CT175 Lý thuyết đồ thị - Ghi chú môn học

Lý thuyết đồ thị (Trường Đại học Cần Thơ)

Lý thuyết đồ thị

CT175

Thầy Khang - pnkhang@ctu.edu.vn

Đánh giá học phần

Giữa kỳ: 30% (thực hành: 10% + lý thuyết: 20%)
Cuối kỳ: 70% (thực hành: 30% + lý thuyết: 40%)
Chuyên cần +1đ
https://csacademy.com/app/graph_editor/

Định nghĩa và phân loại

Định nghĩa

Đồ thị (Graph) G là một bộ đôi <V, E> trong đó:
- V: Tập các đỉnh (vertex set)
- E: Tập các cung (edge set), mỗi cung nối 2 đỉnh trong V

Cho cung e nối 2 đỉnh x và y
- e = (x,y)
- x, y đgl đầu mút (endpoints) của cung e
- x và y đgl **kề** nhau (adjacent) hoặc lân cận của nhau (neighbours)
- e đgl **liên thuộc** (incident) với x và y

Khuyên (loop): cung có 2 đầu mút trùng nhau
Đa cung (multi edges): các cung có cùng chung đầu mút
Đơn đồ thị vô hướng (simple graph):
- Cung không có hướng: (x,y) = (y,x)
- Không chứa khuyên
- Không chứa đa cung

Đa đồ thị (Multigraph)
- Cung không có hướng
- Không chứa khuyên
- Có chứa đa cung

Giả đồ thị (Pseudograph)
- Cung không có hướng
- Có thể chứa đa cung cung
- Có chứa khuyên

Bậc (degree) của đỉnh, kí hiệu deg(x) = số cung liên thuộc với đỉnh x
Đỉnh có bậc = 0 đgl đỉnh cô lập

Đỉnh có bậc = 1 đgl đỉnh treo

Định lý 1 (định lý bắt tay)

Tổng bậc của tất cả các đỉnh trong 1 đồ thị = 2 lần số cung

$$\sum_{v \in V} deg\,(v) = 2\,|E| = 2e$$

Định lý 2

Số đỉnh bậc lẻ của 1 đồ thị vô hướng là số chẵn

Đơn đồ thị có hướng

- Cung có hướng (x, y) ≠ (y,x)
- Không chứa đa cung

Với mỗi cặp đỉnh x, y chỉ có thể tồn tại nhiều nhất 1 cung dạng (x,y)

Đa đồ thị có hướng (directed multigraph)

- Có thể chứa nhiều cung dạng (x,y)
- Có thể / không chứa khuyên

Cho cung (u,v)

- u **kề với** v (có cung đi từ u đến v)
- v là **đỉnh kề của đỉnh** u

Bậc của đồ thị có hướng

Bậc vào của đỉnh (in-degree), kí hiệu: $deg^-(v)$ = số cung đi đến v

Bậc ra của đỉnh (out-degree), kí hiệu: $deg^+(v)$ = số cung đi ra từ v

Tính chất: $deg(v) = deg^-(v) + deg^+(v)$

Định lý 3

Cho G = <V, E> làm một đồ thị có hướng

$$\sum_{v \in V} deg^-(v) = \sum_{v \in V} deg^+(v) = |E|$$

Sự đẳng cấu của đồ thị

2 đồ thị $G_1 = <V_1, E_1>$ và $G_2 = <V_2, E_2>$ đgl đẳng cấu nếu và chỉ nếu tồn tại *song ánh*:

$f : V_1 \rightarrow V_2$

$v_2 = f(v_1) \in V_2$

sao cho $(x,y) \in E_1 \Leftrightarrow (f(x), f(y)) \in V_2$

Biểu diễn đồ thị

Cấu trúc dữ liệu đồ thị

- Danh sách cung
- Ma trận kề (đỉnh - đỉnh)
- Danh sách kề (danh sách con/ danh sách cha)
- Ma trận đỉnh cung (ma trận liên thuộc, ít sử dụng)

Các phép toán trên cấu trúc dữ liệu đồ thị

- init_graph(G): khởi tạo đồ thị
- adjacent(G,x,y): kiểm tra x và y có kề nhau không
- degree(G,x): tính bậc của đỉnh x
- neighbors(G,x): trả về danh sách các đỉnh kề của x
- add_edge(G,e,x,y): thêm cung e=(x,y) vào G
- remove_edge(G,e,x,y): xóa cung e=(x,y) ra khỏi G

Duyệt đồ thị

Đi qua tất cả đỉnh của một đồ thị cập nhật và/ hoặc kiểm tra giá trị của chúng trên đường đi

Ý tưởng:

- Bắt đầu từ đỉnh bất kỳ đánh dấu nó đã duyệt
- Duyệt các đỉnh kề của nó
- Duyệt các đỉnh kề của đỉnh kề của nó

Các phương pháp duyệt đồ thị

Duyệt theo chiều rộng (Breadth first search - BFS)

- Ý tưởng: "bay" từ đỉnh bắt đầu đến đỉnh cần duyệt → đỉnh gần nhất được duyệt trước (ít cung nhất)
- Thuật toán bfs 2 (Phiên bản cài đặt)
    - Sử dụng **hàng đợi**
    - Các đỉnh trong hàng đợi là các đỉnh *sẽ được duyệt*

```
Đưa 1 đỉnh s bất kì vào hàng đợi (vd: đỉnh 1)
while hàng đợi chưa rỗng do
    Lấy đỉnh ở đầu hàng đợi ra ⇒ gọi là đỉnh u
    if (u đã duyệt) continue; //bỏ qua
    Duyệt u (vd: in u ra màn hình)
    Đánh dấu u đã duyệt  (mark[u] = 1)
    for các đỉnh kề v của u do
        Đưa v vào hàng đợi
```

- Thuật toán bfs 3 (Phiên bản bài tập lý thuyết)
    - Sử dụng **hàng đợi**
    - Các đỉnh trong hàng đợi là các đỉnh *sẽ được duyệt*

```
Đưa 1 đỉnh s bất kì vào hàng đợi (vd: đỉnh 1)
while hàng đợi chưa rỗng do
    Lấy đỉnh ở đầu hàng đợi ra ⇒ gọi là đỉnh u
    if (u đã duyệt) continue; //bỏ qua
    Duyệt u (vd: in u ra màn hình)
    Đánh dấu u đã duyệt  (mark[u] = 1)
    for các đỉnh kề v của u do
        if (v chưa duyệt)          (làm giảm số lần v nằm trong hàng đợi)
            Đưa v vào hàng đợi
```

Duyệt theo chiều sâu (Depth first search - DFS)

