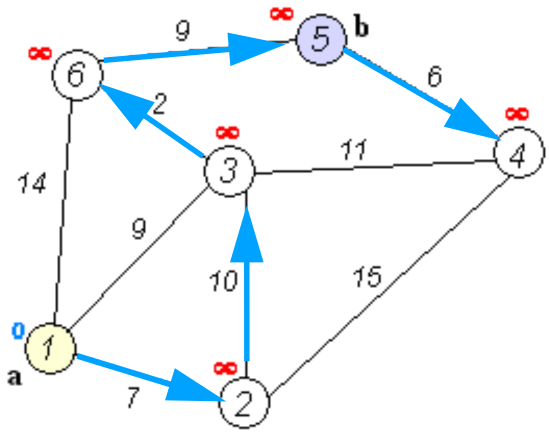
Dijkstra 演算法

原理

Dijkstra使用了廣度優先搜尋解決從源點到其餘各頂點的最短路徑，也就是有向圖的單源最短路徑問題。
首先以某一節點出發，在其相鄰的節點中選擇尚未被存取且cost最小的節點，因為新增的節點cost為最小值，所以每次新增完都要更新到達其他節點的最小距離，直到所有節點都被加入為止。

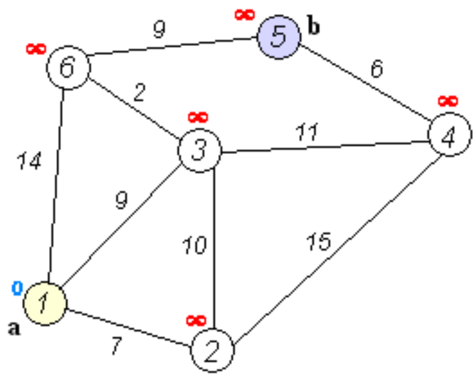


流程圖

Step1

	1	2	3	4	5	6
cost	-	-	-	-	-	-

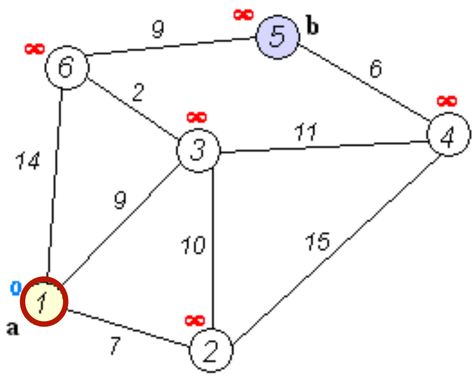
- 代表目前還無法到達，所以距離為無限大。



Step2

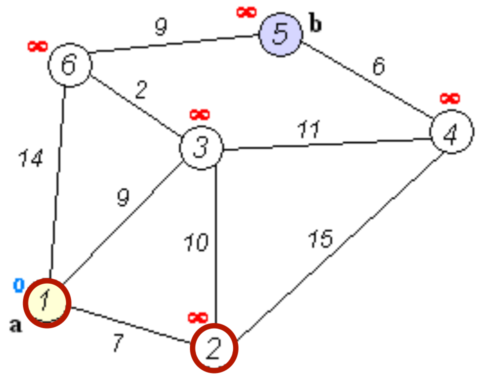
	1	2	3	4	5	6
cost	0	7	9	-	-	14

黃色代表已存取
新增1，與1相鄰的節點有2、3、6。
1與1的距離為0，新增0
1與2的距離為7，新增7
1與3的距離為9，新增9
1與6的距離為14，新增14
距離1最短的點為2



