

单源最短路

吉林大学计算机学院
谷方明

fmgu2002@sina.com





学习目标

- 了解最短路及其分类
- 掌握求解单源最短路问题的4个常用算法
 - ✓ BFS
 - ✓ Dijkstra算法
 - ✓ Bellman-Ford算法
 - ✓ SPFA算法



背景

- 最短路是实际应用中经常遇到的问题。
 - ✓ 旅游： 长春→杭州
 - ✓ 最少时间
 - ✓ 最少费用
- 最短路问题定义： 给定图中，从某顶点出发，找从该点到其它顶点 **cost**最小的路径。路径的**cost** 指该路径边的权值和。



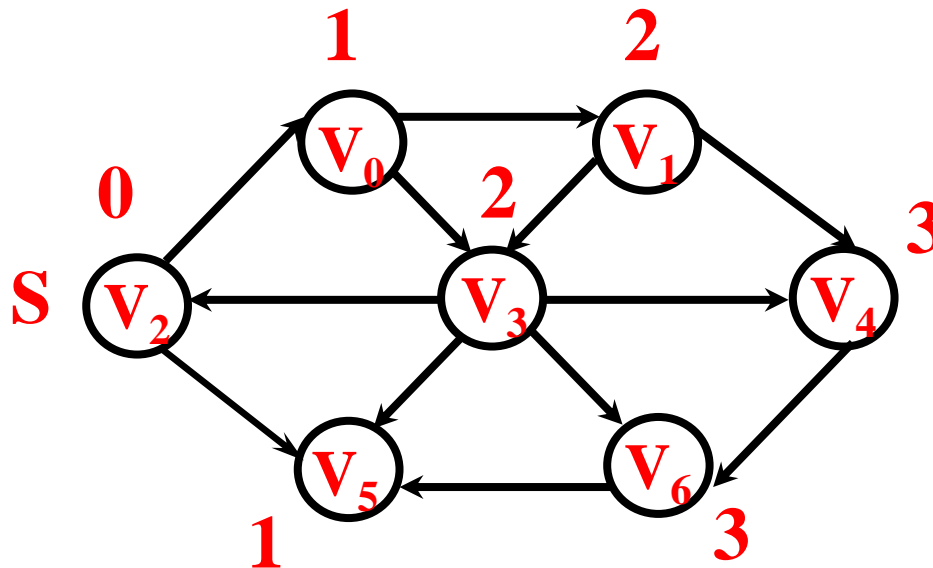
最短路问题分类

- 两个顶点间的最短路：从一个指定的顶点到达另一指定顶点的最短路问题。
- 单源最短路：从一个指定的顶点到其它所有顶点的最短路径问题，Single Source Shortest Path, SSSP.
- 任意顶点间的最短路：所有顶点间的最短路；



1. 无权图的单源最短路

- 无权——所有边的权值都为 1.
- 源点到各顶点的路径长度：所经历的边的数目



- 从源点开始由近及远依次求各顶点的最短路径

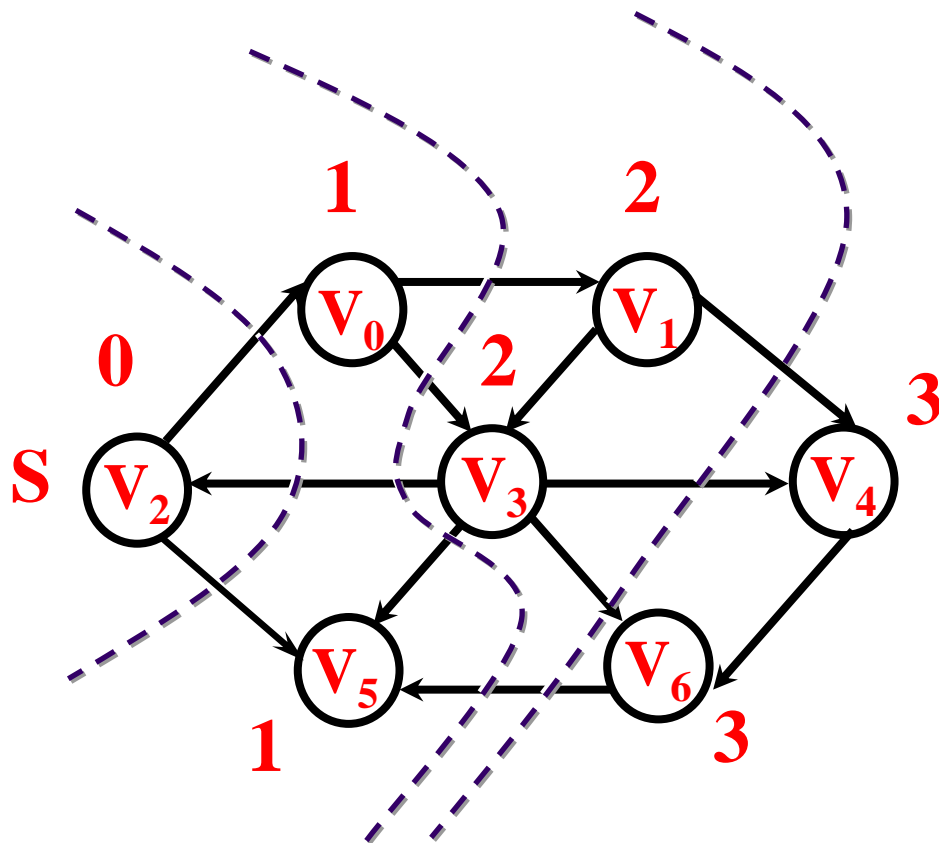


算法思想

设 D_i 为源点 S 到顶点 i 的最短路径长度

- ①访问初始顶点 S ，即令 $D_S = 0$.
- ②从 S 出发，找最短路径为1的顶点，即 S 的所有邻接顶点 w ，令 $D_w = D_S + 1$
- ③从上步访问的顶点 w 出发，找最短路径为2的顶点，即 w 未被访问的邻接顶点 v ，令 $D_v = D_w + 1$
- ④继续上述过程，直至处理完所有顶点。

BFS



- 访问顶点的次序与对图进行广度优先遍历BFS的次序是一致的;



算法实现

- 图采用邻接表存储;
- 用队列保存待处理顶点;
- 使用数组 **dist[]** 存储最短路径值。
 - ✓ 源点初始化为0, 其它初始为-1.
 - ✓ 当 **dist[i]** 从 -1 变为非负时, 表示从源点到i的最短路径已求完
- 为找到最短路径, 可用 **path[]** 存储从源点到顶点 **i** 的最短路上 **i** 之前的顶点。



算法描述

算法**ShortestPath(v)**

*/*计算从顶点v到其他各顶点的最短路径*/*

S1[初始化]

CREATEQueue(Q) . */* 创建一个队列 */*

for(i =1 ; i <= n ; i ++){

path[i] = -1; dist[i] = -1.

