第8章 图

**一、复习要点**

图是一种重要的非线性结构。它的特点是每一个顶点都可以与其它顶点相关联，与树不同，图中各个顶点的地位都是平等的，对顶点的编号都是人为的。通常，定义图由两个集合构成：一个是顶点的非空有穷集合，一个是顶点与顶点之间关系(边)的有穷集合。对图的处理要区分有向图与无向图。它的存储表示可以使用邻接矩阵，可以使用邻接表，前者属顺序表示，后者属链接表示。在本章着重讨论了图的深度优先搜索和广度优先搜索算法，附带引入了生成树与生成森林的概念。对于带权图，给出了最小生成树的两种方法：Prim算法和Kruskal算法，后者使用了最小堆和并查集作为它的辅助求解手段。在解决最短路径问题时，采用了逐步求解的策略。最后讨论了作工程计划时常用的活动网络。涉及的主要概念是拓扑排序和关键路径，在解决应用问题时它们十分有用。

本章复习的要点是：

**1、**基本知识点

主要要求理解图的基本概念，包括图的定义、图的连通性、图的路径和路径长度、图中各顶点的度及度的度量、无向连通图的最大边数和最小边数，有向强连通图的最大边数与最小边数等。掌握图的存储表示，包括邻接矩阵和邻接表，以及这些存储表示上的典型操作，如构造、求根、找第一个邻接顶点、找下一个邻接顶点等操作的实现算法。并要求掌握图的两种遍历算法：深度优先搜索和广度优先搜索算法，以及求解连通性问题的方法。理解求解关节点及构造重连通图的方法。此外，要求掌握构造最小生成树的Prim算法和Kruskal方法，掌握活动网络的拓扑排序算法，掌握求解关键路径的方法。需要注意的是，让某个关键活动提前完成，是否能让整个工程提前完成。

**2、**算法设计

⮚ 建立无向带权图的邻接表的算法，要求输入边的数目随机而定。

⮚ 图的深度优先搜索的递归算法。

⮚ 利用图的深度优先搜索的递归算法建立图的深度优先生成森林(用左子女右兄弟表示)的算法。

⮚ 图的广度优先搜索算法。

⮚ 利用图的广度优先搜索算法建立图的广度优先生成森林(用左子女右兄弟表示)的算法。

⮚ 求解最小生成树的Prim算法，注意nearvex和lowcost辅助数组的变化。

⮚ 求解最小生成树的Kruskal算法，注意minheap和UFset的变化。

⮚ 求解最短路径的dijkstra算法，注意dist辅助数组的变化。

⮚ 有向图中求解拓扑排序的算法，要求用邻接表作为图的存储表示。注意算法执行过程中入度为零的顶点栈的变化。

⮚ 有向图中求解拓扑排序的算法，要求用邻接矩阵作为图的存储表示。

**二、难点和重点**

1、图：图的定义与图的存储表示

⮚ 邻接矩阵表示(通常是稀疏矩阵)

⮚ 邻接表与逆邻接表表示，要求建立算法

⮚ 邻接多重表(十字链表)表示

2、深度优先遍历与广度优先遍历

* 生成树与生成树林的定义
* 深度优先搜索算法和广度优先搜索算法
* 深度优先搜索是个递归的过程，而广度优先搜索是个非递归的过程
* 为防止重复访问已经访问过的顶点，需要设置一个访问标志数组visited

3、图的连通性

⮚ 深度优先搜索可以遍历一个连通分量上的所有顶点

⮚ 对非连通图进行遍历，可以建立一个生成森林

⮚ 对非强连通图进行遍历，可能建立一个生成森林

⮚ 关节点的求解方法和以最少的边构成重连通图的方法

4、最小生成树

⮚ 对于连通网络、可用不会构成环路的权值最小的n-1条边构成最小生成树

⮚ 会画出用Kruskal算法及Prim算法构造最小生成树的过程

5、单源最短路径

⮚ 采用逐步求解的方式求某一顶点到其他顶点的最短路径的方法

⮚ 要求每条边的权值必须大于零

6、活动网络

⮚ 拓扑排序、关键路径、关键活动、AOE网

⮚ 拓扑排序将一个偏序图转化为一个全序图。

⮚ 为实现拓扑排序，要建立一个栈，将所有入度为零的顶点进栈

⮚ 关键路径的计算

**三、教材中习题的解析**

8-1 画出1个顶点、2个顶点、3个顶点、4个顶点和5个顶点的无向完全图。试证明在n个顶点的无向完全图中，边的条数为n(n-1)/2。

【解答】

2个顶点的

无向完全图

1个顶点的

无向完全图

5个顶点的

无向完全图

4个顶点的

无向完全图

3个顶点的

无向完全图

【证明】

在有n个顶点的无向完全图中，每一个顶点都有一条边与其它某一顶点相连，所以每一个顶点有n-1条边与其他n-1个顶点相连，总计n个顶点有n(n-1)条边。但在无向图中，顶点i到顶点j与顶点j到顶点i是同一条边，所以总共有n(n-1)/2条边。

8-2 右边的有向图是强连通的吗？请列出所有的简单路径。

A

B

C

D

E

F

【解答】

判断一个有向图是否强连通，要看从任一顶点出发是否能够回到该顶点。右面的有向图做不到这一点，它不是强连通的有向图。各个顶点自成强连通分量。

所谓简单路径是指该路径上没有重复的顶点。

从顶点A出发，到其他的各个顶点的简单路径有A→B，A→D→B，A→B→C，A→D→B→C，A→D，A→B→E，A→D→E，A→D→B→E，A→B→C→F→E，A→D→B→C→F→E，A→B→C→F，A→D→B→C→F。

从顶点B出发，到其他各个顶点的简单路径有B→C，B→C→F，B→E，B→C→F→E。

从顶点C出发，到其他各个顶点的简单路径有C→F，C→F→E。

从顶点D出发，到其他各个顶点的简单路径有D→B，D→B→C，D→B→C→F，D→E，D→B→E，D→B→C→F→E。

从顶点E出发，到其他各个顶点的简单路径无。

从顶点F出发，到其他各个顶点的简单路径有F→E。

8-3 给出右图的邻接矩阵、邻接表和邻接多重表表示。

A

B

C

D

E

F

【解答】

1. 邻接矩阵

3

2

4

5

1

4

4

∧

∧

∧

∧

∧

∧

1

1. A
2. B
3. C
4. D
5. E
6. F
7. 邻接表

（出边表）

1. A
2. B
3. C
4. D
5. E
6. F

0

3

1

0

1

3

5

2

∧

∧

∧

∧

∧

∧

（入边表）

data fin fout

(3) 邻接多重表（十字链表）

1. A
2. B
3. C
4. D
5. E
6. F

0 3 (A, D)

