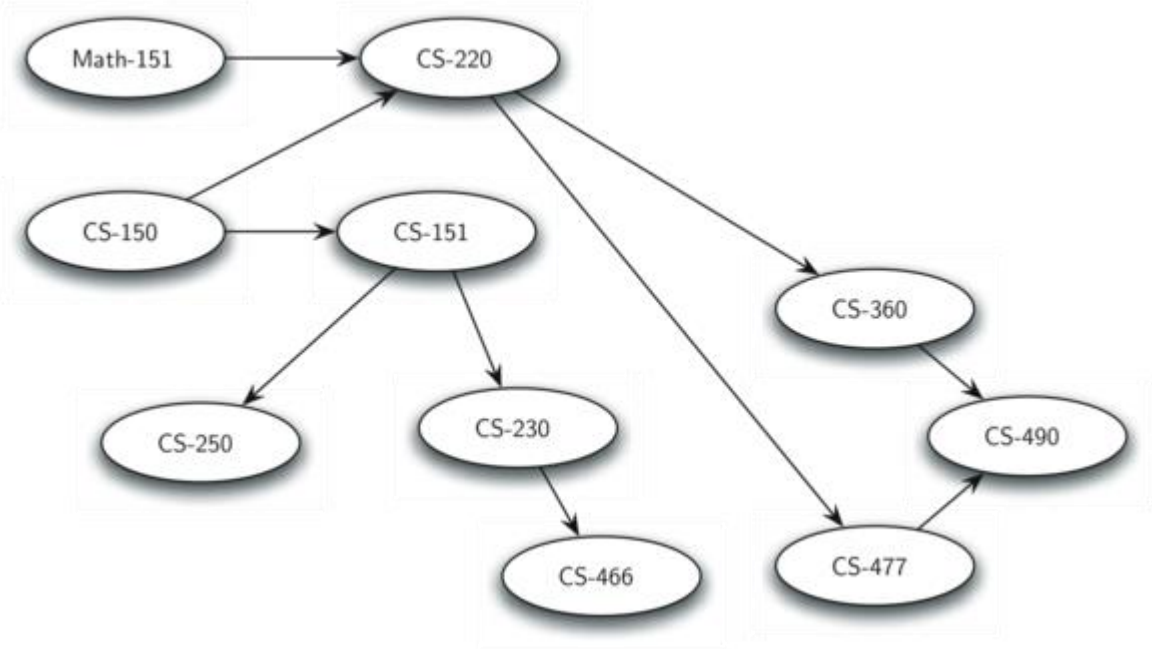
กราฟ
graph

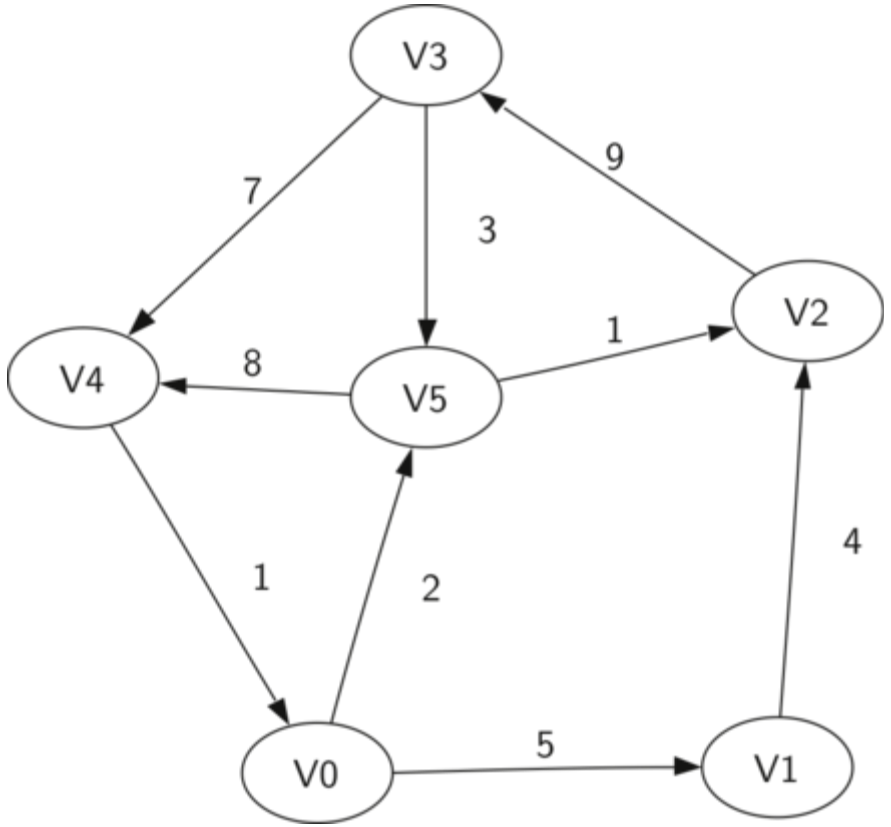
ใช้ทำอะไร?

