第3章 VLSI布图的数学基础和数据结构

VLSI版图是一组有规则的由若干层平面几何图形元素组成的集合。通常，这些图形元素只限于曼哈顿(Manhatten)图形(只由垂直边和水平边构成的图形)，且在同一层内不允许重叠。早期，版图是由人工根据电路功能和设计规则产生的。但随着电路的规模和复杂性的增加，人工设计版图越来越困难甚至是不可能。各种各样的计算机辅助设计工具也相应产生，尤其是自动布图工具发展非常迅速。它们采用了各种有效的布图算法和数据结构以完成整个布图过程。与VLSI自动布图算法有关的数学基础包括图论、组合优化、线性规划和有约束的非线性规则、运筹学以及计算复杂性理论等。本章将介绍与这些数学基础有关的基本内容以及布图中所用的数据结构。

3.1 图的基本概念及其基本数据结构

3.1.1 基本概念

1. 图的定义

图(Graph)：由一个用线或边连接在一起的顶点或节点的集合。是一种比线性表和树更为复杂的非线性数据结构，可称为图状结构或网状结构。

图 G 由两个集合 V 和 E 组成，记为，$G=(V,E)$，其中：V 是顶点的有穷非空集合，E 是 V 中顶点偶对(称为边)的有穷集。通常，也将图 G 的顶点集和边集分别记为 $V(G)$ 和 $E(G)$。$E(G)$可以是空集。若 $E(G)$为空，则图 G 只有顶点而没有边。

2. 无向图和有向图

无向图：若边是顶点的无序对，则称为无向图，并用圆括号表示顶点的无序对。在无向图中，边(v_1,v_2)和(v_2, v_1)是同一条边。

有向图：在图 G 中，如果每条边是顶点的有序对，即每条边在图示时用箭头指明了方向，则称 G 为有向图。有向图中的边也称为弧，用一对尖括号表示，如$<v_i,v_j>$，其中：v_i 称为弧尾，v_j 称为弧头。

3. 子图

假设有两个图 G 和 G'，且满足条件：$V(G')\subset(G)$，$E(G')\subset(G)$，则称 G' 是 G 的子图。

4. 完全图与二分图

每一对不同的顶点都有一条边相连的简单图称为完全图(complete graph)。n 个顶点的完全图记为 K_n。

$G=(V,E)$为一个无向图,若能将 V 分成 V_1 和 V_2($V_1 \cup V_2=V, V_1 \cap V_2=\varphi$),使得 G 中的每条边的两个顶点都是一个属于 V_1,另一个属于 V_2,则称 G 为二分图(称 V_1 和 V_2 为互补顶点子集),常将二分图 G 记为(V_1,V_2,E)。又若 G 是简单二分图,V_1 中每个顶点均与 V_2 中所有的顶点相邻,则称 G 为完全二分图,记为 $K_{r,s}$,其 $r=|V_1|, s=|V_2|$。

5. 连接图和树

如果图 G 中两顶点 u 和 v 之间存在有通路,则称它们是连接的。如果图 G 中所有顶点间两两都有通路存在,则称它为连接图(connected graph)。图 G 的连接分量(connected component)是它最大的连接子图。如果删去图 G 中的某一个边 $e \in E$,会使图 G 至少增加 1 个连接分量,则称此边为割边(cut edge)。没有回路的连接图称为树(tree)。图中的最大连接子图称为团图(clique)。

3.1.2　图的基本数据结构

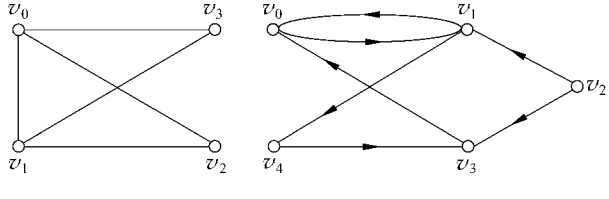
1. 邻接矩阵

邻接矩阵表示法是将图以邻接矩阵(adjacency matrix)的形式存储在计算机中。图 $G=(V,E)$的邻接矩阵是如下定义的:

$$A[i,j]=\begin{cases}1, & (v_i,v_j) \text{ 或 } <v_i,v_j> \text{ 是 } E(G) \text{ 中的边} \\ 0, & (v_i,v_j) \text{ 或 } <v_i,v_j> \text{ 不是 } E(G) \text{ 中的边}\end{cases}$$

