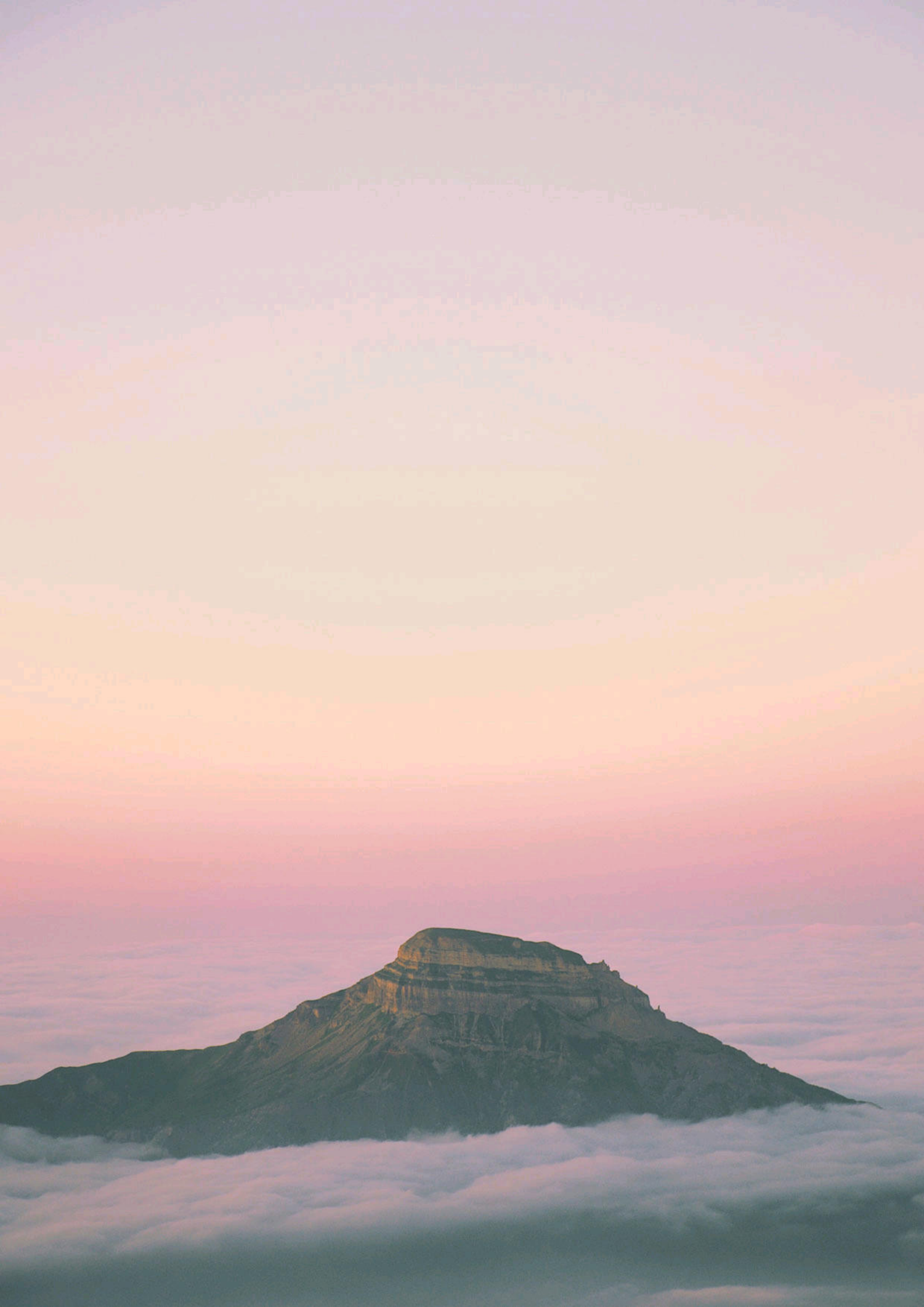


图

一、定义

1. 图中数据元素称为顶点。

2. 无空图

3. 顶点之间的逻辑关系用边表示

   边集可以为空。

4. 无向边：顶点 V_i 到 V_j 的边没有方向，用 (V_i, V_j) 表示（无序偶对）

   有向边：从顶点 V_i 到 V_j 的边有方向，则称这条边为有向边，也称为弧（A指向D：$<A, D>$）

△ 无向边用（ ）　有向边用 < >

5. 无向/有向完全图：任意2个顶点之间存在边。

6. 边或弧很少的图称为稀疏图，反之称为稠密图

7. 图或边相关的数叫权

   带权的图称为网

8. 子图：若图 $G=(V, \{E\})$ 和图 $G'(V', \{E'\})$ 满足 $V' \subseteq V$ 且 $E' \in E$

   则称 G' 为 G 的子图。

9. (V, V')：V 与 V' 相邻接

   $<V, V'>$：V 邻接到 V'

