BUỔI 5. CÂY KHUNG NHỎ NHẤT & LUỒNG CỰC ĐẠI TRÊN MẠNG

Mục đích

- *Thực hành cài đặt các thuật toán tìm cây khung (vô hướng) nhỏ nhất, tìm luồng cực đại trên mạng*
- *Củng cố lý thuyết, rèn luyện kỹ năng lập trình*

Nội dung

- Thuật toán Kruskal
- Thuật toán Prim
- Thuật toán Ford-Fulkerson (thực ra là thuật toán Edmonds-Karp)

Yêu cầu

- Biểu diễn đồ thị
- Các phép toán cơ bản trên đồ thị
- Duyệt đồ thị (theo chiều rộng, chiều sâu)
- Biết sử dụng cấu trúc dữ liệu ngăn xếp + hàng đợi

5.1 Cây khung có trọng số nhỏ nhất

Cây (Tree)

Cây là đồ thị vô hướng liên thông và không chứa chu trình.

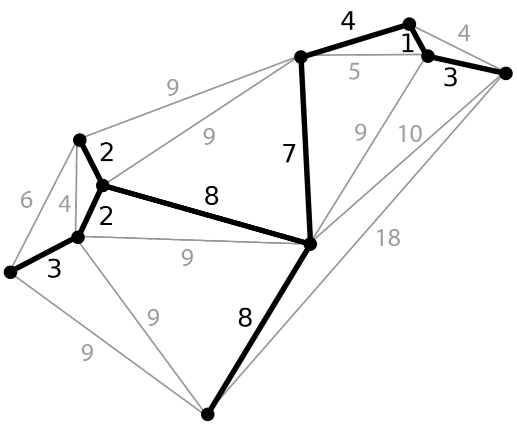
Cây khung (Spanning Tree)

Cây khung T của một đồ thị (liên thông) G là cây bao gồm tất cả các đỉnh của đồ thị G và một số cung của đồ thị G.

Cây khung nhỏ nhất (Minimum Spanning Tree – MST)

Cho đồ thị vô hướng liên thông `G = <V, E>`. Mỗi cung của `G` được gán một trọng số (có thể âm). Bài toán tìm *cây khung có nhỏ nhất* (còn có tên gọi là Cây phủ tối tiểu, cây bao trùm tối tiểu, cây khung tối tiểu. Tên tiếng anh là: Minimum Spanning Tree) là bài toán tìm cây khung của đồ thị `G` sao cho *tổng trọng số các cung trên cây nhỏ nhất*.

Ví dụ bên dưới minh hoạ đồ thị vô hướng liên thông và cây khung có trọng số nhỏ nhất (các cung của cây được in đậm) của nó.



5.1.1 Thuật Kruskal tìm cây khung nhỏ nhất

Cho đồ thị vô hướng và liên thông `G`, ta cần tìm cây khung có trọng số nhỏ nhất của đồ thị này. Kết quả trả về là một cây `T` (cũng là một đồ thị).

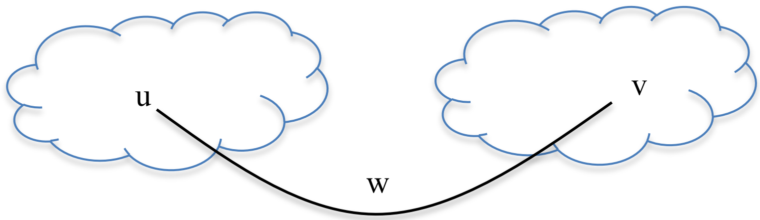
Ý tưởng:

- Sắp xếp các cung của `G` theo thứ tự trọng số tăng dần (thực ra là không giảm vì có trường hợp trọng số của 2 cung bằng nhau)
- Khởi tạo cây `T` gồm các đỉnh của `G` và không chứa cung nào
- `for` (các cung `e = (u, v; w)` theo thứ tự đã sắp xếp)
    - `if` (thêm cung `e` vào `T` mà không tạo nên chu trình) thì thêm `e` vào `T`
    - `else` bỏ qua cung `e`

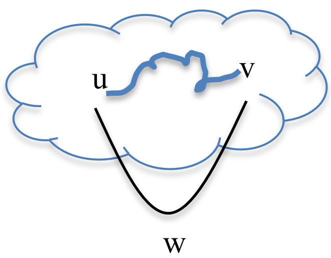
Ta có thể cho thuật toán dừng sớm bằng cách mỗi khi thêm một cung `e` vào `T`, ta tăng số cung của `T` lên. Thuật sẽ dừng khi `T` chứa `n – 1` cung với n là số đỉnh của `G`.