- Ý tưởng: khám phá mê cung với dây và phấn → Đi xuống sâu nhất có thể, không đi được nữa thì quay lên
- Thuật toán dfs 1 (Phiên bản cài đặt)
    - Sử dụng **ngăn xếp**
    - Các đỉnh trong ngăn xếp là các đỉnh *sẽ được duyệt.* Một đỉnh có thể có mặt *nhiều lần* trong ngăn xếp

```
Đưa 1 đỉnh s bất kì vào ngăn xếp (vd: đỉnh 1)
while ngăn xếp chưa rỗng do
    Lấy đỉnh ở đầu ngăn xếp ra ⇒ gọi là đỉnh u
    if (u đã duyệt) continue; //bỏ qua
    Duyệt u (vd: in u ra màn hình)
    Đánh dấu u đã duyệt  (mark[u] = 1)
    for các đỉnh kề của u do
        Đưa v vào ngăn xếp
```

- Thuật toán dfs 2 (Phiên bản bài tập lý thuyết)

- Sử dụng **ngăn xếp**
- Các đỉnh trong ngăn xếp là các đỉnh *sẽ được duyệt*. Một đỉnh có thể có mặt *nhiều lần* trong ngăn xếp

```
Đưa 1 đỉnh s bất kì vào ngăn xếp (vd: đỉnh 1)
while ngăn xếp chưa rỗng do
    Lấy đỉnh ở đầu ngăn xếp ra ⇒ gọi là đỉnh u
    if (u đã duyệt) continue; //bỏ qua
    Duyệt u (vd: in u ra màn hình)
    Đánh dấu u đã duyệt  (mark[u] = 1)
    for các đỉnh kề của u do
        if (v chưa duyệt)                    (làm giảm số lần v nằm trong ngăn xếp)
            Đưa v vào ngăn xếp
```

Duyệt theo chiều sâu (đệ quy)

- Không sử dụng stack
- Sử dụng đệ quy
- Thuật toán

```
void DFS (u) {
    if (u đã duyệt) return; //bỏ qua
    //Duyệt u: in u ra màn hình hoặc đánh số đỉnh u
    Đánh dấu u đã duyệt (mark[u] = 1)
    for (các đỉnh kề v của u) do
        DFS(v); //gọi đệ quy duyệt các đỉnh kề v của u
}
```

- Phiên bản kiểm tra trong vòng lặp, phù hợp khi đồ thị không liên thông cần duyệt nhiều lần

```
void DFS (u) {
    //Duyệt u: in u ra màn hình hoặc đánh số đỉnh u
    Đánh dấu u đã duyệt (mark[u] = 1)
    for (các đỉnh kề v của u) do
        if (v chưa duyệt)
            DFS(v); //gọi đệ quy duyệt các đỉnh kề v của u
}
```

Tính Liên Thông

Định nghĩa

- **Đường đi** (walk): đường đi chiều dài k đi từ đỉnh u = v_0 đến đỉnh v_k = v là danh sách các đỉnh và cung xen kẽ nhau

  $v_0,e_1,v_1,e_2,v_2,e_3,...,e_k,v_k$

  Cung e_i có đỉnh đầu là v_{i-1} và đỉnh cuối là v_i
- **Đường đi đơn cung** (trail): là đường đi các cung đều khác nhau
- **Đường đi đơn đỉnh** (path): là đường đi các đỉnh đều khác nhau
- Đường đi đơn cung có đỉnh đầu và đỉnh cuối trùng nhau gọi là một **đường vòng** (circuit)
- Một đường vòng không có đỉnh nào lặp lại gọi là **chu trình** (cycle)
- Chiều dài của walk, trail, path, circuit hay cycle là số cung của nó

Tính liên thông của đồ thị vô hướng

- Định nghĩa
  - Đồ thị vô hướng G được gọi là **liên thông** nếu và chỉ nếu với mọi cặp đỉnh u,v ∈V luôn tồn tại đường đi từ u → v, Ngược lại, G được gọi là không liên thông
  - Đỉnh u được gọi là **liên thông với** đỉnh v ⇔ tồn tại đỉnh đi từ u → v
  - QUan hệ liên thông trên G là tập các cặp có thứ tự (u,v) sao cho u liên thông với v
- Định lý
  - Quan hệ liên thông là một quan hệ tương đương
- Bổ đề
  - Mỗi đường đi đi từ u → luôn chứa 1 đường đi đơn đỉnh từ u → v
- **Các bộ phận liên thông** của đồ thị G là tập các đồ thị con liên thông lớn nhất của G (là các lớp tương đương của quan hệ liên thông)
- *Đỉnh cô lập* (có bậc = 0) cũng là một bộ phận liên chỉ gồm chính nó và được là **bộ phận liên thông tầm thường** (trivial converted component)
- Khoảng cách giữa 2 đỉnh
  - Trên một đồ thị vô hướng, khoảng cách từ đỉnh u đến đỉnh v, ký hiệu **d(u,v)** được định nghĩa bằng:
    - 0, nếu u ≡ v
    - Chiều dài đường đi ngắn nhất từ u → v, nếu tồn tại đường đi từ u → v
    - ∞, nếu không có đường đi từ u → v
  - Đường kính của đồ thị vô hướng là khoảng cách lớn nhất giữa 2 đỉnh trên đồ thị

Thuật toán kiểm tra tính liên thông của đồ thị

∀ u,v ∈ V ⇔ u,v liên thông với nhau ⇔ có đường đi từ u đến v

⇒Duyệt đồ thị từ đỉnh bất kỳ → kiểm tra xem có đỉnh nào chưa duyệt không? →Nếu không còn đỉnh nào chưa duyệt → Đồ thị liên thông

- Áp dụng thuật toán duyệt đồ thị để đánh số/ đánh dấu (gán nhãn) các đỉnh
- Nếu sau khi duyệt tất cả các đỉnh đều có nhãn ⇒ Liên thông

- Khi duyệt đồ thị từ đỉnh s ta sẽ duyệt qua được các đỉnh "đến được từ đỉnh s" (reachable from s)
- Các đỉnh đến được từ s (bao gồm cả s) là bộ phận liên thông chứa s

Thuật toán kiểm tra tính liên thông của đồ thị

Sử dụng giải thuật duyệt đồ thị

Khởi tạo tất cả các đỉnh có num[u] = -1 (chưa duyệt)

Xét qua các đỉnh, nếu u chưa được đánh dấu ⇒ DFS(u) để duyệt nó

```
for (u=1; u <= n; u++)
    if (mun[u] <0){ //u chưa duyệt
        DFS(G,u);
        //Tìm được 1 bộ phận liên thông chứa u
    }
```

Mỗi lần duyệt xong 1 đỉnh u, ta sẽ tìm được 1 bộ phận liên thông chứa u

Có thể thay DFS bằng BFS

Đếm bộ phận liên thông

```
cut =0;
for u ∈ V
    if (u chưa duyệt){
        // duyệt đồ thị
        cut++;
    }
```

Duyệt toàn bộ đồ thị

```
for u ∈ V
    if (u chưa duyệt)
        // duyệt đồ thị
```