   V' 邻接自顶点 V

   入度 ID　出度 OD　度 TD = ID + OD

10. 路径：

    路径长度是路径上的边或弧的数目。

    序列中顶点不重复出现的路径称为简单路径。

11. 回路或环：第一个顶点和最后一个结点相同的路径

    顶点不重复出现的为简单环或简单回路

12. 连通：v到v'有路径。

<1> 若图中任意2个点均连通，则G为连通图 > 无向图

无向图中的极大连通子图称为连通分量

{ 子图连通
  含有极大顶点数
  包含依附于这些顶点的所有边。

<2> 强连通图 > 有向图
强连通分量

13. 无向图中连通n个顶点n-1条边叫生成树。

有向图中一个顶点入度为0其余顶点入度为1的有向树。

一个有向图由若干棵有向树构成森林

二、抽象数据类型

ADT 图(Graph)

Data

顶点的有穷非空集合和边的集合。

Operation

CreateGraph(*G,V,VR)：按照顶点集V和边弧集VR的定义构造图G。

DestroyGraph(*G)：图G存在则销毁。

LocateVex(G,u)：若图G中存在顶点u，则返回图中的位置。

GetVex(G,v)：返回图G中顶点v的值。

PutVex(G,v,value)：将图G中顶点v赋值value。

FirstAdjVex(G,*v)：返回顶点v的一个邻接顶点，若顶点在G中无邻接顶点返回空。

NextAdjVex(G,v,*w)：返回顶点v相对于顶点w的下一个邻接顶点，若w是v的最后一个邻接点则返回“空”。

InsertVex(*G,v)：在图G中增添新顶点v。

DeleteVex(*G,v)：删除图G中顶点v及其相关的弧。

InsertArc(*G,v,w)：在图G中增添弧<v,w>，若G是无向图，还需要增添对称弧<w,v>。

DeleteArc(*G,v,w)：在图G中删除弧<v,w>，若G是无向图，则还删除对称弧<w,v>。

DFSTraverse(G)：对图G中进行深度优先遍历，在遍历过程中对每个顶点调用。

HFSTraverse(G)：对图G中进行广度优先遍历，在遍历过程中对每个顶点调用。

endADT

三、存储结构

1) 邻接矩阵

用2个数组表示：
- 一个一维数组 存储图中顶点信息
- 一个二维数组 存储图中边或弧信息 (邻接矩阵)

及物为度
行为为出度.

$$arc[i][j] = \begin{cases} 1, & \text{若 } (v_i, v_j) \in E \text{ 或 } \langle v_i, v_j \rangle \in E \\ 0, & \text{其他.} \end{cases}$$

带权网图：
- W_{ij}，若 $(v_i, v_j) \in E$ 或 $\langle v_i, v_j \rangle \in E$ (权值)
- 0，若 $i=j$
- ∞，其他

创建邻接矩阵

```
typedef char VertexType;                /* 顶点类型应由用户定义 */
typedef int EdgeType;                   /* 边上的权值类型应由用户定义 */
#define MAXVEX 100                      /* 最大顶点数，应由用户定义 */
#define INFINITY 65535                  /* 用65535来代表∞ */
typedef struct
{
    VertexType vexs[MAXVEX];            /* 顶点表 */
    EdgeType arc[MAXVEX][MAXVEX];       /* 邻接矩阵，可看作边表 */
    int numNodes, numEdges;             /* 图中当前的顶点数和边数 */
}MGraph;
```

创建 无向网图

```
/* 建立无向网图的邻接矩阵表示 */
void CreateMGraph(MGraph *G)
{
    int i,j,k,w;
    printf("输入顶点数和边数:\n");
    scanf("%d,%d",&G->numNodes,&G->numEdges);          /* 输入顶点数和边数 */
    for(i = 0;i <G->numNodes;i++)                      /* 读入顶点信息,建立顶点表 */
        scanf(&G->vexs[i]);                            — n.
    for(i = 0;i <G->numNodes;i++)
        for(j = 0;j <G->numNodes;j++)                  — n²
            G->arc[i][j]=INFINITY;                     /* 邻接矩阵初始化 */
    for(k = 0;k <G->numEdges;k++)                      /* 读入numEdges条边，建立邻接矩阵 */
    {
        printf("输入边(vi,vj)上的下标i，下标j和权w:\n");
        scanf("%d,%d,%d",&i,&j,&w);                    /* 输入边(vi,vj)上的权w */
        G->arc[i][j]=w;                                — e.
        G->arc[j][i]= G->arc[i][j];                    /* 因为是无向图，矩阵对称 */
    }
}
```

n个顶点，e边条 — 时间复杂度 $O(n+n^2+e)$

2.邻接表

数组和链表相结合的存储方法称为邻接表.

