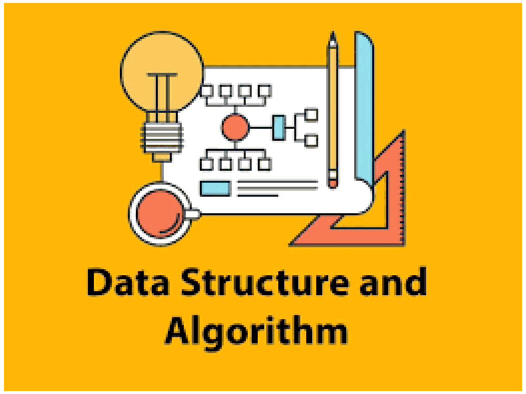
CẤU TRÚC DỮ LIỆU & GIẢI THUẬT

NGÀY 10: CÁC BÀI TOÁN TRÊN ĐỒ THỊ



Giảng viên: Th.S Bùi Văn Kiên

NỘI DUNG

- Tìm đường đi ngắn nhất: Floyd, Bellman-Ford, Dijkstra
- Cây bao trùm (cây khung) nhỏ nhất
- Chu trình Euler
- Chu trình Hamilton

1. Đường đi ngắn nhất

Phát biểu bài toán

- Xét đồ thị G=<V, E>; trong đó | V| = n, | E | = m. Với mỗi cạnh (u, v) $\in$ E, ta đặt tương ứng với nó một số thực A[u][v] được gọi là trọng số của cạnh. Ta sẽ đặt A[u,v]= ∞ nếu (u, v) $\notin$ E.
- Nhiệm vụ của bài toán là tìm đường đi ngắn nhất từ một đỉnh xuất phát s $\in$ V (đỉnh nguồn) đến đỉnh cuối t $\in$ V (đỉnh đích).
- Độ dài của đường đi d(s,t) được gọi là khoảng cách ngắn nhất từ s đến t (trong trường hợp tổng quát d(s,t) có thể âm).
- Nếu như không tồn tại đường đi từ s đến t thì độ dài đường đi d(s,t)= ∞.

1. Đường đi ngắn nhất

Phát biểu bài toán

- Trường hợp 1. Nếu đỉnh START cố định → tìm đường đi ngắn nhất từ START đến tất cả các đỉnh còn lại trên đồ thị.
  - Đối với đồ thị có trọng số không âm → Thuật toán Dijkstra.
  - Đối với đồ thị có trọng số âm nhưng không tồn tại chu trình âm → Thuật toán Bellman-Ford.
  - Trong trường hợp đồ thị có chu trình âm, bài toán không có lời giải.
- Trường hợp 2. Nếu START thay đổi và END (đích) cũng thay đổi

→ tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị.

  - Bài toán luôn có lời giải trên đồ thị không có chu trình âm.
  - Đối với đồ thị có trọng số không âm → thực hiện lặp lại n lần thuật toán Dijkstra.
  - Đối với đồ thị không có chu trình âm, bài toán có thể giải quyết bằng thuật toán Floyd.

1. Đường đi ngắn nhất

Tổng quát:

- Đường đi ngắn nhất trên đồ thị không có trọng số:

→ sử dụng BFS

- Đường đi ngắn nhất trên đồ thị có trọng số
  - Thuật toán Bellman-Ford (nếu đồ thị có trọng số âm)

  → với một cặp đỉnh: $O(n^2)$
  - Thuật toán Floyd:

  tìm đường đi ngắn nhất với mọi cặp đỉnh (s, t) → $O(n^3)$
  - Thuật toán Dijkstra

  → với một cặp đỉnh: O(n log n)

1. Đường đi ngắn nhất

Cách biểu diễn đồ thị có trọng số bằng danh sách kề:

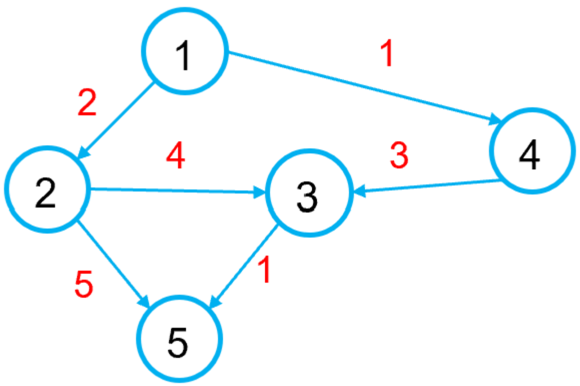
- Sử dụng vector<int> adj cho danh sách kề