也就是说,如果两顶点之间有一条弧,则邻接矩阵中对应的元素为 1;否则为 0。可以看出,这种表示法非常简单、直接。但是,在邻接矩阵的所有 n^2 个元素中,只有 m 个为非零元。如果网络比较稀疏,这种表示法浪费大量的存储空间,从而增加了在网络中查找弧的时间。

如图 3-1(a)是一个 4 个顶点的无向图,图 3-1(b)是一个 5 个顶点的有向图,可以分别用邻接矩阵 $\boldsymbol{A}_1$ 和 $\boldsymbol{A}_2$ 表示。行和列都是按照节点 v_0 v_1 v_2 v_3 的顺序排列的。



(a) 无向图　　(b) 有向图

图 3-1　无向图和有向图

$$\boldsymbol{A}_i=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix}\qquad \boldsymbol{A}_i=\begin{array}{c}\\v_0\\v_1\\v_2\\v_3\\v_4\end{array}\begin{array}{c}\begin{array}{ccccc}v_0&v_1&v_2&v_3&v_4\end{array}\\\begin{bmatrix}0&1&0&0&0\\1&0&0&0&1\\0&1&0&1&0\\1&0&0&0&0\\0&0&0&1&0\end{bmatrix}\end{array}$$

2. 关联矩阵

每行对应于图的一个顶点，每列对应于图的一条弧。如果一个顶点是一条弧的起点，则关联矩阵中对应的元素为 1；如果一个顶点是一条弧的终点，则关联矩阵中对应的元素为 −1；如果一个顶点与一条弧不关联，则关联矩阵中对应的元素为 0。对于简单图，关联矩阵每列只含有两个非零元(一个+1，一个−1)。可以看出，这种表示法也非常简单、直接。但是，在关联矩阵的所有 $n\times m$ 个元素中，只有 $2m$ 个为非零元。如果网络比较稀疏，这种表示法也会浪费大量的存储空间。

3. 边-节点表(数组)

由上可知，对于一般的图，邻接矩阵和关联矩阵都比较稀疏，因此会浪费存储空间。表结构比矩阵节省存储空间。可以采用一个二维数组来表示图的边-节点关系。如图 3-1(b)的有向图可以表示成表 3-1 所示的一个二维数组。假设图的边数为 m，节点数为 n，则边-节点表所需要的存储空间为 $2m$。

表 3-1 与图 3-1(b)对应的边-节点表

边号	1	2	3	4	5	6	7
始节点	0	1	3	1	4	2	2
终节点	1	0	0	4	3	1	3

4. 邻接表

邻接表表示法将图以邻接表(Adjacency Lists)的形式存储在计算机中。图的邻接表，也就是图的所有顶点的邻接表的集合；而对每个顶点，它的邻接表就是它的所有出弧。邻接表表示法就是对图的每个顶点，用一个单向链表列出从该顶点出发的所有弧，链表中每个单元对应于一条出弧。为了记录弧上的权，链表中每个单元除列出弧的另一个端点外，还可以包含弧上的权等作为数据域。图的整个邻接表可以用一个指针数组表示。图 3-2 所示为一种邻接表结构。

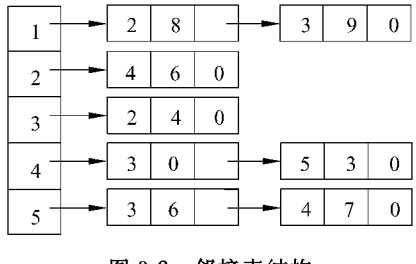


图 3-2 邻接表结构

这是一个 5 维指针数组，每一维(上面表示法中的每一行)对应于一个顶点的邻接表，如第 1 行对应于第 1 个顶点的邻接表(即第 1 个顶点的所有出弧)。每个指针单元的第 1 个数据域表示弧的另一个顶点(弧的头)，后面的数据域表示对应弧上的权。如第 1 行中的 2 表

示弧的另一个顶点为 2(即弧为(1,2)),8 表示对应弧(1,2)上的权为 8;3 表示弧的另一个顶点为 3(即弧为(1,3)),9 表示对应弧(1,3)上的权为 9。又如,第 5 行说明顶点 5 出发的弧有(5,3)、(5,4),它们对应的权分别为 6 和 7。

