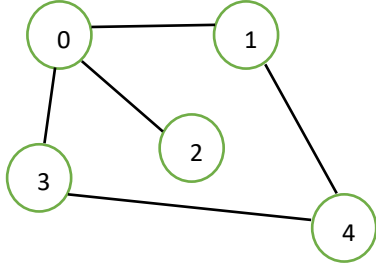
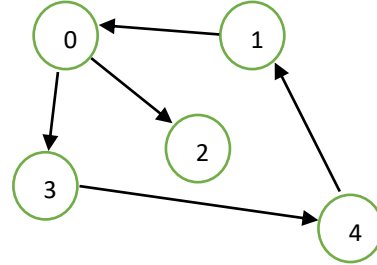


- ماهو ال graph ؟ هو نوع من أنواع ال data structure بحيث أنه لديه vertices or nodes ويربطهم عن ب edges
It's a data structure type, which have vertices connected with edges

- أنواع ال graph ؟ Directed and undirected



Undirected
Edge ليس لها اتجاه



Directed
Edge لها اتجاه

- ماهي الأرقام في vertices ؟ ممكن أن تعتبرها id or value مميز لكل vertex – وأيضا ممكن أن تحتوي على أي قيم أخرى
- في البداية لابد من التكلم عن كيفية تمثيل ال graph في البرمجة (Graph representation)

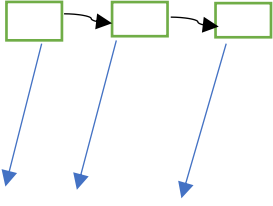
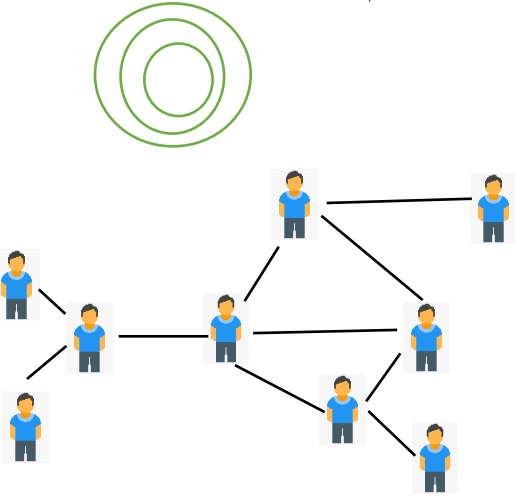
- 2D Matrix
- Adjacency list or Dictionary <Key, []>
- ملحوظة في هذا المثال شغالين على undirected من المثال السابق – طيب في حالة إنه directed أضع حسم الإتجاه

2D Matrix	Adjacency List or Dictionary <Key, []>																																				
<table><tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr><tr><th>0</th><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr><tr><th>1</th><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><th>2</th><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><th>3</th><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><th>4</th><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>		0	1	2	3	4	0	0	1	1	1	0	1	1	0	0	0	1	2	1	0	0	0	0	3	1	0	0	0	1	4	0	1	0	1	0	<pre>graph LR; 0 --> 1; 0 --> 2; 0 --> 3; 1 --> 0; 1 --> 4; 2 --> 0; 3 --> 0; 3 --> 4; 4 --> 1; 4 --> 3;</pre>
	0	1	2	3	4																																
0	0	1	1	1	0																																
1	1	0	0	0	1																																
2	1	0	0	0	0																																
3	1	0	0	0	1																																
4	0	1	0	1	0																																
<p>هنا أقومك بعمل binary matrix لو فيه edge بين 2 vertices أقوم بوضع 1 غير ذلك 0</p> <pre>graph=[] graph.append([0,1,1,1,0]) graph.append([1,0,0,0,1]) graph.append([1,0,0,0,0]) graph.append([1,0,0,0,1]) graph.append([0,1,0,1,0])</pre>	<p>هنا كل vertex هي key و مصاحب لها list of vertices</p> <pre>Vertices = 5 graph = {} # Init dictionary with key and empty list option-1</pre>																																				

