图论

七月算法 邹博

2015年4月28日

图的表述与搜索

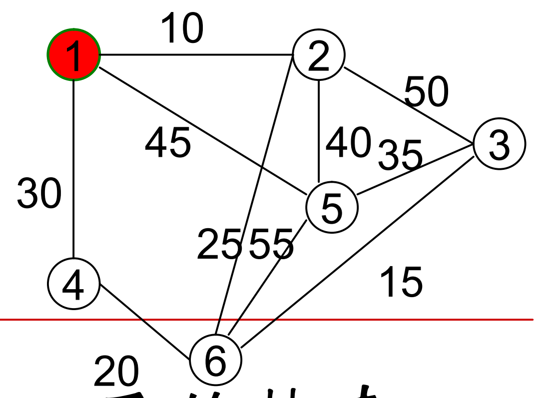
- 图的表示
  - 邻接矩阵
    - n*n的矩阵，有边是1，无边是0
  - 邻接表
    - 为每个点建立一个链表(数组)存放与之连接的点
- 搜索
  - BFS (Breadth-First-Search) 广(宽)度优先
  - DFS (Depth-First-Search) 深度优先

主要内容

- 树的遍历和搜索
- (隐式)图的搜索(连通性)
  - 重点
  - 8皇后
- 最短路径
  - 单源图(Dijkstra)
  - 任意两点(floyd)
  - 有负边(bellman-ford)
- 最小生成树(MST)
  - Prim
  - Krusal
- 拓扑排序(topsort)

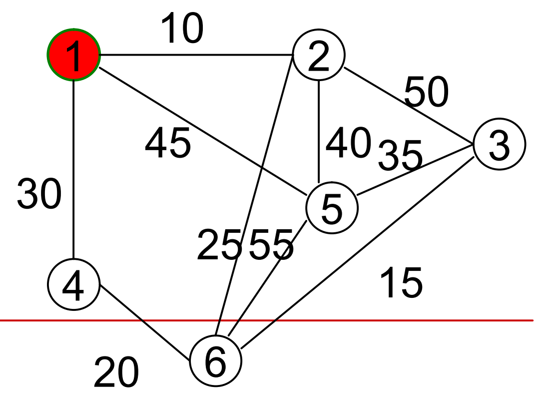
广度优先搜索：Breadth First Search，BFS

- 最简单、直接的图搜索算法
  - 从起点开始层层扩展
    - 第一层是离起点距离为1的
    - 第二层是离起点距离为2的
    - ....
  - 本质就是按层(距离)扩展，无回退



BFS分析

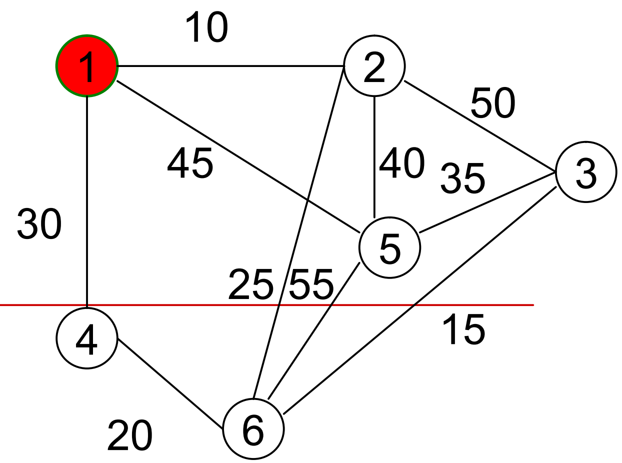
- 给定某起点a，将a放入缓冲区，开始搜索；
- 过程：假定某时刻缓冲区内结点为abc，则访问结点a的邻接点$a_1a_2a_x$，同时，缓冲区变成bc $a_1a_2a_x$，为下一次访问做准备；
  - 辅助数据结构：队列
  - 先进先出
  - 从队尾入队，从队首出队
  - 只有队首元素可见



BFS分析的两个要点

- 结点判重
  - 如果在扩展中发现某结点在前期已经访问过，则本次不再访问该结点；显然，第一次访问到该结点时，是访问次数最少的：最少、最短；
- 路径记录
  - 一个结点可能扩展出多个结点：多后继a a1a2ax，但是任意一个结点最多只可能有1个前驱(起始结点没有前驱)：单前驱
  - 用和结点数目等长的数组pre[0…N-1]：
  - pre[i]=j：第i个结点的前一个结点是j
  - 注：再次用到"存索引，不存数据本身"的思路。

BFS算法框架



- 辅助数据结构
  - 队列q；
  - 结点是第几次被访问到的d[0…N-1]：简称步数；
  - 结点的前驱pre[0…N-1]；
- 算法描述：
- 起点start入队q
  - 记录步数d[start]=0；
  - 记录start的前驱pre[start]=-1；
- 如果队列q非空，则队首结点x出队，尝试扩展x
  - 找到x的邻接点集合{y|(x,y)∈E}
    - 对每个新扩展结点y判重，如果y是新结点，则入队q；
    - 同时，记录步数d[y]=d[x]+1；前驱pre[y]=x；

BFS算法思考

- 对BFS的改进——双向BFS
  - 从起点和终点分别走，直到相遇；
    - 将树形搜索结构变成纺锤形；
  - 为什么这样更“快”？
    - 经典BFS中，树形搜索结构若树的高度非常高时，叶子非常多(树的宽度大)
    - 把一棵高度为h的树，用两个近似为h/2的树代替。宽度相对较小。

单词变换问题Word ladder

- 给定字典和一个起点单词、一个终点单词，每次只能变换一个字母，问从起点单词是否可以到达终点单词？最短多少步？
- 如：
  - start= "hit"
  - end = "cog"
  - dict = ["hot","dot","dog","lot","log"]
  - "hit" -> "hot" -> "dot" -> "dog" -> "cog"

单词变换问题

- 使用临界表，建立单词间的联系
  - 单词为图的结点，若能够变换，则两个单词存在无向边；
    - 若单词A和B只有1个字母不同，则(A-B)存在边；
  - 建图：
    - 预处理：对字典中的所有单词建立map、hash或者trie结构，利于后续的查找
    - 对于某单词w，单词中的第i位记为β，则将β替换为[β+1,Z]，查找新的串nw是否在字典中。如果在，将(w-nw)添加到邻接表项w和nw中(无向边)
    - 循环处理第二步
    - 若使用map，串在字典中的查找认为是O(logN)的，那么，整体时间复杂度为O(N*len*13*logN)，即O(N*logN)。若使用hash或者trie，整体复杂度为O(N)
- 从起始单词开始，广度优先搜索，看能否到达终点单词。若可以到达，则这条路径上的变化是最快的。

思考

- 有趣的是：虽然从起点单词开始到终点单词的路径内的单词，必须在词典内，但起点和终点本身是无要求的。
- 是否需要事先计算图本身？
- 体会路径记录问题。

Code

```
class Solution {
public:
    int ladderLength(const string& start, const string &end,
            const unordered_set<string> &dict) {
        queue<string> current, next;    // 当前层，下一层
        unordered_set<string> visited;  // 判重

        int level = 0;  // 层次
        bool found = false;

        auto state_is_target = [&](const string &s) {return s == end;};
        auto state_extend = [&](const string &s) {
            vector<string> result;

            for (size_t i = 0; i < s.size(); ++i) {
                string new_word(s);
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c == new_word[i]) continue;

                    swap(c, new_word[i]);

                    if ((dict.count(new_word) > 0 || new_word == end) &&
                            !visited.count(new_word)) {
                        result.push_back(new_word);
                        visited.insert(new_word);
                    }
                    swap(c, new_word[i]); // 恢复该单词
                }
            }

            return result;
        };

        current.push(start);
        while (!current.empty() && !found) {
            ++level;
            while (!current.empty() && !found) {
                const string str = current.front();
                current.pop();

                const auto& new_states = state_extend(str);
                for (const auto& state : new_states) {
                    next.push(state);
                    if (state_is_target(state)) {
                        found = true; //找到了
                        break;
                    }
                }
            }
            swap(next, current);
        }
        if (found) return level + 1;
        else return 0;
    }
};
```

