COMP20003 Workshop Week 10

Graphs

Graphs: concepts, properties
Graph representation
Graph Traversal
DFS, BFS, path finding
Implicit Graphs

LAB:
- do Peer Activities & exercises, and (perhaps) write a short program

Graphs: Concepts

Formal definition:

G= (V, E) where

V={v_i} : set of *vertices*

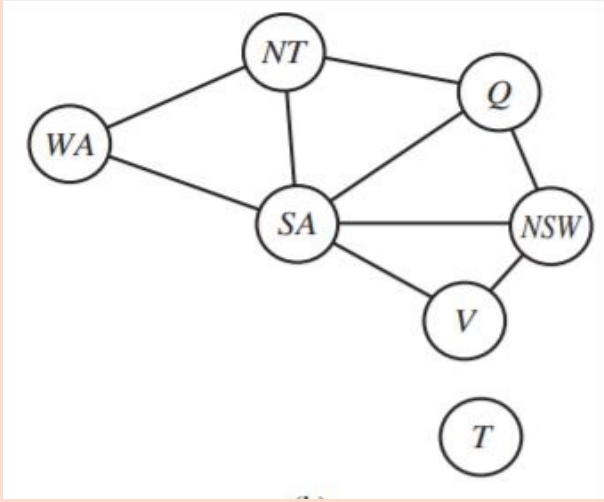
E={ **(v_i, v_j)** | v_i ∈ V, v_j ∈V}

: set of *edges*;

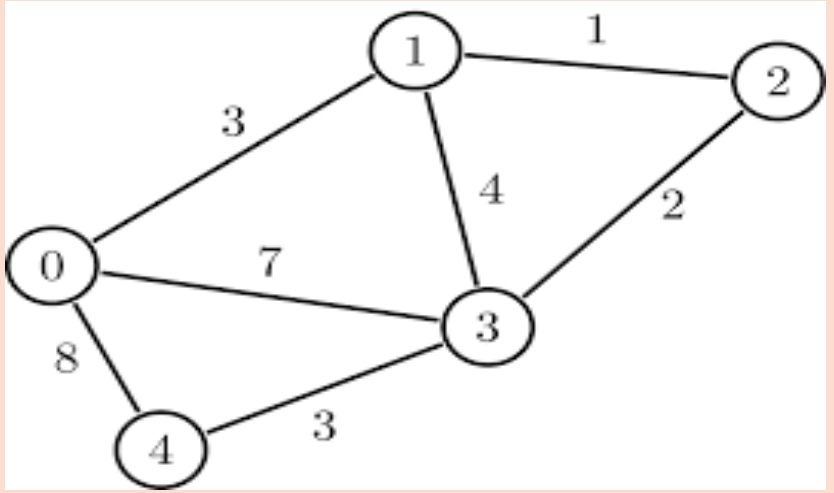
Understanding graph properties:

- *weighted, unweighted*
- *cyclic, acyclic*
- *connected, unconnected graph, connected component*
- *dense, sparse* graphs
- *directed (di-graph), undirected, DAG*
- *weakly and strongly connected di-graph, strongly connected components*

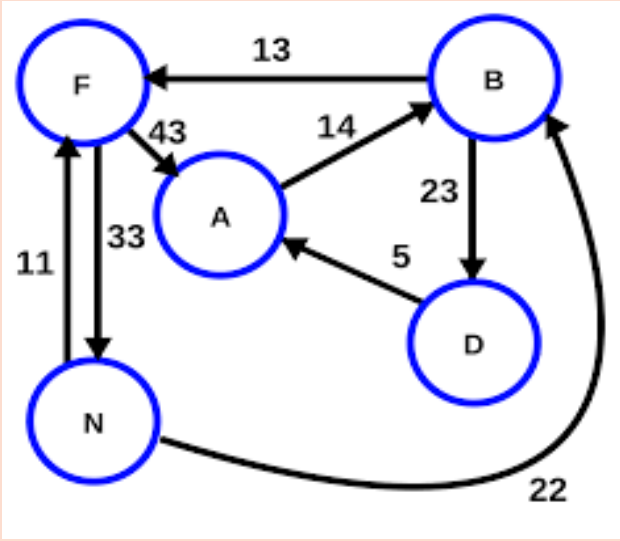
Graph A



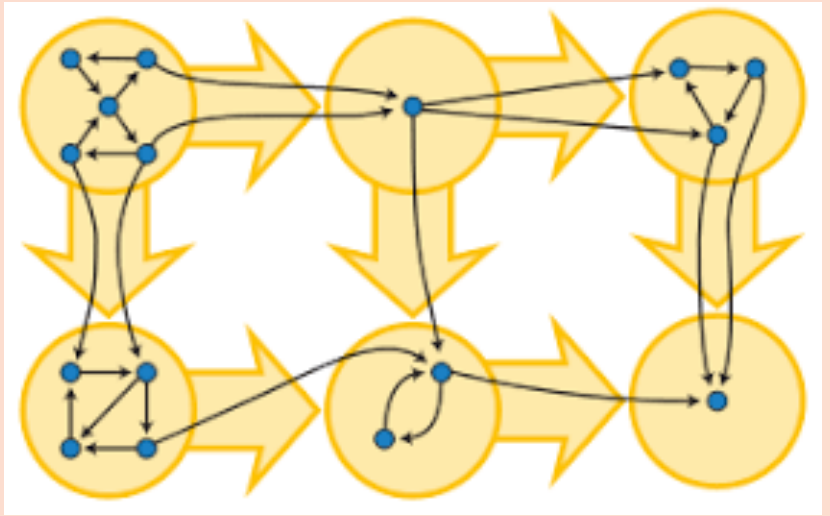
Graph B



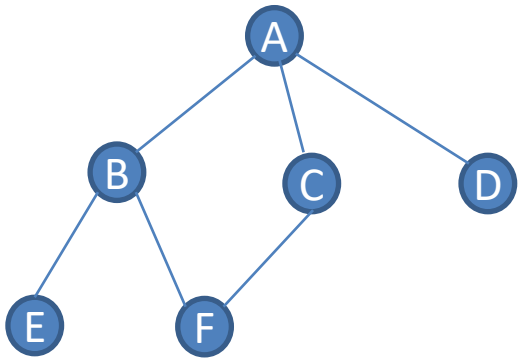
Graph C



Graph D



Graph representation

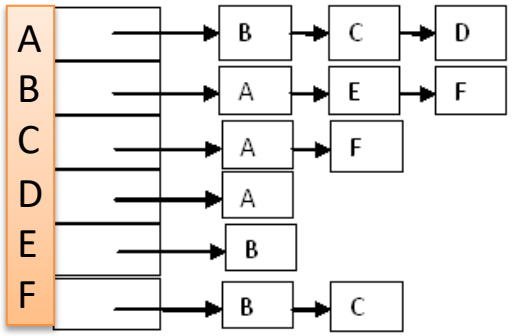


Vertices: Vertices can be represented as integers 0..|V|-1

- If need to store some other properties of each vertices (such as vertex label) we can put them in an array.
- labels[]= {“A”, “B”, “C”, “D”, “E”, “F”};

