# 数据结构相关算法

* **1.稀疏矩阵转置算法思路：设矩阵列数为 cols，对矩阵三元组表扫描cols 次。第 k 次检测列号为 k 的项。**
* **第 k 次扫描找寻所有列号为 k的项，将其行号变列号、列号变行号，顺次存于转置矩阵三元组表。**
* **2.快速转置算法“：为加速转置速度，建立辅助数组 rowSize 和 rowStart，记录矩阵转置后各行非零元素个数和各行元素在转置三元组表中开始存放位置。**

**扫描矩阵三元组表，根据某项列号，确定它转置后的行号, 查 rowStart 表, 按查到的位置直接将该项存入转置三元组表中。**

**处理冲突的方法**

* **开放定址法（又称闭散列法）：**

**使用数组**

* **链地址法（又称开散列法）：**

**使用“拉链式”链表**

* ***Sn* 是搜索一个随机选择的关键码 *xi* (1** ≤ ***i*** ≤ ***n*) 所需的关键码比较次数的期望值**
* ***Un* 是在长度为 *m* 的散列表中 *n* 个桶已装入表项的情况下，装入第 *n*+1 项所需执行的关键码比较次数期望值。**
* **图的深度优先搜索递归算法**
* **（请自行改为非递归算法）**
* **template<class V, class E> void DFS (Graph <V, E>& g, const V& v) {**
* **int loc =g.GetVertexPos(v); if(loc == -1) return;**
* **int i, n = g.NumberOfVertices();**
* **bool\* visited = new bool [n];**
* **assert(visited != NULL);**
* **for ( int i = 0; i < n; i++ )**
* **visited [i] = false; //访问数组 visited 初始化**
* **DFS (g, loc, visited ); //从顶点v开始DFS遍历**
* **delete [ ] visited; //释放 visited**
* **}**
* **template<class V, class E> void DFS (Graph <V, E>& g, int v, bool visited[ ] ) {**
* **cout << g.GetValue (v) << ‘ ’; //访问顶点 v**
* **visited[v] = true; //顶点 v 作访问标记**
* **int w = g.GetFirstNeighbor (v);**
* **//取 v 的第一个邻接顶点 w**
* **while ( w != -1 ) { //若邻接顶点 w 存在**
* **if ( !visited[w] ) DFS ( g, w, visited );**
* **//若顶点 w 未访问过, 递归访问顶点 w**
* **w = g.GetNextNeighbor ( v, w );**
* **//取顶点 v 排在 w 后的下一个邻接顶点**
* **}**
* **}**
* **图的广度优先搜索算法**
* **template<class V, class E> void BFS (Graph <V, E>& g, const V& v ) {**
* **int loc = g.GetVertexPos(); if(loc == -1) return;**
* **int i, w, n = g.NumberOfVertices();**
* **bool\* visited = new bool[n];**
* **assert(visited != NULL);**
* **for ( int i = 0; i < n; i++ )**
* **visited[i] = false; //visited初始化**
* **cout << GetValue (v) << ' '; visited[v] = true;**
* **Queue<int> q;**
* **q.EnQueue (loc); //进队列**
* **while ( !q.IsEmpty ( ) ) { //队空则搜索结束**
* **q.DeQueue ( );**
* **w = g.GetFirstNeighbor (loc);**
* **while ( w != -1 ) { //若邻接顶点 w 存在**
* **if ( !visited[w] ) { //未访问过**
* **cout << g.GetValue (w) << ‘ ’;**
* **visited[w] = true; q.EnQueue (w);**
* **}**
* **w = GetNextNeighbor (loc, w);**
* **} //重复检测 v 的所有邻接顶点**
* **} //外层循环，判队列空否**
* **delete [ ] visited;**
* **}**

**克鲁斯卡尔 (Kruskal) 算法**

* **克鲁斯卡尔算法的基本思想：**

**设有一个有 *n* 个顶点的连通网络 *N* = { *V*, *E* }, 最初先构造一个只有 *n* 个顶点, 没有边的非连通图 *T* = { *V*, ∅ }, 图中每个顶点自成一个连通分量。当在 *E* 中选到一条具有最小权值的边时, 若该边的两个顶点落在不同的连通分量上，则将此边加入到 *T* 中; 否则将此边舍去，重新选择一条权值最小的边。如此重复下去, 直到所有顶点在同一个连通分量上为止。**

**普里姆(Prim)算法**

* **普里姆算法的基本思想：**

**从连通网络 *N* = { *V*, *E* }中的某一顶点 *u*0 出发, 选择与它关联的具有最小权值的边 ( *u*0, *v* ), 将其顶点加入到生成树顶点集合*U*中。**

**以后每一步从一个顶点在 *U* 中,而另一个顶点不在 *U* 中的各条边中选择权值最小的边(*u*, *v*), 把它的顶点加入到集合 *U* 中。如此继续下去, 直到网络中的所有顶点都加入到生成树顶点集合 *U* 中为止。**

* **采用邻接矩阵作为图的存储表示。**

**最短路径(Shortest Path)**

**—**  **Dijkstra算法**