

## 第6章



[6-1] **关联矩阵 (incidence matrix)** 是描述和实现图算法的另一重要方式。对于含有  $n$  个顶点、 $e$  条边的图，对应的关联矩阵  $I[][]$  共有  $n$  行  $e$  列。在无向图中，对于任意的  $0 \leq i < n$  和  $0 \leq j < e$ ，若第  $i$  个顶点与第  $j$  条边彼此关联，则定义  $I[i][j] = 1$ ；否则，定义  $I[i][j] = 0$ 。

#### a) 关联矩阵与邻接矩阵有何联系？

**【解答】**

就无向图而言（不考虑自环等情况），既然每一条边均恰好与两个顶点关联，故该矩阵中的每一列都应恰好包含两个1，总和均为2。

考查关联矩阵与其转置的乘积  $B = I \cdot I^T$ 。该矩阵对角线上的任意元素  $B[i][i]$ ，都应满足：

$$B[i][i] = I[i] \cdot I[i]^T = \text{顶点 } i \text{ 的度数}$$

而对于对角线以外的任意元素  $B[i][j]$ ， $i \neq j$ ，都有

$$B[i][j] = I[i] \cdot I[j]^T = \text{顶点 } i \text{ 与顶点 } j \text{ 之间公共的关联边数}$$

也就是说， $B[i][j]$  等于1/0当且仅当顶点  $i$  与顶点  $j$  是/否彼此邻接。

#### b) 有向图的关联矩阵应如何定义？

通常的定义方式为，对于任意的  $0 \leq i < n$  和  $0 \leq j < e$ ，若第  $j$  条边从第  $i$  个顶点发出（即顶点  $i$  为边  $j$  之尾），则定义  $I[i][j] = -1$ ；若第  $j$  条边进入第  $i$  个顶点（即顶点  $i$  为边  $j$  之头），则定义  $I[i][j] = +1$ ；否则，定义  $I[i][j] = 0$ 。

#### c) 有向图的关联矩阵，与邻接矩阵又有何联系？

与无向图类似地，有向图关联矩阵中的每一列应包含+1和-1各一个，总和应均为0。

为发现此时两种矩阵之间的联系，不妨依然考查关联矩阵与其转置的乘积  $B = I \cdot I^T$ 。

对于该矩阵对角线上的任意元素  $B[i][i]$ ，都有

$$B[i][i] = I[i] \cdot I[i]^T = \text{顶点 } i \text{ 的（出、入总）度数}$$

而对于对角线以外的任意元素  $B[i][j]$ ， $i \neq j$ ，都有

$$-B[i][j] = -I[i] \cdot I[j]^T = \text{顶点 } i \text{ 与顶点 } j \text{ 之间公共的关联（有向）边数}$$

也就是说， $B[i][j]$  等于0当且仅当顶点  $i$  与顶点  $j$  互不邻接； $B[i][j]$  等于-1或-2，当且仅当在顶点  $i$  与顶点  $j$  之间，联接有1或2条有向边。

#### d) 基于关联矩阵，可以解决哪些问题？试举一例。

**【解答】**

以参考文献[20]为例，其中的24.4节针对线性规划（linear programming）问题的一种特例——差分约束系统（system of difference constraints）——介绍了一个高效算法。该算法中最为重要的一个步骤，就是将差分约束系统转化为有向带权图：将差分约束变量视作顶点，将差分约束矩阵视作关联矩阵。如此，原问题即可转化为有向带权图的最短路径问题。

[6-2] 试说明，即便计入向量扩容所需的时间，就分摊意义而言，`GraphMatrix::insert(v)`算法的时间复杂度依然不超过  $\mathcal{O}(n)$ 。

【解答】

首先请注意，`GraphMatrix`类（教材157页代码6.2）在底层，是基于可扩充向量，以二维`Vector`结构的形式来实现邻接矩阵。

按照第2.4节的实现方法及其分析结论，每一向量（即邻接矩阵的每一行）的单次插入操作，在分摊意义上只需  $\mathcal{O}(1)$  时间。这里，在每一节点的插入过程中， $n$  个向量的操作（含扩容操作）完全同步，故总体的运行时间不超过分摊的  $\mathcal{O}(n)$ 。

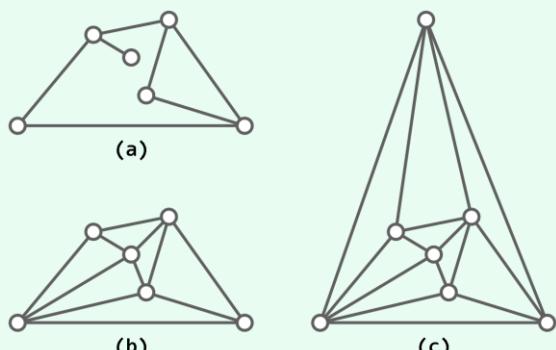
当然，为插入一个顶点，在最坏情况下可能需要访问和修改整个邻接矩阵，共需  $\mathcal{O}(n^2)$  时间。

[6-3] 所谓平面图，即可以将  $n$  个顶点映射为平面上的  $n$  个点，并且顶点之间的所有联边只相交于其公共端点，而不相交于边的内部。

试证明，平面图必满足  $e = \mathcal{O}(n)$ ，亦即，边数与顶点数同阶。（提示：平面图必然遵守欧拉公式  $n - e + f - c = 1$ ，其中  $n$ 、 $e$ 、 $f$  和  $c$  分别为平面图的顶点、边、面和连通域的数目）

【解答】

不妨设这里所讨论的平面图，如图x6.1(a)所示，至少包含3个顶点；自然地，同时也包含  $c \geq 1$  个连通域。考查其中各边与各面之间的关联关系，将其总数记作  $I$ 。



图x6.1 (a)平面图、(b)三角剖分以及  
(c)外面亦为三角形的三角剖分

首先不难看出，悬边仅与一张面关联，其余各条边均与两张面相关联。因此，每条边对  $I$  的贡献至多为2，故有：

$$I \leq 2e \dots \dots \dots \quad (1)$$

另一方面，平面图中仅有张无界的面——即所谓的外面（outer face）——它对  $I$  的贡献至少为3。此外其余的各张面，均由至少三条边围成，对  $I$  的贡献也至少为3，故有：

$$3f \leq I \dots \dots \dots \quad (2)$$

联立不等式(1)和(2)，即有：

$$3f \leq 2e$$

$$f \leq 2e/3 \dots \dots \dots \quad (3)$$

将不等式(3)代入欧拉公式，则有：

$$1 = n - e + f - c \leq n - e + 2e/3 - 1$$

稍作整理，即得：

$$e \leq 3n - 6 = \mathcal{O}(n) \dots \dots \dots \quad (4)$$

由以上证明也可进一步推知，不等式(4)取等号，当且仅当不等式(1~3)均取等号。

此时，每张面（包括外面）应恰好由三条边围成。也就是说，该平面图不仅如图x6.1(b)所示，是习题[6-33]中定义的三角剖分（triangulation），而且如图x6.1(c)所示，外面也必须是一个三角形。

### [6-4] 考查无向图的邻接矩阵表示法。

a ) 试通过将二维邻接矩阵映射至一维向量，提高空间利用率。

**【解答】**

无向图的邻接矩阵必然对称，亦即， $A[i][j] = A[j][i]$  对合法的任意*i*和*j*均成立。因此，邻接矩阵的上或下半角完全可以不必记录，并将剩余部分转化并压缩为一维向量A'。

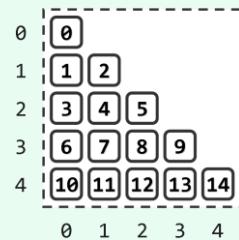
这里，不妨仅保存其中的下半三角区域（含对角线），即所有满足 $0 \leq j \leq i < n$ 的元素A[i][j]。于是如图x6.2所示不难验证，可以在这些元素与向量A'之间建立如下一一对应关系：

$$A[i][j] \leftrightarrow A'[i(i+1)/2 + j]$$

或者等价地：

$$A'[k] \leftrightarrow A[i][k - \frac{i(i+1)}{2}], \text{ 其中 } i = \lfloor \frac{\sqrt{8k+1}-1}{2} \rfloor$$

如此所得一维向量A'的长度为 $n(n+1)/2$ ，大致为未压缩之前的一半。但就渐进意义而言，空间复杂度依然为 $\mathcal{O}(n^2)$ 。



图x6.2 将 $5 \times 5$ 的对称矩阵压缩  
至长度为15的一维向量

b ) 采用你所提出的方法，需额外增加多少处理时间？

**【解答】**

就从A中元素到A'中元素的映射而言，以上转换均属于基本操作，各自仅需 $\mathcal{O}(1)$ 时间。

c ) 采用你所提出的方法，是否会影响到图 ADT 各接口的效率？

**【解答】**

既然以上转换均属于基本操作，故在顶点集保持不变的情况下，各接口所需时间将保持不变。然而在图的规模可能发生改变的场合，无论是新顶点的引入还是原顶点的删除，都有可能需要移动A'中的所有元素，从而造成巨大的额外时间开销，因此得不偿失。

### [6-5] 考查邻接表表示法。

a ) 试按照 158 页 6.4 节的思路，以邻接表的形式实现图 ADT 的各操作接口；

**【解答】**

一种可行的实现方式大致如下。首先，将原定义的整体框架：

```
#include "../Vector/Vector.h" //引入向量
/* ... */
template <typename Tv, typename Te> //顶点类型、边类型
class GraphMatrix : public Graph<Tv, Te> { //基于向量，以邻接矩阵形式实现的图
private:
    Vector< Vertex< Tv > > V; //顶点集（向量）
    Vector< Vector< Edge< Te > * > > E; //边集（邻接矩阵）
    /* ... */
}
```

