数据结构

第十一讲　最短路径

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年11月9日

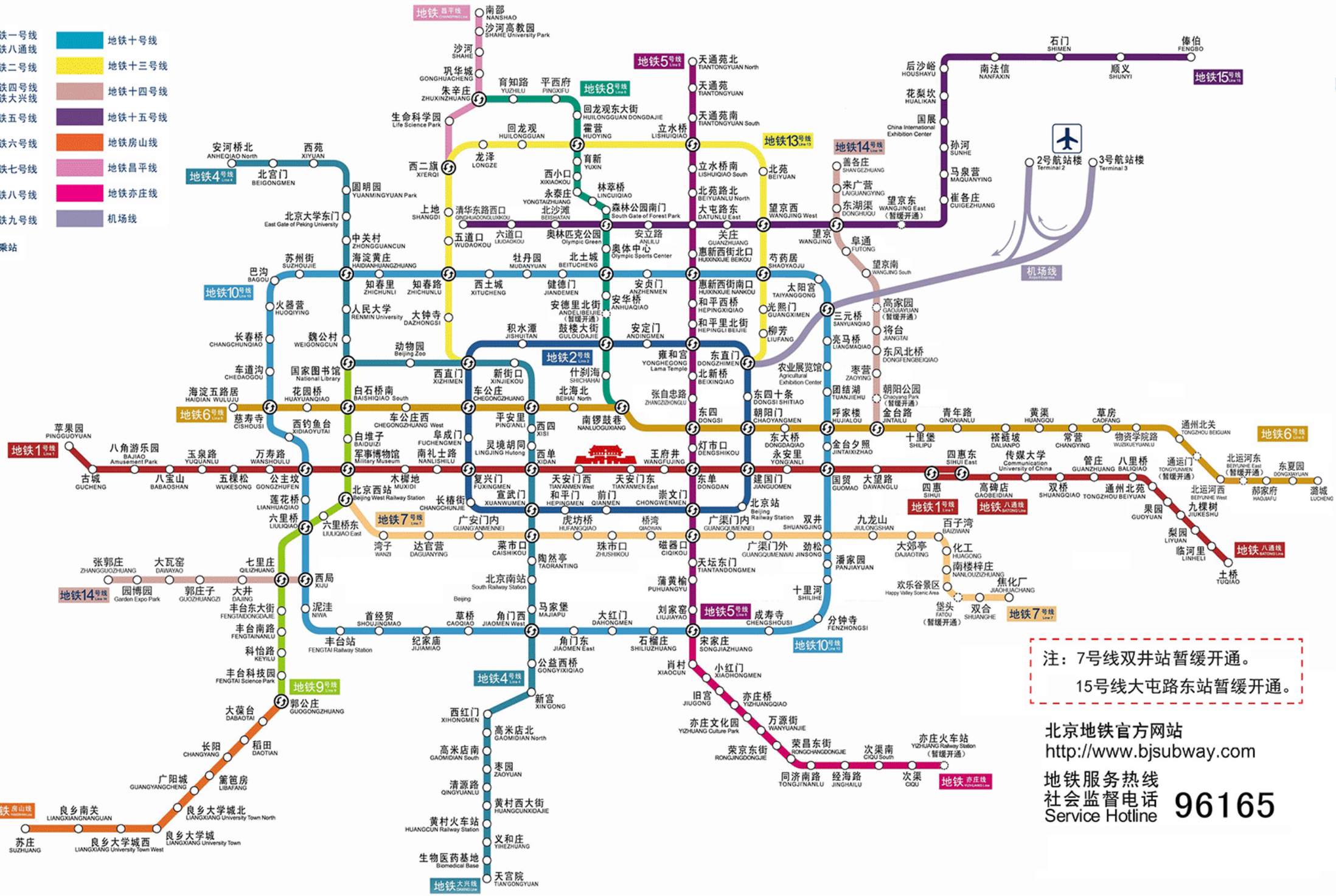
北京地铁线路图

Beijing Subway Map

图例：
Legend

- 地铁一号线 / 地铁八通线
- 地铁二号线
- 地铁四号线 / 地铁大兴线
- 地铁五号线
- 地铁六号线
- 地铁七号线
- 地铁八号线
- 地铁九号线
- 地铁十号线
- 地铁十三号线
- 地铁十四号线
- 地铁十五号线
- 地铁房山线
- 地铁昌平线
- 地铁亦庄线
- 机场线
- 换乘站

