**五、图**

**图的基本概念**

一个图(G)定义为一个偶对(V,E) ，记为G=(V,E) 。其中： V是顶点(Vertex)的非空有限集合，记为V(G)；E是无序集V&V的一个子集，记为E(G) ，其元素是图的弧(Arc)。

**1.图**

图由结点的有穷集合V和边的集合E组成，为了与树形结构进行区别，在图结构中常常将结点称为顶点，边是顶点的有序偶对。若两个顶点之间存在一条边，则表示这两个顶点具有相邻关系。

**2.有向图和无向图**

有向图(Digraph)： 若图G的关系集合E(G)中，顶点偶对<v,w>的v和w之间是有序的，称图G是有向图。

无向图(Undigraph)： 若图G的关系集合E(G)中，顶点偶对<v,w>的v和w之间是无序的，称图G是无向图。

**3.弧**

在有向图中，若 <v,w>(G) ，表示从顶点v到顶点w有一条弧。 其中：v称为弧尾(tail)或始点(initial node)，w称为弧头(head)或终点(terminal node) 。

**4.顶点的度，入度和出度**

无向图中，顶点 v 的度是指和 v 相关联的边的数目  
有向图中，以顶点 v 为弧头的弧的数目称为顶点 v 的入度，以顶点 v 为弧尾的弧的数目称为顶点 v 的出度

**5.有向完全图和无向完全图**

有向图中每两个顶点之间都有两条方向相反的边连接的图称为有向完全图。弧数为 n(n -1)（结点为 n）  
无向图中每一对不同顶点恰有一条边相连的图称为无向完全图。边数为n(n−1)2（结点数为 n）

**6.路径和路径长度**

从顶点 v 经过一系列的边或弧到达顶点 w ，则称这一系列的边或弧为顶点v 到顶点 w 的路径。路径长度是指路径上边的数目。

**7.简单路径**

序列中顶点不重复出现的路径称为简单路径

**8.回路**

若一条路径中第一个顶点和最后一个顶点相同，则这条路径是一条回路

**9.连通,连通图和连通分量**

在无向图中，如果从顶点 v 到顶点 w 有路径，则称顶点 v 和 顶点 w 是连通的。如果对于图中的任意两个顶点都是连同的，则称为连通图。无向图中的极大连通子图为其连通分量。

**10.强连通图和强连通分量**

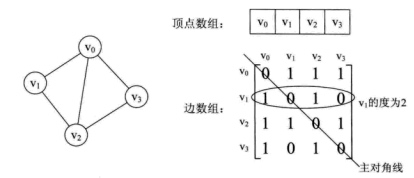
在有向图中，如果对每一对顶点 v 、w 从v 和 从w到v都有路径，则称该有向图是强连通图。有向图中的极大强连通子图称为有向图的强连通分量。

**11.权和网**

图中每条边都可以附带一个数，这种与边相关的数称为权，权可以表示从一个顶点到另一个顶点的距离或者花费的代价。边上带权的图称为带权图。

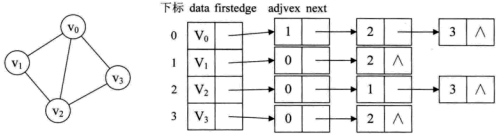
**图的储存结构**

**邻接矩阵**

邻接矩阵是表示顶点之间相邻关系的矩阵，存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组（称为邻接矩阵）存储图中的边或弧的信息。  
[](https://cloud.8hfq.com/img/15330216139508.jpg)  
从上面可以看出，无向图的边数组是一个对称矩阵。所谓对称矩阵就是n阶矩阵的元满足aij = aji。即从矩阵的左上角到右下角的主对角线为轴，右上角的元和左下角相对应的元全都是相等的。  
矩阵中“1”的个数为图中总边数的两倍，矩阵图中第i行和第i列元素之和即为顶点i的度  
对于有向图，矩阵中“1”的个数即为图的边数，矩阵中第i行的元素之和即为顶点i的出度，第j列的元素之和即为顶点j的入度

有权有向图中，无边的话0变成无限大，1变成权值

**邻接表**

邻接矩阵是不错的一种图存储结构，但是，对于边数相对顶点较少的图，这种结构存在对存储空间的极大浪费。因此，找到一种数组与链表相结合的存储方法称为邻接表。  
邻接表的处理方法是这样的：  
（1）图中顶点用一个一维数组存储，当然，顶点也可以用单链表来存储，不过，数组可以较容易的读取顶点的信息，更加方便。  
（2）图中每个顶点vi的所有邻接点构成一个线性表，由于邻接点的个数不定，所以，用单链表存储，无向图称为顶点vi的边表，有向图则称为顶点vi作为弧尾的出边表。  
例如，下图就是一个无向图的邻接表的结构。  
[](https://cloud.8hfq.com/img/15330228248146.jpg)