△逆邻接表

```
typedef char VertexType;          /* 顶点类型应由用户定义 */
typedef int EdgeType;             /* 边上的权值类型应由用户定义 */

typedef struct EdgeNode           /* 边表结点 */
{
    int adjvex;                   /* 邻接点域,存储该顶点对应的下标 */
    EdgeType info;                /* 用于存储权值,对于非网图可以不需要 */
    struct EdgeNode *next;        /* 链域,指向下一个邻接点 */
}EdgeNode;

typedef struct VertexNode         /* 顶点表结点 */
{
    VertexType data;              /* 顶点域,存储顶点信息 */
    EdgeNode *firstedge;          /* 边表头指针 */
}VertexNode, AdjList[MAXVEX];

typedef struct
{
    AdjList adjList;
    int numNodes,numEdges;        /* 图中当前顶点数和边数 */
}GraphAdjList;
```

创造无向图邻接表

```
/* 建立图的邻接表结构 */
void CreateALGraph(GraphAdjList *G)
{
  int i,j,k;
  EdgeNode *e;
  printf("输入顶点数和边数:\n");
  scanf("%d,%d",&G->numNodes,&G->numEdges);   /* 输入顶点数和边数 */
  for(i = 0;i < G->numNodes;i++)              /* 读入顶点信息,建立顶点表 */
  {
      scanf(&G->adjList[i].data);             /* 输入顶点信息 */
      G->adjList[i].firstedge=NULL;           /* 将边表置为空表 */
  }


  for(k = 0;k < G->numEdges;k++)              /* 建立边表 */
  {
      printf("输入边(vi,vj)上的顶点序号:\n");
      scanf("%d,%d",&i,&j);                   /* 输入边(vi,vj)上的顶点序号 */
      e=(EdgeNode *)malloc(sizeof(EdgeNode)); /* 向内存申请空间,生成边表结点 */
      e->adjvex=j;                            /* 邻接序号为j */
      e->next=G->adjList[i].firstedge;        /* 将e的指针指向当前顶点指向的结点 */
      G->adjList[i].firstedge=e;              /* 将当前顶点的指针指向e */
      e=(EdgeNode *)malloc(sizeof(EdgeNode)); /* 向内存申请空间,生成边表结点 */
      e->adjvex=i;                            /* 邻接序号为i */
      e->next=G->adjList[j].firstedge;        /* 将e的指针指向当前顶点指向的结点 */
      G->adjList[j].firstedge=e;              /* 将当前顶点的指针指向e */
  }
}
```

O(n+e)

(3) 十字链表

把邻接表和逆邻接表结合起来

data	firstin	firstout

顶点表结点

tailvex	headvex	headlink	taillink

边表结点

tailvex：弧起点在顶点表中的下标

headvex：弧终点在顶点表中的下标

headlink：入边表指针域，指向终点相同的下一条边

taillink：指向起点相同的下一条边

(4) 邻接多重表

顶点表不变，边表结点结构

ivex	ilink	jvex	jlink

(5) 边集数组

一个一维数组 { 一个存储顶点信息
另一个存储边信息 —— 每个数据元素由起点下标、终点下标、权

begin	end	weight

四. 图的遍历

1. 深度优先遍历 DFS

从图中某个顶点v出发，访问此顶点，然后从v的未被访问的邻接点出发深度优先遍历图。

邻接矩阵的方式

```
#define MAXVEX 9
Boolean visited[MAXVEX];                    /* 访问标志的数组 */

/* 邻接矩阵的深度优先递归算法 */
void DFS(MGraph G, int i)
{
    int j;
    visited[i] = TRUE;
    printf("%c ", G.vexs[i]);               /* 打印顶点，也可以做其他操作 */
    for(j = 0; j < G.numVertexes; j++)
        if(G.arc[i][j] == 1 && !visited[j])
            DFS(G, j);                      /* 对未访问的邻接顶点递归调用 */
}

/* 邻接矩阵的深度遍历操作 */
void DFSTraverse(MGraph G)
{
    int i;
    for(i = 0; i < G.numVertexes; i++)
        visited[i] = FALSE;                 /* 初始所有顶点状态都是未访问过状态 */
    for(i = 0; i < G.numVertexes; i++)
        if(!visited[i])                     /* 对未访问过的顶点调用DFS，若为连通图仅执行
            DFS(G, i);
}
```