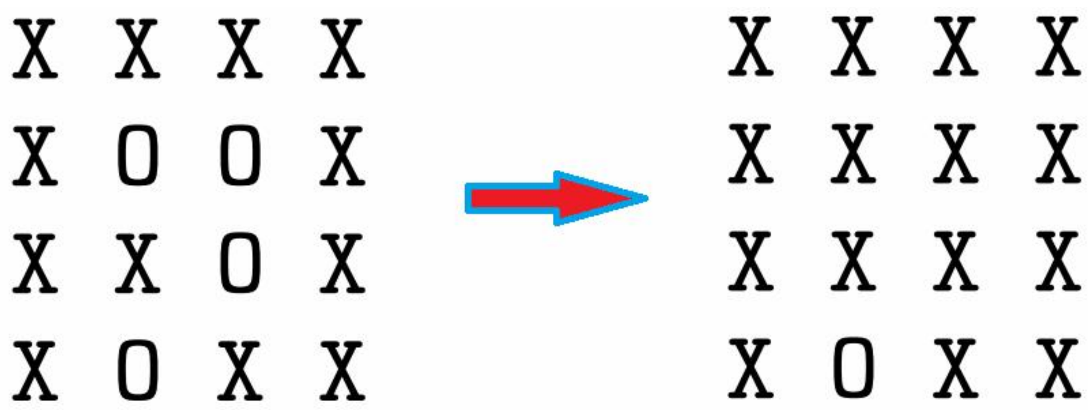
Code (split)

```cpp
auto state_is_target = [&](const string &s)
    {return s == end;};

auto state_extend = [&](const string &s) {
    vector<string> result;

    for (size_t i = 0; i < s.size(); ++i) {
        string new_word(s);
        for (char c = 'a'; c <= 'z'; c++) {
            if (c == new_word[i]) continue;

            swap(c, new_word[i]);

            if((dict.count(new_word)>0||new_word == end)&&
                    !visited.count(new_word)) {
                result.push_back(new_word);
                visited.insert(new_word);
            }
            swap(c, new_word[i]); // 恢复该单词
        }
    }
```

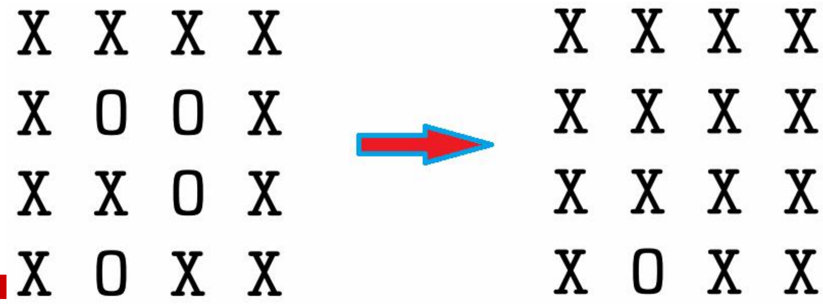
```cpp
class Solution {
public:
    int ladderLength(const string& start, const string &end,
            const unordered_set<string> &dict) {
        queue<string> current, next;    // 当前层, 下一层
        unordered_set<string> visited;  // 判重

        int level = 0;  // 层次
        bool found = false;

        current.push(start);
        while (!current.empty() && !found) {
            ++level;
            while (!current.empty() && !found) {
                const string str = current.front();
                current.pop();

                const auto& new_states = state_extend(str);
                for (const auto& state : new_states) {
                    next.push(state);
                    if (state_is_target(state)) {
                        found = true; //找到了
                        break;
                    }
                }
            }
            swap(next, current);
        }
        if (found) return level + 1;
        else return 0;
    }
};
```

周围区域问题

- 给定二维平面，格点处要么是‘X’，要么是‘O’。求出所有由‘X’围成的区域。
- 找到这样的(多个)区域后，将所有的‘O’翻转成‘X’即可。



分析



- 反向思索最简单：哪些‘O’是应该保留的？
  - 从上下左右四个边界往里走，凡是能碰到的‘O’，都是跟边界接壤的，应该保留。
  - 思路：
    - 对于每一个边界上的‘O’作为起点，做若干次广度优先搜索，对于碰到的‘O’，标记为其他某字符Y；
    - 最后遍历一遍整个地图，把所有的Y恢复成‘O’，把所有现有的‘O’都改成‘X’。

Code

```
class Solution {
public:
    void solve(vector<vector<char>> &board) {
        if (board.empty()) return;

        const int m = board.size();
        const int n = board[0].size();
        for (int i = 0; i < n; i++) {
            bfs(board, 0, i);
            bfs(board, m - 1, i);
        }
        for (int j = 1; j < m - 1; j++) {
            bfs(board, j, 0);
            bfs(board, j, n - 1);
        }
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                if (board[i][j] == 'O')
                    board[i][j] = 'X';
                else if (board[i][j] == '+')
                    board[i][j] = 'O';
    }
private:
    void bfs(vector<vector<char>> &board, int i, int j) {
        typedef pair<int, int> state_t;
        queue<state_t> q;
        const int m = board.size();
        const int n = board[0].size();

        auto is_valid = [&](const state_t &s) {
            const int x = s.first;
            const int y = s.second;
            if (x < 0 || x >= m || y < 0 || y >= n || board[x][y] != 'O')
                return false;
            return true;
        };

        auto state_extend = [&](const state_t &s) {
            vector<state_t> result;
            const int x = s.first;
            const int y = s.second;
            // 上下左右
            const state_t new_states[4] = {{x-1,y}, {x+1,y},
                    {x,y-1}, {x,y+1}};
            for (int k = 0; k < 4;  ++k) {
                if (is_valid(new_states[k])) {
                    // 既有标记功能又有去重功能
                    board[new_states[k].first][new_states[k].second] = '+';
                    result.push_back(new_states[k]);
                }
            }

            return result;
        };

        state_t start = { i, j };
        if (is_valid(start)) {
            board[i][j] = '+';
            q.push(start);
        }
        while (!q.empty()) {
            auto cur = q.front();
            q.pop();
            auto new_states = state_extend(cur);
            for (auto s : new_states) q.push(s);
        }
    }
};
```

Code (split)