3 1 (D, B)

∧

∧

0 1 (A, B)

1 2 (B, C)

1 4 (B, E)

5 4 (F, E)

3 4 (D, E)

2 5 (C, F)

i j ilink jlink

∧

∧

∧

∧

∧

∧

∧

∧

∧

8-4 用邻接矩阵表示图时，若图中有1000个顶点，1000条边，则形成的邻接矩阵有多少矩阵元素？有多少非零元素？是否稀疏矩阵？

【解答】

一个图中有1000个顶点，其邻接矩阵中的矩阵元素有10002 = 1000000个。它有1000个非零元素（对于有向图）或2000个非零元素（对于无向图），因此是稀疏矩阵。

8-5 用邻接矩阵表示图时，矩阵元素的个数与顶点个数是否相关？与边的条数是否相关？

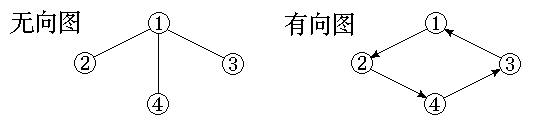
【解答】

用邻接矩阵表示图，矩阵元素的个数是顶点个数的平方，与边的条数无关。矩阵中非零元素的个数与边的条数有关。

8-6 有n个顶点的无向连通图至少有多少条边？有n个顶点的有向强连通图至少有多少条边？试举例说明。

【解答】

n个顶点的无向连通图至少有n-1条边，n个顶点的有向强连通图至少有n条边。例如：



特例情况是当n = 1时，此时至少有0条边。

8-7对于有n个顶点的无向图，采用邻接矩阵表示，如何判断以下问题： 图中有多少条边？任意两个顶点i和j之间是否有边相连？任意一个顶点的度是多少？

【解答】

用邻接矩阵表示无向图时，因为是对称矩阵，对矩阵的上三角部分或下三角部分检测一遍，统计其中的非零元素个数，就是图中的边数。如果邻接矩阵中A[i][j] 不为零，说明顶点i与顶点j之间有边相连。此外统计矩阵第i行或第i列的非零元素个数，就可得到顶点i的度数。

8-8对于如右图所示的有向图，试写出：

①

②

③

④

⑤

(1) 从顶点①出发进行深度优先搜索所得到的深度优先生成树；

(2) 从顶点②出发进行广度优先搜索所得到的广度优先生成树；

【解答】

(1) 以顶点①为根的深度优先生成树（不唯一）：② ③ ④ ⑤ ⑥

①

②

③

④

⑤

①

②

③

④

⑤

或

(2) 以顶点②为根的广度优先生成树：

①

②

③

④

⑤

8-9 试扩充深度优先搜索算法，在遍历图的过程中建立生成森林的左子女-右兄弟链表。算法的首部为 **void** Graph**::**DFS ( **const int** v, **int** visited [ ], TreeNode<**int**> \* t ) 其中，指针t指向生成森林上具有图顶点v信息的根结点。（提示：在继续按深度方向从根v的某一未访问过的邻接顶点w向下遍历之前，建立子女结点。但需要判断是作为根的第一个子女还是作为其子女的右兄弟链入生成树。）

【解答】

为建立生成森林，需要先给出建立生成树的算法，然后再在遍历图的过程中，通过一次次地调用这个算法，以建立生成森林。

**te mplate<Type> void** Graph<**Type**> **::** DFS\_Tree ( **const int** v, **int** visited [ ], TreeNode<**Type>** \*t ) **{**

//从图的顶点v出发, 深度优先遍历图, 建立以t (已在上层算法中建立)为根的生成树。

Visited[v] = 1**; int** first = 1**;**  TreeNode**<Type>** \* p, \* q**;**

**int** w = GetFirstNeighbor ( v )**;** //取第一个邻接顶点

**while** ( w != -1 ) **{** //若邻接顶点存在

**if** ( vosited[w] == 0 ) **{**  //且该邻接结点未访问过

p = **new** TreeNode<**Type>** ( GetValue (w) )**;** //建立新的生成树结点

**if** ( first == 1 ) //若根\*t还未链入任一子女

**{** t->setFirstChild ( p )**;** first = 0**; }** //新结点\*p成为根\*t的第一个子女

**else** q->setNextSibling ( p )**;** //否则新结点\*p成为\*q的下一个兄弟

q = p**;** //指针q总指示兄弟链最后一个结点

DFS\_Tree ( w, visited, q )**;** //从\*q向下建立子树

**}**

w = GetNextNeighbor ( v, w )**;** //取顶点v排在邻接顶点w的下一个邻接顶点

**}**

**}**

下一个算法用于建立以左子女-右兄弟链表为存储表示的生成森林。

**template<Type> void** Graph<**Type**> **::** DFS\_Forest ( Tree<**Type>** **&** T ) **{**

//从图的顶点v出发, 深度优先遍历图, 建立以左子女-右兄弟链表表示的生成森林T。

T.root = NULL**; int** n =NumberOfVertices ( )**;**  //顶点个数

TreeNode<**Type**> \* p, \* q**;**

**int** \* visited = **new int** [ n ]**;** //建立访问标记数组

**for** ( **int** v = 0**;** v < n**;** v++ ) visited[v] = 0**;**

**for** ( v = 0**;** v < n**;** v++ ) //逐个顶点检测

**if** ( visited[v] == 0 ) **{** //若尚未访问过

p = **new** TreeNode<**Type**> ( GetValue ( v ) )**;** //建立新结点\*p

**if** ( T.root == NULL ) T.root = p**;** //原来是空的生成森林, 新结点成为根

**else** q-> setNextSibling ( p )**;** //否则新结点\*p成为\*q的下一个兄弟

q = p**;**

DFS\_Tree ( v, visited, p )**;** //建立以\*p为根的生成树

**}**

**}**

8-10 用邻接表表示图时，顶点个数设为n，边的条数设为e，在邻接表上执行有关图的遍历操作时，时间代价是O(n\*e)？还是O(n+e)？或者是O(max(n,e))？

