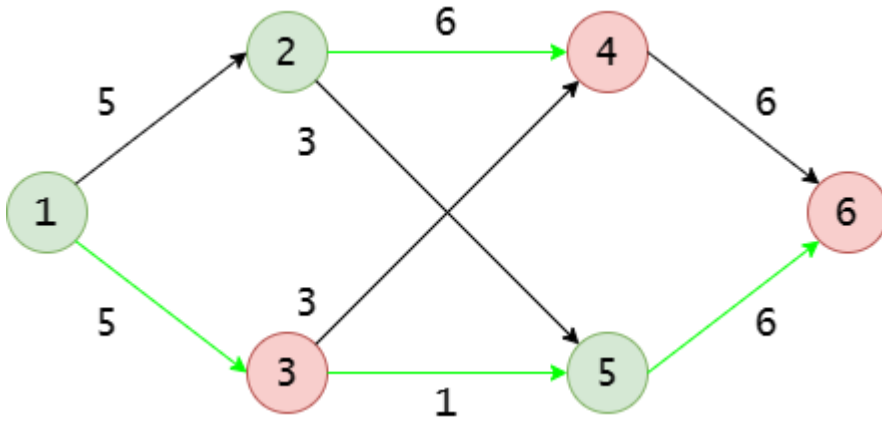


Một luồng hợp lệ. Giá trị f/c trên cạnh biểu diễn luồng/khả năng thông qua.

Một **lát cắt** (cut) (A, B) trên mạng là một cách chia các đỉnh trên đồ thị mạng thành hai tập hợp sao cho $s \in A, t \in B$.

Tổng các giá trị khả năng thông qua trên các cạnh nối giữa một đỉnh thuộc A và một đỉnh thuộc B được gọi là **khả năng thông qua** (cut value) của lát cắt (A, B)

$$c(A, B) = \sum_{u \in A, v \in B} c(u, v)$$



Một lát cắt hợp lệ với hai tập $A = \{1, 2, 5\}$ và $B = \{3, 4, 6\}$. Mỗi tập con của lát cắt được đánh dấu bằng một màu khác nhau. Lát cắt này có khả năng thông qua là $6 + 5 + 1 + 6 = 17$.

Định lý: Trên cùng một mạng, tất cả mọi luồng đều có giá trị không lớn hơn khả năng thông qua của một lát cắt bất kỳ.

► Chứng minh

Nếu ta hiểu mạng như một hệ thống ống nước, nó sẽ như sau:

- Nước chảy qua một hệ thống các ống, từ nguồn nước (đỉnh phát) đến bồn chứa (đỉnh thu).
- Mỗi ống có một giới hạn nhất định. Lượng nước chảy qua ống này không thể vượt quá giới hạn này.
- Hiển nhiên, tại mỗi điểm nút (trừ điểm đầu và điểm cuối), có bao nhiêu nước đến thì sẽ có bấy nhiêu nước chảy đi. Nước không tự sinh ra và mất đi, chúng chỉ chảy từ điểm này sang điểm khác.
- Và tất nhiên tổng lượng nước xuất hiện trong mạng sẽ là lượng nước ta cấp cho nguồn. Bể chứa cũng sẽ thu được từng đó nước.
- Còn một lát cắt là một cách bỏ đi các ống sao cho nước không thể chảy từ nguồn đến bể nữa bằng bất kỳ cách nào.

Bài toán

Đề bài: Cho mạng $G(V, E)$ với m đỉnh và n cạnh có đỉnh phát là s , đỉnh thu là t ($n \leq 1000, 1 \leq s, t \leq n$). Hãy tìm một luồng trong mạng sao cho giá trị của nó là lớn nhất. Luồng này gọi là **luồng cực đại** trên mạng G .

Đề bài VNOI: [NKFLOW](#)

Phương pháp Ford-Fulkerson. Thuật toán Edmonds-Karp.

► Đôi lời về lịch sử thuật toán

Các khái niệm

Giả sử tại một thời điểm, ta đã có một luồng trên đồ thị, với giá trị luồng trên cạnh (u, v) là $f(u, v)$.

Với mọi cạnh (u, v) , ta định nghĩa thêm giá trị $f(v, u) = -f(u, v)$. Về mặt ý nghĩa, việc định nghĩa này cho ta biết luồng hiện tại trên cạnh này có thể giảm đi một lượng bao nhiêu.

Lưu ý rằng ta **không** định nghĩa $c(v, u) = c(u, v)$, giá trị này vẫn được mặc định bằng 0.

