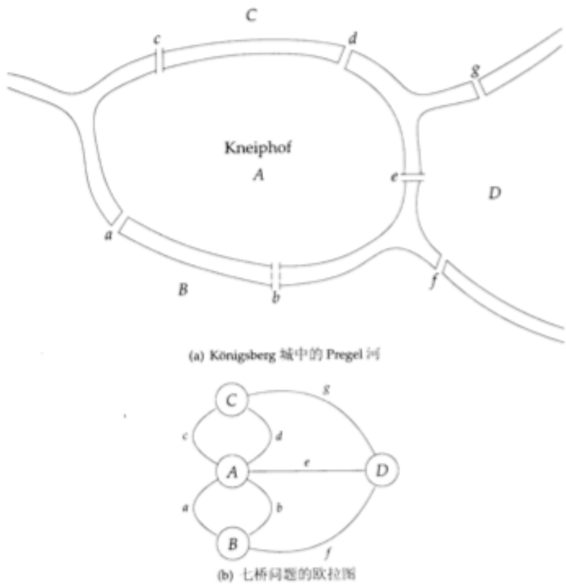
# 图的定义

图论是数学的一个分支，最早可以追溯到1736年，数学家欧拉用图论方法解决了Konigsberg七桥问题，此后七桥问题成为著名的数学经典。Konigsberg城中的Pregel河围绕Kneiphof岛缓缓流过，分成两条支流。Pregel河把Konigsberg城分割成四个区域，四个区域由七座桥连接。




(a) Königsberg 城中的 Pregel 河

(b) 七桥问题的欧拉图

四个区域用$A, B, C, D$标记，七座桥用$a, b, c, d, e, f, g$标记。七桥问题的提法是：从任何一个区域出发，跨过每座桥一次且一次，问最后能否回到出发的那个区域。

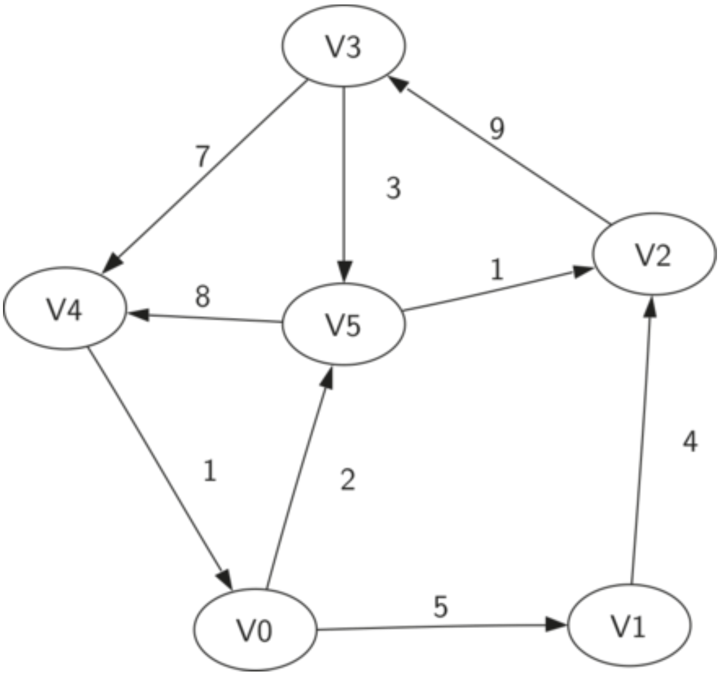
欧拉解决了这一问题，答案是不能。欧拉首先把这个问题描述为抽象的数学对象：图，用图的顶点表示区域，用图的边表示桥梁。图中顶点的度定义为与该顶点邻接的边数，欧拉证明了：如果从图中的一个顶点出发，经过图中所有边一次且仅一次，最后回到出发的顶点，那么当且仅当所有顶点的度都是偶数。后来为了纪念欧拉的发现，这样的回路称为欧拉回路。七桥问题之所以不存在欧拉回路，是因为所有顶点的度均为奇数。