Edges: depending on how to represent edges, we have different graph representations:

using *array of adjacency lists*

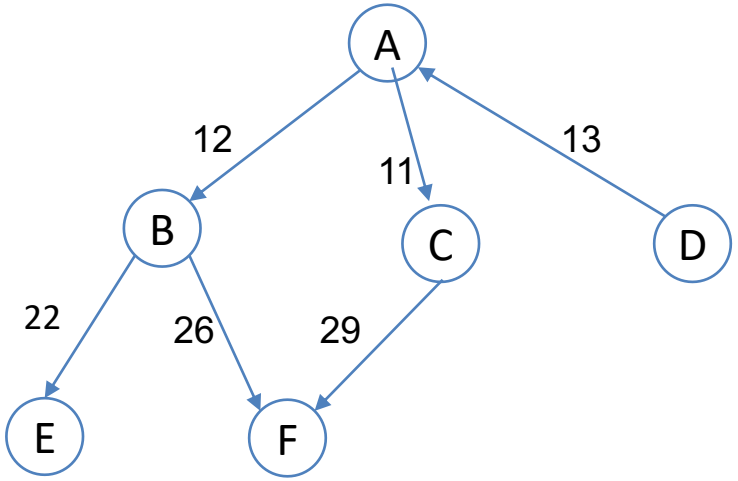


using *adjacency matrix*

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	0	1
D	1	0	0	0	0	0
E	0	1	0	0	0	0
F	0	1	1	0	0	0

Note: adjacency list don’t need to be linked lists

weighted di-graph example: Using Adjacency Lists

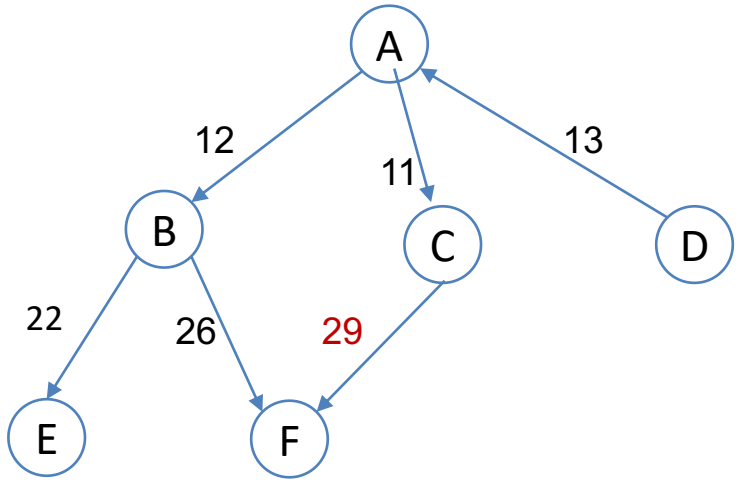


input data

```
"A","B",12
"B","F",26
"B","E",22
"D","A",13
"A","C",11
"C","F",29
```

id	name	adj list
0	"A"	→ 1,12 → 5,11
1	"B"	→ 2,26 → 3,22
2	"F"	
3	"E"	
4	"D"	→ 0,13
5	"C"	→ 2,29

weighted di-graph example: Using Adjacency Matrix



input data

```
"A","B",12
"B","F",26
"B","E",22
"D","A",13
"A","C",11
"C","F",29
```

id	name
0	"A"
1	"B"
2	"F"
3	"E"
4	"D"
5	"C"

A is a matrix of V x V		TO					
		0	1	2	3	4	5
FROM	0		12				11
	1			26	22		
	2						
	3						
	4	13					
	5				29		

NOTES

Aij =

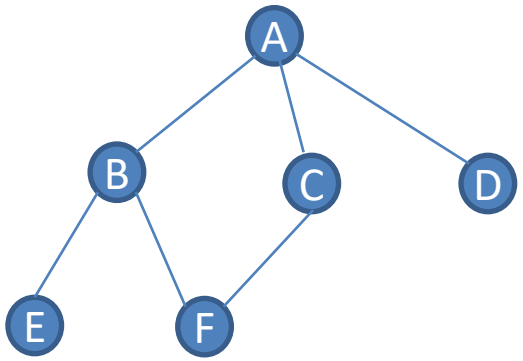
- weight of edge i→j
- 1 or 0 if graph is unweighted

For weighted graph, empty cells:

- are normally "undefined"
- depending on situations, could be set as 0 or ∞

Notes: if graph is undirected, the matrix is symmetric

Graph representation: Space Complexity

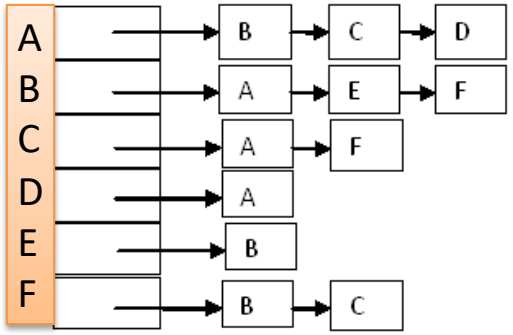


Time/Space Complexity of graph algorithms is represented as a function of:

- $|V|$ *(or just* V*): number of vertices*
- $|E|$ *(or just* E*): number of edges*

Remember $E = O(V^2)$, *and sometimes* $E = O(V)$ → *we can also use just a function of just* V