- Sử dụng vector<pair<int,int> > adj (pair của STL) với trường hợp có trọng số như sau:

  adj[u].first = v and adj[u].second = cost/weight

- Thêm 1 phần tử vào danh sách cạnh:
  - adj[u].push_back(make_pair(v, cost))
  - or adj[u].push_back(II(v, cost)
  - or adj[u].push_back({v, cost})

1.1 Thuật toán BellmanFord

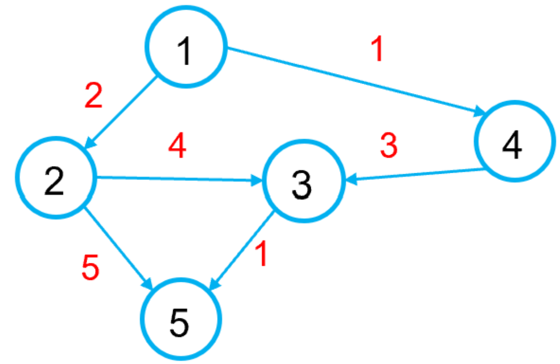


Source = 1
5 6
2 5 5
2 3 4
3 5 1
4 3 3
1 2 2
1 4 1

```
function BellmanFord
    for each u:
        dist[u] = infi
        parent[u] = -1
    dist[source] = 0

    // optimization
    for i from 1 to |V|-1:
        for each edge (u,v):
            if dist[v] > dist[u] + w[u][v]:
                dist[v] = dist[u] + w[u][v]
                parent[v] = u

    // check negative cycle
    for each (u,v) in edge_list:
        if dist[v] > dist[u] + w[u][v]:
            error "has a negative cycle"
```

1.1 Thuật toán BellmanFord

- Độ phức tạp O(E*V)
- Chứng minh:
  - Sau mỗi một vòng lặp, có ít nhất 1 đỉnh được tối ưu
  - Sau |V|-1 lần lặp → |V| đỉnh được tối ưu



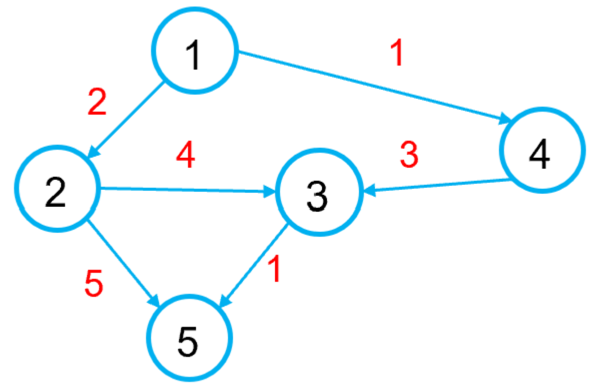
Source = 1
5 6
2 5 5
2 3 4
3 5 1
4 3 3
1 2 2
1 4 1

```
function BellmanFord
    for each u:
        dist[u] = infi
        parent[u] = -1
    dist[source] = 0

    // optimization
    for i from 1 to |V|-1:
        for each edge (u,v):
            if dist[v] > dist[u] + w[u][v]:
                dist[v] = dist[u] + w[u][v]
                parent[v] = u

    // check negative cycle
    for each (u,v) in edge_list:
        if dist[v] > dist[u] + w[u][v]:
            error "has a negative cycle"
```

1.1 Thuật toán BellmanFord

Source = 1
5 6
2 5 5
2 3 4
3 5 1
4 3 3
1 2 2
1 4 1

i = 1
Edge 1: do no thing
Edge 2: do no thing
Edge 3: do no thing
Edge 4: do no thing
Edge 5: dist[2] = 2, p[2] = 1
Edge 6: dist[4] = 1, p[4] = 1

