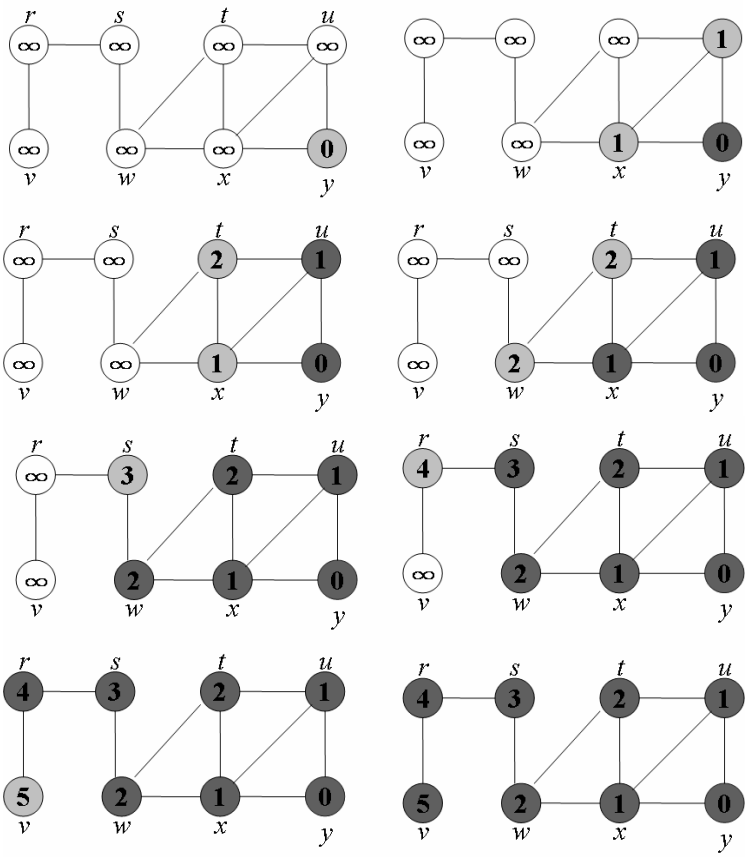
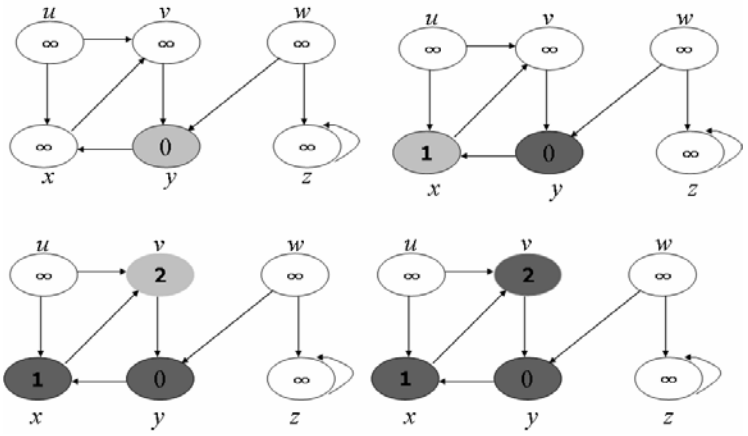
参考答案