【解答】

在邻接表上执行图的遍历操作时，需要对邻接表中所有的边链表中的结点访问一次，还需要对所有的顶点访问一次，所以时间代价是O(n+e)。

8-11 右图是一个连通图，请画出

⑩

①

②

③

④

⑤

⑥

⑦

⑧

⑨

(1) 以顶点①为根的深度优先生成树；

(2) 如果有关节点，请找出所有的关节点。

(3) 如果想把该连通图变成重连通图，至少在图中加几条边？如何加？

【解答】

(1) 以顶点①为根的深度优先生成树：

⑩

①

②

③

④

⑤

⑥

⑦

⑧

⑨

(2) 关节点为 ①,②,③,⑦,⑧

(3) 至少加四条边 (1, 10), (3, 4), (4, 5), (5, 6)。从③的子孙结点⑩到③的祖先结点①引一条边，从②的子孙结点④到根①的另一分支③引一条边，并将⑦的子孙结点⑤、⑥与结点④连结起来，可使其变为重连通图。

⑩

①

②

③

④

⑤

⑥

⑦

⑧

⑨

8-12试证明在一个有n个顶点的完全图中，生成树的数目至少有2n-1-1。

【证明】略

5

①

②

③

④

⑤

⑥

11

7

6

8

10

9

7

8-13 编写一个完整的程序，首先定义堆和并查集的结构类型

和相关操作，再定义

Kruskal求连通网络

的最小生成树算法的

实现。并以右图为

例，写出求解过程中

堆、并查集和最小生成树的变化。

【解答】

求解过程的第一步是对所有的边，按其权值大小建堆：

1 3 11

3 4 5

1 2 7

2 3 10

2 4 9

1 3 11

1 2 7

2 4 9

2 3 10

1 2 7

1 3 11

2 3 10

加(1, 2), (1, 3), (2,3)

加(2, 4)

加(3, 4)

1 3 11

3 4 5

1 2 7

3 5 7

2 4 9

2 3 10

3 6 8

1 3 11

3 4 5

1 2 7

3 5 7

2 4 9

2 3 10

加(3, 5)

加(3, 6)

1 2 7

3 4 5

5 6 6

3 5 7

2 4 9

2 3 10

3 6 8

1 3 11

加(5, 6)

求解过程中并查集与堆的变化：

③

④

1 3 11

5 6 6

1 2 7

3 5 7

2 4 9

2 3 10

3 6 8

1 2 7

⑥

④

3 6 8

3 5 7

⑤

③

2 3 10

2 4 9

1 3 11

选(5,6,6)

选(3,4,5)

1 3 11

3 5 7

2 4 9

2 3 10

3 6 8

③

④

⑤

⑥

①

②

1 3 11

2 4 9

2 3 10

3 6 8

③

④

⑤

⑥

①

②

选(3,5,7)

选(1,2,7)

1 3 11

2 3 10

③

④

⑤

⑥

①

②

③

④

⑤

⑥

①

②

1 3 11

2 3 10

2 4 9

选(2,4,9), 结束

选(3,6,8), 在同一连通分量上, 不加

最后得到的生成树如下

3 1 -6 3 3 5

0 1 2 3 4 5 6

①

②

③

④

⑤

⑥

6

7

5

7

9

并查集的存储表示

完整的程序如下:

#**include** <iostream.h>

**template <class Type> class** MinHeap **{**

**public:**

**enum {** MaxHeapSize = 50 **};**

MinHeap ( **int** Maxsize = MaxHeapSize )**;**

MinHeap ( **Type** Array[ ]**, int** n )**;**

**void** Insert ( **const Type &**ele )**;**

**void** RemoveMin ( **Type &**Min )**;**

**void** Output ()**;**

**private:**

**void** FilterDown ( **int** start**, int** end )**;**

**void** FilterUp ( **int** end )**;**

**Type \***pHeap**;**

**int** HMaxSize**;**

**int** CurrentSize**;**

**};**

**class** UFSets **{**

**public:**

**enum {** MaxUnionSize = 50 **};**

UFSets ( **int** MaxSize = MaxUnionSize )**;**

~UFSets () **{ delete** [ ] m\_pParent**; }**

**void** Union ( **int** Root1**, int** Root2 )**;**

**int** Find ( **int** x )**;**

**private:**

**int** m\_iSize**;**

**int** \*m\_pParent**;**

**};**

**class** Graph **{**

**public:**

**enum {** MaxVerticesNum = 50 **};**

Graph( **int** Vertices = 0) **{** CurrentVertices = Vertices**;** InitGraph()**; }**

**void** InitGraph ()**;**

**void** Kruskal ()**;**

**int** GetVerticesNum () **{ return** CurrentVertices**; }**

**private:**

**int** Edge[MaxVerticesNum][MaxVerticesNum]**;**

**int** CurrentVertices**;**

**};**

**class** GraphEdge **{**

**public:**

**int** head, tail**;**

**int** cost**;**

**int operator** <= ( GraphEdge **&**ed )**;**

**};**

GraphEdge **:: operator** <= ( GraphEdge **&**ed ) **{**

**return this**->cost <= ed.cost**;**

**}**

UFSets **::** UFSets ( **int** MaxSize ) **{**

m\_iSize = MaxSize**;**

m\_pParent = **new int**[m\_iSize];