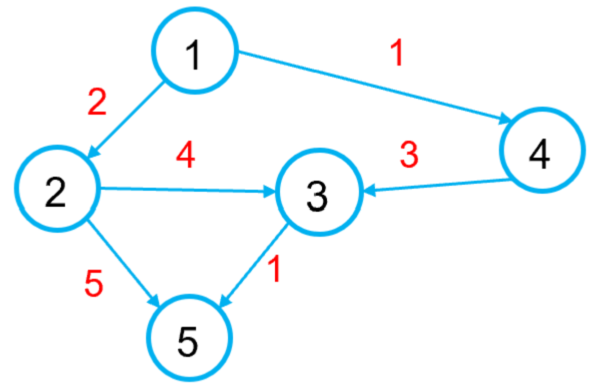



```
function BellmanFord
    for each u:
        dist[u] = infi
        parent[u] = -1
    dist[source] = 0

    // optimization
    for i from 1 to |V|-1:
        for each edge (u,v):
            if dist[v] > dist[u] + w[u][v]:
                dist[v] = dist[u] + w[u][v]
                parent[v] = u

    // check negative cycle
    for each (u,v) in edge_list:
        if dist[v] > dist[u] + w[u][v]:
            error "has a negative cycle"
```

1.1 Thuật toán BellmanFord

Source = 1
5 6
2 5 5
2 3 4
3 5 1
4 3 3
1 2 2
1 4 1

i = 2
Edge 1: dist[5] = 7, p[5] = 2
Edge 2: dist[3] = 6, p[3] = 2
Edge 3: do nothing
Edge 4: dist[3] = 4, p[3] = 4
Edge 5: do nothing
Edge 6: do nothing

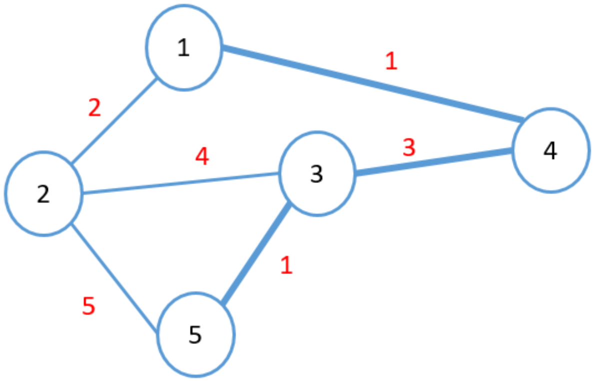


i = 3
Edge 1: do nothing
Edge 2: do nothing
Edge 3: dist[3] = 4, dist[5] = 7
→ dist[5] = 5, p[5] = 3
Edge 4: …
Edge 5: ..
Edge 6: …

i = 4: do nothing

```
function BellmanFord
    for each u:
        dist[u] = infi
        parent[u] = -1
    dist[source] = 0

    // optimization
    for i from 1 to |V|-1:
        for each edge (u,v):
            if dist[v] > dist[u] + w[u][v]:
                dist[v] = dist[u] + w[u][v]
                parent[v] = u

    // check negative cycle
    for each (u,v) in edge_list:
        if dist[v] > dist[u] + w[u][v]:
            error "has a negative cycle"
```

1.1 Thuật toán BellmanFord



➢ **Truy vết tìm đường đi**

Đường đi ngắn nhất 1→ 5
Parent[1] = -1
Parent[4] = 1
Parent[3] = 4
Parent[5] = 3

u = 5
Path.push_back(5), Parent[5] = 3 → u = 3
Path.push_back(3), Parent[3] = 4 → u = 4
Path,push_back(4), Parent[4] = 1
Path.push_back(1), Parent[1] = -1
5 -> 3 -> 4 -> 1

1.2 Thuật toán Floyd

- Tìm đường đi ngắn nhất của V cặp đỉnh
- Ý tưởng: đường đi ngắn nhất từ u→v được tối ưu hóa khi đi qua mọi đỉnh k
- Độ phức tạp $O(V^3)$

```
void floydWarshall(i) {
    // Initialize
    // D[u][v] = infinity for all u, v
    // D[u][u] = 0
    // D[u][v] = cost[u][v] for each edge (u,v)

    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            trace[u][v] = u;
        }
    }

    for (int k = 0; k < n; k++) {
        for (int u = 0; u < n; u++) {
            for (int v = 0; v < n; v++) {
                if (D[u][v] > D[u][k] + D[k][v]) {
                    D[u][v] = D[u][k] + D[k][v];
                    trace[u][v] = trace[k][v];
                }
            }
        }
    }
}
```

1.3 Thuật toán Dijkstra

- Giống như thuật toán Bellman-Ford, thuật toán Dijkstra cũng tối ưu hóa đường đi bằng cách xét các cạnh (u,v), và so sánh hai đường đi S→v sẵn có với đường đi S→u→v.