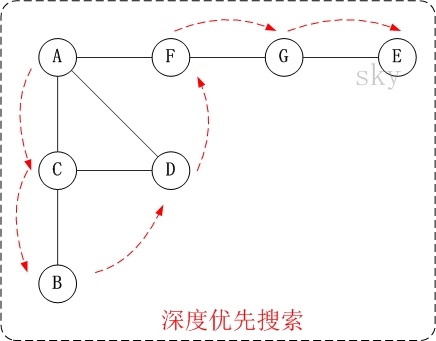
**图的遍历**

**深度优先搜索遍历**

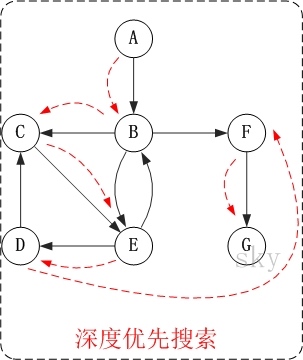
图的深度优先搜索遍历（DFS）类似于二叉树的先序遍历。  
基本思路：假设初始状态是图中所有顶点均未被访问，则从某个顶点v出发，首先访问该顶点，然后依次从它的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和v有路径相通的顶点都被访问到。 若此时尚有其他顶点未被访问到，则另选一个未被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | int visit[maxSize]; /\* V是起点编号，visit[]是一个全局数组，作为顶点的访问标记，初始时所有元素均为0，  \* 表示所有顶点都未被访问，因为图中存在回路，当前经过的点在将来还有可能再次经过，  \* 所以要对每个顶点进行标记，以免重复访问，\*/ void DFS(AGraph \*G,int v){  ArcNode \*p;  visit[v]=1;  Visit(v);  p = G->adjlist[v].firstarc;//p指向顶点v的第一条边  while(p!=NULL){  if(visit[p->adjvex]==0)  DFS(G,p->adjvex);  p=p->nextarc;  } } |

**无向图的深度优先搜索**

下面以”无向图”为例，来对深度优先搜索进行演示。  
对上面的图进行深度优先遍历，从顶点A开始。  
[](https://cloud.8hfq.com/img/15330233458653.jpg)  
因此访问顺序是：A -> C -> B -> D -> F -> G -> E

**有向图的深度优先搜索**

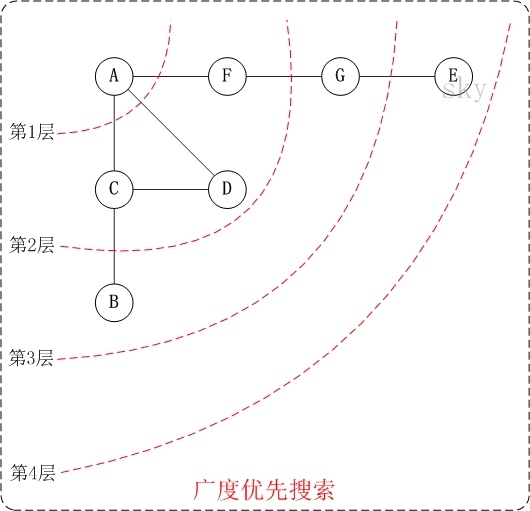
下面以”有向图”为例，来对深度优先搜索进行演示。  
对上面的图进行深度优先遍历，从顶点A开始。  
[](https://cloud.8hfq.com/img/15330235140601.jpg)  
因此访问顺序是：A -> B -> C -> E -> D -> F -> G

**广度优先搜索遍历**

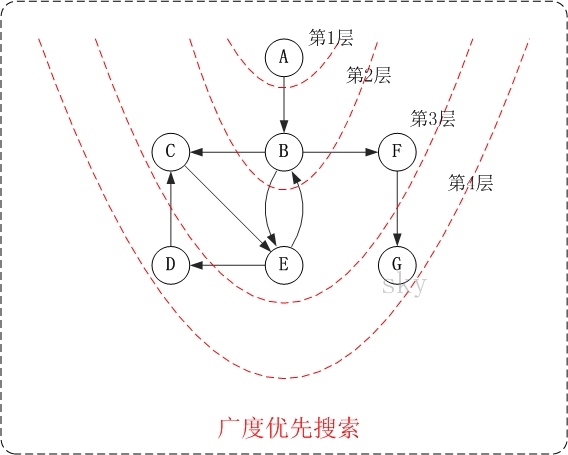
广度优先搜索算法（BFS）类似于树的层次遍历  
基本思想：从图中某顶点v出发，在访问了v之后依次访问v的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使得“先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问，直至图中所有已被访问的顶点的邻接点都被访问到。如果此时图中尚有顶点未被访问，则需要另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。  
换句话说，广度优先搜索遍历图的过程是以v为起点，由近至远，依次访问和v有路径相通且路径长度为1,2…的顶点。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | void BFS(AGraph \*G, int v ,int visit[maxSize]) {  ArcNode \* p;  int que[maxSize],front=0,rear=0;  itn j;  Visit(v);  visit[v]=1;  rear=(rear+1)%maxSize;  que[rear]=v;  while(front!=rear)  {  front=(front+1)%maxSize;  j=que[front];  p = G->adjlist[j].firstarc;  while (p!=NULL){  if (visit[p->adjvex]==0)  {  Visit(p->adjvex);  visit[p->adjvex]=1;  rear = (rear+1)%maxSize;  que[rear] = p->adjvex;  }  p = p->nextarc;  }  } } |