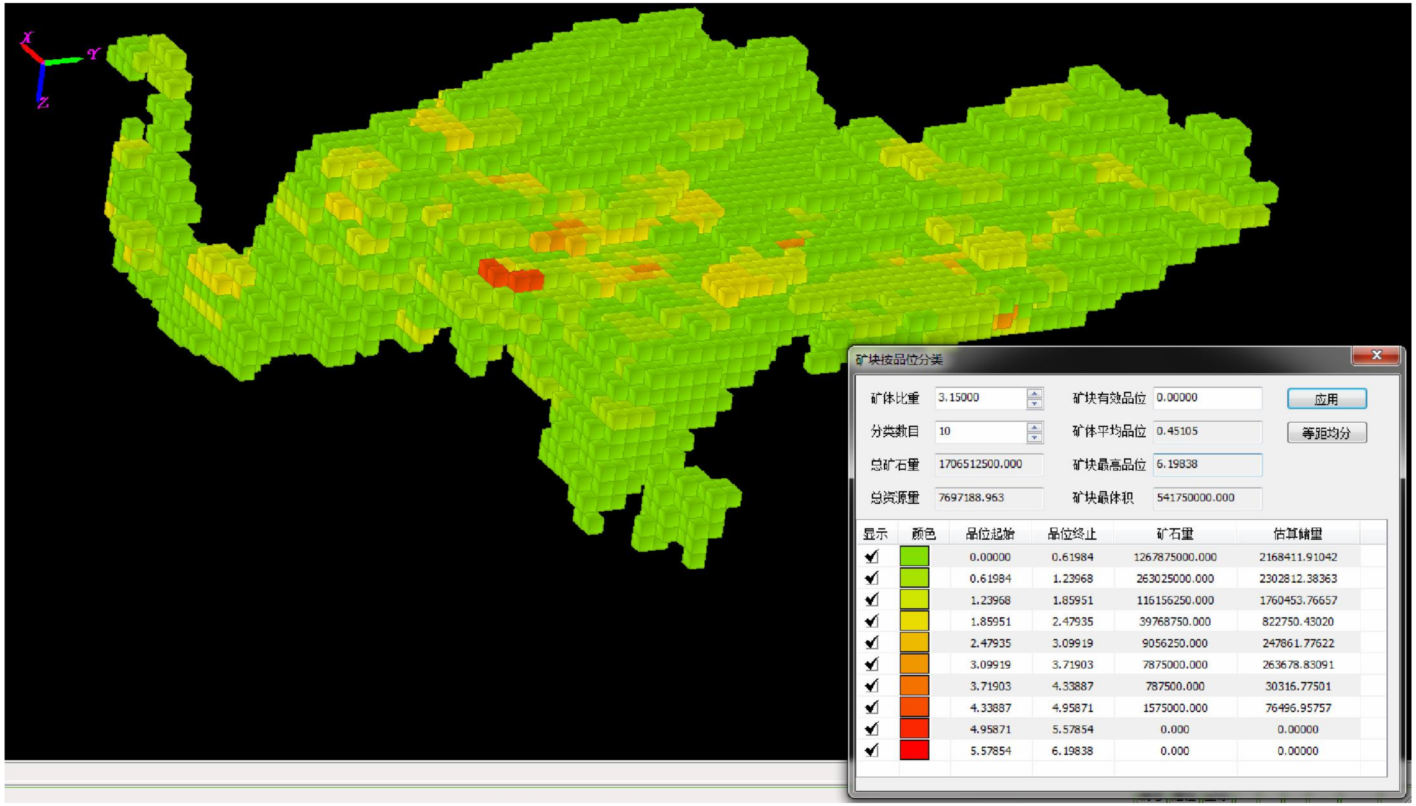
```
class Solution {
public:
    void solve(vector<vector<char>> &board) {
        if (board.empty()) return;

        const int m = board.size();
        const int n = board[0].size();
        for (int i = 0; i < n; i++) {
            bfs(board, 0, i);
            bfs(board, m - 1, i);
        }
        for (int j = 1; j < m - 1; j++) {
            bfs(board, j, 0);
            bfs(board, j, n - 1);
        }
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                if (board[i][j] == 'O')
                    board[i][j] = 'X';
                else if (board[i][j] == '+')
                    board[i][j] = 'O';
    }
};
```

```
private:
    void bfs(vector<vector<char>> &board, int i, int j) {
        typedef pair<int, int> state_t;
        queue<state_t> q;
        const int m = board.size();
        const int n = board[0].size();

        state_t start = { i, j };
        if (is_valid(start)) {
            board[i][j] = '+';
            q.push(start);
        }
        while (!q.empty()) {
            auto cur = q.front();
            q.pop();
            auto new_states = state_extend(cur);
            for (auto s : new_states) q.push(s);
        }
    }
```

```
auto is_valid = [&](const state_t &s) {
    const int x = s.first;
    const int y = s.second;
    if (x < 0 || x >= m || y < 0 || y >= n || board[x][y] != 'O')
        return false;
    return true;
};

auto state_extend = [&](const state_t &s) {
    vector<state_t> result;
    const int x = s.first;
    const int y = s.second;
    // 上下左右
    const state_t new_states[4] = {{x-1,y}, {x+1,y},
            {x,y-1}, {x,y+1}};
    for (int k = 0; k < 4;  ++k) {
        if (is_valid(new_states[k])) {
            // 既有标记功能又有去重功能
            board[new_states[k].first][new_states[k].second] = '+';
            result.push_back(new_states[k]);
        }
    }

    return result;
};
```

思考与拓展

- 如果目标区域是三维的呢？



显示	颜色	品位起始	品位终止	矿石量	估算储量
✔		0.00000	0.61984	1267875000.000	2168411.91042
✔		0.61984	1.23968	263025000.000	2302812.38363
✔		1.23968	1.85951	116156250.000	1760453.76657
✔		1.85951	2.47935	39768750.000	822750.43020
✔		2.47935	3.09919	9056250.000	247861.77622
✔		3.09919	3.71903	7875000.000	263678.83091
✔		3.71903	4.33887	787500.000	30316.77501
✔		4.33887	4.95871	1575000.000	76496.95757
✔		4.95871	5.57854	0.000	0.00000
✔		5.57854	6.19838	0.000	0.00000

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

DFS

- 优点
  - 不妨回忆一下Tarjan算法求解LCA问题
  - 由于只保存了一条路径，空间重复利用
- 缺点
  - 找到的解不一定是“最少”步数的
- 无论BFS，DFS找到解都和解的“位置”有关

回文划分问题

- 给定一个字符串s，将s划分成若干子串，使得每一个子串都是回文串。计算s的所有可能的划分。
- 如：s=“aab”，返回
  - “aa”，“b”；
  - “a”，“a”，“b”。

问题分析

- 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。
- 一个长度为n的字符串，有n-1个位置可以截断，每个位置可断可不断，因此时间复杂度为$O(2^{n-1})$。

Code

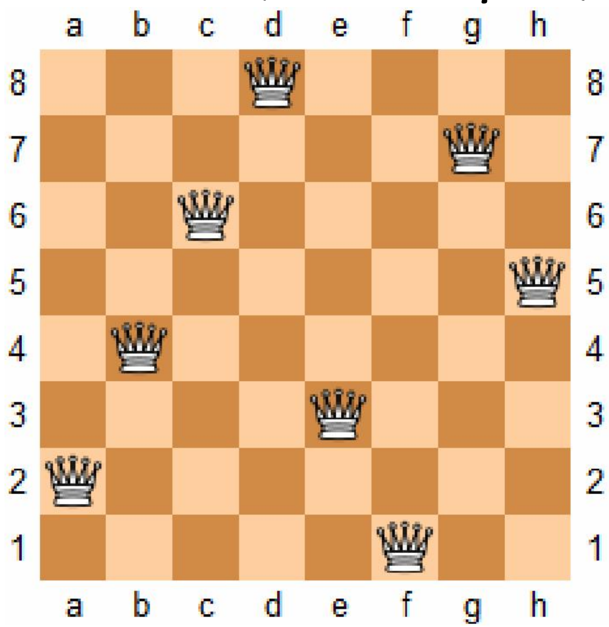
```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;  // 一个 partition 方案
        DFS(s, path, result, 0);
        return result;
    }
    // 搜索必须以 s[start] 开头的 partition 方案
    void DFS(string &s, vector<string>& 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;
    }
};
```

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。

八皇后问题

- 在8×8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种解法。



思路分析

- 分析：显然任意一行有且仅有1个皇后，使用数组queen[0…7]表示第i行的皇后位于哪一列。
  - 对于“12345678”这个字符串，调用全排列问题的代码，并且加入分支限界的条件判断是否相互攻击即可；
- 此外，也可以使用深度优先的想法，将第i个皇后放置在第j列上，如果当前位置与其他皇后相互攻击，则剪枝掉该结点。
  - 上述分析完全可以扩展到N皇后问题。

Code

```
class Solution {
public:
    vector<vector<string> > solveNQueens(int n) {
        this->columns = vector<int>(n, 0);
        this->main_diag = vector<int>(2 * n, 0);
        this->anti_diag = vector<int>(2 * n, 0);

        vector<vector<string> > result;
        vector<int> C(n, 0);  // C[i] 表示第 i 行皇后所在的列编号
        dfs(C, result, 0);
        return result;
    }
private:
    // 这三个变量用于剪枝
    vector<int> columns;  // 表示已经放置的皇后占据了哪些列
    vector<int> main_diag;  // 占据了哪些主对角线
    vector<int> anti_diag;  // 占据了哪些副对角线

    void dfs(vector<int> &C, vector<vector<string> > &result, int row) {
        const int N = C.size();
        if (row == N) { // 终止条件，也是收敛条件，意味着找到了一个可行解
            vector<string> solution;
            for (int i = 0; i < N; ++i) {
                string s(N, '.');
                for (int j = 0; j < N; ++j) {
                    if (j == C[i]) s[j] = 'Q';
                }
                solution.push_back(s);
            }
            result.push_back(solution);
            return;
        }

        for (int j = 0; j < N; ++j) {  // 扩展状态，一列一列的试
            const bool ok = columns[j] == 0 && main_diag[row + j] == 0 &&
                    anti_diag[row - j + N] == 0;
            if (! ok) continue;  // 剪枝：如果合法，继续递归
            // 执行扩展动作
            C[row] = j;
            columns[j] = main_diag[row + j] = anti_diag[row - j + N] = 1;
            dfs(C, result, row + 1);
            // 撤销动作
            // C[row] = 0;
            columns[j] = main_diag[row + j] = anti_diag[row - j + N] = 0;
        }
    }
};
```