Tính liên thông của đồ thị có hướng

Cho đồ thị có hướng G = <V,E>

G được gọi là **liên thông yếu** ⇔ đồ thị vô hướng nền của nó liên thông

G được gọi là **liên thông mạnh** ⇔ giữa hai đỉnh x, y bất kỳ, luôn có đường đi từ x đến y

Bộ phận liên thông mạnh

- Đồ thị con liên thông mạnh: có đường đi giữa 2 đỉnh bất kỳ
- Đồ thị có hướng không liên thông mạnh bao gồm nhiều bộ phận liên thông mạnh

Các đỉnh trong một chu trình liên thông với nhau

Giải thuật tìm các bộ phận liên thông mạnh:

- Tìm các chu trình (lớn nhất có thể) của một đồ thị

Thuật toán

- Áp dụng duyệt theo chiều sâu để đánh số các đỉnh
- Để tìm được chu trình, với mỗi đỉnh v, ngoài num[v], ta lưu thêm min_num[u] (là chỉ số nhỏ nhất trong các đỉnh có thể đi đến được từ v). Tròng quá trình duyệt, min_num[v] sẽ được cập nhật
- Khi duyệt xong 1 đỉnh, nếu num[v] = min_num[v] thì v là đỉnh bắt đầu (đỉnh gốc/ đỉnh khớp) của bộ phận liên thông mạnh

Các biến hỗ trợ

- **S**: stack lưu các đỉnh chưa tìm được bộ phận liên thông mạnh
- **on_stack[v]**: kiểm tra v còn trên stack không
- **num[v]**: chỉ số của đỉnh v trong quá trình duyệt
- **min_num[v]**: chỉ số nhỏ nhất trong các chỉ số của các đỉnh trong stack S mà v đi đến nó được
- **k**: chỉ số dùng để gán cho num của các đỉnh (tăng dần)

```
void SCC (int u){
    //1. Đánh số cho đỉnh u, đưa u vào ngăn xếp
    num[u] = k;
    min_num[u] = k;
    k++;
    push(S,u);
    on_stack[u] = true;
```

```
        //2. Lần lượt xét các đỉnh kề của u
        for (v là các đỉnh kề của u)
            if (v chưa duyệt){
                //2a. Duyệt v là cập nhật lại min_num[u]
                SCC(v);
                min_num[u] = min(min_num[u], min_num[v]);
            }
            else
                if (v còn trên stack){
                    //2b. Cập nhật lại min_num[u]
                    min_num[u] = min(min_num[u], num[v]);
                }
                else            //2c. bỏ qua, không làm gì cả
        //3. Kiểm tra num[u] == min_num[u]
        if (num[u] == min_num[u]) {
            //Lấy các đỉnh trong stack ra cho đến khi gặp u
            //Các đỉnh này thuộc về một thành phần liên thông
            int w;
            do {
                w = top (&S);
                pop (&S);
                on_stack[w] = 0;
                //làm gì đó trên w, vd: in ra màn hình
            } while (w != u);
        }
    }
```

Lưu ý:

- Thứ tự các đỉnh kề của 1 đỉnh được sắp xếp từ bé đến lớn
- Khi duyệt xong 1 đỉnh v, qua về đỉnh cho u (đỉnh trước), cập nhật lại min_num[u] (so với **min_num[v]**)
- Khi xét 1 đỉnh kề với u mà v đang có mặt trong stack ⇒ cập nhật lại min_num[u] (so với **num[v]**)

Cây duyệt theo chiều sâu

- Quá trình duyệt cây kết hợp với đánh số (num, min_num) tạo ra cây “duyệt đồ thị”

Ứng dụng của duyệt đồ thị

Dựng cây duyệt đồ thị

Tổ chức hàng đợi / ngăn xếp có khả năng lưu

- Đỉnh sắp sửa duyệt và đỉnh cha của nó (đỉnh đưa nó vào hàng đợi / ngăn xếp)

```
typedef struct{
        int u; //đỉnh sẽ được duyệt
        int p; //đỉnh cha của u
} ElementType;
typedef struct{
        int front, rear;
        ElementType data[MAX_SIZE];
} Queue;
int parent[MAX_N];
Đưa 1 đỉnh (s,-1) vào hàng đợi
while hàng đợi chưa rỗng do
        Lấy phần tử ở đầu hàng đợi ra ⇒ (u,p)
        parent[u] = p; //Cho u là con của p
        for các đỉnh kề v của u do
                if (v chưa duyệt)
                        Đưa (v,u) vào hàng đợi
```

Duyệt theo chiều sâu dùng đệ quy (DFS đệ quy)

```
void DFS (int u, int p) {
        if (u đã duyệt)
        return;
        //Duyệt u, đánh dấu u đã duyệt
        //cho p là cho của u
        parent[u] = p
        for (các đỉnh kề v của u)

                DFS(v,u);
}
```

```
void DFS (int u, int p) {


        //Duyệt u, đánh dấu u đã duyệt
        //cho p là cho của u
         parent[u] = p
        for (các đỉnh kề v của u)
                if (v chưa duyệt)
                        DFS(v,u);
}
```

Tìm đường đi

Tìm đường đi từ s đến t

- Duyệt đồ thị từ đỉnh s và dựng cây trong quá trình duyệt
- Nếu trong quá trình duyệt, đỉnh t được duyệt ⇒ có đường đi
- Nếu t không được duyệt ⇒ không có đường đi

Kiểm tra đồ thị chứa chu trình

Chu trình (cycle) là đường đi (path) có đỉnh đầu trùng đỉnh cuối

Để kiểm tra đồ thị có hướng chứa chu trình

- Duyệt u
- Xét các đỉnh kề v của u

- Có 3 trường hợp
  - v chưa duyệt ⇒ Không tạo chu trình
  - v đã duyệt và xét kết các kề của nó → đã duyệt ⇒ Không tạo chu trình
  - v đã duyệt nhưng chưa xét hết kề của nó → đang duyệt ⇒ Tạo chu trình

Thuật toán

- DFS + tô màu các đỉnh trong quá trình duyệt (TRẮNG / XÁM / ĐEN)
- Nếu khi xét 1 đỉnh kề v của u mà v có có màu xám ⇒ Tạo chu trình

```
void DFS (u) {
        color [u] = GRAY;                   //1. Đang duyệt u
        for (các đỉnh kề v của u)           //2. Xét kề
            if (v có màu trắng)             //2a. v chưa duyệt
                DFS (v);
            else if (v có màu xám)          //2b. v còn đang duyệt
                has_cycle = 1;              //Tạo chu trình
            else {}                         //2c. v đã duyệt xong ⇒ bỏ qua
        color [u] = BLACK.                  //3. Duyệt xong u
}
```