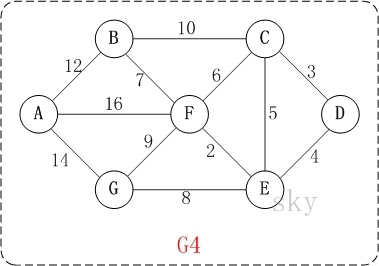
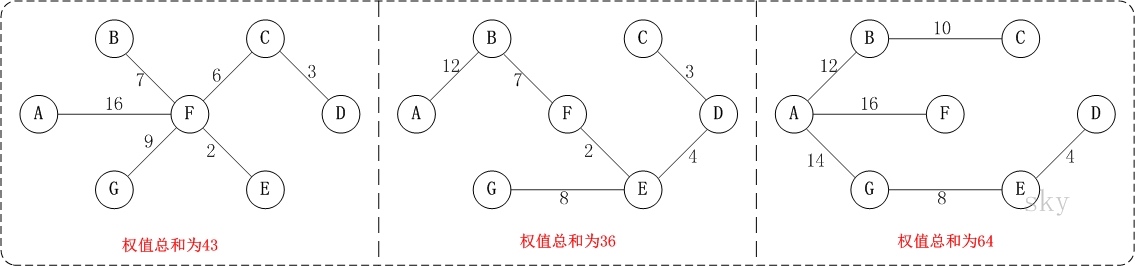
**无向图的广度优先搜索**

[](https://cloud.8hfq.com/img/15330237300952.jpg)  
因此访问顺序是：A -> C -> D -> F -> B -> G -> E

**有向图的广度优先搜索**

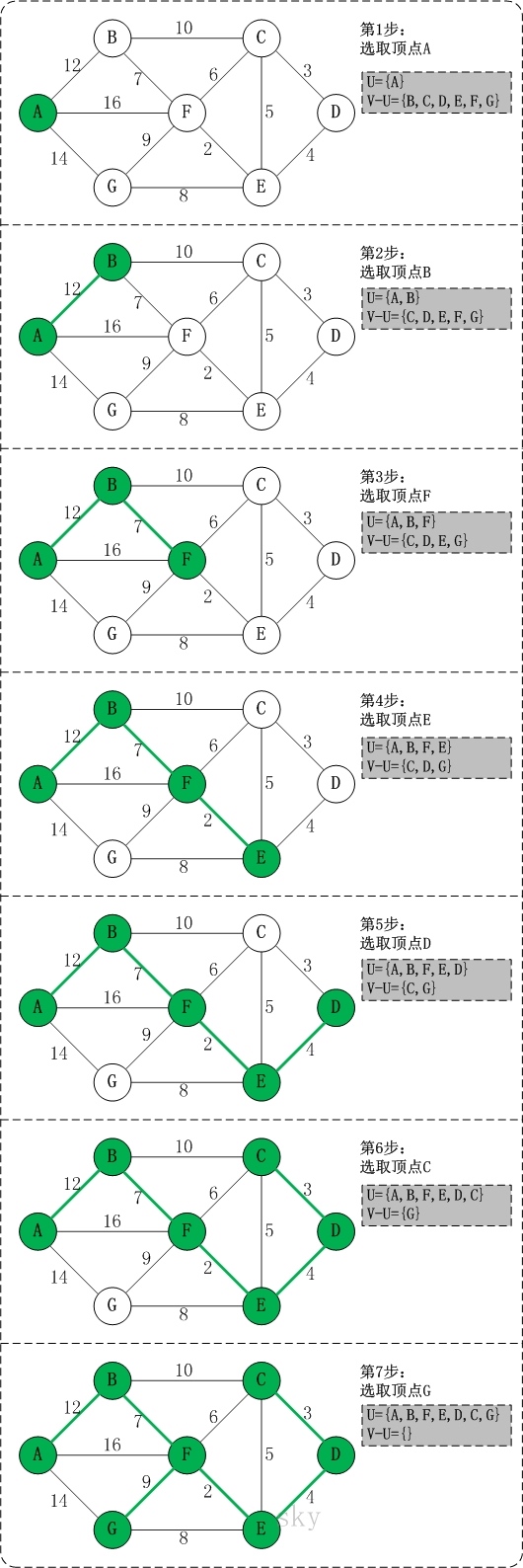
[](https://cloud.8hfq.com/img/15330237892270.jpg)  
因此访问顺序是：A -> B -> C -> E -> F -> D -> G

**最小（代价）生成树**

最小生成树：在含有n个顶点的连通图中选择n-1条边，构成一颗极小连通子图，并使该连通子图中n-1条边上权值之和达到最小，则称其为连通网的最小生成树。  
[](https://cloud.8hfq.com/img/15330291715117.jpg)  
例如，对于如上图G4所示的连通网可以有多棵权值总和不相同的生成树。[](https://cloud.8hfq.com/img/15330291878638.jpg)

**普里姆（Prim）算法**

**普里姆(Prim)算法思想**

是用来求加权连通图的最小生成树的算法。  
基本思想  
对于图G而言，V是所有顶点的集合；现在，设置两个新的集合U和T，其中U用于存放G的最小生成树中的顶点，T存放G的最小生成树中的边。 从所有uЄU，vЄ(V-U) (V-U表示出去U的所有顶点)的边中选取权值最小的边(u, v)，将顶点v加入集合U中，将边(u, v)加入集合T中，如此不断重复，直到U=V为止，最小生成树构造完毕，这时集合T中包含了最小生成树中的所有边。  
[](https://cloud.8hfq.com/img/15330294651987.jpg)  
此时，最小生成树构造完成！它包括的顶点依次是：A B F E D C G。

**基本定义代码**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | public class MatrixUDG {   private char[] mVexs; // 顶点集合  private int[][] mMatrix; // 邻接矩阵  private static final int INF = Integer.MAX\_VALUE; // 最大值   ... } |

MatrixUDG是邻接矩阵对应的结构体。mVexs用于保存顶点，mEdgNum用于保存边数，mMatrix则是用于保存矩阵信息的二维数组。例如，mMatrix[i][j]=1，则表示”顶点i(即mVexs[i])”和”顶点j(即mVexs[j])”是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