Step3

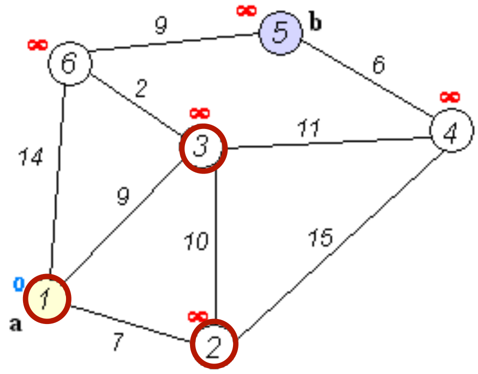
	1	2	3	4	5	6
cost	0	7	9	22	-	14



新增2，與2相鄰的節點有3、4。
透過2，1與3的距離為17，保留9
透過2，1與4的距離為22，新增22
距離1最短的點為3

Step4

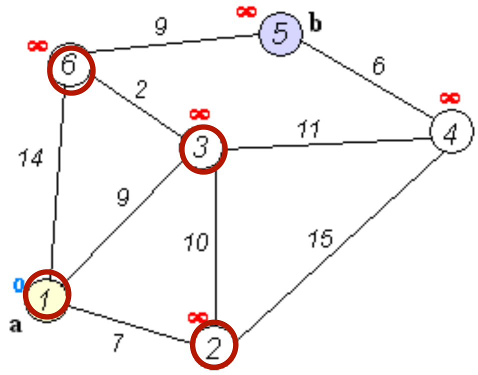
	1	2	3	4	5	6
cost	0	7	9	20	-	11



新增3，與3相鄰的節點有4、6。
透過3，1與4的距離為9+11=20，更新20
透過3，1與6的距離為9+2=11，更新11
距離1最短的點為6

Step5

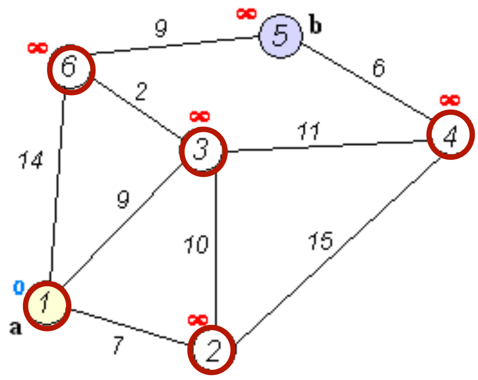
	1	2	3	4	5	6
cost	0	7	9	20	23	11



新增6，與6相鄰的節點有5。
透過6，1與5的距離為14+9=23，新增23
距離1最短的點為4

Step6

	1	2	3	4	5	6
cost	0	7	9	20	23	11



新增4，與4相鄰的節點有5。
透過4，1與5的距離為20+6=26，保留23

Step7

	1	2	3	4	5	6
cost	0	7	9	20	23	11

加入5後，所有節點已存取完，所有節點與1的最小距離為上表。

Dijkstra學習歷程

一開始覺得就跟BFS差不多，就先建一些list來暫存節點的狀態。

```python
from collections import defaultdict

#Class to represent a graph
class Graph():

    def __init__(self, vertices):
        self.v = vertices
        self.graph = []
        self.graph_matrix = [[0 for column in range(vertices)]
                    for row in range(vertices)]

    def addEdge(self,u,v,w):
        self.graph.append([u,v,w]) # u:head v:next w:cost

    def Dijkstra(self, s):
        visited = [False] * self.v #紀錄未走訪的點
        dist = [99999] * self.v #預設所有邊為無限大
        dist[s] = 0
        for v in self.graph[i]:
            if 0 < v < min and visited[v] == False:
                dist[v] = v
                visited[v] = True
                print(v,dist[v])
```

＊self.graph[v]跟range(self.v)的資料型態不一樣，使用int比大小的話必須用後者。

＊用for迴圈挑選距離母節點最短距離的時候，沒辦法即時跟上一次的結果做比較，所以把找尋最短距離拉出來重新定義。

```python
    def Dijkstra(self, s):
        visited = [False] * self.v #紀錄未走訪的點
        dist = [99999] * self.v #預設所有邊為無限大
        dist[s] = 0
        for i in range(self.v):
            m = self.find_min(dist,visited,i)
            for r in range(self.v):
                if self.graph[i][r] > 0 and  visited[r] == False and dist[r] > dist[i] + self.graph[i][r]:
                    dist[r] = dist[i] + self.graph[i][r]
        for node in range(self.v):
            print (node, ":", dist[node])


    def find_min(self,dist,visited,root):
        min = 99999
        for v in range(self.v):
            if dist[v]<min and visited[v] == False:
                min = dist[v]
                visited[v] = True
                return v
```