- แสดงความสัมพันธ์ของข้อมูล
- อธิบายได้กว้างกว่าต้นไม้



คำศัพท์

- จุดยอด (vertex, node)
  - key
  - payload
- เส้นเชื่อม (edge, arc)
  - มีทิศทาง
  - ไม่มีทิศทาง
- น้ำหนัก (weight)
  - ระยะทาง
- G = (V,E)

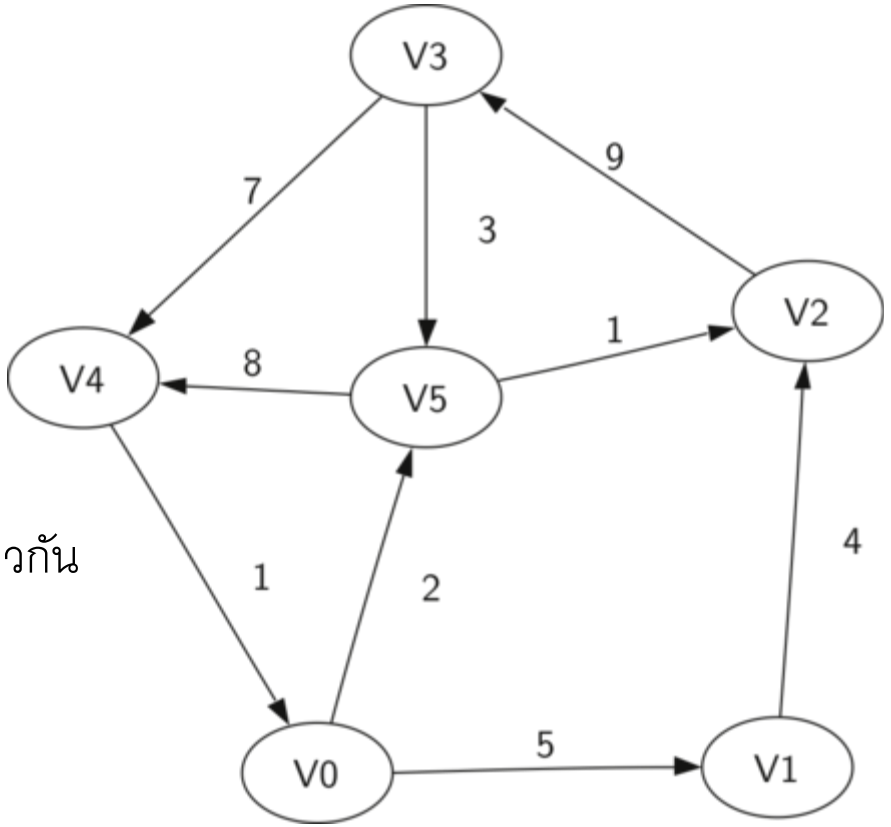


V={V0,V1,V2,V3,V4,V5}

E={(v0,v1,5),(v1,v2,4),(v2,v3,9),(v3,v4,7),(v4,v0,1),
(v0,v5,2),(v5,v4,8),(v3,v5,3),(v5,v2,1)}

คำศัพท์

- เส้นทาง (path)
  - ลำดับของจุดยอดที่เชื่อมโยงกัน
  - ความยาวของเส้นทางคำนวณจากน้ำหนัก
- วงวน (cycle)
  - จุดยอดเริ่มต้นและจุดยอดสุดท้ายของเส้นทางเป็นจุดเดียวกัน



(V3,V4,V0,V1)

{(v3,v4,7),(v4,v0,1),(v0,v1,5)}

graph ADT

- Graph() สร้าง graph ว่าง.
- addVertex(vert) เพิ่ม vertex ให้กับ graph.
- addEdge(fromVert, toVert) กำหนด edge ที่มีทิศทางให้กับ 2 vertex
- addEdge(fromVert, toVert, weight) กำหนด edge ที่มีทิศทางและน้ำหนักให้กับ 2 vertex
- getVertex(vertKey) ค้นหา vertex ด้วย key
- getVertices() คืนค่า list ของ vertex ทั้งหมดใน graph
- in คืนค่า True เมื่อมี vertex ที่ถามอยู่ใน graph คืนค่า False เมื่อไม่มี