## 图的定义

- 顶点
- 边：一条边连接两个顶点，表示它们之间有关系。边可以有方向，也可以没有方向。
  - 若所有的边都有方向，则称该图为有向图。
  - 若所有的边都无方向，则称该图为无向图。
- 权重：边可能有权重，以表明从一个顶点到另一个顶点的代价。
  - 比如交通图中，一条边的权重可能表示两个城市之间的距离。

图$G$由两个集合$V, E$组成，其中$V$是顶点集合，$E$是边集合，顶点二元组称为边。用$G = (V, E)$表示图。

子图$s$是边$e$和顶点$v$的集合，其中$e \subset E, v \subset V$。




上图是一个带权有向图，该图由六个顶点

$$V=\{v_0,v_1,v_2,v_3,v_4,v_5\}$$

和九条边

$$E=\{(v_0,v_1,5),(v_1，v_2,4),(v_2,v_3,9)(v_3,v_4,7),(v_4，v_0,1),(v_0,v_5,2)(v_5,v_4,8),(v_3，v_5,3),(v_5,v_2,1)\}$$

构成。

- 路径(path):

  图$G$中的顶点序列$u,w_1,w_2,\cdots,w_k,v$称为从$u$到$v$的路径。 该路径有$E$中的边$(u,w_1),(w_1,w_2),\cdots,(w_k,v)$组成。
  - 无权图的路径长度为路径中边的条数，即$n-1$。
  - 带权图的路径长度为路径中边的权重之和。

上图中，路径$v_3\rightarrow v_1$为顶点序列$(v_3,v_4,v_0,v_1)$，路径长度为$7+1+5=13$。

- 简单路径：除起点和终点可以相同，其他顶点都不相同的路径。
- 环路(Cycle)：起点和终点都相同的简单路径。$(v_5,v_2,v_3,v_5)$为环路。

## 图的抽象数据类型

图的抽象数据类型定义如下：

- `Graph()` 创建新的空图
- `addVertex(vert)` 往图中增加图的一个实例
- `addEdge(fromVert, toVert)` 往图中增加一条新的有向边，以连接两个顶点
- `addEdge(fromVert, toVert, weight)` 往图中增加一条新的有向带权边，以连接两个顶点
- `getVertex(vertKey)` 查找图中名为`vertKey`的顶点
- `getVertices()` 返回图中由所有顶点构成的列表
- `in` 若给定顶点在图中，则`vertex in graph`返回True，否则返回False.

# 邻接矩阵

令图$G=(V,E)$有$n\geq 1$顶点，$G$的邻接矩阵是$n\times n$的矩阵，矩阵的每一行和每一列表示一个顶点，第$v$行和第$w$列的元素表示顶点$v$和$w$之间是否存在边。若两个顶点之间有边相连，则称它们是邻接的。第$v$行和第$w$列的元素可能表示边$(v,w)$的权重。