Kiểm tra trên toàn bộ đồ thị: Vòng lặp (u từ 1 đến n) + DFS(u)

```
int color [MAX_N];
int is_cycle;
void DFS (int u) {
        …
}
int main() {
        …
        for (u = 1; u <= n; u++)
            color [u] = WHITE;
        has_cycle = 0;
        for (u = 1; u <= n; u++)
            if(color[u] == WHITE)
                DFS (u);
        if(has_cycle)
            //Chứa chu trình
        else
            //Không chứa chu trình
        …
}
```

Để kiểm tra đồ thị vô hướng chứa chu trình

- Duyệt u
- Xét các đỉnh kề v của u
- Có 3 trường hợp
  - v chưa duyệt ⇒ Không tạo chu trình
  - v đang duyệt, v là cho của u ⇒ Không tạo chu trình
  - v đang duyệt ⇒ Tạo chu trình

Thuật toán

```
void DFS (u, p) {
        color [u] = GRAY;                   //1. Đang duyệt u
        for (các đỉnh kề v của u)           //2. Xét kề
            if (v==p)                       //2a. v là cha của p ⇒ bỏ qua
```

```
            else if (v có màu trắng)      //2b. v chưa duyệt
                DFS (v);
            else if (v có màu xám)        //2c. v còn đang duyệt
                has_cycle = 1;            //Tạo chu trình
            else {}                       //2d. v đã duyệt xong (KHÔNG CÓ)
        color [u] = BLACK;                //3. Duyệt xong u
    }
```

Kiểm tra đồ thị phân đôi / đồ thị 2 bên / đồ thị 2 phần / đồ thị 2 phía

⇒ Đồ thị VÔ HƯỚNG
⇒ Có thể chia các đỉnh của đồ thị thành 2 phần (không giao nhau) sao cho các đỉnh trong mỗi phần *không* kề nhau

- Duyệt u
- Xét các đỉnh kề v của u
- Có 3 trường hợp
  - v chưa có màu ⇒ Tô màu ngược lại với màu của u
  - v có màu GIỐNG với u → Không tô được ⇒ Đồ thị không phân đôi
  - v có màu KHÁC với u ⇒ Bỏ qua

Thuật toán

- Duyệt đồ thị + tô màu các đỉnh bằng 1 trong 2 màu (XANH, ĐỎ).
- Hai đỉnh kề nhau có màu khác nhau

```
#define NO_COLOR    -1
#define BLUE         1
#define RED          2
void colorize (u, c){
    color[u] = c;                          //1. Đang duyệt u
    for (các đỉnh kề v của u)              //2. Xét kề
        if (v == NO_COLOR){                //2a. v chưa có màu
            colorize(v, 3-c);              //tô màu ngược với c
        }
        else if (color[v] == color[u])     //2b. màu giống với màu u
            conflict = 1;                  // Đụng độ, không tô được
        else {}                            //2c. Màu khác nhau, bỏ qua
}
```

Đồ thị Euler & đồ thị Hamilton

Đồ thị Euler

Chu trình Euler (Euler circuit) trong đồ thị G là một *chu trình đơn cung* chứa tất cả các cung của G
Đường đi Euler (Euler path) trong đồ thị G là *đường đi đơn cung* chứa qua tất các đỉnh của G

Định lý Euler

Đa đồ thị (vô hướng) liên thông có *ít nhất 2 đỉnh* có **chu trình Euler** khi và chỉ khi tất cả các đỉnh của nó đều có bậc chẵn.

Đa đồ thị (vô hướng) liên thông có **đường đi Euler** (nhưng không có chu trình Euler) khi và chỉ khi nó có đúng 2 đỉnh bậc lẻ.

Thuật toán Fleury xây dựng chu trình Euler

- Chọn 1 đỉnh bất kỳ
- Chọn tiếp các cung liên tiếp cho đến khi tạo thành chu trình. Mỗi khi một cung được chọn, xóa nó ra khỏi đồ thị
- Các cung được chọn sao cho đỉnh đầu của cung mới là đỉnh cuối của cung cũ VÀ nó không phải là cầu (cầu: xóa bỏ cung này sẽ làm đồ thị mất liên thông) trừ phi không có lựa chọn khác.

Thuật toán đồ thị Euler

- **Cài đặt đệ quy**

```
void findEudlerPath (u) {
        for (các đỉnh kề v của u)
                xóa cung (u, v)
                findEudlerPath (v);
        Thêm u và path
}
```

- **Cài đặt dùng ngăn xếp**

S: stack

```
void findEudlerPath(u) {
        while (S != rỗng) {
                Lấy phần tử trên đỉnh S, gọi nó là u
                if (u không còn đỉnh kề)
                        Thêm u vào path
                for (v = 1; v <= n; v++)
                        if (v kề với u) {
                                xóa cung (u, v)
                                push v vào S
                                break; //Chỉ cần lấy 1 kề đầu tiên
                        }
        }
}
```

Ứng dụng

- Đi qua các con đường
- Giao hàng
- Chinese postman problem

Đồ thị Hamilton

Chu trình Hamilton (Hamilton circuit): *Chu trình đơn đỉnh* (simple circuit) đi quá các đỉnh của đồ thị, mỗi đỉnh đúng 1 lần (trừ đỉnh xuất phát)

Đường đi Hamilton (Hamilton path): *Đường đi đơn đỉnh* (simple path) đi qua các đỉnh của đồ thị G, mỗi đỉnh đúng 1 lần

Định lý Dirac:

Nếu G là đơn đồ thị có số đỉnh n ≥ 3 và bậc của các đỉnh đều ≥ n/2 thì G có chu trình Hamilton

Định lý Ore:

Nếu G là đơn đồ thị có số đỉnh n ≥ 3 và deg(u) + deg(v) ≥ n với mọi cặp đỉnh không kề nhau trong G, thì G có chu trình Hamilton.

Ứng dụng

- Xếp tour du lịch
- Bài toán người giao hàng (Traveling Salesman Problem – TSP)
- Mã Gray

Tìm đường đi ngắn nhất

Biểu diễn đồ thị có trọng số

Chiều dài đường đi ⇒ Trọng số / chiều dài cung

- Ma trận kề (đỉnh – đỉnh) => ma trận trọng số
  - Nếu có (u, v) thì **W[u][v]** = trọng số của cung (u, v)
  - Nếu không có (u, v) thì W[u][v] = NO_EDGE (vd: -1)

```
#define NO_EDGE -1
#define MAX_N 100
typedef struct {
        int n, m;
        int/double W[MAX_N][MAX_N];
} Graph;
```

- Nếu đồ thị có đa cung ⇒ tạo Danh sách cung
  - Mỗi cung lưu trữ: đỉnh đầu, đỉnh cuối và **trọng số**

```
#define NO_EDGE -1
#define MAX_N 100
typedef struct {
        int u, v;
        int/double w;
} Edge;
```

Graph = danh sách Edge

Bài toán đường đi ngắn nhất (shortest path)

