# 图论基础算法与应用

文皓

# 综述

- 起源于著名的“戈尼斯堡七桥问题”
- 数据元素之间是多对多的关系

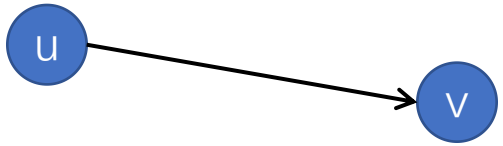
# 本讲目录

# 1.图的基本概念

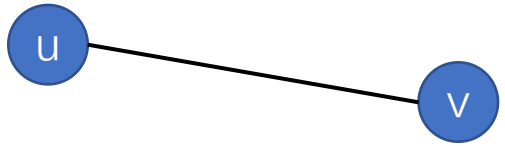
- **图**：二元组 G(V,E)称为图(graph)。 其中，V为结点(node)或顶点(vertex)集。 E为图中结点之间的边的集合
- **点**：用数字0…n-1表示
- **边**：点对(u,v)称为边(edge)或称弧(arc)
  - 对于边(u,v)∈E，u和v邻接(adjacent)，e和u、v关联(incident)
- **子图**(subgraph):边的子集和相关联的点集

# 1.图的基本概念

- 有向边：表示以u为起点，v为终点的边。
- 无向边：表示既能从u到v，又能从v到u的边。




有向边



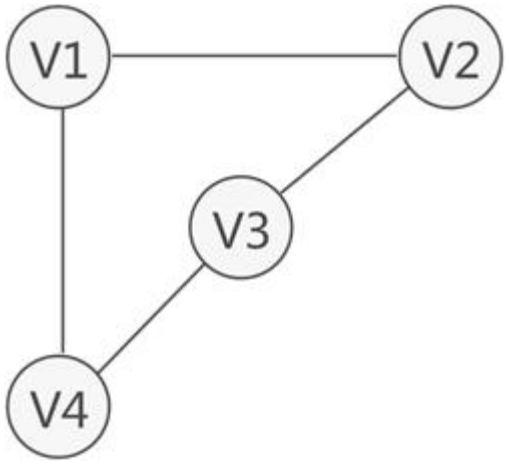

无向边

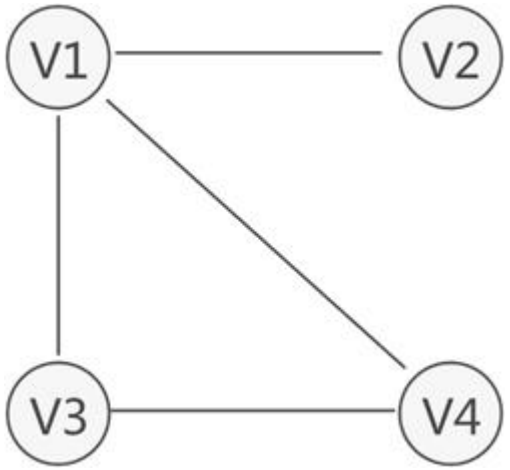
# 1.图的基本概念

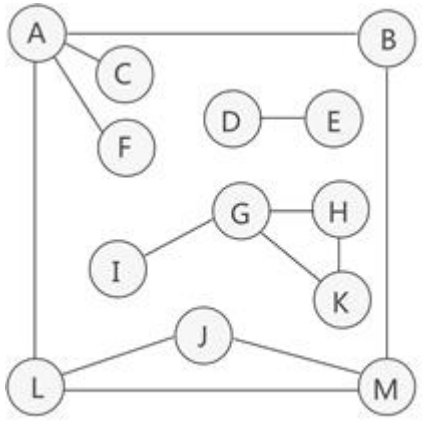
- 有向图：边都是有向边，是单向的
- 无向图：边都是无向边，是双向的
- 带权图：图的边带一个权值，表示具体的含义，例如距离、费用、拥堵程度等等，权值可以是正值，也可以是0或负值
- 图的稠密性：n为顶点数，边数和n(n-1)/2相比非常少的图称为稀疏图，反之为稠密图
- 完全图：边数为n(n-1)/2的图称为完全图

# 1.图的基本概念

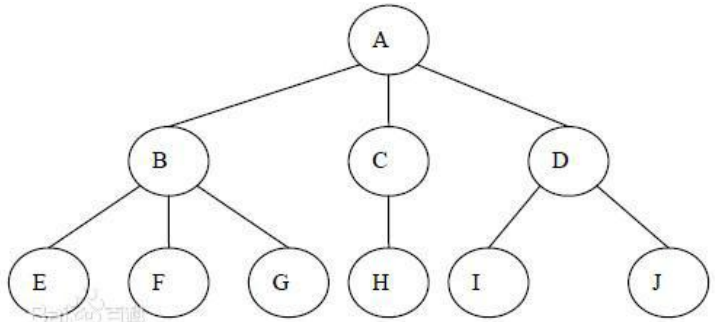
- 点的**度**：图中与某一个点相连的边数称为该点的度
- 连通图：如果图中任意两点都有路径相连，则称图是连通图，否则称图是非连通的
- 树：一种特殊的图，是n个点、n-1条边的连通图(无环连通图)




a) 非连通图

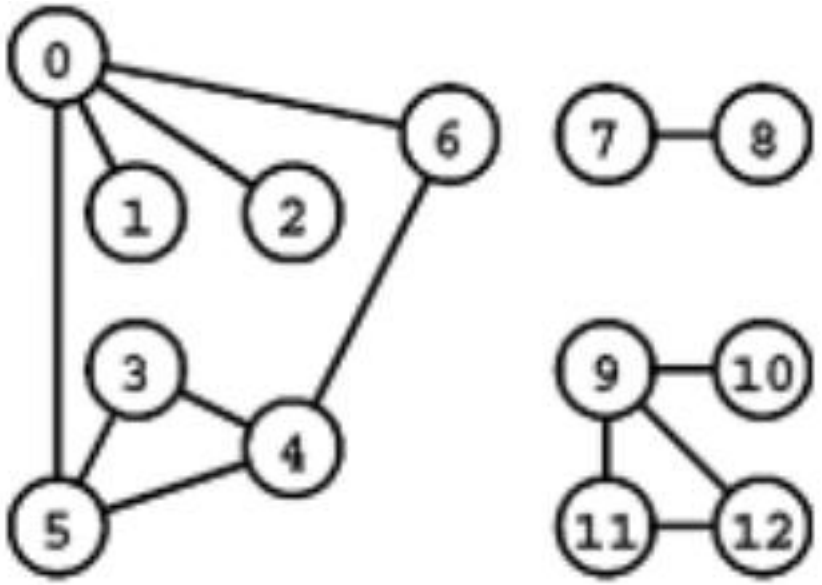


# 2.图的存储方式

- 邻接矩阵存储 （二维数组存储）
- 邻接表存储 （数组模拟链表【较难】、vector写法【较简单】）

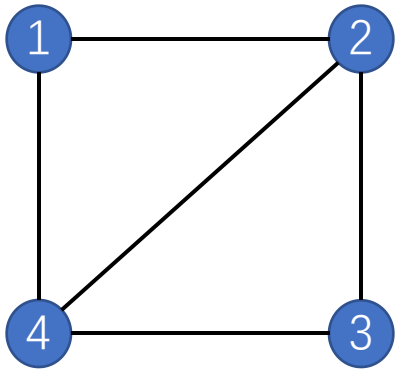
# 2.图的存储方式——输入数据格式说明

- 13 13
- 0 1
- 0 2
- 0 5
- 0 6
- 3 4
- 3 5
- 4 5
- 4 6
- 7 8
- 9 10
- 9 11
- 9 12
- 11 12



# 2.图的存储方式——邻接矩阵

- 设图中有n个点，用一个n*n的矩阵(二维数组)g保存图的信息。
- g[i][j]=以i为起点，j为终点的边数(没有权值)或 g[i][j]=以i为起点，j为终点的权值(无边则为INF)



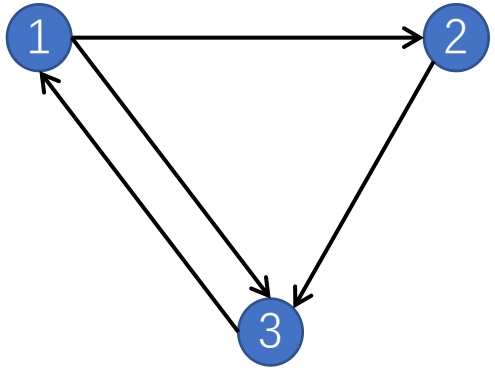

(a)无向图G3

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

(a) G3 的 邻 接 矩 阵

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

(b) G4 的 邻 接 矩




(b)有向图G4

# 2.图的存储方式——邻接矩阵

- 1313
- 0 1
- 0 2
- 0 5
- 0 6
- 3 4
- 3 5
- 4 5
- 4 6
- 7 8
- 9 10
- 9 11
- 9 12
- 11 12

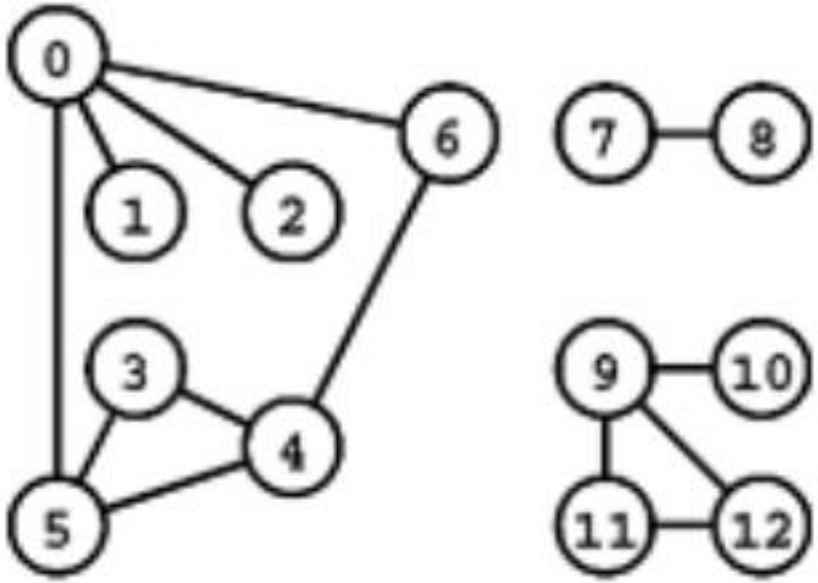
```cpp
#include<iostream>
using namespace std;
const int MAXN = 1005;
int G[MAXN][MAXN],n,e;
// 该图有 n 个点，e 条边
int main(){
    cin>>n>>e;
    for(int i=1;i<=e;++i){
        int u,v;
        cin>>u>>v;
        G[u][v] = 1;
        G[v][u] = 1;
    }
    for(int i=0;i<=n;++i){
        cout<<i<<"的邻接点有: ";
        for(int j=0;j<=n;++j){
            if(G[i][j]==1){
                cout<<j<<" ";
            }
        }
        cout<<endl;
    }
    return 0;
}
```

