第八讲 图算法

概述

- 定义和表示
- 最小生成树：Prim算法
- 单源最短路径：Dijkstra算法
- 全部顶点对间的最短路径
- 传递闭包
- 连通分量
- 稀疏图算法

1 定义和表示

定义和表示

- 无向图G是一对(V,E)，其中V是有限个称为顶点的点集，E是有限条边的边集。
- 边$e \in E$是无顺序的对(u,v)，这里$u,v \in V$。
- 有向图G也是一对(V,E)，和上面定义的顶点集合相同，但边(u,v)是有序对，即表示有一个从u到v的连接。
- 从顶点v到顶点u的路径是顶点序列$<v_0,v_1,v_2,\ldots,v_k>$，其中$v_0 = v$，$v_k = u$，且对$i = 0, 1,\ldots, k-1$，$(v_i, v_{i+1}) \in E$。
- 路径的长度被定义为路径中边的数目。

定义和表示

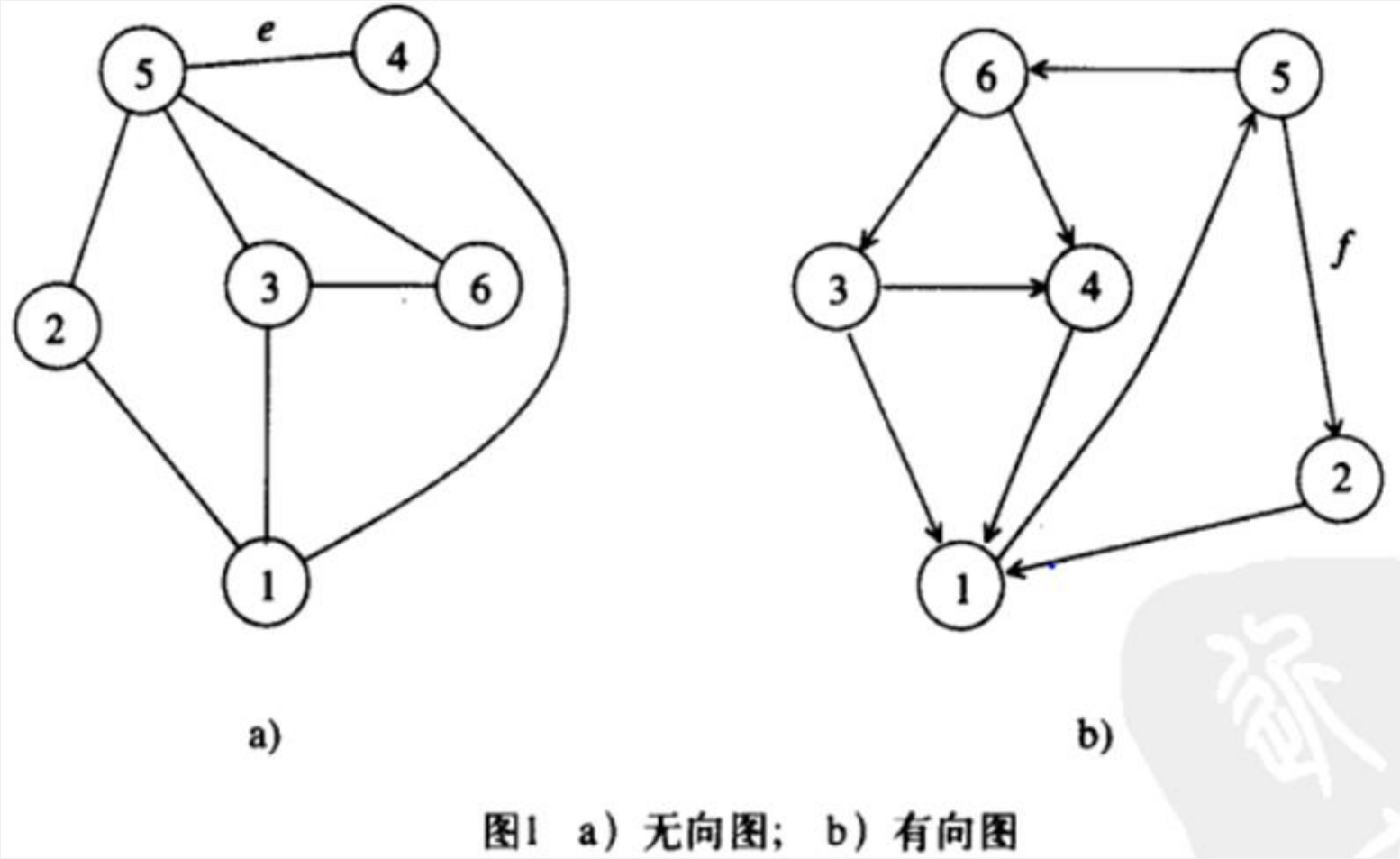


图1 a）无向图； b）有向图

定义和表示

- 图可以用邻接矩阵或者一个边（或点）表来表示。
- 如果点*i*和*j*在同一条边，则邻接矩阵有值$a_{i,j} = 1$；否则该值为0。在有权图中，$a_{i,j} = w_{i,j}$，$w_{i,j}$表示为边的权值。

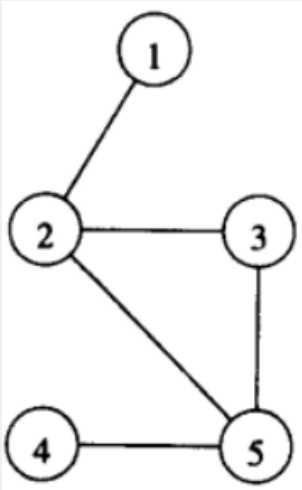
定义和表示

- 如果无向图中的每一对顶点通过一条路径连接，则称无向图是连通的。
- 森林是无环图，而树是连通无环图。
- 如果图中的每条边都有权与之对应，则该图称为加权图。

定义和表示

- 图可以用它们的邻接矩阵或边列表（或者点列表）来表示。
- 如果节点 i 和 j 在同一条边，邻接矩阵就有值 $a_{i,j} = 1$；否则为0。如果是一个加权图， $a_{i,j} = w_{i,j}$ 是该边的权。
- 一个图 $G = (V,E)$ 的邻接表由表的数组 $Adj[1../V/]$ 构成。每一个表 $Adj[v]$ 是所有与 v 邻接的顶点的列表。
- 对于一个包含 n 个节点的图，邻接矩阵占用 $\Theta(n^2)$ 的空间，而邻接表占用 $\Theta(/E/)$ 的空间。

定义和表示



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

图2　无向图及其邻接矩阵表示

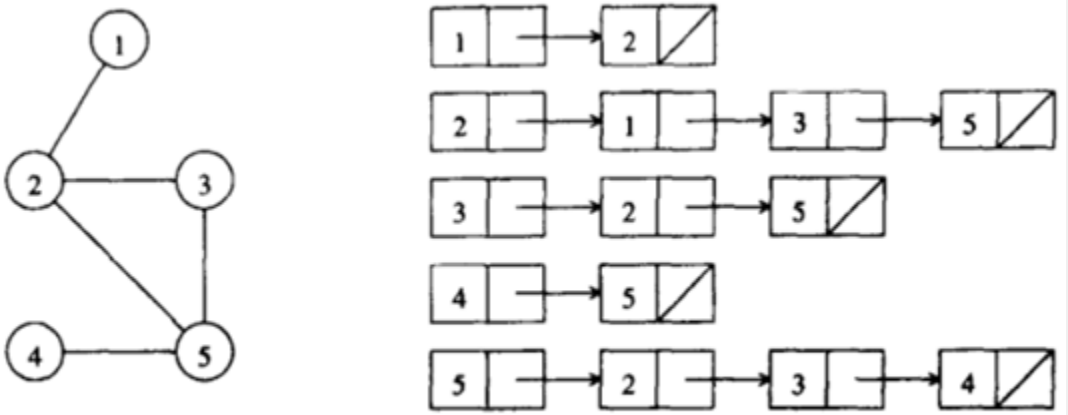


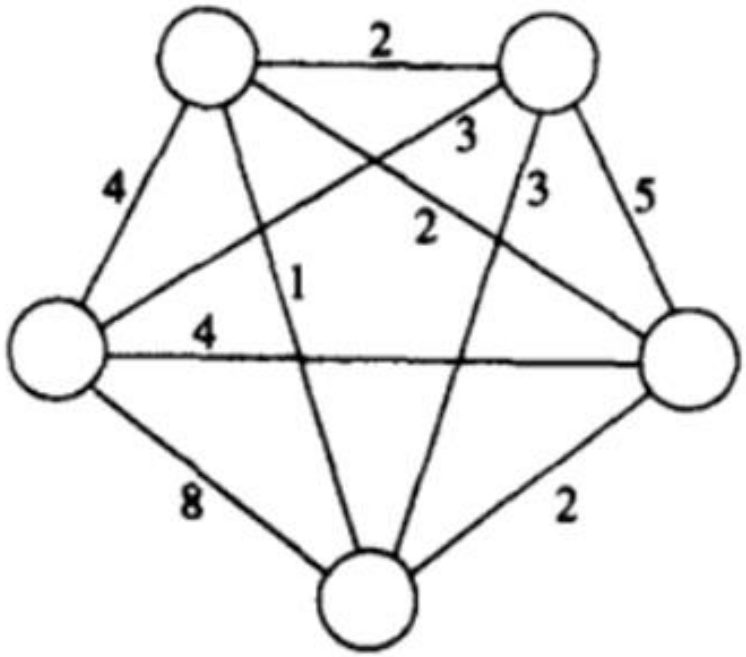
图3　无向图及其邻接表表示

2 最小生成树：Prim算法

最小生成树

- 无向图G的生成树是图G的子图，它是一棵包含图G的所有顶点的树。
- 在加权图中，子图的权是子图中的边的权值之和。
- 加权无向图的最小生成树（MST）是权值最小的生成树。