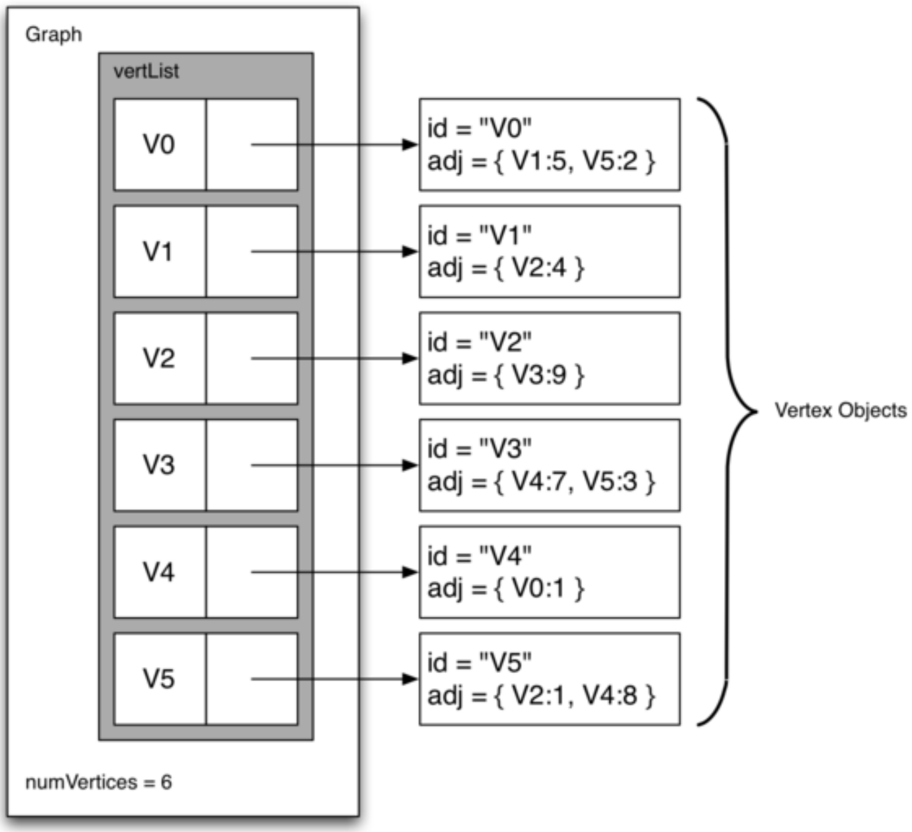
| | V0 | V1 | V2 | V3 | V4 | V5 |
|---|---|---|---|---|---|---|
| V0 | | 5 | | | | 2 |
| V1 | | | 4 | | | |
| V2 | | | | 9 | | |
| V3 | | | | | 7 | 3 |
| V4 | 1 | | | | | |
| V5 | | | 1 | | 8 | |

邻接矩阵的好处是它很简单，对小图来说非常容易看出顶点之间的连接关系。

上图的邻接矩阵中，有大量元素都为零，即为稀疏矩阵。而存储稀疏数据采用矩阵并不是一种有效的方式。

# 邻接表

实现稀疏连接图的一种有效方式是采用邻接表。




邻接表的好处是可以紧凑的表示稀疏图，同时还可以容易地找到与某个顶点直接相邻关系。

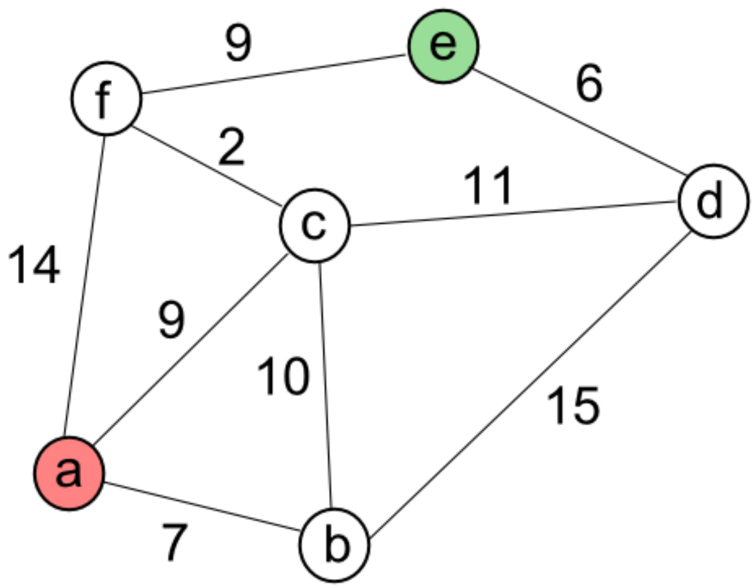
# 实现

In [244]:
```
class Vertex:
    def __init__(self, node):
        self.id = node
        self.adjacent = {}
        self.distance = 999
        self.visited = False
        self.previous = None

    def addNeighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def getConnections(self):
        return self.adjacent.keys()

    def getId(self):
        return self.id

    def getWeight(self, neighbor):
        return self.adjacent[neighbor]

    def setDistance(self, dist):
        self.distance = dist

    def getDistance(self):
        return self.distance

    def setPrevious(self, prev):
        self.previous = prev

    def setVisited(self):
        self.visited = True

    def __str__(self):
        return str(self.id) + ' adjacent: ' + str([x.id for x in self.adjacent]
)
```