对于有向图,一般用表示顶点的邻接表,即顶点的所有出弧构成的集合或链表(实际上只需要列出弧的另一个顶点,即弧的头)。

5. 十字链表

十字链表(Orthogonal List)是有向图的另一种链式存储结构。可以把它看成是将有向图的邻接表和逆邻接表结合起来形成的一种链表。有向图中的每一条弧对应十字链表中的一个弧节点,同时有向图中的每个顶点在十字链表中对应有一个节点,叫做顶点节点。

在十字链表中既能够很容易地找到以 v_i 为尾的弧,也能够容易地找到以 v_i 为头的弧,因此对于有向图,若采用十字链表作为存储结构,则很容易求出顶点 v_i 的度。此外,为有向图建立一个邻接表和建立一个十字链表的算法的时间复杂度是相同的。所以,在某些有向图的应用中,十字链表表示法得到了广泛的应用。

3.2　算法及其复杂度

用于布图设计中的算法,大多数是基于通用算法和数学技巧之上的。在设计算法时,比较关注该算法的时间和空间,即算法的复杂度。这主要是因为所要解决的问题规模太大所致,现在的 VLSI 芯片上有数以亿计的图形,这对于二次方量级的算法时间都是无法实现的。本节将介绍算法及其复杂性的基本概念,然后叙述几种解决 NP-困难问题的策略。

3.2.1　算法问题及算法复杂性

定义 3.1　算法问题:一个算法问题 Π 是一个映射 Π:$I \to 2^S$,其中 I 是问题实例的集合,S 是构形的集合。符号 2^S 表示集合 S 的所有子集的总体。对于一个问题实例 $P \in I$,称 $\Pi(P)$ 的构成为 P 的解,或 P 的合法构形。这里 I 和 S 都可以是无穷集合。当 S 是单一的集合时,则 Π 叫做判定问题。

例如,求无向图 G 上的生成树问题。此时,$G \in I$ 是一个问题实例,S 是所有 G 上的生成树的集合。映射 Π 把这些生成树赋予图 G,每个 G 上的生成树就是对应算法问题 Π 的 G 上的一个解。

定义 3.2　算法复杂性:它是算法运行所需要的计算机资源的量,需要的时间资源的量称作时间复杂性,需要的空间(即存储器)资源的量称作空间复杂性。这个量应该集中反映算法中所采用的方法的效率,而从运行该算法的实际计算机中抽象出来。换句话说,这个量应该是只依赖于算法要解问题的规模、算法的输入和算法本身的函数,如果分别用 N,I 和 A 来表示算法要解问题的规模、算法的输入和算法本身,用 C 表示算法的复杂性,则 $C = F(N,I,A)$,其中 $F(N,I,A)$ 是 N,I,A 的一个确定的三元函数。如果把时间复杂性和空间复杂性分开,并分别用 T 和 S 来表示,则 $T = T(N,I,A)$ 和 $S = S(N,I,A)$。

定义 3.3　最优化问题:就是在满足一定的约束条件下,寻找一组参数值,使得系统的某些性能指标达到最大或最小。

定义 3.4 可行解问题：一个可行的算法 A 是指在 $T(N,I,A)=O(N^a)$ 且 $a\geqslant 0$ 的算法，若存在一个可行算法 A 可求解算法问题 Ⅱ，则称该算法问题 Ⅱ 是可行的，或称在多项式时间可求解。这类问题记作 P。

定义 3.5 NP-困难问题：一类不能在多项式时间内求解的问题。

3.2.2 求解 NP-困难问题的方法

由于计算复杂性，只有在问题规模不大时才能精确求解 NP-困难问题。当规模较大时，通常采用一些近似方法求解。鉴于 VLSI 布图中经常会遇到 NP-困难问题，下面介绍一些常用的求解策略以帮助了解求解 NP-困难问题的基本方法。

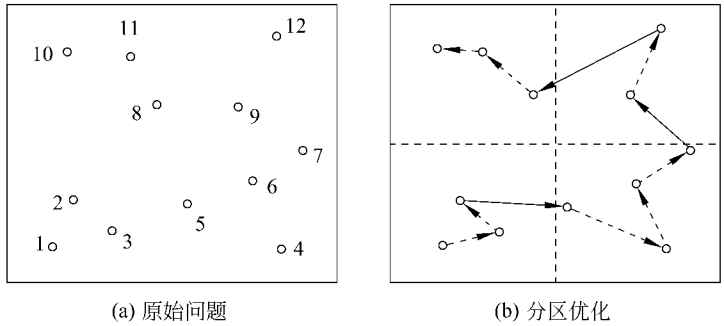
1. 限制问题的范围，只对某一类问题求解

例如在求图上的最小树时只求最小生成树(Minimum Spanning Tree)，即限制树的交叉点只能是原有的顶点。求最小生成树是一个多项式时间内可求解的，但它不一定能获得最小树。在下一节将会更详细地讨论它。