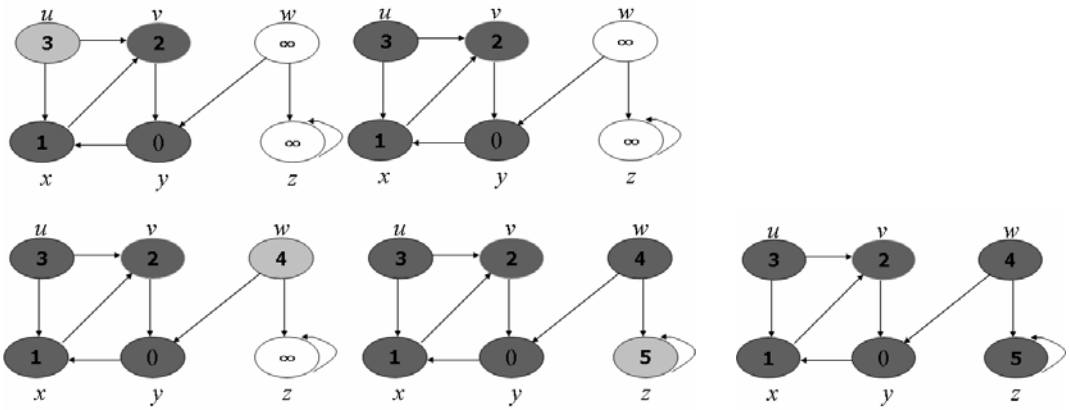
8.1 完成以下练习：

1) 对图8.3(a)从顶点 y 开始，类似图8.3产生一棵宽度优先生成树。



2) 对图8.4(a)从顶点 y 开始，类似图8.4产生一棵深度优先生成树。





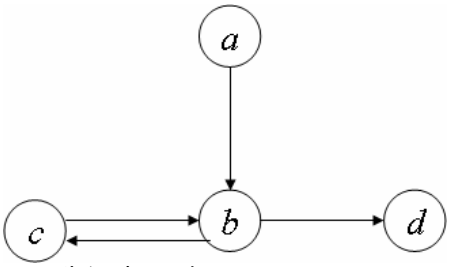
8.2 重写 DFS 算法，用数据结构栈来消除递归。

```
DFSstack(G)
1   for each vertex do
2       colour[u]←White
3   InitStack(s)
4   for each vertex do
5       if colour[u] =White then
6           clcour[u]←Gray
7           push(s,u)
8           while !empty(s) do
9                   pop(s,v)
10                  colour[v]←DarkGray
11                  for each u∈V do
12                      if colour[k]=White then
13                          colour[k]←Gray
14                          push(s,k)
```

8.3

如果用邻接表表示，则只需扫描每个顶点的邻接表，因此所需要的时间为 $O(|V|+|E|)$，如果用邻接矩阵表示，则实现非常简单，但是需要的时间为 $O(|V|^2)$。

8.4



以 a 为源点。则 $d(a)=1, d(b)=2, d(c)=3, d(d)=4, d(c)>d(d)$,且从 c 到 d 存在路径，但是 d 不是 c 的后继。

8.5

深度优先搜索无向图，只产生树边和后向边。注意到无向图不包含回路当且仅当深度优先搜索不产生后向边。利用深度优先搜索，若发现后向边，则说明有回路。

8.6

We have at our disposal an $O(V + E)$-time algorithm that computes strongly connected

components. Let us assume that the output of this algorithm is a mapping $scc[u]$, giving the number of the strongly connected component containing vertex u, for each vertex u. Without loss of generality, assume that $scc[u]$ is an integer in the set $\{1, 2, \ldots, |V|\}$.

Construct the multiset (a set that can contain the same object more than once) $T = \{scc[u] : u \in V\}$, and sort it by using counting sort. Since the values we are sorting are integers in the range 1 to $|V|$, the time to sort is $O(V)$. Go through the sorted multiset T and every time we Þnd an element x that is distinct from the one before it, add x to V_{SCC}. (Consider the Þrst element of the sorted set as .distinct from the one before it..) It takes $O(V)$ time to construct V_{SCC}.

Construct the set of ordered pairs $S = \{(x, y) : \text{there is an edge } (u, v) \in E, x = scc[u], \text{ and } y = scc[v]\}$.

We can easily construct this set in $\_(E)$ time by going through all edges in E and looking up $scc[u]$ and $scc[v]$ for each edge $(u, v) \in E$.

Having constructed S, remove all elements of the form (x, x). Alternatively, when we construct S, do not put an element in S when we Þnd an edge (u, v) for which $scc[u] = scc[v]$. S now has at most $|E|$ elements.

Now sort the elements of S using radix sort. Sort on one component at a time. The order does not matter. In other words, we are performing two passes of counting sort. The time to do so is $O(V + E)$, since the values we are sorting on are integers in the range 1 to $|V|$.

Finally, go through the sorted set S, and every time we Þnd an element (x, y) that is distinct from the element before it (again considering the Þrst element of the sorted set as distinct from the one before it), add (x, y) to E_{SCC}. Sorting and then adding (x, y) only if it is distinct from the element before it ensures that we add (x, y) at most once. It takes $O(E)$ time to go through S in this way, once S has been sorted.