调整为

---

```
#include "../Vector/Vector.h" //引入向量
#include "../List/List.h" //引入列表
/* ... */
template <typename Tv, typename Te> //顶点类型、边类型
class GraphList : public Graph<Tv, Te> { //基于向量和列表，以邻接表形式实现的图
private:
    Vector< Vertex< Tv >> V; //顶点集（向量）
    Vector< List< Edge< Te >*>> E; //边集（邻接表）
    /* ... */
}
```

---

可见，所有顶点依然构成一个向量，且分别将各自的关联边组织为一个列表（即所谓边表）。既然同一边表内的边都关联于同一顶点，故为了便于查找另一关联顶点，接下来还需相应地在原Edge边结构的基础上，再增加一个域v：

---

```
template <typename Te> struct Edge { //边对象
    Te data; int weight; EStatus status; //数据、权重、状态
    int v; //关联顶点
    Edge( Te const& d, int w ) : data( d ), weight( w ), status( UNDETERMINED ) {} //构造新边
};
```

---

对于有向图，可以统一约定各边分别归属于其尾顶点所对应的边表（出边表），或统一归属于其头顶点（入边表）。而为了提高查找效率，甚至可以同时为各顶点设置出边表和入边表。

Graph各标准接口的具体实现，也要做相应的调整，凡涉及边表的操作都要将此前Vector结构的操作替换为List结构的操作。请读者独立完成这些工作。

### b) 分析这一实现方式的时间、空间效率，并与基于邻接矩阵的实现做一对比。

**【解答】**

在空间复杂度方面，邻接表可以动态地与图结构的实际规模相匹配，而不再是固定的 $\Theta(n^2)$ 。具体地，若当时的图结构共含n个顶点、e条边，则实际的空间消耗应不超过 $\mathcal{O}(n + e)$ 。

与邻接矩阵相比，多数针对顶点的操作的时间复杂度几乎不变，但涉及边的操作则不尽相同。

在这里，边确认操作exists(i, j)的作用至关重要。改为邻接表之后，我们需要遍历顶点i所对应的边表，方可判定其中是否存在与顶点j相关联者，因此其所需时间由 $\mathcal{O}(1)$ 增加至 $\mathcal{O}(\deg(i)) = \mathcal{O}(n)$ 。相应地，涉及exists()操作的顶点删除操作remove(i)也需要更多的时间。此外，在改用邻接表之后，边删除操作remove(i, j)也需要以类似的方式确认边(i, j)的确存在，并在存在时确定该边记录的存放位置，因此该操作也将不能在 $\mathcal{O}(1)$ 时间内完成。

请读者根据自己的具体实现方式，对其它操作接口时间效率的变化补充分析。

[6-6] 试基于BFS搜索设计并实现一个算法，在 $\mathcal{O}(n + e)$ 时间内将任一无向图分解为一组极大连通域。

【解答】

反观如教材160页代码6.3所示的广度优先搜索算法，其子算法BFS(v)只有在访遍顶点v所属的极大连通域之后，方可返回；此后，若还有其它尚未访问的连通域，则算法主入口bfs()中的循环必然会继续检查其余的所有顶点，而一旦发现尚处于UNDISCOVERED状态的顶点，即会再次调用子算法BFS()并遍历该顶点所属的极大连通域。

由此可见，只需按照BFS()的各次调用顺序，分批输出所访问的顶点以及边，即可实现无向图的极大连通域分解。

相对于基本的广度遍历算法，除了顶点和边的输出，该算法并未引入更多操作，因此其时间复杂度依然是 $\mathcal{O}(n + e)$ 。

实际上，上述分析以及结论，同样适用于如教材162页代码6.4所示的深度优先遍历算法，请读者自行验证。

[6-7] 若在图G中存在从顶点s通往顶点v的道路，则其中最短道路的长度称作s到v的(最小)距离，记作 $\pi(v)$ ；不存在通路时，取 $\pi(v) = +\infty$ 。

试证明，在起始于s的广度优先搜索过程中：

- a) 前沿集内的各顶点始终按其在BFS树中的深度，在辅助队列中单调排列；且任何时刻同处于辅助队列中的顶点，深度彼此相差不超过一个单位；

【解答】

采用数学归纳法，证明该不变性在每一顶点入队后都成立。

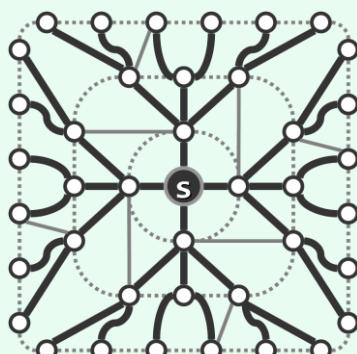
初始时队列为空，自然成立。

一般地，考查下一入队顶点u，其在BFS树中的深度 $\text{depth}(u)$ ，在其入队的同时确定。而就在u入队的那一步迭代之前，必有某一顶点v刚刚出队，且在BFS树中u是v的孩子，故有：

$$\text{depth}(u) = \text{depth}(v) + 1$$

因此，倘若题中所述不变性在该步迭代之前成立，则在v出队、u入队后应该继续成立。

- b) 所有顶点按其在BFS树中的深度，以非降次序接受访问。

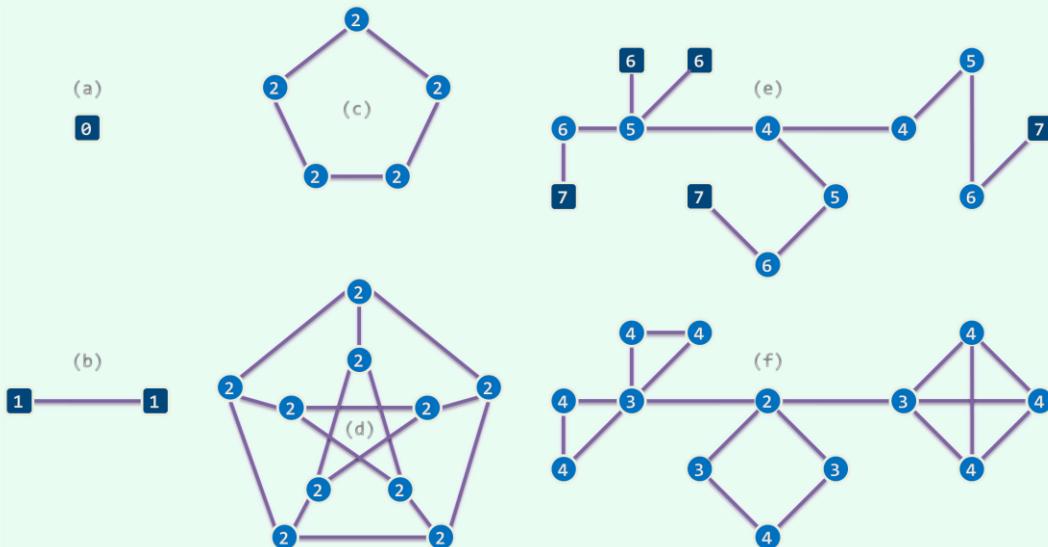


图x6.3 图的BFS搜索，等效于BFS树的层次遍历



## [6-9] 考查无向图。

- a) 对于图中任一顶点  $v$ , 其余顶点到它的距离的最大值, 称作其偏心率 (eccentricity), 记作  $\epsilon(v)$ 。试验证, 在如图 x6.4 所示的六幅图中, 各顶点所标注的就是其对应的偏心率。



图x6.4 偏心率实例 : (a) 单个顶点 ; (b) 两个顶点的完全图 ; (c) 周长为奇数 (5) 的环形 ;  
(d) Petersen图 (Petersen's Graph) ; (e) 13个顶点、12条边构成的树 ; (f) 13个顶点、18条边构成的图

## 【解答】

请读者逐一验证。

- b) 若将图中偏心率最小的顶点称作中心点 (central vertex 或 center), 并将这个最小偏心率称作图的半径 (radius), 则如图 x6.4 所示的六幅图各有几个中心点? 它们的半径各是多少?

## 【解答】

按定义, 分别有1个、2个、5个、10个、2个、1个中心点, 半径分别为0、1、2、2、4、2。

- c) 若将图中偏心率最大的顶点称作边缘点 (peripheral vertex), 并将这个最大值称作图的直径 (diameter), 则如图 x6.4 所示的六幅图各有几个边缘点? 它们的直径各是多少?

