**最短路径算法总结**

几个最短路径算法Floyd、Dijkstra、Bellman-Ford、SPFA的比较

<http://blog.csdn.net/v_july_v/article/details/6181485>

其中，Dijkstra、Bellman-Ford属于单源最短路径算法

**一、Floyd算法**

<http://www.cnblogs.com/biyeymyhjob/archive/2012/07/31/2615833.html>

<http://blog.csdn.net/qq_35644234/article/details/60875818>

**1、定义概述**

Floyd-Warshall算法（Floyd-Warshall algorithm）是*解决任意两点间的最短路径的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包*。Floyd-Warshall算法的时间复杂度为O(N3)，空间复杂度为O(N2)。

**2、算法描述**

* **算法思想原理：**Floyd算法是一个经典的动态规划算法。用通俗的语言来描述的话，首先我们的目标是寻找**从点i到点j的最短路径**。从动态规划的角度看问题，我们需要为这个目标重新做一个诠释（这个诠释正是动态规划最富创造力的精华所在）

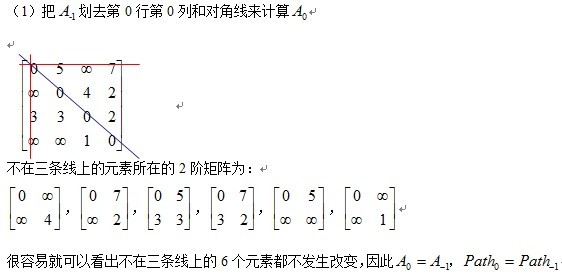
从**任意节点i到任意节点j的最短路径不外乎2种可能**，*1是直接从i到j，2是从i经过若干个节点k到j*。所以，我们假设Dis(i,j)为节点u到节点v的最短路径的距离，对于每一个节点k，我们检查**Dis(i,k) + Dis(k,j) < Dis(i,j)**是否成立，如果成立，证明从i到k再到j的路径比i直接到j的路径短，我们便设置**Dis(i,j) = Dis(i,k) + Dis(k,j)**，这样一来，当我们遍历完所有节点k，Dis(i,j)中记录的便是i到j的最短路径的距离

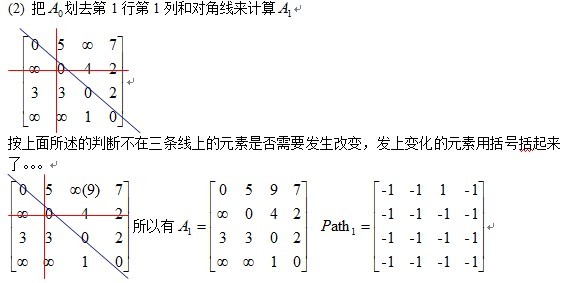
* **算法描述：**
* a. 从任意一条单边路径开始。**所有两点之间的距离是边的权**，如果两点之间没有边相连，则权为无穷大。
* b.对于**每一对顶点 u 和 v**，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比己知的路径更短。如果是更新它。
* Floyd算法过程矩阵的计算----**十字交叉法**

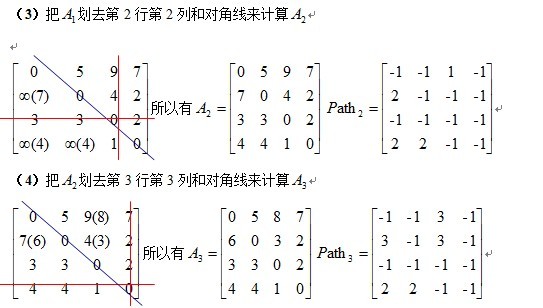
**方法：**两条线，从*左上角开始计算一直到右下角*，如下所示给出矩阵，其中矩阵A是邻接矩阵，而***矩阵Path记录u,v两点之间最短路径所必须经过的点***



相应计算方法如下：

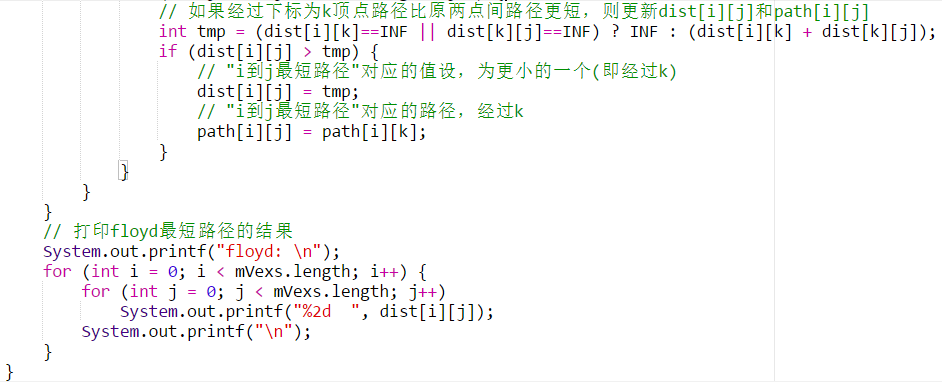






**3、算法代码实现**





**4、算法实例**

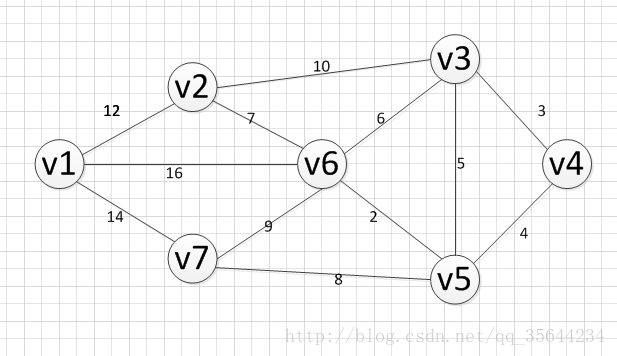
**1）实例1**

**算法的思路：**

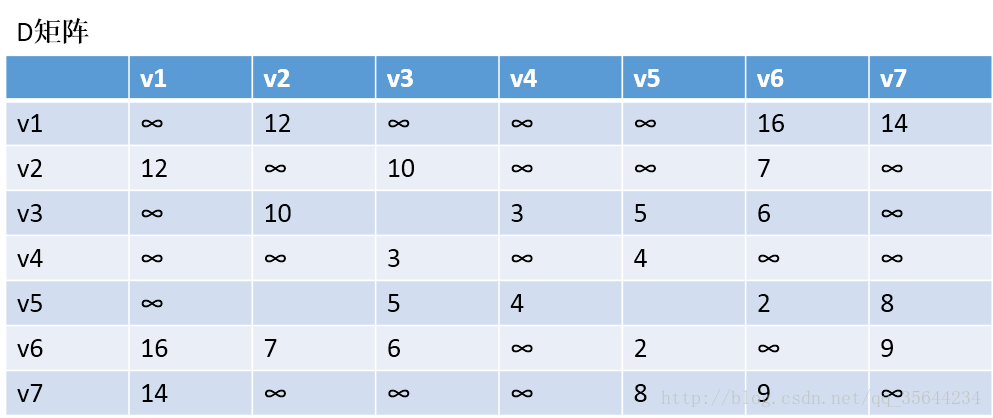
通过Floyd计算图G=(V,E)中**各个顶点的最短路径**时，需要引入两个矩阵，**矩阵S**中的元素*a[i][j]表示顶点i(第i个顶点)到顶点j(第j个顶点)的距离*。**矩阵P**中的元素*b[i][j]表示顶点i到顶点j经过了b[i][j]记录的值所表示的顶点*。

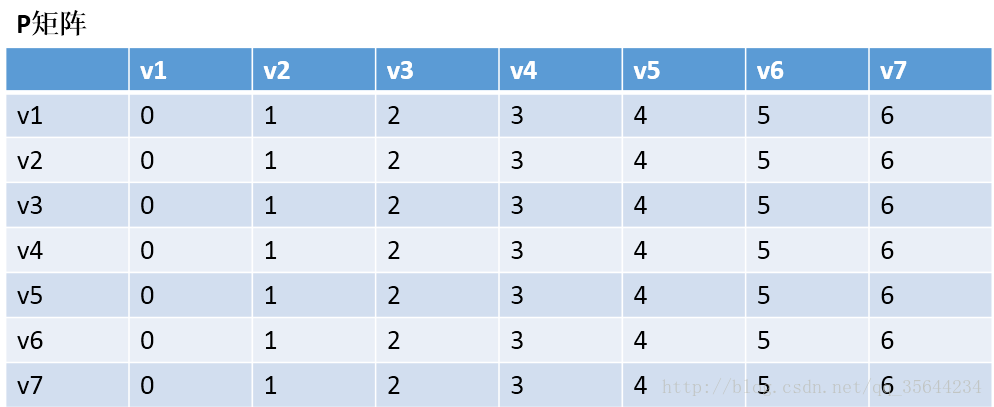
假设图G中顶点个数为N，则需要对矩阵D和矩阵P进行**N次更新**。初始时，矩阵D中顶点a[i][j]的距离为顶点i到顶点j的权值；如果i和j不相邻，则a[i][j]=∞，矩阵P的值为顶点b[i][j]的j的值。 接下来开始，**对矩阵D进行N次更新**。第1次更新时，如果”a[i][j]的距离” > “a[i][0]+a[0][j]”(a[i][0]+a[0][j]表示”i与j之间经过第1个顶点的距离”)，则更新a[i][j]为”a[i][0]+a[0][j]”,更新b[i][j]=b[i][0]。 同理，第k次更新时，如果”a[i][j]的距离” > “a[i][k-1]+a[k-1][j]”，则更新a[i][j]为”a[i][k-1]+a[k-1][j]”,b[i][j]=b[i][k-1]。更新N次之后，操作完成！