邻接表的方式

```
/* 邻接表的深度优先递归算法 */
void DFS(GraphAdjList GL, int i)
{
    EdgeNode *p;
    visited[i] = TRUE;
    printf("%c ",GL->adjList[i].data);  /* 打印顶点，也可以做其他操作 */
    p = GL->adjList[i].firstedge;
    while(p)
    {
        if(!visited[p->adjvex])
            DFS(GL, p->adjvex);
        p = p->next;                    /* 对未访问的邻接顶点递归调用 */
    }
}

/* 邻接表的深度遍历操作 */
void DFSTraverse(GraphAdjList GL)
{
    int i;
    for(i = 0; i < GL->numVertexes; i++)
        visited[i] = FALSE;             /* 初始所有顶点状态都是未访问过状态 */
    for(i = 0; i < GL->numVertexes; i++)
        if(!visited[i])                 /* 对未访问过的顶点调用DFS，若是连通图，只会执行一次 */
            DFS(GL, i);
```

邻接矩阵 $O(n^2)$

邻接表 $O(n+e)$ ——> 适用于点多边少的稀疏图

2. 广度优先遍历 BFS

邻接矩阵

```
/* 邻接矩阵的广度遍历算法 */
void BFSTraverse(MGraph G)
{
    int i, j;
    Queue Q;
    for(i = 0; i < G.numVertexes; i++)
        visited[i] = FALSE;
    InitQueue(&Q);                              /* 初始化一辅助用的队列 */
    for(i = 0; i < G.numVertexes; i++)          /* 对每一个顶点做循环 */
    {
        if (!visited[i])                        /* 若是未访问过就处理 */
        {
            visited[i]=TRUE;
            printf("%c ", G.vexs[i]);           /* 设置当前顶点访问过 */
            EnQueue(&Q,i);                      /* 打印顶点，也可以做其他操作 */
            while(!QueueEmpty(Q))               /* 将此顶点入队列 */
            {                                   /* 若当前队列不为空 */
                DeQueue(&Q,&i);
                for(j=0;j<G.numVertexes;j++)    /* 将队首元素出队列，赋值给i */
                {
                                                /* 判断其他顶点，若与当前顶点存在 */
                    if(G.arc[i][j] == 1 && !visited[j])  /* 边且未访问过 */
                    {
                        visited[j]=TRUE;        /* 将找到的此顶点标记为已访问 */
                        printf("%c ", G.vexs[j]);  /* 打印顶点 */
                        EnQueue(&Q,j);          /* 将找到的此顶点入队列 */
                    }
                }
            }
        }
    }
}
```

邻接表

```
/* 邻接表的广度遍历算法 */
void BFSTraverse(GraphAdjList GL)
{
    int i;
    EdgeNode *p;
    Queue Q;
    for(i = 0; i < GL->numVertexes; i++)
        visited[i] = FALSE;
    InitQueue(&Q);
    for(i = 0; i < GL->numVertexes; i++)
    {
        if (!visited[i])
        {
            visited[i]=TRUE;
            printf("%c ",GL->adjList[i].data);  /* 打印顶点,也可以做其他操作 */
            EnQueue(&Q,i);
            while(!QueueEmpty(Q))
            {
                DeQueue(&Q,&i);
                p = GL->adjList[i].firstedge;   /* 找到当前顶点的边表链的表头指针 */
                while(p)
                {
                    if(!visited[p->adjvex])     /* 若此顶点未被访问 */
                    {
                        visited[p->adjvex]=TRUE;
                        printf("%c ",GL->adjList[p->adjvex].data);
                        EnQueue(&Q,p->adjvex);  /* 将此顶点入队列 */
                    }
                    p = p->next;                /* 指针指向下一个邻接点 */
                }
            }
        }
    }
}
```

