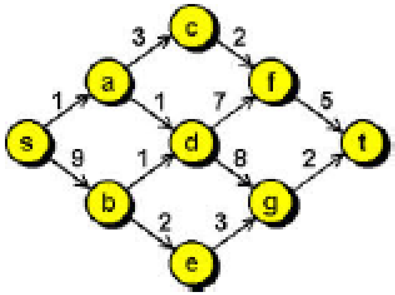
Part 3
Graph algorithms

Algorithm

- Graph Traversal
- Find shortest Path
- Minimum Spanning Tree

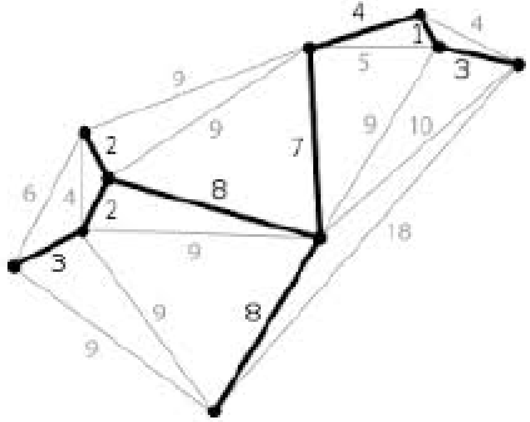
Graph Traversal

- Depth-First Search Algorithm (DFS), Breadth-First Search Algorithm (BFS).



Shortest Path

Dijkstra's Algorithm, *Bellman-Ford Algorithm (BFS)*



Minimum Spanning Tree

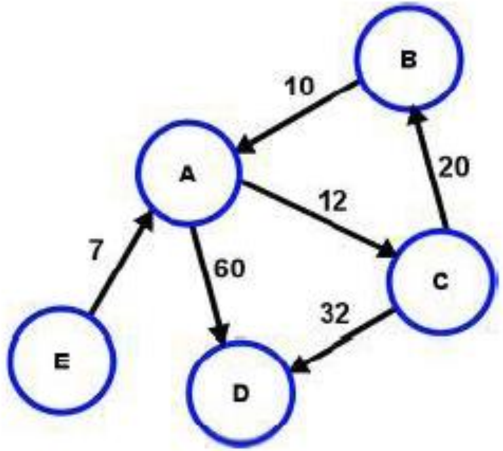
Prim's Algorithm, *Kruskal's Algorithm*

DFS+BFS

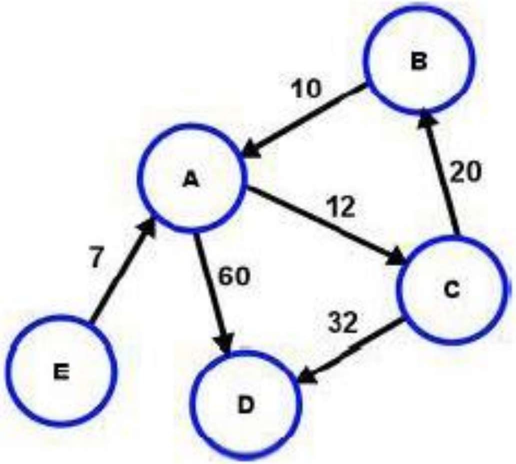
- To **visit all vertices**
  - output will be the sequence of visited vertex.

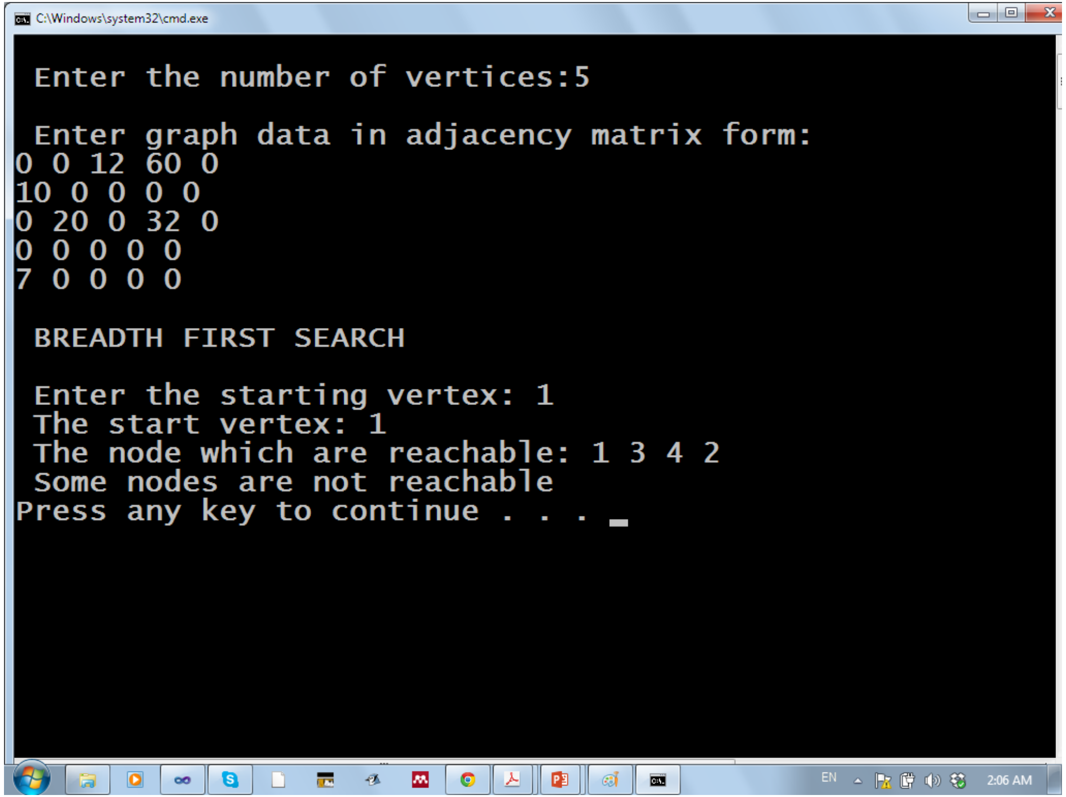
- In tree structure, traversal begin with root to leaf node : In-order, Pre-order, Post-order.