**普里姆算法代码**

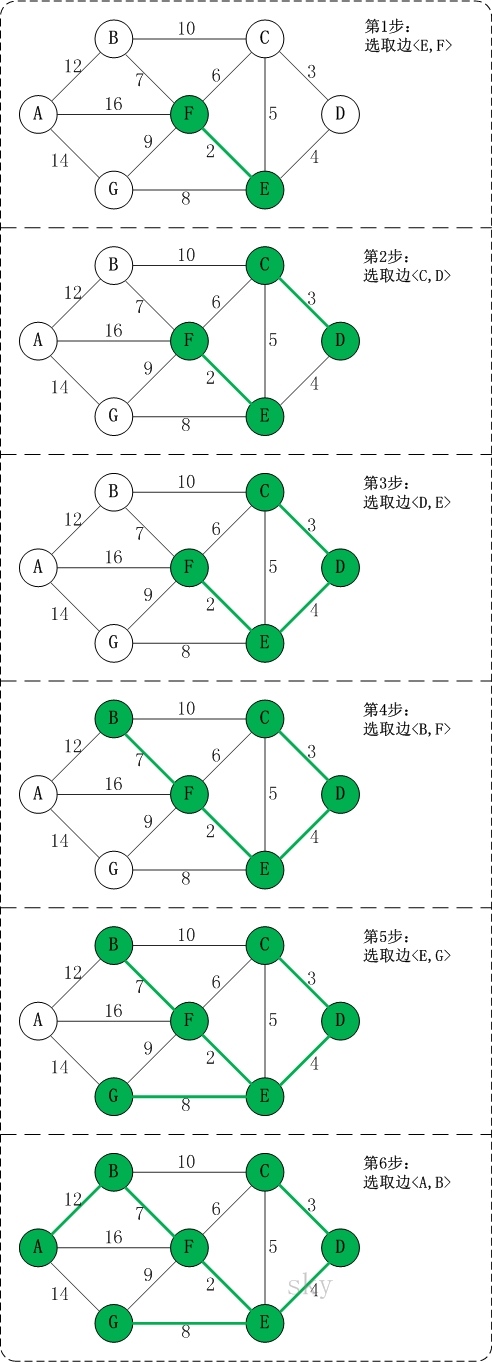
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 | /\*  \* prim最小生成树  \*  \* 参数说明：  \* start -- 从图中的第start个元素开始，生成最小树  \*/ public void prim(int start) {  int num = mVexs.length; // 顶点个数  int index=0; // prim最小树的索引，即prims数组的索引  char[] prims = new char[num]; // prim最小树的结果数组  int[] weights = new int[num]; // 顶点间边的权值   // prim最小生成树中第一个数是"图中第start个顶点"，因为是从start开始的。  prims[index++] = mVexs[start];   // 初始化"顶点的权值数组"，  // 将每个顶点的权值初始化为"第start个顶点"到"该顶点"的权值。  for (int i = 0; i < num; i++ )  weights[i] = mMatrix[start][i];  // 将第start个顶点的权值初始化为0。  // 可以理解为"第start个顶点到它自身的距离为0"。  weights[start] = 0;   for (int i = 0; i < num; i++) {  // 由于从start开始的，因此不需要再对第start个顶点进行处理。  if(start == i)  continue;   int j = 0;  int k = 0;  int min = INF;  // 在未被加入到最小生成树的顶点中，找出权值最小的顶点。  while (j < num) {  // 若weights[j]=0，意味着"第j个节点已经被排序过"(或者说已经加入了最小生成树中)。  if (weights[j] != 0 && weights[j] < min) {  min = weights[j];  k = j;  }  j++;  }   // 经过上面的处理后，在未被加入到最小生成树的顶点中，权值最小的顶点是第k个顶点。  // 将第k个顶点加入到最小生成树的结果数组中  prims[index++] = mVexs[k];  // 将"第k个顶点的权值"标记为0，意味着第k个顶点已经排序过了(或者说已经加入了最小树结果中)。  weights[k] = 0;  // 当第k个顶点被加入到最小生成树的结果数组中之后，更新其它顶点的权值。  for (j = 0 ; j < num; j++) {  // 当第j个节点没有被处理，并且需要更新时才被更新。  if (weights[j] != 0 && mMatrix[k][j] < weights[j])  weights[j] = mMatrix[k][j];  }  }   // 计算最小生成树的权值  int sum = 0;  for (int i = 1; i < index; i++) {  int min = INF;  // 获取prims[i]在mMatrix中的位置  int n = getPosition(prims[i]);  // 在vexs[0...i]中，找出到j的权值最小的顶点。  for (int j = 0; j < i; j++) {  int m = getPosition(prims[j]);  if (mMatrix[m][n]<min)  min = mMatrix[m][n];  }  sum += min;  }  // 打印最小生成树  System.out.printf("PRIM(%c)=%d: ", mVexs[start], sum);  for (int i = 0; i < index; i++)  System.out.printf("%c ", prims[i]);  System.out.printf("\n"); } |

**普里姆算法时间复杂度分析**

观察代码发现，普里姆算法主要部分是一个双重循环，外层循环内有两个并列的单层循环，单层循环内的操作都是常量级别的，其执行次数为n的平方，因此普里姆算法的时间复杂度为O(n22)