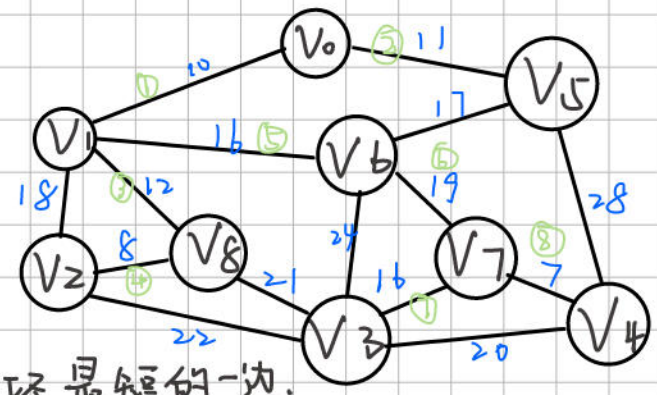
时间复杂度与广度相同

五、最小生成树

构造连通网的最小代价生成树

1、普里姆算法

例：



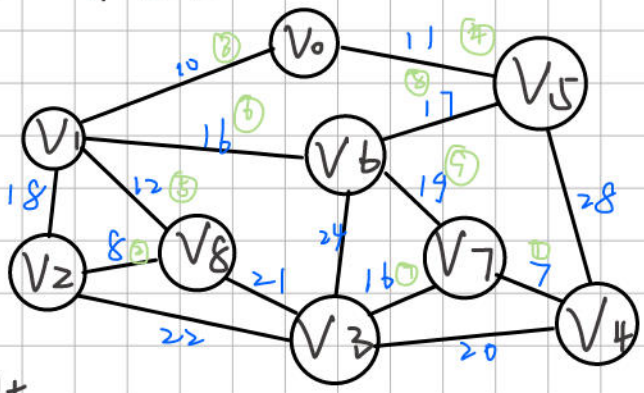
依次选择最短的一边

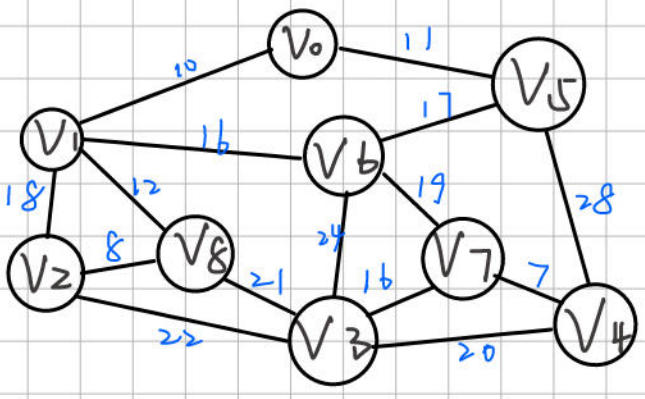
从哪个顶点开始都一样

```
/* Prim算法生成最小生成树 */
void MiniSpanTree_Prim(MGraph G)
{
    int min, i, j, k;
    int adjvex[MAXVEX];
    int lowcost[MAXVEX];
    lowcost[0] = 0;                 /* 保存相关顶点间边的权值点下标 */
    adjvex[0] = 0;                  /* 保存相关顶点间边的权值 */
    for(i = 1; i < G.numVertexes; i++)  /* 初始化第一个权值为0，即v0加入生成树。*/
    {                               /* 初始化第一个顶点下标为0 */
        lowcost[i] = G.arc[0][i];   /* 循环除下标为0外的全部顶点 */
        adjvex[i] = 0;              /* 将v0顶点与之有边的权值存入数组 */
    }                               /* 初始化都为v0的下标 */
    for(i = 1; i < G.numVertexes; i++)
    {
        min = INFINITY;             /* 初始化最小权值为∞，可以是较大数字如65535等 */
        j = 1;k = 0;
        while(j < G.numVertexes)    /* 循环全部顶点 */
        {
            if(lowcost[j]!=0 && lowcost[j] < min)
            {                       /* 如果权值不为0且权值小于min */
                min = lowcost[j];   /* 则让当前权值成为最小值 */
                k = j;              /* 将当前最小值的下标存入k */
            }
            j++;
        }
        printf("(%d, %d)\n", adjvex[k], k); /* 打印当前顶点边中权值最小的边 */
        lowcost[k] = 0;                     /* 将当前顶点权值设置为0,此顶点已完成任务 */
        for(j = 1; j < G.numVertexes; j++)  /* 循环所有顶点 */
        {   /* 如果下标为k的顶点的各边权值小于此前这些顶点未被加入生成树的权值 */
            if(lowcost[j]!=0 && G.arc[k][j] < lowcost[j])
            {                               /* 将较小的权值存入lowcost相应位置 */
                lowcost[j] = G.arc[k][j];   /* 将下标为k的顶点存入adjvex */
                adjvex[j] = k;
            }
        }
    }
}
```