最小生成树



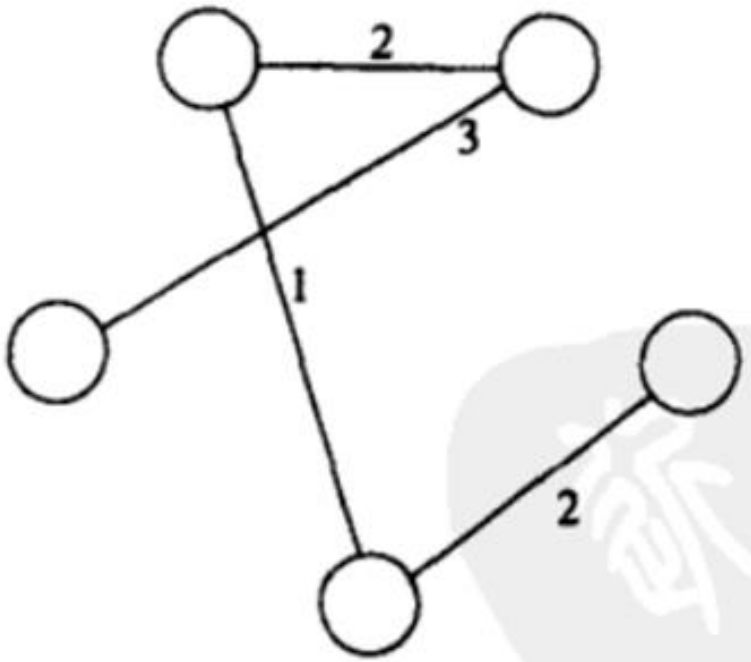


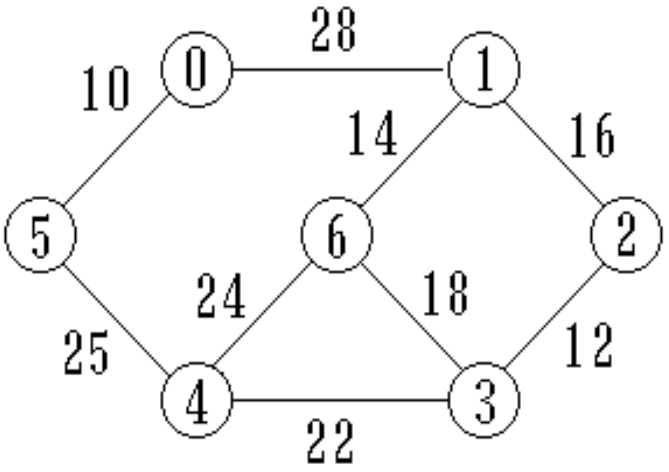
图4　无向图及其最小生成树

最小生成树：Prim算法

- 寻找最小生成树的Prim算法是一个贪婪算法。
- 算法首先选择任意的起点，将它作为现在最小生成树的点。
- 然后不断加入能保证生成树成本最小的顶点和边，扩大最小生成树。

用Prim算法构造最小生成树的例子

构造下图(a)的最小生成树

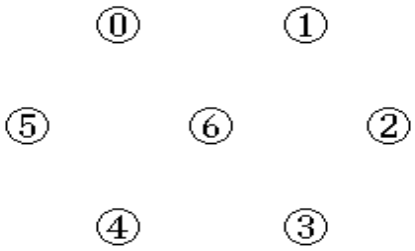


（a）连通网络

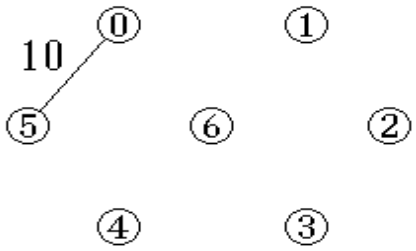
	0	1	2	3	4	5	6	
	0	28	∞	∞	∞	10	∞	0
	28	0	16	∞	∞	∞	14	1
	∞	16	0	12	∞	∞	∞	2
	∞	∞	12	0	22	∞	18	3
	∞	∞	∞	22	0	25	24	4
	10	∞	∞	∞	25	0	∞	5
	∞	14	∞	18	24	∞	0	6

（b）邻接矩阵表示

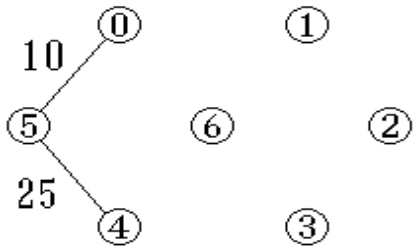
用Prim算法构造最小生成树的过程



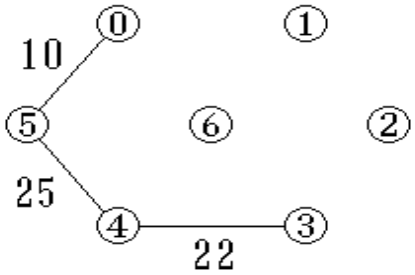
(a)



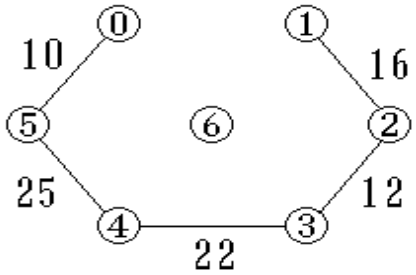
(b)



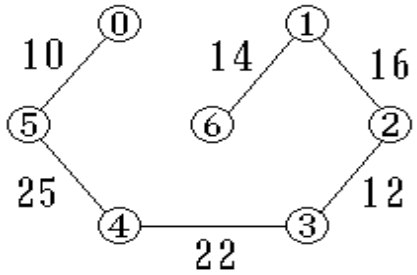
(c)



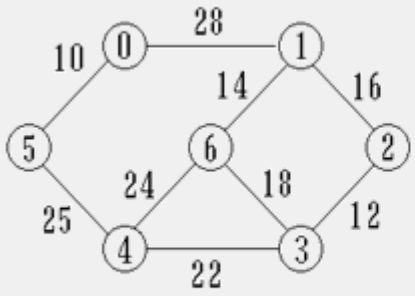
(d)



(e)



(f)

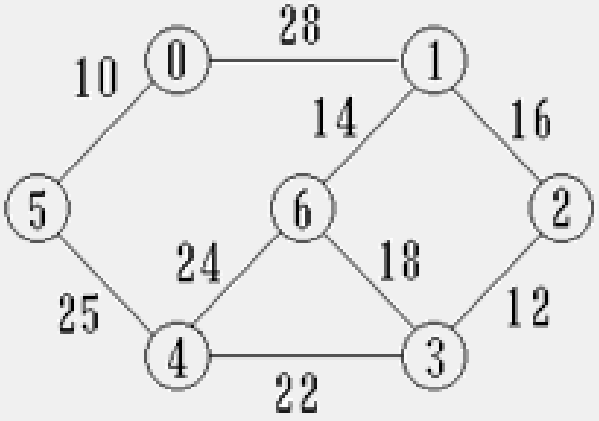


(a) 连通网络

	0	1	2	3	4	5	6	
0	0	28	∞	∞	∞	10	∞	0
1	28	0	16	∞	∞	∞	14	1
2	∞	16	0	12	∞	∞	∞	2
3	∞	∞	12	0	22	∞	18	3
4	∞	∞	∞	22	0	25	24	4
5	10	∞	∞	∞	25	0	∞	5
6	∞	14	∞	18	24	∞	0	6

(b) 邻接矩阵表示

- 在构造过程中，设置了两个辅助数组：
  - *d* [ ] 存放生成树顶点集合内顶点到生成树外各顶点的各边上的当前最小权值*lowcost*:距离；
  - *nearvex*[ ] 记录生成树顶点集合外各顶点距离集合内哪个顶点最近(即权值最小)。
- 例子

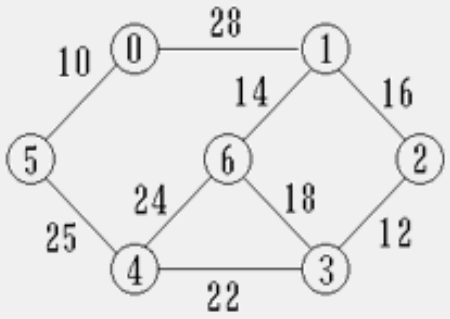


（a）连通网络

	0	1	2	3	4	5	6	
	0	28	∞	∞	∞	10	∞	0
	28	0	16	∞	∞	∞	14	1
	∞	16	0	12	∞	∞	∞	2
	∞	∞	12	0	22	∞	18	3
	∞	∞	∞	22	0	25	24	4
	10	∞	∞	∞	25	0	∞	5
	∞	14	∞	18	24	∞	0	6

（b）邻接矩阵表示

Prim算法的计算过程



（a）连通网络

	0	1	2	3	4	5	6	
	0	28	∞	∞	∞	10	∞	0
	28	0	16	∞	∞	∞	14	1
	∞	16	0	12	∞	∞	∞	2
	∞	∞	12	0	22	∞	18	3
	∞	∞	∞	22	0	25	24	4
	10	∞	∞	∞	25	0	∞	5
	∞	14	∞	18	24	∞	0	6

