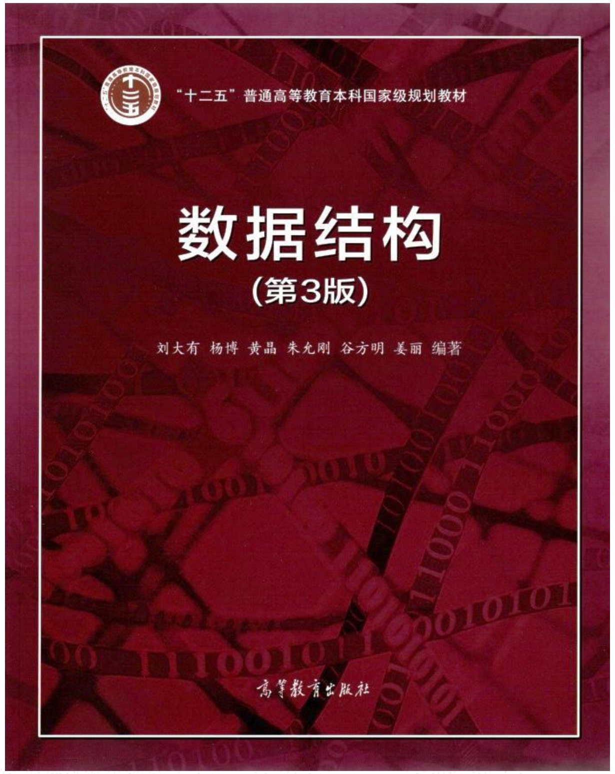
think.create.solve



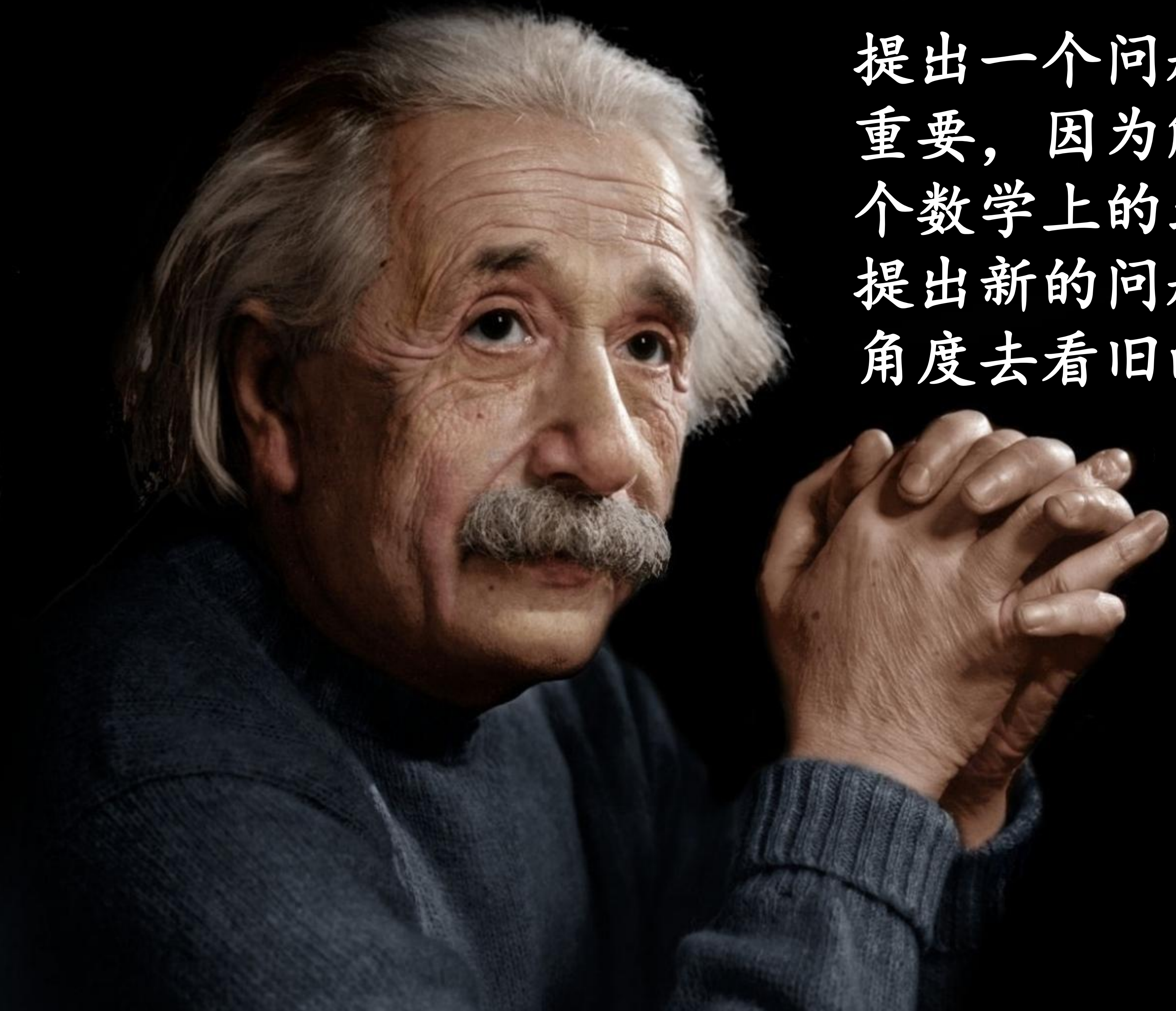
图的最短路径

- **无权图最短路径**
- Dijkstra算法
- A*算法
- Floyd算法
- Bellman-Ford算法
- 满足约束的最短路径

数据之法
结构之美
算法之道

Last updated on 2022.11

zhuyungang@jlu.edu.cn



提出一个问题往往比解决一个问题更重要，因为解决一个问题也许仅是一个数学上的或实验上的技能而已。而提出新的问题，新的可能性，从新的角度去看旧的问题，却需要有创造性的想象力，而且标志着科学的真正进步。

——— 爱因斯坦

路径长度

- 无权图：路径包含的边的条数。

- 带权图：路径包含的各边权值之和。

- 长度最小的路径称为最短路径，最短路径的长度也称为最短距离。

问题

无权图的单源最短路径问题

无权图的单源最短路径问题

- 单源最短路径：给定顶点到其它所有顶点的最短路径问题
- 源点到各顶点的路径所经历的边的数目就是路径的长度。
- BFS过程中，当访问某个顶点时，就确定了该点与源点的最短距离。
- 可以通过BFS，从源点开始由近及远求各顶点的最短路径。

回顾：广度优先遍历

- ✓ 首先访问起点（源点）v_1；
- ✓ 访问v_1的邻接顶点$w_1, w_2 ,\dots , w_k$；（与v_1最短距离为1的点）
- ✓ 然后，再顺次访问w_1，w_2，…，w_k的**未访问的**邻接顶点（与v_1最短距离为2的顶点）；
- ✓ 再从这些被访问过的顶点出发，逐个访问与它们邻接的尚未访问过的全部顶点（与v_1最短距离为3的顶点）
- ✓ ……
- ✓ 依此类推，直到连通图中的所有顶点全部访问完为止。

广度优先遍历算法

① 将所有顶点的*dist*[]、*path*[]值置为-1，对源点*s*置*dist*[*s*]=0，*s*入队；

② 检测队列是否为空，若队列为空，则迭代结束；

③ 从队头取出一个顶点*v*，检测其每个邻接节点*w*：

如果*w*未被访问过，则

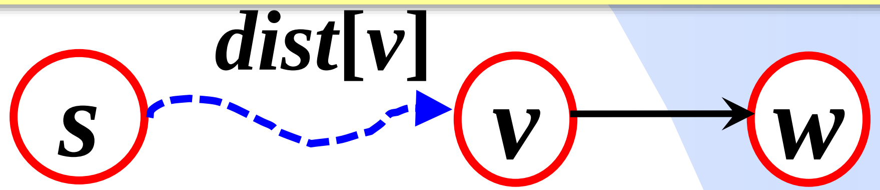
$dist[w] \leftarrow dist[v] + 1.$

$path[w] \leftarrow v.$

将*w*入队；

④ 转至②。

当*dist*[*i*]由-1变成正数（**BFS第一次被访问时**），该值就是源点到 *i* 的最短距离。



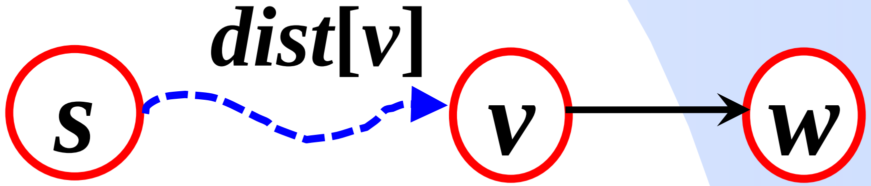
解决最短路径问题，使用两个数组 *dist*[ ]和 *path*[ ].

- *dist*[*i*]:从源点到顶点 *i* 的最短距离，初值-1；
- *path*[*i*]:从源点到顶点 *i* 的最短路径上，顶点 *i* 的前驱顶点，初值-1.

```
void BFS(Vertex* Head,int n,int s,int dist[],int path[]){
   Queue Q;  //创建队列Q，队列需预先实现
   for(int i=1; i<=n; i++) {path[i]=-1; dist[i]=-1};
   dist[s]=0; Q.Enqueue(s);  //源点s入队
   while(!Q.Empty()){
      int v=Q.Dequeue();    //出队一个点
      for(Edge* p=Head[v].adjacent; p!=NULL; p=p->link){
         int w=p->VerAdj;
         if(dist[w]==-1){ //考察v的邻接顶点p
             dist[w]=dist[v]+1; path[w]=v;
             Q.Enqueue(w);
         } //end if
      } //end while
   } //end while
} //end BFS
```

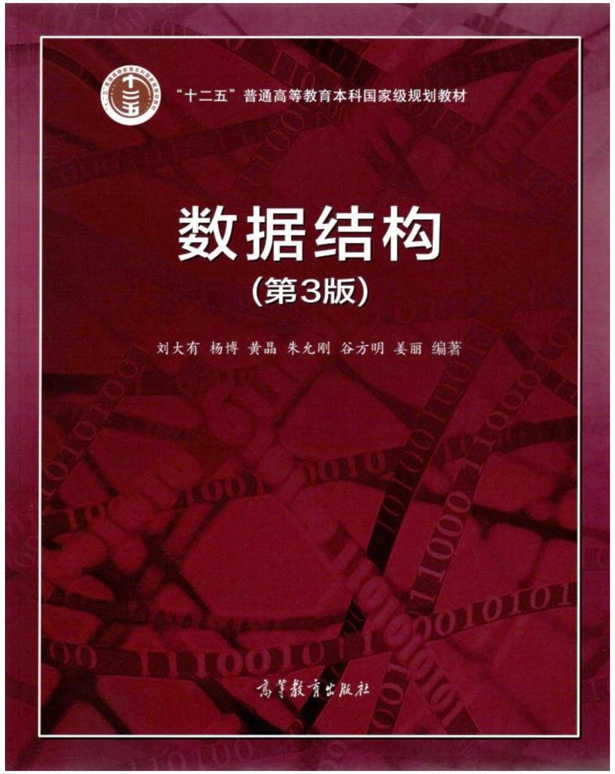
时间复杂度

$O(n+e)$



dist[i]:从源点到顶点 *i* 的最短距离

path[i]:从源点到顶点 *i* 的最短路径上顶点 *i* 的前驱顶点



图的最短路径

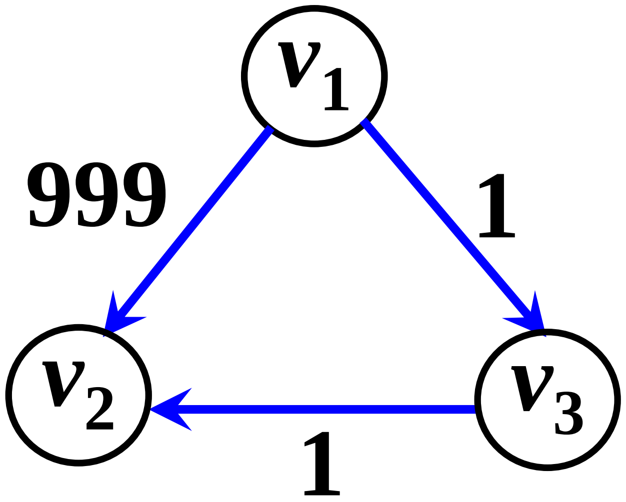
- 无权图最短路径
- **Dijkstra算法**
- A*算法
- Floyd算法
- Bellman-Ford算法
- 满足约束的最短路径

数据之法
结构之美
算法之道

Last updated on 2022.11

zhuyungang@jlu.edu.cn

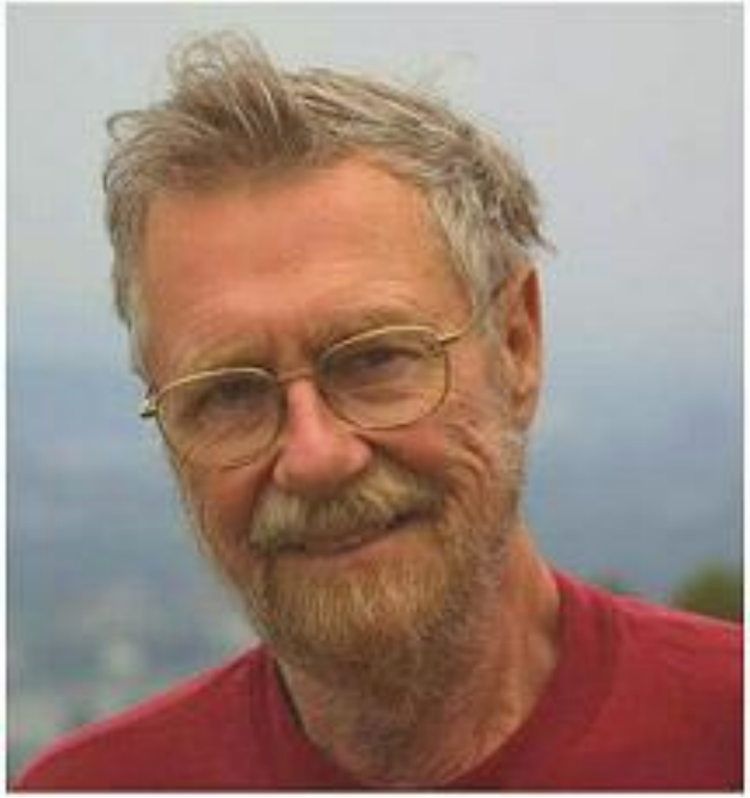
正权图的单源最短路径问题



- 带权图，权值非负。BFS可行么？
- 若直接按BFS：设v_1为源点，v_1 的邻接顶点为 v_2，从而有dist[2] =999. 但是，从v_1到v_2还有一条经由v_3的路径，长度为2。
- 两个顶点间的最短路径不一定就是连接这两个顶点的边，而可能是一条包括多个中间顶点的路径。

问题

无权图的单源最短路径问题 → 正权图的单源最短路径问题

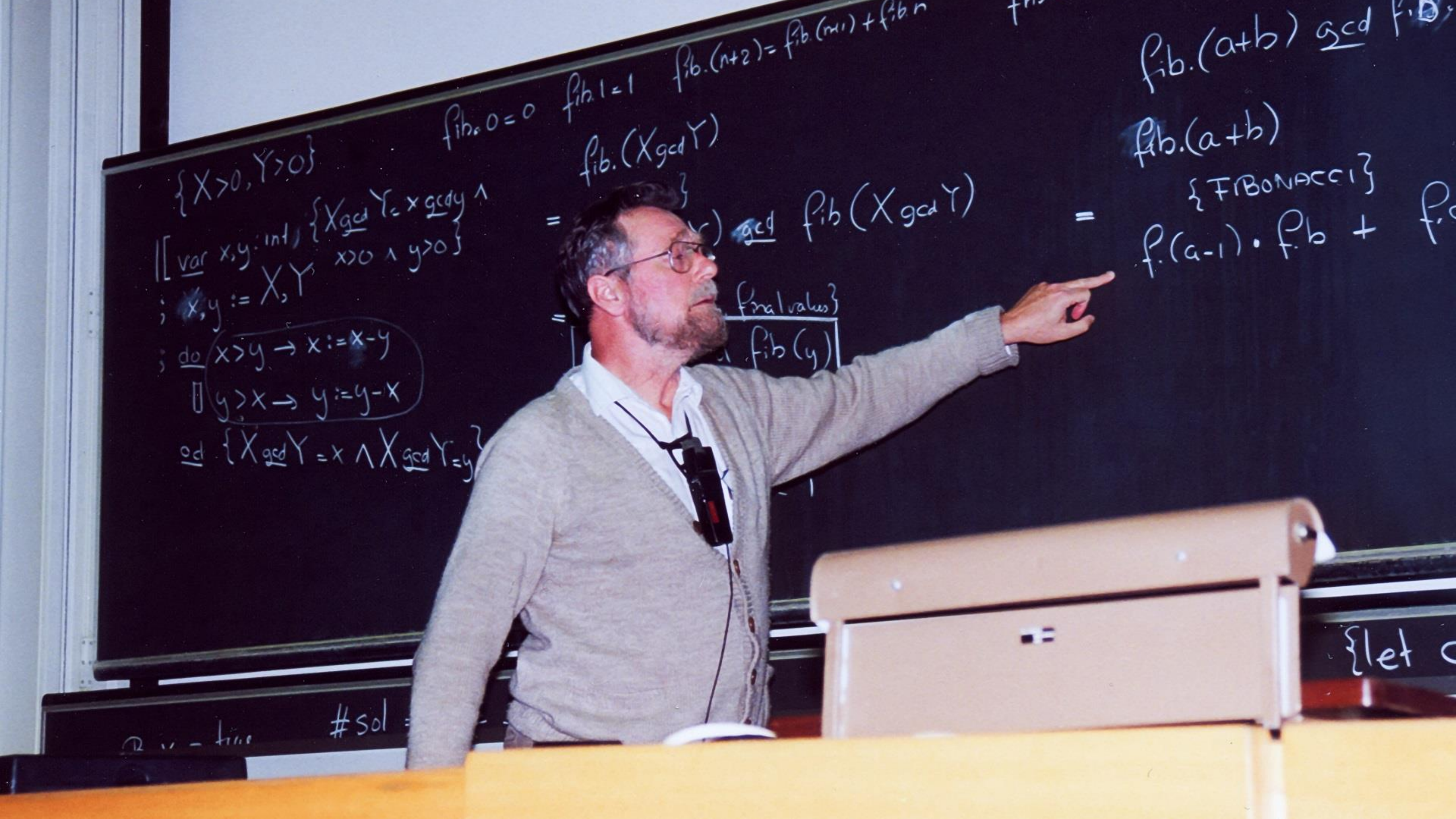


Edsger W. Dijkstra（1930-2002）

德州大学奥斯汀分校教授

图灵奖获得者（首位非英美籍获奖者）

荷兰皇家科学院院士


fib.0 = 0
fib.1 = 1
fib.(n+2) = fib.(n+1) + fib.n
fib.(a+b) gcd
{X>0, Y>0}
fib.(X gcd Y)
fib.(a+b)
{FIBONACCI}
f.(a-1) · f.b +
[[var x,y: int
x>0 ∧ y>0
; x,y := X,Y
; do x>y → x := x-y
[] y>x → y := y-x
od {X gcd Y = x ∧ X gcd Y = y}
final values
fib(y)
{let
#sol