如下图，求下图的**每个点对之间的最短路径的过程如下：**

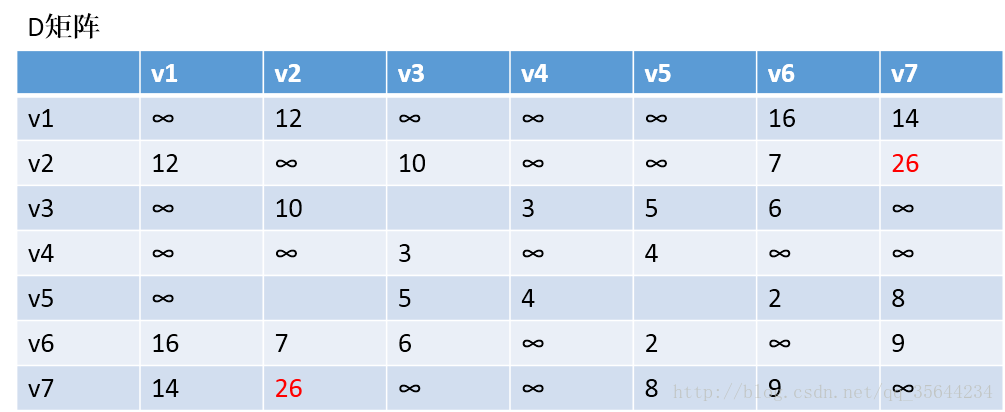


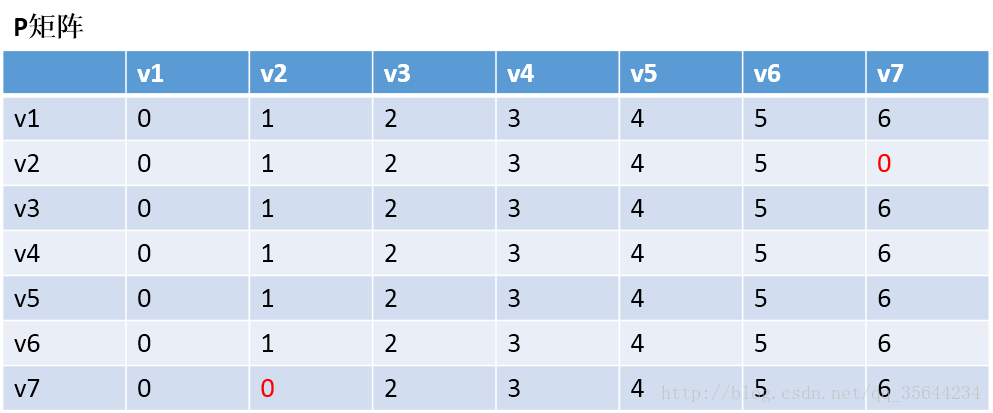
* 第一步，我们先初始化两个矩阵，得到下图两个矩阵：





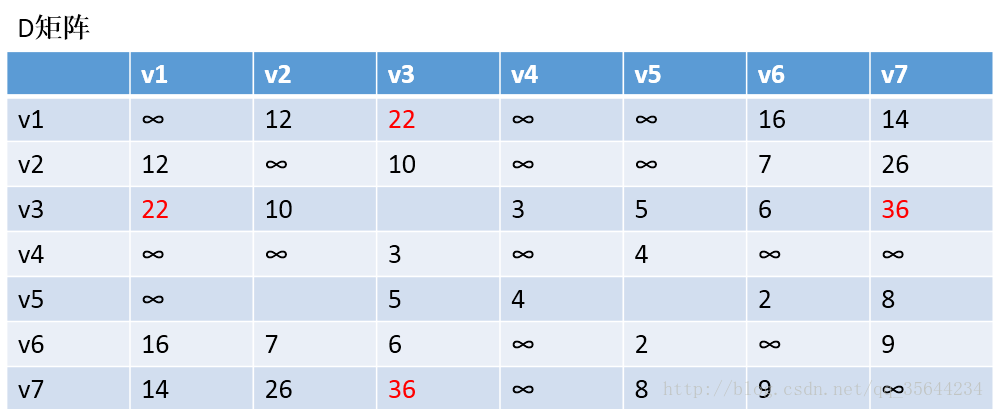
* 第二步，以v1为中阶，更新两个矩阵： 发现，a[1][0]+a[0][6] < a[1][6] 和a[6][0]+a[0][1] < a[6][1]，所以我们只需要矩阵D和矩阵P，结果如下：

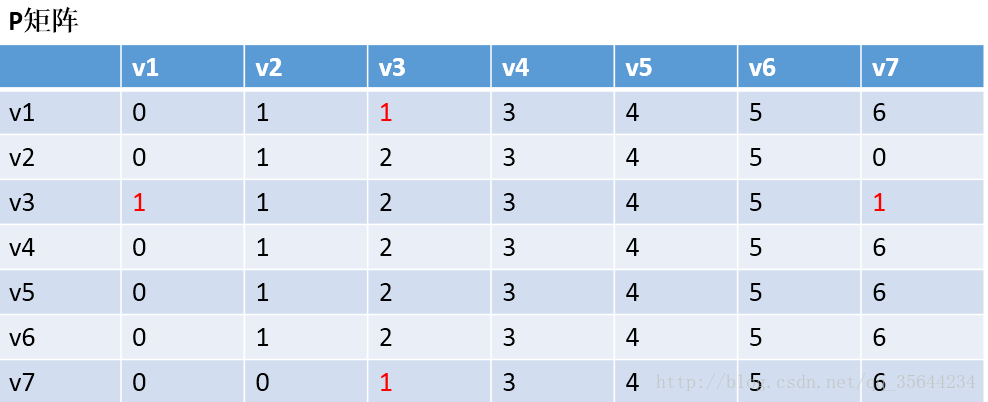




通过矩阵P，我发现v2–v7的最短路径是：v2–v1–v7

* 第三步：以v2作为中介，来更新我们的两个矩阵，使用同样的原理，扫描整个矩阵，得到如下图的结果：

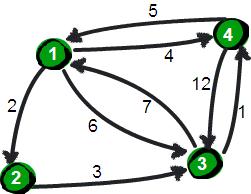




OK，到这里我们也就应该明白Floyd算法是如何工作的了，他**每次都会选择一个中介点**，**然后，遍历整个矩阵，查找需要更新的值**，下面还剩下五步，就不继续演示下去了，理解了方法，我们就可以写代码了。

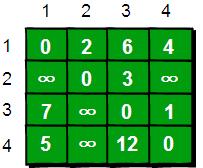
**2）实例2**

<http://developer.51cto.com/art/201403/433874.htm>



上图中有4个城市8条公路，公路上的数字表示这条公路的长短。请注意这些公路是单向的。我们现在需要求**任意两个城市之间的最短路程**，也就是求任意两个点之间的最短路径。这个问题这也被称为**“多源最短路径”问题**。

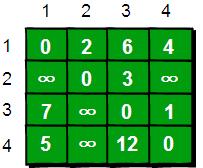
现在需要一个数据结构来存储图的信息，我们仍然可以用一个4\*4的矩阵（二维数组e）来存储。比如1号城市到2号城市的路程为2，则设e[1][2]的值为2。2号城市无法到达4号城市，则设置e[2][4]的值为∞。另外此处约定一个城市自己是到自己的也是0，例如e[1][1]为0，具体如下。



现在回到问题：如何求任意两点之间最短路径呢？通过之前的学习我们知道通过**深度或广度优先搜索可以求出两点之间的最短路径**。所以进行**n2遍深度或广度优先搜索**，即对每两个点都进行一次深度或广度优先搜索，便可以求得任意两点之间的最短路径。可是还有没有别的方法呢？

我们来想一想，根据我们以往的经验，如果要让任意两点（例如从顶点a点到顶点b）之间的路程变短，只能引入第三个点（顶点k），并通过这个顶点k中转即a->k->b，才可能**缩短原来从顶点a点到顶点b的路程**。那么这个中转的顶点k是1~n中的哪个点呢？甚至有时候不只通过一个点，而是经过两个点或者更多点中转会更短，即a->k1->k2-> b或者a->k1->k2…->k->i…->b。比如上图中从4号城市到3号城市（4->3）的路程e[4][3]原本是12。如果只通过1号城市中转（4->1->3），路程将缩短为11（e[4][1]+e[1][3]=5+6=11）。其实1号城市到3号城市也可以通过2号城市中转，使得1号到3号城市的路程缩短为5（e[1][2]+e[2][3]=2+3=5）。所以如果同时经过1号和2号两个城市中转的话，从4号城市到3号城市的路程会进一步缩短为10。通过这个的例子，我们发现**每个顶点都有可能使得另外两个顶点之间的路程变短**。好，下面我们将这个问题一般化。

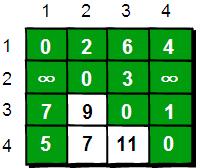
当任意两点之间不允许经过第三个点时，这些城市之间最短路程就是初始路程，如下。



如现在**只允许经过1号顶点，求任意两点之间的最短路程**，应该如何求呢？*只需判断e[i][1]+e[1][j]是否比e[i][j]要小即可*。e[i][j]表示的是从i号顶点到j号顶点之间的路程。e[i][1]+e[1][j]表示的是从i号顶点先到1号顶点，再从1号顶点到j号顶点的路程之和。其中i是1~n循环，j也是1~n循环，代码实现如下。