（b）邻接矩阵表示

顶点0与其他顶点连接边：

	0	1	2	3	4	5	6	
d [ ]								
lowcost	0	28	∞	∞	∞	10	∞	
nearvex	−1	0	0	0	0	0	0	

v = 5 选边（0，5）

顶点5加入生成树顶点集合： *v* = 5

	0	1	2	3	4	5	6	
d [ ]								
lowcost	0	28	∞	∞	25	10	∞	
nearvex	−1	0	0	0	5	−1	0	

v = 4 选边（5，4）

继续重复得： *v* = 4

	0	1	2	3	4	5	6	
d [ ]								
lowcost	0	28	∞	22	25	10	24	
nearvex	−1	0	0	4	−1	−1	4	

v = 3 选边（4，3）

lowcost	0	28	12	22	25	10	18	
nearvex	−1	0	3	−1	−1	−1	3	

v = 2 选边（3，2）

lowcost	0	16	12	22	25	10	18	
nearvex	−1	2	−1	−1	−1	−1	3	

v = 1 选边（2，1）

lowcost	0	16	12	22	25	10	14	
nearvex	−1	−1	−1	−1	−1	−1	1	

v = 6 选边（1，6）

lowcost	0	16	12	22	25	10	14	
nearvex	−1	−1	−1	−1	−1	−1	−1	

最小生成树：Prim算法

```
1.    procedure PRIM_MST(V, E, w, r)
2.    begin
3.        V_T := {r};
4.        d[r] := 0;
5.        for all v ∈ (V − V_T) do
6.            if edge (r, v) exists set d[v] := w(r, v);
7.            else set d[v] := ∞;
8.        while V_T ≠ V do
9.        begin
10.           find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.           V_T := V_T ∪ {u};
12.           for all v ∈ (V − V_T) do
13.               d[v] := min{d[v], w(u, v)};
14.       endwhile
15.   end PRIM_MST
```

Prim算法：并行形式

- 算法执行n步外部迭代——同时执行这些迭代很困难。
- 算法的内部循环相对容易并行化，设p为处理器个数，n为点的个数。
- 邻接矩阵用一维块映射分划，距离数组d也相应地进行分划。
- 在每一步，每个进程选择本地最近点，再通过全局归约选择全局最近点。
- 将全局最近点插入到最小生成树中，并且广播到所有进程中。
- 每个进程更新本地的数组d。

Prim算法：并行形式

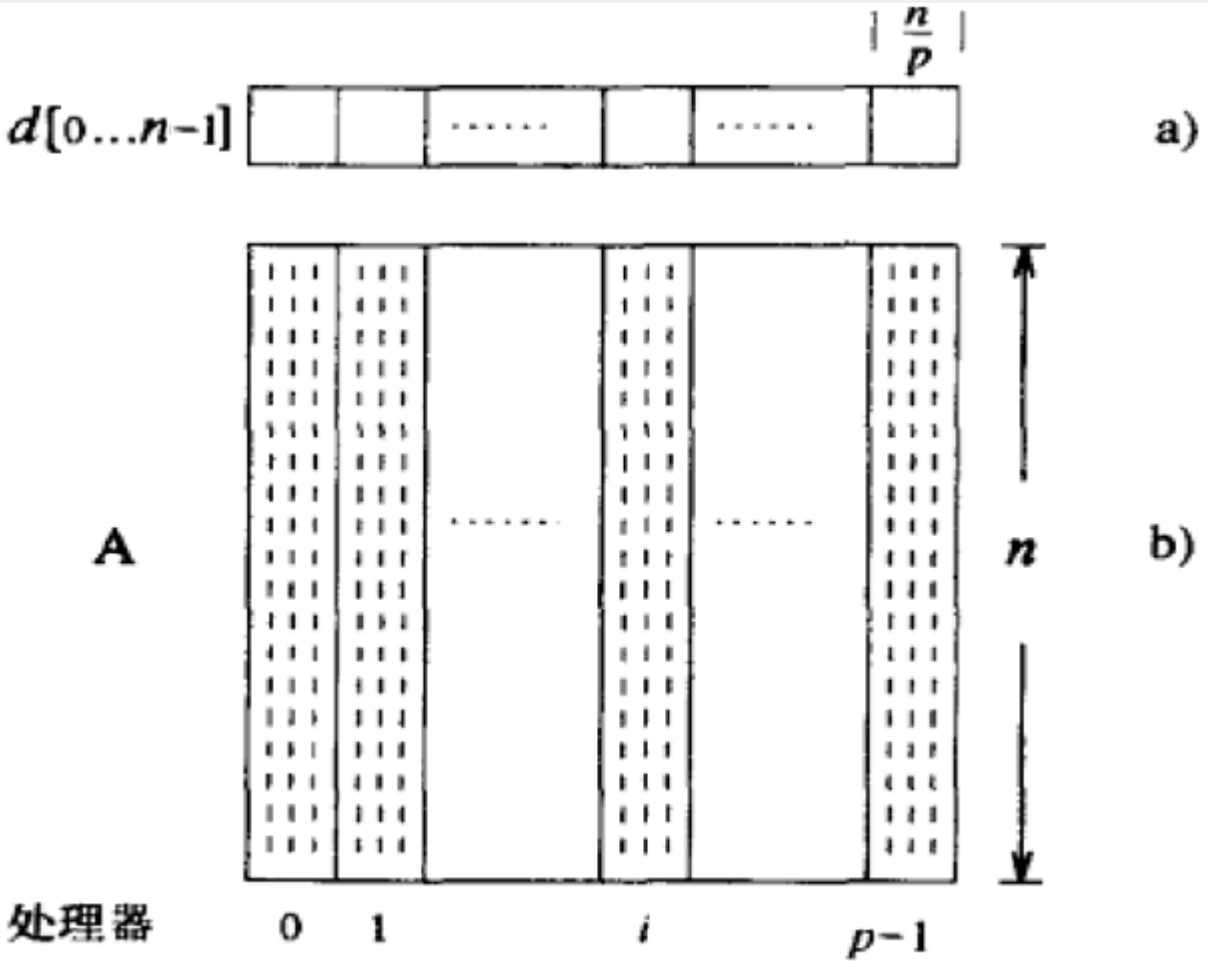


图6　在p个进程中划分距离数组d以及邻接矩阵A

Prim算法：并行形式

- 选择最小元素的成本是$O(n/p + \log p)$。
- 一个广播的成本是$O(\log p)$。
- 对本地数组d的更新成本是$O(n/p)$。
- 每次迭代的并行时间是$O(n/p + \log p)$。
- 总并行时间是$O(n^2/p + n \log p)$。
- 相应的等效率函数为$\Theta(p^2 \log^2 p)$。

3 单源最短路径：Dijkstra算法

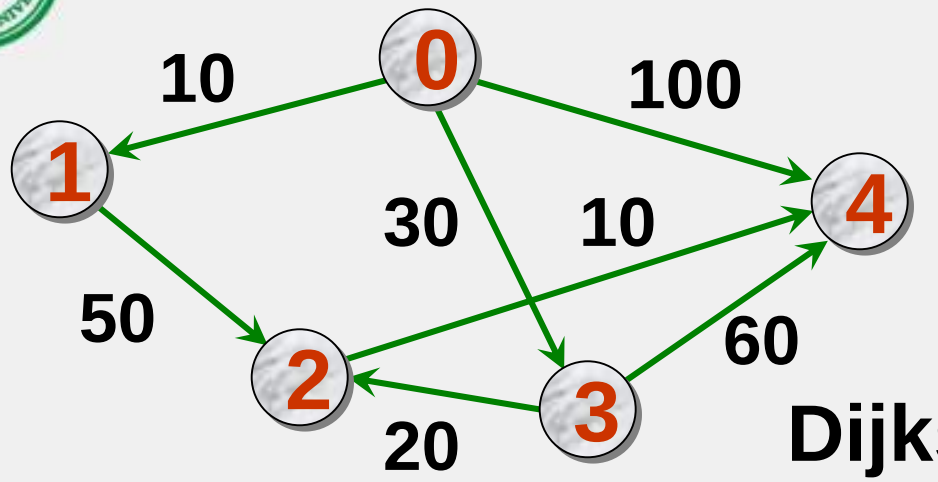
单源最短路径

- 对于一个加权图$G = (V,E,w)$，单源最短路径问题是找出从$v \in V$点到其他所有在V的点的最短路径。
- Dijkstra算法类于Prim算法。它维护一组已知最短路径的节点。
- 该算法通过使用最短路径集中的点来找源的最近点，从而扩展最短路径集合。

边上权值非负情形的单源最短路径问题

- 问题的提法：给定一个带权有向图*D*与源点 *v*，求从*v*到*D*中其他顶点的最短路径。

  限定各边上的权值大于或等于0。
- 为求得这些最短路径, Dijkstra提出按路径长度的递增次序，逐步产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点v到其它各顶点的最短路径全部求出为止。



Dijkstra逐步求解的过程

源点	终点	最短路径			路径长度
v_0	v_1	(v_0,v_1)			10
	v_2		(v_0,v_1,v_2)	(v_0,v_3,v_2)	∞,60,50
	v_3	(v_0,v_3)			30
	v_4	(v_0,v_4)	(v_0,v_3,v_4)	(v_0,v_3,v_2,v_4)	100,90,60

- 引入辅助数组l。它的每一个分量l[i]表示当前找到的从源点 v_0 到终点 v_i 的最短路径的长度。初始状态：
  - 若从v_0到顶点v_i有边，则l[i]为该边的权值；
  - 若从v_0到顶点v_i无边，则l[i]为∞ 。
- 假设S是已求得的最短路径的终点的集合，则可证明：下一条最短路径必然是从v_0 出发，中间只经过S中的顶点便可到达的那些顶点v_x ($v_x \in V-S$)的路径中的一条。
- 每次求得一条最短路径后，其终点v_k 加入集合S，然后对所有的$v_i \in V-S$，修改其 l[i]值。

Dijkstra算法可描述如下：

① 初始化： $S \leftarrow \{v_0\}$;

$l[j] \leftarrow A[0][j]$, $j = 1, 2, \ldots, n-1$;

// n为图中顶点个数

② 求出最短路径的长度：

$l[k] \leftarrow \min\{l[i]\}$, $i \in V-S$;

$S \leftarrow S \cup \{k\}$;

③ 修改：

$l[i] \leftarrow \min\{l[i], l[k]+A[k][i]\}$,

对于每一个 $i \in V-S$;