- Try traverse the graph in Figure (c) in your own way.



(c)


B
10
A
20
12
7
60
C
32
E
D
(c)

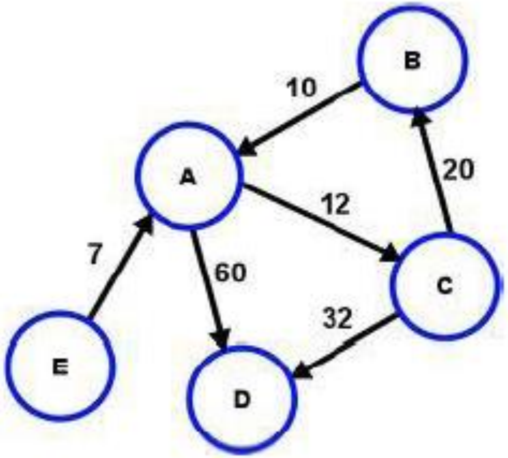

C:\Windows\system32\cmd.exe
Enter the number of vertices:5
Enter graph data in adjacency matrix form:
0 0 12 60 0
10 0 0 0 0
0 20 0 32 0
0 0 0 0 0
7 0 0 0 0
BREADTH FIRST SEARCH
Enter the starting vertex: 1
The start vertex: 1
The node which are reachable: 1 3 4 2
Some nodes are not reachable
Press any key to continue . . .
EN
2:06 AM

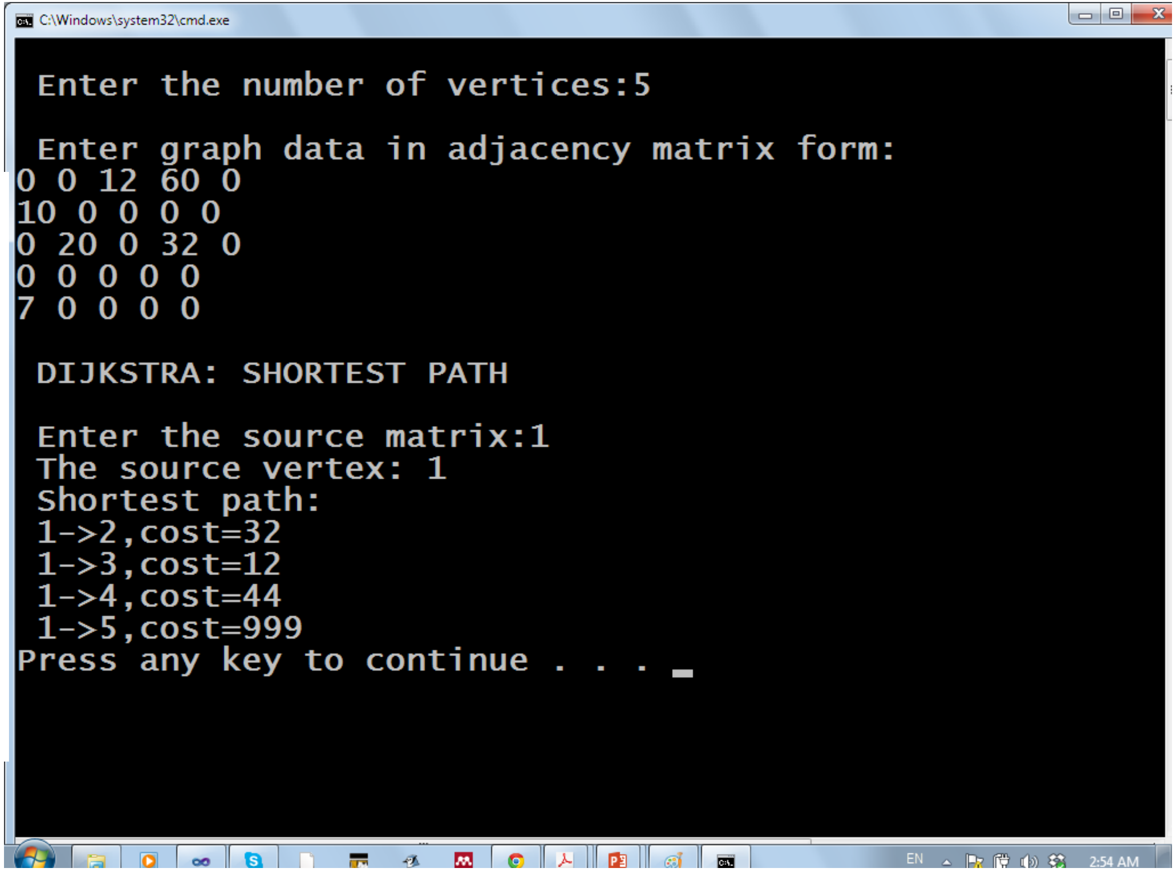
- Strategy:
  - from top to bottom (DFS Algorithm)
  - from left to right (BFS Algorithm)