在只允许经过1号顶点的情况下，任意两点之间的最短路程更新为：

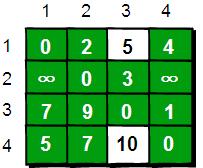


通过上图我们发现：在只通过1号顶点中转的情况下，3号顶点到2号顶点（e[3][2]）、4号顶点到2号顶点（e[4][2]）以及4号顶点到3号顶点（e[4][3]）的路程都变短了。

接下来继续求在只允许经过1和2号两个顶点的情况下任意两点之间的最短路程。如何做呢？我们需要在只允许经过1号顶点时任意两点的最短路程的**结果下**，再判断如果经过2号顶点是否可以使得i号顶点到j号顶点之间的路程变得更短。即判断e[i][2]+e[2][j]是否比e[i][j]要小，代码实现为如下。

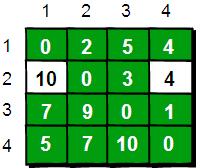


在只允许经过1和2号顶点的情况下，任意两点之间的最短路程更新为：

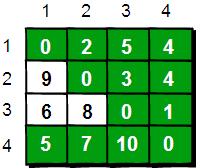


通过上图得知，在相比只允许通过1号顶点进行中转的情况下，这里允许通过1和2号顶点进行中转，使得e[1][3]和e[4][3]的路程变得更短了。

同理，继续在只允许经过1、2和3号顶点进行中转的情况下，求任意两点之间的最短路程。任意两点之间的最短路程更新为：



最后允许通过所有顶点作为中转，任意两点之间最终的最短路程为：

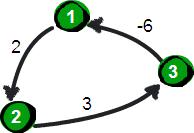


整个算法过程虽然说起来很麻烦，但是代码实现却非常简单，核心代码只有五行：



这段代码的**基本思想就是：**最开始只允许经过1号顶点进行中转，接下来只允许经过1和2号顶点进行中转……*允许经过1~n号所有顶点进行中转，求任意两点之间的最短路程*。用一句话概括就是：从i号顶点到j号顶点只经过前k号点的最短路程。其实这是一种“动态规划”的思想。

另外需要注意的是：**Floyd-Warshall算法不能解决带有“负权回路”（或者叫“负权环”）**的图，因为带有**“负权回路”的图**没有最短路。例如下面这个图就不存在1号顶点到3号顶点的最短路径。因为1->2->3->1->2->3->…->1->2->3这样路径中，每绕一次1->-2>3这样的环，**最短路就会减少1，永远找不到最短路**。其实如果一个图中带有“负权回路”那么这个图则没有最短路。



**3）实例3**

<http://www.cnblogs.com/skywang12345/p/3711532.html>

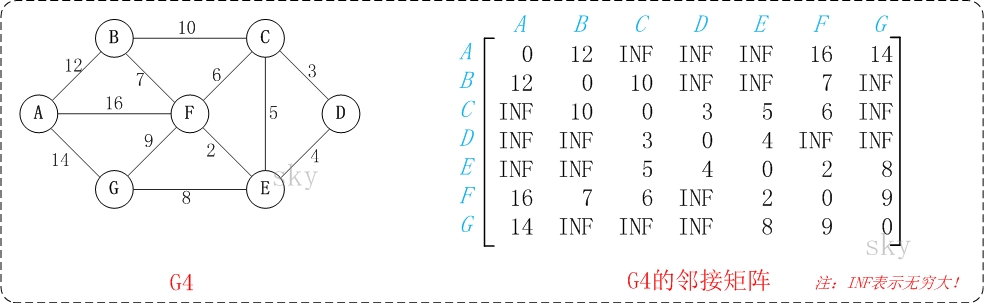
基本思想：

通过Floyd计算图G=(V,E)中各个顶点的最短路径时，需要引入一个矩阵S，矩阵S中的元素a[i][j]表示顶点i(第i个顶点)到顶点j(第j个顶点)的距离。

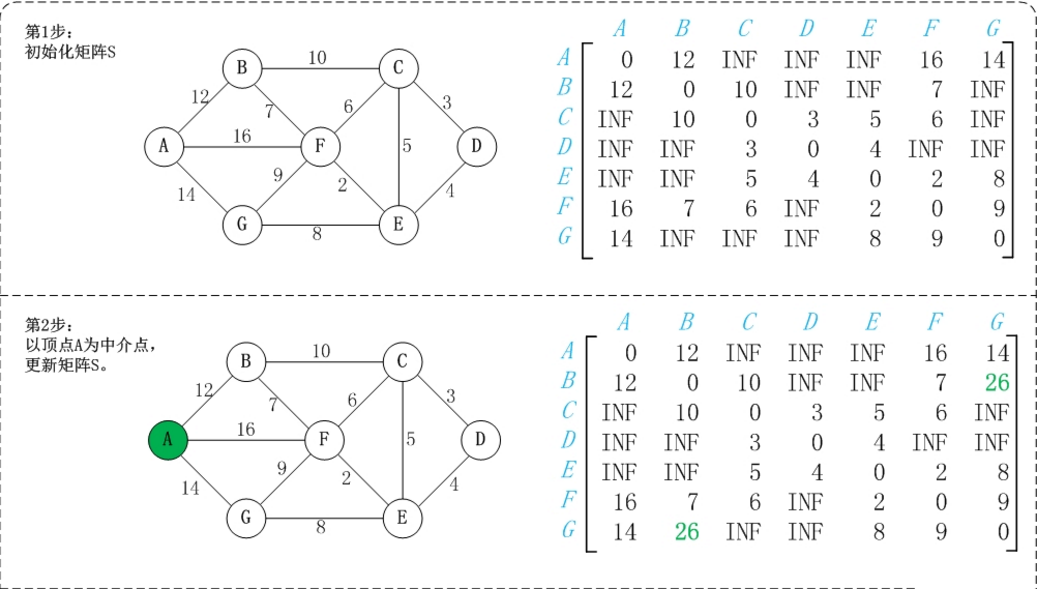
假设图G中顶点个数为N，则需要对矩阵S进行N次更新。初始时，矩阵S中顶点a[i][j]的距离为顶点i到顶点j的权值；如果i和j不相邻，则a[i][j]=∞。 接下来开始，对矩阵S进行N次更新。第1次更新时，如果"a[i][j]的距离" > "a[i][0]+a[0][j]"(a[i][0]+a[0][j]表示"i与j之间经过第1个顶点的距离")，则更新a[i][j]为"a[i][0]+a[0][j]"。 同理，第k次更新时，如果"a[i][j]的距离" > "a[i][k]+a[k][j]"，则更新a[i][j]为"a[i][k]+a[k][j]"。更新N次之后，操作完成！