④ 判断：若 $S = V$, 则算法结束，否则转②。

单源最短路径：Dijkstra算法

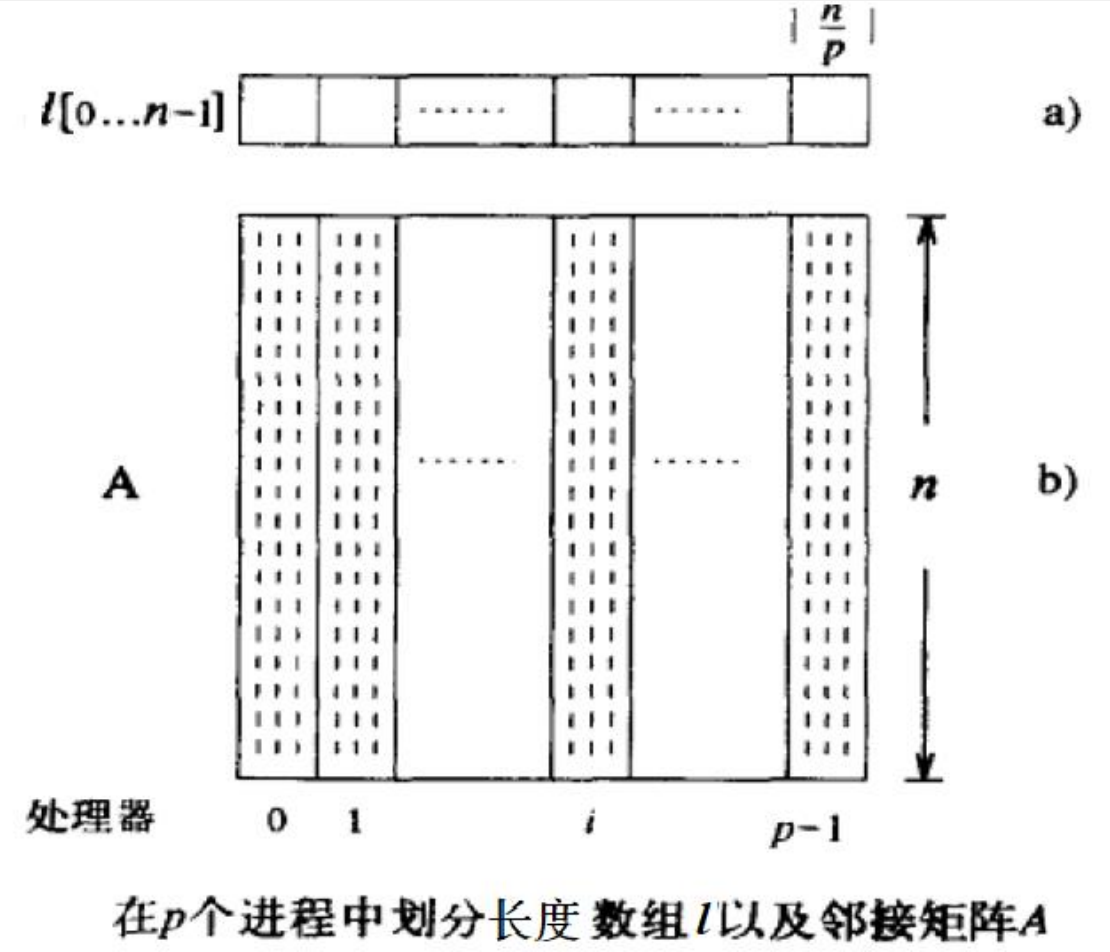
算法 2　Dijkstra的串行单源最短路径算法

```
1.  procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2.  begin
3.      V_T := {s};
4.      for all v ∈ (V − V_T) do
5.          if (s, v) exists set l[v] := w(s, v);
6.          else set l[v] := ∞;
7.      while V_T ≠ V do
8.      begin
9.          find a vertex u such that l[u] := min{l[v]|v ∈ (V − V_T)};
10.         V_T := V_T ∪ {u};
11.         for all v ∈ (V − V_T) do
12.             l[v] := min{l[v], l[u] + w(u, v)};
13.     endwhile
14. end DIJKSTRA_SINGLE_SOURCE_SP
```

Dijkstra算法：并行形式

- 非常类似于关于最小生成树问题的Prim算法并行形式。
- 加权邻接矩阵使用一维块映射分划。
- 每一个进程在本地选择离源点最接近的点，然后使用全局归约找出全局最近点。
- 这个点被广播到所有进程中，并且更新本地的l数组值。
- Dijkstra算法的并行性能与Prim算法相同。

Dijkstra算法：并行形式



在p个进程中划分长度数组l以及邻接矩阵A

4 全部顶点对间的最短路径

- 基于Dijkstra算法
- 基于矩阵“乘法”算法
- Floyd算法
- 性能比较

全部顶点对间的最短路径

- 对于加权图$G(V,E,w)$，全部顶点间的最短路径问题就是找出全部顶点对$v_i, v_j \in V$间的最短路径。
- 已知有一定数量的算法用于解决这个问题。

基于Dijkstra算法

- 单源最短路径的Dijkstra算法负责度为$O(n^2)$。
- 全部顶点对间的最短路径算法的复杂度为$O(n^3)$。

基于Dijkstra算法：并行形式

- 两种并行策略-一种方法是将顶点分划到不同的进程，让每个进程计算分配给它的所有顶点间的单源最短路径，称这种方法为源划分形式；另一种方法分配每个顶点给一组进程，并用单源算法的并行形式求解每组进程上的问题，称这种方法为源并行形式。

基于Dijkstra算法：源划分形式

- 使用n个进程，通过执行Dijkstra的串行单源最短路径算法，每个进程P_i找出从顶点v_i到所有其他顶点间的最短路径。
- 这种形式不需要进程间的通信（假设邻接矩阵在所有进程处被复制）。
- 并行运行时间为$\Theta(n^2)$。
- 由并发带来的等效率函数为$\Theta(p^3)$。

基于Dijkstra算法：源并行形式

- 在这个例子中，每个最短路径问题被进一步地进行并行执行。因此我们能够使用最多个n^2处理器。
- 给定p个处理器(p>n)，每个单源最短路径问题被p/n个处理器执行。
- 并行运行时间为：

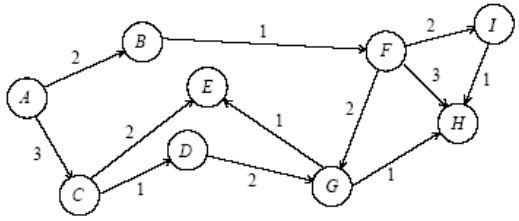
$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n \log p).}^{\text{communication}}$$

- 对于成本最优的并行形式，我们有$p = O(n^2/\log n)$，并且等效率函数为$\Theta((p \log p)^{1.5})$。

全部顶点对间的最短路径
基于矩阵乘法算法

- 考虑加权邻接矩阵与自己相乘——但是在这个例子中，我们将矩阵相乘中的元素相乘再求和的操作替换为元素相加再求最小值的操作。
- 可以注意到加权邻接矩阵与自己“相乘”可以得到一个包含所有顶点对间的长度为2的最短路径的矩阵。
- 这样计算下去A^n包含所有最短路径。

基于矩阵乘法算法



$$A^1 = \begin{pmatrix} 0 & 2 & 3 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & 2 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^2 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & \infty & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^8 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

基于矩阵乘法算法

- A^n是通过指数的翻倍计算出来的。例如A, A^2, A^4, A^8等等。
- 我们需要$\log n$个矩阵乘法，每个花费$O(n^3)$时间。
- 这个算法的串行复杂度为$O(n^3 \log n)$。
- 这个算法不是最优的，因为已知最优算法有复杂度$O(n^3)$。

基于矩阵乘法算法：并行形式

- 在*log n*个矩阵乘法中，每次乘法都能被并行执行。
- 我们可以使用$n^3/\log n$个处理器在*log n*时间内计算每个矩阵乘矩阵操作。
- 所有进程使用$O(\log^2 n)$时间。

所有顶点之间的最短路径

Floyd算法

- 问题的提法：已知一个各边权值均大于0的带权有向图，对每一对顶点 $v_i \neq v_j$，要求求出v_i与v_j之间的最短路径和最短路径长度。
- Floyd算法的基本思想：

定义一个n阶方阵序列：

$$D^{(0)},\ D^{(1)},\ \ldots,\ D^{(n)}.$$

其中 $D^{(0)}[i][j] = w[i][j]$;

$$D^{(k)}[i][j] = \min\{ D^{(k-1)}[i][j],\ D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \},\ k = 1,\ldots, n$$

Floyd算法

- 对于任一对顶点v_i, $v_j \in V$，考虑从v_i到v_j的所有路径，其中间顶点属于集合$\{v_1,v_2,\ldots,v_k\}$，令$p_{i,j}^{(k)}$（权值为$d_{i,j}^{(k)}$）为这些路径中权最小的路径。
- 如果顶点不是从v_i到v_j的最短路径，则$p_{i,j}^{(k)}$与$p_{i,j}^{(k-1)}$相同。
- 如果v_k在$p_{i,j}^{(k)}$中，则可将$p_{i,j}^{(k)}$分成两条路径——一条从v_i到v_k，另一条从v_k到v_j。这两条路径都使用$\{v_1,v_2,\ldots,v_{k-1}\}$中的顶点。

Floyd算法

- 以上结论由下面递归公式表示：

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\left\{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right\} & \text{if } k \geq 1 \end{cases}$$

- 这个等式对于每对顶点都必须计算，并且$k = 1 \ldots n$，所以串行复杂度为$O(n^3)$。

Floyd算法

算法 3　Floyd的全部顶点对间的最短路径算法

```
1.  procedure FLOYD_ALL_PAIRS_SP(A)
2.  begin
3.      D^(0) = A;
4.      for k := 1 to n do
5.          for i := 1 to n do
6.              for j := 1 to n do
7.                  d_{i,j}^(k) := min(d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1));
8.  end FLOYD_ALL_PAIRS_SP
```

注：程序用邻接矩阵A计算图$G = (V, E)$的全部顶点对间的最短路径。

Floyd算法：使用2维块映射的并行形式

- 矩阵$D^{(k)}$分为大小为$(n / \sqrt{p}) \times (n / \sqrt{p})$的p个块。
- 在每次迭代中，每个进程更新矩阵中自己的部分。
- 进程$P_{i,j}$为了计算$d_{l,r}^{(k)}$必须得到$d_{l,k}^{(k-1)}$和$d_{k,r}^{(k-1)}$。
- 在算法的第k次迭代，$\sqrt{p}$个进程中包含第k行一部分的每个进程都将这一部分发送给同一列的$\sqrt{p} - 1$个进程。
- 同样，$\sqrt{p}$个进程中包含第k列一部分的每个进程都将这一部分发送给同一行的$\sqrt{p} - 1$个进程。

