Graph

Breadth-First Search (BFS)

```
#include<iostream>
#include<queue>
#include<vector>
using namespace std;

vector<vector<int>> graph;

void BFS(int start)
{
    int n=graph.size();
    // 记录顶点是否被访问过
    vector<bool> visited(n, false);
    //记录从起点到各顶点的距离
    vector<int> distance(n, -1);
    //初始化起点
    visited[start] = true;
    distance[start]=0;
    queue<int> q;
    q.push(start);
    while(!q.empty())
    {
        int u=q.front();
        q.pop();
        //遍历u的所有邻接顶点
        for(int v:graph[u])
        {
            if(!visited[v])
            {
                visited[v] = true;
                distance[v]=distance[u]+1;
                q.push(v);
            }
        }
    }
```

```
    for (int i = 0;i<n;i++)
    {
        cout<<"distance from"<<start<<"to"<<i<<"is"<<distance[i]<<endl;
    }
}



int main()
{
    // int n,m;
    // cin>>n>>m;
    // graph.resize(n);
    // for(int i=0;i<m;i++)
    // {
    //     int u,v;
    //     cin>>u>>v;
    //     graph[u].push_back(v);
    //     graph[v].push_back(u);
    // }
    BFS(0);

    return 0;
}
```

Depth-First Search(DFS)

Explore

Figure 3.3 Finding all nodes reachable from a particular node.

```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

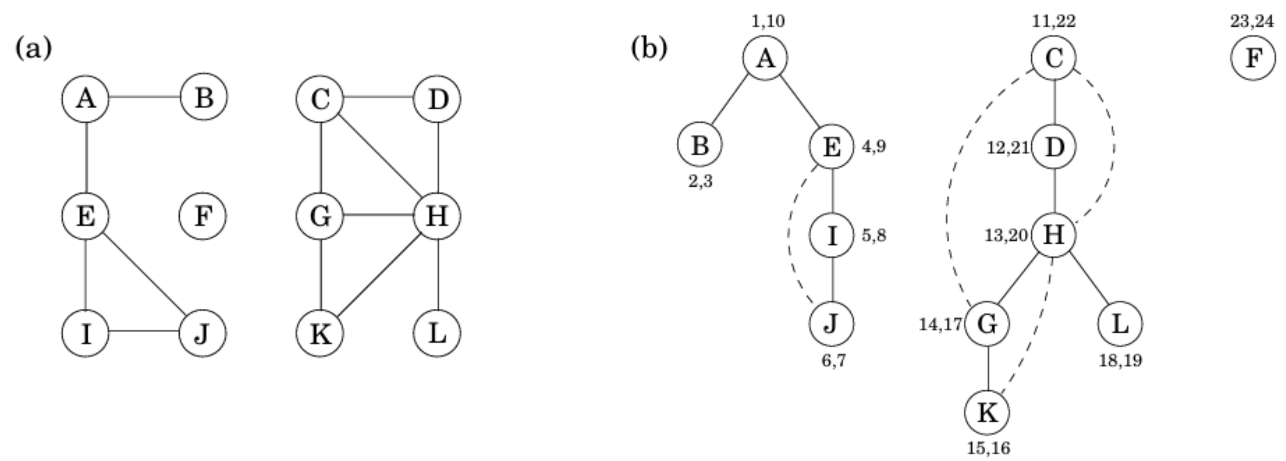
undirected graph

Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

connectivity in undirected graphs

Figure 3.6 (a) A 12-node graph. (b) DFS search forest.



To achieve this goal:

for all v belongs to V: visited[v]=false;

......ccnum[v]=null;

cc=0;

for all v belongs to V: if not visited[v]

..cc++;

..explore(G,v);

and we alse need to modify the **explore(G,v)**

Explore(G,v):

visited[v]=true;

ccnum[v]=cc;

for all (v,u) belongs to E(this pseudo code means that all u that connected with v):

if not visited[u] :

explore(G,u);

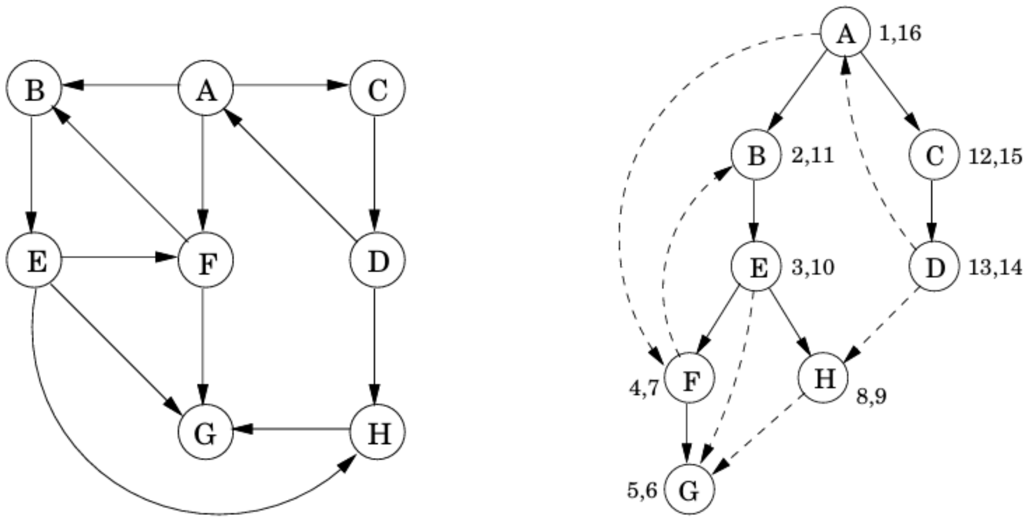
Previsit and postvisit orderings

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

pseudo code:

```
// 全局变量，用于记录时间戳
time = 0

// 维护一个栈来记录节点在栈中的时间
Maintain a global stack for time on stack

DFS(G):
    for u in V:
        u.visited = False
        u.previsit = -1
        u.postvisit = -1
    for u in V:
        if not u.visited:
            Explore(u)

Explore(u):
    u.visited = True
    // Pre - visit操作
    time = time + 1
    u.previsit = time
    for v in adj(u):
        if not v.visited:
            Explore(v)
    // Post - visit操作
    time = time + 1
    u.postvisit = time
```

Figure 3.7 DFS on a directed graph.

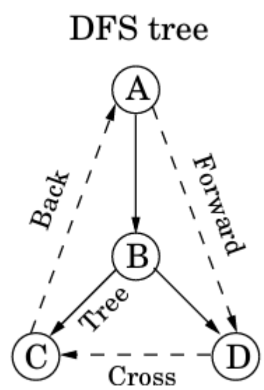


Tree edges are actually part of the DFS forest.

Forward edges lead from a node to a *nonchild* descendant in the DFS tree.

Back edges lead to an ancestor in the DFS tree.

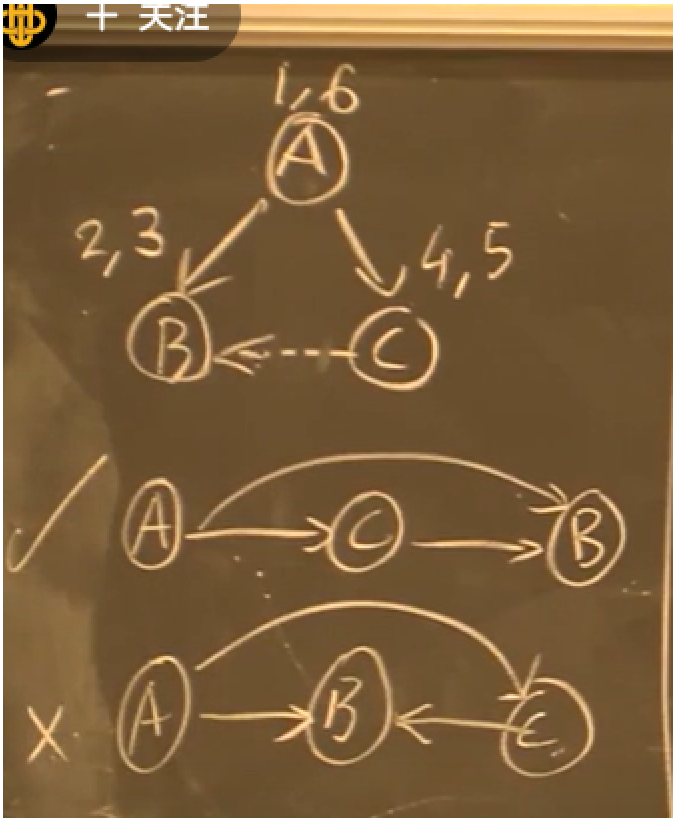
Cross edges lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).



it follows that they, too, can be read off from `pre` and `post` numbers. Here is a summary of the various possibilities for an edge (u, v):

`pre`/`post` *ordering for* (u, v)	*Edge type*
$\underset{u}{[} \quad \underset{v}{[} \quad \underset{v}{]} \quad \underset{u}{]}$	Tree/forward
$\underset{v}{[} \quad \underset{u}{[} \quad \underset{u}{]} \quad \underset{v}{]}$	Back
$\underset{v}{[} \quad \underset{v}{]} \quad \underset{u}{[} \quad \underset{u}{]}$	Cross

we want all pots to be forward and here is a subroutine called **topological sort**. it is easy:we just need output vectices in descending post numbers(降序输出顶点后序)



here is an example.it tells us we can in desending post numbers but can not increasing pre number.

isCyclicUtil

here is a code,but please focus on the isCyclicUtil function.

```
#include <iostream>
#include <vector>

class Graph {
private:
    int V;
    std::vector<std::vector<int>> adjMatrix;
    bool isCyclicUtil(int v, std::vector<int>& visited, std::vector<int>&
recStack);
public:
    Graph(int vertices);
    bool isCyclic();
};