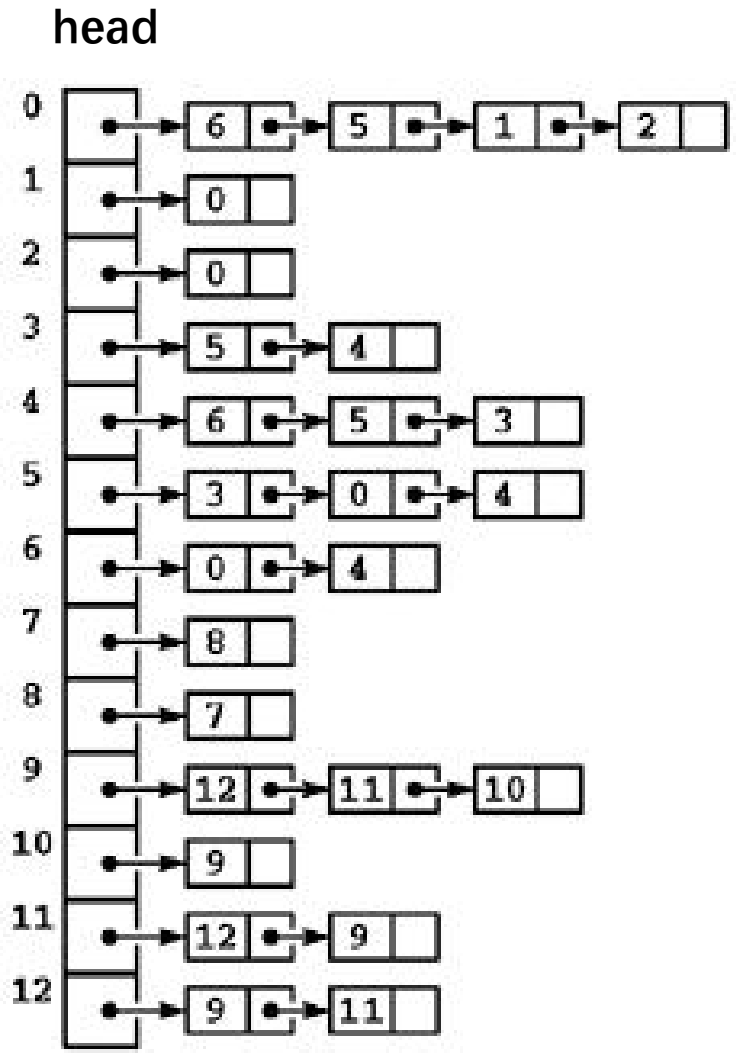
# 2.图的存储方式——邻接矩阵的特点

- 代码实现简单
- 无向图的邻接矩阵是对称的
- 查找和删除某条边是O(1)的
- 遍历某个点的邻居是O(n)的，存储空间是O(n2)的

# 2.图的存储方式——邻接表

- 每个结点的邻接点形成一个链



# 2.图的存储方式——邻接表链表实现（有向图）

```cpp
#include<iostream>
using namespace std;
const int MAXN = 10005,MAXE = 10005<<1;
int head[MAXN],e,n,cnt;//总共n个点 e条边
// cnt 记录当前统计到第几条边
struct Edge{
    int next,v,dis;
};
Edge edge[MAXE];
void add_edge(int u,int v,int dis){
    ++cnt;
    edge[cnt].next = head[u];
    head[u] = cnt;
    edge[cnt].v = v;
    edge[cnt].dis = dis;
}
```

```cpp
int main(){
    cin>>n>>e;
    for(int i=1;i<=e;++i){
        int u,v;
        cin>>u>>v;
        add_edge(u,v,1);
    }
    for(int i=0;i<=n;++i){
        cout<<i<<"的邻接点有：";
        for(int j=head[i];j!=0;j=edge[j].next){
            cout<<edge[j].v<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

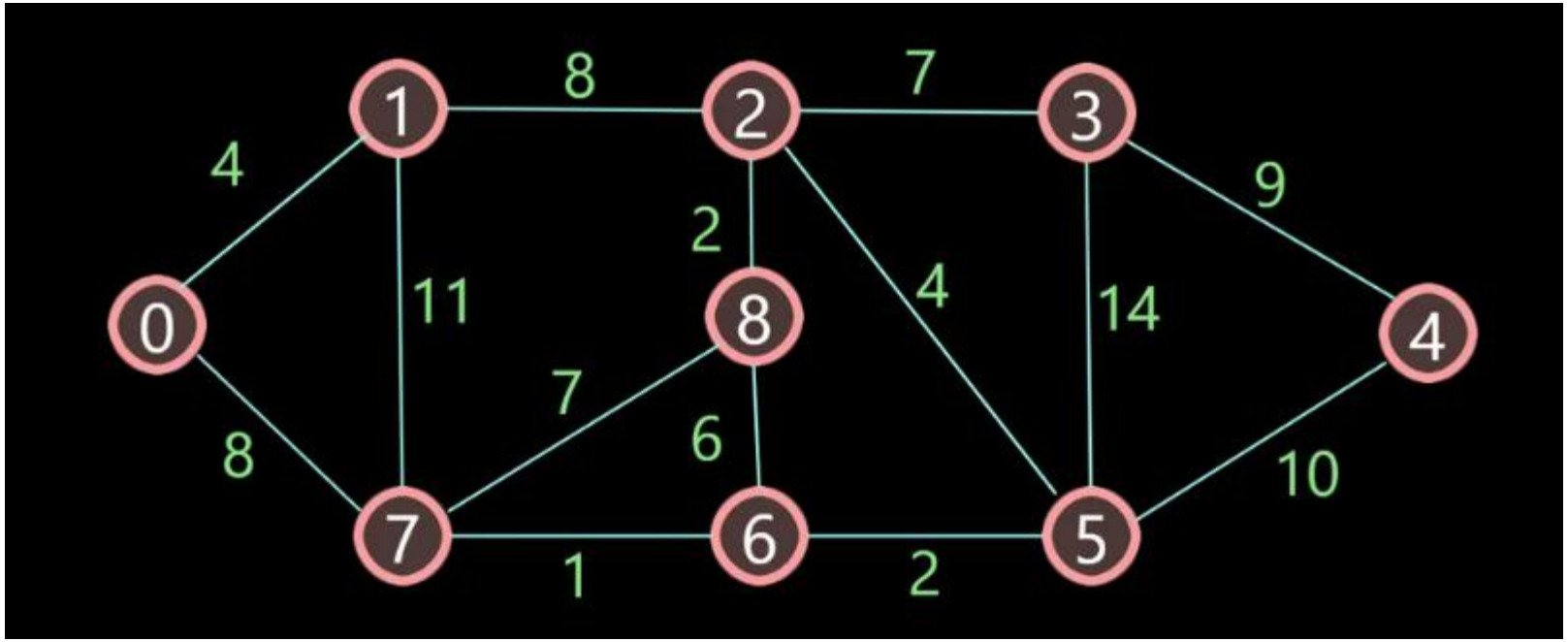
# 2.图的存储方式——邻接表vector实现（有向图）

```cpp
#include<iostream>
#include<vector>
using namespace std;
const int MAXN = 10005;
struct Edge{
    int v,d;
};
vector<Edge> G[MAXN];
void add_edge(int u,int v,int d){
    Edge e;e.v = v;e.d = d;
    G[u].push_back(e);
}
int n,e;
```

```cpp
int main(){
    cin>>n>>e;
    for(int i=1;i<=e;++i){
        int u,v;
        cin>>u>>v;
        add_edge(u,v,1);
        add_edge(v,u,1);
    }
    for(int i=0;i<=n;++i){
        int sz = G[i].size();
        cout<<i<<"的邻接点有: ";
        for(int j=0;j<sz;++j){
            cout<<G[i][j].v<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

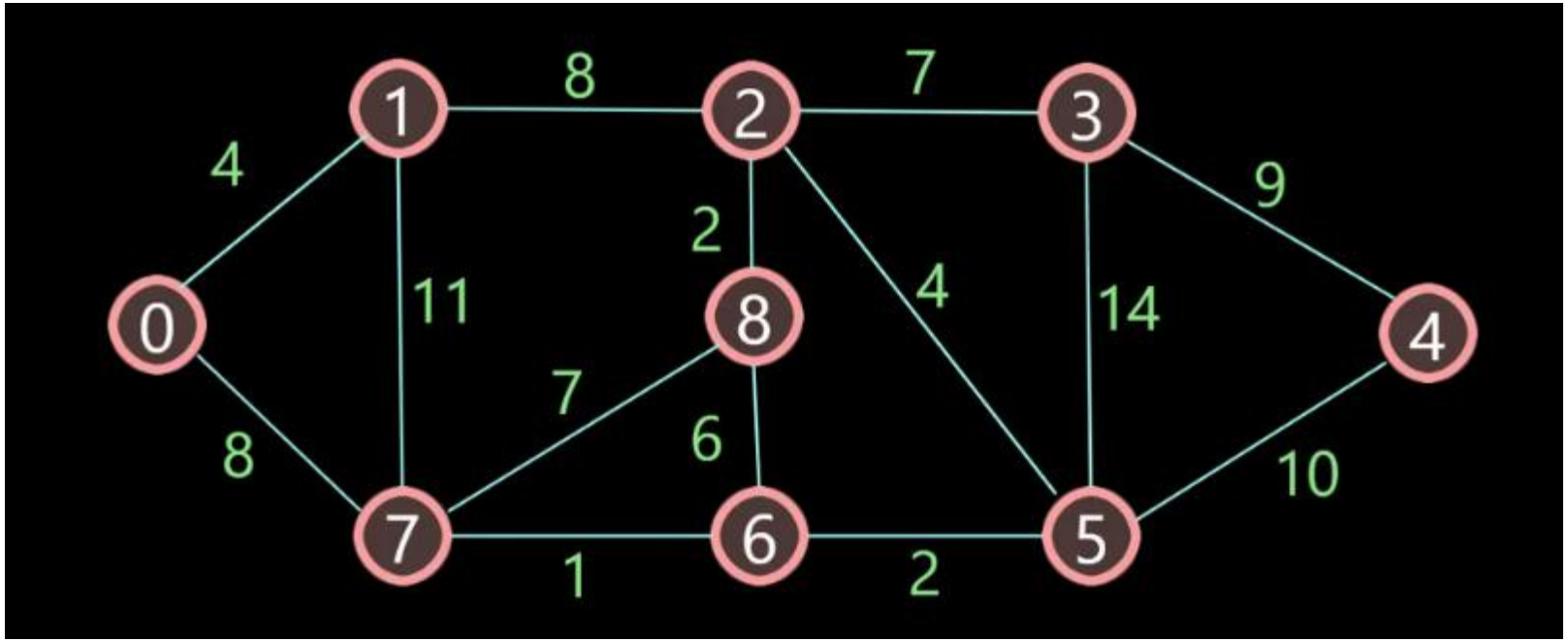
# 带权图

- 给边赋予 “长度” 等属性



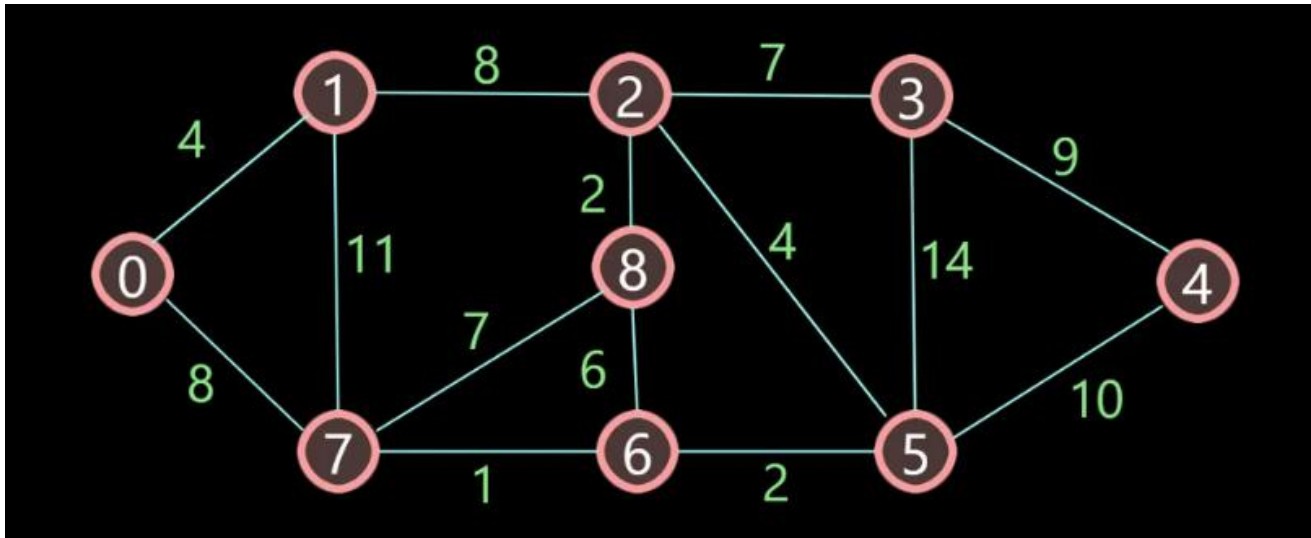
# 图的遍历

- 8 14
- 0 1 4
- 0 7 8
- 1 2 8
- 1 7 11
- 2 3 7
- 2 5 4
- 2 8 2
- 3 5 14
- 3 4 9
- 4 5 10
- 5 6 2
- 6 7 1
- 6 8 6
- 7 8 7



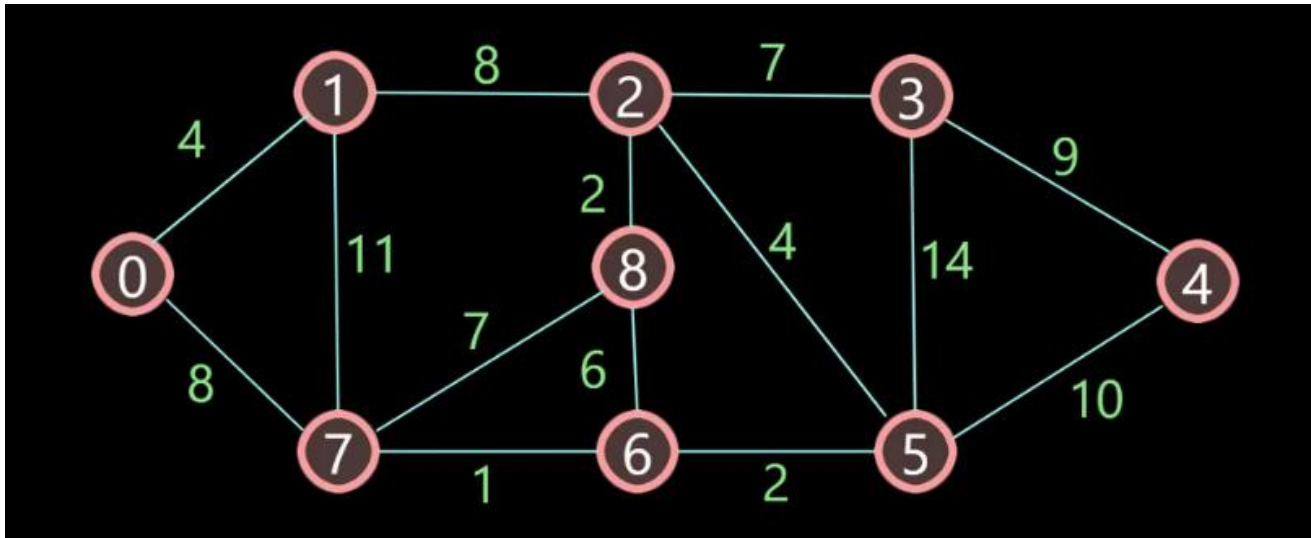
# 图的深度优先遍历

- 思路
- 递归{
  - 遍历起点的所有邻接点
    - 如果能够访问：则把该点作为起点，进行递归
  - 返回
- }



# 图的宽度优先遍历

- 思路
- 起点进入队列
- while(队列非空){
  - 获取队首
  - 将队首的所有未访问邻接点加入队列
  - 队首出队
- }

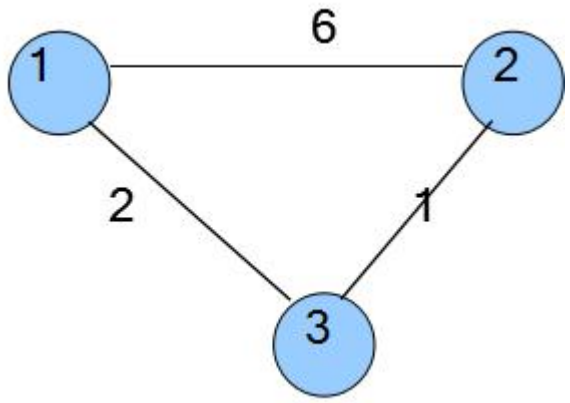


# 谈一谈“松弛”操作

- 文老到地图上各个地点的距离为 dis[i]
- 文老到 A 的距离 dis[A] = 10
- 文老到 B 的距离 dis[B] = 3
- 有一天，从 A 到 B 之间修了一条道路，距离为 6。G[A][B] = 6
- 显然，对于文老来说，直接到 A 点不如经由 B 再到达 A 更优。
- 那么对于文老来说，dis[A] = dis[B] + G[A][B]

