

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Thuật toán ứng dụng

Nguyễn Khánh Phương

**Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn**

Nội dung khóa học

Chương 1. Các cấu trúc dữ liệu và thư viện

Chương 2. Kỹ thuật đệ quy và nhánh cận

Chương 3. Chia để trị

Chương 4. Quy hoạch động

Chương 5. Các thuật toán trên đồ thị và ứng dụng

Chương 6. Các thuật toán xử lý xâu và ứng dụng

Chương 7. Lớp bài toán NP-đầy đủ

Nội dung

1. Đồ thị và cách biểu diễn đồ thị
2. Duyệt đồ thị
- 3. Một số bài toán cơ bản trên đồ thị**

3. Một số bài toán cơ bản trên đồ thị

3.1. Bài toán cây khung nhỏ nhất

3.2. Bài toán đường đi ngắn nhất

3.3. Bài toán phủ đỉnh

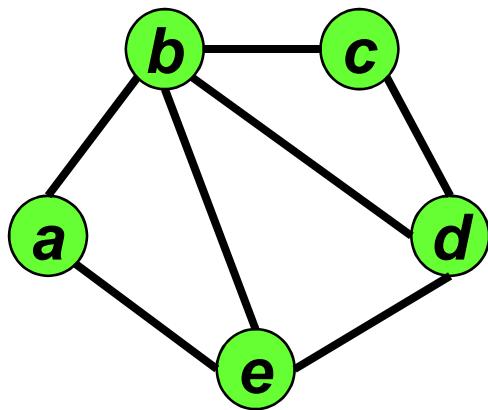
3.4. Bài toán tập độc lập

3.5. Bài toán ghép cặp

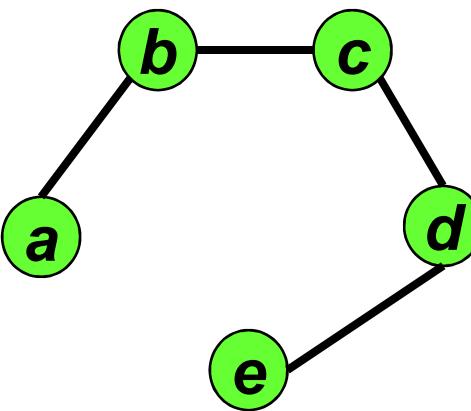
3.6. Bài toán tô màu đồ thị

3.1. Bài toán cây khung nhỏ nhất

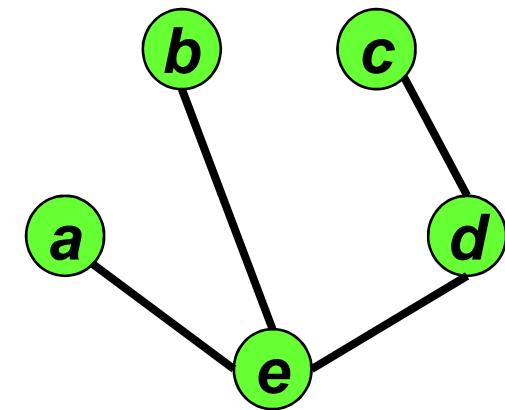
- Cho đồ thị $G=(V,E)$ là đồ thị vô hướng liên thông có trọng số.
- Cây $T=(V,F)$ với $F \subseteq E$ được gọi là cây khung của đồ thị G .
vô hướng liên thông không có chu trình



G



Cây khung T_1



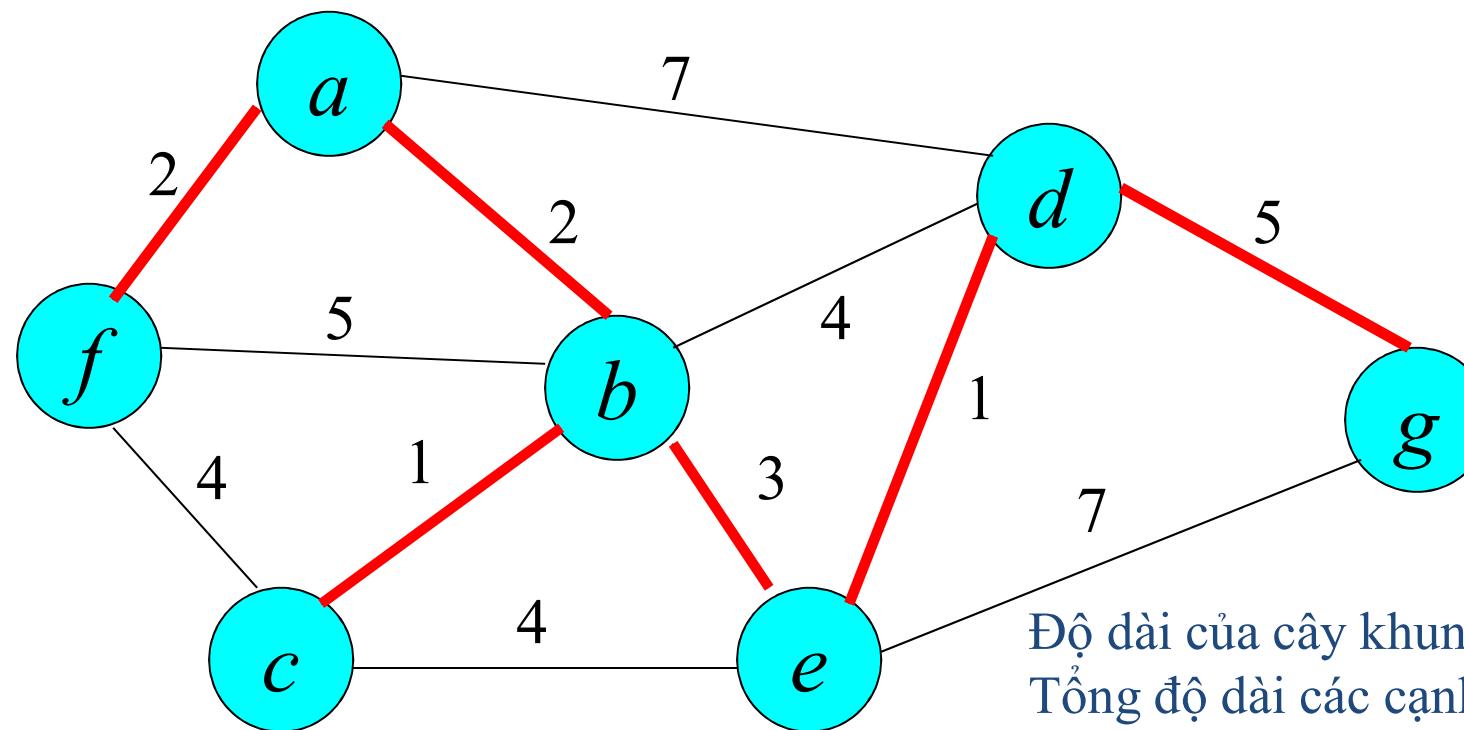
Cây khung T_2

Đồ thị G và 2 cây khung T_1 và T_2 của nó

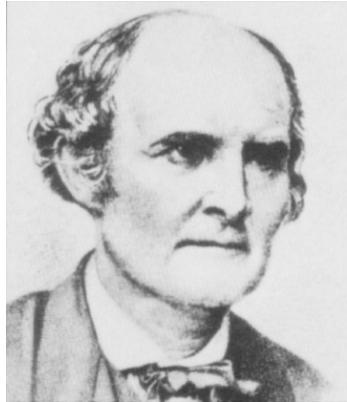
Độ dài của cây khung là tổng trọng số trên các cạnh của nó. Cần tìm cây khung có độ dài nhỏ nhất.

3.1. Bài toán cây khung nhỏ nhất

Bài toán: Cho đồ thị vô hướng $G=(V,E)$ với trọng số $c(e)$, $e \in E$. Độ dài của cây khung là tổng trọng số trên các cạnh của nó. Cần tìm cây khung có độ dài nhỏ nhất.



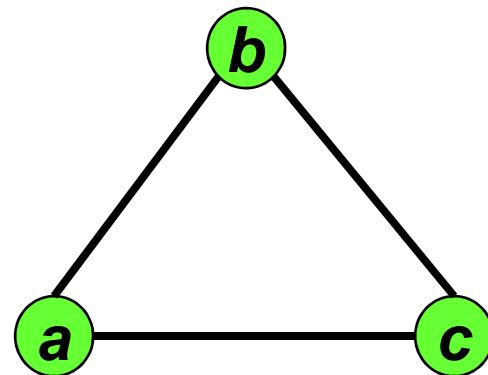
Số lượng cây khung của đồ thị



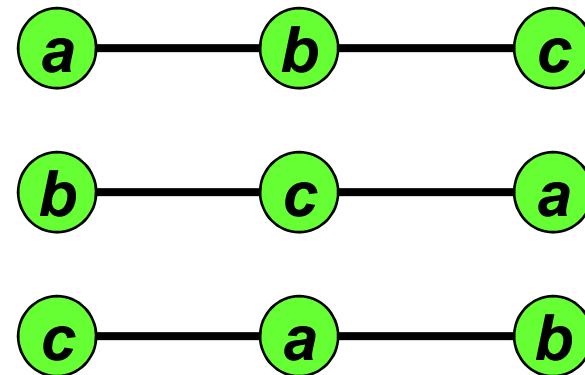
Arthur Cayley
(1821 – 1895)

Định lý sau đây cho biết số lượng cây khung của đồ thị đầy đủ K_n :

- Định lý Cayley. Số cây khung của đồ thị K_n là n^{n-2} .



K_3



Ba cây khung của K_3

Do số lượng cây khung của G là rất lớn (xem định lý Cayley), nên không thể giải nhò duyệt toàn bộ

3.1. Bài toán cây khung nhỏ nhất

Bài toán: Cho đồ thị vô hướng $G=(V,E)$ với trọng số $c(e)$, $e \in E$. Độ dài của cây khung là tổng trọng số trên các cạnh của nó. Cần tìm cây khung có độ dài nhỏ nhất.

- Thuật toán Kruskal:
 - Sử dụng DFS: $O(V^* (V+E))$
 - Sử dụng cấu trúc dữ liệu các tập không giao nhau: $O(E \log_2 E)$
- Thuật toán PRIM:
 - Cài đặt bình thường: $O(V^2)$
 - Sử dụng cấu trúc hàng đợi có ưu tiên: $O((V+E)^* \log V)$

Thuật toán Kruskal

- Thuật toán sẽ xây dựng tập cạnh E_T của cây khung nhỏ nhất $T = (V, E_T)$ theo từng bước.
- Bắt đầu từ tập $E_T = \emptyset$, ở mỗi bước ta sẽ lần lượt duyệt trong danh sách cạnh đã sắp xếp, từ cạnh có độ dài nhỏ đến cạnh có độ dài lớn hơn, để tìm ra cạnh mà việc bổ sung nó vào tập E_T không tạo thành chu trình trong tập này.
- Thuật toán sẽ kết thúc khi ta thu được tập E_T gồm $n-1$ cạnh.

Kruskal_Algorithm

$E_T := \emptyset;$

while $|E_T| < (n-1)$ and ($E \neq \emptyset$) **do**

{

Chọn e là cạnh có độ dài nhỏ nhất trong E;

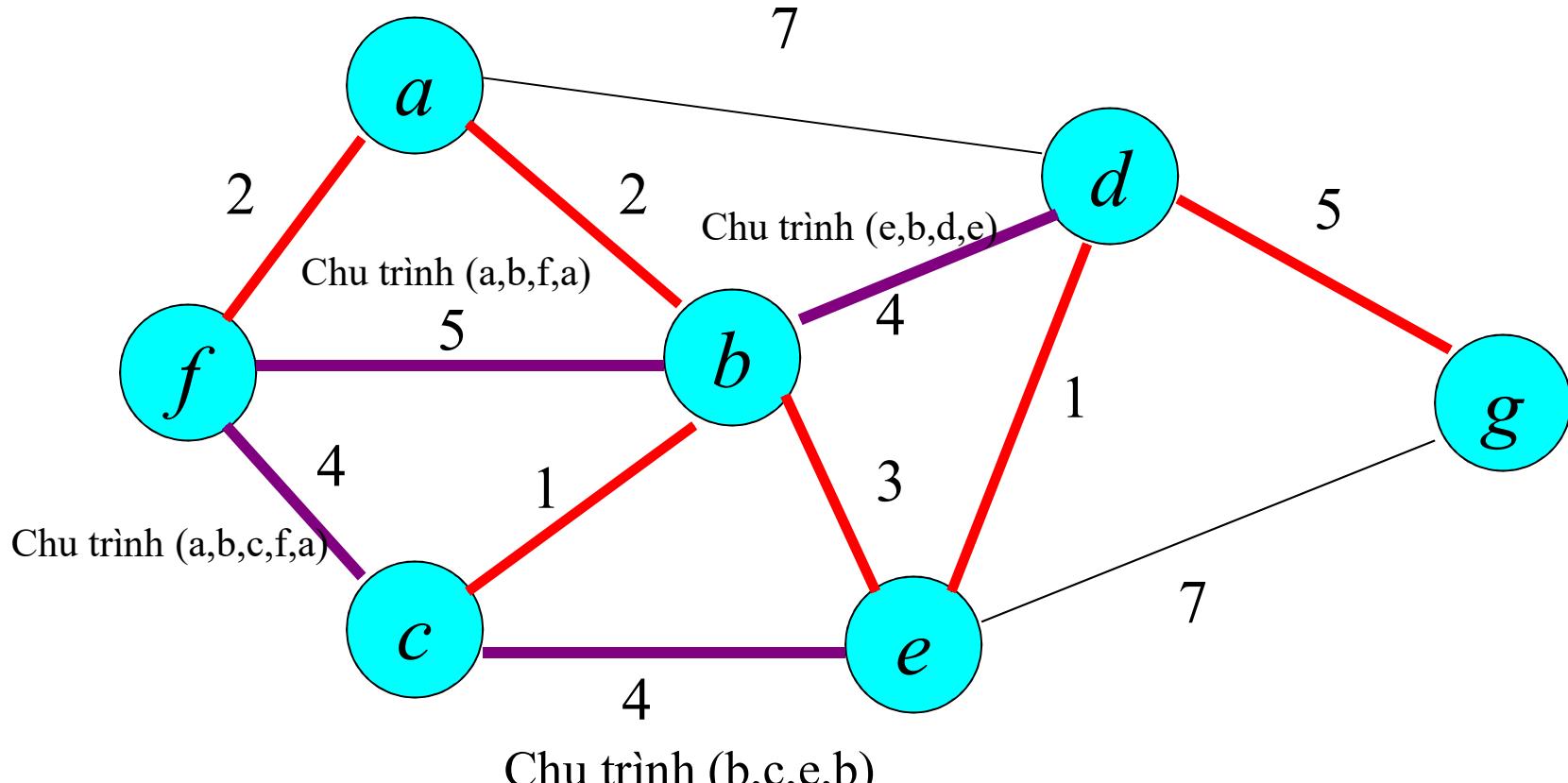
$E := E \setminus \{e\};$

if ($E_T \cup \{e\}$ không chứa chu trình) **then** $E_T := E_T \cup \{e\};$

}

if ($|E_T| < n-1$) **then** *Đồ thị không liên thông;*

Thuật toán Kruskal – Ví dụ



Cài đặt thuật toán Kruskal

- Có 2 thao tác đòi hỏi nhiều tính toán nhất trong 1 bước lặp của thuật toán Kruskal:
 - Chọn e là cạnh có độ dài nhỏ nhất trong E ;
 - Kiểm tra xem tập cạnh $E_T \cup \{e\}$ có chứa chu trình hay không?

Kruskal_Algorithm

$E_T := \emptyset;$

while $|E_T| < (n-1)$ and ($E \neq \emptyset$) **do**

{

Chọn e là cạnh có độ dài nhỏ nhất trong E ;

$E := E \setminus \{e\};$

if ($E_T \cup \{e\}$ không chứa chu trình) **then** $E_T := E_T \cup \{e\};$

}

if ($|E_T| < n-1$) **then** *Đồ thị không liên thông;*

Thao tác 1: Chọn e là cạnh có độ dài nhỏ nhất trong E

- Cách 1: Ta sẽ thực hiện trước việc sắp xếp các cạnh của đồ thị theo thứ tự tăng dần của độ dài. Đối với đồ thị có m cạnh, bước này đòi hỏi thời gian $O(m \log m)$. Khi đó, trong bước lặp việc chọn cạnh có độ dài nhỏ nhất đòi hỏi thời gian $O(1)$.
- Cách 2: Tuy nhiên, để xây dựng cây khung nhỏ nhất với $n-1$ cạnh, nói chung, thường ta chỉ phải xét $p < m$ cạnh. Do đó thay vì sắp xếp toàn bộ dãy cạnh ta sẽ sử dụng min-heap:
 - Để tạo đống đầu tiên ta mất thời gian $O(m)$,
 - Việc vun lại đống sau khi lấy ra phần tử nhỏ nhất ở gốc đòi hỏi thời gian $O(\log m)$

→ Suy ra thuật toán sẽ đòi hỏi thời gian $O(m + p \log m)$ cho việc sắp xếp các cạnh.

Trong việc giải các bài toán thực tế, số p thường nhỏ hơn rất nhiều so với m .

Thao tác 2: Kiểm tra: Tập $E_T \cup \{e\}$ có chứa chu trình hay không?

Ký hiệu $E^* = E_T \cup \{e = (j, k)\}$

- **Cách 1:** Thao tác 2 có thể thực hiện nhờ sử dụng thuật toán kiểm tra xem đồ thị $G^* = (V, E^*)$ có chứa chu trình hay không bằng cách sử dụng DFS trên G^* với thời gian cần thiết là $O(n+m)$.

Cần thực hiện tổng cộng n lần thao tác 2 nên thời gian là: **O($n^* (n+m)$)**

- **Cách 2:** Khi cạnh $e = (j, k)$ được xét, ta cần biết có phải j và k thuộc hai **thành phần liên thông (tplt)** khác nhau hay không. Nếu đúng, thì cạnh này được bổ sung vào cây khung vì nó sẽ nối tplt chứa j và tplt chứa k . Nếu không, thì bổ sung (j, k) sẽ tạo nên chu trình.

Thời gian tổng cộng: **O($m \log_2 m$)**

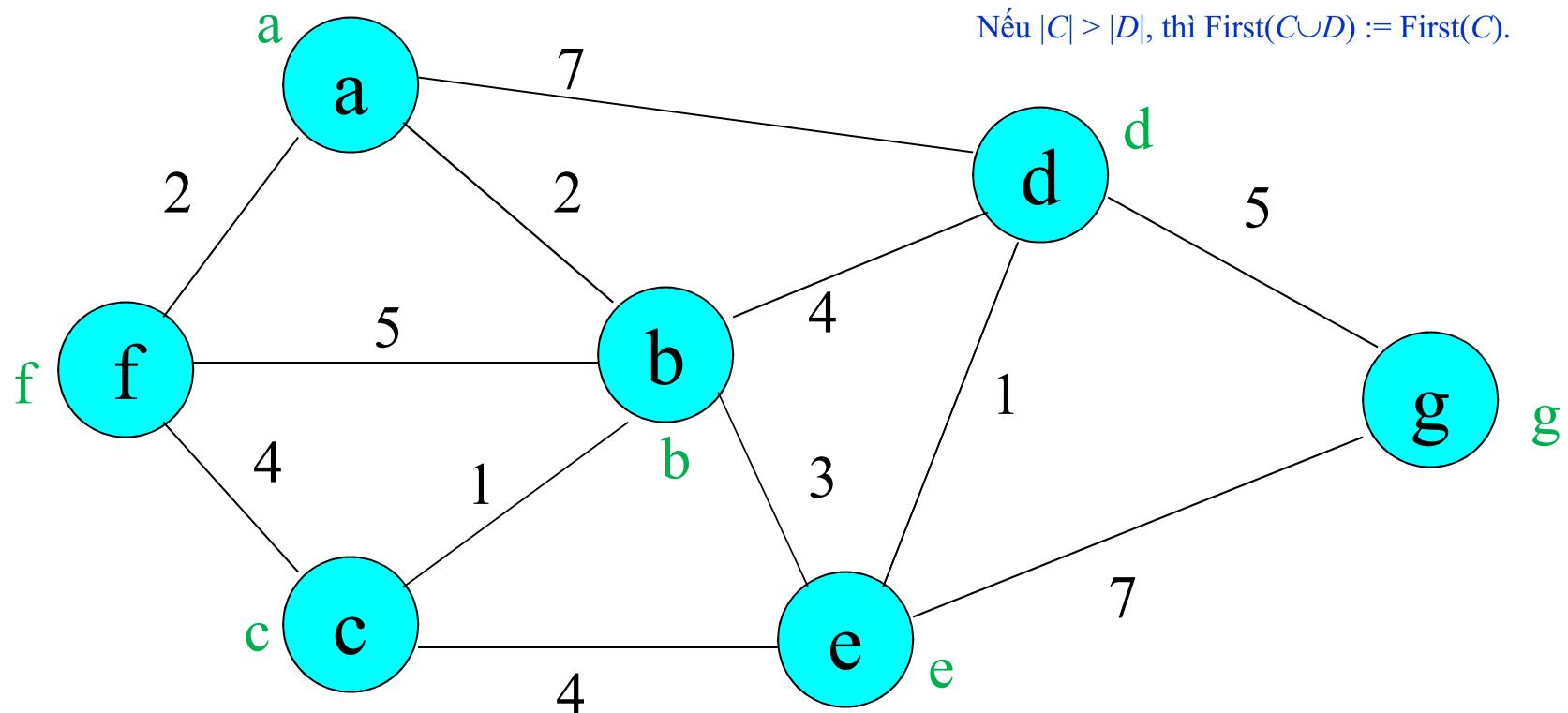
Để thực hiện điều này một cách hiệu quả, ta làm như sau:

- Mỗi tplt C của rừng F được cất giữ như một tập. Ký hiệu First(C) đỉnh đầu tiên trong tplt C .
- Với mỗi đỉnh j trong tplt C , đặt First(j) = First(C) = đỉnh đầu tiên trong C .
- Chú ý: Thêm cạnh (i, j) vào rừng F tạo thành chu trình iff i và j thuộc cùng một tplt, tức là First(i) = First(j).
- Khi nối tplt C và D , sẽ nối tplt **nhỏ hơn** (ít đỉnh hơn) vào tplt **lớn hơn** (nhiều đỉnh hơn): Nếu $|C| > |D|$: thì First($C \cup D$) := First(C).

Thuật toán Kruskal – Ví dụ

- Mỗi tplt C của rừng F được cất giữ như một tập.
- Ký hiệu $\text{First}(C)$ đỉnh đầu tiên trong tplt C .
- Với mỗi đỉnh j trong tplt C , đặt $\text{First}(j) = \text{First}(C) =$ đỉnh đầu tiên trong C .
- Chú ý: Thêm cạnh (i,j) vào rừng F tạo thành chu trình iff i và j thuộc cùng một tplt, tức là $\text{First}(i) = \text{First}(j)$.
- Khi nối tplt C và D , sẽ nối tplt **nhỏ hơn** (ít đỉnh hơn) vào tplt **lớn hơn** (nhiều đỉnh hơn):

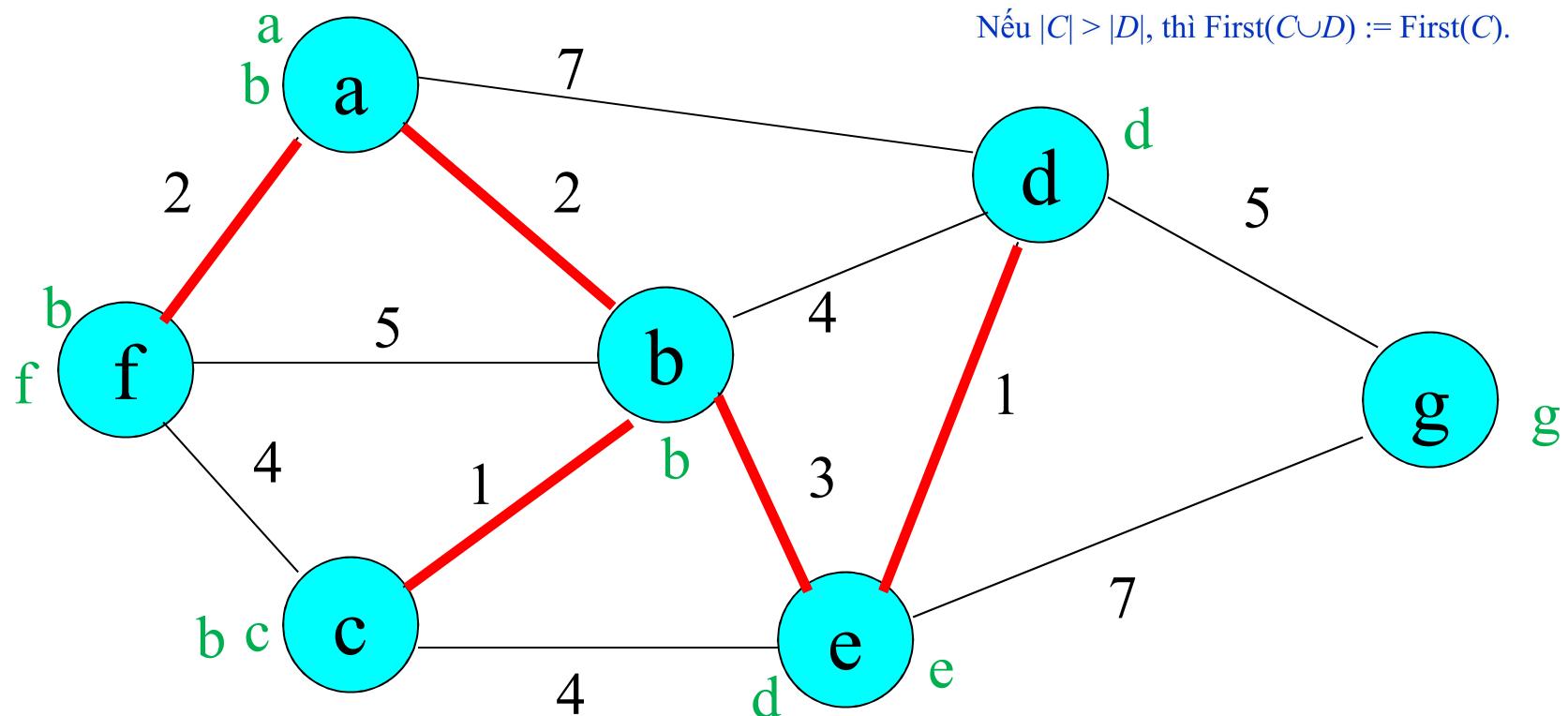
Nếu $|C| > |D|$, thì $\text{First}(C \cup D) := \text{First}(C)$.