Floyd算法：使用2维块映射的并行形式

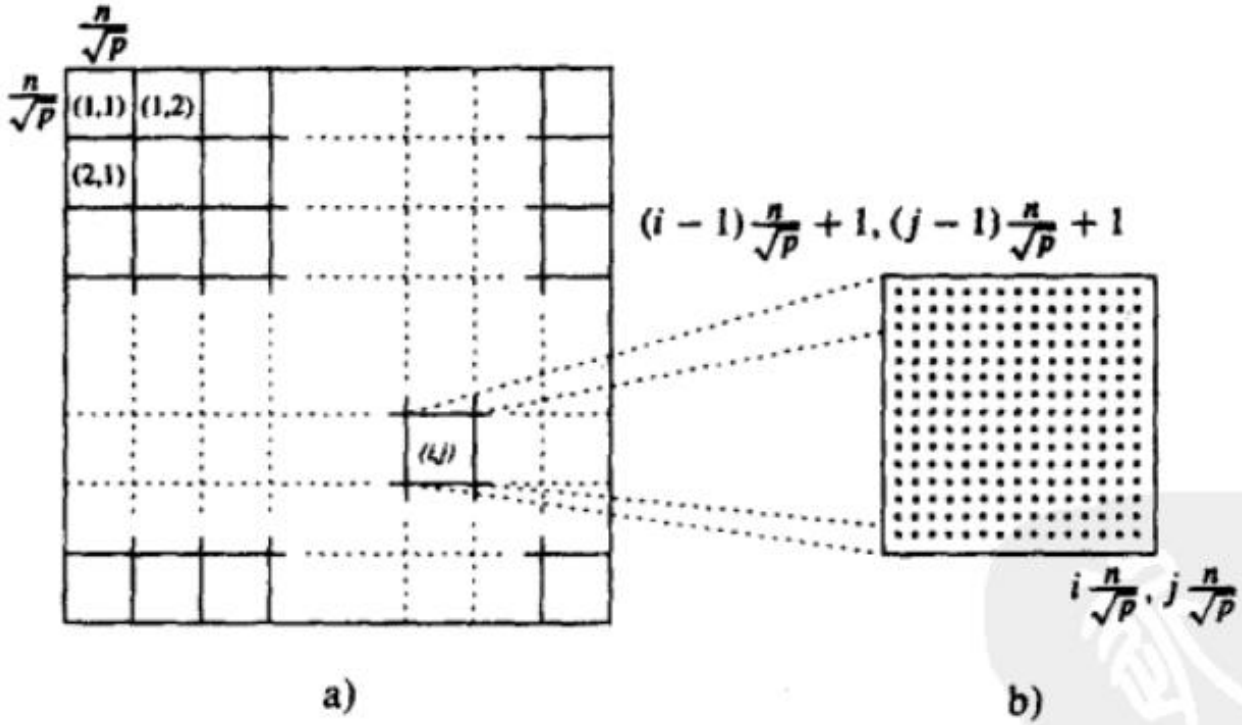


图 7　a）用2维块映射将矩阵$D^{(k)}$分成 $\sqrt{p} \times \sqrt{p}$ 个子块；　b）$D^{(k)}$的子块分配给进程$P_{i,j}$

Floyd算法：使用2维块映射的并行形式

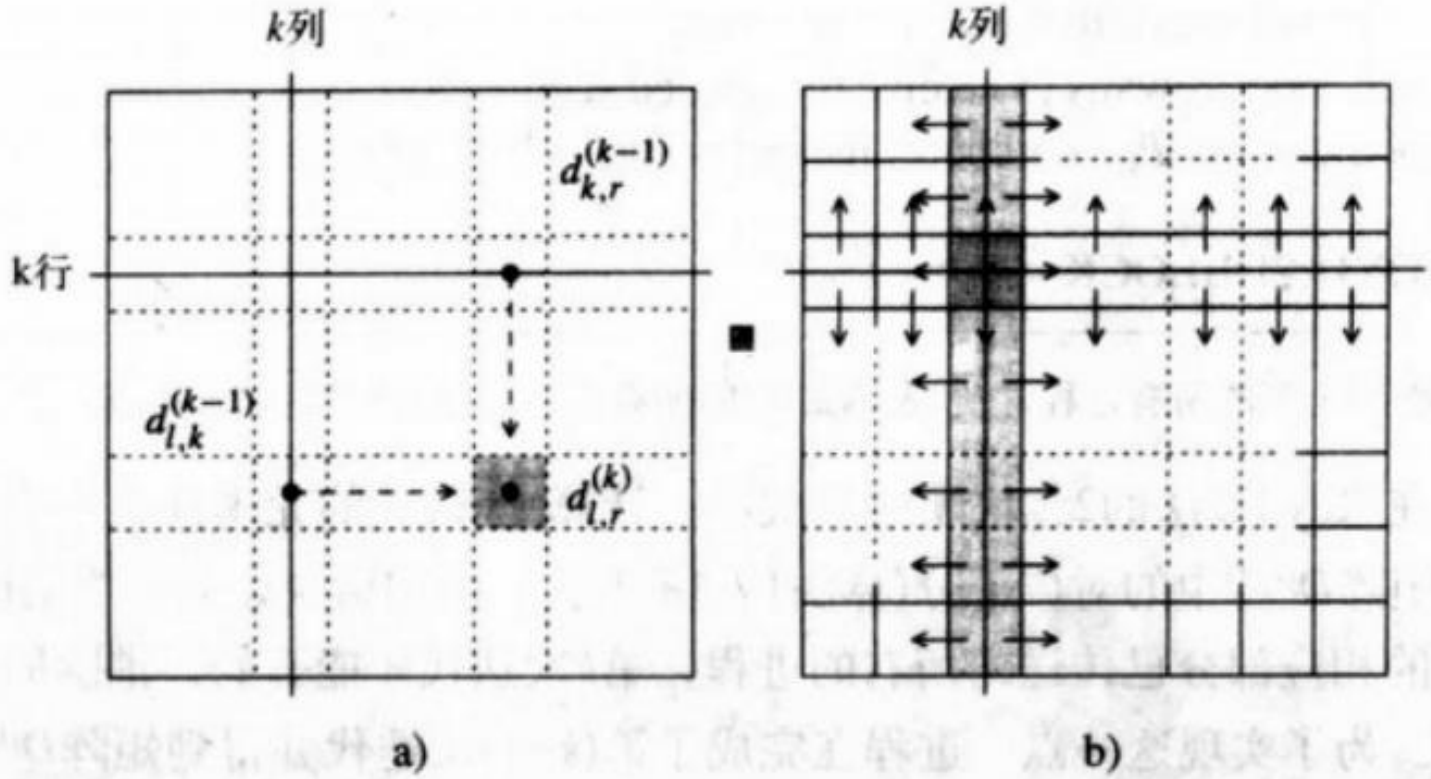


图 8 a）使用2维块映射的通信模式。计算 $d_{i,j}^{(k)}$ 时，信息必须从同一行和同一列的其他两个进程发送到高亮的进程； b）包含第 k 行和 k 列的 $\sqrt{p}$ 个进程的行和列沿进程的列和行发送信息

Floyd算法：使用2维块映射的并行形式

- 使用2维块映射的Floyd算法并行形式

```
1.      procedure FLOYD_2DBLOCK(D^(0))
2.      begin
3.          for k := 1 to n do
4.          begin
5.              each process P_{i,j} that has a segment of the k^th row of D^(k-1);
                    broadcasts it to the P_{*,j} processes;
6.              each process P_{i,j} that has a segment of the k^th column of D^(k-1);
                    broadcasts it to the P_{i,*} processes;
7.              each process waits to receive the needed segments;
8.              each process P_{i,j} computes its part of the D^(k) matrix;
9.          end
10.     end FLOYD_2DBLOCK
```

注：$P_{*,j}$表示第j列的所有进程，$P_{i,*}$表示第i行的所有进程。矩阵$D^{(0)}$是邻接矩阵。

Floyd算法：使用2维块映射的并行形式

- 在算法的每次迭代中，拥有第k行和第k列的进程沿着它们的列/行进行一对多广播。
- 这样的广播发送的元素数目为$n/\sqrt{p}$，广播需要的时间为$\Theta((n \log p)/\sqrt{p})$。
- 同步操作需要的时间为$\Theta(\log p)$。
- 计算时间为$\Theta(n^2/p)$。
- 使用2维块映射的Floyd算法的并行时间为：

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta\left(\frac{n^2}{\sqrt{p}}\log p\right)}^{\text{communication}}.$$

Floyd算法：使用2维块映射的并行形式

- 对于成本最优的并行形式，上述公式可以使用$O(n^2 / \log^2 n)$个进程。
- 等效率函数为$\Theta(p^{1.5} \log^3 p)$。
- 通过放宽每次迭代中同步的限制，该算法还能得到进一步的改进。

Floyd算法：使用流水线加速

- 消除并行Floyd算法的同步步骤不会影响算法 的正确性。
- 每个进程只要完成了第k-1次迭代并且得到$D^{(k-1)}$矩阵的相关部分即可开始第k次迭代的计算。

