算法设计与分析

Computer Algorithm Design & Analysis

文明
mwenaa@hust.edu.cn

第十一讲
Single-Source Shortest Paths

单源最短路径

最短路径问题：

给定一个带权重的有向图G=(V,E)和权重函数ω:E→R。图中一条路径$p=\langle v_0,v_1,\ldots,v_k\rangle$的权重ω(p)是构成该路径的所有边的权重之和：

$$w(p)=\sum_{i=1}^{k} w(v_{i-1}, v_i)\ .$$

从结点u到结点v的最短路径权重δ(u,v)定义如下：

$$\delta(u,v)=\begin{cases}\min\{w(p): u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v\ ,\\ \infty & \text{otherwise}\ .\end{cases}$$

单源最短路径问题：

给定一个图G=(V,E)，找出从给定的源点s∈V到其它每个结点v∈V的最短路径。

- ## 最短路径的最优子结构

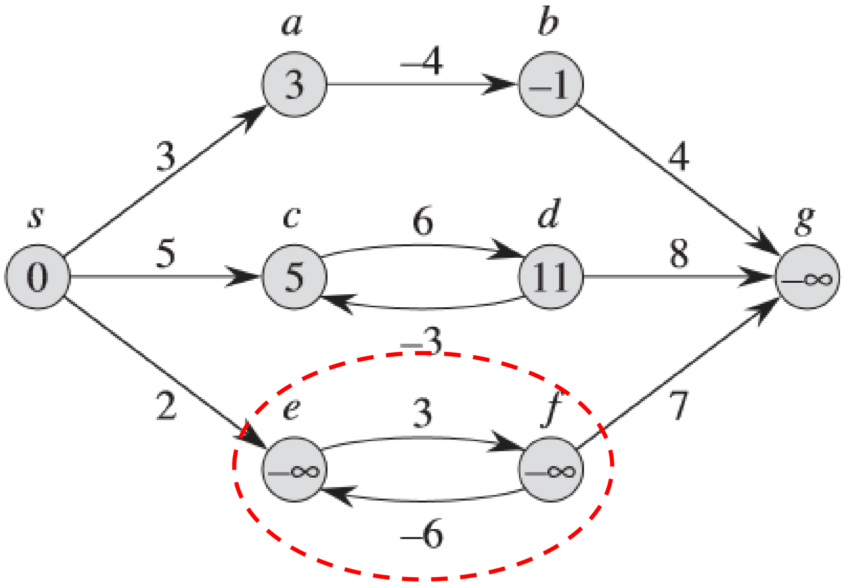
这样最短路径具有最优子结构性：两个结点之间的最短路径的任何子路径都是最短的。

引理24.1 给定一个带权重的有向图G=(V,E)和权重函数ω:E→R。设$p=\langle v_0, v_1, \cdots, v_k\rangle$为从结点$v_0$到结点$v_k$的一条最短路径，并且对于任意的i和j，$0 \le i \le j \le k$，设$p_{ij}=\langle v_i, v_{i+1}, \cdots, v_j\rangle$为路径p中从结点$v_i$到结点$v_j$的子路径，则$p_{ij}$是从结点$v_i$到结点$v_j$的一条最短路径。（证明略）

▪ 负权重的边

权重为负值的边称为负权重的边。

- 如果存在负权重的边，则有可能存在权重为负值的环路，而造成图中最短路径无定义（路径的权重为-∞）。



- 环路
- 最短路不应包含环路。
    - 不包含环路的路径称为简单路径。
        - 对任何简单路径最多包含|V|-1条边和|V|个结点。
        - 不失一般性，假设后续算法寻找的最短路径都不包含环路。
- 最短路径的表示
- 一个结点的前驱结点记为：v.π
    - 前驱结点：为NIL或者为另一个结点
- 利用v.π的记录可以搜索出最短路径上的所有结点。

■ 前驱子图

定义前驱子图为$G_\pi=(V_\pi, E_\pi)$，其中，

- 结点集合$V_\pi=\{v\in V: v.\pi\neq NIL\}\cup\{s\}$
  - 即V_π是图G中的前驱结点不为NIL的结点的集合，再加上源点s。
- 边集合$E_\pi=\{(v.\pi,v)\in E: v \in V_\pi-\{s\}\}$
  - 即E_π是由V_π中的结点的π值所“诱导”(induced)的边的集合。