Thuật toán Kruskal – Ví dụ

- Mỗi tplt C của rừng F được cất giữ như một tập.
- Ký hiệu $\text{First}(C)$ đỉnh đầu tiên trong tplt C .
- Với mỗi đỉnh j trong tplt C , đặt $\text{First}(j) = \text{First}(C) =$ đỉnh đầu tiên trong C .
- Chú ý: Thêm cạnh (i,j) vào rừng F tạo thành chu trình iff i và j thuộc cùng một tplt, tức là $\text{First}(i) = \text{First}(j)$.
- Khi nối tplt C và D , sẽ nối tplt **nhỏ hơn** (ít đỉnh hơn) vào tplt **lớn hơn** (nhiều đỉnh hơn):

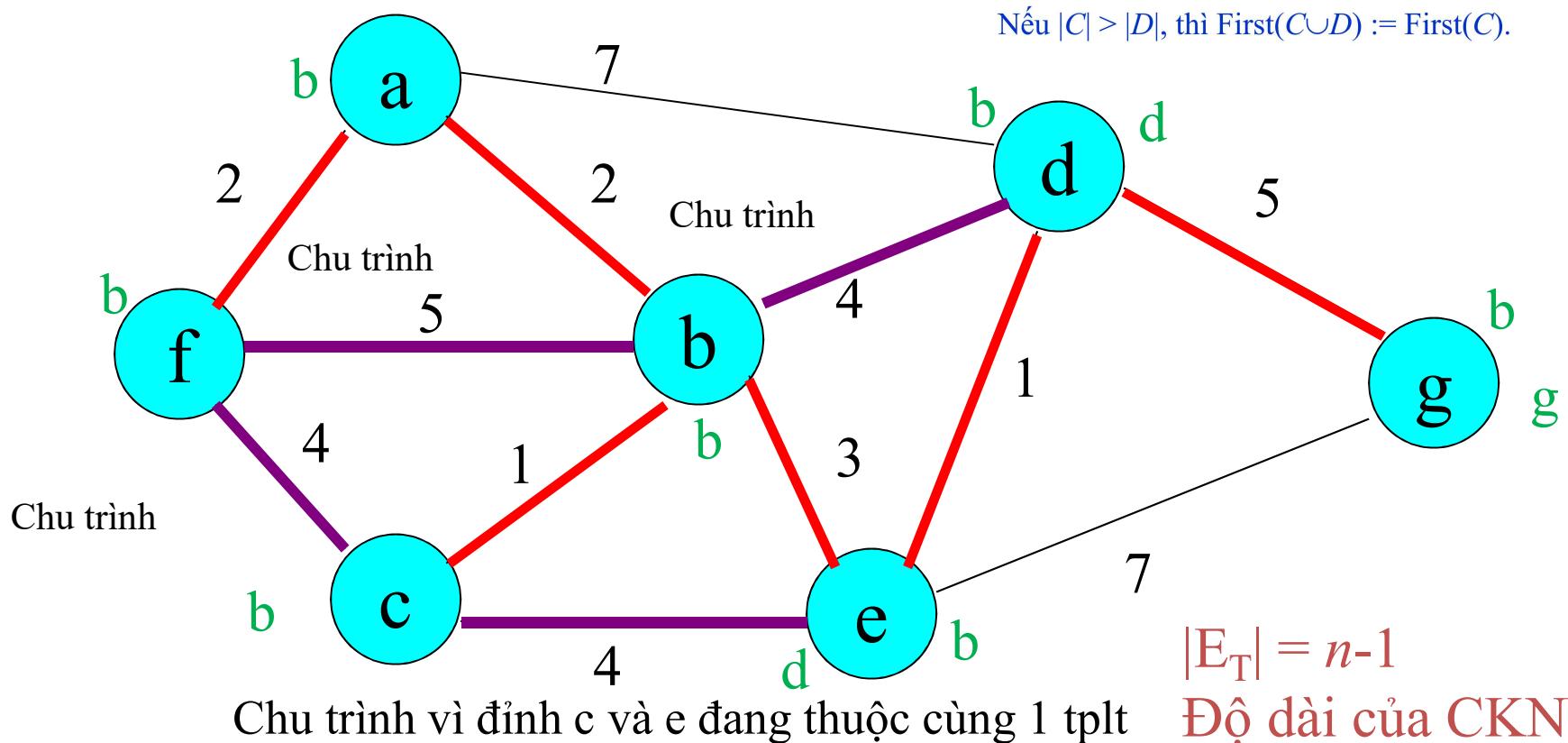
Nếu $|C| > |D|$, thì $\text{First}(C \cup D) := \text{First}(C)$.



Thuật toán Kruskal – Ví dụ

- Mỗi tplt C của rừng F được cất giữ như một tập.
- Ký hiệu $\text{First}(C)$ đỉnh đầu tiên trong tplt C .
- Với mỗi đỉnh j trong tplt C , đặt $\text{First}(j) = \text{First}(C) =$ đỉnh đầu tiên trong C .
- Chú ý: Thêm cạnh (i,j) vào rừng F tạo thành chu trình iff i và j thuộc cùng một tplt, tức là $\text{First}(i) = \text{First}(j)$.
- Khi nối tplt C và D , sẽ nối tplt **nhỏ hơn** (ít đỉnh hơn) vào tplt **lớn hơn** (nhiều đỉnh hơn):

Nếu $|C| > |D|$, thì $\text{First}(C \cup D) := \text{First}(C)$.



Cấu trúc dữ liệu các tập không giao nhau cho thuật toán Kruskal

- Thoát tiên, E_T là rỗng. Có $|V|$ tplt, mỗi thành phần gồm 1 đỉnh.

1 3 5 7

2 4 6 8

- Các tập khởi tạo là:
 - $\{1\} \ {2\} \ {3\} \ {4\} \ {5\} \ {6\} \ {7\} \ {8\}$
- Nếu việc bổ sung cạnh (i, j) vào E_T không tạo thành chu trình thì cạnh này được bổ sung và E_T .

$r_1 = \text{find}(i); r_2 = \text{find}(j); // r_1 (r_2) – tên của tập con chứa i (j)$

if ($r_1 \neq r_2$) **then**

$$E_T = E_T \cup (i, j);$$

$\text{union}(r_1, r_2); // Nối hai tập con } r_1, r_2$

Cấu trúc dữ liệu các tập không giao nhau (Disjoint-set Data Structures)

Vấn đề đặt ra là: Cho tập V gồm n phần tử, ta cần xây dựng cấu trúc dữ liệu biểu diễn phân hoạch tập V ra thành các tập con V_1, V_2, \dots, V_k hỗ trợ thực hiện hiệu quả các thao tác sau:

- **Makeset(x):** Tạo một tập con chứa duy nhất phần tử x .
- **Union(x, y):** Thay thế các tập V_i và V_j (trong đó $x \in V_i$ và $y \in V_j$) bởi tập $V_i \cup V_j$ trong phân hoạch đang xét.
- **Find(x):** Tìm tên của tập chứa phần tử x .

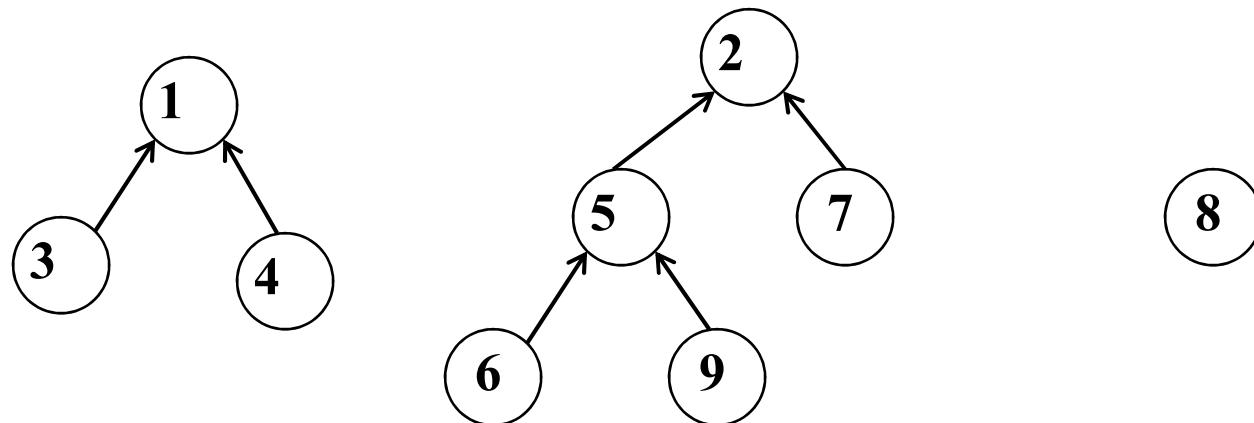
Như vậy, $\text{Find}(x)$ và $\text{Find}(y)$ trả lại cùng một giá trị khi và chỉ khi x và y thuộc cùng một tập con trong phân hoạch.

Cấu trúc dữ liệu đáp ứng yêu cầu này có tên là **cấu trúc dữ liệu Union-Find** (hoặc Disjoint-set data structure).

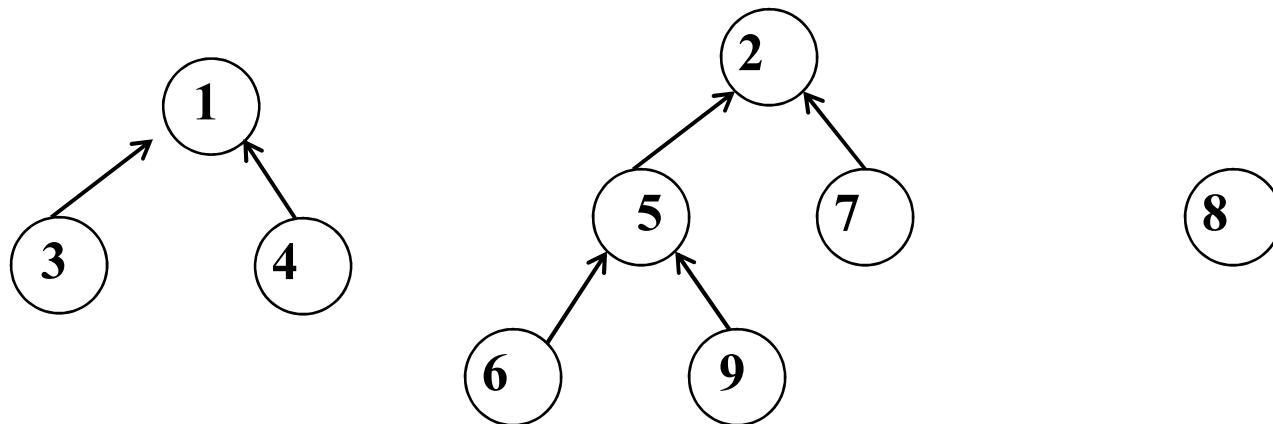
Cấu trúc dữ liệu các tập không giao nhau (Disjoint-set Data Structures)

- Trước hết để biểu diễn mỗi tập con $X \subset V$, chúng ta sẽ sử dụng cấu trúc cây có gốc: Chọn một phần tử nào đó của X làm gốc (tên của tập con X chính là phần tử tương ứng với gốc), mỗi phần tử $x \in X$ sẽ có một biến trả parent[x] trỏ đến cha của nó, nếu x là gốc thì parent[x] = x.

Ví dụ: Giả sử có $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. $V_1 = \{1, 3, 4\}$, $V_2 = \{2, 5, 6, 7, 9\}$, $V_3 = \{8\}$. Ta có ba cây mô tả ba tập V_1, V_2, V_3



Cấu trúc dữ liệu các tập không giao nhau (Disjoint-set Data Structures)



Mảng parent để biểu diễn rừng gồm 3 cây tương ứng với V_1, V_2, V_3 :

v	1	2	3	4	5	6	7	8	9
parent[v]	1	2	1	1	2	5	2	8	5

Makeset(x) và FindSet

MakeSet(x)

– **Makeset(x):** Tạo một tập con chứa duy nhất phần tử x .

{

 parent[x] := x;

}

Thời gian: $O(1)$.

Find(x)

– **Find(x):** Tìm tên của tập chứa phần tử x .

{

(tên của tập V_i chính là phần tử tương ứng với gốc)

while $x \neq \text{parent}[x]$ **do** $x = \text{parent}[x]$;

return x ;

}

Thời gian: $O(h)$, trong đó h là độ cao của cây chứa x .

Cấu trúc dữ liệu các tập không giao nhau (Disjoint-set Data Structures)

- **Union(x, y):** Thay thế các tập V_i và V_j (trong đó $x \in V_i$ và $y \in V_j$) bởi tập $V_i \cup V_j$ trong phân hoạch đang xét.

Để nối tập con chứa x và tập con chứa y chúng ta có thể chia lại biến trả của gốc của cây chứa x để cho nó trở đến gốc của cây con chứa y . Điều đó được thực hiện nhờ thủ tục sau:

Union(x, y) {

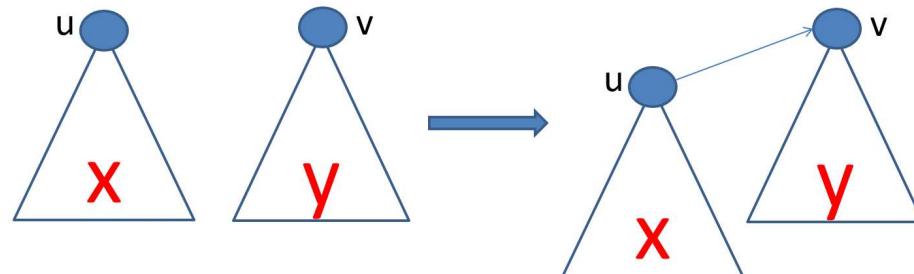
$u := \text{Find}(x);$ (* Tìm u là gốc của cây con chứa x *)

$v := \text{Find}(y);$ (* Tìm v gốc của cây con chứa y *)

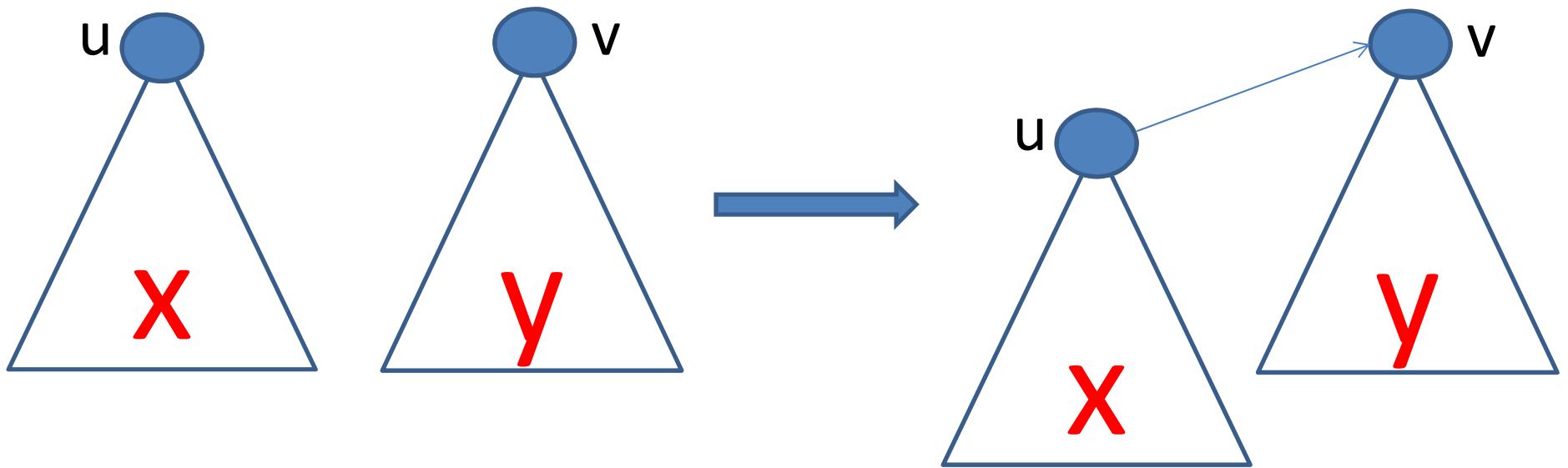
$\text{parent}[u] := v;$

}

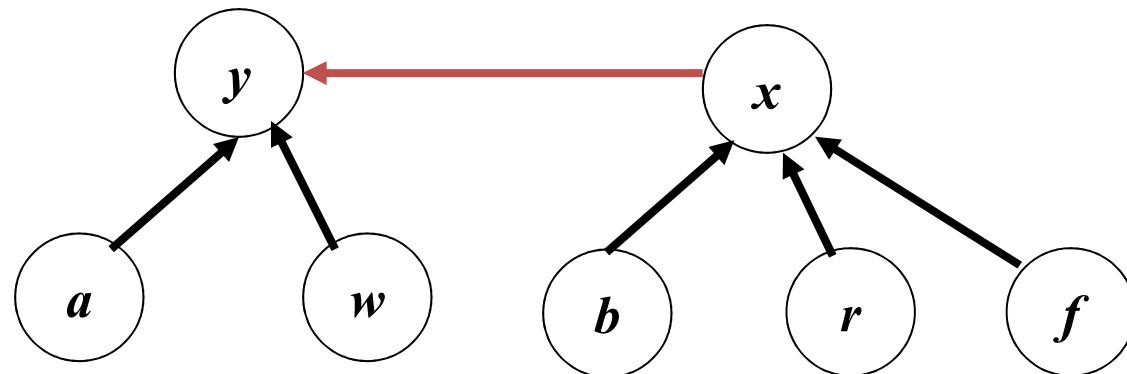
Thời gian: $O(h)$



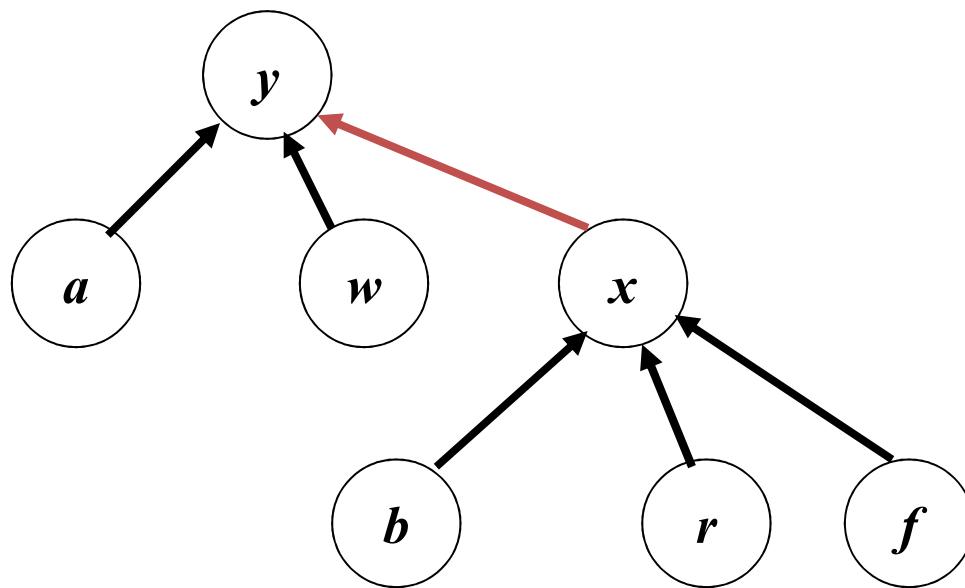
Ví dụ Union(x,y)



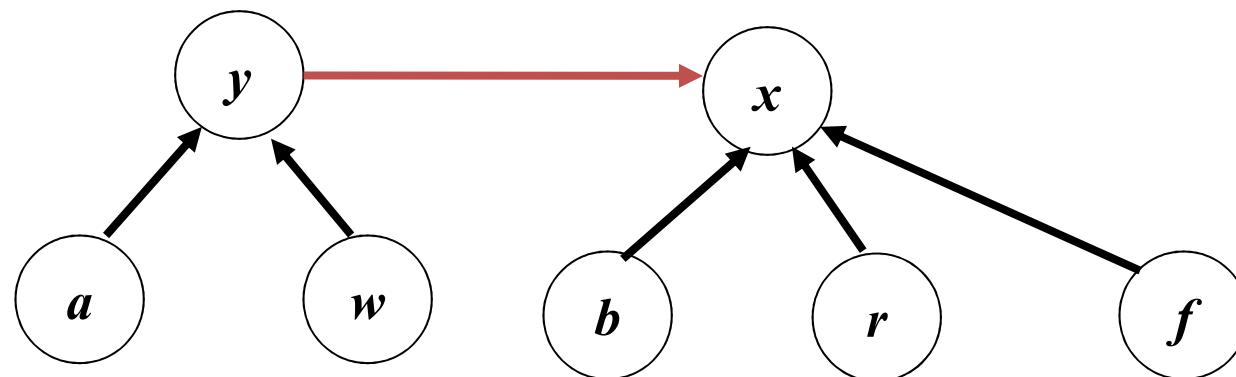
Ví dụ Union(x,y)



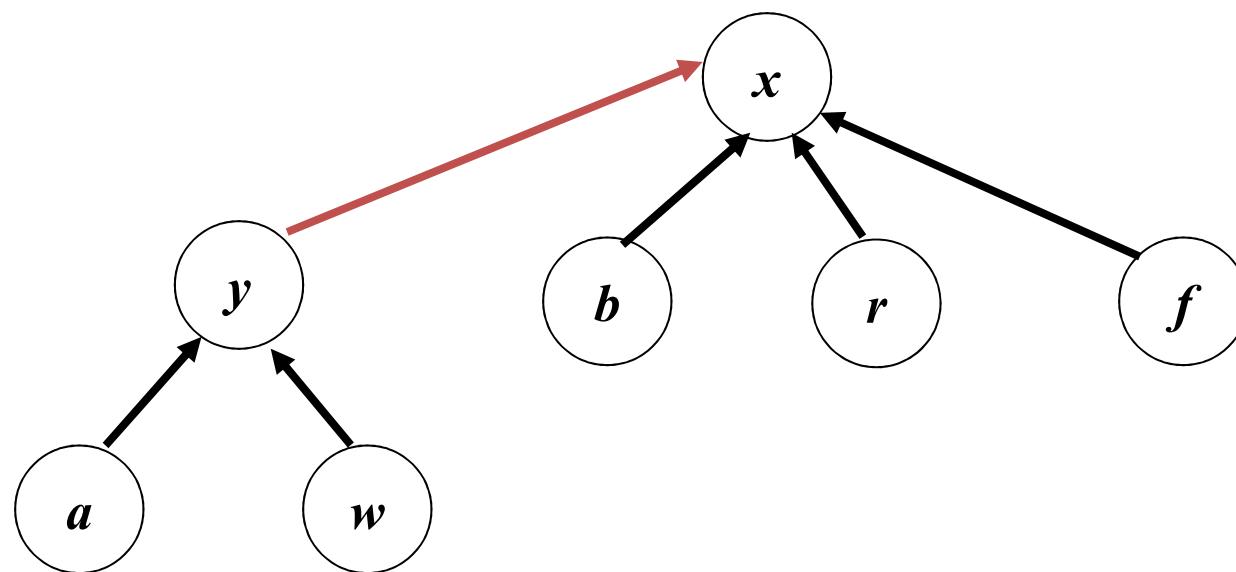
↓
 x trỏ đến y
 b, r và f chìm xuống sâu hơn



Ví dụ Union(y, x)



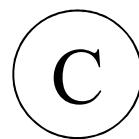
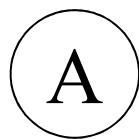
y trở đến x
 a và w chìm xuống sâu hơn



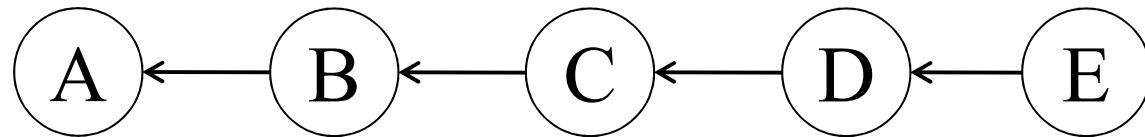
Phân tích độ phức tạp

Có thể thấy thời gian tính của hàm Find(x) phụ thuộc vào độ cao của cây chứa x.