Dijkstra算法

基本思想：将图中所有顶点分成两个集合。

- 集合S包括已确定最短路径的顶点。
- 集合V-S包括尚未确定最短路径的顶点。
- 按照**最短路径长度递增的顺序**逐个把顶点加入集合S。

Dijkstra算法

初始时（s为源点），令D_s=0且 $D_i = \infty$ （$\forall\ i\neq s$）；

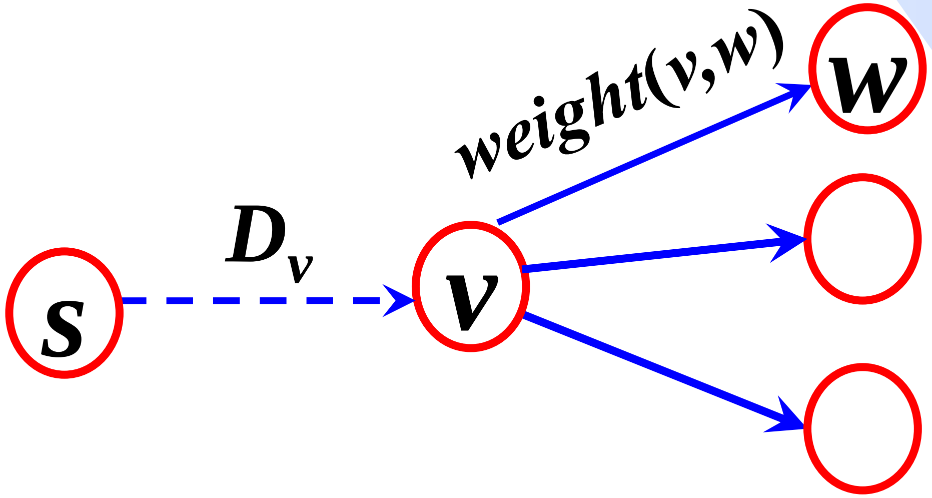
①从不在集合S中的顶点选择D_v最小的顶点v，将其放入集合S；

此时已求出s到v的最短距离，即D_v

②考察v的不在S中的邻接顶点w，若D_v+weight(v,w) < D_w，则更新D_w，使$D_w \leftarrow D_v$+weight(v,w)．【松弛操作】

③重复① ②，直至所有顶点都放入集合S。

D_i表示从源点到顶点 i 的最短距离

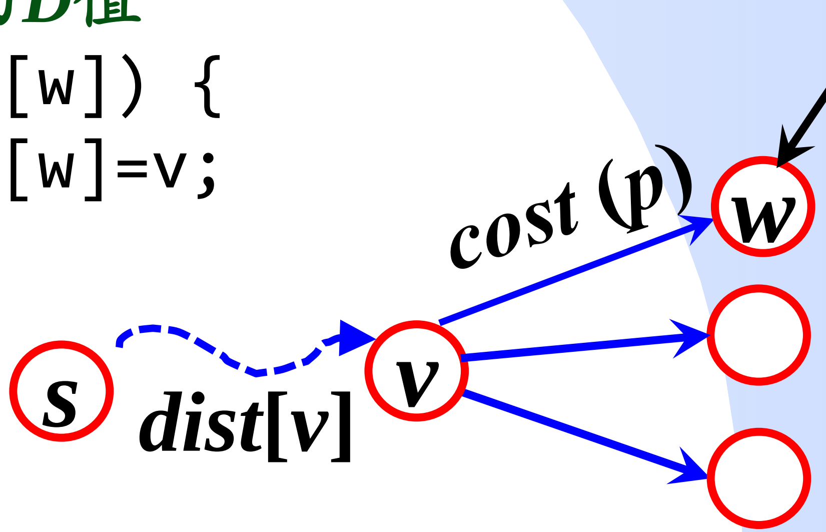


算法实现

- 引入3个辅助数组 *dist*[]、*path*[]、*S*[].
- ✓ *dist*[*i*]：从源点 *s* 到顶点 *i* 的最短距离，初始时 *dist*[*s*]=0, *dist*[*i*]=+∞（∀ *i* ≠ *s*）.
- ✓ *path*[*i*]：*s*到*i*最短路径上*i*的前驱顶点编号，初始时*path*[*i*]= -1 .
- ✓ *S*[*i*]：顶点*i*最是否在集合S中，初始时*S*[*i*]= 0.

- C/C++如何表示 +∞
- ✓ INT_MAX：2^{31}-1，需包含limits.h头文件，做加法运算或翻倍会溢出。
- ✓ 0x3f3f3f3f：1061109567，加上一个数或翻倍都不会溢出，将数组a中每个元素都初始化为无穷大：memset(a,0x3f,sizeof(a))

```
const int INF=0X3f3f3f3f, N=1e5+10;
void Dijkstra(Vertex *Head,int n,int s,int dist[],int path[]){
   int S[N], i, j, min, v, w;
   for(i=1; i<=n; i++) {path[i]=-1; dist[i]=INF; S[i]=0;}//初始化
   dist[s]=0;
   for(i=1; i<=n; i++) {
     min=INF;                  //从不在S集合中的顶点中选D值最小的顶点v
     for(j=1;j<=n;j++) if(S[j]==0 && dist[j]<min){min=dist[j];v=j;}
     S[v]=1;                   //将顶点v放入S集合
     for(Edge* p=Head[v].adjacent; p!=NULL; p=p->link) {
         w=p->VerAdj;          //更新v的邻接顶点的D值
         if(S[w]==0 && dist[v]+p->cost<dist[w]) {
             dist[w]=dist[v]+p->cost;  path[w]=v;
         }
      }
   }
}
```



Dijkstra算法时间复杂度（邻接表存图）

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

其中d_i为各顶点的邻接顶点数，n为顶点数，e为边数。

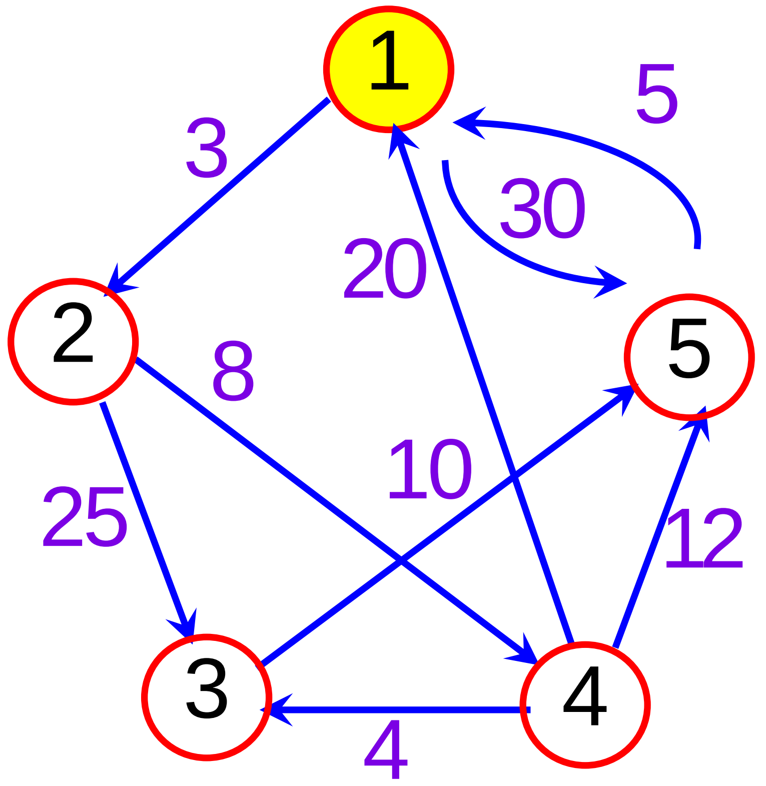
课下思考：若用邻接矩阵存图，时间复杂度是多少？

初始时（s为源点），令D_s=0且 $D_i = \infty$（$\forall\ i \neq s$）；

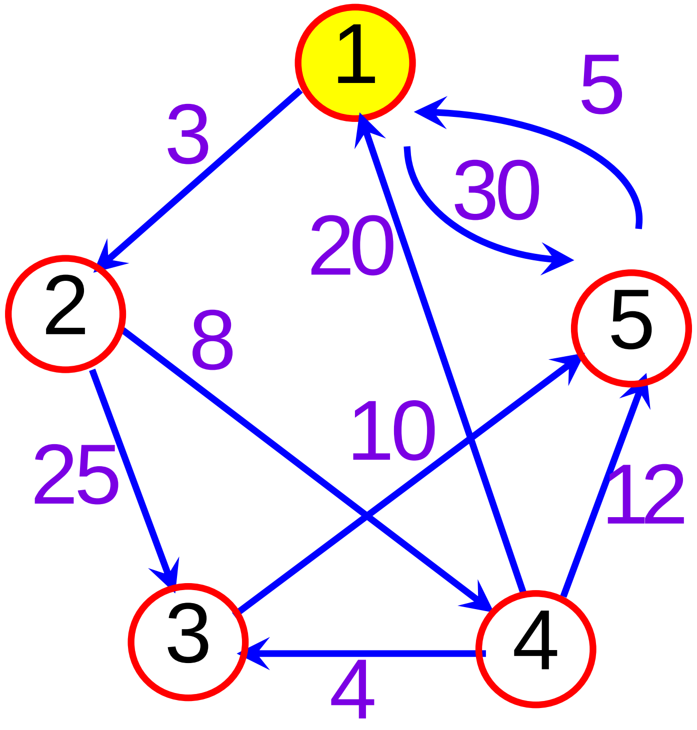
①从不在集合S中的顶点选择D_v最小的顶点v，将其放入集合S；

②考察v的不在S中的邻接顶点w，若D_v+weight(v,w) ＜ D_w，则更新D_w，使$D_w \leftarrow D_v$+weight(v,w)．

③重复① ②，直至所有顶点都放入集合S。



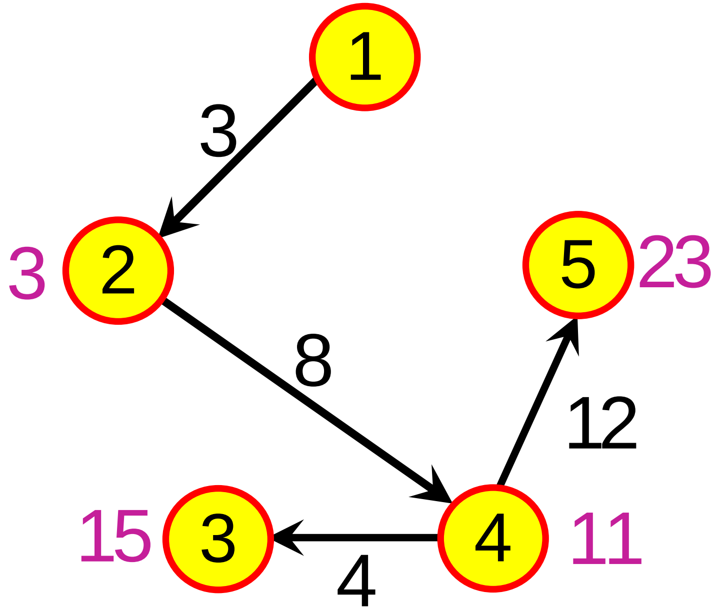
	1	2	3	4	5
S	1	1	1	1	1
dist	0	3	15	11	23
path		V_1	V_4	V_2	V_4

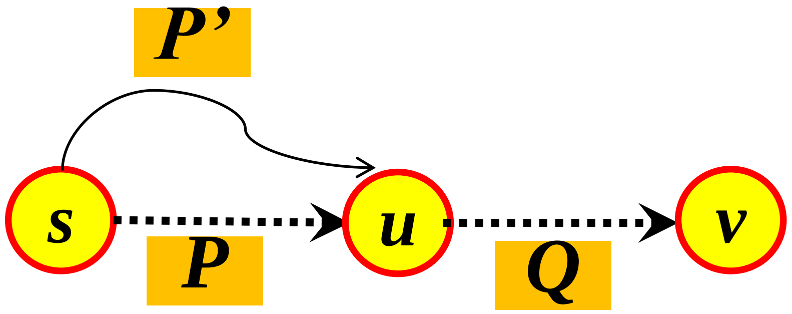


	1	2	3	4	5
S	1	1	1	1	1
dist	0	3	15	11	23
path		v_1	v_4	v_2	v_4

源点到V_5的最短路　v_1 v_2 v_4 v_5

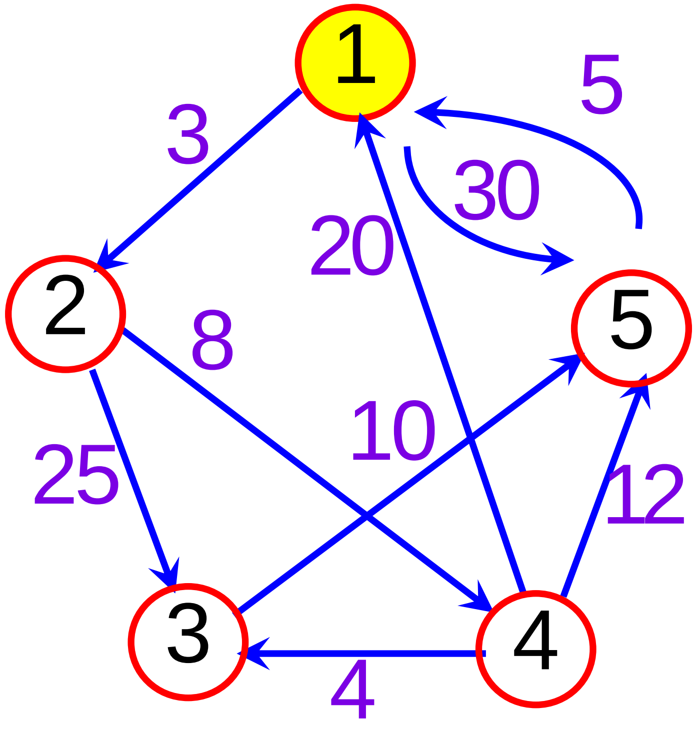
path数组优势：可存源点到其他所有点的最短路





原理：任意最短路的前缀，也是一条最短路

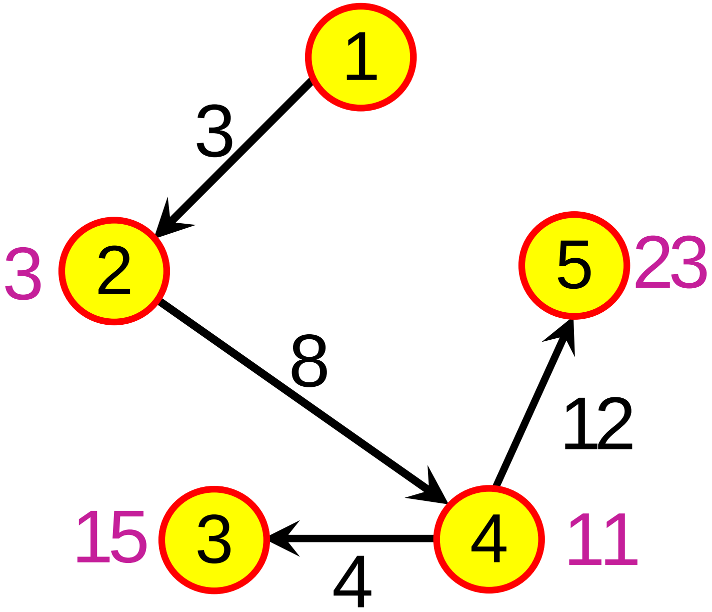
反证法：若s到u有一条更短的路径P'



