图论(下)

七月算法 邹博

2015年4月30日

主要内容

- 继续关于深度优先搜索的讨论
  - 以某例题来观察DFS和动态规划的关系
- 最短路径
  - 单源点最短路径
  - 任意两点间的最短路径
  - 边存在负权
- 最小生成树
  - Prim算法
  - Kruskal算法
- 拓扑排序
  - 深化“拓扑”的概念

深度优先搜索DFS

- 理念：
  - 不断深入，“走到头”回退。(回溯思想)
- 一般所谓“暴力枚举”搜索都是指DFS
  - 回忆数组章节中“N-Sum问题”的解法
- 实现
  - 一般使用堆栈，或者递归
- 用途：
  - DFS的过程中，能够获得的信息
    - “时间戳”、“颜色”、父子关系、高度

回文划分问题Palindrome Partitioning

- 给定一个字符串str，将str划分成若干子串，使得每一个子串都是回文串。计算str的所有可能的划分。
  - 单个字符构成的字符串，显然是回文串；所以，这个的划分一定是存在的。
- 如：s=“aab”，返回
  - “aa”，“b”；
  - “a”，“a”，“b”。

回文划分问题Palindrome Partitioning

- 分析：
  - 在每一步都可以判断中间结果是否为合法结果：回溯法——如果某一次发现划分不合法，立刻对该分支限界。
  - 事实上，一个长度为n的字符串，有n-1个位置可以截断，每个位置有两种选择，因此时间复杂度为$O(2^{n-1})=O(2^n)$。

Code

```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;  // 一个 partition 方案
        DFS(s, path, result, 0);
        return result;
    }
    // 搜索必须以 s[start] 开头的 partition 方案
    void DFS(string &s, vector<string>& path,
            vector<vector<string>> &result, int start) {
        if (start == s.size()) {
            result.push_back(path);
            return;
        }
        for (int i = start; i < s.size(); i++) {
            if (isPalindrome(s, start, i)) { // 从 i 位置砍一刀
                path.push_back(s.substr(start, i - start + 1));
                DFS(s, path, result, i + 1);  // 继续往下砍
                path.pop_back(); // 撤销上上行
            }
        }
    }
    bool isPalindrome(const string &s, int start, int end) {
        while (start < end && s[start] == s[end]) {
            ++start;
            --end;
        }
        return start >= end;
    }
};
```

Palindrome Partitioning思考

- 与之类似的：
  - 给定仅包含数字的字符串，返回所有可能的有效IP地址组合。如：“25525511135”，返回“255.255.11.135”，“255.255.111.35”。
  - 该问题只插入3个分割位置。
  - 只有添加了第3个分割符后，才能判断当前划分是否合法。
    - 如：2.5.5.25511135，才能判断出是非法的。
      - 当然，它可以通过“25511135”大于“255.255”等其他限界条件“事先”判断。

Palindrome Partitioning思考：动态规划

- 如果已知：
- str[i+1,i+2…n-1]的所有回文划分ϕ(i+1)，
  - 注：ϕ是个集合，存储了所有可能的回文划分，下同
- str[i+2,i+3…n-1]的所有回文划分ϕ(i+2)，
- str[i+3,i+4…n-1]的所有回文划分ϕ(i+3)，
- ……
- str[n-1]的所有回文划分ϕ(n-1)，
- 如何求str[i,i+1…n-1]的所有划分呢？
  - 分析：如果已知以str[i]为起点的子串中str[i,i+1…j]是回文串，那么，该子串添加到ϕ(j)中，即为一种可能的划分方案。
- 算法：
  - 将集合ϕ(i)置空；
  - 遍历j(i≤j≤n-1)，若str[i,i+1…j]是回文串，则将{str[i,i+1…j]， ϕ(j)}添加到ϕ(i)中；
  - i从n-1到0，依次调用上面两步，最终返回ϕ(0)即为所求。

动态规划解回文划分的Trick

- 在计算str[i,i+1…j]是否是回文串这一子问题中，暴力也是无可厚非的。线性探索：j从i到n-1遍历即可。
- 事实上，可以事先缓存所有的str[i,i+1...j]是回文串的那些记录：用二维布尔数组p[n][n]就够了：p[i][j]的true/false表示了str[i,i+1...j]是否是回文串；
- 它本身是个小的动态规划：
  - 如果已知str[i+1...j-1]是回文串，那么，判断str[i,i+1...j]是否是回文串，只需要判断str[i]==str[j]就可以。

Code

```
class Solution {
public:
    vector<vector<string> > partition(string s) {
        const int n = s.size();
        bool p[n][n]; // whether s[i,j] is palindrome
        fill_n(&p[0][0], n * n, false);
        for (int i = n - 1; i >= 0; --i)
            for (int j = i; j < n; ++j)
                p[i][j] = s[i] == s[j] && ((j - i < 2) || p[i + 1][j - 1]);

        vector<vector<string> > sub_palins[n]; // sub palindromes of s[0,i]
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i; j < n; ++j)
                if (p[i][j]) {
                    const string palindrome = s.substr(i, j - i + 1);
                    if (j + 1 < n) {
                        for (auto v : sub_palins[j + 1]) {
                            v.insert(v.begin(), palindrome);
                            sub_palins[i].push_back(v);
                        }
                    } else {
                        sub_palins[i].push_back(vector<string> { palindrome });
                    }
                }
        }
        return sub_palins[0];
    }
};
```

回文子串问题的DFS与DP深刻认识

- DFS的过程，是计算完成了str[0…i]的切分，然后递归调用，继续计算str[i+1,i+2…n-1]的过程；
- 而DP中，故意使用了从后向前的方法：假定得到了str[i+1,i+2…n-1]的所有可能切分方案，如何扩展得到str[i,i+1…n-1]的切分；
  - 当然可以先计算str[0…i]的所有可能切分，然后考察str[0…i+1]的所有切分。
- 从本质上说，二者是等价的：最终都搜索了一颗隐式树。
  - 当然，如果题目要求“切分成最少数目的回文子串”，动态规划就有优势了——事实上是快速删减了不可能的子树分支。

附： Palindrome Partitioning II

- 给定一个字符串str，将str划分成若干子串，使得每一个子串都是回文串。在所有可能的划分中，子串数目最少是多少？
  - 如：s=“aab”，最少子串数目为2
    - “aa”，“b”
  - 注：原leetcode题目是返回切分的数目
    - “aab”返回1：切1刀。

- 设d[i]为字符0~i划分的最小回文串的个数，则d[i]=min{d[j]+1 | s[j+1~i]是回文串}。

Palindrome Partitioning II

- 假设当前已经计算完成前缀串str[0…i-1]的最少划分数目d[0…i-1]，其中d[j]表示前缀串str[0…j]的最少划分数目；
- 则：要划分前缀串str[0…i]，假定划分方案的最后一个子串为str[j+1…i]，显然，根据题意str[j+1…i]为回文串，并且str[0…j]一定是最少划分，即：
  - d[i]=min{d[j]+1|s[j+1…i]是回文串，i-1≥j≥0}
- 注意：
  - str[j…i]是否为回文串，本身是个小的动态规划
    - 若已知str[j+1…i-1] 是回文串，则只需判断str[j]==str[i]；
  - 上述整个过程从后向前分析仍然可以得到类似的结论；