2. 克鲁斯卡尔算法

利用边集数组，并将边集按权的大小排序





算法：

```c
/* Kruskal算法生成最小生成树 */
void MiniSpanTree_Kruskal(MGraph G)
{
    int i, n, m;
    Edge edges[MAXEDGE];/* 定义边集数组,edge的结构为begin,end,weight,均为整型 */
    int parent[MAXVEX]; /* 定义一数组用来判断边与边是否形成环路 */

    /* 此处省略将邻接矩阵G转化为边集数组edges并按权由小到大排序的代码 */

    for (i = 0; i < G.numVertexes; i++)
        parent[i] = 0;                          /* 初始化数组值为0 */
    for (i = 0; i < G.numEdges; i++)            /* 循环每一条边 */
    {
        n = Find(parent,edges[i].begin);
        m = Find(parent,edges[i].end);
        if (n != m) /* 假如n与m不等，说明此边没有与现有的生成树形成环路 */
        {/* 将此边的结尾顶点放入下标为起点的parent中。表示此顶点已经在生成树集合中 */
            parent[n] = m;
            printf("(%d, %d) %d\n", edges[i].begin,
                edges[i].end, edges[i].weight);
        }
    }
}

/* 查找连线顶点的尾部下标 */
int Find(int *parent, int f)
{
    while ( parent[f] > 0)
    {
        f = parent[f];
    }
    return f;
}
```

Dleloge)

稀疏图：克鲁斯卡尔优势大

稠密图：用普里姆算法

六、最短路径

网图：顶点——终点

1、迪杰斯特拉算法

数据结构

```
#define MAXVEX 20
#define INFINITY 65535

typedef struct
{
    int vexs[MAXVEX];
    int arc[MAXVEX][MAXVEX];
    int numVertexes, numEdges;
}MGraph;

typedef int Patharc[MAXVEX];           /* 用于存储最短路径下标的数组 */
typedef int ShortPathTable[MAXVEX]; /* 用于存储到各点最短路径的权值和 */
```

算法

```
/* Dijkstra算法，求有向网G的v0顶点到其余顶点v的最短路径P[v]及带权长度D[v] */
/* P[v]的值为前驱顶点下标,D[v]表示v0到v的最短路径长度和 */
void ShortestPath_Dijkstra(MGraph G, int v0, Patharc *P, ShortPathTable *D)
{
    int v,w,k,min;
    int final[MAXVEX];                  /* final[w]=1表示求得顶点v0至vw的最短路径 */
    for(v=0; v<G.numVertexes; v++)      /* 初始化数据 */
    {
        final[v] = 0;                   /* 全部顶点初始化为未知最短路径状态 */
        (*D)[v] = G.arc[v0][v];         /* 将与v0点有连线的顶点加上权值 */
        (*P)[v] = -1;                   /* 初始化路径数组P为-1 */
    }
    (*D)[v0] = 0;                       /* v0至v0路径为0 */
    final[v0] = 1;                      /* v0至v0不需要求路径 */
    /* 开始主循环，每次求得v0到某个顶点v的最短路径 */
    for(v=1; v<G.numVertexes; v++)
    {
        min=INFINITY;                   /* 当前所知离v0顶点的最近距离 */
        for(w=0; w<G.numVertexes; w++)  /* 寻找离v0最近的顶点 */
        {
            if(!final[w] && (*D)[w]<min)
            {
                k=w;
                min = (*D)[w];          /* w顶点离v0顶点更近 */
            }
        }
        final[k] = 1;                   /* 将目前找到的最近的顶点置为1 */
        for(w=0; w<G.numVertexes; w++)  /* 修正当前最短路径及距离 */
        {
            /* 如果经过v顶点的路径比现在这条路径的长度短的话 */
            if(!final[w] && (min+G.arc[k][w]<(*D)[w]))
            {                           /* 说明找到了更短的路径，修改D[w]和P[w] */
                (*D)[w] = min + G.arc[k][w]; /* 修改当前路径长度 */
                (*P)[w]=k;
            }
        }
    }
}
```