}

dist[v] = 0 ;

Q.insert(v);



S2[求从顶点v到其他各顶点的最短路径]

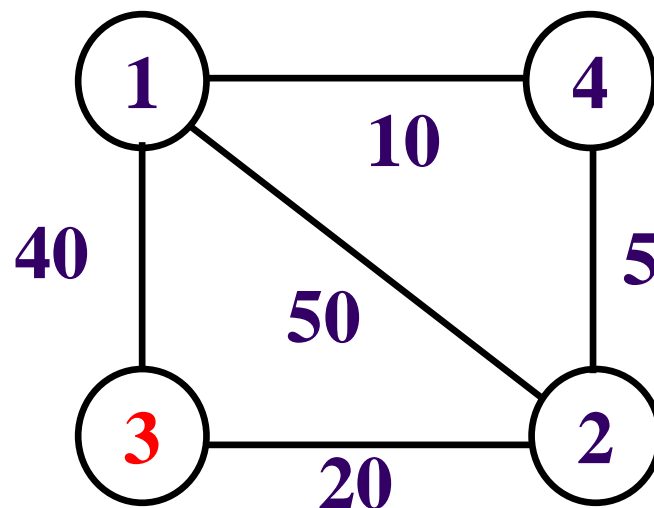
```
while ( ! Q.empty() ) {  
    u = Q.delete();          /* 队头顶点u出队 */  
    for( p = Head[u].adjacent ; p ; p = p->link ){  
        k = p -> VerAdj ;  
        if (dist[k] == -1) {  
            Q.insert(k); // u未访问的邻接顶点入队  
            dist[k] = dist[u] + 1; path[k] = u; }  
        }  
    }  
}
```



算法分析

- 邻接表：在最短路径的计算中，一个顶点入队出队各一次，时间复杂性为 $O(n)$ ；而对每个顶点，都要对它的边链表进行遍历，其遍历邻接表的开销为 $O(e)$ ，于是整个算法的时间复杂性为 $O(n+e)$ 。
- 邻接矩阵： $O(n^2)$

2. 权图（非负）



- ❑ 从源点开始，按广度优先扩展；
 - ✓ 失败！权值只更新一次
- ❑ 允许权值多次更新，按广度优先扩展；
 - ✓ 失败！原因：盲目扩展，没利用当前的最短路信息
- ❑ 一种思想：选择未求完最短路的点中当前最短路径最小的点扩展



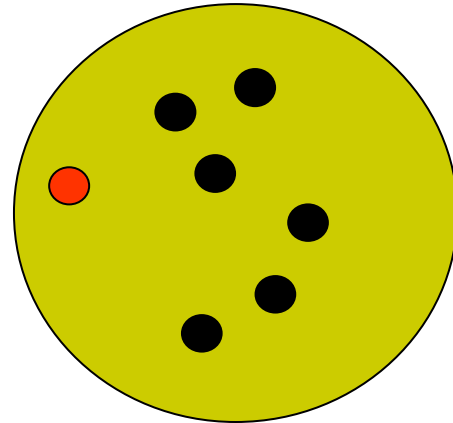
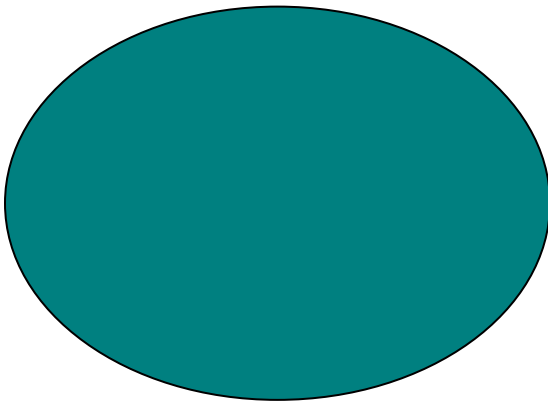
Dijkstra算法

- 把图中所有顶点分成两组，**第一组**：已求完最短路径的顶点；**第二组**：未求完最短路径的顶点。
- 每次从第二组中选择当前最短路径最小的点处理，逐个把第二组的顶点加到第一组。



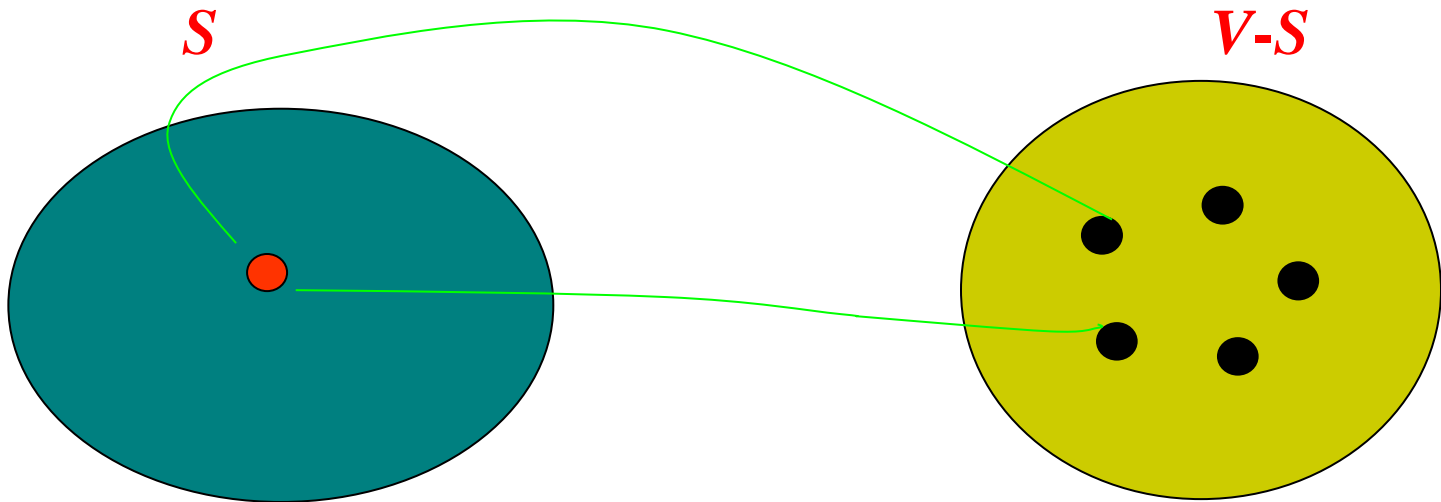
具体操作

- 设数组 D , D_i 表示当前找到的源点 v_0 到 v_i 的最短路径长度。
- 设 S 是已求得最短路径的顶点集合。
- 初始时: $S = \{ \}$, $D_0 = 0$, $D_i = +\infty$