	<pre> [graph.setdefault(v, []) for v in range(Vertices)] # or use for-loop option-2 for v in range(Vertices): graph[v] = [] graph[0].append(1) graph[0].append(2) graph[0].append(3) graph[1].append(0) graph[1].append(1) graph[2].append(0) graph[3].append(0) graph[3].append(4) graph[4].append(1) graph[4].append(3) </pre>
<p>مشكلة الطريقة دي:</p> <p>Time complexity, Space, Sparse</p> <p>ال sparse هي أنه توجد كثير من 0 وقليل من 1 بمعنى</p> <p>قليل من Edge</p> <p>Time complexity $O(N^2)$</p>	<p>هنا تم حل مشاكل 2D Matrix</p> <p>$O(V + E)$</p> <p>لذلك أغلب التمثيل لل Graph يكون بهذه الطريقة</p>

• كيفية Graph traverse

DFS	BFS
<pre> graph = dict() # initialize like previous example seen = set() def dfs(v): if v not in graph: return seen.add(v) for w in graph[v]: if w not in seen: dfs(w) def main(): for v in range(V): if not v in seen: continue dfs(v) </pre>	<pre> from queue import Queue Vertices = 5 graph = dict() # initialize like previous example seen = set() def bfs(v): q = Queue() q.put(v) seen.add(v) while not q.empty(): curr = q.get() for w in graph[curr]: if w not in seen: seen.add(w) q.put(w) </pre>

	<pre>def main(): for v in range(Vertices): if v not in seen: bfs(v)</pre>
<p>البحث أو traverse لمسك كل branch إلى آخره – ثم يبدأ ب branch التالي</p> 	<p>هنا البحث level-level بمعنى الأقرب بالأبعد – مثل الفيس بوك الأصدقاء ثم أصدقاء الأصدقاء</p> 

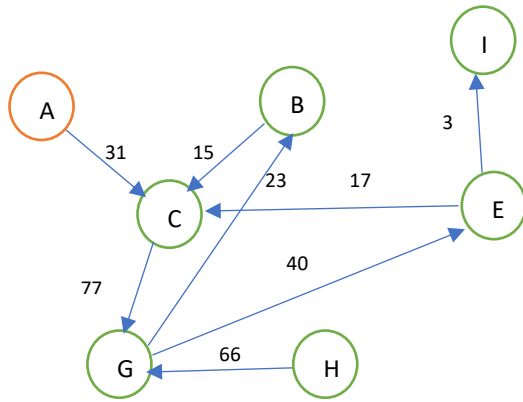
• ماهو ال PriorityQueue؟

- هو نوع من ال Queue يستخدم ال Binary Tree لترتيب العناصر ويستخدم طريقة Heapify وSwim Sink
- تستخدم هذه العمليات مع عمليات الإضافة والإزالة وتكلفة كل عملية $O(\log N)$
- يوجد منه نوعين Max PQ, Min PQ الماكس يجيب من الأكبر للأصغر و المين يجيب من الأصغر للأكبر
- كيفية استخدامه في البايثون

<pre>from queue import PriorityQueue pq = PriorityQueue()</pre>	<pre>import heapq heapq.heapify() heapq.heappush() heapq.heappop()</pre>
--	---

• ماهو ال Dijkstra algorithm؟

- من أنواع الخوارزميات التي تعمل على Graph بحيث تحسب أفضل مسار من نقطة إلى باقي النقاط المتصلة
- شكل ال Graph



يمكن أن يكون ال graph إما directed or undirected في هذا المثال directed

Dictionary <Key, Set(>

كل vertex هي ال key ومتصل بكل واحدة ال set - كل عنصر في set هو tuple(Vertex, Cost)

```
G = {
  'A': {'C', 31},
  'B': {'C', 15},
  'C': {'G', 77}, {'H', 40},
  'E': {'C', 17}, {'I', 3},
  'G': {'B', 22}, {'E', 23},
  'H': {'G', 66},
  'I': {'J', 70}, {'K', 31},
}
```

○ فكرة عمله كالتالي

1. أبدأ من عند نقطة البداية S وأضفها لل PQ
2. أسحب من ال PQ القيمة الأقل
3. أشوف كل النقط المتصلة بها
4. أشوف ال vertex زرتها من قبل أو لا؟
5. لو لا ، أحسب تكلفتها عن طريق حساب تكلفة vertex مع edge لو أصغر من تكلفة النقطة الحالية
6. أضعها لل priority queue
7. وأعمل تحديث للتكلفة في cost
8. وأعمل تحديث لل parent_path وأضيف ال vertex اللي رايحة له كأصغر تكلفة
9. أكرر الخطوات من 2 إلى 8 إلى أن يبقى Priority Queue فاضي