In [245]:
```
class Graph:
    def __init__(self):
        self.vertDict = {}
        self.numVertices = 0

    def __iter__(self):
        return iter(self.vertDict.values())

    def addVertex(self, node):
        self.num_vertices = self.numVertices + 1
        newVertex = Vertex(node)
        self.vertDict[node] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertDict:
            return self.vertDict[n]
        else:
            return None

    def addEdge(self, frm, to, cost = 0):
        if frm not in self.vertDict:
            self.addVertex(frm)
        if to not in self.vertDict:
            self.addVertex(to)

        self.vertDict[frm].addNeighbor(self.vertDict[to], cost)
#         self.vertDict[to].addNeighbor(self.vertDict[frm], cost)

    def getVertices(self):
        return self.vert_dict.keys()

    def setPrevious(self, current):
        self.previous = current

    def getPrevious(self):
        return self.previous
```

In [246]:
```
g = Graph()
for i in range(6):
    g.addVertex(i)

g.addEdge(0, 1, 5)
g.addEdge(0, 5, 2)
g.addEdge(1, 2, 4)
g.addEdge(2, 3, 9)
g.addEdge(3, 4, 7)
g.addEdge(3, 5, 3)
g.addEdge(4, 0, 1)
g.addEdge(5, 4, 8)
g.addEdge(5, 2, 1)

for v in g:
    print(v)
```

```
0 adjacent: [1, 5]
1 adjacent: [2]
2 adjacent: [3]
3 adjacent: [5, 4]
4 adjacent: [0]
5 adjacent: [2, 4]
```




In [254]:
```
g = Graph()

g.addVertex('a')
g.addVertex('b')
g.addVertex('c')
g.addVertex('d')
g.addVertex('e')
g.addVertex('f')

g.addEdge('a', 'b', 7)
g.addEdge('a', 'c', 9)
g.addEdge('a', 'f', 14)
g.addEdge('b', 'c', 10)
g.addEdge('c', 'f', 2)
g.addEdge('d', 'b', 15)
g.addEdge('d', 'c', 11)
g.addEdge('e', 'd', 6)
g.addEdge('f', 'e', 9)


print(g.getVertex('a').getWeight(g.getVertex('b')))
```

```
7
```

# 图的基本操作

给定图$G=(V,E)$，从顶点$v$出发，要达到其他顶点，有两种办法：

- 广度优先搜索
- 深度优先搜索

# 广度优先算法

广度优先搜索的过程如下：

先访问顶点$v$，并把它标记为已访问，然后访问$v$的邻接表中的所有顶点，这些顶点访问之后，接着访问这个邻接表第一个顶点的邻接表。

为实现广度优先搜索，每次将当前顶点入队列保存，在处理完一个邻接表之后，出列一个顶点，然后处理这个顶点的邻接表，表中每个顶点如果未访问则访问之后入队列，已访问过的顶点忽略，直到队列空为止。

In [142]:
```
class Queue():
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

In [189]:
```
def bfs(aGraph, start):
    vertQueue = Queue()
    start.visited = True
    print(start.getId())
    vertQueue.enqueue(start)
    while vertQueue.size() > 0:
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if not nbr.visited:
                print(nbr.getId())
                vertQueue.enqueue(nbr)
                nbr.visited = True
for v in g:
    v.visited = False
start = g.vert_dict[0]
bfs(g, start)
```