数独Sudoku

- 解数独问题，初始化时的空位用‘.’表示。
  - 每行、每列、每个九宫内，都是1-9这9个数字。

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

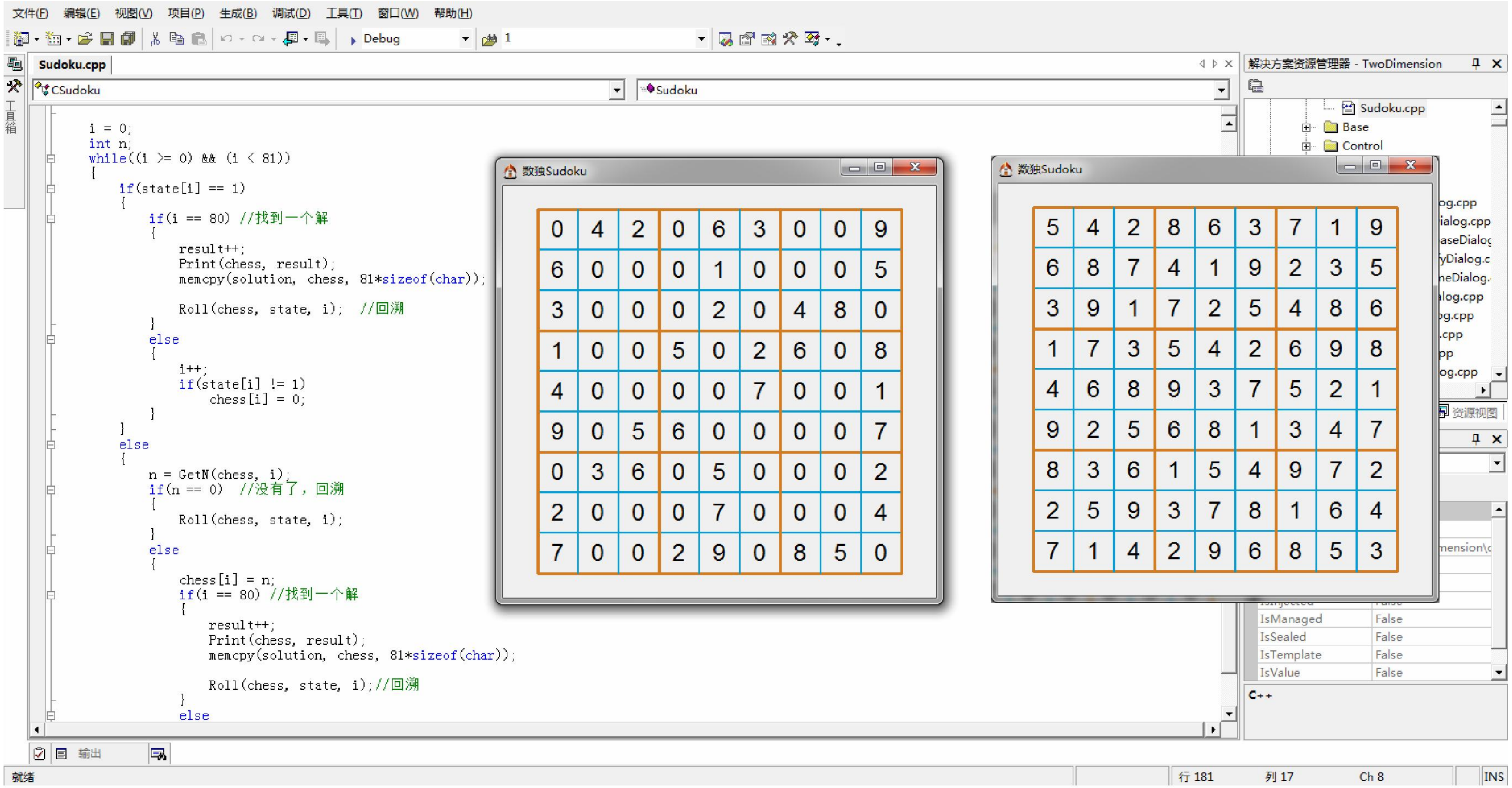
数独Sudoku分析

- 若当前位置是空格，则尝试从1到9的所有数；如果对于1到9的某些数字，当前是合法的，则继续尝试下一个位置——调用自身即可。

Code

```
class Solution {
public:
    bool solveSudoku(vector<vector<char> > &board) {
        for (int i = 0; i < 9; ++i)
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') {
                    for (int k = 0; k < 9; ++k) {
                        board[i][j] = '1' + k;
                        if (isValid(board, i, j) && solveSudoku(board
                            return true;
                        board[i][j] = '.';
                    }
                    return false;
                }
            }
        return true;
    }
private:
    // 检查 (x, y) 是否合法
    bool isValid(const vector<vector<char> > &board, int x, int y) {
        int i, j;
        for (i = 0; i < 9; i++) // 检查 y 列
            if (i != x && board[i][y] == board[x][y])
                return false;
        for (j = 0; j < 9; j++) // 检查 x 行
            if (j != y && board[x][j] == board[x][y])
                return false;
        for (i = 3 * (x / 3); i < 3 * (x / 3 + 1); i++)
            for (j = 3 * (y / 3); j < 3 * (y / 3 + 1); j++)
                if ((i != x || j != y) && board[i][j] == board[x][y])
                    return false;
        return true;
    }
};
```

非递归数独Sudoku



再谈LCA：Tarjan算法

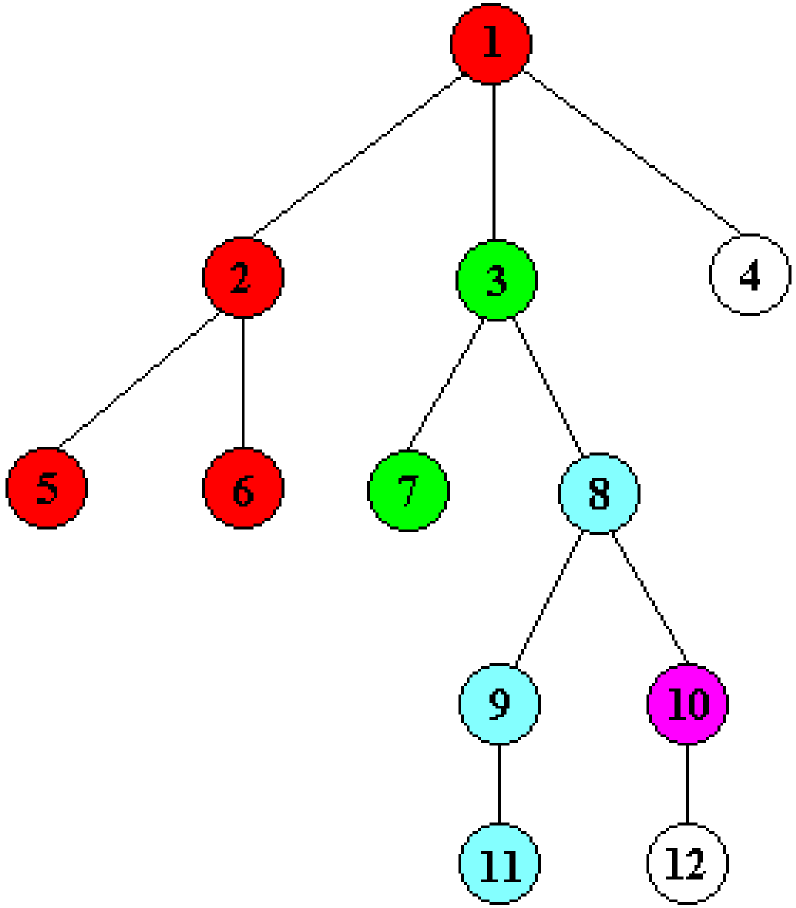
- Tarjan算法是由Robert Tarjan在1979年发现的一种高效的离线算法，也就是说，它要首先读入所有的询问(求一次LCA叫做一次询问)，然后并不一定按照原来的顺序处理这些询问，该算法的时间复杂度O(N*α(N)+Q)，其中，α(x)不大于4，N表示问题规模，Q表示询问次数。

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

- (2,10)
- (10,7)
- (9,10)
- (10,8)