The total time is $O(V + E)$.

8.7

令T是一个$|V| \times |V|$的矩阵，用来表示传递闭包，使得G中存在一条从i到j的路径时，$T[i, j] = 1$，否则$T[i, j] = 0$，初始化T如下

$$T[i, j] = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

当边(u, v)增加到G，T能够更新如下

TRANSITIVE-CLOSURE-UPDATE(u, v)

```
1  for i ← 1 to |V| do
2      for j ← 1 to |V| do
3          if T [i, u] = 1 and T [v, j ] = 1 then
4              T [i, j ] ← 1
```

这表示增加边的效果是产生一条每个已经到达u的顶点到每个可能已经从v可达顶点的一条路径。值得注意的是算法设置$T [u, v] \leftarrow 1$是因为初始值$T [u, u] = T [v, v] = 1$。

由于只有两个for循环，因此时间复杂度为$O(|V|^2)$。

8.8

```
Prim(G, v)     //以为 v 起点
1  for i←0 to |V| do
```

```
2       closest[i]←v //点 i 到已完成顶点集 U 的最小值顶点
3       lowcost[i]←w(v,i)//U 中顶点到 V-U 中顶点的最小值
4   for i←0 to |V|-1 do
5       min←∞
6       for j←0 to |V| do
7             if lowcost[j] !=0 and lowcost[j]<min then
8                   min←lowcost[j]
9                   k←j         // (closest[k],k)是最小值的边
10      lowcost[k]←0
11      for j←0 to |V| do
12          if w(k, j) !=0 and w(k,j)<lowcost[j] then
13              lowcost[j]←w(k,j)
14              closest[j]←k
```

8.9

算法思想：把修改的边加入到最小生成树里，这样在树中就会产生了一条含有修改边的回路。在这个回路中去掉权值最大的，就可以得到一颗新的最小生成树。

8.10 证明 PrimMST 算法的正确性。

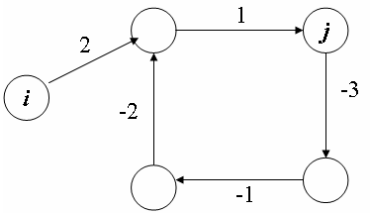
类似书上证明。略。

8.11 是否存在这样的图，对这些图，PrimMST 算法要慢于 KruskalMST 算法？

Prim 算法的时间复杂度为 O(|*E*|lg|*V*|), Kruskal 算法的时间复杂度为 O(|*E*|lg|*E*|),要使 Prim 算法慢于 Kruskal 算法，即要求|*E*|lg|*V*|>|*E*|lg|*E*|，可得 *V*>*E*。又因为这两个算法是用来求图的最小生成树，所以 *E*≥*V*-1。综上所述，当 *E*=*V*-1 时，Prim 算法要慢于 Kruskal 算法。故存在这样的图。

8.12

存在负权回路



8.13 修改 BellmanFood 算法，使得对任意顶点 v，当从源点到 v 的某些路径上存在一个负权回路时，则设置 $d[v] = -\infty$。

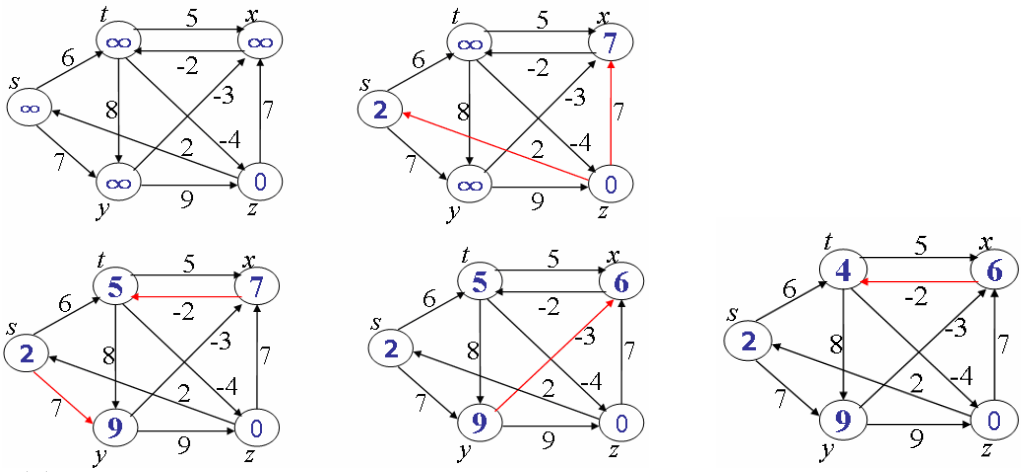
```
BellmanFord(G, w, s)
1   InitializeSingleSource(G, s)
2   for i ← 1 to |V| - 1 do
3       for each edge (u, v)∈ E do
4             Relax(u, v, w)
5   for each edge (u, v) ∈ E do
6       if d[v] > d[u] + w(u, v) then
7             d[v]←-∞
8             return False
9   return True
```