```
0
1
4
5
2
3
```

# 深度优先搜索

深度优先搜索的过程是：

先访问$v$，然后在所有$v$的邻接表中选择一个顶点$w$，接着进行深度优先搜索。

为了记录搜索过程的当前位置，当前的访问顶点$v$入栈保存。搜索过程中如果遇到一个顶点$u$，它的邻接表中不再有未被访问的顶点，则从栈中弹出一个顶点，如果该顶点已被访问，那么忽略，否则访问这个顶点并把这个顶点入栈。搜索在栈空时结束

该搜索过程看似复杂，其实可以简单地递归实现。

In [257]:
```
def dfs(aGraph, start):
    start.visited = True
    print(start.getId())
    for node in start.getConnections():
        if not node.visited:
            dfs(aGraph, node)

for v in g:
    print(v)
    v.visited = False
start = g.vertDict['a']
dfs(g, start)
```

```
f adjacent: ['e']
a adjacent: ['c', 'b', 'f']
c adjacent: ['f']
d adjacent: ['c', 'b']
e adjacent: ['d']
b adjacent: ['c']
a
c
f
e
d
b
```

# Dijkstra算法：单源点至所有其它顶点（边权值非负）

给定有向图$G=(V,E)$，每条边$e$的权值函数$w(e)>0$，以及源点$v_0$，找出由源点$v_0$到图中其它所有顶点的最短路径。

要找出上图列出的最短路径，可用贪婪算法(Greedy alogrithom)。

用$S$记录已找到最短路径的顶点（包括$v_0$），对不在$S$中的顶点$w$，令 `distance[w]` 表示从$v_0$开始，途径$S$中的顶点到$w$的最短路径长度。有以下观察：

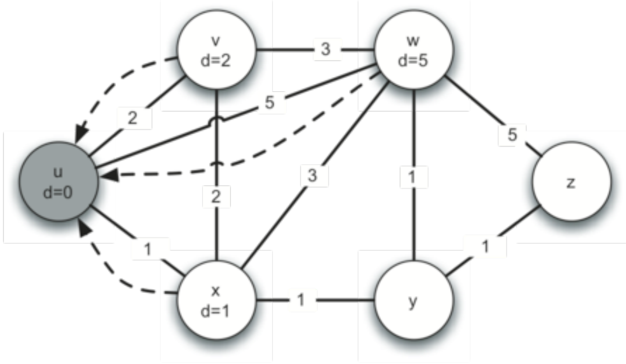
- 如果下一条最短路径将达到$u$，那么从$v_0$到$u$的最短路径只能途经$S$中的顶点。
- 由 `distance` 的定义可知，顶点$u$是所有当前不在$S$中而相距$S$最近的顶点，即`distant[u]`最小。如果有多于一个这样的顶点，则从中人选一个。
- 一旦选定$u$，则$u$归入$S$中。$u$的加入，有可能减少从$v_0$出发，途经$S$中顶点，到达目前还不在$S$中的某顶点$w$的路径长度。如果有这样路径的话，则该路径一定经过$u$。因此，应将原路径$v_0\to w$修改为$v_0\to u\to w$，即由$v_0$途经$S$中的顶点到$u$，然后直接到$w$，并将$w$的原路径长度 `distance[w]` 修改为 `distance[u]+length(<u, w>)`