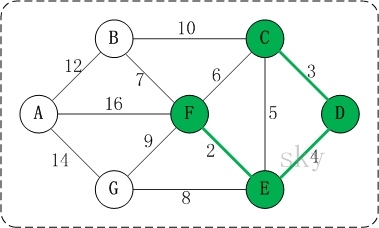
**克鲁斯卡尔（Kruskal）算法**

**克鲁斯卡尔算法思想**

克鲁斯卡尔(Kruskal)算法，是用来求加权连通图的最小生成树的算法。  
基本思想：按照权值从小到大的顺序选择n-1条边，并保证这n-1条边不构成回路。  
具体做法：首先构造一个只含n个顶点的森林，然后依权值从小到大从连通网中选择边加入到森林中，并使森林中不产生回路，直至森林变成一棵树为止。  
[](https://cloud.8hfq.com/img/15330311350970.jpg)  
此时，最小生成树构造完成！它包括的边依次是：<E,F> <C,D> <D,E> <B,F> <E,G> <A,B>。

根据前面介绍的克鲁斯卡尔算法的基本思想和做法，我们能够了解到，克鲁斯卡尔算法重点需要解决的以下两个问题：  
问题一 对图的所有边按照权值大小进行排序。  
问题二 将边添加到最小生成树中时，怎么样判断是否形成了回路。

问题一很好解决，采用排序算法进行排序即可。

问题二，处理方式是：记录顶点在”最小生成树”中的终点，顶点的终点是”在最小生成树中与它连通的最大顶点”(关于这一点，后面会通过图片给出说明)。然后每次需要将一条边添加到最小生存树时，判断该边的两个顶点的终点是否重合，重合的话则会构成回路。 以下图来进行说明：  
[](https://cloud.8hfq.com/img/15330896124450.jpg)  
(01) C的终点是F。  
(02) D的终点是F。  
(03) E的终点是F。  
(04) F的终点是F。  
关于终点，就是将所有顶点按照从小到大的顺序排列好之后；某个顶点的终点就是”与它连通的最大顶点”。 因此，接下来，虽然<C,E>是权值最小的边。但是C和E的重点都是F，即它们的终点相同，因此，将<C,E>加入最小生成树的话，会形成回路。这就是判断回路的方式。

**基本定义代码**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | // 边的结构体 private static class EData {  char start; // 边的起点  char end; // 边的终点  int weight; // 边的权重   public EData(char start, char end, int weight) {  this.start = start;  this.end = end;  this.weight = weight;  } }; |

EData是邻接矩阵边对应的结构体。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | public class MatrixUDG {   private int mEdgNum; // 边的数量  private char[] mVexs; // 顶点集合  private int[][] mMatrix; // 邻接矩阵  private static final int INF = Integer.MAX\_VALUE; // 最大值   ... } |

MatrixUDG是邻接矩阵对应的结构体。mVexs用于保存顶点，mEdgNum用于保存边数，mMatrix则是用于保存矩阵信息的二维数组。例如，mMatrix[i][j]=1，则表示”顶点i(即mVexs[i])”和”顶点j(即mVexs[j])”是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

**克鲁斯卡尔算法代码**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 | /\*  \* 克鲁斯卡尔（Kruskal)最小生成树  \*/ public void kruskal() {  int index = 0; // rets数组的索引  int[] vends = new int[mEdgNum]; // 用于保存"已有最小生成树"中每个顶点在该最小树中的终点。  EData[] rets = new EData[mEdgNum]; // 结果数组，保存kruskal最小生成树的边  EData[] edges; // 图对应的所有边   // 获取"图中所有的边"  edges = getEdges();  // 将边按照"权"的大小进行排序(从小到大)  sortEdges(edges, mEdgNum);   for (int i=0; i<mEdgNum; i++) {  int p1 = getPosition(edges[i].start); // 获取第i条边的"起点"的序号  int p2 = getPosition(edges[i].end); // 获取第i条边的"终点"的序号   int m = getEnd(vends, p1); // 获取p1在"已有的最小生成树"中的终点  int n = getEnd(vends, p2); // 获取p2在"已有的最小生成树"中的终点  // 如果m!=n，意味着"边i"与"已经添加到最小生成树中的顶点"没有形成环路  if (m != n) {  vends[m] = n; // 设置m在"已有的最小生成树"中的终点为n  rets[index++] = edges[i]; // 保存结果  }  }   // 统计并打印"kruskal最小生成树"的信息  int length = 0;  for (int i = 0; i < index; i++)  length += rets[i].weight;  System.out.printf("Kruskal=%d: ", length);  for (int i = 0; i < index; i++)  System.out.printf("(%c,%c) ", rets[i].start, rets[i].end);  System.out.printf("\n"); } |

**克鲁斯卡尔算法时间复杂度分析**

算法时间花费主要在函数sort()和单层循环上。循环是线性级的，可以认为算法时间主要花费在sort()上，因为排序算法时间复杂度一般大于常量级，因此，克鲁斯卡尔算法的时间复杂度主要由选取的排序算法决定，排序算法所处理的数据的规模由图的边e决定，与顶点无关，因此克鲁斯卡尔算法适用于稀疏图。