Định nghĩa **luồng thặng dư** (residual flow) trên một cạnh tại một thời điểm là hiệu của khả năng thông qua và giá trị luồng hiện tại trên cạnh đó:

$$r(u, v) = c(u, v) - f(u, v)$$

Giá trị này cũng áp dụng cho cả các cạnh đảo (cạnh có luồng âm), khi đó

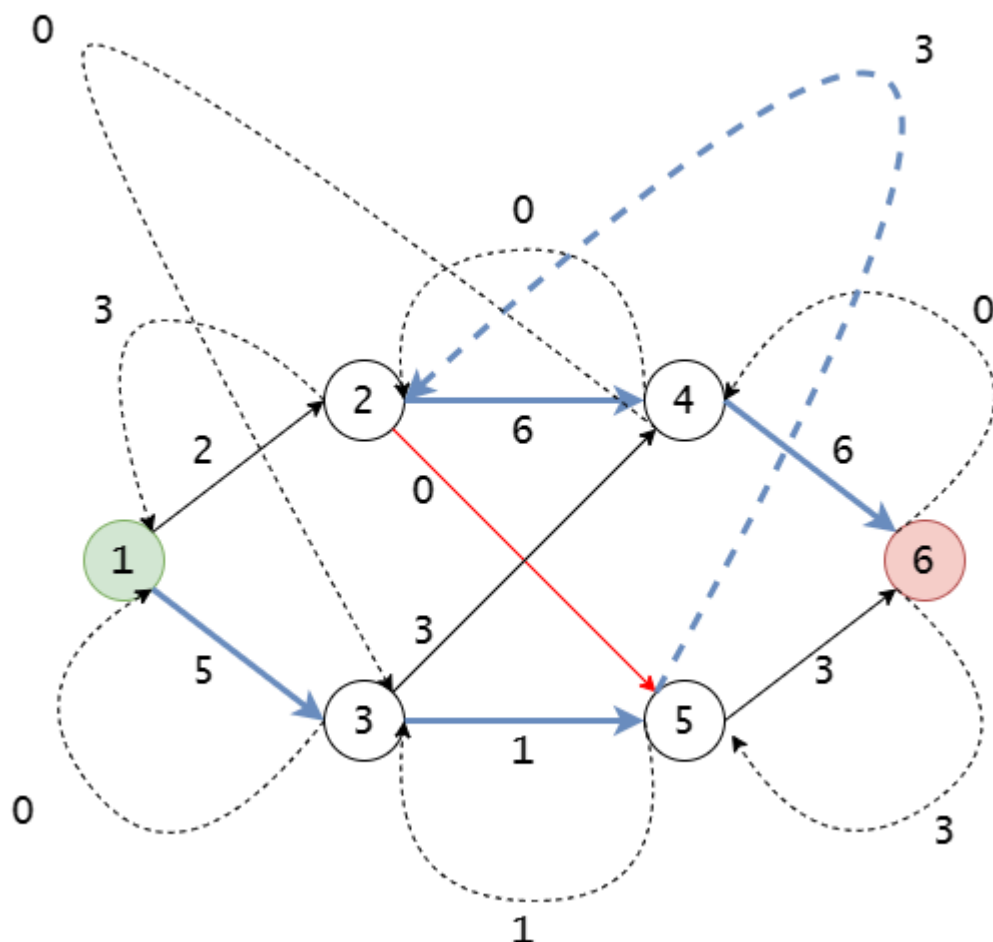
$$r(v, u) = 0 - f(v, u) = f(u, v).$$

Ta có thể hiểu rằng giá trị luồng thặng dư cho biết còn có thể thêm vào luồng này một lượng bao nhiêu.

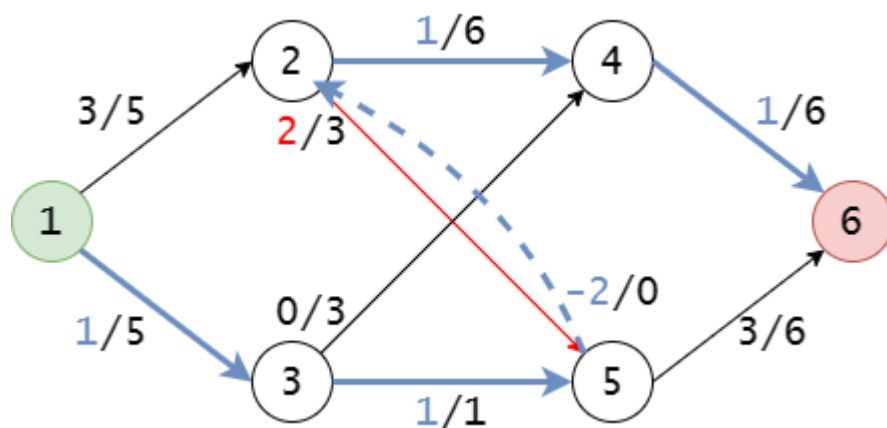
Với các giá trị $r(u, v)$ này, ta có thể xây dựng một **đồ thị thặng dư/đồ thị tăng luồng** (residual network). Ứng với mỗi cạnh (u, v) trên mạng ban đầu, trên đồ thị thặng dư sẽ có hai cạnh:

- Cạnh (u, v) , với trọng số là $r(u, v)$. Mỗi cạnh loại này cho ta biết có thể tăng luồng trên mạng ban đầu bao nhiêu.
- Cạnh (v, u) , với trọng số là $f(u, v)$. Mỗi cạnh loại này cho ta biết có thể giảm luồng trên mạng ban đầu bao nhiêu.

Một **đường tăng luồng** (augmenting path) là một đường đi đơn trên đồ thị thặng dư. Đối chiếu lại với đồ thị gốc, đó sẽ là một đường đi đơn (có thể đi ngược chiều cạnh) qua những cạnh có $r(u, v) > 0$. Trên đường này, chúng ta có thể thực hiện tăng giá trị của luồng trên mỗi cạnh.



Đường màu xanh là một đường tăng luồng trên đồ thị thặng dư trên. Các cạnh đứt chính là các cạnh "ngược" so với mạng ban đầu; chúng có giá trị f âm.



Đem đối chiếu đồ thị thặng dư trên về đồ thị gốc, ta được đường tăng luồng như hình trên. Trong hình dưới, giá trị của luồng (f) trên các cạnh thuộc đường tăng luồng đã được tăng 1 đơn vị so với đồ thị thặng dư bên trên.

Việc xây dựng cả một đồ thị thặng dư sau từng bước rất tốn thời gian và bộ nhớ. Vì vậy, trong phương pháp Ford-Fulkerson chúng ta sẽ chỉ sử dụng đồ thị gốc, và thực hiện tìm đường tăng luồng trực tiếp trên đồ thị này.

Còn nếu bạn muốn hiểu theo kiểu "ống nước" thì đường tăng luồng có thể coi như một đường nước chảy từ nguồn đến bể chứa. Đối với các "ống đi ngược" như "ống" (5, 2) trên hình, ta hiểu đây là một cách phân phối lại nước: thêm 1 đơn vị nước vào nút 5 sẽ dẫn đến việc phải bớt 1 đơn vị từ ống (2, 5) để đảm bảo đoạn sau vẫn đủ nước; ở đầu 2 phần nước thay vì chảy vào ống này đi ra đầu 5 thì nó sẽ đưa phần nước này sang ống (2, 4).

Thuật toán

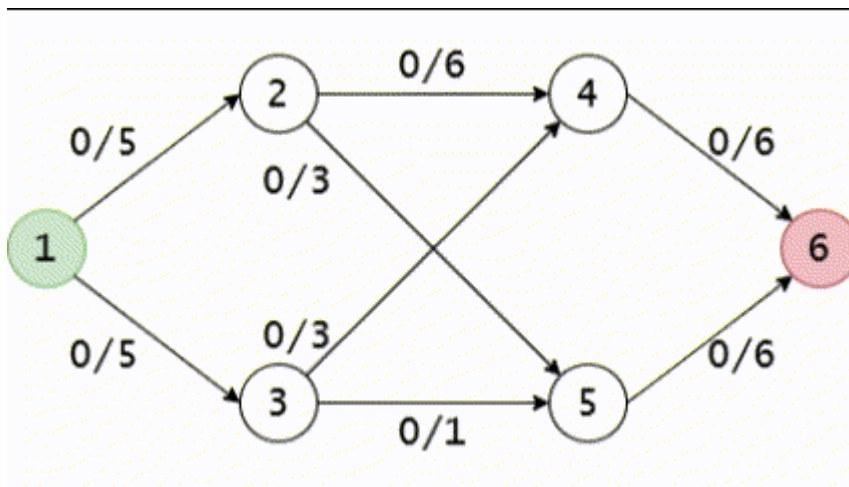
Đầu tiên ta gán giá trị mọi luồng trên tất cả mọi cạnh thành 0.

Ta đi tìm một đường tăng luồng có thể có trên đồ thị. Nhắc lại rằng, đường tăng luồng chỉ chứa các cạnh (kể cả "cạnh" ngược) có $r > 0$, hay $c - f > 0$.