为了跟踪从源点到每个终点的总成本，我们将使用顶点类中的 `distance` 实例变量。

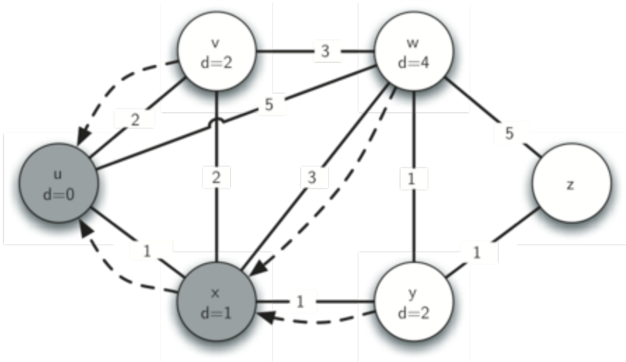
`distance` 包含从源点到终点的最小权重路径的当前总权重。

该算法遍历图中的每个顶点；在顶点上迭代的顺序由优先级队列控制。用于确定优先级队列中对象顺序的值为 `distance`。

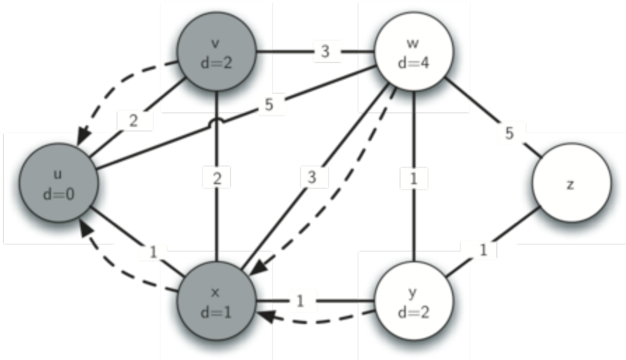
首次创建顶点时，`distance` 被设置为非常大的数。



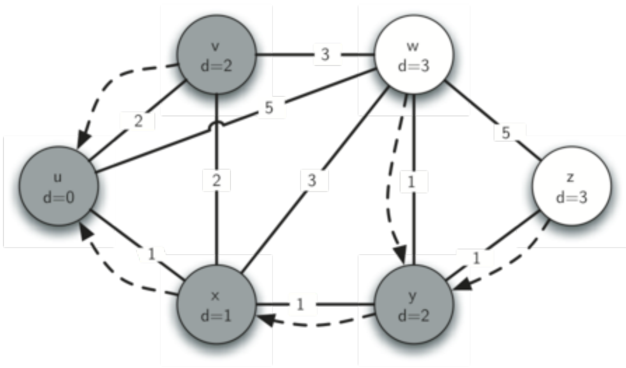

PQ = x,v,w



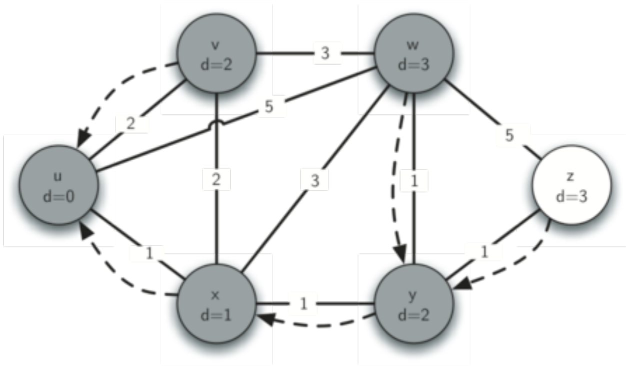

PQ = vy,w



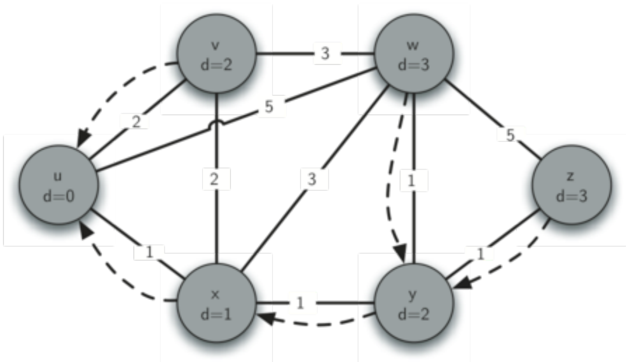

PQ = yw




PQ = wz




PQ = z




PQ = None

In [219]:
```
class BinHeap():
    def __init__(self):
        self.heapList = [(0,0)]
        self.currentSize = 0

    def percUp(self,i):
        while i // 2 > 0:
            if self.heapList[i][0] < self.heapList[i // 2][0]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2

    def insert(self,k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def percDown(self,i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i][0] > self.heapList[mc][0]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self,i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i*2][0] < self.heapList[i*2+1][0]:
                return i * 2
            else:
                return i * 2 + 1