北 NORTH



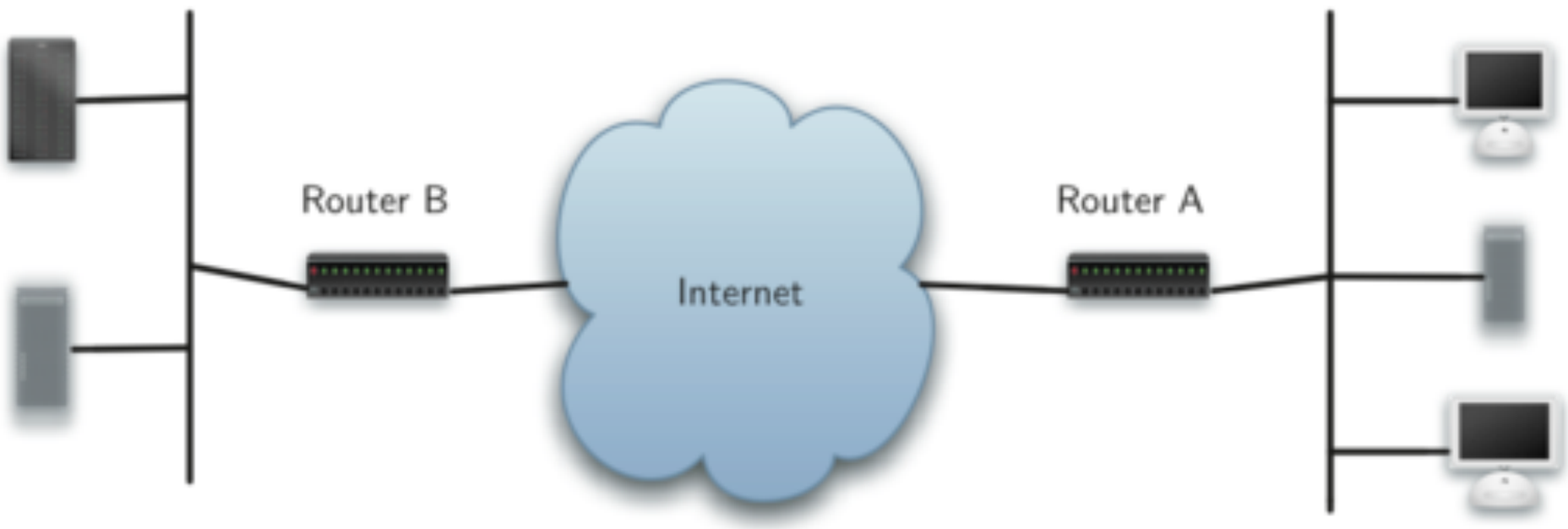
注：7号线双井站暂缓开通。
15号线大屯路东站暂缓开通。

北京地铁官方网站
http://www.bjsubway.com

地铁服务热线
社会监督电话 96165
Service Hotline

课程内容

- **Dijkstra算法**
- **Floyd算法**

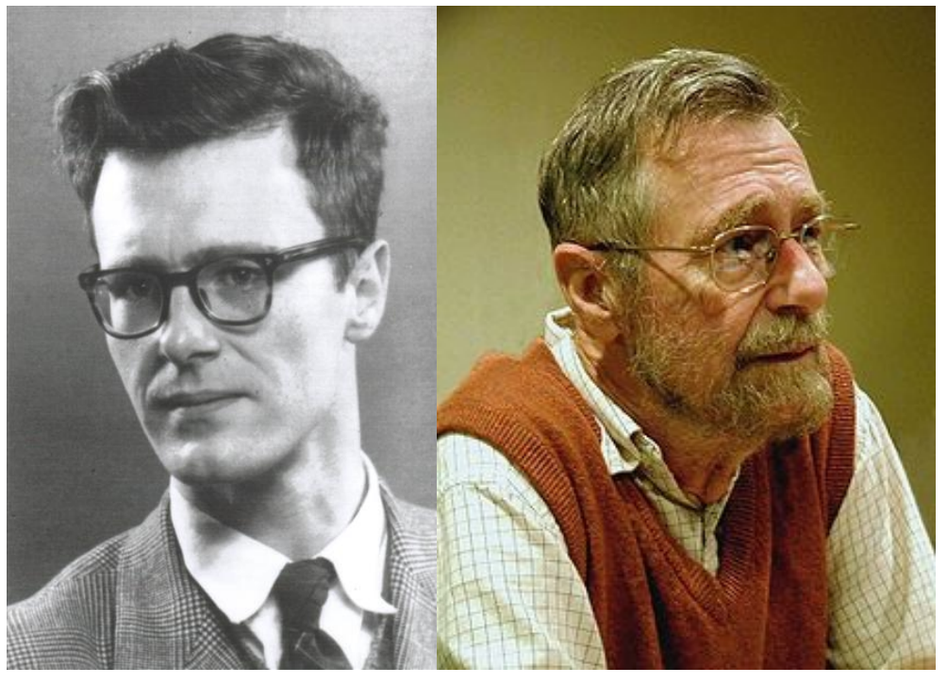


最短路径

- 如果图中从一个顶点可以到达另一个顶点，则称这两个顶点间存在一条路径。
- 从一个顶点到另一个顶点间可能存在多条路径，而每条路径上经过的边数并不一定相同。
- 如果图是一个带权图，则路径长度为路径上各边的权值的总和。
- 两个顶点之间路径长度最短的那条路径称为两个顶点间的**最短路径**，其路径长度称为**最短路径长度**。

Dijkstra算法

- Dijkstra算法求解从顶点v_0出发到其它各顶点最短路径。
- 按路径长度递增的次序产生最短路径，该算法假设所有边的权都大于等于零。



Edsger W. Dijkstra
(1930-2002)

基本思想

- 设置一个集合U存放已求出最短路径的顶点，$V-U$是尚未确定最短路径的顶点集合。
- 每个顶点对应一个距离值，集合U中顶点的距离值是从顶点v_0到该顶点的最短路径长度；
- 集合$V-U$中顶点的距离值是从顶点v_0到该顶点的只包括集合U中顶点为中间顶点的最短路径长度。
- U的初始状态为$\{v_0\}$。
- 按路径长度递增的次序逐个产生v_x的最短路径，把v_x加入U中。直到$U=V$时终止。

如何为 U 扩充顶点，在 $V-U$ 里找到下一能确定最短路径的顶点?

$v_i \in V-U$ 的已知最短距离定义为：

若存在 $v_j \in U$ 有边(v_j, v_i)，则

$\mathbf{cdis}(v_i)=\mathbf{min}\{\mathbf{dis}(v_j)+\mathbf{w}(v_j,v_i)|v_j \in U\}$。