**for** ( **int** i = 0**;** i < m\_iSize**;** i++ ) m\_pParent[i] = -1**;**

**}**

**void** UFSets **::** Union ( **int** Root1**, int** Root2 ) **{**

m\_pParent[Root2] = Root1**;**

**}**

**int** UFSets **::** Find ( **int** x ) **{**

**while** ( m\_pParent[x] >= 0 ) x = m\_pParent[x]**;**

**return** x**;**

**}**

**template <class Type>** MinHeap**<Type> ::** MinHeap ( **int** Maxsize ) **{**

HMaxSize = Maxsize**;**

pHeap = **new Type**[HMaxSize]**;**

CurrentSize = -1**;**

**}**

**template <class Type>** MinHeap**<Type> ::** MinHeap ( **Type** Array[]**, int** n ) **{**

HMaxSize = ( n < MaxHeapSize ) **?** MaxHeapSize **:** n**;**

pHeap = **new Type**[HMaxSize]**;**

**for** ( **int** i = 0**;** i < n**;** i++ ) pHeap[i] = Array[i]**;**

CurrentSize = n-1**;**

**int** iPos = ( CurrentSize - 1 ) / 2**;**

**while** ( iPos >= 0 ) **{**

FilterDown ( iPos**,** CurrentSize )**;**

iPos--**;**

**}**

**}**

**template <class Type> void** MinHeap<**Type**> **::** FilterDown ( **int** start**, int** end ) **{**

**int** i = start**,** j = 2 \* start + 1**;**

**Type** Temp = pHeap[i]**;**

**while** ( j <= end ) **{**

**if** ( j < end **&&** pHeap[j+1] <= pHeap[j] ) j++**;**

**if** ( Temp <= pHeap[j] ) **break;**

pHeap[i] = pHeap[j]**;**

i = j**;** j = 2 \* j + 1**;**

**}**

pHeap[i] = Temp**;**

**}**

**template <class Type> void** MinHeap<**Type**> **::** FilterUp ( **int** end ) **{**

**int** i = end**,** j = ( end - 1 ) / 2**;**

**Type** Temp = pHeap[i]**;**

**while** ( i > 0 ) **{**

**if** ( pHeap[j] <= Temp ) **break;**

pHeap[i] = pHeap[j]**;**

i = j**;** j = ( j - 1 ) / 2**;**

**}**

pHeap[i] = Temp**;**

**}**

**template <class Type> void** MinHeap**<Type> ::** Insert ( **const Type &**ele ) **{**

CurrentSize++**;**

**if** ( CurrentSize == HMaxSize ) **return;**

pHeap[CurrentSize] = ele**;**

FilterUp ( CurrentSize )**;**

**}**

**template <class Type> void** MinHeap<**Type> ::** RemoveMin ( **Type &**Min ) **{**

**if** ( CurrentSize < 0 ) **return;**

Min = pHeap[0]**;**

pHeap[0] = pHeap[CurrentSize--]**;**

FilterDown ( 0, CurrentSize )**;**

**}**

**template <class Type> void** MinHeap**<Type> ::** Output ( ) **{**

**for** ( **int** i = 0**;** i <= CurrentSize**;** i++ ) **cout** << pHeap[i] << " "**;**

**cout** << **endl;**

**}**

**void** Graph **::** InitGraph( ) **{**

Edge[0][0] = -1**;** Edge[0][1] = 28**;** Edge[0][2] = -1**;** Edge[0][3] = -1**;** Edge[0][4] = -1**;**

Edge[0][5] = 10**;**  Edge[0][6] = -1**;**

Edge[1][1] = -1**;** Edge[1][2] = 16**;** Edge[1][3] = -1**;** Edge[1][4] = -1**;** Edge[1][5] = -1**;**

Edge[1][6] = 14**;**

Edge[2][2] = -1**;** Edge[2][3] = 12**;** Edge[2][4] = -1**;**  Edge[2][5] = -1**;** Edge[2][6] = -1**;**

Edge[3][3] = -1**;** Edge[3][4] = 22**;** Edge[3][5] = -1**;** Edge[3][6] = 18**;**

Edge[4][4] = -1**;** Edge[4][5] = 25**;**  Edge[4][6] = 24**;**

Edge[5][5] = -1**;** Edge[5][6] = -1**;**

Edge[6][6] = -1**;**

**for** ( **int** i = 1**;** i < 6**;** i++ )

**for** ( **int** j = 0**;** j < i**;** j ++ ) Edge[i][j] = Edge[j][i]**;**

**}**

**void** Graph **::** Kruskal( ) **{**

GraphEdge e**;**

**int** VerticesNum = GetVerticesNum ( )**;**

**int** i**,** j**,** count**;**

MinHeap<GraphEdge> heap ( VerticesNum \*VerticesNum )**;**

UFSets set ( VerticesNum )**;**

**for** ( i = 0**;** i < VerticesNum**;** i++ )

**for** ( j = i + 1**;** j < VerticesNum**;** j++ )

**if** ( Edge[i][j] > 0 )  **{**

e.head = i**;** e.tail = j**;** e.cost = Edge[i][j]**;**

heap.Insert ( e )**;**

**}**

count = 1**;**

**while** ( count < VerticesNum ) **{**

heap.RemoveMin ( e )**;**

i = set.Find ( e.head )**;**

j = set.Find ( e.tail )**;**

**if** ( i != j ) **{**

set.Union ( i, j )**;**

count++**;**

**cout** << "( " << e.head << ", " << e.tail << ", " << e.cost << " )" << **endl;**

**}**

**}**

**}**

8-14 利用Dijkstra算法的思想，设计一个求最小生成树的算法。

【解答】

计算连通网络的最小生成树的Dijkstra算法可描述如下：将连通网络中所有的边以方便的次序逐步加入到初始为空的生成树的边集合T中。每次选择并加入一条边时，需要判断它是否会与先前加入T的边构成回路。如果构成了回路，则从这个回路中将权值最大的边退选。