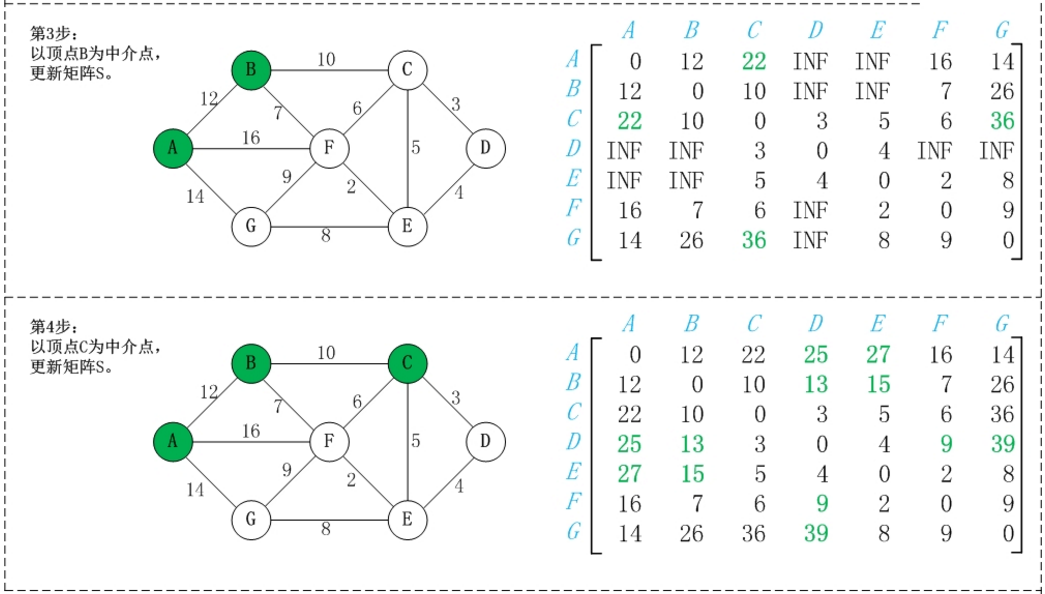
单纯的看上面的理论可能比较难以理解，下面通过实例来对该算法进行说明。

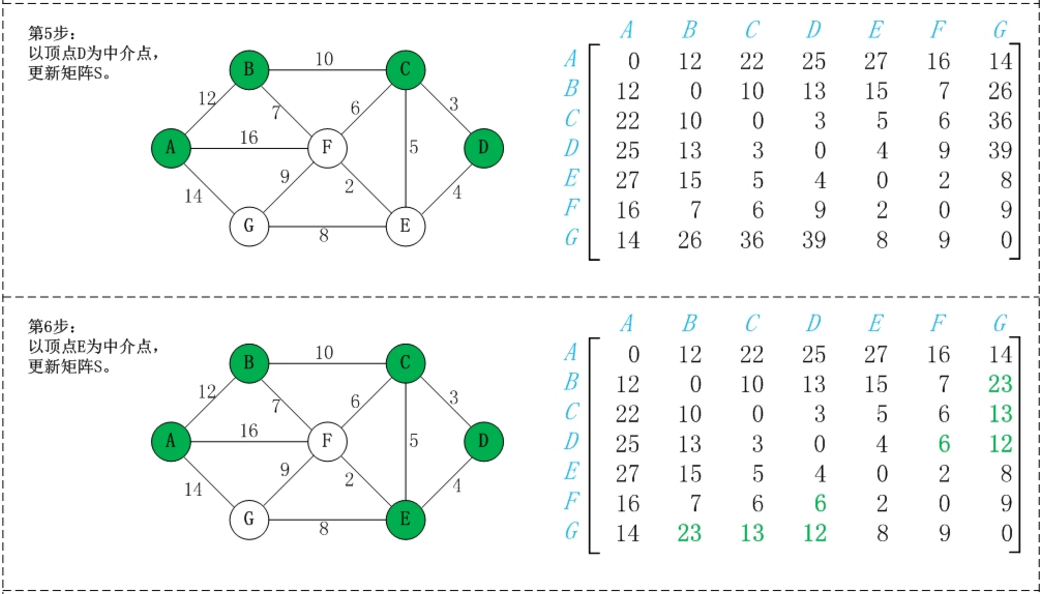
图解：

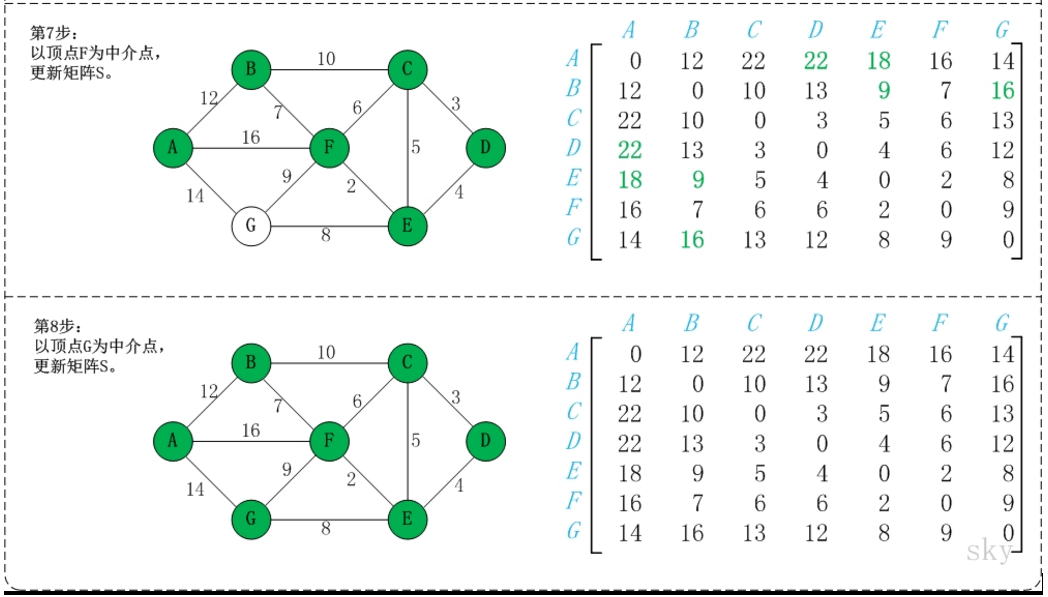


以上图G4为例，来对弗洛伊德进行算法演示。**（十字交叉法分析）**







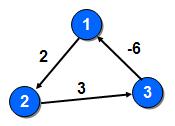


* **初始状态：**S是记录各个顶点间最短路径的矩阵。
* **第1步：**初始化S。矩阵S中顶点a[i][j]的距离为顶点i到顶点j的权值；如果i和j不相邻，则a[i][j]=∞。实际上，就是将图的原始矩阵复制到S中。

注：a[i][j]表示矩阵S中顶点i(第i个顶点)到顶点j(第j个顶点)的距离。

* **第2步：**以**顶点A(第1个顶点)为中介点**，若a[i][j] > a[i][0]+a[0][j]，则设置a[i][j]=a[i][0]+a[0][j]。 以顶点a[1]6，上一步操作之后，a[1][6]=∞；而将**A作为中介点**时，(B,A)=12，(A,G)=14，因此B和G之间的距离可以更新为26。
* 同理，依次将顶点B,C,D,E,F,G作为中介点，并更新a[i][j]的大小。

另外需要注意的是，**Floyd算法不能解决带有“负权回路”(或者叫“负权环”)的图**，因为带有“负权回路”的图没有最短路径。例如下面的图就不存在1号顶点到3号顶点的最短路径，因为1->2->3->1->2->3->……1->2->3这样路径中，每绕一次1->2->3这样的环，最短路径就会减少1，永远找不到最短路径。其实如果一个图中带有“负权回路”，那么这个图则没有最短路径。



**二、Dijkstra算法**

<http://www.cnblogs.com/biyeymyhjob/archive/2012/07/31/2615833.html>

<http://blog.csdn.net/qq_35644234/article/details/60870719>

Dijkstra算法解决的是带权重的有向图上单源最短路径问题，该算法要求所有边的权重都为非负值。

**1、定义概述**

Dijkstra(迪杰斯特拉)算法是典型的**单源最短路径算法**，用于*计算一个节点到其他所有节点的最短路径*。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra算法是很有代表性的最短路径算法，在很多专业课程中都作为基本内容有详细的介绍，如数据结构，图论，运筹学等等。注意该算法要求图中不存在负权边。

Dijkstra算法使用了*广度优先搜索解决赋权有向图或者无向图的单源最短路径问题*，算法最终得到一个**最短路径树**。该算法常用于路由算法或者作为其他图算法的一个子模块。

**问题描述：**在无向图 G=(V,E) 中，假设**每条边 E[i] 的长度为 w[i]**，找到由顶点 V0 到其余各点的最短路径。（单源最短路径）

**2、算法描述**

* **算法思想：**设G=(V,E)是一个带权有向图，把图中**顶点集合V分成两组**，*第一组*为已求出最短路径的顶点集合（用S表示，初始时S中只有一个源点，以后每求得一条**最短路径**, 就将加入到集合S中，直到**全部顶点**都加入到S中，算法就结束了），*第二组*为其余未确定最短路径的顶点集合（用U表示），按**最短路径长度的递增**次序依次把第二组的顶点加入S中。在加入的过程中，*总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度*。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。
* 算法步骤：
* a. 初始时，S只包含源点，即S＝{v}，v的距离为0。U包含除v外的其他顶点，即:U={其余顶点}，若v与U中顶点u有边，则**<u,v>正常有权值**，若u不是v的出边邻接点，则<u,v>权值为∞。
* b. 从U中选取一个距离v最小的顶点k，把k，加入S中（该**选定的距离**就是v到k的最短路径长度）。
* c. 以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则**修改顶点u的距离值**，修改后的距离值为顶点k的距离加上边上的权。
* d. 重复步骤b和c直到所有顶点都包含在S中。

**3、算法代码实现**

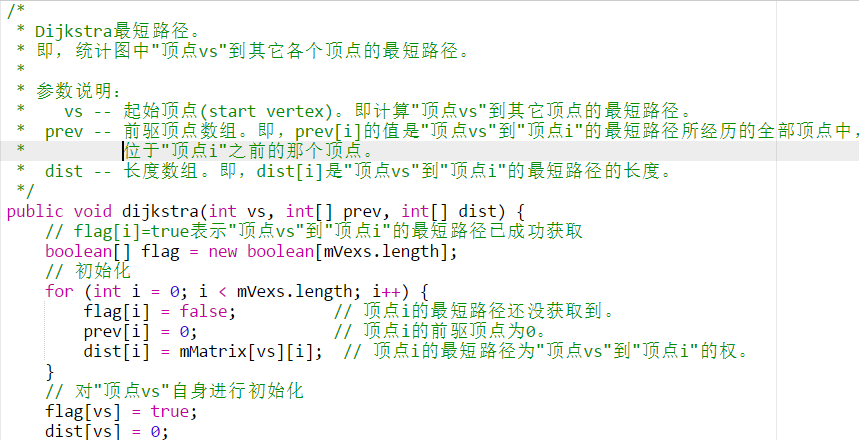
**1）伪代码（算法导论）**

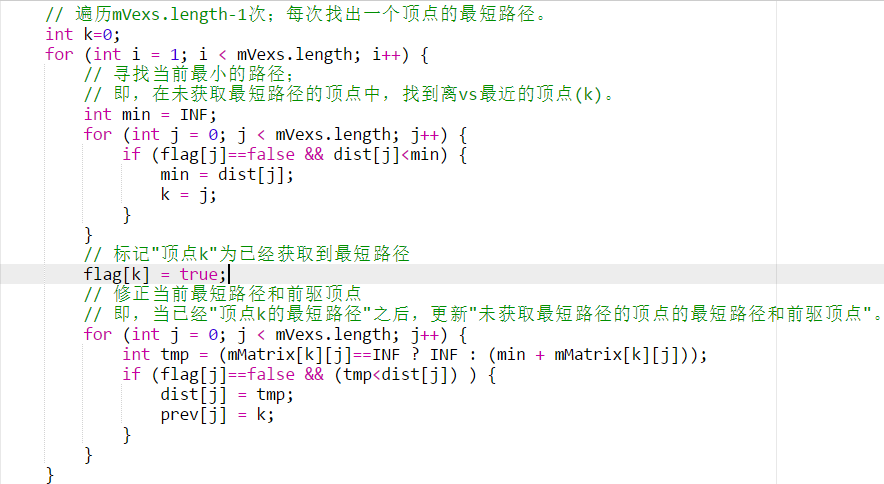
Dijkstra算法在运行过程中维持的关键信息是一组节点集合S。从源节点s到该集合中每个节点之间的最短路径已经被找到。算法重复从节点机**V-S**中选择最短路径估计最小的节点u，将u加入到集合S，然后对所有从u发出的边进行松弛。在下面给出的实现方式中，使用一个**最小优先队列Q来保存节点集合**，每个节点的关键值为其d值。