Code

```
class Solution {
public:
    int minCut(string s) {
        const int n = s.size();
        int f[n+1];
        bool p[n][n];
        fill_n(&p[0][0], n * n, false);
        //the worst case is cutting by each char
        for (int i = 0; i <= n; i++)
            f[i] = n - 1 - i; // 最后一个 f[n]=-1
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i; j < n; j++) {
                if (s[i] == s[j] && (j - i < 2 || p[i + 1][j - 1])){
                    p[i][j] = true;
                    f[i] = min(f[i], f[j + 1] + 1);
                }
            }
        }
        return f[0];
    }
};
```

再谈LCA：Tarjan算法

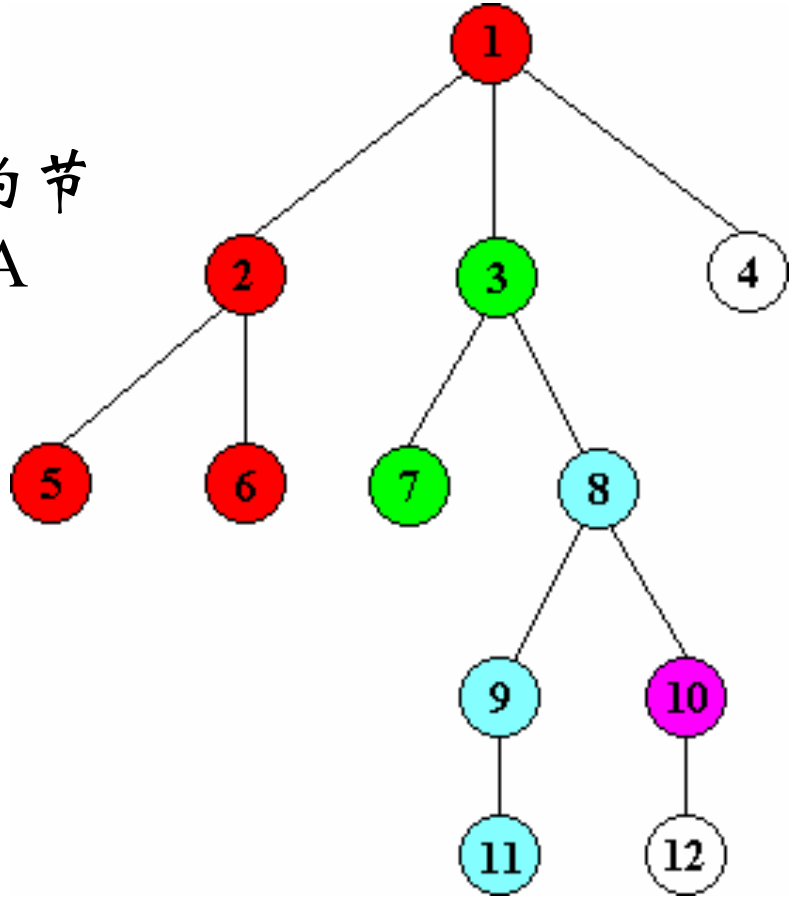
- Tarjan算法是由Robert Tarjan在1979年发现的一种高效的离线算法，也就是说，它要首先读入所有的询问(求一次LCA叫做一次询问)，然后并不一定按照原来的顺序处理这些询问，该算法的时间复杂度O(N*α(N)+Q)，其中，α(x)不大于4，N表示问题规模，Q表示询问次数。

Tarjan概览

- Tarjan算法基于深度优先搜索，对于新搜索到的一个结点u，首先创建由这个结点u构成的集合setU，再对当前结点u的每一个子树subTree进行搜索，每搜索完一棵子树sub，则可确定子树sub内的LCA询问都已解决。其他的LCA询问的结果必然在这个子树sub之外，这时把子树sub所形成的集合setSub与当前结点的集合setU合并成set，并将当前结点u设为这个集合set的祖先Ua。之后继续搜索下一棵子树subNext，直到当前结点u的所有子树搜索完。这时把当前结点u设为checked，同时可以处理有关当前结点u的LCA询问，如果有一个从当前结点u到结点v的询问，且v已被检查过，则由于进行的是深度优先搜索，当前结点u与v的最近公共祖先LCA一定是未checked，而这个最近公共祖先LCA包含v的子树subV一定已经搜索过了，那么LCA一定是v所在集合的祖先Va。

Tarjan算法：深度优先

- 关键：
  - 计算当前正要被回溯的节点10和其他结点的LCA
  - (2,10)
  - (10,7)
  - (9,10)
  - (10,8)

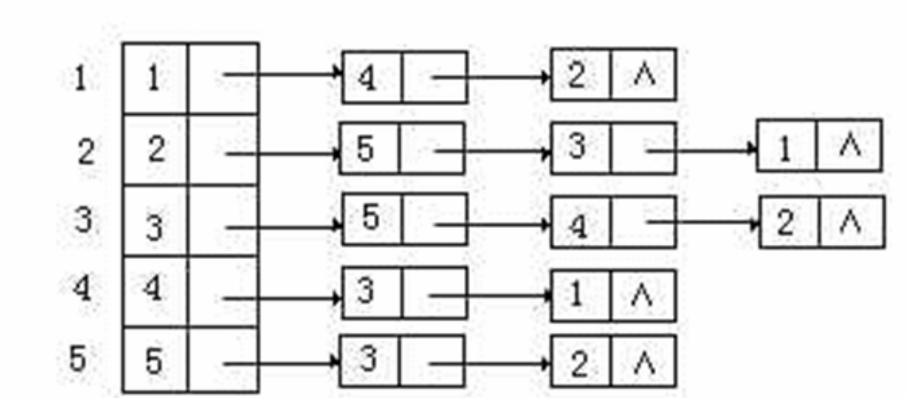


Tarjan Code

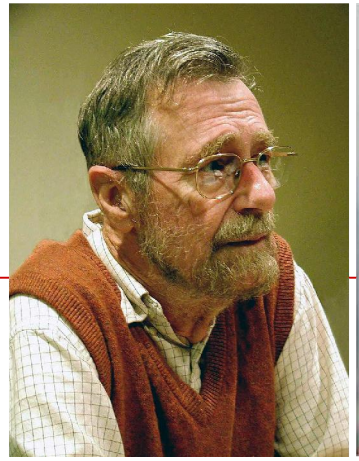
```
function TarjanOLCA(u)
  MakeSet(u);
  u.ancestor := u;    // 将集合u的祖先指向自己
  for each v in u.children do  // DFS所有孩子
     TarjanOLCA(v);
     Union(u,v);     // 与根结点U合并（并查集操作）
     Find(u).ancestor := u;    // 将u所在集合根的祖先指向u
  u.colour := black;  // 当所有孩子都已遍历，则标记根已完成
  for each v such that {u,v} in P do  // 找所有与 u相关的查询
    if v.colour == black  // 如果另一结点 v是前面标记过的,则输出递归向上返回根的祖先
       print "Tarjan's Least Common Ancestor of " + u +
            " and " + v + " is " + Find(v).ancestor + ".";
```

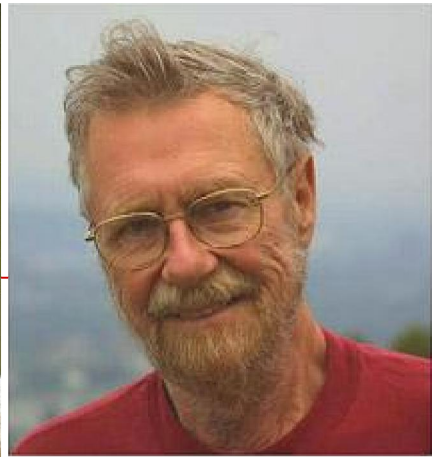
Tarjan算法的思考：如何存储候选查询

- 遍历P中的查询(a,b)，将a插入到b的列表中，b插入到a的列表中——隐式图；
- 常见的空间检索方案
  - 有点像链式Hash表：只是这里的“冲突”太多了——凡是与该结点邻接的，都在冲突链中。