下面以邻接矩阵作为连通网络的存储表示，并以并查集作为判断是否出现回路的工具，分析算法的执行过程。

① ② ③ ④ ⑤ ⑥

26

21

11

①

②

⑤

④

③

⑥

18

14

16

19

9

5

6



①

②③④⑤⑥

①

②

⑤

⑥

③

④

16

①

②

⑤

④

③

⑥

14

21

19

9

5

×

①

②

⑤

⑥

①

②

⑤

④

⑥

14

16

21

③

并查集, 表明4个结点在同一连通分量上

①

②

⑤

⑥

③

④

①

②

⑤

④

③

⑥

14

16

5

6

19

11

×

①

②

⑤

⑥

③

④

9

①

②

⑤

④

③

⑥

14

16

5

6

×

19

最终得到的最小生成树为

①

②

⑤

④

③

⑥

14

16

5

6

11

算法的思路如下：

1. 并查集初始化：将所有顶点置为只有一个顶点的连通分量；
2. 检查所有的边：

ⅰ）若边的两个端点i与j不在同一连通分量上(i与j在并查集中不同根)，则连通之(合并)；

ⅱ) 若边的两个端点i与j在同一连通分量上(i与j在并查集中同根)，则

* 在并查集中寻找离i与j最近的共同祖先结点
* 分别从i与j向上检测具有最大权值的边
* 在并查集上删除具有最大权值的边，加入新的边。

下面给出实现算法：

**const int** MaxNum = 10000**;**

**void** Graph **::** Dijkstra ( ) **{**

GraphEdge e**;**

**int** VerticesNum = GetVerticesNum ( )**;**

**int** i**,** j**,** p, q, k**;**

int disJoint[VerticesNum]**;** //并查集

**for** ( i = 0**;** i < VerticesNum**;** i++ ) disJoint[i] = -1**;** //并查集初始化

**for** ( i = 0**;** i < VerticesNum-1**;** i++ ) //检查所有的边

**for** ( j = i + 1**;** j < VerticesNum**;** j++ )

**if** ( Edge[i][j] < MaxNum )  **{** //边存在

p = i**;** q = j**;** //判结点i与j是否在同一连通分量上

**while** ( disJoint[p] >= 0 ) p = disJoint[p]**;**

**while** ( disJoint[q] >= 0 ) p = disJoint[q]**;**

**if** ( p != q ) disJoint[j] = i**; //** i与j不在同一连通分量上, 连通之

**} else { //** i与j在同一连通分量上

p = i**;** //寻找离结点i与j最近的祖先结点

**while** ( disJoint[p] >= 0 ) **{** //每变动一个p, 就对q到根的路径检测一遍

q = j**;**

**while** ( disJoint[q] >= 0 **&&** disJoint[q] == disJoint[p] )

q = disJoint[q]**;**

**if** ( disJoint[q] == disJoint[p] ) **break;**

**else** p = disJoint[p]**;**

**}**

k = disJoint[p]**;** //结点k是i和j的最近的共同祖先

p = i**;** q = disJoint[p]**;** max = -MaxNum**;** //从i到k找权值最大的边(s1, s2)

**while** ( q <= k ) **{**

**if** ( Edge[q][p] > max ) **{** max = Edge[q][p]**;**  s1 = p**;**  s2 = q**; }**

p =q**;** q = disJoint[p]**;**

**}**

p = j**;** q = disJoint[p]**;** max = -MaxNum**;** //从j到k找权值最大的边(t1, t2)

**while** ( q <= k ) **{**

**if** ( Edge[q][p] > max ) **{** max = Edge[q][p]**;**  t1 = p**;**  t2 = q**; }**

p =q**;** q = disJoint[p]**;**

**}**

max = Edge[i][j]**;** k1 = i**;** k2 = j**;**

**if** ( max < Edge[s1][s2] ) **{** max = Edge[s1][s2]**;** k1 = s1**;** k2 = s2**; }**

**if** ( max < Edge[t1][t2] ) **{** max = Edge[t1][t2]**;** k1 = t1**;** k2 = t2**; }**

**if** ( max != Edge[i][j] ) **{** //当Edge[i][j] == max时边不改

**if** ( disJoint[k1] == k2 ) disJoint[k1] = -1**;**

**else** disJoint[k2] = -1**;** //删除权值最大的边

disJoint[j] = i**;** //加入新的边

Edge[j][i] = - Edge[j][i]**;**

**}**

**}**

**}**

8-15以右图为例，按Dijkstra算法计算得到的从顶点①(A)到其它各个顶点的最短路径和最短路径长度。

10

18

5

5

2

2

2

A

B

C

D

E

【解答】

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 源点 | 终点 | 最短路径 | | | | 最短路径长度 | | | |
| A | B | (A,B) | (A,B) | (A,B) | (A,B) | 10 | 10 | 10 | 10 |
|  | C | (A,C) | (A,C) | (A,C) | (A,C) | 18 | 18 | 18 | 18 |
|  | D | ⎯ | (A,B,D) | (A,B,D) | (A,B,D) | ∞ | 15 | 15 | 15 |
|  | E | ⎯ | ⎯ | (A,B,D,E) | (A,B,D,E) | ∞ | ∞ | 17 | 17 |

8-16 在以下假设下，重写Dijkstra算法：

(1) 用邻接表表示带权有向图G，其中每个边结点有3个域：邻接顶点vertex，边上的权值length和边链表的链接指针link。

(2) 用集合T = V(G) - S代替S (已找到最短路径的顶点集合)，利用链表来表示集合T。

试比较新算法与原来的算法，计算时间是快了还是慢了，给出定量的比较。

【解答】

1. 用邻接表表示的带权有向图的类定义：

**const int** DefaultSize = 10**;** //缺省顶点个数

**class** Graph**;**  //图的前视类定义

**struct** Edge **{** //边的定义

**friend class** Graph**;**

**int**vertex**;** //边的另一顶点位置

**float** length**;** //边上的权值

Edge \*link**;** //下一条边链指针

Edge ( ) **{ }** //构造函数

Edge ( **int** num, **float** wh ) **:** vertex (num), length (wh), link (NULL) **{ }** //构造函数

**int operator** < ( **const** Edge **&** E ) **const { return** length != E.length**;** **}** //判边上权值小否

**}**

**struct** Vertex **{** //顶点的定义

**friend class** Graph**;**

**char** data**;** //顶点的名字

Edge \*adj**;** //边链表的头指针

**}**

**class** Graph **{**  //图的类定义

**private:**

Vertex \*NodeTable**;** //顶点表 (各边链表的头结点)

**int** NumVertices**;** //当前顶点个数

**int** NumEdges**;** //当前边数

**int** GetVertexPos ( **const Type** vertex )**;** //给出顶点vertex在图中的位置

**public:**

Graph ( **int** sz )**;** //构造函数

~Graph ( )**;** //析构函数

**int** NumberOfVertices ( ) **{ return** NumVertices**;** **}** //返回图的顶点数

**int** NumberOfEdges ( ) **{ return** NumEdges**;** **}** //返回图的边数

**char** GetValue ( **int** i ) //取位置为i的顶点中的值

**{ return** i >= 0 **&&** i < NumVertices **?** NodeTable[i].data **:** ‘ ’**;** **}**

**float** GetWeight ( **int** v1, **int** v2 )**;** //返回边(v1, v2)上的权值

**int** GetFirstNeighbor ( **int** v )**;** //取顶点v的第一个邻接顶点

**int** GetNextNeighbor ( **int** v, **int** w )**;** //取顶点v的邻接顶点w的下一个邻接顶点

**}**

(2) 用集合T = V(G) - S代替S (已找到最短路径的顶点集合)，利用链表来表示集合T。

集合T用有序链表表示，数据域为顶点序号，链表T中的顶点都是未找到最短路径的顶点。另外设置一个数组S，其作用是记录已找到的顶点0到其他各顶点的最短路径path及最短路径长度len。

算法的主要思路是：

1. 对数组S及链表T初始化，记录顶点0到各个顶点的初始最短路径及其长度；
2. 扫描链表T，寻找链表T中各个顶点到顶点0的当前最短路径中长度最小者，记为u；
3. 在邻接表中扫描第u个顶点的出边表，确定每一边的邻接顶点号k。