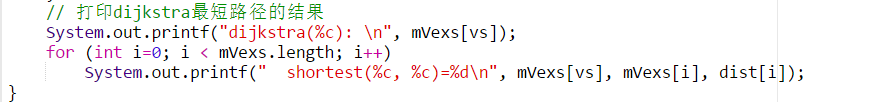


<http://www.cnblogs.com/skywang12345/p/3711516.html>

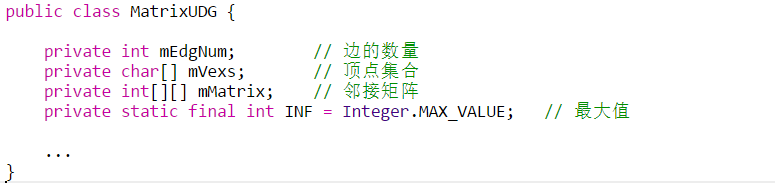
**2）邻接矩阵**





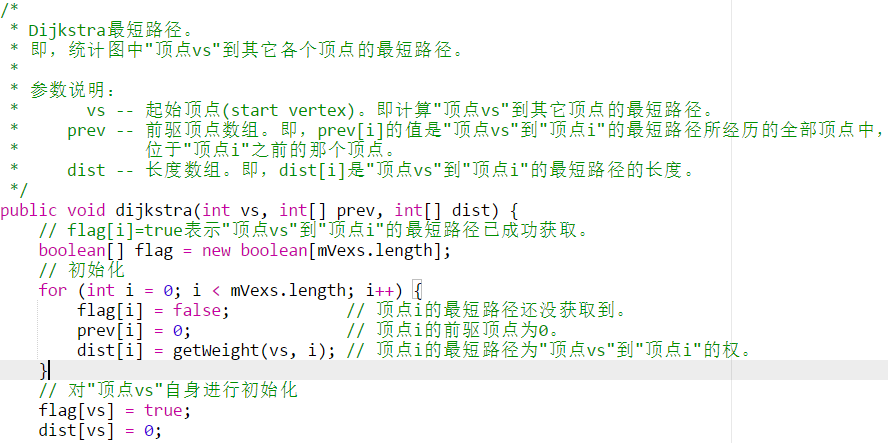


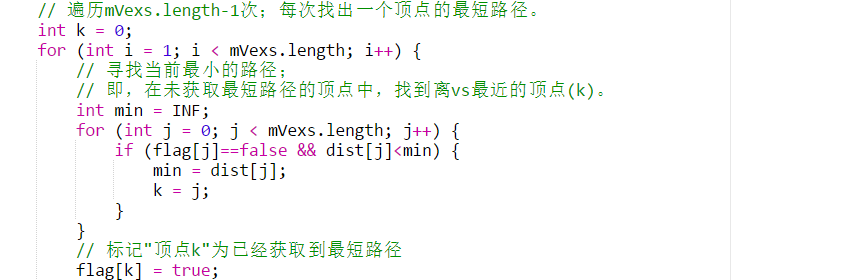
基本定义：

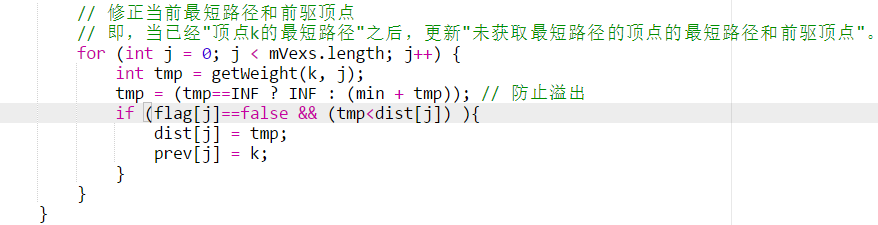


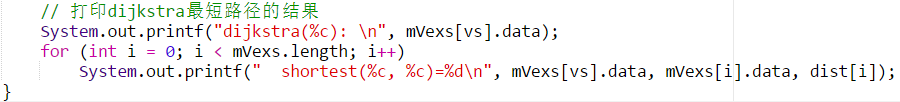
MatrixUDG是**邻接矩阵对应的结构体**。mVexs用于保存顶点，mEdgNum用于**保存边数**，mMatrix则是用于**保存矩阵信息的二维数组**。例如，mMatrix[i][j]=1，则表示"顶点i(即mVexs[i])"和"顶点j(即mVexs[j])"是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

**3）邻接表**

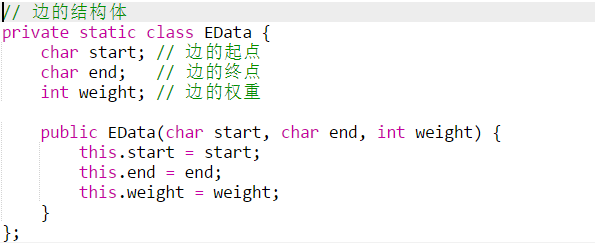


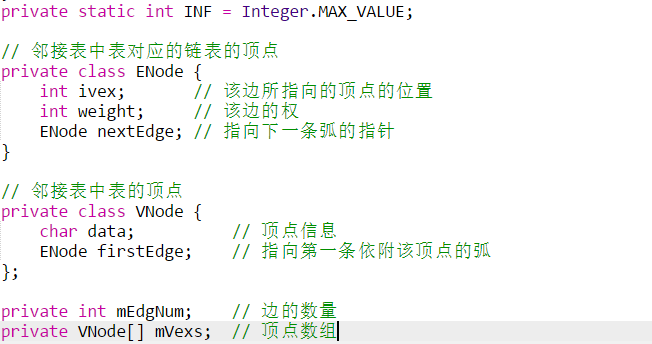






**数据结构构成：**

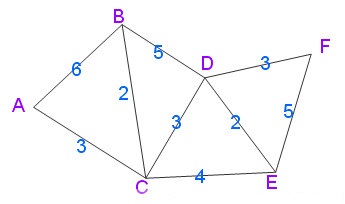




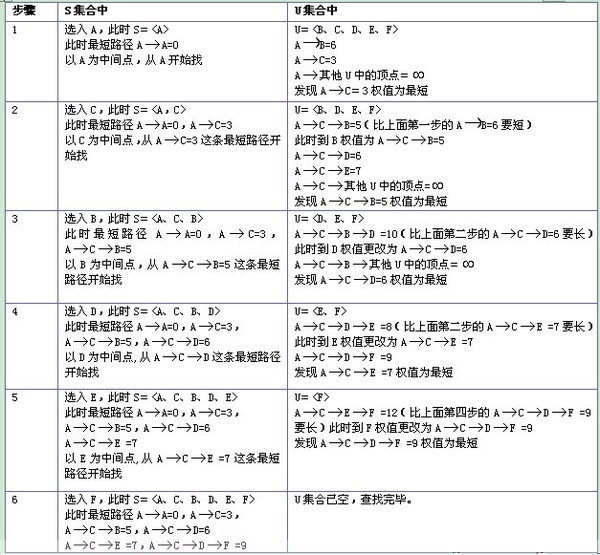
**4、算法实例**

**1）实例1（无向图）**

先给出一个无向图



用Dijkstra算法找出以A为起点的单源最短路径步骤如下：

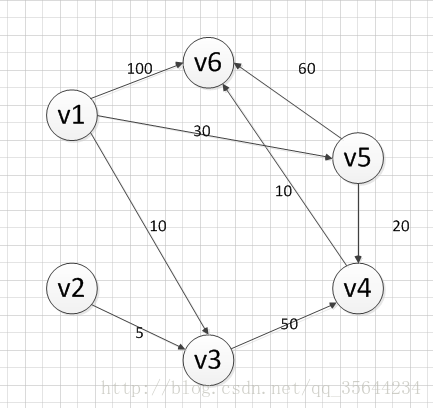


**2）实例2（有向图）**

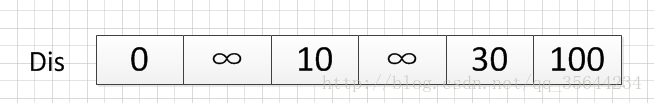
算法的思路：

* Dijkstra算法采用的是一种**贪心的策略**，声明一个*数组dis来保存源点到各个顶点的最短距离和一个保存已经找到了最短路径的顶点的集合*：T，初始时，原点 s 的路径权重被赋为 0 （dis[s] = 0）。若对于顶点 s 存在能直接到达的边（s,m），则把dis[m]设为w（s, m）,同时把所有其他（s不能直接到达的）顶点的路径长度设为无穷大。初始时，集合T只有顶点s。
* 然后，从dis数组选择最小值，则该值就是源点s到该值对应的顶点的**最短路径**，并且把该点加入到T中，OK，此时完成一个顶点，
* 然后，我们需要看看**新加入的顶点**是否可以到达其他顶点并且看看通过该顶点到达其他点的路径长度是否比源点直接到达短，如果是，那么就***替换这些顶点在dis中的值***。
* 然后，又从dis中找出最小值，重复上述动作，直到T中包含了图的所有顶点。

下面我求下图，从顶点v1到其他各个顶点的最短路径



首先第一步，我们先声明一个dis数组，该数组初始化的值为：

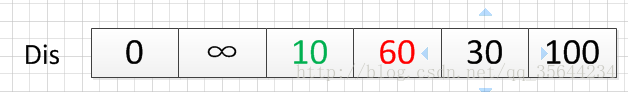


我们的**顶点集T**的初始化为：T={v1}

既然是求 **v1顶点到其余各个顶点的最短路程**，那就先找一个离 1 号顶点最近的顶点。通过数组 dis 可知当前离v1顶点最近是 **v3顶点**。当选择了 2 号顶点后，dis[2]（下标从0开始）的值就已经从**“估计值”变为了“确定值”**，即 v1顶点到 v3顶点的最短路程就是当前 dis[2]值。将V3加入到T中。

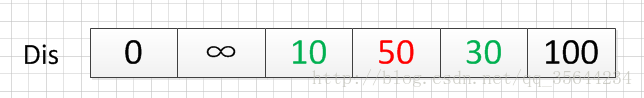
为什么呢？因为目前离 v1顶点最近的是 v3顶点，并且这个图所有的边都是**正数**，那么肯定不可能通过第三个顶点中转，使得 v1顶点到 v3顶点的路程进一步缩短了。因为 v1顶点到其它顶点的路程肯定没有 v1到 v3顶点短.