- Đường đi ngắn nhất (shortest path) là đường đi có chiều dài nhỏ nhất. ⇒ Chia để trị
- Một đoạn đường đi (subpath) của đường đi ngắn nhất cũng là đường đi ngắn nhất.
  - Tính chất ***cấu trúc con tối ưu*** (optimal substructure) => có thể áp dụng kỹ thuật *quy hoạch động* để thiết kế các giải thuật tìm đường đi ngắn nhất

Thuật toán Moore - Dijkstra

- Tìm đường đi ngắn nhất từ 1 đỉnh đến các đỉnh khác trên đồ thị có trọng số
- Điều kiện áp dụng:
  - Đồ thị có trọng số không âm

- Có hướng hoặc vô hướng đều được

- Ý tưởng chính:
  - Khởi tạo đường đi trực tiếp từ s đến các đỉnh
  - Lần lượt cập nhật lại đường đi nếu tìm được đường đi mới tốt hơn đường đi cũ
- Thuật toán qua ví dụ

  // x = pi [u]: chiều dài đường đi ngắn nhất tính đến thời điểm hiện tại

  1. Chọn 1 đỉnh chưa chắc chắn u có pi[u] nhỏ nhất
  2. Đánh dấu u là đỉnh chắc chắn
  3. Cập nhật các đỉnh kề v của u pi[v] = min(pi[v], pi[u] + L[u][v])
- Thuật toán tổng quát

  // W không âm, Đỉnh bắt đầu: s

  // pi [u]: chiều dài đường đi ngắn nhất (tạm thời) từ s đến u.

  // p[u]: đỉnh trước của u trên đường đi ngắn nhất (tạm thời) từ s → u

  // mark[u]: đánh dấu đỉnh u

  1. Khởi tạo: pi [s] = 0;
     pi[v] = ∞ (v ≠ s) với mọi u
     p[u] = -1 với mọi u
     mark [u] = 0 ∀ u (0: chưa chắc chắn; 1: chắc chắn)
  2. Lặp (n-1): Tìm đỉnh mark[u] = 0 và pi bé nhất
     Đánh dấu chắc chắn mark[u] = 1;
     Xét các kề v chưa chắc chắn của u:
     if (pi[v] > pi[u] + W[u][v])
     pi[v] = pi[u] + W[u][v];
     p[v] = u;
- Thuật toán

```
void Dijkstra(Graph *pG, int s) {
    Khởi tạo
    int u, v, it;
    for (u = 1; u <= pG->n; u++) {
        mark[u] = 0;
        pi[u] = INFINITY;
    }
    pi[s] = 0;
    Lặp n – 1 lần:
    for (it = 1; it < pG->n; it++) {
        //1. Tìm u chưa chắc chắn và có pi[u] nhỏ nhất
        //2. Đánh dấu u là đỉnh chắc chắn: mark[u] = 1;
        //3. Xét các kề của v để cập nhật pi và p (nếu thoả)
        for (v = 1; v <= G->n; v++)
            if (pG->W[u][v] != NO_EDGE && mark[v] == 0) {
                if (pi[u] + pG->W[u][v] < pi[v]) {
                    pi[v] = pi[u] + pG->W[u][v];
                    p[v] = u;//cập nhật đỉnh trước của v
                }
            }
    }
}
```

Thứ tự topo và ứng dụng

Thứ tự topo

- Sắp xếp các đỉnh sao cho đỉnh gốc (của 1 cung) phải đứng trước đỉnh ngọn
- Đồ thị có hướng không chu trình (Directed Acyclic Graph - DAG) → Tồn tại ít nhất 1 thứ tự topo
- Thuật toán sắp xếp topo dựa trên BFS
  - Khởi tạo

    Tính bậc vào của các đỉnh, d[u]
    Q chứa các đỉnh có bậc vào = 0
    L = rỗng
  - Lặp

    while (Q khác rỗng)
        Lấy phần tử đầu hàng đợi Q, gọi nó là u
        Thêm u vào L
        for các đỉnh kề v của u
            Giảm bậc vào của đỉnh v
        if bậc vào của v = 0
            Thêm v vào Q

Thuật toán sắp xếp topo dựa trên DFS

```
DFS(u)
    mark[u] = 1 //1. Đánh dấu đã duyệt
    for các đỉnh kề v của u //2. Xét các đỉnh kề
        if mark[v] == 0
            DFS(v)
    Đưa u vào L //3. Sau khi duyệt xong, đưa u vào L
```

- Đảo ngược danh sách L => Thứ tự topo
- Xếp hạng các đỉnh của đồ thị (dựa trên BFS)
  - Xếp hạng cho (các) gốc (hạng 0)
  - Loại bỏ gốc ra khỏi đồ thị => xuất hiện gốc mới
  - Xếp hạng cho gốc mới (hạng 1)
  - Loại bỏ gốc ra khỏi đồ thị => xuất hiện gốc mới
  - Nhận xét:
    - Xoá đỉnh u: giảm bậc vào của các đỉnh kề của u
    - Đỉnh kề nào có bậc vào giảm về 0 => gốc mới
  - Cài đặt
    - Các biến hỗ trợ

      d[u]: bậc vào của đỉnh u
      S[k]: danh sách các gốc ở bước k
      rank[u]: hạng của đỉnh u
      k: bước lặp
    - Khởi tạo

      Tính bậc vào của các đỉnh: d[u]
      Đưa các gốc vào S[0]
      k = 0
      while (S[k] != rỗng)
          //1. Làm rỗng danh sách S[k+1]

```
        //2. Xếp hạng và giảm bậc của đỉnh kề
        for (các đỉnh u trong danh sách S[k])
            rank[u] = k // Xếp hạng cho u
            for (các đỉnh kề v của u)
                d[v]-- //Xoá u (giảm bậc vào của các đỉnh kề của u)
                if (d[v] == 0)
                    đưa v vào S[k+1]
k++
```

Quản lý dự án

- Dự án (project)
  - Danh sách công việc (activities/tasks), mỗi công việc có thời gian hoàn thành
  - Có sự phụ thuộc giữa các công việc (vd: công việc B chỉ có thể bắt đầu làm khi làm xong công việc A)
- Vấn đề
  - Kế hoạch thực hiện các công việc
  - Ước tính thời gian hoàn thành
- Mô hình hoá về đồ thị
  - Công việc (CV) => Đỉnh
  - Sự phụ thuộc giữa 2 CV => Cung
  - Thời gian hoàn thành CV u (d[u]) => trọng số của đỉnh u
  - Thêm 2 đỉnh: α, β (tương ứng với 2 công việc giả)
  - Thêm cung nối α với các đỉnh có bậc vào bằng 0
    Thời gian hoàn thành α: d[α] = 0
  - Thêm cung nối các đỉnh có bậc ra bằng 0 với β
- Thời điểm bắt đầu sớm nhất (early start time) của công việc, ký hiệu là **t[u]**
  Thời điểm sớm nhất có thể bắt đầu công việc u
  t[β]: thời điểm sớm nhất hoàn thành dự án
- Tính t[u]
  - t[α] = 0
  - Xét các đỉnh u theo thứ tự hạng tăng dần (theo thứ tự topo)
    t[u] = max {t[x] + d[x]} với x là đỉnh tương ứng với công việc trước của cung việc u
- Thời điểm bắt đầu trễ nhất (late start time) của công việc u, ký hiệu **T[u]**
  Thời điểm trễ nhất để bắt đầu công việc u mà không ảnh hưởng đến tiến độ dự án
- Tính T[u]
  - T[β] = t[β]
  - Xét các đỉnh u theo thứ tự hạng giảm dần
    T[u] = min {T[v]} - d[u] với v là đỉnh kề của u
- Công việc then chốt (critical activities) **t[u] = T[u]**
- Đường nối các công việc then chốt: đường then chốt (critical path)