2. 限制问题的规模

例如现在要求解旅行商问题。一个商人要在 n 个城市中推销他的商品。他从其中的一个城市出发，到每个城市去一次，求一个最短路程的旅行路线。一个求解方法是枚举所有可能的旅行方案可以得到精确解。由于该问题的复杂性为 $(n-1)!/2$。当 $n=20$ 时，此值为 6×10^{16}。假如计算机计算一次总路径长度的运行时间为 10^{-8}s，则计算所有旅行方案的总时间为 $(6\times 10^{16}\times 10^{-8})/(365\times 24\times 3600)\approx 20$ 年。这显然行不通。但当 $n=10$ 时，$(n-1)!/2\approx 181440$。总时间约为 0.0018s。因此，一种求解旅行商问题的近似方法是分区优化。用图 3-3 给出分区优化的示意图。

在图 3-3(a)中，用圆圈表示城市，并加以编号。总共有 12 个城市。在图 3-3(b)中，把整个区域划分成 4 个分区。在每个分区枚举所有旅行方案。由于分区降低了每个分区内的节点的数目，就有可能求得精确解。然后把每个分区看成一个节点，再求解该问题。图 3-3(b)所示的旅行路线就是问题的一个近似解。



(a) 原始问题　　(b) 分区优化

图 3-3 旅行商问题的分区优化

3. 分支定界法

分支定界法(Branch and Bound)[1]是解决 NP-困难问题的一个通用的方法。它把原始问题划分成一些子问题求解,这就是“分枝”。为了防止枚举所有可能的解,还需要设置下(或上)界,在求解过程中以剪掉小于下界(或大于上界)但还未求解的枝,这就是“定界”。下面仍以旅行商问题为例来说明分支定界法的基本思想。

第一步,可以选择 12 个城市中任何一个作为起点,例如城市 1。然后只能选择剩下的 11 个城市中的任何一个作为第二个访问地,例如是城市 2。再在其余的 10 个城市中选择一个作为第三个访问地。依此类推,可以得到一个旅行方案,并计算其路径长度。此值可作为下界。路径长度的计算是每加入一个城市就累计长度,不是到最后才计算。

第二步,保持前 10 个城市顺序不动,更改第 11 和第 12 个城市的顺序,得到另一个方案及该方案的路径长度并与下界值比较,取小者作为新的下界。保持前 9 个城市顺序不动,更改后 3 个访问地顺序,同样计算路径长度并确定新的下界值。依此类推直到只剩下第一个不动其余 11 个都被改变过顺序。注意:在上述累计路径长度过程中,一旦发现其值超过下界就放弃此方案,并且不再继续此城市以前的顺序不动只改变以后访问顺序的所有方案。即剪掉这些方案。

第三步,取另一个城市作为第一个访问地,然后执行类似于上述第一步和第二步的操作。直到所有的城市都当过第一个访问地。

图 3-4 给出了整个过程的搜索树。这是一个按深度优先原则搜索的过程。分支定界法的关键是下界值的设定。一个好的下界可以剪掉很多枝,减少计算工作量。最坏情况下仍然要枚举所有方案。一个实用的方法是先用一个“启发式”算法求得一个近似解,然后用这个近似解作为分支定界法的下界。

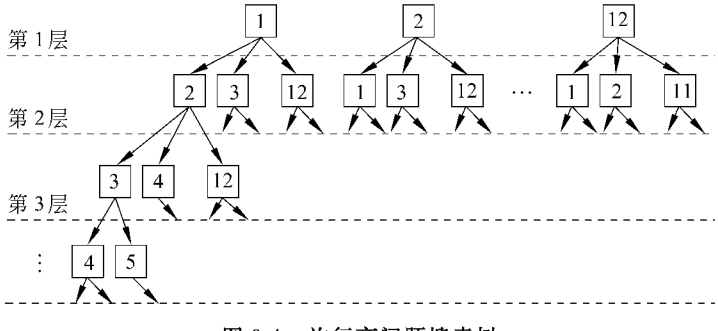


图 3-4　旅行商问题搜索树

4. 启发式算法

启发式(Heuristic)算法是求解 NP-困难问题最常用的方法。它利用问题本身的特点,借鉴物理学、化学、数学,甚至日常生活中的一些常识构造一个算法。它往往抓住问题的本质而忽略一些次要因素以降低计算复杂性。下面仍以旅行商问题为例来设计两个启发式算法,借以说明构造启发式算法的基本方法。

