第八讲：图（下）

8.1 最小生成树问题

8.1.1 最小生成树（Minimum Spanning Tree）

如图 1 所示。

图 1

它是一棵树：无回路；|V|个顶点一定有|V|-1 条边；

它是生成树：包含全部顶点；|V|-1 条边都在图里。在图 1 中，第 2/3/4 个图都是图 1 的生成树，可以看出，**生成树中任加一条边都一定构成回路**。

最小：边的权重和**最小**。

显然可以得出，**最小生成树存在<->图连通**。

8.1.2 贪心算法

贪：每一步都要最好的。好：权重最小的边。

需要约束：只能用图里有的边；只能正好用掉|V|-1 条边；不能有回路。

8.1.3 Prim 算法（密集图）—让一棵小树长大

如图 2 所示，我们选择 v1 作为源节点，选择权重最小的（为 1），到 v4；然后我们看这个图，权重最小的为 2 有两个边，分别到 v2 和 v3，我们先选择连到 v2，再用 v4 连接到 v3；为了不构成回路，我们需要选择权重为 4 的由 v4 到 v7 的边；再选择权重为 1 的 v6，再选择权重为 6 的 v5。

图 2

其伪码描述如下，其中 dist[V]=E(s,v)或正无穷，parent[s]=-1：

```
void Prim()
{
	MST={s};
	while(1){
		V=未收录顶点中 dist 最小者;
		if(V 不存在)
			break;
		将 V 收录进 MST,dist[V]=0;
		for(V 的每一个邻接点 W)
			if(dist[W]!=0)/*即这个点未被收入*/
				if(E[V,W]<dist[W]){
					dist[W]=E[V,W];
					parent[W]=v;
				}
	}
	if(MST 中收的顶点不到|V|个)
		Error("生成树不存在");
```

```
}
```

完整代码:

```
/* 邻接矩阵存储 - Prim 最小生成树算法 */

Vertex FindMinDist( MGraph Graph, WeightType dist[] )
{ /* 返回未被收录顶点中 dist 最小者 */
    Vertex MinV, V;
    WeightType MinDist = INFINITY;

    for (V=0; V<Graph->Nv; V++) {
        if ( dist[V]!=0 && dist[V]<MinDist) {
            /* 若 V 未被收录，且 dist[V]更小 */
            MinDist = dist[V]; /* 更新最小距离 */
            MinV = V; /* 更新对应顶点 */
        }
    }
    if (MinDist < INFINITY) /* 若找到最小 dist */
        return MinV; /* 返回对应的顶点下标 */
    else return ERROR;  /* 若这样的顶点不存在，返回-1 作为标记 */
}

int Prim( MGraph Graph, LGraph MST )
{ /* 将最小生成树保存为邻接表存储的图 MST，返回最小权重和 */
    WeightType dist[MaxVertexNum], TotalWeight;
    Vertex parent[MaxVertexNum], V, W;
    int VCount;
    Edge E;

    /* 初始化。默认初始点下标是 0 */
       for (V=0; V<Graph->Nv; V++) {
        /* 这里假设若 V 到 W 没有直接的边，则 Graph->G[V][W]定义为 INFINITY */
           dist[V] = Graph->G[0][V];
           parent[V] = 0; /* 暂且定义所有顶点的父结点都是初始点 0 */
    }
    TotalWeight = 0; /* 初始化权重和       */
    VCount = 0;      /* 初始化收录的顶点数 */
    /* 创建包含所有顶点但没有边的图。注意用邻接表版本 */
    MST = CreateGraph(Graph->Nv);
    E = (Edge)malloc( sizeof(struct ENode) ); /* 建立空的边结点 */

    /* 将初始点 0 收录进 MST */
    dist[0] = 0;
    VCount ++;
    parent[0] = -1; /* 当前树根是 0 */
```

```
    while (1) {
        V = FindMinDist( Graph, dist );
        /* V = 未被收录顶点中 dist 最小者 */
        if ( V==ERROR ) /* 若这样的 V 不存在 */
            break;      /* 算法结束 */

        /* 将 V 及相应的边<parent[V], V>收录进 MST */
        E->V1 = parent[V];
        E->V2 = V;
        E->Weight = dist[V];
        InsertEdge( MST, E );
        TotalWeight += dist[V];
        dist[V] = 0;
        VCount++;

        for( W=0; W<Graph->Nv; W++ ) /* 对图中的每个顶点 W */
            if ( dist[W]!=0 && Graph->G[V][W]<INFINITY ) {
            /* 若 W 是 V 的邻接点并且未被收录 */
                if ( Graph->G[V][W] < dist[W] ) {
                /* 若收录 V 使得 dist[W]变小 */
                    dist[W] = Graph->G[V][W]; /* 更新 dist[W] */
                    parent[W] = V; /* 更新树 */
                }
            }
    } /* while 结束*/
    if ( VCount < Graph->Nv ) /* MST 中收的顶点不到|V|个 */
       TotalWeight = ERROR;
    return TotalWeight;    /* 算法执行完毕，返回最小权重和或错误标记 */
}
```

