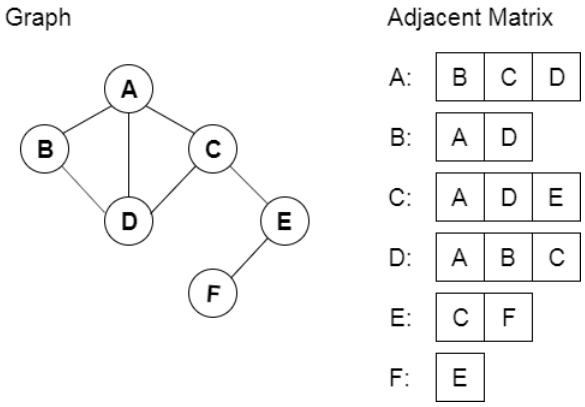
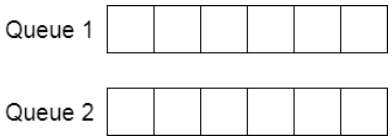
一、 BFS 與 DFS 流程圖

BFS 流程圖



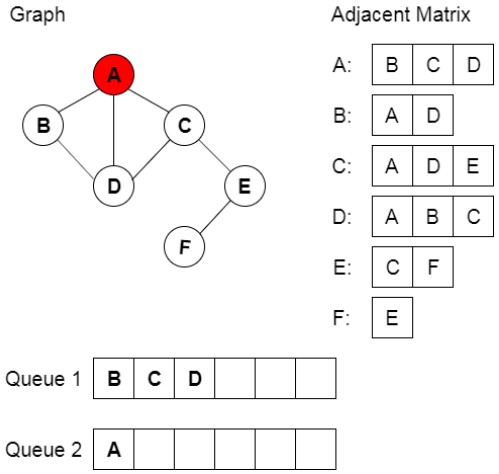
Step1:

1. 建立兩個 Queue。



Step2:

1. 選擇 A 作為起始點，並做上一個已拜訪過的記號，並將 A 放進 Queue2 中。
2. 將所有與 A 相連的點放進 Queue1 中，放入的順序需照 Adjacent Matrix 中與 A 相連的點的順序放進 Queue1 中。

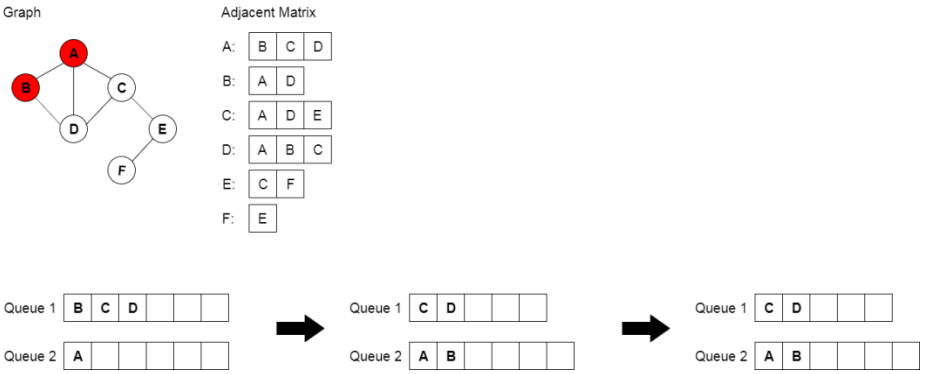


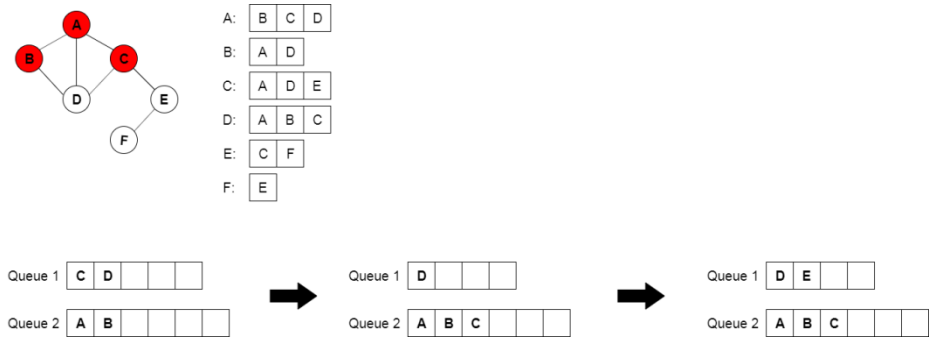
Step3:

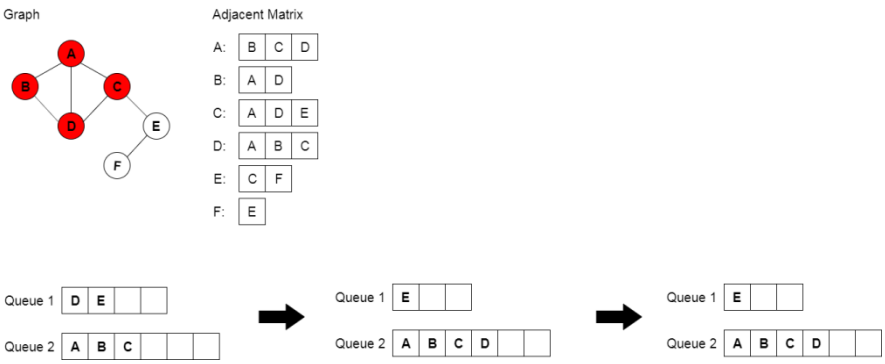
依序選擇與前一個點相鄰的所有點作為下一個拜訪的點。然而需按照順序選取，上一個與 A 相鄰全部的點落在 Queue1 中，依照 Queue 先進先出的原則，應先選

擇 B 作為下一個拜訪的點，然後是 C，再來是 D。

1. 選擇 B 作為拜訪點，並做上已拜訪過的記號，並將 B 放進 Queue2 中。
2. 將所有與 B 相連的點放進 Queue1 中，放入的順序需照 Adjacent Matrix 中與 B 相連的點的順序放進 Queue1 中，如果該相鄰的點已做上記號或已放進 Queue1 中則不選取。
3. 選擇 C 作為拜訪點，並做上已拜訪過的記號，並將 C 放進 Queue2 中。
4. 將所有與 C 相連的點放進 Queue1 中，放入的順序需照 Adjacent Matrix 中與 C 相連的點的順序放進 Queue1 中，如果該相鄰的點已做上記號或已放進 Queue1 中則不選取。
5. 選擇 D 作為拜訪點，並做上已拜訪過的記號，並將 D 放進 Queue2 中。
6. 將所有與 D 相連的點放進 Queue1 中，放入的順序需照 Adjacent Matrix 中與 D 相連的點的順序放進 Queue1 中，如果該相鄰的點已做上記號或已放進 Queue1 中則不選取。



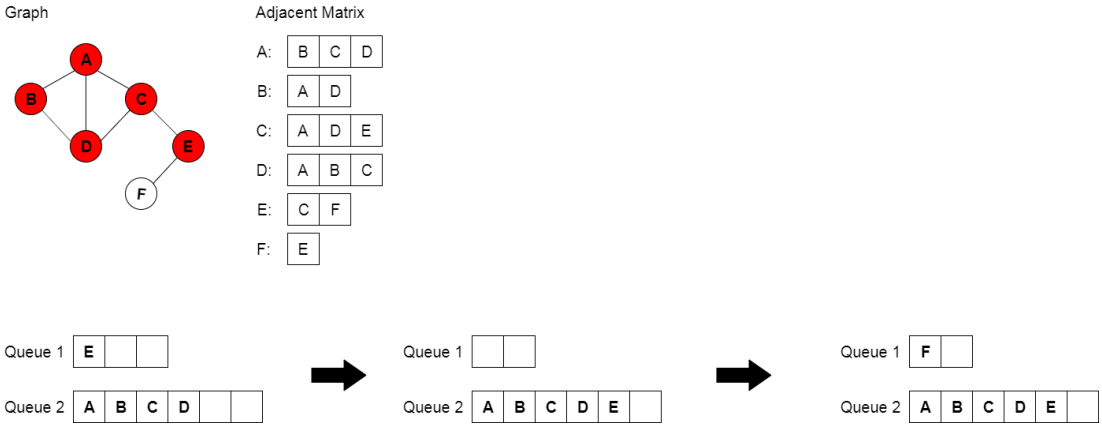




Step3:

依序選擇與之前訪尋過的點相鄰的所有點作為下一個拜訪的點。然而需按照順序選取，與之前點相鄰的所有點落在 Queue1 中，依照 Queue 先進先出的原則，應先選擇 E 作為下一個拜訪的點。

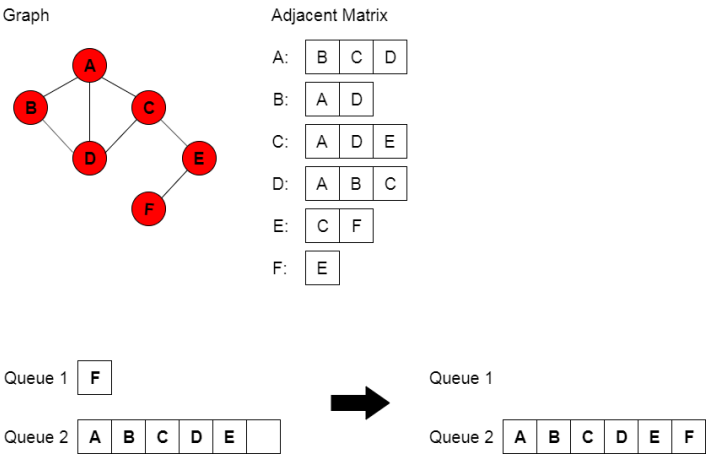
1. 選擇 E 作為拜訪點，並做上已拜訪過的記號，並將 E 放進 Queue2 中
2. 將所有與 E 相連的點放進 Queue1 中，放入的順序需照 Adjacent Matrix 中與 E 相連的點的順序放進 Queue1 中，如果該相鄰的點已做上記號或已放進 Queue1 中則不選取。。