Ví dụ:



- Sau khi thực hiện Union(A,B); Union(B,C); Union(C,D); Union(D,E) có thể thu được cây:



Trong trường hợp cây có k đỉnh và có dạng như một đường đi thì độ cao của cây sẽ là $k - 1$.

Do đó hàm Find(x) có đánh giá thời gian tính là $O(n)$.

Cấu trúc dữ liệu các tập không giao nhau

Liệu có cách nào để giảm độ cao của các cây con?

- Có một cách thực hiện rất đơn giản: Khi nối hai cây chúng ta sẽ điều chỉnh con trỏ của gốc của cây con có ít đỉnh hơn, chứ không thực hiện việc nối một cách tuỳ tiện.
- Để ghi nhận số phần tử của một cây chúng ta sẽ sử dụng thêm biến Num [v] chứa số phần tử của cây con với gốc tại v.

MAKESET và Union cải tiến

```
MAKESET(x) {  
    parent[x] := x;  
    Num[x]:=1;  
}
```

```
MakeSet(x)  
{  
    parent[x] := x;  
}
```

– **Makeset(x):** Tạo một tập con chứa duy nhất phần tử x.

Thời gian: $O(1)$.

```
Union(x, y){  
    u:= Find(x); (* Tìm u là gốc của cây con chứa x *)  
    v:= Find(y); (* Tìm v là gốc của cây con chứa y *)  
    if Num[u] <= Num[v] { //gán u vào v  
        parent[u] := v;  Num[v]:= Num[u]+Num[v];  
    } else {  
        parent[v] := u;  Num[u]:= Num[u]+Num[v];  
    }  
}
```

```
}
```

```
Union(x, y) {
```

```
    u:= Find(x); (* Tìm u là gốc của cây con chứa x *)  
    v:= Find(y); (* Tìm v là gốc của cây con chứa y *)  
    parent[u] := v;
```

Cấu trúc dữ liệu các tập không giao nhau

- **Bổ đề.** Giả sử quá trình thực hiện nối cây bắt đầu từ các cây chỉ có 1 đỉnh. Khi đó độ cao của các cây xuất hiện khi thực hiện thủ tục nối không vượt quá $\log n$.

CM. Qui nạp theo số đỉnh của cây.

Từ bổ đề trên, suy ra các thao tác **Find** và **Union** được thực hiện với thời gian $O(\log n)$ nhờ sử dụng cách nối cây cải tiến.

[Trước cải tiến: $O(h)$: với h là chiều cao cây; $h \sim n$]

Thuật toán Kruskal sử dụng cấu trúc dữ liệu Union-Find

Kruskal(G, w)

1. $E_T \leftarrow \emptyset$
2. **for** $v \in V$ **do**
 3. Make-Set(v)
 4. Sắp xếp các cạnh trong E theo thứ tự không giảm của trọng số
 5. **for** $(u, v) \in E$ **do**
 6. **if** $\text{Find}(u) \neq \text{Find}(v)$ **then**
 7. $E_T \leftarrow E_T \cup \{(u, v)\}$
 8. Union(u, v)
 9. **return** E_T

Kruskal_Algorithm

```
ET := ∅;  
while |ET| < (n-1) and ( E ≠ ∅ ) do  
{  
    Chọn e là cạnh có độ dài nhỏ nhất trong E;  
    E := E \ {e};  
    if ( ET ∪ {e} không chứa chu trình ) then ET := ET ∪ {e};  
}  
if ( |ET| < n-1 ) then Đồ thị không liên thông;
```

Phân tích thời gian tính

Thuật toán Kruskal sử dụng cấu trúc dữ liệu Union-Find

- Dòng 1-3 (khởi tạo): $O(n)$
 - Dòng 4 (sắp xếp): $O(m \log_2 m)$
 - Dòng 6-8 (các thao tác với phân hoạch): $O(\log_2 m)$
- ➔ Dòng 5-8: $O(m \log_2 m)$

Tổng cộng: **$O(m \log_2 m)$**

Kruskal(G, w)

1. $E_T \leftarrow \emptyset$
2. **for** $v \in V$ **do**
3. Make-Set(v)
4. Sắp xếp các cạnh trong E theo thứ tự không giảm của trọng số
5. **for** $(u, v) \in E$ **do**
6. **if** $\text{Find}(u) \neq \text{Find}(v)$ **then**
7. $E_T \leftarrow E_T \cup \{(u, v)\}$
8. Union(u, v)
9. **return** E_T

Cài đặt thuật toán Kruskal dùng cấu trúc các tập không giao nhau

```
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

typedef pair<int, int> iPair;

// To represent Disjoint Sets
struct DisjointSets
{
    int *parent, *num;
    int n;
    DisjointSets(int n)
    {
        this->n = n;
        parent = new int[n+1];
        num = new int[n+1];
        //Khai tao: Makeset
        for (int i = 0; i <= n; i++)
        {
            num[i] = 1;
            //every element is parent of itself
            parent[i] = i;
        }
    }
}
```

Cài đặt thuật toán Kruskal dùng cấu trúc các tập không giao nhau

```
int Find(int u)
{
    while (u != parent[u]) u = parent[u];
    return u;
}
// Union by num
void Union(int x, int y)
{
    int u = Find(x), v = Find(y);
    /* Nơi cây có ít node vào cây có nhiều node hơn */
    if (num[u] > num[v])
    {
        parent[v] = u; num[u] += num[v];
    }
    else
    {
        parent[u] = v; num[v] += num[u];
    }
};
```

Cài đặt thuật toán Kruskal dùng cấu trúc các tập không giao nhau

```
class Graph
{
    int V, E;
    vector< pair<int, iPair> > edges;

public:
    Graph(int V, int E); // Constructor
    void addEdge(int u, int v, int weight); // Cạnh (u, v) có trọng số weight
    int kruskalMST_DisjointSet();
};

// Constructor
Graph::Graph(int V, int E)
{
    this->V = V;
    this->E = E;
}

// Thêm cạnh (u, v) có trọng số = weight
void Graph::addEdge(int u, int v, int weight)
{
    //đo thị vô hướng
    edges.push_back({weight, {u, v}});
}
```

Cài đặt thuật toán Kruskal dùng cấu trúc các tập không giao nhau

```
/* Functions returns weight of the MST*/
int Graph::kruskalMST_DisjointSet()
{
    int mst_wt = 0; //luu do dai MST
    //Sap xep cac canh theo thu tu khong giam ve trong so
    sort(edges.begin(), edges.end());

    //Khai tao disjoint sets
    DisjointSets ds(V);

    //Duyet qua lan luot tung canh trong day canh da sap xep:
    vector< pair<int, iPair> >::iterator it;
    for (it=edges.begin(); it!=edges.end(); it++)
    {
        int u = it->second.first;
        int v = it->second.second;

        int set_u = ds.Find(u);
        int set_v = ds.Find(v);

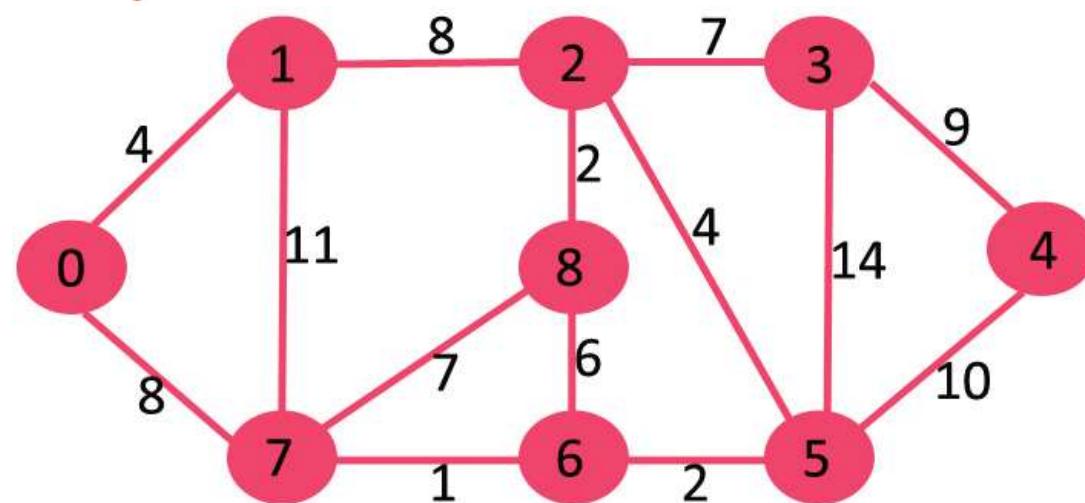
        //Kiem tra 2 dinh u va v co thuoc cung 1 tfilt hay ko:
        if (set_u != set_v)
        {
            cout << "(" << u << ", " << v << "), "; //Bo sung canh (u, v) vao MST
            mst_wt += it->first; // Update do dai MST
            ds.Union(set_u, set_v); //noi 2 tap
        }
    }
    return mst_wt;
}
```

Cài đặt thuật toán Kruskal dùng cấu trúc các tập không giao nhau

```
int main()
{
    int V = 9, E = 14;
    Graph g(V, E);
    g.addEdge(0, 1, 4);   g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);   g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);   g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);   g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);  g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);   g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);   g.addEdge(7, 8, 7);

    cout<<"CAI DAT THUAT TOAN KRUSKAL DUNG DISJOINT SET \n";
    cout<<"- Cay khung nho nhat gom cac canh: ";
    int mst_wt = g.kruskalMST();
    cout << "\n- Do dai cua cay khung nho nhat: " << mst_wt;

    return 0;
}
```

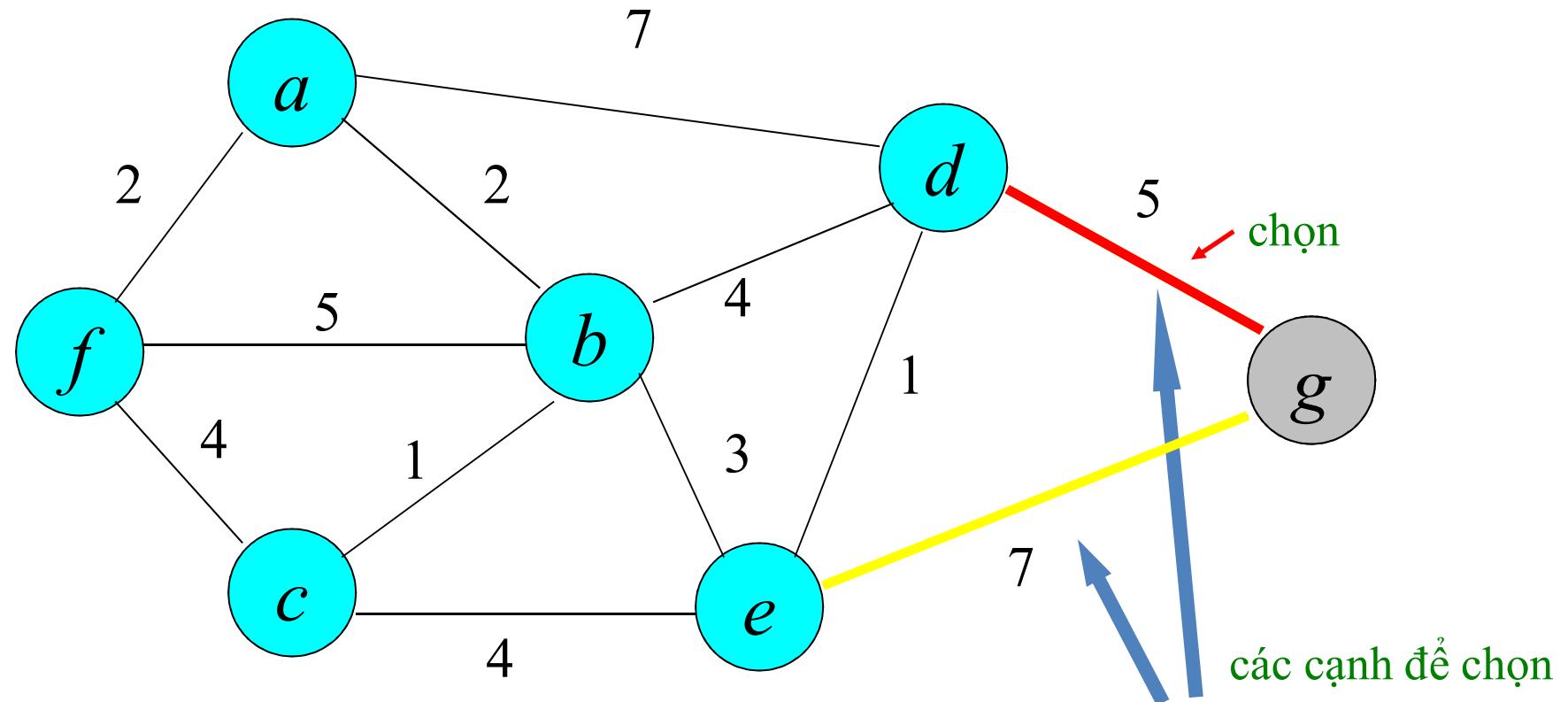


3.1. Bài toán cây khung nhỏ nhất

Bài toán: Cho đồ thị vô hướng $G=(V,E)$ với trọng số $c(e)$, $e \in E$. Độ dài của cây khung là tổng trọng số trên các cạnh của nó. Cần tìm cây khung có độ dài nhỏ nhất.

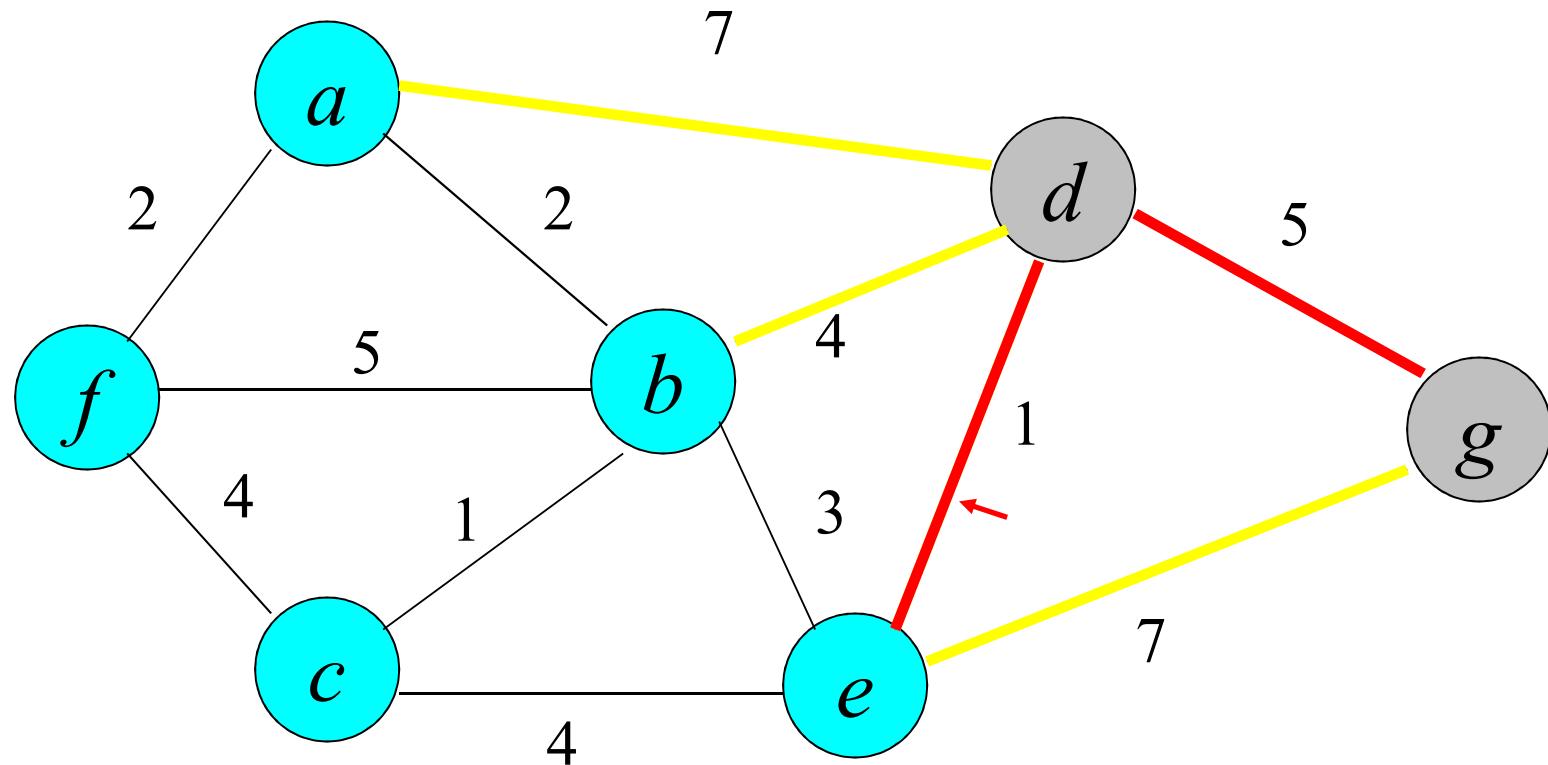
- Thuật toán Kruskal:
 - Sử dụng DFS: $O(V^* (V+E))$
 - Sử dụng cấu trúc dữ liệu các tập không giao nhau: $O(E \log_2 E)$
- Thuật toán PRIM:
 - Cài đặt bình thường: $O(V^2)$
 - Sử dụng cấu trúc hàng đợi có ưu tiên: $O((V+E)^* \log V)$

Thuật toán PRIM



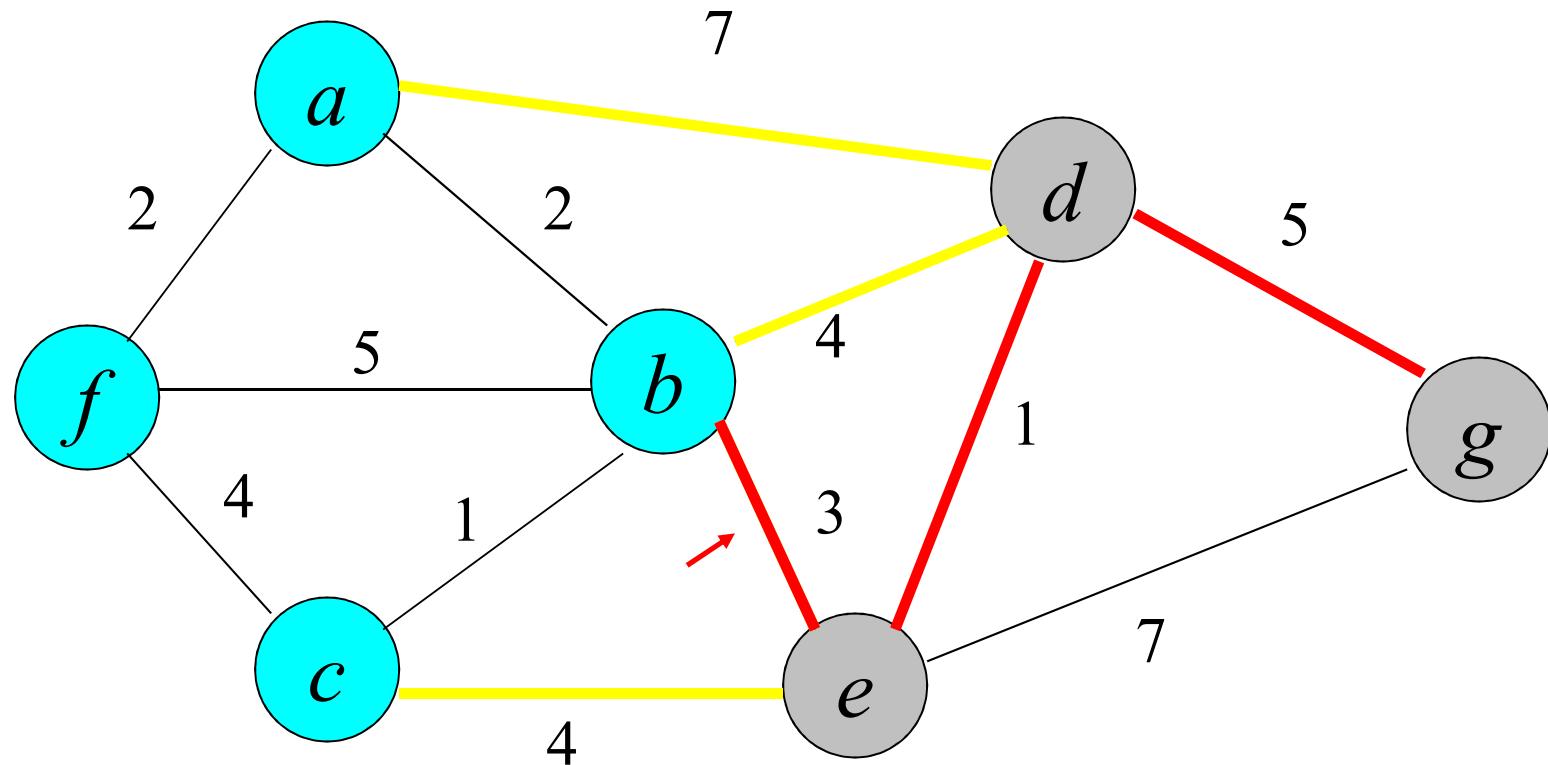
- ★ T là cây (bắt đầu từ cây chỉ có 1 đỉnh)
- ★ Cạnh được bổ sung vào T là cạnh nhẹ nhất trong số các cạnh nối đỉnh trong T với một đỉnh không ở trong T .

Thuật toán PRIM – Ví dụ



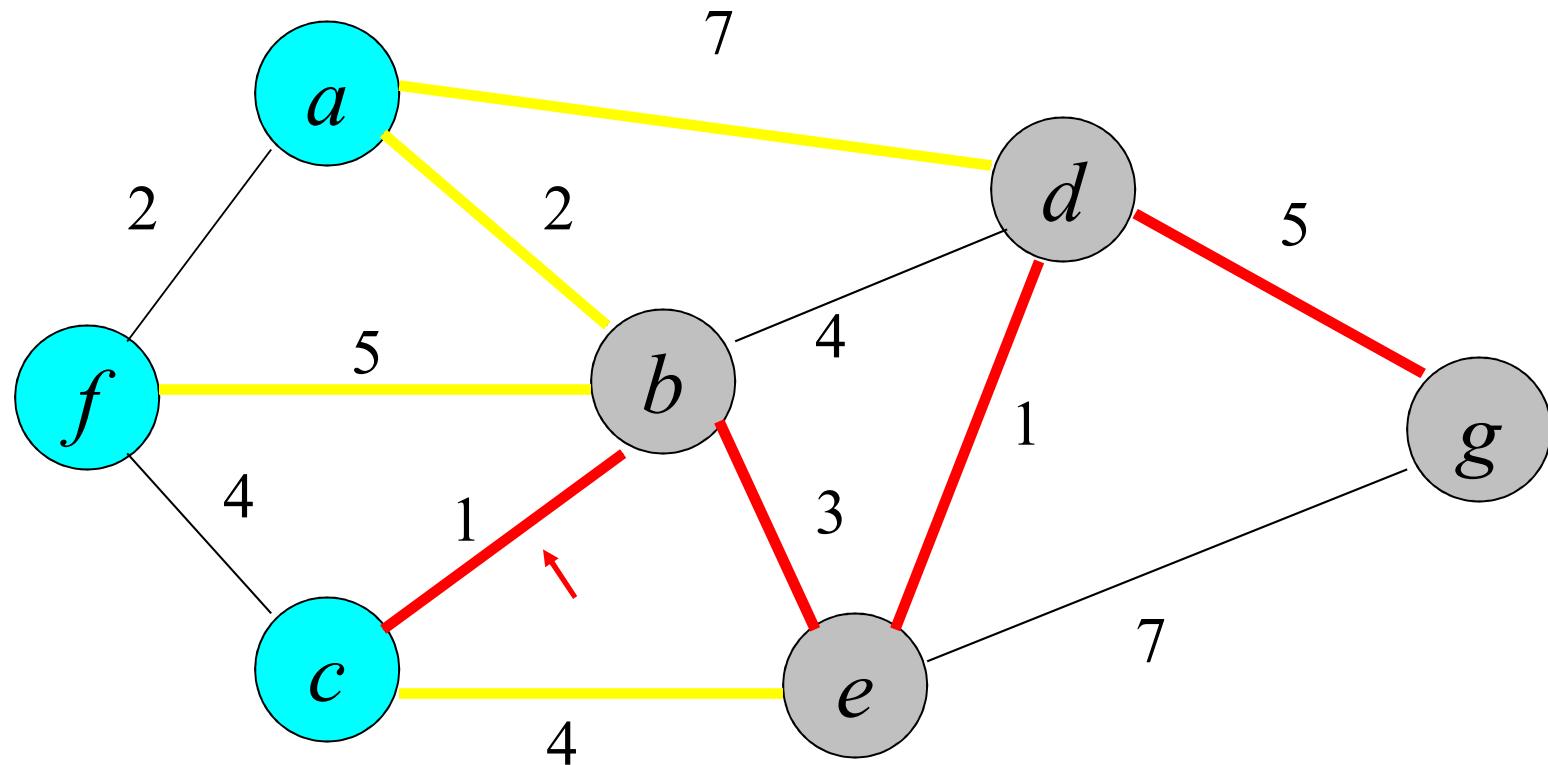
- ★ T là cây (bắt đầu từ cây chỉ có 1 đỉnh)
- ★ Cạnh được bổ sung vào T là cạnh nhẹ nhất trong số các cạnh nối đỉnh trong T với một đỉnh không ở trong T .