# 谈一谈“松弛”操作

- 比如说从1到2可以有2种解法，一种是直接走，另一种就是用一个点来中转;
- 从这两条路上选最短的走法的操作就叫松弛。




```
if(dis[i]+G[i][j]<dis[j]){
    dis[j] = dis[i]+G[i][j];
}
```

# 单源最短路径——Dijkstra 算法

- 单源：从图中一个点出发，到其余所有点的最短距离
- 思路：
  - 选定一个起点，初始状态下标记起点到任一点 i dis[i] 为无穷大
  - 遍历起点所有的邻接点

当前距离存储:

| DIS | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A至各点距离 | 0 | 1 | 12 | 4 | 17 | 19 |
| 已访问的节点 | Y | Y | Y | Y | Y | |

数据存储:

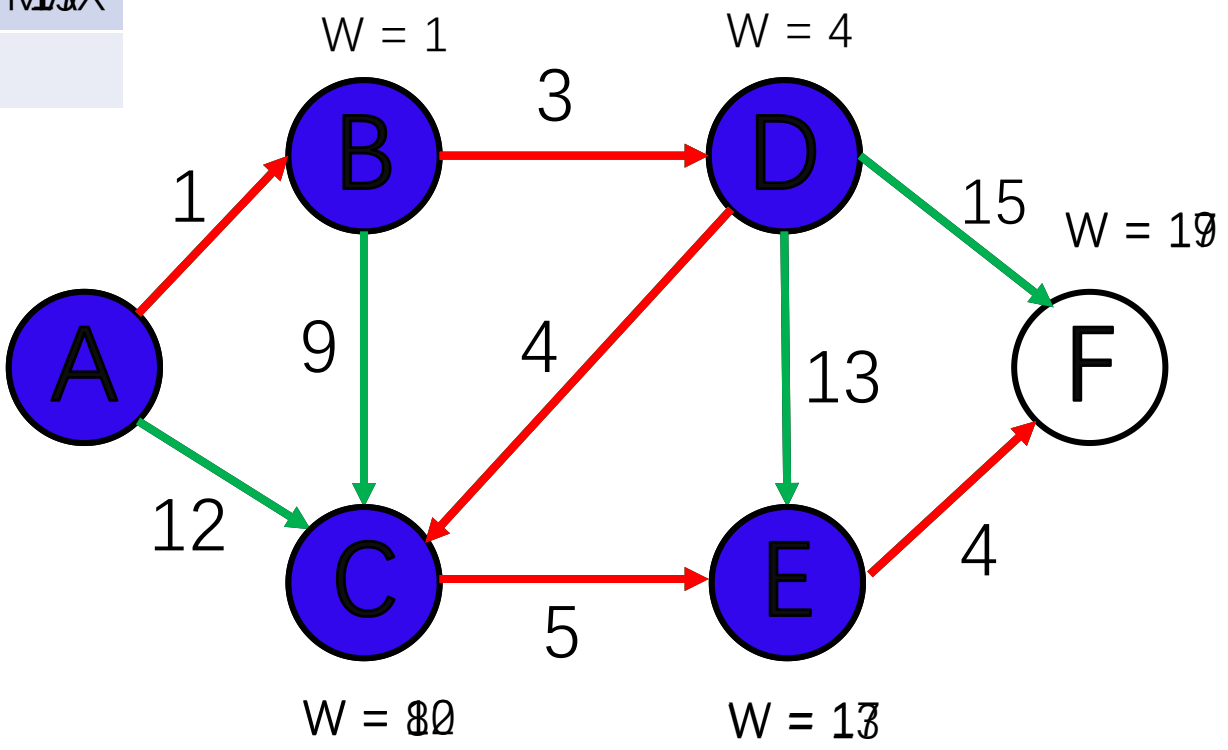
| map | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 12 | MAX | MAX | MAX |
| B | MAX | 0 | 9 | 3 | MAX | MAX |
| C | MAX | MAX | 0 | MAX | 5 | MAX |
| D | MAX | MAX | 4 | 0 | 13 | 15 |
| E | MAX | MAX | MAX | MAX | 0 | 4 |
| F | MAX | MAX | MAX | MAX | MAX | 0 |

最小顶点个数为17

MIN = 13 选择E

并生成最后一个元素

结束程序




A->F的最短路径为17

# 单源最短路径——Dijkstra 算法 邻接矩阵

```
void Dijkstra(int be){
    for(int i=0;i<=MAXN;++i){
        dis[i] = 0x7fffffff;
    }
    dis[be] = 0;
    for(int k=1;k<=n;++k){
        int tx = -1,tp = 0x7fffffff;
        for(int i=0;i<=n;++i){
            if(vis[i]==0){
                if(dis[i]<tp){
                    tx = i;
                    tp = dis[i];
                }
            }
        }
        for(int i=0;i<=n;++i){
            if(G[tx][i]!=0&&vis[i]==0)
            if(dis[tx]+G[tx][i]<dis[i]){
                dis[i] = dis[tx]+G[tx][i];
            }
        }
        vis[tx] = 1;
    }
}
```

```
#include<iostream>
#include<cstdio>
using namespace std;
const int MAXN = 110;
int G[MAXN][MAXN],dis[MAXN],vis[MAXN],pre[MAXN];
int n,e,u,v,d;

int main(){
    cin>>n>>e;
    for(int i=1;i<=e;++i){
        cin>>u>>v>>d;
        G[u][v] = d;
        G[v][u] = d;
    }
    Dijkstra(0);
    for(int i=0;i<=n;++i){
        cout<<i<<" "<<dis[i]<<endl;
    }
    return 0;
}
```

# 单源最短路径——Dijkstra 算法 邻接表

```cpp
#include<iostream>
#include<cstdio>
using namespace std;
const int MAXN = 110,MAXE = 10010;
int head[MAXN];
struct Edge{
    int to,dis,next;
};
Edge edge[MAXE];
int cnt;
void add_edge(int u,int v,int d){
    ++cnt;
    edge[cnt].next = head[u];
    head[u] = cnt;
    edge[cnt].to = v;
    edge[cnt].dis = d;
}
int n,e,u,v,d,dis[MAXN],pre[MAXN],vis[MAXN];
```

```cpp
void Dijkstra(int be){
    for(int i=0;i<=n;++i){
        dis[i] = 0x7fffffff;
    }
    dis[be] = 0;
    for(int k=0;k<=n;++k){
        int tx = 0x7fffffff,tp = -1;
        for(int i=0;i<=n;++i){
            if(vis[i]==0){
                if(dis[i]<tx){
                    tx = dis[i];
                    tp = i;
                }
            }
        }
        vis[tp] = 1;
        for(int i=head[tp];i!=0;i=edge[i].next){
            int to = edge[i].to;
            if(dis[to]>dis[tp]+edge[i].dis){
                dis[to] = dis[tp] + edge[i].dis;
                pre[to] =tp;
            }
        }
    }
}
```

# 单源最短路径——Dijkstra 算法 STL

```cpp
#include<iostream>
#include<cstdio>
#include<vector>
using namespace std;
const int MAXN = 110;
const int INF = 0x7fffffff;
struct Edge{
    int to,dis;
};
vector<Edge> G[MAXN];
void add_edge(int u,int v,int d){
    Edge e;
    e.to = v;e.dis = d;
    G[u].push_back(e);
}
int n,e,u,v,d,dis[MAXN],vis[MAXN],pre[MAXN];
```

```cpp
void Dijkstra(int be){
    for(int i=0;i<=MAXN;++i){
        dis[i] = INF;
        vis[i] = 0;
    }
    dis[be] = 0;
    for(int k=0;k<=n;++k){
        int tx = INF,tp = -1;
        for(int i=0;i<=n;++i){
            if(vis[i]==0&&dis[i]<tx){
                tx = dis[i];
                tp = i;
            }
        }
        vis[tp] = 1;
        int sz = G[tp].size();
        for(int i=0;i<sz;++i){
            if(dis[G[tp][i].to]>dis[tp] + G[tp][i].dis){
                dis[G[tp][i].to] = dis[tp] + G[tp][i].dis;
                pre[G[tp][i].to] = tp;
            }
        }
    }
}
```

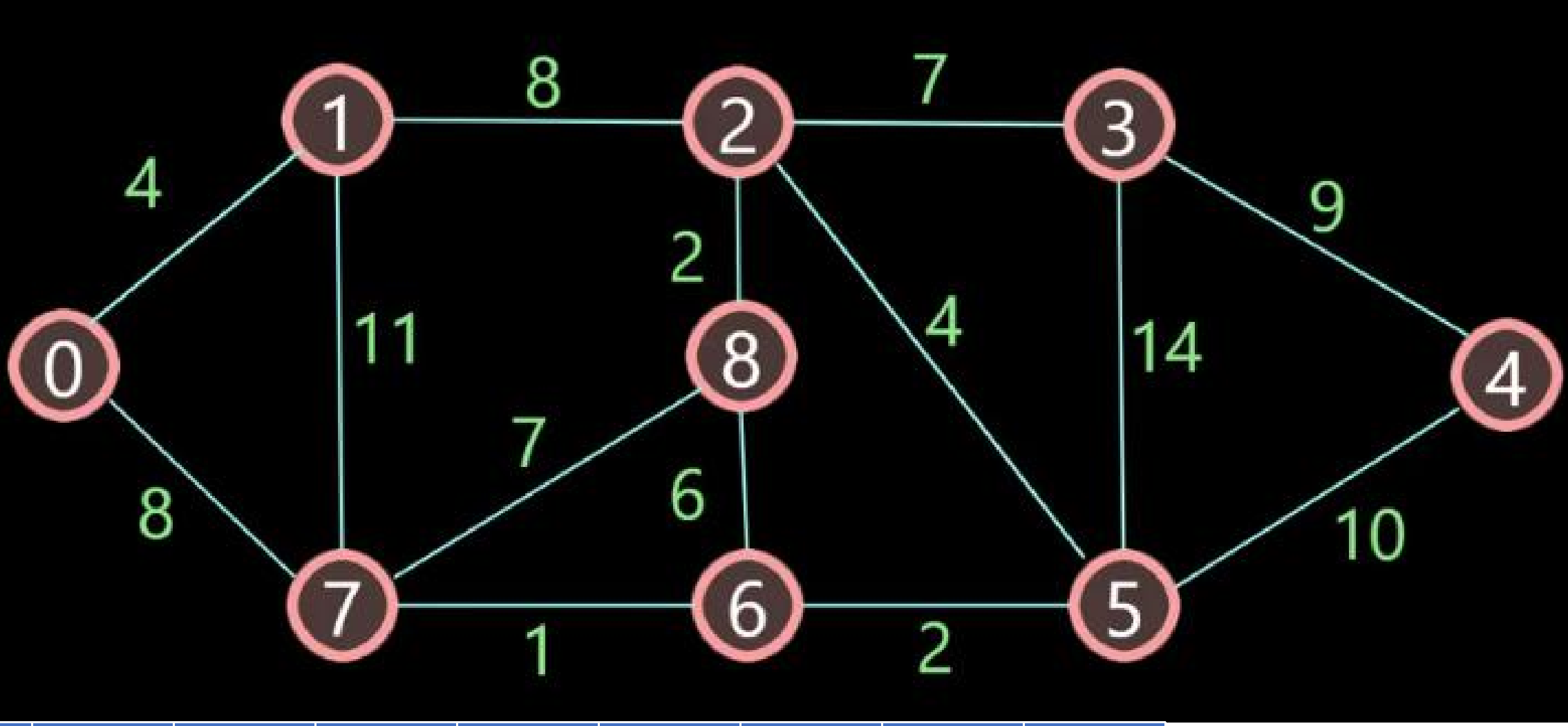
# 单源最短路径——Dijkstra 算法 优先队列优化

```cpp
#include<iostream>
#include<cstdio>
#include<vector>
#include<queue>
using namespace std;
const int MAXN = 110;
const int INF = 0x7fffffff;
struct Edge{
    int to,dis;
};
vector<Edge> G[MAXN];
struct Dis{
    int i,dis;
    friend bool operator < (Dis x,Dis y){
        x.dis>y.dis;
    }
};
priority_queue<Dis> pq;
void add_edge(int u,int v,int d){
    Edge e;
    e.to = v;e.dis = d;
    G[u].push_back(e);
}
```