- Thuật toán sẽ duy trì đường đi ngắn nhất đến tất cả các đỉnh. Ở mỗi bước, chọn đường đi S→u có tổng trọng số nhỏ nhất trong tất cả các đường đi đã được duy trì. Sau đó tiến hành tối ưu các đường đi S→v bằng cách thử kéo dài thành S→u→v.
- Đpt trường hợp cơ bản: O(V^2 + E)

1.3 Thuật toán Dijkstra

Code C++ dpt O(N^2)

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] = INFINITY
        parent[v] = -1
    add source to Q
    dist[source] = 0

    while Q is not empty:
        u = vertex in Q with dist[u] is minimum
        remove u from Q

        for each neighbor v of u:
            tmp = dist[u] + Edges(u, v).cost
            if tmp < dist[v]:
                dist[v] = tmp
                parent[v] = u
                push v to Q

    return dist[], parent[]
```

1.3 Thuật toán Dijkstra

- Cải tiến:
  - Sử dụng priority_queue hoặc set
  - priority_queue<ii, vector<ii>, greater<ii>> Q;

  với ii là pair<int, int>
  - Thao tác: Q.push, Q.top, Q.pop (lấy ra phần tử nhỏ nhất trong queue)
  - Lấy 1 đỉnh u với dist[u] nhỏ nhất trong O(log n)

- Độ phức tạp: O (E log V)

2. Cây khung

- Khái niệm cây khung
- Cây khung nhỏ nhất
- Thuật toán Prim
- Thuật toán Krusal

2.1 Cây khung

- Định nghĩa
  - Cây là đồ thị vô hướng liên thông không có chu trình.
  - Đồ thị không liên thông được gọi là rừng.
- Định lý: Giả sử T= <V, E> là đồ thị vô hướng n đỉnh. Khi đó những khẳng định sau là tương đương:
  - T là một cây;
  - T không có chu trình và có n-1 cạnh;
  - T liên thông và có đúng n-1 cạnh;
  - T liên thông và mỗi cạnh của nó đều là cầu;
  - Giữa hai đỉnh bất kỳ của T được nối với nhau bởi đúng một đường đi đơn;
  - T không chứa chu trình nhưng hễ cứ thêm vào nó một cạnh ta thu được đúng một chu trình.

2.1 Cây khung

Xây dựng cây khung bằng DFS

```
int n, m, st, des, visitedEdgeCnt = 0;
int visited[maxN];
vector<int> List[maxN];
vector<II> spanningTree;