Trên đường này, với mỗi cạnh (u, v) , ta tăng giá trị của luồng trên cạnh này (tức $f(u, v)$) lên Δ đơn vị, với Δ là giá trị $r(u, v)$ nhỏ nhất trên đường tăng luồng vừa tìm được. Đồng thời, ta cũng phải giảm $f(v, u)$ đi Δ để luôn có $f(u, v) = -f(v, u)$.

Một cách dễ hiểu hơn thì tại bước này, ta tăng giá trị của luồng trên đường vừa tìm được đến mức tối đa có thể.

Ta lặp đi lặp lại việc tăng luồng cho đến khi nào không thể tìm được đường tăng luồng nữa thì thôi. Khi đó, giá trị của luồng trong cả mạng chính là luồng cực đại mà ta cần tìm.



Hình GIF trên mô tả phương pháp Ford-Fulkerson trên mạng ta vừa lấy ví dụ trong bài viết này. Chú ý rằng có một bước, chúng ta đã phải sử dụng cạnh ngược.

Để tìm đường tăng luồng, ta chỉ phải tìm một đường để đi từ s tới t , qua các cạnh có $r(u, v) = c(u, v) - f(u, v) > 0$. Đây chỉ là một bài toán duyệt đồ thị đơn giản, ta có thể thử áp dụng DFS, BFS, ... để duyệt.

Hai thuật BFS và DFS có độ phức tạp giống nhau, nhưng trên thực tế BFS chạy nhanh hơn DFS khi đi tìm đường tăng luồng. Thuật Edmonds-Karp sử dụng BFS.

Tính đúng đắn

Định lý: Phương pháp Ford-Fulkerson cho kết quả là luồng cực đại.

► Chứng minh

Hệ quả:

- Khả năng thông qua của lát cắt hẹp nhất trên một mạng bằng giá trị của luồng cực đại trên mạng đó. **Lát cắt hẹp nhất** (mincut) là lát cắt có khả năng thông qua nhỏ nhất trong số mọi lát cắt thuộc mạng.
- Nếu mọi giá trị c trên luồng đều là số nguyên thì giá trị luồng cực đại cũng là số nguyên.

Cài đặt

```

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 1001;

int n, m, s, t;
vector <int> adj[MAXN];    //đồ thị lưu kiểu danh sách kề
int c[MAXN][MAXN], f[MAXN][MAXN], trace[MAXN], maxFlow;

//BFS để tìm đường tăng luồng
void bfs()
{
    fill(trace, trace + n + 1, 0);
    trace[s] = -1;

    queue <int> bfsQueue;
    bfsQueue.push(s);

    while (!bfsQueue.empty())
    {
        int u = bfsQueue.front();
        bfsQueue.pop();
        for (auto v : adj[u])
        {
            //Không dẫm lại đường cũ theo đúng luật BFS
            if (trace[v]) continue;

            //Không đi qua cạnh có  $r(u, v) = c(u, v) - f(u, v) = 0$ 
            if (f[u][v] - c[u][v] == 0) continue;

            //Các công việc còn lại của BFS
            trace[v] = u;
            bfsQueue.push(v);
        }
    }
}

//Hàm tăng luồng
void incFlow()
{
    //Đi ngược theo đường tăng luồng để tìm giá trị  $\delta = c - f$  tốt nhất
    int delta = INT_MAX;
    int v = t;
    while (v != s)
    {
        int u = trace[v];
        delta = min(delta, c[u][v] - f[u][v]);
        v = u;
    }
}

```

```


51
52
53     maxFlow += delta;
54
55     //Đi ngược theo đường tăng luồng một lần nữa để cập nhật giá trị f
56     v = t;
57     while (v != s)
58     {
59         int u = trace[v];
60         f[u][v] += delta;
61         f[v][u] -= delta;
62         v = u;
63     }
64 }
65
66 int main()
67 {
68     cin >> n >> m >> s >> t;
69     for (int u, v, i = 1; i <= m; i++)
70     {
71         cin >> u >> v;
72         cin >> c[u][v];
73         adj[u].push_back(v);
74         adj[v].push_back(u); //lưu thêm cạnh ngược để có thể chạy qua nó khi
75     }
76
77     maxFlow = 0;
78
79     //Tăng luồng đến khi không tăng được nữa
80     do
81     {
82         bfs();
83         if (trace[t]) incFlow();
84     } while (trace[t]);
85
86     cout << maxFlow;
87 }

```

Độ phức tạp

Trong bài toán chúng ta xét, tất cả các khả năng thông qua của các cạnh đều là số nguyên. Do đó, mỗi bước tăng luồng đều làm tăng giá trị của luồng lên ít nhất 1 đơn vị. Khi sử dụng thuật BFS hoặc DFS để tìm đường tăng luồng, độ phức tạp sẽ vào cỡ $O(E)$. Do đó, độ phức tạp của phương pháp Ford-Fulkerson sẽ là $O(Ef)$, với f là giá trị của luồng cực đại trên mạng. Đây không phải là một độ phức tạp với thời gian đa thức trên kích thước đồ thị.