$\mathbf{dis}(v_j)$表示从v_0到v_j的距离，$\mathbf{w}(v_j,v_i)$是(v_j, v_i) 的权；

若没有这种 v_j，定义$\mathbf{cdis}(v_i)=\infty$。

性质：若v_i是$V-U$中cdis值最小的结点，那么$\mathbf{dis}(v_i)=\mathbf{cdis}(v_i)$。

即：从v_0到v_i的最短路径已知，可以把 v_i 加入集合 U。

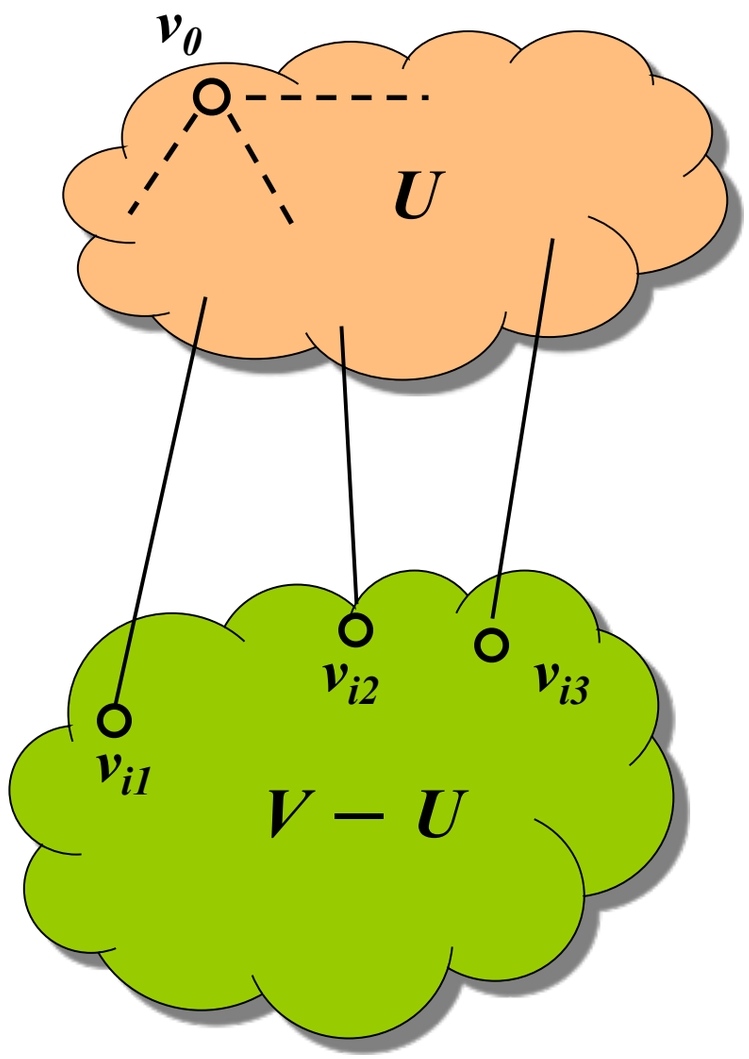


图 G

按路径长度递增的次序逐个产生最短路径

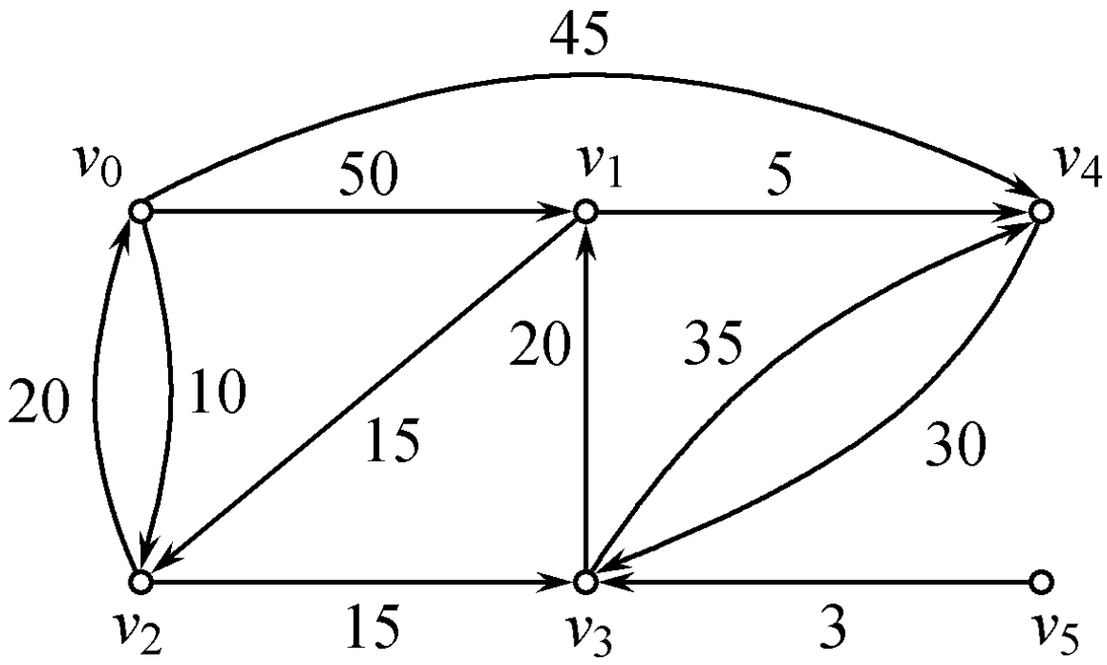
- 初始状态：
  - 集合U中只有顶点v_0，顶点v_0对应的距离值为0，集合$V-U$中顶点v_i的距离值为边(v_0,v_i) (i=1,2,…,n-1)的权，如果v_0和v_i间无边直接相连，则v_i的距离值为∞（实际程序中可以用一个足够大的数代替）。
- 处理框架：

（1）在集合$V-U$中选择距离值最小的顶点v_{min}加入集合U；

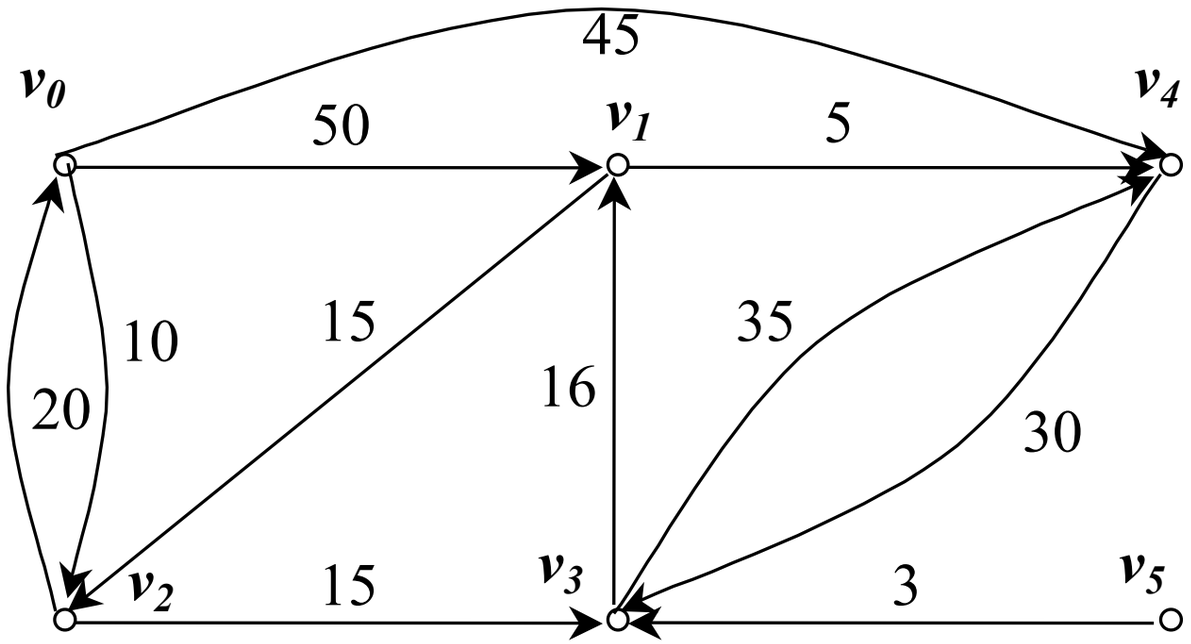
（2）由于v_{min}加入，$V-U$中某些顶点的已知最短路径长度可能发生改变。对集合$V-U$中各顶点的距离值进行修正：如果加入顶点v_{min}为中间顶点后，使v_0经过v_{min}到v_i的距离值比原来的距离值更小，则修改v_i的距离值，该路径为新的最短路径。