void DFS(int u) {
    visited[u] = true;
    for(int i = 0; i < List[u].size(); i++) {
        int v = List[u][i];
        if(visited[v] == false) {
            visitedEdgeCnt++;
            spanningTree.push_back(make_pair(u, v));
            DFS(v);
        }
    }
}
```

- visitedEdgeCnt = n-1 → tìm được cây khung
- visitedEdgeCnt < n-1 → đồ thị không liên thông

2.1 Cây khung

Xây dựng cây khung bằng BFS

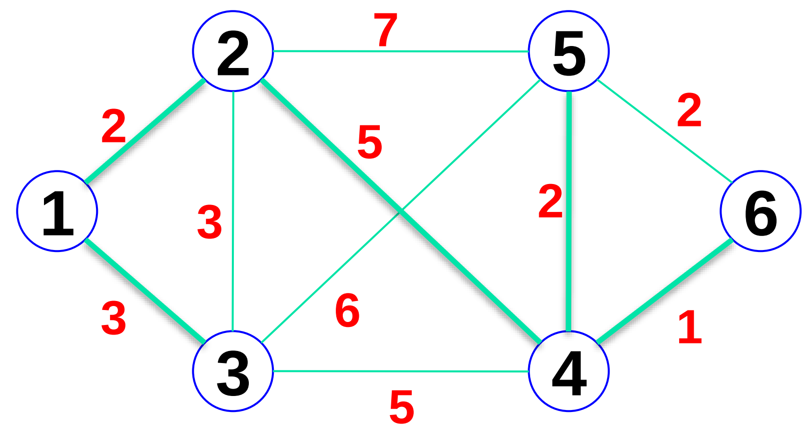
2.2 Cây khung nhỏ nhất

- Cho G= <V, E> là đồ thị vô hướng liên thông với tập đỉnh V = {1, 2, . . ., n } và tập cạnh E gồm m cạnh. Mỗi cạnh e của đồ thị được gán với một số không âm c(e) được gọi là trọng số của cạnh.
- Giả sử H= <V, T> là một cây khung của đồ thị G. Ta gọi chi phí cost(H) của cây khung H được tính bằng tổng độ dài các cạnh.
- Bài toán được đặt ra là, trong số các cây khung của đồ thị hãy tìm cây khung có độ dài nhỏ nhất.

  (Minimum spanning tree)

2.2 Cây khung nhỏ nhất

- Có nhiều cây khung nhỏ nhất
- In ra một cấu hình bất kì



Input:

N = 6, M = 10

1	2	2
1	3	3
2	3	3
2	4	5
2	5	7
3	4	5
3	5	6
4	5	2
4	6	1
5	6	2

Output:

Cost(H) = 13

4 5 2

4 6 1

1 2 2

1 3 3

2 4 5

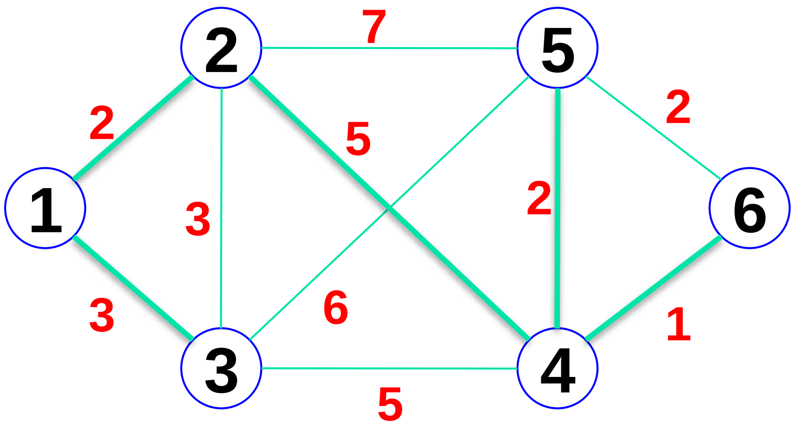
2.2 Cây khung nhỏ nhất

- **So sánh**

	Prim	Krusal
Cách tiếp cận	Vertex-base	Edge-base
CTDL	Priority queue	Disjoint set
Biểu diễn đồ thị	Ma trận kề hoặc danh sách kề	Danh sách cạnh
Khởi tạo	Xuất phát ở 1 đỉnh bất kì	Các đỉnh riêng biệt, chưa có cạnh nào
Phù hợp với	Đồ thị dày	Đồ thị thưa
Độ phức tạp	O((E+V) log V)	O(E log E) hoặc O(E log V)

2.3 Cây khung nhỏ nhất – Prim

- Thuật toán tham lam




```
// Input: A weighted connected graph G = {V, E}
// Output: MST, the set of edges composing a minimum spanning tree
// Start at any vertex