// 构造函数
```

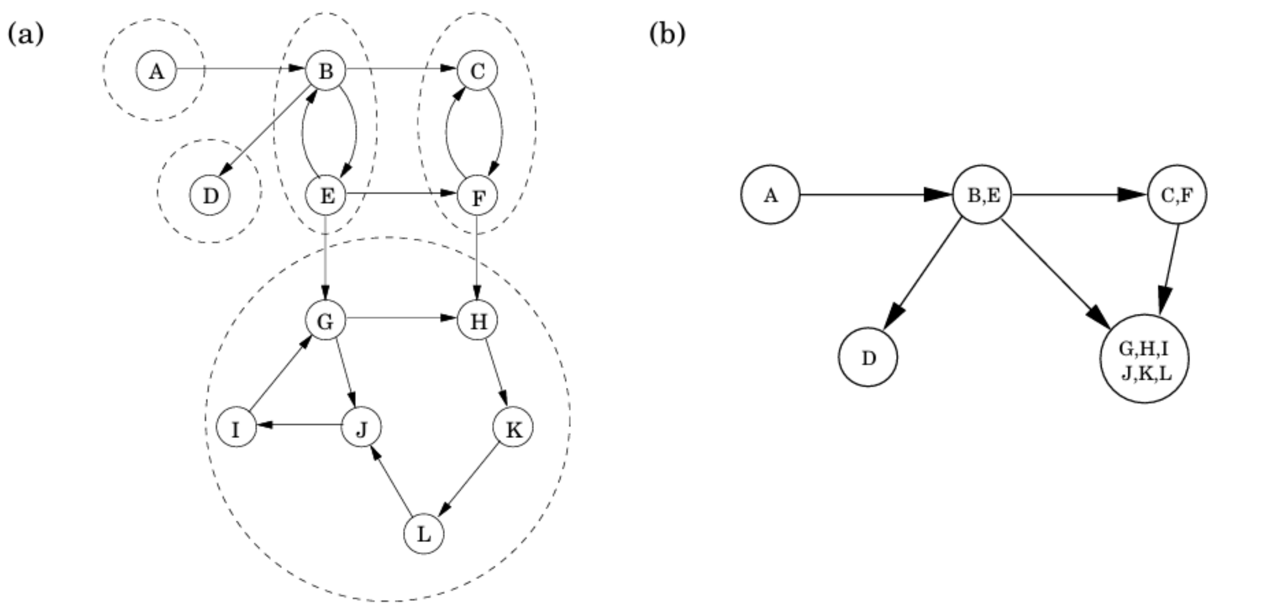
```cpp
Graph::Graph(int vertices) {
    V = vertices;
    adjMatrix.resize(V, std::vector<int>(V, 0));
}

// 判断是否有环的辅助函数
bool Graph::isCyclicUtil(int v, std::vector<int>& visited, std::vector<int>&
recStack) {
    if (visited[v] == 0) {
        // 标记节点为正在访问
        visited[v] = 1;
        recStack[v] = 1;

        for (int u = 0; u < V; u++) {
            if (adjMatrix[v][u] == 1) {
                if (!visited[u] && isCyclicUtil(u, visited, recStack))
                    return true;
                else if (recStack[u])
                    return true;
            }
        }
    }
    // 标记节点为已访问且不在当前搜索路径
    recStack[v] = 0;
    return false;
}

// 判断是否有环
bool Graph::isCyclic() {
    std::vector<int> visited(V, 0);
    std::vector<int> recStack(V, 0);
    for (int i = 0; i < V; i++) {
        if (isCyclicUtil(i, visited, recStack))
            return true;
    }
    return false;
}

int main() {
    // 示例：创建一个4个顶点的图，这里手动设置邻接矩阵
```

```
    Graph g(4);
    g.adjMatrix[0][1] = 1;
    g.adjMatrix[0][2] = 1;
    g.adjMatrix[1][2] = 1;
    g.adjMatrix[2][0] = 1;
    g.adjMatrix[2][3] = 1;
    g.adjMatrix[3][3] = 1;

    if (g.isCyclic())
        std::cout << "Graph contains a cycle" << std::endl;
    else
        std::cout << "Graph doesn't contain a cycle" << std::endl;

    return 0;
}
```

Connectivity for directed graphs

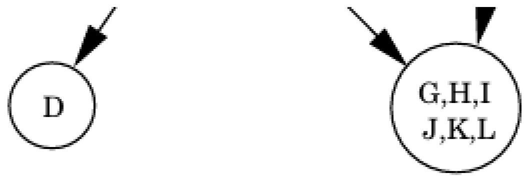
102 *Algorithms*

Figure 3.9 (a) A directed graph and its strongly connected components. (b) The meta-graph.

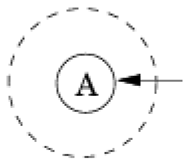


all those in the same circle is **Strongly connected components(SCC)**.

this two who had no next SCC is **Sink SCC**.



this one who had no previous SCC is **Source SCC**.



if we start **Explore(G,x)** when **x** is a element of one of Sink SCC,we could find all the Sink SCC. that is useful!
because then we can delete the Sink SCC we found and then Explore a new Sink SCC then we will get all SCC and some other informations!!!
But how can we find the Sink SCC? here is a way:
we can find the source in the directed graph
here is code in python to achieve this:

- 在每次从一个未访问顶点开始 DFS 时，递归地访问其所有未访问的邻接顶点。
- 例如，使用递归方式实现的 DFS 伪代码如下：

收起

python