- Graph traversal can have a **cycle** = visit the same vertex more than once.
  - While traverse, **record all visited vertex**, and only display the new visited vertex.

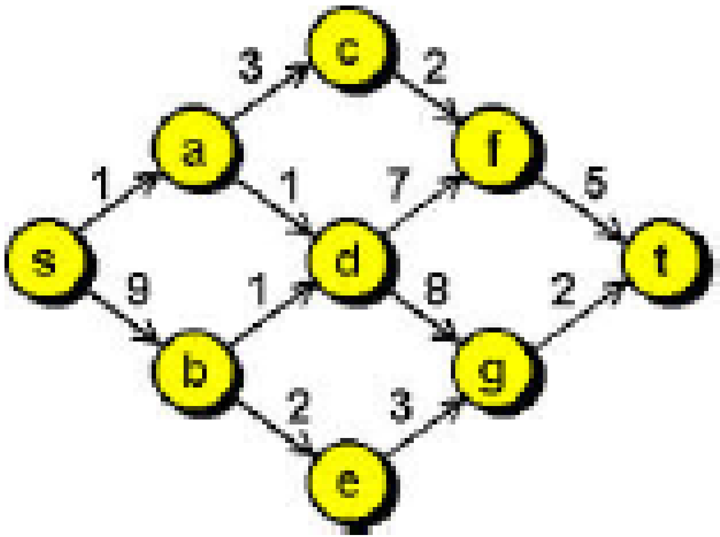
- Vertices might **not be reachable if the graph is disconnected.**

- To find shortest path from vertex A to vertex B.
- Case study : shortest path of cities in Malaysia
- Strategy:
  - Find all minimum distance from a given vertex to another vertices.
  - In order to find the minimum distance, we need to know the distance by using all possible paths to the respective immediate vertex.
  - How to do this? It is like traverse the graph, and calculate the distance.
- Algorithm: **Dijkstra's Algorithm**, *Bellman-Ford Algorithm (BFS)*


B
10
A
20
12
7
60
C
32
E
D
(c)

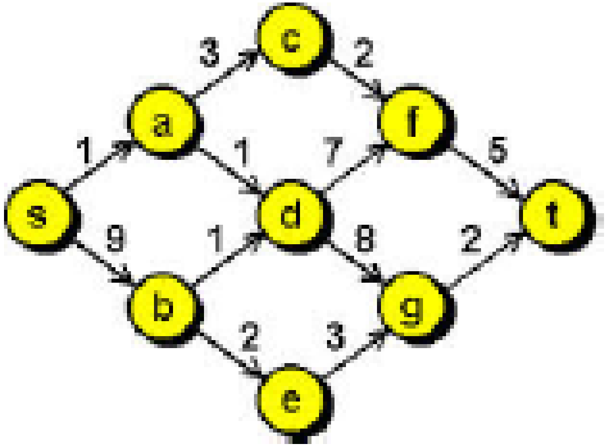

C:\Windows\system32\cmd.exe
Enter the number of vertices:5
Enter graph data in adjacency matrix form:
0 0 12 60 0
10 0 0 0 0
0 20 0 32 0
0 0 0 0 0
7 0 0 0 0
DIJKSTRA: SHORTEST PATH
Enter the source matrix:1
The source vertex: 1
Shortest path:
1->2,cost=32
1->3,cost=12
1->4,cost=44
1->5,cost=999
Press any key to continue . . .
EN
2:54 AM

Find shortest path from vertex **s** to **t**.



Find shortest path from vertex **s** to **t**.

1. **sa-ac-cf-ft = 11**
2. sa-ad-dg-gt=12
3. sa-ad-df-ft=14
4. sb-bd-dg-gt=20
5. sb-be-eg-gt=16



- A graph may have many spanning trees (ST).
- Only **connected and undirected** graph can produce a spanning tree.