则，算法终止时，**G_π是一棵最短路径树**。

- 该树包含了从源结点s到每个可以从s到达的结点的一条最短路径。

设G=(V,E)是一条带权重的有向图，其权重函数为ω:E→R，假定G不包含从s可以到达的权重为负值的环路，因此，所有的最短路径都有定义。

一棵根结点为s的最短路径树是一个有向子图G′ =(V′, E′)，这里 $V' \subseteq V$，$E' \subseteq E$。且有以下性质：

（1） V′是图G中从源结点s可以到达的所有结点的集合。

（2） G′形成一棵根结点为s的树。

（3）对于所有的结点v∈V′，图G′中从结点s到结点v的唯一简单路径是图G中从结点s到结点v的一条最短路径。

松弛操作（Relax）

对于每个结点v，维持一个属性v.d，记录**从源点s到结点v的最短路径权重的上界**。称v.d为s到v的**最短路径估计**。

- 过程INITIALIZE-SINGLE-SOURCE实现对结点最短路径估计和前驱结点的**初始化**：

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

初始化后，对所有的结点v∈V有，

- v.π = NIL;
- s.d = 0;
- 对所有的结点v∈V-{s}有：v.d = ∞。

INITIALIZE-SINGLE-SOURCE 的时间：Θ(V)

松弛操作： 首先测试一下是否可以对从s到v的最短路径进行改善（即有没有更短的路径）。如果可以改善，则v.d更新为新的最短路径估计值，v的前驱v.π更新为新的前驱结点。

RELAX(u, v, w)

1 **if** $v.d > u.d + w(u, v)$
2 $\quad v.d = u.d + w(u, v)$
3 $\quad v.\pi = u$

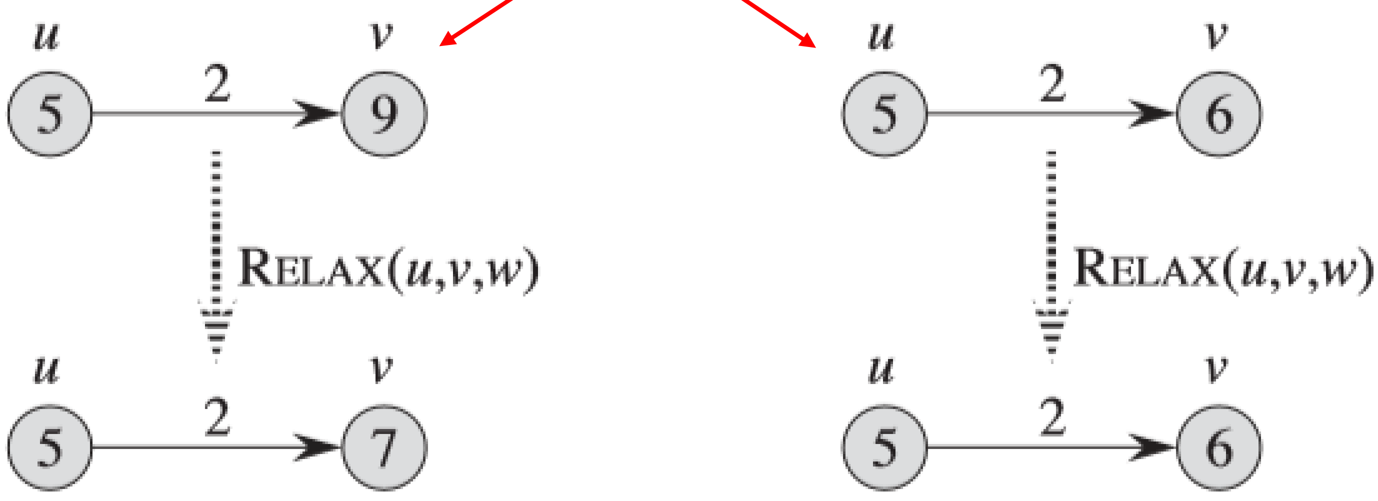
RELAX 的时间：O(1)

测试：对s到v所经过的**最后一个中间结点u**，按下列方式计算从s出发，经过u而到达v的路径的权重：

将从结点s到结点u之间最短路径加上结点u到v之间的边的权重，然后与当前的s到v的最短路径估计v.d进行比较，看有没有变小。如果变小，则对v.d和v.π进行更新——这一操作就称为“**松弛**”

结点中的数字：每个结点的最短路径估计

例：



(a)　　　　　　(b)

对边(u,v)进行松弛操作，权重：ω(u,v)=2：

(a)：因为v.d > u.d + ω(u,v)，所以v.d的值减小(9 > 5+2)。

(b)：因为v.d ≤ u.d + ω(u,v)，所以v.d的值没有改变。

最短路径和松弛操作的性质

1. 三角不等式性质

引理24.11 设G=(V, E)为一个带权重的有向图，其权重函数为ω:E→R，设其源结点为s。那么对于所有的边(u, v)∈E，有

$$\delta(s,v) \leqslant \delta(s,u) + w(u,v)$$

证明:

假定p是从源结点s到结点v的一条最短路径，则p的权重不会比任何从s到v的其它路径的权重大，因此路径p的权重也不会比这样的一条路径的权重大：从源结点s到结点u的一条最短路径，再加上边(u, v)而到达结点v的这条路径。