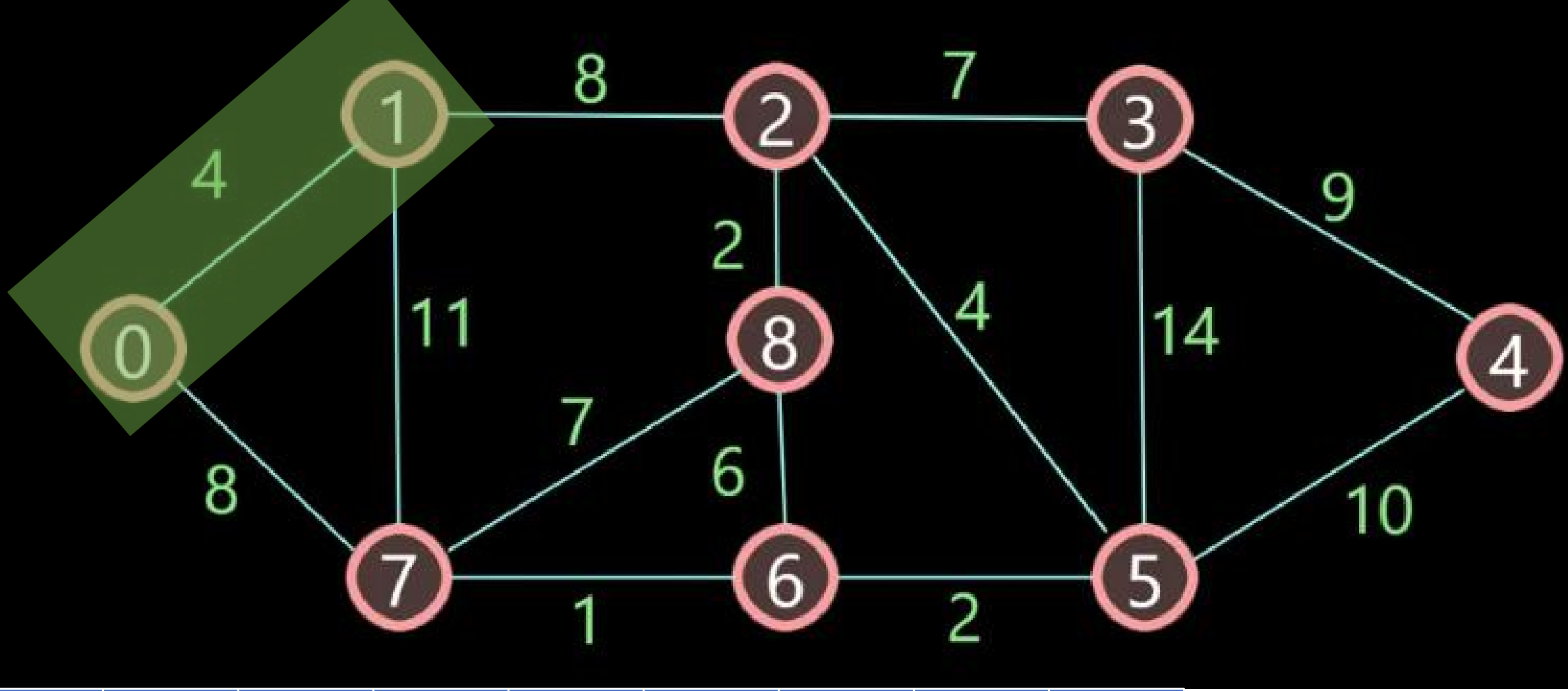
```cpp
int n,e,u,v,d,pre[MAXN],vis[MAXN],dis[MAXN];
int Dijkstra(int be){
    for(int i=0;i<=n;++i){
        dis[i] = INF;
    }
    dis[be] = 0;
    Dis temp;
    temp.i = be;
    temp.dis = 0;
    pq.push(temp);
    while(!pq.empty()){
        int td = pq.top().dis;
        int tp = pq.top().i;
        int sz = G[tp].size();
        pq.pop();
        for(int i=0;i<sz;++i){
            int des = G[tp][i].to;
            int cost = G[tp][i].dis;
            if(dis[des]>dis[tp]+cost){
                dis[des] = dis[tp]+cost;
                pre[des] = tp;
                pq.push({des,dis[des]});
            }
        }
    }
}
```

# 单源最短路径——Bellman-Ford算法

- 思路：
- 使用 dis 数组记录从起点到各点的最短路径，dis[i] 表示从起点到 i 点的最短路径
- 枚举n-1次图中每一条边E(x->y){
  枚举图中每个点 i {
- 如果 dis[i] > dis[x] + lenE
- 那么进行松弛
- }
  如果没有发生松弛，则表示最优。
- }
- 再次枚举一次图中每条边
- 如果发生过松弛　　则说明图中有 负环
- 否则说明到达最优



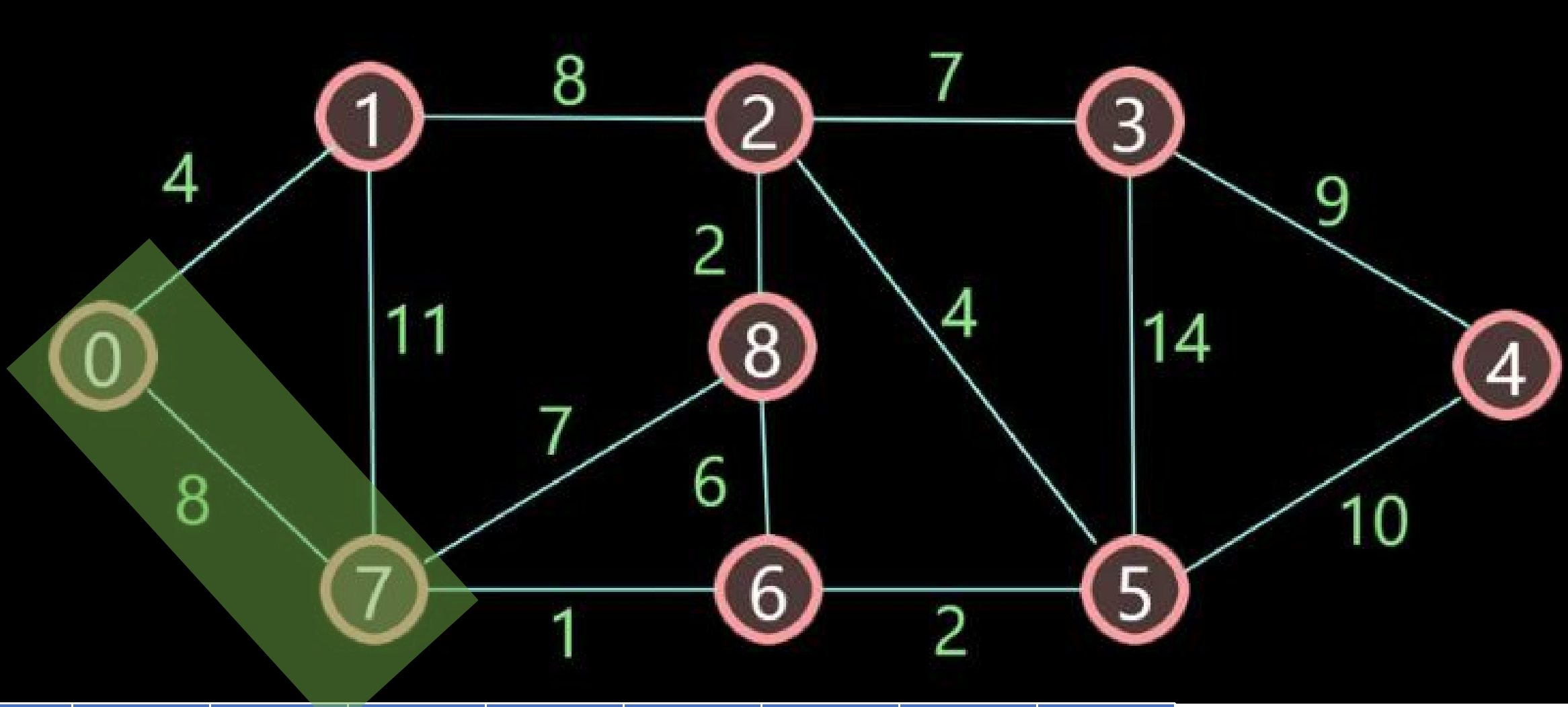

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | INF | INF | INF | INF | INF | INF | INF | INF |




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | INF | INF | INF | INF | INF | INF | INF |

dis[1] > dis[0]+w(0,1)
relax
dis[1] = dis[0]+w(0,1)

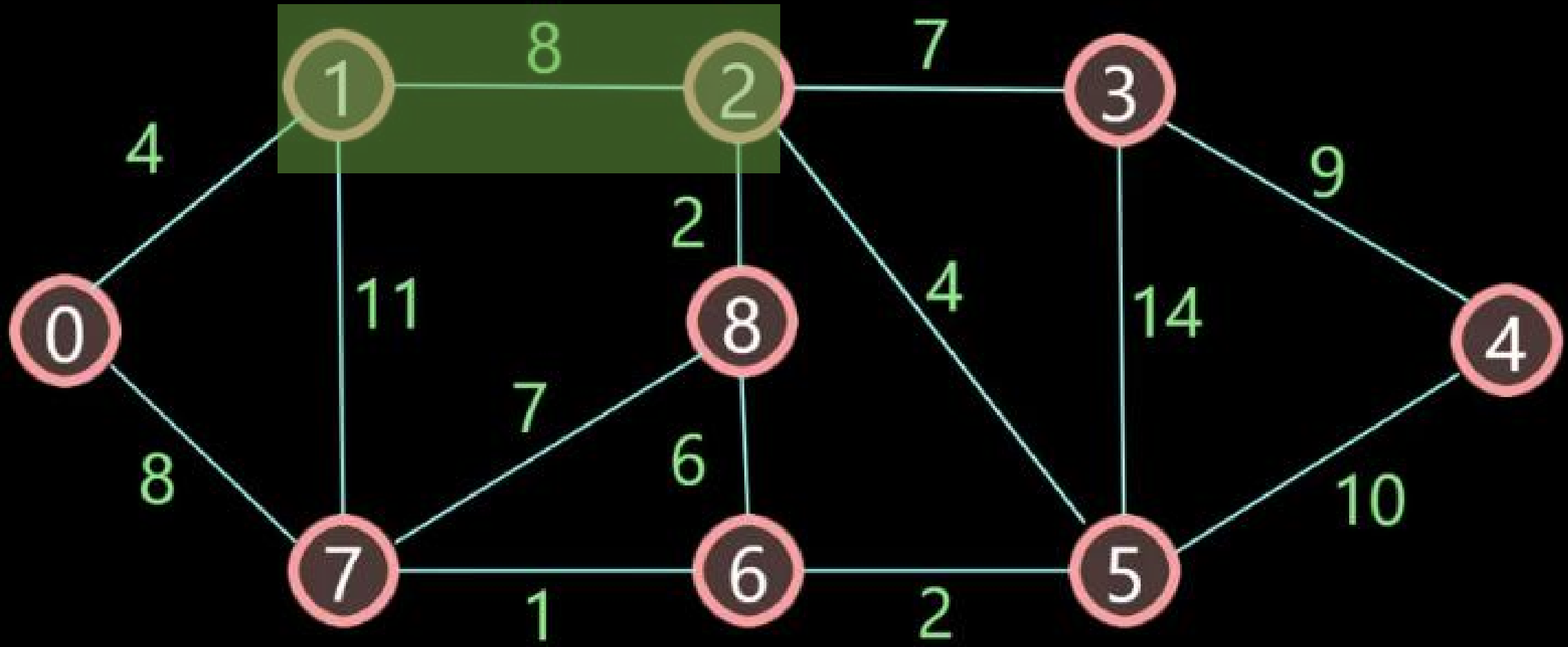
dis[0] < dis[1]+w(1,0)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | INF | INF | INF | INF | INF | 8 | INF |

dis[7] > dis[0]+w(0,7)
relax
dis[0] = dis[0]+w(0,7)