## 【解答】

按定义, 分别有1个、2个、5个、10个、3个、8个边缘点, 直径分别为0、1、2、2、7、4。

[6-10] 考查图的特例——树  $T$ 。

- a) 试证明: 【Jordan, 1869】  $T$  的中心点或者唯一, 或者是一对相邻顶点。

## 【解答】

对树的规模 (亦即顶点数目) 做数学归纳。图x6.4中的(a)和(b), 即是归纳基。

以下考查至少包含3个顶点的树  $T$  (比如图x6.4(e))。不难看出,  $T$  至少有两匹叶子, 也至少有一个内部顶点。因此, 若令  $\hat{T}$  为从  $T$  中剪除所有叶子之后所剩余的子树, 则  $\hat{T}$  必然非空。

接下来我们注意到，在（包括 $T$ 在内的）任何一棵树中，每个顶点的偏心率都是由某个叶顶点决定的。也就是说，对于任何顶点 $v, u \in T$ ，若 $\epsilon(v) = \text{dist}(v, u)$ ，则 $u$ 必是叶子（否则 $\epsilon(v)$ 应更大）。因此 $\hat{T}$ 中各顶点的偏心率，相对于此前在 $T$ 中将至少减少一个单位。而从另一方面看，此前在 $T$ 中不含叶子的每一条路径，都将在 $\hat{T}$ 中继续完整地保留。因此 $\hat{T}$ 中各顶点的偏心率，相对于此前在 $T$ 中将至多减少一个单位。

综合而言， $\hat{T}$ 中各顶点的偏心率必然恰好减少一个单位。因此， $T$ 中偏心率最小的顶点（中心点），必然对应于 $\hat{T}$ 中偏心率最小的顶点（中心点）。

实际上，以上也给出了一个确定半径的算法。

**b) 试证明：** $\text{diameter}(T) = 2 \cdot \text{radius}(T)$ ，或者， $\text{diameter}(T) = 2 \cdot \text{radius}(T) - 1$

【解答】

只需注意到：按照以上证明中的归纳过程，每次从 $T$ 归纳到 $\hat{T}$ ，直径都恰好减少两个单位。

不难看出，这两种可能分别对应于中心点是否唯一。

由此也可推知：树的直径必然就是最长的通路；该通路必然经过中心点。

**c) 试证明：**在从任一顶点 $v$ 出发的BFS过程中，最后一个（出队并）接受访问的顶点 $u$ 必是边缘点。

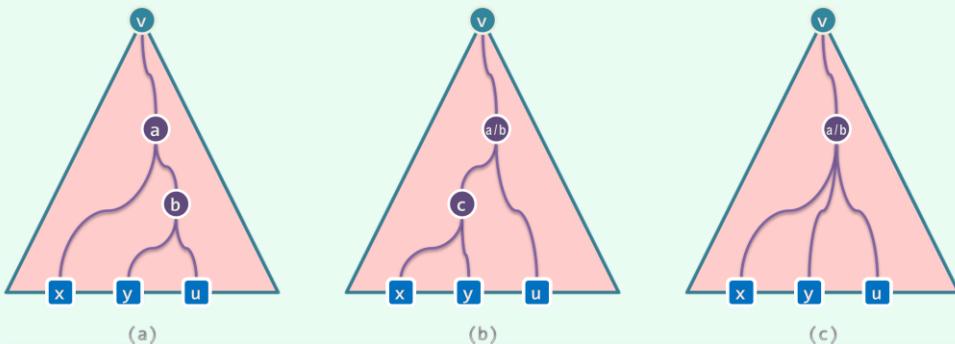
【解答】

根据习题[6-7]的结论，有：

$$\epsilon(v) = \text{dist}(v, u)$$

故若 $v$ 本身就是一个边缘点，则实现其偏心率的 $u$ 必然也是。因此以下仅需考查 $v$ 非边缘点的情况，此时各节点的偏心率都实现于通往某个叶顶点的通路。

任取实现直径的一对边缘点 $x$ 和 $y$ ，二者均应为叶子。设 $u$ 与 $x$ 的最低公共祖先为 $a$ ，与 $y$ 的最低公共祖先为 $b$ ；且不失一般性地，设 $a$ 是 $b$ 的祖先（但未必是 $v$ ）。



图x6.5 无论如何，BFS在树中最后访问的顶点 $u$ 必是边缘点（还有一些情况，比如 $a$ 可能就是 $v$ ）

尽管如图x6.5所示存在多种可能的情况，但依然根据习题[6-7]的结论可知总有：

$$\max\{\text{dist}(a, x), \text{dist}(a, y)\} \leq \text{dist}(a, u)$$

并进一步地有：

$$\text{diameter}(G) \leq \text{dist}(x, a) + \text{dist}(a, y) \leq \text{dist}(x, a) + \text{dist}(a, u) = \text{dist}(x, u) \leq \epsilon(u)$$

这就意味着， $u$ 的确是一个边缘点。

d) 试设计并实现一个算法，在 $O(n)$ 时间内确定 $T$ 的直径。

【解答】

可以先从任一顶点 $v$ 做一趟BFS，找到与之距离最远的顶点 $u$ ；然后从 $u$ 出发再做一趟BFS，找到与之距离最远的顶点 $w$ 。根据以上已证明的性质， $uw$ 之间的通路即对应于树 $T$ 的直径。

若 $T$ 已表示为有根树形式（比如第5.1.3节中的“长子-兄弟”法），则亦可通过递归求解。

在b)中我们已经推知：树的直径必然就是最长的通路（尽管有环图未必具有这一性质）。这样的最长通路，无非两种可能：

- 1) 它不经过 $v$ ，从而完全属于 $v$ 的某棵子树；或者
- 2) 它经过 $v$ ，且联接于最高的两棵子树中各自最深的叶子之间。而此时的直径，应在这两棵子树高度之和的基础上，再加上2。

无论如何，只要递归地得到了各棵子树的高度及直径，即可进而得到树 $v$ 的高度及直径。

读者可在教材所提供的示例代码的基础上，独立完成以上算法的编码和调试任务。

[6-11] 试基于深度优先搜索的框架设计并实现一个算法，在 $O(n + e)$ 时间判定任一无向图是否存在欧拉环路；并且在存在时，构造出一条欧拉环路。

【解答】

根据图论的基本结论，只需遍历全图确定其连通性，再核对各顶点的度数。若连通且没有奇度数的顶点，则必然存在欧拉环路；否则，不存在欧拉环路。其中特别地，若奇度数的顶点恰有两个，则必然存在以这两个顶点为起点、终点的欧拉通路。

构造欧拉环路的一种算法，过程大致如下。从任一顶点出发做一趟DFS，依次记录沿途经过的各边并随即将其从图中删除；一旦有顶点度数归零，则随即将其删除。每当回到起点，即得到一条欧拉子环路。此时若还存在已访问但尚未被删除的顶点，则任选其一并从它出发，再做一趟DFS，过程相同。每次所新得的子环路，都需要在搜索的起始点处与此前的子环路合并为一条更大的子环路。最终不剩任何顶点时，算法结束，当前的子环路即为原图的一条欧拉环路。

如果采用邻接表实现图结构，则以上算法中的每一基本操作（访问或删除当前顶点的一条关联边、访问度数非零的顶点、删除度数为零的顶点、将两条子环路在公共顶点处合并等）都可以在常数时间内完成，故总体运行时间线性正比于原图自身的规模。

上述关于欧拉环路存在性的判定依据以及环路的构造算法，不仅适用于无向图，实际上也不难推广至有向图。

128

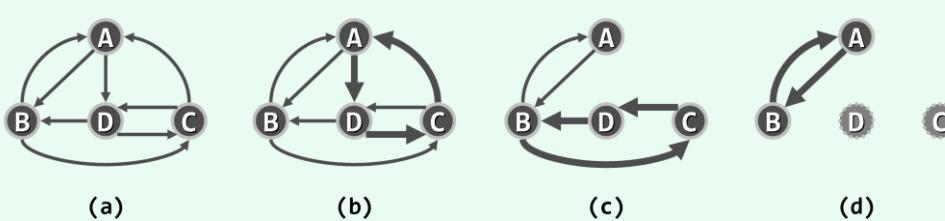


图6.6 构造有向图的欧拉环路：各子环路加粗示意，删除的边不再画出，删除的顶点以灰色示意

考查如图x6.6(a)所示的有向图实例——也就是教材152页图6.4(a)中的实例。

首先，经核对确认各顶点的出、入度数分别相等，故可判定该有向图存在欧拉环路。

接下来，从任一顶点出发做深度优先搜索。比如，若从顶点C出发，可能如图(b)所示得到一条子环路：

{ C A D }

删除已访问过的边，并继续从顶点C出发做深度优先搜索，即可能如图(c)所示得到子环路：

{ C D B }

并与上一子环路在顶点C处合并为：

{ C D B C A D }

删除已访问过的边，并删除度数为零的顶点C和D之后，继续从已经访问但尚未删除的任一顶点出发做深度优先搜索。实际上此时只能从顶点B出发，得到子环路{ B, A }，并与上一子环路在顶点B处合并为：

{ C D [ B A ] B C A D }

#### [6-12] BFS 算法（教材 160 页代码 6.3）的边分类，采用了简化的策略：

**树边（TREE）之外，统一归为跨边（CROSS）**

**试分别针对无向图和有向图，讨论跨边的可能情况。**

#### 【解答】

根据此前的分析，无向图中任意一对邻接顶点在BFS树中的深度之差至多为1。因此在经过广度优先搜索之后，无向图的各边无非两类：树边，亦即被BFS树采用的边；跨边，亦即联接于来自不同分支、深度相同或相差一层的两个顶点之间的边。

类似地，有向图中的每一条边(v, u)均必然满足：

$$\text{depth}(u) \leq \text{depth}(v) + 1$$

这一不等式取等号时，(v, u)即是（由v指向u的）一条树边。

若满足：

$$\text{depth}(u) = \text{depth}(v)$$

则v和u在BFS树中分别属于不同的分支，(v, u)跨越于二者之间。

若满足：

$$\text{depth}(u) < \text{depth}(v)$$

则在BFS树中，u既可能与v属于不同的分支，也可能就是v的祖先。

#### [6-13] 考查采用 DFS 算法（教材 162 页代码 6.4）遍历而生成的 DFS 树。试证明：

##### a) 顶点 v 是 u 的祖先，当且仅当

$$[\text{dTime}(v), \text{fTime}(v)] \supseteq [\text{dTime}(u), \text{fTime}(u)]$$