如果s到v没有最短路径，则不可能存在到v的路径。 ■

2. 上界性质： v.d是s到v的最短路径权重δ(s, v)的上界

引理24.11 设G=(V,E)为一个带权重的有向图，其权重函数为ω:E→R，设其源结点为s，该图由算法 *INITIALIZE-SINGLE-SOURCE(G,s)* 执行初始化。那么对于所有的结点v∈V，v.d≥δ(s,v)。并且该不变式在对图G的边进行任何次序的松弛过程中都保持成立，而一旦v.d取得其下界δ(s,v)后，将不再发生变化。

用数学归纳法证明：对于所有的结点v∈V，v.d≥δ(s,v)。

- 注：归纳的主体是松弛步骤的数量。

基础步：在经 *INITIALIZE-SINGLE-SOURCE(G, s)* 初始化之后，对于所有的结点v∈V-{s}，置 v.d=∞ ，而s.d=0 ，显然s.d≥δ(s, s)，而其它的结点 v.d≥δ(s, v)，结论成立。

归纳步：考虑对边(u, v)的松弛操作。

归纳假设：在对边(u, v)进行松弛之前，对所有的结点x∈V，

x. d≥δ(s, x)。

而在对边(u, v)进行松弛的过程中，唯一可能发生改变的d值只有v. d，如果该值发生变化，则有：

$$\begin{aligned} v.d &= u.d + w(u,v) & \\ &\geq \delta(s,u) + w(u,v) & \text{(by the inductive hypothesis)} \\ &\geq \delta(s,v) & \text{(by the triangle inequality)}\ , \end{aligned}$$

同时，根据计算的规则，在v. d达到其下界δ(s, v)后，就无法再减小（也不可能增加）。

引理得证。

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

3. 非路径性质

推论24.12　给定一个带权重的有向图G=(V,E)，其权重函数为ω:E→R。假定从源结点s到给定点v之间不存在路径，则该图在由算法 *INITIALIZE-SINGLE-SOURCE(G,s)* 进行初始化后，有

$$v.d \geq \delta(s,v) = \infty,$$

并且该等式作为不变式一直维持到图G的所有松弛操作结束。

证明:

因为从源点s到给定点v之间不存在路径，所以$\delta(s,v)=\infty$。而根据上界性质，总有$v.d \geqslant \delta(s,v)$，所以，$v.d \geqslant \delta(s,v)=\infty$。

得证。

引理24.13 设G=(V,E)为一个带权重的有向图，其权重函数为ω:E→R，并且边(u,v)∈E。那么在对边(u,v)进行松弛操作RELAX(u,v,ω)后，有v.d≤ u.d+ω(u,v) 。

证明:

如果在对边(u, v)进行松弛操作前，有v. d>u. d+ω (u, v)，则松弛操作时，置v. d=u. d+ω (u, v)。

如果在松弛操作前有v. d≤ u. d+ω (u, v)，则松弛操作不会改变v. d和u. d的值，因此在松弛操作后仍有v. d≤u. d+ω (u, v).

得证。

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

4. 收敛性质

引理24.14　设G=(V,E)为一个带权重的有向图，其权重函数为ω:E→R。设s∈V为某个源结点，$s \rightsquigarrow u \rightarrow v$ 为图G中的一条最短路径（u，v∈V）。

假定图G由算法*INITIALIZE-SINGLE-SOURCE(G,s)*进行初始化，并在这之后进行了一系列边的松弛操作，其中包括对边(u,v)的松弛操作RELAX(u,v,ω)。如果在对边(u,v)进行松弛操作之前的某时刻有u.d=δ(s,u)，则在该松弛操作之后的所有时刻有v.d=δ(s,v) 。

证明：

根据上界性质，如果在对边(u, v)进行松弛前的某个时刻有u. d=δ(s, u)，则该等式在松弛之后仍然成立。

特别地，在对边(u, v)进行松弛后，有

$$
\begin{aligned}
v.d &\le u.d + w(u,v) && \text{(by Lemma 24.13)} \\
&= \delta(s,u) + w(u,v) \\
&= \delta(s,v) && \text{(by Lemma 24.1)}\ .
\end{aligned}
$$

最优子结构性

而根据上界性质，有v.d≥δ(s,v)。所以有v.d=δ(s,v)，并且该等式在此之后一直保持成立。

得证。

4. 路径松弛性质

引理24.15　设G=(V,E)为一个带权重的有向图，其权重函数为ω:E→R。设s∈V为某个源结点，考虑从源结点s到结点v_k的任意一条最短路径$p=\langle v_0,v_1,\ldots,v_k\rangle$，$v_0=s$。

