Exercise 1 chart for recursive Graph-Exploration

(Take s as starting vertex and scan adjacency list from left to right)

Vertex	Adj. Vertices	seen	p (predecessor)
s	a m		
a	m c d		
m	b d		
b	d		
c	t		
d	t		
t			

Exercise 2 chart for recursive Graph-Exploration

(Take d as starting vertex and scan adjacency list from left to right)

Vertex	Adj. Vertices	seen	p (predecessor)
s	a m		
a	m c d		
m	b d		
b	d c s		
c	t s m		
d	a c t		
t	b		

Exercise chart for stack-based Graph-Exploration

Vertex	Adj. Vertices	seen	p (predecessor)
s	a m		
a	m c z		
m	b z		
b	z		
c	t		
z	t		
t			

Stack based exploration algorithm for destination "z"

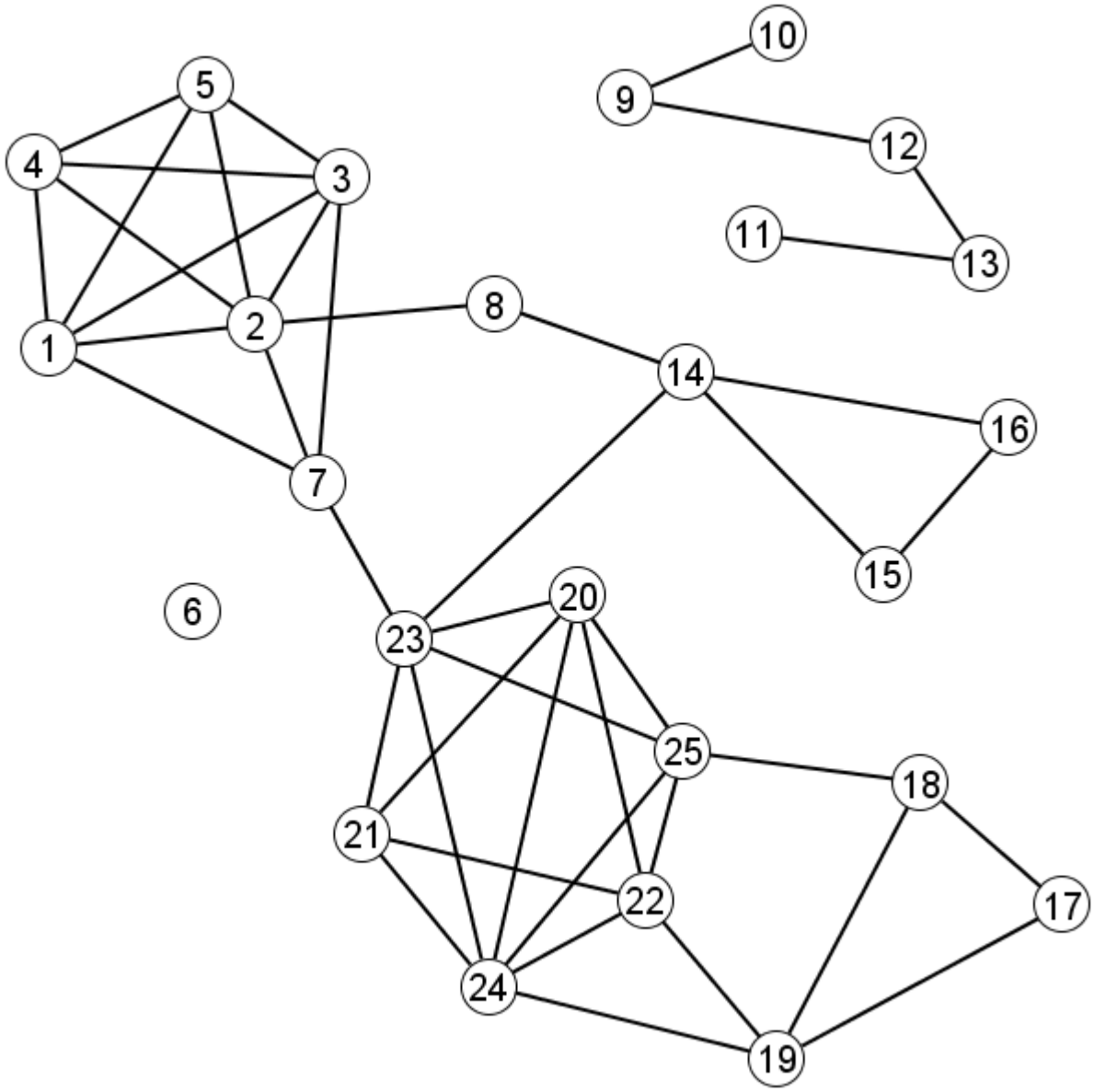
```
Tick mark "s" as seen, push it onto a stack
while stack is not empty
{
        cc = value popped from stack
        if cc is destination
        {
                job done: path is found
                display it using predecessor data
        }
        else
        {
                for each of its (cc) unseen Adj. Vertices
                        Tick mark it (Adj. Vertex) as seen,
                        p = cc // Assign cc to its (Adj. Vertex) p (predecessor)
                        push it (Adj. Vertex) onto the stack
        }
}
If Stack is empty: Path not found ➔ display path not found message
```