Floyd算法：使用流水线加速

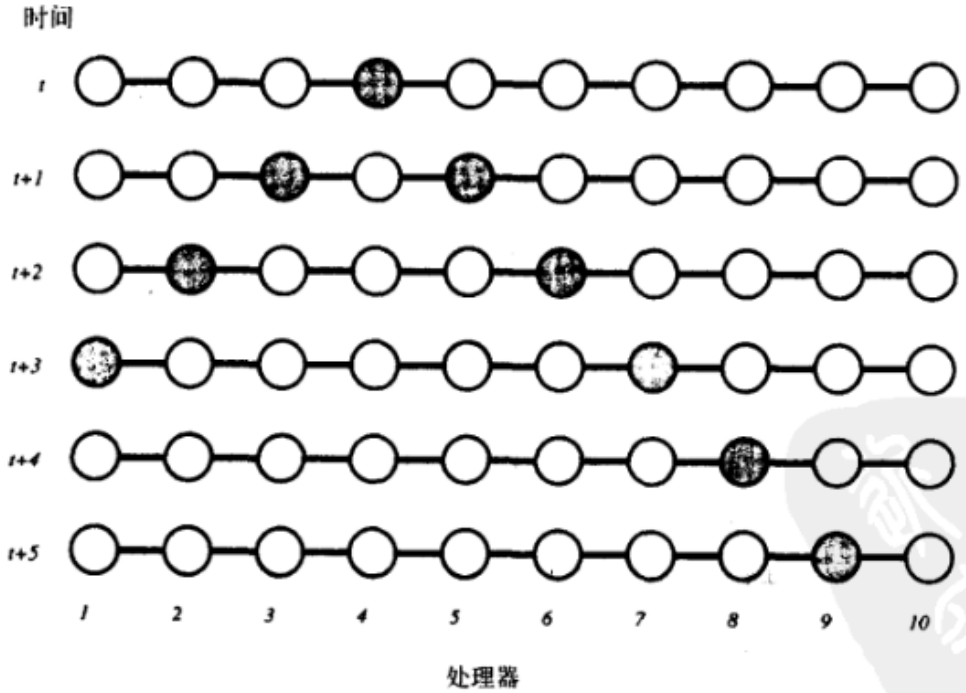


图 9 使用流水线2维映射的Floyd算法并行形式的通信协议

注：假设进程4在时刻t刚计算出矩阵$D^{(k-1)}$的第k列的一部分。它将此部分发送给进程3和进程5。在$t+1$时刻，这两个进程收到这个部分（时间单位为矩阵部分在邻接的进程间的通信链路上传送所需时间）。同样，远离进程4的进程收到该部分的时间更迟。进程1（在边界上）收到该部分后并不转发。

Floyd算法：使用流水线加速

- 考虑第一次迭代中元素的移动。在每一步，第一行的$n/\sqrt{p}$个元素从进程$P_{i,j}$发送到进程$P_{i+1,j}$。
- 同样，第一列的元素从进程$P_{i,j}$发送到进程$P_{i,j+1}$。
- 每一步耗时$\Theta(n/\sqrt{p})$。
- 在$\Theta(\sqrt{p})$步后，进程$P_{\sqrt{p},\sqrt{p}}$得到第一行和第一列的相应元素，时间为$\Theta(n)$。
- 在流水线模式下，后续的行和列的元素在耗时$\Theta(n^2/p)$后继续传输过来。
- 进程$P_{\sqrt{p},\sqrt{p}}$完成最短路径中自己部分的时间为$\Theta(n^3/p) + \Theta(n)$。
- 当进程$P_{\sqrt{p},\sqrt{p}}$完成第(n-1)迭代后，就将第n行和第n列的相应值发送给其他的进程。

Floyd算法：使用流水线加速

- 总体并行时间为：

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n).}^{\text{communication}}$$

- 在成本最优的并行形式下，Floyd算法的流水线形式最多能有效使用$O(n^2)$个进程。
- 相关的等效率函数为$\Theta(p^{1.5})$。

全部顶点对的最短距离：性能比较

表 1　对分带宽为O(p)的不同体系结构上，全部顶点对间的最短路径算法的性能及可扩展性，如果进程能正确映射到底层处理器，则k-d立方体系结构的运行时间也相同

	$E=\Theta(1)$时的最大进程数	对应的并行运行时间	等效率函数
Dijkstra源划分	$\Theta(n)$	$\Theta(n^2)$	$\Theta(p^3)$
Dijkstra源并行	$\Theta(n^2/\log n)$	$\Theta(n \log n)$	$\Theta((p \log p)^{1.5})$
Floyd一维块映射	$\Theta(n/\log n)$	$\Theta(n^2 \log n)$	$\Theta((p \log p)^3)$
Floyd二维块映射	$\Theta(n^2/\log^2 n)$	$\Theta(n \log^2 n)$	$\Theta(p^{1.5}\log^3 p)$
Floyd流水线二维块映射	$\Theta(n^2)$	$\Theta(n)$	$\Theta(p^{1.5})$

5 传递闭包

传递闭包

- 如果$G = (V,E)$是一个图，G的传递闭包定义为$G^* = (V,E^*)$，其中$E^* = \{(v_i,v_j) \mid G$中存在从v_i到v_j的路径$\}$。
- G的连通性矩阵为这样一个矩阵$A^* = (a_{i,j}^*)$，如果存在从v_i到v_j的路径或$i = j$，则$a_{i,j}^* = 1$，否则$a_{i,j}^* = \infty$。
- 为了计算A^*，我们对E的每条边赋权值1，并对这个加权图使用任一全部顶点对间的最短路径算法。

6 连通分量

连通分量

- 一个无向图的连通分量是顶点在连通关系下的等价类。

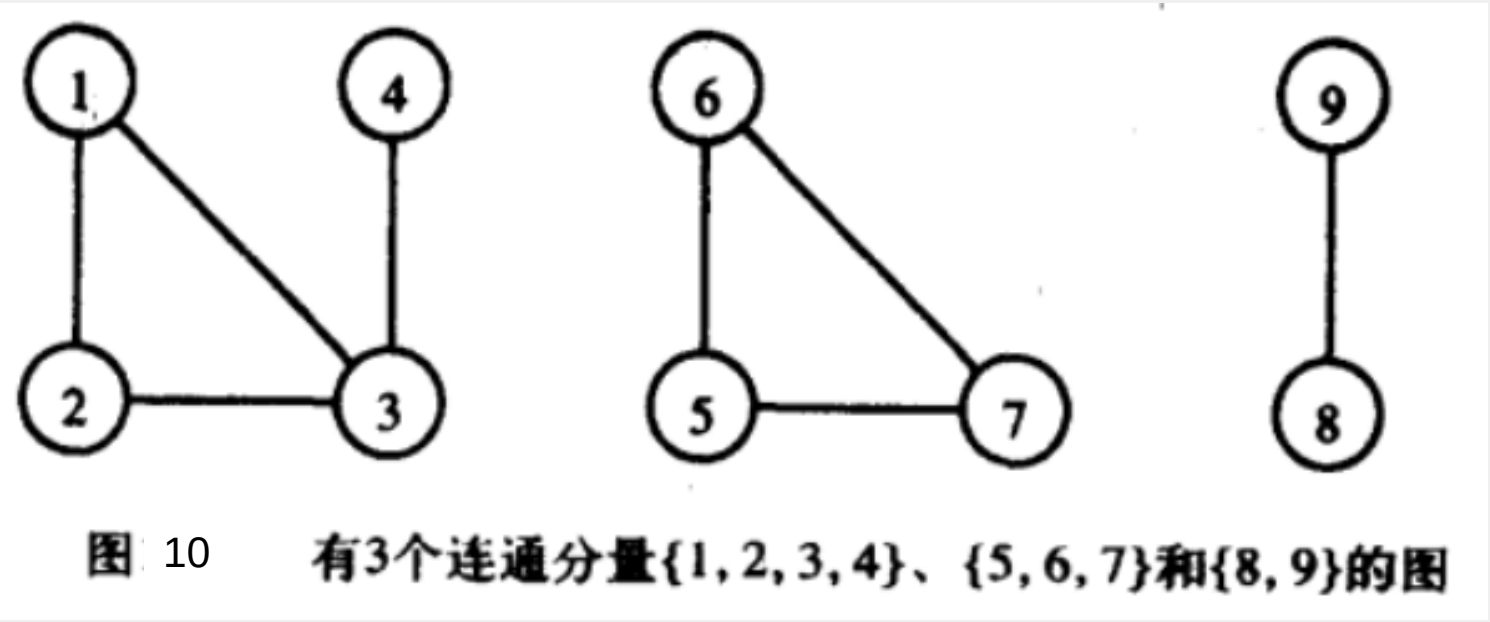


图 10　有3个连通分量{1, 2, 3, 4}、{5, 6, 7}和{8, 9}的图

连通分量：深度优先搜索算法

- 深度优先遍历的结果是深度优先树的树林，树林中每一棵树都包含属于不同连通分量的顶点。

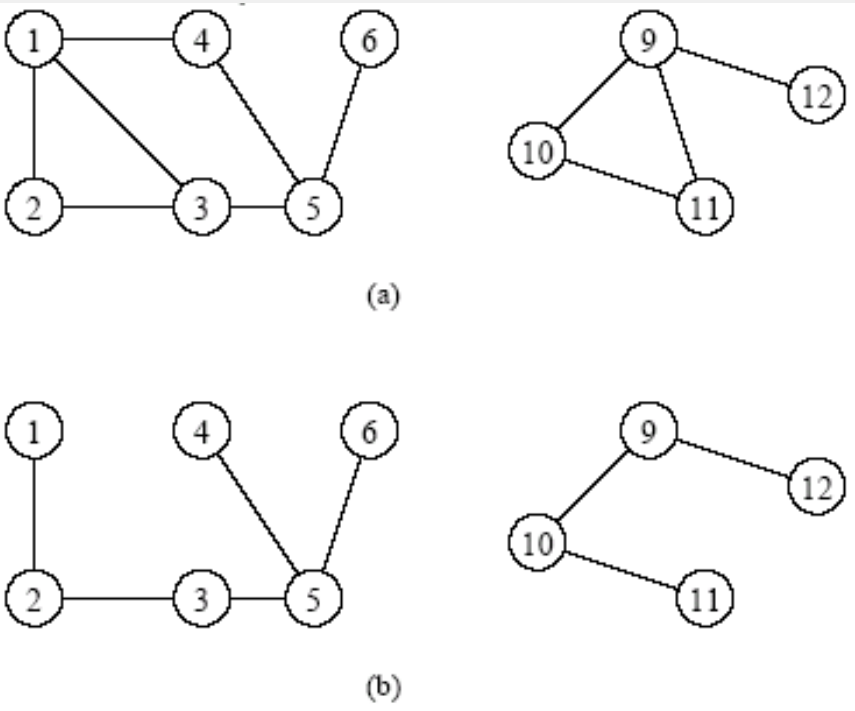


图 11　图b）是对图 a）进行深度优先遍历得到的深度优先树林。这些树的每一棵都是图 a）的连通分量

连通分量：并行形式

- 根据处理器数量分划图，并且每个处理器在各自对应的子图上独立运行连通分量搜索算法。我们就得到了p个生成树林。
- 第二步，生成树林按对合并直到只剩下一个生成树。