＊def find_min 來尋找最短edge，return出來跟dist裡的暫時最短路徑比較。

]:

```python
g.Dijkstra(0)
```

```
0 : 0
1 : 4
2 : 12
3 : 19
4 : 28
5 : 16
6 : 18
7 : 8
8 : 14
```

n [199]:

```python
g = Graph(9)
g.graph = [[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]
        ];

print("Dijkstra", g.Dijkstra(0))
```

```
0 : 0
1 : 4
2 : 12
3 : 19
4 : 26
5 : 16
6 : 18
7 : 8
8 : 14
Dijkstra None
```

＊有些成功有些失敗。發現return的位置在回圈裡，將他移出回圈。4的cost有減少一點，但還不是最短路徑。

＊我後來發現紅框裡我一開始因為在迴圈裡所以就直覺的設self.graph[i][r]當成邊，但實際上應該用從find_min找來的變數才對，所以我用m把i取代掉就成功了。
這樣的話，我的find_min裡的root參數其實可以省略。

```python
def Dijkstra(self, s):
    visited = [False] * self.v #紀錄未走訪的點
    dist = [99999] * self.v #預設所有邊為無限大
    dist[s] = 0
    for i in range(self.v):
        m = self.find_min(dist,visited,i)
        visited[m] = True
        for r in range(self.v):
            if self.graph[m][r] > 0 and  visited[r] == False and dist[r] > dist[m] + self.graph[m][r]:
                dist[r] = dist[m] + self.graph[m][r]
    for node in range(self.v):
        print (node, ":", dist[node])


def find_min(self,dist,visited,root):
    min = 99999
    for v in range(self.v):
        if dist[v]<min and visited[v] == False:
            min = dist[v]
            min_index = v
    return min_index
```

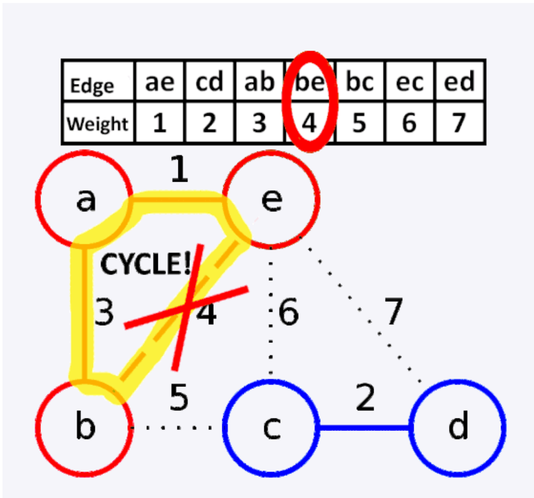
```python
        [0, 0, 4, 14, 10, 0, 2, 0, 0],
        [0, 0, 0, 0, 0, 2, 0, 1, 6],
        [8, 11, 0, 0, 0, 0, 1, 0, 7],
        [0, 0, 2, 0, 0, 0, 6, 7, 0]
        ];

print("Dijkstra", g.Dijkstra(0))
```

```
0 : 0
1 : 4
2 : 12
3 : 19
4 : 21
5 : 11
6 : 9
7 : 8
8 : 14
Dijkstra None
```