Thuật toán PRIM – Ví dụ



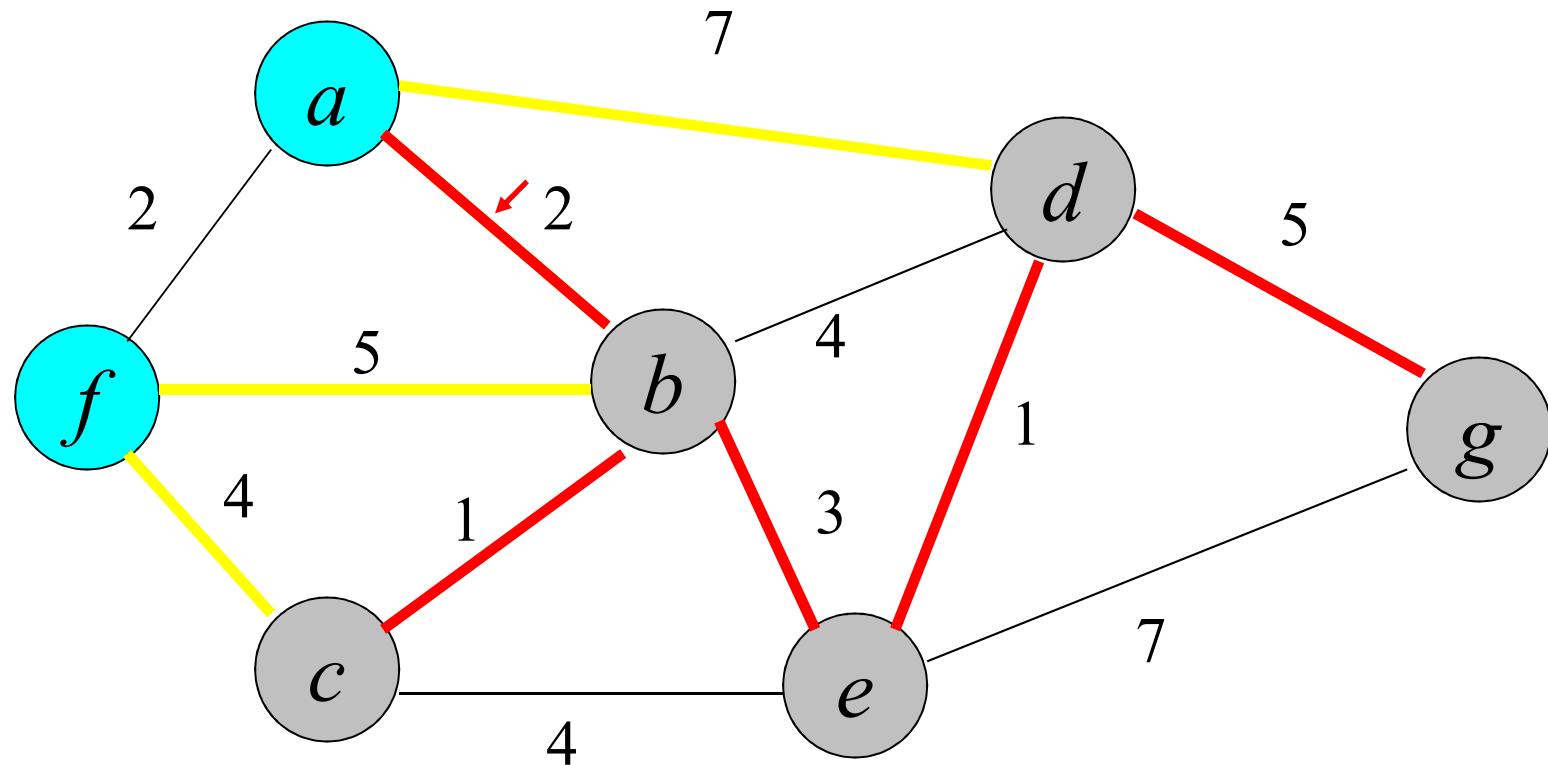
- ★ T là cây (bắt đầu từ cây chỉ có 1 đỉnh)
- ★ Cạnh được bổ sung vào T là cạnh nhẹ nhất trong số các cạnh nối đỉnh trong T với một đỉnh không ở trong T .

Thuật toán PRIM – Ví dụ



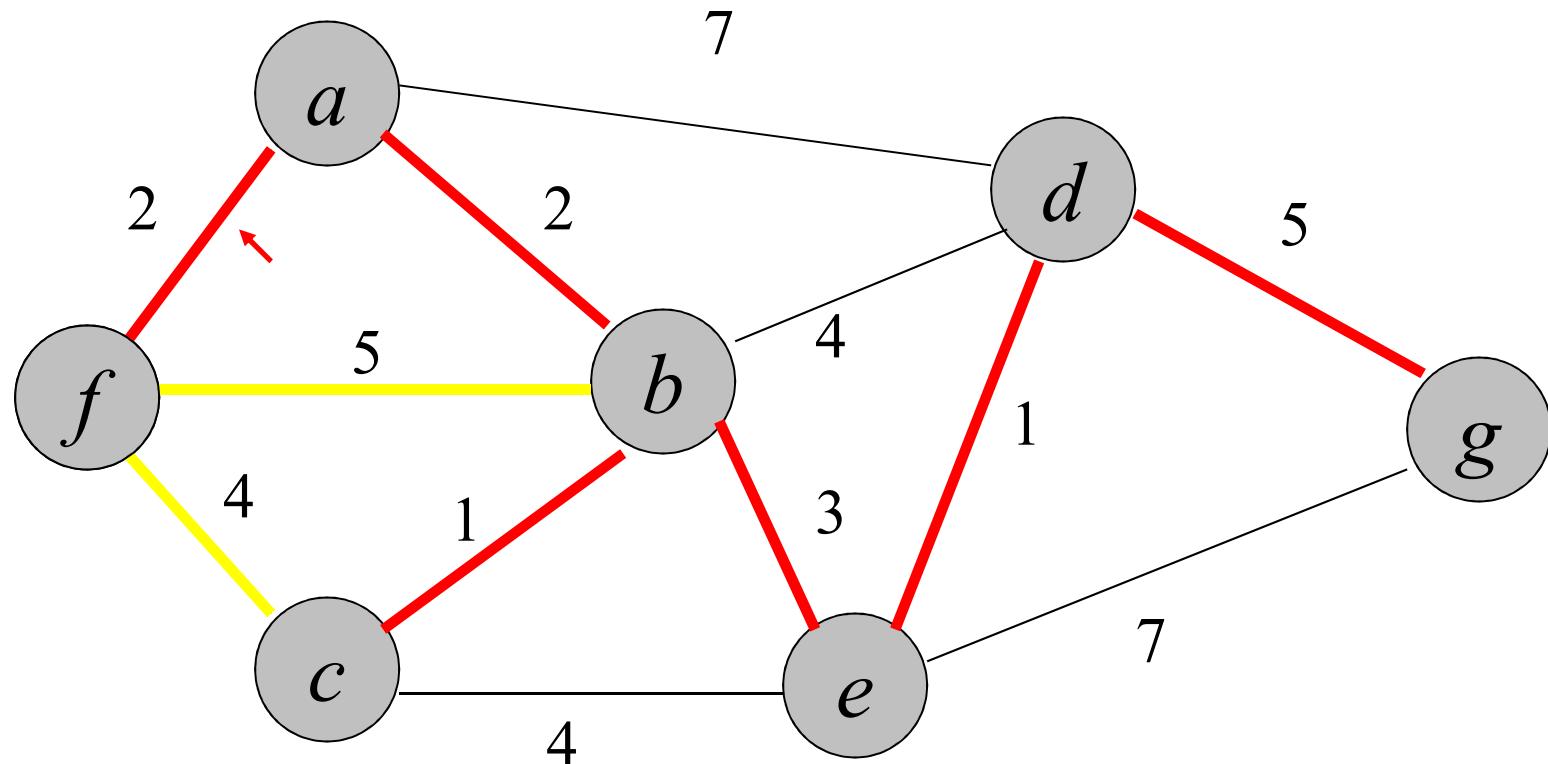
- ★ *T là cây* (bắt đầu từ cây chỉ có 1 đỉnh)
- ★ Cạnh được bổ sung vào *T* là cạnh nhẹ nhất trong số các cạnh nối đỉnh trong *T* với một đỉnh *không ở* trong *T*.

Thuật toán PRIM – Ví dụ



- ★ *T là cây* (bắt đầu từ cây chỉ có 1 đỉnh)
- ★ *Cạnh được bổ sung vào T là cạnh nhẹ nhất trong số các cạnh nối đỉnh trong T với một đỉnh không ở trong T.*

Thuật toán PRIM – Ví dụ



CKNN gồm các cạnh: $(g,d), (d,e), (e,b), (b,c), (b,a), (a,f)$

Độ dài của CKNN: 14

$$5+1+3+1+2+2 = 14$$

Mô tả thuật toán PRIM

void Prim(G, C) // G : đồ thị; C : ma trận trọng số biểu diễn trọng số các cạnh trên đồ thị

{

Chọn đỉnh tuỳ ý $r \in V$;

Cây T chỉ có 1 đỉnh duy nhất r

Khởi tạo cây $T = (V(T), E(T))$ với $V(T) = \{r\}$ và $E(T) = \emptyset$;

while (T có $< n$ đỉnh)

Trong số các cạnh nối 1 đỉnh thuộc T với 1 đỉnh không
thuộc T , tìm cạnh có trọng số nhỏ nhất

Gọi (u, v) là cạnh nhẹ nhất với $u \in V(T)$ và $v \in V(G) - V(T)$

$E(T) \leftarrow E(T) \cup \{(u, v)\};$

$V(T) \leftarrow V(T) \cup \{v\}$

bổ sung đỉnh v và cạnh (u, v) vào cây T

}

}

Cài đặt thuật toán PRIM

- Giả sử đồ thị cho bởi ma trận trọng số $C = \{c[i,j], i, j = 1, 2, \dots, n\}$.
- Ở mỗi bước để nhanh chóng chọn đỉnh và cạnh cần bổ sung vào cây khung, các đỉnh của đồ thị sẽ được gán cho các nhãn:

Nhãn của một đỉnh $v \in V \setminus V(T)$ có dạng $[bestW[v], bestAdj[v]]$:

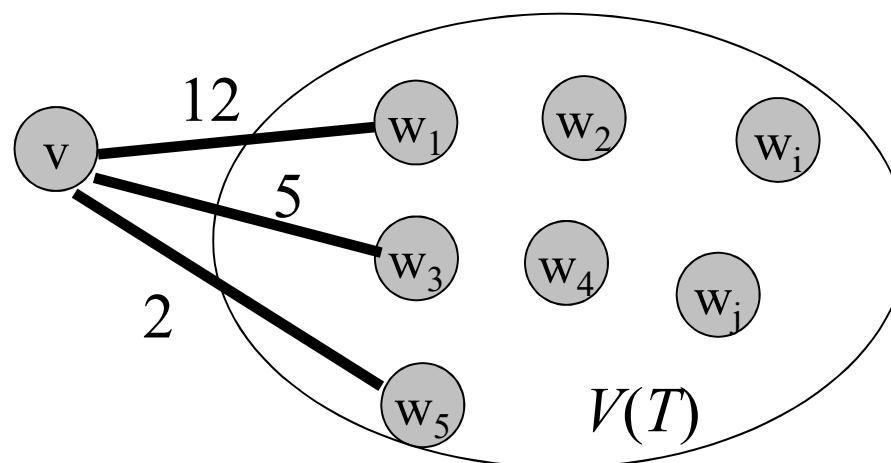
- $bestW[v]$ dùng để ghi nhận khoảng cách từ đỉnh v đến tập đỉnh $V(T)$:

$$bestW[v] := \min \{ c[v, w] : w \in V(T) \} (= c[v, z])$$

Cạnh có trọng số nhỏ nhất trong số các cạnh nối v với 1 đỉnh trong tập $V(T)$

- $bestAdj[v] := z$ ghi nhận đỉnh của cây khung gần v nhất (tức là $c[v, z] = bestW[v]$)

$$\begin{aligned}bestW(v) &= 2 \\bestAdj[v] &= w_5\end{aligned}$$



```

void Prim () {
    // Bước khởi tạo:
    V(T) = { r }; E(T) = ∅ ;
    bestW[r] = 0; bestAdj[r] = r;
    for v ∈ V \ V(T) {
        bestW[v] = c[r,v];
        bestAdj[v] = r;
    }
    // Bước lắp:
    for (k=2; k <= n; k++)
    {
        Tìm u ∈ V \ V(T) thoả mãn: bestW[u] = min { bestW[i] : i ∈ V \ V(T) };
        V(T) = V(T) ∪ { u }; E(T) = E(T) ∪ { ( u, bestAdj[u] ) } ;
        for v ∈ V \ V(T)
            if (bestW[v] > c[u,v]) {
                bestW[v] = c[u,v] ;
                bestAdj[v] = u;
            }
    }
}

```

T là cây khung nhỏ nhất của đồ thị ;

```

void Prim(G, C) //G: đồ thị; C: ma trận trọng số biểu diễn trọng số các cạnh trên đồ thị
{
    Chọn đỉnh tùy ý r ∈ V;
    Khởi tạo cây T=(V(T), E(T)) với V(T)= {r} và E(T)=∅;
    while (T có < n đỉnh )
    {
        Gọi (u, v) là cạnh nhẹ nhất với u ∈ V(T) và v ∈ V(G) – V(T)
        E(T) ← E(T) ∪ { (u, v) };
        V(T) ← V(T) ∪ { v }
    }
}

```

Chuẩn bị dữ liệu cho việc tìm cạnh an toàn
 $\text{bestW}[v]$: trọng số cạnh nhẹ nhất nối v (*chưa thuộc cây khung T*) với một đỉnh trong cây khung T

Vì cây khung T có sự thay đổi: đỉnh u vừa được bổ sung vào cây khung T
 → cập nhật lại nhãn của các đỉnh chưa được bổ sung vào cây T nếu cần thiết

Thời gian tính: $O(|V|^2)$

3.1. Bài toán cây khung nhỏ nhất

Bài toán: Cho đồ thị vô hướng $G=(V,E)$ với trọng số $c(e)$, $e \in E$. Độ dài của cây khung là tổng trọng số trên các cạnh của nó. Cần tìm cây khung có độ dài nhỏ nhất.

- Thuật toán Kruskal:
 - Sử dụng DFS: $O(V^* (V+E))$
 - Sử dụng cấu trúc dữ liệu các tập không giao nhau: $O(E \log_2 E)$
- Thuật toán PRIM:
 - Cài đặt bình thường: $O(V^2)$
 - Sử dụng cấu trúc hàng đợi có ưu tiên: $O((V+E)^* \log V)$

Thuật toán PRIM: cài đặt Priority Queue min

Với mỗi đỉnh v của đồ thị, ta lưu vào trong Priority Queue Q min cặp giá trị $(v, \text{key}(v))$: với

- v là đỉnh chưa thuộc MST
- $\text{key}(v) = \text{BestW}(v)$ (trong số các cạnh nối v với 1 đỉnh thuộc MST, tìm cạnh có trọng số nhỏ nhất, và lưu trọng số nhỏ nhất đó vào $\text{BestW}(v)$)

for each vertex $v \in G$

$\text{key}[v] = \infty$

$\text{pred}[v] = \text{NULL}$

$\text{Q.Insert}(v, \text{key}(v))$

$\text{key}[s] = 0$

 while (! $\text{Q.IsEmpty}()$)

$v = \text{Q.ExtractMin}()$ //Lấy đỉnh v có giá trị key nhỏ nhất ra khỏi Q

 for each edge (v, w) //Duyệt qua danh sách kề của đỉnh v

 if (vertex $w \in Q$) \longrightarrow Sử dụng mảng phụ $\text{MST}[w]$: =true nếu $w \in \text{MST}$;

 false nếu $w \notin \text{MST}$

 if ($\text{key}[w] > c[v, w]$)

$\text{Q.DecreaseKey}(w, c[v, w])$

$\text{pred}[w] = v$

Thuật toán PRIM: cài đặt Priority Queue min

```
for each vertex v ∈ G
    key[v] = ∞
    pred[v] = NULL; MST[v] = false;
    Q.Insert(v, key(v))
key[s]= 0
while (!Q.IsEmpty())
    v= Q.ExtractMin() //Lấy đỉnh v có giá trị key nhỏ nhất ra khỏi Q
    MST[v]=true;
    for each edge (v, w)
        if (!MST[w]) //nếu w thuộc Q, tức là chưa thuộc MST
            if (key[w] > c[v, w])
                Q.DecreaseKey(w, c[v,w])
                pred[w] = v
```

Q.Insert () : V đỉnh
Q.IsEmpty () : V đỉnh
Q.ExtractMin () : V đỉnh
Q.DecreaseKey () : E cạnh

Thuật toán PRIM: cài đặt Priority Queue min

	Priority Queue		
Thao tác	Array	Binary Heap	Fibonacci Heap
Insert	V	$\log V$	1
ExtractMin	V	$\log V$	$\log V$
DecreaseKey	1	$\log V$	1
IsEmpty	1	1	1
PRIM	V^2	$(V+E)\log V$	$E + V\log V$

```

for each vertex v ∈ G
    key[v] = ∞
    pred[v] = NULL; MST[v] = false;
    Q.Insert(v, key(v))
key[s]= 0
while (!Q.IsEmpty())
    v= Q.ExtractMin() //Lấy đỉnh v có giá trị key nhỏ nhất ra khỏi Q
    MST[v]=true;
    for each edge (v, w)
        if (!MST[w]) //nếu w thuộc Q, tức là chưa thuộc MST
            if (key[w] > c[v, w])
                Q.DecreaseKey(w, c[v,w])
            pred[w] = v

```

$Q.\text{Insert}()$: V đỉnh
 $Q.\text{IsEmpty}()$: V đỉnh
 $Q.\text{ExtractMin}()$: V đỉnh
 $Q.\text{DecreaseKey}()$: E cạnh

3.1. Bài toán cây khung nhỏ nhất

- Thuật toán PRIM: cài đặt nào nhanh hơn ?
 - Cài đặt bình thường: $O(V^2)$
 - Sử dụng cấu trúc hàng đợi có ưu tiên: $O((V+E)^* \log_2 V)$

Câu trả lời tùy thuộc vào đồ thị là thưa (sparse) hay dày (dense):

- 2000 đỉnh, 1 triệu cạnh:
 - Heap: 2-3 lần CHẬM HƠN
- 100K đỉnh, 1 triệu cạnh:
 - Heap: 500 lần NHANH HƠN
- 1 triệu đỉnh, 2 triệu cạnh:
 - Heap: 10K lần NHANH HƠN

Kết luận:

- Cài đặt bình thường: tối ưu trong trường hợp đồ thị dày
- Cài đặt sử dụng Heap: tốt hơn trong trường hợp đồ thị thưa

Advanced MST Algorithms

Deterministic comparison based algorithms.

- $O(E \log V)$ Prim, Kruskal, Boruvka.
- $O(E \log \log V)$. Cheriton-Tarjan (1976), Yao (1975).
- $O(E \log^* V)$. Fredman-Tarjan (1987).
- $O(E \log (\log^* V))$. Gabow-Galil-Spencer-Tarjan (1986).
- $O(E \alpha (E, V))$. Chazelle (2000).
- $O(E)$. Holy grail.

Worth noting.

- $O(E)$ randomized. Karger-Klein-Tarjan (1995).
- $O(E)$ verification. Dixon-Rauch-Tarjan (1992).



3. Một số bài toán cơ bản trên đồ thị

3.1. Bài toán cây khung nhỏ nhất

3.2. Bài toán đường đi ngắn nhất

3.3. Bài toán phủ đỉnh

3.4. Bài toán tập độc lập

3.5. Bài toán ghép cặp

3.6. Bài toán tô màu đồ thị

Các dạng bài toán đường đi ngắn nhất

1. **Bài toán một nguồn một đích:** Cho hai đỉnh s và t , cần tìm đường đi ngắn nhất từ s đến t .
2. **Bài toán một nguồn nhiều đích:** Cho s là đỉnh nguồn, cần tìm đường đi ngắn nhất từ s đến tất cả các đỉnh còn lại.
3. **Bài toán mọi cặp:** Tìm đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị.

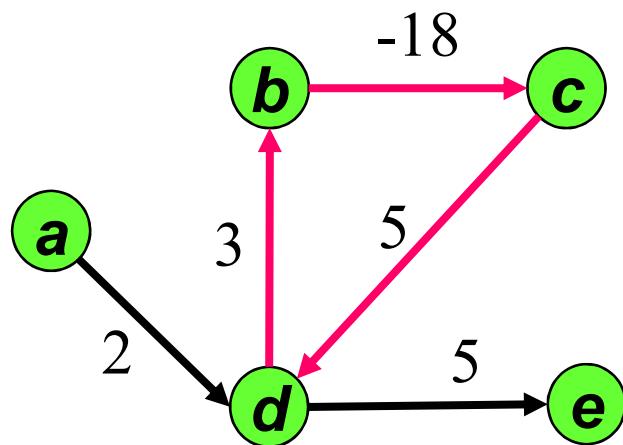
Đường đi ngắn nhất theo số cạnh - BFS.

Nhận xét:

- Các bài toán được xếp theo thứ tự từ đơn giản đến phức tạp
- Nếu có thuật toán hiệu quả để giải một trong ba bài toán thì thuật toán đó cũng có thể sử dụng để giải hai bài toán còn lại

Giả thiết cơ bản

- Nếu đồ thị có chu trình âm thì độ dài đường đi giữa hai đỉnh nào đó có thể làm nhỏ tùy ý:



Chu trình: $(d \rightarrow b \rightarrow c \rightarrow d)$

Độ dài = -10

Xét đường đi từ a đến e :

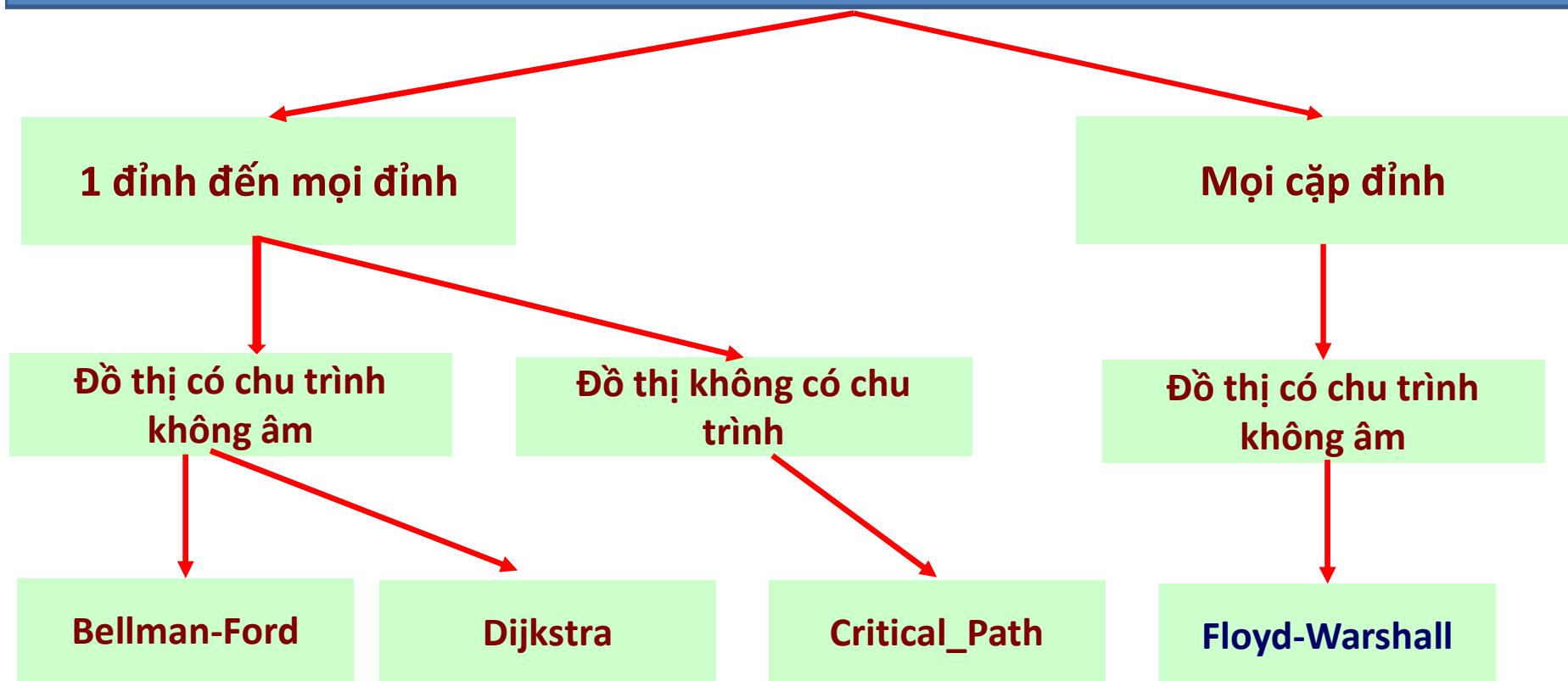
$P: a \rightarrow \sigma(d \rightarrow b \rightarrow c \rightarrow d) \rightarrow e$

$w(P) = 7 - 10\sigma \rightarrow -\infty$, khi $\sigma \rightarrow +\infty$

Giả thiết:

Đồ thị không chứa chu trình độ dài âm (gọi tắt là chu trình âm)

Đường đi ngắn nhất



Trọng số trên cạnh: số thực tùy ý

Độ phức tạp: $O(|V||E|)$

Trọng số trên cạnh ≥ 0

Độ phức tạp: $O(|V|^2)$

Trọng số trên cạnh: số thực tùy ý

Độ phức tạp: $O(|E|)$

Trọng số trên cạnh: số thực tùy ý

Độ phức tạp: $O(|V|^3)$



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Đường đi ngắn nhất xuất phát từ một đỉnh



Biểu diễn đường đi ngắn nhất

Các thuật toán tìm đường đi ngắn nhất làm việc với hai mảng:

- ★ $d(v)$ = độ dài đường đi từ s đến v ngắn nhất hiện biết
(cận trên cho độ dài đường đi ngắn nhất thực sự).
- ★ $p(v)$ = đỉnh đi trước v trong đường đi nói trên $\delta(s, v) \leq d(v)$
(sẽ sử dụng để truy ngược đường đi từ s đến v) .