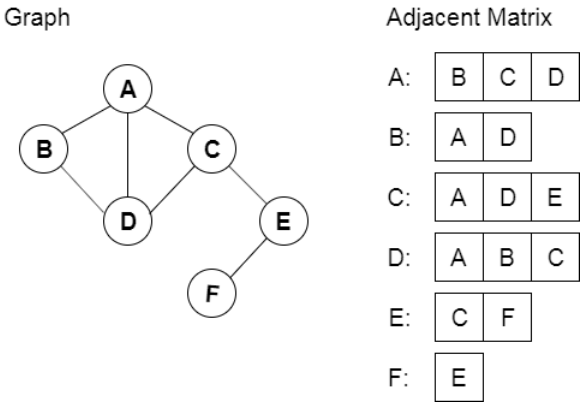
Step4:

依序選擇與之前訪尋過的點相鄰的所有點作為下一個拜訪的點。然而需按照順序選取，與之前點相鄰的所有點落在 Queue1 中，依照 Queue 先進先出的原則，應先選擇 F 作為下一個拜訪的點。

1. 選擇 F 作為拜訪點，並做上已拜訪過的記號，並將 F 放進 Queue2 中。
2. 當 Queue1 為空時，代表已沒有需訪尋的點，因此 BFS 完成，訪尋 Graph 的順序為: A, B, C, D, E, F。

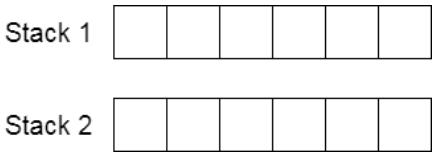


DFS 流程圖



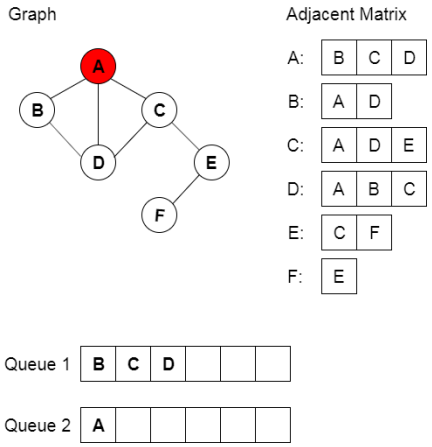
Step1:

1. 建立兩個 Stack。



Step2:

1. 選擇 A 作為起始點，並做上一個已拜訪過的記號，並將 A 放進 Stack2 中。
2. 將所有與 A 相連的點放進 Stack1 中，放入的順序需照 Adjacent Matrix 中與 A 相連的點的順序放進 Stack1 中。

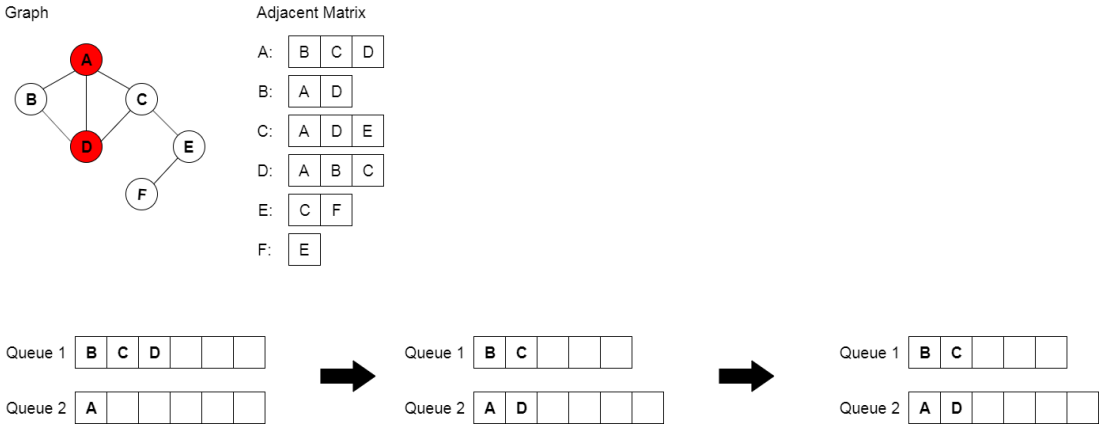


Step3:

選擇與之前訪尋過的點相鄰的點作為下一個拜訪的點。然而需按照順序選取，與之前點相鄰的所有點落在 Stack1 中，依照 Stack 後進先出的原則，應先選擇 D 作為下一個拜訪的點。