- MST = **minimum connection of all vertices**.
- Case : UNIFI at FTSM

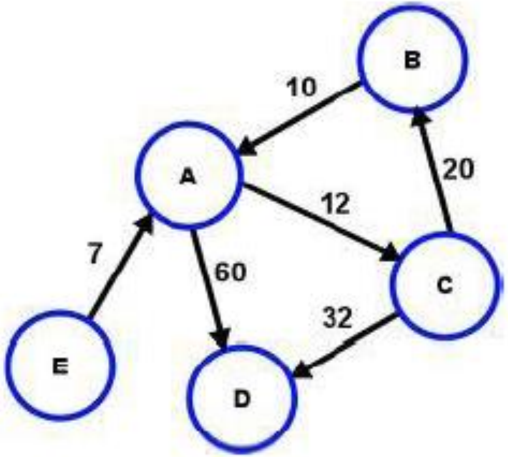
```
o---o
|\ /|
| X |
|/ \|
o---o
```

```
o---o
|\ /|
| X |
|/ \|
o---o
```

has sixteen spanning trees:

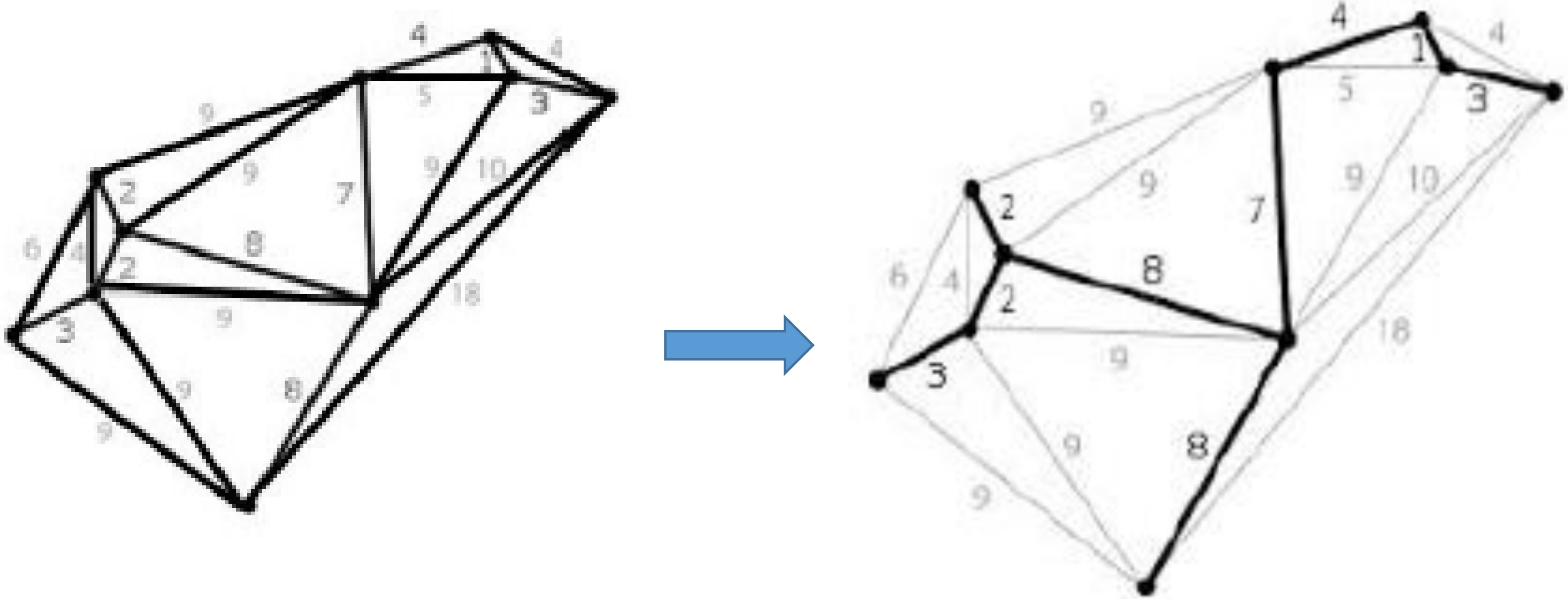
```
o---o   o---o   o   o   o---o
|   |   |       |   |       |
|   |   |       |   |       |
|   |   |       |   |       |
o   o   o---o   o---o   o---o

o---o   o   o   o   o   o   o
 \ /    |\ /     \ /     \ /|
  X     | X       X       X |
 / \    |/ \     / \     / \|
o   o   o   o   o---o   o   o

o   o   o---o   o   o   o---o
|\  |      /    |  /|    \
| \ |     /     | / |     \
|  \|    /      |/  |      \
o   o   o---o   o   o   o---o

o---o   o   o   o   o   o---o
|\      |  /     \  |      /|
| \     | /       \ |     / |
|  \    |/         \|    /  |
o   o   o---o   o---o   o   o
```

- Spanning tree can be :
  - graph without cycle. Or,
  - a tree without root, the nodes are either intermediate node or leaf.
- Algorithm : Prim, Kruskal, DFS+BFS



(c)