using *array of adjacency lists*

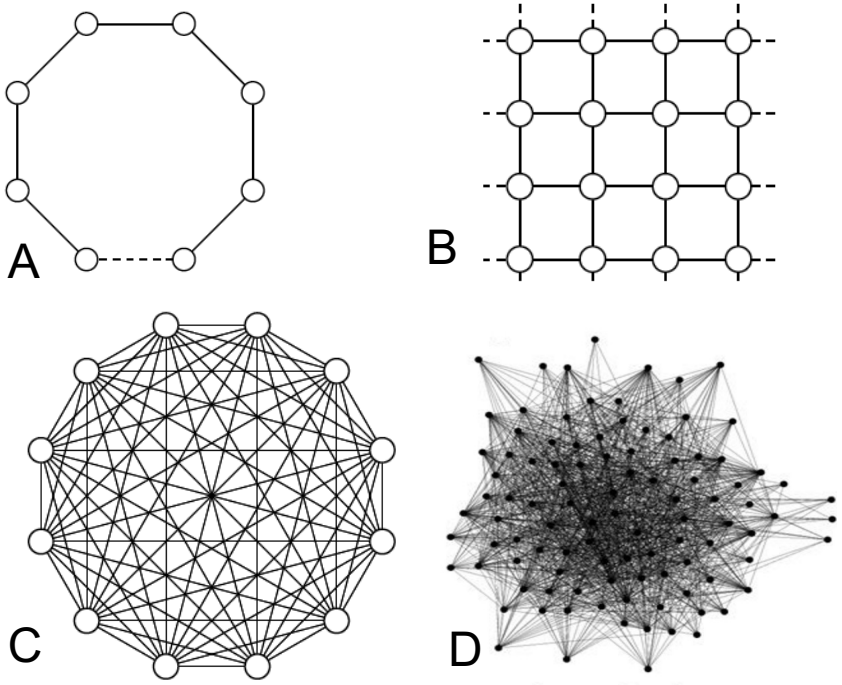


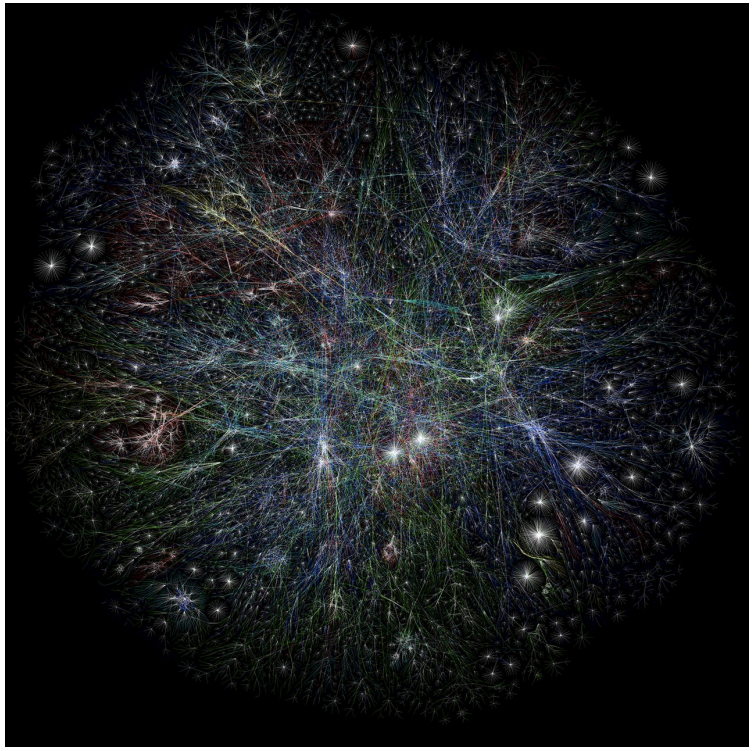
using *adjacency matrix*

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	0	1
D	1	0	0	0	0	0
E	0	1	0	0	0	0
F	0	1	1	0	0	0

Space complexity

Graphs Representation: sparse & dense graphs





E: The Internet

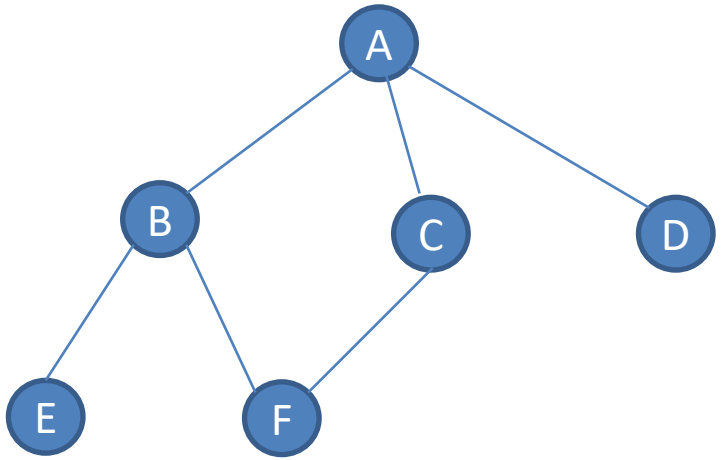
Is each graph dense/sparse?

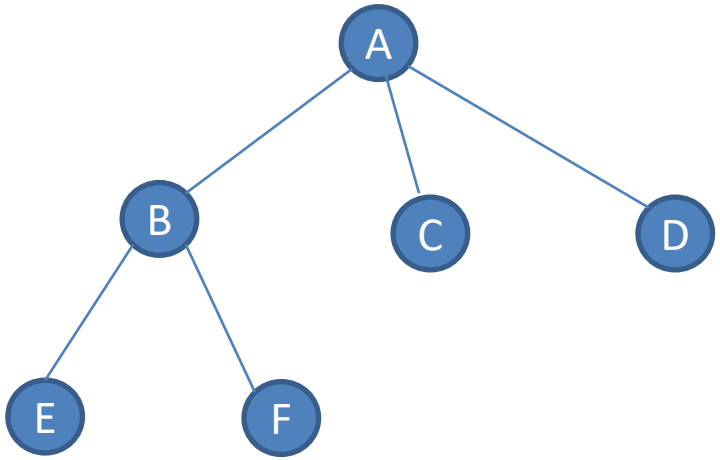
What is the criteria for:

- *Sparse graphs: E= O(V)*
- *Dense graphs: E= $\Theta(V^2)$ (or $\Omega(V^2)$)*

What representation is better for each case?

Graph Traversal = What? How?





Traversal: visit nodes of a graph in a systematic way.

Understanding the 2 main strategies for graph traversal:

- DF (Depth-First) Traversal
- BF (Breath-First) Traversal

Compare with binary tree travel

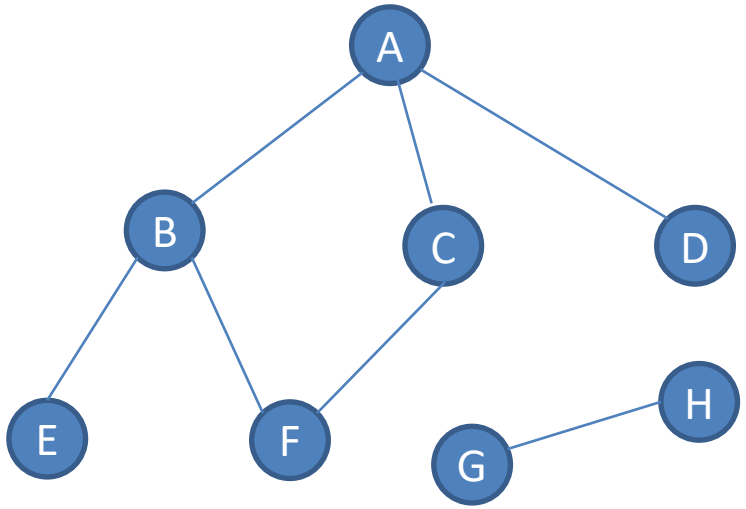
- pre-order: DF
- level-order: BF

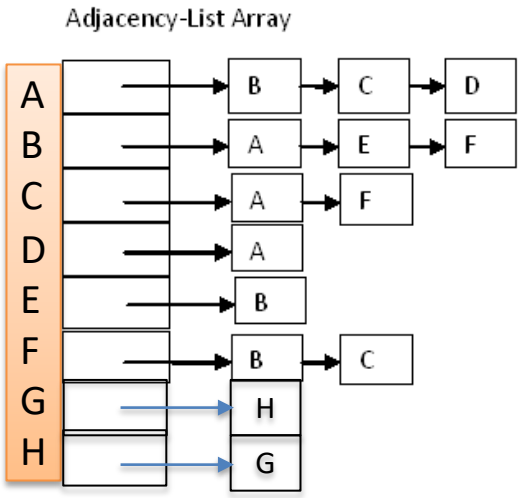
Exercise: for the top graph, list the nodes in visited order by:

DF: A ...

BF: A ...

Breath-First Traversal= level-by-level using a queue



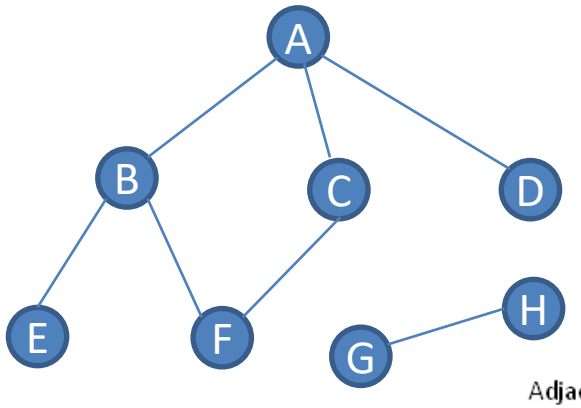


What's wrong with the following function?