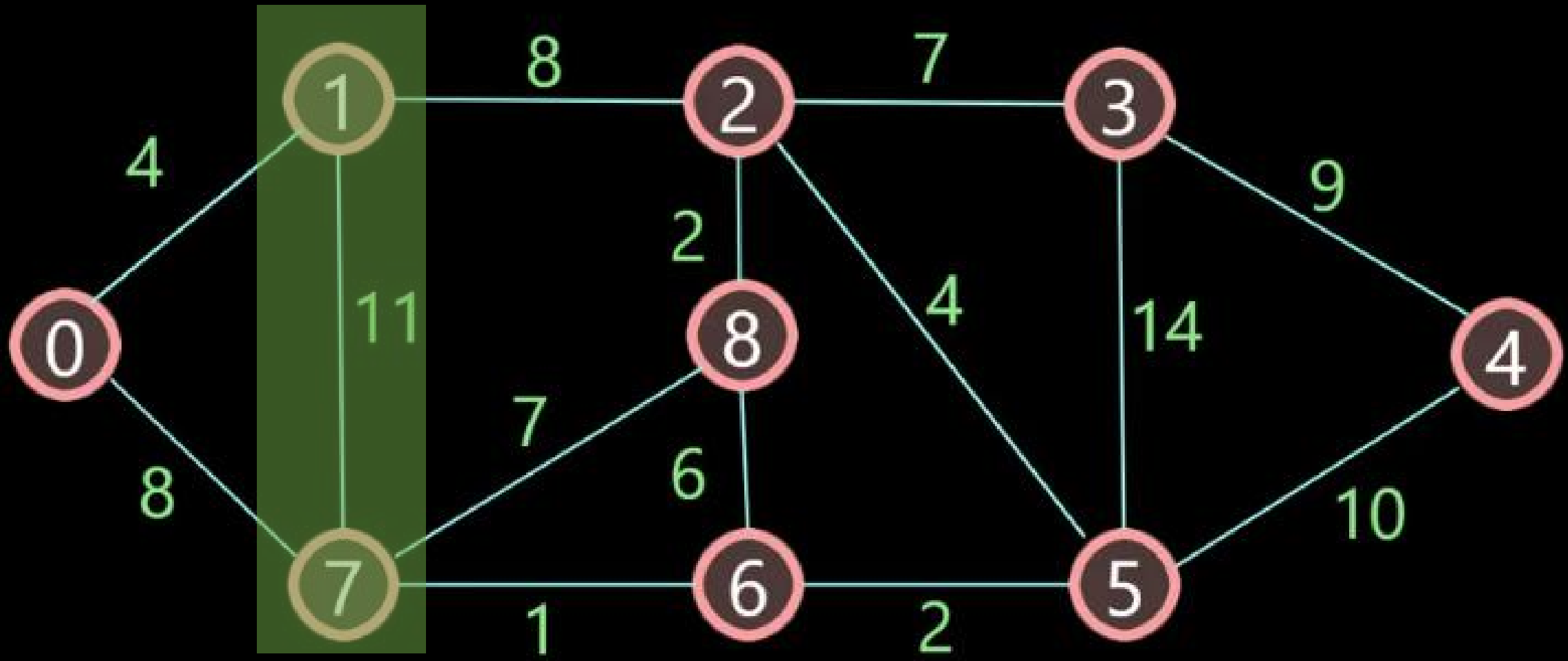
dis[0] < dis[7]+w(7,0)
no relax



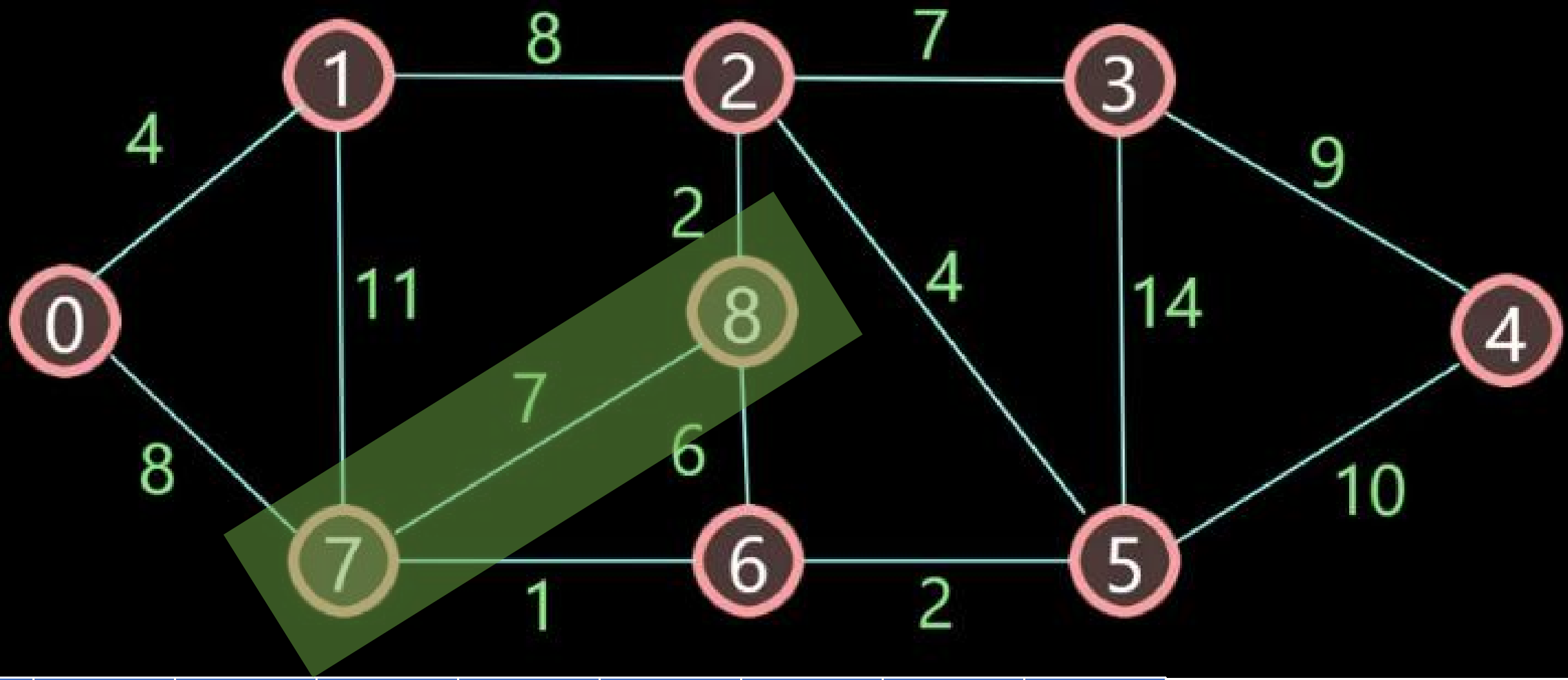

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | INF | INF | INF | INF | 8 | INF |

dis[2] > dis[1]+w(1,2)
relax
dis[2] = dis[1]+w(1,2)

dis[1] < dis[2]+w(2,1)
no relax



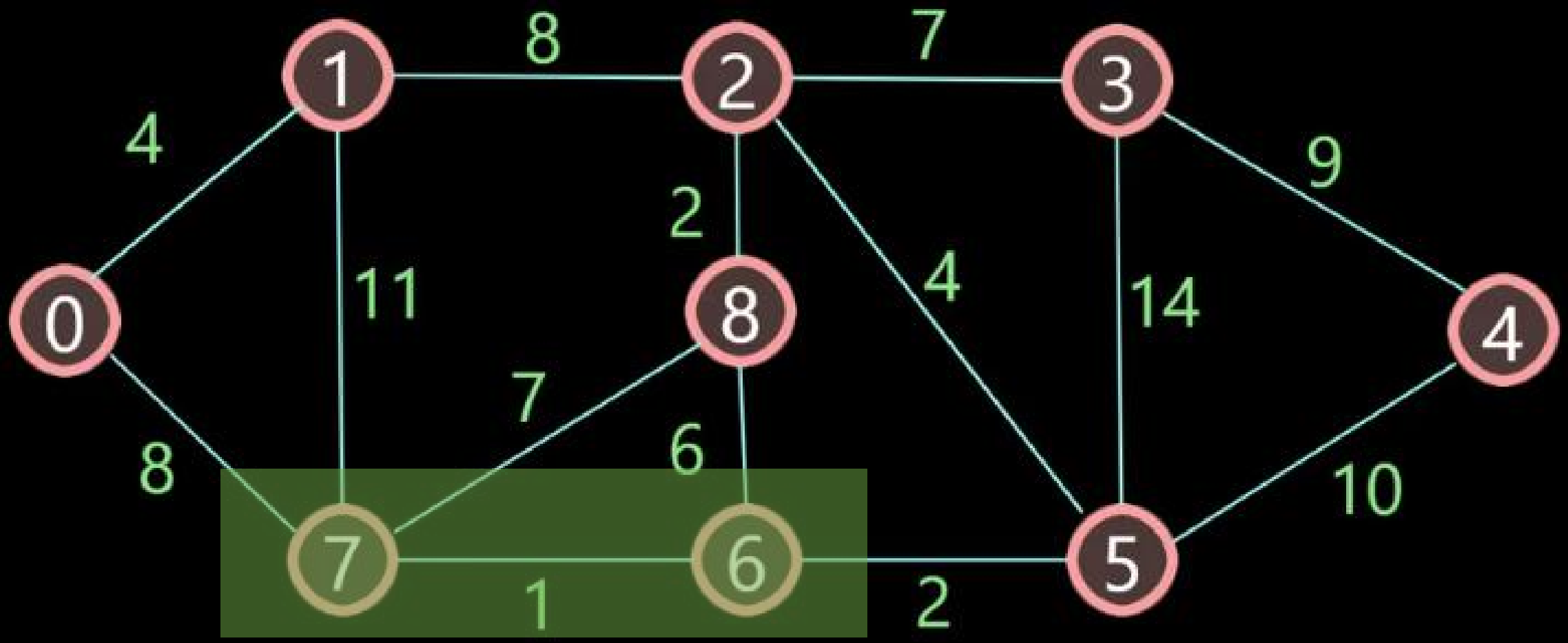

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | INF | INF | INF | INF | 8 | INF | dis[7] < dis[1]+w(1,7)<br>no relax |
| 0 | 4 | 12 | INF | INF | INF | INF | 8 | INF | dis[1] < dis[7]+w(7,1)<br>no relax |




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | INF | INF | INF | INF | 8 | 15 |

dis[8] > dis[7]+w(7,7)
relax
dis[8] = dis[7]+w(7,7)

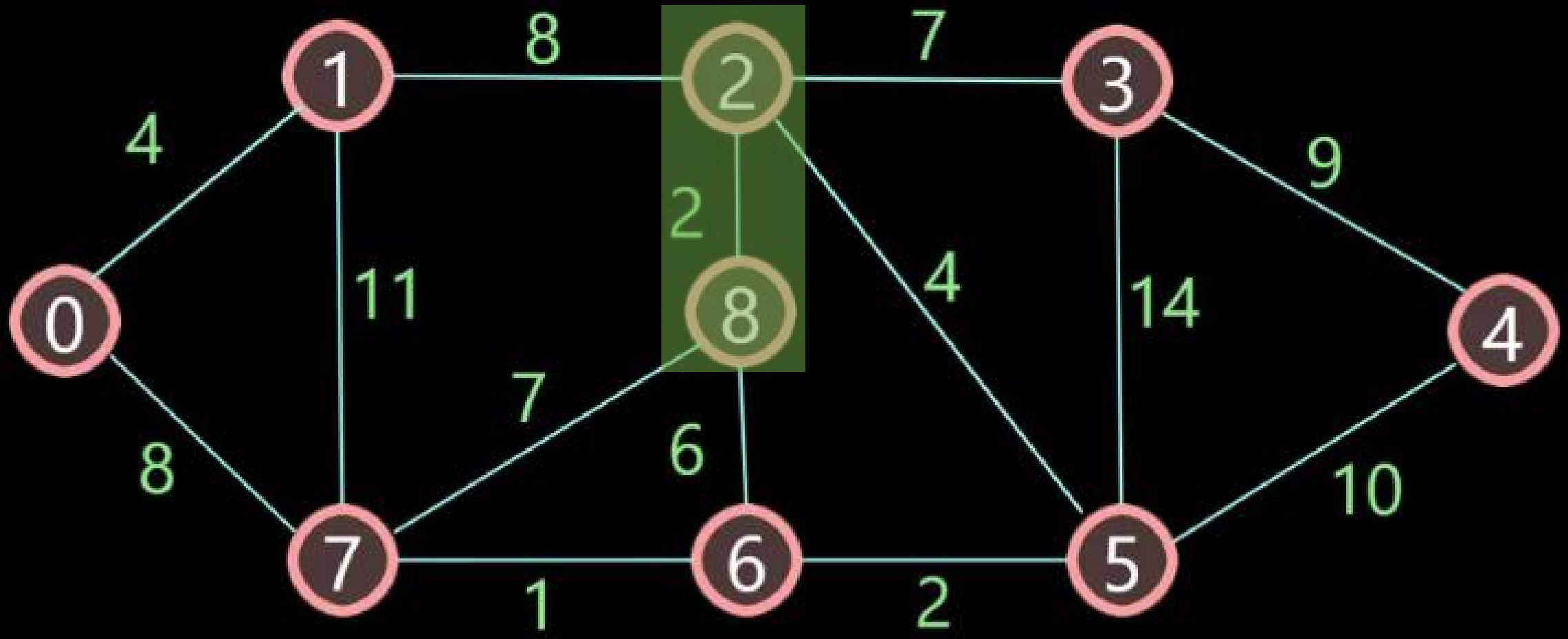
dis[7] < dis[8]+w(8,7)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | INF | INF | INF | 9 | 8 | 15 |

dis[6] > dis[7]+w(7,6)
relax
dis[6] = dis[7]+w(7,6)

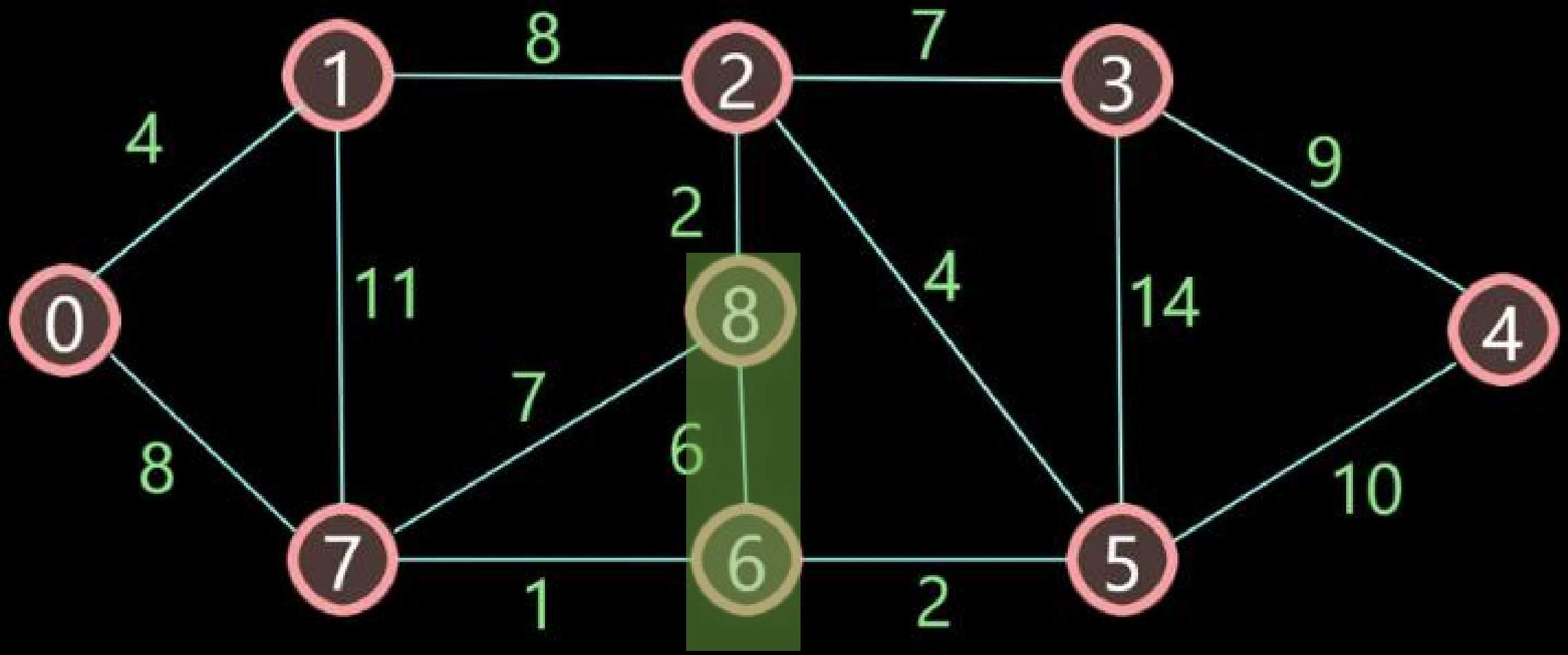
dis[7] < dis[6]+w(6,7)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | INF | INF | INF | 9 | 8 | 14 |

dis[8] > dis[2]+w(2,8)
relax
dis[8] = dis[2]+w(2,8)