Edsger Wybe Dijkstra

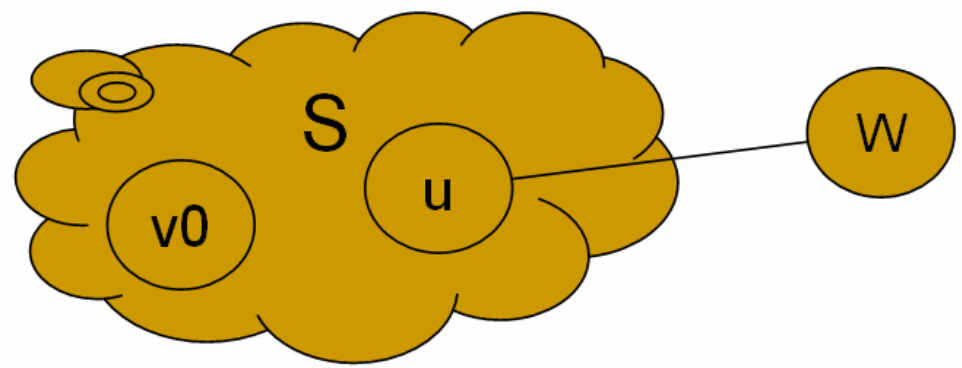




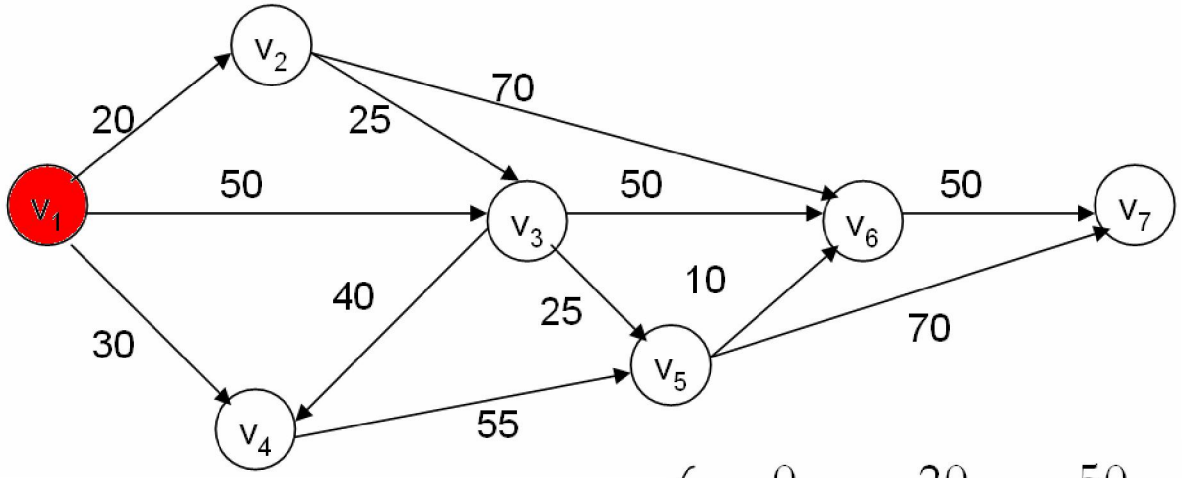
- 提出“goto有害论”;
- 提出信号量和PV原语;
- 解决了“哲学家聚餐”问题;
- 最短路径算法(SPF)和银行家算法的创造者;
- 第一个Algol 60编译器的设计者和实现者;
- THE操作系统的设计者和开发者;
  - 还有提过的“荷兰国旗问题”。

最短路径SPF：Shortest Path First(Dijkstra)

- 对于从v0至w，且经过最后一个中间结点为u的最短路径，有：
  - DIST(w) = DIST(u) + c(u,w)
- 随着u的加入， DIST(w)调整为
  - DIST(w) = min(DIST(w),DIST(u) + c(u,w))

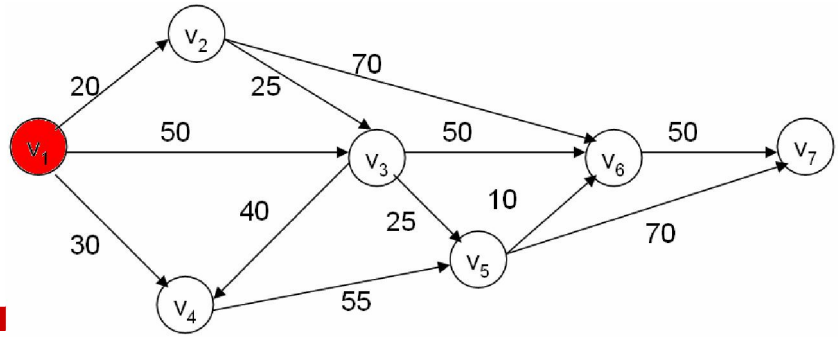


最短路径示例



$$\begin{pmatrix} 0 & 20 & 50 & 30 & +\infty & +\infty & +\infty \\ +\infty & 0 & 25 & +\infty & +\infty & 70 & +\infty \\ +\infty & +\infty & 0 & 40 & 25 & 50 & +\infty \\ +\infty & +\infty & +\infty & 0 & 55 & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty & 0 & 10 & 70 \\ +\infty & +\infty & +\infty & +\infty & +\infty & 0 & 50 \\ +\infty & +\infty & +\infty & +\infty & +\infty & +\infty & 0 \end{pmatrix}$$

最短路径示例



迭代	选取的结点	S	DIST						
			(1)	(2)	(3)	(4)	(5)	(6)	(7)
置初值	—	1	0	20	50	30	+∞	+∞	+∞
1	2	1,2	0	20	45	30	+∞	90	+∞
2	4	1,2,4	0	20	45	30	85	90	+∞
3	3	1,2,4,3	0	20	45	30	70	90	+∞
4	5	1,2,4,3,5	0	20	45	30	70	90	140
5	6	1,2,4,3,5,6	0	20	45	30	70	90	130

- 算法的执行在有n-1个结点加入到S中后终止，此时求出了v0至其它各结点的最短路径。
- 问题：如何求出所有这些最短路径？
  - 记录前驱

生成最短路径的贪心算法

```
procedure SHORTEST-PATHS(v,COST,DIST,n)
    //G是一个n结点有向图，它由其成本邻接矩阵COST(n,n)表示。DIST(j)被置
      从结点v到结点j的最短路径长度，这里1≤j≤n。特殊的，DIST(v)被置成零//
    boolean S(1:n);real COST(1:n,1:n),DIST(1:n)
    integer u,v,n,num,i,w
    for i←1 to n do //将集合S初始化为空//
        S(i) ←0;DIST(i) ←COST(v,i)  //若v到i没有边，DIST(i)=∞//
    repeat
    S(v) ←1;DIST(v) ←0  //结点v计入S//
    for num←2 to n-1 do //确定由结点v出发的n-1条路//
        选取结点u,它使得DIST(u)= min_{S(w)=0} {DIST (w)}
        S(u) ←1 //结点u计入S//
        for 所有S(w) = 0的结点w do  //修改DIST(w)//
            DIST(w) = min(DIST(w), DIST(u) + COST(u,w))
        repeat
    repeat
end SHORTEST-PATHS
```

Floyd算法

- Floyd算法又称为插点法，是一种用于寻找给定的加权图中多源点之间最短路径的算法。该算法名称以创始人之一、1978年图灵奖获得者罗伯特·弗洛伊德命名。
- 通过一个图的权值矩阵求出它的每两点间的最短路径矩阵。