Find the minimum cost that connect all vertices



Graph API & Implementations

`public class Graph`

	`Graph(int V)`	*create an empty graph with V vertices*
	Scanner in `Graph(`~~`In in`~~`)`	*create a graph from input stream*
`void`	`addEdge(int v, int w)`	*add an edge v-w*
`Iterable<Integer>`	`adj(int v)`	*vertices adjacent to v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`String`	`toString()`	*string representation*

Reference: http://algs4.cs.princeton.edu/41graph

Adjacency-list graph representation: Java implementation

```
public class Graph
{
   private final int V;
   private Bag<Integer>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

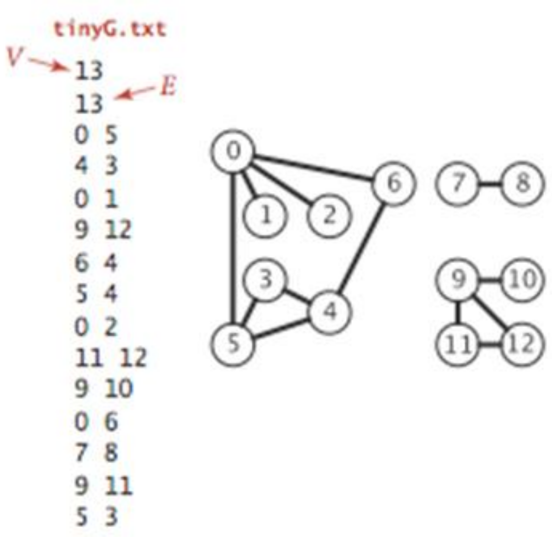
adjacency lists
(using Bag data type)

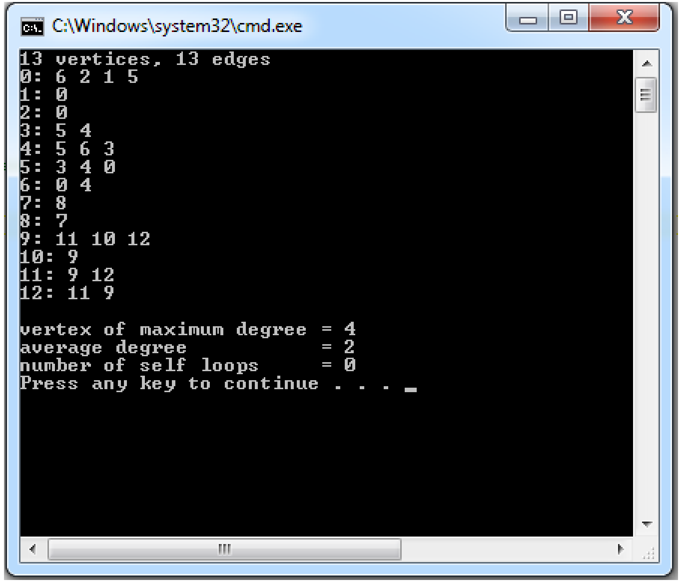
create empty graph
with V vertices

add edge v–w
(parallel edges and
self-loops allowed)

iterator for vertices adjacent to v

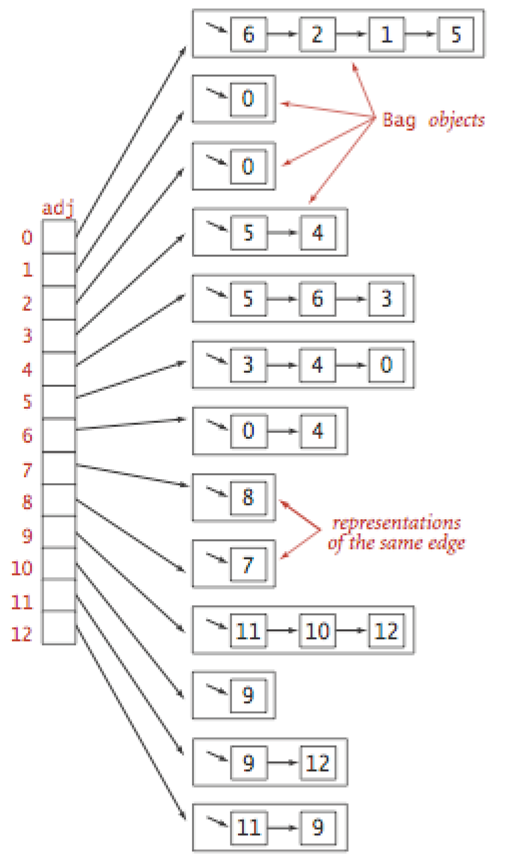
Data file: tinyG.txt



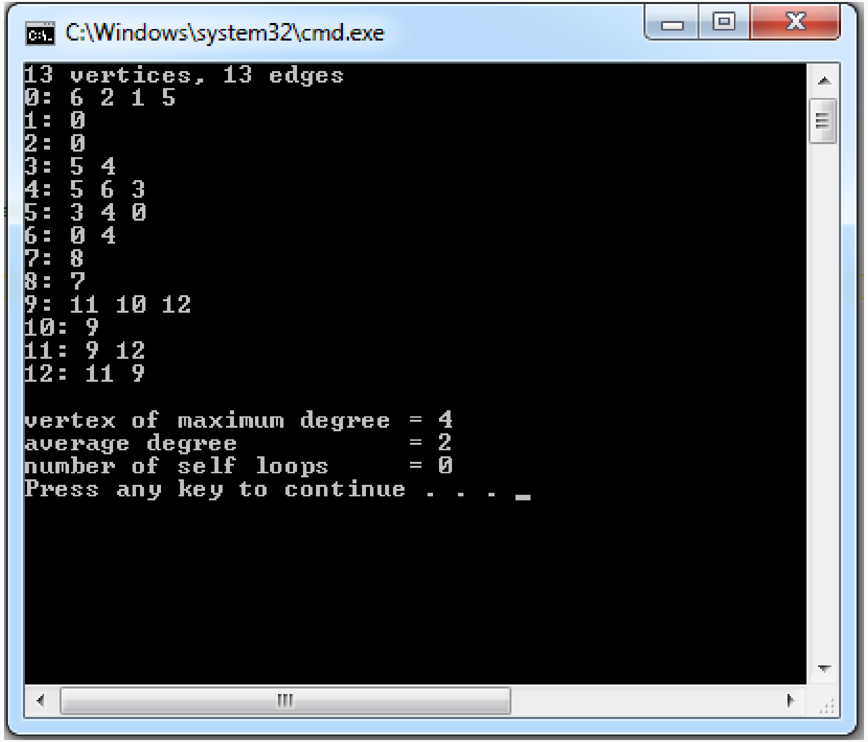


In practice use *adjacency-lists representation*
- Algorithm based on iterating over vertices adjacent to *v*
- Real world graphs tend to be sparse

adjacent-list representation



Adjacency-lists representation (undirected graph)



- Goal : systematically search through a graph
- Typical Applications
  - find all vertices connected to a given source vertex
  - Find a path between two vertices

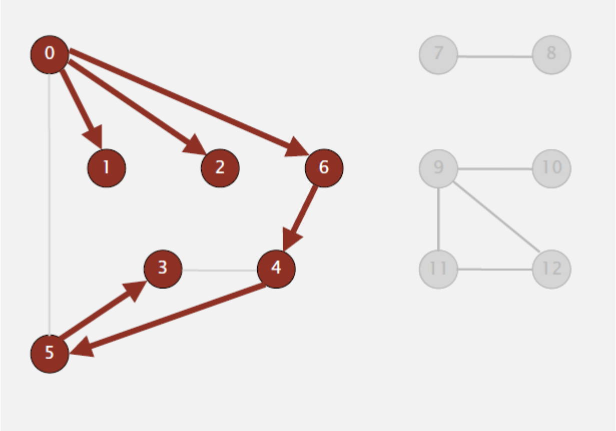
DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked
    vertices w adjacent to v.

- Decouple graph data from graph processing
  - Create Graph object
  - Pass the Graph to a graph-processing methods
  - Query the graph-processing methods for information

	public class Paths	
	Paths(Graph G, int s)	*find paths in G from source s*
boolean	hasPathTo(int v)	*is there a path from s to v?*
Iterable<Integer>	pathTo(int v)	*path from s to v; null if no such path*



vertices reachable from 0