8.14

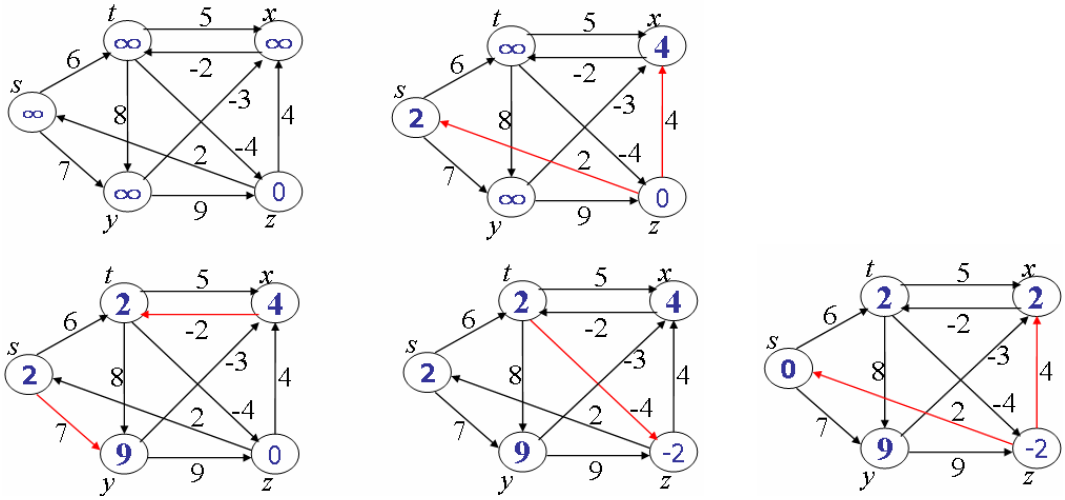
```
Pathnum(G, w)
1   Count←0
2   for each vertex v∈V do
3       BellmanFord(G, w, v)

BellmanFord(G, w, s)
1   InitializeSingleSource(G, s)
2   for i←1 to |V| - 1 do
3       for each edge (u, v)∈E do
4               if d[u]+w(u,v)≠∞ and π[v]≠u then
5                       count←count+1
6                       d[v]←d[u]+w(u,v)
7                       π[v]←u
```

8.15 令顶点 z 为源点，对图 8.10(a)所示的有向图运行 BellmanFord 算法。在每趟运行中，按图中的顺序对边进行松弛，显示每一趟运行后的 d 值和 π 值。现在把边 (z,x) 的权值改为 4 并把 z 作为源点再运行该算法的结果。

(1)



(2)