算法分析

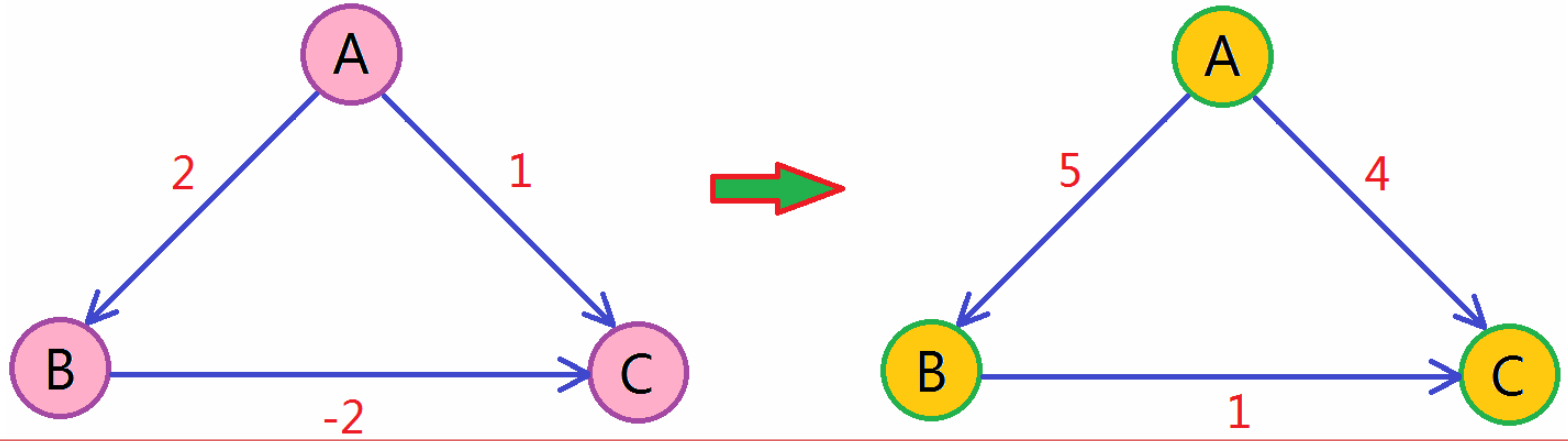
- 记录map[i,j]为结点i到结点j的最短路径的距离；则：
- map[i,j] = min {map[i,k] + map[k,j], map[i,j]}
  - k取所有结点
- 同时，map[n,n]==0
  - i,j,k各自从0到N-1，所以时间复杂度为$O(n^3)$
- 此外，如果图中存在负的权值，算法也是适用的。
  - 思考：Dijkstra算法允许边存在负权吗？

Floyd算法

```
D[u,v]=A[u,v]   //初始化
For k:=1 to n
   For i:=1 to n
      For j:=1 to n
         If D[i,j]>D[i,k]+D[k,j] Then
            D[I,j]:=D[I,k]+D[k,j];
```

带负权的最短路径

- 思考：计算图的最小权值，若该权值大于0，按照Dijkstra算法正常计算；若小于0，则所有权值都加上该权值的绝对值+1，则修改后的图不再含有负权。使用Dijkstra算法计算最小路径。



带负权的最短路径：Bellman-ford算法

- 本质：动态规划
- 适用：单源结点到其他所有结点的最短路径
- 若u->v是有向边，则d[v]<=d[u] + dis(u,v)
- 这个操作被成为松弛操作。
- 优点：对边权无要求，可以发现负环。

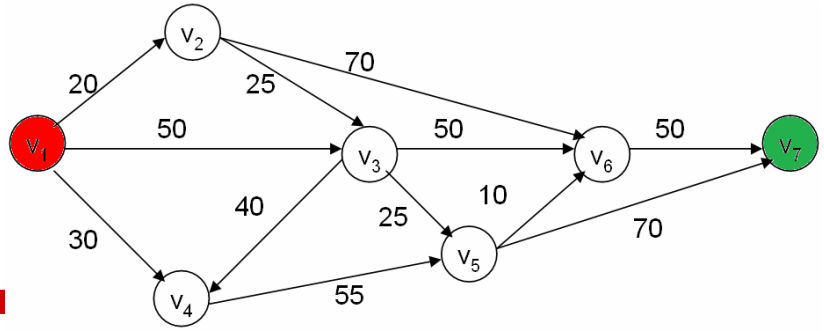
Bellman-ford算法

```
Bellman-Ford()
{
  for each vertex v ∈ G do    //初始化
       d[v] = +∞
  d[s] = 0

  for i = 1 to n-1 do
    for each edge(u, v) ∈ G do
      if d[v] > d[u] + w(u, v) then   //松弛操作
        d[v] = d[u] + w(u, v)

  for each edge(u, v) ∈ G do
     if d[v] > d[u] + w(u, v) then    //检查是否存在回路
       return false
  return true
 }
```

Bellman-ford算法分析

- 图的任意一条最短路径既不能包含负权回路，也不会包含正权回路，因此它最多包含|v|-1条边。
- 从源点s可达的所有顶点如果存在最短路径，则这些最短路径构成一个以s为根的最短路径树。Bellman-Ford算法的迭代松弛操作，实际上就是按顶点距离s的层次，逐层生成这棵最短路径树的过程。
- 在对每条边进行第1遍松弛的时候，生成了从s出发，层次至多为1的那些树枝。也就是说，找到了与s至多有1条边相联的那些顶点的最短路径；对每条边进行第2遍松弛的时候，生成了第2层次的树枝，就是说找到了经过2条边相连的那些顶点的最短路径。因为最短路径最多只包含|v|-1条边，所以，只需要循环|v|-1次。
- 略做优化：如果第k次松弛操作后，最短路径没有得到更新，显然，后面仍然无法得到更新，可提前退出。并且，如果k<n-1，一定不存在负环。

最短路径例题



- A traveler's map gives the distances between cities along the highways, together with the cost of each highway. Now you are supposed to write a program to help a traveler to decide the shortest path between his/her starting city and the destination. If such a shortest path is not unique, you are supposed to output the one with the minimum cost, which is guaranteed to be unique.
- 城市间由高速路连接，从城市A到城市B，有两个代价：高速距离、旅行价格。给定一张有向图和某个起点、终点，计算它们的最短路径：如果高速距离相同的路段，选择更小的旅行价格——旅行价格保证是唯一的。

输入输出

Sample Input: 4 5 0 3
0 1 1 20
1 3 2 30
0 3 4 10
0 2 2 20
2 3 1 20
Output: 0 2 3 3 40

- Input Specification:
  - Each input file contains one test case. Each case starts with a line containing 4 positive integers N, M, S, and D, where N (<=500) is the number of cities (and hence the cities are numbered from 0 to N-1); M is the number of highways; S and D are the starting and the destination cities, respectively. Then M lines follow, each provides the information of a highway, in the format:City1 City2 Distance Cost，where the numbers are all integers no more than 500, and are separated by a space.
- Output Specification:
  - For each test case, print in one line the cities along the shortest path from the starting point to the destination, followed by the total distance and the total cost of the path. The numbers must be separated by a space and there must be no extra space at the end of output.
- 输入文件仅包含一个测试用例。第一行为4个整数N、M、S、D，分别表示城市的数目N、高速公路的数目M、起点城市S、终点城市D。紧接着的M行格式相同，每行都是4元组：(城市1 城市2 距离 价格)
- 输出最短路径，路径后面也输出最近路径的长度和总价。