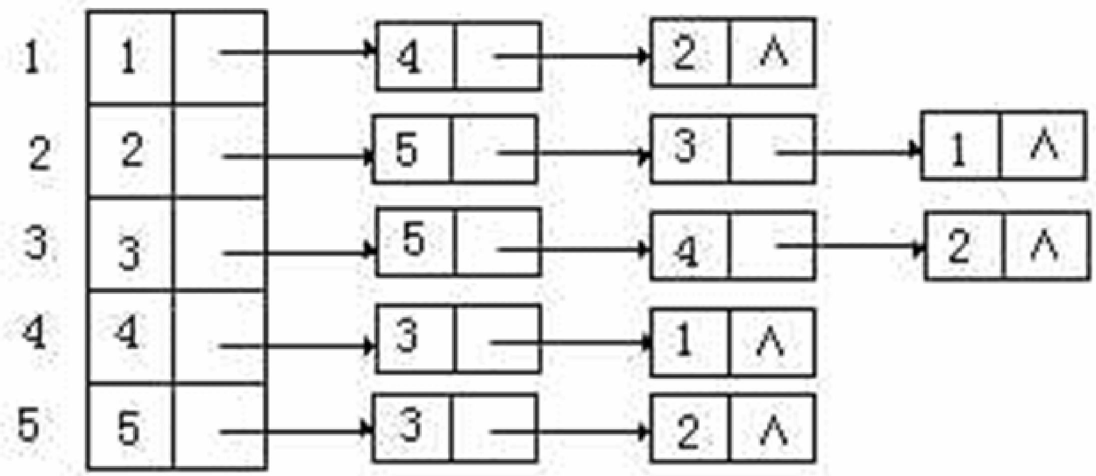


Tarjan Code

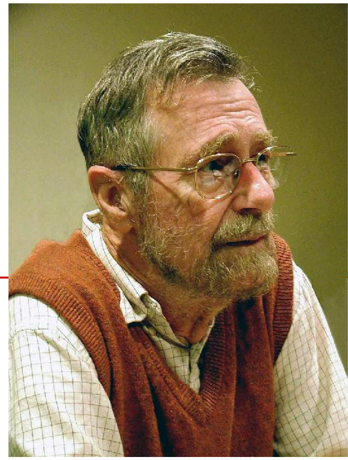
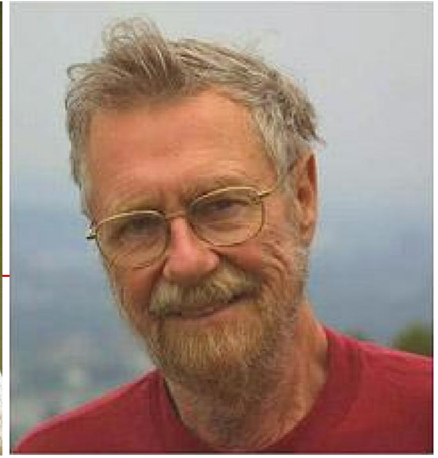
```
function TarjanOLCA(u)
  MakeSet(u);
  u.ancestor := u;   // 将集合u的祖先指向自己
  for each v in u.children do  // DFS所有孩子
     TarjanOLCA(v);
     Union(u,v);    // 与根结点U合并（并查集操作）
     Find(u).ancestor := u;   // 将u所在集合根的祖先指向u
  u.colour := black;  // 当所有孩子都已遍历，则标记根已完成
  for each v such that {u,v} in P 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 + ".";
```

处理查询的方法 -邻接表

- 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；
- 常见的空间检索方案。



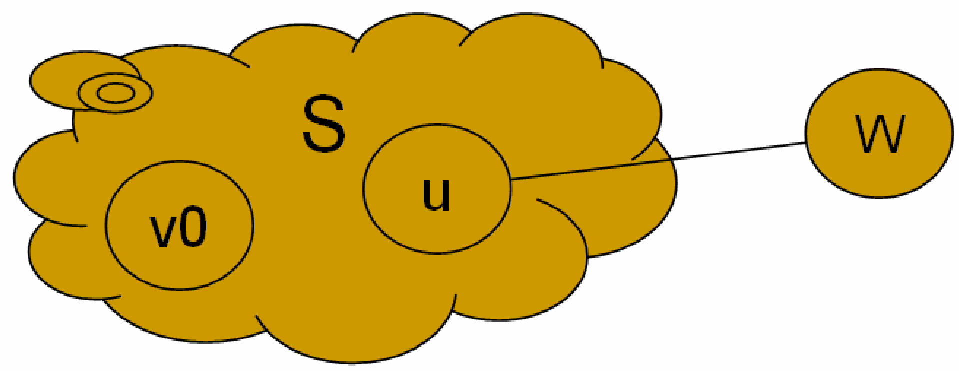
Edsger Wybe Dijkstra

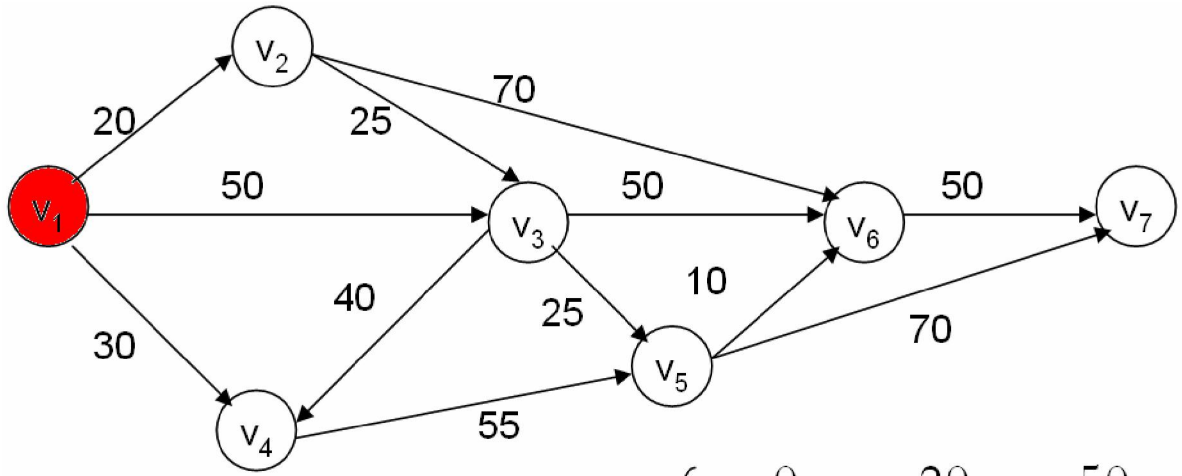
- 提出“goto有害论”;
- 提出信号量和PV原语;
- 解决了“哲学家聚餐”问题;
- 最短路径算法(SPF)和银行家算法的创造者;
- 第一个Algol 60编译器的设计者和实现者;
- THE操作系统的设计者和开发者;
  - 还有提过的“荷兰国旗问题”。

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))