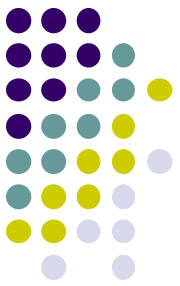




- 第一次操作, $S=\{v_0\}$, 若 v_0 到 v_i 有边, 则 D_i 更新为边上的权值; v_0 到 v_i 无边, 则 D_i 仍为 $+\infty$ 。



- 每次求最短路径, 就是在 $V-S$ 中找具有最小 D 值的顶点 v_k , 将 v_k 加入集合 S , 然后对 $v_i \in V-S$, 修改 D_i



Dijkstra算法描述-自然语言

- ① 设 s 为初始顶点, $D_s=0$ 且 $\forall i \neq s, D_i = +\infty$
- ② 在未求完的顶点中选择 D_v 最小的顶点 v , 访问 v , 令 $S[v]=1$ 。
- ③ 依次考察 v 的邻接顶点 w , 若
$$D_v + \text{weight}(\langle v, w \rangle) < D_w ,$$
则使 $D_w = D_v + \text{weight}(\langle v, w \rangle)$ 。
- ④ 重复② ③ , 直至所有顶点被访问。



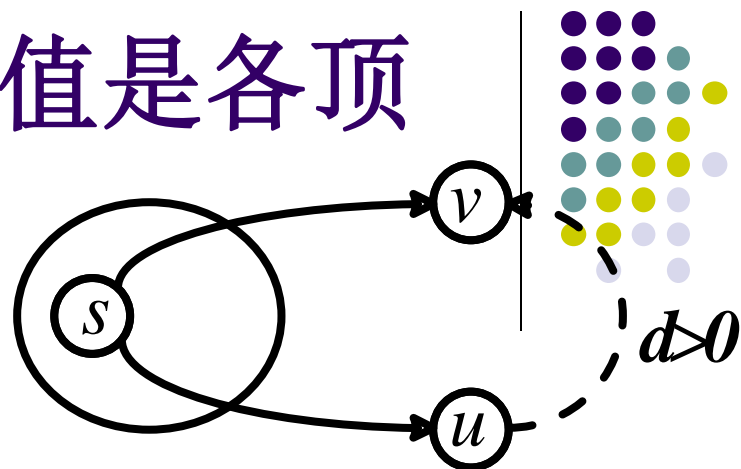
Dijkstra算法的正确性

- 定理6.4 Dijkstra算法可以按照非递减次序依次得到各顶点的最小路径长度。
- 建议分开证：
 - ✓ 算法得到的路径值是各顶点的最小路径长度；
 - ✓ 算法得到的路径值是按非递减次序得到的。

证明1:算法得到的路径值是各顶点的最小路径长度

反证法

- 分析算法可知: D_i 是从 s 出发, 仅经由 S 中的点得到的最短路径值。
- 假设选择 v 点, 但 D_v 不是 s 到 v 的最短路径值
- 取 s 到 v 的一条最短路径, 令 u 为这条路径上第一个不在 S 中的点, $D_u + d < D_v$ 。由于权值非负, $d \geq 0$, 从而 $D_u < D_v$
- 但选择 v 扩展而未选择 u , 则 $D_v \leq D_u$, 矛盾



证明2:算法得到的路径值是按非递减次序得到的



数学归纳法

□ 基础步骤

✓ $D_S=0$

□ 归纳步骤: 设 $D_S \leq D_1 \leq D_2 \leq \dots \leq D_k$, 往证 $D_k \leq D_{k+1}$ 。 $D_{k+1} = D_i + \text{weight}\langle i, k+1 \rangle$ 。

✓ $i=k$ 时, 显然成立

✓ $i < k$ 时, 先扩展 k , 后扩展 $k+1$, 也有 $D_k \leq D_{k+1}$



Dijkstra算法实现

算法DShortestPath(v)

/* 计算 v 到其他各顶点的最短路径 */

D1[初始化]

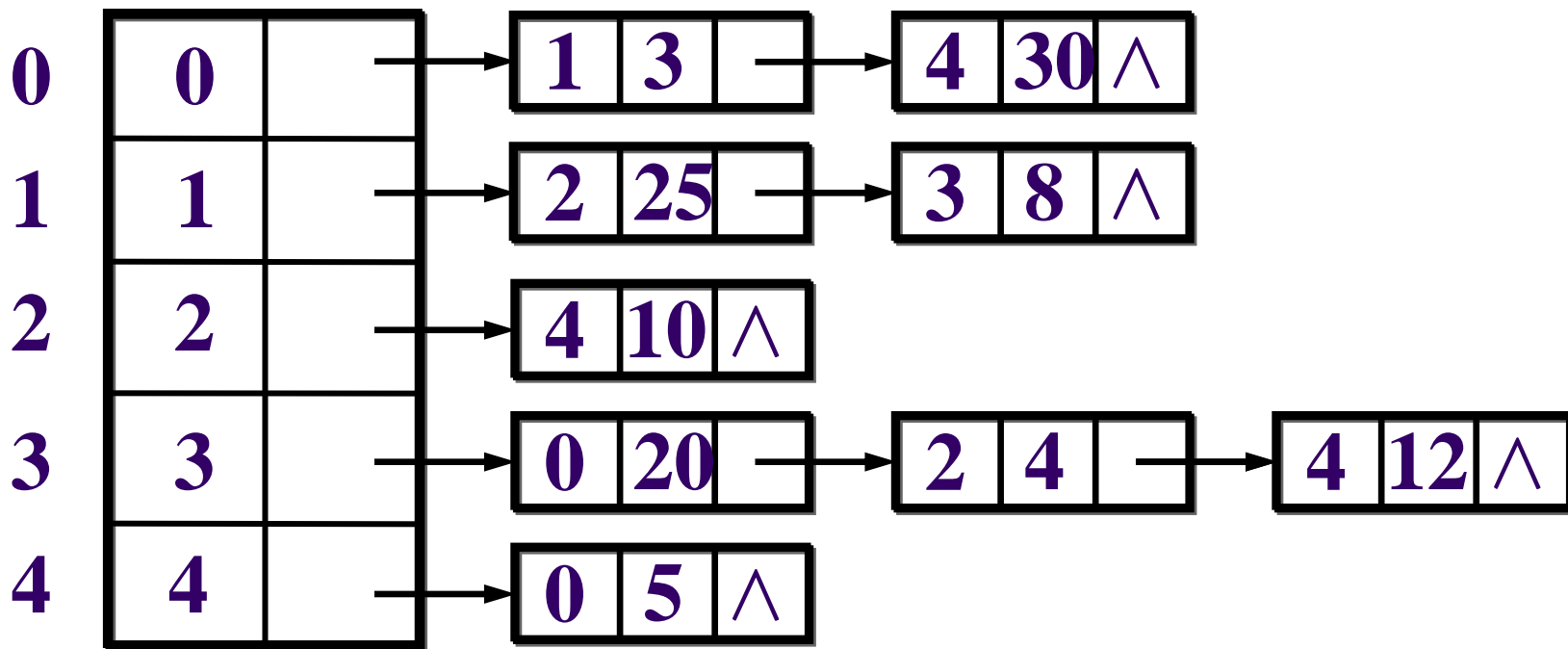
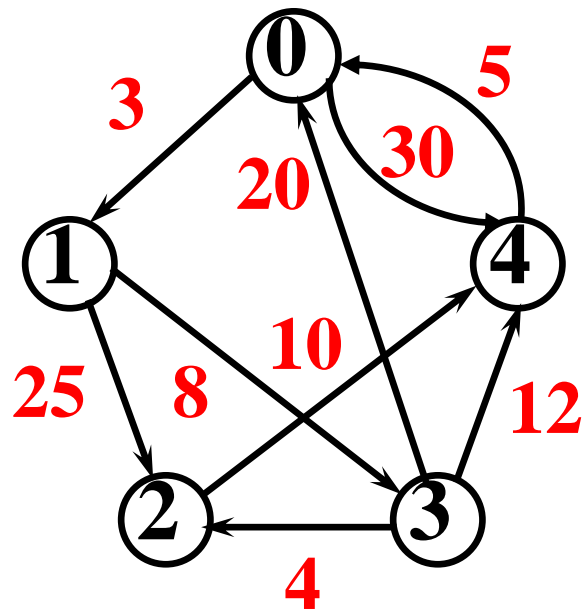
```
for(  $i = 1$  ;  $i \leq n$  ;  $i++$  ){  
     $path[i] = -1$ ;  $dist[i] = \max$ ;  
     $s[i] = 0$ ; // 数组 $s[i]$  记录  $i$  是否已计算完  
}  
  
 $dist[v] = 0$ ;
```