Phần khó nhất của thuật toán này là *kiểm tra việc thêm 1 cung vào cây* `T` *có tạo nên chu trình hay không*. Để kiểm tra khi thêm 1 cung vào một đồ thị có tạo nên chu trình hay không cách đơn giản nhất là thêm cung này vào đồ thị, sau đó áp dụng thuật toán kiểm tra chu trình (trong bài trước) để kiểm tra. Tuy nhiên cách này kém hiệu quả vì độ phức tạp cao. Ta xét một phương pháp khác hiệu quả hơn để kiểm tra việc tạo chu trình như sau:

- Nếu cung `e = (u, v; w)` có `u` và `v` *thuộc về hai bộ phận liên thông khác nhau* thì việc thêm `e` vào `T` sẽ *không tạo nên chu trình* vì chỉ có một đường đi duy nhất từ `u` đến `v` thông qua cung `e`. Khi thêm cung `e = (u, v; w)` (có `u` và `v` ở hai bộ phân liên thông khác nhau) vào cây `T`, Hai bộ phân liên thông của `u` và của `v` sẽ được *gom lại thành 1 bộ phân liên thông mới*.



- Ngược lại, nếu cung `e` có đỉnh `u` và đỉnh `v` *cùng thuộc về một bộ phận liên thông* thì khi thêm `e` vào `T`, sẽ tồn tại ít nhất 2 đường đi khác nhau từ `u` đến `v` (một theo bộ phân liên thông và một đi trực tiếp theo cung `e`) hay *sẽ tồn tại chu trình*.



Từ nhận xét trên, ta cần phải quản lý các bộ phận liên thông của `T` trong quá trình xét từng cung của `G`.

Quản lý các bộ phận liên thông (BPLT) của đồ thị

Mỗi bộ phận liên thông được biểu diễn dưới dạng 1 cấu trúc dữ liệu cây (xem cấu trúc dữ liệu cây trong học phần Cấu trúc dữ liệu). Để đơn giản ta chỉ cần sử dụng phương pháp biểu diễn cây bằng mảng (lưu nút cha của các nút).

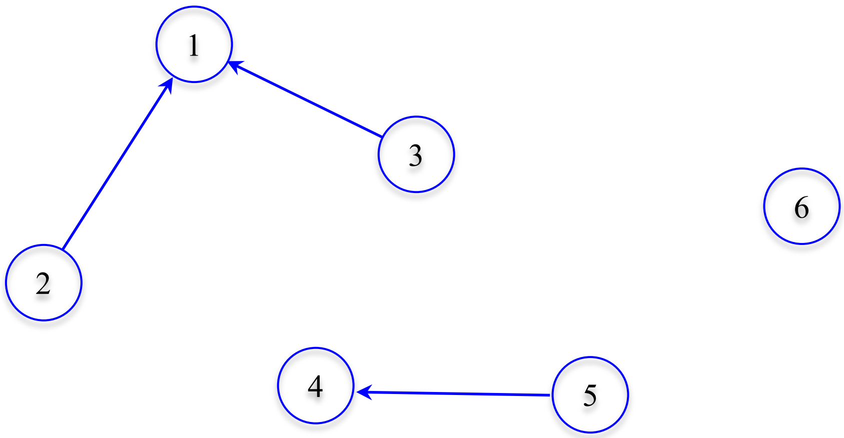
Sử dụng mảng `parent[]` để lưu đỉnh cha của các đỉnh.

- `parent[u]`: đỉnh cha của đỉnh u.
- *Quy ước:* đỉnh ứng với gốc của cây có cha là chính nó (`parent[r] = r`).

Ví dụ: Đồ thị bên dưới có 3 bộ phận liên thông, mảng `parent[]` tương ứng của nó sẽ là:

u	**1**	2	3	**4**	5	**6**
parent[u]	1	1	1	4	4	6

3 đỉnh gốc là 1, 4 và 6.



Như thế mỗi BPLT được lưu trong một cấu trúc dữ liệu cây. Đỉnh (nút) gốc của cây là đại diện của BPLT tương ứng. Để tìm xem đỉnh u thuộc về BPLT nào ta chỉ cần tìm gốc của cây chứa u.