VT = {v0}
MST = empty
for i = 1 to |V| - 1 do
        find a minimum-weight edge e = (u, v) among all the edges
        such that u is in (VT) and v is in (V - VT)
        VT = VT ∪ {v}
        MST = MST ∪ {e}
return MST[]
```

2.4 Cây khung nhỏ nhất – Krusal

- Ý tưởng: giải thuật tham lam

Ban đầu mỗi đỉnh là một cây riêng biệt, xây dựng MST bằng cách duyệt các cạnh theo trọng số từ nhỏ đến lớn rồi hợp nhất các cây lại với nhau.

Cụ thể:

- Sort các cạnh theo thứ tự trọng số tăng dần (tập E)
- Thực hiện phép lặp sau |E|-1 lần
  - Chọn cạnh u-v nhỏ nhất trong tập E
  - Nếu cạnh tạo thành chu trình (kiểm tra bằng DFS/BFS or DSU)

  → loại bỏ
  - Ngược lại, thêm u-v vào cây khung

2.4 Cây khung nhỏ nhất – Krusal

Kiểm tra có tạo chu trình

Thuật toán Kruskal:

Begin

  Bước 1 (Khởi tạo):

    T = ∅; *//Khởi tạo tập cạnh cây khung là ∅*

    d(H) = 0; *//Khởi tạo độ dài nhỏ nhất cây khung là 0*

  Bước 2 (Sắp xếp):

    <Sắp xếp các cạnh của đồ thị theo thứ tự giảm dần của trọng số>;

  Bước 3 (Lặp):

    while (|T<n-1| && E≠∅) do { *// Lặp nếu E≠∅ và |T|<n-1*

      e = <Cạnh có độ dài nhỏ nhất>;

      E = E \ {e}; *//Loại cạnh e ra khỏi đồ thị*

      if (T∪{e} không tạo nên chu trình) then {

        T = ∪{e}; *// Kết nạp e vào tập cạnh cây khung*

        d(H) = d(H) + d(e); *//Độ dài của tập cạnh cây khung*

      endif;

    endwhile;

  Bước 4 (Trả lại kết quả):

    if (|T| < n-1) then <Đồ thị không liên thông>;

    else

      Return(T, d(H));

end.

2.4 CTDL Disjoint-set

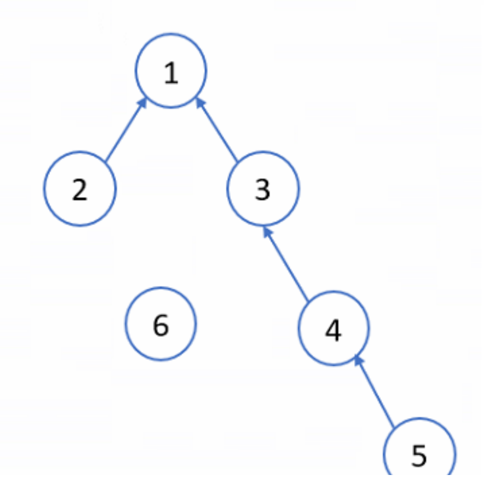
Kiểm tra thêm cạnh có tạo nên chu trình?

- Khởi tạo mỗi đỉnh thành một tập hợp rời rạc riêng biệt.
- Với mỗi cạnh của đồ thị, nối mỗi hai đỉnh (đại diện cho tập hợp) với nhau bởi một cạnh.
- Thêm cạnh (u, v) vào tập cây khung T, kiểm tra có tạo nên chu trình hay không? Tương đương với kiểm tra xem u và v cùng một thành phần liên thông hay không?

- Độ phức tạp:
  - Nếu sử dụng DFS/BFS, mỗi truy vấn cần đpt O(n)
  - Với DSU, mỗi truy vấn ~ O(log n)
- Sử dụng DSU trong thuật toán Krusal.

2.4 CTDL Disjoint-set

- Thuật toán



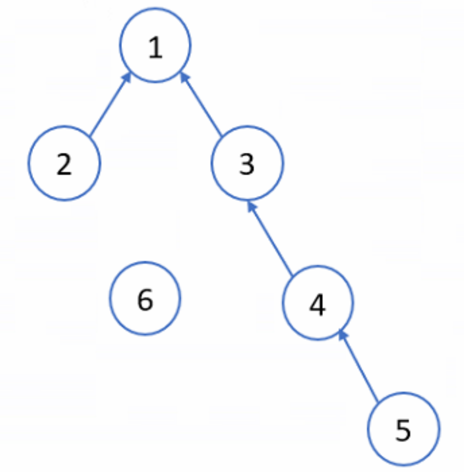
- Ví dụ:
- Union(2, 1) →
- Union(3, 1) →
- Union(4, 3) →
- Union(5, 4) →

- Findset(6)
- Findset(5)

```
FOR(x,1,n) parent[x] = x;

int findSet(int x) {
    if (parent[x] == x) return x;
    else findSet(parent[x]);
}