Với thuật toán Edmonds-Karp, khi sử dụng BFS, sau $O(EV)$ lần tìm đường tăng luồng, chúng ta sẽ tìm được kết quả. Độ phức tạp của thuật toán này là $O(E^2V)$.

Bạn có thể tham khảo chứng minh độ phức tạp này tại [đây](#) .

Khi thực hiện giải thuật Edmonds-Karp, các đánh giá ban đầu về độ phức tạp có thể sai lệch nhiều so với thực tế. Mặc dù độ phức tạp của thuật toán là tương đối lớn trong trường hợp tệ nhất, nó vẫn hoạt động hiệu quả trong hầu hết các trường hợp.

Thuật toán Dinic

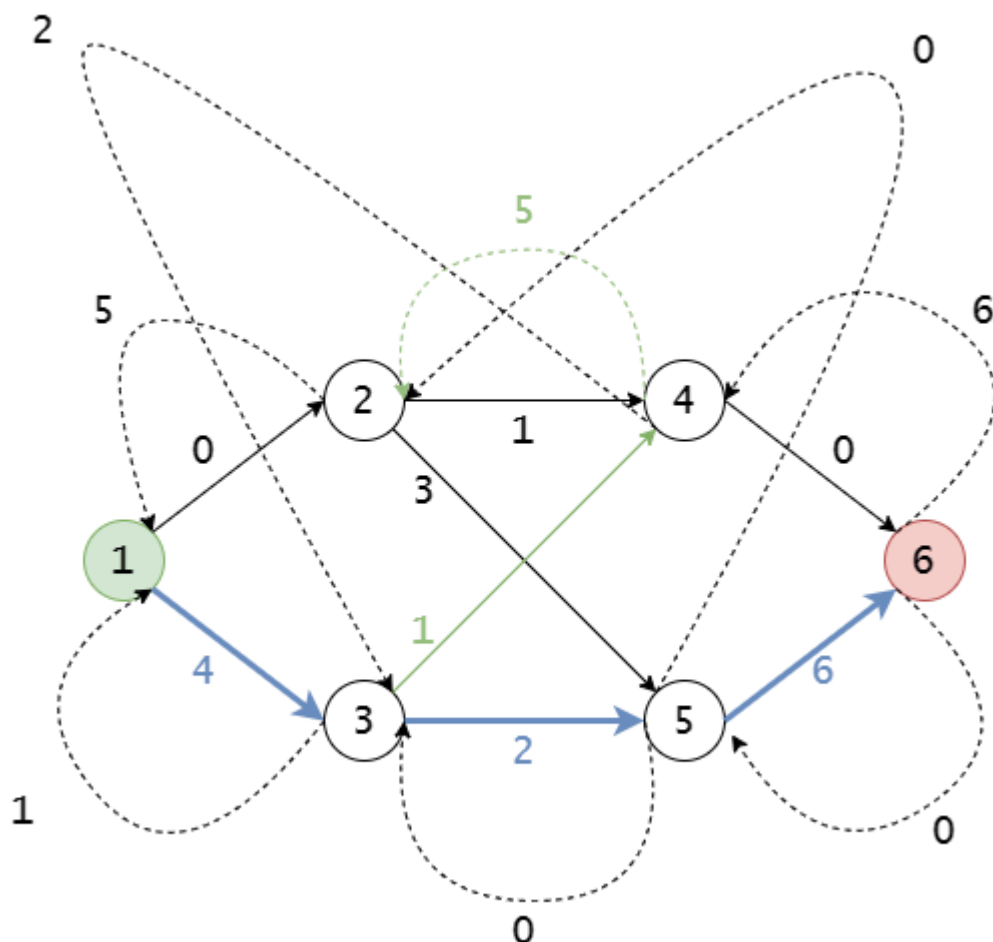
Như đã nói ở trên, tuy đánh giá về độ phức tạp của thuật Edmonds-Karp không hề đẹp, nó vẫn chạy đủ nhanh trong thực tế. Tất nhiên, vẫn có những trường hợp thuật này chạy chưa được ổn lắm, điển hình là khi mạng có rất nhiều cạnh, ví dụ có dạng của đồ thị đầy đủ với $\frac{V(V-1)}{2}$ cạnh thì độ phức tạp của thuật toán sẽ là $O(V^5)$, rất khủng khiếp. Thuật toán Dinic sẽ làm giảm độ phức tạp của thuật đi một chút.