（3）重复（1）、（2）操作，直到从v_0出发可以到达的所有顶点都在U中为止。

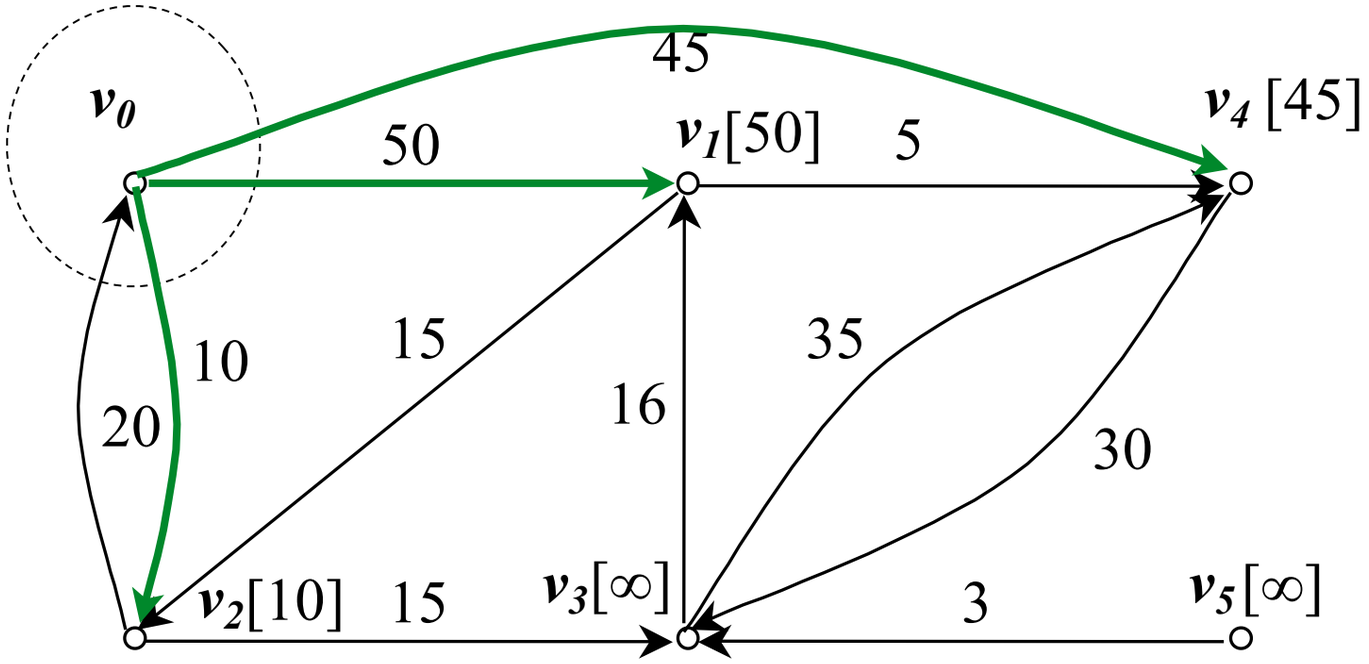
例子



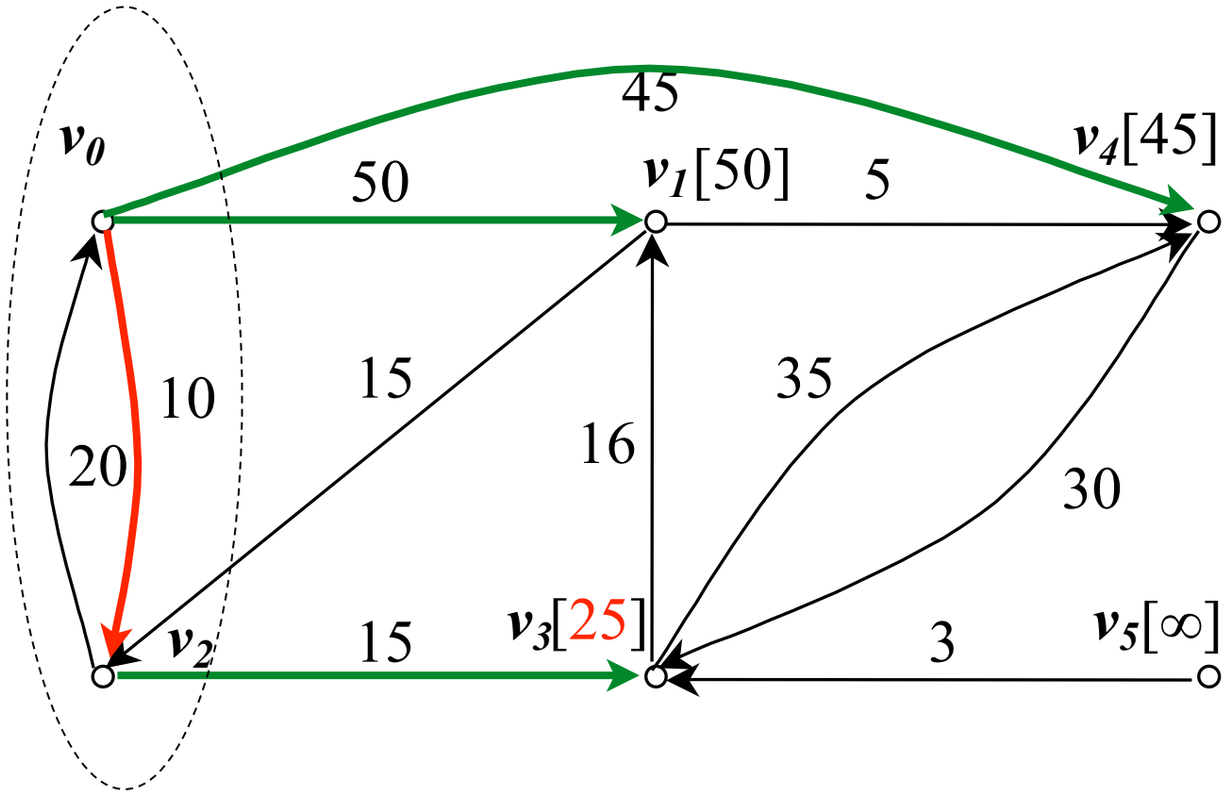
$$\text{arcs} = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 20 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$



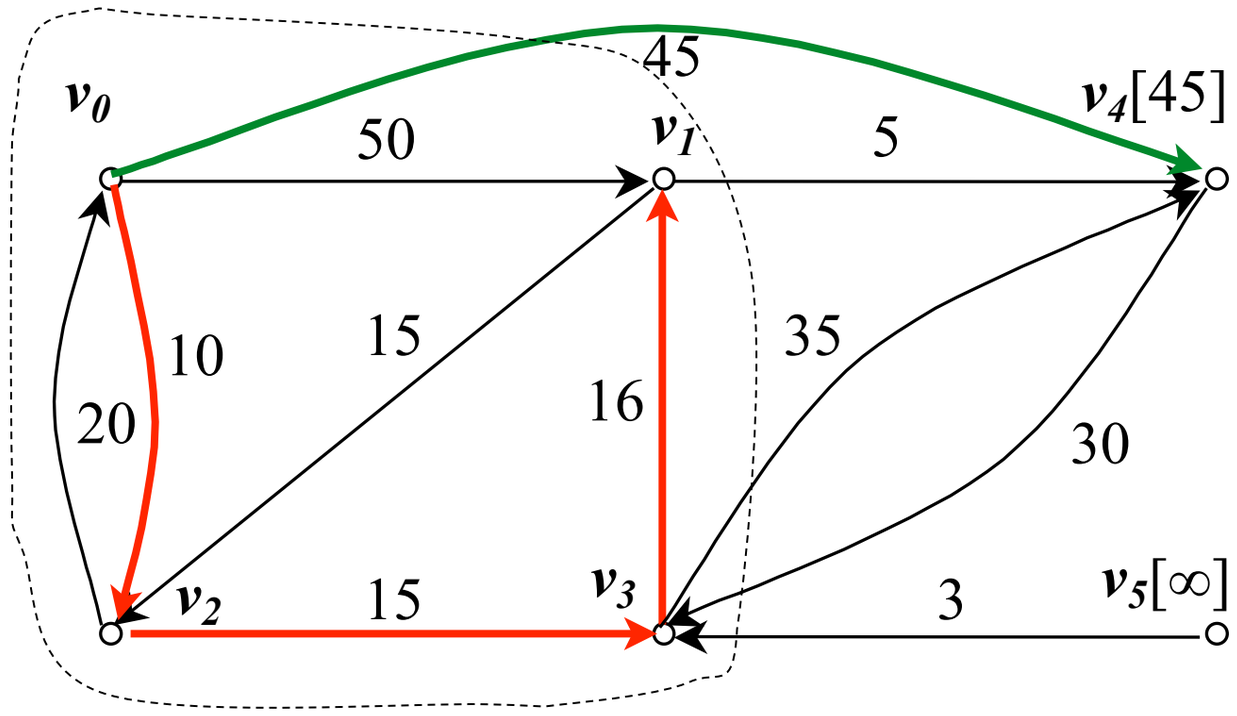
$$A = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 16 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$