```
private void dfs(Graph G, int v)
{
   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w])
      {
         dfs(G, w);
         edgeTo[w] = v;
      }
}
```

C:\Windows\system32\cmd.exe

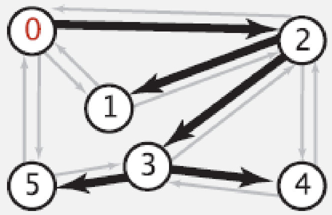
```
Starting node: 0
0 1 2 3 4 5 6
NOT connected
Press any key to continue . . .
```

C:\Windows\system32\cmd.exe

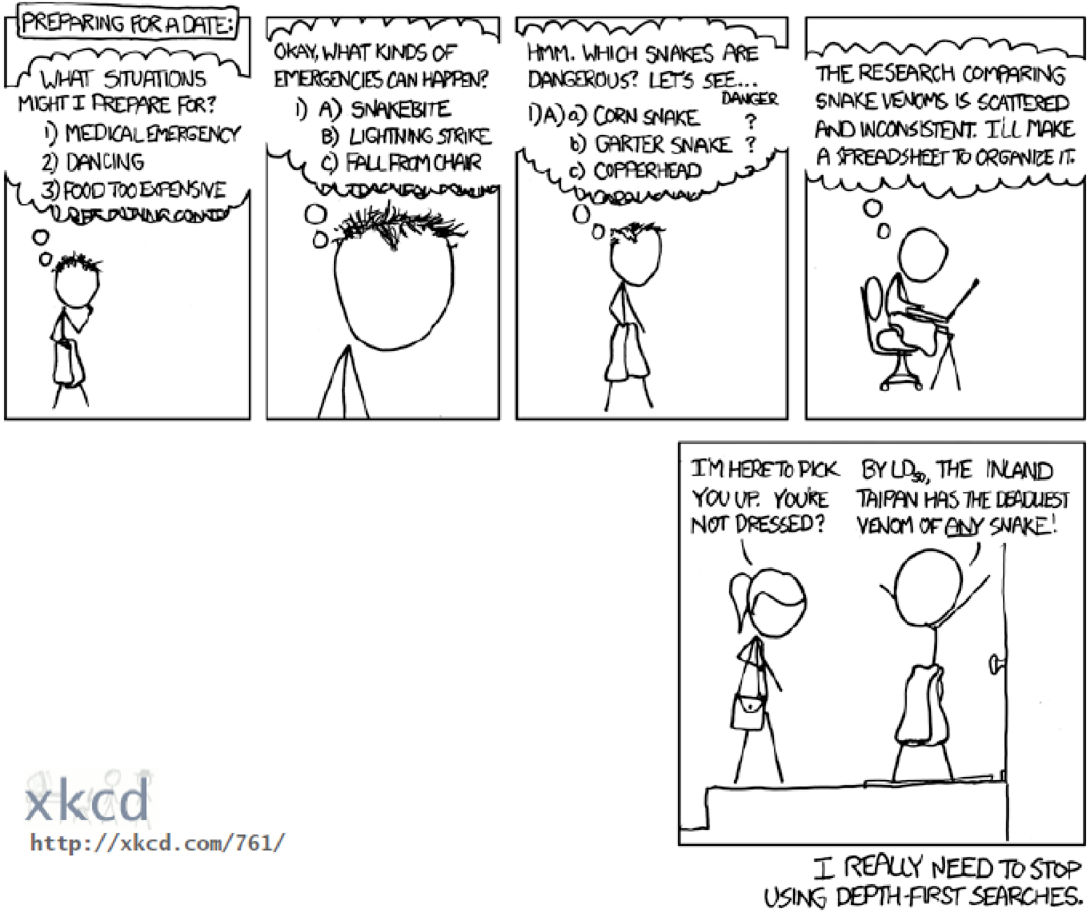
```
Starting node: 9
9 10 11 12
NOT connected
Press any key to continue . . .
```

- After DFS, we can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportion to its length