#### 【解答】

先证明“仅当”。若v是u的祖先，则遍历过程的次序应该是“v被发现...u被发现...u被访问完毕...v被访问完毕”。也就是说，u的活跃期包含于v的活跃期中。由此也可得出一条推论：在任一顶点v刚被发现的时刻，其每个后代顶点u都应处于UNDISCOVERED状态。

反之，若 $u$ 的活跃期包含于 $v$ 的活跃期中，则意味着当 $u$ 被发现（由UNDISCOVERED状态转入DISCOVERED状态）时， $v$ 应该正处于DISCOVERED状态。因此， $v$ 既不可能与 $u$ 处于不同的分支，也不可能 $u$ 的后代。故“当”亦成立。

实际上由以上分析可进一步看出，此类顶点活跃期之间是严格的包含关系。

### b) $v$ 与 $u$ 无承袭关系，当且仅当

$$[\text{dTime}(v), \text{fTime}(v)] \cap [\text{dTime}(u), \text{fTime}(u)] = \emptyset$$

#### 【解答】

作为a)的推论，“当”显然成立，故只需证明“仅当”。

考查没有承袭关系的顶点 $v$ 与 $u$ ，且不妨设 $\text{dTime}(v) < \text{dTime}(u)$ 。于是根据a)，只需证明 $\text{fTime}(v) < \text{dTime}(u)$ 。

若不然( $\text{dTime}(u) < \text{fTime}(v)$ )，则意味着当 $u$ 被发现时， $v$ 应该仍处于DISCOVERED状态。此时，必然有一条从 $v$ 通往 $u$ 的路径，且沿途的顶点都处于DISCOVERED状态。此时在DFS()算法的函数调用栈中，沿途各顶点依次分别存有一帧。在DFS树中，该路径上的每一条边都对应于一对父子顶点，故说明 $u$ 是 $v$ 的后代——这与假设矛盾。

**[6-14] 在起始于顶点s的DFS搜索过程中的某时刻，设当前顶点为v。试证明：任一顶点u处于DISCOVERED状态，当且仅当u来自s通往v的路径沿途——或者等效地，在DFS树中u必为v的祖先。**

#### 【解答】

由题所述条件，可知必有：

$$\text{dTime}(u) < \text{dTime}(v) < \text{fTime}(u)$$

于是由以上根据顶点活跃期之间相互包含关系的结论，必有：

$$\text{dTime}(u) < \text{dTime}(v) < \text{fTime}(v) < \text{fTime}(u)$$

亦即：

$$[\text{dTime}(v), \text{fTime}(v)] \subseteq [\text{dTime}(u), \text{fTime}(u)]$$

故知 $u$ 必为 $v$ 的祖先。

由以上规律亦可进一步推知：起始顶点 $s$ 既是第一个转入DISCOVERED状态的，也是最后一个转入VISITED状态的，其活跃期纵贯整个DFS()算法过程的始末；在此期间的任一时刻，任何顶点处于DISCOVERED状态，当且仅当它属于从起始顶点 $s$ 到当前顶点 $v$ 的通路上。

从类比的角度可以看出，这一通路的作用，正相当于第4.4.1节所介绍的忒修斯的线绳。

**[6-15] 通过显式地维护一个栈结构，将DFS算法（教材162页代码6.4）改写为迭代版本。**

#### 【解答】

实际上，这里引入的栈结构，只需动态记录从起始顶点 $s$ 到当前顶点之间通路上的各个顶点，其中栈顶对应于当前顶点。每当遇到处于UNDISCOVERED状态的顶点，则将其转为DISCOVERED状态，并令其入栈；一旦当前顶点的所有邻居都不再处于UNDISCOVERED状态，则将其转为VISITED状态，并令其出栈。



也就是说，各边权重均有所增加，且增量构成以 $1/2$ 为公比的几何级数，其总和不过：

$$1/2 + 1/2^2 + 1/2^3 + \dots + 1/2^e < 1$$

同时，即便 $W$ 中可能存在等权的边，在如此构造的 $W'$ 中各边的权重同样必然互异。因此与上一方法同理，亦可以消除最小支撑树构造过程中的歧义。

然而不幸的是，这种“方法”要求计算机的数值精度达到 $1/2^e$ ——按指数级的速度随边数 $e$ 提高，或者等效地，其数位应按线性级速度随 $e$ 增加。也就是说，随着 $e$ 的增大，计算机的字长很快就会溢出。而反观上一方法，数值精度为 $1/e^2$ ，相对而言不致轻易即溢出。

### b) 这种方法可否推广至实数权重的网络？

#### 【解答】

以上方法之所以行之有效，是因为事先能够在不等权的边之间，确定边权重的最小差值(1)，从而既能够保证 $W'$ 中的各权重彼此互异，同时又能保证通过向下取整运算，可以从 $|T'|$ 确定对应的 $|T|$ 。若权重可以取自任意实数，则这二个性质不能直接兼顾。

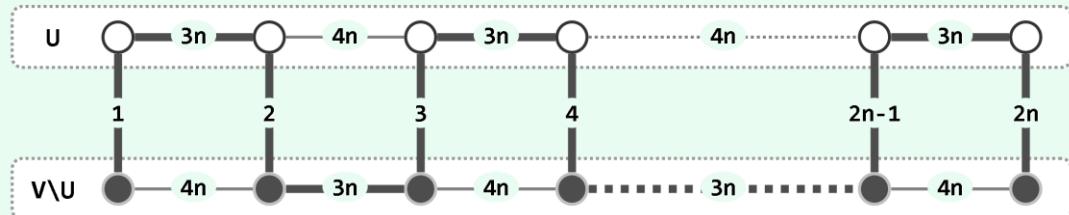
当然，若各边权重均取自浮点数（正如实际计算环境中的情况），则仍可以套用上述方法，只不过需要做必要的预处理——通过统一的放缩，将各边的权重转换为整数。

[6-18] 在教材 176 页图 6.20 中，出于简洁的考虑，将通路  $us$  和  $vt$  分别画在构成割的两个子图中。然而这样有可能造成误解，比如读者或许会认为，组成这两条通路的边也必然分别归属于这两个子图。

试举一实例说明， $us$  或  $vt$  均可能在两个子图之间穿越多（偶数）次——亦即，除了该割的最短跨越边  $uv$ ，最小支撑树还可能采用同一割的其它跨越边，其长度甚至可能严格大于 $|uv|$ 。

#### 【解答】

一个（组）通用的实例如图x6.7所示。



图x6.7 最小支撑树（粗线条）可能反复地穿过于割的两侧

这里的子集 $U$ 包含 $2n$ 个顶点（白色），其中 $2n - 1$ 条非跨越边的权重依次为：

$$W_U = \{ 3n, 4n, 3n, 4n, \dots, 3n \}$$

补集 $V \setminus U$ 也包含 $2n$ 个顶点（黑色），其中 $2n - 1$ 条非跨越边的权重依次为：

$$W_{V \setminus U} = \{ 4n, 3n, 4n, 3n, \dots, 4n \}$$

在这两个子集之间，共有 $2n$ 条跨越边，其权重依次为：

$$X = \{ 1, 2, 3, 4, \dots, 2n - 1, 2n \}$$

不难验证，该图的最小支撑树唯一确定，由权重不超过 $3n$ 的所有 $4n - 1$ 条边组成，亦即图x6.7中所有的粗边。由图易见，该支撑树就是一条通路，它在割的两侧总共穿越 $2n$ 次——（权重为1的）最短跨越边只是其中之一。

实际上， $X$ 中各跨越边的权重未必需要按次序排列。

另外，基于二部图，完全可以构造出更为精简的实例。读者不妨照此思路，独立尝试。

**[6-19] 利用“有向无环图中极大顶点入度必为零”的性质，实现一个拓扑排序算法，若输入为有向无环图，则给出拓扑排序，否则报告“非有向无环图”。该算法时间、空间复杂度各是多少？**

**【解答】**

基于这一策略的拓扑排序算法，过程大致如算法x6.1所示。

```
1 将所有入度为零的顶点存入栈S //O(n)
2 取空队列Q //记录拓扑排序序列，O(1)
3 while ( ! S.empty() ) { //O(n)
4     Q.enqueue( v = S.pop() ); //栈顶顶点v转入队列Q
5     for each edge(v, u) //考查v的所有邻接顶点u
6         if ( inDegree(u) < 2 ) S.push(u); //凡入度仅为1者，均压入栈S中
7     G = G \ {v}; //删除顶点v及其关联边（邻接顶点入度相应地递减）
8 } //总体O(n + e)
9 return |G| ? Q : "NOT_DAG"; //残留的G空（Q覆盖所有顶点），当且仅当原图可拓扑排序（Q即是排序序列）
```

#### 算法x6.1 基于“反复删除零入度节点”策略的拓扑排序算法

这里，栈S和队列Q的初始化共需 $O(n)$ 时间。主体循环共计迭代 $O(n)$ 步，其中涉及的操作无非以下五类：出、入栈，共计 $O(n)$ 次；入队，共计 $O(n)$ 次；递减邻接顶点的入度，共计 $O(e)$ 次；删除（零入度）顶点，共计 $O(n)$ 个；（删除顶点时一并）删除关联边，共计 $O(e)$ 条。以上各类操作均属于基本操作，故总体运行时间为 $O(n + e)$ ，线性正比于原图的规模。