	1	2	3	4	5
S	1	1	1	1	1
dist	0	3	15	11	23
path		v_1	v_4	v_2	v_4

源点到V_5的最短路 v_1 v_2 v_4 v_5

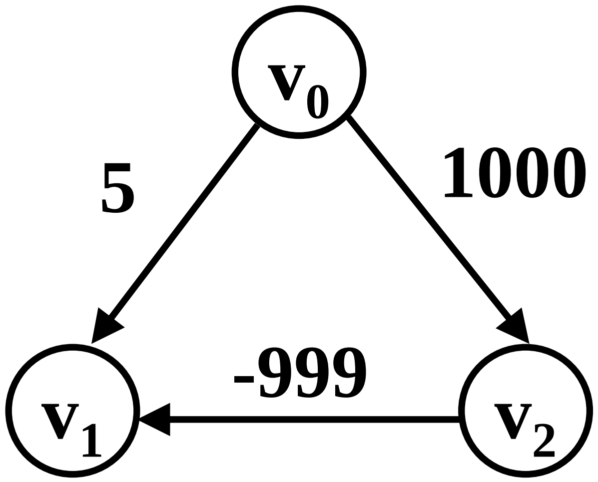
path数组优势：可存源点到其他所有点的最短路

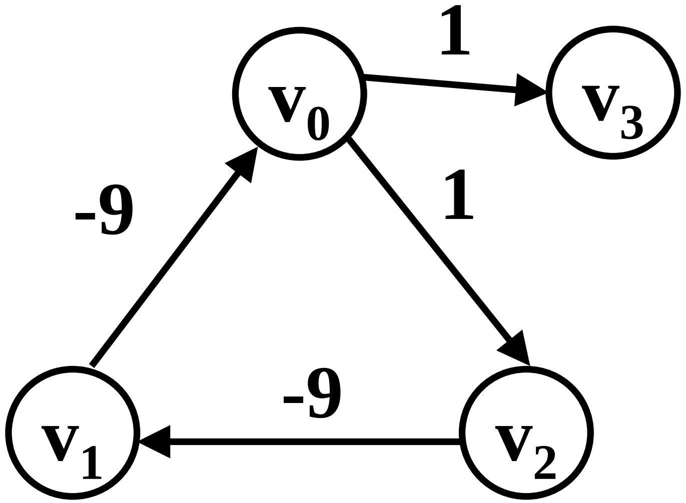


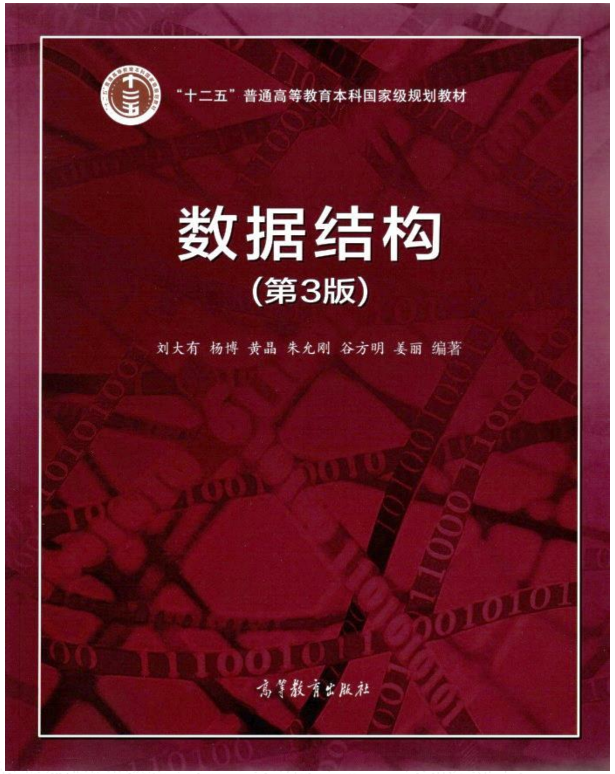

```
void PrintPath(int path[], int s, int t){
   //输出s到t的最短路
   if(s==t) {printf(“%d ”,s); return;}
   PrintPath(s,path[t]);//输出s到t前驱的最短路
   printf(“%d ”,t);      //输出t
}
```

Dijkstra算法总结

- ➢适用于有向图，也适用于无向图。
- ➢算法运行过程中，一旦顶点*v*添加入集合*S*，源点到*v*的最短路径既已确定。
- ➢按照最短距离递增顺序依次把顶点加入集合*S*。
- ➢能处理非负权图，不能处理负权图。
- ➢也可以求源点*s*到某一个顶点*t*的最短路。
- ➢用贪心策略的“形”，包裹了动态规划的“魂”。







图的最短路径

- 无权图最短路径
- Dijkstra算法
- **A*算法**
- Floyd算法
- Bellman-Ford算法
- 满足约束的最短路径

数据之法
结构之美
算法之道

Last updated on 2022.11

zhuyungang@jlu.edu.cn

游戏中的自动寻路

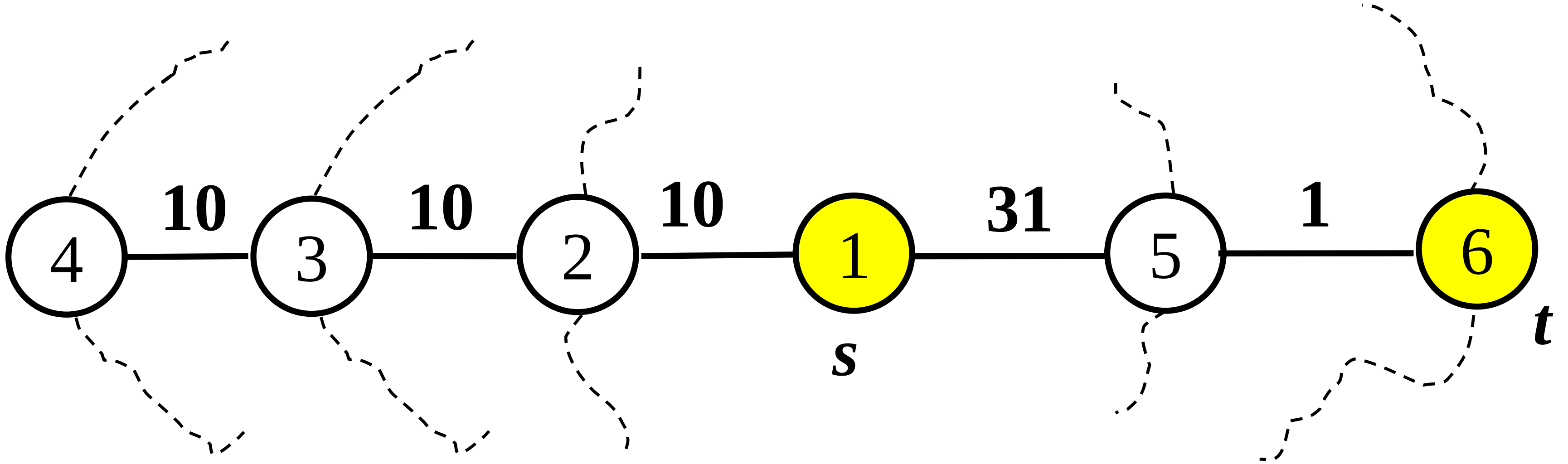
起点s到目标点t的最短路径是该问题的最优解。

问题

无权图的单源最短路径问题 → 正权图的单源最短路径问题 → 正权图单源单点最短路径问题

例：找顶点1到顶点6的最短路

- 按照Dijkstra算法，则搜索顺序2、3、4……
- 原因：Dijkstra只考虑顶点与起点的距离，未考虑顶点与目标点的距离，搜索过程中略显盲目。
- 如何避免跑偏？把顶点到目标点的距离也考虑进去。



A*算法

$$f(v) = g(v) + h(v)$$

估价函数　　启发函数

- $g(v)$：顶点v与起点的最短距离，即dist[v]。
- $h(v)$：启发函数，顶点v与目标点距离的估计值。
- 算法运行过程中，每次选f值最小的顶点v
- A*算法属于启发式搜索算法，相当于对Dijkstra算法的改造。

A*算法

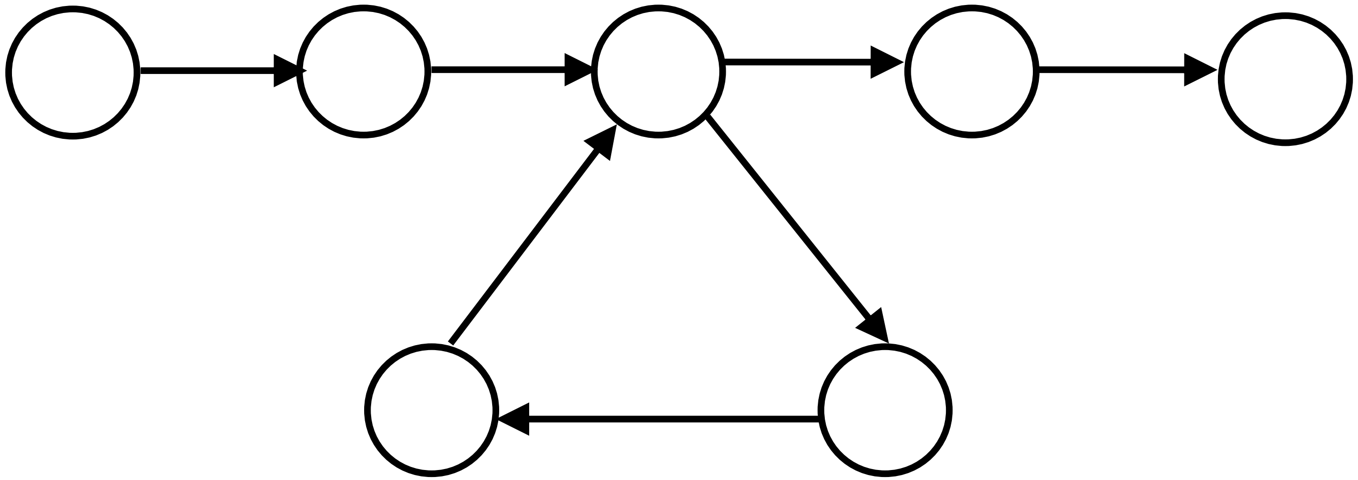
$$f(v) = g(v) + h(v)$$

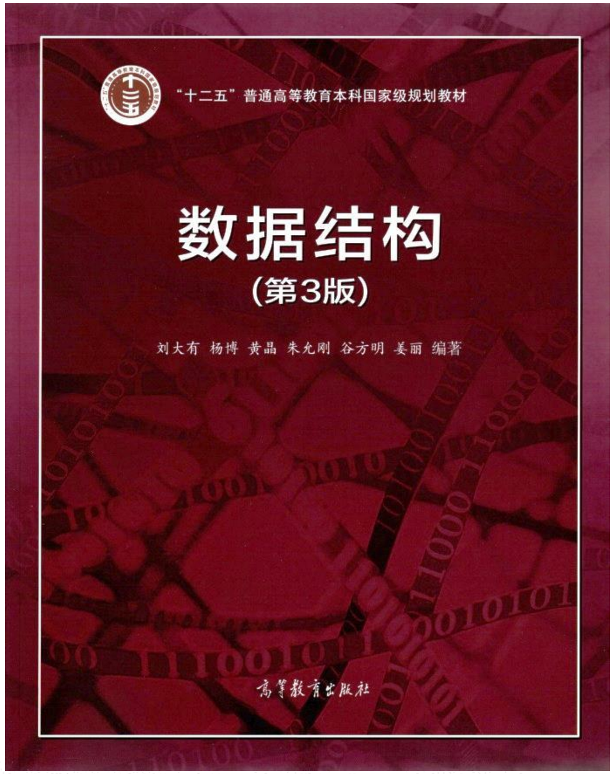
估价函数　　　　启发函数

- 若$h(v) \le h^*(v)$，则可获得最优解，否则无法确保获得最优解，只能获得近似解。（$h^*(v)$为v到目标点的实际最优距离）
- 实际面对的往往是非常大的地图和海量的寻路请求，对算法时间效率要求高。
- 并不一定非要求最优解（最短路），在结果质量和时间效率做一个折中，选择次优解，在实际开发中的应用更加广泛。

最短路径中是否含有环

- 负权环？
- 正权环？
- 0权环？
- 不失一般性地，我们可以假定最短路径中没有环，最多包含 n个顶点，n-1条边。





图的最短路径

- 无权图最短路径
- Dijkstra算法
- A*算法
- **Bellman-Ford算法**
- Floyd算法
- 满足约束的最短路径

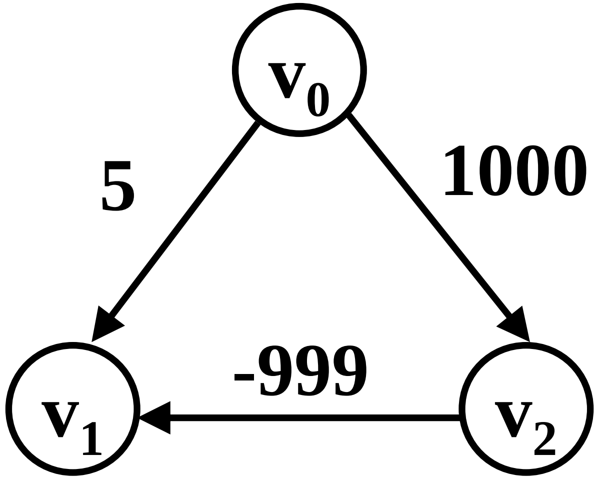
数据结构之美 结构之美 算法之道

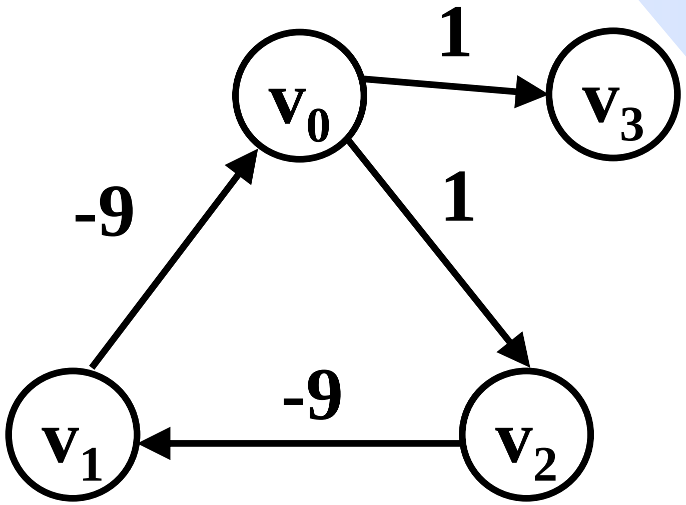
Last updated on 2022.11

zhuyungang@jlu.edu.cn

负权图单源最短路径问题

- Dijkstra算法总结不能处理负权图。选出D值最小的顶点v，加入集合S后，源点到v的最短距离即确定。但在负权图中，v加入集合S后，源点到v的最短距离也不确定。
- 目标：对负权图能求单源最短路径，若图中含负环，亦能识别。





问题

无权图的单源最短路径问题

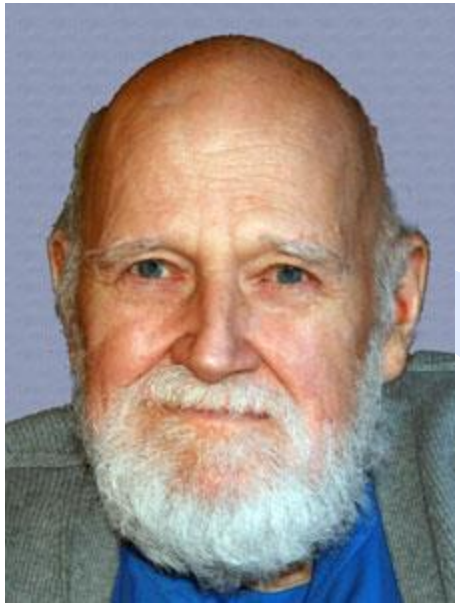
正权图的单源最短路径问题

正权图单源单点最短路径问题

负权图的单源最短路径问题

Bellman-Ford算法

Richard E. Bellman
南加州大学教授
美国科学院院士
美国工程院院士
动态规划之父



Lester R. Ford Jr
美国数学家

Bellman-Ford算法

步骤1：初始化

FOR $i = 1$ **TO** n **DO** $D[i] \leftarrow +\infty$.

$D[s] \leftarrow 0$.