Cây & Cây khung tối thiểu

Cây vô hướng

- Cây (tree): đồ thị vô hướng liên thông và không có chu trình ⇒ đơn đồ thị
- Rừng (forest): đồ thị vô hướng không có chu trình
- Mỗi BPLT của rừng là 1 cây.
- Đồ thị liên thông tối thiểu (minimally connected)
  - Đồ thị liên thông và nếu xoá một cung bất kỳ thì nó không còn liên thông nữa.
  - Đồ thị liên thông tối thiểu không chứa chu trình
- Định lý:
  1. Đồ thị G là cây ⇔ giữa mỗi cặp đỉnh của G có đúng 1 đường đi (path).
  2. Cây có n đỉnh sẽ có n-1 cung
  3. Đồ thị liên thông n đỉnh, n-1 cung là 1 cây
  4. Đồ thị G là cây ⇔ G liên thông tối thiểu
  5. Đồ thị G có n đỉnh, n-1 cung, không chứa chu trình thì liên thông.
  6. Cây có ít nhất 2 đỉnh thì sẽ có ít nhất 2 đỉnh treo (đỉnh bậc 1)
  7. 1 rừng n đỉnh, k cây sẽ có n – k cung

Cây khung (spanning tree)

- Cây khung (spanning tree): cây bao trùm/cây phủ
  - G =là đồ thị vô hướng liên thông
  - Đồ thị con T = là cây khung của G:
    - T là cây
    - Chứa tất cả các đỉnh của G Có |V| - 1 cung
- Đồ thị (liên thông) có thể có nhiều cây khung

Thuật toán tìm cây khung nhỏ nhất (MST)

- Minimum spanning tree (MST)
  - Cây khung nhỏ nhất/cây khung tối thiểu
  - Cây khung có trọng lượng nhỏ nhất
- Cây T là cây khung tối tiểu của G
  - T là cây khung của G
  - T có tổng trọng số nhỏ nhất trong tất cả các cây khung của G

Bài toán xây dựng đường

Chọn các cung giữ lại sao cho đồ thị liên thông
Cây chứa tất cả các đỉnh của đồ thị ban đầu ⇒ Cây khung
Xây dựng đường đi có chi phí thấp nhất
→ Đồ thị vô hướng có trọng số
Chọn các cung để giữ lại sao cho tạo thành 1 cây và tổng trọng số bé nhất
⇒ Tìm 1 cây khung trọng lượng nhỏ nhất ⇒ Cây khung nhỏ nhất (MST)

Thuật toán:

1. Khởi tạo các đỉnh (không có cung)
2. Lần lượt xét từng cung của đồ thị gốc
   - Không tạo chu trình → thêm cung

- Tạo chu trình → bỏ qua

Thuật toán Kruskal

1. Sắp xếp các cung của G theo thứ tự trọng số tăng dần
2. Khởi tạo cây T rỗng (không chứa cung nào cả)
3. Lần lượt xét từng cung của G (theo thứ tự đã sắp xếp)
   - Nếu thêm cung e = (u, v) vào T mà không tạo thành chu trình thì thêm e vào T
   - Ngược lại, bỏ qua cung e
   - Điều kiện dừng: T có n – 1 cung hoặc tất cả các cung của G đều đã được xét

⇒ Kiểm tra thêm 1 cung (u,v) có tạo chu trình không
(Khi u và v nằm ở 2 BPLT khác nhau => không tạo chu trình)
⇒ Tìm bộ phận liên thông
Cài đặt:

- 1 BPLT được tổ chức như 1 CTDL cây
- Đỉnh gốc: đại diện cho BPLT
- Mỗi đỉnh đều có 1 mũi tên chỉ về đỉnh cha (parent) của nó.
- Đỉnh gốc chỉ vào chính nó (có đỉnh cha là chính nó)

Cài đặt thuật toán:

```
Sắp xếp các cung của G theo thứ tự trọng số tăng dần
Khởi tạo cây T rỗng (không chứa cung nào cả)
for (các đỉnh u của G)
        parent[u] = u; //Mỗi đỉnh là 1 BPLT
Lần lượt xét từng cung của G
        r_u = find_root(u); //r_u là gốc của u
        r_v = find_root(v); //r_v là gốc của v
        if (r_u != r_v) //2 BPLT khác nhau => không tạo CT
                Thêm cung (u, v) vào T
                parent[r_v] = r_u; //cho gốc của v làm con của gốc của u
Điều kiện dừng: T có n – 1 cung hoặc tất cả các cung của G đều đã được xét
```

Thuật toán Prim

1. Khởi tạo cây T rỗng
2. Chọn 1 đỉnh u bất kỳ làm đỉnh bắt đầu, đánh dấu nó đã xét (mark[u] = 1), các đỉnh khác đều chưa xét.
3. Lặp n - 1 lần
   - Tìm đỉnh v chưa xét, gần với 1 trong các đỉnh u đã xét nhất
   - Thêm cung (u, v) vào T
   - Đánh dấu v đã xét

⇒ Kết quả có thể khác Kruskal
⇒ Cài đặt giống Moore -Dijkstra
Các biến hỗ trợ

- mark[u]: đỉnh u đã xét (1) hay chưa (0)
- pi[u]: khoảng cách ngắn nhất từ u đến 1 trong các đỉnh đã xét
- p[u]: đỉnh đã xét gần với u nhất

Cài đặt thuật toán

- Khởi tạo

```
Với mọi u, mark[u] = 0; ![u] = ∞
for (các đỉnh kề u của 1)
    π[u] = oo; //khoảng cách ngắn nhất đến u
    p[u] = -1;  //đỉnh kết nối u vào cây
```

mark[u] = 0; //tất cả các đỉnh đều chưa đưa vào cây
π[s] = 0 ; //s là đỉnh bắt đầu, có thể chọn s = 1

- Lặp n lần
  Chọn đỉnh u chưa xét, có π[u] nhỏ nhất
  mark[u] = 1; //Đánh dấu u đã xét
  if u != -1, thêm cung (p[u], u) vào T
  for (các đỉnh kề v chưa xét của u)
  Cập nhật lại π[v] và p[v] (nếu W[u][v] < π[v])