最近邻法:这是一种贪婪算法。它的主要思想是把离当前点最近的一个点作为下一个

访问地。从当前这一步来说，这是最佳的选择。但从总体观点来说，这不一定能得到最好的结果。该方法简单且计算复杂性低。算法可以这样描述：假如当前为第 k 步，已求得前 k 步的路径总长度为 T_k，寻找离当前点最近的，但还未访问过的点作为第 $k+1$ 点。$T_{k+1}=T_k+d(k,k+1)$，其中 $d(k,k+1)$ 为第 k 点到第 $k+1$ 点的距离。若设精确解为 T_n^*，此方法的近似解为 T_n，可以证明 $T_n<\frac{1}{2}(\log_2 n+1)T_n^*$。

3.3 解决布图问题的基本算法

3.3.1 图论算法

1. 图搜索算法

1）深度优先搜索

深度优先搜索(Depth-First Search)遍历类似于树的先序遍历，是树的先序遍历的推广。其基本思想如下：假定以图中某个顶点 v_i 为出发点，首先访问出发点，然后选择一个 v_i 未访问过的邻接点 v_j，以 v_j 为新的出发点继续进行深度优先搜索，直至图中所有顶点都被访问过。显然，这是一个递归的搜索过程。

分析上述算法。假设图有 $|V|$ 个顶点，那么，当用邻接矩阵表示图时，搜索一个顶点的所有邻接点需花费的时间为 $O(|V|)$，则从 $|V|$ 个顶点出发搜索的时间应为 $O(|V|^2)$；如果使用邻接表来表示图时，需花费时间为 $O(|V|+|E|)$，其中 $|E|$ 为无向图中边的数目或有向图中弧的数目。

2）广度优先搜索

广度优先搜索(Breadth-First Search)遍历图类似于树的按层次遍历。其基本思想是：首先访问图中某指定的起始点 V_i 并将其标记为已访问过，然后由 V_i 出发访问与它相邻接的所有顶点 V_j、V_k……并均标记为已访问过，然后再按照 V_j、V_k……的次序，访问每一个顶点的所有未被访问过的邻接顶点，并均标记为已访问过，下一步再从这些顶点出发访问与它们相邻接的尚未被访问的顶点，如此做下去，直到所有的顶点均被访问过为止。

分析上述算法，假设图有 $|V|$ 个顶点、$|E|$ 条边，在广度优先搜索图的过程中，每个顶点至多进一次队列，所以算法中的内外循环次数均为 $|V|$ 次，故算法 BFS 的时间复杂度为 $O(|V|^2)$；若采用邻接链表存储结构，广度优先搜索遍历图的时间复杂度与深度优先搜索是相同的，也是 $O(|V|+|E|)$。

2. 最短路径

VLSI 布图设计中的布线问题实际上就是一个特殊图上的最短路径(shortest path)问题，因此，最短路径算法在布线中占有重要地位。在此介绍几种经典的最短路径算法。

最短路径问题：如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小。

1）Dijkstra 算法

Dijkstra 算法[2]解决的是边上权值非负情形的单源最短路径问题。

在图 G 中，给定 s 和 t 两个顶点。从 s 到 t 可以有多条路径，从这多条路中找出长度最小的路，这样的路称为从 s 到 t 的最短路。设每条弧的长度均为非负值。

Dijkstra 算法的基本思想是：设已知图中最接近于顶点 s 的 m 个顶点以及从顶点 s 到这些顶点中每一个顶点的最短路(从 s 到其本身的最短路是零路，即没有弧的路，其长度为 0)。对顶点 s 和这 m 个顶点着色。然后，最接近于 s 的第 $m+1$ 个顶点可如下求之：