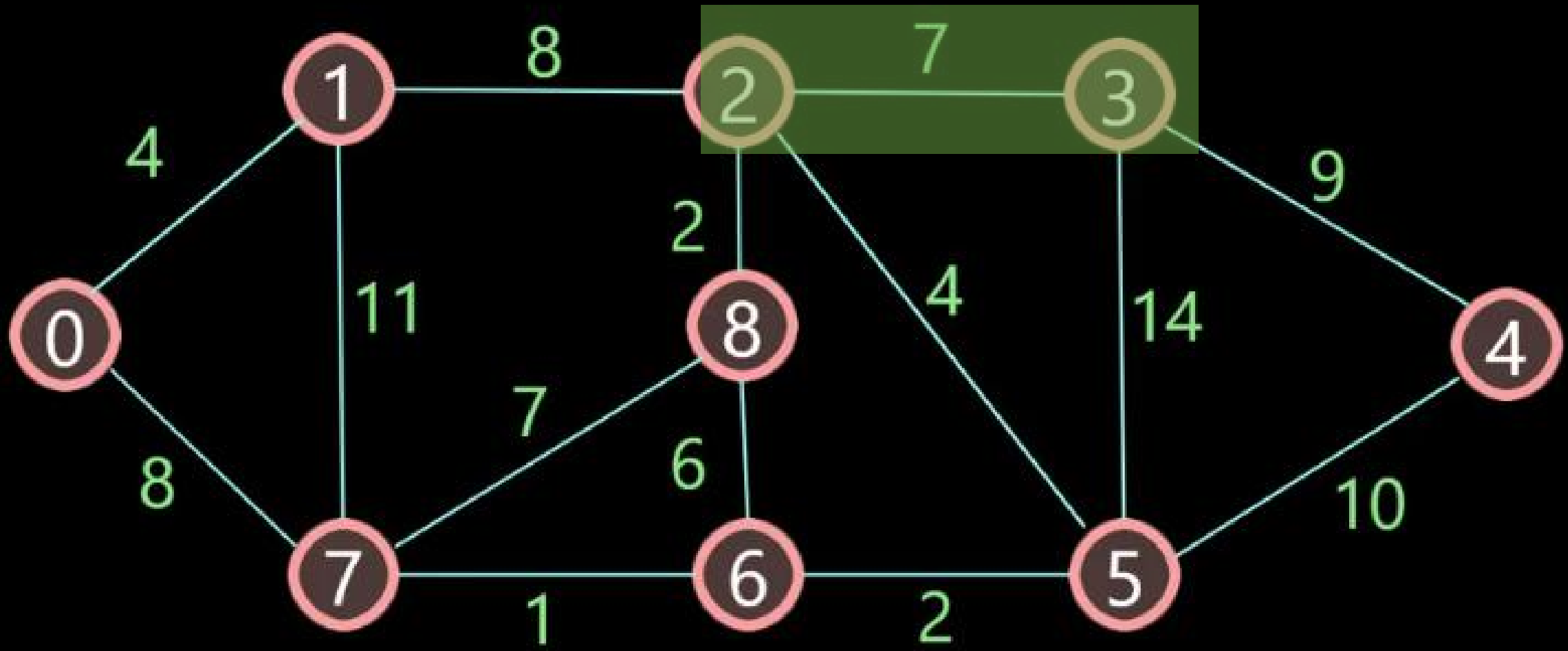
dis[2] < dis[8]+w(8,2)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | INF | INF | INF | 9 | 8 | 14 |

dis[6] < dis[8]+w(8,6)
no relax

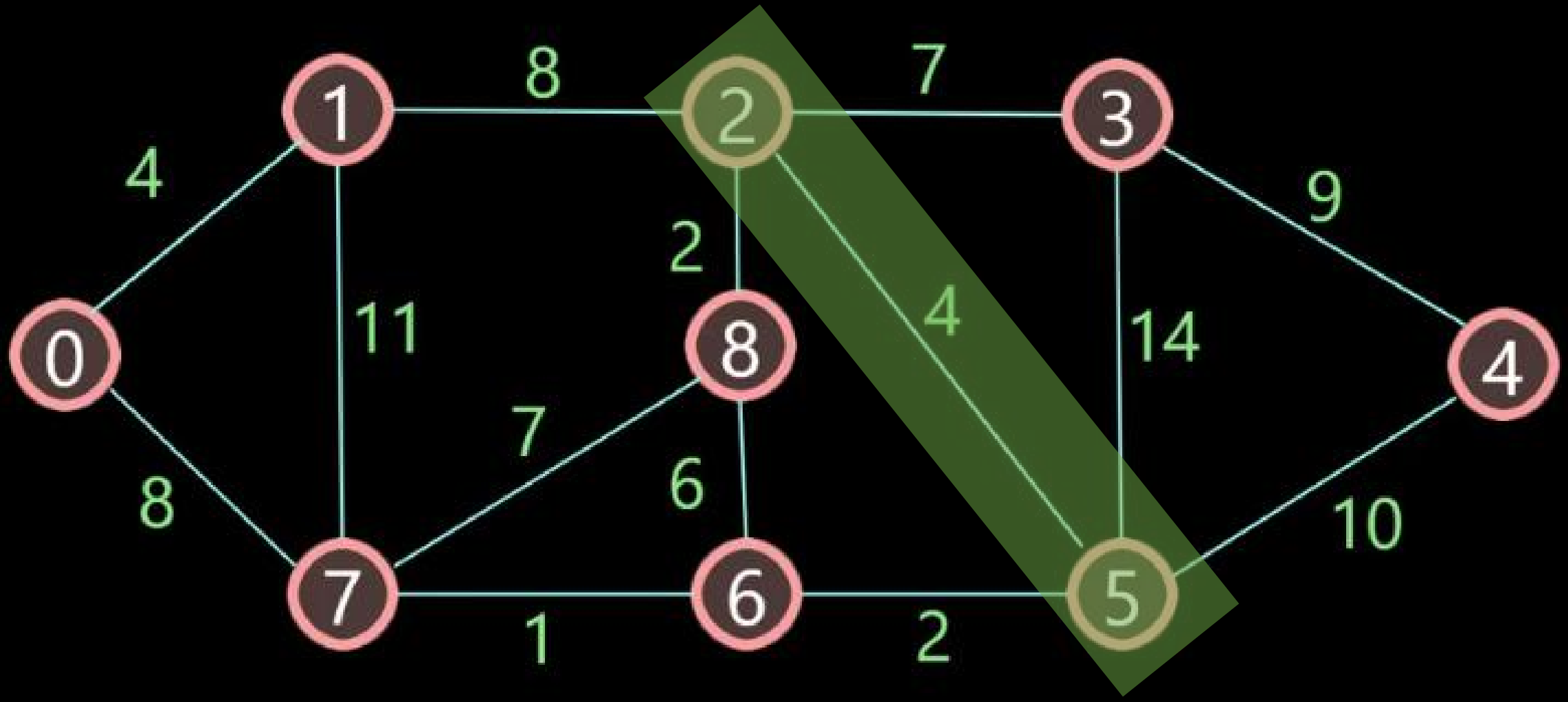
dis[8] = dis[6]+w(6,8)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | 19 | INF | INF | 9 | 8 | 14 |

dis[3] > dis[2]+w(2,3)
relax
dis[3] = dis[2]+w(2,3)

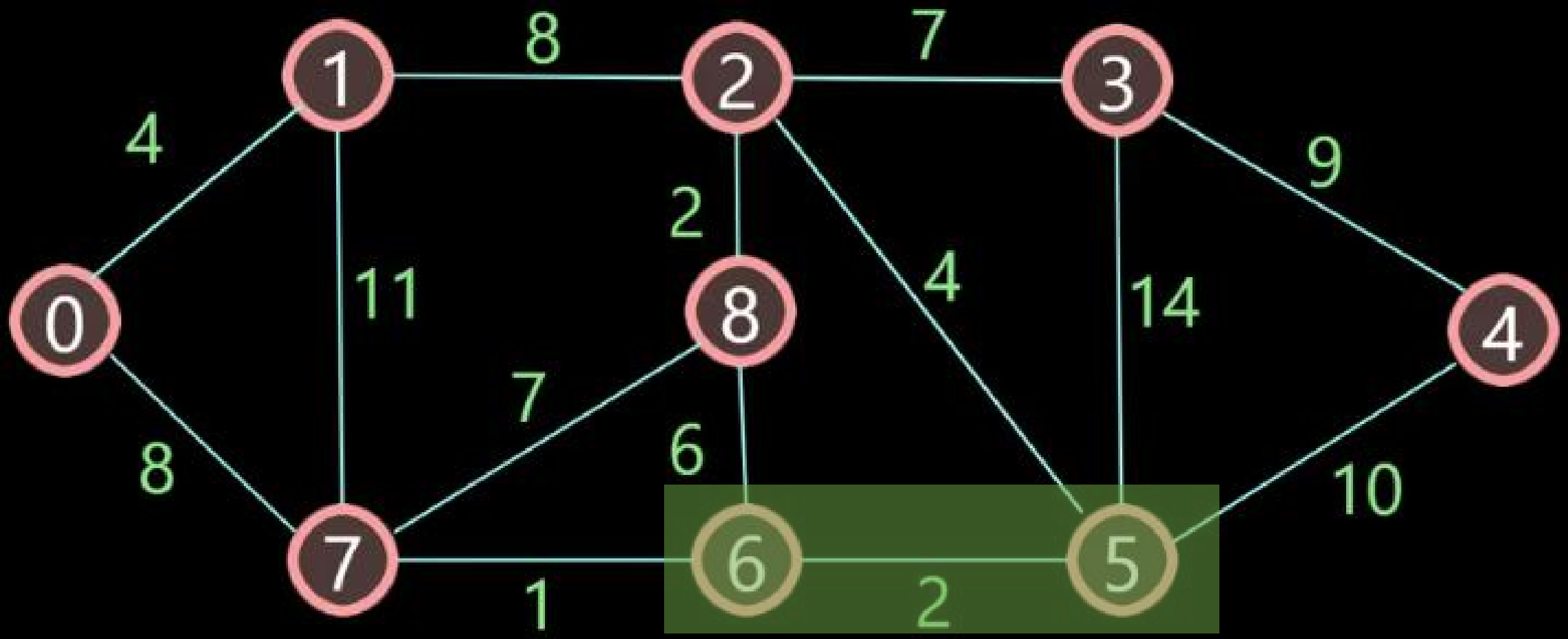
dis[2] < dis[3]+w(3,2)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | 19 | INF | 16 | 9 | 8 | 14 |

dis[5] > dis[2]+w(2,5)
relax
dis[5] = dis[2]+w(2,5)

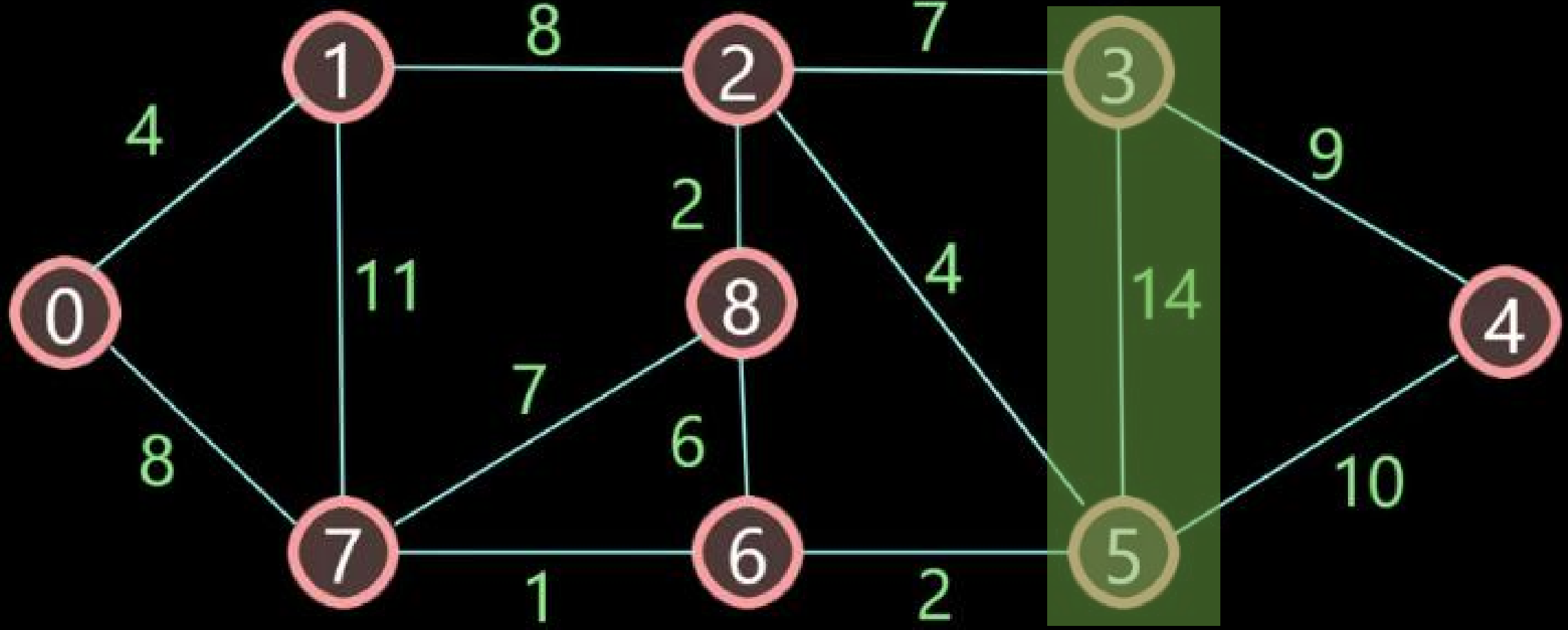
dis[2] < dis[3]+w(5,2)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | 19 | INF | 11 | 9 | 8 | 14 |

dis[5] > dis[2]+w(6,5)
relax
dis[5] = dis[2]+w(6,5)