```python
visited = [False] * n  # n是图中的顶点数量

def dfs(u):
    visited[u] = True
    for v in GR[u]:  # GR是有向图的邻接表表示
        if not visited[v]:
            dfs(v)
```

- 然后在主程序中，对每个未访问顶点调用 `dfs` 函数：

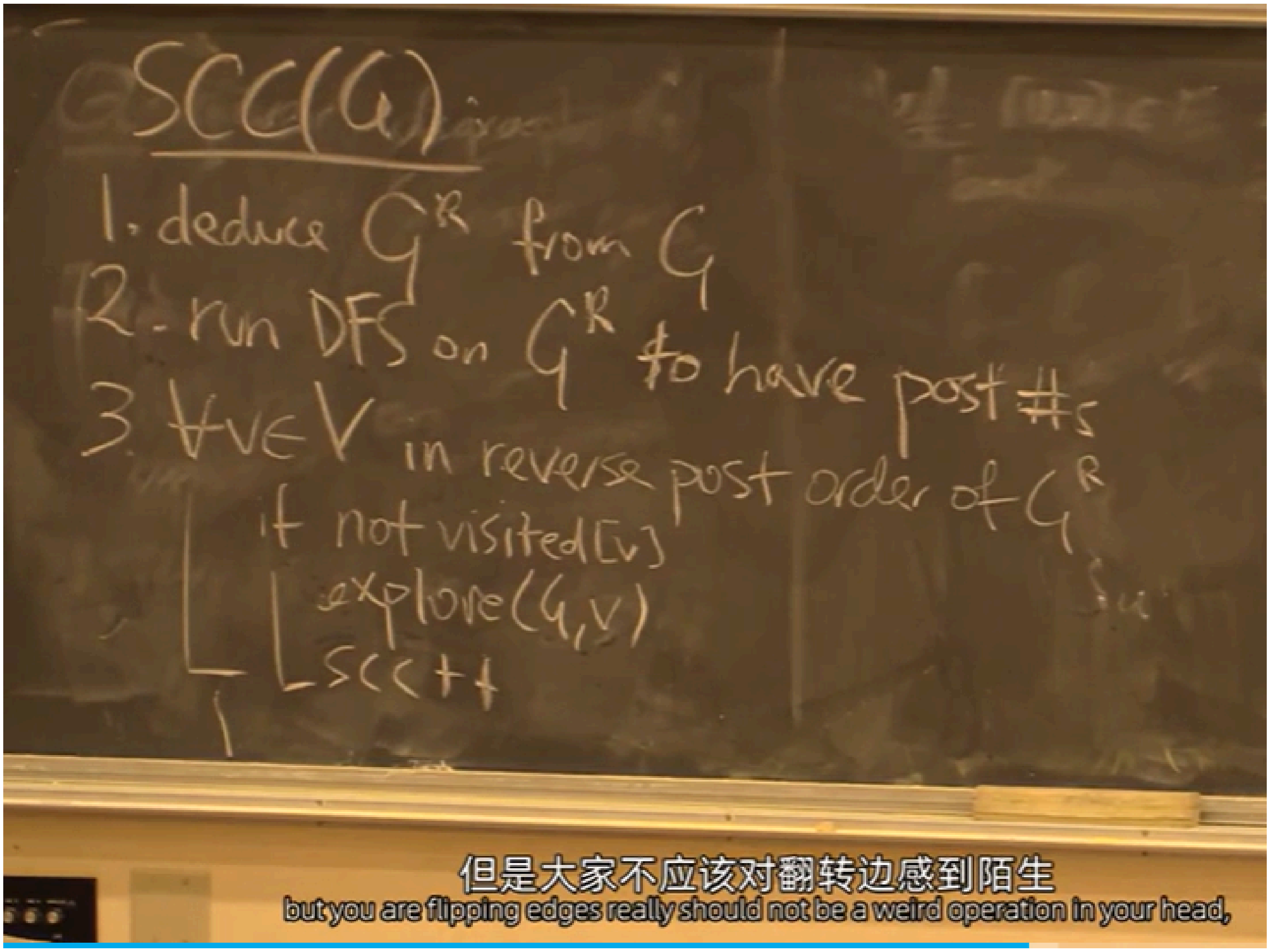
收起

python

```python
for u in range(n):
    if not visited[u]:
        dfs(u)
```

then we let all arrows in directed graph reverse.

we could find a element **x** in Sink SCC. Then we could Explore(G,**x**),we will get the Sink SCC!



Rreadth-First Search(BFS)

for those which paths are all 1.

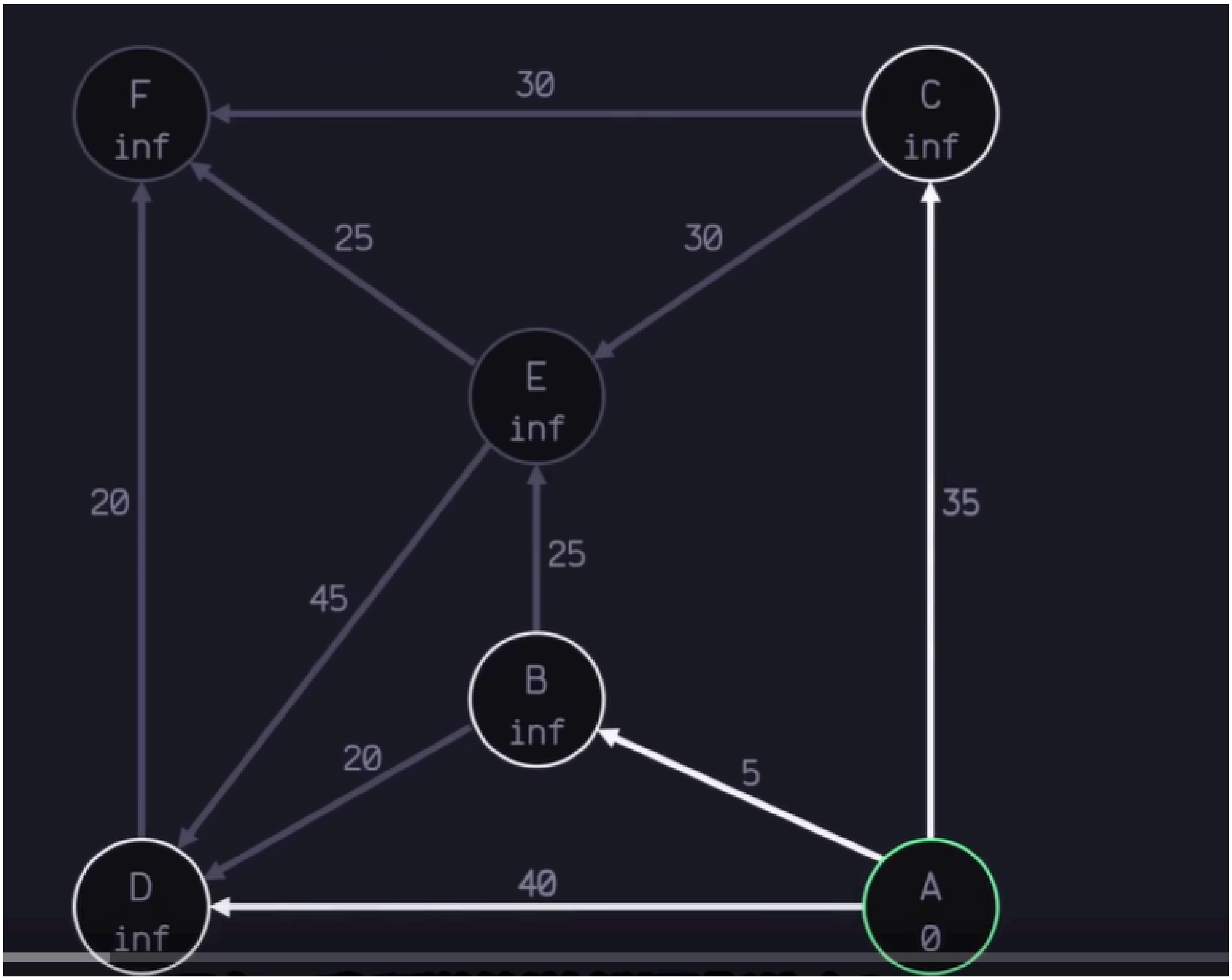
```
#include<iostream>
#include<queue>
#include<vector>
using namespace std;