Ta có thể cài đặt hàm tìm gốc của một đỉnh bằng đệ quy:

```
//Tìm gốc của đỉnh u (đệ quy)
int findRoot(int u) {
	if (parent[u] == u)
		return u;
	return findRoot(parent[u]);
}
```

hoặc sử dụng giải thuật lặp:

```
//Tìm gốc của đỉnh u (lặp)
int findRoot(int u) {
	while (parent[u] != u)
		u = parent[u];
	return u;
}
```

Cài đặt thuật toán Kruskal

Tham số và các biến hỗ trợ

- `parent[u]`: đỉnh cha của u.
- `pG`: đồ thị đầu vào
- `pT`: Cây kết quả (có kiểu con trỏ `Graph` như `pG`)
- Trả về tổng trọng số các cung của cây `pT`.

Biểu diễn đồ thị

- Sử dụng phương pháp **danh sách cung**

```
//Cấu trúc dữ liệu của 1 cung
typedef struct {
	int u, v;
	int w;
} Edge;
```

Thuật toán Kruskal

```
//Thuật toán Kruskal tìm cây khung nhỏ nhất
int Kruskal(Graph *pG, Graph *pT) {
	//1. Sắp xếp các cung của G theo thứ tự trọng số tăng dần
	...
	//2.  Khởi tạo pT không chứa cung nào, khởi tạo bộ quản lý các BPLT
	init_graph(pT, pG->n);
	for (int u = 1; u <= pG->n; u++)
		parent[u] = u;     //Mỗi đỉnh u là một bộ phận liên thông
	int sum_w = 0;         //Tổng trọng số các cung của cây

	//3. Duyệt qua các cung của G (đã sắp xếp)
	for (e = 0; e < pG->m; e++) {
		int u = pG->edges[e].u;
		int v = pG->edges[e].v;
		int w = pG->edges[e].w;
		int root_u = findRoot(u);     //Tìm BPLT của u
		int root_v = findRoot(v);     //Tìm BPLT của v
		if (root_u != root_v) {       //u và v ở 2 BPLT khác nhau

			//Thêm cung (u, v; w) vào cây pT
			add_edge(pT, u, v, w);

			//Gộp 2 BPLT root_u và root_v lại
			parent[root_v] = root_u;
			sum_w += w;
		}
	}
	return sum_w;
}
```

Để sử dụng thuật Kruskal, sau khi xây dựng đồ thị `G` ta gọi `Kruskal(&G, &T)`. Cây khung nhỏ nhất sẽ được lưu vào cây `T`. Ví dụ bên dưới sẽ đọc một đồ thị từ tập tin *dt.txt*, tìm cây khung nhỏ nhất bằng thuật toán Kruskal và in ra màn hình: tổng trọng số và các cung của cây kết quả.

```
//Sử dụng thuật toán Kruskal
int main() {
	Graph G, T;
	int n, m, u, v, w, e;

	//Đọc dữ liệu từ tập tin dt.txt.
	freopen("dt.txt", "r", stdin);
	scanf("%d%d", &n, &m);
	init_graph(&G, n);
	for (e = 1; e <= m; e++) {
		scanf("%d%d%d", &u, &v, &w);
		add_edge(&G, u, v, w);
	}

	int sum_w = Kruskal(&G, &T); //Gọi hàm Kruskal
	printf("Tong trong so cua cay T la: %d\n", sum_w);

	for (e = 0; e < T.m; e++)
		printf("(%d, %d): %d\n", T.edges[e].u, T.edges[e].u,

		T.edges[e].w);
	return 0;
}
```

Để hoàn chỉnh thuật toán trên bạn cần cài đặt thêm phần sắp xếp các cung của đồ thị `pG` theo thứ tự tăng dần. Hãy tham khảo các thuật toán sắp xếp trong các học phần đã được học như: Lập trình căn bản, Phân tích thiết kế thuật toán.

Dưới đây là một thuật toán sắp xếp đơn giản (biến thể của sắp xếp chọn) dễ nhớ, dễ cài đặt (nhưng không hiệu quả về mặt thời gian chạy).

```
//Sắp xếp các cung theo thứ tự trọng số tăng dần
for (i = 0; i < m; i++)
      for (j = i + 1; j < m; j++)
            if (trọng số cung j < trọng số cung i) {
                  //Hoán  đổi cung i và cung j cho nhau
            }
```

Thuật toán sắp xếp trên có độ phức tạp thời gian là $O(n^2)$. Để làm giảm độ phức tạp của toàn bộ thuật toán, ta cần phải sử dụng các thuật toán có độ phức tạp thời gian nhỏ hơn như: quick sort, merge sort.