**最短路径**

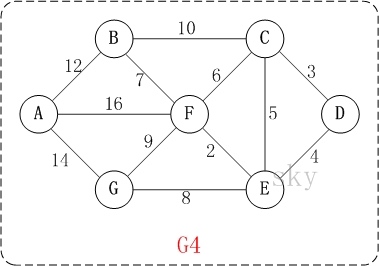
**迪杰斯特拉算法（Dijkstra）**

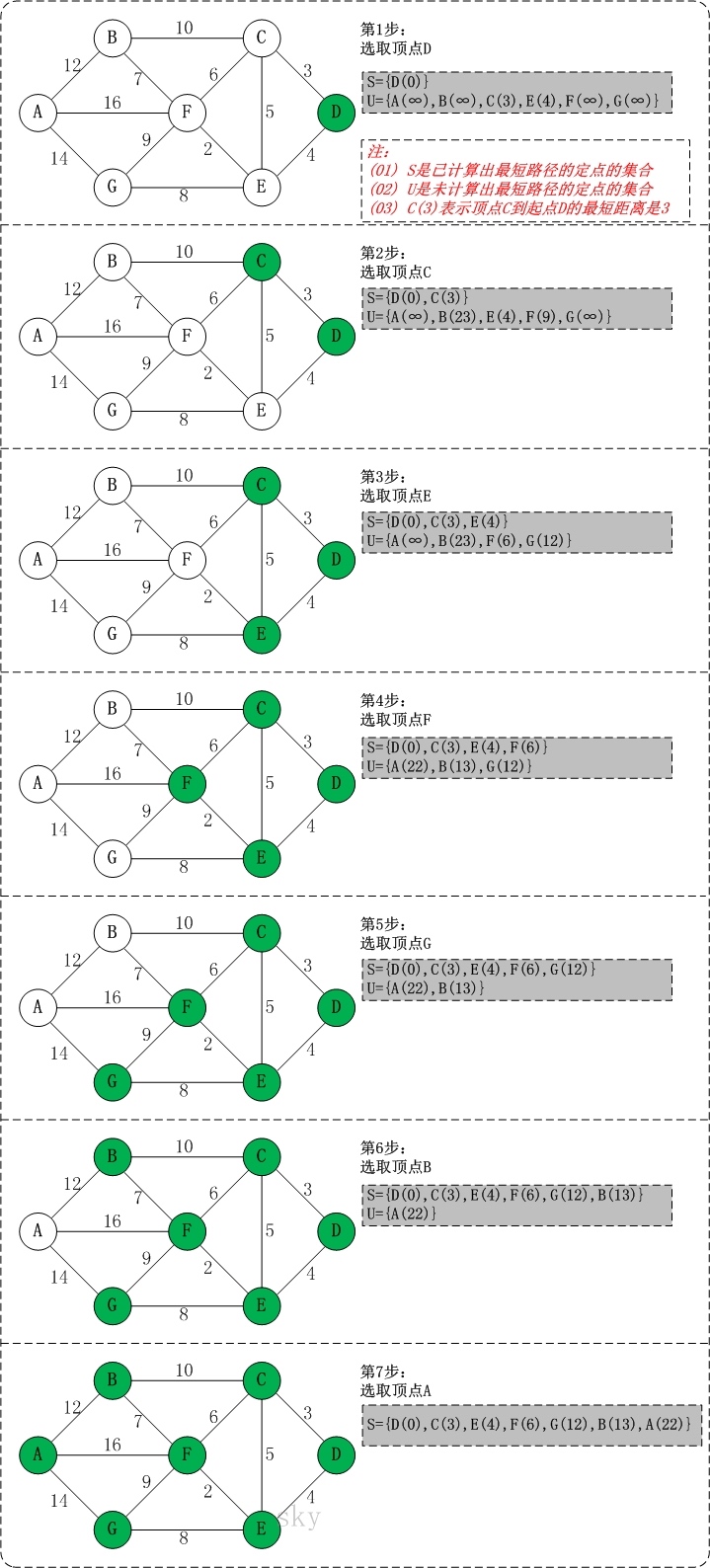
迪杰斯特拉(Dijkstra)算法是典型最短路径算法，用于计算一个节点到其他节点的最短路径。  
它的主要特点是以起始点为中心向外层层扩展(广度优先搜索思想)，直到扩展到终点为止。

**迪杰特斯拉算法基本思想**

通过Dijkstra计算图G中的最短路径时，需要指定起点s(即从顶点s开始计算)。此外，引进两个集合S和U。S的作用是记录已求出最短路径的顶点(以及相应的最短路径长度)，而U则是记录还未求出最短路径的顶点(以及该顶点到起点s的距离)。初始时，S中只有起点s；U中是除s之外的顶点，并且U中顶点的路径是”起点s到该顶点的路径”。然后，从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。 然后，再从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。 … 重复该操作，直到遍历完所有顶点。

**操作步骤**

(1) 初始时，S只包含起点s；U包含除s外的其他顶点，且U中顶点的距离为”起点s到该顶点的距离”[例如，U中顶点v的距离为(s,v)的长度，然后s和v不相邻，则v的距离为∞]。  
(2) 从U中选出”距离最短的顶点k”，并将顶点k加入到S中；同时，从U中移除顶点k。  
(3) 更新U中各个顶点到起点s的距离。之所以更新U中顶点的距离，是由于上一步中确定了k是求出最短路径的顶点，从而可以利用k来更新其它顶点的距离；例如，(s,v)的距离可能大于(s,k)+(k,v)的距离。  
(4) 重复步骤(2)和(3)，直到遍历完所有顶点。  
单纯的看上面的理论可能比较难以理解，下面通过实例来对该算法进行说明。  
[](https://cloud.8hfq.com/img/15330917841894.jpg)

[](https://cloud.8hfq.com/img/15330917268661.jpg)