vector<vector<int>> graph;

void BFS(int start)
{
    int n=graph.size();
    // 记录顶点是否被访问过
    vector<bool> visited(n, false);
    //记录从起点到各顶点的距离
```

```
    vector<int> distance(n, -1);
    //初始化起点
    visited[start] = true;
    distance[start]=0;
    queue<int> q;
    q.push(start);
    while(!q.empty())
    {
        int u=q.front();
        q.pop();
        //遍历u的所有邻接顶点
        for(int v:graph[u])
        {
            if(!visited[v])
            {
                visited[v] = true;
                distance[v]=distance[u]+1;
                q.push(v);
            }
        }
    }
    for (int i = 0;i<n;i++)
    {
        cout<<"distance from"<<start<<"to"<<i<<"is"<<distance[i]<<endl;
    }
}


int main()
{
    // int n,m;
    // cin>>n>>m;
    // graph.resize(n);
    // for(int i=0;i<m;i++)
    // {
    //     int u,v;
    //     cin>>u>>v;
    //     graph[u].push_back(v);
    //     graph[v].push_back(u);
    // }
```

```
    BFS(0);

    return 0;
}
```

The Shortest Path

background:if we have this problem:in the graph as follow, if we start from A what is the shortest paths from A to others?



Dijkstra Algorithm

when the paths all are positive numbers, we can use Dijkstra Algorithm,one of Greedy Algorithm.

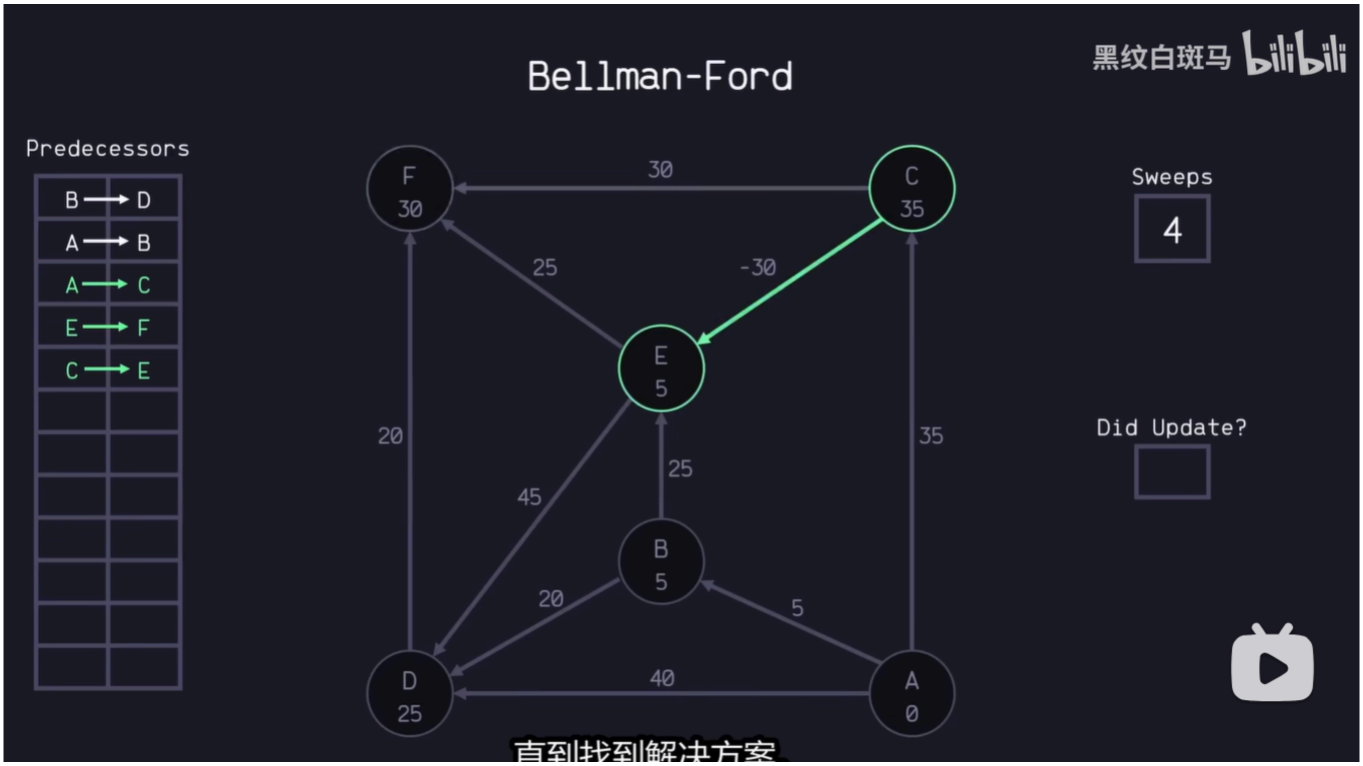
```
#include<iostream>
#include<vector>
#include<climits>
```