Nâng cao

Thư viện `stdlib.h` có hỗ trợ hàm `qsort()` dùng để sắp xếp các phần tử của một mảng theo thứ tự tuỳ ý. Tuy nhiên, việc sử dụng nó cũng không đơn giản. Ví dụ bên dưới cho phép sắp xếp các phần tử của một mảng số nguyên A theo thứ tự tăng dần.

```
//Thư viện chứa hàm qsort()
#include <stdlib.h>

//Hàm so sánh hai phần tử a và b hỗ trợ việc sắp xếp
int cmpfunc(const void *a, const void *b) {
      return (*(int*)a - *(int*)b); //Phải biết kiểu dữ liệu của a và b
      //Trả về giá trị < 0 nếu muốn a đứng trước b
}

int main () {
      int n = 5;
      int A[] = { 88, 56, 100, 2, 25 };

      //Sắp xếp 5 phần tử đầu tiên của mảng a theo thứ tự tăng dần
      qsort(A, 5, sizeof(int), cmpfunc);
      ...
      return 0;
}
```

Khuôn dạng (prototype) của hàm `qsort()`:

```
//Thư viện chứa hàm qsort()
void qsort(void *base, size_t nitems, size_t size,
                       int (*compar)(const void*, const void*));
```

Các tham số:

- `base`: mảng cần sắp xếp
- `nitems`: số phần tử cần sắp xếp tính từ đầu mảng
- `size`: kích thước (theo byte) của mỗi phần tử trong mảng
- `compar`: hàm so sánh hai phần tử của mảng.

Hãy áp dụng hàm `qsort()` để sắp xếp các cung của đồ thị `pG` trong thuật toán Kruskal.

Bài tập 1. Viết chương trình đọc đồ thị từ bàn phím, áp dụng thuật toán Kruskal tìm cây khung có trọng số nhỏ nhất và in kết quả ra màn hình (như chương trình mẫu).

5.2 Thuật toán Prim tìm cây khung nhỏ nhất

Khác với thuật toán Kruskal dựa trên cung, thuật toán Prim dựa trên đỉnh để tìm cây khung. Ý tưởng tương tự với thuật toán Moore – Dijkstra.

Ý tưởng:

- Gọi S là tập đỉnh đã xét (đã đánh dấu), khởi tạo S rỗng
- Chọn 1 đỉnh s bất kỳ, đưa nó vào tập hợp S
- Lặp (n – 1 lần) làm các công việc sau:
    - Tìm đỉnh u không có trong S và *gần với S nhất*. Gọi p[u] là đỉnh trong S mà u gần với nó nhất.
    - Đưa u vào S
    - Đưa cung tương ứng (p[u], u) vào T

Phân tích ý tưởng

Mấu chốt của giải thuật Prim là việc *tìm đỉnh u không thuộc S mà gần với S nhất*. Thay vì phải tính lại khoảng cách từ 1 đỉnh `u` đến `S` trong mỗi lần lặp, ở mỗi đỉnh ta cần lưu `pi[u]` là *khoảng cách từ u đến S* và `p[u]` là *đỉnh trong S gần với u nhất*. Trong mỗi lần lặp, ta sẽ chọn đỉnh `u` chưa đánh dấu có `pi[u]` nhỏ nhất và cập nhật lại `pi[v]` và `p[v]` của các đỉnh kề `v` (không nằm trong `S`) của `u`.

Tham số và biến hỗ trợ

- `pG`: Đồ thị đầu vào
- `pT`: cây kết quả
- `s`: đỉnh bắt đầu
- `mark[u]`: cho biết đỉnh u đã nằm trong `S` chưa (1: rồi, 0: chưa)
- `pi[u]`: khoảng cách gần nhất từ đỉnh u đến 1 đỉnh nào đó trong `S`
- `p[u]`: Đỉnh trong `S` gần với `u` nhất

Ta không cần lưu trữ S, chỉ cần lưu trữ `mark[u]`. Những đỉnh có `mark[u] = 0` là đỉnh nằm trong S và ngược lại.

Biểu diễn đồ thị

- Để đơn giản, ta sử dụng phương pháp ma trận trọng số để biểu diễn đồ thị. Thực tế thì phương pháp danh sách các đỉnh kề sẽ hiệu quả hơn: tiết kiệm hơn, chạy nhanh hơn.