```
public boolean hasPathTo(int v)
{  return marked[v];  }

public Iterable<Integer> pathTo(int v)
{
   if (!hasPathTo(v)) return null;
   Stack<Integer> path = new Stack<Integer>();
   for (int x = v; x != s; x = edgeTo[x])
      path.push(x);
   path.push(s);
   return path;
}
```



edgeTo[]	
0	
1	2
2	0
3	2
4	3
5	3



http://xkcd.com/761/

Graph traversal : breadth-first search

- Repeat until queue is empty:
  - Remove vertex *v* from queue.
  - Add to queue all unmarked vertices adjacent to *v* and mark them.

BFS (from source vertex s)

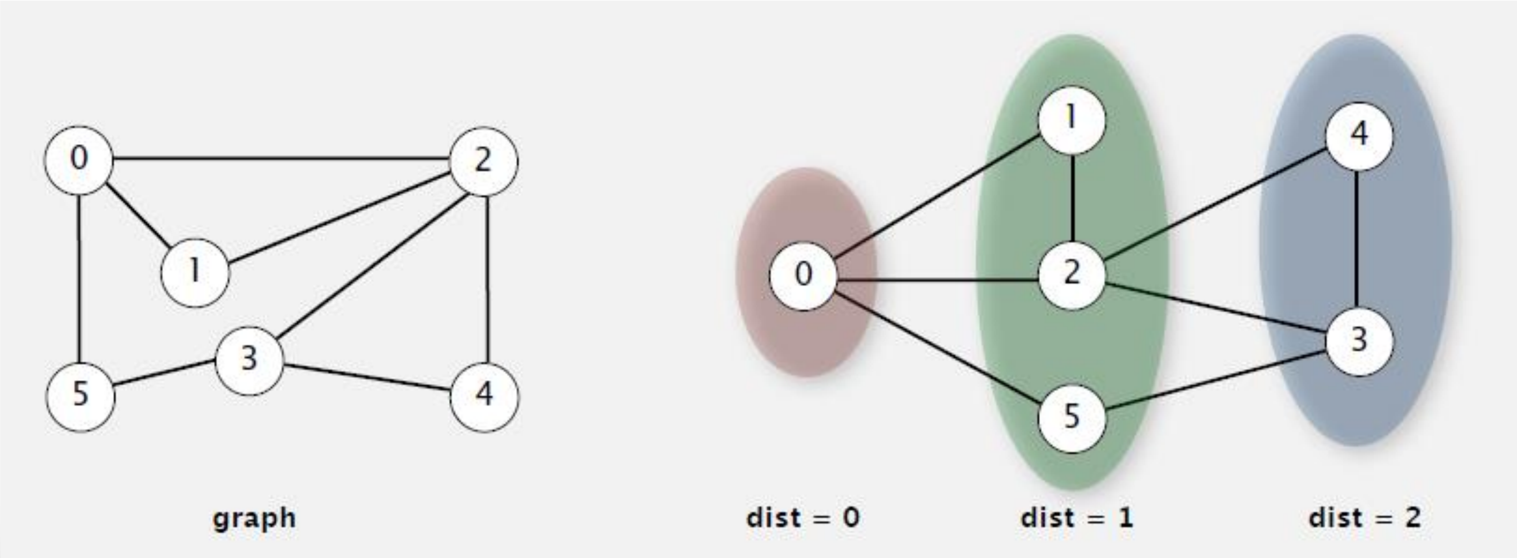
Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- remove the least recently added vertex v
- add each of v's unvisited neighbors to the queue, and mark them as visited.

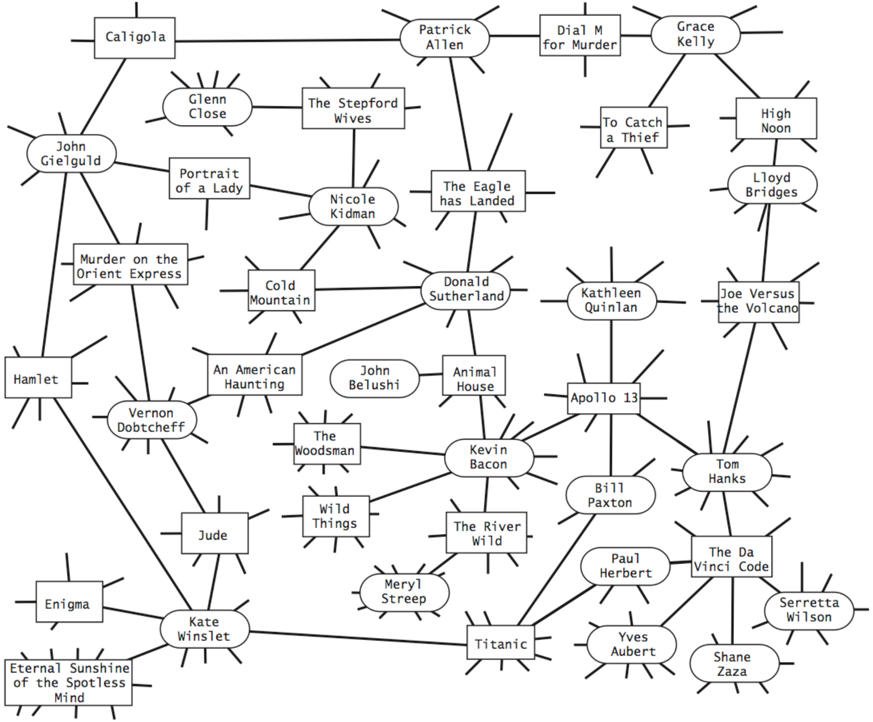
Breadth-first search properties

- BFS computes shortest paths (fewest number of edges) from *s* to all other vertices in a graph in time proportional to *E* + *V*.