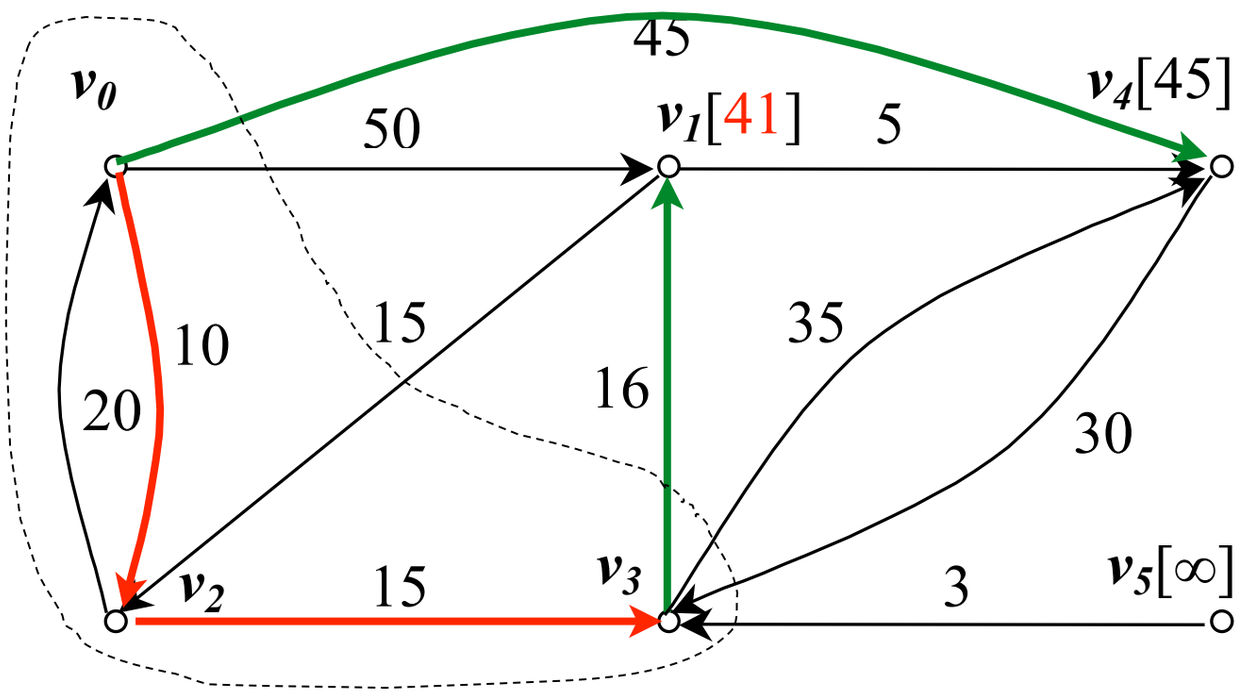
绿色为从U到$V-U$的边，
选 (v_0, v_2) 把v_2加入U



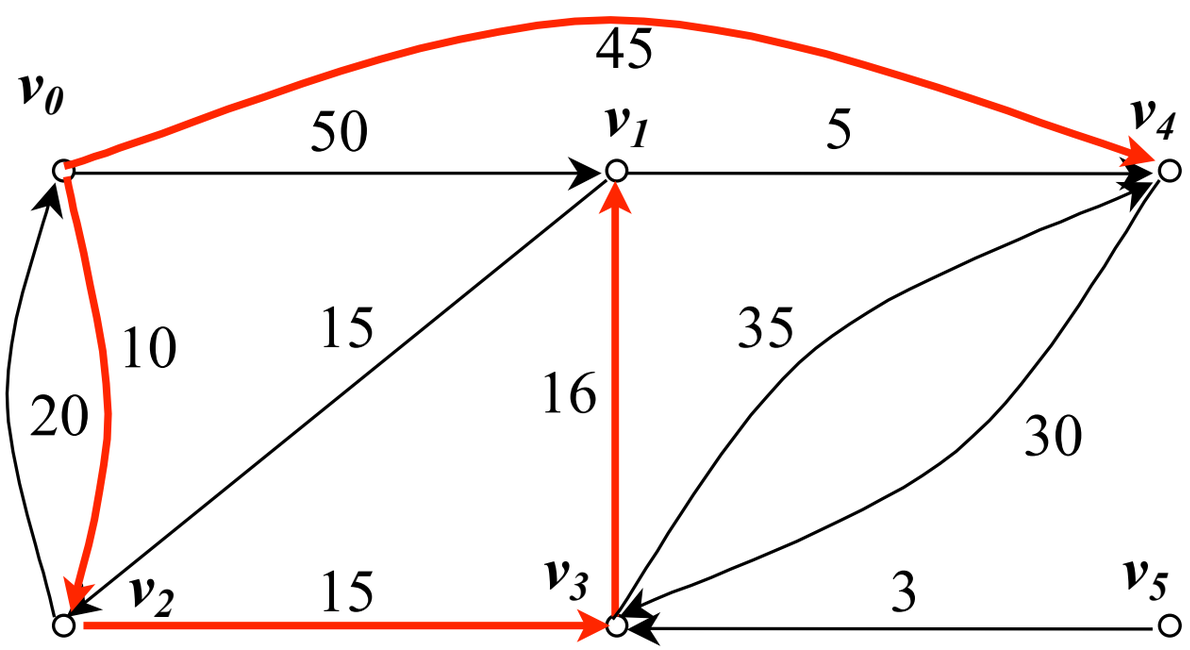
选 (v_2, v_3)，把v_3加入U



选 (v_0, v_4)，把v_4加入U



选 (v_3, v_1)，把v_1加入U



存储结构

- 图选择邻接矩阵表示法，其中关系矩阵的对角线初值均取0。算法中，将放入集合U中结点对应的关系矩阵中对角线元素值修改为1；
- 另外设置一个数组dist，dist[i]用于存放v_0到顶点v_i的最短路径及其最短路径长度（计算过程中为距离值）：

```
typedef struct {
 AdjType length;    /* 最短路径长度 */
 int prevex;        /*从v0到达vi(i=1,2,…n-1)的最短路径上
                      vi 的前驱顶点*/
} Path;
Path dist[VN];
```

修正距离值方法的精化

如果加入顶点v_{min}为中间顶点后

dist[i].length>dist[min].length+G.arcs[min][i],

则将顶点v_i的距离值改为dist[min].length+G.arcs[min][i],

并修改路径上v_i的前驱顶点：dist[i].prevex=min。

程序实现：

```
void dijkstra(GraphMatrix graph, Path* dist)
```

①. 初始时，集合U中只有顶点v0。

结果dist[n]为[(0, 0), (50, 0), (10, 0), (MAX, -1), (45, 0), (MAX, -1)]

②. 在集合V-U中找出距离值最小的顶点v2，将顶点v2加入集合U中。

结果dist[n]为[(0, 0), (50, 0), (10, 0), (MAX, -1), (45, 0), (MAX, -1)]