时间复杂度：$O(n^3)$

2. 弗洛伊德算法

$$D^{0}[v][w] = \min\{D^{-1}[v][w], D^{-1}[v][0] + D^{-1}[0][w]\}$$

```
typedef int Patharc[MAXVEX][MAXVEX];
typedef int ShortPathTable[MAXVEX][MAXVEX];
/* Floyd算法，求网图G中各顶点v到其余顶点w的最短路径P[v][w]及带权长度D[v][w] */
void ShortestPath_Floyd(MGraph G, Patharc *P, ShortPathTable *D)
{
    int v,w,k;
    for(v=0; v<G.numVertexes; ++v)        /* 初始化D与P */
    {
        for(w=0; w<G.numVertexes; ++w)
        {
            (*D)[v][w]=G.arc[v][w];       /* D[v][w]值即为对应点间的权值 */
            (*P)[v][w]=w;                 /* 初始化P */
        }
    }
    for(k=0; k<G.numVertexes; ++k)
    {
        for(v=0; v<G.numVertexes; ++v)
        {
            for(w=0; w<G.numVertexes; ++w)
            {
                if ((*D)[v][w]>(*D)[v][k]+(*D)[k][w])
                {/* 如果经过下标为k顶点的路径比原两点间路径更短 */
                    (*D)[v][w]=(*D)[v][k]+(*D)[k][w];/* 将当前两点间权值设更小一个 */
                    (*P)[v][w]=(*P)[v][k];           /* 路径设置为经过下标为k的顶点 */
                }
            }
        }
    }
}
```

也可以这样：

```
printf("各顶点间最短路径如下:\n");
for(v=0; v<G.numVertexes; ++v)
{
    for(w=v+1; w<G.numVertexes; w++)
    {
        printf("v%d-v%d weight: %d ",v,w,D[v][w]);
        k=P[v][w];                  /* 获得第一个路径顶点下标 */
        printf(" path: %d",v);      /* 打印源点 */
        while(k!=w)                 /* 如果路径顶点下标不是终点 */
        {
            printf(" -> %d",k);     /* 打印路径顶点 */
            k=P[k][w];              /* 获得下一个路径顶点下标 */
        }
        printf(" -> %d\n",w);       /* 打印终点 */
    }
    printf("\n");
}
```

七. 拓扑排序

1. 定义

在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系这样的有向图为顶点表示活动的网。称为 AOV 网

设 G(V, E) 是一个具有 n 个顶点的有向图，V中顶点序列 $v_1, v_2 \cdots v_n$ 若从顶点 v_i 到 v_j 有一条路径，在顶点序列中 v_i 必在顶点 v_j 之前。则此顶点序列为拓扑序列

拓扑排序 算法

```c
typedef struct EdgeNode          /* 边表结点 */
{
    int adjvex;                  /* 邻接点域，存储该顶点对应的下标 */
    int weight;                  /* 用于存储权值，对于非网图可以不需要 */
    struct EdgeNode *next;       /* 链域，指向下一个邻接点 */
}EdgeNode;

typedef struct VertexNode        /* 顶点表结点 */
{
    int in;                      /* 顶点入度 */
    int data;                    /* 顶点域，存储顶点信息 */
    EdgeNode *firstedge;         /* 边表头指针 */
}VertexNode, AdjList[MAXVEX];

typedef struct
{
    AdjList adjList;
    int numVertexes,numEdges;    /* 图中当前顶点数和边数 */
}graphAdjList,*GraphAdjList;
```

找

```c
/* 拓扑排序，若GL无回路，则输出拓扑排序序列并返回1，若有回路返回0 */
Status TopologicalSort(GraphAdjList GL)
{
    EdgeNode *e;
    int i,k,gettop;
    int top=0;                                  /* 用于栈指针下标 */
    int count=0;                                /* 用于统计输出顶点的个数 */
    int *stack;                                 /* 建栈将入度为0的顶点入栈 */
    stack=(int *)malloc(GL->numVertexes * sizeof(int) );
    for(i = 0; i<GL->numVertexes; i++)
        if(0 == GL->adjList[i].in)              /* 将入度为0的顶点入栈 */
            stack[++top]=i;
    while(top!=0)
    {
        gettop=stack[top--];                    /* 出栈 */
        printf("%d -> ",GL->adjList[gettop].data);  /* 打印此顶点 */
        count++;                                /* 统计输出顶点数 */
        for(e = GL->adjList[gettop].firstedge; e; e = e->next)/* 对此顶点弧表遍历 *
        {
            k=e->adjvex;
            if( !(--GL->adjList[k].in) )        /* 将k号顶点邻接点的入度减1*/
                stack[++top]=k;                 /* 若为0则入栈，以便下次循环输出
        }
    }
    if(count < GL->numVertexes)                 /* count小于顶点数，说明存在环 *
        return ERROR;
    else
        return OK;
}
```