```
#include<queue>
using namespace std;

//表示无穷大，用于初始化距离
const int INF = INT_MAX;

//迪杰斯特拉算法实现
void Dijkstra(vector<vector<int>>&graph,int source)
{
    int n=graph.size();
    vector<int> dist(n, INF);//距离数组，初始化为无穷大
    dist[source] = 0;//起点到自己的距离为0

    //优先队列，存储（距离，顶点）对
    priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>
pq;
    pq.push({0,source});

    while(!pq.empty())
    {
        int u=pq.top().second;
        pq.pop();

        //遍历所有邻接顶点
        for (int v = 0; v < n;++v)
        {
            if(graph[u][v]!=0&&dist[v]>dist[u]+graph[u][v])
            {
                dist[v] = dist[u] + graph[u][v];
                pq.push({dist[v],v});
            }


        }
    }

    //输出最短路径
    cout<<"顶点"<<source<<"到其他顶点的最短路径距离"<<endl;
    for (int i = 0;i<n;i++)
    {
        if(dist[i]==INF)
```

```
        {
            cout << "顶点" << i << "不可到达" << endl;
        }
        else
        {
            cout << "到顶点" << i << "的距离为" << dist[i] << endl;
        }
    }
}
int main() {
    // 示例图的邻接矩阵表示
    vector<vector<int>> graph = {
        {0, 10, 20, 0, 0, 0},
        {10, 0, 0, 50, 10, 0},
        {20, 0, 0, 20, 33, 0},
        {0, 50, 20, 0, 20, 2},
        {0, 10, 33, 20, 0, 1},
        {0, 0, 0, 2, 1, 0}
    };

    int source = 0; // 源点
    Dijkstra(graph, source);

    return 0;
}
```

Bellman-Ford Algorithm

If there has minus numbers,



we should use Bellman-Ford Algorithm.

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

// 边的结构体，用于表示图中的边
struct Edge
{
    int src;
    int dest;
    int weight;
};

// bellman-Ford算法的实现
void bellmanFord(const vector<Edge> &edges, int numVertices, int source)
{
    // 存储从源点到各个顶点的最短距离，初始化为无穷大
    vector<int> dist(numVertices, INT_MAX);
    dist[source] = 0;

    // 进行numVertices-1次迭代，对所有边进行松弛操作
```

```
    for (int i = 0; i < numVertices - 1; ++i)
    {
        bool updated = false;
        for (const Edge &edge : edges)
        {
            int u = edge.src;
            int v = edge.dest;
            int weight = edge.weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            {
                dist[v] = dist[u] + weight;
                updated = true;
            }
        }
        // 如果没有状态更新，提前终止
        if (!updated)
        {
            break;
        }
    }

    // 检查是否存在负权重环
    for (const Edge &edge : edges)
    {
        int u = edge.src;
        int v = edge.dest;
        int weight = edge.weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
        {
            cout << "图中存在负权重环" << endl;
            return;
        }
    }

    // 输出最短路径
    cout << "顶点 " << source << " 到其他顶点的最短路径距离：" << endl;
    for (int i = 0; i < numVertices; ++i)
    {
        if (dist[i] == INT_MAX)
        {
```

```
            cout << "顶点 " << i << " 不可达" << endl;
        }
        else
        {
            cout << "到顶点 " << i << " 的距离为 " << dist[i] << endl;
        }
    }
}

int main()
{
    // 示例图的边列表表示
    vector<Edge> edges =
        {
            {0, 1, -1},
            {0, 2, 4},
            {1, 2, 3},
            {1, 3, 2},
            {1, 4, 2},
            {3, 2, 5},
            {3, 1, 1},
            {4, 3, -3}};

    int numVertices = 5; // 顶点数量
    int source = 0;      // 源点
    bellmanFord(edges, numVertices, source);

    return 0;
}
```