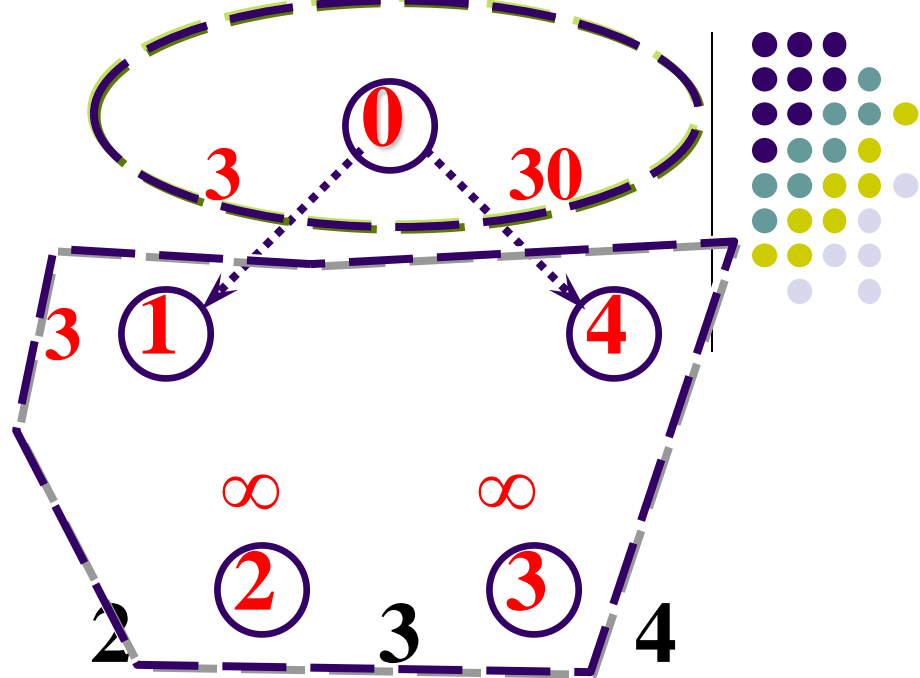


D2[求从v到其他各顶点的最短路径]

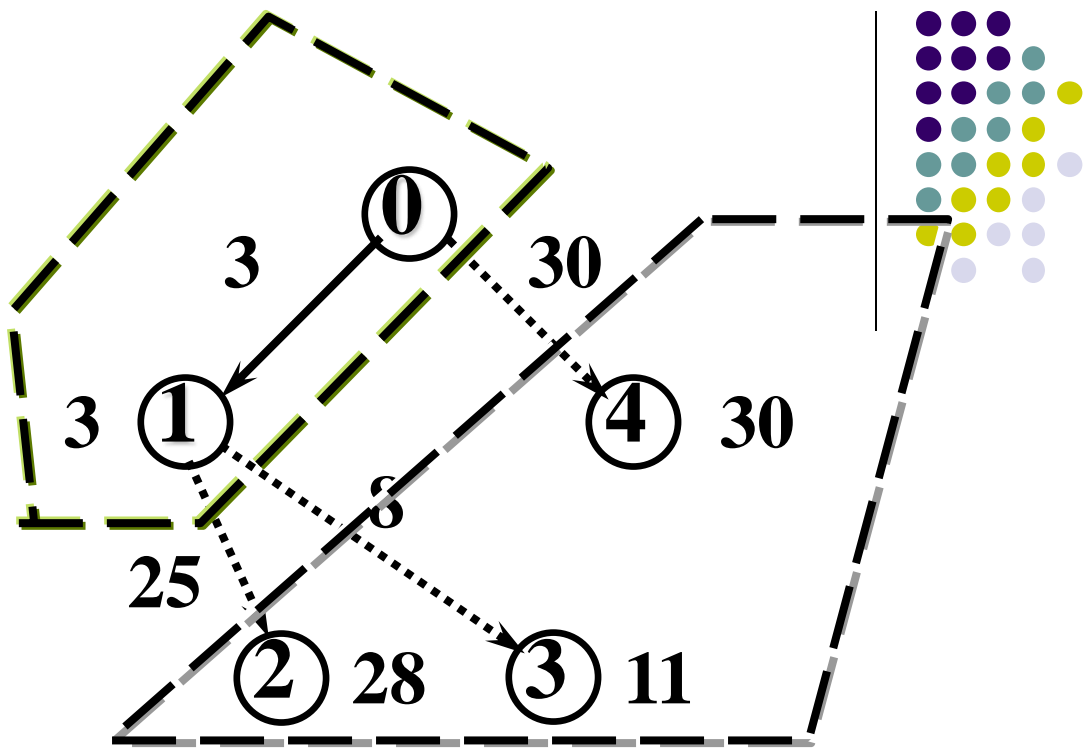
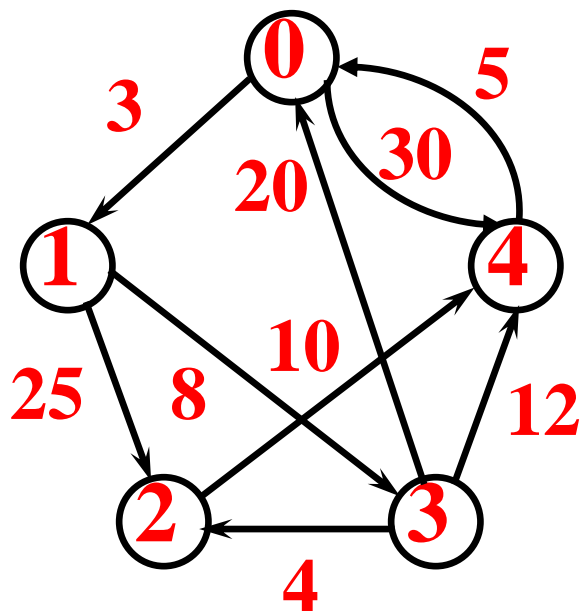
```
for( j = 1 ; j < n ; j ++ ){  
    mindist = max; //循环:确定即将被访问的顶点u  
    for( i = 1 ; i <= n ; i ++ )  
        if ( dist[i] < mindist && s[i] == 0 )  
            { mindist = dist[i] ; u = i ; }  
    s[u] = 1.  
    for( p = Head[u].adjacent ; p ; p = p->link ){  
        k = p -> VerAdj ;  
        if ( dist[u] + cost(p) < dist[k] ) //松弛  
            { dist[k] ← dist[u] + cost(p) ; path[k] = u ; }  
    }  
}
```