1. 選擇 D 作為拜訪點，並做上已拜訪過的記號，並將 D 放進 Stack2 中。

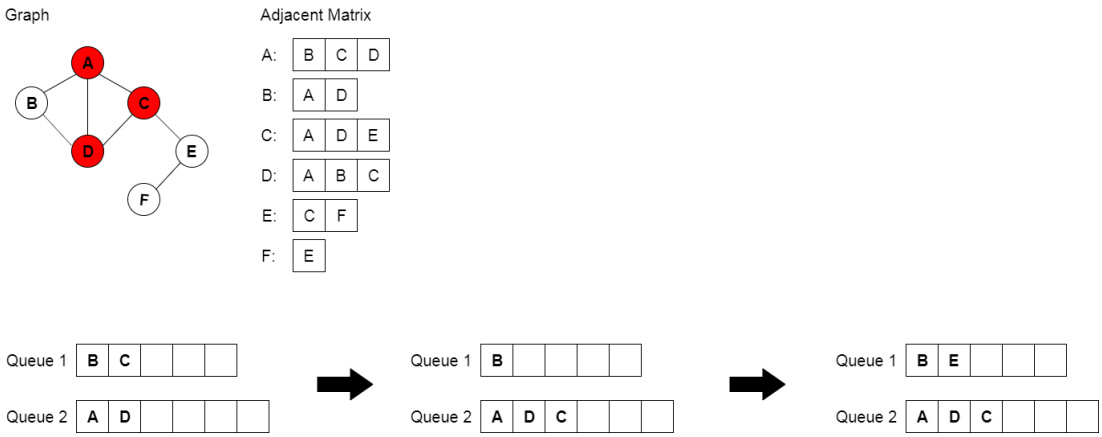
2. 將所有與 D 相連的點放進 Stack1 中，放入的順序需照 Adjacent Matrix 中與 D 相連的點的順序放進 Stack1 中，如果該相鄰的點已做上記號或已放進 Stack1 中則不選取。



Step4:

選擇與之前訪尋過的點相鄰的點作為下一個拜訪的點。然而需按照順序選取，與之前點相鄰的所有點落在 Stack1 中，依照 Stack 後進先出的原則，應先選擇 C 作為下一個拜訪的點。

1. 選擇 C 作為拜訪點，並做上已拜訪過的記號，並將 C 放進 Stack2 中。
2. 將所有與 C 相連的點放進 Stack1 中，放入的順序需照 Adjacent Matrix 中與 C 相連的點的順序放進 Stack1 中，如果該相鄰的點已做上記號或已放進 Stack1 中則不選取。

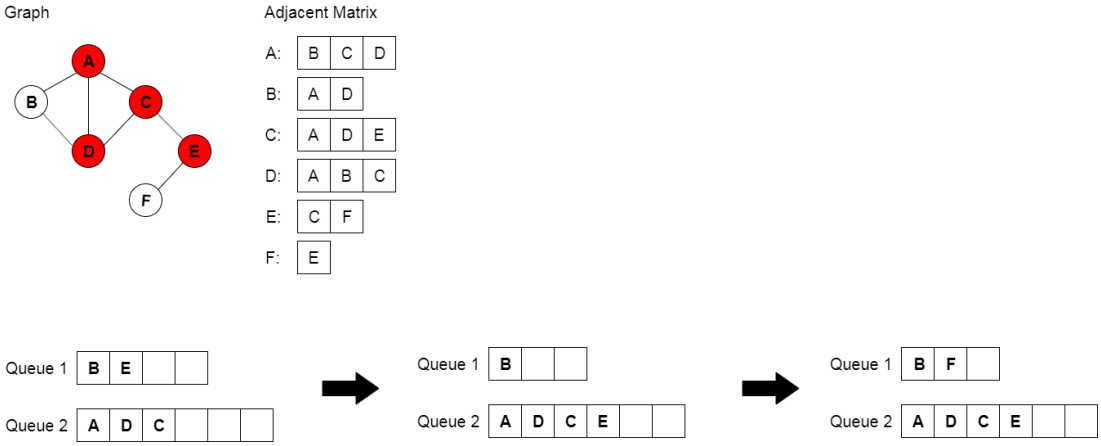


Step5:

選擇與之前訪尋過的點相鄰的點作為下一個拜訪的點。然而需按照順序選取，與之前點相鄰的所有點落在 Stack1 中，依照 Stack 後進先出的原則，應先選擇 E 作為下一個拜訪的點。