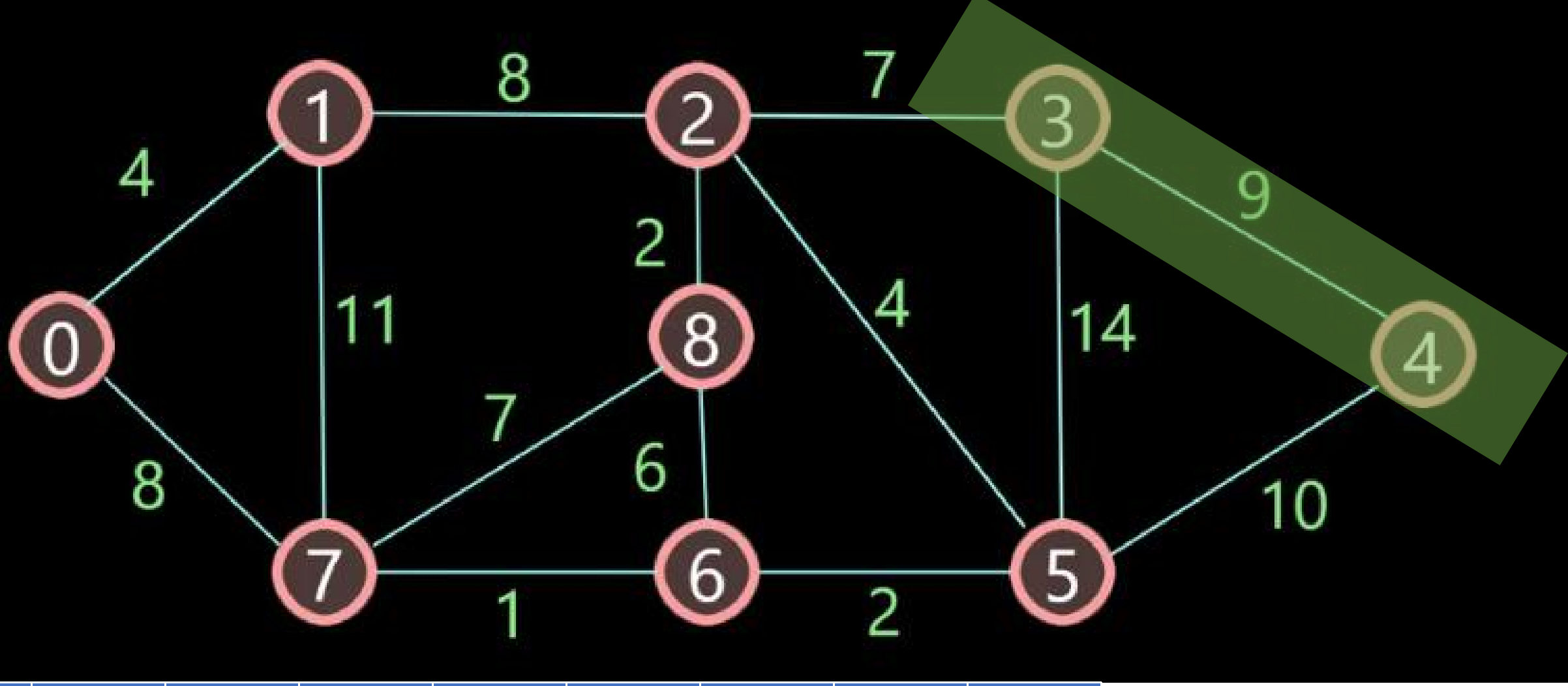
dis[6] < dis[5]+w(5,6)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | 19 | INF | 11 | 9 | 8 | 14 |

dis[5] < dis[3]+w(3,5)
no relax

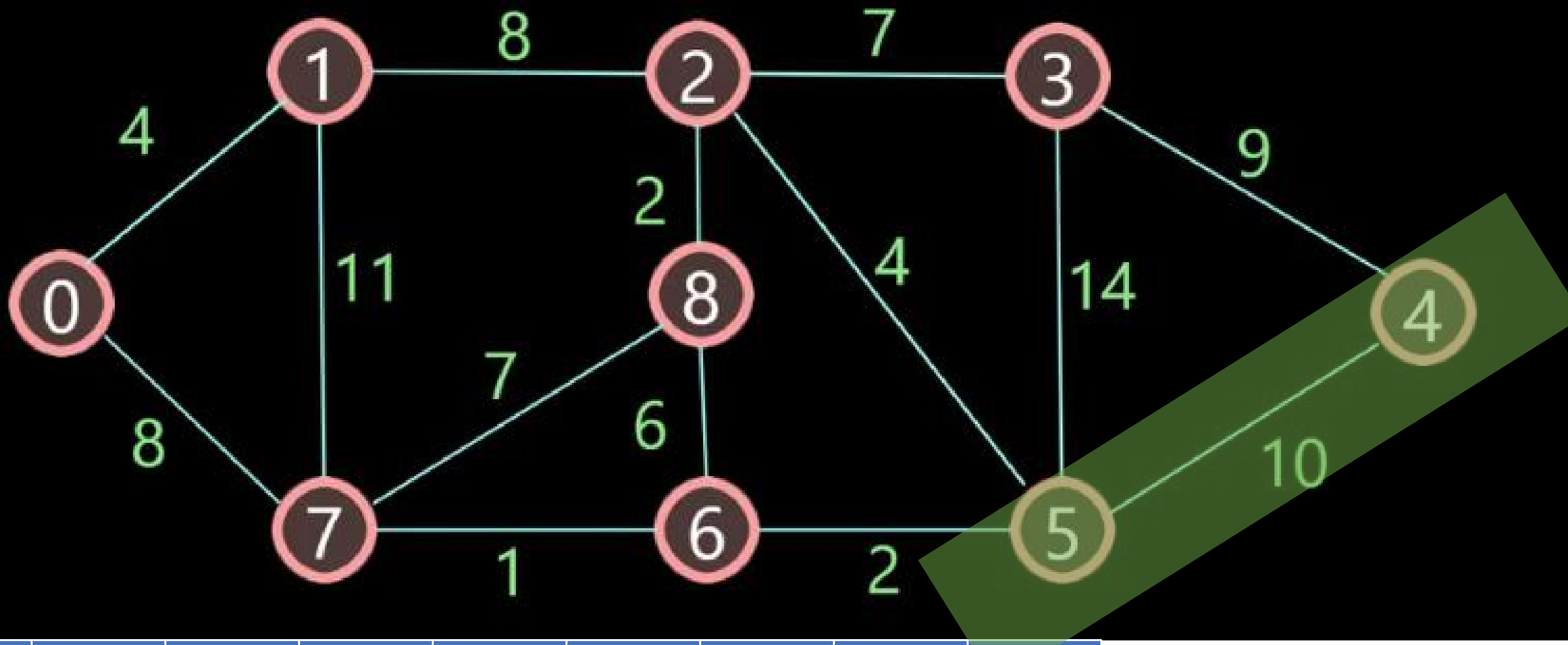
dis[3] < dis[5]+w(5,3)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | 19 | 28 | 11 | 9 | 8 | 14 |

dis[4] > dis[3]+w(3,4)
relax
dis[4] = dis[3]+w(3,4)

dis[3] < dis[4]+w(4,3)
no relax




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 12 | 19 | 28 | 11 | 9 | 8 | 14 |

dis[4] > dis[5]+w(5,4)
relax
dis[4] = dis[5]+w(5,4)

dis[5] < dis[4]+w(4,5)
no relax

# 单源最短路径——Bellman-Ford算法

- 暴力枚举所有的边
- 第k-1轮遍历结束，理应得到最优解
- 再进行第k轮，如果还能发生松弛，那么图中有 负权 边。

# 单源最短路径——Bel

```cpp
#include<iostream>
#include<cstdio>
using namespace std;
const int MAXN = 110,MAXE = 10010;
const int INF = 1e9;
struct Edge{
    int from,to,dis,next;
};
Edge edge[MAXE];
int cnt,head[MAXN],dis[MAXN];
void add_edge(int u,int v,int d){
    ++cnt;
    edge[cnt].next = head[u];
    head[u] = cnt;
    edge[cnt].from = u;
    edge[cnt].to = v;
    edge[cnt].dis = d;
}
int u,v,d,n,e;
```

```cpp
int Bellman_Ford(int be){
    for(int i=0;i<=n;++i){
        dis[i] = INF;
    }
    dis[be] = 0;
    for(int k=1;k<n;++k){
        int flag = 0;
        for(int i=1;i<=cnt;++i){
            int from = edge[i].from;
            int to = edge[i].to;
            int ds = edge[i].dis;
            if(dis[to]>dis[from]+ds){
                dis[to] = dis[from] + ds;
                flag = 1;
            }
        }
        if(flag == 0){
            return 0;
        }
    }
    int flag = 0;
    for(int i=1;i<=cnt;++i){
        int from = edge[i].from;
        int to = edge[i].to;
        int ds = edge[i].dis;
        if(dis[to]>dis[from]+ds){
            dis[to] = dis[from] + ds;
            flag = 1;
        }
    }
    if(flag!=0){
        return 1;
    }else{
        return 0;
    }
}
```

```cpp
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 1e5+1;
const int INF = 1e9+7;
vector<pair<int,int> > E[MAXN];
```

```cpp
int main(){
    scanf("%d %d",&n,&e);
    for(int i=1;i<=e;++i){
        scanf("%d %d %d",&u,&v,&d);
        E[u].push_back(pair<int,int>(v,d));
        E[v].push_back(pair<int,int>(u,d));
    }
    Bellman_Ford(0);
    for(int i=0;i<=n;++i){
        printf("%d %d\n",i,dis[i]);
    }
    return 0;
}
```

```cpp
int Bellman_Ford(int be){
    for(int i=0;i<=n;++i){
        dis[i] = INF;
    }
    dis[be] = 0;
    for(int k=1;k<= n;++k){
        int f = 0;
        for(int i=0;i<=n;++i){
            for(int j=0;j<E[i].size();++j){
                int to = E[i][j].first;
                int dist = E[i][j].second;
                if(dis[to]>dis[i]+dist){
                    dis[to] = dis[i] + dist;
                    f = 1;
                }
            }
        }
        if(f==0){
            return 0;
        }
    }
    int f = 0;
    for(int i=0;i<=n;++i){
        for(int j=0;j<E[i].size();++j){
            int to = E[i][j].first;
            int dist = E[i][j].second;
            if(dis[to]>dis[i]+dist){
                dis[to] = dis[i]+dist;
                f = 1;
            }
        }
    }
    if(f==0){
        return 0;
    }else{
        return 1;
    }
}
```

# SPFA (Shortest Path Faster Algorithm)

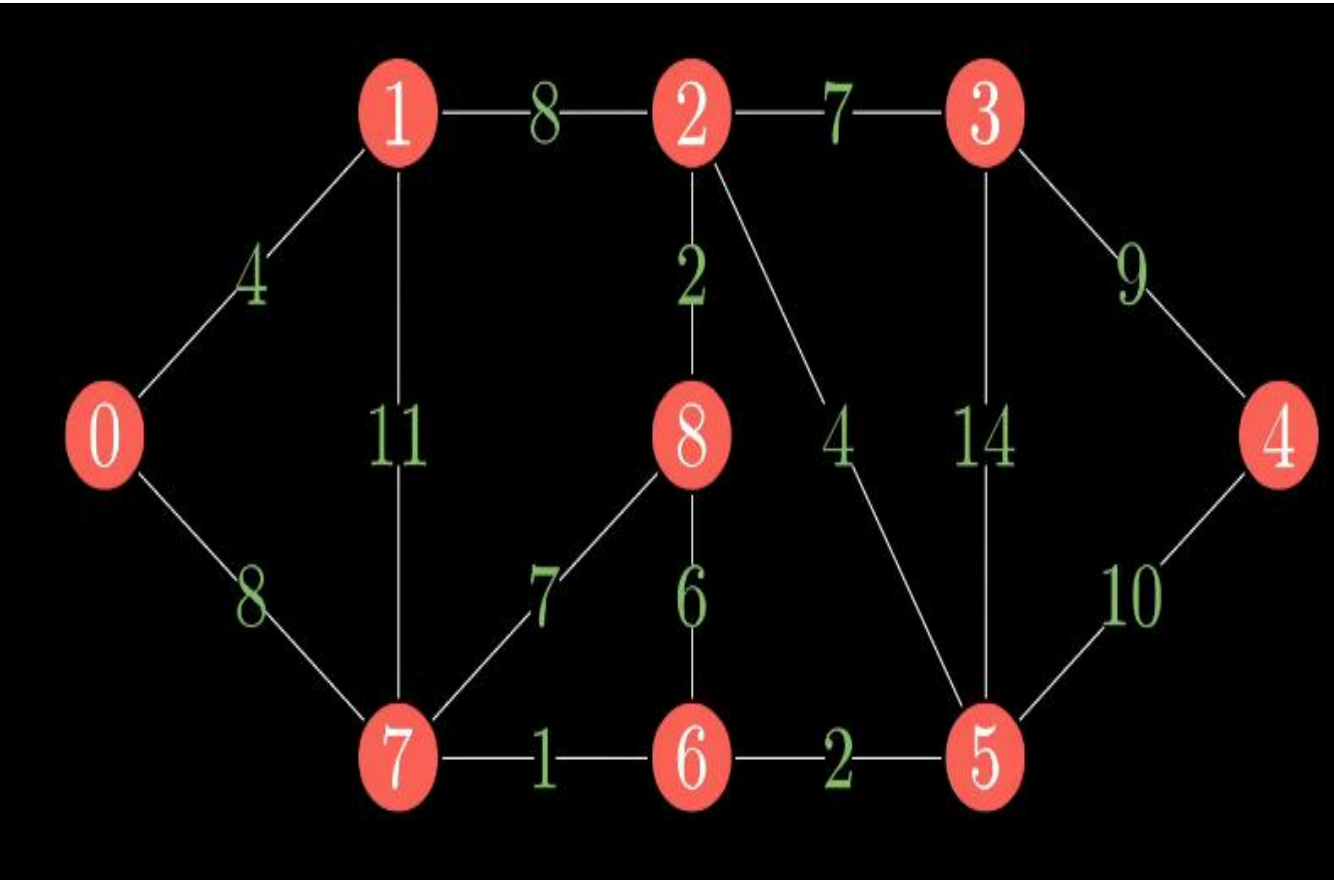
- 快速最短路径算法（这个名字太*了）
- 思路:
- 可以看做是队列优化的Bellman-ford
- 起点入队，使用起点连接的边松弛更新整个dis数组。
- 使用队列中第一个节点进行整个dis数组的松弛，头结点出队，并取消标记。
- 对进制松弛过的节点进行重新判断如果该节点不在队列中则入队。