例

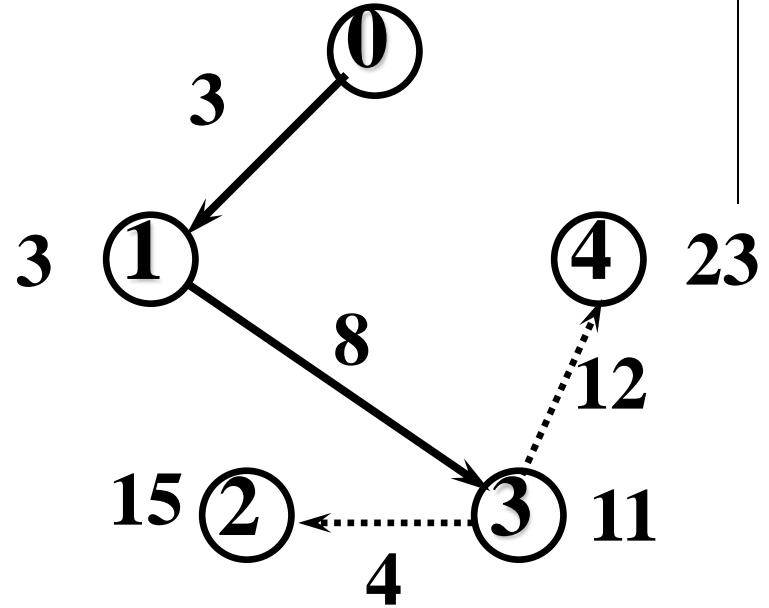
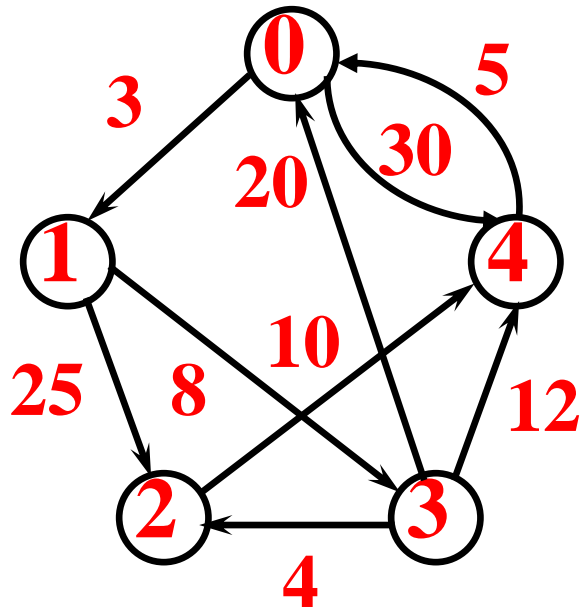




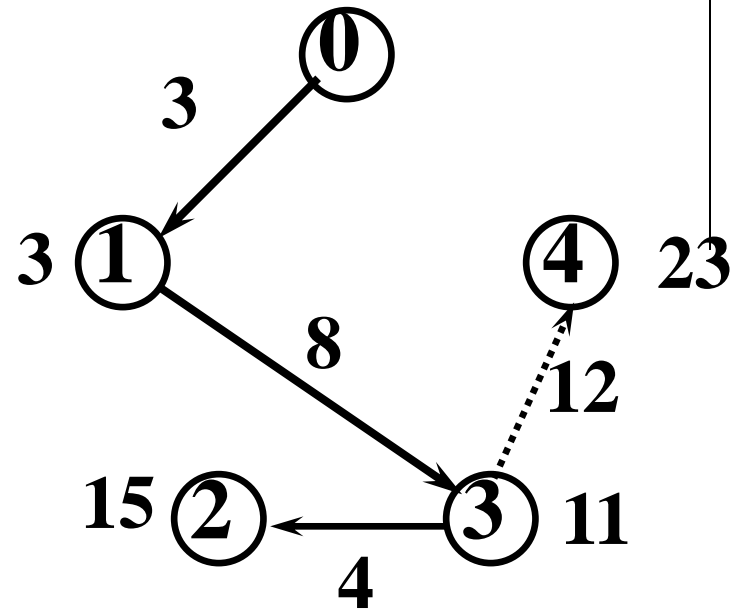
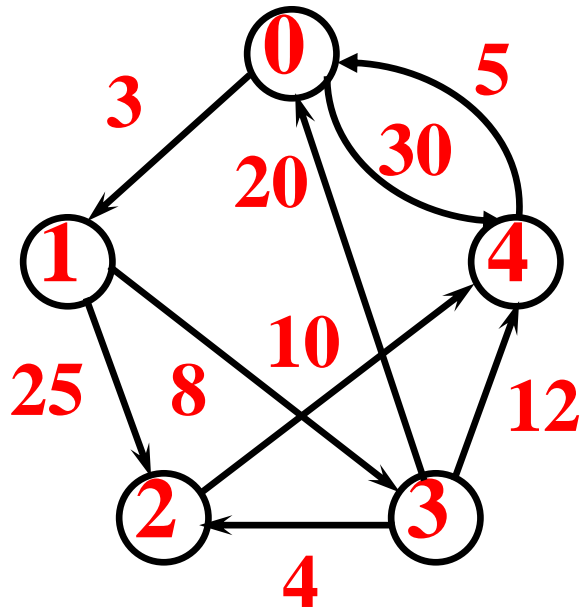
s dist path	1	0	0	0	0
	0	3	∞	∞	30
		v_0			v_0