Thuật toán này được Yefim A. Dinits (nhiều tài liệu để tên là E. A. Dinic) đề xuất năm 1970. Nó được chứng minh là có độ phức tạp $O(EV^2)$, tốt hơn thuật toán Edmonds-Karp.

Thuật toán Dinic sử dụng nhiều ý tưởng của phương pháp Ford-Fulkerson để tìm đường tăng luồng. Để đọc và hiểu được phần dưới đây, bạn nên có kiến thức về phương pháp này trước.

Các khái niệm

- ▶ Thuật toán Dinic vẫn sử dụng khái niệm **đồ thị thặng dư** giống như trong phương pháp Ford-Fulkerson. Nhắc lại, đồ thị thặng dư là đồ thị mà ứng với mỗi cạnh (u, v) sẽ có hai cạnh, một cạnh (u, v) có trọng số $r(u, v) = c(u, v) - f(u, v)$ và một cạnh (v, u) có trọng số $f(u, v)$.
- ▶ Một **luồng cản** (blocked flow) là một tập các cạnh trên đồ thị có dạng giống như luồng trên mạng sao cho mọi đường đi từ s đến t đều chứa ít nhất một cạnh thuộc tập này.
- ▶ Gọi $d(u)$ là **mức/cấp** (level) của đỉnh u - đường đi ngắn nhất (tính bằng số cạnh) để đi từ s đến u trên đồ thị thặng dư. Định nghĩa **đồ thị phân cấp** (layered network) của đồ thị ban đầu là đồ thị chỉ chứa các cạnh (u, v) **có trọng số dương** thoả mãn $d(v) = d(u) + 1$, tức là các cạnh tham gia tạo thành đường đi ngắn nhất đến tất cả các đỉnh.



Đồ thị phân cấp (tất cả các đường có màu) và luồng cản (xanh lam) của đồ thị thặng dư

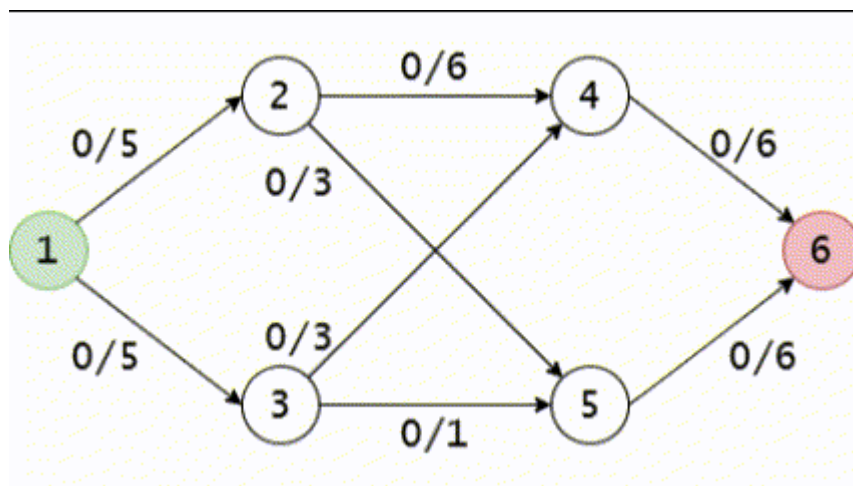
Thuật toán

Ta dựng đồ thị phân cấp của đồ thị thặng dư. Trên đồ thị này, ta tìm một luồng cản rồi tăng luồng ở tất cả các cạnh trên luồng cản này càng nhiều càng tốt. Nói cách khác, đây là phương pháp Ford-Fulkerson với các đường tăng luồng là các đường cản trong luồng cản. Lặp lại quá trình trên cho tới khi ta không thể tìm được đường đi từ s tới t trên đồ thị phân cấp nữa, hay $d(t)$ không xác định.

Để tìm luồng cản, ta sử dụng DFS để tìm từng đường cản một. Mỗi đường cản là một đường đi có trọng số dương từ s tới t trên đồ thị phân cấp. Đây là lý do thuật Dinic được gọi là "dùng cả BFS và DFS để tìm luồng".