Cây có hướng và Cây khung tối thiểu

Bài toán xây dựng hệ thống dẫn nước

- Đỉnh: nhà/nhà máy nước, gọi tắt là địa điểm
- Cung: đường ống nối giữa các địa điểm với nhau. Nước chỉ chảy 1 chiều => cung có hướng
- Trọng số cung: chi phí xây dựng đường ống tương ứng

⇒ Đồ thị có hướng, có trọng số

Định nghĩa

Cây có hướng

- Đồ thị có hướng G = <V, E> là một cây có hướng, gốc r khi và chỉ khi:
  - G không có chu trình vô hướng
  - Luôn có đường đi từ r đến các đỉnh khác
- Định lý: G là cây có hướng gốc r
  1. Tồn tại đỉnh r được nối với mỗi một đỉnh khác bằng một đường đi duy nhất xuất phát từ r.
  2. Gần liên thông mạnh và cực tiểu đối với tính chất này.
  3. Liên thông và tồn tại một đỉnh r có bậc trong bằng không và bậc trong của những đỉnh khác r là bằng 1.
  4. Không có chu trình và tồn tại một đỉnh r có bậc trong bằng không và bậc trong của những đỉnh khác r là bằng 1.
  5. Gần liên thông mạnh và không có chu trình.
  6. Gần liên thông mạnh và có n-1 cung.

Cây khung có hướng

Cây khung của đồ thị G

- Cây có hướng
- Gồm tất cả các đỉnh của đồ thị G

Cây khung có hướng nhỏ nhất

Cây khung có hướng có tổng số trọng số các cung nhỏ nhất

Thuật toán Chu-Liu/Edmonds

Pha co

Gọi đồ thị gốc là G_0, t = 0

Lặp

Xây dựng đồ thị xấp xỉ H_t từ G_t

Trừ gốc ra, với mỗi đỉnh còn lại giữ lại 1 cung đi đến nó có trọng số nhỏ nhất (bỏ các cung khác đi)

Nếu H_t không chứa chu trình => thoát vòng lặp chuyển sang pha giãn

Ngược lại co G_t thành G_{t+1}

t = t + 1

Pha giãn

Giãn cây khung T_{t+1} thành cây khung T_t của đồ thị G_t

Mở đỉnh (được gom lại trong pha co) => chu trình

Điều chỉnh trọng số của cung đi đến chu trình

Xóa bỏ 1 cung trong chu trình

Luồng cực đại trong mạng

Mạng (các luồng) & luồng trong mạng

Mạng (các luồng) Network (of flows)/Flow network là 1 bộ 5: N = **<V, E, s, t, c>** trong đó

- <V, e> là đồ thị có hướng biểu diễn cho mạng
  V: đỉnh/nút
  E: cung
- s: đỉnh phát (source)
- t: đỉnh thu (sink)
- c: (u,v) → c(u,v) hàm mô tả khả năng thông qua của 1 cung
- c(u,v): luồng lớn nhất có thể đi qua cung (u, v)

Luồng (trên mạng) (Network) Flow: thứ lưu thông trên mạng

- Một luồng đi từ đỉnh phát (s) đến đỉnh thu (t) là 1 hàm f: (u, v) -> f(u,v)
  thoả mãn các điều kiện:
  - Với mỗi cung: 0 <= f(u,v) <= c(u,v)
  - Với mỗi đỉnh khác s và t: tổng luồng vào = tổng luồng ra
  - Tổng luồng ra khỏi s = tổng luồng vào t

  Giá trị luồng |f| = tổng luồng ra khỏi s
- Mẹo: luồng s-t là cách gán các giá trị luồng cho từng cung của mạng sao cho thoả mãn điều kiện luồng

Luồng cực đại

Cho N = **<V, E, s, t, c>**, tìm luồng f (từ s đến t) có giá trị luồng |f| lớn nhất.
Luồng cực đại → luồng có giá trị lớn nhất

Lát cắt

Một lát cắt cát tách s và t, (gọi là: **s-t cut**) Là một cách chia (phân hoạch) tập đỉnh V thành hai phần rời nhau (S, T) sao cho s nằm trong S và t nằm trong T
Khả năng thông qua của 1 lát cắt (tách s và t) = Tổng khả năng thông qua của các cung của lát cắt
(Tổng các cung từ S đến T)
Luồng đi qua lát cắt = luồng ra khỏi s = luồng đi vào t
Tổng luồng đi ra khỏi S - Tổng luồng vào S = Giá trị luồng
Giá trị luồng không vượt quá khả năng thông qua của lát cắt
Nếu |f| = c(S, T) thì f là luồng lớn nhất (luồng cực đại) và (S, T) là lát cắt hẹp nhất

Thuật toán/Phương pháp Ford – Fulkerson

- Giá trị luồng cực đại = khả năng thông qua của lát cắt hẹp nhất
- Ý tưởng: tìm lát cắt hẹp nhất bằng cách tăng luồng đi qua mạng → Tìm đường đi từ s đến t sao cho có thể tăng thêm luồng trên đường đi đó
  ⇒ Xây dựng đồ thị còn dư/thặng dư (residual graph)
- Đường tăng luồng = đường đi từ s đến t trên đồ thị thặng dư
- Gán nhãn (đánh dấu) các đỉnh để tìm đường tăng luồng trên đồ thị thặng dư
  - Không cần xây dựng tường minh đồ thị thặng dư
  - Sử dụng duyệt đồ thị để tìm đường đi từ s đến t
- Mỗi đỉnh u được đánh dấu/gán nhãn với các thông tin sau:

- d[u]: hướng của cung (+: cung thuận, -: cung nghịch "undo" luồng)
- p[u]: đỉnh trước đỉnh u
- σ[u]: lượng tăng luồng lớn nhất có thể tăng

Thuật toán Edmonds – Karp sử dụng tìm kiếm theo *chiều rộng* để tìm đường tăng luồng

1. Gán nhãn s: (+, s, ∞)
2. Push s vào hàng đợi Q
3. while (Q không rỗng) {
    a. u = front(Q); dequeue(Q);
    b. for (các đỉnh kề v của u và có f(u, v) < c(u, v))
        Gán nhãn v: (+, u, min{σ[u], c(u, v) – f(u, v)})
        Đưa v vào Q
    c. for (các đỉnh x có cung đi đến u có f(x, u) > 0)
        Gán nhãn x: (-, u, min{σ[u], f(x, u)})
        Đưa x vào Q
    d. Nếu t được đánh dấu => thoát vòng lặp while
}

Thuật toán Tăng luồng theo đường tăng luồng

```
u = t;
sigma = σ[t]
while (u != s) {
    if (d[u] == '+') //Cung thuận
        f(p[u], u) += sigma; //TĂNG LUỒNG
    else //Cung nghịch
        f(u, p[u]) -= sigma; //GIẢM LUỒNG
    u = p[u];
}
```