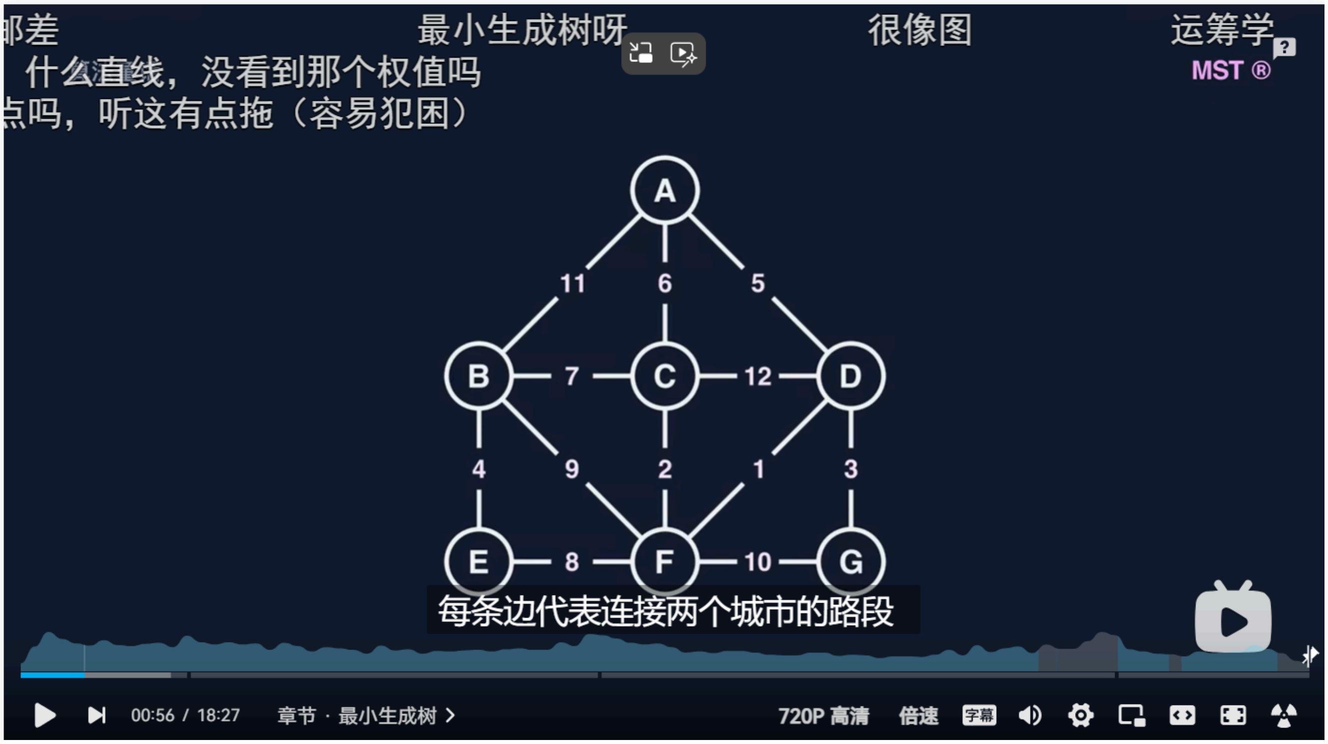
Minimum Spanning Tree

kruskal Algorithm

We need find the minimum ways to connect all citys in the follow picture.

Kruskal一往无前，并查集鼎力相助（算法童话第一回）

8.3万 530 2022-08-25 11:11:11 未经作者授权，禁止转载

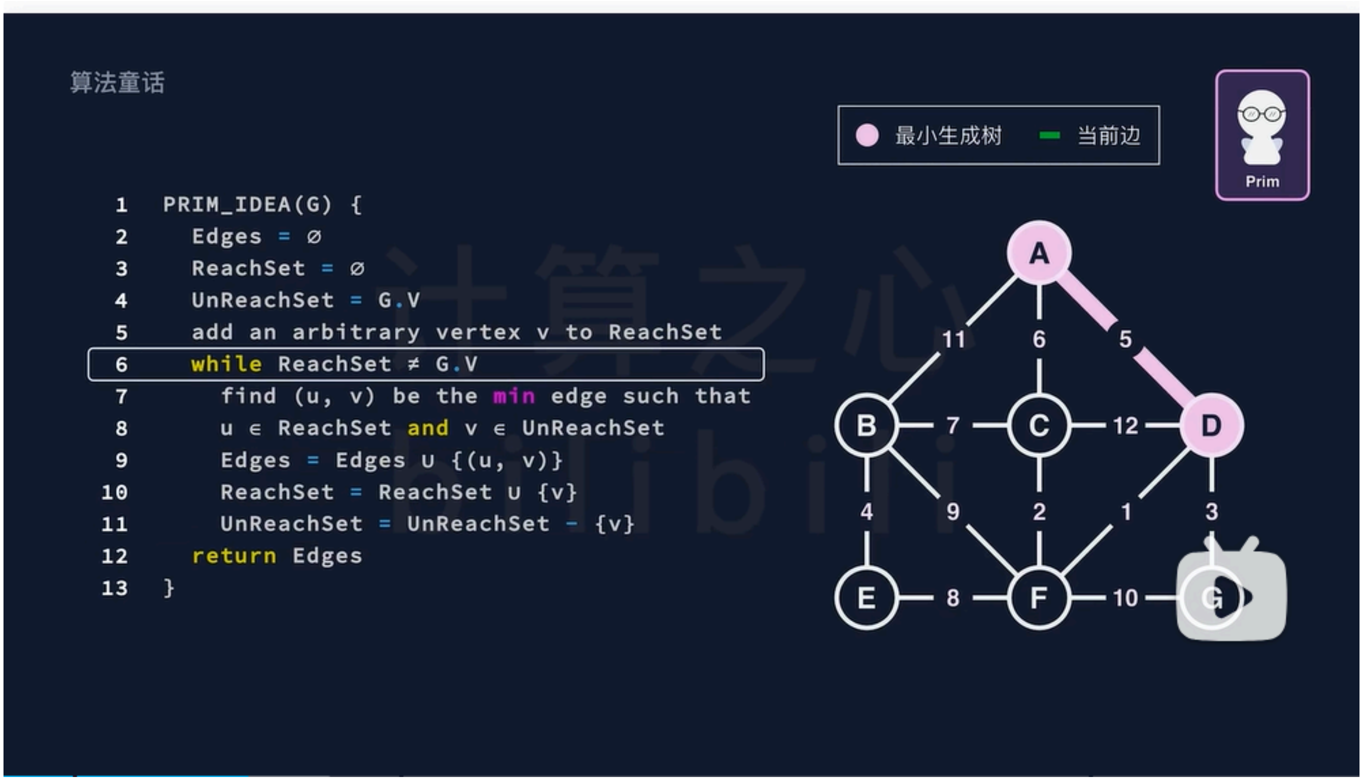



```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

// 边，记录边两边结点，记录边的长度
struct Edge
{
    int src;
    int dest;
    int weight;
};

// 并查集结构体,用于检查环
struct DisjointSets
{
    vector<int> parent;
    vector<int> rank;
    // 构造函数,相当于make_set()
    DisjointSets(int n)
```

```
    {
        // 初始化指针指向自己，且rank初始化为0
        parent.resize(n);
        rank.resize(n, 0);
    }
    int find(int x)
    {
        return parent[x] = x ? x : find(parent[x]);
    }

    void unite(int x, int y)
    {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY)
        {
            return;
        }
        else
        {
            if (rank[rootX] < rank[rootY])
            {
                parent[rootX] = rootY;
            }
            else if (rank[rootX] > rank[rootY])
            {
                parent[rootY] = rootX;
            }
            else
            {
                rank[rootY]++;
                parent[rootX] = rootY;
            }
        }
    }
};

vector<Edge> Kruskal(vector<Edge> &edges, int numVertices)
{
    DisjointSets ds(numVertices); // 初始化那些结点
```

```
    vector<Edge> mst;               // 存储最小生成树的边
    sort(edges.begin(), edges.end(), [](Edge a, Edge b)
         { return a.weight < b.weight; }); // 按照从小到大将边排序
    for (Edge &edge : edges)
    {
        int one = edge.src;
        int another = edge.dest;
        if (ds.find(one) != ds.find(another)) // 判断加入这条边后有没有成环
        {
            mst.push_back(edge);    // 加入最小生成树
            ds.unite(one, another); // 将两个点加入到同一个并查集中
        }
    }
    return mst;
}

int main()
{
    return 0;
}
```