空间方面，除了原图本身，这里引入了辅助栈S和辅助队列Q，分别用以存放零入度顶点和排序序列。不难看出，无论是S或Q，每个顶点从始至终至多在其中存放一份，故二者的规模始终不超过 $O(n)$ 。实际上，通过更进一步地观察还可以发现，S和Q之间在任何时刻都不可能有公共顶点，因此二者总体所占的空间亦不过 $O(n)$ 。

请注意，既然不是基于DFS，便无需维护各顶点的时间标签及状态、各边的分类。因此相对于基于DFS的拓扑排序算法而言，这一实现方式所需的附加空间更少——对稠密图而言尤其如此。

**[6-20] a) 试从教材 167 页代码 6.5 中，删除与拓扑排序无关的操作，以精简其实现；**

**【解答】**

仅就拓扑排序而言，这里并不需要对各边做分类，也不必记录各顶点的时间标签，相关的数据项及操作均可省略。顶点的状态虽然仍需区分，但只要足以判别顶点是否已经访问即可。相应地，循环体内的三个switch分支也只需保留一个。请读者按照这些提示，独立完成精简工作。

**b) 精简之后，整体的渐进复杂度有何变化？**

**【解答】**

经以上精简之后，运行时间虽有所减少，但渐进的复杂度依然保持为 $O(n + e)$ 。

空间方面，因为不必维护各边的分类标签，故除原图本身外仅需使用 $O(n)$ 辅助空间。

[6-21] a) 试从教材 170 页代码 6.6 中，删除与双连通分量分解无关的操作，以精简其实现；

【解答】

仅就双连通分量分解这一问题而言，这里并不需要对各边进行分类，相关的数据项及操作均可删除。各顶点的时间标签（其中的`fTime`用作`hca`）和状态，都仍然需要记录。

请读者按照以上思路，独立完成精简工作。

b) 精简之后，整体的渐进复杂度有何变化？

【解答】

经以上精简之后，运行时间虽有所减少，但渐进的复杂度依然保持为 $O(n + e)$ 。

空间方面，因为不必维护各边的分类标签，故除原图本身外仅需使用 $O(n)$ 辅助空间。

[6-22] 试按照 PFS 搜索的统一框架（教材 173 页代码 6.7），通过设计并实现对应的 `prioUpdater` 函数对象，分别实现 BFS 和 DFS 算法。

【解答】

BFS 算法对应的优先级更新器，可实现如代码 x6.1 所示。



```

1 template <typename Tv, typename Te> struct BfsPU { //针对BFS算法的顶点优先级更新器
2     virtual void operator() ( Graph<Tv, Te>* g, int uk, int v ) {
3         if ( g->status ( v ) == UNDISCOVERED ) //对于uk每一尚未被发现的邻接顶点v
4             if ( g->priority ( v ) > g->priority ( uk ) + 1 ) { //将其到起点的距离作为优先级数
5                 g->priority ( v ) = g->priority ( uk ) + 1; //更新优先级(数)
6                 g->parent ( v ) = uk; //更新父节点
7             } //如此效果等同于，先被发现者优先
8     }
9 };

```

代码x6.1 基于PFS框架的BFS优先级更新器

例如，若有某个图结构`g`的顶点为`char`类型、边为`int`类型，则可以通过如下形式的调用，基于PFS框架完成对`g`的BFS搜索。

```
g->pfs( 0, BfsPU<char, int>() );
```

与Dijkstra算法的顶点优先级更新器`DijkstraPU()`（教材179页代码6.9）做一对比即可看出，此处`BfsPU()`的作用，只不过是将 $u_k$ 到其邻接顶点 $v$ 的距离统一取作：

```
g->weight(uk, v) = 1
```

也就是说，所谓的BFS实际上完全等效于，在所有边的权重统一为1个单位的图中，采用Dijkstra算法构造最短路径树。从这个意义上讲，BFS也可以视作Dijkstra算法的一个特例。

类似地，DFS算法对应的优先级更新器，亦可实现如代码x6.2所示。



**[6-24] 合成数 ( composite number ) 法，是消除图算法歧义性的一种通用方法。**

首先在顶点的标识之间，约定某一次序。比如，标识为整数或字符时，可直接以整数或字符为序；对于字符串类标识，可按字典序。于是，对于权重为  $w$  的边  $(v, u)$ ，合成数可以取作向量： $(w, \min(v, u), \max(v, u))$ 。如此，任何两条边总能明确地依照字典序比较大小。

试在 6.11.5 节 Prim 算法和 6.12.2 节 Dijkstra 算法中引入这一方法，以消除其中的歧义性。

**【解答】**

采用这一方法，实质的调整无非只是比较器的重新定义，算法的整体框架及流程均保持不变。具体地，也就是将原先各边的权重，替换为其对应的合成数，并按照字典序判定各边的优先级。

反观 Prim 算法。在最短跨越边同时存在多条时，该算法可以任取其中之一。虽然如此必然能够构造出一棵最小支撑树，但却不能保证其唯一确定性。

对于所有边的权重为整数或浮点数的情况，此前介绍过通过扰动使之唯一确定化的技巧。按照这一方法，输入序列中各边的扰动量严格单调递增，故在扰动之后，原先权重相等的边必然可以按照“前小后大”的准则判定相对的优先级。从这个角度来看，其效果完全等同于将各边的权重依次替换为如下合成数： $W' = \{(w_1, 1), (w_2, 2), (w_3, 3), \dots, (w_e, e)\}$

请注意，这一方法不再局限于整数或浮点数的权重，因此适用范围更广。

**[6-25] 考查某些边的权重不是正数的带权网络。试证明：**

a) 对此类网络仍可以定义最小支撑树——此时，Prim 算法是否依然可行？

**【解答】**

依然可行。为此首先需要确认，含负权边的带权网络依然拥有最小支撑树。设带权网络  $G$  中各边权重的最小值为  $-\delta < 0$ ，现在令  $G$  中所有边的权重统一地增加  $2\delta$ ，得到另一带权网络  $G'$ 。

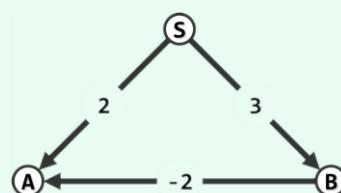
不难看出， $G$  和  $G'$  的支撑树必然一一对应。另外，二者的每一棵支撑树都应恰好包含  $v - 1$  条边，其中  $v$  为顶点总数。因此就总权重而言， $G$  的每一棵支撑树与  $G'$  所对应的支撑树之间均相差一个常数： $2\delta \cdot (v - 1)$ 。既然  $G'$  中各边权重均为正数，故其最小支撑树必然存在；而与之对应的，即为原网络  $G$  的最小支撑树。

b) 若不含负权重环路，则仍可以定义最短路径树——此时，Dijkstra 算法是否依然可行？

**【解答】**

任意两点之间的通路数目有限，其中最短者必然存在，故最短路径树必然存在。但如图x6.9 所示，Dijkstra 算法在此时却未必依然可行。

一种补救的方法是，在 PFS 优先级更新器 DijkstraPU（代码 6.9）中，不再忽略非 UNDISCOVERED 状态的顶点；而且一旦它们的优先级经更新所有增加，都要随即将其恢复为 UNDISCOVERED 状态，并进而参与下一顶点  $u_k$  的竞选。然而不难看出，时间复杂度也会因而提高。



图x6.9 含负权边时，即便不存在负权环路，Dijkstra算法依然可能出错：设起点为  $S$ ，算法将依次确定  $A$ 、 $B$  对应的最短距离分别为  $2$ 、 $3$ 。而实际上，从  $S$  经  $B$  通往  $A$  的距离为  $3 + (-2) = 1 < 2$ 。这里的关键在于，在顶点  $A$  所对应的最短路径上，有另一顶点  $B$  被算法发现得更晚。

[6-26] 各边权重未必互异时，带权网络的“最小生成树”未必唯一，故应相应地，将其改称作“极小支撑树”更为妥当。对于任一此类的带权网络G，试证明：

a) 每一割的极短跨越边都会被G的某棵极小支撑树采用；

【解答】

教材176页图6.20已在“各边权重互异”的前提下，证明了Prim算法的正确性。在废除这一前提之后，证明的技巧依然类似。

反证。如该图所示，假设uv是割( $U : V \setminus U$ )的极短跨越边（之一），但uv却未被任何极小支撑树采用。任取一棵极小支撑树T，则T至少会采用该割的一条跨越边st。于是同理，将st替换为uv，将得到另一棵支撑树T'，而且其总权重不致增加。这与假设相悖。

b) G的每棵极小支撑树中的每一条边，都是某一割的极短跨越边。

【解答】

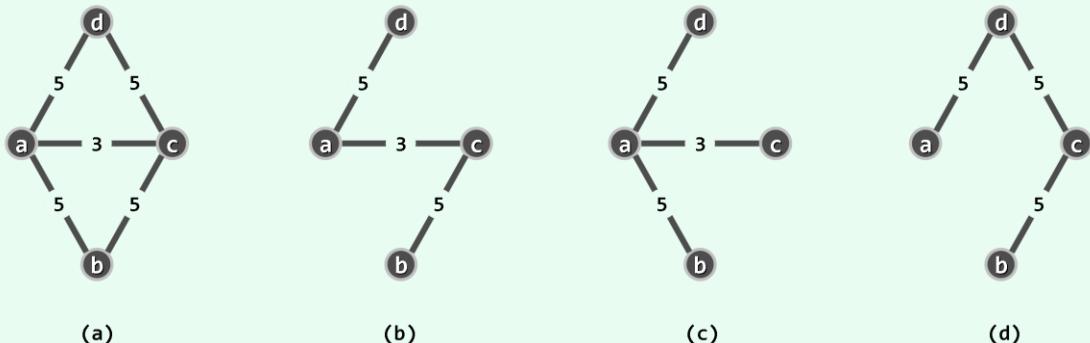
任取G的一棵极小支撑树T，考查其中的任何一条树边uv。将该边删除之后，T应恰好被分成两棵子树，它们对应的两个顶点子集也构成G的一个割( $U : V \setminus U$ )。