    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [(0,0)] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1
```

In [237]:
```
def dijkstra(aGraph, start):
    print ('''Dijkstra's shortest path''')
    # Set the distance for the start node to zero
    start.setDistance(0)

    # Put tuple pair into the priority queue
    pq = BinHeap()
    unvisited_queue = [(v.getDistance(),v) for v in aGraph]
    pq.buildHeap(unvisited_queue)

    while pq.currentSize > 0:
        # Pops a vertex with the smallest distance
        uv = pq.delMin()
        current = uv[1]
        current.setVisited()

        #for next in v.adjacent:
        for next in current.adjacent:
            # if visited, skip
            if next.visited:
                continue
            new_dist = current.getDistance()
            + current.getWeight(next)

            if new_dist < next.getDistance():
                next.setDistance(new_dist)
                next.setPrevious(current)
                print ('updated : current = %s next = %s new_dist = %s' \
                        %(current.getId(), next.getId(), next.getDistance()))
            else:
                print ('not updated : current = %s next = %s new_dist = %s' \
                        %(current.getId(), next.getId(), next.getDistance()))

        # Rebuild heap
        # 1. Pop every item
        while pq.currentSize > 0:
            pq.delMin()
        # 2. Put all vertices not visited into the queue
        unvisited_queue = [(v.getDistance(),v) for v in aGraph if not v.visited
]
        pq.buildHeap(unvisited_queue)
```

In [241]:
```
def shortest(v, path):
    ''' make shortest path from v.previous'''
    if v.previous:
        path.append(v.previous.getId())
        shortest(v.previous, path)
    return
```

In [242]:
```
if __name__ == '__main__':

    g = Graph()

    g.addVertex('a')
    g.addVertex('b')
    g.addVertex('c')
    g.addVertex('d')
    g.addVertex('e')
    g.addVertex('f')

    g.addEdge('a', 'b', 7)
    g.addEdge('a', 'c', 9)
    g.addEdge('a', 'f', 14)
    g.addEdge('b', 'c', 10)
    g.addEdge('b', 'd', 15)
    g.addEdge('c', 'd', 11)
    g.addEdge('c', 'f', 2)
    g.addEdge('d', 'e', 6)
    g.addEdge('e', 'f', 9)

    print('Graph data:')
    for v in g:
        for w in v.getConnections():
            vid = v.getId()
            wid = w.getId()
            print ('( %s , %s, %3d)'  % ( vid, wid, v.getWeight(w)))

    dijkstra(g, g.getVertex('a'), g.getVertex('e'))

    target = g.getVertex('e')
    path = [target.getId()]
    shortest(target, path)
    print ('The shortest path : %s' %(path[::-1]))
```

```
Graph data:
( f , e,   9)
( f , c,   2)
( f , a,  14)
( a , c,   9)
( a , b,   7)
( a , f,  14)
( c , f,   2)
( c , a,   9)
( c , d,  11)
( c , b,  10)
( d , e,   6)
( d , c,  11)
( d , b,  15)
( e , f,   9)
( e , d,   6)
( b , c,  10)
( b , a,   7)
( b , d,  15)
Dijkstra's shortest path
updated : current = a next = c new_dist = 9
updated : current = a next = b new_dist = 7
updated : current = a next = f new_dist = 14
not updated : current = b next = c new_dist = 9
updated : current = b next = d new_dist = 22
updated : current = c next = f new_dist = 11
updated : current = c next = d new_dist = 20
updated : current = f next = e new_dist = 20
not updated : current = d next = e new_dist = 20
The shortest path : ['a', 'c', 'f', 'e']
```

需要注意的是，Dijkstra的算法只有当权重都是正数时才起作用。如果你在图的边引入一个负权重，算法永远不会退出。