**基本定义代码**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // 邻接矩阵 typedef struct \_graph {  char vexs[MAX]; // 顶点集合  int vexnum; // 顶点数  int edgnum; // 边数  int matrix[MAX][MAX]; // 邻接矩阵 }Graph, \*PGraph;  // 边的结构体 typedef struct \_EdgeData {  char start; // 边的起点  char end; // 边的终点  int weight; // 边的权重 }EData; |

Graph是邻接矩阵对应的结构体。  
vexs用于保存顶点，vexnum是顶点数，edgnum是边数；matrix则是用于保存矩阵信息的二维数组。例如，matrix[i][j]=1，则表示”顶点i(即vexs[i])”和”顶点j(即vexs[j])”是邻接点；matrix[i][j]=0，则表示它们不是邻接点。  
EData是邻接矩阵边对应的结构体。

**迪杰斯特拉算法代码**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 | /\*  \* Dijkstra最短路径。  \* 即，统计图(G)中"顶点vs"到其它各个顶点的最短路径。  \*  \* 参数说明：  \* G -- 图  \* vs -- 起始顶点(start vertex)。即计算"顶点vs"到其它顶点的最短路径。  \* prev -- 前驱顶点数组。即，prev[i]的值是"顶点vs"到"顶点i"的最短路径所经历的全部顶点中，位于"顶点i"之前的那个顶点。  \* dist -- 长度数组。即，dist[i]是"顶点vs"到"顶点i"的最短路径的长度。  \*/ void dijkstra(Graph G, int vs, int prev[], int dist[]) {  int i,j,k;  int min;  int tmp;  int flag[MAX]; // flag[i]=1表示"顶点vs"到"顶点i"的最短路径已成功获取。   // 初始化  for (i = 0; i < G.vexnum; i++)  {  flag[i] = 0; // 顶点i的最短路径还没获取到。  prev[i] = 0; // 顶点i的前驱顶点为0。  dist[i] = G.matrix[vs][i];// 顶点i的最短路径为"顶点vs"到"顶点i"的权。  }   // 对"顶点vs"自身进行初始化  flag[vs] = 1;  dist[vs] = 0;   // 遍历G.vexnum-1次；每次找出一个顶点的最短路径。  for (i = 1; i < G.vexnum; i++)  {  // 寻找当前最小的路径；  // 即，在未获取最短路径的顶点中，找到离vs最近的顶点(k)。  min = INF;  for (j = 0; j < G.vexnum; j++)  {  if (flag[j]==0 && dist[j]<min)  {  min = dist[j];  k = j;  }  }  // 标记"顶点k"为已经获取到最短路径  flag[k] = 1;   // 修正当前最短路径和前驱顶点  // 即，当已经"顶点k的最短路径"之后，更新"未获取最短路径的顶点的最短路径和前驱顶点"。  for (j = 0; j < G.vexnum; j++)  {  tmp = (G.matrix[k][j]==INF ? INF : (min + G.matrix[k][j])); // 防止溢出  if (flag[j] == 0 && (tmp < dist[j]) )  {  dist[j] = tmp;  prev[j] = k;  }  }  }   // 打印dijkstra最短路径的结果  printf("dijkstra(%c): \n", G.vexs[vs]);  for (i = 0; i < G.vexnum; i++)  printf(" shortest(%c, %c)=%d\n", G.vexs[vs], G.vexs[i], dist[i]); } |

**迪杰斯特拉算法时间复杂度分析**

由算法代码可知，本算法主要部分为一个双重循环，外层循环内部有两个并列的单层循环，可以任取一个循环内的操作为基本操作，基本操作执行的总次数为n2n2 因此时间复杂度为O(n2)O(n2).

**弗洛伊德（Floyd）算法**

弗洛伊德算法是解决任意两点间的最短路径的一种算法，可以正确处理有向图或有向图或负权（但不可存在负权回路)的最短路径问题，同时也被用于计算有向图的传递闭包。

**弗洛伊德算法思想**

通过Floyd计算图G=(V,E)中各个顶点的最短路径时，需要引入两个矩阵，矩阵S中的元素a[i][j]表示顶点i(第i个顶点)到顶点j(第j个顶点)的距离。矩阵P中的元素b[i][j]，表示顶点i到顶点j经过了b[i][j]记录的值所表示的顶点。

假设图G中顶点个数为N，则需要对矩阵D和矩阵P进行N次更新。初始时，矩阵D中顶点a[i][j]的距离为顶点i到顶点j的权值；如果i和j不相邻，则a[i][j]=∞，矩阵P的值为顶点b[i][j]的j的值。 接下来开始，对矩阵D进行N次更新。第1次更新时，如果”a[i][j]的距离” > “a[i][0]+a[0][j]”(a[i][0]+a[0][j]表示”i与j之间经过第1个顶点的距离”)，则更新a[i][j]为”a[i][0]+a[0][j]”,更新b[i][j]=b[i][0]。 同理，第k次更新时，如果”a[i][j]的距离” > “a[i][k-1]+a[k-1][j]”，则更新a[i][j]为”a[i][k-1]+a[k-1][j]”,b[i][j]=b[i][k-1]。更新N次之后，操作完成！

**弗洛伊德算法过程**

1）设置两个矩阵A 和PATH，初始时将图的邻接矩阵赋值给A，将矩阵Path中元素全部设置为01.  
2）以顶点K为中间顶点，k取0~n-1(n为图中顶点个数)，对图中所有顶点对{i，j}进行如下检测：  
如果A[i][j]>A[i][k]+A[k][j]，则将A[i][j]改成A[i][k]+A[k][j]的值，将Path[i][j]改成K，否则什么也不做