如果图G由算法 *INITIALIZE-SINGLE-SOURCE(G,s)* 进行初始化，并在这之后进行了一系列边的松弛操作，其中包括对边(v_0,v_1)、(v_1,v_2)、…、(v_{k-1},v_k)按照所列次序而进行的松弛操作，则在所有这些松弛操作之后，有$v_k.d=\delta(s,v_k)$，并且在此之后该等式一直保持。

该性质的成立与其他边的松弛操作及次序无关，即使这些松弛操作是与对p上的边所进行的松弛操作穿插进行的。

归纳法证明：在最短路径p的第i条边被松弛之后，有$v_i.d=\delta(s,v_i)$

基础步：在对路径p的任何一条边进行松弛操作之前，从初始化算法可以得出：$v_0.d=s.d=\delta(s,s)$。结论成立，且s.d的取值在此之后不再发生变化。

归纳步：假定依次经过(v_0,v_1)、(v_1,v_2)、…、(v_{i-2},v_{i-1})松弛操作之后，$v_{i-1}.d=\delta(s,v_{i-1})$。

则在对边(v_{i-1},v_i)进行松弛操时，根据收敛性质，必有在对该边进行松弛后$v_i.d=\delta(s,v_i)$，并且该等式在此之后一直保持成立。

得证。

24.1 Bellman-ford算法

Bellman-ford算法可以求解一般情况下的单源最短路径问题

—— 可以有负权重的边，但不能有负权重的环。

设G=(V,E)为一个带权重的有向图，其权重函数为ω:E→R。s∈V为源结点。

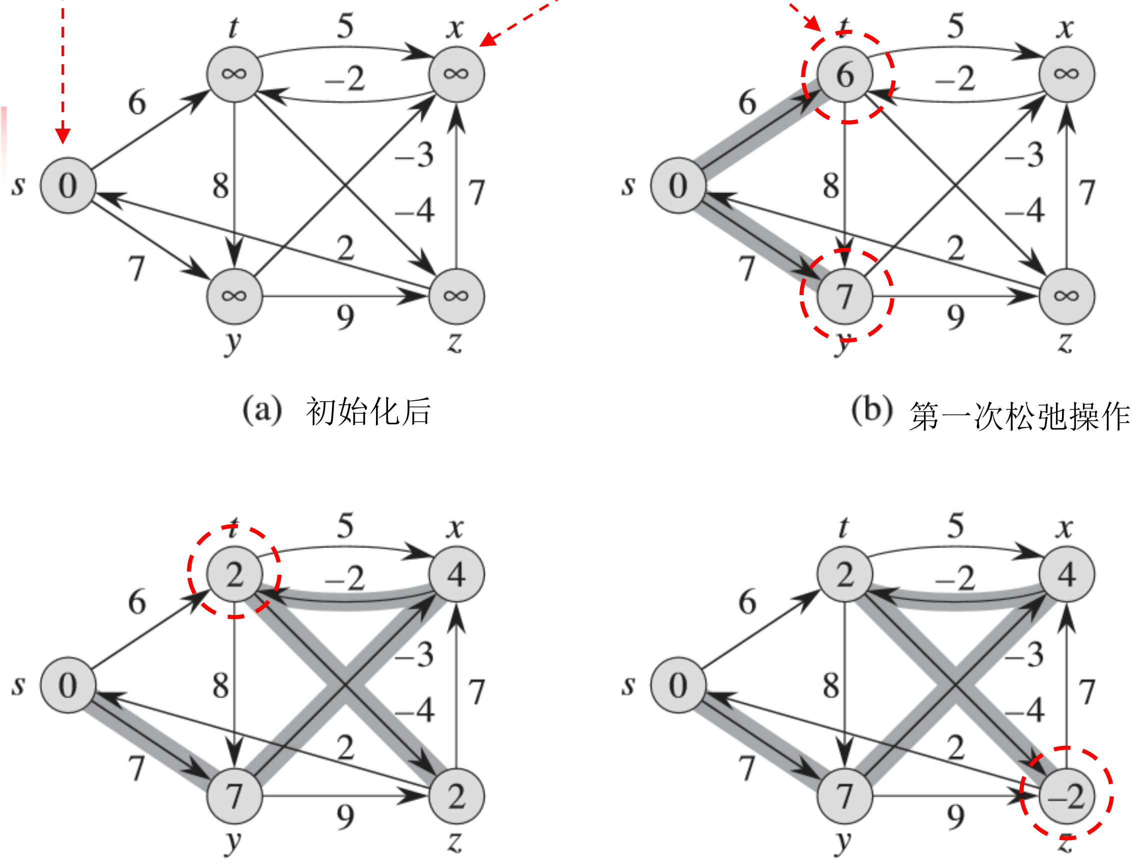
```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

算法返回TRUE当且仅当图G中不包含从源结点可达的权重为负值的环路。

源结点：s

结点中的数值是结点的d值