对于每一个未着色的顶点 y，考虑所有已着色顶点 x，把弧(x,y)接在从 s 到 x 的最短路后面，这样就得到从 s 到 y 的 m 条不同路。从这 m 条路中选出最短的路，它就是从 s 到 y 的最短路。相应的 y 点就是最接近于 s 的第 $m+1$ 个顶点。因为所有弧的长度都是非负值，所以从 s 到最接近于 s 的第 $m+1$ 个顶点的最短路必然只使用已着色的顶点作为中间顶点。其伪代码如下：

```
void Dijkstra(MGraph G,int v0,PathMatrix &P,ShortPathTable &D)
{for (v=0;v<G,vexnum;++v)
  {final[v]=FALSE; D[v]=G.arcs[v0][v];
for (w=0;w<G,vexnum;++w) P[v][w]=FALSE;
if (D[v]<INFINITY){p[v][v0]=TRUE;P[v][v]=TRUE;}
  }
D[v0]=0; final[v0]=TRUE;
for (i=0;i<G,vexnum;++i)
  {min=INFINITY;
for (w=0;w<G,vexnum;++w)
  if(!final[w])
    if(D[w]<min) {v=w; min=D[w];}
final[v]=TRUE;
for (w=0;w<G,vexnum;++w)
  if(!final[w] &&(min+G.arcs[v][w]<D[w]))
    {D[w]=min+G.arcs[v][w];
     P[w]=P[v]; P[w][w]=TRUE;
    }
    }
  }
```

对于有$|N|$个顶点的图，求一个顶点到其余顶点的最短路径，循环$|N|-1$次，加上修改最短路径的循环，是两层循环，故时间复杂度为 $O(|N|^2)$。若只求从源点到某一顶点间的最短路径，和求到其余顶点间的最短路径相同，时间复杂度也是 $O(|N|^2)$。

2) floyd 算法

floyd 算法[3]解决的是所有顶点之间的最短路径问题。它的基本思想是：设求顶点 v_i 到 v_j 间的最短路径，若 v_i 到 v_j 有弧，则弧上的权值是一条路径，但未必是最短路径，要经过 $n-1$ 次测试。首先将顶点 v_1 加入，即看(v_i, v_1)和(v_1, v_j)是否有路径，且比(v_i, v_j)低，如是，则用后两段路径代替，并称这是 v_i 到 v_j 中间顶点序号不大于 1 的最短路径。再将顶点 v_2 加入，得到 v_i 到 v_j 中间顶点序号不大于 2 的最短路径。如此下去，直到 v_n加入，得到 v_i 到 v_j 中间顶点序号不大于 n 的最短路径，算法结束。其伪代码如下：

```
void Floyd(MGraph G, PathMatrix &P[],DistanceMatrix &D)
{for (v=0;v<G,vexnum;++v)
for (w=0;w<G,vexnum;++w)
    {D[v][w]=G.arcs[v][w];
for (u=0;u<G,vexnum;++u) P[v][w][u]=FALSE;
if(D[v][w]<INFINITY){p[v][w][v]=TRUE;P[v][w][w]=TRUE;}
}
  for (u=0;u<G.vexnum;u++)
    for (v=0;v<G.vexnum;v++)
for (w=0;w<G.vexnum;w++)
  if (D[v][u]+D[u][w]<D[v][w])
    { D[v][w]=D[v][u]+D[u][w];
for (i=0;i<G,vexnum;++i)
    P[v][w][i]=P[v][u][i]||P[u][w][i];
    }
  }
```

Floyd 算法的时间复杂度是 $O(n^3)$。

3) Bellman-Ford 算法

Bellman-Ford 算法[4]解决的是边上权值为任意值的单源最短路径问题。

用 Dijkstra 算法求有向图的单源最短路，条件是图中任意一条边都是正权。但在求单源最短路中可能存在有负权边的有向图，如果不包括从源点可达的负权回路，Dijkstra 算法依然可以求得最短路的长度；但如果存在从源点可达的负权回路，Dijkstra 算法就无法计算了，因为总可以顺着“最短路”再穿过负权回路从而得到更小的最短路径值，一直这样下去总可以找到更小的最短路。针对这种情况 Bellman-Ford 算法对带负权的有向图返回一个布尔值，当图中存在负权回路时，返回 false；若不存在负权回路，则返回 true，并产生单源最短路径。