步骤2：迭代求解源点s到各点的最短距离

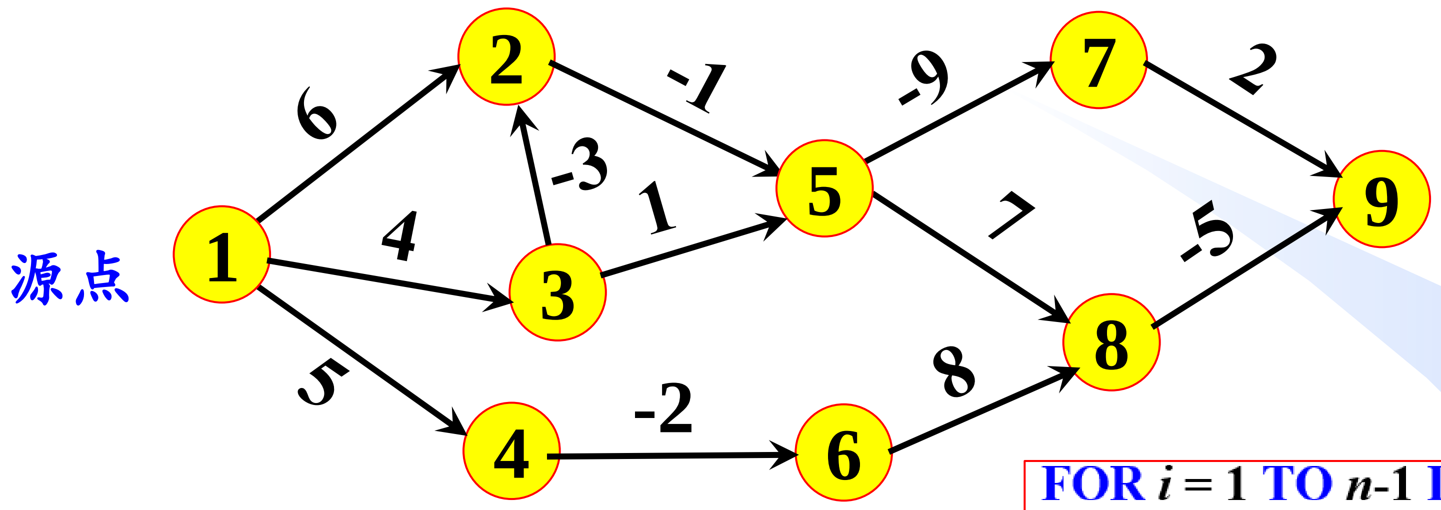
FOR $i = 1$ **TO** n-1 **DO**

  FOR each edge <u, v> DO

    IF $D[u] + weight(u, v) < D[v]$

     THEN $D[v] \leftarrow D[u] + weight(u, v)$.

Bellman-Ford算法的正确性




```
FOR i = 1 TO n-1 DO
    FOR each edge <u, v> DO
        IF D[u]+ weight(u, v) < D[v]
          THEN D[v] ← D[u] + weight(u, v)
```

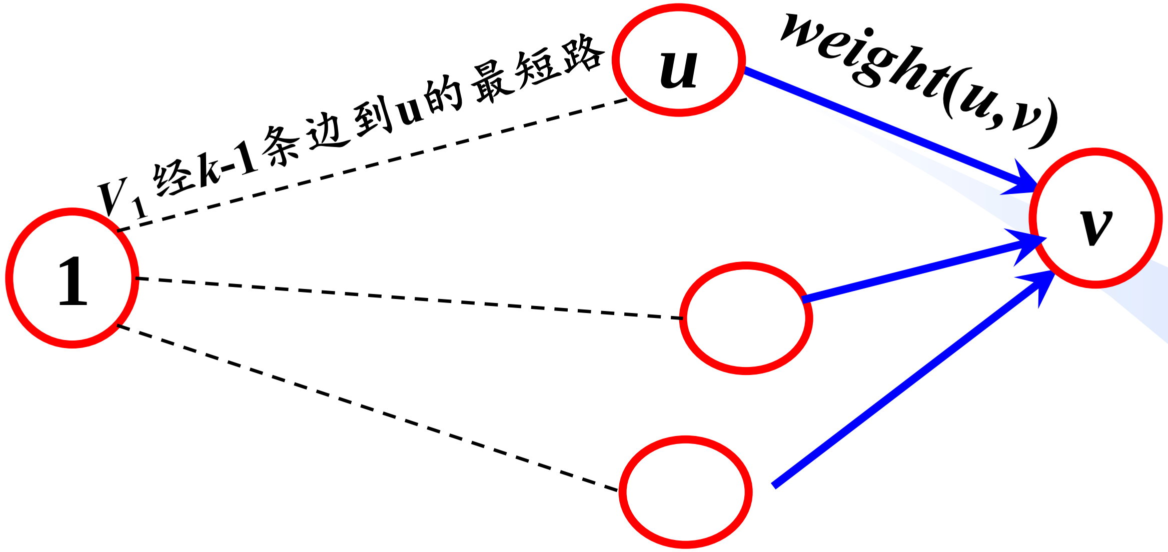
第1次迭代，一定找到V_1到各点的至多包含1条边的最短路径

第2次迭代，一定找到V_1到各点的至多包含2条边的最短路径

第3次迭代，一定找到V_1到各点的至多包含3条边的最短路径

Bellman-Ford算法的正确性

第k次迭代



第k次迭代，一定找到V_1到各点的至多包含k条边的最短距离

```
FOR each edge <u, v> DO
  IF D[u]+ weight(u, v) < D[v]
    THEN D[v] ← D[u] + weight(u, v) .
```

Bellman-Ford算法的正确性

- 第k次迭代，一定找到V_1到各点的至多包含k条边的最短距离。
- 一共需要几次迭代？最短路径最多包含全部n个顶点，n-1条边，即V_1到各点的最短路径至多包含n-1条边，故至多迭代n-1次。
- 如果n-1次迭代后，再做1次迭代，最短路径还有变化，则肯定含有负环。

Bellman-Ford算法

步骤1：初始化

FOR $i = 1$ **TO** n **DO** $D[i] \leftarrow +\infty$.

$D[s] \leftarrow 0$.

步骤2：迭代求解源点s到各点的最短距离

FOR $i = 1$ **TO** $n-1$ **DO**

    FOR each edge <u, v> DO

        IF $D[u] + weight(u, v) < D[v]$

         THEN $D[v] \leftarrow D[u] + weight(u, v)$.

步骤3：检验负权环

FOR each edge <u, v> DO

        IF $D[u] + weight(u, v) < D[v]$

            THEN RETURN FALSE.

RETURN TRUE.

时间复杂度
$O(ne)$

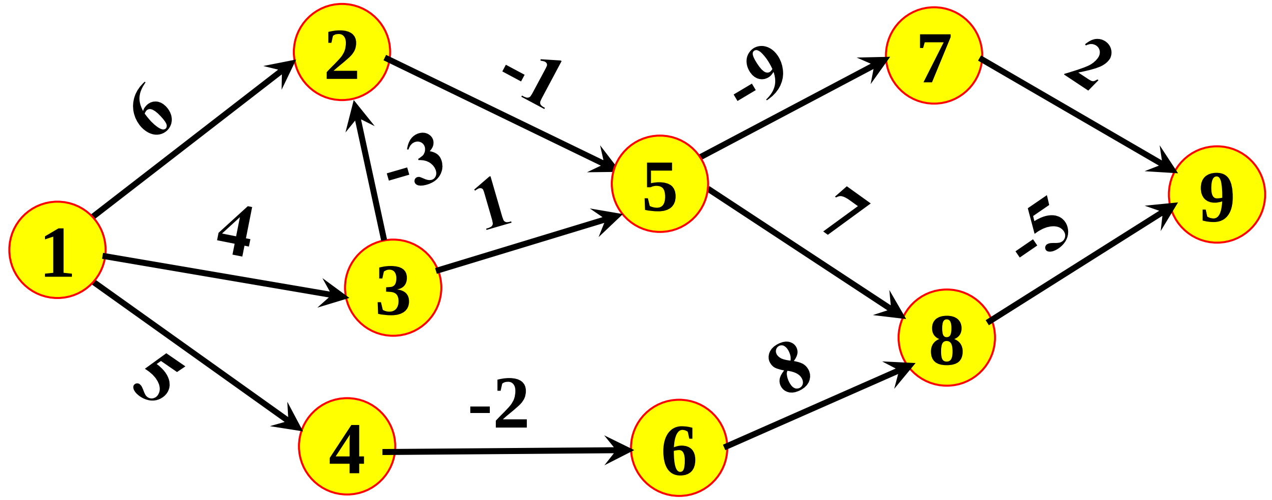
Bellman-Ford算法

```
bool Bellman_Ford(Edge E[], int s, int n, int e, int D[]){
  for(int i=1;i<=n;i++) D[i]=INF; //初始化
  D[s]=0;
  for(int i=1; i<=n-1; i++)        //n-1轮松弛操作
     for(int k=0; k<e; k++) {      //扫描所有边
          int u=E[k].u; int v=E[k].v; int w=E[k].weight;
          if(D[u]+w<D[v]) D[v]=D[u]+w;
     }
  for(int k=0; k<e; k++) {          //检测负环
     int u=E[k].u; int v=E[k].v; int w=E[k].weight;
     if(D[u]+w<D[v]) return false; //有负环
  }
  return true;
}
```

```
struct Edge{
   int u,v;
   int weight;
};
Edge E[1000];
```

队列优化的Bellman-Ford算法

- 本次迭代中*D*值被更新的顶点，其邻接顶点的*D*值可能在下次迭代时更新。
- 更新*D*[*v*]之后，下一步只计算和调整顶点*v*的邻接顶点，可加快收敛速度。



队列优化的Bellman-Ford算法

- 算法采用一个队列。 初始时将源点入队，每次从队列中取出一个顶点，并考察其邻接顶点，若某个顶点*D*值被更新，则将其入队。 直至队列为空时算法结束。
- 也称为SPFA算法（Shortest Path Faster Algorithm）
- 最坏情况下时间复杂度与Bellman-Ford算法相同，为O(ne)。但在稀疏图上运行效率较高，为O(ke)，其中k为一个较小的常数。

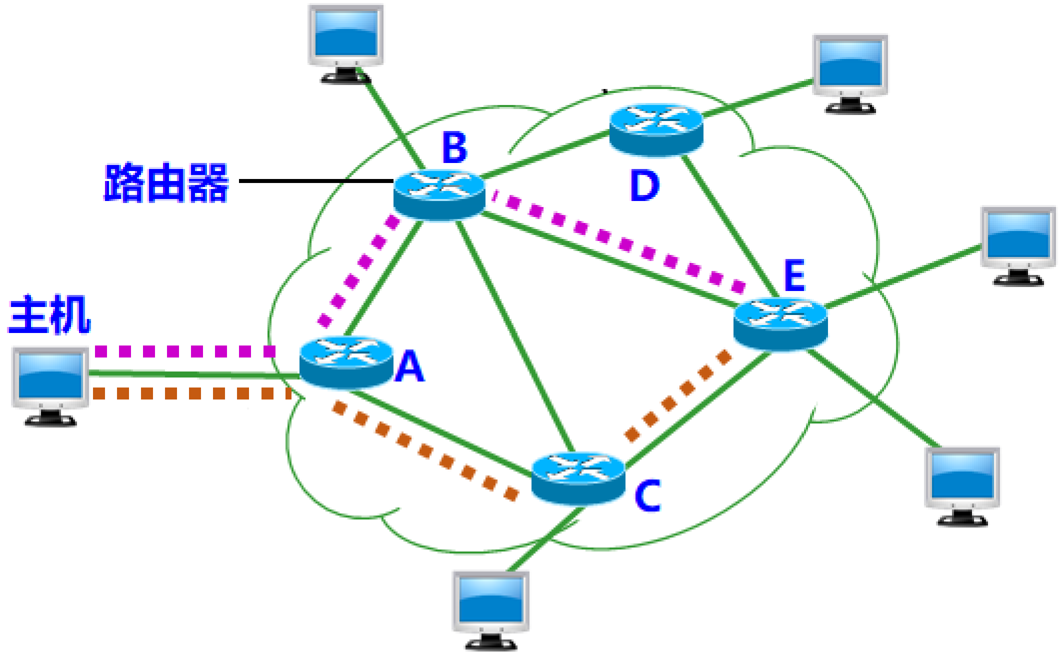
```
bool SPFA(Vertex* Head, int n, int s, int D[]) {
    Queue Q;  int InQueue[N]={0}, times[N]={0}, D[N];
    memset(D, 0x3f, sizeof(D));
    D[s] = 0;  Q.ENQUEUE(s); InQueue[s] = 1; times[s]++;
    while(!Q.Empty()) {
        int u = Q.DEQUEUE();  InQueue[u] = 0;
        for (Edge* p = Head[u].adjacent; p != NULL; p = p->link) {
            int v = p->VerAdj;
            if(D[u] + p->cost < D[v]) {
                D[v] = D[u] + p->cost;
                if(!InQueue[v]) {
                    Q.ENQUEUE(v); InQueue[v] = 1; times[v]++; //入队次数+1
                    if(times[v] >= n) return false; //入队次数>=n则存在负环
                }
            }
        }
    }
    return true;
}
```

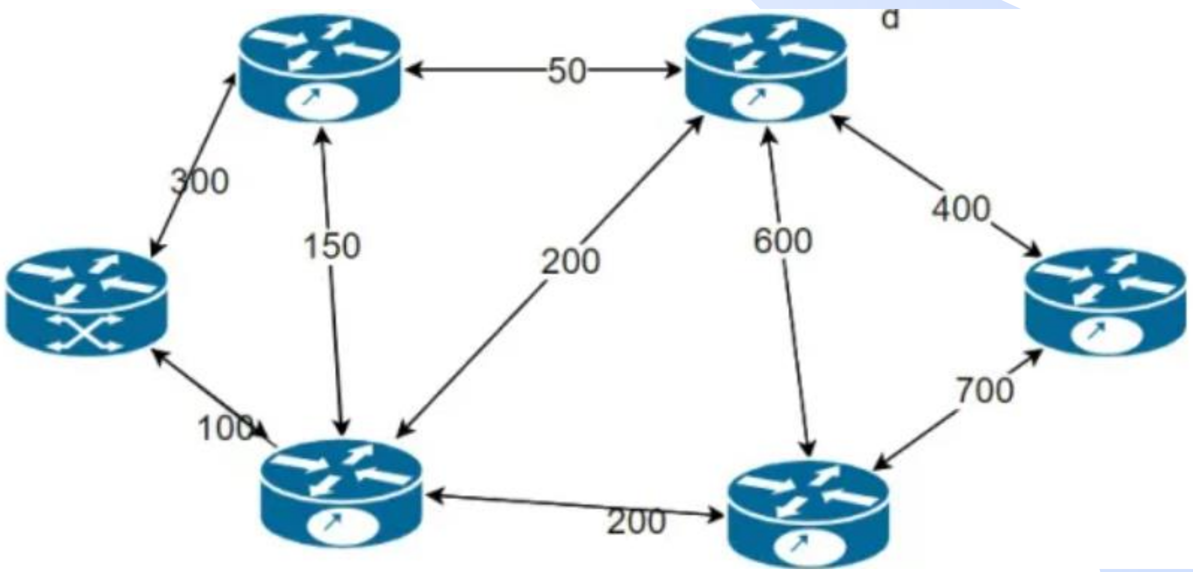
SPFA算法

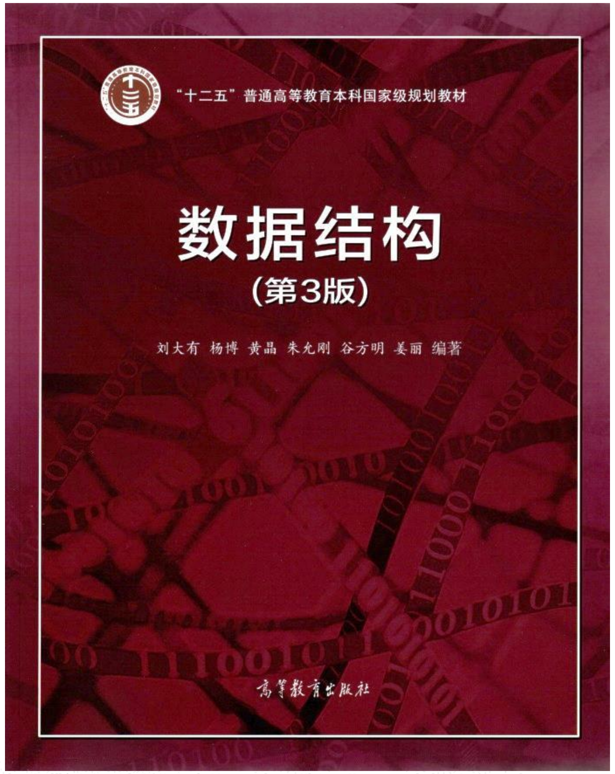
- ✓ 顶点可以多次进出队
- ✓ 顶点每次入队*D*值更新1次
- ✓ 若入队次数大于等于*n*次，则存在负环