Khởi tạo f(u, v) = 0 với mọi cung (u, v)

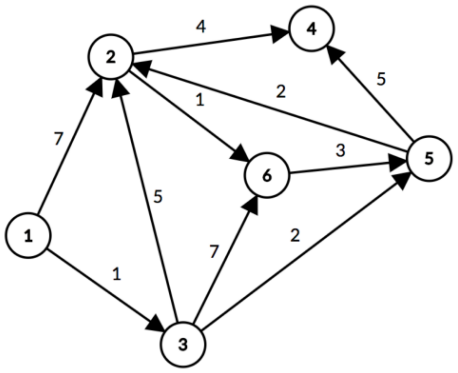
```
while (1) {
    Tìm đường tăng luồng
    Nếu không tìm thấy => thoát vòng lặp (break)
    Tăng luồng theo đường tăng luồng
}
```

Lát cắt hẹp nhất = (S, T)

- S: các đỉnh đã có nhãn
- T: là các đỉnh chưa có nhãn

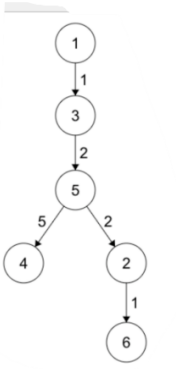
Bài tập

Moore - Dijkstra: Tìm đường đi ngắn nhất từ 1 đến các đỉnh khác trên đồ thị có hướng

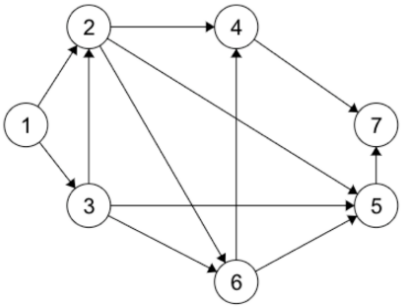


	1	2	3	4	5	6	Công việc
Khởi tạo	1	oo	oo	oo	oo	oo	mark[u] = 0 pi[u] = oo (u != 1) pi[1] = 0
1	*	7/1	1/1				Chọn 1, Cập nhật 2,3
2		6/3	*		3/3	8/3	Chọn 3, CN 2,5,6
3		5/5		8/5	*		Chọn 5, CN 2,4
4		*				6/2	Chọn 2, CN 6
5						*	Chọn 6

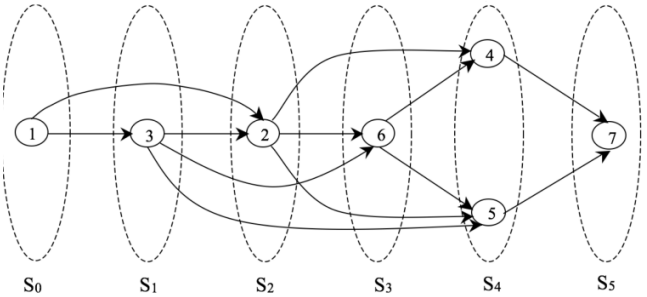
Cây đường đi ngắn nhất:



Thứ tự topo: Xếp hạng đồ thị



	1	2	3	4	5	6	7	S[k+1]
Khởi tạo	0	2	1	2	3	2	2	1
0	*	1	0					3
1		0	*		2	1		2
2		*		1	1	0		6
3				0	0	*		4,5
4				*	*		0	7
5							*	
Kết quả	0	2	1	4	4	3	5	

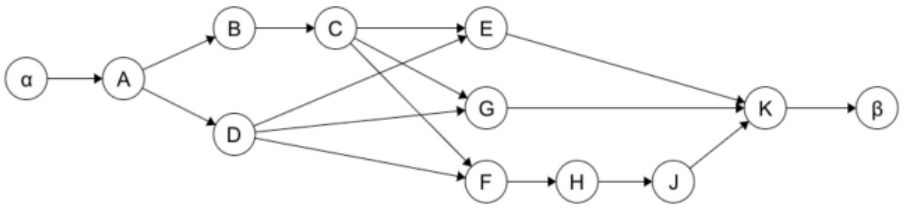


Thứ tự topo: 1, 3,2,6,4,5,7
1,3,2,6,5,4,7

Quản lý dự án

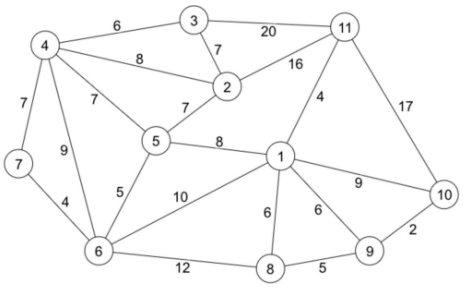
Công việc	Thời gian thực hiện d[i] tính theo tuần	Công việc trước đó
A	7	
B	3	A
C	1	B
D	8	A
E	2	C,D

F	1	C,D
G	1	C,D
H	2	F
J	2	H
K	1	E,G,J



	α	A	B	C	D	E	F	G	H	J	K	**β**
d[u]	0	7	3	1	8	2	1	1	2	2	1	0
t[u]	0	0	7	10	7	15	15	15	16	18	20	21
T[u]	0	0	11	14	7	18	15	19	16	18	20	21
CV then chốt	*	*			*		*		*	*	*	*

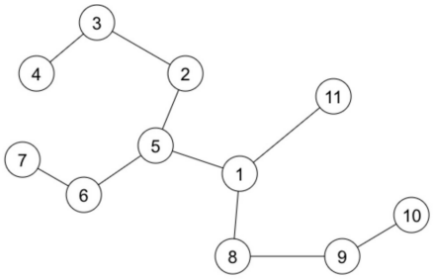
Thuật toán Kruskal



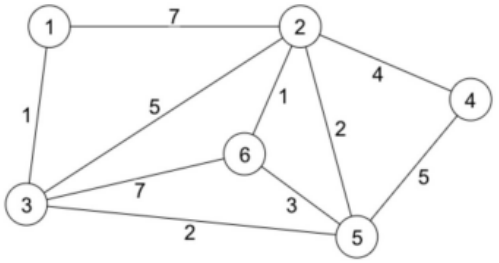
	u	v	w	Thêm vào cây
1	9	10	2	**Thêm**
2	1	11	4	**Thêm**
3	6	7	5	**Thêm**
4	5	6	5	**Thêm**
5	8	9	5	**Thêm**
6	1	8	6	**Thêm**
7	1	9	6	Không

8	3	4	6	**Thêm**
9	2	3	7	**Thêm**
10	2	5	7	**Thêm**
11	4	5	7	Không
12	4	7	7	Không
13	1	5	8	**Thêm**

Cây khung nhỏ nhất:



Thuật toán Prim



	1	2	3	4	5	6
Khởi tạo	0	oo	oo	oo	oo	oo
1	*	7/1	1/1			
2		5/3	*		2/3	7/3
3		2/5		5/5	*	3/5
4		*		4/2		1/2
5						*