③. 按min=2调整集合V-U中顶点的距离值。

因为dist[1].length=50，

dist[2].length+graph.arcs[2][1]=10+MAX，顶点v1的距离值不需要调整。

因为dist[3].length=MAX，

dist[2].length+graph.arcs[2][3]=10+15=25，顶点v3的距离值调整为25，其前驱顶点为v2。

同理，顶点v4 ， v5的距离值不需要调整。

结果dist[n]为[(0, 0), (50, 0), (10, 0), (25, 2), (45, 0), (MAX, -1)]

④．同理在集合V-U中找出当前距离值最小的顶点v3，并调整集合V-U中顶点的距离值。

结果dist[n]为[(0, 0), (41, 3), (10, 0), (25, 2), (45, 0), (MAX, -1)]

⑤．在集合V-U中找出当前距离值最小的顶点v1。并调整集合V-U中顶点的距离值。

结果dist[n]为[(0, 0), (41, 3), (10, 0), (25, 2), (45, 0), (MAX, -1)]

⑥．在集合V-U中找出当前距离值最小的顶点v4。并调整集合V-U中顶点的距离值。

结果dist[n]为[(0, 0), (41, 3), (10, 0), (25, 2), (45, 0), (MAX, -1)]

⑦．没有可以再加入集合U的顶点了，说明从顶点v0到顶点v5之间无路径相通。过程结束。

由dist的prevex字段得到顶点v_0到各顶点的最短路径

[(0, 0), (41, 3), (10, 0), (25, 2), (45, 0), (MAX, -1)]

- 如从v_0到v_1的最短路径，dist[1].prevex=3可知路径上顶点v_1的前一个顶点是v_3，
- dist[3].prevex=2可知路径上顶点v_3的前一个顶点是v_2，
- dist[2].prevex=0可知路径上前一个顶点是v_0，即最短路径为(v_0,v_2,v_3,v_1)。
- 其路径长度为dist[1].length=41。

Dijkstra算法的实现

```
void init(GraphMatrix* pgraph, Path dist[]) {
    int i;
    dist[0].length = 0;
    dist[0].prevex = 0;
    pgraph->arcs[0][0] = 1;
                        /* 用矩阵对角线元素记录集合 U，为 1 表示在U中。
                           也可修改 Path 类型的定义，在数组 dist 里记录 */
    for(i = 1; i < VN; ++ i) {        /* 初始化V-U中顶点的距离 */
        dist[i].length = pgraph->arcs[0][i];
        if (dist[i].length != MAX)
            dist[i].prevex = 0;
        else  dist[i].prevex = -1;
    }
}
```

Dijkstra算法的实现

```
void dijkstra(GraphMatrix *graph, Path dist[ ]) {
    int i, j, mv; AdjType min;
    init(graph, dist); /* 初始化，集合U中只有顶点v0 */
    for(i = 1; i < VN; ++ i) {
        min = MAX; mv = 0;
        for (j = 1; j < VN; ++ j)/*在V-U中选出距 v0 最近的顶点*/
          if( graph->arcs[j][j] == 0 && dist[j].length < min )
          {   mv = j;    min = dist[j].length;   }
       if (mv == 0) break;  /* v0 与 V-U 的顶点不连通，结束 */
       graph->arcs[mv][mv] = 1;/* 顶点mv加入U */
       for (j = 1; j < VN; ++ j){ /*调整V-U 顶点的已知最短路径*/
          if (graph->arcs[j][j] == 0 &&  /*为V-U的顶点*/
           dist[j].length>dist[mv].length + graph->arcs[mv][j]) {
              dist[j].prevex = mv; /* 调整已知最短路径信息 */
              dist[j].length=dist[mv].length+graph->arcs[mv][j];
    }  }  }
}
```

算法分析

- **Dijkstra算法中的初始化部分的时间复杂度为O(n)，求最短路径部分由一个大循环组成，其中外循环运行n-1次，内循环为两个，均运行n-1次，因此，算法的时间复杂度为$O(n^2)$。**
- **另外需要注意，在算法中改变了关系矩阵中的初始状态，如果要求算法不能破坏原始数据，就需要另外增加数据结构记录U集合的值。空间代价是O(n) 。**

Floyd算法

- 求每一对顶点间的最短路径。
- 通过多次迭代计算每一对顶点间的最短路径的缩短变化。
- Floyd算法的基本想法来自Warshall 的强连通子图算法（基于邻接矩阵），实际上是求可达性（有边相邻）的传递闭包。

Robert W. Floyd
(1936-2001)

基本思想

- 图采用邻接矩阵作为存储结构。把关系矩阵A看成是没有经过任何中间结点，直接可以到达的每一对顶点间的最短路径的完整表示。
- 如果有边 $(v, v')\in E$，那么它就是从顶点 v 到 v' 的路径，其长度可以直接得到，即是 $A[v][v']$。但这一路径未必是从 v 到 v' 的最短路径，有可能存在从 v 到 v' 途中经过其它顶点的更短路径。
- 然后经过多次迭代，每次增加一个新结点，在允许这个结点作为中间结点的条件下，计算每一对顶点间的最短路径的缩短变化。
- 直到把所有结点都考虑进去为止，结果得到每一对顶点间的最短路径。

数据结构

- 为了在算法中不破坏原始的关系矩阵，需要定义一个与关系矩阵同样大小，并以关系矩阵作为其初始状态的矩阵，用于处理中存放每对顶点间的距离值（或最短路径长度）。
- 为了保存全部最短路径的轨迹，需要另外设计一个与关系矩阵同样大小的整数矩阵，存放v_i到v_j最短路径上v_i的后继顶点的下标值。把它们封装在下面定义的结构类型ShortPath 中：

```
typedef struct{
   AdjType a[VN][VN];     /*存放每对顶点间最短路径长度 */
   int nextvex[VN][VN];   /*存放vi到vj最短路径上vi的后继顶点
                            的下标值 */
}ShortPath;
```

Floyd算法

- 开始：每对 v 到 v' 的途中不经任何结点的路径长度为已知
  - 有 v 到 v' 的边时就是边的权，无边时认为存在长度为 ∞ 的路径
- $k = 0$：对每对 v 和 v'，除已知直接路径外，从 v 到 v' 途经顶点的下标不大于k（此时是不大于0）的路径可分为两段：
  - $<v, v_0>$，$<v_0, v'>$（如果没有，就认为有长度为 ∞ 的路径）
  - 其长度是两段路径的长度和。比较这一路径和直接路径（是已知最短的），可确定 v 到 v' 的途径顶点下标不大于 0 的最短路径

Floyd算法

- k=1：对每对v和v'，除至此已知路径外（途经顶点下标≤0），从v到v'途经顶点下标不大于k（不大于1）的新路径可分为两段：
  - $<v, \ldots, v_1>$，$<v_1, \ldots, v'>$
  - 这两段内部所经过的顶点下标都不大于0，路径及其长度都已在前一步确定。这种新路径的长度是两段路径的长度之和。
  - 用这样的新路径与从 v 到 v' 的已知最短路径（途径顶点下标 ≤ 0）比较，就可确定 v 到 v' 的途经顶点下标 ≤ 1 的最短路径。