OK，既然确定了一个顶点的最短路径，下面我们就要根据这个新入的顶点V3会有出度，发现以**v3 为弧尾的有： < v3,v4 >,**那么我们看看路径：v1–v3–v4的长度是否比v1–v4短，其实这个已经是很明显的了，因为dis[3]代表的就是v1–v4的长度为无穷大，而v1–v3–v4的长度为：10+50=60，所以更新dis[3]的值,得到如下结果：

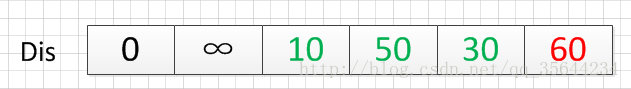


因此 dis[3]要更新为 60。这个过程有个专业术语叫做**“松弛”**。即 v1顶点到 v4顶点的路程即 dis[3]，通过 < v3,v4> 这条边松弛成功。这便是 **Dijkstra 算法的主要思想**：通过“边”来松弛v1顶点到其余各个顶点的路程。

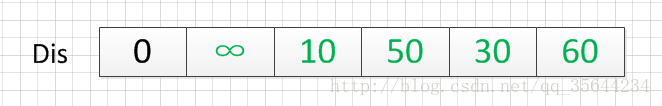
然后，我们又从**除dis[2]和dis[0]外的其他值中寻找最小值**，发现dis[4]的值最小，通过之前是解释的原理，可以知道v1到v5的最短距离就是dis[4]的值，然后，我们把v5加入到集合T中，然后，**考虑v5的出度是否会影响我们的数组dis的值**，v5有一条出度：< v4,v6>,然后我们发现：v1–v5–v4的长度为：50，而dis[3]的值为60，所以我们要更新dis[3]的值，更新后的dis数组如下图：



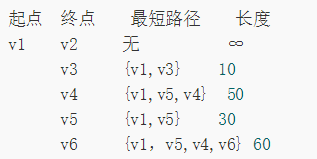
然后，继续从dis中选择**未确定的顶点的值中选择一个最小的值**，发现dis[3]的值是最小的，所以把v4加入到集合T中，此时集合**T={v1,v3,v5,v4}**,然后，考虑v4的出度是否会影响我们的数组dis的值，v4有一条出度：< v5,v4>,然后我们发现：v1–v5–v4–v6的长度为：60，而dis[5]的值为100，所以我们要更新dis[5]的值，更新后的dis数组如下图：



然后，我们使用同样原理，分别确定了v6和v2的最短路径，最后dis的数组的值如下：



因此，从图中，我们可以发现v1-v2的值为：∞，代表没有路径从v1到达v2。所以我们得到的最后的结果为：



**3）实例3（有向图）**

<http://www.cnblogs.com/skywang12345/p/3711516.html>

* **基本思想：**

通过Dijkstra计算图G中的最短路径时，需要指定起点s(即从顶点s开始计算)。

此外，引进两个集合S和U。S的作用是*记录已求出最短路径的顶点(以及相应的最短路径长度)，而U则是记录还未求出最短路径的顶点(以及该顶点到起点s的距离)*。

初始时，S中只有起点s；U中是除s之外的顶点，并且U中顶点的路径是"起点s到该顶点的路径"。然后，从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。 然后，再从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。 ... 重复该操作，直到遍历完所有顶点。

* **操作步骤：**

(1) 初始时，S只包含起点s；U包含除s外的其他顶点，且U中顶点的距离为"起点s到该顶点的距离"[例如，U中顶点v的距离为(s,v)的长度，然后s和v不相邻，则v的距离为∞]。

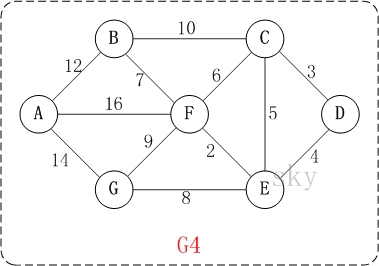
(2) 从U中选出"距离最短的顶点k"，并将顶点k加入到S中；同时，从U中移除顶点k。

(3) 更新U中各个顶点到起点s的距离。之所以更新U中顶点的距离，是由于上一步中确定了k是**求出最短路径的顶点**，从而可以**利用k来更新其它顶点的距离**；例如，(s,v)的距离可能大于(s,k)+(k,v)的距离。