```
RELAX(u, v, w)
  if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```



(a) 初始化后

(b) 第一次松弛操作后

(c) 第二次松弛操作后

(d) 第三次松弛操作后

(e) 第四次松弛操作后

初始化后，算法将对图中每条边进行|V|-1次松弛处理：

for循环执行|V|-1次，每次对所有的边进行进行一次松弛处理。

例，Bellman-ford算法的执行过程

- 加了阴影的边表示前驱值：如果边(u,v)加了阴影，则v.π=u.
- 本例中Bellman-ford算法执行4次松弛操作后返回TRUE。

Bellman-ford算法的运行时间

初始化：Θ(V)

松弛处理：for循环执行|V|-1次，

每次的时间是Θ(E)

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

O(E)

- Bellman-ford算法总的运行时间O(VE)。

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

Θ(V)

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Θ(1)

Bellman-ford算法的证明

设G=(V, E)为一个带权重的源点为s的有向图，其权重函数为ω:E→R，并假定图G中不包含从源结点s可以到达的权重为负值的环路。

引理24.2 Bellman-ford算法的第2~4行的for循环在执行|V|-1次之后，对于所有从源结点s可以到达的结点v有

$$v.d=\delta(s,v)。$$

v.d到达下界

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

引理24.2 假定图G中不包含从源结点s可以到达的权重为负值的环路。则Bellman-ford算法的第2~4行的for循环在执行|V|-1之后，对于所有从源结点s可以到达的结点v有v.d=δ(s,v)。

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

证明：（使用路径松弛性质证明）

考虑任意从源结点s可以到达的结点v。设$p=\langle v_0, v_1, \cdots, v_k\rangle$是从结点s到结点v之间的任意一条最短路径，这里$v_0=s$，$v_k=v$。

- 因为最短路径都是简单路径，所以p中最多包含|V|-1条边，故$k \leqslant |V|-1$。
- 同时，算法第2~4行的for循环每次松弛所有的|E|条边，每一次为p扩展一条边。所以对序列$\langle v_0, v_1, \cdots, v_k\rangle$在其第i次松弛操作时，被松弛的边中包含边$(v_{i-1}, v_i)$，这里$i=1, 2, \cdots, k$。
- 根据路径松弛性质有：$v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. 得证。

引理24.3　对所有结点$v \in V$，存在一条从源结点s到结点v的路径当且仅当Bellman-ford算法终止时有$v.d < \infty$。

证明：（略）

定理24.4（Bellman-ford算法的正确性）　设Bellman-ford算法运行在一个带权重的源点为s的有向图$G=(V,E)$上，其权重函数为$\omega: E \rightarrow R$。

- 如果图G中不包含从源结点s可以到达的权重为负值的环路，则算法将返回TRUE，且对于所有结点$v \in V$，前驱子图G_π是一个根结点为s的最短路径树。
- 而如果图G中包含一条从源结点s可以到达的权重为负值的环路，则算法将返回FALSE。

定理24.4（Bellman-ford算法的正确性）的证明：

1）首先证明：如果图G中不包含从源结点s可以到达的权重为负值的环路，则算法将返回TRUE，且对于所有结点v∈V，前驱子图G_π是一个根结点为s的最短路径树。

证明：

（1）证明：对于所有结点v∈V，在算法终止时，有v. d= δ（s, v）。

- 如果结点v是从s可以到达的，则论断可以从引理24.2得到证明。
- 如果结点v不能从s可达，则论断可以从非路径性质获得。

因此，对于所有结点v∈V，在算法终止时，有v.d=δ(s,v)。

（2）综合前驱子图性质和本论断，可以推导出G_π是一棵最短路径树

(3）终止时，算法是否返回TRUE?

算法终止时，对所有的边(u,v)∈E，有

$$
\begin{aligned}
v.d &= \delta(s,v) \\
&\le \delta(s,u) + w(u,v) \quad \text{(by the triangle inequality)} \\
&= u.d + w(u,v)\,,
\end{aligned}
$$

因此，算法第6行中没有任何测试可以让算法返回FALSE（ G中不包含从源结点s可以到达的权重为负值的环路），因此一定返回TRUE值。

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

2）然后证明：如果图G中包含一条从源结点s可以到达的权重为负值的环路，则算法将返回FALSE。

证明：

假定图G包含一个权重为负值的环路，并且该环路可以从源结点s到达。设该环路为c=<v_0,v_1,...v_k>，这里v_0=v_k。

因为环路的权重为负值，所以有：

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \qquad (24.1)$$

反证法证明：假设此种情况下Bellman-ford算法返回TRUE值，

则有：对所有的i=1,2,...,k，

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

将环路c上的所有这种不等式都加起来，有：

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

由于$v_0=v_k$，环路c上面的每个结点在上述求和表达式 $\sum_{i=1}^{k} v_i.d$ 和$\sum_{i=1}^{k} v_{i-1}.d$ 中都刚好各出现一次。因此有