最短路径问题

- 分析
  - 题目即明确考察图中两点间最短路径的算法，因为权值为正(如果权值为0，物理意义其实是无穷大)，给定数据是每条边的权值，只不过，在使用Dijkstra算法扩展当前集合V的时候，权值是(dist,cost)双关键字：当dist相等时，再比较cost即可。

最小生成树MST

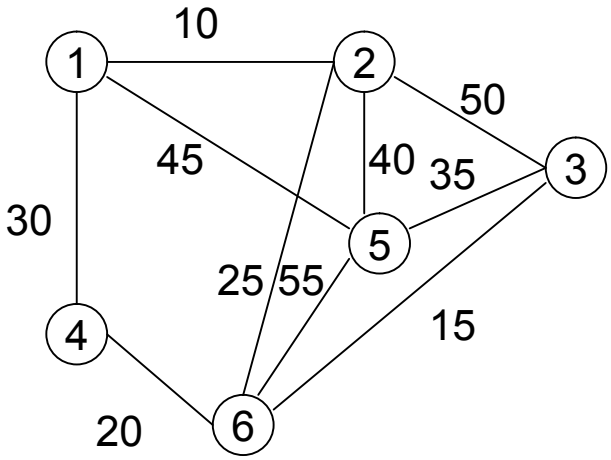
- 最小生成树要求从一个带权无向完全图中选择n－1条边并使这个图仍然连通(也即得到了一棵生成树)，同时还要考虑使树的权最小。最小生成树最著名算法是Prim算法和Kruskal算法。

Prim算法

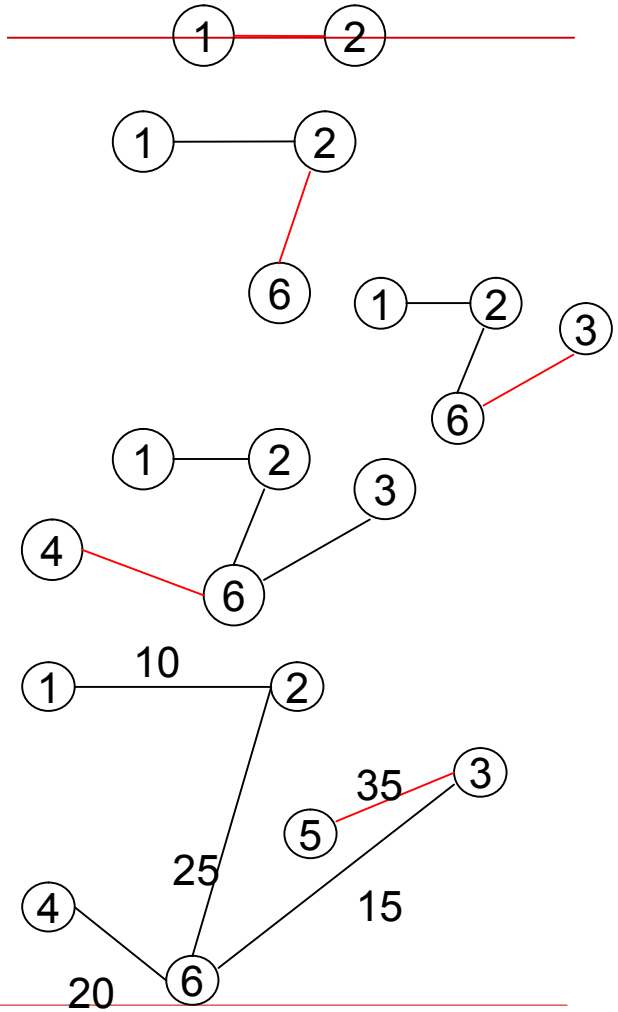
- 首先以一个结点作为最小生成树的初始结点，然后以迭代的方式找出与最小生成树中各结点权重最小边，并加入到最小生成树中。加入之后如果产生回路则跳过这条边，选择下一个结点。当所有结点都加入到最小生成树中之后，就找出了连通图中的最小生成树了。

Prim算法

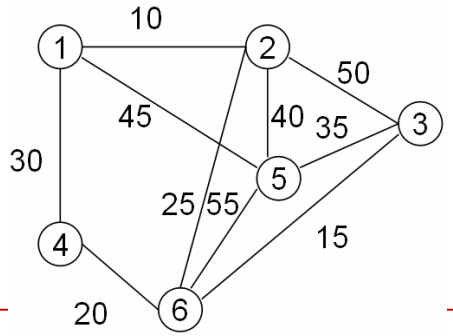
策略：使得迄今所选择的边的集合A构成一棵树；则将要计入到A中的下一条边(u,v)，应是E中一条当前不在A中且使得A∪{(u,v)}也是一棵树的最小成本边。



边	成本
(1,2)	10
(2,6)	25
(3,6)	15
(6,4)	20
(3,5)	35



Prim算法描述



- 1：初始化：U={u0},TE=∅。此步骤设立一个只有结点u0的结点集U和一个空的边集TE作为最小生成树的初始形态，在随后的算法执行中，这个形态会不断的发生变化，直到得到最小生成树为止。
- 2：在所有u∈U,v∈V－U的边(u,v)∈E中，找一条权最小的边(u0,v0)，将此边加进集合TE中，并将此边的非U中顶点加入U中。
- 3：如果U=V，则算法结束；否则重复步骤2。

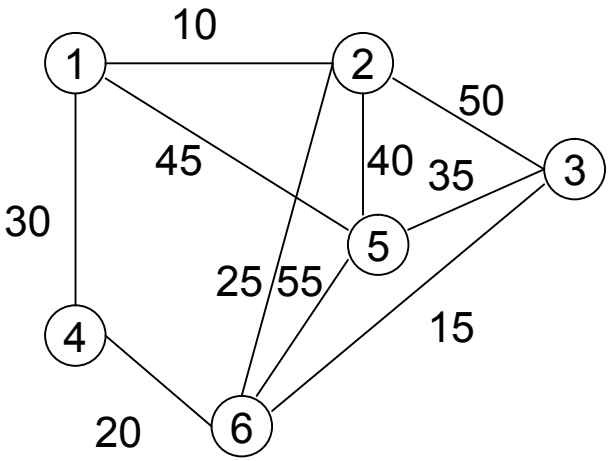
- 显然，当U=V时，步骤2共执行了n-1次(设n为图中顶点的数目)，TE中也增加了n-1条边，这n-1条边就是需要求出的最小生成树的边。