Prim Algorithm



```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

// 边的结构体
struct Edge
{
    int dest;
    int weight;
};

//使用优先队列的Prim算法实现
vector<Edge>Prim(vector<vector<Edge>>graph,int numVertices)//输入存储边的容器，并
输入点的个数
{
    vector<Edge> mst;//用以存储最终的最小生成树
    vector<bool> in_mst(numVertices, false);//记录该点是否在最小生成树
    vector<int> key(numVertices, INT_MAX);//记录每个顶点的最小权重
    vector<int> parents(numVertices, -1);//记录每个顶点的父节点

    //优先队列，存储（权重，顶点）对
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
    //priority_queue是一个容器适配器，第一个参数代表存储的数据类型（此处为（权重，顶点）
对），
    //第二个参数代表用什么样的容器存储数据，vector是一种动态大小的数组容器，
    //第三个参数是定义优先级比较规则，此处的greater会先比较容器中所有pair对的第一个元素的
大小，然后比较第二个
    key[0] = 0;
    pq.push({0, 0});

    while(!pq.empty())
    {
        int u = pq.top().second;//pq.top()用来取到pq优先队列中优先级最高的那个pair，
second取到pair中的第二个元素，即顶点
        pq.pop();//优先队列保证队列中首个元素是优先级最高的，所以此处是将优先级最高的元素
踢出队列
```

```
        if(in_mst[u])
        {
            continue;//如果u已经在最小生成树中，则可以直接结束本次关于u的循环
        }
        in_mst[u] = true;//如果u不在最小生成树，先将其加入最小生成树（因为这是目前检查
到的可以从最小生成树中接触到的路程最小的点）
        //然后更新该点可以接触到的点
        for(const Edge&edge:graph[u])
        {
            int v = edge.dest;//v(vertice)代表edge将接触到的点
            int weight = edge.weight;//weight代表edge的权重
            if(!in_mst[v]&&(weight<key[v]))//如果v没在mst中且该边的权重小于更新前
mst到达v的最小权重
            {
                key[v] = weight;
                pq.push({key[v], v});
                parents[v] = u;
            }
        }
    }

    //构建最小生成树
    for (int i = 1; i < numVertices;++i)//从1开始是因为在上面已经将最小生成树的起点
{0，0}加入
    {
        if(parents[i]!=-1)
        {
            mst.push_back({parents[i], key[i]});
        }
    }
    return mst;
}

int main() {
     // 示例图的邻接表表示
    vector<vector<Edge>> graph = {
        {{1, 2}, {3, 6}},
        {{0, 2}, {2, 3}, {3, 8}, {4, 5}},
        {{1, 3}, {4, 7}},
        {{0, 6}, {1, 8}, {4, 9}},
```

```
        {{1, 5}, {2, 7}, {3, 9}}
    };

    int numVertices = 5; // 顶点数量

    vector<Edge> mst = Prim(graph, numVertices);

    // 输出最小生成树的边
    cout << "最小生成树的边:" << endl;
    for (const Edge& edge : mst) {
        cout << edge.dest << " -- " << edge.weight << endl;
    }

    return 0;
}
```