void Union(int u, int v) {
    int parentU = findSet(u);
    int parentV = findSet(v);
    if (parentU == parentV) return;
    else parent[parentU] = parentV;
}
```

2.4 CTDL Disjoint-set

- Thuật toán

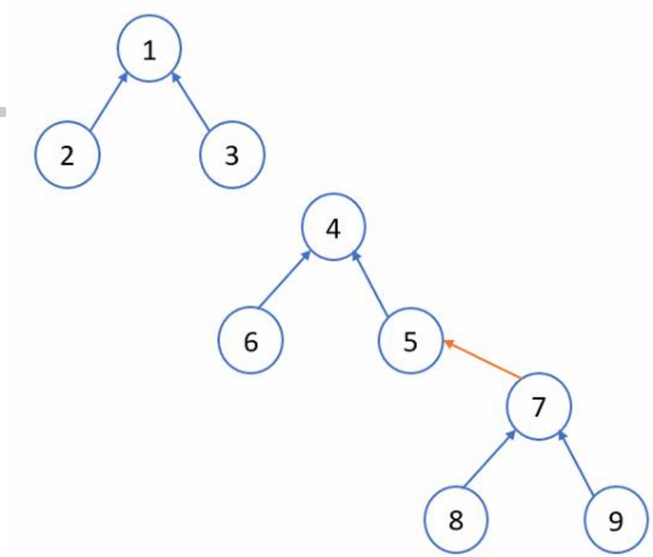


- Ví dụ:
- Union(2, 1) → parent[2] = parent[1] = 1
- Union(3, 1) → parent[3] = parent[1] = 1
- Union(4, 3) → parent[4] = parent[3] = 1
- Union(5, 4) →
  - findSet(5) return 5
  - findSet(4) → findSet(1) = 1
  - Gán parent[5] = 1

- Findset(6) return 6;
- Findset(5) → findset(1) and return 1

```
FOR(x,1,n) parent[x] = x;

int findSet(int x) {
    if (parent[x] == x) return x;
    else findSet(parent[x]);
}

void Union(int u, int v) {
    int parentU = findSet(u);
    int parentV = findSet(v);
    if (parentU == parentV) return;
    else parent[parentU] = parentV;
}
```

2.4 CTDL Disjoint-set



Code gốc

- Union(2,1) → parent[2] = 1
- Union(3,1) → parent[3] = 1
- Union(5,4) → parent[5] = 4
- Union(6, 4) → parent[6] = 4
- Union(8, 7) → parent[8] = 7
- Union(9, 7) → parent[9] = 7
- Union(9, 6) → parent[7] = 4
- Union(9, 3) → parent[4] = 1
- Đã xong thao tác Union(9, 3)

Mỗi lần gọi findSet(9)

Gọi findSet(9) → findSet(7) → findSet(4) → findSet(1) = 1 → TLE

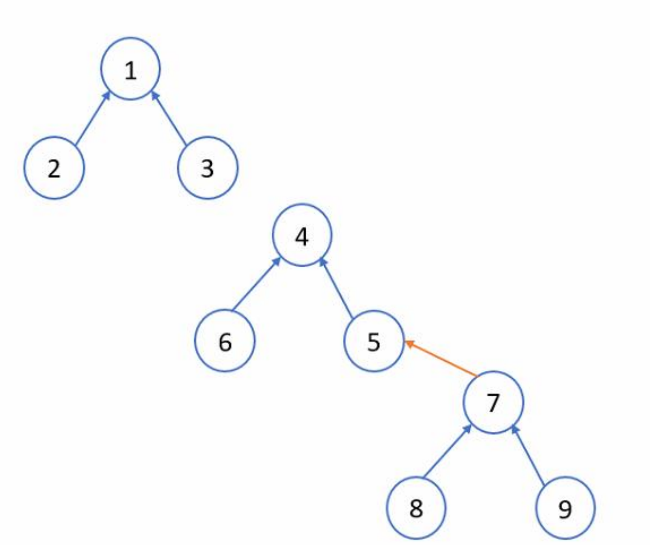
```
FOR(x,1,n) parent[x] = x;

int findSet(int x) {
    if (parent[x] == x) return x;
    else findSet(parent[x]);
}