八. 关键路径

1. 定义

在一个表示工程的带权有向图中，用顶点表示事件，用有向边表示活动，用边上权值表示活动的持续时间，这种有向图的边表示活动的网，AOE网。

路径上各个活动所持续的时间之和为路径长度，

从源点到汇点具有最大长度的路径叫关键路径，

在关键路径上的活动叫关键活动。

2. 算法

改进拓扑排序算法

```
int *etv, *ltv;
int *stack2;
int top2;
```

一些局变量

$$etv[k]=\begin{cases}0, & k=0\text{ 时}\\ \max\{etv[i]+len\langle v_i,v_k\rangle\}, & k\neq 0\text{ 时}\end{cases}$$

其中 $\langle v_i,v_k\rangle\in P[k]$ 时

```
1  /* 拓扑排序 */
2  Status TopologicalSort(GraphAdjList GL)
3  {   /* 若GL无回路，则输出拓扑排序序列并返回1，若有回路返回0 */
4      EdgeNode *e;
5      int i,k,gettop;
6      int top=0;                                  /* 用于栈指针下标 */
7      int count=0;                                /* 用于统计输出顶点的个数 */
8      int *stack;                                 /* 建栈将入度为0的顶点入栈 */
9      stack=(int *)malloc(GL->numVertexes * sizeof(int) );
10     for(i = 0; i<GL->numVertexes; i++)
11         if(0 == GL->adjList[i].in)              /* 将入度为0的顶点入栈 */
12             stack[++top]=i;
13     top2=0;                                     /* 初始化 */
14     etv=(int *)malloc(GL->numVertexes * sizeof(int) );  /* 事件最早发生时间数组 */
15     for(i=0; i<GL->numVertexes; i++)
16         etv[i]=0;                               /* 初始化 */
17     stack2=(int *)malloc(GL->numVertexes * sizeof(int) );/* 初始化拓扑序列栈 */
18     while(top!=0)
19     {
20         gettop=stack[top--];
21         count++;                                /* 输出i号顶点，并计数 */
22         stack2[++top2]=gettop;                  /* 将弹出的顶点序号压入拓扑序列的栈 */
23         for(e = GL->adjList[gettop].firstedge; e; e = e->next)
24         {
25             k=e->adjvex;
26             if( !(--GL->adjList[k].in))
27                 stack[++top]=k;
28             if((etv[gettop] + e->weight) > etv[k]) /* 求各顶点事件的最早发生时间etv的值 */
29                 etv[k] = etv[gettop] + e->weight;
30         }
31     }
32     if(count < GL->numVertexes)
33         return ERROR;
34     else
35         return OK;
36 }
```

关键路径算法

```
/* 求关键路径,GL为有向网, 输出G的各项关键活动 */
void CriticalPath(GraphAdjList GL)
{
    EdgeNode *e;
    int i,gettop,k,j;
    int ete,lte;                    /* 声明活动最早发生时间和最迟发生时间变量 */
    TopologicalSort(GL);            /* 求拓扑序列, 计算数组etv和stack2的值 */
    ltv=(int *)malloc(GL->numVertexes*sizeof(int));/* 事件最早发生时间数组 */
    for(i=0; i<GL->numVertexes; i++)
        ltv[i]=etv[GL->numVertexes-1];              /* 初始化ltv */
    while(top2!=0)                                  /* 计算ltv */
    {
        gettop=stack2[top2--];
        for(e = GL->adjList[gettop].firstedge; e; e = e->next)
        {
            k=e->adjvex;
            if(ltv[k] - e->weight < ltv[gettop])   /* 求各顶点事件最晚发生时间ltv */
                ltv[gettop] = ltv[k] - e->weight;
        }
    }
    for(j=0; j<GL->numVertexes; j++)                /* 求ete,lte和关键活动 */
    {
        for(e = GL->adjList[j].firstedge; e; e = e->next)
        {
            k=e->adjvex;
            ete = etv[j];                           /* 活动最早发生时间 */
            lte = ltv[k] - e->weight;               /* 活动最迟发生时间 */
            if(ete == lte)                          /* 两者相等即在关键路径上 */
                printf("<v%d - v%d> length: %d \n",
                    GL->adjList[j].data,GL->adjList[k].data,e->weight);

        }
    }
```

$$ltv[k]=\begin{cases} etv[k] & k=n-1\text{时;} \\ \min\{ltv[j]-len\langle v_k,v_j\rangle\} & k<n-1\text{且}\langle v_k,v_j\rangle\in S[k]\text{时} \end{cases}$$