若顶点k的最短路径没有选中过，比较绕过顶点u到顶点k的路径长度和原来顶点0到顶点k的最短路径长度，取其小者作为从顶点0到顶点k的新的最短路径。

1. 重复执行②、③步，直到图中所有顶点的最短路径长度都已选定为止。

算法的实现如下：

**const float** MaxNum = 10000000**;**

**typedef struct** info **{** //辅助数组元素: 各顶点最短路径信息

**int** pre**;** //在最短路径上前一顶点的顶点序号

**float** len**;** //当前最短路径长度

**}**

info S [NumVertices]**;** //辅助数组: 最短路径数组

List<**int**> T**;** //未选定最短路径顶点链表

**int** i, k, u**;** ListNode<**int**> \* p**;**

T. MakeEmpty ()**;**

**for** ( i = 1**;** i < NumVertices**;** i++ ) **{**

S[i].pre = 0**;** S[i].len = MaxNum**;** //辅助数组初始化

T.Locate ( i )**;** T.Insert( i )**;** //形成有序链表T

**}**

p = NodeTable[0].adj**;**

**while** ( p != NULL )

**{** S[p->vertex].len = p->length**;** p = p->link**; }**

**while** (1) **{**

T.First ()**;** //循环检测链表T

**if** ( ! T.NextNotNull( ) ) **break;** //链表仅剩一个顶点, 跳出循环, 算法结束

**float** min = MaxNum**;** u = 0**;**

**while** ( T.NotNull( ) ) **{** //链表不空, 还有剩余顶点未确定最短路径

i = T.GetData( )**;**  //取剩余顶点号

**if** ( S[i].len < min ) **{** min = S[i].len**;** u = i**; }** //比较, 寻找最短路径长度结点u

T.next ( )**;**

**}**

p = NodeTable[u].adj**;**

**while** ( p != NULL ) **{** //比较绕过顶点u 到其他顶点k的路径长度

k = p->vertex**;** //顶点k在链表T中表示该顶点未最终选定最短路径

**if** ( T.Find(k) != NULL **&&** S[u].len + p->length < S[k].len )

**{** s[k].len = S[u].len + p->length**;** S[k].pre = u**; }** //修改

p = p->link**;**

**}**

T.Find(u)**;** T.Remove()**;**  //在链表T中删除顶点u

**}**

8-17 试证明：对于一个无向图G = (V, E)，若G中各顶点的度均大于或等于2，则G中必有回路。

【解答】

反证法：对于一个无向图G=（V，E），若G中各顶点的度均大于或等于2，则G中没有回路。此时从某一个顶点出发，应能按拓扑有序的顺序遍历图中所有顶点。但当遍历到该顶点的另一邻接顶点时，又可能回到该顶点，没有回路的假设不成立。

8-18 设有一个有向图存储在邻接表中。试设计一个算法，按深度优先搜索策略对其进行拓扑排序。并以右图为例检验你的算法的正确性。

①

②

③

④

⑤

【解答】

(1) 利用题8-16定义的邻接表结构。

增加两个辅助数组和一个工作变量：

⬩ 记录各顶点入度 **int** indegree[NumVertices]。

⬩ 记录各顶点访问顺序 **int** visited[NumVertices]，初始时让visited[i] = 0, i = 1, 2, …, NumVertices。

⬩ 访问计数**int** count，初始时为0。

(2) 拓扑排序算法

**void** Graph **::** dfs ( **int** visited[ ], **int** indegree[ ], **int** v, **int** **&** count ) **{**

count++**;**  visited[v] = count**;**

**cout <<** NodeTable[v].data << **endl;**

Edge \*p = NodeTable[v].adj**;**

**while** ( p != NULL ) **{**

**int** w = p->vertex**;**

indegree[w]--**;**

**if** ( visited[w] == 0 **&&** indegree[w] == 0 ) dfs ( visited, indegree, w, count )**;**

p = p->link**;**

**}**

**}**

主程序

**int** i, j**;**  Edge \*p**; float** w**;**

**cin** >> NumVertices**;**

**int** \* visited = **new int**[NumVertices+1]**;**

**int** \* indegree **= new int**[NumVertices+1]**;**

**for** ( i = 1**;** i <= NumVertices**;** i++ ) **{**

NodeTable[i].adj = NULL**; cin** >> NodeTable[i].data**; cout << endl;**

visited[i] = 0**;** indegree[i] = 0**;**

**}**

**int** count = 0**;**

**cin** >> i >> j >> w**; cout << endl;**

**while** ( i != 0 **&&** j != 0 ) **{**

p = **new** Edge ( j, w )**;**

**if** ( p == NULL ) **{ cout** << “存储分配失败！” << **endl; exit**(1)**; }**

indegree[j]++**;**

p->link = NodeTable[i].adj**;** NodeTable[i].adj = p**;**

NumEdges++**;**

**cin** >> i >> j >> w**; cout << endl;**

**}**

**for** (i = 1**;** i <= NumVertices**;** i++ )

**if** ( visited[i] == 0 **&&** indegree[i] == 0 )dfs ( visited, indegree, i, count )**;**

**if** (count < NumVertices ) **cout** << “排序失败!” << **endl;**

**else cout** << “排序成功!” << **endl;**

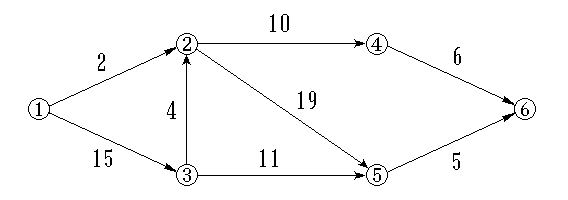
**delete** [ ] visited**; delete** [ ] indegree**;**

8-19 试对下图所示的AOE网络，解答下列问题。

(1) 这个工程最早可能在什么时间结束。

(2) 求每个事件的最早开始时间Ve[i]和最迟开始时间Vl[i]。

(3) 求每个活动的最早开始时间e( )和最迟开始时间l( )。

 (4) 确定哪些活动是关键活动。画出由所有关键活动构成的图，指出哪些活动加速可使整个工程提前完成。

【解答】

按拓扑有序的顺序计算各个顶点的最早可能开始时间Ve和最迟允许开始时间Vl。然后再计算各个活动的最早可能开始时间e和最迟允许开始时间l，根据l - e = 0? 来确定关键活动，从而确定关键路径。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| Ve | 0 | 19 | 15 | 29 | 38 | 43 |
| Vl | 0 | 19 | 15 | 37 | 38 | 43 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | <1, 2> | <1, 3> | <3, 2> | <2, 4> | <2, 5> | <3, 5> | <4, 6> | <5, 6> |
| e | 0 | 0 | 15 | 19 | 19 | 15 | 29 | 38 |
| l | 17 | 0 | 15 | 27 | 19 | 27 | 37 | 38 |
| l-e | 17 | 0 | 0 | 8 | 0 | 12 | 8 | 0 |