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d \, .$$

因此有 $$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

与 $$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \, .$$ 相矛盾。

因此，如果图G中不包含从源结点s可以到达的权重为负值的环路，则算法将返回TRUE，否则返回FALSE。 得证。

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

struct Edge
{
    // This structure is equal to an edge. Edge contains two end points. These edges are d
    //contain source and destination and some weight. These 3 are elements in this structu
    int source, destination, weight;
};

// a structure to represent a connected, directed and weighted graph
struct Graph
{
    int V, E;
    // V is number of vertices and E is number of edges

    struct Edge* edge;
    // This structure contain another structure which we already created edge.
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph));
    //Allocating space to structure graph

    graph->V = V;   //assigning values to structure elements that taken form user.

    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
    //Creating "Edge" type structures inside "Graph" structure, the number of edge type st

    return graph;
}

void FinalSolution(int dist[], int n)
{
    // This function prints the final solution
    printf("\nVertex\tDistance from Source Vertex\n");
    int i;

    for (i = 0; i < n; ++i){
        printf("%d \t\t %d\n", i, dist[i]);
    }
}
```

```c
void BellmanFord(struct Graph* graph, int source)
{
    int V = graph->V;

    int E = graph->E;

    int StoreDistance[V];

    int i,j;

    // This is initial step that we know , we initialize all distance to infinity except s
    // We assign source distance as 0(zero)

    for (i = 0; i < V; i++)
        StoreDistance[i] = INT_MAX;

    StoreDistance[source] = 0;

    //The shortest path of graph that contain V vertices, never contain "V-1" edges. So we
    for (i = 1; i <= V-1; i++)
    {
        for (j = 0; j < E; j++)
        {
            int u = graph->edge[j].source;

            int v = graph->edge[j].destination;

            int weight = graph->edge[j].weight;

            if (StoreDistance[u] + weight < StoreDistance[v])
                StoreDistance[v] = StoreDistance[u] + weight;
        }
    }

    // Actually upto now shortest path found. But BellmanFord checks for negative edge cyc
    // shortest distances if graph doesn't contain negative weight cycle.

    // If we get a shorter path, then there is a negative edge cycle.
    for (i = 0; i < E; i++)
    {
        int u = graph->edge[i].source;

        int v = graph->edge[i].destination;

        int weight = graph->edge[i].weight;

        if (StoreDistance[u] + weight < StoreDistance[v])
            printf("This graph contains negative edge cycle\n");
    }

    FinalSolution(StoreDistance, V);

    return;
}
```

```
int main()
{
    int V,E,S;  //V = no.of Vertices, E = no.of Edges, S is source vertex

    printf("Enter number of vertices in graph\n");
    scanf("%d",&V);

    printf("Enter number of edges in graph\n");
    scanf("%d",&E);

    printf("Enter your source vertex number\n");
    scanf("%d",&S);

    struct Graph* graph = createGraph(V, E);    //calling the function to allocate space t

    int i;
    for(i=0;i<E;i++){
        printf("\nEnter edge %d properties Source, destination, weight respectively\n",i+1
        scanf("%d",&graph->edge[i].source);
        scanf("%d",&graph->edge[i].destination);
        scanf("%d",&graph->edge[i].weight);
    }

    BellmanFord(graph, S);
    //passing created graph and source vertex to BellmanFord Algorithm function