实际上，uv必然是该割的极短跨越边（之一）。否则，与a)同理，只需将其替换为一条极短跨边st，即可得到一棵总权重更小的支撑树T'——这与T的极小性矛盾。

[6-27] 试举例说明，在允许多边等权的图G中，即便某棵支撑树T的每一条边都是G某一割的极短跨越边，T也未必是G的极小支撑树。

【解答】

考查如图x6.10(a)所示的带权网络G。



图x6.10 完全由极短跨越边构成的支撑树，未必是极小的

首先不难验证，每一条边都是G某一割的极短跨越边：ac为全图的最短边；ad则为割：

$$(\{a, b, c\} : \{d\})$$

的极短跨边（之一）——根据对称性，其余权重同为5的各边亦是如此。

既然该网络的支撑树都由三条边组成，故如图(b)和(c)所示，总权重为 $3 + 5 + 5 = 13$ 的支撑树必然是极小的。然而如图(d)所示，同样亦由三条极短跨边构成的支撑树，总权重却为15，显然并非极小支撑树。

[6-28] 试证明，尽管在允许多边等权时，同一割可能同时拥有多条最短跨过边，6.11.5 节中 Prim 算法所采用的贪心迭代策略依然行之有效。

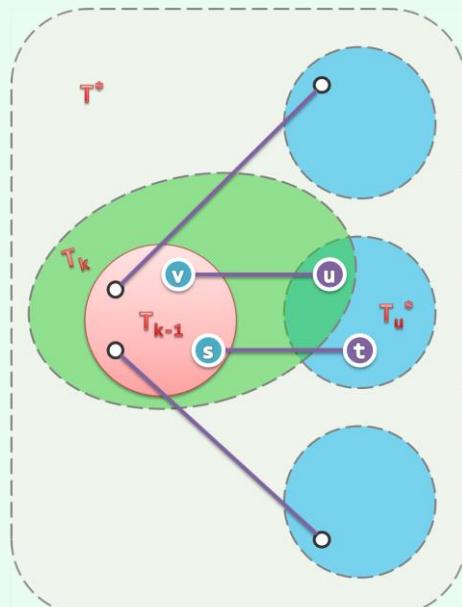
(提示：只需证明，只要  $T_k$  是某棵极小支撑树的子树，则  $T_{k+1}$  也必是（尽管可能与前一棵不同的）某棵极小支撑树的子树。)

【解答】

任取一棵极小支撑树  $T^* = (V, E^*)$ ，以下采用数学归纳法证明：

对于 Prim 算法过程中所生成的每一棵（子）树  $T_k$ ，都可以在总权重不致增加的前提下，将  $T^*$  转换为其一棵超树

果真如此，则意味着 Prim 算法最终生成的  $T_n$  也是一棵极小支撑树。



图x6.11 在各边权重未必互异时，  
Prim算法依然正确

作为归纳基，以上命题对于  $T_1$  显然成立。

以下，假定上述命题对于  $T_1, \dots, T_{k-1}$  ( $2 \leq k$ ) 均成立。如图x6.11所示，考查：

$T_k = (V_{k-1} \cup \{u\}, E_{k-1} \cup \{vu\})$   
且不妨设边  $vu \notin E^*$ 。

既然  $T^*$  是  $T_{k-1}$  的超树，故在将后者从前者中删除之后， $T^* \setminus T_{k-1}$  应该是个非空的森林。将顶点  $u$  在其中所属的子树记作  $T_u^*$ 。在  $T^*$  中，子树  $T_{k-1}$  与子树  $T_u^*$  之间必然有且仅有一条边相联，设为  $st$ （有可能  $s = v$  或  $t = u$ ，但是不能同时成立）。

现在，在  $T^*$  中将  $st$  替换为  $vu$ 。经如此转换之后， $T^*$  的连通性和无环性依然满足，故仍是原图的一棵支撑树。就权重而言， $T^*$  新、旧两个版本之间的差异为  $|vu| - |st|$ 。鉴于 Prim 算法挑选的  $vu$  必然是极短跨过边，故新版本的权重不致增加（当然，也不可能下降）。由此可见，归纳假设对  $T_k$  也依然成立。

[6-29] Joseph Kruskal 于 1956 年<sup>[31]</sup> 提出了构造极小支撑树的另一算法：

将每个顶点视作一棵树，并将所有边按权重非降排序；  
依次考查各边，只要其端点分属不同的树，则引入该边，并将端点所分别归属的树合二为一；  
如此迭代，直至累计已引入  $n - 1$  条边时，即得到一棵极小支撑树。

试证明：

a) 算法过程中所引入的每一条边，都是某一割的极短跨过边（因此亦必属于某棵极小支撑树）；

【解答】

考查 Kruskal 算法每一步迭代中所引入的边  $vu$ 。在此步迭代即将执行之前， $v$  必属于当时森林中的某棵子树，将其记作  $T_v$ 。 $T_v$  及其补集，构成原图的一个割。不难看出， $vu$  既然（作为当时不致造成环路的极短边）被选用，则它必然也是该割的极短跨过边。

b) 算法过程中的任一时刻，由已引入的边所构成的森林，必是某棵极小支撑树的子图；  
 (请注意，这一结论方足以确立 Kruskal 算法的正确性，但前一结论并不充分)

【解答】

任取一棵极小支撑树  $T^* = (V, E^*)$ ，以下采用数学归纳法证明：

对于 Kruskal 算法过程中的每一个森林  $F_k$

都可以在总权重不致增加的前提下，将  $T^*$  转换为其一棵超树

如果这是事实，则意味着 Kruskal 算法最终生成的  $T_n = F_n$  也是一棵极小支撑树。

作为归纳基，以上命题对于  $F_0$  显然成立。

以下，设上述命题对于  $F_0, \dots, F_{k-1}$  ( $1 \leq k$ ) 均成立。如图 x6.12 所示，考查：

$$F_k = (V, E_{k-1} \cup \{vu\})$$

且不妨设边  $vu \notin E^*$ 。

将  $v$  和  $u$  所属的子树分别记作  $T_v$  和  $T_u$ 。既然  $T^*$  是  $F_{k-1}$  的超树，故在  $T^*$  中， $T_v$  和  $T_u$  之间必存在（唯一的）通路。该通路可能就是  $vu$  之外的另一跨越边，也可能是辗转穿过其它子树的一条迂回通路。无论如何，如图所示将该通路的起始边设为  $st$ 。

现在，若将  $st$  替换为  $vu$ ，则  $T^*$  依然连通且无环，故仍是原图的一棵支撑树。

$T^*$  的新、旧两个版本，权重的差异为  $|vu| - |st|$ 。鉴于 Kruskal 算法挑选的  $vu$  必然是极短跨越边，故新版本的权重不致增加（当然，也不致下降）。可见，归纳假设对  $F_k$  也依然成立。

[6-30] 试举例说明，在最坏情况下，Kruskal 算法的确可能需要检查  $\Omega(n^2)$  条边。

【解答】

一类最坏的情况，如图 x6.13 所示。

该实例中，原网络中的  $n - 1$  个顶点构成完全图  $K_{n-1}$ ，其中各边的权重相对不大；最后一个顶点  $u$ ，则仅通过一条权重足够大的边  $vu$  与完全图相联。

若将  $(\{u\} : K_{n-1})$  视作割，则  $vu$  是唯一的跨越边。因此，尽管该边的权重在全局最大，但该带权网络的任何一棵极小支撑树  $T^*$ ，都必会采用该边。

而按照 Kruskal 算法的策略， $vu$  必然是作为最后一条边接受检查；在此前，该算法也必然已经遍历过  $K_{n-1}$  中所有的边，累计耗时总量为：

$$n(n - 1)/2 + 1 = \Omega(n^2)$$

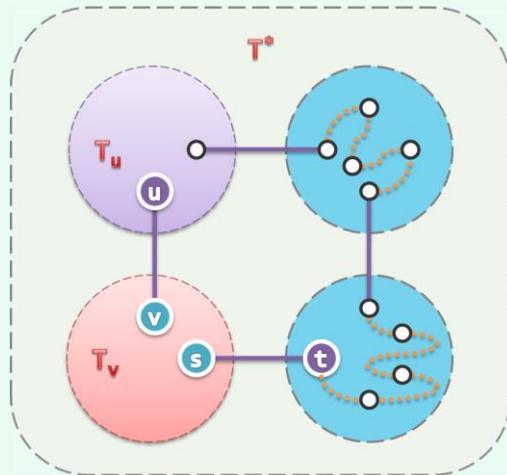


图 x6.12 Kruskal 算法的正确性

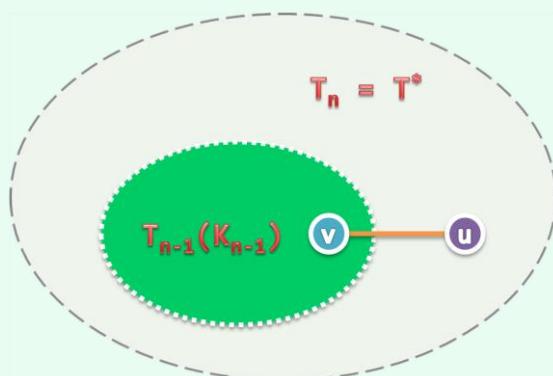


图 x6.13 Kruskal 算法的最坏情况

[6-31] 若将森林中的每棵树视作一个等价类，则 Kruskal 算法迭代过程所涉及的计算不外乎两类：

```
T = find(x) //查询元素(顶点)x所属的等价类(子树)T
union(x, y) //将元素(顶点)y所属的等价类(子树), 合并至元素(顶点)x所属的等价类(子树)
```