连通分量：并行形式

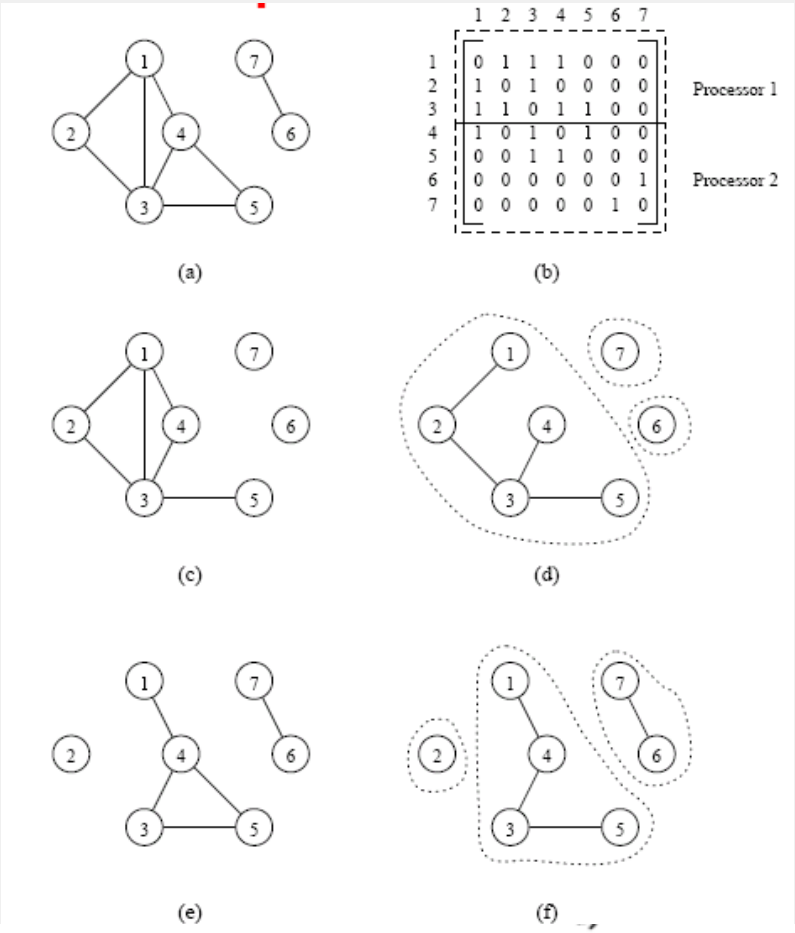


图 12　并行计算连通分量

注：a）中图G的邻接矩阵被分成如图 b）所示的两个部分。接下来，每个进程得到如图 c）和图 e）所示的G的子图。然后每个进程计算子图的生成树林，如图 d）和图 f）所示。最后，两棵生成树合并得到最终结果。

连通分量：并行形式

- 为了有效地合并成对的生成树林，算法只使用不相连的边集。
- 我们定义在不相连的边集上的操作：
- ***find*(*x*)**——返回一个指针，指向包含x的集合中的代表元素。每个集合都有其唯一的代表。
- ***union*(*x*, *y*)**——合并包含元素x和y的集合。假定这两个集合在操作前不相连。

连通分量：并行形式

- 假设要把树林A合并到树林B。对A的每一条边*(u,v)*，将边上的每一个顶点指向一次find操作确定是否两个顶点已处于B中同一棵树上。
- 如果不是，则B中包含u和v的两棵树（集合）通过union操作合并。
- 否则不需要union操作。
- 因此，合并A和B最多需要*2(n-1)*次find操作和*(n-1)*次union操作。

连通分量：并行一维块映射

- $n \times n$的邻接矩阵被划分为p个条带。
- 每个处理器能够在$\Theta(n^2/p)$时间内计算它本地的生成树林。
- 通过在拓扑中嵌入逻辑树来完成合并。总共有$\log p$个合并阶段，每阶段使用$\Theta(n)$时间。因此合并的时间为$\Theta(n \log p)$。
- 最后，在每个合并阶段，生成树林在最近的邻居间传送。回忆生成树林要传送$\Theta(n)$条边。

连通分量：并行一维块映射

- 连通分量算法的并行运行时间为：

$$T_P = \overbrace{\Theta\left(\frac{n^2}{p}\right)}^{\text{local computation}} + \overbrace{\Theta(n \log p).}^{\text{forest merging}}$$

- 对于成本最优的并行形式为$p = O(n / \log n)$。相应的等效率函数为$\Theta(p^2 \log^2 p)$。

7 稀疏图算法

- 查找最大独立集
- 单源最短路径

稀疏图算法

- 如果$|E|$远小于$|V|^2$，则图$G = (V,E)$为稀疏图。

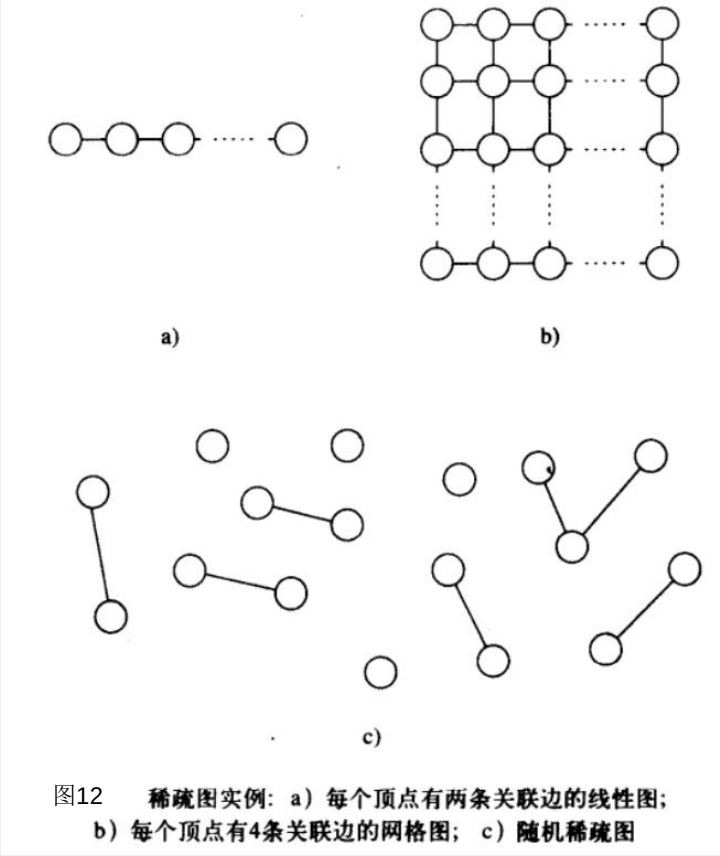


图12　稀疏图实例：a）每个顶点有两条关联边的线性图；b）每个顶点有4条关联边的网格图；c）随机稀疏图

稀疏图算法

- 如果我们利用稀疏性，稠密算法能够被显著地改良。例如，Prim算法的最小生成树算法可以从$\Theta(n^2)$的时间复杂度下降到$\Theta(|E| \log n)$。
- 稀疏图算法使用邻接表而不是邻接矩阵。
- 分划稀疏图的邻接表是一个更为困难的问题——我们是否平衡了点或者边的数量呢？
- 并行算法一般会利用图结构或者度数信息来提高性能。