    return 0;
}
```

24.3 Dijkstra算法

- Dijkstra算法解决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均为非负值**，即对于所有的边(u,v)∈E，ω(u,v)≥0，—— 不能有负权重的边和环。

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

算法从结点集V-S中选择当前最短路径估计最小的结点u，将u从Q中删除，并加入到S中，u.d就是源结点s到u的最短路径的长度。这里Q是一个**最小优先队列**，保存结点集V-S。

然后对所有从u出发的边进行松弛。然后重复上述过程，直到Q=Ø。

Dijkstra算法是一个贪心算法：每次总是选择V-S集合中最短路径估计值最小的结点加入S中。

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

每个结点有且仅有一次机会被从Q中抽取并加入S中。一旦u被从Q中抽取出来，u.d就是s到u的最短路径长度（不再改变，上界性质）。

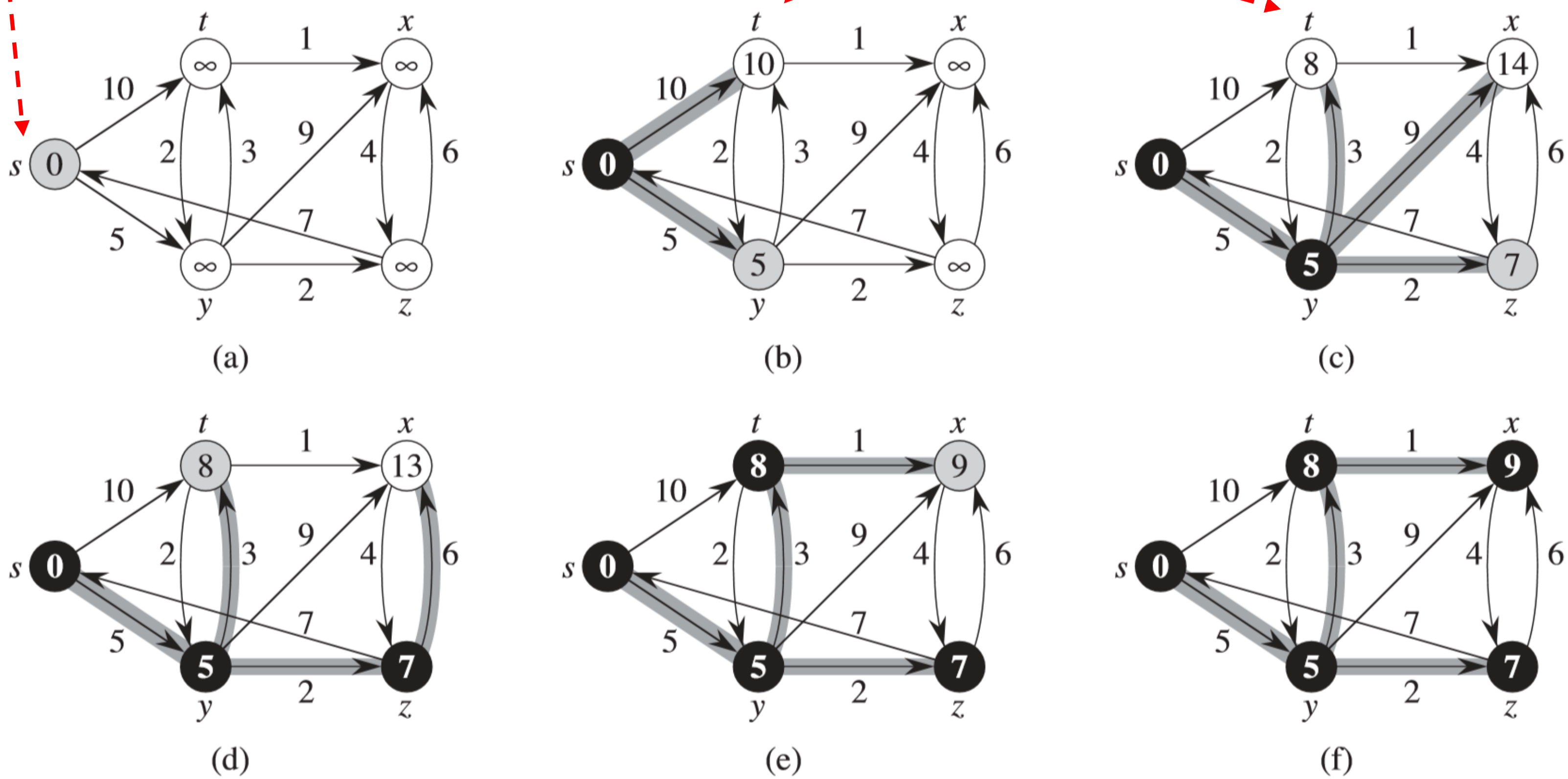
u加入S后，对从u出发的边的(u,v)进行松弛。而如果v.d变小，则是因为存在从s经过u到达v的更短路径所致。此时，修改v.d=u.d +ω(u,v)，v.π=u，即最短路径上v结点的新前驱为u。

while循环总共执行了|V|次

源结点：s

开始的时候s∈S

结点中的数值是s到该结点的最短路径的估计值



例， Dijkstra算法的执行过程

- 加了阴影的边表明前驱值（当前u出发的边）。
- 黑色的结点属于S，白色的结点属于V-S。加阴影的结点是算法下一次循环将选择加入S的点。

定理24.6（Dijkstra算法的正确性）设Dijkstra算法运行在带权重的有向图G=(V,E)上。如果所有边的权重为非负值，则在算法终止时，对于所有结点u∈V，有u.d=δ(s,u)。

证明：利用循环不变式证明

循环不变式：算法在while语句的每次循环开始前，对于每个结点u∈S，有u.d=δ(s,u)

只需证明：对于每个结点u∈V，当u被加入到S时，有u.d=δ(s,u)。

注：一旦u加入S，就不会再修正u.d。且根据上界性质，该等式将一直保持。

证明过程：

（1）初始化：初始时，S=Ø，因此循环不变式直接成立。

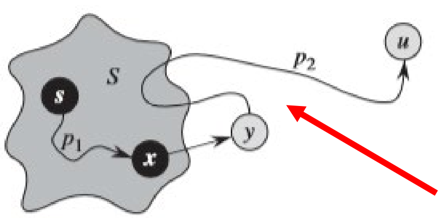
（2）保　持：在每次循环中，对于加入到集合S中的结点u而言，u.d=δ(s,u)。

用反证法证明：**设结点u是第一个在加入到集合S时u.d≠δ(s,u)的结点。**

由于s是第一个加入到集合S中的结点，并且s.d= δ(s,s)=0，所以u≠s，

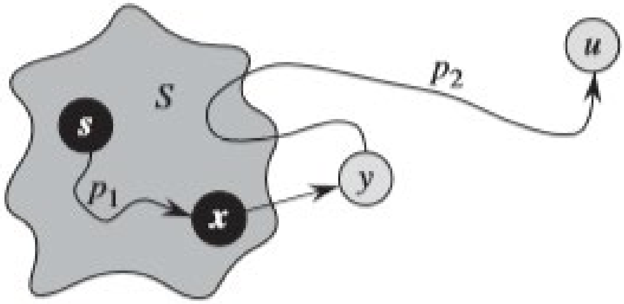
并且在u即将加入S时，S≠Ø，因为S中至少包含了s。

故，此时必存在至少一条从s到u的路径（否则，根据非路径性质将有



一定存在s到u的最短路径p

u.d=δ(s,u) =∞，与假设的u.d≠δ(s,u)相矛盾，故这样路径一定存在），这样也必存在一条从s到u的最短路径，记为p。



考虑路径p上第一个满足y∈V-S的结点y，并设y的前驱是结点x，x∈S，如图所示。路径分为：

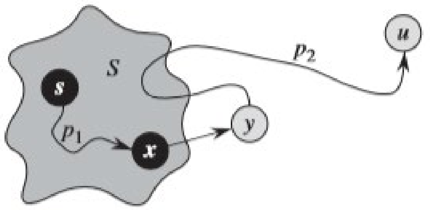
$$s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$$

注：x有可能是s本身，y也有可能是u本身（事实上也只能是u本身，除非δ(y,u)=0）。