3. 選擇 E 作為拜訪點，並做上已拜訪過的記號，並將 E 放進 Stack2 中。

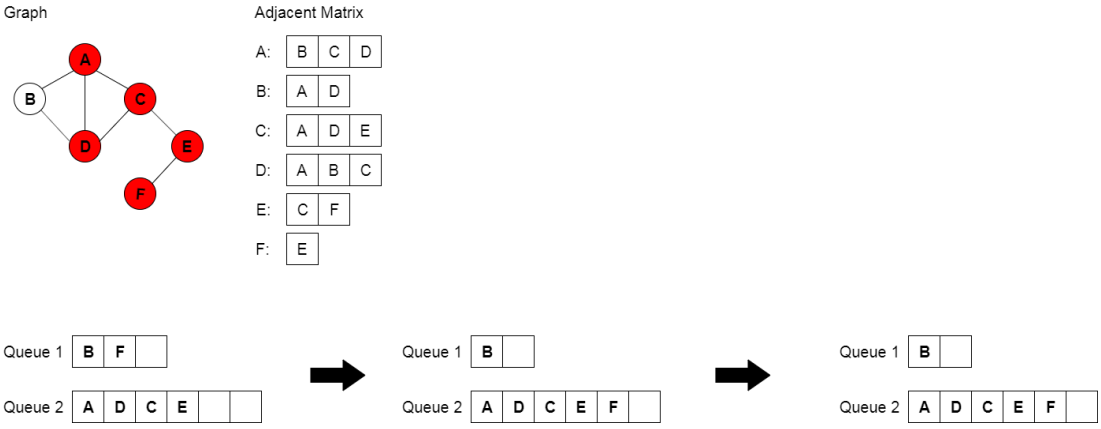
4. 將所有與 E 相連的點放進 Stack1 中，放入的順序需照 Adjacent Matrix 中與 E 相連的點的順序放進 Stack1 中，如果該相鄰的點已做上記號或已放進 Stack1 中則不選取。



Step6:

選擇與之前訪尋過的點相鄰的點作為下一個拜訪的點。然而需按照順序選取，與之前點相鄰的所有點落在 Stack1 中，依照 Stack 後進先出的原則，應先選擇 F 作為下一個拜訪的點。

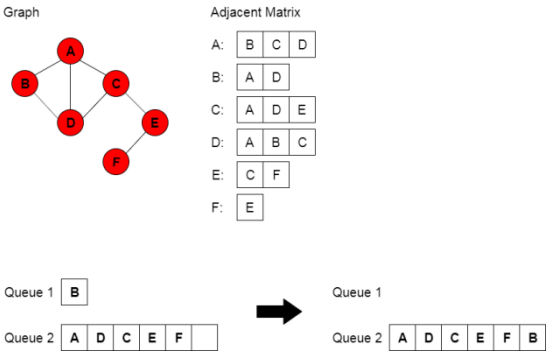
5. 選擇 F 作為拜訪點，並做上已拜訪過的記號，並將 F 放進 Stack2 中。
6. 將所有與 F 相連的點放進 Stack1 中，放入的順序需照 Adjacent Matrix 中與 F 相連的點的順序放進 Stack1 中，如果該相鄰的點已做上記號或已放進 Stack1 中則不選取。



Step7:

3. 選擇與之前訪尋過的點相鄰的點作為下一個拜訪的點。然而需按照順序選取，與之前點相鄰的所有點落在 Stack1 中，依照 Stack 後進先出的原則，應先選擇 B 作為下一個拜訪的點。
4. 選擇 B 作為拜訪點，並做上已拜訪過的記號，並將 B 放進 Stack2 中。

5. 當 Stack1 為空時，代表已沒有需訪尋的點，因此 DFS 完成，訪尋 Graph 的順序為: A, D, C, E, F, B。



二、 程式碼學習歷程

1. 程式碼說明

```
1  from collections import defaultdict
2
3  class Queue:
4      def __init__(self):
5          self.stack_in=[]
6          self.stack_out=[]
7
8      def pop(self):
9          if len(self.stack_out) > 0:
10             return self.stack_out.pop()
11         elif len(self.stack_in)==0:
12             return
13         else:
14             while len(self.stack_in) > 0:
15                 self.stack_out.append(self.stack_in.pop())
16             return self.stack_out.pop()
17 class Graph:
18     def __init__(self):
19         self.graph = defaultdict(list)
20         self.state1 = []
21         self.state2 = []
22
23     def addEdge(self,u,v):
24         self.graph[u].append(v)
25     def BFS(self, s, initialize=True):
27         if initialize == True:
28             self.state2.append(s)
29         Q1 = Queue()
30         Q1.stack_in = self.graph[s].copy()
31         while True:
32             v1=Q1.pop()
33             if v1 not in self.state1 and v1 not in self.state2:
34                 self.state1.append(v1)
35             if len(Q1.stack_out) == 0:
36                 break
37         Q2 = Queue()
38         Q2.stack_in = self.state1.copy()
39         v2 = Q2.pop()
40         if len(self.state1) > 0:
41             self.state1.remove(v2)
42             self.state2.append(v2)
43             return self.BFS(v2, initialize=False)
44         elif len(self.state1) == 0:
45             output=self.state2
46             self.state2=[]
47             return output
48
```

BFS:

第 21、22 行。建立兩個 Queue: state1、state2。

第 28、29 行。在函數 BFS 裡預設參數 initialize 的目的是了區分 input 為起始點還是非起始點。處理起始點和非起始點方式並不相同，處理起始點時只需將該點放進 state1；處理非起始點時則需要後續動作。在程式運行方面，因為是使用遞迴的方式，必須區分遞迴時的 input 為起始點還是非起始點，當 initialize 預設為 True 時，代表輸入的 input 為起始值；當 initialize 預設為 False 時，代表輸入的 input 非起始值。