Kruskal算法

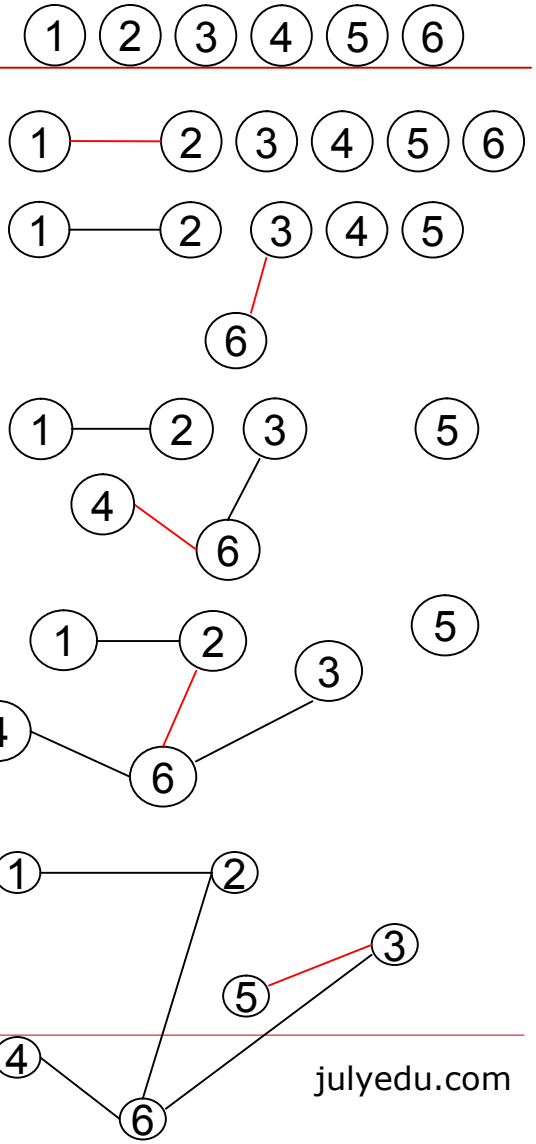
- Kruskal在找最小生成树结点之前，需要对所有权重边做从小到大排序。将排序好的权重边依次加入到最小生成树中，如果加入时产生回路就跳过这条边，加入下一条边。当所有结点都加入到最小生成树中之后，就找出了最小生成树。
  - Prim算法在得到最小生成树的过程中，始终保持是一颗树；而Kruskal算法最开始是森林，直到最后一条边加入，才得到树。

Kruskal算法

策略：图G的所有边按成本非降次序排列，下一条生成树T中的边是尚未加入树的边中具有最小成本、且和T中现有的边不会构成环路的边。



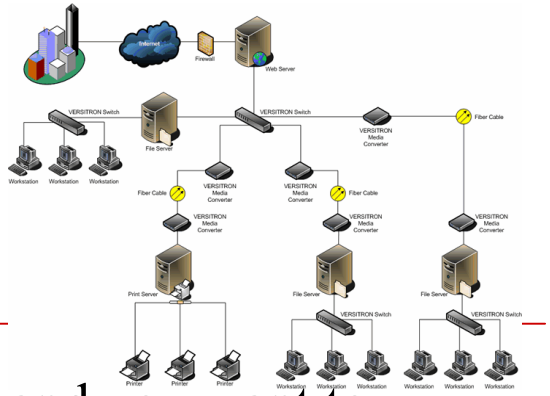
边	成本
(1,2)	10
(3,6)	15
(4,6)	20
(2,6)	25
(3,5)	35



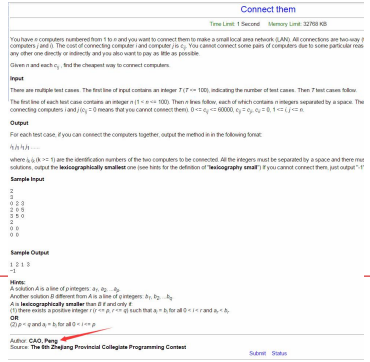
Kruskal算法几点说明

- 边集E以小顶堆的形式保存，一条当前最小成本边可以在O(loge)的时间内找到；
  - 当然，也可以用其他排序方法对边完全排序。
- 为了快速判断候选边e的加入是否会形成环，可考虑用并查集的方法：把当前状态的每个连通子图保存在各自的集合中；候选边是否可以加入，转化成边的两个顶点是否位于同一集合中；
- 算法的计算时间是O(eloge)。

最小生成树例题



- You have ***n*** computers numbered from 1 to n and you want to connect them to make a small local area network (LAN). All connections are two-way (that is connecting computers i and j is the same as connecting computers j and i). The cost of connecting computer i and computer j is c_{ij}. You cannot connect some pairs of computers due to some particular reasons. You want to connect them so that every computer connects to any other one directly or indirectly and you also want to pay as little as possible.
- Given n and each c_{ij} , find the cheapest way to connect computers. If there are multiple solutions, output the **lexicographically smallest** one。
  - Thanks to Dr. Cao Peng


Connect them

Time Limit: 1 Second Memory Limit: 32768 KB

You have n computers numbered from 1 to n and you want to connect them to make a small local area network (LAN). All connections are two-way (that is connecting computers i and j is the same as connecting computers j and i). The cost of connecting computer i and computer j is c_{ij}. You cannot connect some pairs of computers due to some particular reasons. You want to connect them so that every computer connects to any other one directly or indirectly and you also want to pay as little as possible.

Given n and each c_{ij} , find the cheapest way to connect computers.

Input

There are multiple test cases. The first line of input contains an integer T (T <= 100), indicating the number of test cases. Then T test cases follow.

The first line of each test case contains an integer n (1 < n <= 100). Then n lines follow, each of which contains n integers separated by a space. The connecting computers i and j (c_{ij} > 0 means that you cannot connect them). 0 <= c_{ij} <= 60000, c_{ij} = c_{ji}, c_{ii} = 0, 1 <= i, j <= n.

Output

For each test case, if you can connect the computers together, output the method in in the following format:

i_1 i_2 i_3 i_4 ....

where i_k i_{k+1} (k >= 1) are the identification numbers of the two computers to be connected. All the integers must be separated by a space and there must be no extra spaces at the end of the line. If there are multiple solutions, output the **lexicographically smallest** one (see hints for the definition of "**lexicography small**"). If you cannot connect them, just output "-1".

Sample Input

```
2
3
0 2 3
2 0 5
3 5 0
3
0 0 0
0 0 0
0 0 0
```

Sample Output

```
1 2 1 3
-1
```

Hints:

A solution A is a line of p integers: a_1, a_2 ... a_p.

Another solution B different from A is a line of q integers: b_1, b_2, ... b_q.

A is **lexicographically smaller** than B if and only if:

(1) there exists a positive integer r (r <= p, r <= q) such that a_r < b_r for all 0 < i < r and a_i = b_i.

OR

(2) p < q and a_i = b_i for all 0 < i <= p.

Author: CAO, Peng

Source: The 6th Zhejiang Provincial Collegiate Programming Contest

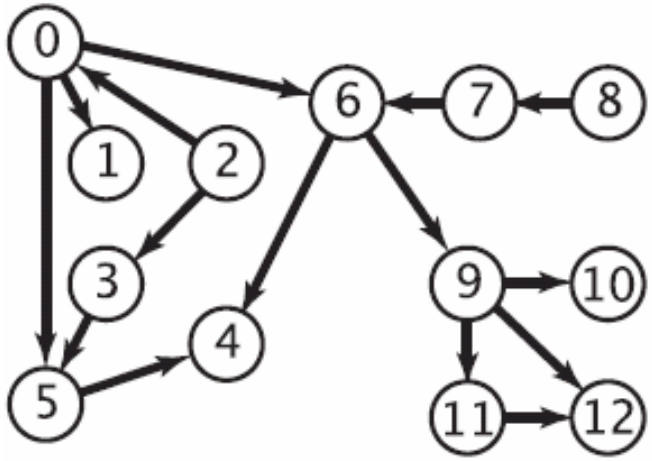
Submit Status

解题过程

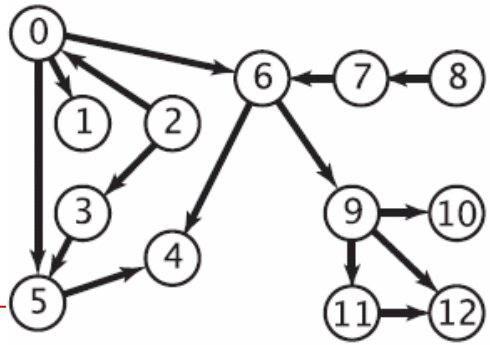
- 问题分析：题目本身即给定了n个节点的图的邻接矩阵，求该图的最小生成树。此外，如果最小生成树有多个（如果有多个，它们的权值比如都相等），输出字典序最小的那个。
  - 多解时，输出字典序最小的，才是本题的关键难点。
  - 注意：如果某个图的最小生成树有m个，则这m颗树的权值分别相等。即：如果该图的某个MST的边权值为：$e_1e_2e_3...e_x$，则其他MST的边权值必然也是$e_1e_2e_3...e_x$。
- 算法设计：
  - 使用Kruskal算法可以完成该问题，只是，对于排序的m条边$\{(v_i,v_j)\}$，如果当前要添加到MST的边权值相等，添加字典序小的那个(v_i,v_j)。
  - 完成MST后，再用快排等排序算法，将MST的边按照字典序得到从小到大的序列。