○ كيف أأخذن التكلفة؟ عن طريق استخدام dictionary<key, int> ال key هي النقط والتكلفة

Vertex	Cost	Parent (Incoming vertex)
A	0	None
B	INF	
C	INF	

هنا عندي graph, seen, cost, parent_path

الـ graph تمثيل الجراف – seen من أجل هل زرت الـ vertex أو لا؟ – cost هو dictionary فيه كل vertex والتكلفة لها –
parent_path هو dictionary أأخزن كل vertex أنا رويحت لها منين

```
from queue import PriorityQueue

def dijkstra(graph, s):
    seen = set()
    cost = {s: 0} # store vertices' cost
    parent_path = {s: None} # store incoming vertex
    pq = PriorityQueue()

    pq.put((0, s))
    while not pq.empty():
        _, v = pq.get() # get the min value (ascending order)
        seen.add(v)

        for w, distance in graph[v]:
            if w in seen: continue

            # get w from costs, if not exists then default INF
            old_cost = cost.get(w, float('inf'))
            new_cost = cost[v] + distance
            if new_cost < old_cost:
                pq.put((new_cost, w)) # put to the PQ
                cost[w] = new_cost
                parent_path[w] = v

    return parent_path

G = {
    'A': {('C', 31)},
    'B': {('C', 15), ('J', 58)},
    'C': {('C', 60), ('G', 77), ('H', 40)},
    'E': {('C', 17), ('E', 55), ('I', 3)},
    'G': {('B', 22), ('E', 23), ('G', 31)},
    'H': {('G', 66)},
    'I': {('J', 70), ('K', 31)},
    'J': {('H', 8), ('K', 28)},
    'K': {('I', 13)}
}

parent_path = dijkstra(G, 'A')
```

• ماهو ال Prims algorithm؟

- هو من خوارزميات الجراف التي تجد MST (Minimum Spanning Tree) والتي تصل كل النقاط من غير مايكون في cycle بأفضل تكلفة
- شكل Graph

يمكن أن يكون ال graph إما directed or undirected في هذا المثال directed

<Key, Set()> Dictionary

كل vertex هي ال key ومتصل بكل واحدة ال set - كل عنصر في set هو tuple(Vertex, Cost)

```
G = {
'A': {('C', 31)},
'B': {('C', 15)},
'C': {('G', 77), ('H', 40)},
'E': {('C', 17), ('I', 3)},
'G': {('B', 22), ('E', 23)},
'H': {('G', 66)},
'I': {('J', 70), ('K', 31)},
}
```

○ فكرة عمله كتالي

1. أبدأ من عند نقطة البداية S وأضفها لل PQ
2. أسحب من ال PQ القيمة الأقل
3. أشوف ال vertex زرتها من قبل أو لا؟
4. لو لا ، أضيفها لل MST
5. أشوف كل النقط المتصلة بها
6. وأشوف هل زرتها أو لا؟
7. لو لا ، أضيفها لل priority queue
8. أكرر الخطوات من 2 إلى 7 إلى أن يبقى Priority Queue فاضي

الـ graph تمثيل الجراف – seen من أجل هل زرت ال vertex أو لا؟ - mst هو dictionary فيه كل vertex و النقط المتصلة بها – edges هي list بها tuple(cost,v,w) النقطتين والتكلفة ما بينهم - وأستخدمه ك PQ

```
from collections import defaultdict
import heapq

def prims(graph, s):
    mst = defaultdict(set)      # declare empty dictionary default value set
    seen = set([s])             # store seen vertex

    # edges are list of tuple(weight/cost, v, w)
    # get all vertices connected with start and add them to edges
    edges = [
        (cost, s, w)
        for w, weight in graph[s].items()
    ]
    # use edges as PQ
    heapq.heapify(edges)

    while edges:
        cost, v, w = heapq.heappop(edges)      # get min value
        if w not in visited and w in graph:
            seen.add(w)
            mst[v].add(w)

            for adj, cost in graph[w]:
                if adj not in seen:
                    heapq.heappush(edges, (cost, w, adj))

    return mst

graph = {
    'A': {('C', 31)},
    'B': {('C', 15)},
    'C': {('G', 77), ('H', 40)},
    'E': {('C', 17), ('I', 3)},
    'G': {('B', 22), ('E', 23)},
    'H': {('G', 66)},
    'I': {('J', 70), ('K', 31)},
}

print(prims(graph, 'A'))
```