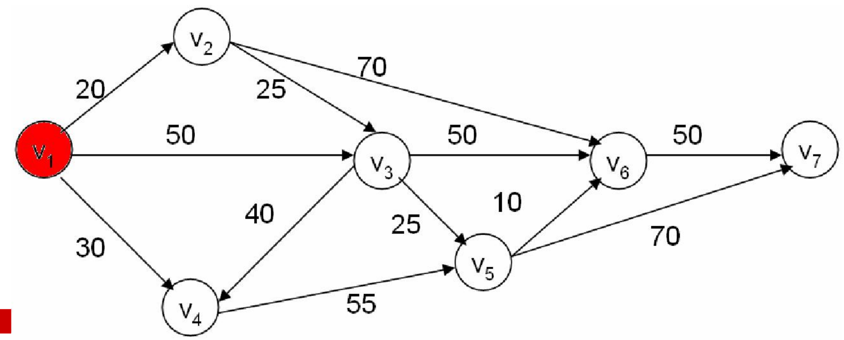


最短路径示例



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

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

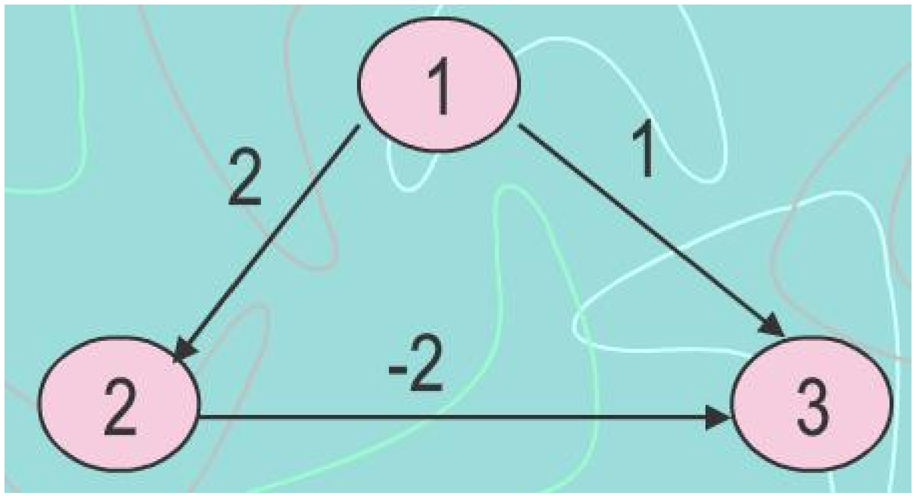
- 记录map[i,j]为结点i到结点j的最短路径的距离；则：
- $map[i,j] = \min\{map[i,k] + map[k,j], map[i,j]\}$
  - k取所有结点
- 同时，$map[n,n]==0$
  - i,j,k各自从0到N-1，所以时间复杂度为$O(n^3)$
- 此外，如果图中存在负的权值，算法也是适用的。
  - 思考：Dijkstra算法允许边存在负权吗？

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

- 计算图中最小的权值，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。
- 是否可行？



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

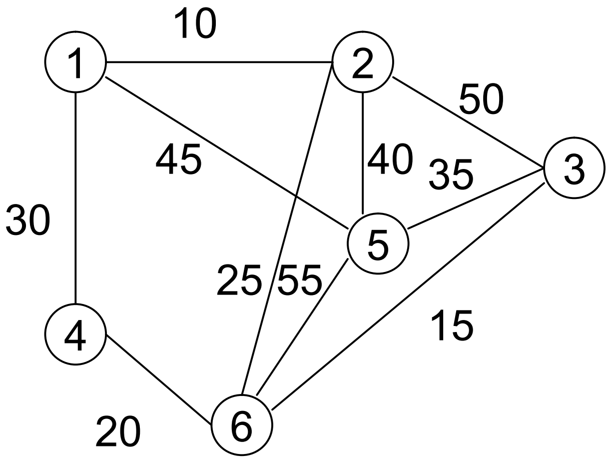
- 最小生成树要求从一个带权无向完全图中选择n－1条边并使这个图仍然连通(也即得到了一棵生成树)，同时还要考虑使树的权最小。最小生成树最著名算法是Prim算法和Kruskal算法。

Prim算法

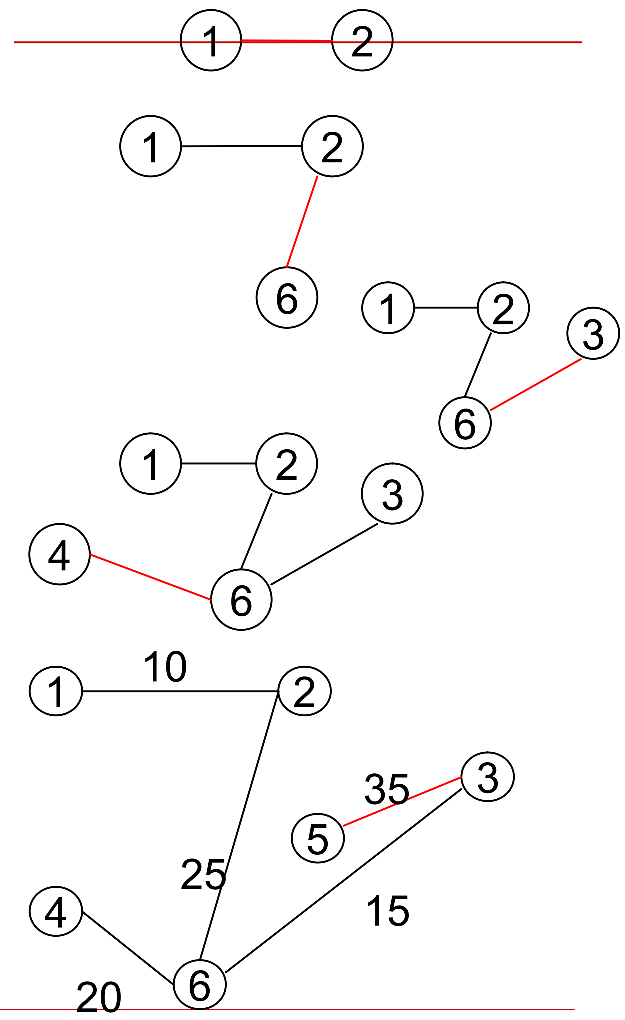
- 首先以一个结点作为最小生成树的初始结点，然后以迭代的方式找出与最小生成树中各结点权重最小边，并加入到最小生成树中。加入之后如果产生回路则跳过这条边，选择下一个结点。当所有结点都加入到最小生成树中之后，就找出了连通图中的最小生成树了。

Prim算法

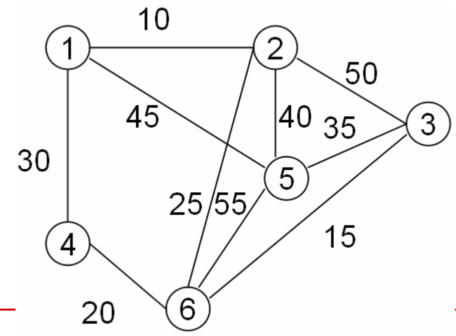
策略：使得迄今所选择的边的集合A构成一棵树；则将要计入到A中的下一条边(u,v)，应是E中一条当前不在A中且使得A∪{(u,v)}也是一棵树的最小成本边。



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。

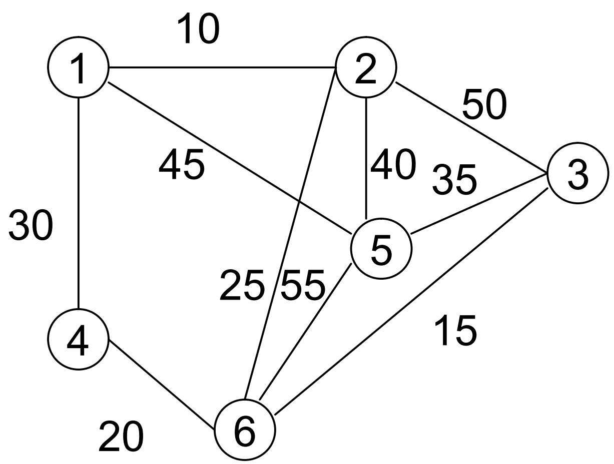
- 显然，当U=V时，步骤2共执行了n-1次(设n为图中顶点的数目)，TE中也增加了n-1条边，这n-1条边就是需要求出的最小生成树的边。