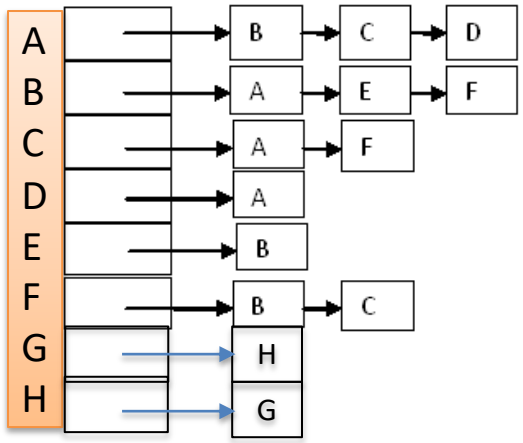
The function can be used for a tree, but is likely incorrect for a general graph. Why? How to fix?

```
function BFSvisit(int u) {
  Q= empty queue
  enQ(Q, u)
  while (Q not empty) {
    u = deQ(Q)   // visit u
    for each (u,v)
      enQ(Q,v)
  }
}
```

Breath-First Traversal



Adjacency-List Array



	adj list	adj matrix
sparse	θ( )	θ( )
dense	θ( )	θ( )

```
for each node u
  visited[u] = 0;     //mark all nodes as "unvisited"
for each node u
  if (!visited(u)) BFSvisit(u);

function BFSvisit(int u) {
  Q= empty queue
  enQ(Q, u)
  while (Q not empty) {
    u = deQ(Q)
    if (!visited[u]) {
       visited[u]= 1; // visit u, and mark as visited
       for each (u,v)
          if (!visited[v]) enQ(Q,v)
    }
  }
}
```

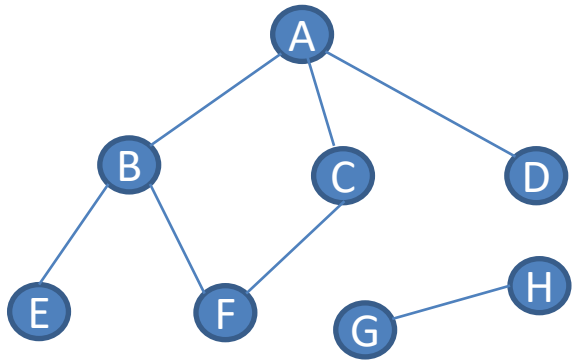
Q: List the nodes in order of visited by BFS. *On tie choose the smallest element.*

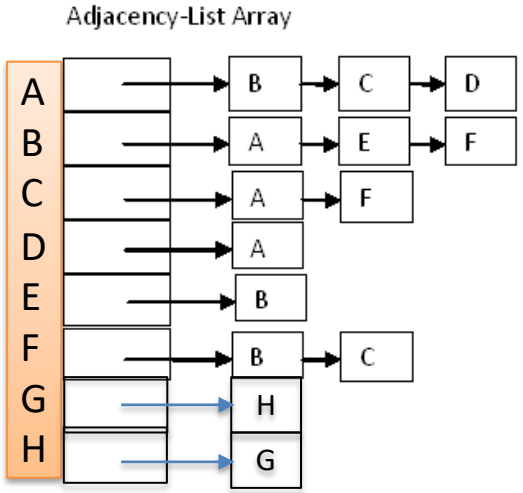
? A

Complexity=

- using Adj Lists: O(?) θ(?)
- using Adj matrix: O(?) θ(?)
- what if graph is sparse/dense?

Depth-First Traversal= using recursion or a stack





	adj list	adj matrix
sparse	θ ( )	θ ( )
dense	θ ( )	θ ( )

```
timestamp= 0;
for each node u
  visited[u] = timestamp;    //mark as "unvisited"
for each node u
  if (!visited(u)) DFSvisit(u);


function DFSvisit(int u) {

  visited[u]= ++timestamp;
  for each edge (u,v) {
    if (!visited[u])
      DFSvisit(v);
  }

}
```

The Visit

Q: List the nodes in order of visited by DFS.

? A

Other questions:

- What's the use of `timestamp`
- Complexity in comparison with BF traversal?

Graph Search

Basic Concept:

- *Graph Search* is *Path Finding*.
- The Task: given a source node S and a destination node D, find a path from S to D.
- Output: a path $S \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{n-1} \rightarrow D$ with n is the path length
- Note: in most of the cases, the objective ss to find a *Shortest Path*: a path that has minimal possible length

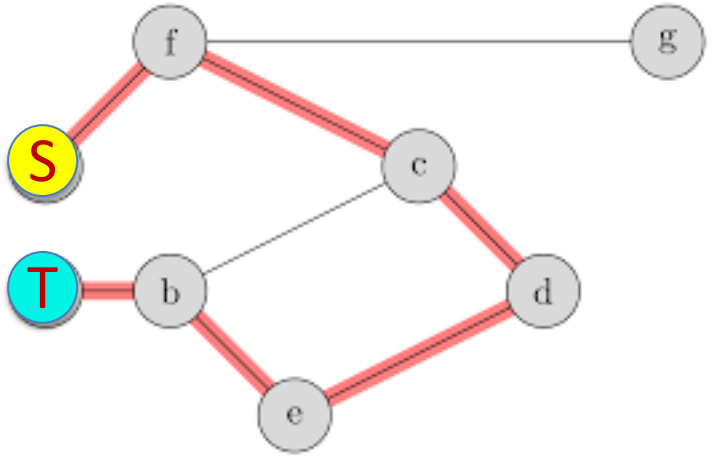
Examples:

- GPS or Google Map

A **classification** of Graph Search (not quite popular):

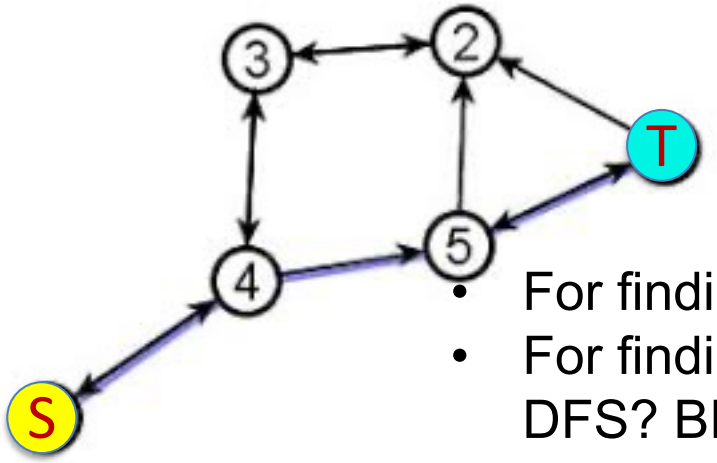
- 1S1D: search for a path from a single Source to a single Destination
- 1S*D: single source, all possible destinations
- *S*D: ...