计算机网络的路由算法

- OSPF路由协议：基于Dijkstra算法
- RIP、BGP路由协议：基于Bellman-Ford算法







图的最短路径

- 无权图最短路径
- Dijkstra算法
- A*算法
- Bellman-Ford算法
- **Floyd算法**
- 满足约束的最短路径

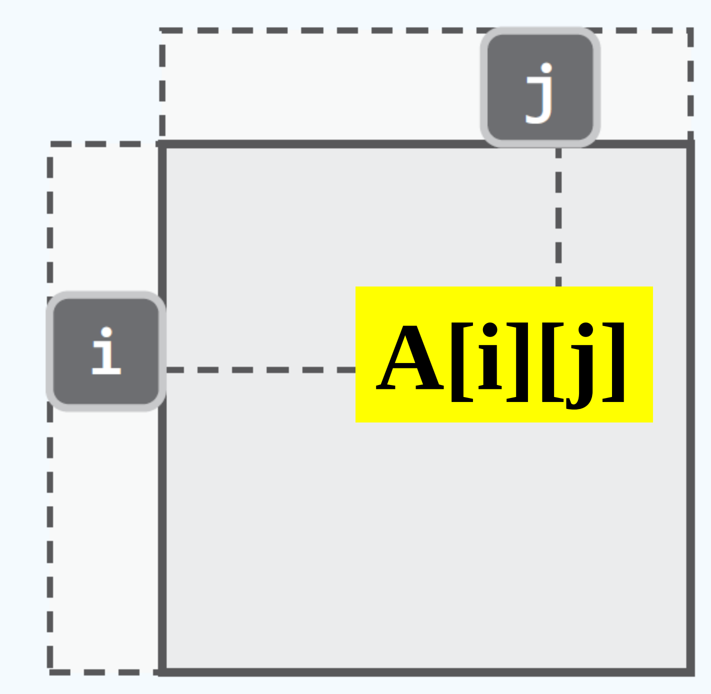
数据之法
结构之美
算法之道

Last updated on 2022.11

zhuyungang@jlu.edu.cn

每对顶点间的最短路径

- 问题：G是带权图，对每一对顶点 $v_i \neq v_j$，求 v_i 与 v_j 间的最短路径。
- 方案一：若权值为正，可依次把每个顶点作为源点，执行Dijkstra算法。
- 方案二：Floyd算法



问题

无权图的单源最短路径问题

正权图的单源最短路径问题

正权图单源单点最短路径问题

负权图的单源最短路径问题

任意两点间的最短路径问题

Floyd算法

Robert W. Floyd（弗洛伊德）

斯坦福大学教授

图灵奖获得者

文学转专业计算机

成功逆袭终获图灵奖

Floyd算法

对于图中所有顶点$v_1, v_2, \ldots, v_n$，考察如下集合：

$I_1=\{v_1\}$

$I_2=\{v_1, v_2\}$

$I_3=\{v_1, v_2, v_3\}$

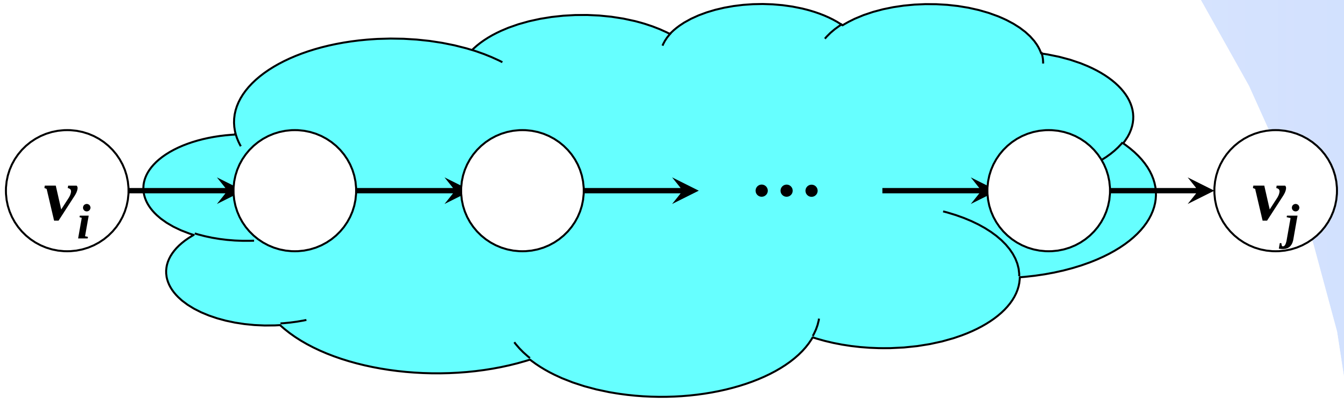
$I_4=\{v_1, v_2, v_3, v_4\}$

….

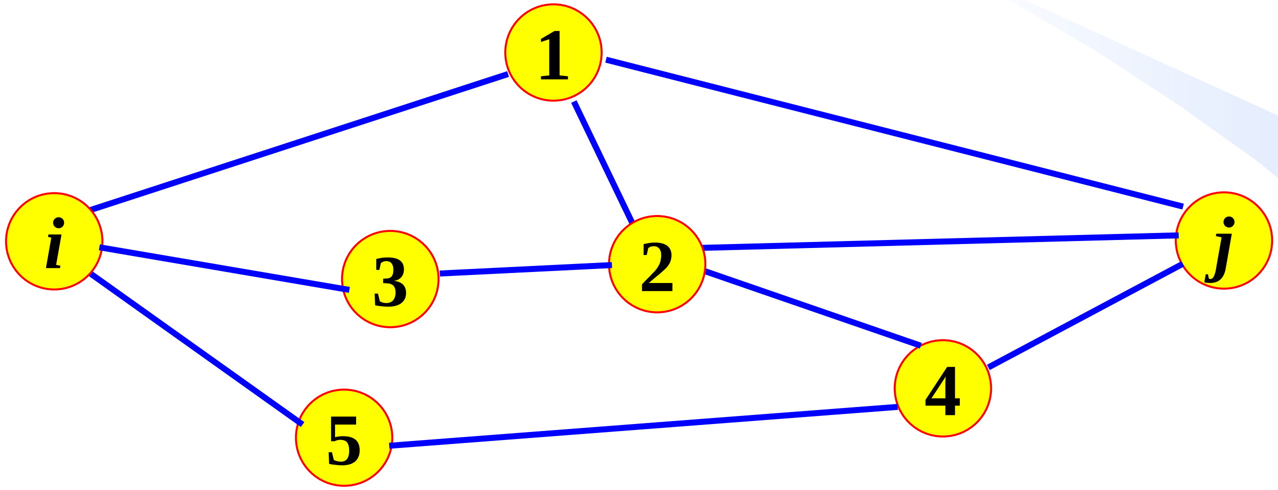
$I_n=\{v_1, v_2, \ldots, v_n\}$

可以把v_i到v_j的最短路径分为3部分

v_i　中间顶点　v_j

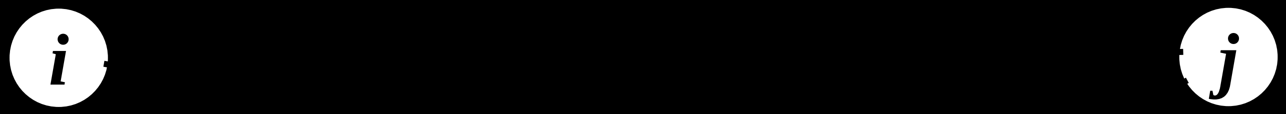


Floyd算法



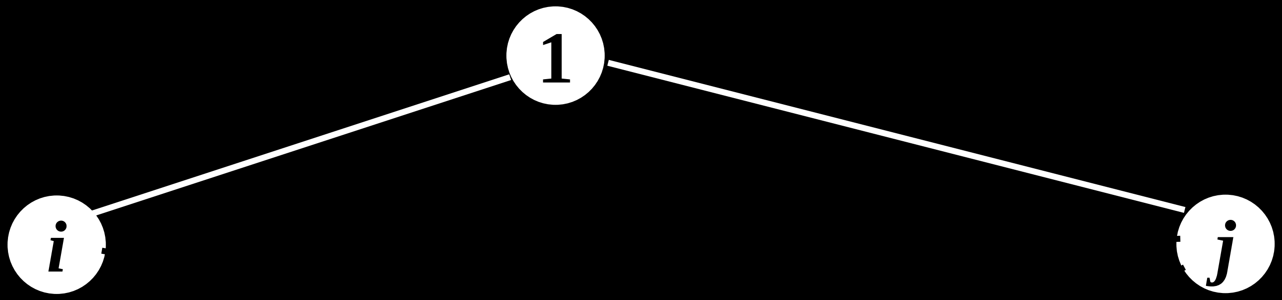
用邻接矩阵存储图

Floyd算法



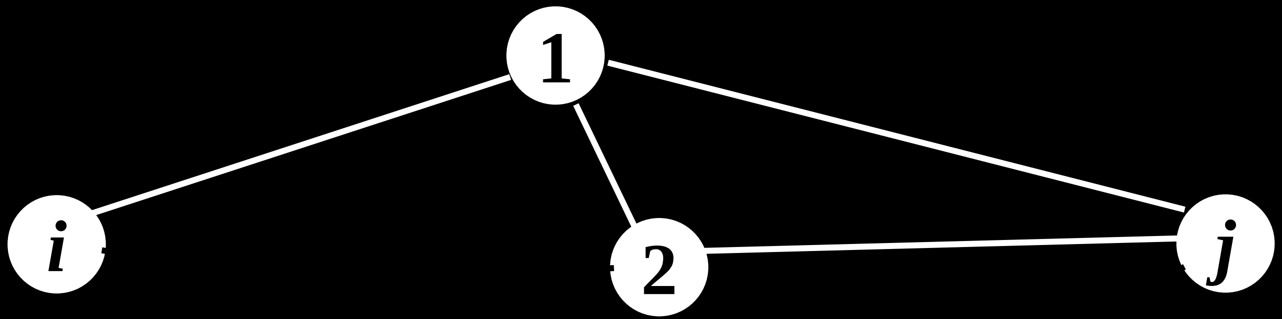
初始矩阵: $D^{(0)}[i][j]=A[i][j]$

Floyd算法



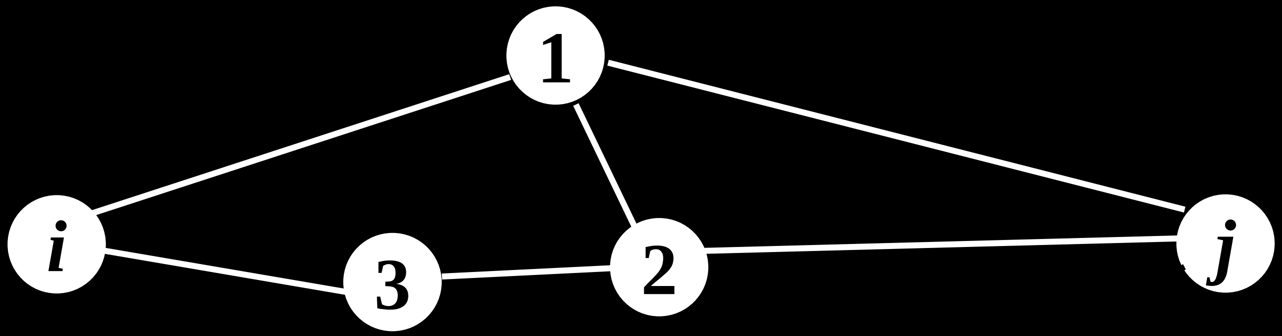
求v_i经由$\{v_1\}$中顶点到达v_j的最短路径，最短距离存入矩阵$D^{(1)}$

Floyd算法



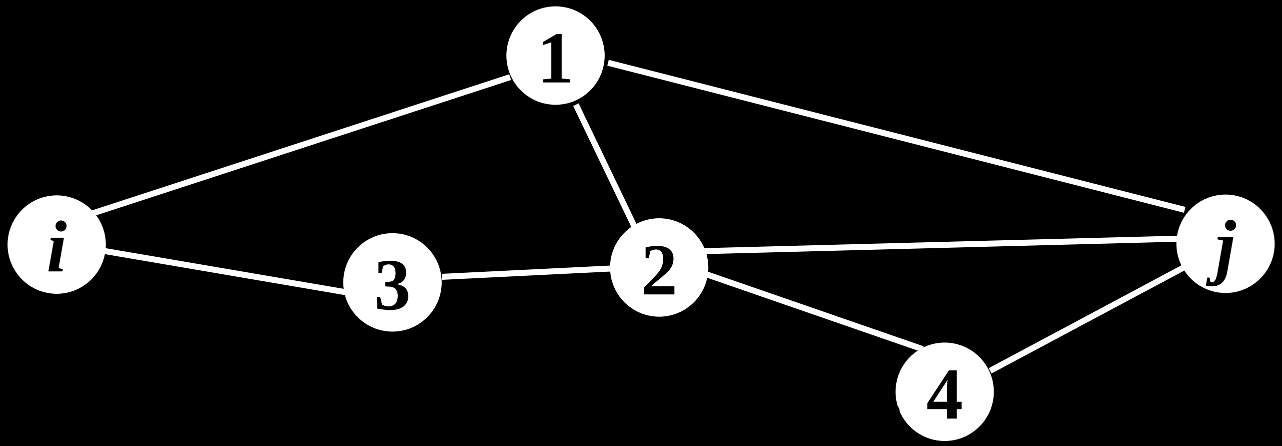
求v_i经由$\{v_1, v_2\}$中顶点到达v_j的最短路径，最短距离存入矩阵$D^{(2)}$

Floyd算法



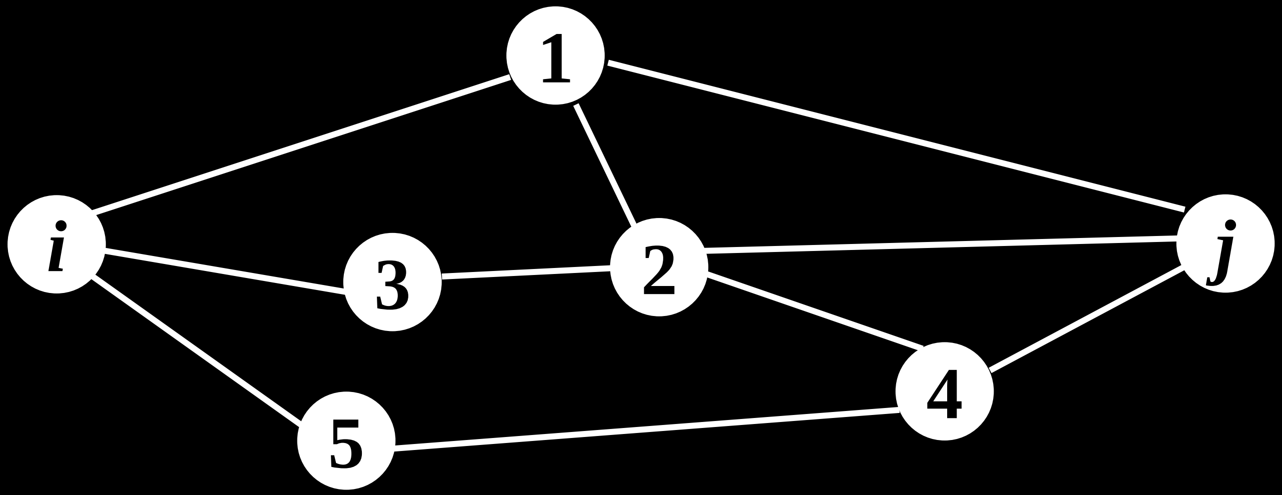
求v_i经由$\{v_1, v_2, v_3\}$中顶点到达v_j的最短路径，最短距离存入矩阵$D^{(3)}$

Floyd算法



求v_i经由$\{v_1, v_2, v_3, v_4\}$中顶点到达 v_j的最短路径，最短距离存入矩阵$D^{(4)}$

Floyd算法



求v_i经由$\{v_1, v_2, v_3, v_4, v_5\}$中顶点到达$v_j$的最短路径，最短距离存入矩阵$D^{(5)}$

用邻接矩阵存储图，定义初始矩阵：$D^{(0)}[i][j]=A[i][j]$

1. $D^{(1)}$ 是从 v_i 经由 $\{v_1\}$ 中顶点到达 v_j 的最短路径长度；

$$v_i \longrightarrow v_1 \longrightarrow v_j$$

2. $D^{(2)}$ 是从 v_i 经由 $\{v_1, v_2\}$ 中顶点到达 v_j 的最短路径长度；

…

$$v_i \longrightarrow v_1, v_2 \longrightarrow v_j$$

$D^{(k)}[i][j]$:v_i 至 v_j 在包含前 k 个顶点的子图上的最短路径的长度

k. $D^{(k)}$ 是从 v_i 经由 $\{v_1, v_2, \dots, v_k\}$ 中顶点到达 v_j 的最短路径长度；

…

$$v_i \longrightarrow v_1, v_2, \dots, v_k \longrightarrow v_j$$

$D^{(k)}[i][j]$: v_i 至 v_j 的中间顶点序号小于等于 k 的最短路径的长度

n. $D^{(n)}$ 是从 v_i 经由 $\{v_1, v_2, \dots, v_n\}$ 中顶点到达 v_j 的最短路径长度；

$$v_i \longrightarrow v_1, \dots, v_n \longrightarrow v_j$$

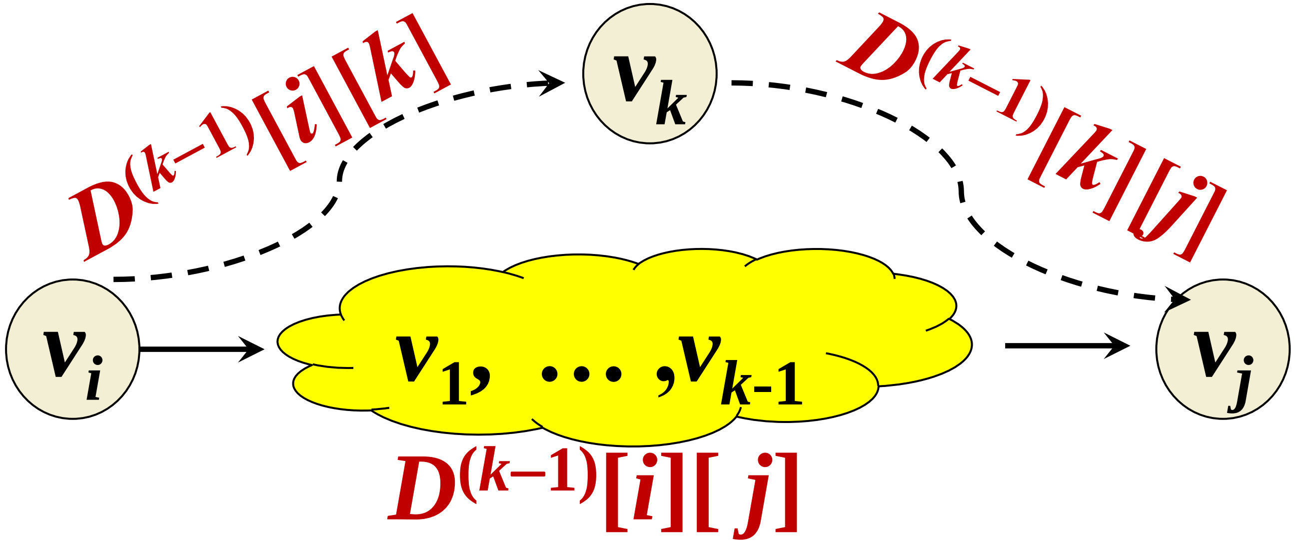
则 $D^{(n)}$ 即为图中从 v_i 到 v_j 的最短路径长度。

递推方法

用邻接矩阵存储图，定义初始矩阵: $D^{(0)}[i][j]=A[i][j]$

k-1. $D^{(k-1)}[i][j]$是从 v_i 经由$\{v_1, v_2, \dots, v_{k-1}\}$中顶点至$v_j$的最短路径长度； $v_i \dots \{v_1, v_2, \dots, v_{k-1}\} \dots v_j$

k. $D^{(k)}[i][j]$ 是从 v_i经由$\{v_1, v_2, \dots, v_{k-1}, v_k\}$中顶点至$v_j$的最短路径长度； $v_i \dots \{v_1, v_2, \dots, v_{k-1}, v_k\} \dots v_j$