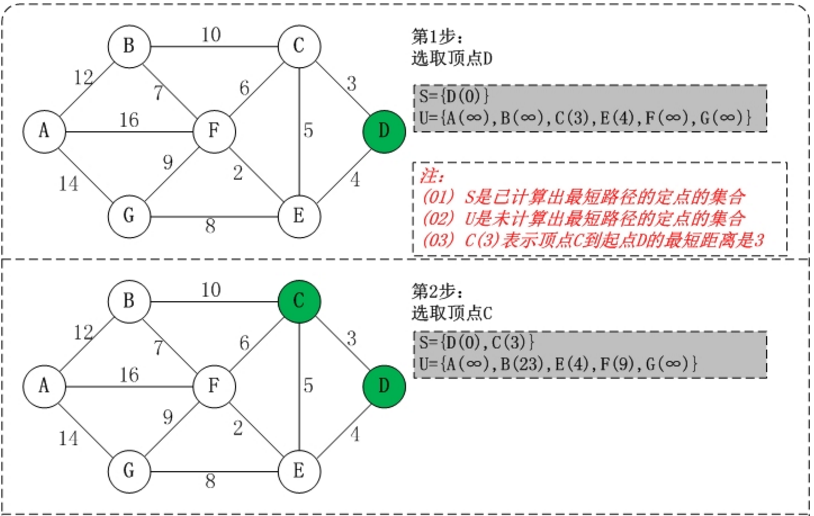
(4) 重复步骤(2)和(3)，直到遍历完所有顶点。

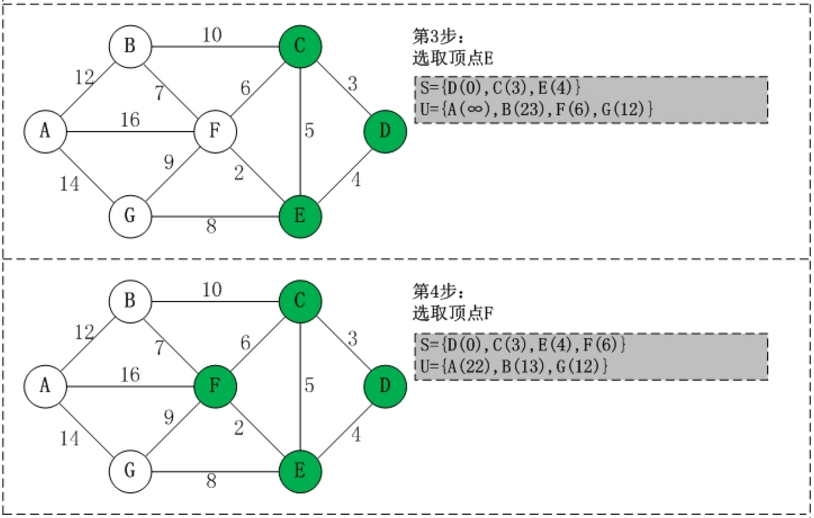
单纯的看上面的理论可能比较难以理解，下面通过实例来对该算法进行说明。

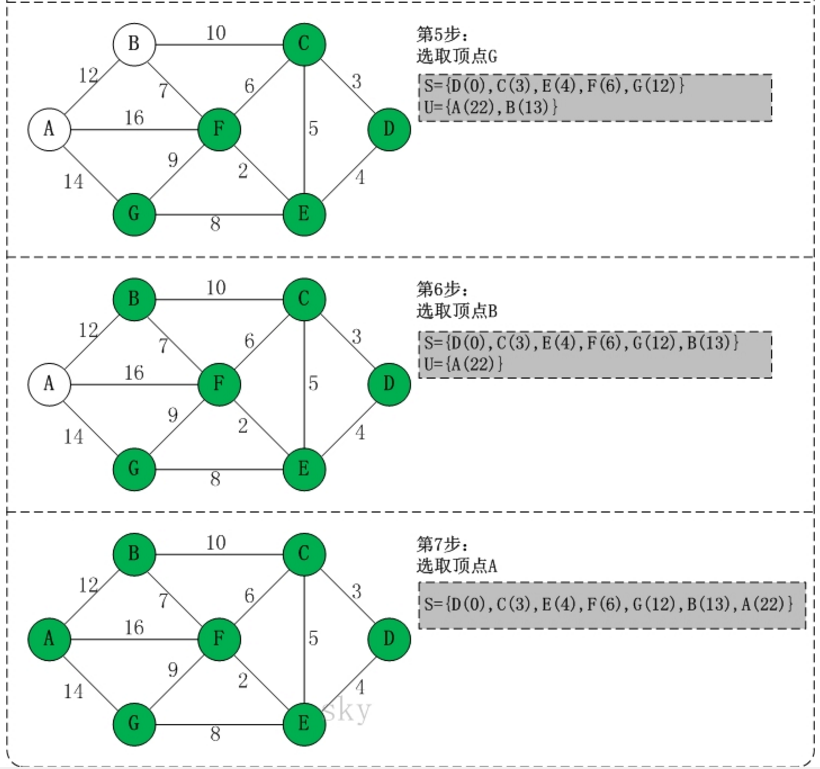
Dijkstra算法图解：



以上图G4为例，来对Dijkstra算法演示(以第4个顶点D为起点)。**注：BC边权重改为20**







* **初始状态：**S是已计算出最短路径的顶点集合，U是未计算除最短路径的顶点的集合！
* **第1步：**将顶点D加入到S中。此时，S={D(0)}, U={A(∞),B(∞),C(3),E(4),F(∞),G(∞)}。 注：C(3)表示C到起点D的距离是3。
* **第2步：**将顶点C加入到S中。上一步操作之后，U中顶点C到起点D的距离最短；因此，将C加入到S中，同时更新U中顶点的距离。以顶点F为例，之前F到D的距离为∞；但是将C加入到S之后，F到D的距离为9=(F,C)+(C,D)。 此时，S={D(0),C(3)}, U={A(∞),B(23),E(4),F(9),G(∞)}。
* **第3步：**将顶点E加入到S中。上一步操作之后，U中顶点E到起点D的距离最短；因此，将E加入到S中，同时更新U中顶点的距离。还是以**顶点F为例**，之前F到D的距离为9；但是将E加入到S之后，F到D的距离为6=(F,E)+(E,D)。 此时，S={D(0),C(3),E(4)}, U={A(∞),B(23),F(6),G(12)}。
* **第4步：**将顶点F加入到S中。此时，S={D(0),C(3),E(4),F(6)}, U={A(22),B(13),G(12)}。
* **第5步：**将顶点G加入到S中。此时，S={D(0),C(3),E(4),F(6),G(12)}, U={A(22),B(13)}。
* **第6步：**将顶点B加入到S中。此时，S={D(0),C(3),E(4),F(6),G(12),B(13)}, U={A(22)}。
* **第7步：**将顶点A加入到S中。此时，S={D(0),C(3),E(4),F(6),G(12),B(13),A(22)}。
* 此时，起点D到各个顶点的最短距离就计算出来了：A(22) B(13) C(3) D(0) E(4) F(6) G(12)。

**三、Bellman-Ford算法**

<http://www.cnblogs.com/godfray/p/4077146.html>

贝尔曼-福特算法，它的原理是**对图进行V-1次松弛操作，得到所有可能的最短路径**。其优于Dijkstra算法的方面是边的权值可以为负数、实现简单，缺点是时间复杂度过高，高达O(VE)。

**1、算法流程**

给定图G(V, E)（其中V、E分别为图G的**顶点集与边集**），源点s，数组*Distant[i]记录从源点s到顶点i的路径长度*，初始化数组Distant[n]为, Distant[s]为0；

以下**操作循环执行至多n-1次，n为顶点数：**

对于每一条**边e(u, v)**，如果**Distant[u] + w(u, v) < Distant[v]**，则另Distant[v] = Distant[u]+w(u, v)。w(u, v)为边e(u,v)的权值；

若上述操作没有对Distant进行更新，说明最短路径已经查找完毕，或者部分点不可达，跳出循环。否则执行下次循环；

为了**检测图中是否存在负环路，即权值之和小于0的环路**。对于每一条边e(u, v)，如果存在**Distant[u] + w(u, v) < Distant[v]**的边，则图中存在负环路，即是说改图无法求出单源最短路径。否则数组Distant[n]中记录的就是源点s到各顶点的最短路径长度。

我个人倒是觉得有点像DFS啊，对刚刚更新的结点继续下一层的搜索、计算权值，取更小的那个作为新的权值。当每个结点在这一轮都不再更新的时候，算法结束。

**2、算法三部分**

* 第一，初始化所有点。每一个点保存一个值，表示从原点到达这个点的距离，将原点的值设为0，其它的点的值设为**无穷大（表示不可达）**。
* 第二，进行循环，**循环下标**为从1到n－1（n等于**图中点的个数**）。在*循环内部，遍历所有的边，进行松弛计算*。
* 第三，遍历**途中所有的边（edge(u,v)）**，判断是否存在这样情况：

**d(v)>d(u) + w(u,v)**

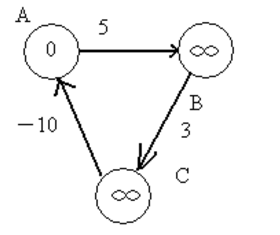
则返回false，表示途中存在从源点可达的权为负的回路。

或者：

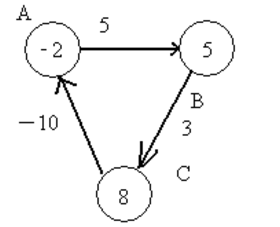
* **初始化：**将除源点外的所有顶点的**最短距离估计值** d[v] ← +∞, d[s] ←0;
* **迭代求解：***反复对边集E中的每条边进行松弛操作*，使得*顶点集V中的每个顶点v的最短距离估计值逐步逼近其最短距离*；（运行|v|-1次）
* **检验负权回路：**判断*边集E中的每一条边的两个端点是否收敛*。如果存在未收敛的顶点，则算法返回false，表明问题无解；否则算法返回true，并且从源点可达的顶点v的最短距离保存在 d[v]中。

之所以需要第三部分的原因，是因为，如果存在从源点可达的权为负的回路。则应为无法**收敛**而*导致不能求出最短路径*。可知，Bellman-Ford算法寻找单源最短路径的时间复杂度为**O(VE)**.

考虑如下的图：

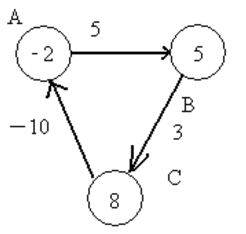


经过第一次遍历后，**点B的值变为5，点C的值变为8**，这时，注意权重为－10的边，这条边的存在，导致点A的值变为－2。（8＋ －10＝－2）



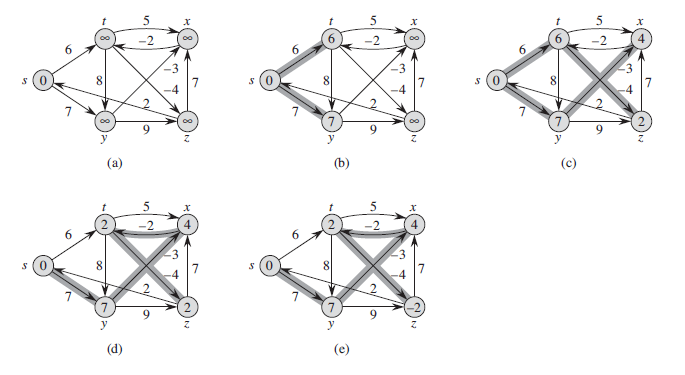
第二次遍历后，点B的值变为3，点C变为6，点A变为－4。正是因为有一条负边在回路中，导致*每次遍历后，各个点的值不断变小*。

在回过来看一下**bellman－ford算法的第三部分**，遍历所有边，检查是否存在**d（v） > d (u) + w(u,v)**。因为第二部分循环的次数是**定长的**，所以如果存在无法收敛的情况，则肯定能够在第三部分中检查出来。比如



此时，点A的值为－2，点B的值为5，边AB的权重为5，5 > -2 + 5. 检查出来这条边**没有收敛**。所以，*Bellman－Ford算法可以解决图中有权为负数的边的单源最短路径问题*。