8.1.4 Kruskal 算法（稀疏图）—将森林合并成树

寻找权重最短的边，初始的情况下认为每一个顶点都是一棵树，通过不断把边收进来，就把两棵树并成了一棵树，最后把所有节点并成一棵树。

伪码如下：

```
void Kruskal(Graph G)
{
    MST = {};
    while(MST 中不到|V-1|条边&&E 中还有边){
        从 E 中去一条权重最小的边 E)(V,W);/*最小堆*/
        将 E(V,W)从 E 中删除;
        if(E(V,W)不在 MST 中构成回路)/*并查集*/
            将 E(V,W)加入 MST;
        else
```

```
            彻底无视 E(V,W);
    }
    if(MST 中不到|V|-1 条边)
        ERROR("生成树不存在");
}
```

复杂度为 T=O(|E|log|E|)

完整代码：

```
/* 邻接表存储 - Kruskal 最小生成树算法 */

/*-------------------- 顶点并查集定义 --------------------*/
typedef Vertex ElementType; /* 默认元素可以用非负整数表示 */
typedef Vertex SetName;      /* 默认用根结点的下标作为集合名称 */
typedef ElementType SetType[MaxVertexNum]; /* 假设集合元素下标从 0 开始 */

void InitializeVSet( SetType S, int N )
{ /* 初始化并查集 */
    ElementType X;

    for ( X=0; X<N; X++ ) S[X] = -1;
}

void Union( SetType S, SetName Root1, SetName Root2 )
{ /* 这里默认 Root1 和 Root2 是不同集合的根结点 */
    /* 保证小集合并入大集合 */
    if ( S[Root2] < S[Root1] ) { /* 如果集合 2 比较大 */
        S[Root2] += S[Root1];     /* 集合 1 并入集合 2  */
        S[Root1] = Root2;
    }
    else {                        /* 如果集合 1 比较大 */
        S[Root1] += S[Root2];     /* 集合 2 并入集合 1  */
        S[Root2] = Root1;
    }
}

SetName Find( SetType S, ElementType X )
{ /* 默认集合元素全部初始化为-1 */
    if ( S[X] < 0 ) /* 找到集合的根 */
        return X;
    else
        return S[X] = Find( S, S[X] ); /* 路径压缩 */
}

bool CheckCycle( SetType VSet, Vertex V1, Vertex V2 )
{ /* 检查连接 V1 和 V2 的边是否在现有的最小生成树子集中构成回路 */
```

```
    Vertex Root1, Root2;

    Root1 = Find( VSet, V1 ); /* 得到 V1 所属的连通集名称 */
    Root2 = Find( VSet, V2 ); /* 得到 V2 所属的连通集名称 */

    if( Root1==Root2 ) /* 若 V1 和 V2 已经连通，则该边不能要 */
        return false;
    else { /* 否则该边可以被收集，同时将 V1 和 V2 并入同一连通集 */
        Union( VSet, Root1, Root2 );
        return true;
    }
}
/*-------------------- 并查集定义结束 --------------------*/

/*-------------------- 边的最小堆定义 --------------------*/
void PercDown( Edge ESet, int p, int N )
{ /* 改编代码 4.24 的 PercDown( MaxHeap H, int p )    */
  /* 将 N 个元素的边数组中以 ESet[p]为根的子堆调整为关于 Weight 的最小堆 */
    int Parent, Child;
    struct ENode X;

    X = ESet[p]; /* 取出根结点存放的值 */
    for( Parent=p; (Parent*2+1)<N; Parent=Child ) {
        Child = Parent * 2 + 1;
        if( (Child!=N-1) && (ESet[Child].Weight>ESet[Child+1].Weight) )
            Child++;  /* Child 指向左右子结点的较小者 */
        if( X.Weight <= ESet[Child].Weight ) break; /* 找到了合适位置 */
        else  /* 下滤 X */
            ESet[Parent] = ESet[Child];