Thuật toán Prim

1. Khởi tạo
- `pi[u]   = oo` với mọi `u`
- `p[u]    = -1` với mọi `u`
- `mark[u] = 0` với mọi `u`
- `pi[s]   = 0`

2. Lặp `n - 1` lần
- Chọn `u` trong số các đỉnh chưa xét (`mark[u] = 0`) mà có `pi[u]` nhỏ nhất
- Cập nhật `mark[u] = 1`
- `for` (`v` là các đỉnh kề chưa xét của `u`)
    - `if` (`pi[v]` > trọng số cung `(u, v)`)
        - `pi[v] =` trọng số cung `(u, v)`
        - `p[v]  = u`

3. Dựng cây (dựa vào các `p[u]` tìm được ở bước 2)
- `for (u = 1; u <= n; u++)`
    - `if (p[u] != -1)` thêm cung `(p[u], u)` vào `pT`

Cài đặt thuật toán Prim

```
//Các biến hỗ trợ
int pi[MAXN];
int p[MAXN];
int mark[MAXN];

//Hàm Prim tìm cây khung nhỏ nhất của đồ thị pG bắt đầu từ đỉnh s
int Prim (Graph *pG, Graph *pT, int s) {
      int i, u, v, x;

      //1. Khởi tạo
      for (u = 1; u <= pG->n; u++) {
            pi[u] = oo;        //Khởi tạo pi = vô cùng, ví dụ: 999999
            p[u]  = -1;        //p[u] chưa xác định
            mark[u] = 0;       //Chưa đỉnh nào được đánh dấu, S = rỗng
      }
      pi[s] = 0;               //Chỉ có p[s] = 0

      //2. Lặp n - 1 lần
      for (i = 1; i < pG->n; i++) {

            //2a. Tìm u gần với S nhất (tìm u có pi[u] nhỏ nhất)
            int min_dist = oo;
            for (x = 1; x <= pG->n; x++)
                  if (mark[x] == 0 && pi[x] < min_dist) {
                        min_dist = pi[x];
                        u = x;
                  }

            //2b. Đánh dấu u
            mark[u] = 1;


            //3c. Cập nhật lại pi và p của các đỉnh kề v của u
            for (v = 1; v <= pG->n; v++)
                  if (pG->A[u][v] != NO_EDGE)
                        if (mark[v] == 0 && pi[v] > pG->W[u][v]) {
                              pi[v] = pG->W[u][v];
                              p[v]  = u;
                        }
      }

      //3. Dựng cây dựa vào p[u]: thêm các cung (p[u], u) vào cây T
      init_graph(pT, pG->n);  //khởi tạo cây T rỗng
      int sum_w = 0;           //Tổng trọng số của cây T

      for (u = 1; u <= pG->n; u++) {       //Cây n đỉnh có n – 1 cung
            if (p[u] != -1) {              //Đỉnh s có p[s] = -1
                  int w = pG->W[p[u]][u];
                  add_edge(pT, p[u], u, w);
                  sum_w += w;
            }
      }

      return sum_w;
}
```

Ta cũng có thể lồng ghép bước 3 (xây dựng cây) vào bước 2 (lặp).

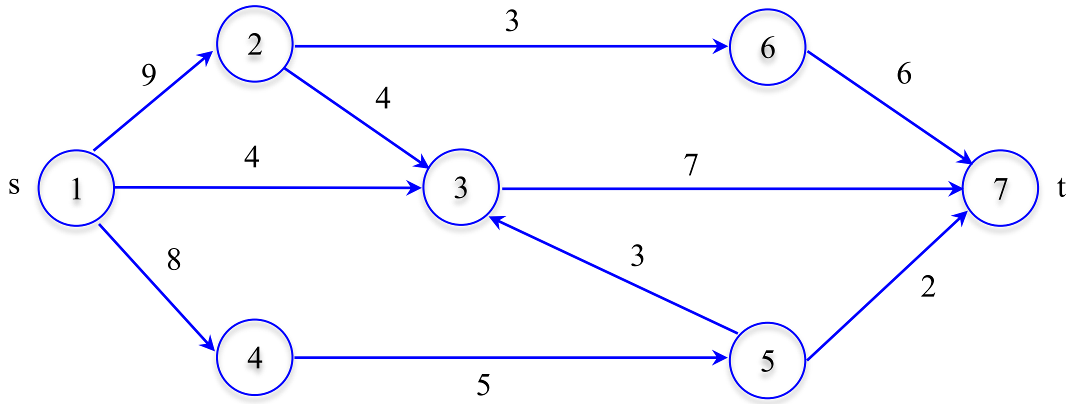
5.3 Luồng cực đại trong mạng

Cho mạng được biểu diễn bằng đồ thị có hướng `G = <V, E>`. Mỗi cung `e = (u, v)` được gán một khả năng thông qua lớn nhất là `c(u,v)`.