拓扑排序

- 对一个有向无环图(Directed Acyclic Graph，DAG)G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边(u,v)∈E(G)，则u在线性序列中出现在v之前。
- 一种可能的拓扑排序结果
  2->8->0->3->7->1->5->6
  ->9->4->11->10->12



拓扑排序的方法



- 从有向图中选择一个没有前驱(即入度为0)的顶点并且输出它；
- 从网中删去该顶点，并且删去从该顶点发出的全部有向边；
- 重复上述两步，直到剩余的网中不再存在没有前趋的顶点为止。

Code

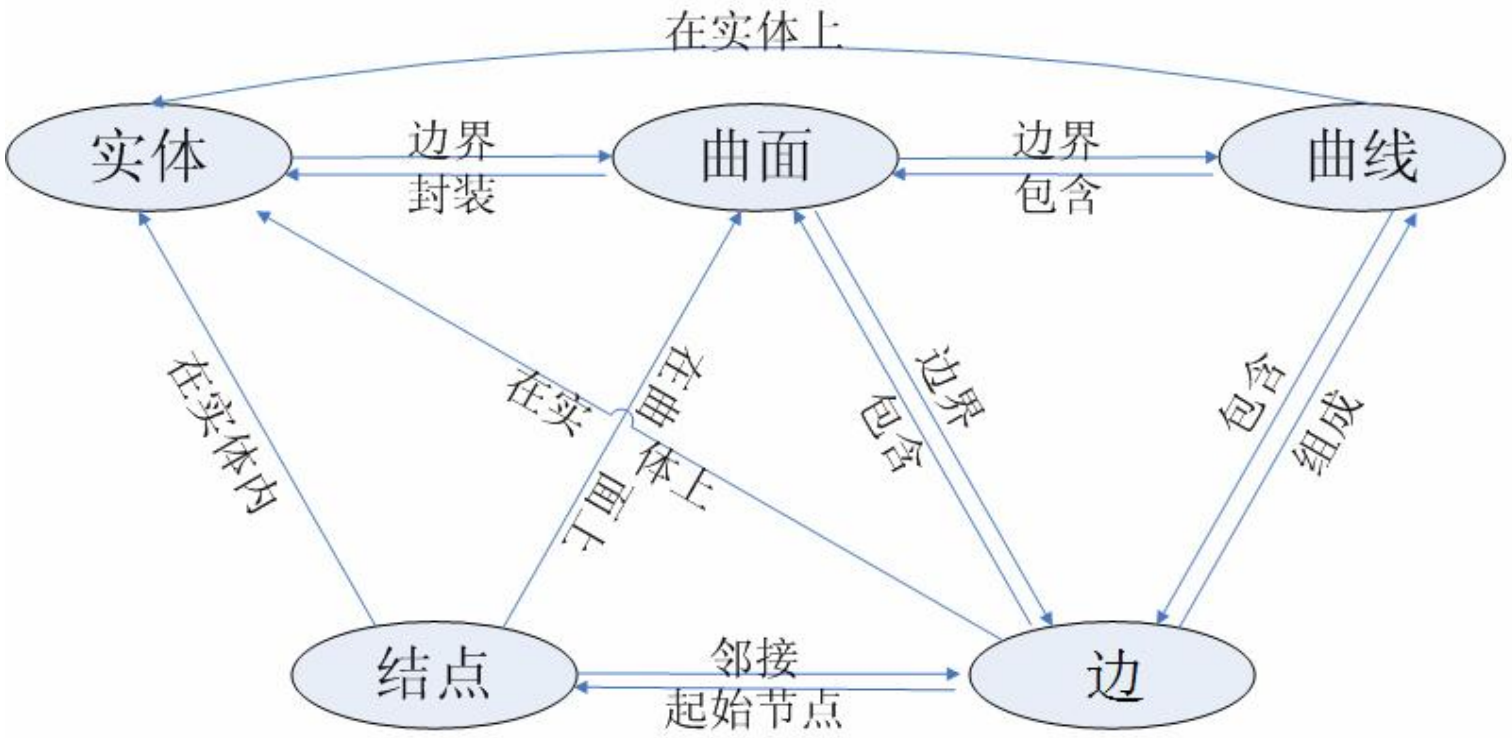
```
//结点数为n，用邻接矩阵gragh[n][n]存储边权，
//用indegree[n]存储每个结点的入度
void topologic(int* toposort)
{
    int cnt = 0;    //当前拓扑排序列表中有多少结点
    queue<int> q;   //保存入度为0的结点：还可以用栈甚至随机取
    int i;
    for(i = 0; i < n; i++)
    {
        if(indegree[i] == 0)
            q.push(i);
    }
    int cur;    //当前入度为0的结点
    while(!q.empty())
    {
        cur = q.front();
        q.pop();
        toposort[cnt++] = cur;
        for(i = 0; i < n; i++)
        {
            if(gragh[cur][i] != 0)
            {
                indegree[i]--;
                if(indegree[i] == 0)
                    q.push(i);
            }
        }
    }
}
```