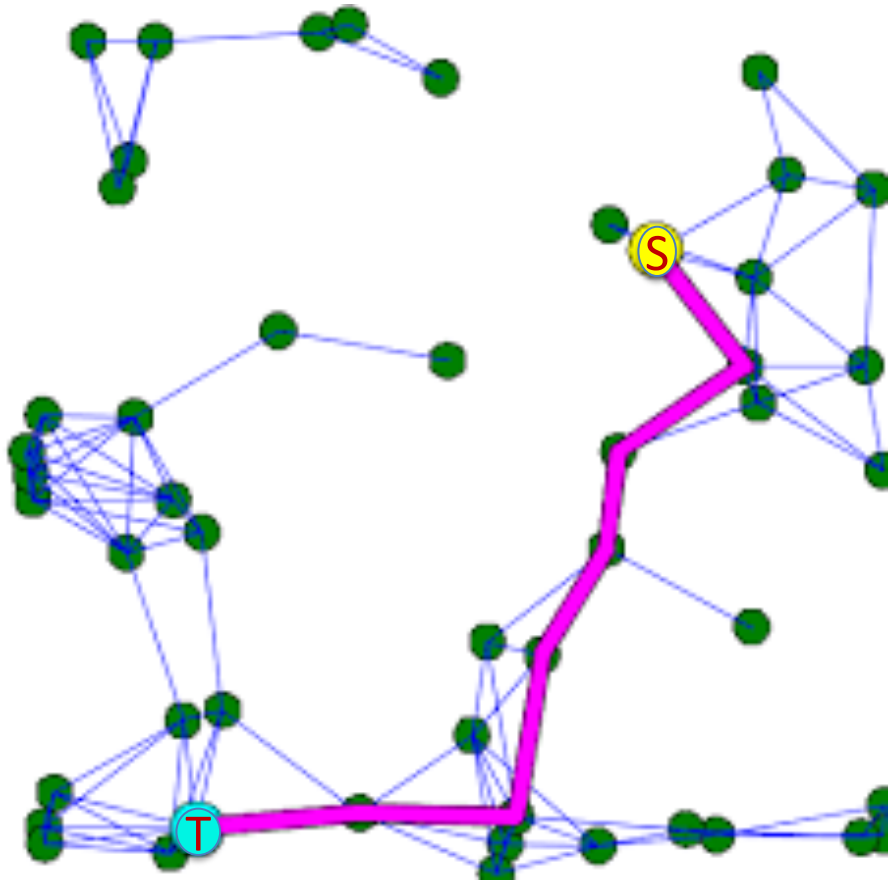
Paths in unweighted graphs: path length, shortest path



Path= S→f→c→d→e→b→T, length=6

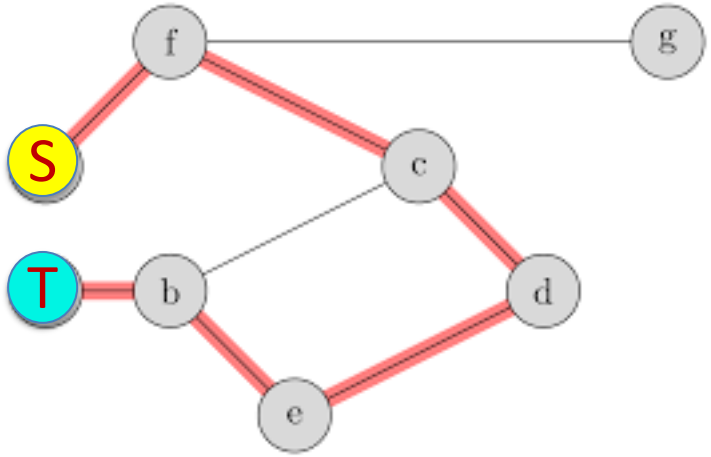
Path= S→4→5→T, length= 3





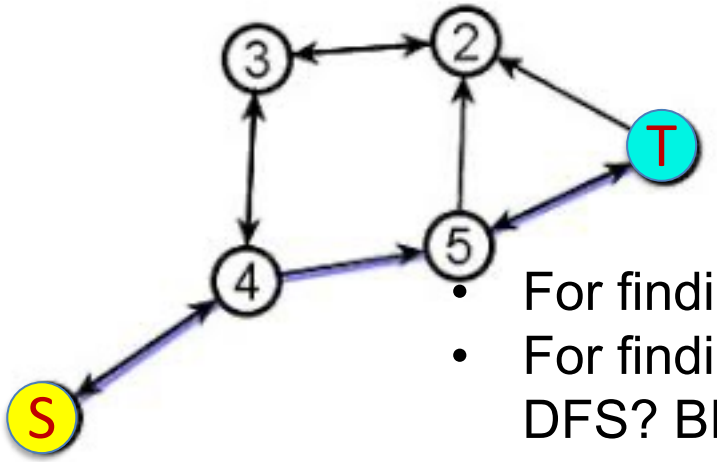
- For finding a path from S to T can we use DFS? BFS?
- For finding a shortest path from S to T can we use DFS? BFS?
- Applied to directed graphs? cyclic graphs?

BFS for shortest path



Path= S→f→c→d→e→b→T, length=6

Path= S→4→5→T, length= 3



- For finding a path
- For finding a shor
  DFS? BFS?
- Applied to directe

Adapting BF traversal for shortest path in unweighted graphs

```
function BFS(int source, int dest) {
  Q= empty queue
  enQ(Q, source)
  while (Q not empty) {
    u = deQ(Q)
    ??
    ??
    if (!visited[u]) {
       visited[u]= 1; // visit u, and mark as visited
       for each (u,v)
          if (!visited[v]) enQ(Q,v)
    }
  }
}
```

Q:

- Can we use DFS for the same purpose?
- Can we use this algorithm for weighted graphs?

Lab Time

- Start with the peer activity W10.10
- Do W10.1, W10.10, W10.2
- Programming: Build an undirected graph in a simple way such as
  - using adjacency matrix, or
  - using adjacency list, but use static arrays for adjacency lists

  then use DFS to find a maximal connected component (a component with highest number of nodes) of the graph.

Input graphs:

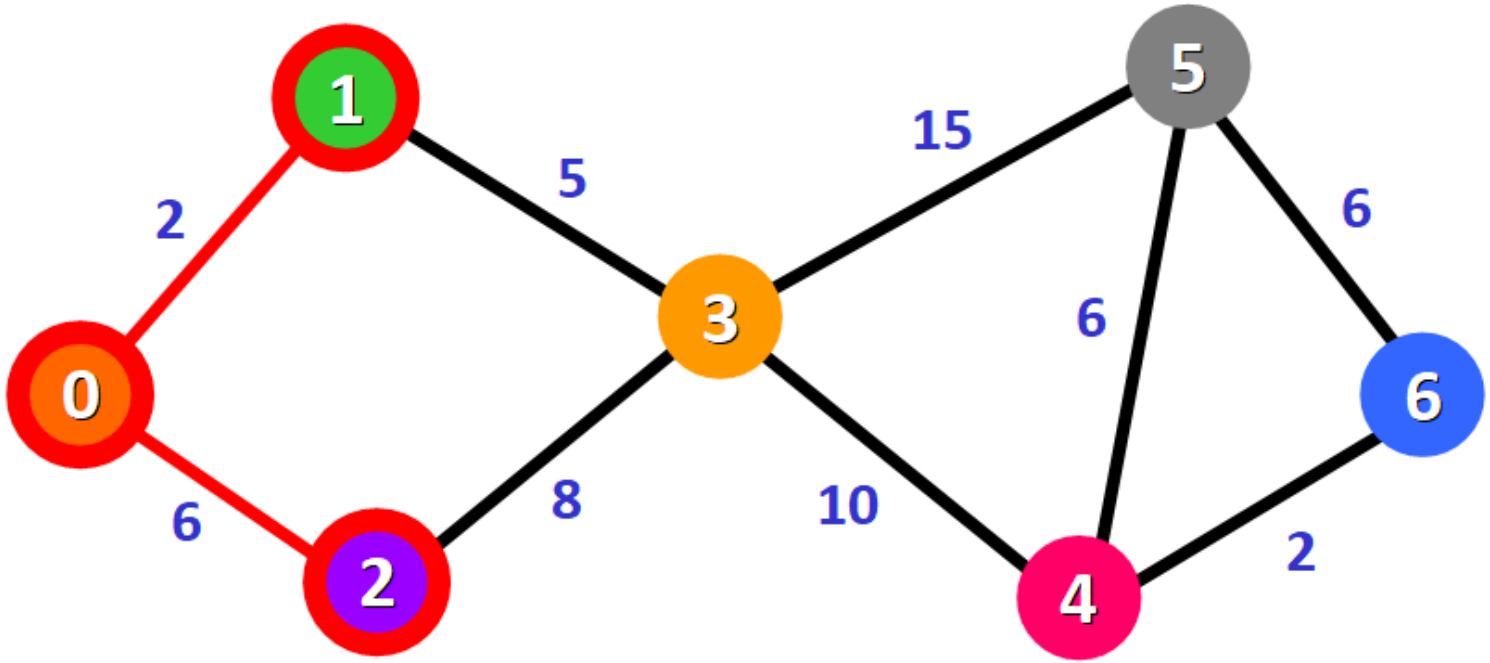
Example 1: unweighted graph

```
0   1        # pair   i   j  means there is an edge between i and j
1   2
0   3
4   5
6   5
7   8
```

Example 2: weighted graph described in question W10.2

Additional Slides

How about shortest path on weighted graphs?