Quy ước:

- Mạng chỉ có 1 đỉnh phát `s` và 1 đỉnh thu `t`
- Mạng không chứa đồng thời 2 cung dạng `(u, v)` và `(v, u)`. Nếu có, ta có thể xen thêm 1 đỉnh vào giữa một trong hai cung này.

Ví dụ: mạng có 7 đỉnh với đỉnh phát `s = 1` và đỉnh thu `t = 7`. Khả năng thông qua của các cung được cho trên các cung tương ứng.



5.4 Thuật toán đánh dấu Ford – Fulkerson

Ý tưởng:

- Khởi tạo luồng hợp lệ f = 0: luồng đi qua tất các cung đều bằng 0.
- Lặp:
    - Tìm đường tăng luồng (bằng cách đánh dấu các đỉnh)
    - Nếu tìm thấy => tăng luồng
    - Nếu không còn đường tăng luồng => dừng

Sau khi kết thúc, các đỉnh được đánh dấu thuộc về `S`, các đỉnh chưa được đánh dấu thuộc về `T`, ta thu được lát cắt hẹp nhất `(S, T)`.

Cài đặt:

- Biểu diễn đồ thị: sử dụng phương pháp ma trận trọng số.
- Để có thể lưu được luồng trên các cung ta sử dụng thêm ma trận F.

```
//Các biến hỗ trợ
#define MAXN 100
#define NO_EDGE 0
#define oo 999999

typedef struct {
	int C[MAXN][MAXN]; //khả năng thông qua của cung
	int F[MAXN][MAXN]; //Luồng qua cung
	int n;
} Graph;
```

Các biến/cấu trúc dữ liệu bổ sung

- Cấu trúc dữ liệu `Label`: lưu nhãn của một đỉnh, nhãn của các đỉnh: `labels[]`. Trường 'dir' cho biết nhãn là +, - hay chưa có nhãn (dir = 0).

```
//Các biến hỗ trợ
typedef struct {
	int dir;	// +1: +, -1: -, 0: chưa có nhãn
	int p;		//đỉnh trước
	int sigma;	//lượng tăng luồng khi qua đỉnh này
} Label;

Label labels[MAXN]; //nhãn các đỉnh
```

- Cấu trúc dữ liệu hàng đợi `Queue` để lưu các đỉnh đã được đánh dấu nhưng chưa xét (xem các bài thực hành trước)

Phát hoạ giải thuật

```
//Các biến hỗ trợ
int FordFullkerson(Graph *pG, int s, int t) {
	//I. Khởi tạo luồng = 0, gán F[u][v] = 0 với mọi u, v.
	init_flow(pG);
	int max_flow = 0;
	//II. Lặp
	Queue Q;
	do {
		//Bước 1 - xoá nhãn các đỉnh và gán nhãn cho s
		//Xoá tất cả các nhãn, gán nhãn s: (+, s, oo)
		//Khởi tạo Q rỗng, Đưa s vào Q
		//Bước 2 - lặp gán nhãn cho các đỉnh để tìm đường tăng luồng
		while (Q is chưa rỗng) {
			//Lấy 1 đỉnh trong Q ra, gọi nó là u
			//Xét gán nhãn cho các đỉnh kề với u
			//Xét gán nhãn cho các đỉnh đi đến u
			//Nếu t được gán nhãn =>
			//          tìm được đường tăng luồng, thoát vòng lặp.
		}
		if (tìm được đường tăng luồng) {
			//Bước 3 - Tăng luồng
			int sigma = labels[t].sigma;
			//3.1 Cập nhật các luồng trên cung
			//3.2 Tăng giá trị luồng
			max_flow += sigma;
		} else
			break; //thoát vòng lặp
	} while (1);

	return max_flow;
}
```

Các công việc cần thực hiện

1. Viết hàm `init_flow(Graph *pG)`: gán luồng `F[u][v] = 0`.

```
//Các biến hỗ trợ
void init_flow(Graph *pG) {
      //Gán pG->F[u][v] = 0 với u, v = 1..n.
      ...
}
```