Kruskal 演算法

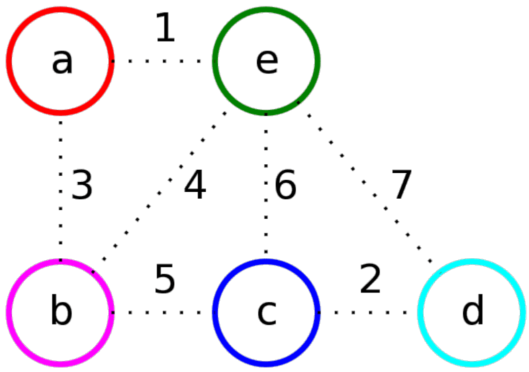
原理

Kruskal演算法是一種尋找最小生成樹的演算法。將原圖中所有的邊按照權值從小到大排序，從值最小的邊開始添加，總共會選取n–1條邊，每條邊在選取的同時，都要確定是連接兩個不同的連通分量的權值最小的邊。
需要注意的是：若這條邊的兩個節點已經在同一條分量的話，就會產生一個Loop，對最小生成樹T1之中的某一環C，假設e屬於最小生成樹T1，那麼將e刪去將會使得T1變為兩個樹。因為環C必然還存在另一橫切邊 f 可以連接兩個子樹形成生成樹T2，這樣就對最小生成樹T1矛盾了。
其平均時間複雜度為O(| E | log | V |)，其中E和V分別是圖的邊集和點集。



流程圖

STEP1:



Edge	ae	cd	ab	be	bc	ec	ed
Cost	1	2	3	4	5	6	7

Disjoint set:

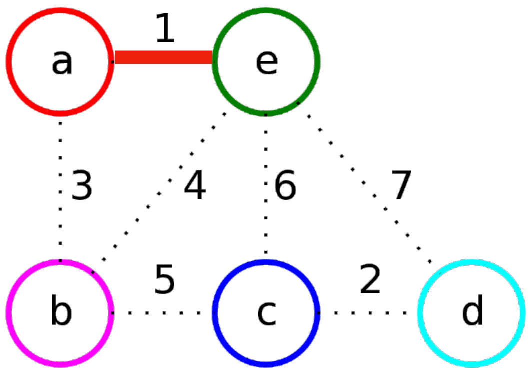
a	b	c	d	E
-1	-1	-1	-1	-1

STEP2:加入ae邊

Edge	ae	cd	ab	be	bc	ec	ed
Cost	1	2	3	4	5	6	7

Disjoint set:

a	b	c	d	e
-1	-1	-1	-1	a

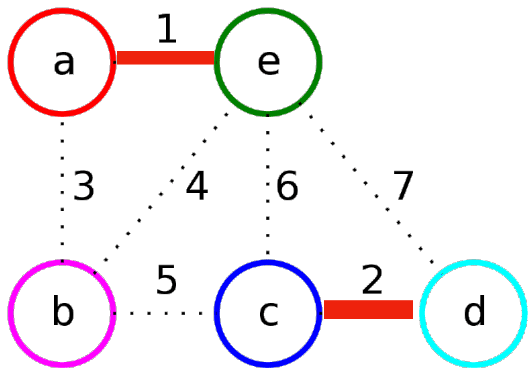


STEP3:加入cd邊

Edge	ae	cd	ab	be	bc	ec	ed
Cost	1	2	3	4	5	6	7

Disjoint set:

a	b	c	d	e
-1	-1	-1	c	a

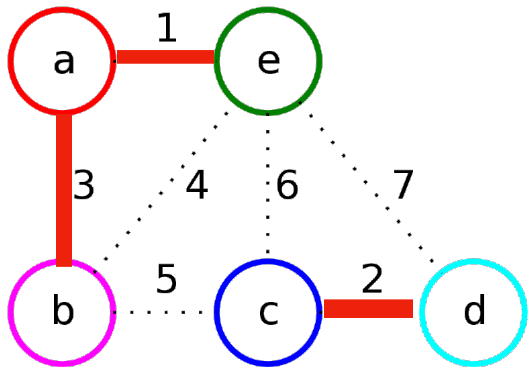


STEP4:加入ab邊

Edge	ae	cd	ab	be	bc	ec	ed
Cost	1	2	3	4	5	6	7

Disjoint set:

a	b	c	d	e
-1	a	-1	c	a

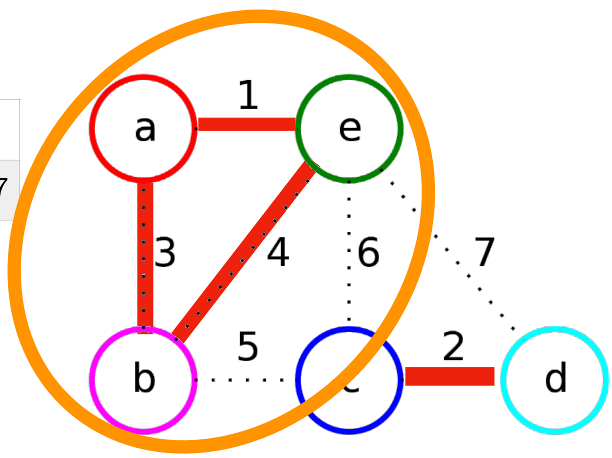


STEP5:加入be邊會形成一個Loop，故跳過

Edge	ae	cd	ab	be	bc	ec	ed
Cost	1	2	3	4	5	6	7

Disjoint set:

a	b	c	d	e
-1	a	-1	c	a

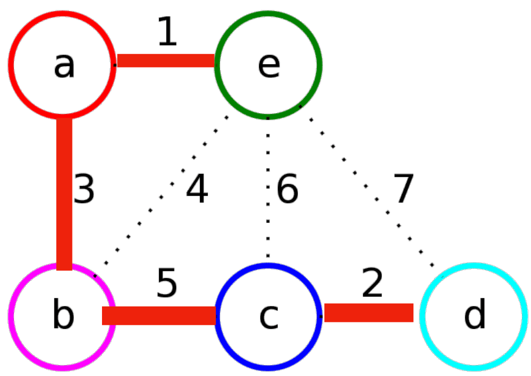


STEP6:加入bc邊

Edge	ae	cd	ab	be	bc	ec	ed
Cost	1	2	3	4	5	6	7

Disjoint set:

a	b	c	d	e
-1	a	a	a	a



當所有節點都被存取，或邊的數量等於節點的數量-1時，代表最小生成樹已完成。總路徑為1+2+3+5=11。

學習歷程

其實dijkstra比較麻煩，kruskal就照著邏輯打一打就出來了。

*set集合不能直接拿來跟list比較
*t.issubset(T)：t是否為T的子集

In [180]:
```python
from collections import defaultdict

#Class to represent a graph
class Graph():

    def __init__(self, vertices):
        self.v = vertices
        self.graph = []
        self.graph_matrix = [[0 for column in range(vertices)]
                    for row in range(vertices)]

    def addEdge(self,u,v,w):
        self.graph.append([u,v,w]) # u:head v:next w:cost

    def Kruskal(self):
        h=0
        subset=set() #檢查是否迴圈用
        MST=[]
        self.graph =  sorted(self.graph,key=lambda item: item[2]) #排序

        for m in self.graph:
            t = set(m[:2]) #轉成set
            if t.issubset(subset) == False:
                subset.update(m[:2])
                MST.append(m)

        while h < len(MST):
            print(MST[h][0],"–",MST[h][1],":",MST[h][2])
            h = h+1
```

In [181]:
```python
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

print("Kruskal",g.Kruskal() )
```

```
2 – 3 : 4
0 – 3 : 5
0 – 1 : 10
Kruskal None
```


參考資料：
https://zh.wikipedia.org/wiki/戴克斯特拉算法
https://zh.wikipedia.org/wiki/克魯斯克尔演算法
http://nthucad.cs.nthu.edu.tw/~yyliu/personal/nou/04ds/dijkstra.html
https://www.itread01.com/content/1549051928.html
https://blog.csdn.net/business122/article/details/7541486