第 30~37 行。將所有與訪尋到的點相連的點放進 state1 中，作為待訪尋的點，放入的順序需照 Adjacent Matrix 中與該點相連的點的順序放進 state1 中。放入的順序是將 self.graph[s]中的數值由左至右抽出並依序放入 state1 中。抽出的順序與

Queue 先進先出的觀念相同，因此可以把 self.graph[s]當成一個 Queue，並使用 Queue 物件中 pop()功能將 self.graph[s]中的數值抽出，Queue 於課程第二堂課已教過，因此不多敘述。其中為了不破壞 self.graph[s]的內容，因此另外建一個 self.graph[s].copy()，將 self.graph[s].copy()裡的數值按照先進先出的原理抽出並放入 state1 中。

第 34、35 行。如果該相連的點已訪尋或已列為待訪尋，則不放入 state1 中。

第 36、37 行。當已經將訪尋到的點相連的點全部放進 state1 中時則結束動作。

第 38~40 行。選擇一個與之前訪尋過的點相鄰的點作為下一個訪尋的點。因為 BFS 的概念是依序選擇一個與之前訪尋過的點相鄰的所有點作為下一個訪尋的點，因此下一個訪尋的點需從目前最早列為待訪尋的點開始往後尋找，順序極為 state1 中由左至右的數值。尋找的順序與 Queue 先進先出的觀念相同，因此將 state1 視為一個 Queue，並使用 Queue 物件將 state1 中的數值抽出並放進 state2 中。其中因為在做 Queue 物件中的 pop()時會先將陣列中所有值先取出並 append 進一個新陣列，為了不破壞 state1，因此外建一個 state1.copy()，並從中取出下一個訪尋值，然後用 remove()的方式將該值從 state1 中剔除並放進 state2 中。

第 44 行。將該值作為新的訪尋值，放進 BFS 函式做遞迴，其中 intitailize 設為 False。

第 45~48 行。直到已沒待訪尋的值時，輸出 BFS 的訪尋結果。其中為了可以重複使用 BFS 物件，因此會先將__init__中 state2 清空。

```
1  from collections import defaultdict
2
3  class Queue:
4      def __init__(self):
5          self.stack_in=[]
6          self.stack_out=[]
7
8      def pop(self):
9          if len(self.stack_out) > 0:
10             return self.stack_out.pop()
11         elif len(self.stack_in)==0:
12             return
13         else:
14             while len(self.stack_in) > 0:
15                 self.stack_out.append(self.stack_in.pop())
16             return self.stack_out.pop()
18 class Graph:
19     def __init__(self):
20         self.graph = defaultdict(list)
21         self.state1 = []
22         self.state2 = []
23
24     def addEdge(self,u,v):
25         self.graph[u].append(v)
50     def DFS(self, s, initialize=True):
51         if initialize == True:
52             self.state2.append(s)
53         Q1 = Queue()
54         Q1.stack_in = self.graph[s].copy()
55         while True:
56             v1=Q1.pop()
57             if v1 not in self.state1 and v1 not in self.state2:
58                 self.state1.append(v1)
59             if len(Q1.stack_out) == 0:
60                 break
61         if len(self.state1) > 0:
62             v2 = self.state1.pop()
63             self.state2.append(v2)
64             return self.DFS(v2, initialize=False)
65         if len(self.state1) == 0:
66             output = self.state2
67             self.state2 = []
68             return output
```

DFS:

第 21、22 行。建立兩個 Stack: state1、state2。

第 51、52 行。在函數 DFS 裡預設參數 initialize 的目的是了區分 input 為起始點還是非起始點。處理起始點和非起始點方式並不相同，處理起始點時只需將該點放進 state1；處理非起始點時則需要後續動作。在程式運行方面，因為是使用遞迴的方式，必須區分遞迴時的 input 為起始點還是非起始點，當 initialize 預設為 True 時，代表輸入的 input 為起始值；當 initialize 預設為 False 時，代表輸入的 input 非起始值。

第 53~60 行。將所有與訪尋到的點相連的點放進 state1 中，作為待訪尋的點，放入的順序需照 Adjacent Matrix 中與該點相連的點的順序放進 state1 中。放入的順序是將 self.graph[s]中的數值由左至右抽出並依序放入 state1 中。抽出的順序與 Queue 先進先出的觀念相同，因此可以把 self.graph[s]當成一個 Queue，並使用 Queue 物件中 pop()功能將 self.graph[s]中的數值抽出，Queue 於課程第二堂課已教過，因此不多敘述。其中為了不破壞 self.graph[s]的內容，因此另外建一個 self.graph[s].copy()，將 self.graph[s].copy()裡的數值按照先進先出的原理抽出並放