2. Cài đặt cấu trúc dữ liệu `Queue` với các phép toán:
   - `make_null_queue(Queue *pQ)`: tạo hàng đợi rỗng
   - `enqueue(Queue *pQ, int x)`: thêm phần tử x vào cuối hàng đợi.
   - `front(Queue *pQ)`: trả về phần tử đầu hàng đợi.
   - `dequeue(Queue *pQ)`: xoá phần tử đầu hàng đợi.
   - `empty_queue(Queue *pQ)`: kiểm tra hàng đợi rỗng hay không.
3. Bước 1 – xoá nhãn các đỉnh và gán nhãn cho `s`

```
...
      do {
            //Bước 1 - Xóa nhãn và gán nhãn đỉnh s
            //1.1 Xoá tất cả các nhãn cũ
            for (u = 1; u <= G->n; u++)
                  labels[u].dir = 0;
            //1.2 Gán nhãn s: (+, s, oo)
            labels[s].dir = +1;
            labels[s].p   = s;
            labels[s].sigma = oo;
            //1.3 Khởi tạo Q rỗng và đưa s vào Q
            make_null_queue(&Q);
            enqueue(&Q, s);
            ...

      } while (1);
}
```

4. Bước 2 – lặp gán nhãn để tìm đường tăng luồng

Sử dụng kỹ thuật duyệt đồ thị để gán nhãn cho các đỉnh. Ta có thể sử dụng hàng đợi (queue) hoặc ngăn xếp (stack) hoặc hàng đợi ưu tiên (priority queue) để đánh dấu các đỉnh. Trong phần cài đặt này ta *sử dụng hàng đợi để lưu các đỉnh đã được đánh dấu nhưng chưa xét*. Sinh viên có thể cải tiến phần này để cài đặt bước đánh dấu bằng Stack hoặc hàng đợi ưu tiên theo sigma(u) để ưu tiên tìm đường tăng luồng lớn nhất có thể.

```
...
      do {

            //Bước 2 - Lặp gán nhãn cho các đỉnh tìm đường tăng luồng
            int found = 0;
            while (!empty_queue(&Q)) {
                  //2.1 Lấy 1 đỉnh trong Q ra, gọi nó là u
                  int u = front(&Q); dequeue(&Q);
                  //2.2 Xét gán nhãn cho các đỉnh kề với u: cung thuận
                  for (int v = 1; v <= G->n; v++) {
                        if (G->C[u][v] != NO_EDGE &&
                                    labels[v].dir == 0 &&
                                          G->F[u][v] < G->C[u][v]) {
                              labels[v].dir = +1; //cung thuận
                              labels[v].p    = u;
                              labels[v].sigma = min(labels[u].sigma,
                                                G->C[u][v] - G->F[u][v]);
                              enqueue(&Q, v);
                        }
                  }

                  //2.3 Xét gán nhãn cho các đỉnh đi đến u: cung nghịch
                  for (int x = 1; x <= G->n; x++) {
                        if (G->C[x][u] != NO_EDGE &&
                                    labels[x].dir == 0 &&
                                          G->F[x][u] > 0) {
                              labels[x].dir = -1; //cung nghịch
                              labels[x].p    = u;
                              labels[x].sigma = min(labels[u].sigma,
                                                            G->F[x][u]);
                              enqueue(&Q, x);
                        }
                  }

                  //2.4 Nếu t được gán nhãn => tìm được đường tăng luồng
                  if (labels[t].dir != 0) {
                        found = 1;
                        break;
                  }
            }
            ...
      } while (1);
}
```

5. Bước 3 – tăng luồng

Sau khi thoát khỏi vòng `while(!empty_queue(&Q))`, có hai trường hợp xảy ra:

- `found = 0`: không gán nhãn được `t`. Điều này đồng nghĩa với việc không thể tìm được đường tằng luồng => thuật toán kết thúc.
- `found = 1`: đỉnh `t` được gán nhãn, dựa vào nhãn của `t` ta lần ngược theo trường `p` của các nhãn để tăng/giảm luồng.

```
...
	int max_flow = 0;
	do {
		...
		if (found == 1) {
			//Bước 3 - Tăng luồng
			int sigma = labels[t].sigma; //lượng luồng tăng thêm

			//3.1 cập nhật luồng của các cung trên đường tăng luồng
			int u = t;
			while (u != s) {
				int p = labels[u].p;
				if (labels[u].dir > 0)  //tăng luồng
					G->F[p][u] += sigma;
				else                    //giảm luồng
					G->F[u][p] -= sigma;
				u = p;
			}

			//3.2 Tăng giá trị luồng trên mạng
			max_flow += sigma;
		} else
			break; //thoát vòng lặp
	} while (1);

	return max_low; //trả về luồng cực đại trên mạng
}
```