Khởi tạo (Initialization)

```
for  $v \in V(G)$ 
    do  $d[v] \leftarrow \infty$ 
         $p[v] \leftarrow \text{NULL}$ 
     $d[s] \leftarrow 0$ 
```

Giảm cận trên

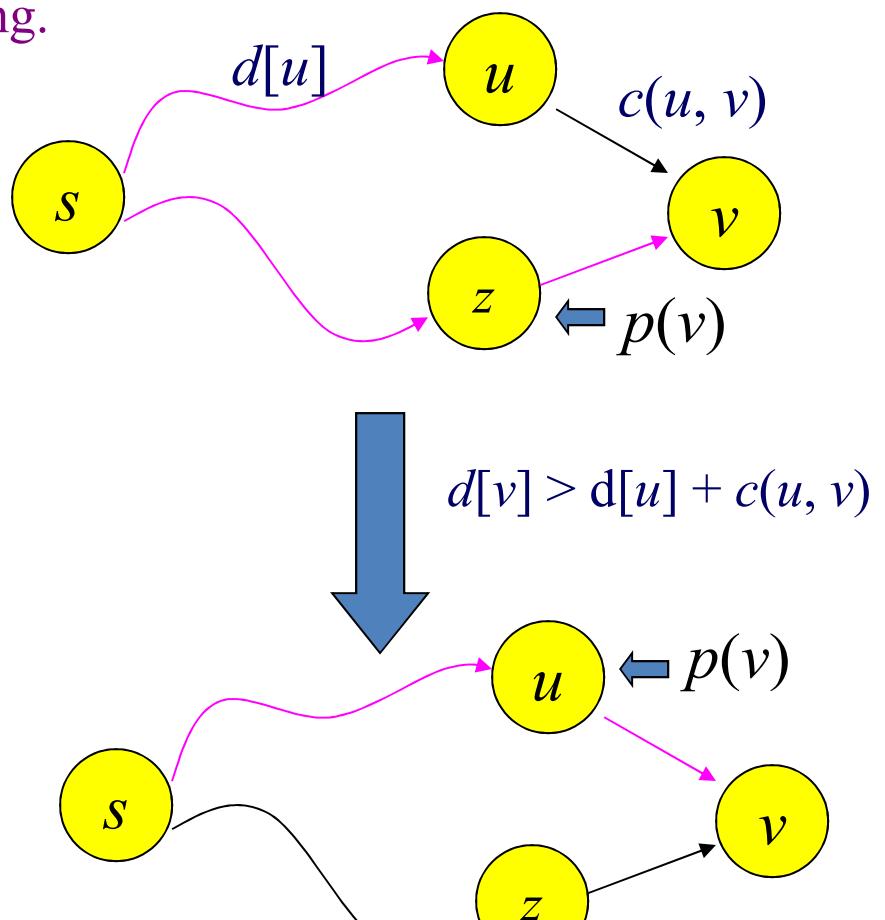
Đang xây dựng đường đi ngắn nhất từ s đến v :

Giả sử đường đi ngắn nhất hiện biết từ s đến v : $s \dots z \rightarrow v$

Sử dụng cạnh (u, v) để kiểm tra xem đường đi đến v đã tìm được có thể làm ngắn hơn nhờ đi qua u hay không.

Relax(u, v)

```
if ( $d[v] > d[u] + c(u, v)$ )
{
     $d[v] \leftarrow d[u] + c(u, v)$ 
     $p[v] \leftarrow u$ 
}
```



Các thuật toán tìm đường khác nhau ở
số lần dùng các cạnh và trình tự duyệt
chúng để thực hiện giảm cận

Nhận xét chung

- Việc cài đặt các thuật toán được thể hiện nhờ ***thủ tục gán nhãn***:
 - Mỗi đỉnh v sẽ có nhãn gồm 2 thành phần $(d[v], p[v])$. Nhãn sẽ biến đổi trong quá trình thực hiện thuật toán
- Nhận thấy rằng để tính khoảng cách từ s đến t , ở đây, ta phải tính khoảng cách từ s đến tất cả các đỉnh còn lại của đồ thị.
- Hiện nay vẫn chưa biết thuật toán nào cho phép tìm đường ngắn nhất giữa hai đỉnh làm việc thực sự hiệu quả hơn những thuật toán tìm đường từ một đỉnh đến tất cả các đỉnh còn lại.

Bài toán đường đi ngắn nhất

1. Thuật toán Bellman-Ford

2. Thuật toán Dijkstra

3. Thuật toán Floyd-Warshall

Thuật toán Bellman-Ford



Richard Bellman
1920-1984



Lester R. Ford, Jr.
1927~2017

Thuật toán Bellman-Ford

- Thuật toán Bellman-Ford tìm đường đi ngắn nhất từ đỉnh s đến tất cả các đỉnh còn lại của đồ thị.
- Thuật toán làm việc trong trường hợp trọng số của các cung là tuỳ ý.
- **Giả thiết rằng** trong đồ thị không có chu trình âm.
- **Đầu vào:** $\text{Đồ thị } G=(V,E)$ với n đỉnh xác định bởi ma trận trọng số $c[u,v]$, $u,v \in V$, đỉnh nguồn $s \in V$;
- **Đầu ra:** V ới mỗi $v \in V$
 - $d[v] = \delta(s, v)$; Độ dài đường đi ngắn nhất từ s đến v
 - $p[v]$ - đỉnh đi trước v trong đđnn từ s đến v .

Mô tả thuật toán

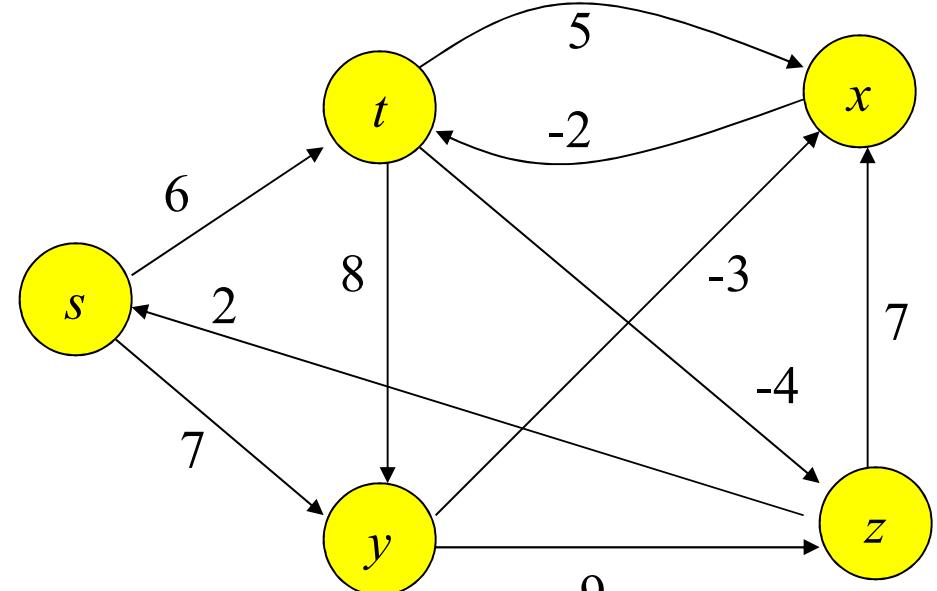
```

void Bellman-Ford ()
{
    for  $v \in V \setminus s$  // Khởi tạo
    {
         $d[v] = c[s, v]$ ;
         $p[v] = s$ ;
    }
    d[s]=0; p[s]=s;
    for ( $k = 1$ ;  $k \leq |V| - 1$ ;  $k++$ )
        for each  $(u, v) \in E$ 
            RELAX( $u, v$ );
}

```

	d	p
t	6	s
x	∞	s
y	7	s
z	∞	s
s	0	s

$v \neq s$
if $(d[v] > d[u] + c[u, v])$ {
 $d[v] = d[u] + c[u, v]$;
 $p[v] = u$;
}



Độ phức tạp tính toán của thuật toán là $O(|V||E|)$.

Chú ý: Có thể chấm dứt vòng lặp theo k khi phát hiện trong quá trình thực hiện vòng lặp trong (for each $(u, v) \in E$) không có biến $d[v]$ nào bị đổi giá trị. Việc này có thể xảy ra đối với $k < |V|-2$, và điều đó làm tăng hiệu quả của thuật toán trong việc giải các bài toán thực tế.

Nhận xét

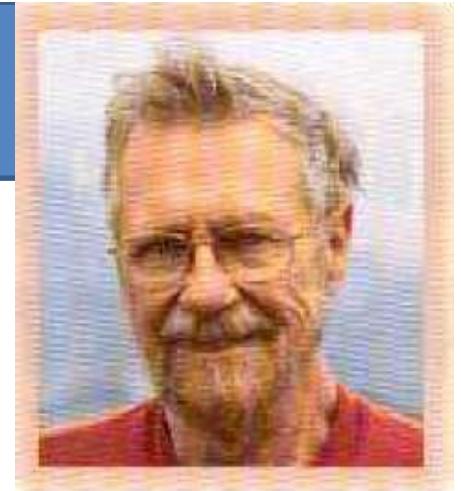
- Độ phức tạp tính toán của thuật toán là $O(|V||E|)$.
- **Chú ý:** Có thể chấm dứt vòng lặp theo k khi phát hiện trong quá trình thực hiện vòng lặp trong không có biến $d[v]$ nào bị đổi giá trị. Việc này có thể xảy ra đối với $k < |V|-2$, và điều đó làm tăng hiệu quả của thuật toán trong việc giải các bài toán thực tế.

Bài toán đường đi ngắn nhất

1. Thuật toán Bellman-Ford
- 2. Thuật toán Dijkstra**
3. Thuật toán Floyd-Warshall

Thuật toán Dijkstra

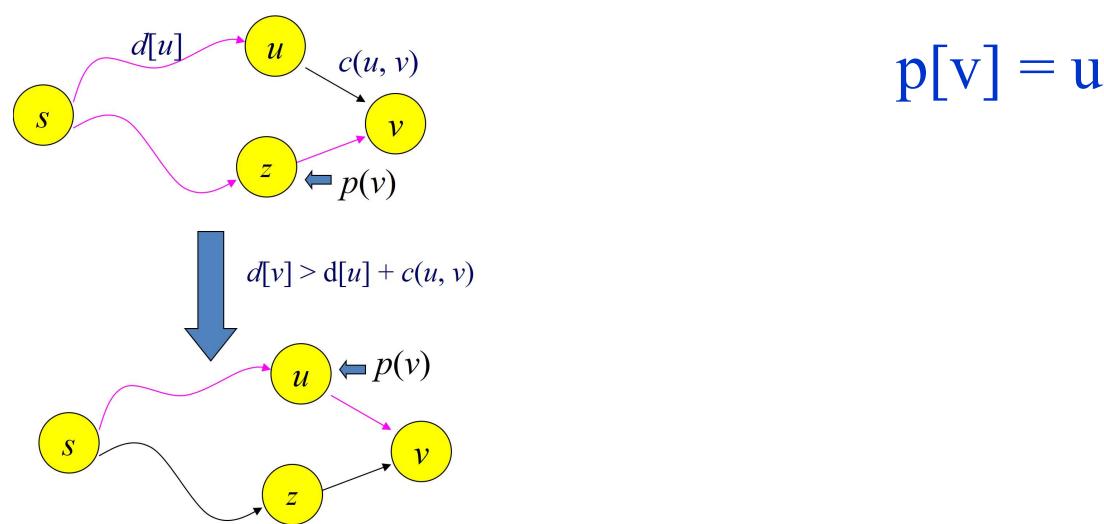
- Trong trường hợp trọng số trên các cung là không âm, thuật toán do Dijkstra đề nghị hiệu quả hơn rất nhiều so với thuật toán Bellman-Ford.
- Thuật toán được xây dựng dựa trên thủ tục gán nhãn. Trong quá trình thực hiện thuật toán, với mỗi đỉnh v ta sẽ lưu trữ nhãn của đỉnh gồm các thông tin sau:
 - $k[v]$: biến bun có giá trị true ($=1$) nếu ta đã tìm được đường đi ngắn nhất từ s đến v , ban đầu biến này được khởi tạo giá trị false ($=0$).
 - $d[v]$: khoảng cách ngắn nhất hiện biết từ s đến v . Ban đầu biến này được khởi tạo giá trị $+\infty$ đối với mọi đỉnh, ngoại trừ $d[s]$ được đặt bằng 0.
 - $p[v]$: là đỉnh đi trước đỉnh v trong đường đi có độ dài $d[v]$. Ban đầu, các biến này được khởi tạo rỗng (chưa biết).



Edsger W.Dijkstra
(1930-2002)

Thuật toán Dijkstra

- Thuật toán lặp lại các thao tác sau đây cho đến khi tất cả các đỉnh được khảo sát xong (nghĩa là $k[u] = \text{true}$ với mọi u):
 - Trong tập đỉnh với $k[u] = \text{false}$: chọn đỉnh u có $d[u]$ là nhỏ nhất.
 - Đặt $k[u] = \text{true}$.
 - For each $v \in \text{Adj}[u]$
 - if ($k[v] == \text{false}$): gọi $\text{Relax}(u, v)$
 - if ($d[v] > d[u] + c(u, v)$): thì đặt lại $d[v] = d[u] + c(u, v)$ và $p[v] = u$



Cài đặt thuật toán với các cấu trúc dữ liệu

Để cài đặt thuật toán Dijkstra chúng ta sử dụng bộ nhãn của các đỉnh:

- Nhãn của mỗi đỉnh v gồm 3 thành phần cho biết các thông tin:
 - $k[v]$ - đã tìm được đường đi ngắn nhất từ đỉnh nguồn đến v hay chưa,
 - $d[v]$ - khoảng cách (độ dài đường đi) từ s đến v hiện biết
 - $p[v]$ - đỉnh đi trước đỉnh v trong đường đi tốt nhất hiện biết.
- Các thành phần này sẽ được cất giữ tương ứng trong các biến $k[v]$, $d[v]$ và $p[v]$.

(1) Cài đặt trực tiếp

Dijkstra_Table(G, s)

```
1. for u  $\in$  V do {  
2.     d[u]  $\leftarrow$  infinity;  
3.     p[u]  $\leftarrow$  NIL;  
4.     k[u]  $\leftarrow$  FALSE;  
5. }  
6. d[s]  $\leftarrow$  0;  
// s là đỉnh nguồn  
7. T = V;
```

```
8. while T  $\neq \emptyset$  do {  
9.     u  $\leftarrow$  đỉnh có d[u] là nhỏ nhất trong T;  
10.    k[u]=TRUE;  
11.    T = T-{u};  
12.    for (v  $\in$  Adj(u) && !k[v]) do  
13.        if (d[v]  $>$  d[u] + c[u, v]) {  
14.            d[v] = d[u] + c[u, v];  
15.            p[v] = u;  
16.        }  
17.    }
```

Dễ dàng nhận thấy rằng Dijkstra_Table(G, s) đòi hỏi thời gian $O(|V|^2+|E|)$.

(2) Cài đặt thuật toán Dijkstra sử dụng hàng đợi có ưu tiên

Do tại mỗi bước ta cần tìm ra đỉnh với nhãn khoảng cách nhỏ nhất, nên để thực hiện thao tác này một cách hiệu quả ta sẽ sử dụng hàng đợi có ưu tiên.

Dijkstra_Table(G, s)

```
1. for  $u \in V$  do {  
2.      $d[u] \leftarrow \text{infinity}$ ;  
3.      $p[u] \leftarrow \text{NIL}$ ;  
4.      $k[u] \leftarrow \text{FALSE}$ ;  
5. }  
6.  $d[s] \leftarrow 0$ ;  
    // s là đỉnh nguồn  
7.  $T = V$ ;
```

```
8. while  $T \neq \emptyset$  do {  
9.      $u \leftarrow \text{đỉnh có } d[u] \text{ là nhỏ nhất trong } T$ ;  
10.     $k[u] = \text{TRUE}$ ;  
11.     $T = T - \{u\}$ ;  
12.    for ( $v \in \text{Adj}(u) \text{ && } !k[v]$ ) do  
13.        if ( $d[v] > d[u] + c[u, v]$ ) {  
14.             $d[v] = d[u] + c[u, v]$ ;  
15.             $p[v] = u$ ;  
16.        }  
17.    }
```

(2) Cài đặt thuật toán Dijkstra sử dụng hàng đợi có ưu tiên

Dijkstra_Heap(G, s)

```
1.  for u ∈ V do {  
2.      d[u] ← infinity;  
3.      p[u] ← NIL;  
4.      k[u] ← FALSE;  
5.  }  
6.  d[s] ← 0; // s là đỉnh nguồn  
7.  Q ← Build_Min_Heap (d[V]) ; // Khởi tạo hàng đợi có ưu tiên Q từ d[v], v∈V  
8.  while Not Empty(Q) do {  
9.      u ← Extract-Min(Q) ; // loại bỏ gốc của Q và đưa vào u  
10.     k[u]=TRUE;  
11.     for (v ∈ Adj(u) && !k[v]) do  
12.         if d[v] > d[u] + c[u, v] {  
13.             d[v] = d[u] + c[u, v];  
14.             p[v] = u;  
15.             Decrease_Key(Q,v,d[v]);  
16.         }  
17.     }
```

Phân tích thời gian tính của thuật toán

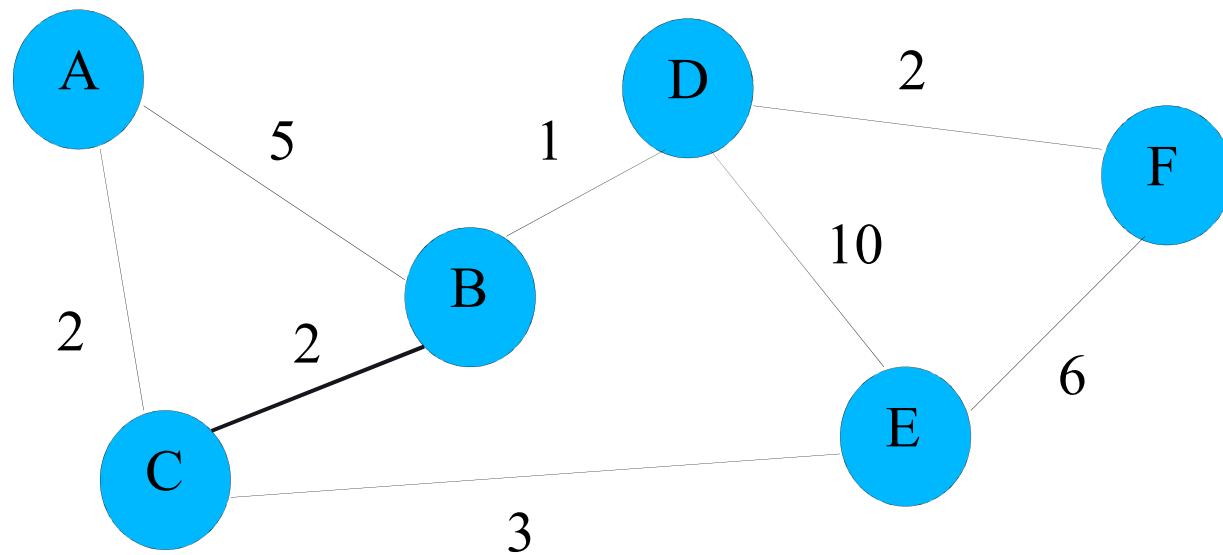
- Vòng lặp for ở dòng 1-5 đòi hỏi thời gian $O(|V|)$.
- Việc khởi tạo min-heap (đống min) dòng 7 đòi hỏi thời gian $O(|V|)$.
- Vòng lặp while ở dòng 8 lặp $|V|$ lần, do đó thao tác Extract-Min thực hiện $|V|$ lần, vậy đòi hỏi thời gian tổng cộng là $O(|V| \log|V|)$.
- Thao tác Decrease_Key ở dòng 15 phải thực hiện không quá $O(|E|)$ lần. Do đó, thời gian thực hiện thao tác này trong thuật toán là $O(|E| \log|V|)$.
- Vậy tổng cộng thời gian tính của thuật toán là $O((|E| + |V|) \log|V|)$

Dijkstra_Heap(G, s)

```
1. for u ∈ V do {  
2.     d[u] ← infinity;  
3.     p[u] ← NIL;  
4.     k[u] ← FALSE;  
5. }  
6. d[s] ← 0; // s là đỉnh nguồn  
7. Q ← Build_Min_Heap(d[V]); // Khởi tạo hàng đợi có ưu tiên Q từ d[v], v∈V  
8. while Not Empty(Q) do {  
9.     u ← Extract-Min(Q); // loại bỏ gốc của Q và đưa vào u  
10.    k[u]=TRUE;  
11.    for (v ∈ Adj(u) && !k[v]) do  
12.        if d[v] > d[u] + c[u, v] {  
13.            d[v] = d[u] + c[u, v];  
14.            p[v] = u;  
15.            Decrease_Key(Q, v, d[v]);  
16.        }  
17.    }
```