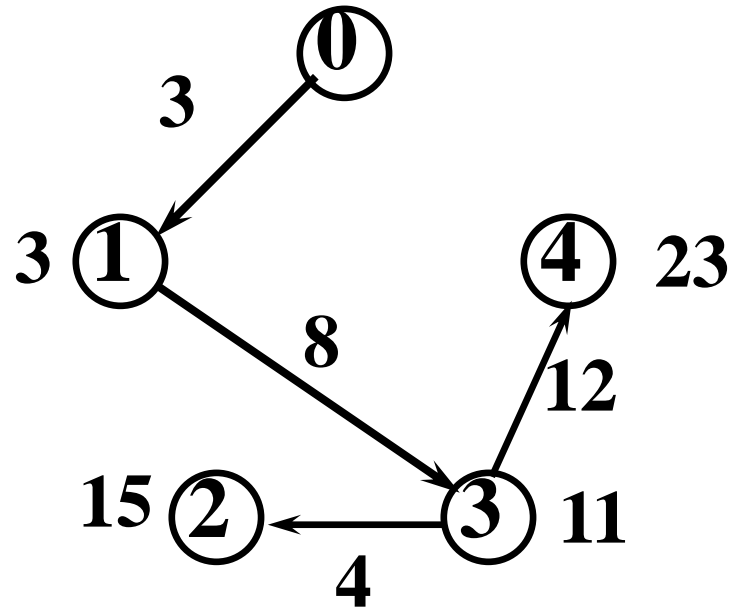
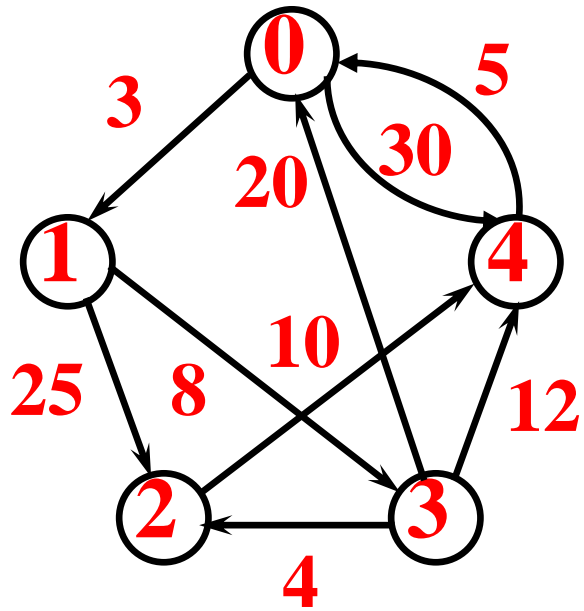
	0	1	2	3	4
s	1	1	0	0	0
dist	0	3	28	11	30
path		v_0	v_1	v_1	v_0



	0	1	2	3	4
s	1	1	0	1	0
dist	0	3	15	11	23
path		v_0	v_3	v_1	v_3



	0	1	2	3	4
s	1	1	1	1	0
dist	0	3	15	11	23
path		v_0	v_3	v_1	v_3



	0	1	2	3	4
s	1	1	1	1	1
dist	0	3	15	11	23
path		v_0	v_3	v_1	v_3



算法分析

- 时间复杂性：在**Dijkstra**算法中，循环扫描被访问顶点的边链表，时间复杂性为 $O(d_u)$ （ d_u 为顶点 u 的邻接顶点个数）；循环扫描顶点表求最小dist，时间复杂性为 $O(n)$ ；循环体要被执行 n 次，因此整个算法的时间复杂性为

$$O\left(\sum_{i=0}^{n-1} (n + d_i)\right) = O\left(n^2 + \sum_{i=0}^{n-1} d_i\right) = O(n^2 + e)$$

- 与存储方式无关；与边数无关

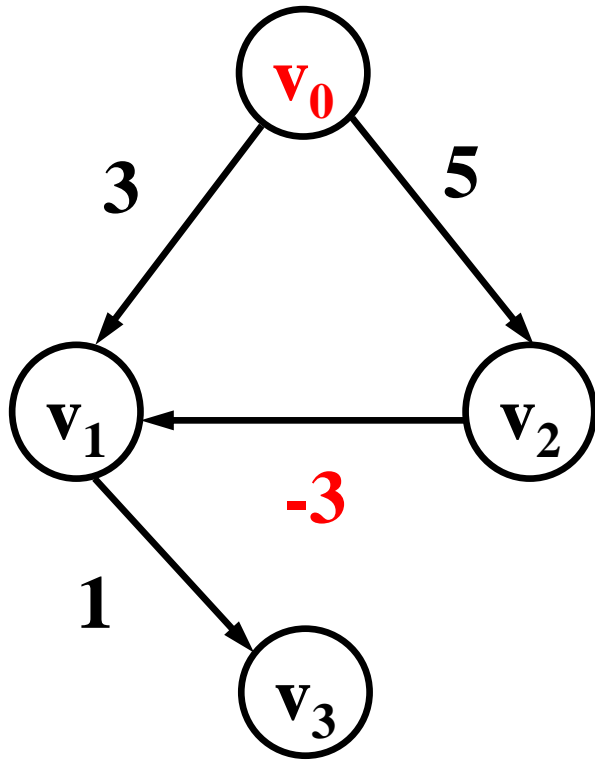


Dijkstra算法小结

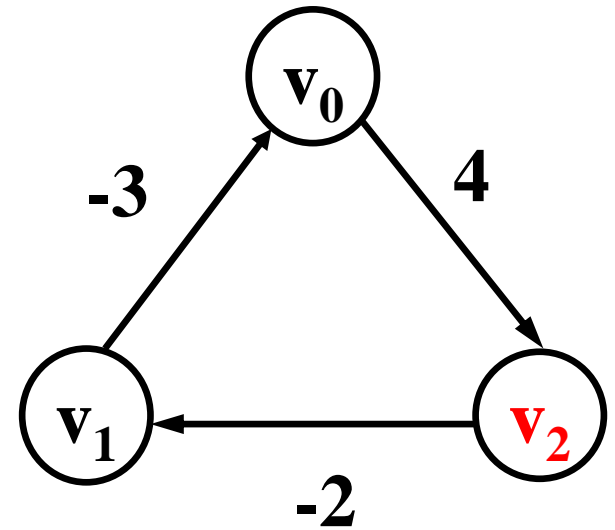
- 标记法（基于点）
- 松弛操作 $\text{relax}(u,v)$
若 $D_u + \text{weight}(\langle u,v \rangle) < D_v$,
则 $D_v = D_u + \text{weight}(\langle u,v \rangle)$
- 三角不等式: $\delta(s,v) \leq \delta(s,u) + w(u,v)$
- 时间复杂度: $O(n^2)$
- 适用条件: 权值非负