Để tối ưu việc cài đặt, ta có thể:

- Không dựng đồ thị thặng dư và đồ thị phân cấp. Cũng như thuật toán Edmonds-Karp, ta hoàn toàn có thể sử dụng thêm các "cạnh" ngược với giá trị luồng âm để biểu diễn các cạnh ngược trong đồ thị thặng dư. Việc sử dụng đồ thị phân cấp thì chỉ là đánh các nhãn $d(u)$ cho các đỉnh u của đồ thị, rồi kiểm tra $c(u, v) - f(u, v) > 0$ và $d(u) + 1 = d(v)$ để biết cạnh (u, v) (kể cả ngược) có thuộc đồ thị phân cấp không.
- Tại mỗi đỉnh, chỉ DFS từ cạnh cuối cùng được xét trong lần tìm đường cản trước đó với cùng một bộ d (hay cùng một đồ thị phân cấp) (xem code để hiểu phần này hơn). Việc tiếp tục sử dụng một cạnh nào đó của các đường trước đó để tăng luồng là vô nghĩa, vì trong những lần tìm trước đó, ta đã khẳng định là chúng không thể tạo ra đường cản mới rồi. Khi không tìm được bất kỳ đường cản nào nữa, luồng cản hiện tại coi như đã xong. Ta tăng luồng và đánh lại d cho các đỉnh.



Hình GIF trên mô tả thuật toán Dinic. Tất cả các cạnh có màu đều là các cạnh nằm trên đồ thị phân cấp. Các cạnh màu xanh và đỏ là các cạnh nằm trên luồng cần tìm được sau mỗi bước.

Tính đúng đắn

Định lý: Thuật toán Dinic cho kết quả là luồng cực đại

► Chứng minh

Cài đặt

Trong bước DFS, để lập trình đơn giản hơn một chút, ta sẽ kết hợp DFS và tăng luồng. Mỗi lần đi tìm đường cần, ta có thể kết hợp lưu lại giá trị Δ nhỏ nhất trên đường này luôn, và khi đường này đến được t , ta thực hiện tăng luồng trên những cạnh đã xét.

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const int MAXN = 1001;
6  const int INF = 1e9 + 7;
7
8  int n, m, s, t;
9  vector <int> adj[MAXN];
10 int c[MAXN][MAXN], f[MAXN][MAXN], d[MAXN], maxFlow;
11
12 //chỉ số của cạnh cuối cùng được xét trong danh sách kề
13 int curVertexId[MAXN];
14
15 //BFS để tìm mức (d) của mỗi đỉnh
16 void bfs()
17 {
18     fill(d, d + n + 1, INF);
19     d[s] = 0;
20
21     queue <int> bfsQueue;
22     bfsQueue.push(s);

```

```

23
24     while (!bfsQueue.empty())
25     {
26         int u = bfsQueue.front();
27         bfsQueue.pop();
28         for (auto v : adj[u])
29         {
30             if (d[v] != INF) continue;
31             if (f[u][v] - c[u][v] == 0) continue; //chỉ xét cạnh dương
32             d[v] = d[u] + 1;
33             bfsQueue.push(v);
34         }
35     }
36 }
37
38 //DFS tìm luồng cản.
39 //curDelta: giá trị delta tốt nhất hiện có trên đường từ s tới u
40 //Hàm trả về giá trị delta tốt nhất sau khi tìm xong đường cản.
41 int dfs(int u, int curDelta)
42 {
43     if (curDelta == 0) return 0;
44     if (u == t) return curDelta;
45
46     //Chỉ xét từ cạnh cuối cùng
47     for (; curVertexId[u] < adj[u].size(); curVertexId[u]++)
48     {
49         int v = adj[u][curVertexId[u]];
50
51         //Chỉ xét cạnh thuộc đồ thị phân cấp
52         if (d[v] != d[u] + 1) continue;
53         if (f[u][v] == c[u][v]) continue;
54
55         //Thực hiện tăng luồng
56         int delta = dfs(v, min(c[u][v] - f[u][v], curDelta));
57         if (delta == 0) continue;
58         f[u][v] += delta;
59         f[v][u] -= delta;
60         return delta;
61     }
62     return 0;
63 }
64
65 int32_t main()
66 {
67     cin >> n >> m >> s >> t;
68     for (int u, v, i = 1; i <= m; i++)
69     {
70         cin >> u >> v;
71         cin >> c[u][v];
72         adj[u].push_back(v);
73         adj[v].push_back(u);
74

```