void Union(int u, int v) {
    int parentU = findSet(u);
    int parentV = findSet(v);
    if (parentU == parentV) return;
    else parent[parentU] = parentV;
}
```

2.4 CTDL Disjoint-set

Cải tiến: Code mới findSet (Path Compression)

Đệ quy có nhớ

→ log(n)



2.4 CTDL Disjoint-set

➢ **Cải tiến 2**

Dùng hạng để giảm độ sâu của cây (rank)

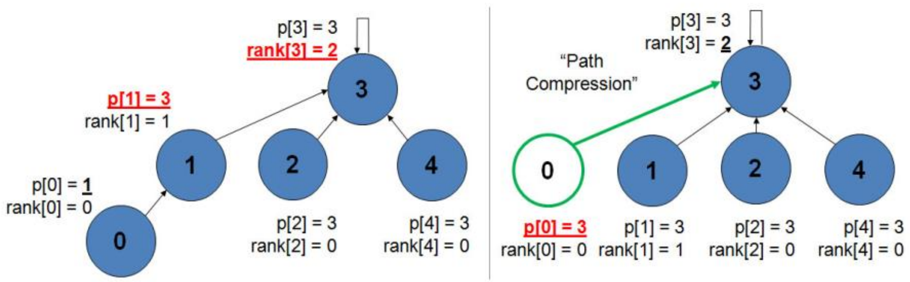
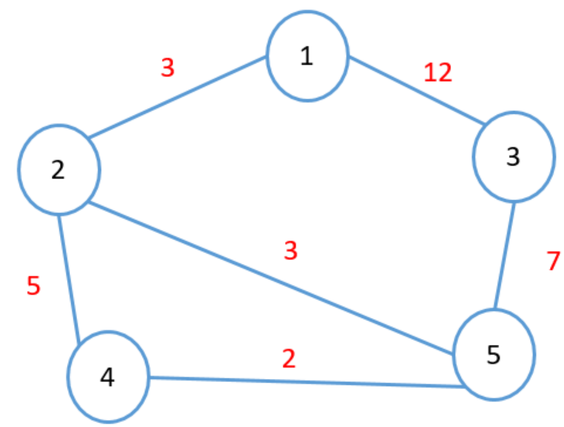


Figure 2.7: `unionSet(0, 3)` → `findSet(0)`

2.4 Cây khung nhỏ nhất – Krusal

N = 4, M = 6
4 6
1 2 3
1 3 12
2 4 5
2 5 3
4 5 2
3 5 7



Running
weight[4-5] = 2
weight[2-5] = 3
weight[1-2] = 3
weight[2-4] = 5
weight[3-5] = 7
weight[1-3] = 12

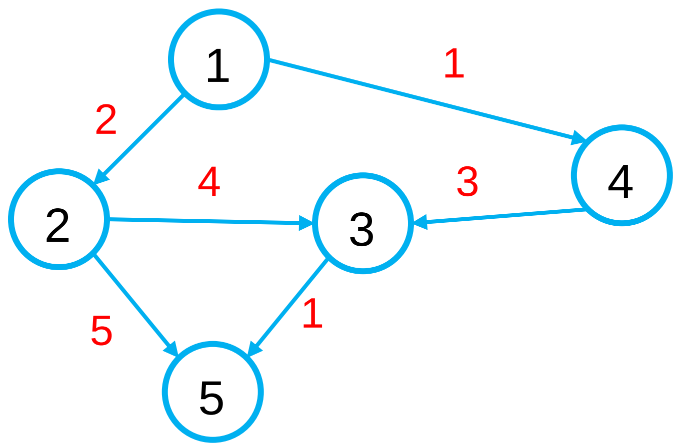
Parent[i] = -1 với mọi i
4-5: Union(4, 5)
2-5: Union(2, 5)
1-2: Union(1, 2)
2-4: không union
3-5: Union(3, 5)
1-3: không union

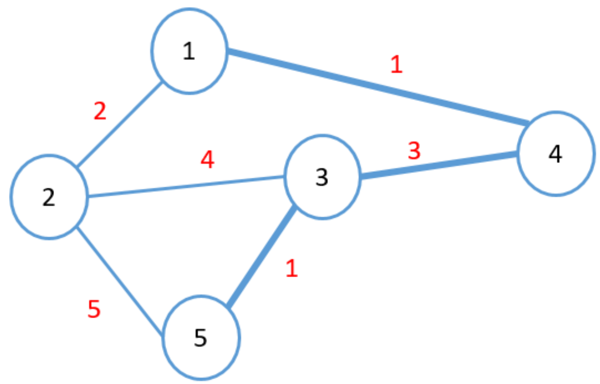
5. Đồ thị Hamilton

- Đpt: O(N!)

```
Thuật toán Hamilton( int k) {
    /* Liệt kê các chu trình Hamilton của đồ thị bằng cách phát triển dãy đỉnh
    (X[1], X[2], . . ., X[k-1] ) của đồ thị G = (V, E) */
    for y∈ Ke(X[k-1]) {
        if (k==n+1) and (y == v0) then
                Ghinhan(X[1], X[2], . . ., X[n], v0);
        else {
                X[k]=y;
                visited[y] = true;
                Hamilton(k+1);
                visited[y] = false;
        }
    }
}
```

5. Đồ thị Hamilton





Source = 1
5 6
2 5 5
2 3 4
3 5 1
4 3 3
1 2 2
1 4 1

QUESTIONS & ANSWERS