Floyd算法

- 一般而言，如果已考察过从 v 到 v' 的途经顶点的下标 $\leq k-1$ 的所有路径，而且已经获知了这样的路径中的最短路径及其长度。
- 考虑 k：对每对 v 和 v'，除至此已知的路径（途经顶点的下标 $\leq k-1$）外，途经顶点的下标 $\leq k$ 的其他路径必定可分为两段：
  - $<v, \ldots, v_k>$，$<v_k, \ldots, v'>$
  - 这两段路径中途径顶点的下标都 $\leq k-1$，两段的长度也在前一步已知，这种新路径的长度是两段路径的长度之和。
  - 用该路径与已知的从 v 到 v' 的最短路径比较，就可确定从 v 到 v' 的途经顶点的下标 $\leq k$ 的最短路径。

Floyd算法

- 如此继续直到做完 $k=n-1$（途径结点下标不大于$n-1$），也就对于每对 v 和 v'，确定了从 v 到 v' 的所有路径中的最短路径
- 实现 Floyd 算法，需要迭代式地算出一系列方阵
- 为此要生成一系列 $n\times n$ 方阵 A_k $(0 \le k \le n)$，其中 $A_k[i][j]$ 表示从 v_i 到 v_j 的途径顶点可为 $v_0, v_1, \ldots, v_{k-1}$ 的最短路径的长度：
  - A_0就是图的邻接矩阵A（由于是计算路径，对角线元素取 0 值），$A_0[i][j]$ 表示从 v_i 到 v_j 不经过任何顶点的最短路径长度
  - $A_n[i][j]$ 是从 v_i 到 v_j 的最短路径长度

Floyd算法

- 矩阵序列$A_0, A_1, \ldots, A_n$可递推计算（其中$0 \le i \le n-1, 0 \le j \le n-1$）：
  - $A_0[i][j] = A[i][j]$ 直接由邻接矩阵得到。
  - $A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k]+A_k[k][j]\}$，$0 \le k \le n-1$，这时新考虑了途径顶点 v_k 的路径，因此，$A_{k+1}[i][j]$ 为 v_i 到 v_j 的途经顶点的下标不大于 k 的最短路径的长度。
  - $A_n[i][j]$ 为 v_i 到 v_j 的最短路径的长度。
- 实现算法时还需要设法做出所有路径的记录：
  - 下面考虑另外安排一系列n阶方阵 nvertex_k，其中 $\text{nvertex}_k[v][v']$ 的值为在从 v 到 v' 的中间允许有顶点 $v_0, v_1, \ldots, v_{k-1}$ 的最短路径上，顶点 v 的后继顶点 v'' 的下标（与前面 A_k 对应）。
  - 到最后 v'' 到 v' 的最短路径也应已知，可以根据后继顶点追溯。

Floyd算法

- 初始时，若$A_0[i][j] = \infty$（没有边），则令$\text{nvertex}_0[i][j] = -1$，否则令$\text{nvertex}_0[i][j] = j$，表示 v_j 是 v_i 的后继顶点。
- 在由A_k计算A_{k+1}时，若$A_{k+1}[i][j]$被设置为$A_k[i][k] + A_k[k][j]$，则设 $\text{nvextex}_{k+1}[i][j] = \text{nvertex}_k[i][k]$（在从 v_i 到 v_j 的路径上 v_i 的后继顶点就是原来 v_i 到 v_k 的路径上 v_i 的后继顶点）。
- 这轮计算完成时，$\text{nvertex}_{k+1}[i][j]$ 就是在从 v_i 到 v_j 的可以途径顶点 $v_0, v_1, \ldots, v_k$ 的最短路径上，顶点 v_i 的后继顶点。
- 最后$\text{nvertex}_n[i][j]$ 就是从 v_i 到 v_j 的最短路径上 v_i 的后继结点。追溯这个矩阵，可得到任何一对结点之间的最短路径。

Floyd算法

- Floyd算法从$A_0=A$（图的邻接矩阵）开始。递推生成一系列矩阵A_1, A_2, …, A_n。后一矩阵可能与前一矩阵不同。

  问题：是否真需要用一个新的两维表存放下一个矩阵?

- 假设已经算出A_k保存在矩阵A里，考虑A_{k+1}的计算。公式是：

$$A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k]+A_k[k][j]\}$$

- 注意：所有新的$A_{k+1}[i][j]$或者为$A_k[i][j]$（如果它比较小），或者由下面的一列和一行中的元素求和计算出来：

  $A_k[0][k]$，$A_k[1][k]$，...，$A_k[n-1][k]$

  $A_k[k][0]$，$A_k[k][1]$，...，$A_k[k][n-1]$

- 注意：如果在计算 A_{k+1} 过程中可能修改矩阵第 k 行或第 k 列元素，在后面取元素 $[i][k]$ 或 $[k][j]$ 得到的将不是 A_k 的元素，就必须保留 A_k。

Floyd算法

- 实际上 A_{k+1} 计算中不会修改矩阵第 k 行或者第 k 列的元素，因为：

  $A_{k+1}[i][k] = \min\{A_k[i][k], A_k[i][k]+ A_k[k][k]\}$

  $A_{k+1}[k][j] = \min\{A_k[k][j], A_k[k][k]+ A_k[k][j]\}$

  而A_{k-1} 的对角线元素 $A_{k-1}[m, m]$ 总是 0（对所有m），所以

  $A_{k+1}[i][k] = A_k[i][k]$　　第 k 行不变，

  $A_{k+1}[k][j] = A_k[k][j]$　　第 k 列不变。

- 因此，计算中可用同一个 A 实现所有的 A_k，矩阵递推计算过程可通过直接更新 A 里元素实现。
  - 计算 $A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k]+ A_k[k][j]\}$，其中新矩阵元素的生成用赋值 $A[i][j] := A[i][k] + A[k][j]$ 实现。
  - 计算所有顶点对之间最短路径的长度，只需一个矩阵。

Floyd算法

```
void Floyd(GraphMatrix * pgraph, ShortPath * path) {
    int i, j, k;
    for (i = 0; i < VN; ++ i)
        for (j = 0; j < VN; ++ j) {
            if (pgraph->arcs[i][j] != MAX) path->nextvex[i][j] = j;
            else path->nextvex[i][j] = -1;
            path->a[i][j] = pgraph->arcs[i][j]; /* 复制邻接矩阵 */
        }
    for (k = 0; k < VN; ++ k)
        for (i = 0; i < VN; ++ i)
            for (j = 0; j < VN; ++ j) {









            }
}
```