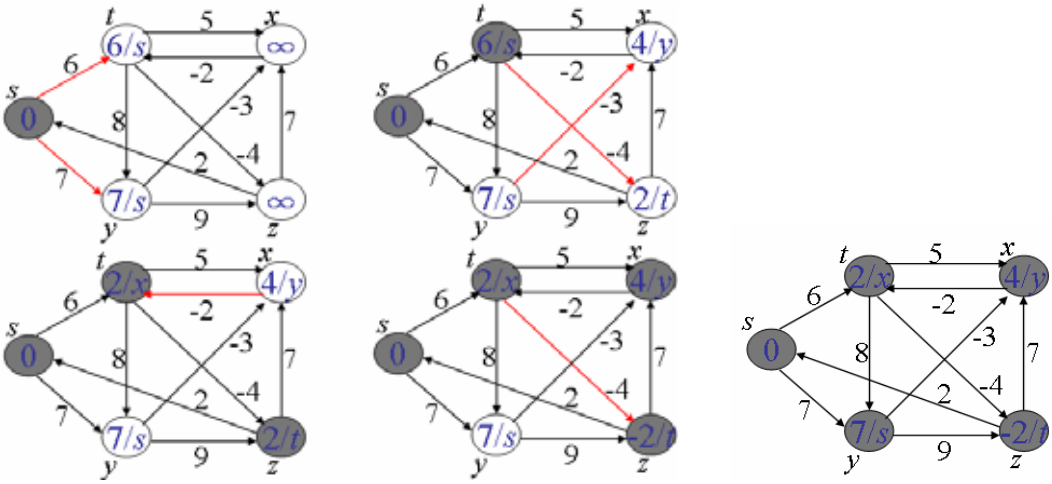


8.16

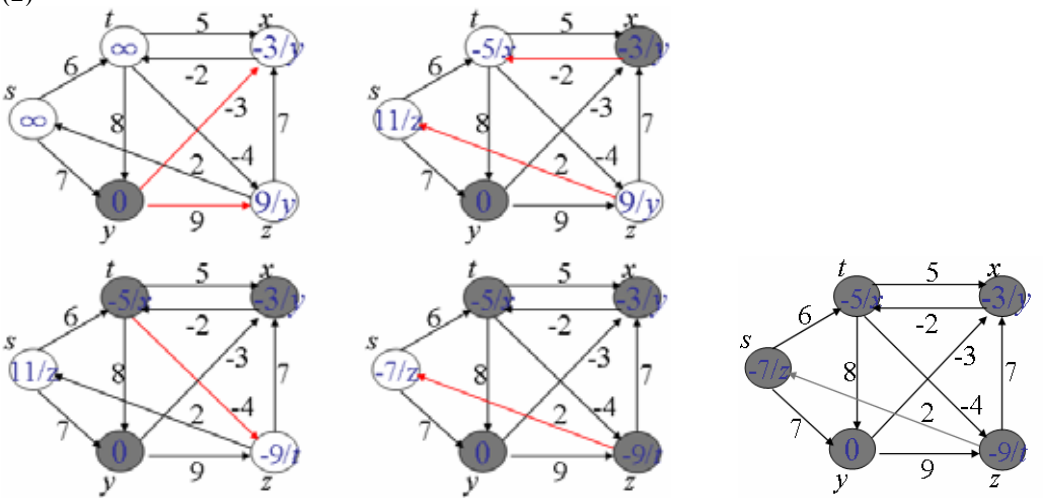
```
BellmanFordM1(G,w, s)
1    InitializeSingleSource(G, s)
2    changes ← True
3    while changes = True do
4              changes ← False
5              for each edge (u, v)∈E do
6                        RelaxM(u, v,w)
RelaxM(u, v,w)
1     if d[v] > d[u] + w(u, v) then
2           d[v]←d[u] + w(u, v)
3           π[v]←u
4           changes←True
```

8.17 令顶点 s 和顶点 y 分别为源点，对图 8.10(a)所示的有向图运行 Dijkstra 算法，类似图 8.11 所示的方式，给出 **while** 循环的每次迭代后的 d 和 π 值以及集合 S 中的顶点。

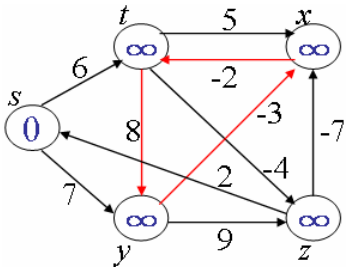
(1) 以 s 为源点(黑色点表示该点属于集合 S，圆圈中列出此时各个点的 d 值和 π值)



(2)



8.18 给出一个带负权边的有向图的简单实例，说明 Dijkstra 算法计算该例子会产生错误的结果。如果允许图中边的权为负，说明定理 8.8 的证明不能成立的原因。