    }
    ESet[Parent] = X;
}

void InitializeESet( LGraph Graph, Edge ESet )
{ /* 将图的边存入数组 ESet，并且初始化为最小堆 */
    Vertex V;
    PtrToAdjVNode W;
    int ECount;

    /* 将图的边存入数组 ESet */
    ECount = 0;
    for ( V=0; V<Graph->Nv; V++ )
        for ( W=Graph->G[V].FirstEdge; W; W=W->Next )
            if ( V < W->AdjV ) { /* 避免重复录入无向图的边，只收 V1<V2 的边 */
```

```
                ESet[ECount].V1 = V;
                ESet[ECount].V2 = W->AdjV;
                ESet[ECount++].Weight = W->Weight;
            }
    /* 初始化为最小堆 */
    for ( ECount=Graph->Ne/2; ECount>=0; ECount-- )
        PercDown( ESet, ECount, Graph->Ne );
}

int GetEdge( Edge ESet, int CurrentSize )
{ /* 给定当前堆的大小 CurrentSize，将当前最小边位置弹出并调整堆 */

    /* 将最小边与当前堆的最后一个位置的边交换 */
    Swap( &ESet[0], &ESet[CurrentSize-1]);
    /* 将剩下的边继续调整成最小堆 */
    PercDown( ESet, 0, CurrentSize-1 );

    return CurrentSize-1; /* 返回最小边所在位置 */
}
/*-------------------- 最小堆定义结束 --------------------*/


int Kruskal( LGraph Graph, LGraph MST )
{ /* 将最小生成树保存为邻接表存储的图 MST，返回最小权重和 */
    WeightType TotalWeight;
    int ECount, NextEdge;
    SetType VSet; /* 顶点数组 */
    Edge ESet;    /* 边数组 */

    InitializeVSet( VSet, Graph->Nv ); /* 初始化顶点并查集 */
    ESet = (Edge)malloc( sizeof(struct ENode)*Graph->Ne );
    InitializeESet( Graph, ESet ); /* 初始化边的最小堆 */
    /* 创建包含所有顶点但没有边的图。注意用邻接表版本 */
    MST = CreateGraph(Graph->Nv);
    TotalWeight = 0; /* 初始化权重和     */
    ECount = 0;      /* 初始化收录的边数 */

    NextEdge = Graph->Ne; /* 原始边集的规模 */
    while ( ECount < Graph->Nv-1 ) {  /* 当收集的边不足以构成树时 */
        NextEdge = GetEdge( ESet, NextEdge ); /* 从边集中得到最小边的位置 */
        if (NextEdge < 0) /* 边集已空 */
            break;
        /* 如果该边的加入不构成回路，即两端结点不属于同一连通集 */
        if ( CheckCycle( VSet, ESet[NextEdge].V1, ESet[NextEdge].V2 )==true ) {
```

```
                /* 将该边插入 MST */
                InsertEdge( MST, ESet+NextEdge );
                TotalWeight += ESet[NextEdge].Weight; /* 累计权重 */
                ECount++; /* 生成树中边数加 1 */
            }
        }
        if ( ECount < Graph->Nv-1 )
            TotalWeight = -1; /* 设置错误标记，表示生成树不存在 */

        return TotalWeight;
}
```

8.2 拓扑排序

8.2.1 拓扑排序

AOV（Activity On Vertex）网络

如图 3 所示，是指所有的真实活动是表现在顶点上的，顶点与顶点之间的有向边表现了顶点间的先后顺序。

图 3

所谓拓扑序是指：如果图中从 V 到 W 有一条有向路径，则 V 一定排在 W 之前，满足此条件的顶点序列称为一个拓扑序。获得一个拓扑序的过程就是拓扑排序。

AOV 网络如果有**合理（所谓不合理是指，网络形成了一个环，那就代表着 V 必须在 V 开始之前结束，自然不合理）**的拓扑序，则必定是**有向无环图**（Directed Acyclic Graph, DAG）。