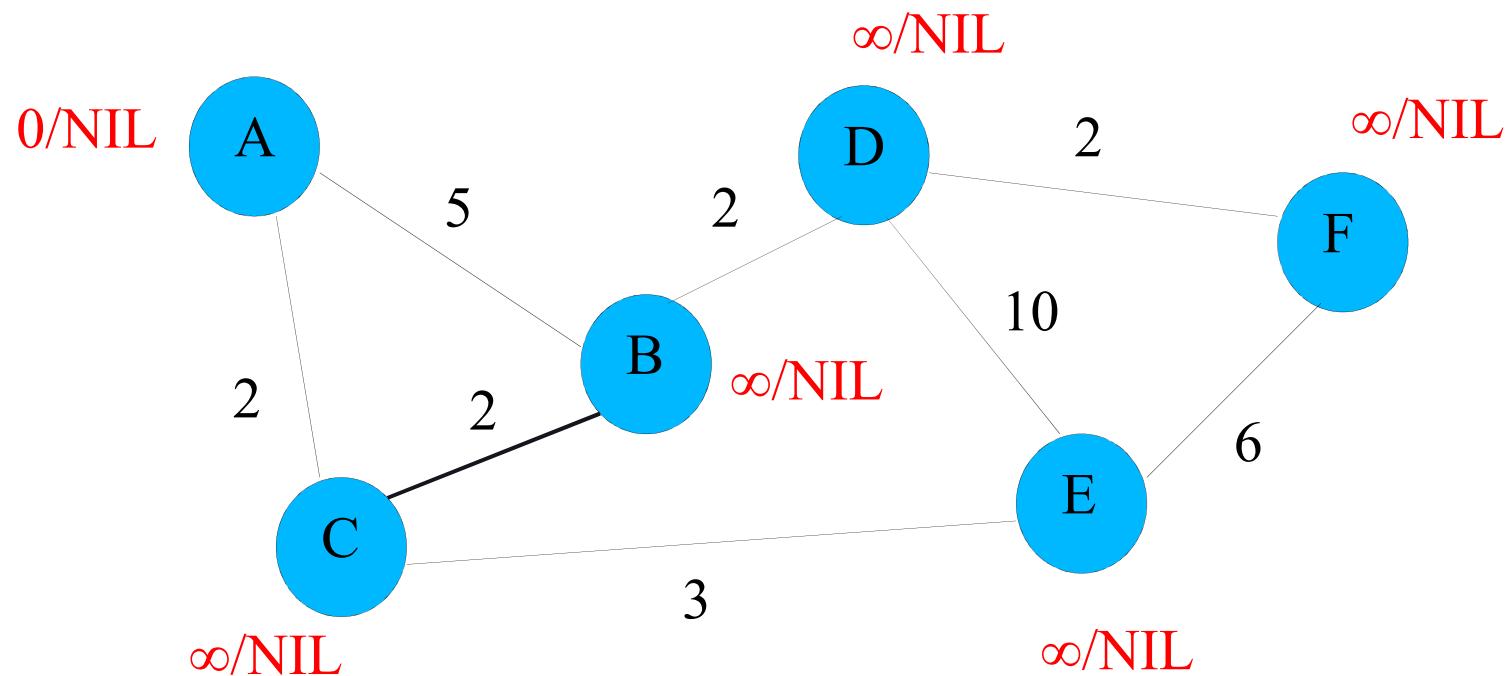
Ví dụ: Dijkstra algorithm

Tìm đường đi ngắn nhất từ đỉnh A đến tất cả các đỉnh còn lại



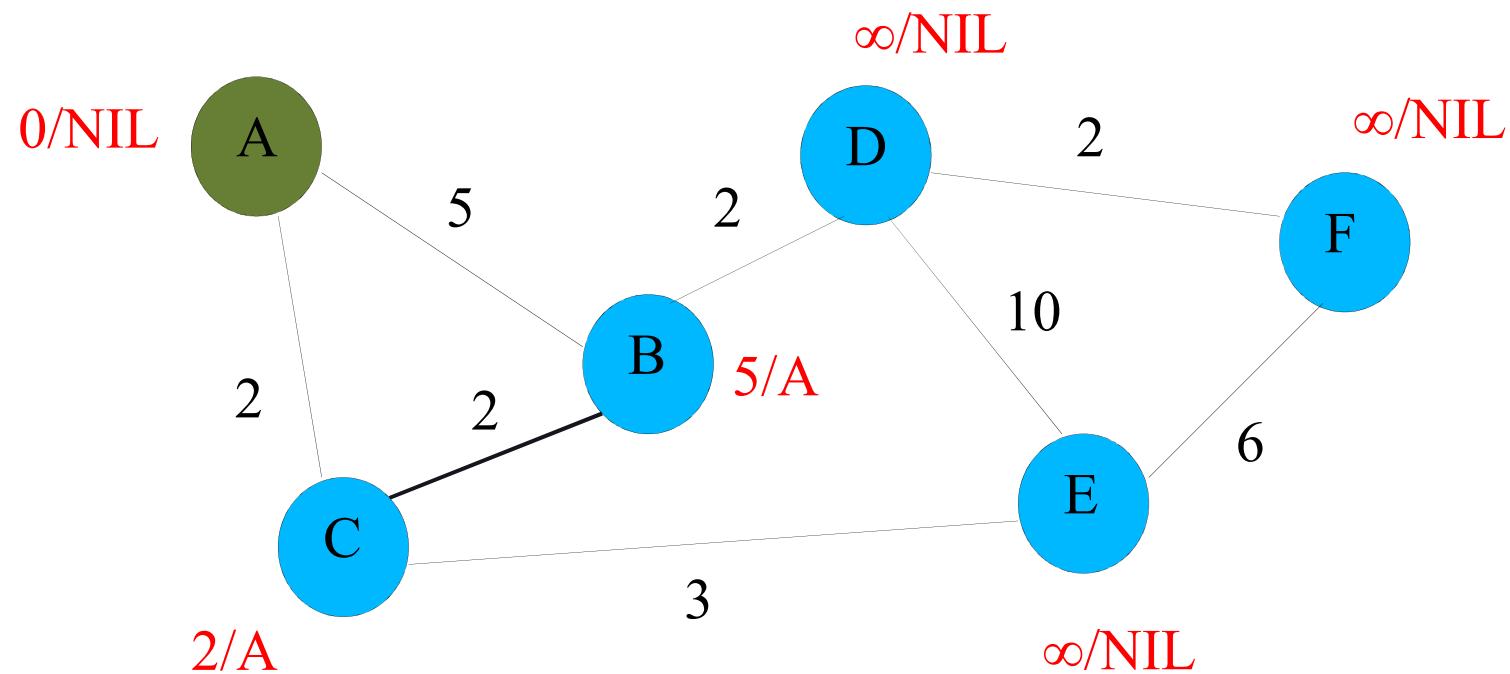
Thuật toán Dijkstra: Khởi tạo

Priority queue: A, B, C, D, E, F



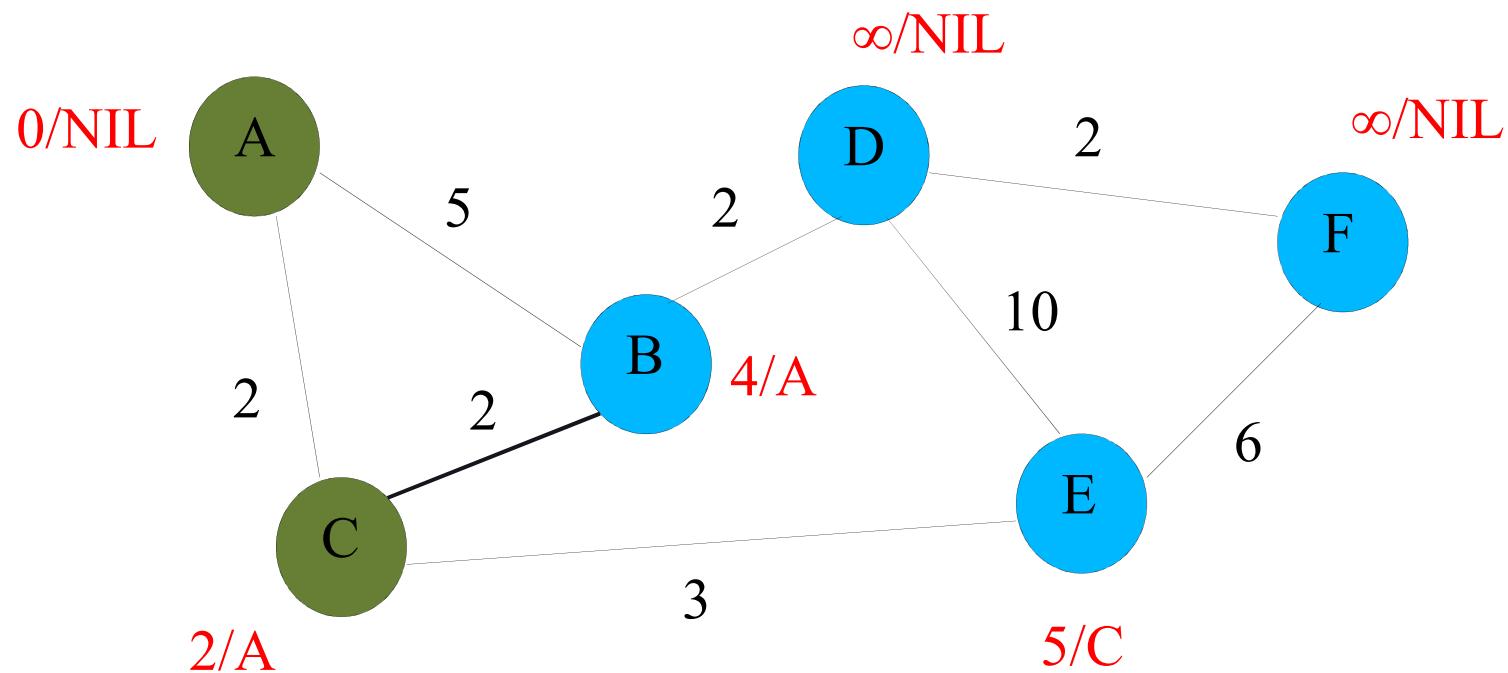
Thuật toán Dijkstra: Cố định nhãn đỉnh A

Priority queue: C, B, D, E, F



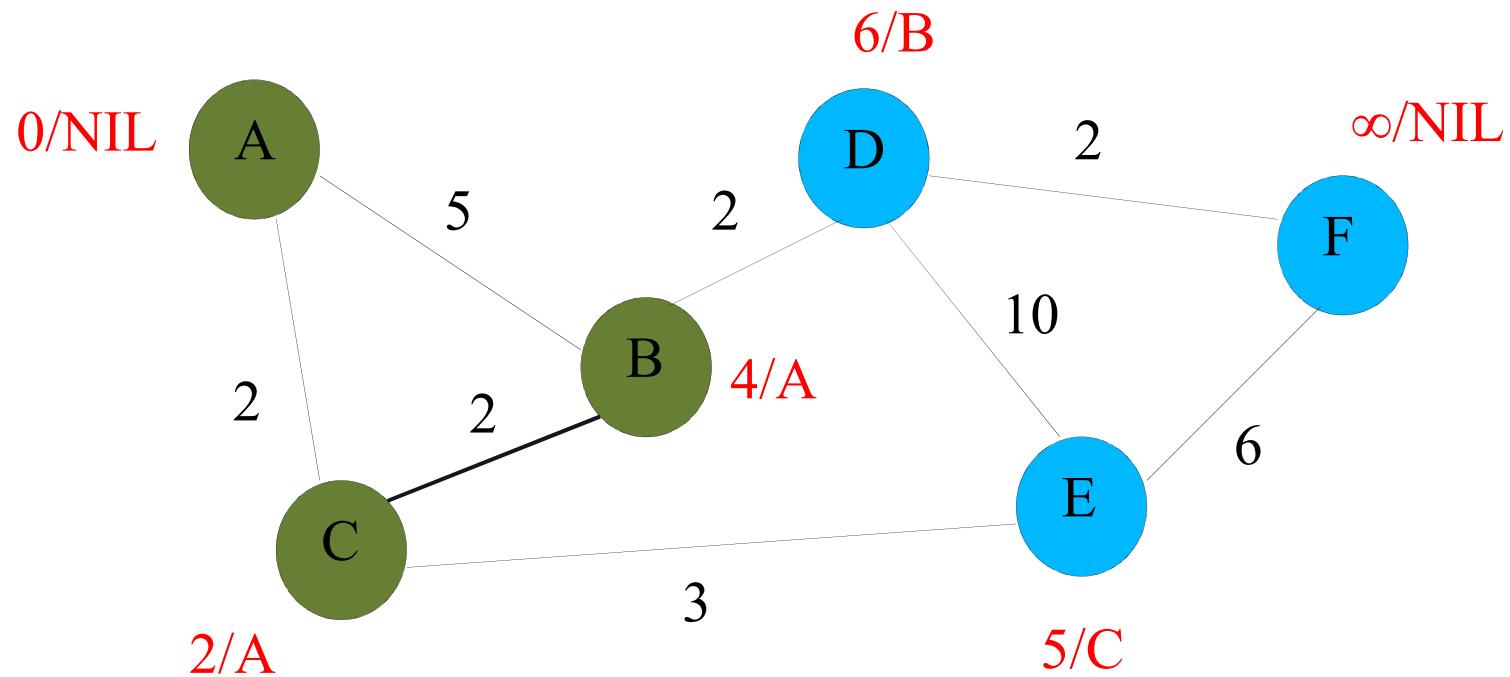
Thuật toán Dijkstra: Cố định nhãn đỉnh C

Priority queue: B, E, D, F



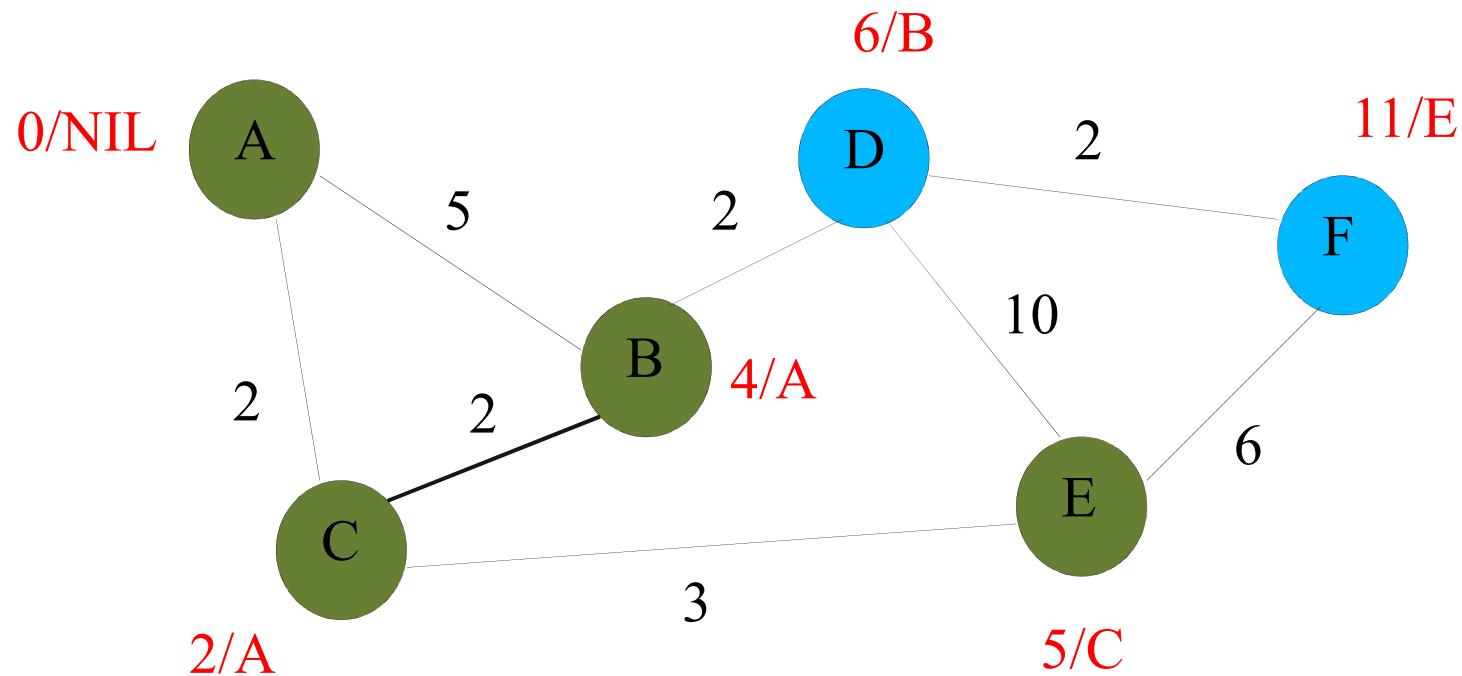
Thuật toán Dijkstra: Cố định nhãn đỉnh B

Priority queue: E, D, F



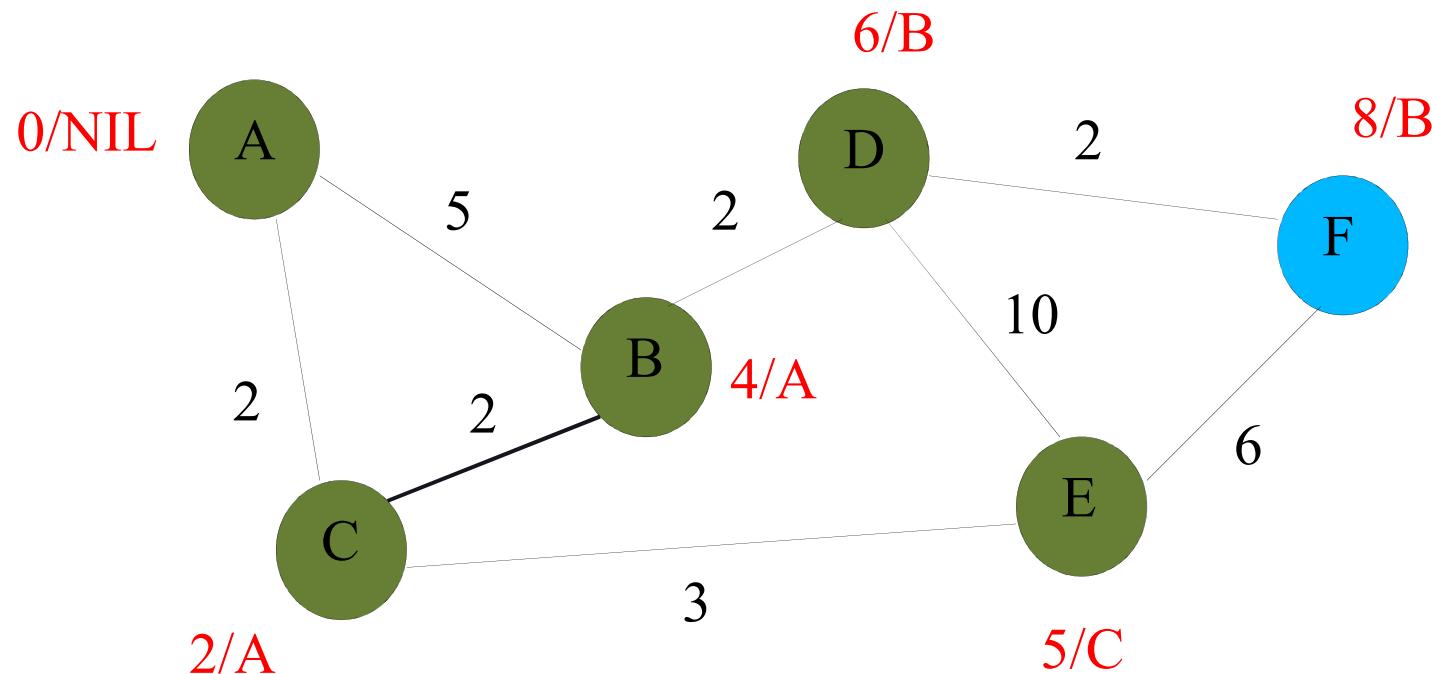
Thuật toán Dijkstra: Cố định nhãn đỉnh E

Priority queue: D, F



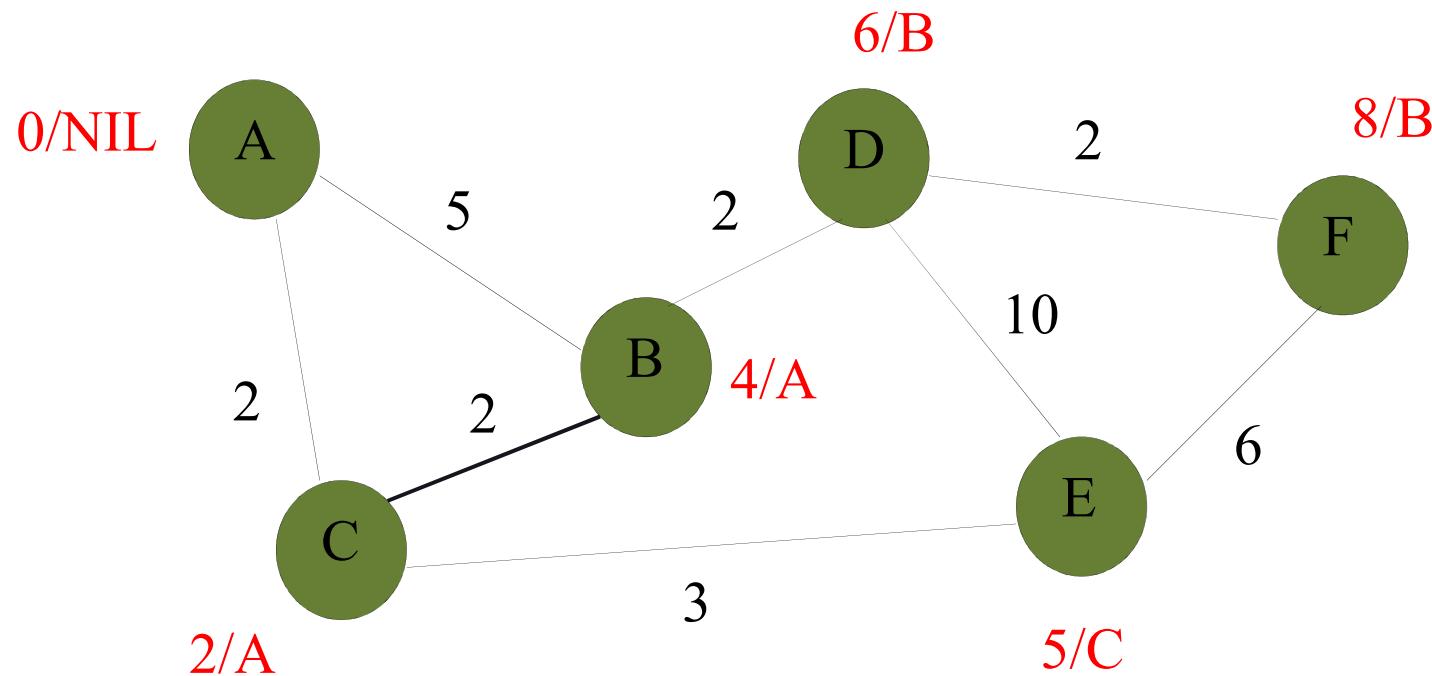
Thuật toán Dijkstra: Cố định nhãn đỉnh D

Priority queue: F



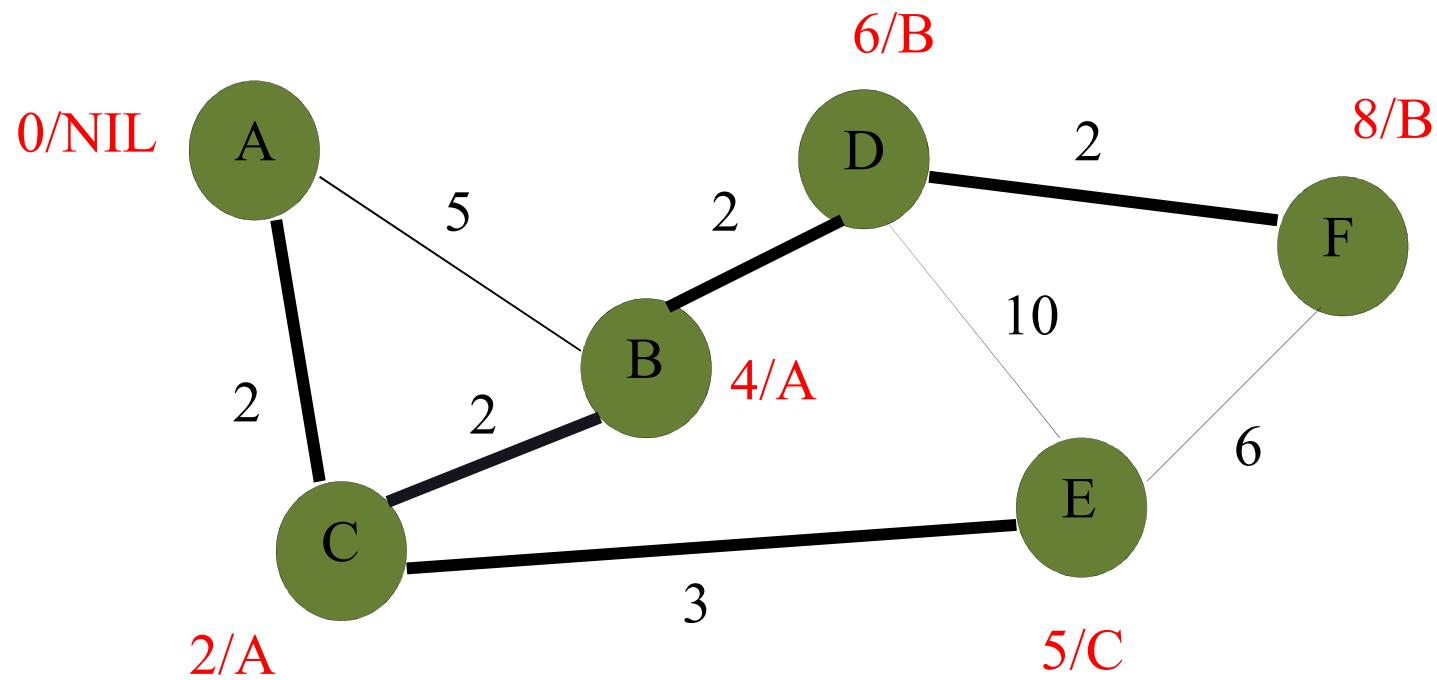
Thuật toán Dijkstra: Cố định nhãn đỉnh F

Priority queue:



Kết quả

Cây đường đi ngắn nhất xuất phát từ A:



Cài đặt Dijkstra theo 2 cách

```
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

typedef pair<int, int> iPair;

class Graph
{
    int V;      // No. of vertices
    // In a weighted graph, we need to store vertex and weight pair for every edge
    list< pair<int, int> > *adj;
    //vector <pair<int, int> > *adj1;

public:
    Graph(int V); // Constructor
    ~Graph() { delete [] adj; } // To avoid memory leak
    //Them canh (u, v) co trong so = weight vao do thi:
    void addEdge(int u, int v, int weight);

    //Thuat toan Dijkstra tim duong di ngan nhat tu dinh s den tat ca cac dinh con lai cua do thi
    void Dijkstra(int s);

    //Thuat toan Dijkstra tim duong di ngan nhat tu dinh s den tat ca cac dinh con lai cua do thi
    void Dijkstra_PriorityQueue(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair> [V];
    //adj1 = new vector <iPair> [V];
}

void Graph::addEdge(int u, int v, int weight)
{
    adj[u].push_back(make_pair(v, weight));
    adj[v].push_back(make_pair(u, weight)); //do thi vo huong
}
```

```
//Dijkstra cai dat truc tiep
void Graph::Dijkstra(int s)
{
    vector<int> d(V, INF);
    vector<int> p(V, s);
    vector <bool> k(V, 0);
    d[s] = 0;

    int count=1;
    int mindistance,i,j,u;
    while(count < V-1)
    {
        mindistance = INF;
        for(i=0; i<V; i++) //Tim dinh u co d[u] min
            if(!k[i] && d[i]<mindistance)
            {
                mindistance = d[i];
                u=i;
            }
        k[u]=1;
        list< pair<int, int> >::iterator i;
        //Duyet qua danh sach ke cua dinh u:
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
        {
            // Get vertex label and weight of current adjacent of u.
            int v = (*i).first;
            int weight = (*i).second; //trong so tren cung (u, v)
```

```

    //Goi Relax(u, v):
    if (d[v] > d[u] + weight)
    {
        d[v] = d[u] + weight; //cap nhat d[v]
        p[v] = u;
    }
    count++;
}

cout<<" Dinh      Do dai DDNN tu dinh "<<s<<"      DDNN \n";
for(i=0; i<V; i++)
{
    if(i != s)
    {
        cout<<i<<" \t \t "<<d[i]<<"\t\t";
        cout<<i;
        j=i;
        do {
            j=p[j];
            cout<<" - "<<j;
        }while(j != s);
        cout<<endl;
    }
}

```

```
//Dijkstra cai dat priority queue:  
void Graph::Dijkstra_PriorityQueue(int s)  
{  
    /* Create a priority queue to store vertices*/  
    priority_queue< iPair, vector <iPair> , greater<iPair> > Q;  
    vector<int> d(V, INF);  
    vector<int> p(V, s);  
    vector <bool> k(V,0);  
  
    // Insert source itself in priority queue and initialize its distance as 0.  
    d[s] = 0; Q.push(make_pair(d[s], s));  
  
    /* Looping till priority queue becomes empty (or all  
       distances are not finalized) */  
    while (!Q.empty())  
    {  
        int u = Q.top().second; //dinh u co d[u] min  
        int temp = Q.top().first;  
        Q.pop();  
  
        if (!k[u])  
        {  
            k[u] = true;  
            list< pair<int, int> ::iterator i;  
            for (i = adj[u].begin(); i != adj[u].end(); ++i)  
            {  
                // Get vertex label and weight of current adjacent of u.  
                int v = (*i).first;  
                int weight = (*i).second; //trong so tren cung (u, v)  
                if (d[v] > d[u] + weight)  
                    d[v] = d[u] + weight;  
                p[v] = u;  
            }  
        }  
    }  
}
```

```

        //Goi Relax(u, v):
        if (d[v] > d[u] + weight)
        {
            d[v] = d[u] + weight;
            Q.push(make_pair(d[v], v));
            p[v]=u;
        }
    }

int i, j;
cout<<" Dinh      Do dai DDNN tu dinh "<<s<<"      DDNN \n";
for(i=0; i<V; i++)
{
    if(i != s)
    {
        cout<<i<<" \t \t "<<d[i]<<"\t\t";
        cout<<i;
        int j=i;
        do {
            j=p[j];
            cout<<" <- "<<j;
        }while(j != s);
        cout<<endl;
    }
}

```

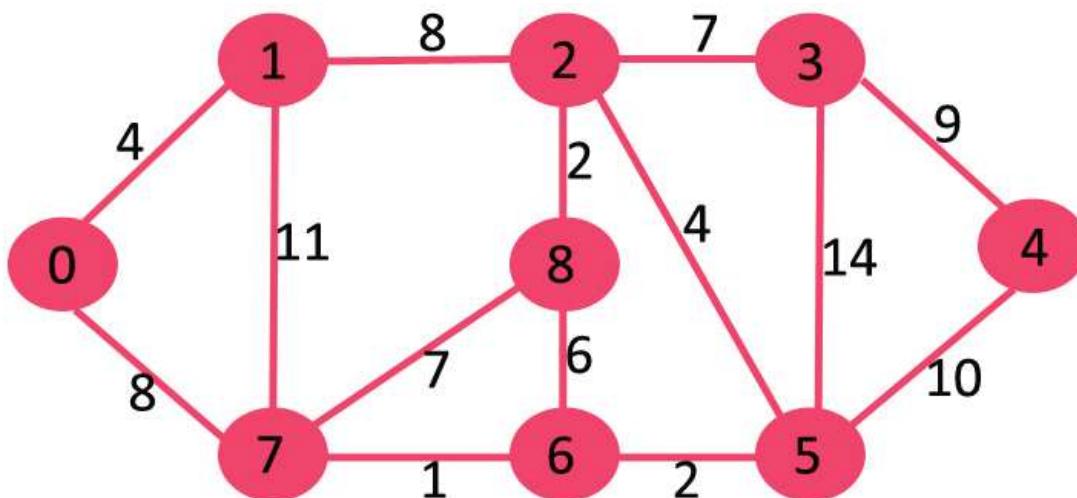
```

int main()
{
    int V = 9;
    Graph g(V);
    g.addEdge(0, 1, 4);    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);   g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);    g.addEdge(7, 8, 7);

    cout<<"__ KET QUA CAI DAT TRUC TIEP THUAT TOAN DIJKSTRA_____ "<<endl;
    g.Dijkstra(0);
    cout<<"__ KET QUA CAI DAT DIJKSTRA DUNG PRIORITY QUEUE _____ "<<endl;
    g.Dijkstra_PriorityQueue(0);

    return 0;
}

```



ĐƯỜNG ĐI NGẮN NHẤT GIỮA MỌI CẶP ĐỈNH

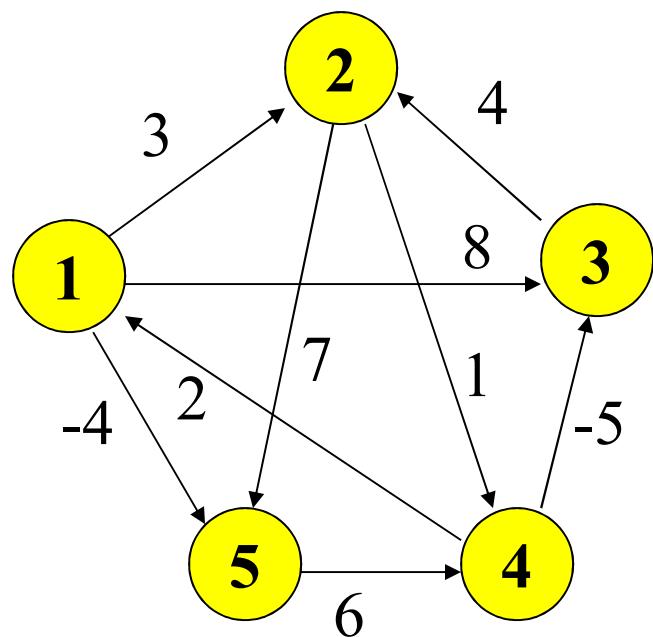
All-Pairs Shortest Paths

Đường đi ngắn nhất giữa mọi cặp đỉnh

Bài toán Cho đồ thị $G = (V, E)$, với trọng số trên cạnh e là $w(e)$,
đối với mỗi cặp đỉnh u, v trong V , tìm đường đi ngắn
nhất từ u đến v .

- ✳ Đầu vào: *ma trận trọng số*.
- ✳ Đầu ra *ma trận*: phần tử ở dòng u cột v là độ dài đường
đi ngắn nhất từ u đến v .
- ✳ Cho phép có trọng số âm
- ✳ **Giả thiết: Đồ thị không có chu trình âm.**

Ví dụ



Đầu vào

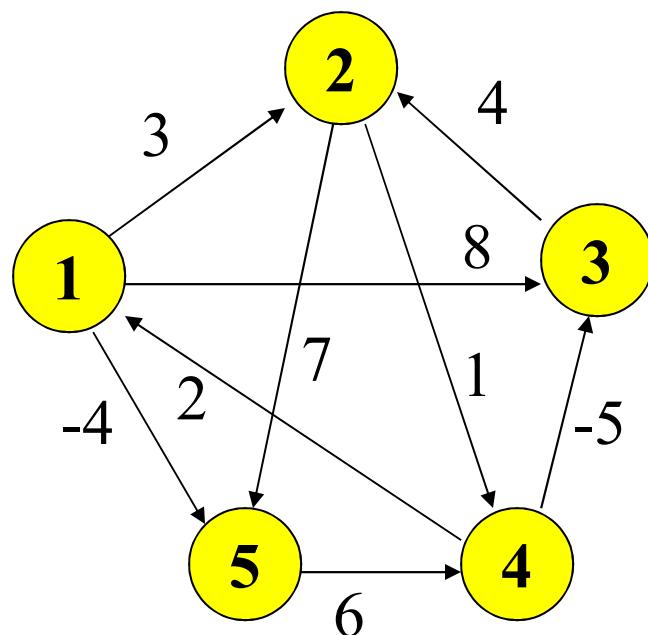
Ma trận trọng số $C_{nxn} = (c)_{ij}$ với

$$c_{ij} = \begin{cases} 0 & \text{nếu } i=j \\ c(i,j) & \text{nếu } i \neq j \text{ & } (i,j) \in E \\ \infty & \text{còn lại} \end{cases}$$

$$C_{5 \times 5} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Đầu ra

Ma trận: phần tử ở dòng u cột v là độ dài đường đi ngắn nhất từ u đến v .



Đường đi từ **1** đến **2**: 1 - 5 - 4 - 3 - 2

$$\left(\begin{array}{ccccc} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{array} \right)$$

\downarrow

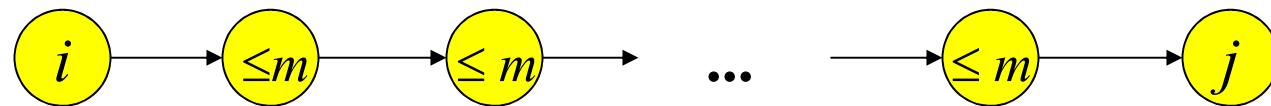
= -4 + 6 - 5 + 4

Đường đi từ **5** đến **1**: 5 - 4 - 1

Thuật toán Floyd-Warshall

- Sử dụng quy hoạch động để tính đường đi ngắn nhất:

$d_{ij}^{(m)}$ = độ dài đường đi ngắn nhất từ i đến j chỉ sử dụng các đỉnh trung gian trong tập đỉnh $\{1, 2, \dots, m\}$.



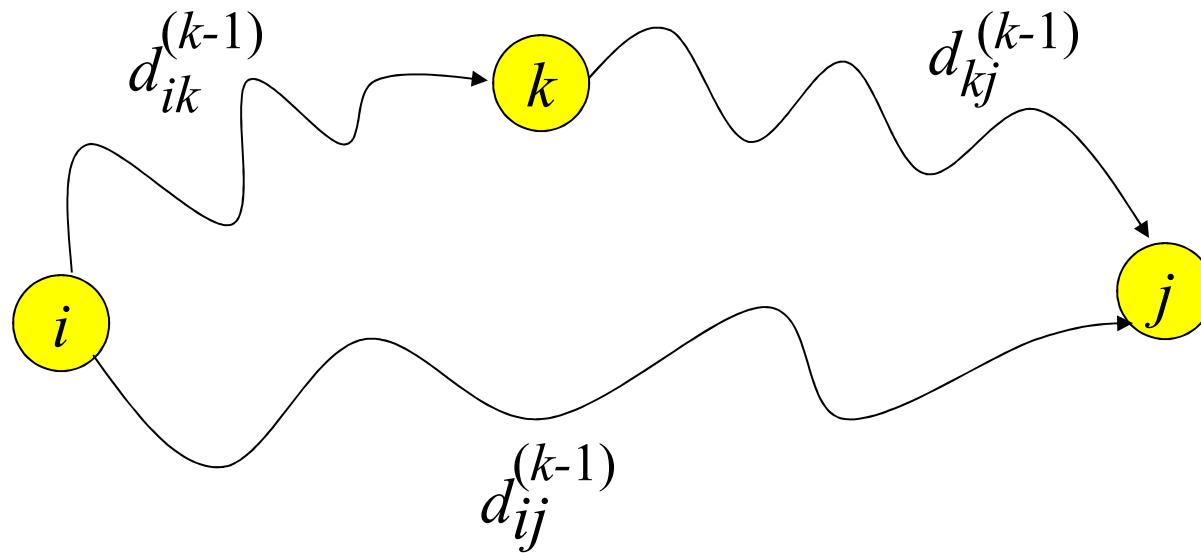
Đồ thị có n đỉnh $\{1, 2, \dots, n\} \rightarrow$ độ dài đường đi ngắn nhất từ i đến j trên đồ thị là $d_{ij}^{(n)}$

Công thức đê qui tính $d^{(k)}$

★ $d_{ij}^{(0)} = c_{ij}$



★ $d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ nếu $k \geq 1$



Thuật toán Floyd-Warshall

```
void Floyd-Warshall(n, W)
```

```
{
```

```
     $D^{(0)} \leftarrow W$ 
```

```
    for ( $k=1; k \leq n; k++$ )
```

Đường đi chỉ qua các đỉnh trung gian trong $\{1, \dots, k\}$

```
        for ( $i=1; i \leq n; i++$ )
```

```
            for ( $j = 1; j \leq n; j++$ )
```

Mọi cặp đỉnh (i, j)

```
                 $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
```

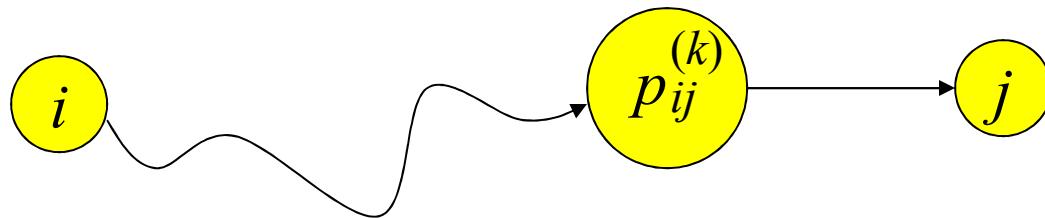
```
return  $D^{(n)}$ ;
```

```
}
```

Thời gian tính $\Theta(n^3)$!

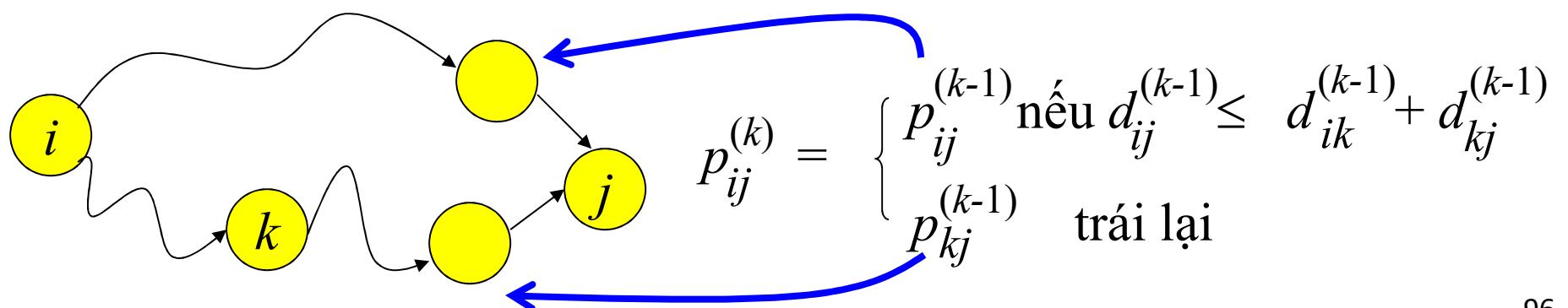
Xây dựng đường đi ngắn nhất

Predecessor matrix $P^{(k)} = (p_{ij}^{(k)})$:

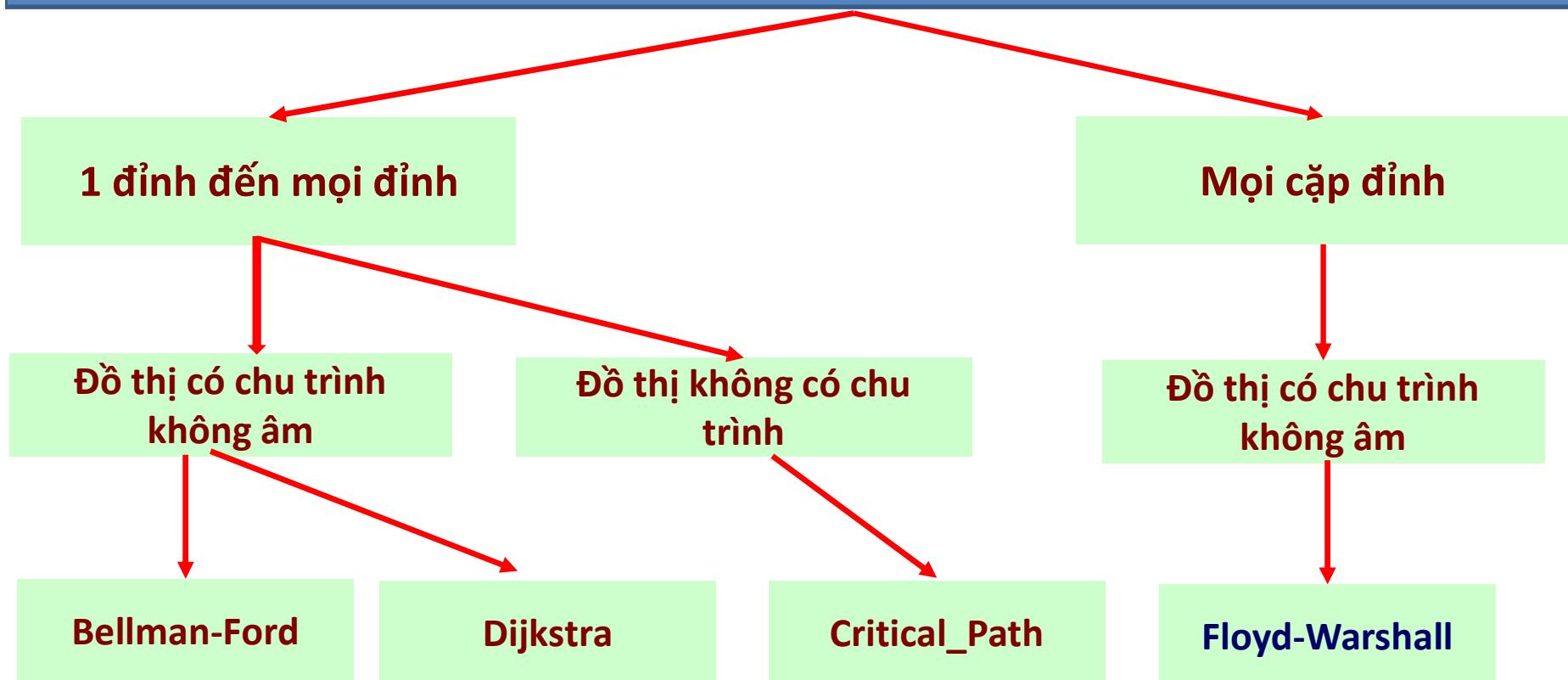


đường đi ngắn nhất từ i đến j chỉ qua các đỉnh trung gian trong $\{1, 2, \dots, k\}$.

$$p_{ij}^{(0)} = \begin{cases} i, & \text{nếu } (i, j) \in E \\ \text{Nil}, & \text{nếu } (i, j) \notin E \end{cases}$$



Đường đi ngắn nhất



Trọng số trên cạnh: số thực tùy ý

Độ phức tạp: $O(|V||E|)$

Trọng số trên cạnh ≥ 0

Độ phức tạp: $O(|V|^2)$

Trọng số trên cạnh: số thực tùy ý

Độ phức tạp: $O(|E|)$

Trọng số trên cạnh: số thực tùy ý

Độ phức tạp: $O(|V|^3)$

3. Một số bài toán cơ bản trên đồ thị

3.1. Bài toán cây khung nhỏ nhất

3.2. Bài toán đường đi ngắn nhất

3.3. Bài toán phủ đỉnh

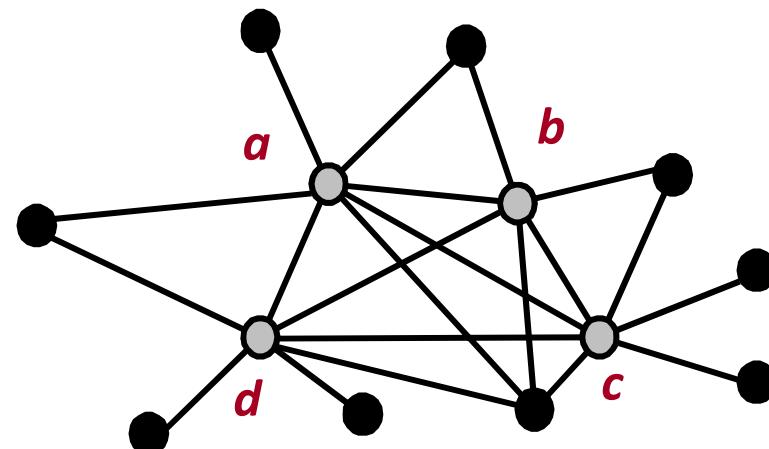
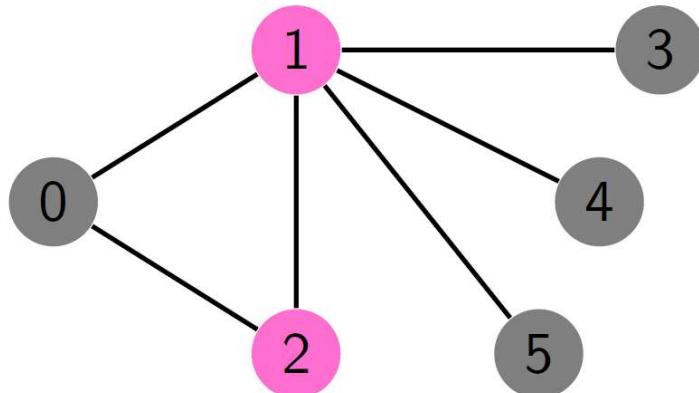
3.4. Bài toán tập độc lập

3.5. Bài toán ghép cặp

3.6. Bài toán tô màu đồ thị

3.3. Bài toán phủ đỉnh (Vertex Cover)

- Phủ đỉnh của đồ thị vô hướng $G=(V,E)$ là tập con S của V , sao cho mỗi cạnh của đồ thị đều có ít nhất một đầu mút trong S .
- Bài toán VC: Cho đồ thị G , tìm phủ đỉnh với lực lượng nhỏ nhất của G .
- VC là NP-khó (NP-hard) khi đồ thị là bất kỳ.