上图中存在一条负权回路，所以从 S 到 Z 不存在最短路径。但是利用 Dijkstra 算法时却能找到一条错误的路径。

定理 8.8 证明不成立的原因是：当图中的边存在负权时，则有可能存在负权回路。

则图中含有该负权回路的所有路径，都没有最短路径。不妨设该路径为从 s 到顶点 u，即 $\delta(s, u) = -\infty$。

但是由于从 s 到顶点 u 存在着路径，又 **while** 循环是有限次的，所以最后得到的 $d(s, u) \neq -\infty$,即 $d(s,u) \neq \delta(s, u)$ ，故定理不成立。

8.19 利用合计方法分析 Dijkstra 算法的时间复杂度。

第一行的初始化部分的运行时间为 O(|*V*|)，**while** 循环的循环体需要执行|*V*|次，循环体中每次 ExtractMin 操作需要 O(lg|*V*|)时间，而对于第 7 至 8 行总运算需要 O(|*E*|)时间，所以算法的总运行时间为 O(|*V*|+|*V*|lg|*V*|+|*E*|)=O(|*V*|lg|*V*|)

8.20 假设将 Dijkstra 算法的第 4 行改为：

4 **while** $|Q|>1$

这一修改使得 **while** 循环执行 $|V|-1$ 次而不是 $|V|$ 次，算法是否仍然正确？

算法仍然正确。令 u 是队列 Q 中剩余的顶点。如果 u 不能从 s 可达，则有 $d[u]=\delta(s,u)=\infty$。如果 u 从 s 可达，则有一条路径 p，它从 s 经过 x 再直接到 u，由于 $d[x]=\delta(s,x)$，按照最短路径松弛性质，可得 $d[u]=\delta(s,u)$。故算法正确。

8.21

只需要修改 Dijkstra 算法，使它能够求解最大化路径上每条边可靠性的乘积问题。具体可以修改如下：

(1) 将队列运算 ExtractMin 改为 ExtractMax

(2) 在松弛运算步，+改为×

例如，对于 Relax 可以修改如下

RelaxReliability*(u, v, r)*

1 **if** $d[v] < d[u] \cdot r(u, v)$ **then**

2 $d[v] \leftarrow d[u] \cdot r(u, v)$

3 $\pi[v] \leftarrow u$

另一种解法是，对任意边 (u,v)，权值修改为 $w(u,v) = \lg r(u,v)$，然后运行 Dijkstra 算法，最可靠路径即为从 s 到 t 的最短路径，路径的可靠性为路径上边可靠性的乘积。

8.22

Observe that if a shortest-path estimate is not ∞, then it's at most $(|V| - 1)W$. Why? In order to have $d[v] < \infty$, we must have relaxed an edge (u, v) with $d[u] < \infty$. By induction, we can show that if we relax (u, v), then $d[v]$ is at most the number of edges on a path from s to v times the maximum edge weight. Since any acyclic path has at most $|V| - 1$ edges and the maximum edge weight is W, we see that $d[v] \le (|V|-1)W$. Note also that $d[v]$ must also be an integer, unless it is ∞.

We also observe that in Dijkstra's algorithm, the values returned by the ExtractMin calls are monotonically increasing over time. Why? After we do our initial $|V|$ Insert operations, we never do another. The only other way that a key value can change is by a DecreaseKey operation. Since edge weights are nonnegative, when we relax an edge *(u, v)*, we have that $d[u] \le d[v]$. Since u is the minimum vertex that we just extracted, we know that any other vertex we extract later has a key value that is at least $d[u]$.When keys are known to be integers in the range 0 to k and the key values extracted are monotonically increasing over time, we can implement a min-priority queue so that any sequence of m Insert, ExtractMin, and DecreaseKey operations takes $O(m + k)$ time. Here.s how. We use an array, say $A[0 . . k]$, where $A[j]$ is a linked list of each element whose key is j . Think of $A[j]$ as a bucket for all elements with key j . We implement each bucket by a circular, doubly linked list with a sentinel, so that we can insert into or delete from each bucket in $O(1)$ time.