所谓拓扑排序，就是每一次输出没有前序顶点的顶点。

伪码描述如下：

```
void TopSort()
{
        for(cnt=0;cnt<|V|;cnt++){
            v = 为输出的入度为 0 的顶点;/*简单粗暴法此步 O(|V|)，总为 O(|V|2)*/
            /*聪明的算法如下文所述*/
            if(这样的 V 不存在_{
                Error("图中有回路");
                break;
            }

            输出 V，或者记录 V 的输出序号;
            for(V 的每个邻接点 W)
                Indegree[W]--;
        }
}
```

聪明的算法：随时将入度变为 0 的顶点放到一个容器（数组、堆栈、队列等都行）中，下一次直接从这里面直接拿数据就可以了。利用队列的新的伪码描述：

```
void TopSort()
```

```
{
    for(图中每个顶点 V){
        if(Indegree[V]==0)
            Enqueue(V,Q);
    }
        while(!IsEmpty(Q){
            V=Dequeue(Q);
            输出 V，或者记录 V 的输出序号;cnt++;
            for(V 的每个邻接点 W)
            {
                if(--Indegree[W]==0)
                    Enqueue(W,Q);
            }
        }
        if(cnt!=|V|)
            Error("图中有回路");
}
```

时间复杂度 T=O(|V|+|E|)

完整代码：

```
/* 邻接表存储 - 拓扑排序算法 */

bool TopSort( LGraph Graph, Vertex TopOrder[] )
{ /* 对 Graph 进行拓扑排序， TopOrder[]顺序存储排序后的顶点下标 */
    int Indegree[MaxVertexNum], cnt;
    Vertex V;
    PtrToAdjVNode W;
       Queue Q = CreateQueue( Graph->Nv );

    /* 初始化 Indegree[] */
    for (V=0; V<Graph->Nv; V++)
        Indegree[V] = 0;

    /* 遍历图，得到 Indegree[] */
    for (V=0; V<Graph->Nv; V++)
        for (W=Graph->G[V].FirstEdge; W; W=W->Next)
            Indegree[W->AdjV]++; /* 对有向边<V, W->AdjV>累计终点的入度 */

    /* 将所有入度为 0 的顶点入列 */
    for (V=0; V<Graph->Nv; V++)
        if ( Indegree[V]==0 )
            AddQ(Q, V);

    /* 下面进入拓扑排序 */
    cnt = 0;
```

```
    while( !IsEmpty(Q) ){
        V = DeleteQ(Q); /* 弹出一个入度为 0 的顶点 */
        TopOrder[cnt++] = V; /* 将之存为结果序列的下一个元素 */
        /* 对 V 的每个邻接点 W->AdjV */
        for ( W=Graph->G[V].FirstEdge; W; W=W->Next )
            if ( --Indegree[W->AdjV] == 0 )/* 若删除 V 使得 W->AdjV 入度为 0 */
                AddQ(Q, W->AdjV); /* 则该顶点入列 */
    } /* while 结束*/

    if ( cnt != Graph->Nv )
        return false; /* 说明图中有回路, 返回不成功标志 */
    else
        return true;
}
```

8.2.2 关键路径

AOE（Activity On Edge）网络一般用于安排项目的工序。与 AOV 不同，AOE 的活动表示在边上，而节点代表活动到此结束。一般情况下，AOE 网络的图示为以下结构：

图 4

如图 5 所示，图中虚线，且权重为 0 表示的是，要继续执行 9/7 权重那里，4/5 这里必须都得走到了。

图 5

问题 1：整个工期有多长？

Earliest[0]=0;

Earliest[j]=max(<i,j>∈E){Earliest[i]+C<i,j>}

故 Earliest[8]=18

问题 2：哪几个组有机动时间（保证工期最长 18 天）？

方法是设置最后一个最晚完成时间为 18 天，然后往前推。

注意：虽然 7 倒推回去 5 只需要在第 10 天完工就可以，但是考虑到 4 必须在第 7 天完工，而 6 必须在第 16 天完工，又由于 4、5 同时完工才能往下走，所以 5 的最晚完成时间也必须和 4 同样，为第 7 天。

Latest[8]=18;

Latest[i]=(min<i,j>∈E){Latest[j]-C<i,j>}

所谓机动时间就是哪些组可以不用急着赶工

机动时间 D<i,j>=Latest[j]-Earliest[i]-C<i,j>

所谓关键路径就是整个流程中最需要关注的地方，哪些步骤是一点也不能耽误的，只要它耽误了，整个流程都要耽误，所以它是**绝对不允许延误的活动**组成的路径。