3. Một số bài toán cơ bản trên đồ thị

3.1. Bài toán cây khung nhỏ nhất

3.2. Bài toán đường đi ngắn nhất

3.3. Bài toán phủ đỉnh

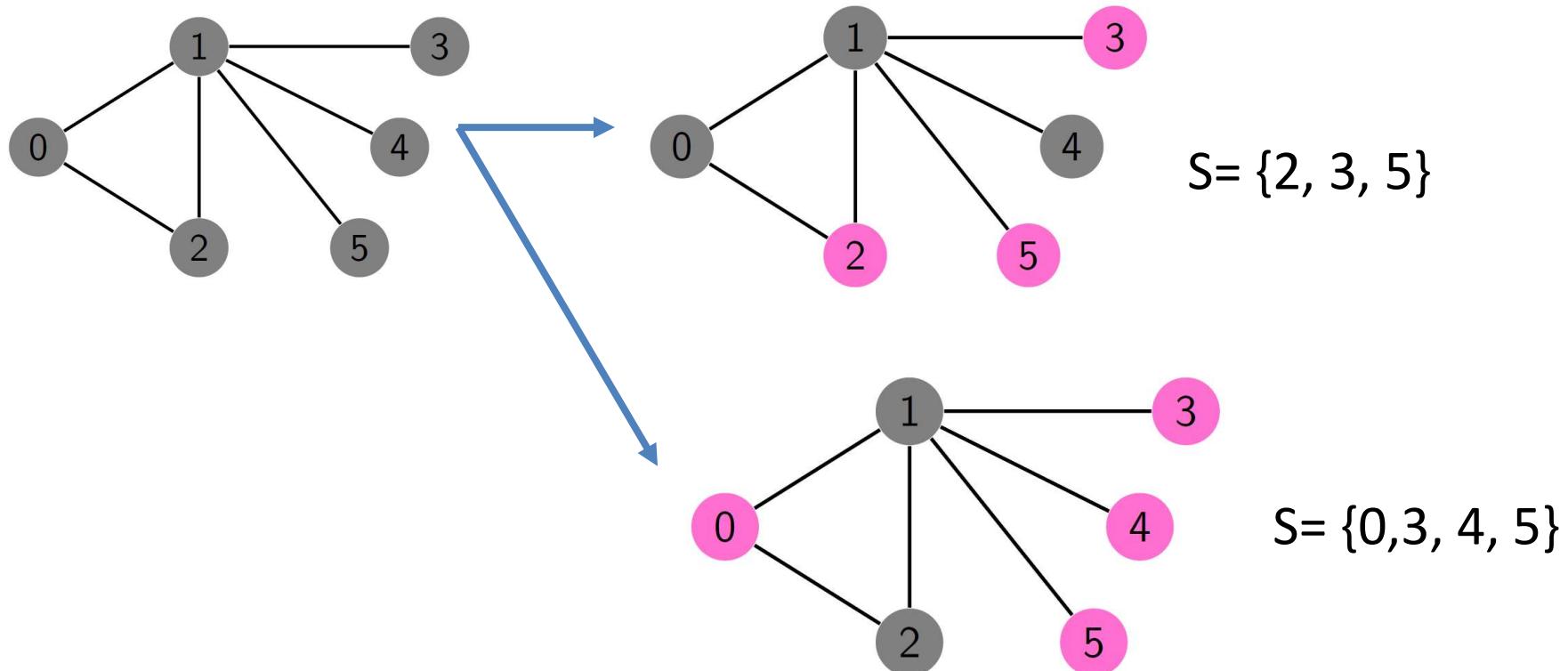
3.4. Bài toán tập độc lập

3.5. Bài toán ghép cặp

3.6. Bài toán tô màu đồ thị

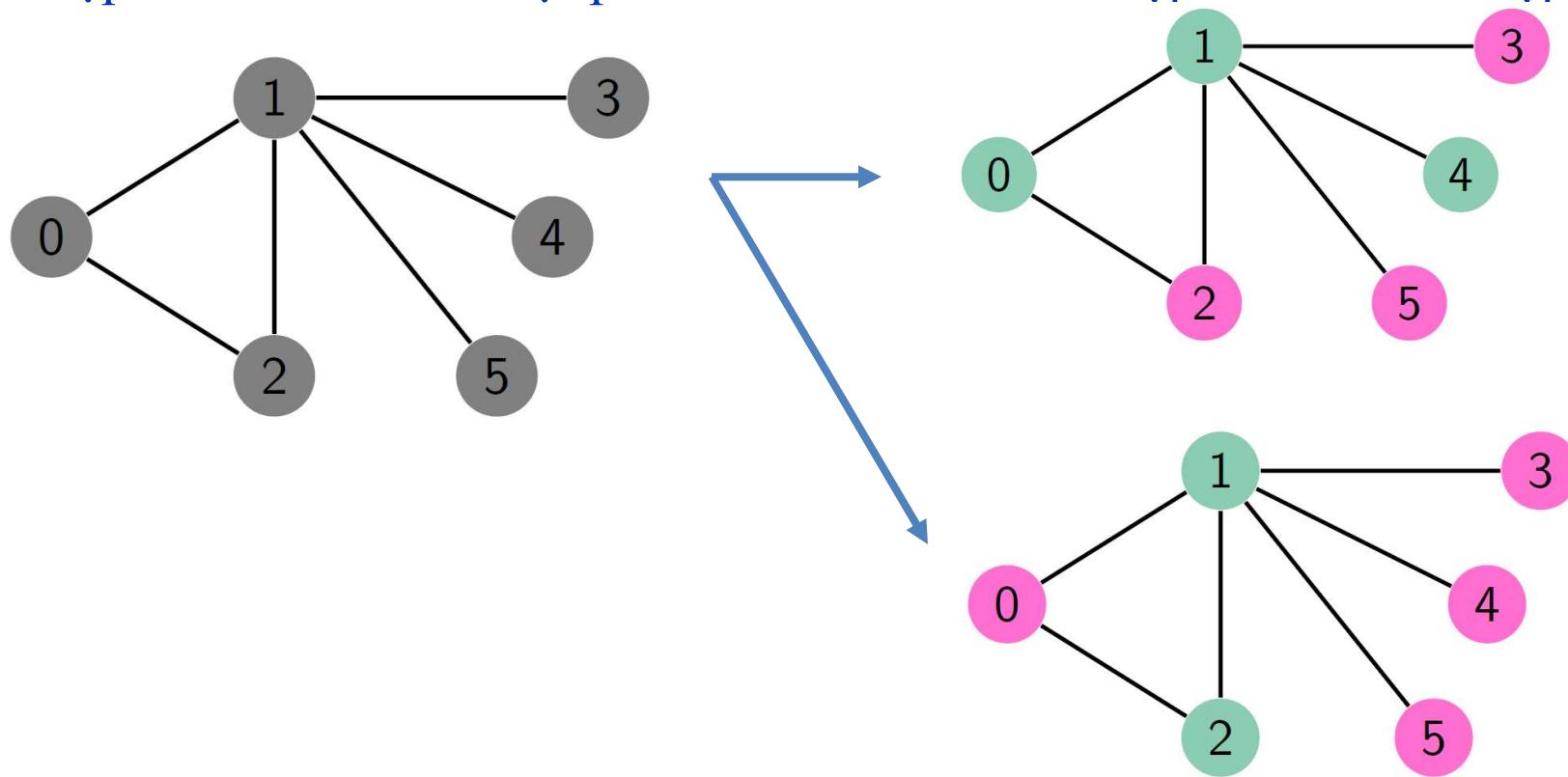
3.4. Bài toán tập độc lập (Independent Set)

- Tập độc lập của đồ thị vô hướng $G=(V,E)$ là tập con S của V , sao cho không có hai đỉnh u, v nào trong S kề với nhau trong G .
- Bài toán IS: Tìm tập độc lập với lực lượng lớn nhất của G .
- IS là NP-khó khi đồ thị là bất kỳ.



Mối quan hệ giữa Tập phủ đỉnh và Tập độc lập

- Cho đồ thị vô hướng $G=(V,E)$
- Tập con S của V là một phủ đỉnh khi và chỉ khi tập bù của nó là tập độc lập.



- Lực lượng của một phủ đỉnh có kích thước nhỏ nhất + lực lượng của tập độc lập có kích thước lớn nhất = số đỉnh của đồ thị

3. Một số bài toán cơ bản trên đồ thị

3.1. Bài toán cây khung nhỏ nhất

3.2. Bài toán đường đi ngắn nhất

3.3. Bài toán phủ đỉnh

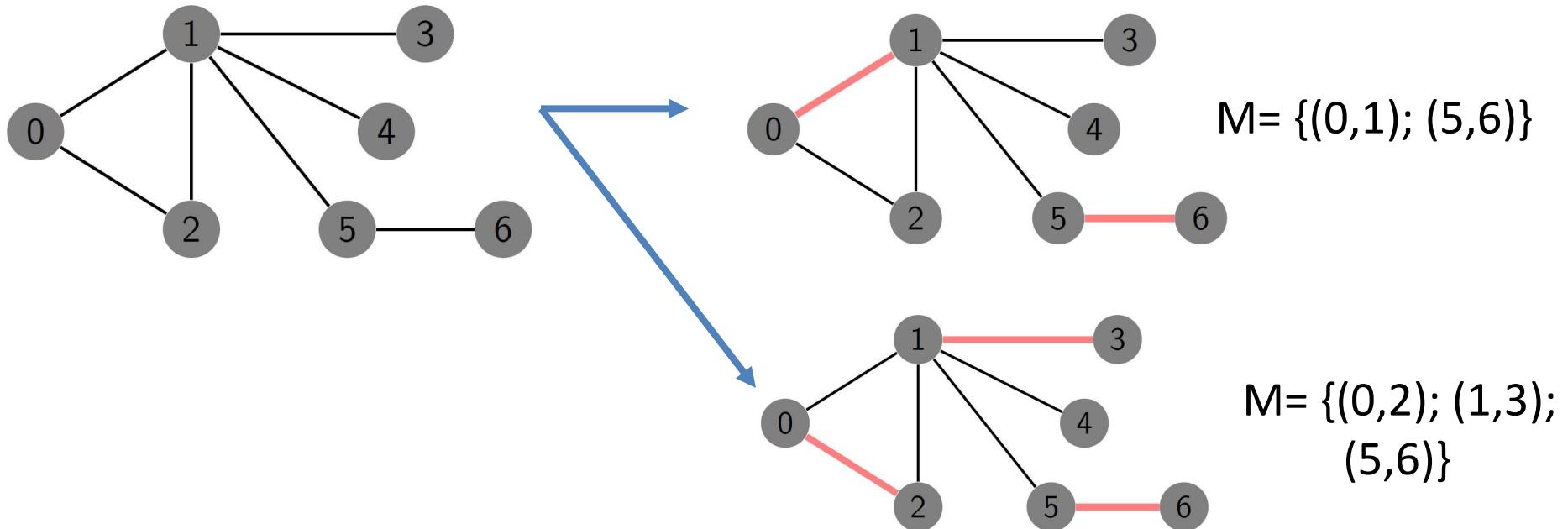
3.4. Bài toán tập độc lập

3.5. Bài toán ghép cặp

3.6. Bài toán tô màu đồ thị

3.5. Bài toán ghép cặp

- Cặp ghép M trên đồ thị vô hướng $G = (V, E)$ là tập các cạnh của đồ thị trong đó không có hai cạnh nào có đỉnh chung.



- Cặp ghép với lực lượng lớn nhất được gọi là *cặp ghép cực đại*.
- Cặp ghép với trọng lượng lớn nhất được gọi là *cặp ghép lớn nhất* (trong trường hợp đồ thị có trọng số).
- Có thuật toán $O(|V|^4)$ cho đồ thị tổng quát

3. Một số bài toán cơ bản trên đồ thị

3.1. Bài toán cây khung nhỏ nhất

3.2. Bài toán đường đi ngắn nhất

3.3. Bài toán phủ đỉnh

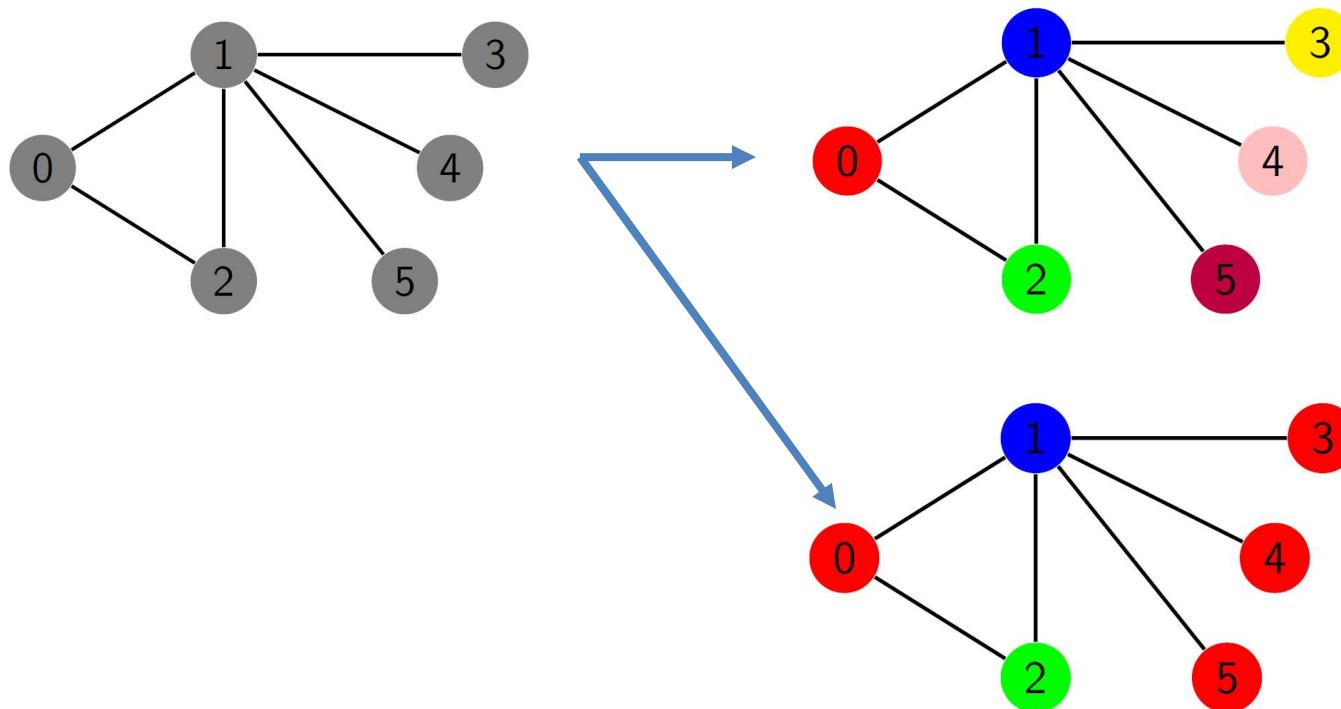
3.4. Bài toán tập độc lập

3.5. Bài toán ghép cặp

3.6. Bài toán tô màu đồ thị

3.6. Bài toán tô màu đồ thị

- Cho đồ thị vô hướng $G = (V, E)$.
- Tô màu đồ thị G là việc gán các màu cho các đỉnh của đồ thị sao cho không có 2 đỉnh nào kề nhau được tô cùng màu



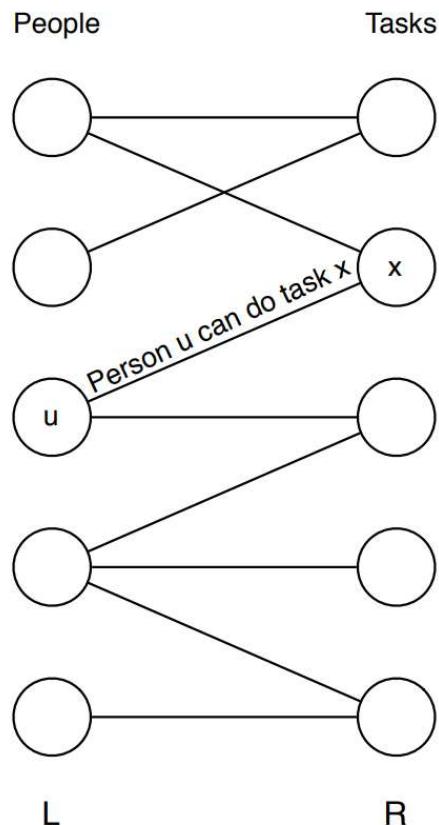
- Hãy tìm cách tô sao cho số màu sử dụng là ít nhất.
- Là bài toán NP-khó trên đồ thị bất kỳ

Xét trường hợp đặc biệt trên đồ thị hai phía

Các bài toán 3.3, 3.4, 3.6 là các bài toán NP-khó cho đồ thị bất kỳ. Giờ ta sẽ xét trên đồ thị đặc biệt: đồ thị hai phía

Ghép cặp trên đồ thị hai phía

Cặp ghép với lực lượng lớn nhất được gọi là *cặp ghép cực đại*



- Tập công nhân: L
 - Tập công việc: R
 - Mỗi người $i \in L$ chỉ có thể làm một số việc $j \in R$
- Hãy gán việc cho mỗi công nhân (mỗi người nhận 1 việc, mỗi việc được thực hiện bởi 1 công nhân): sao cho số lượng công nhân được nhận việc là nhiều nhất có thể

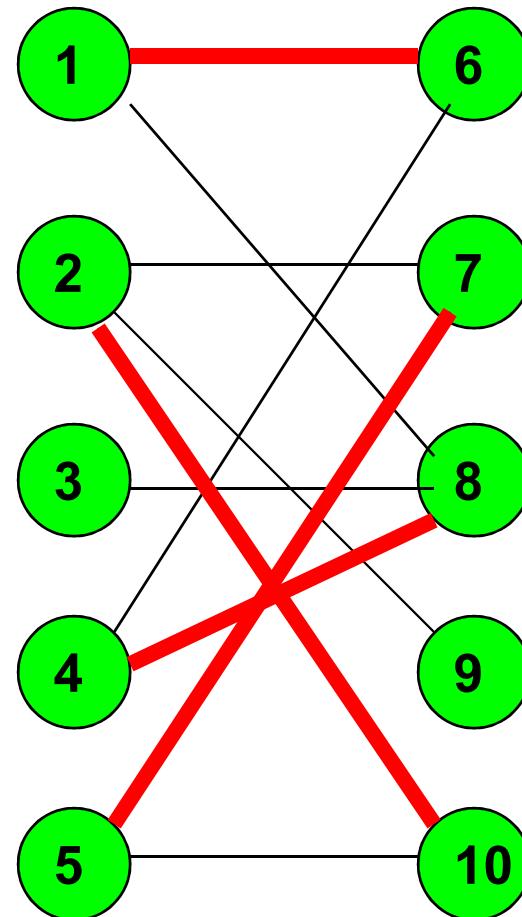
Ghép cặp trên đồ thị hai phía

Xét đồ thị hai phía

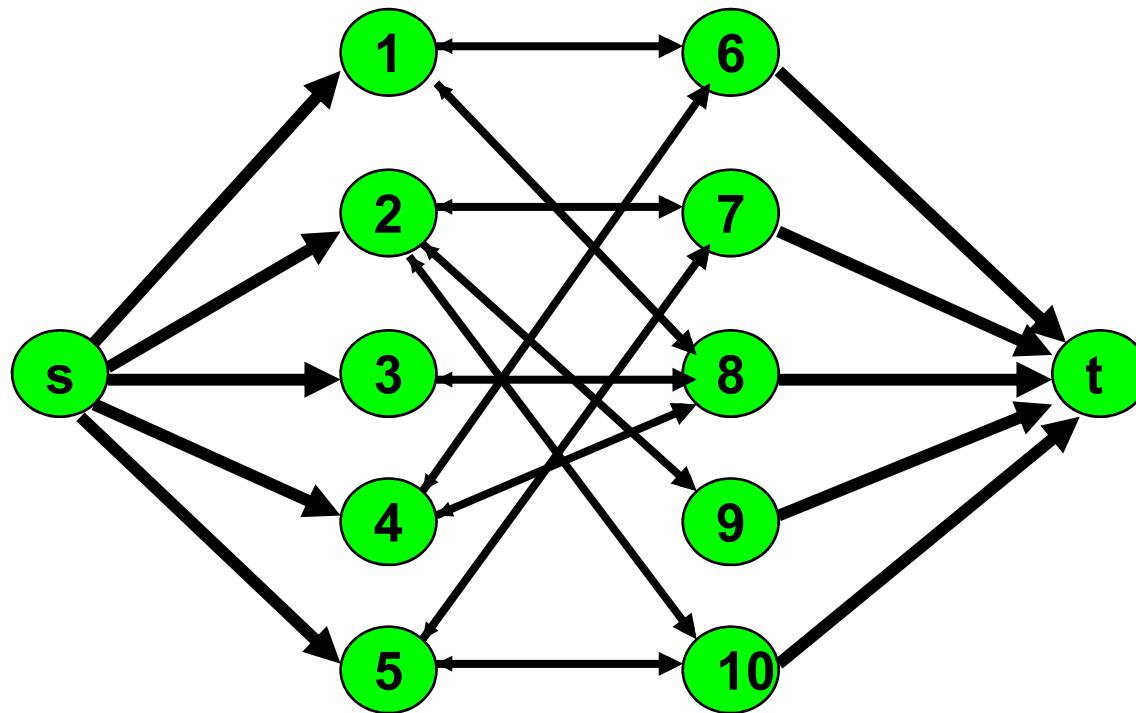
$$G = (X \cup Y, E)$$

Cặp ghép là tập cạnh mà không
có hai nào có chung tập đỉnh

Bài toán: Tìm cặp ghép có kích
thước lớn nhất



Ghép cặp trên đồ thị hai phía: Qui về Bài toán luồng cực đại



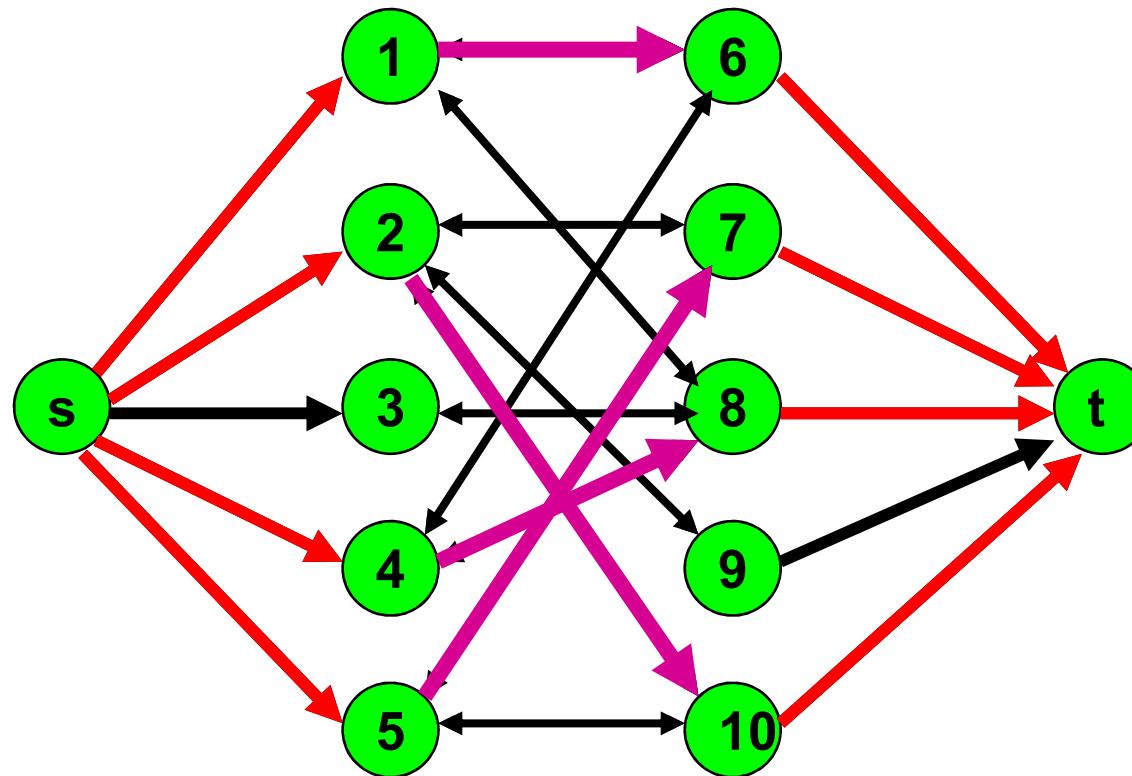
Thêm 2 đỉnh: s và t

Mỗi cung (s, i) có kntq 1.

Mỗi cung (j, t) có kntq 1.

Mỗi cạnh được thay thế
bởi cung có kntq 1.

Tìm luồng cực đại: thuật toán Ford-Fulkerson

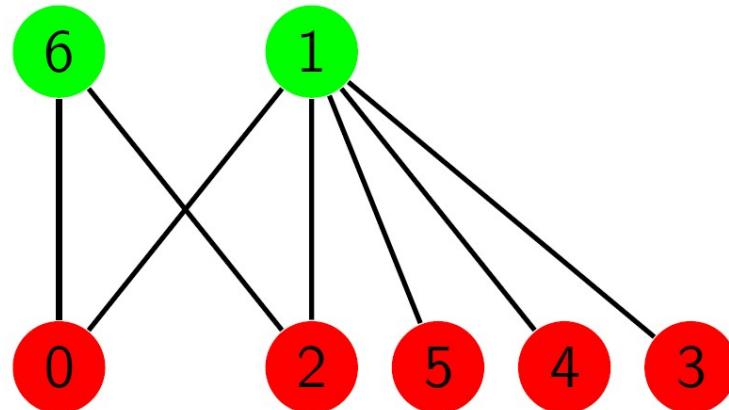


Luồng cực đại từ $s \rightarrow t$ có giá trị 4.

Cặp ghép cực đại có kích thước 4.

Bài toán tô màu trên đồ thị hai phía

Bài toán tô màu: làm thế nào để tô ít màu nhất trên đồ thị hai phía?



Trả lời: mỗi phía tô bởi 1 màu; tổng cộng sử dụng 2 màu

Bài toán tập độc lập trên đồ thị hai phía

Định lý König:

- Định lý chỉ ra rằng: lực lượng của một phủ đindh nhỏ nhất trên đồ thị hai phía bằng với lực lượng của ghép cặp lớn nhất trên đồ thị đó

Do đó, để tìm một phủ đindh nhỏ nhất trên đồ thị hai phía, ta chỉ cần tìm ghép cặp lớn nhất.

Và do lực lượng của tập độc lập lớn nhất chính là tổng số đindh của đồ thị trừ đi lực lượng của phủ đindh nhỏ nhất, nên ta có thể tính được tập độc lập lớn nhất