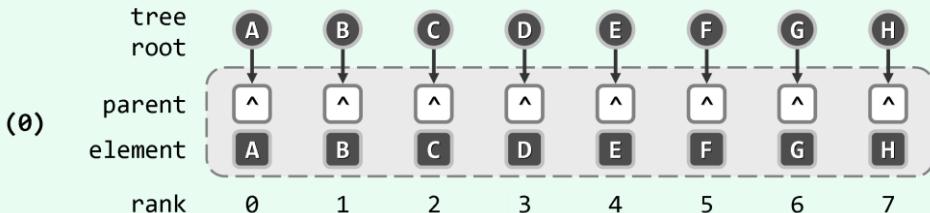
支持以上基本操作接口的数据结构，即所谓的独立集（disjoint set），或者称作并查集（union-find set）<sup>[32][33][34][35]</sup>。

a) 试基于此前介绍过的基本数据结构实现并查集，并用以组织 Kruskal 算法中的森林；

【解答】

并查集中的等价类，可视作某一全集经划分之后得到的若干互不相交的子集。最初状态下，全集中的每个元素自成一个子集，并以该元素作为其标识。此后，每经过一次`union(x, y)`操作，都将元素y所属的子集归入元素x所属的子集，并继续沿用元素x此前的标识。

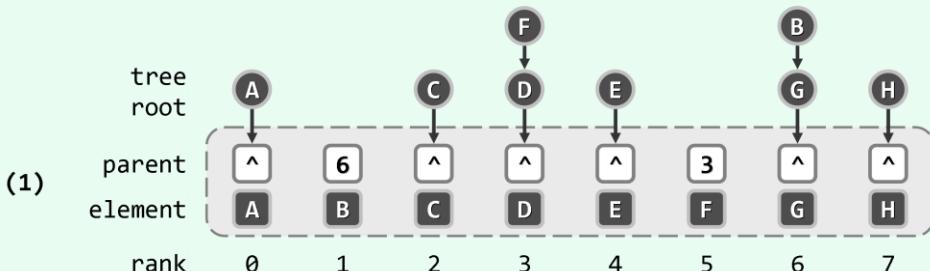
一种可行的方法是仿照父节点表示法（教材112页图5.3），将每个子集组织为一棵多叉树，并令所有多叉树共存于同一个向量之中。如图x6.14所示，即是并查集初始状态的一个实例。



图x6.14 并查集：初始时每个元素逻辑上自成一个子集，并分别对应于一棵多叉树，parent统一取作-1

于是，子集的合并即对应于树的合并。如图x6.15所示，元素所属的子集即是其所属的树，也对应于该树的根，三者始终一致，不必再刻意区分。故不妨约定，就以树根作为各子集的标识。

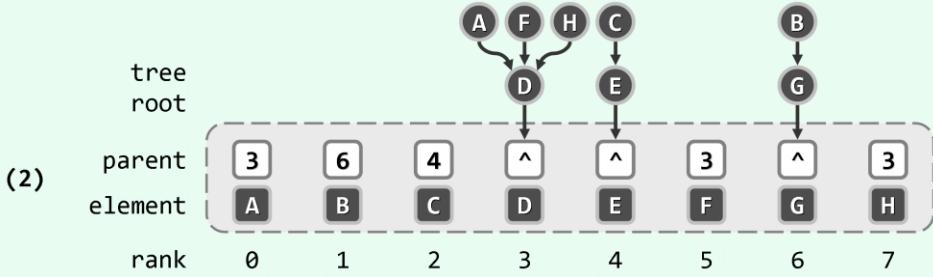
这样，`find(x)`查找问题也便转化为了在多叉树中查找根节点的问题。为此，我们只需从元素x出发，沿着`parent`引用逐层上行，直到x的最高祖先。例如在图x6.14~图x6.18中，`find(F)`将分别返回F、D、D、G和E。



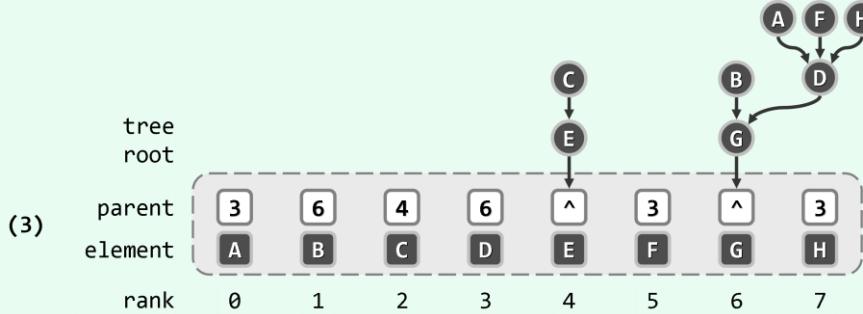
图x6.15 并查集：经过`union(D, F)`和`union(G, B)`，F所属的子集(树)被归入D所属的子集(树)，沿用标识D；B所属的子集(树)被归入G所属的子集(树)，沿用标识G

对以上实例进一步的具体操作结果，依次如图x6.16~图x6.18所示。请留意这里各元素`parent`值的更新，以及树结构的调整与并查集逻辑结构之间的动态对应关系。

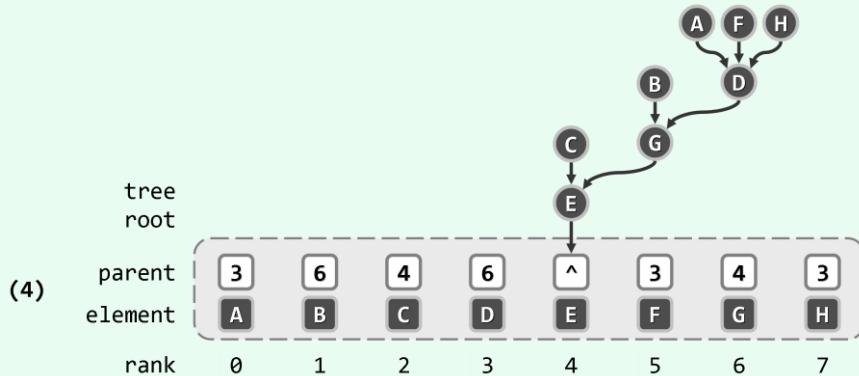
由于`union()`操作并不可逆，故经过n-1次`union()`之后该结构将不会再有实质性的调整。



图x6.16 并查集：再经union(D, A)和union(F, H), A和H被归入子集D中；再经union(E, C), C被归入子集E中



图x6.17 并查集：再经union(B, A), A所属的子集(树)D, 被归入B所属的子集(树)G中



图x6.18 并查集：再经union(C, F), F所属的子集(树)G, 被归入C所属子集(树)E中(树高未能有效控制)

b) 按你的实现，`find()`和`union()`接口的复杂度各是多少？相应地，Kruskal 算法的复杂度呢？

**【解答】**

无论`find()`或`union()`，都需首先确定相关元素所属子集的标识（树根），为此花费的时间量主要取决于元素在树中所处的深度；在最坏情况下，也就是树的高度。然而在目前的实现中，我们并不能有效地控制树高。仍以如图x6.14所示的初始并查集为例，假若接下来依次执行：

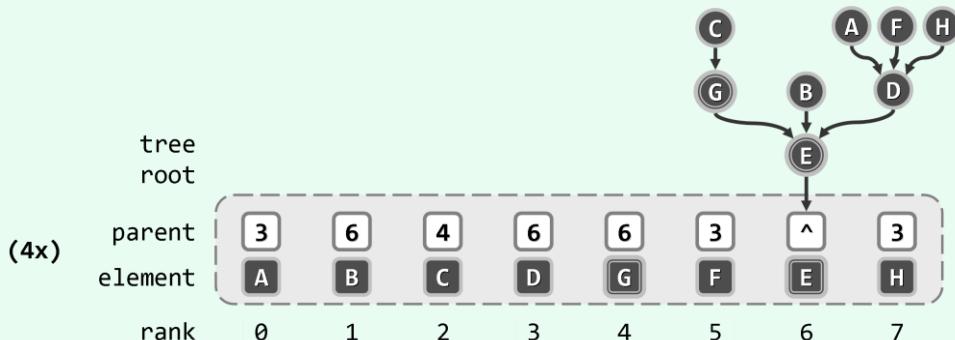
`union(A, B), union(B, C), union(C, D), ..., union(G, H)`

则不难验证，各次`union()`操作所需的时间成本将按算术级数递增。

一般地在最坏情况下，对于包含n个元素的并查集，以上过程共需 $\mathcal{O}(n^2)$ 时间，单次操作平均需要 $\mathcal{O}(n)$ 时间——退化到蛮力算法。相应地，基于该版本并查集实现的Kruskal算法，有可能每步迭代都平均需要线性时间，累计共需 $\mathcal{O}(n \cdot e)$ 时间。