负权边和负环



带负权边的图：
Dijkstra算法失效



带负环的图：最短路径的定义失效



Bellman-Ford算法思想

- 松弛操作 $\text{relax}(u,v)$: 对有向边 $\langle u,v \rangle$, if $d[v] > d[u] + w(u,v)$ then $d[v] \leftarrow d[u] + w(u,v)$ 。
- 对边集执行 n 遍松弛。
 - ✓ $n = |V|$, 即图的顶点数
 - ✓ 一遍松弛: 对边集的每条边都执行一次松弛操作;
- 如果第 n 遍还有边可被松弛, 则判定图中存在负环; 否则, 已求出最短路径



Bellman-Ford算法描述

1. 初始化:

for each vertex $v \in V(G)$ do $d[v] \leftarrow +\infty$; $d[s] \leftarrow 0$;

2. 迭代求解: (运行 $|V|-1$ 次)

for $i=1$ to $|V|-1$ do

for each edge $\langle u, v \rangle \in E(G)$ do
relax(u, v);

3. 检验负权回路:

for each edge $(u, v) \in E(G)$ do
if $d[v] > d[u] + w(u, v)$ then Exit false;
Exit true;

例1：带负权边的图

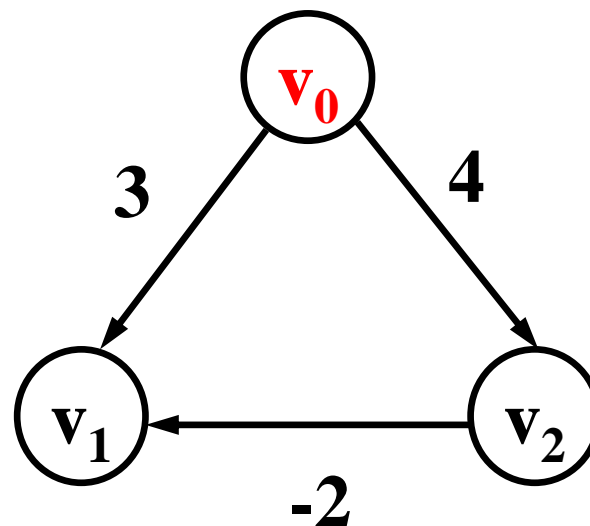


0	1	2
0	∞	∞

0	1	2
0	3	4

 \Rightarrow

0	1	2
0	2	4



$$E = \{ \langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 2, 1 \rangle \}$$



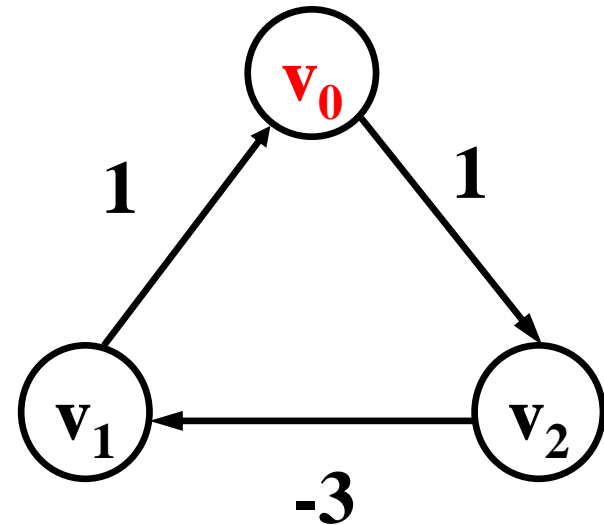
例2：带负环的图

0	1	2
0	∞	∞

0	1	2
0	-2	1

0	1	2
-1	-2	1

0	1	2
-1	-3	0

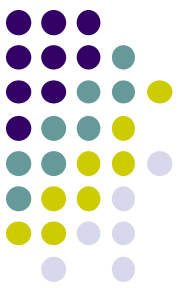


$$E = \{ \langle 0, 2 \rangle, \langle 1, 0 \rangle, \langle 2, 1 \rangle \}$$



Bellman-Ford算法的正确性

- 图的任意一条最短路径既不能包含负权回路，也不会包含正权回路，因此最多包含 $n-1$ 条边；
- 从源点 s 可达的所有顶点如果存在最短路径，则这些最短路径构成一个以 s 为根的最短路径树。
- **Bellman-Ford**算法的 n 遍迭代松弛，就是按每个顶点距离 s 的层次（**最短路径树上的层次，不是原图的层次**），逐层生成最短路径树的过程。



- 对边集进行第 1 遍松弛的时候，生成了从 **s** 出发，层次至多为 1 的那些树枝，也就是说，找到了与 **s** 至多有 1 条边相连的那些顶点的最短路径；
- 对边集进行第 2 遍松弛的时候，生成了第 2 层次的树枝，就是说找到了从 **s** 出发经过 2 条边相连的那些顶点的最短路径.....
- 因为最短路径最多只包含 $n-1$ 条边，所以最多需要循环 $n - 1$ 次。



- 如果没有负权回路，由于最短路径树的高度最多只能是 $n - 1$ ，所以最多经过 $n - 1$ 遍松弛操作后，所有从 s 可达的顶点必将求出最短距离。如果 $d[v]$ 仍保持 $+\infty$ ，则表明从 s 到 v 不可达。
- 如果有负权回路，那么第 n 遍松弛操作仍然会成功，这时，负权回路上的顶点不会收敛。



Bellman-Ford算法描述（ADL）

算法Bellmanford(s)

/*求单源最短路， s为源点， 有负环false， 否则true*/

B1[初始化]

for(i = 1; i <= n ; i ++) d[i] = INF;

d[s] = 0;



B2[迭代求解]

```
for( i = 1; i <= n-1 ; i ++ )
    for(u = 1; u <= n; u++)
        for( p = head[u].adjacent; p ; p = p->link){
            v = p -> VerAdj ;
            if ( dist[u] + cost(p) < dist[v] ) //松弛
                dist[v] = dist[u] + cost(p) ;
        }
```



B3 [判负环]

```
for(u = 1; u <= n; u++)  
    for( p = head[u].adjacent; p ; p = p->link)  
        v = p -> VerAdj ;  
        if ( dist[u] + cost(p) < dist[v] )  
            return false;  
return true; ■
```