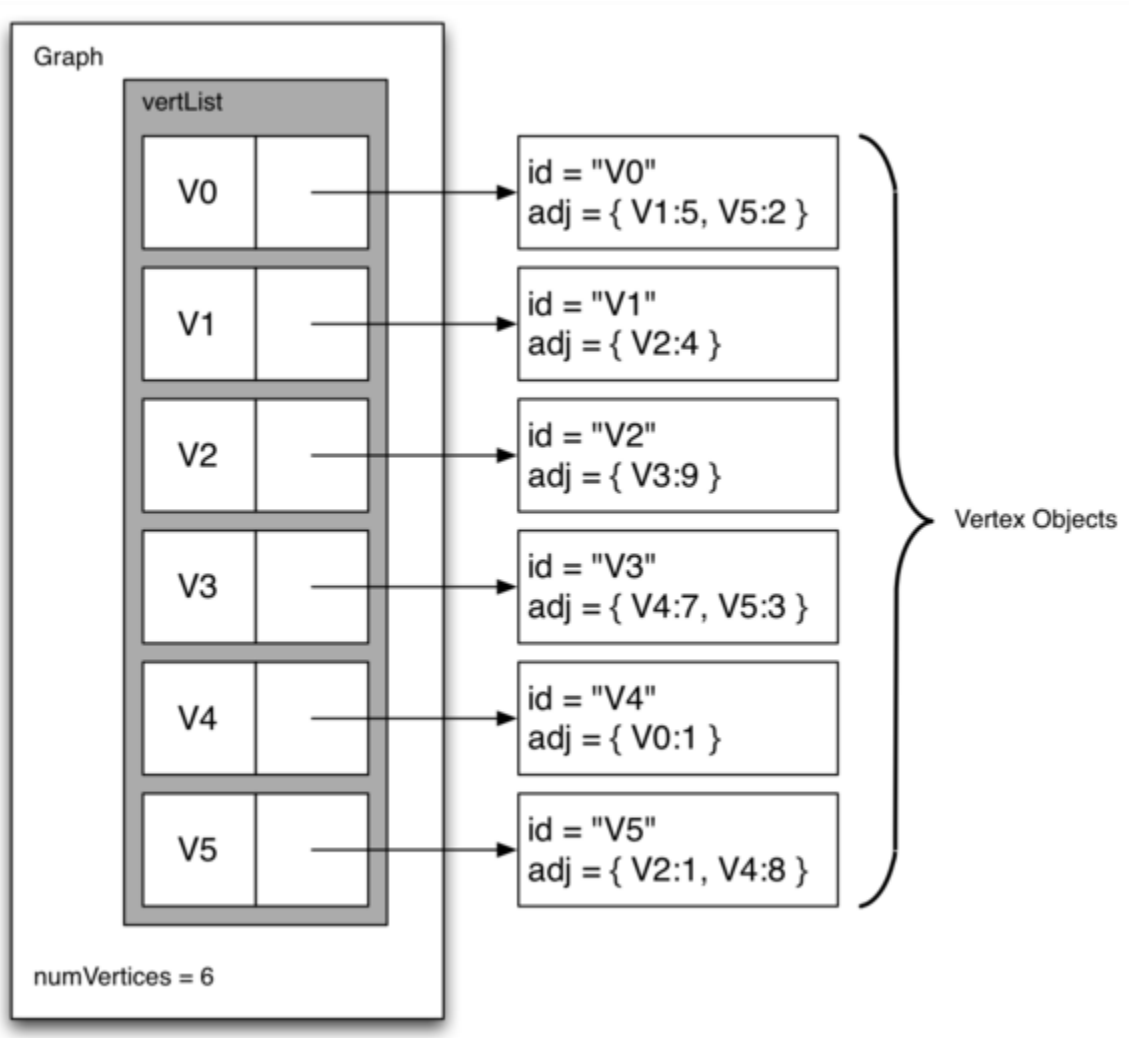
เมตริกซ์การเชื่อมโยง (adjacency matrix)

- ใช้ตารางแสดงความเชื่อมโยง
  - array 2 มิติ
- เห็นความเชื่อมโยงได้ง่าย
- เปลืองเนื้อที่
- ดีเมื่อมี edge เยอะ

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

ลิสต์การเชื่อมโยง (adjacency list)

- กะทัดรัด
- หาความเชื่อมโยงได้ง่ายกว่า



```python
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

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

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

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

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

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)

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

    def __iter__(self):
        return iter(self.vertList.values())
```

```
>>> g = Graph()
>>> for i in range(6):
...     g.addVertex(i)
>>> g.vertList
{0: <adjGraph.Vertex instance at 0x41e18>,
 1: <adjGraph.Vertex instance at 0x7f2b0>,
 2: <adjGraph.Vertex instance at 0x7f288>,
 3: <adjGraph.Vertex instance at 0x7f350>,
 4: <adjGraph.Vertex instance at 0x7f328>,
 5: <adjGraph.Vertex instance at 0x7f300>}
>>> 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:
...     for w in v.getConnections():
...         print("( %s , %s )" % (v.getId(), w.getId()))
...
( 0 , 5 )
( 0 , 1 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
```