当然，为了有效控制树的高度，完全可以做进一步的改进。比如在树合并时，可采取“低者优先归入高者”的策略。也就是说，比较待合并的树的高度，并倾向于将更低者归入更高者。

仍考查如图x6.17所示的并查集，假设接下来同样要执行 $\text{union}(C, F)$ 。在找出对应的树G和E之后，经比较发现前者更高。故只要与如图x6.18所示的合并方向相反，改为将树E归入树G，则如图x6.19所示，合并之后的树高即可由3降至2。



图x6.19 并查集：经 $\text{union}(C, F)$ 操作之后（“低者归入高者”以控制树高）

当然，为此还需要给每个元素增添一个域，以动态记录其高度：初始统一取作0；合并时若两树高度相等，则合并后树根的高度值相应地递增；若高度不等，则无调整。

最后，为了遵守此前关于子集标识的沿用约定，还有可能需要交换原先的两个树根——比如此例中的元素E和G——包括它们各自的标识以及parent值。

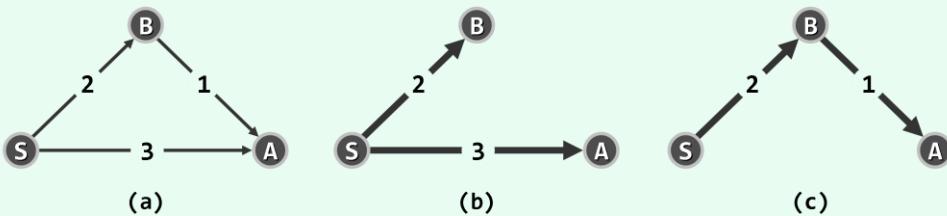
有效控制树高的另一策略是路径压缩（path compression），它源自如下观察事实：

- 1) 就此问题而言，树中元素之间的拓扑联接关系并不重要；
- 2) 因为仅涉及（沿parent引用的）上行查找而无需下行查找，故孩子的数目并不影响查找效率

因此在每次查找的过程中，可将上行通路沿途的元素逐个取出，并作为树根的孩子重新接入树中。如此，树将被尽可能地平坦化，从而进一步地控制树高。请读者按照这些思路完成以上改进。

### [6-32] 试举例说明，即便带权网络中不含权重相等的边，其最短路径树依然可能不唯一。

【解答】



图x6.20 各边权重互异时，最短路径树依然可能不唯一

最简单的一个实例如图x6.20(a)所示。相对于起始顶点S，顶点A有两条最短路径 ( $\{S, A\}$  和  $\{S, B, A\}$ )，相应地如图(b)和(c)所示有两棵最短路径树。

[6-33] 若图G的顶点取自平面上的点，各顶点间均有联边且权重就是其间的欧氏距离，则G的最小支撑树亦称作欧氏最小支撑树（Euclidean Minimum Spanning Tree, EMST），记作EMST(G)。

a) 若套用Kruskal或Prim算法构造EMST(G)，各需多少时间？

【解答】

带权网络G是由平面点集隐式定义的完全图，故若G由n个顶点构成，则所含边数应为：

$$e = n(n - 1)/2 = O(n^2)$$

因此若直接调用（借助优先级队列结构改进之后的）Prim算法，渐进的时间复杂度应为：

$$O((n + e) \cdot \log n) = O(n^2 \log n)$$

而直接调用Kruskal算法，渐进的时间复杂度应为：

$$O(e \log n) = O(n^2 \log n)$$

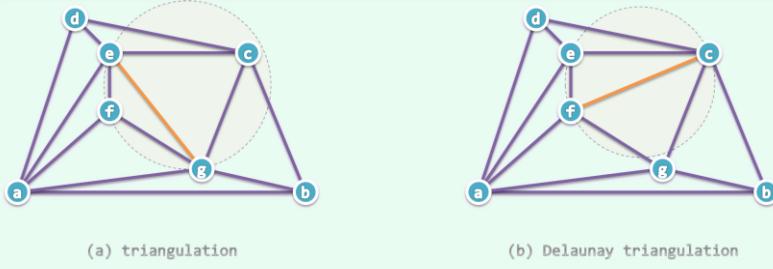
b) 试设计一个算法，在 $O(n \log n)$ 时间内构造出EMST(G)；（提示：Delaunay三角剖分）

【解答】

由上可见，为了能够使用Prim等常规算法，应先设法把由欧氏距离隐式定义的完全图，转化为某种邻近图（proximity graph），从而将候选边的数量从 $\Omega(n^2)$ 减至 $O(n)$ 。

例如，图x6.21(a)即为三角剖分（triangulation），也就是原隐式完全图的任一极大平面子图。因每个子区域都是三角形，故此得名。不难发现，同一点集的三角剖分尽管往往并不唯一确定，但其所保留的边总数固定，所分出的三角形总数已固定。

任一点集都有一个特殊的三角剖分，称作Delaunay三角剖分（Delaunay triangulation）。如图(b)所示，在这种三角剖分中，任意三角形的外接圆都不包含第四个点——亦即所谓的空圆性质（empty-circle property）。反观图(a)，其中三角形efg的外接圆内还有c点，故而不属于Delaunay三角剖分。



图x6.21 平面点集的三角剖分(a)，及其特例Delaunay三角剖分(b)

还可按照以下准则，从Delaunay三角剖分中进一步地剔除若干条边：

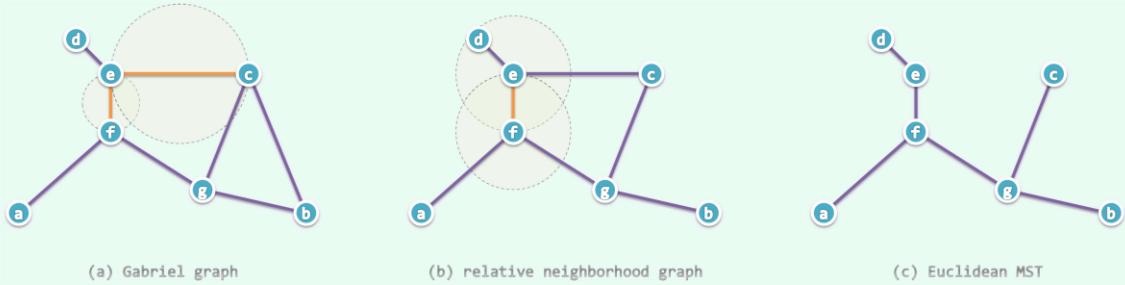
**一条边被剔除，当且仅当以其为直径的圆内（除该边端点外）还包含第三个点**

如此保留下来的子图如图x6.22(a)所示，称作Gabriel图（Gabriel graph）。

进一步地，可以按照以下原则，从Gabriel图中再剔除若干条边：

**一条边被剔除，当且仅当以其为半径、分别以其端点为圆心的两个圆，不会同时包含第三个点**

如此保留下来的子图如图x6.22(b)所示，称作RNG图（relative neighborhood graph）。



图x6.22 平面点集的邻近图：(a)Gabriel图，(b) RNG图，(c) 欧氏最小支撑树

由上可见，Delaunay三角剖分、Gabriel图、RNG图依次构成“超图-子图”的关系。

有趣的是，尽管进一步地删除了更多的边，但RNG图依然是连通的。进一步地还可以证明，如图x6.22(c)所示，欧氏最小支撑树必是RNG图的子图。证明的方法和技巧，与教材第6.11.5节“最小支撑树必然采用极短跨越边”的证明极为相似，我们将此留给读者独立完成。

反过来，既然三角剖分是平面图，故以上介绍的所有邻近图亦是。而根据习题[6-3]的结论，其中边的总数必然与顶点数渐进地相当，也就是从 $\mathcal{O}(n^2)$ 降低到 $\mathcal{O}(n)$ 。

另一好消息是，以上邻近图均可在 $\mathcal{O}(n \log n)$ 的时间内构造出来。因此这里的预处理通常并不会增加整体的渐进时间复杂度。有兴趣的读者，可自学具体的转换算法。

### c ) 试证明你的算法已是最优的（亦即，在最坏情况下，任何此类算法都需要 $\Omega(n \log n)$ 时间）。

**【解答】**

对于任意平面点集 $G$ ，可以定义最近邻图（nearest neighbor graph, NNG）如下：

边 $pq$ 属于该图，当且仅当点 $q$ 在 $G \setminus \{p\}$ 中距离 $p$ 最近

请注意，最近邻图是有向图。也就是说，即便 $q$ 是 $p$ 的最近邻，反之却未必亦然。

考查经典的 $\varepsilon$ -间距问题（ $\varepsilon$ -closeness）：

设 $P$ 为由任意 $n$ 个实数构成的集合，对于任意的 $\varepsilon > 0$ ，判定 $P$ 中是否存在两个实数的差距不大于 $\varepsilon$

该问题的难度，已经证明为 $\Omega(n \log n)$ 。

实际上可以证明，最近邻图NNG必为欧氏最小支撑树EMST的子图。特别地，该NNG图中的最短边，便是点集 $G$ 中的最近点对（nearest pair）。

不难看出， $\varepsilon$ -间距问题可在线性时间内归约至最近点对问题，而最近点对问题又可以进一步地在线性时间内归约至欧氏最小支撑树问题。根据线性归约的传递性， $\varepsilon$ -间距问题可在线性时间内归约至欧氏最小支撑树问题。

于是自然地， $\Omega(n \log n)$ 的复杂度下界，亦适用于欧氏最小支撑树问题。这意味着，在没有其它附加条件的前提下，以上所设计的 $\mathcal{O}(n \log n)$ 算法，已属于最坏情况下最优的。