We perform the min-priority queue operations as follows:

• Insert: To insert an element with key j , just insert it into the linked list in $A[j]$. Time: $O(1)$ per Insert.

• ExtractMin: We maintain an index *min* of the value of the smallest key extracted. Initially, *min* is 0. To Þnd the smallest key, look in $A[min]$ and, if this list is nonempty, use any element in it, removing the element from the list and returning it to the caller. Otherwise, we rely on the monotonicity property (and that there is no IncreaseKey operation) and increment *min* until we either Þnd a list $A[min]$ that is nonempty (using any element in $A[min]$ as before)or we run off the end of the array A (in which case the min-priority queue is empty).Since there are at most m Insert operations, there are at most m elements in the min-priority queue. We increment *min* at most k times, and we remove and return some element at most m times. Thus, the total time over all ExtractMin operations is $O(m + k)$.

• DecreaseKey: To decrease the key of an element from j to i , Þrst check whether $i \le j$, ßagging an error if not. Otherwise, we remove the element from its list $A[j]$ in $O(1)$ time and insert it into the list $A[i]$ in $O(1)$ time.Time: $O(1)$ per DecreaseKey.

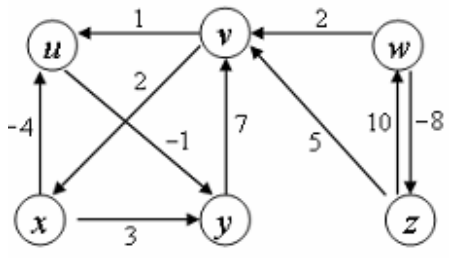
To apply this kind of min-priority queue to Dijkstra's algorithm, we need to let $k = (|V| - 1)W$, and we also need a separate list for keys with value ∞. The number of operations m is $O(|V|+|E|)$ (since there are $|V|$ Insert and $|V|$ ExtractMin operations and at most $|E|$ DecreaseKey operations), and so the total time is $O(|V| + |E| + |V||W|) = O(|V||W| + |E|)$.

8.23 根据递归方程（8.2）和（8.3），写出动态规划算法的伪代码，并分析其时间复杂度。

```
Shortestpaths(G,W)
1   for i=1 to |V| do
2       for j= 1 to |V| do
3           if i=j then  l_ij^0 ← 0
4           else   l_ij^0 ← ∞
5   for m=1 to |V|-1 do
6       for i=1 to |V| do
7           for    j=1 to |V| do
8               min ← ∞
9               for   each k∈adi[j] do
10                  if   l_ik^(m-1) + w_kj < min  then  min ← l_ik^(m-1) + w_kj
11              l_ij^m ← min
```

8.24 修改 FloydWarshall(W)算法，以便构造出最优解。

```
FloydWarshall(W)
1   D^(0) ← W
2   s[1..|V|,1..|V|]←0
```



8.16

```
3    for k ← 1 to |V| do
4        for i ← 1 to |V| do
5            for j ← 1 to |V| do
6                if d_ij^(k-1) < d_ik^(k-1) + d_kj^(k-1) then d_ij^(k) ← d_ij^(k-1)
7                else
8                    d_ij^(k) ← d_ik^(k-1) + d_kj^(k-1)
9                    s[i,j] ← k
10
11   return D^(|V|) and s
```

8.25 给定图 8.16，运行 FloydWarshall 算法，并写出外层循环中每次迭代所生成的矩阵 $D^{(k)}$。

k 为 0 时的矩阵:

0	32767	-1	32767	32767	32767
-4	0	3	32767	32767	32767
32767	32767	0	7	32767	32767
1	2	32767	0	32767	32767
32767	32767	32767	2	0	-8
32767	32767	32767	5	10	0

k 为 1 时的矩阵:

0	32767	-1	32767	32767	32767
-4	0	-5	32763	32763	32763
32767	32767	0	7	32767	32767
1	2	0	0	32767	32767
32767	32767	32766	2	0	-8
32767	32767	32766	5	10	0

k 为 2 时的矩阵

0	32767	-1	32767	32767	32767
-4	0	-5	32763	32763	32763
32763	32767	0	7	32767	32767
-2	2	-3	0	32765	32765
32763	32767	32762	2	0	-8
32763	32767	32762	5	10	0

k 为 3 时的矩阵

0	32766	-1	6	32766	32766
-4	0	-5	2	32762	32762
32763	32767	0	7	32767	32767
-2	2	-3	0	32764	32764
32763	32767	32762	2	0	-8
32763	32767	32762	5	10	0

k 为 4 时的矩阵

0	8	-1	6	32766	32766
-4	0	-5	2	32762	32762
5	9	0	7	32767	32767
-2	2	-3	0	32764	32764
0	4	-1	2	0	-8
3	7	2	5	10	0

k 为 5 时的矩阵

0	8	-1	6	32766	32758
-4	0	-5	2	32762	32754
5	9	0	7	32767	32759
-2	2	-3	0	32764	32756
0	4	-1	2	0	-8
3	7	2	5	10	0

k 为 6 时的矩阵

0	8	-1	6	32766	32758
-4	0	-5	2	32762	32754
5	9	0	7	32767	32759
-2	2	-3	0	32764	32756
-5	-1	-6	-3	0	-8
3	7	2	5	10	0

8.26 由于要计算 $d_{ij}^{(k)}$，$i,j,k=1,2,\cdots,|V|$，所以 FloydWarashall 算法的空间要求为 $O(|V|^3)$。

给出仅仅去掉所有上标所得的算法

```
FloydWarshall(W)
1   D ← W
2   for k ← 1 to |V| do
3       for i ← 1 to |V| do
4           for j ← 1 to |V| do
5               if d_ij < d_ik + d_kj then d_ij ← d_ij
6               else      d_ij ← d_ik + d_kj
7   return D
```

证明该算法是正确的，而且所需空间仅为 $O(|V|^2)$。

当把所有上标去掉时，则原递归方程变为：

$d_{ij} \leftarrow \min(d_{ij},\ d_{ik}+d_{kj})$，计算该算式包含的情况有以下三种：

$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$

$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)}+d_{kj}^{(k)})$

$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)},\ d_{ik}^{(k)}+d_{kj}^{(k-1)})$

如果从节点 i 到 k 存在最短路径，则在这条路径的中间节点中一定不包含 i 或 k。假设有包含，说明存在着回路，当把回路去掉时就可以得到一条更短的路径，与已知的矛盾。所以 $d_{ik}^{(k)}=d_{ik}^{(k-1)}$，$d_{kj}^{(k)}=d_{kj}^{(k-1)}$，所以以上三种情况仍可全部转化为：$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$，即与原递归方程相等。所以可以去掉上标，得到的结果仍是正确的。在空间上由于只要保存 d_{ij} 即可。所以需要的空间为 $O(|V|^2)$

8.27

只要把 FloydWarshall 算法多迭代一次，检查 d 值的变化。如果有负权回路，则某些最短路径的权和会变小。如果没有负权回路，则 d 值不会改变。

8.28

(1)

$$\phi_{ij}^{k}=\begin{cases}0 & other\\ \phi_{ij}^{k-1} & d_{ij}^{k-1}\le d_{ik}^{k-1}+d_{kj}^{k-1}\\ k & d_{ij}^{k-1}>d_{ik}^{k-1}+d_{kj}^{k-1}\end{cases}$$

(2)修改 FloydWarshall 算法类似 8.25.见 8.25

(3) 与矩阵链乘相似之处是都指出了把原问题分成两部分的分割位置。

```
PrintPath(Q,i,j)
1   if Q[i, j]=0 then
2           if w[i, j]≠∞ then   print(i);   print(j);   return
3           else output "there is no path from i to j"; return
4   else
5       PrintPath(Q,i,Q[i, j])
6       PrintPath(Q,Q[i, j], j)
```

实验题

8.29 编程实现求解最短路径问题的类似矩阵乘法的动态规划算法和 FloydWarshall 算法，并用实验分析方法比较两种动态规划算法。

8.30 完成 XOJ 如下题目：1073，1074，1075，1076，1077，1078，1040，1043，1049，1050，1063。

8.31 完成 POJ 如下题目：1062，1094，1125，1158，1161，1178，1181，1364，1639，1665，1679，1860，2394，2421，2553，2728，3009，3026，3259，3660。