Floyd算法

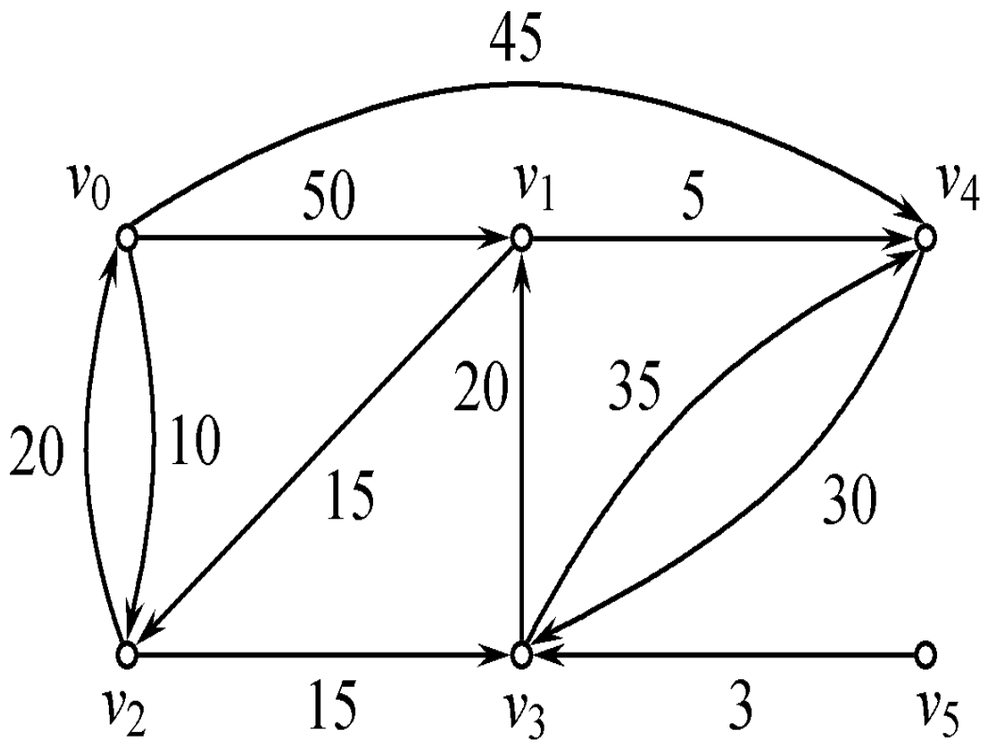
```
void Floyd(GraphMatrix * pgraph, ShortPath * path) {
    int i, j, k;
    for (i = 0; i < VN; ++ i)
        for (j = 0; j < VN; ++ j) {
            if (pgraph->arcs[i][j] != MAX) path->nextvex[i][j] = j;
            else path->nextvex[i][j] = -1;
            path->a[i][j] = pgraph->arcs[i][j]; /* 复制邻接矩阵 */
        }
    for (k = 0; k < VN; ++ k)
        for (i = 0; i < VN; ++ i)
            for (j = 0; j < VN; ++ j) {
                if ( path->a[i][k] == MAX || path->a[k][j] == MAX )
                    continue;
                if ( path->a[i][j] > path->a[i][k]+ path->a[k][j] ) {
                    path->a[i][j] = path->a[i][k] + path->a[k][j];
                    path->nextvex[i][j] = path->nextvex[i][k];
                }
            }
}
```

代价分析

- 算法中初始化部分由一个循环组成，其中外循环运行n次，内循环也运行n次，初始化部分的时间复杂度为$O(n^2)$。
- 迭代生成最短路径长度矩阵A和路径矩阵nextvex的部分为三重循环的嵌套，其时间复杂度为$O(n^3)$。因此，Floyd算法的时间复杂度为$O(n^3)$。
- 空间代价是$O(n^2)$（两个矩阵）。

例子

- **用Floyd方法求下图各顶点间的最短路径长度。**



$$\text{arcs} = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 20 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$

$$A_0 = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 20 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$

$$\text{nvertex}_0 = \begin{bmatrix} 0 & 1 & 2 & -1 & 4 & -1 \\ -1 & 1 & 2 & -1 & 4 & -1 \\ 0 & -1 & 2 & 3 & -1 & -1 \\ -1 & 1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & -1 & 5 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & \boxed{70} & 0 & 15 & \boxed{65} & \infty \\ \infty & 20 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$

$$\text{nvertex}_1 = \begin{bmatrix} 0 & 1 & 2 & -1 & 4 & -1 \\ -1 & 1 & 2 & -1 & 4 & -1 \\ 0 & \boxed{0} & 2 & 3 & \boxed{0} & -1 \\ -1 & 1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & -1 & 5 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & 70 & 0 & 15 & 65 & \infty \\ \infty & 20 & \boxed{35} & 0 & \boxed{25} & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$

$$\text{nvertex}_2 = \begin{bmatrix} 0 & 1 & 2 & -1 & 4 & -1 \\ -1 & 1 & 2 & -1 & 4 & -1 \\ 0 & 0 & 2 & 3 & 0 & -1 \\ -1 & 1 & \boxed{1} & 3 & \boxed{1} & -1 \\ -1 & -1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & -1 & 5 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 0 & 50 & 10 & 25 & 45 & \infty \\ 35 & 0 & 15 & 30 & 5 & \infty \\ 20 & 70 & 0 & 15 & 65 & \infty \\ 55 & 20 & 35 & 0 & 25 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$

$$\text{nvertex}_3 = \begin{bmatrix} 0 & 1 & 2 & 2 & 4 & -1 \\ 2 & 1 & 2 & 2 & 4 & -1 \\ 0 & 0 & 2 & 3 & 0 & -1 \\ 1 & 1 & 1 & 3 & 1 & -1 \\ -1 & -1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & -1 & 5 \end{bmatrix}$$

$$A_4 = \begin{bmatrix} 0 & 45 & 10 & 25 & 45 & \infty \\ 35 & 0 & 15 & 30 & 5 & \infty \\ 20 & 35 & 0 & 15 & 40 & \infty \\ 55 & 20 & 35 & 0 & 25 & \infty \\ 85 & 50 & 65 & 30 & 0 & \infty \\ 58 & 23 & 38 & 3 & 28 & 0 \end{bmatrix}$$

$$\text{nvertex}_4 = \begin{bmatrix} 0 & 2 & 2 & 2 & 4 & -1 \\ 2 & 1 & 2 & 2 & 4 & -1 \\ 0 & 3 & 2 & 3 & 3 & -1 \\ 1 & 1 & 1 & 3 & 1 & -1 \\ 3 & 3 & 3 & 3 & 4 & -1 \\ 3 & 3 & 3 & 3 & 3 & 5 \end{bmatrix}$$

例如，想知道v_0到v_1的最短路径：

- 由$A[0][1]$可知v_0到v_1的最短路径长度为45，

- 路径由 nvertex[0][1]=2 可知顶点 v_0 的下一顶点为v_2，
- 由nvertex[2][1]=3可知 v_2 的下一顶点为v_3，

- 由nvertex[3][1]=1可知 v_3 的下一顶点为v_1，

- 因此从v_0到v_1的最短路径为$v0 \rightarrow v2 \rightarrow v3 \rightarrow v1$。

本讲重点

- 图中从一个顶点到另一个顶点间路径长度最短的路径称为两个顶点间的最短路径。
- Dijkstra 算法基于类似于最小生成树的想法，或者说是一种“宽度优先搜索”。其中用到了类似 MST 的性质。
  - 它逐个找出可以确定最短路径的顶点，同时也找到了到新确定最短路径的顶点的路径。
  - 做完前一步后更新信息，保证记录的都是至今已知的最短路径。
  - 这是典型的动态规划方法（在计算中保留一些信息支持动态决策）。
- Floyd 算法基于完全不同的考虑，求解所有顶点间的最短路径。
  - 其基本方法是为最终问题的解决逐步积累信息，根据已有的信息更新包含部分信息的解的雏形，最终得到问题的解。
  - 这一算法也是一个典型的动态规划方法。
  - 过程中求子结构（子问题）的最优解，最后得到原问题的最优解。