Path from 0 to 3:

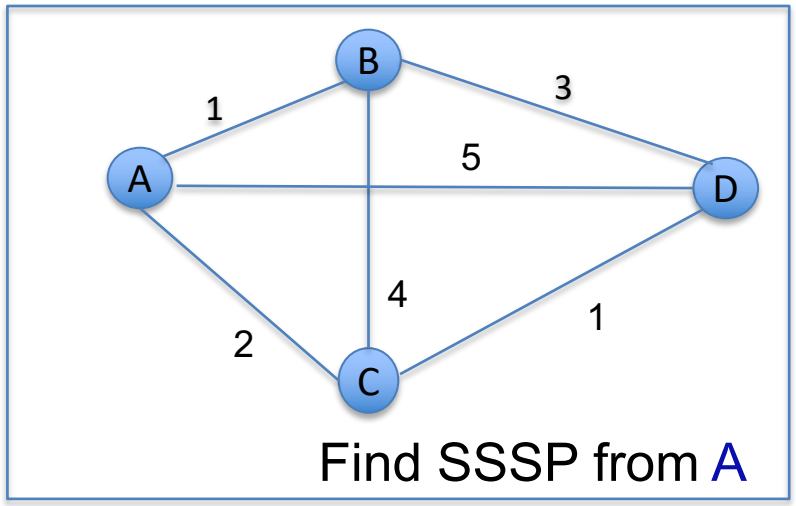
- possible paths: 0→1→3, 0→2→3
- shortest path: 0→1→3 with total weight= 7

Dijkstra's Algorithm: Single Source Shortest Path SSSP

The task:

- Given a weighted graph $G=(V,E,w(E))$, and $s \in V$, and supposing that *all weights are positive*.
- Find shortest path (path with min total weight / min distance) from s to all other vertices.

Input

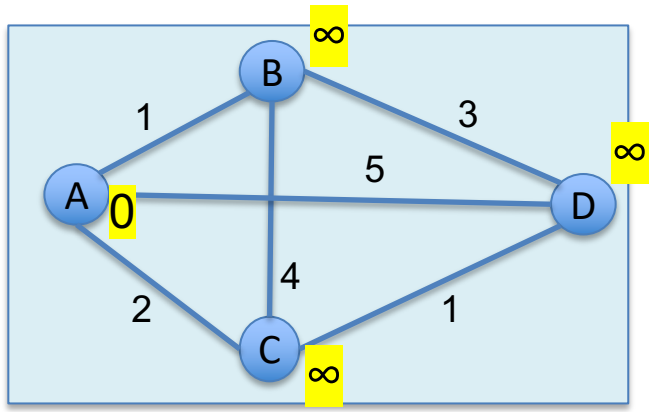


Output

Destination	Shortest paths	Distance
A	A→A	0
B	A→B	1
C	A→C	2
D	A→C→D	3

Very useful (think about Google Map, where weight can be time or distance).

But how?



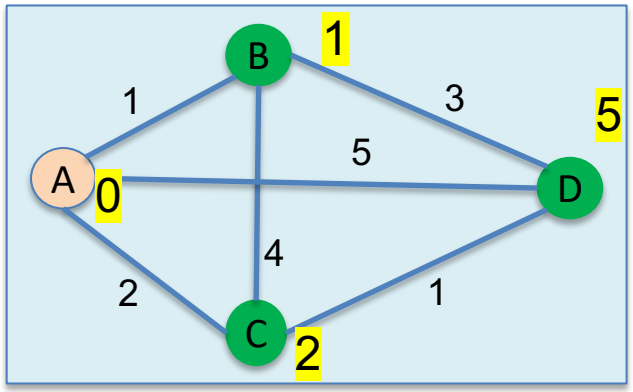
Find a shortest path:

- From A to any other node

A is the source:

- shortest path A→A has `dist[A]= 0`
- and distance-so-far `dist[]= {0, ∞, ∞, ∞}` (for {A,B,C,D})

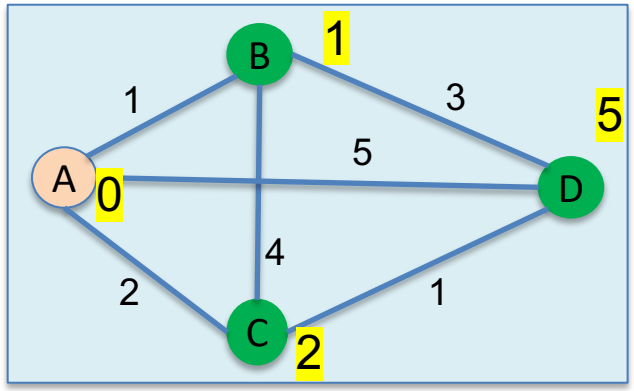
→ Clearly, we've done with A, and can remove it from the job list!
But with that, we should use the information from A to update
`dist[]= {0, 1, 2, 5}`



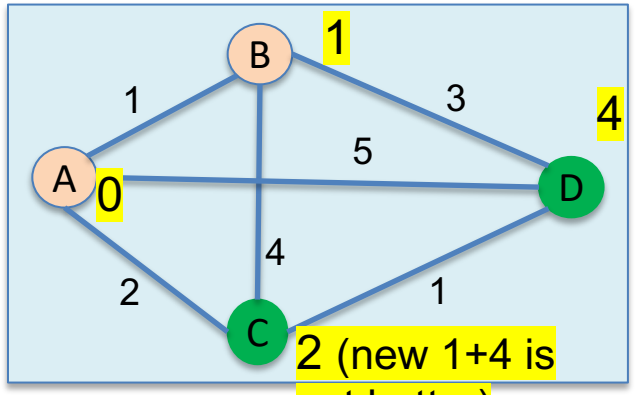
What's next?

But with that, we should use the information from A to update

`dist[]= {0, 1, 2, 5}`

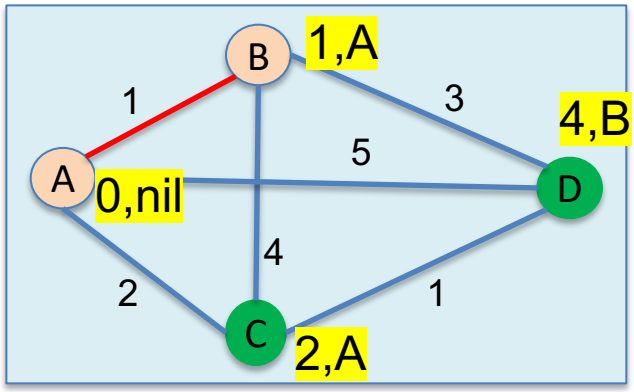


What's next?
explore C, B, or D?