它的基本思想是：对每一点 v，设置变量 $d[v]$，描述从 s 到 v 的最短路径的权的上限。设置变量 $p[v]$，保存当前最短路径中 v 的前驱顶点。初始化 $d[v]=\infty$；$d[s]=0$；$p[v]=\infty$；对每一条边，做 $|v|-1$ 次松弛操作。判断是否有负权回路。松弛操作：对于边 (u,v) $d[v]>d[u]+w[u,v]$，作如下修改：$d[v]=d[u]+w[u,v]$，$p[v]=u$；否则，$d[v]$ 保持不变。判断是否有负权回路再做一次松弛操作，也就是说如果存在一条边 (u,v)，$d[v]>d[u]+w[u,v]$，那么图中一定存在负权回路，返回 false；否则，说明图中没有负权回路，返回 true，并且根据 $p[v]$ 得到 s 到每一点的最短路径。Bellman-Ford 算法时间复杂度为 $O(|N|^2)$。

```
void Graph::BellmanFord ( int n,  int v ) {
    for ( int i=0; i<n; i++) {
        dist[i]=Edge[v][i];
        if ( i !=v && dist[i]<MAXINT )
       path[i]=v;
        else path[i]=-1;
    }
for ( int k=2; k<n; k++)
        for ( int u=0; u<n; u++)
```

```
        if ( u !=v )
           for ( i=0; i<n; i++)
                if ( Edge[i][u]>0 &&
                           Edge[i][u]<MAXNUM &&
                           dist[u]>dist[i]+Edge[i][u] ) {
                     dist[u]=dist[i]+Edge[i][u];
                     path[u]=i;
                  }
}
```

4) A* 算法

A* 算法[5]是解决特定两个顶点之间的最短路径问题。

A* 算法实际是一种启发式搜索。启发式搜索，就是利用一个评价函数评估每次的决策的价值，决定先尝试哪一种方案。这样可以极大地优化普通的广度优先搜索。一般来说，从出发点 s 到目的地 t 的最短距离是固定的，可以写一个函数 judge()估计 s 到 t 的最短距离，如果已经尝试着从出发点 s 沿着某条路线移动到了 m 点，那么认为这个方案的 s 到 t 间的估计距离为 s 到 m 实际已经行走了的距离 H，加上用 judge() 估计出的 m 到 t 的距离。如此，无论搜索展开到哪一步，都会算出一个评估值，每一次决策后，将评估值和等待处理的方案一起排序，然后挑出待处理的各个方案中最有可能是最短路线的一部分的方案展开到下一步，一直循环到对象移动到目的地，或所有方案都尝试过却没有找到一条通向目的地的路径时则结束。

3. 最小生成树

在设计电子线路时，常常要把数个元件的引脚连在一起，使其电位相同。要使几个引脚互相连通，可以使用 $n-1$ 条接线，每条接线连接两个引脚。连接方法很多，通常需要找出连线总长度最短的接法。

把这一连线问题模型化为一个具有边权的图 $G=(V,E)$，其中 V 是引脚集合，E 是每对引脚间可能存在的连线集合。对图中每一边 $(u,v)\in E$，有一个权值 $w(u,v)$ 来表示连接 u 和 v 的代价(需要的接线长度)。希望找出一无回路子集 $T\subseteq E$，使其连接所有节点，且其权值之和为最小。因为 T 无回路且连接了所有节点，所以它必然是一棵树，称为生成树。把确定树 T 的问题称为最小生成树问题。

下面将阐述解决最小生成树问题的两种算法：Kruskal 算法和 Prim 算法。这两个算法中都使用了堆，运行时间均为 $O(E\log V)$。通过采用 Fibonacci 堆，Prim 算法的运行时间可以减少到 $O(E+V\log V)$，若 $|V|$ 远小于 $|E|$，这将是对算法的较大改进。

1) Prim 算法

在 Prim 算法[6]中设置了一个优先队列 q，它存放待处理的节点。每次 q 中节点都按其相应的 d 值(边长)进行排序，使得 d 值小的节点排在队首。此外，T 数组存放求得的 Spanning 树的边，如 $T[v]=u$，则表示边 (u, v) 是树的一个边。$d[v]$ 存放连接节点 v 的某一边的边长，若有多个边与 v 连接，则 $d[v]$ 会逐渐更新为其中最小的边长值。not_in_tree 存放节点是否在树中的标志。$A[v]$ 是与节点 v 相邻的节点集。s 是树的第一个节点。$n=|V|$ 是节点数。下面是算法的伪代码表示：