**弗洛伊德算法代码**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | void Floyd(MGraph g,int Paht[][maxSize]){  int i,j,k;  int A[maxSize][maxSize];  //这个双循环对数组A和Path进行了初始化  for(i=0;i<g.n;++i){  A[i][j]=g.edges[i][j];  Path[i][j]=-1;  }  for (int k = 0; k < g.n; ++k) {  for (int i = 0; i < g.n; ++i) {  for (int j = 0; j < g.n; ++j) {  if(A[i][j]>A[i][k]+A[k][j]){  A[j][j] = A[i][k]+A[k][j];  Paht[i][j]=k;  }  }  }  } } |

**弗洛伊德算法时间复杂度分析**

由算法可知，主要是有三层循环，基本操作执行次数为n3n3，所以时间复杂度为O(n3)O(n3)

**拓扑排序**

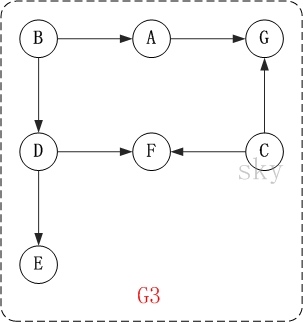
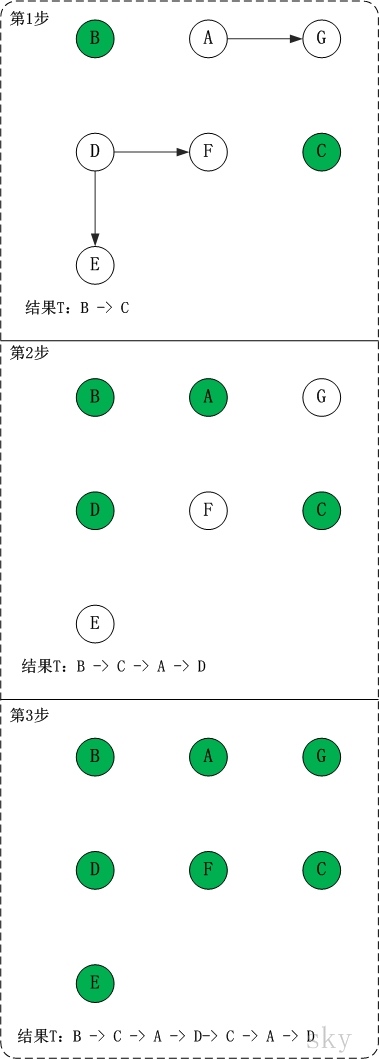
**AOV网**

活动在顶点上的网（Activity On Vertex network,AOV）是一种可以形象的反映出整个工程中各个活动之间的先后关系的有向图。

**拓扑排序介绍**

拓扑排序(Topological Order)是指，将一个有向无环图(Directed Acyclic Graph简称DAG)进行排序进而得到一个有序的线性序列。  
这样说，可能理解起来比较抽象。下面通过简单的例子进行说明！  
例如，一个项目包括A、B、C、D四个子部分来完成，并且A依赖于B和D，C依赖于D。现在要制定一个计划，写出A、B、C、D的执行顺序。这时，就可以利用到拓扑排序，它就是用来确定事物发生的顺序的。  
在拓扑排序中，如果存在一条从顶点A到顶点B的路径，那么在排序结果中B出现在A的后面。

**拓扑排序算法思路**

1. 构造一个队列Q(queue) 和 拓扑排序的结果队列T(topological)；
2. 把所有没有依赖顶点的节点放入Q；
3. 当Q还有顶点的时候，执行下面步骤：  
   3.1 从Q中取出一个顶点n(将n从Q中删掉)，并放入T(将n加入到结果集中)；  
   3.2 对n每一个邻接点m(n是起点，m是终点)；  
   3.2.1 去掉边<n,m>;  
   3.2.2 如果m没有依赖顶点，则把m放入Q;  
   注：顶点A没有依赖顶点，是指不存在以A为终点的边。  
   [](https://cloud.8hfq.com/img/15331914116919.jpg)  
   以上图为例，来对拓扑排序进行演示  
   [](https://cloud.8hfq.com/img/15331914212521.jpg)

第1步：将B和C加入到排序结果中。  
顶点B和顶点C都是没有依赖顶点，因此将C和C加入到结果集T中。假设ABCDEFG按顺序存储，因此先访问B，再访问C。访问B之后，去掉边<B,A>和<B,D>，并将A和D加入到队列Q中。同样的，去掉边<C,F>和<C,G>，并将F和G加入到Q中。  
(01) 将B加入到排序结果中，然后去掉边<B,A>和<B,D>；此时，由于A和D没有依赖顶点，因此并将A和D加入到队列Q中。  
(02) 将C加入到排序结果中，然后去掉边<C,F>和<C,G>；此时，由于F有依赖顶点D，G有依赖顶点A，因此不对F和G进行处理。  
第2步：将A,D依次加入到排序结果中。  
第1步访问之后，A,D都是没有依赖顶点的，根据存储顺序，先访问A，然后访问D。访问之后，删除顶点A和顶点D的出边。  
第3步：将E,F,G依次加入到排序结果中。

因此访问顺序是：B -> C -> A -> D -> E -> F -> G