Run BFS (start from 20) and DFS on it

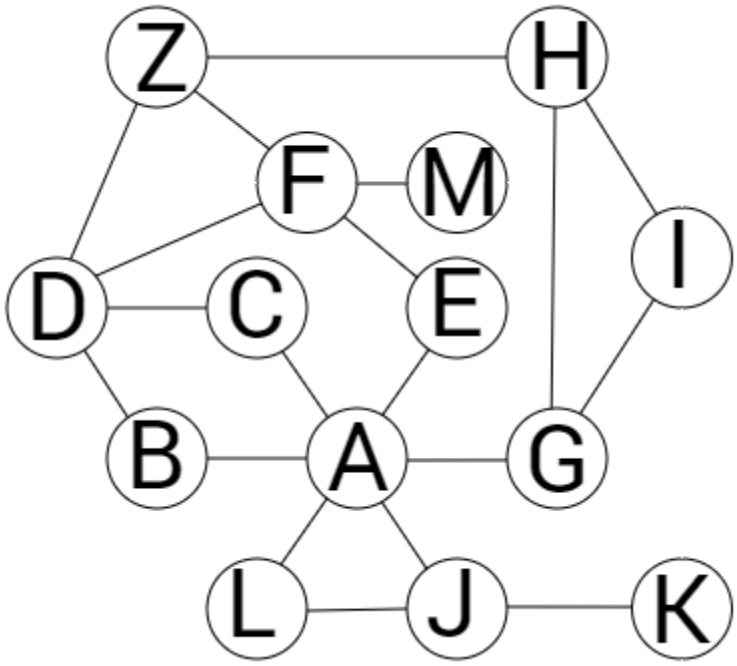
1. to compute d, p for BFS and d, f and p

2. Can you make its adjacency list and show working on it

3. Put random arrows and run DFS again

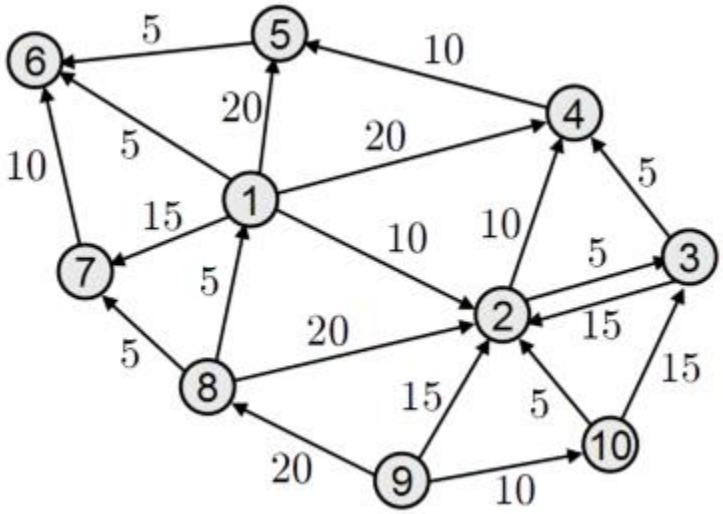


Make its adjacency matrix and run BFS (start M) on it to compute P only



Run Dijkstra algo on the following graph twice

1. Take 9 as source vertex
2. Take 3 as source vertex



Dijkstra's Algorithm PQ

Best	Average	Worst
O((V + E)*log V)	*same*	*same*

Weighted Directed Graph

Priority queue

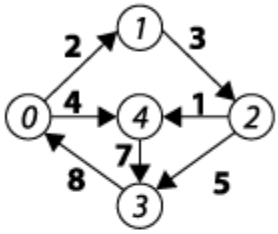
Array

Overflow

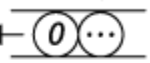
```
singleSourceShortest (G, s)
    PQ = new Priority Queue
    foreach v ∈ V do
        dist[v] = ∞
        pred[v] = −1
    dist[s] = 0
    foreach v ∈ V do
        insert (v, dist[v]) into PQ

    while (PQ is not empty) do
        u = getMin(PQ)
        foreach neighbor v of u do
            w = weight of edge (u,v)
            newLen = dist[u] + w
            if (newLen < dist[v]) then
                decreaseKey (PQ, v, newLen)
                dist[v] = newLen
                pred[v] = u
end
```

Create PQ from neighbors **v** of vertex **s** = 0 based on dist[**v**]



PQ



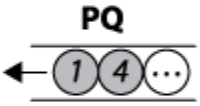
0	1	2	3	4
0	∞	∞	∞	∞

dist[v]

Remove vertex **u** from PQ with least distance from **s**. If path from (**s**,**u**) and (**u**,**v**) is shorter than best computed distance (**s**,**v**), adjust dist[**v**] and PQ.

1st iteration: remove 0 and adjust

PQ



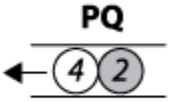
0	1	2	3	4
0	2	∞	∞	4

(0,0) + (0,1) < (0,1)

(0,0) + (0,4) < (0,4)

2nd iteration: remove 1 and adjust

PQ

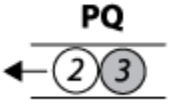


0	1	2	3	4
0	2	5	∞	4

(0,1) + (1,2) < (0,2)

3rd iteration: remove 4 and adjust

PQ

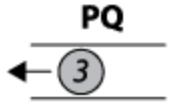


0	1	2	3	4
0	2	5	11	4

(0,4) + (4,3) < (0,3)

4th iteration: remove 2 and adjust

PQ

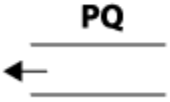


0	1	2	3	4
0	2	5	10	4

(0,2) + (2,3) < (0,3)

5th iteration: remove 3 and done

PQ



0	1	2	3	4
0	2	5	10	4