进一步思考

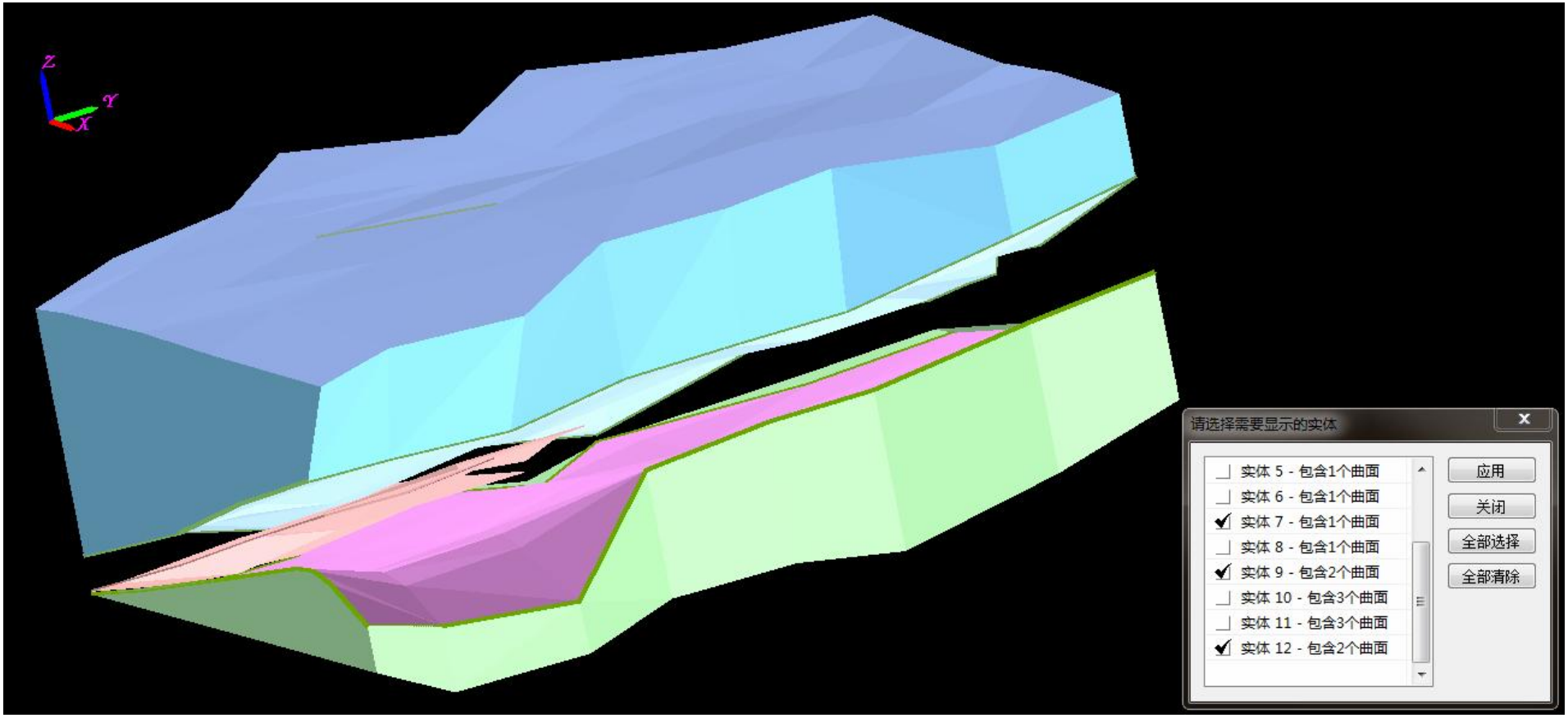
- 拓扑排序算法可以发现圈；其本质是不断输出入度为0的点；
- 可以用队列(或者栈)保存入度为0的点，避免每次遍历所有点查找入度；
  - 排序列表中的点需要更新与之连接的点的入度。入度减小1之后，如果为0，放入队列(或者栈)中。
- 拓扑排序其实是给定了结点的一组偏序关系。
- “拓扑”一词的本意不限于此，在GIS中，它往往指点、线、面、体之间的相互邻接关系，即“橡皮泥集合”——通过揉捏几何形体而不受影响的空间相互关系。存储这些关系，往往能够对某些算法带来好处。
  - 如：计算不自交的空间曲面是否能够围成三维体——任意三维边都邻接两个三维曲面。

扩展：拓扑的几何含义

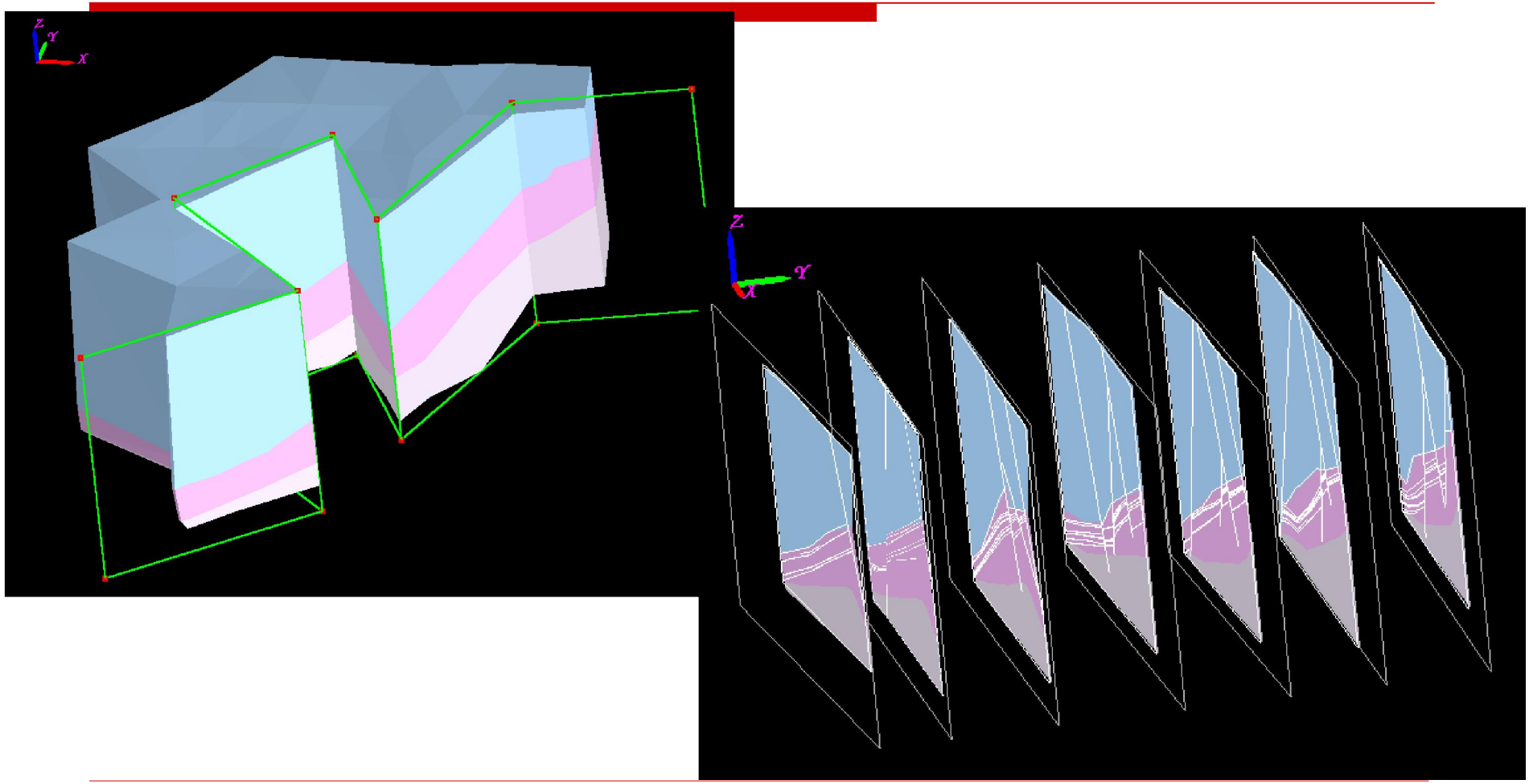
- 一种关系：如三维数据间的拓扑关系



三维拓扑重建


Z
Y
X
请选择需要显示的实体
实体 5 - 包含1个曲面
实体 6 - 包含1个曲面
实体 7 - 包含1个曲面
实体 8 - 包含1个曲面
实体 9 - 包含2个曲面
实体 10 - 包含3个曲面
实体 11 - 包含3个曲面
实体 12 - 包含2个曲面
应用
关闭
全部选择
全部清除

视角再次放大——面面、面线的拓扑



参考文献

- 余祥宣等，计算机算法基础[M]，华中科技大学出版社，2001
- 戴方勤,LeetCode 题解,2014
- http://www.patest.cn/contests/pat-a-practise/1030(最短路例题)
- http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=3204(MST例题)

我们在这里

- 更多算法面试题在7|七月算法
  - http://www.julyedu.com/
    - 免费视频
    - 直播课程
    - 问答社区

- contact us：微博
  - @研究者July
  - @七月问答
  - @邹博_机器学习

感谢大家！

欢迎大家提出宝贵的意见！