Kruskal算法

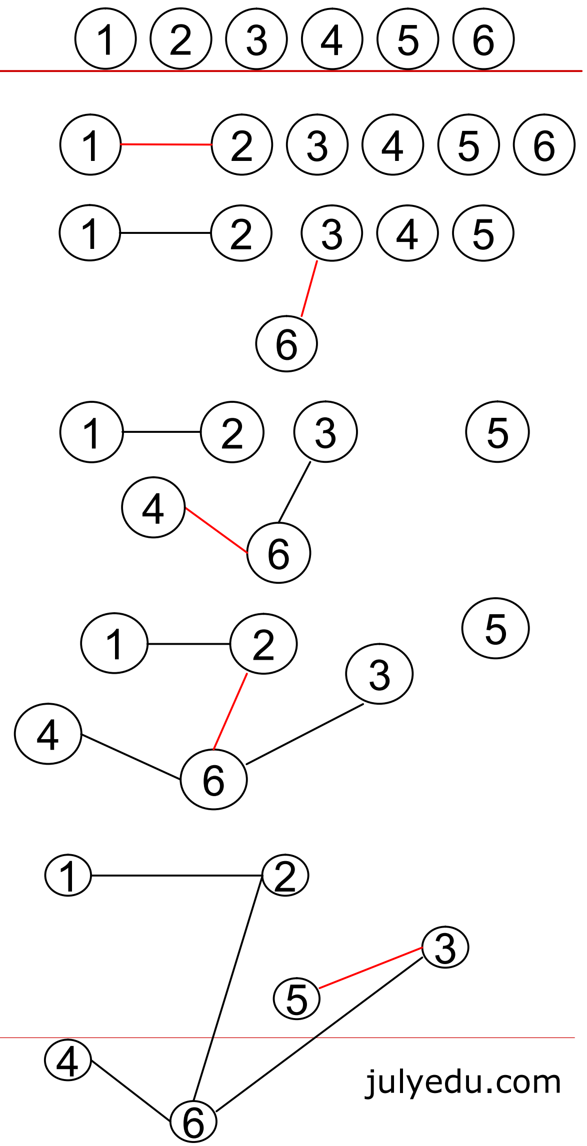
- Kruskal在找最小生成树结点之前，需要对所有权重边做从小到大排序。将排序好的权重边依次加入到最小生成树中，如果加入时产生回路就跳过这条边，加入下一条边。当所有结点都加入到最小生成树中之后，就找出了最小生成树。
  - Prim算法在得到最小生成树的过程中，始终保持是一颗树；而Kruskal算法最开始是森林，直到最后一条边加入，才得到树。

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

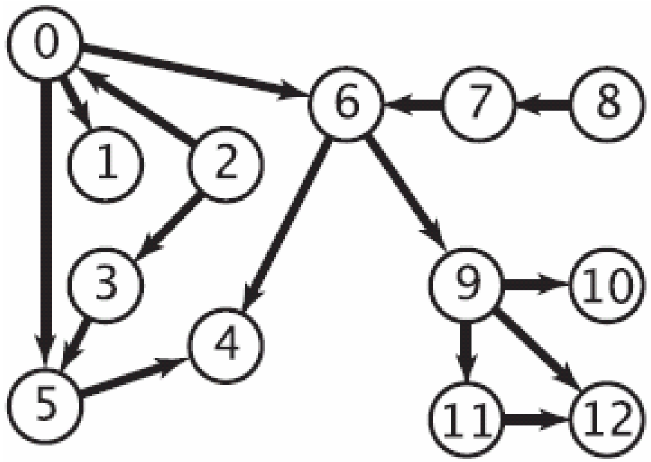


Kruskal算法几点说明

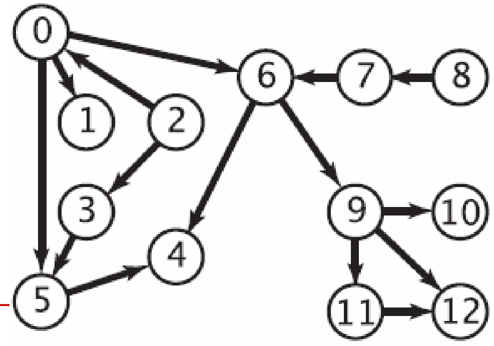
- 边集E以小顶堆的形式保存，一条当前最小成本边可以在O(loge)的时间内找到；
  - 当然，也可以用其他排序方法对边完全排序。
- 为了快速判断候选边e的加入是否会形成环，可考虑用并查集的方法：把当前状态的每个连通子图保存在各自的集合中；候选边是否可以加入，转化成边的两个顶点是否位于同一集合中；
- 算法的计算时间是O(eloge)。

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



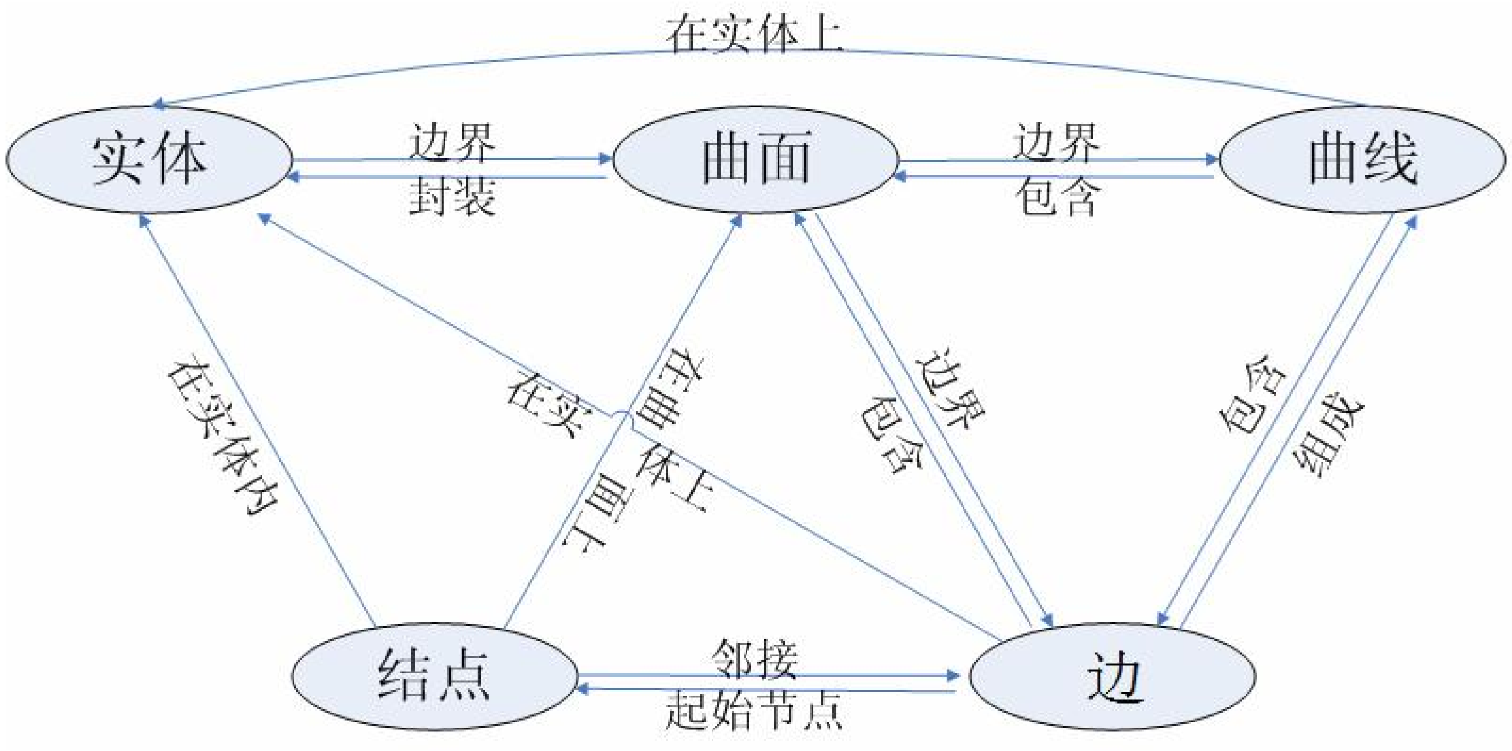
- 从有向图中选择一个没有前驱(即入度为0)的顶点并且输出它；
- 从网中删去该顶点，并且删去从该顶点发出的全部有向边；
- 重复上述两步，直到剩余的网中不再存在没有前趋的顶点为止。