此工程最早完成时间为43。关键路径为<1, 3><3, 2><2, 5><5, 6>

8-20 若AOE网络的每一项活动都是关键活动。令G是将该网络的边去掉方向和权后得到的无向图。

(1) 如果图中有一条边处于从开始顶点到完成顶点的每一条路径上，则仅加速该边表示的活动就能减少整个工程的工期。这样的边称为桥(bridge)。证明若从连通图中删去桥，将把图分割成两个连通分量。

(2) 编写一个时间复杂度为O(n+e)的使用邻接表表示的算法，判断连通图G中是否有桥，若有。输出这样的桥。

【解答】

(1) 反证法（略）

1. 借助于求重连通分量的算法。如果一条边的两个端点满足下列条件之一，即为桥：

① 两个端点都是关节点；

② 一个端点是关节点，另一端点是整个图的开始点；

③ 一个端点是关节点，另一端点是整个图的完成点。

**四、其他练习题**

8-21 判断题

(1) 用邻接矩阵存储一个图时，在不考虑压缩存储的情况下，所占用的存储空间大小只与图中的顶点个数有关，而与图的边数无关。

(2) 邻接表只能用于有向图的存储，邻接矩阵对于有向图和无向图的存储都适用。

(3) 邻接矩阵只适用于稠密图（边数接近于顶点数的平方），邻接表适用于稀疏图（边数远小于顶点数的平方）。

(4) 有n (n≥1) 个顶点的无向连通图最少有n-1条边。

(5) 有n (n≥1) 个顶点的有向强连通图最少有n条边。

(6) 存储无向图的邻接矩阵是对称的，因此只要存储邻接矩阵的下（上）三角部分就可以了。

(7) 连通分量是无向图中的极小连通子图。

(8) 强连通分量是有向图中的极大强连通子图。

(9) 对任何用顶点表示活动的网络（AOV网）进行拓扑排序的结果都是唯一的。

(10) 有回路的有向图不能完成拓扑排序。

(11) 在AOE网络中一定只有一条关键路径。

(12) 关键活动不按期完成就会影响整个工程的完成时间。

(13) 任何一个关键活动提前完成, 那么整个工程将会提前完成。

(14) 所有的关键活动都提前完成, 那么整个工程将会提前完成。

(15) 任何一个关键活动延迟，那么整个工程将会延迟。

【解答】

(1) √ (2) × (3) √ (4) √ (5) √ (6) √ (7) × (8) √

(9) × (10) √ (11) × (12) √ (13) × (14) √ (15) √

8-22 填空题

(1) 在一个无向图中，所有顶点的度数之和等于所有边数的\_\_\_\_\_\_\_\_倍。

(2) 在一个具有n个顶点的无向完全图中，包含有\_\_\_\_\_\_\_\_条边，在一个具有n个顶点的有向完全图中，包含有\_\_\_\_\_\_\_\_条边。

(3) 在一个具有n个顶点的无向图中，要连通所有顶点则至少需要\_\_\_\_\_\_\_\_条边。

(4) 表示图的三种存储结构为\_\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_。

(5) 对于一个具有n个顶点的图，若采用邻接矩阵表示，则矩阵大小为\_\_\_\_\_\_\_\_。

(6) 对于一个具有n个顶点和e条边的有向图和无向图，在其对应的邻接表中，所含边结点分别为\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_条。

(7) 在有向图的邻接表和逆邻接表表示中，每个顶点的边链表中分别链接着该顶点的所有\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_结点。

(8) 对于一个具有n个顶点和e条边的有向图和无向图，若采用邻接多重表表示，则存于顶点表中的边链表指针分别有\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_个，所有边结点有\_\_\_\_\_\_\_\_个。

(9) 对于一个具有n个顶点和e条边的无向图，当分别采用邻接矩阵、邻接表和邻接多重表表示时，求任一顶点度数的时间复杂度依次为\_\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_。

(10) 假定一个图具有n个顶点和e条边，则采用邻接矩阵、邻接表和邻接多重表表示时，其相应的空间复杂度分别为\_\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_。

(11) 对用邻接矩阵表示的图进行任一种遍历时，其时间复杂度为\_\_\_\_\_\_\_\_，对用邻接表表示的图进行任一种遍历时，其时间复杂度为\_\_\_\_\_\_\_\_。

(12) 对于一个具有n个顶点和e条边的连通图，其生成树中的顶点数和边数分别为\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_。

【解答】

(1) 2 (2) n(n-1)/2, n(n-1)

(3) n-1 (4) 邻接矩阵, 邻接表, 邻接多重表

(5) n2 (6) e, 2e

(7) 出边, 入边 (8) 2n, n, e

(9) O(n), O(e/n), O(e) (10) O(n2), O(n+e), O(n+e)

(11) O(n2), O(e) (12) n, n-1

8-23 从供选择的答案中选择与下面有关图的叙述中各括号相匹配的词句，将其编号填入相应的括号内。

(1) 采用邻接表存储的图的深度优先遍历算法类似于二叉树的（ A ）。

(2) 采用邻接表存储的图的广度优先遍历算法类似于二叉树的（ B ）。

(3) 对于含有n个顶点和e条边的无向连通图，利用prim算法产生最小生成树，其时间复杂性为（ C ），利用Kruskal算法产生最小生成树，其时间复杂性为（ D ）。

(4) 设图中有n个顶点和e条边，进行深度优先搜索的时间复杂度至多为（ E ），进行广度优先搜索的时间复杂性至多为（ F ）。

(5) 对于一个具有n个顶点和e条边的无向图，进行拓扑排序时，总的时间为（ G ）。

(4) 判断有向图是否存在回路，除了可以利用拓扑排序方法外，还可以利用（ H ）。

供选择的答案

A, B：① 中序遍历 ② 前序遍历 ③ 后序遍历 ④ 按层次遍历

C, D：① O(n2) ② O(n\*e) ③ O(nlog2n) ④ O(elog2e)

E, F：① O(n+e) ② O(n\*e) ③ O(nlog2n) ④ O(elog2e)