入 state1 中。
第 57、58 行。如果該相連的點已訪尋或已列為待訪尋，則不放入 state1 中。
第 59、60 行。當已經將訪尋到的點相連的點全部放進 state1 中時則結束動作。
第 61~63 行。選擇一個與之前訪尋過的點相鄰的點作為下一個訪尋的點。因為 BFS 的概念是選擇一個與目前訪尋到的點相鄰的點作為下一個訪尋的點，因此下一個訪尋的點需從目前最晚列為待訪尋的點開始往前尋找，順序極為 state1 中由右至左的數值。尋找的順序與 Stack 後進先出的觀念相同，因此將 state1 視為一個 Stack，並將 state1 中的數值抽出並放進 state2 中。
第 64 行。將該值作為新的訪尋值，放進 DFS 函式做遞迴，其中 intitailize 設為 False。
第 65~68 行。直到已沒待訪尋的值時，輸出 DFS 的訪尋結果。其中為了可以重複使用 DFS 物件，因此會先將__init__中 state2 清空。

2. BFS 另外寫法

沒有額外使用 Queue 物件，原理與前一點相似，因此不多敘述。

```
1 from collections import defaultdict
2
3 class Graph:
4     def __init__(self):
5         self.graph = defaultdict(list)
6         self.state1 = []
7         self.state2 = []
8         self.stack_in = []
9         self.stack_out = []
10
11    def addEdge(self,u,v):
12        self.graph[u].append(v)
13
14    def DFS(self, s):
15        l = self.graph[s]
16        stack1 = l.copy()
17        stack2=[]
18        while len(stack1) > 0:
19            stack2.append(stack1.pop())
20        while len(stack2) > 0:
21            v1=stack2.pop()
22            if v1 not in self.state1 and v1 not in self.state2:
23                self.state1.append(v1)
24        self.stack_in=self.state1.copy()
25        v2=self.pop()
26        if len(self.state1) > 0:
27            self.state1.remove(v2)
28            self.state2.append(s)
29            return self.DFS(v2)
30        elif len(self.state1) == 0:
31            output=self.state2
32            self.state2=[]
35    def pop(self):
36        if len(self.stack_out) > 0:
37            return self.stack_out.pop()
38        elif len(self.stack_in)==0:
39            return
40        else:
41            while len(self.stack_in) > 0:
42                self.stack_out.append(self.stack_in.pop())
43            return self.stack_out.pop()
```

三、 BFS 與 DFS 原理與比較

追蹤圖形的作法是從圖形的某一頂點出發，然後走訪圖形的其他頂點。經由圖形追蹤可以判斷該圖形的某些頂點是否連通。樹 (Tree) 的追蹤目的是欲拜訪樹的每一個節點一次，可用的方法有中序法、前序法和後序法等三種，而圖形追蹤的方法有兩種: 「先深後廣走訪」及「先廣後深走訪」。

1. BFS 與 DFS 原理

I. 先廣後深搜尋法 (Breadth-First Search, BFS):

從圖形的某一頂點開始走訪，被拜訪過的頂點就做上已走訪的記號，接著走訪此頂點的所有相鄰且未拜訪過的任意一個頂點，並標上已走訪記號，再以該頂點維新的起點繼續進行 BFS。

II. 先深後廣搜尋法 (Depth-First Search, DFS):

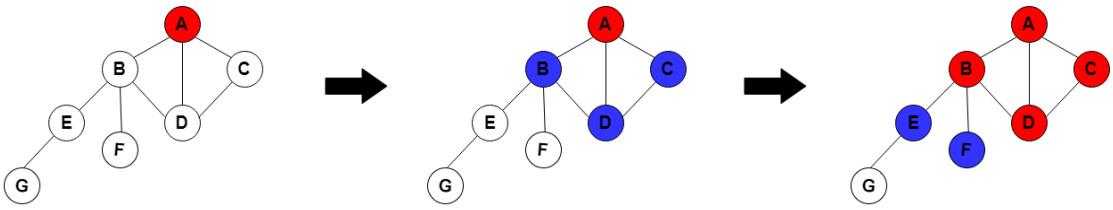
從圖形的某一頂點開始走訪，被拜訪過的頂點就做上已走訪的記號，接著走訪此頂點第一個相鄰且未拜訪過的頂點，並標上已走訪記號，再以該頂點維新的起點繼續進行 DFS。

2. BFS 與 DFS 比較

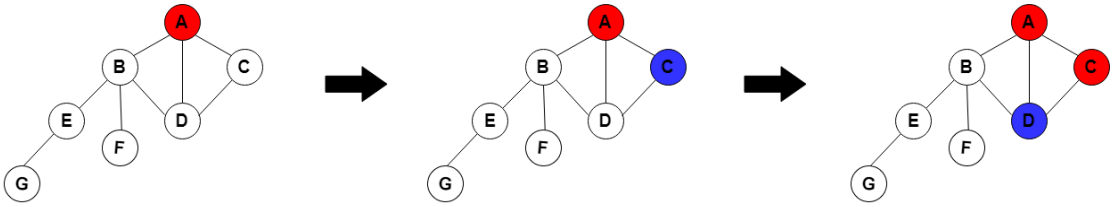
I. 走訪方式不同:

當走訪完某個點後，BFS 會依序走訪該點所有相鄰且未走訪過的點；DFS 則接著走訪該點第一個相鄰且未走訪過的點。

當走訪完 A 點時，BFS 會走訪與 A 點所有相鄰且未走訪過的點: B, C, D，走訪完後再走訪所有與 B 點相鄰且為走訪過的點: E, F。DFS 則會走訪某一個與 A 點相鄰且未走訪過的點: C，走訪完後再走訪某一個與 C 點相鄰且未走訪過的點: D。



BFS



DFS

在撰寫程式方面，BFS 所使用的待訪尋的陣列 state1 本身是一個 Queue，因為 BFS 的概念是依序選擇一個與之前訪尋過的點相鄰的所有點作為下一個訪尋的點，因此下一個訪尋的點需從目前最早列為待訪尋的點開始往後尋找，順序極為 state1 中由左至右的數值。尋找的順序與 Queue 先進先出的觀念相同，因此將 state1 視為一個 Queue。DFS 所使用的待訪尋的陣列 state1 本身是一個 Stack，因為 BFS 的概念是選擇一個與目前訪尋到的點相鄰的點作為下一個訪尋的點，因此下一個訪尋的點需從目前最晚列為待訪尋的點開始往前尋找，順序極為 state1 中由右至左的數值。尋找的順序與 Stack 後進先出的觀念相同，因此將 state1 視為一個 Stack。

```
50    def DFS(self, s, initialize=True):
51        if initialize == True:
52            self.state2.append(s)
53        Q1 = Queue()
54        Q1.stack_in = self.graph[s].copy()
55        while True:
56            v1=Q1.pop()
57            if v1 not in self.state1 and v1 not in self.state2:
58                self.state1.append(v1)
59            if len(Q1.stack_out) == 0:
60                break
61        if len(self.state1) > 0:
62            v2 = self.state1.pop()
63            self.state2.append(v2)
64            return self.DFS(v2, initialize=False)
65        if len(self.state1) == 0:
66            output = self.state2
67            self.state2 = []
68            return output
```

```
27    def BFS(self, s, initialize=True):
28        if initialize == True:
29            self.state2.append(s)
30        Q1 = Queue()
31        Q1.stack_in = self.graph[s].copy()
32        while True:
33            v1=Q1.pop()
34            if v1 not in self.state1 and v1 not in self.state2:
35                self.state1.append(v1)
36            if len(Q1.stack_out) == 0:
37                break
38        Q2 = Queue()
39        Q2.stack_in = self.state1.copy()
40        v2 = Q2.pop()
41        if len(self.state1) > 0:
42            self.state1.remove(v2)
43            self.state2.append(v2)
44            return self.BFS(v2, initialize=False)
45        elif len(self.state1) == 0:
46            output=self.state2
47            self.state2=[]
48            return output
```

II. 時間複雜度:

BFS: 最壞的情況下，每個頂點至少訪問一次，每條邊至少訪問 1 次，這是因為在搜尋的過程中，若某結點向下搜尋時，其子結點都訪問過了，這時候就會回退，故時間復雜度為 O(E)，演算法總的時間複雜度為 O(|V|+|E|)。

DFS: 概念與 BFS 相同。

III. BFS 相對 DFS 更耗記憶體空間:

BFS 每次找尋下一個訪尋的點時都需藉助一個 Queue 來儲存，把待訪尋的陣列 state1 用 Queue 來儲存，並把最先進入 state1 的點取出，因此空間複雜度與點數成正比。DFS 在找尋下一個訪尋的點時則不需借助一個 Stack 來儲存，因為待訪尋的陣列 state1 本身就可視作一個 stack，只需直接把下一個訪尋點從 state1 中 pop 出來。

IV. DFS 有些情況下會相當耗時:

深度搜尋法雖然可以搜尋整個圖形，可是搜尋方向固定，因此若將子節點個數一多，或遇到特殊情況，必須要繞所有子節點才可以找到目標，那麼，會相當耗時間，情況壞的話，電腦可能無法執行，無法負荷，相當的不方便。

四、 參考資料

BFS 與 DFS 流程圖、原理與比較:

1. 李淑馨，使用資料結構 Python，8.3 節，深石數位科技。
2. http://alrightchiu.github.io/SecondRound/graph-breadth-first-searchbfsguang-du-you-xian-sou-xun.html
3. http://alrightchiu.github.io/SecondRound/graph-depth-first-searchdfsshen-du-you-xian-sou-xun.html
4. https://zh.wikipedia.org/wiki/%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2
5. https://zh.wikipedia.org/wiki/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2
6. https://www.itread01.com/content/1549064200.html
7. https://www.shs.edu.tw/works/essay/2017/03/2017033023453259.pdf
8. 老師上課講義

程式碼參考資料:

1. 老師 Queue 程式碼

https://github.com/albert0796/DSA.git

Queue/Queue_參考程式碼

2. 自己原創 Queue 程式碼

https://github.com/albert0796/DSA.git

Queue/Queue_修改程式碼