So shortest path from A to B found with `dist[B]= 1`.

But how can we retrieve the detailed path A→B?
need to keep track of ???



	A	B	C	D
	0, nil	∞,nil	∞,nil	∞,nil
A		**1,A**	2,A	**5,A**
B			2,A	**4,B**

set {B,C,D} includes not-yet-done elements at this stage.
How to best keep track of not-yet-done distance?

this column: nodes with found shortest path

dist[B]: shortest-so-far distance from A

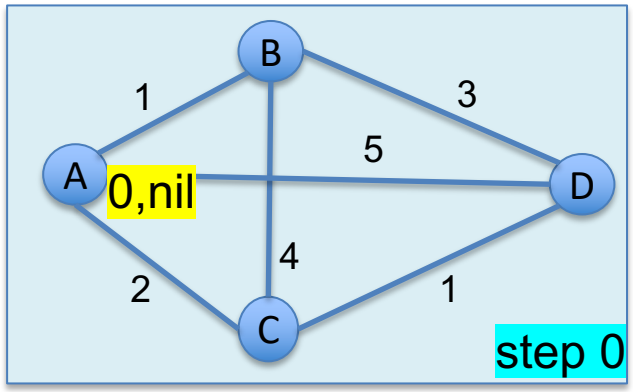
prev[D]: node that precedes D in the path A→D

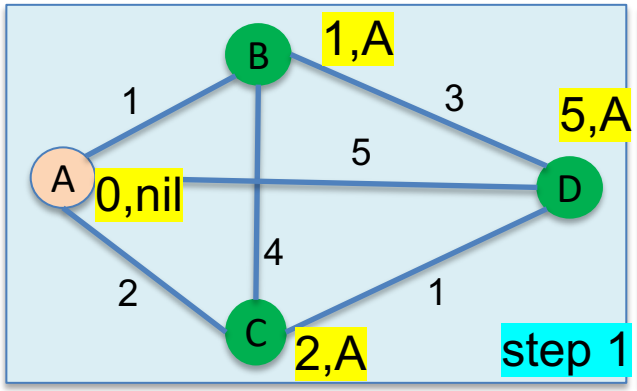
Dijkstra's algorithm

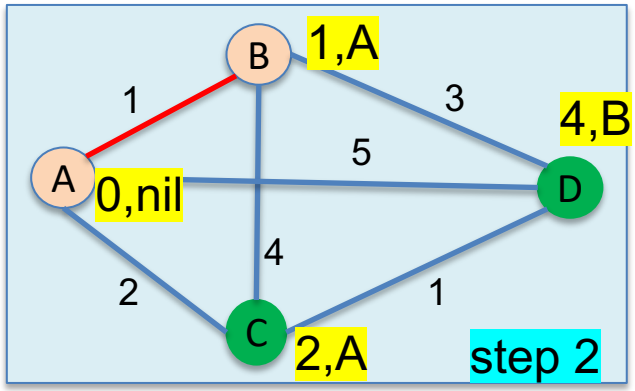
```
set dist[u]= ∞, prev[u]=nil for all u
set dist[s]= 0
set PQ= makePQ(V)
while (PQ not empty)
    u= deleteMin(PQ)
    visit  u                    // practice: just mark u as visited
    for all (u,v) in G:
     if (dist[u]+w(u,v)<dist[v]):
        update dist[v] and pred[v]
```

Programming note: "update dist[v] and pred[v]" means

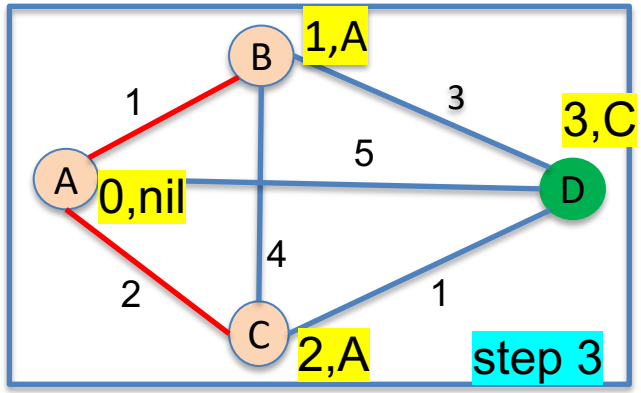
1. dist[v]= dist[u]+w(u,v), pred[v]= u
2. decrease weight of v in PQ to dist[v], hence, need to locate v in PQ, change weight and upheap. There is a way to do the whole operation in O(log n) [not for this course]

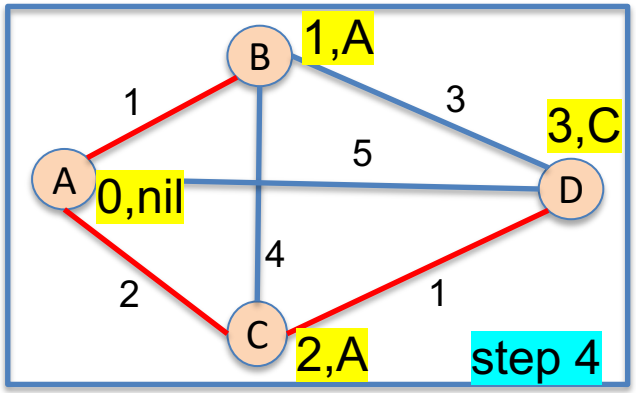






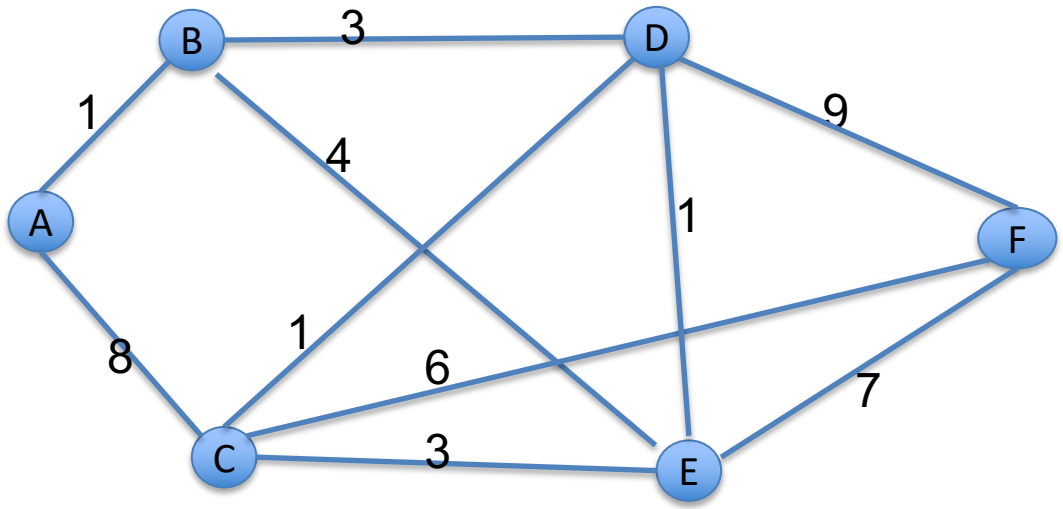
Running Dijkstra from node A, ie. finding shortest paths from A to all nodes.
The number at each **node** is the total distance from **A** to this **node** = distance of *previous node* + weight of the edge (*previous node*, **node**)