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

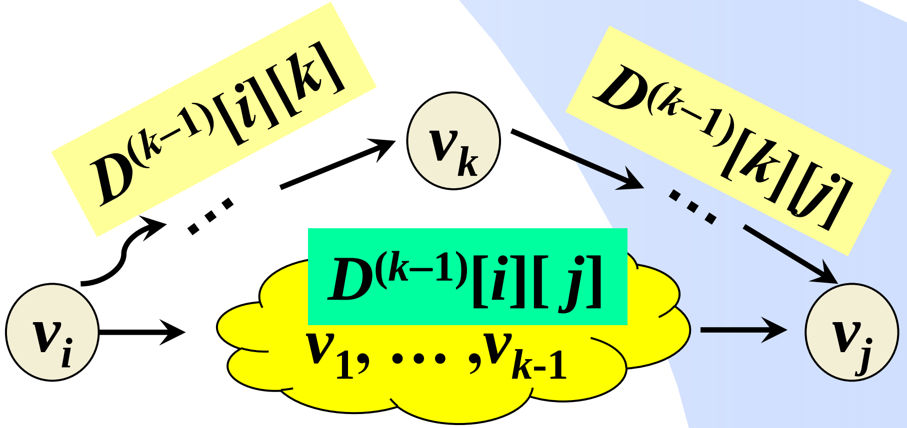
算法实现

- ➢ 引入3个辅助数组 *D*[][]、*path*[][]、*A*[][].
- ✓ *D*[*i*][*j*]：从顶点*i*到顶点*j*的最短距离，初值等于邻接矩阵.
- ✓ *path*[*i*][*j*]：*i*到*j*最短路径上*i*的下一个顶点编号.
- ✓ *A*[ ][ ]：邻接矩阵.

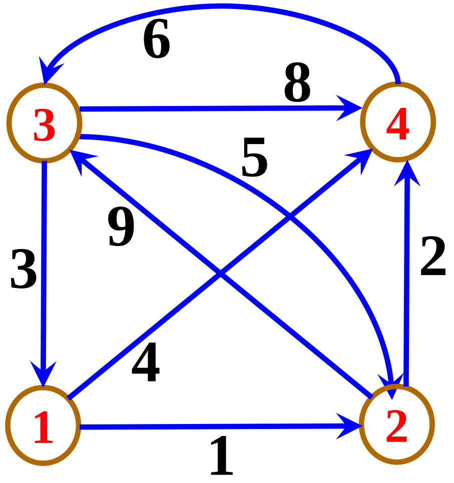
Floyd算法

```c
void Floyd(int A[N][N],int n,int D[N][N],int path[N][N]){
  for(int i=1; i<=n; i++)    //初始化
    for(int j=1; j<=n; j++) {
        D[i][j]=A[i][j];
        if(i!=j && A[i][j]<INF) path[i][j]=j; //i和j间有边
        else path[i][j]=-1;
    }
  for(int k=1; k<=n; k++)     //递推构造D^n
    for(int i=1; i<=n; i++)
      for(int j=1; j<=n; j++)
        if(D[i][k]+D[k][j]<D[i][j]){
            D[i][j]=D[i][k]+D[k][j];
            path[i][j]=path[i][k];
        }
}
```

$O(n^3)$



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



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & \infty & 4 \\ 2 & \infty & 0 & 9 & 2 \\ 3 & 3 & 5 & 0 & 8 \\ 4 & \infty & \infty & 6 & 0 \end{array}$$

从$D^{(4)}$知，顶点 2 到 1 的最短距离为 $D[2][1]=11$，其最短路径：

path[2][1]=4，表示顶点2→顶点4;

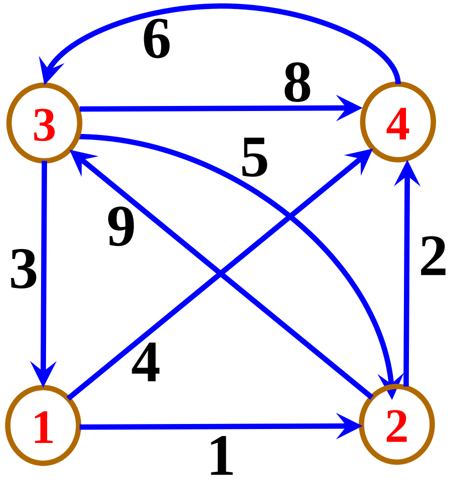
path[4][1]=3，表示顶点4→顶点3;

path[3][1]=1，表示顶点3→顶点1;

综上：2→ 4→ 3→ 1

	$D^{(0)}$				$D^{(1)}$				$D^{(2)}$				$D^{(3)}$				$D^{(4)}$			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
2	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
3	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
4	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0

	path$^{(0)}$				*path*$^{(1)}$				*path*$^{(2)}$				*path*$^{(3)}$				*path*$^{(4)}$			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	-1	2	-1	4	-1	2	-1	4	-1	2	2	2	-1	2	2	2	-1	2	2	2
2	-1	-1	3	4	-1	-1	3	4	-1	-1	3	4	3	-1	3	4	4	-1	4	4
3	1	2	-1	4	1	1	-1	1	1	1	-1	1	1	1	-1	1	1	1	-1	1
4	-1	-1	3	-1	-1	-1	3	-1	-1	-1	3	-1	3	3	3	-1	3	3	3	-1

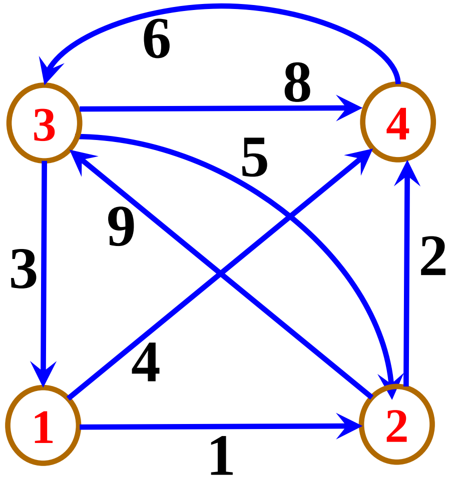


$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & \infty & 4 \\
2 & \infty & 0 & 9 & 2 \\
3 & 3 & 5 & 0 & 8 \\
4 & \infty & \infty & 6 & 0
\end{array}
$$

```c
void PrintPath(int s, int t) {
    printf("%d ", s);
    int k = s;
    while (k != t) {
        printf("%d ", path[k][t]);
        k = path[k][t];
    }
}
```

	$D^{(0)}$				$D^{(1)}$				$D^{(2)}$				$D^{(3)}$				$D^{(4)}$			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
2	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
3	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
4	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0

	$path^{(0)}$				$path^{(1)}$				$path^{(2)}$				$path^{(3)}$				$path^{(4)}$			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	-1	2	-1	4	-1	2	-1	4	-1	2	2	2	-1	2	2	2	-1	2	2	2
2	-1	-1	3	4	-1	-1	3	4	-1	-1	3	4	3	-1	3	4	4	-1	4	4
3	1	2	-1	4	1	1	-1	1	1	1	-1	1	1	1	-1	1	1	1	-1	1
4	-1	-1	3	-1	-1	-1	3	-1	-1	-1	3	-1	3	3	3	-1	3	3	3	-1



$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & \infty & 4 \\
2 & \infty & 0 & 9 & 2 \\
3 & 3 & 5 & 0 & 8 \\
4 & \infty & \infty & 6 & 0
\end{array}
$$

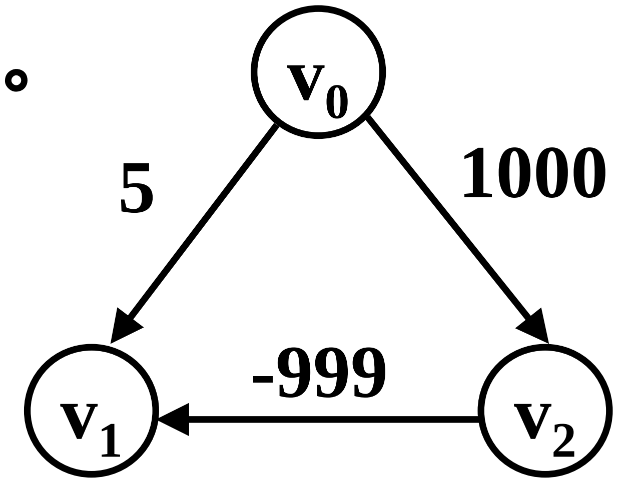
```
void PrintPath1(int s, int t){
    printf("%d ", s);
    for(int k=s; k!=t; k=path[k][t])
        printf("%d ", path[k][t]);
}
```

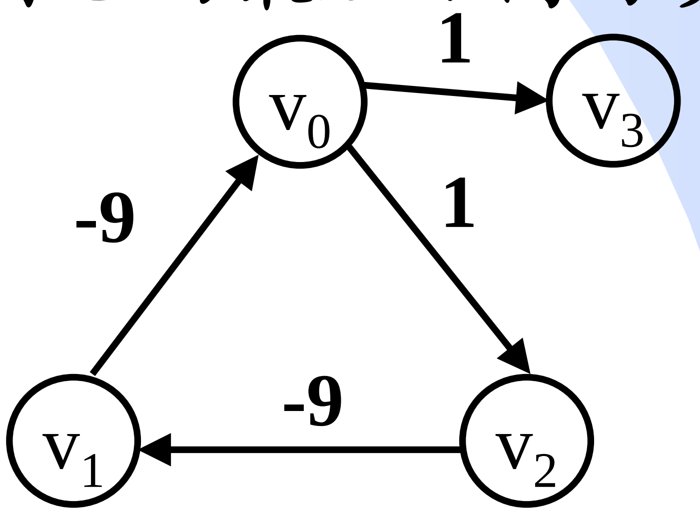
	$D^{(0)}$				$D^{(1)}$				$D^{(2)}$				$D^{(3)}$				$D^{(4)}$			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
2	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
3	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
4	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0

	$path^{(0)}$				$path^{(1)}$				$path^{(2)}$				$path^{(3)}$				$path^{(4)}$			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	-1	2	-1	4	-1	2	-1	4	-1	2	2	2	-1	2	2	2	-1	2	2	2
2	-1	-1	3	4	-1	-1	3	4	-1	-1	3	4	3	-1	3	4	4	-1	4	4
3	1	2	-1	4	1	1	-1	1	1	1	-1	1	1	1	-1	1	1	1	-1	1
4	-1	-1	3	-1	-1	-1	3	-1	-1	-1	3	-1	3	3	3	-1	3	3	3	-1

Floyd算法总结

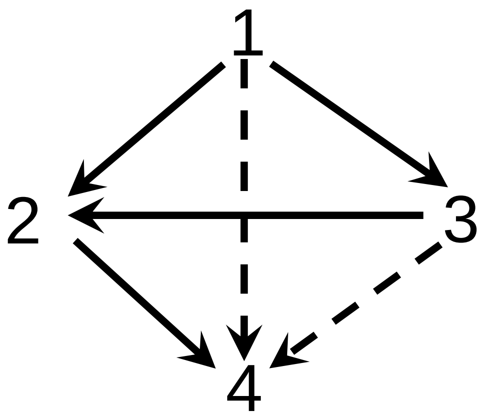
- Floyd算法的时间复杂度为O(n^3)，与调用n次Dijkstra算法求每对顶点的最短路的时间基本相当，但Floyd算法形式简单、算法紧凑、便于实现，允许负权边。
- 适用于有向图，也适用于无向图。
- 不允许存在负环（环上所有边的权值和为负数，理论上不存在最短路）。Floyd算法可以判负环，初始时D[i][i]=0，算法执行过程中若D[i][i]<0，表示顶点i到自己的最短距离为负值，即存在负环。





传递闭包问题

- 不关注路径长度，而是仅关注是否存在路径。
- v_i和v_j是有向图G的两个顶点，若从 v_i 到 v_j 存在一条路径，则称 v_i 到 v_j 可及；
- 传递性：若 v_i 到 v_j，v_j 到 v_k皆可及，则 v_i 到v_k 可及.
- 描述有向图顶点间可及关系的n阶方阵R称为可及矩阵，若顶点v_i 到v_j 可及，则R_{ij} =1，否则R_{ij} =0.
- 传递闭包：由有向图 G 的顶点集 V、边集 *E*，以及新添加的虚边（表示两顶点可及）构成的图被称为 G 的传递闭包。



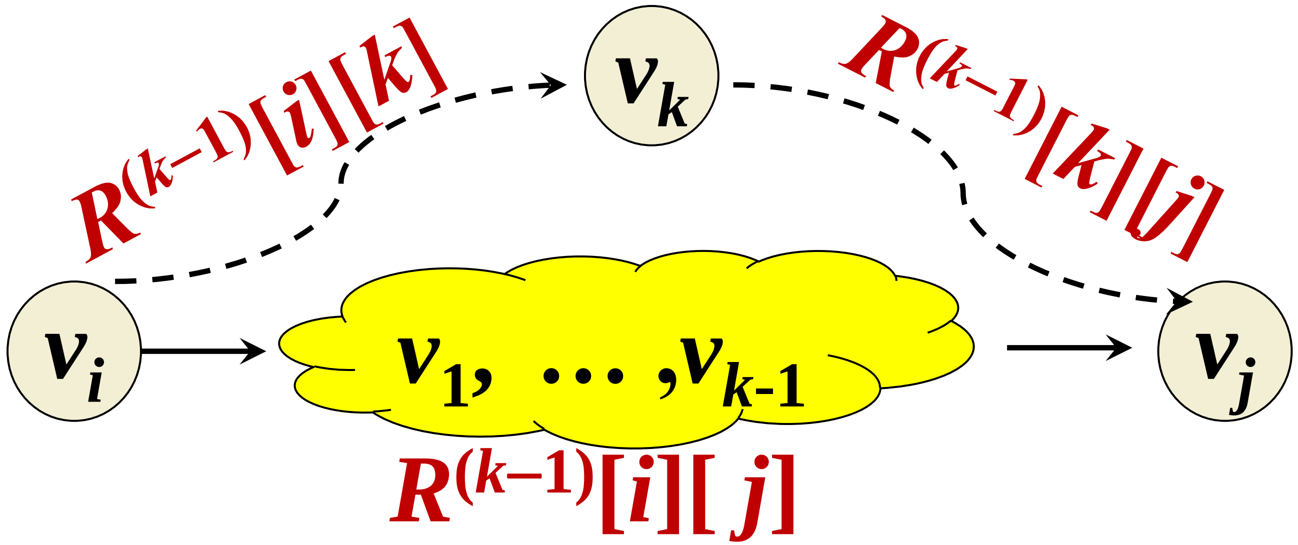
$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

可及矩阵

$R^{(k-1)}[i][j]$表示顶点 i经由$\{v_1, \ldots, v_{k-1}\}$中顶点到达 j 的可及性.