**3、算法示例**

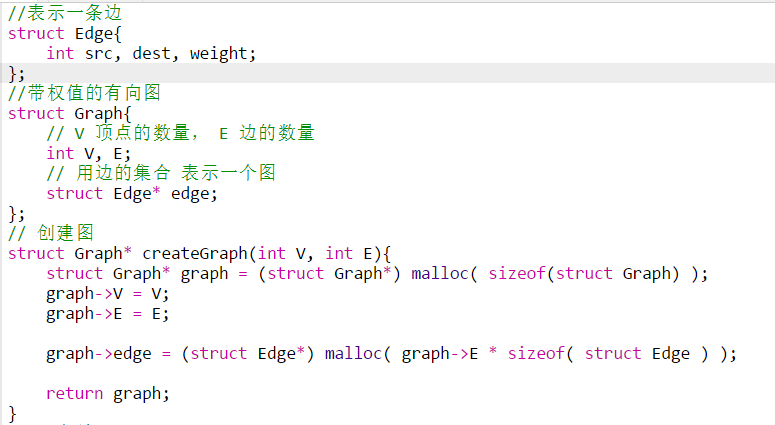


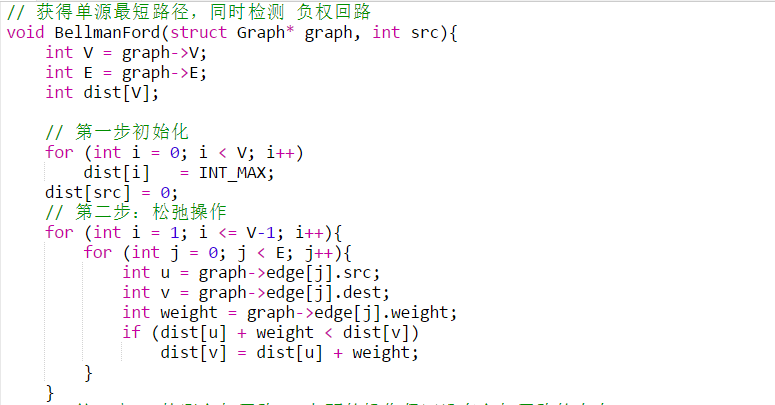
**算法伪代码：**

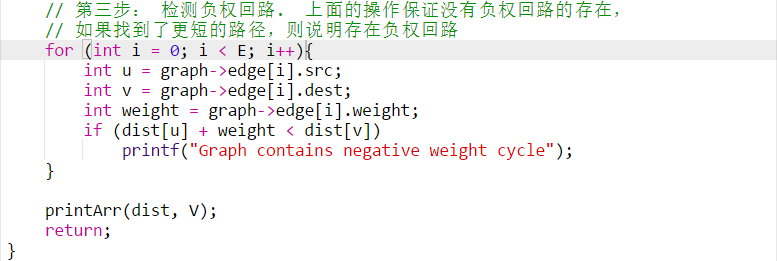


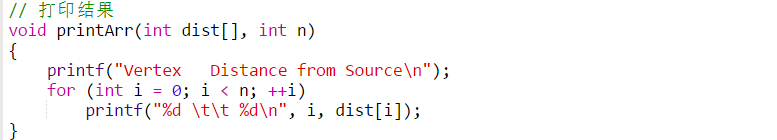
**4、代码实现**

C++实现：









**四、SPFA算法 – Bellman-Ford算法的改进（动态逼近法）**

<http://blog.sina.com.cn/s/blog_a46817ff01015g9h.html>

单源最短路径的算法最常用的是**Dijkstra**，些算法从时间复杂度来说为O（n^2),但是面对含**有负权植的图（权值特点）**来说就无能为力了，此时**Bellman-ford算法**就有用了，算法是采用的是*动态规化*的思想，但是1994年西南交通大学段凡丁发表了*SPFA(Shortest Path Faster Algorithm)*听这个名字就懂了，这种算法在时间上一定很快了。它是对Dellman-ford的优化，所以建议今后直接学SPFA。很多时候，**给定的图存在负权边**，这时类似Dijkstra等算法便没有了用武之地，而Bellman-Ford算法的复杂度又过高，SPFA算法便派上用场了。

**1、思路**

简洁起见，我们*约定有向加权图G不存在负权回路，即最短路径一定存在*。当然，我们可以在执行该算法前做一次**拓扑排序**，以**判断是否存在负权回路，**但这不是我们讨论的重点。和上文一样，我们用*数组d记录每个结点的最短路径估计值，而且用邻接表来存储图G*。我们采取的方法是**动态逼近法**：*设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点u，并且用u点当前的最短路径估计值对离开u点所指向的结点v进行松弛操作，如果v点的最短路径估计值有所调整，且 v点不在当前的队列中，就将v点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止*。

**定理3 只要最短路径存在，上述SPFA算法必定能求出最小值**。

**证明：**每次将点放入队尾，都是经过松弛操作达到的。换言之，*每次的优化将会有某个点v的最短路径估计值d[v]变小*。所以算法的执行会使d越来越小。由于我们假定图中不存在负权回路，所以每个结点都有最短路径值。因此，算法不会无限执行下去，随着d值的逐渐变小，直到到达最短路径值时，算法结束，这时的最短路径估计值就是对应结点的最短路径值。（证毕）

刚才我们只是笼统地说SPFA算法在效率上有过人之处，那么到底它的**复杂度**是怎样的？

**定理4 在平均情况下，SPFA算法的期望时间复杂度为O(E)**。

**证明：**上述算法每次取出队首结点u，并访问u的所有**临结点**的复杂度为O(d)，其中d为点u的**出度**。运用均摊分析的思想，对于V个点E条边的图，**点的平均出度为E/V**，所以每处理一个点的复杂度为**O(E/V)**。假设结点入队的次数h，显然h随图的不同而不同。但它仅与边的权值分布有关。我们设 **h=kV**，则算法SPFA的时间复杂度为**O(h\*E/V)=O(kE)**。在平均的情况下，可以将k看成一个比较小的常数，所以SPFA算法在一般情况下的时间复杂度为O(E)。（证毕）

聪明的读者一定发现了，SPFA和经过简单优化的Bellman-Ford无论在思想上还是在复杂度上都有相似之处。确实如此。两者的思想都属于标号修正的范畴。*算法是迭代式的，最短路径的估计值都是临时的。*算法思想是不断地逼近最优解，只在最后一步才确定想要的结果。但是他们实现的方式上存在差异。正因为如此，它们的时间复杂度其实有较大差异的。在Bellman-Ford算法中，要是**某个点的最短路径估计值更新**了，那么我们必须**对所有边指向的终点再做一次松弛操作**；在SPFA算法中，某个点的最短路径估计值更新，只有以该点为起点的边指向的终点需要再做一次松弛操作。在极端情况下，后者的效率将是前者的n倍，一般情况下，后者的效率也比前者高出不少。基于两者在思想上的相似，可以这样说，SPFA算法其实是Bellman-Ford算法的一个进一步优化的版本。

**2、算法流程（队列维护松弛成功的节点）**

算法大致流程是*用一个队列来进行维护*。初始时将源加入队列。***每次从队列中取出一个元素，并对所有与他相邻的点进行松弛，若某个相邻的点松弛成功，则将其入队***。直到队列为空时算法结束。

这个算法，简单的说就是队列优化的bellman-ford，利用了每个点不会更新次数太多的特点发明的此算法

SPFA -- Shortest Path Faster Algorithm，它可以**在O(kE)的时间复杂度内求出源点到其他所有点的最短路径**，可以处理负边。SPFA的实现甚至比Dijkstra或者Bellman\_Ford还要简单：

设**Dist代表S到I点的当前最短距离，Fa代表S到I的当前最短路径中I点之前的一个点的编号**。开始时Dist全部为+∞，只有Dist[S]=0，Fa全部为0。

维护一个队列，里面存放所有需要进行迭代的点。初始时队列中只有一个点S。用一个布尔数组记录每个点是否处在队列中。

每次迭代，取出队头的点v，依次**枚举**从v出发的边v->u，设边的长度为len，判断Dist[v]+len是否小于 Dist[u]，若小于则改进Dist[u]，将**Fa[u]记为v**，并且由于S到u的最短距离变小了，**有可能u可以改进其它的点，所以若u不在队列中，就将它放入队尾**。这样一直迭代下去直到队列变空，也就是S到所有的最短距离都确定下来，结束算法。**若一个点入队次数超过n，则有负权环**。

SPFA 在形式上和**宽度优先搜索**非常类似，不同的是宽度优先搜索中一个点出了队列就不可能重新进入队列，但是SPFA中一个点可能在出队列之后再次被放入队列，也就是一个点改进过其它的点之后，过了一段时间可能本身被改进，于是再次用来改进其它的点，这样反复迭代下去。设一个点用来作为迭代点对其它点进行改进的平均次数为k，有办法证明对于通常的情况，k在2左右。

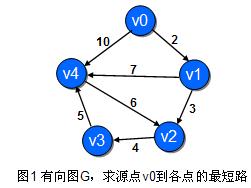
**3、算法示例**

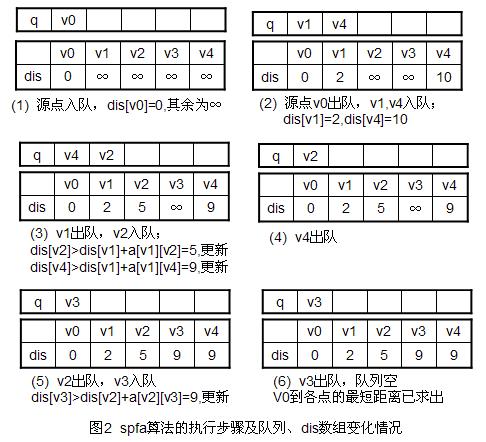
<http://www.layz.net/LAOJ/suanfa/s9-4.html>

设立一个**先进先出的队列q**用来保存待优化的结点，优化时每次取出队首结点u，并且用u点**当前的最短路径估计值**对离开u点所指向的结点v进行松弛操作，如果v点的最短路径估计值有所调整，且v点不在当前的队列中，就将v点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止。

松弛操作的原理是**著名的定理：“三角形两边之和大于第三边”**，在信息学中我们叫它三角不等式。所谓对结点i,j进行松弛，就是判定是否**dis[j]>dis[i]+w[i,j]**，如果该式成立则将dis[j]减小到dis[i]+w[i,j]，否则不动。

下面举一个实例来说明SFFA算法是怎样进行的：





**与BFS的区别：**

SPFA 在形式上和广度(宽度)优先搜索非常类似，不同的是bfs中一个点出了队列就不可能重新进入队列，但是SPFA中一个点可能在出队列之后再次被放入队列，也就是一个点改进过其它的点之后，过了一段时间可能本身被改进(重新入队)，于是再次用来改进其它的点，这样反复迭代下去。

**4、代码实现**

**1）伪代码**



**2）代码**