拓扑排序说明

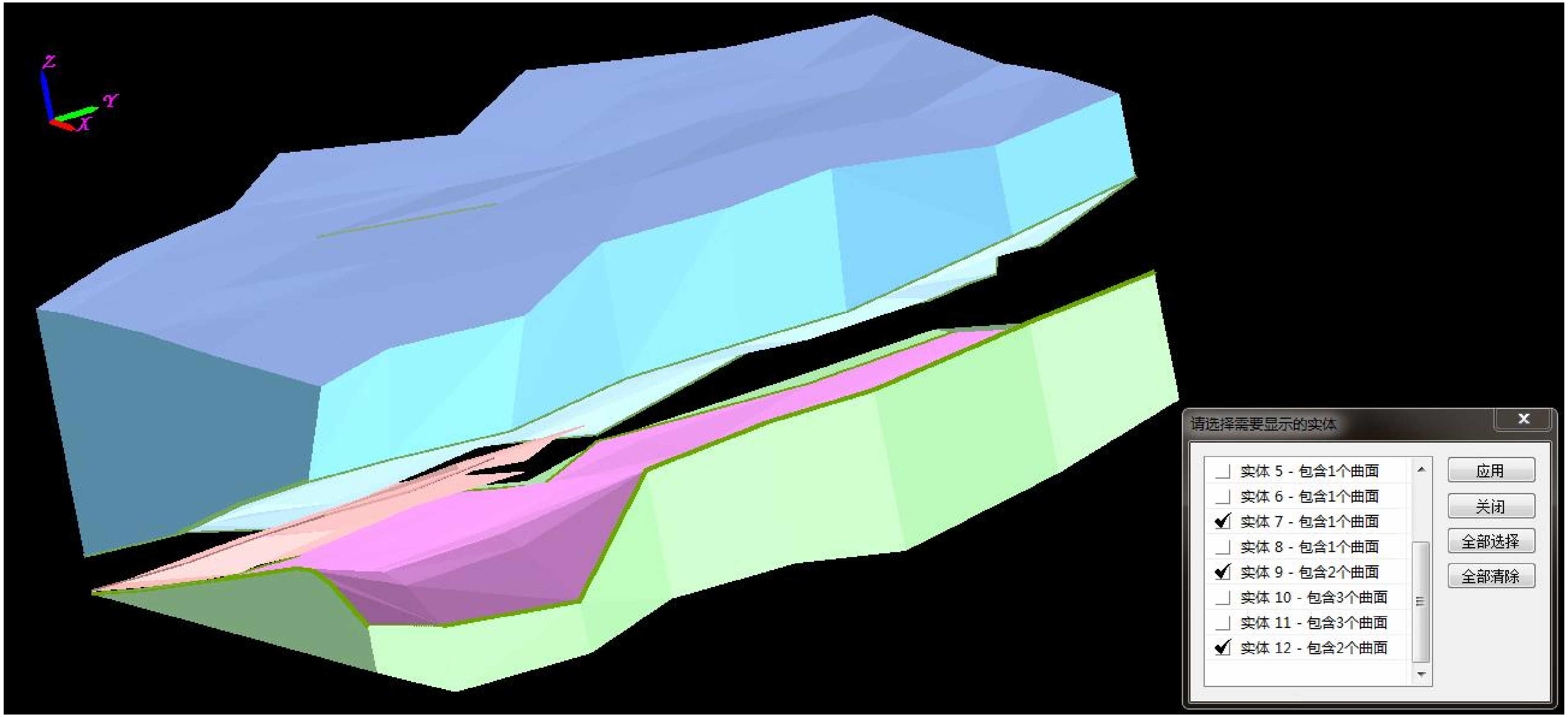
- 不断找入度为0的点
- 可以用队列(或者栈)保存入度为0的点，避免每次遍历所有点查找入度;
- 可以发现圈
- 排序列表中的点需要更新与之连接的点的入度。入度减小1之后，如果为0，放入队列(或者栈)中;

扩展：拓扑的几何含义

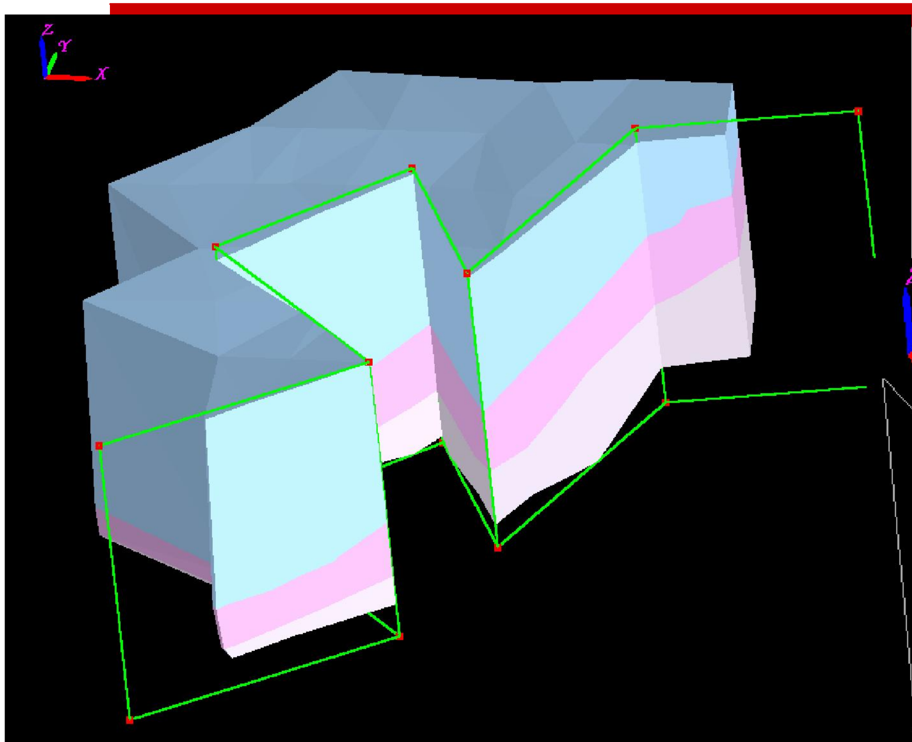
- 一种关系：如三维数据间的拓扑关系

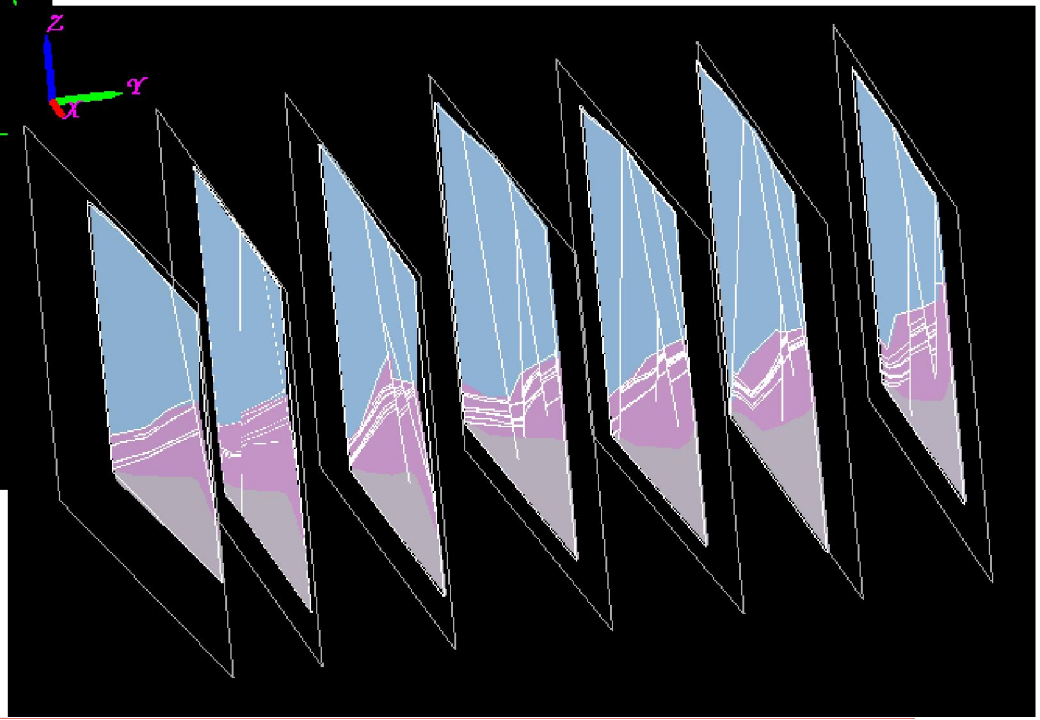


三维拓扑重建


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://baike.baidu.com/view/14495.htm(Floyd算法)
- http://zh.wikipedia.org/wiki/Floyd-Warshall%E7%AE%97%E6%B3%95(Floyd算法)
- http://blog.csdn.net/tsaid/article/details/6853736(Bellman-Ford算法)
- http://baike.baidu.com/view/1481053.htm(Bellman-Ford算法)
- http://zh.wikipedia.org/wiki/Prim%E6%BC%94%E7%AE%97%E6%B3%95 (Prim算法)
- http://squirrelrao.iteye.com/blog/1044867 (Prim算法)

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！