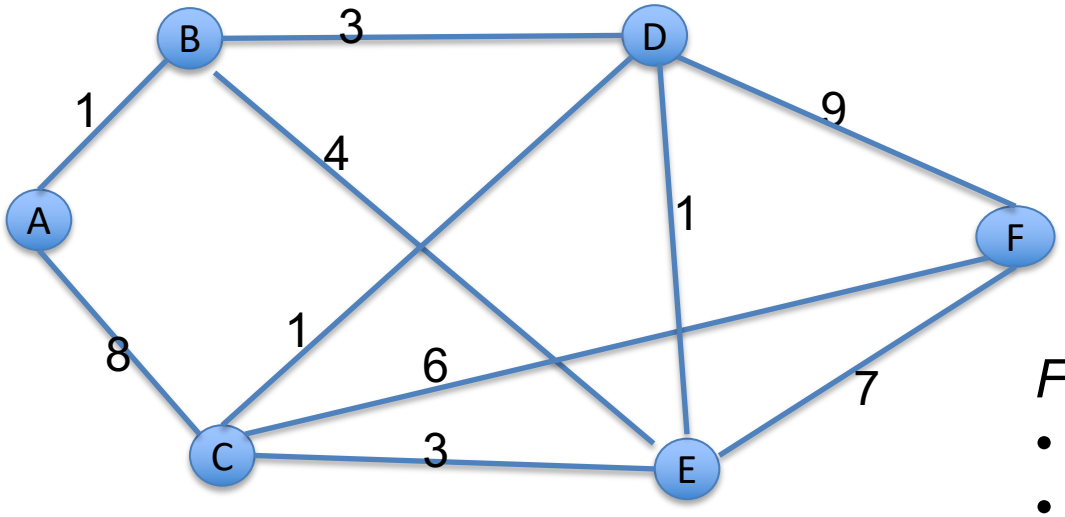
step	node with found shortest path	A	B	C	D
0		0,nil	∞,nil	∞,nil	∞,nil
1	A		1,A	2,A	5,A
2	B			2,A	4,B
3	C				3,C
4	D				

DIY: run Dijkstra's Algorithm for this graph



Find a shortest path:

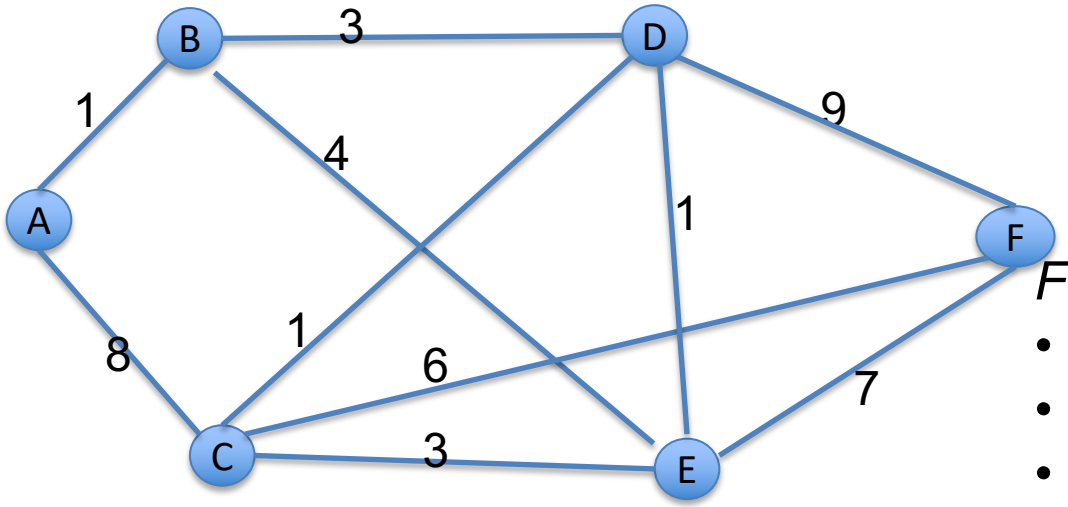
- From A to B
- From A to C
- From A to F
- From A to any other node



Find a shortest path:

- From A to B
- From A to C
- From A to F
- From A to any other node

	A	B	C	D	E	F
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
A		**1,A**	8,A	∞,nil	∞,nil	∞,nil
B			8,A	**4,B**	**5,B**	∞,nil
D						



Find a shortest path:

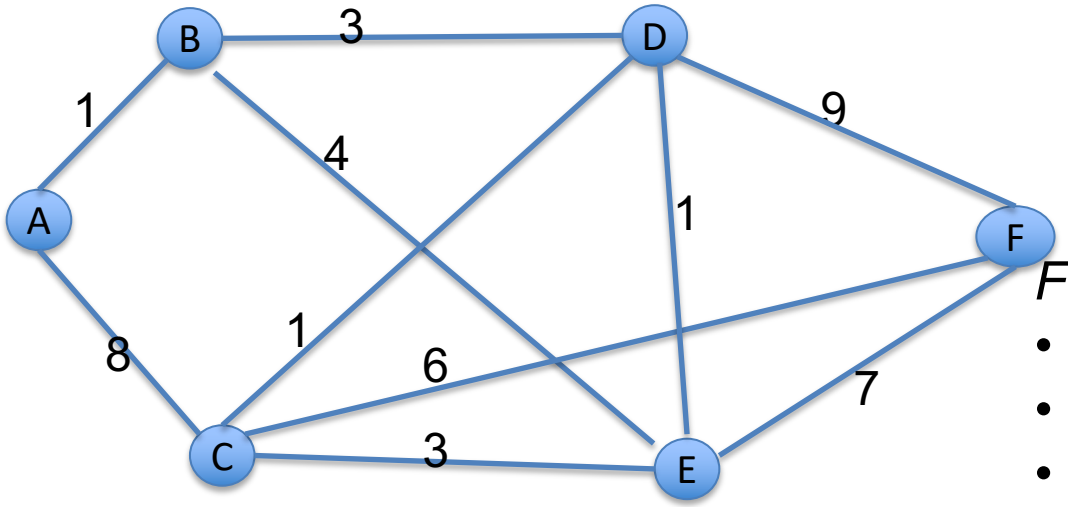
- From A to B
- From A to C
- From A to F
- SP A->F=

4 The dist at SA is 0, there is an edge A->C with length 8, so we can reach C from A with distance 0+8, and 8 is better than previuosly-found distance of ∞

done	A	B	C	D	E	F
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
A		**1,A**	8,A	∞,nil	∞,nil	∞,nil
B			8,A	**4,B**	**5,B**	∞,nil
D			5,D		**5,B**	**13,D**
C					**5,B**	**11,C**
E						**11,C**
C						

Update this cell because now we can reach C from D with distance 4 (of D) + 1 (of edge D→C), and 5 is **better** than 8

At this pointy, we can reach E from D with distance 4 (of D) + 1 (of edge D→E), but new distance 5 is **not better** than the previously found 5, so no update!



Find a shortest path:

- From A to B
- From A to C
- From A to F
- SP A->F=

What's the found shortest path from A to F?
distance= 11, path=A→B→D→ C→F

done	A	B	C	D	E	F
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
A		**1,A**	8,A	∞,nil	∞,nil	∞,nil
B			8,A	**4,B**	**5,B**	∞,nil
D			5,D		**5,B**	**13,D**
C					**5,B**	**11,C**
E						**11,C**
C						

pred[B]= A:
A→B→D→ C→F

pred[D]= B:
B→D→ C→F

pred[C]= D:
D→ C→F

pred[F]= C, that is we came to F from C: C→F

the shortest distance from A to F is 11