$R^{(k)}[i][j]$表示顶点 i经由$\{v_1, \ldots, v_{k-1}, v_k\}$中顶点到达$j$的可及性.

$$R^{(k)}[i][j] = R^{(k-1)}[i][j] \text{ OR } (R^{(k-1)}[i][k] \text{ AND } R^{(k-1)}[k][j])$$



Warshall算法

```
void Warshall(int A[N][N], int n, int R[N][N]) {
   for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
         if (i==j || A[i][j]<INF) R[i][j] = 1;
         //i和j之间存在边，注意对无权图是A[i][j]==1
         else R[i][j] = 0;
   for (int k = 1; k <= n; k++)   //递推构造Rn
      for (int i = 1; i <= n; i++)
         for (int j = 1; j <= n; j++)
            R[i][j]=R[i][j]||(R[i][k] && R[k][j]);
}
```

$O(n^3)$

$$R^{(k)}[i][j] = R^{(k-1)}[i][j] \text{ OR } (R^{(k-1)}[i][k] \text{ AND } R^{(k-1)}[k][j])$$

Warshall算法

```
void Warshall(int A[N][N], int n, int R[N][N]) {
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        if (i==j || A[i][j]<INF) R[i][j] = 1;
        //i和j之间存在边，注意对无权图是A[i][j]==1
        else R[i][j] = 0;
  for (int k = 1; k <= n; k++)   //递推构造Rⁿ
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
          if(R[i][j]==1||(R[i][k]==1 && R[k][j]==1))
              R[i][j] = 1;
}
```

$$R^{(k)}[i][j] = R^{(k-1)}[i][j] \text{ OR } (R^{(k-1)}[i][k] \text{ AND } R^{(k-1)}[k][j])$$

Warshall算法

```
void Warshall(int A[N][N], int n, int R[N][N]) {
   for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
         if (i==j || A[i][j]<INF) R[i][j] = 1;
         //i和j之间存在边，注意对无权图是A[i][j]==1
         else R[i][j] = 0;
   for (int k = 1; k <= n; k++)   //递推构造Rⁿ
      for (int i = 1; i <= n; i++)
         for (int j = 1; j <= n; j++)
            if(R[i][k]==1 && R[k][j]==1)
               R[i][j] = 1;
}
```

$$R^{(k)}[i][j] = R^{(k-1)}[i][j] \text{ OR } (R^{(k-1)}[i][k] \text{ AND } R^{(k-1)}[k][j])$$

Warshall算法——简单优化

```
void Warshall(int A[N][N], int n, int R[N][N]) {
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        if (i==j || A[i][j]<INF) R[i][j] = 1;
        //i和j之间存在边，注意对无权图是A[i][j]==1
        else R[i][j] = 0;
  for (int k = 1; k <= n; k++)   //递推构造Rⁿ
    for (int i = 1; i <= n; i++)
      if(R[i][k]==1)
          for (int j = 1; j <= n; j++)
            if(R[k][j]==1)
                R[i][j] = 1;
}
```

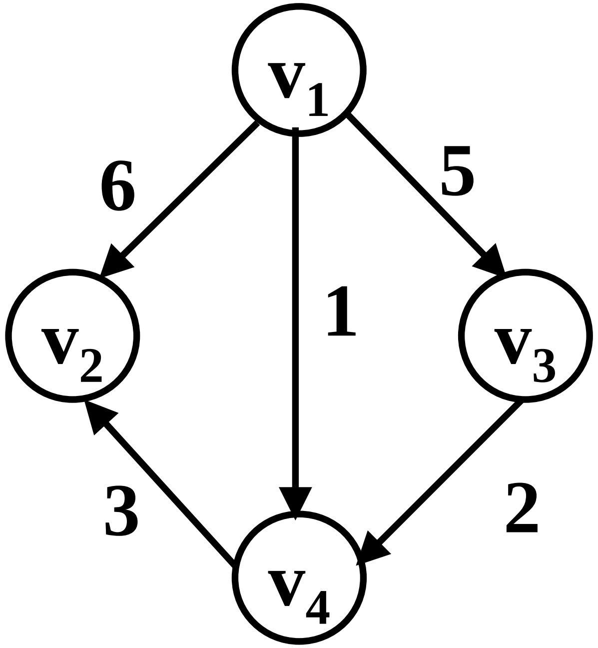
$$R^{(k)}[i][j] = R^{(k-1)}[i][j] \text{ OR } (R^{(k-1)}[i][k] \text{ AND } R^{(k-1)}[k][j])$$

课下思考

- 对于负权图最短路径问题，可否采用如下方案？
- 若图中最小边权是-x（x>0），做一个变换，将所有边的权值加上x+1，使所有边权值变为正数，再调用Dijkstra求最短路径，该最短路就是原图的最短路。

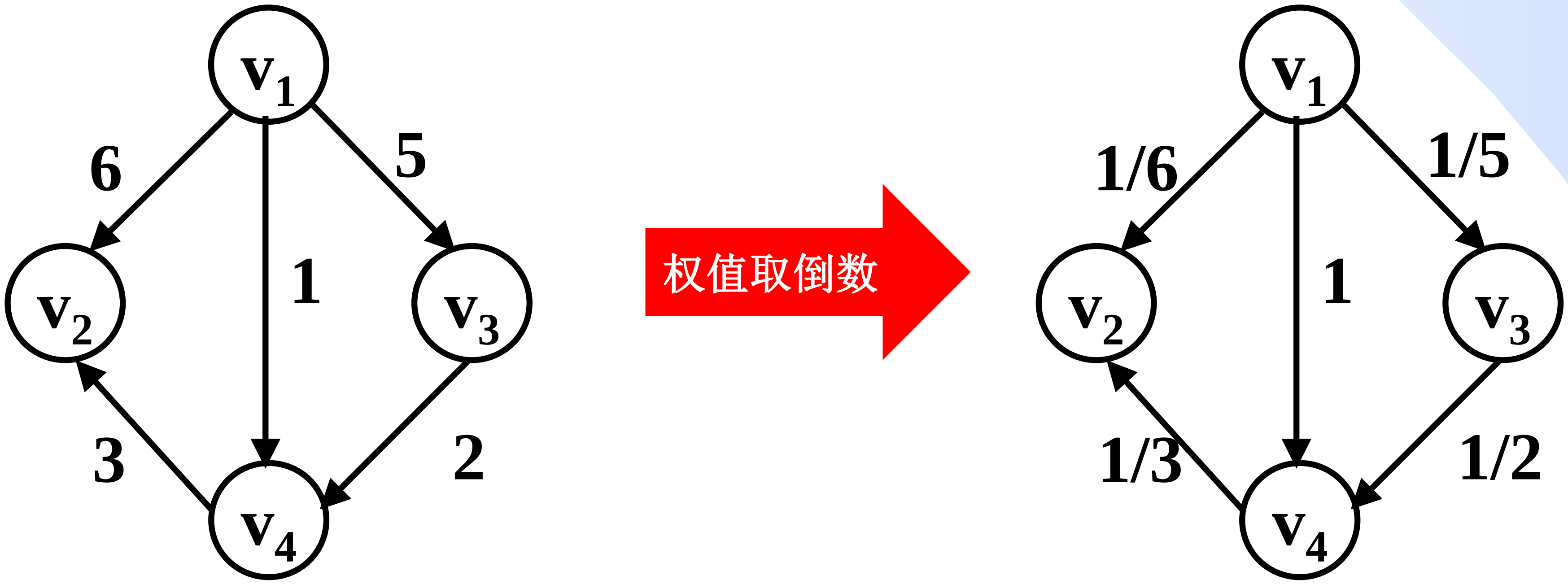
课下思考

- 求图的最长路径，可否采用如下方案？
- 对Dijkstra算法稍加修改，将算法中D[v]+weight(v,w)<D[w]改为D[v]+weight(v,w)>D[w]。



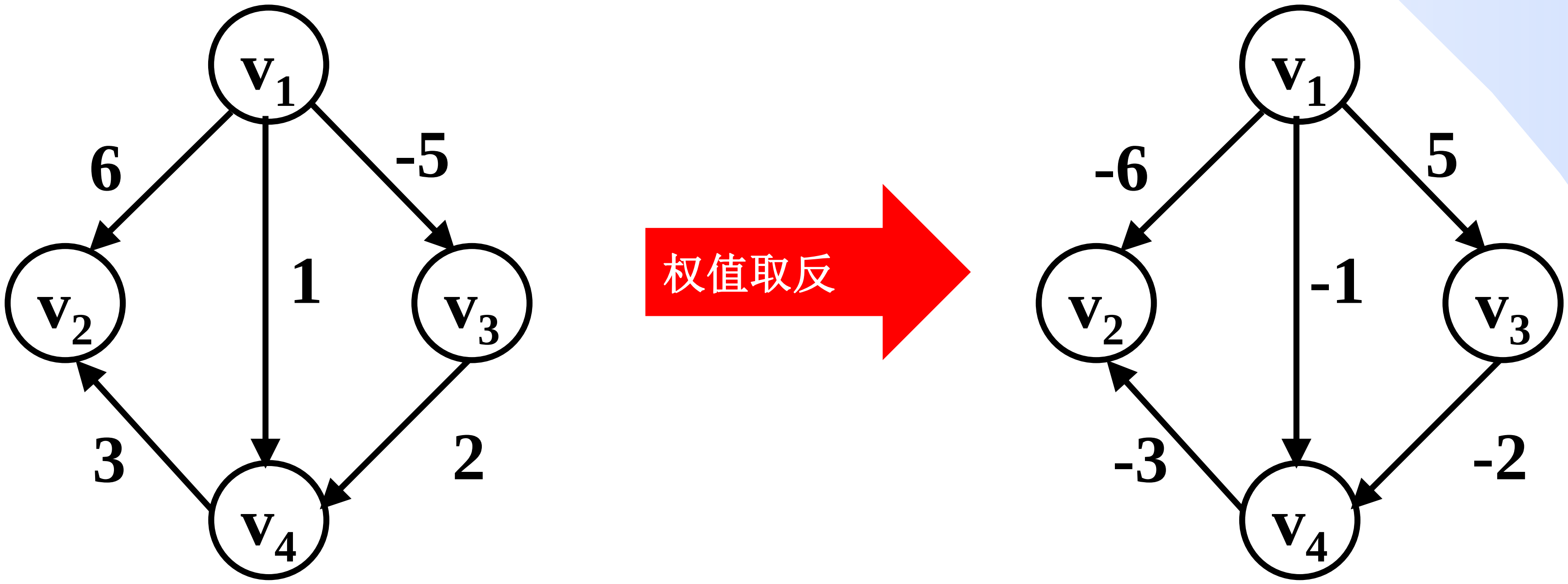
课下思考

- 求图的最长路径，可否采用如下方案？
- 若权值都为正，则对所权值取倒数，用Dijkstra算法求最短路径，该路径就是原图的最长路径。



课下思考

- 求图的最长路径，可否采用如下方案？
- 若图中不存在正环，对所有边的权值取相反数，执行Floyd或Bellman-Ford算法求最短路径，该路径就是原图的最长路。

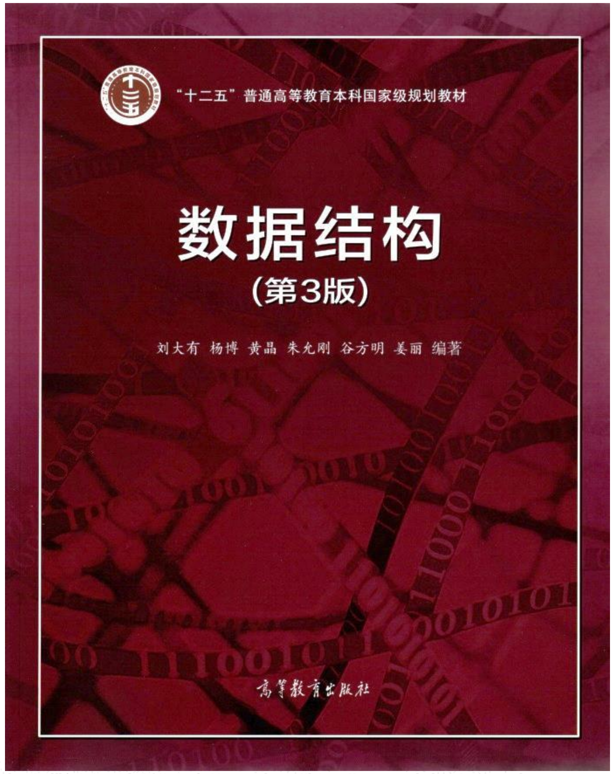


课下思考

- 求图的单源最短路径，可否采用如下方案？
- 将关键路径算法中求*ve*值的算法中max修改为min，则*ve*(*j*)即源点到顶点*j*的最短路径长度，时间复杂度O(*n*+*e*)。若该方案可行，那么O(n^2+*e*)的Dijkstra和O(*ne*)的Bellman-Ford算法的意义何在？

$$ve(j)=\begin{cases} 0, & j=1 \\ \max_i \{ve(i)+w(\langle i,j\rangle) \mid \langle i,j\rangle \in E(G), j=2,\ldots,n\} \end{cases}$$

$$ve(j)=\begin{cases} 0, & j=1 \\ \min_i \{ve(i)+w(\langle i,j\rangle) \mid \langle i,j\rangle \in E(G), j=2,\ldots,n\} \end{cases}$$



图的最短路径

- 无权图最短路径
- Dijkstra算法
- A*算法
- Bellman-Ford算法
- Floyd算法
- **满足约束的最短路径**

数据之法
结构之美
算法之道

Last updated on 2022.11

zhuyungang@jlu.edu.cn

第k短路径

吉林大学(前卫校区)...
长春站
智行
打车
驾车
公交
步行
长春站
预计20:48到达 >
终
限行
未来出行
抽现金
西部快速路
绿园区
亚泰大街
二道区
卫星路
路线偏好
常规路线
24分钟
17公里 6
方案二
27分钟
19公里 9
距离短 红绿灯多
31分钟
14公里 14
打车出发约40元，最高领30元红包
6条出行建议
雷达
开始导航

问题

无权图的单源最短路径问题

正权图的单源最短路径问题

正权图单源单点最短路径问题

负权图的单源最短路径问题

任意两点间的最短路径问题

第K短路径问题

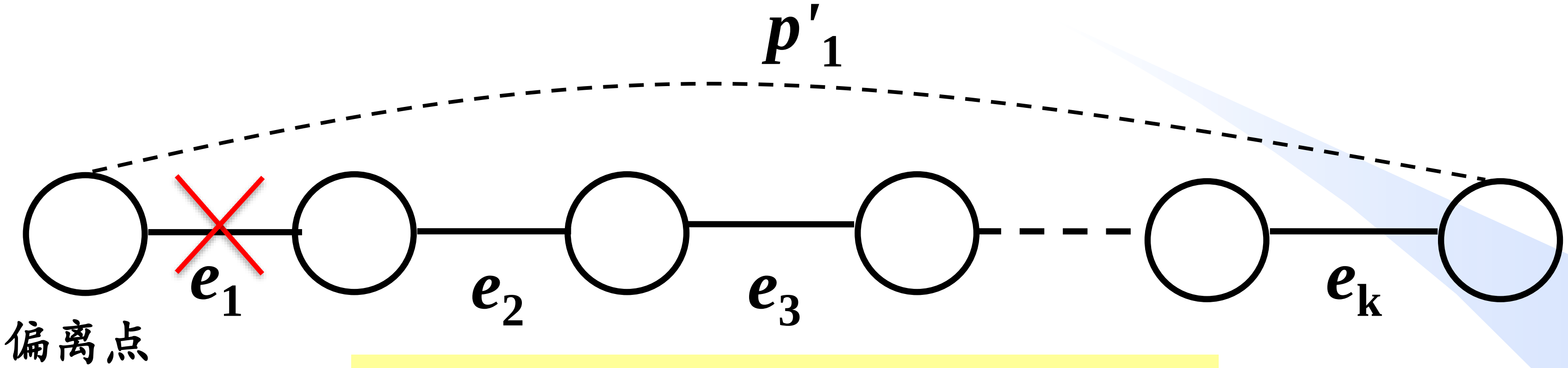
第k短路径

假设$G=(V, E)$是一个包含n个顶点的有向图。s是初始顶点，t是目标顶点，每条边的权值都为正，要求给出s到t的最短路径、第2短路径、第3短路径、……、第k短路径。