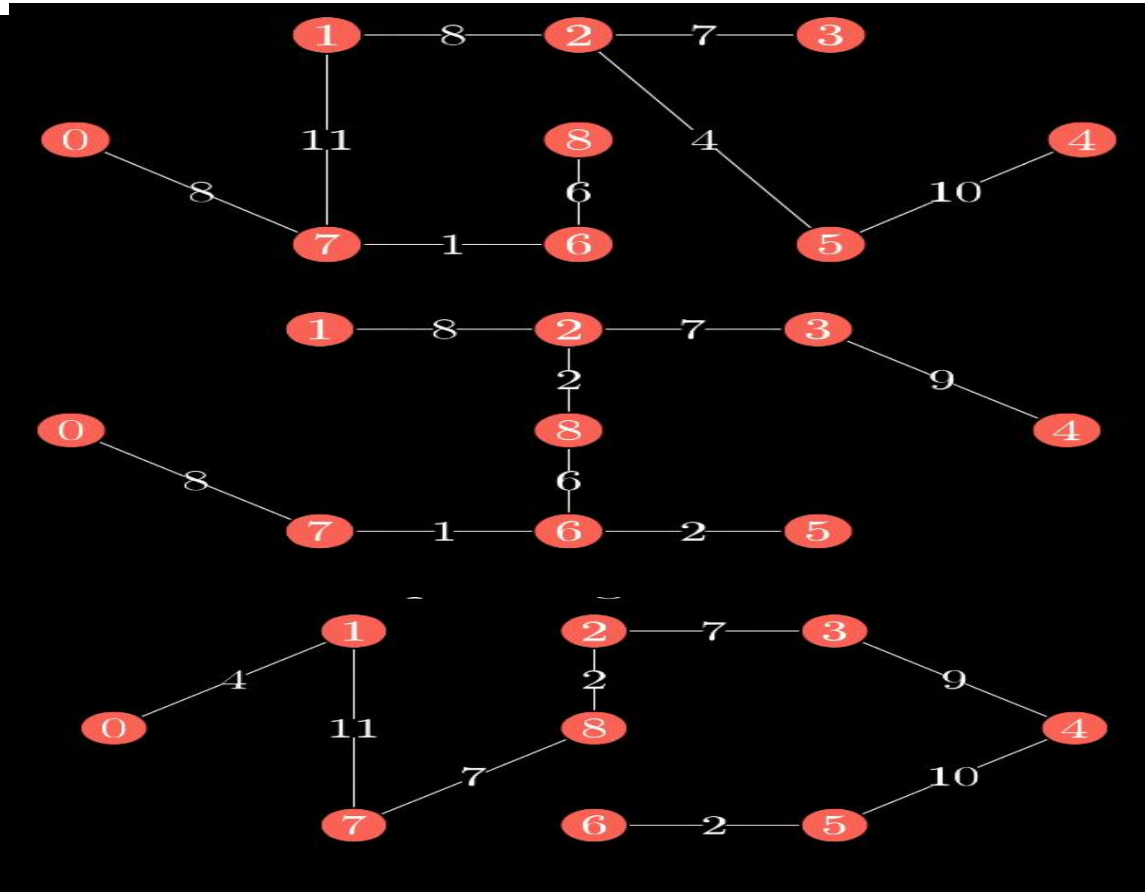
```cpp
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 1e5+10;
const int INF = 1e9+7;
vector<pair<int,int> > E[MAXN];
queue<int> Q;
int n,e,u,v,d,inq[MAXN],dis[MAXN];
void SPFA(int be){
    for(int i=0;i<=n;++i)   dis[i] = INF;
    dis[be] = 0;
    Q.push(be);
    inq[be] = 1;
    while(!Q.empty()){
        int now = Q.front();
        Q.pop();
        inq[now] = 0;
        for(int i=0;i<E[now].size();++i){
            int to = E[now][i].first;
            int dist = E[now][i].second;
            if(dis[to]>dis[now]+dist){
                dis[to] = dis[now]+dist;
                if(inq[to]==0){
                    Q.push(to);
                    inq[to] = 1;
                }
            }
        }
    }
}
```

```cpp
int main(){
    cin>>n>>e;
    for(int i=1;i<=e;++i){
        cin>>u>>v>>d;
        E[u].push_back(pair<int,int>(v,d));
        E[v].push_back(pair<int,int>(u,d));
    }
    SPFA(0);
    for(int i=0;i<=n;++i){
        printf("%d %d\n",i,dis[i]);
    }
    return 0;
}
```

# 生成树

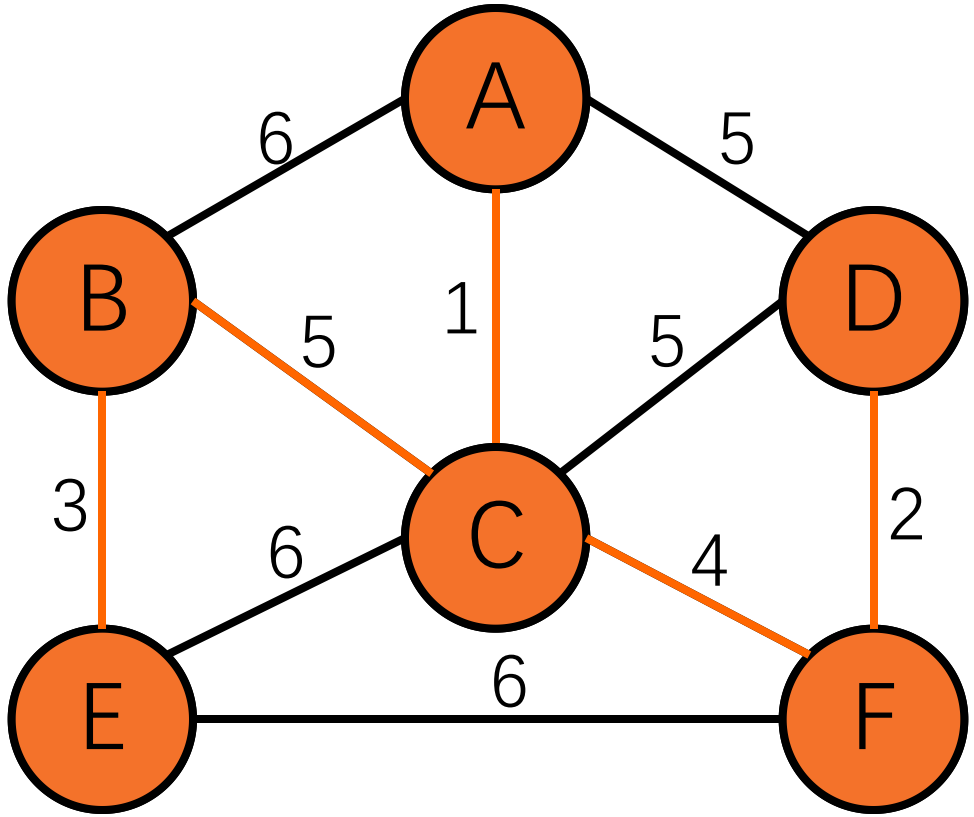
- 如果有 N 个顶点，则至少有 N-1 个边才能将 N 顶点连接起来形成连通无环图，我们称为**图的生成树**。



## 生活案例：

生活中最小生成树的应用十分广泛，比如：要连通n个城市需要n－1条边线路，那么怎么样建设才能使工程造价最小呢？可以把线路的造价看成权值求这几个城市的连通图的最小生成树。求最小造价的过程也就转化成求最小生成树的过程，则最小生成树表示使其造价最小的生成树。



# 最小生成树问题 Minimum Spanning Tree (MST)

- 一个有 n 个结点的连通图的生成树是原图的极小连通子图。
- 且包含原图中的所有 n 个结点。
- 并且有保持图连通的最少的边。
- 可以用kruskal（克鲁斯卡尔）算法或prim（普里姆）算法求出。

# 最小生成树问题 Minimum Spanning Tree (MST)

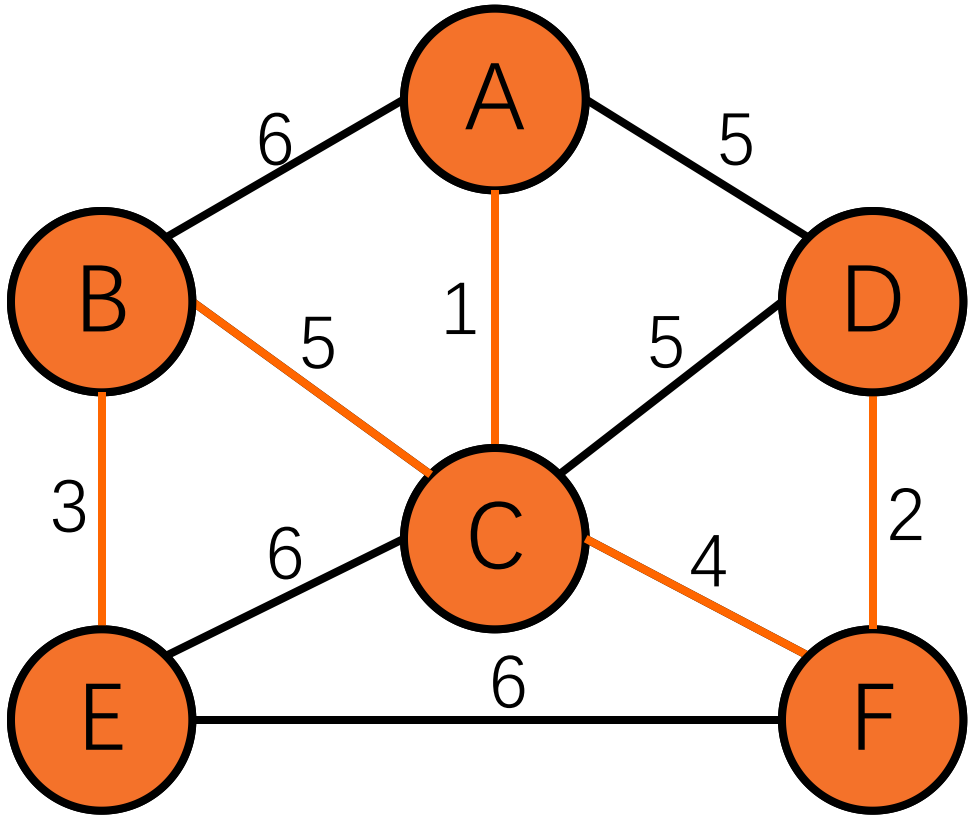
- 最小生成树：n个顶点的生成树很多，最小生成树就需要从这很多树中选一棵代价最小的生成树（即该树各边的代价之和最小）。

- 构造最小生成树的准则：
  - 必须只使用该网络中的边来构造最小生成树;
  - 必须使用且仅使用n-1条边来联络网络中的n个顶点;
  - 使用产生回路的边。

# Prime算法

- 思路:
- 此算法可以称为“**加点法**”，每次**迭代选择代价最小的边对应的点**，加入到最小生成树中。算法从某**一个顶点s**开始，逐渐长大覆盖整个连通网的所有顶点。
  - 1、图的所有顶点集合为V；初始令集合u={s},v=V−u;
  - 2、在两个集合u,v能够组成的边中，选择一条代价最小的边(u0,v0)，加入到最小生成树中，并把v0并入到集合u中。
  - 3、重复上述步骤，直到最小生成树有**n-1条边或者n个顶点**为止。

| pri | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Dis[] | 0 | 6 | 1 | 5 | inf | inf |
| Visit[] | Y | N | N | N | N | N |




通过[illegible]。

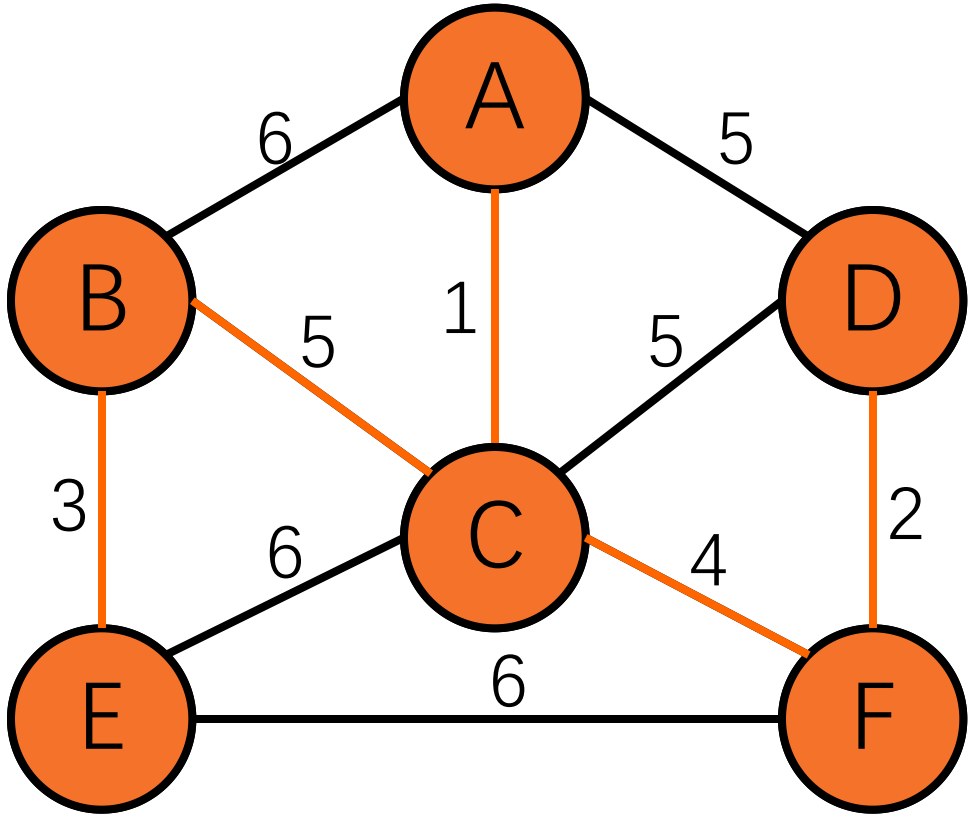
小生成树。

最小生成树和为：15

# Kruskal 算法

- **思路：**
  - 此算法可以称为“**加边法**”，初始最小生成树边数为0，每迭代一次就**选择一条**满足条件的**最小代价边**，加入到最小生成树的边集合里。
  - 1. 把图中的所有边按代价**从小到大排序**;
  - 2. 把图中的n个顶点看成独立的**n棵树组成的森林**;
  - 3. 按权值从小到大选择边，所选的边连接的**两个顶点ui,vi,**应属于**两颗不同的树**，则成为最小生成树的一条边，并将这两颗树**合并作为一颗树**。
  - 4. 重复(3),直到所有顶点都在一颗树内或者有**n-1条边为止**。

| krus | A | B | C | D | E | F |
| --- | --- | --- | --- | --- | --- | --- |
| 并查集 | 1 | 2 | 3 | 4 | 5 | 6 |




最小生成树长度为：1+2+3+4+5 = 15

# 并查集?