□ 时间复杂度 $O(n * e)$



Bellman-Ford算法小结

- 基于边的思想；
- 能处理负权边、判负环；
- 时间复杂度 $O(n \cdot e)$
- 优点是简单；缺点是效率略低。



Bellman-Ford算法的基本优化

- 思想：如果在某一遍迭代中，松弛操作未执行，说明该遍迭代所有的边都没有被松弛。此后，边集中所有的边肯定都不会再被松弛，从而提前结束迭代过程。
- 设计算法时，引入松弛标识**relaxed**来实现



基本优化的算法描述

B2[迭代求解, 引入松弛标识]

```
for( i = 1; i <= n-1 ; i ++ ){  
    relaxed = false;  
    for(u = 1; u <= n; u++)  
        for( p = head[u].adjacent; p ; p = p->link)  
            v = p -> VerAdj ;  
            if ( dist[u] + cost(p) < dist[v] ) { //松弛  
                dist[v] = dist[u] + cost(p) ;  
                relaxed= true;  
            }  
    if (!relaxed) break;  
}
```



基本优化的算法分析

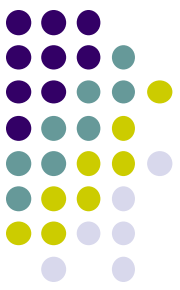
- 有研究表明，对于随机生成数据的平均情况，时间复杂度的估算公式为

✓ $1.13|E|$ if $|E| < |V|$

✓ $0.95 * |E| * \lg|V|$ if $|E| > |V|$

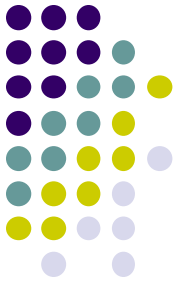
基本优化的**Bellman-Ford**算法效率和堆优化的**Dijkstra**算法效率相近。

- 在处理带负环的图时，时间复杂度仍为 $O(n * e)$



SPFA算法

- 对Bellman-Ford算法优化的关键之处在于意识到：
只有那些在前一遍松弛中改变了最短路径值的点，才可能引起它们的邻接点的最短路径值的改变。
- 因此，用一个先进先出的队列来存放被成功松弛的顶点。初始时，源点 s 入队。当队列不为空时，取出队首顶点，对它的邻接点进行松弛。如果某个邻接点松弛成功，且该邻接点不在队列中，则将其入队。经过有限次的松弛操作后，队列将为空，算法结束。



算法实现

- 邻接表
- queue
- 标志数组 mark



SPFA

算法**SPFA**(v)

*/*v为源点求单源最短路*/*

S1[初始化]

for(i = 1; i <= n ; i ++) d[i] = INF;

d[v] = 0;

CreateQueue Q;

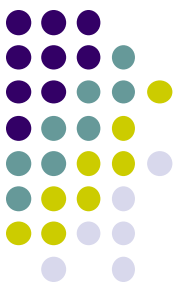
Q.insert(v);

mark[v] = 1;



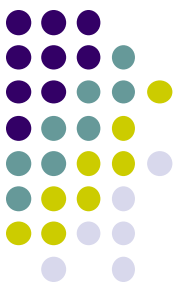
S2[迭代求解]

```
while( ! Q.empty() ){  
    u = Q.delete(); mark[u] = 0;  
    for( p = head[u].adjacent; p ; p = p->link){  
        k = p -> VerAdj ;  
        if ( dist[u] + cost(p) < dist[k] ) {//松弛  
            dist[k] = dist[u] + cost(p) ;  
            if( ! mark[k]) {Q.insert(k) ; mark[k]=1; }  
        }  
    }  
}
```

分析

- 仅当图不存在负权回路时，**SPFA**能正常工作。如果图存在负权回路，由于负权回路上的顶点无法收敛，总有顶点在入队和出队往返，队列无法为空，这种情况下**SPFA**无法正常结束。
- 判断负权回路的方案很多
 - ✓ 记录每个结点进队次数，超过 $|V|$ 次表示有负环；
 - ✓ 记录这个结点在路径中处于的位置 $ord[i]$ ，每次更新的时候 $ord[i]=ord[x]+1$ ，若超过 $|V|$ 则表示有负环；
- **SPFA**的时间复杂度一般认为是 $O(kE)$ ， k 是常数；最坏 $O(VE)$



总结

- 无权单源最短路使用**BFS**。
- **Dijkstra**算法是基于点的算法，常用算法，适用边权负权的单源最短路，时间复杂度 $O(n^2)$ 。
- **Bellman-Ford**算法实现简单，时间复杂度 $O(n*e)$ ，效率低；基本优化的**Bellman-Ford**算法相对较好；
- **SPFA**算法的期望时间复杂度 $O(k*e)$ ，性能优秀。但如果需要判断是否存在负权回路，推荐使用基本优化的**Bellman-Ford**形式。



迷宫问题 (Maze)

- $n \times m$ 的迷宫，X代表墙，.代表空地
- 机器人起始位置 sx, sy ；每步走一格，上、下、左、右任意一个相邻的合法格子；
- 出口位置 tx, ty
- 求机器人走出迷宫的最少步数

样例



5 X 5 迷宫

* * * * *

R . *O*

* *
 ■ ■ ■

* * *
 ■ ■

* * * * *



广度优先搜索算法（BFS）

□ 隐含图的广度优先搜索

- ✓ 状态（解）
- ✓ 初始状态和状态扩展规则（隐含定义一个状态图）
- ✓ 目标状态

□ 适用于找最短解路径问题

- ✓ 最少步数、最少边数

□ 如果求路径，同时要保存多条解路径。



分析

- 机器人的初始位置(s_x, s_y)花费0步即可到达;
 - 利用(s_x, s_y), 检查花费1步到达的格子;
 - 利用1步到达的格子, 检查花费2步到达的格子;
 -
 - 如此下去, 直到到达出口位置, 或能访问的格子都被访问过
-
- 性质: 每个格子可能被多次访问; 但第一次访问时是最小的花费步数;



- 步数矩阵：保存每个格子的最小花费步数，与迷宫同样大小；**(sx,sy)**设为**0**；空地设为**-1**；墙设为无穷大；
- 队列：**按步数的大小依次保存**访问过的位置；访问过的位置不再进队
- 算法结束时**(tx,ty)**中的步数即为所求



BFS算法核心

```
q.insert((sx,sy));map[sx][sy] = 0;
while( ! q.empty() ){
    ( x,y ) = q.delete();
    for( i = 0;i < 4; i++ ) {
        nx = x + dx[i],ny = y + dy[i];
        if( check( nx , ny ) ) {
            map[nx][ ny] = map[x][ y] + 1;
            q.insert ( (nx, ny) );
            //if (nx == tx && ny==ty) ...
        }
    }
}
```