```

75     }
76     maxFlow = 0;
77
78     while (true)
79     {
80         bfs();
81         if (d[t] == INF) break;
82         for (int i = 1; i <= n; i++) curVertexId[i] = 0;
83         while (int delta = dfs(s, INF))
84             maxFlow += delta;
85     }
86
87     cout << maxFlow;
    }


```

Độ phức tạp

Định lý: Thuật toán Dinic có độ phức tạp là $O(EV^2)$

► Chứng minh

Bài toán ví dụ

Đề bài VNOI: [FLOW1](#) 

Tóm tắt đề bài: Có $2n$ học sinh đến từ hai trường SP, TH và m bài toán. Mỗi học sinh có thể giải tốt một số bài toán cho trước. Cần chọn n bài toán sao cho:

- Với mỗi bài toán, mỗi trường có đúng một học sinh giải được bài đó
- Không có học sinh nào làm hai bài toán
- Không có hai học sinh nào cùng trường làm cùng một bài toán.

► Phân tích

Một số chú ý

- Khi giải các bài toán về luồng hoặc lát cắt, loại bài liên quan đến **mạng đơn vị** (mạng có các khả năng thông qua trên các cạnh là 1) khá phổ biến. Trên những mạng này, khi tìm thành công luồng cực đại, luồng các cạnh sẽ chỉ ở một trong hai trạng thái: đầy ($f = 1$) hoặc rỗng ($f = 0$); còn luồng cực đại sẽ có dạng một số đường đi không giao nhau.
- Khi mạng đơn vị có dạng đồ thị hai phía (đồ thị có thể chia các đỉnh thành 2 tập hợp sao cho không có hai đỉnh nào cùng một tập hợp có cạnh nối đến nhau) cùng với đỉnh nguồn và đỉnh đích, bài toán trở thành dạng **cặp ghép cực đại trên đồ thị hai phía**. Bạn đọc nên tìm hiểu thêm về bài toán này để đưa ra những giải thuật linh hoạt hơn trong từng trường hợp cụ thể.
- Như đã nói ở trên, cách đánh giá độ phức tạp của các thuật toán trên có thể sai lệch tương đối so với thực tế. Vì vậy, khi làm những bài luồng, đôi lúc bạn có thể tính ra một độ phức tạp rất lớn, nhưng thuật toán lại chạy tốt. Ngay như bài NKFLOW ở trên, chúng ta vẫn AC được với độ phức tạp $O(E^2V)$.

- ▶ Tuy chênh lệch về độ phức tạp giữa thuật Edmonds-Karp và Dinic là có thể thấy ngay, nhưng khi chạy, thuật Dinic thường cũng không cải thiện được quá nhiều. Các tác giả của *Competitive Programing 3* cũng thừa nhận họ "chưa từng gặp một trường hợp đồ thị nào cho kết quả AC bằng Dinic mà chạy TLE bằng thuật Edmonds-Karp". Tuy nhiên, nếu muốn có sự tối ưu, hãy sử dụng thuật Dinic. Còn nếu bạn muốn một thuật dễ cài đặt, dễ nhớ và dễ hiểu hơn, có thể sử dụng Edmonds-Karp.
- ▶ Edmonds-Karp và Dinic là hai thuật phổ biến nhưng không phải duy nhất để tìm luồng cực đại. Bạn có thể tìm hiểu thêm về thuật [push-relabel](#) (1985) và [MPM](#) (1978) tại CP Algorithms. Gần đây, đã có những thuật tinh vi hơn tìm được luồng với độ phức tạp $O(EV)$, như thuật của King, Rao, and Tarjan (1994), của Orlin (2012). Thậm chí, năm 2022, đã có thêm một thuật toán giải bài toán gần với bài này là min-cost flow với thời gian gần tuyến tính $O(E^{1+o(1)})$.

Luyện tập

- ▶ [BAOVE](#)
- ▶ [STNODE](#)
- ▶ [KWAY](#)
- ▶ [JOBSET](#)
- ▶ [Delivery Bears](#)
- ▶ [Soldier and Traveling](#)
- ▶ [Array and Operations](#)
- ▶ [Red-Blue Graph](#)
- ▶ [Download Speed](#)
- ▶ [Police Chase](#)
- ▶ [School Dance](#)

Ngoài ra, bạn đọc có thể luyện tập bằng các bài tập khác có gắn tag [flows](#) trên VNOJ và các OJ khác.

Tài liệu tham khảo

- ▶ Lê Minh Hoàng (2003), *Giải thuật và lập trình*
- ▶ Steven Halim, Felix Halim (2013), *Competitive Programing 3*
- ▶ CP Algorithms:
 - ▶ [Maximum flow - Ford-Fulkerson and Edmonds-Karp](#)
 - ▶ [Maximum flow - Dinic's algorithm](#)
- ▶ Wikipedia (về lịch sử của các thuật toán)
- ▶ VNOI Wiki: [Luồng cực đại trên mạng - Maxflow network](#) (bài viết cũ)
- ▶ [Phần chứng minh trên brilliant.org](#)
- ▶ Reza Zadeh (2014), [CME 305: Discrete Mathematics and Algorithms - Lecture 3](#)