6. Hàm `main()`

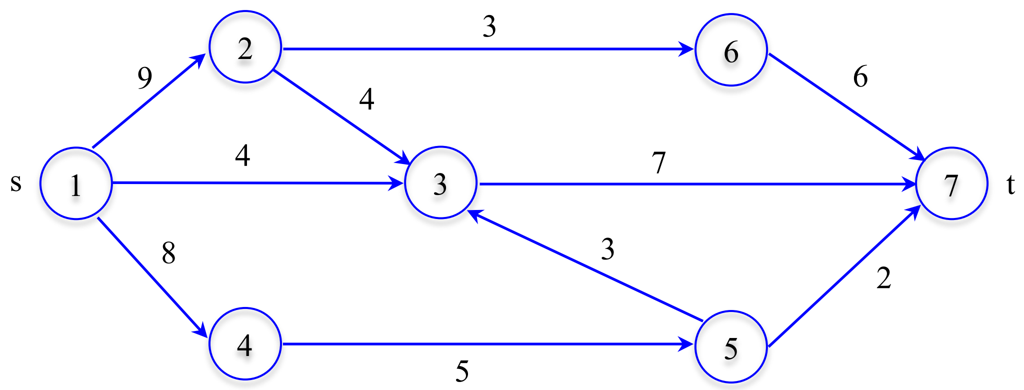
Trong hàm `main()`, ta đọc dữ liệu từ tập tin và gọi hàm `FordFulkerson(&G, 1, n)` với quy ước: đỉnh phát `1` và đỉnh thu `n`. Sinh viên có thể sửa chương trình để cho phép chọn đỉnh phát và đỉnh thu bất kỳ.

```
//Đọc dữ liệu từ tập tin và gọi hàm FordFulkerson
int main() {
	Graph G;
	int n, m, u, v, e, c;
	freopen ("Ten file", "r", stdin);
	scanf("%d%d", &n, &m);
	init_graph(&G, n);


	//Đọc trực tiếp các cung vào ma trận trọng số không cần hàm add_edge()
	for (e = 0; e < m; e++) {
		scanf("%d%d%d", &u, &v, &c);
		G.C[u][v] = c;
	}
	int max_flow = FordFulkerson(&G, 1, n);
	printf("Max flow: %d\n", max_flow);

	return 0;
}
```

Tập tin dữ liệu đầu vào giống như các đồ thị có trọng số:

```
n m
u1 v1 c1
u2 v2 c2
…
um vm cm
```

Bài tập 1. Tích hợp các phần trên thành chương trình hoàn chỉnh cho phép đọc một mạng từ tập tin và in ra màn hình luồng cực đại của mạng. Kiểm thử với mạng sau đây:



Một phần của tập tin dữ liệu:

```
7 10
1 2 9
1 3 4
1 4 8
...
```

Bài tập 2. Cải tiến chương trình để in ra lát cắt hẹp nhất của mạng theo dạng:

$$(x_1, x_2, \ldots / y_1, y_2, \ldots)$$

với $x_1, x_2, \ldots$ là các đỉnh thuộc tập `S` và $y_1, y_2, \ldots$ là các đỉnh thuộc tập `T`.

Gợi ý: sau khi vòng lặp `do … while (1)` kết thúc, sẽ có một số đỉnh được gán nhãn và một số đỉnh không được gán nhãn. Nhãn các đỉnh được lưu trong mảng `labels[]`. Trong hàm `main()` ta có thể in các đỉnh đã được gán nhãn và các đỉnh chưa gán nhãn.

```
//Hàm main() cho phép in lát cắt hẹp nhất (S, T)
int main() {
      ...
      int max_flow = FordFulkerson(&G, 1, n);
      printf("Max flow: %d\n", max_flow);
      ...
      //In lát cắt (S, T)

      //In S: các đỉnh đã có nhãn
      for (u = 1; u <= n; u++)
            if (labels[u].dir != 0) //in u ra màn hình

      //In T: các đỉnh chưa có nhãn
      for (u = 1; u <= n; u++)
            if (labels[u].dir == 0) //in u ra màn hình

      return 0;
}
```

Bài tập 3. Cài đặt lại bài tập 1 bằng cách sử dụng ngăn xếp (stack) thay vì hàng đợi.

Bài tập 4. Cải tiến bài tập 1 bằng cách sử dụng chiến lược đánh dấu như sau: mỗi lần lấy 1 đỉnh từ trong Q, thay vì chọn theo chiến lược vào trước ra trước (FIFO – queue) hoặc vào trước ra sau (FILO – stack) ta sẽ chọn đỉnh có nhãn (+/-, p, sigma) với **sigma lớn nhất** ra xét trước. Với chiến lược, mỗi lần tìm đường tăng luồng ta luôn có đường tăng luồng lớn nhất.