则有：**在结点u加入到集合S时，应有y.d=δ(s,y)。**

- 这是因为x∈S，u是第一个u.d≠δ(s,u)的结点，在将x加入到集合S时，有x.d=δ(s,x)，y是x的邻接点，所以此时边(x,y)将被松弛。由于y是最短路径p上的结点，根据最短路径的最优子结构性和收敛性质，此时应有y.d=δ(s,y)。

因为结点y是从结点s到结点u的一条最短路径上位于u前面的一个结点，所以应有δ(s,y)≤ δ(s,u)，因此



$$\begin{aligned} y.d &= \delta(s, y) \\ &\le \delta(s, u) \\ &\le u.d \qquad \text{(by the upper-bound property)} \end{aligned}$$

而在算法第5行选择结点u时，结点u和y都还在集合V-S里，所以有u.d≤y.d。因此上式的不等式事实上只能是等式，即：

$$y.d = \delta(s, y) = \delta(s, u) = u.d \,.$$

这与假设的u.d≠δ(s,u)相矛盾。因此假设不成立。所以，u在加入S时，将有u.d=δ(s,u)，该等式在随后的循环中一直保持。

终止：在算法终止时，Q=Ø，S=V。

根据前面保持性的证明，终止时对于所有的结点u∈V，有 u.d=δ(s,u)。

证毕。

推论24.7 如果在带权重的有向图G=(V,E)上运行Dijkstra算法，其中的权重皆为非负值，源结点为s，则在算法终止时，前驱子图G_π是一棵根结点为s的最短路径树。

从定理24.6和前驱子图性质可证（证明略）。

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra算法运行时间分析

- 根据算法的处理规则，每个结点u仅被加入集合S一次，邻接链表Adj[u]中的每条边在整个运行期间也只被检查一次。因此**算法第7-8行的for循环执行次数总共为|E|次**（即松弛判定总次数）。

- Dijkstra算法的总运行时间依赖于最小优先队列Q的实现。
  - 如果用线性数组(无序或者按序插入)实现，每次找d最小的结点u需要O(V)的时间，所以算法的总运行时间为$O(V^2+E)=O(V^2)$。
  - 如果用二叉堆实现，每次找d最小的结点u需要$O(\lg V)$的时间，所以算法的总运行时间为$O((V+E)\lg V)$。

```c
#include <limits.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d tt %d\n", i, dist[i]);
}
```

```c
// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the shortest
    // distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not
        // yet processed. u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from
            // u to v, and total weight of path from src to  v through u is
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}
```

```c
// driver program to test above function
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;
}
```