G： ① n ② n+1 ③ n-1 ④ n+e

H： ① 求关键路径的方法 ② 求最短路径的Dijkstra方法

③ 深度优先遍历算法 ④ 广度优先遍历算法

【解答】

A．②B．④C．① D．④ E．① F．① G．① H．③

8-24 若设一个图采用邻接矩阵表示，试编写进行深度优先搜索的非递归算法。

【解答】

邻接矩阵采用教材中的定义。在相应的深度优先搜索的非递归算法中，同样使用了一个辅助数组int visited[ ]，在visited[i]中记忆第i个顶点是否访问过。另外，算法中使用了一个栈S，记忆回退的路径。具体算法实现如下：

#**include** <iostream.h>

**#include** “stack.h”

**template <class Type> void** Graph**<Type> : :** dfss ( **int** v ) **{**

//从顶点v开始进行深度优先搜索，一次遍历一个连通分量的所有顶点

**int** i, j, k**;**  Stack <**int**> S**;** S.Push ( v )**;**

**int** vosited[MaxNumVertices]**;**

**for** (i = 0**;** i < MaxNumVertices**;** i++ ) visited[i] = 0**;**

**while** ( !S.IsEmpty ( ) ) **{**

k = S.GetTop( )**;** S.Pop ( )**; //**栈中退出一个顶点

**if** ( !visited[k] ) **{ //**若未访问过

**cout** << k << ' '**;** visited[k] = 1**;** //访问，并作访问标记

**for** ( j = MaxNumVertices-1**;** j >= 0; j*--* ) //检查k的所有邻接顶点

**if** ( k != j **&&** Edge[k][j] != MaxValue ) //所有邻接顶点进栈

S.Push ( j )**;**

**}**

**}**

**}**

**}**

8-25 应用Prim算法求解连通网络的最小生成树问题。

(1) 针对右图所示的连通网络，试按如下格式给出在构造最小生成树过程中顺序选出的各条边。

0

1

2

3

4

5

6

5

1

5

7

5

3

4

2

6

( 始顶点号，终顶点号， 权值 )

( , , )

( , , )

( , , )

( , , )

( , , )

(2) 下面是Prim算法的实现，中间有5个地方缺失，请阅读程序后将它们补上。

**const int** MaxInt = INT\_MAX**;** //INT\_MAX的值在<limits.h>中

**const int** n = 6**;** //图的顶点数, 应由用户定义

**typedef int** AdjMatrix[n][n]**;** //用二维数组作为邻接矩阵表示

**typedef struct {** //生成树的边结点

**int** fromVex**,** toVex**;** //边的起点与终点

**int** weight**;** //边上的权值

**}** TreeEdgeNode**;**

**typedef** TreeEdgeNode MST[n-1]**;**  //最小生成树定义

**void** PrimMST ( AdjMatrix G**,** MST T, **int** rt ) **{**

//从顶点rt出发构造图G的最小生成树T，rt成为树的根结点

TreeEdgeNode e**; int** i, k = 0, min, minpos, v**;**

**for** ( i = 0**;** i < n**;** i++ ) //初始化最小生成树T

**if** ( i != rt ) **{**

T[k].fromVex = rt**;**

**;**

①

T[k++].weight = G[rt][i]**;**

**}**

**for** ( k = 0**;** k < n-1**;** k++ ) { //依次求MST的候选边

**;**

②

**for** ( i = k**;** i < n-1**;** i++ ) //遍历当前候选边集合

**if** ( T[i].weight < min ) //选具有最小权值的候选边

③

**{** min = T[i].weight**;** **; }**

**if** ( min *==* MaxInt ) //图不连通, 出错处理

④

**{ cerr <<** “Graph is disconnected!” << **endl;**  **; }**

e = T[minpos]**;** T[minpos] = T[k] **;** T[k] = e**;**

v = T[k].toVex**;**

**for** ( i = k+1**;** i < n-1**;** i++ ) //修改候选边集合

if ( G[v][T[i].toVex] < T[i].weight ) **{**

T[i].weight = G[v][T[i].toVex]**;**

**;**

⑤

**}**

**}**

**}**

【解答】

(1) 选出的边顺序为: 【括号中的内容为：（始顶点号，终顶点号，权值）】

(0, 3, 1)，(3, 5, 4)，(5, 2, 2)，(3, 4, 5) 或 (3, 1, 5)，(4, 1, 3)，(1, 4, 3 )

(2) 应填入的语句是：① T[k].toVex = i**;** ② min = MaxInt**;** ③ minpos = i**;**

④ exit (1)**;** ⑤ T[i].fromVex = v**;**

8-26 给定n个小区之间的交通图。若小区i与小区j之间有路可通，则将顶点i与顶点j之间用边连接，边上的权值wij表示这条道路的长度。现在打算在这n个小区中选定一个小区建一所医院，试问这家医院应建在哪个小区，才能使距离医院最远的小区到医院的路程最短？试设计一个算法解决上述问题。

【解答】

将n个小区的交通图视为带权无向图，并利用邻接矩阵来存放带权无向图。算法的思想是：

(1) 应用Floyed算法计算每对顶点之间的最短路径；

(2) 找出从每一个顶点到其他各顶点的最短路径中最长的路径；

(3) 在这n条最长路径中找出最短的一条，则它的出发点即为所求。

算法描述如下：

**const int** MaxInt = INT\_MAX**;** //INT\_MAX的值在<limits.h>中

**const int** n = 6**;** //图的顶点数, 应由用户定义

**typedef int** AdjMatrix[n][n]**;** //用二维数组作为邻接矩阵表示

**int** distance ( AdjMatrix A**, int&** min ) **{**

**int** i, j, k, s**;**

**for** ( k = 0**;** k < n**;** k++ ) //计算每对顶点之间的最短路径

**for** ( i = 0**;** i < n**;** i++ )

**for** ( j = 0**;** j < n**;** j++ )

**if** ( A[i][k] + A[k][j] < A[i][j] ) A[i][j] = A[i][k] + A[k][j]**;**

k = 0**;** min = MaxInt**;** //求各最长路径中最短的一条

**for** ( i = 0**;** i < n**;** i++ ) **{**

s = 0**;**

**for** ( j = 0**;** j < n**;** j++ )

**if** ( A[i][j] > s ) s = A[i][j]**;** //求顶点i到其他顶点最短路径中的长者

**if** ( s < min ) **{** k = i**;** min = s**; }** //在各最长路径中选最短者

**}**

**return** k**;**

**}**

8-27 画出下图所示的AOV网的所有拓扑有序序列。

A

B

C

D

E

F

【解答】

共有8种： A D B E C F A D B E F C A D E B C F A D E B F C

D A B E C F D A B E F C D A E B C F D A E B F C