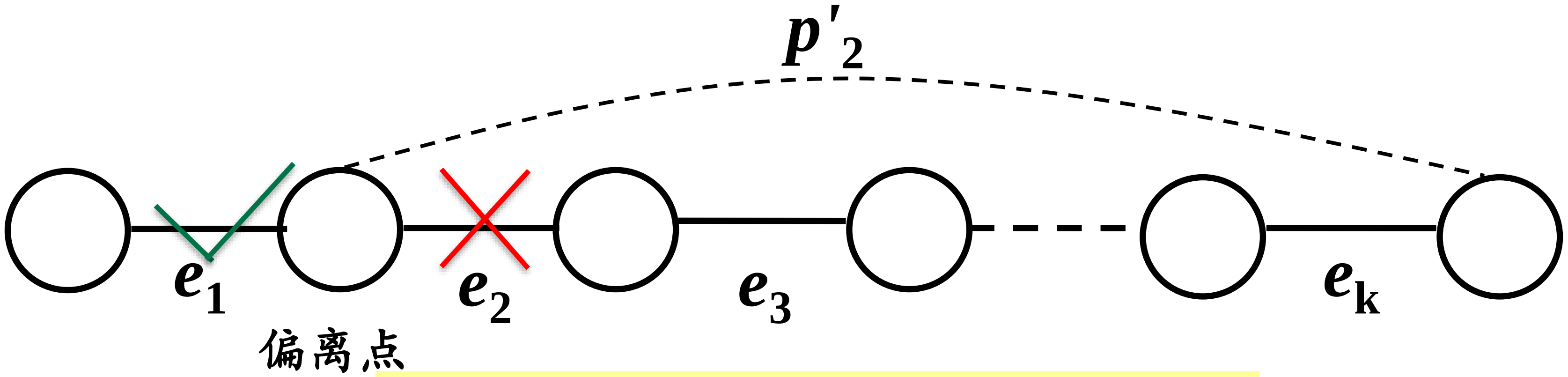
Yen 算法

（1）先求得s到t的最短路径，假设其经过的边为$e_1, e_2, \ldots e_{k-2}, e_{k-1}, e_k$；

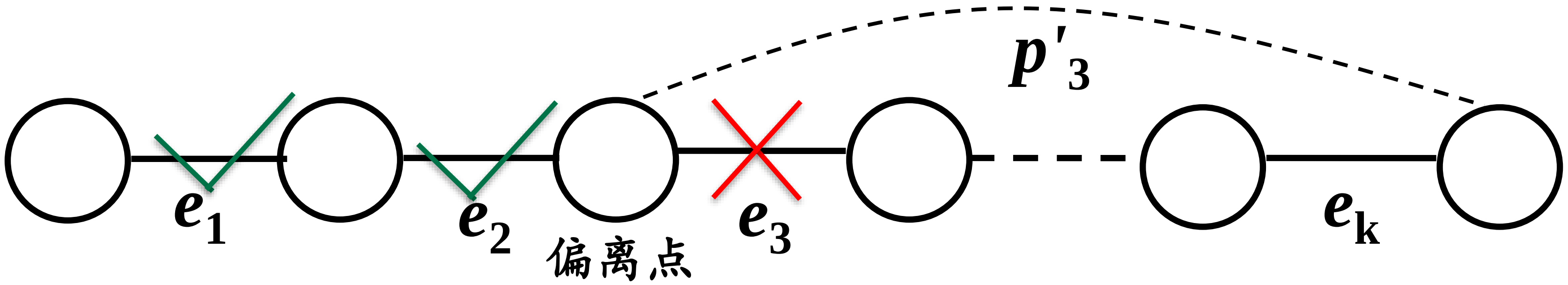
（2）



$p_1 = p'_1$送入候选集合

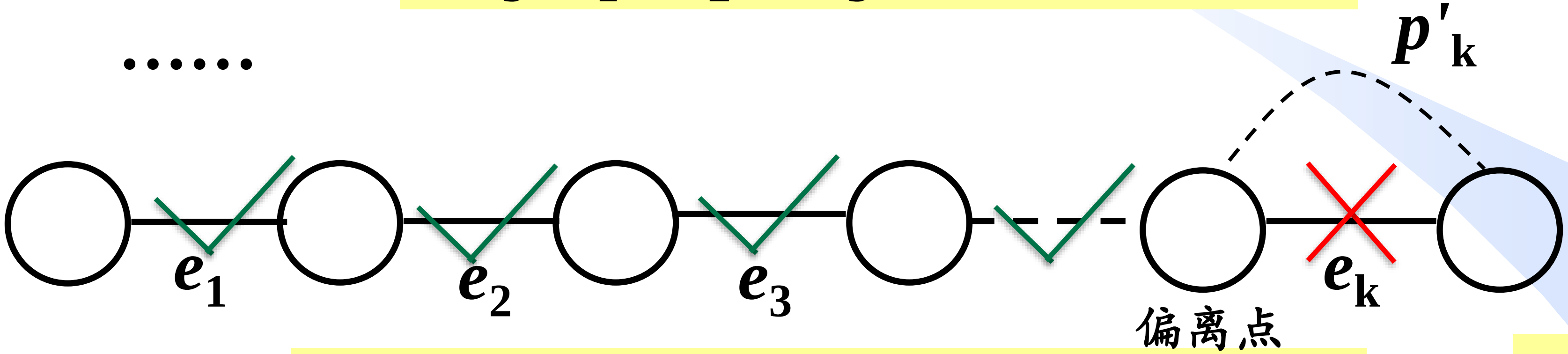


$p_2=e_1+p'_2$送入候选集合



$p_3=e_1+e_2+p'_3$送入候选集合

……



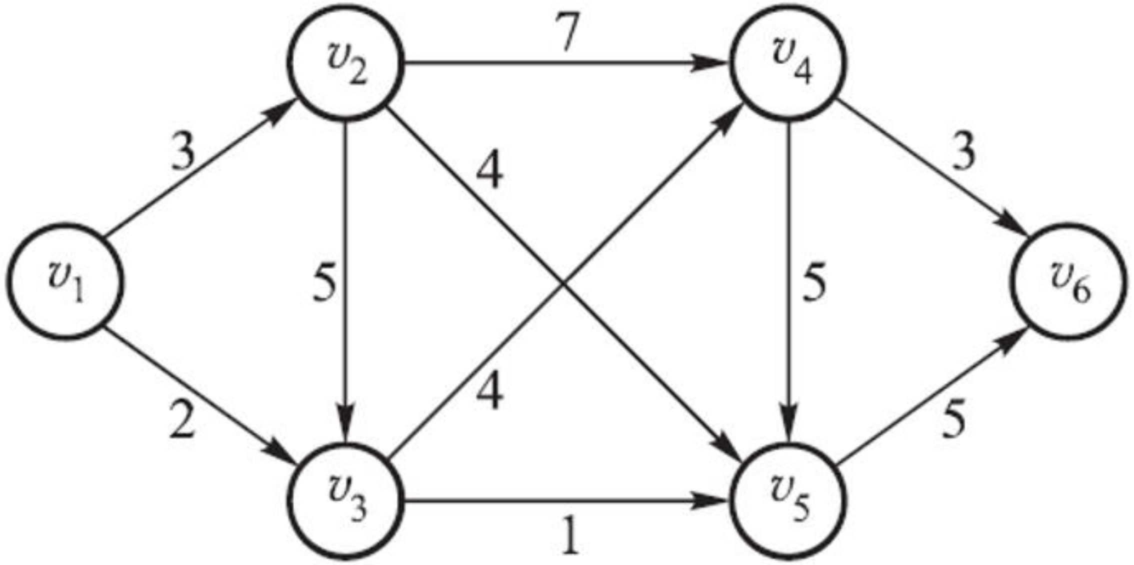
$p_k=e_1+\ldots+e_{k-1}+p'_k$送入候选集合

时间复杂度 $O(kn^3)$

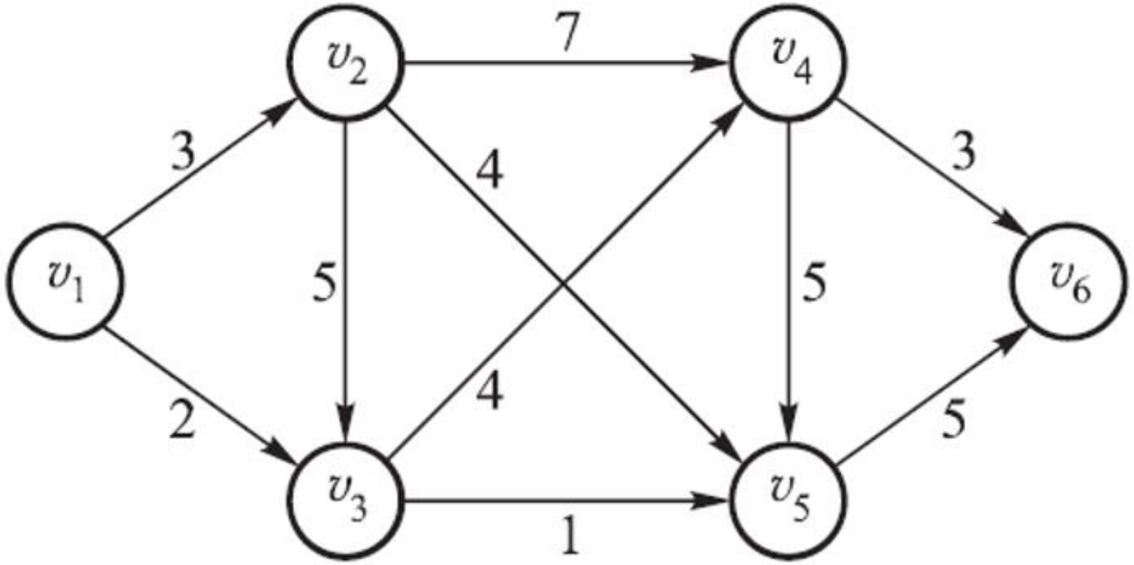
（3）候选集合中选出最短路径即为第2短路径。

（4）对第2短路径重复上述操作(2)-(3)，直至求出第k短路径。

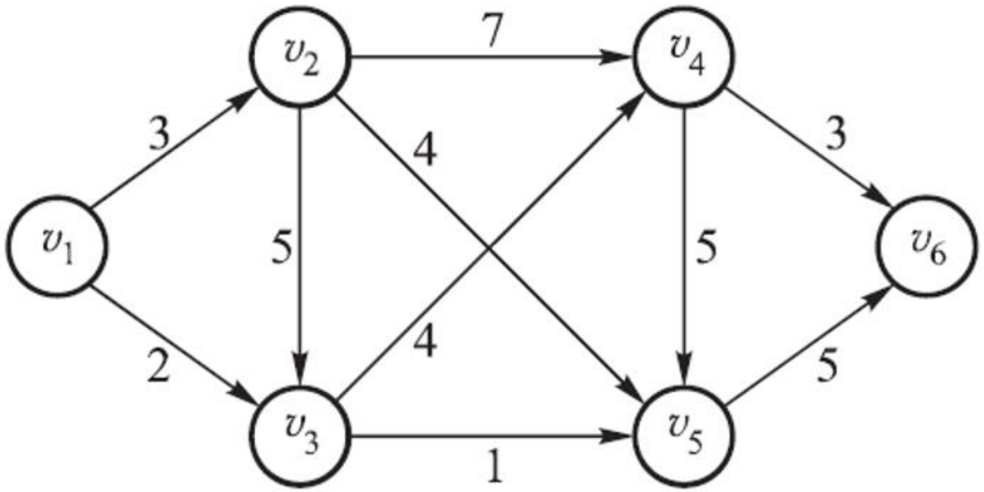
- 防止路径中有环：从偏离点v到t的最短路径不能包含s到v的顶点
- 避免与已求路径重复：从偏离点v出发的边不能与已求出的路径中v出发的边相同



最短路径	路径长度	包含的边	不包含的边	候选路径
$v_1v_3v_5v_6$	8			
			v_1v_3	$v_1v_2v_5v_6$=12
		v_1v_3	v_3v_5	$v_1v_3v_4v_6$=9
		v_1v_3 , v_3v_5	v_5v_6	∞
$v_1v_3v_4v_6$	9		v_1v_3	$\cancel{v_1v_2v_5v_6=12}$
		v_1v_3	v_3v_4 , v_3v_5	∞
		v_1v_3 , v_3v_4	v_4v_6	$v_1v_3v_4$ v_5v_6=16
$v_1v_2v_5v_6$	12		v_1v_2 ,v_1v_3	∞
		v_1v_2	v_2v_5	v_1v_2 v_4v_6=13
		v_1v_2 ,v_2v_5	v_5v_6	∞



最短路径	路径长度	包含的边	不包含的边	候选路径
$v_1v_2\ v_4v_6$	13		v_1v_2 ,v_1v_3	∞
		v_1v_2	v_2v_4 ,v_2v_5	$v_1v_2v_3v_5v_6$=14
		v_1v_2 ,v_2v_4	v_4v_6	$v_1v_2v_4v_5v_6$=20
$v_1v_2v_3v_5v_6$	14		v_1v_2,v_1v_3	∞
		v_1v_2	v_2v_3 ,v_2v_4 ,v_2v_5	∞
		v_1v_2 ,v_2v_3	v_3v_5	$v_1v_2v_3v_4v_6$=15
		v_1v_2 ,v_2v_3 ,v_3v_5	v_5v_6	∞
$v_1v_2v_3v_4v_6$	15		v_1v_2,v_1v_3	∞
		v_1v_2	v_2v_3 ,v_2v_4 ,v_2v_5	∞
		v_1v_2 ,v_2v_3	v_3v_4 , v_3v_5	∞
		v_1v_2 ,v_2v_3 ,v_3v_4	v_4v_6	$v_1v_2v_3v_4\ v_5v_6$=22



最短路径	路径长度	包含的边	不包含的边	候选路径
$v_1v_3v_4\ v_5v_6$	16		v_1v_2,v_1v_3	∞
		v_1v_3	v_3v_4 , v_3v_5	∞
		v_1v_3 , v_3v_4	v_4v_5 ,v_4v_6	∞
		v_1v_3 , v_3v_4 ,v_4v_5	v_5v_6	∞
$v_1v_2v_4v_5v_6$	20		v_1v_2,v_1v_3	∞
		v_1v_2	v_2v_3 ,v_2v_4 ,v_2v_5	∞
		v_1v_2 ,v_2v_4	v_4v_5 ,v_4v_6	∞
		v_1v_2 ,v_2v_4 ,v_4v_5	v_5v_6	∞
$v_1v_2v_3v_4\ v_5v_6$	22		v_1v_2,v_1v_3	∞
		v_1v_2	v_2v_3 ,v_2v_4 ,v_2v_5	∞
		v_1v_2 ,v_2v_3	v_3v_4 , v_3v_5	∞
		v_1v_2 ,v_2v_3 ,v_3v_4	v_4v_5 ,v_4v_6	∞
		v_1v_2 ,v_2v_3 ,v_3v_4 ,v_4v_5	v_5v_6	∞

方案2：修改Dijkstra算法

优先级队列

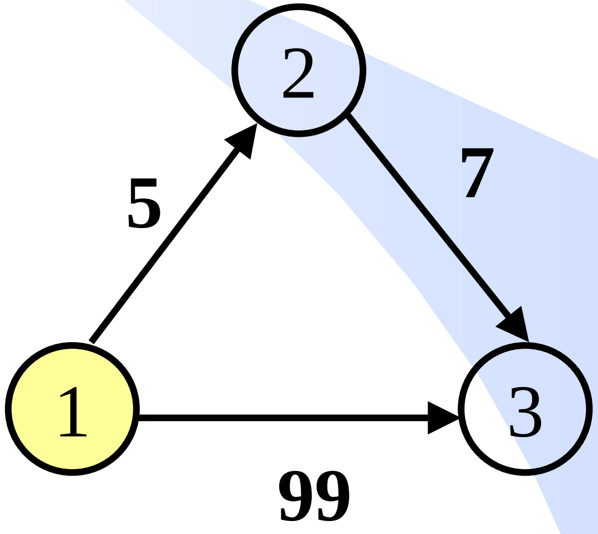
初始时（ s为源点），将$(s, 0)$放入队列Q

①从Q中出队D_v最小元素的(v, D_v)。

②依次考察v的邻接顶点w，更新D_w的值，使$D_w \leftarrow D_v + weight(<v,w>)$．将$(w, D_w)$入队（队列中可能包含顶点$w$对应的多个二元组）。

③重复①②，直至目标点t第k次出队。

维护一个队列Q，队列中存储二元组(v, D_v)。出队时并不是队头元素队列，而是D值最小的元素出队。



顶点t第i次出队时，D_t就是s到t的第i短距离

可以用A*算法加速

	(2, 5)	(3, 12)
(1, 0)	(3, 99)	(3, 99)

方案2：修改Dijkstra算法

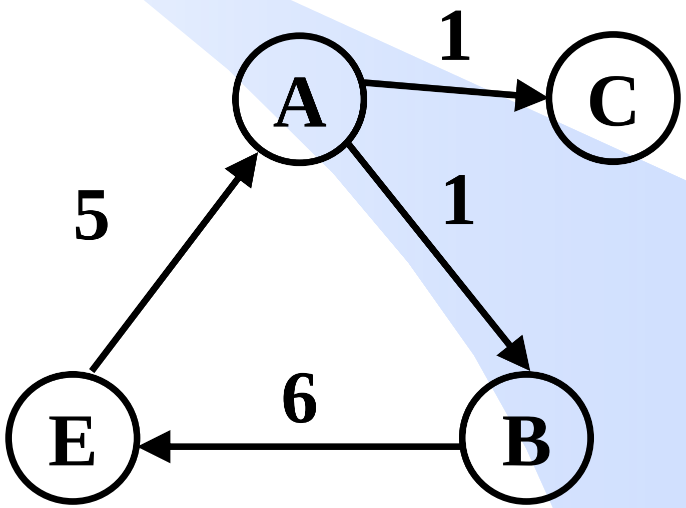
初始时（s为源点），将(s, 0)放入队列Q

①从Q中出队D_v最小元素的(v, D_v)。

②依次考察v的邻接顶点w，更新D_w的值，使$D_w \leftarrow D_v$ + $weight(<v,w>)$．将 (w, D_w)入队（队列中可能包含顶点w对应的多个二元组）。

③重复①②，直至目标点t第k次出队。

找到的第k短路可能存在环



顶点t第i次出队时，D_t就是s到t的第i短距离

满足约束的最短路径

某些问题需要找在某些约束条件下的最短路径，一种可行的方案是依次生成两个顶点间的第1、2、3、…、k短路径，然后逐一测试每条路径是否满足给定的约束条件，第一条满足约束条件的路径即为所求。

最短路径
第2短路径
第3短路径
……
第k短路径
……

问题

无权图的单源最短路径问题

正权图的单源最短路径问题

正权图单源单点最短路径问题

负权图的单源最短路径问题

任意两点间的最短路径问题

第K短路径问题

满足约束的最短路径问题