稀疏图算法

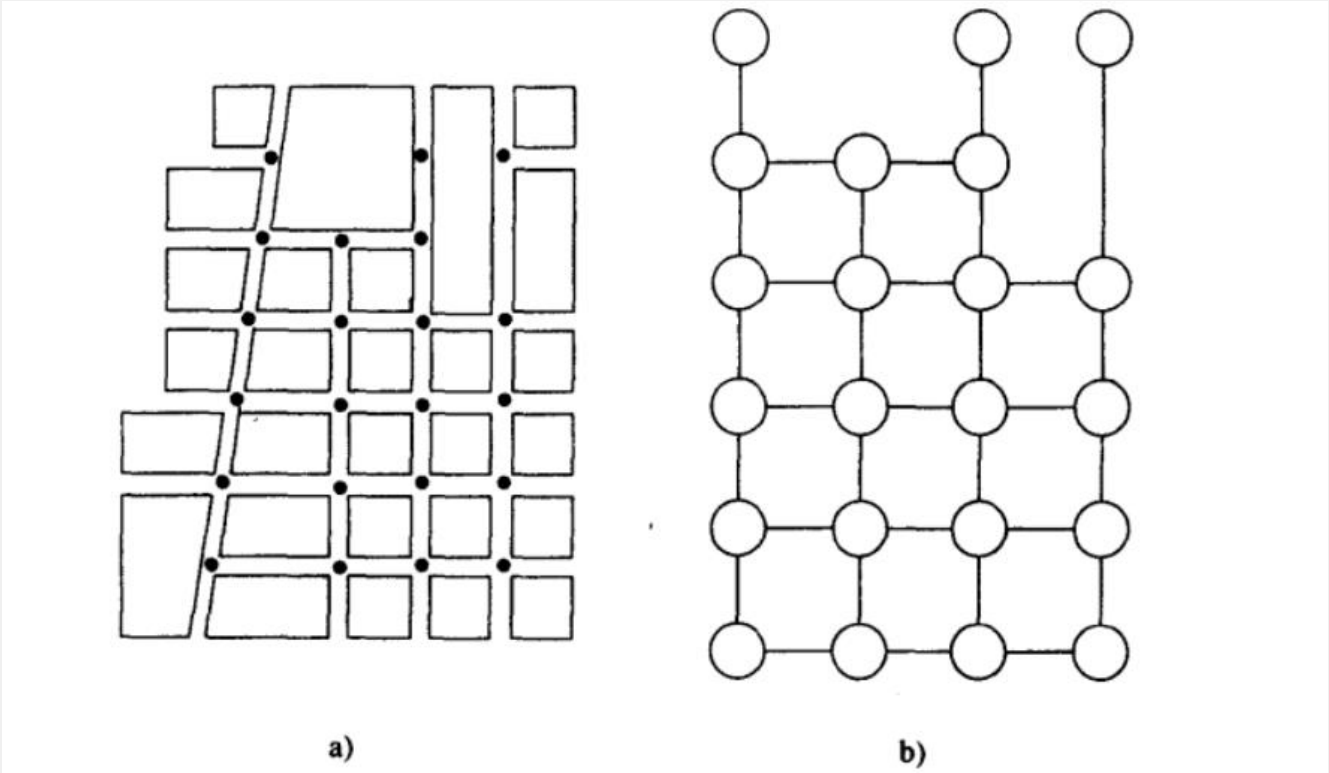
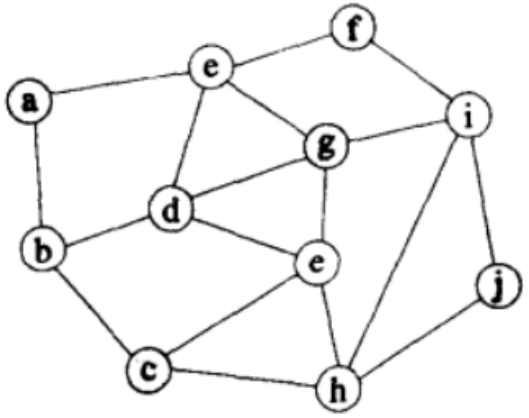


图14　街道地图

注：街道地图 a）可用图 b）来表示。在图 b）中，每个街道的交点是一个顶点，每条边是一段街道。图 b）中的顶点是图 a）中用黑点标出的交点。

查找最大独立集

- 给定无向图$G(V,E)$，如果在I中任何一对顶点都不能由G中的一条边相连，则称顶点集合$I \subset V$为独立集。如果在I中加入不属于I的任何其他顶点集合就不再独立，则独立集I被称为最大的。



{a, d, i, h}是独立集
{a, c, j, f, g}是最大独立集
{a, d, h, f}是最大独立集

图15 独立集及最大独立集示例

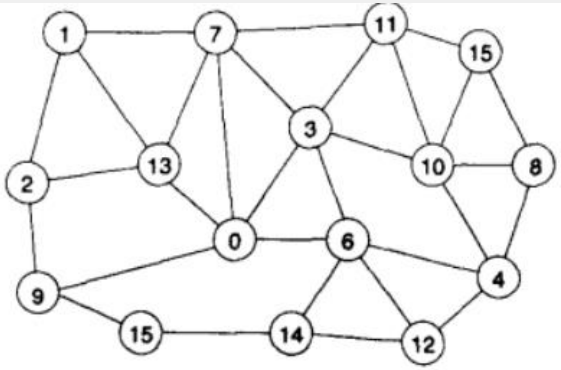
查找最大独立集（MIS）

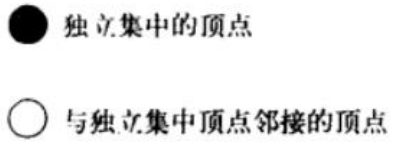
- 最简单的一类算法从初始设置集合*I*为空开始，将所有的顶点放入集合*C*中。
- 将顶点*v*从*C*移到*I*中，并从*C*中除去所有和*v*邻接的顶点。
- 这个过程一直重复以上步骤直到为空。
- 这个过程在本质上是串行的！

查找最大独立集（MIS）

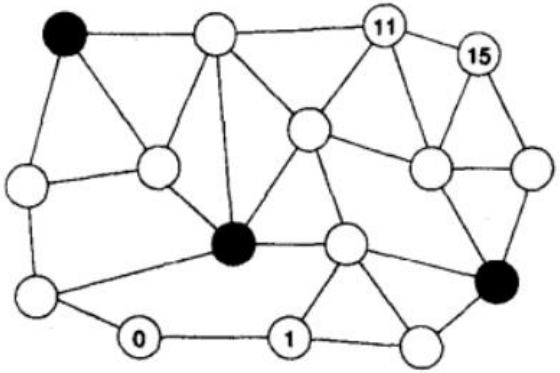
- 并行MIS算法使用随机性来获得并发（由Luby开发的用于计算图的着色问题的随机算法）。
- 初始时每个顶点都在候选集*C*中。一个唯一的随机数被分配给*C*中的每一个顶点，并且使用自己的随机数与邻居通信。
- 如果一个顶点的随机数是它所有相邻顶点中最小的，它被加入到集合*I* 。所选顶点和它的所有相邻顶点从*C*中删去。
- 这个过程持续到*C*变为空集。
- 平均而言，这个算法在$O(\log|V|)$步后收敛。

查找最大独立集（MIS）

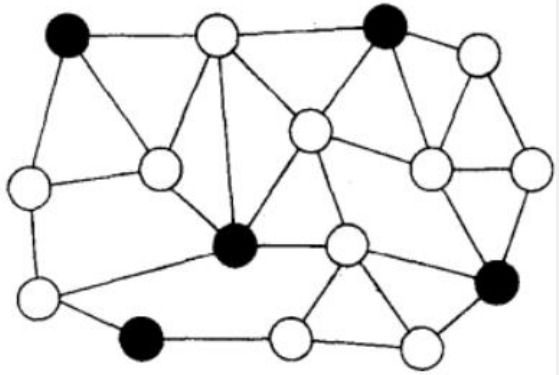




a）第一次随机数赋值后



b）第二次随机数赋值后



c）最后的最大独立集

图16　Luby随机最大独立集算法的不同扩大步骤。每个顶点中的数字对应于分配给每个顶点的随机数

查找最大独立集（MIS）：并行形式

- 我们使用长度为*n*的三个数组： *I*用于存储最大独立集的顶点； *C*用于存储候选顶点； *R*用于存储随机数。
- 根据*p*个处理器划分*C*。每个处理器生成数组*R*的相应值，据此计算那些候选点可以进入最大独立集。
- 通过删除所有被选顶点以及相邻顶点，数组*C*被更新。
- 此算法的性能依赖于图结构。

单源最短路径

- 将Dijkstra算法修改成处理稀疏矩阵的算法称为Johnson算法。
- 算法的修改是根据这样一个因素：Dijkstra算法中只需要对那些已被选择顶点的相邻顶点执行求最小值步骤。
- Johnson算法使用优先队列Q来存储每个顶点$v \in (V - V_T)$的$l[v]$值。

单源最短路径：Johnson算法

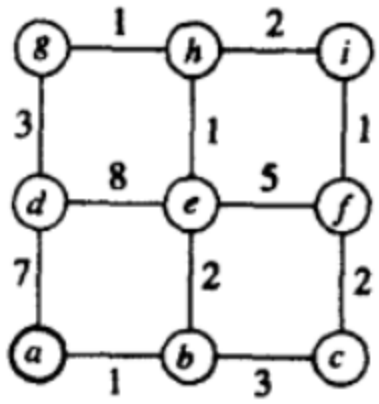
```
1.    procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
2.    begin
3.        Q := V;
4.        for all v ∈ Q do
5.            l[v] := ∞;
6.        l[s] := 0;
7.        while Q ≠ ∅ do
8.        begin
9.            u := extract_min(Q);
10.           for each v ∈ Adj[u] do
11.               if v ∈ Q and l[u] + w(u, v) < l[v] then
12.                   l[v] := l[u] + w(u, v);
13.       endwhile
14.   end JOHNSON_SINGLE_SOURCE_SP
```

算法 5　Johnson的串行单源最短路径算法

单源最短路径：并行Johnson算法

- 严格保持Johnson算法的顺序会导致非常严格的一类并行算法。
- 我们需要让多个顶点同时探索。这样做是通过同时从优先队列提取p个顶点，更新相邻顶点的路径成本，并且扩充最短路径。
- 如果造成了一个错误，它可以（通过一条更短的路径）被发现，并且该顶点连同它的最短路径会被重新插入到优先队列中。

单源最短路径：并行Johnson算法



优先队列		数组l[]								
		a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
(1)	*b:1, d:7, c:inf, e:inf, f:inf, g:inf, h:inf, i:inf*	0	1	∞	7	∞	∞	∞	∞	∞
(2)	*e:3, c:4, g:10, f:inf, h:inf, i:inf*	0	1	4	7	3	∞	10	∞	∞
(3)	*h:4, f:6, i:inf*	0	1	4	7	3	6	10	4	∞
(4)	*g:5, i:6*	0	1	4	7	3	6	5	4	6

图17　用修正后的Johnson算法并发处理不安全顶点的实例

单源最短路径：并行Johnson算法

- 即使我们可以从队列中提取和处理多个顶点，队列本身是主要的瓶颈。
- 由于这个原因，我们使用多个队列，每个队列对应于一个处理器。每个处理器只使用自己的顶点来建立自己的优先队列。
- 当进程P_i提取顶点$u \in V_i$，它将发送一个消息给那些存储u的邻接顶点的进程。
- 收到此消息的进程P_j会将存储在优先队列的$l[v]$值设置为$min\{l[v],l[u] + w(u,v)\}$。

单源最短路径：并行Johnson算法

- 如果到顶点v的更短路径被发现，它会被从新插入到本地优先队列。
- 只有当所有队列都变为空时，算法才会终止。
- 许多点划分方案可以通过利用图结构来提升性能。