```
    private void bfs(Graph G, int s)
    {
      Queue<Integer> q = new Queue<Integer>();
       q.enqueue(s);
       marked[s] = true;
       while (!q.isEmpty())
       {
          int v = q.dequeue();
          for (int w : G.adj(v))
          {
             if (!marked[w])
             {
                q.enqueue(w);
                marked[w] = true;
                edgeTo[w] = v;
             }
          }
       }
    }
}
```

- Small world phenomenon : Six handshakes away from anyone.
- Find shortest path in performer-movie graph.



Type of Problem	Objective	Algorithm
Graph Traversal	To visit all vertices.	Depth First Search (DFS) Breadth First Search (BFS)
Shortest Path	To find shortest path from vertex A to vertex B. Focus on 2 vertex only: start-vertex, stop-vertex.	Dijsktra's *Bellman-Ford's*
Minimum Spanning Tree (MSP)	To visit all vertices in Spanning Tree with minimum cost. Focus on all vertices.	DFS, BFS Prim's, *Kruskal's,*

Additional slides

public class Digraph		
	Digraph(int V)	*create an empty digraph with V vertices*
	Digraph(In in)	*create a digraph from input stream*
void	addEdge(int v, int w)	*add a directed edge* $v \to w$
Iterable<Integer>	adj(int v)	*vertices pointing from v*
int	V()	*number of vertices*
int	E()	*number of edges*
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

Shortest Path : Dijkstra

- Set distance to *startNode* to zero.
- Set all other distances to an infinite value.
- We add the *startNode* to the unsettled nodes set.
- While the unsettled nodes set is not empty we:
  - Choose an evaluation node from the unsettled nodes set, the evaluation node should be the one with the lowest distance from the source.
  - Calculate new distances to direct neighbors by keeping the lowest distance at each evaluation.
  - Add neighbors that are not yet settled to the unsettled nodes set.

http://algs4.cs.princeton.edu/44sp/

```
public DijkstraSP(EdgeWeightedDigraph G, int s) {
  for (DirectedEdge e : G.edges()) {
    if (e.weight() < 0)
      throw new IllegalArgumentException("edge " + e +
        " has negative weight");
      }
      distTo = new double[G.V()];
      edgeTo = new DirectedEdge[G.V()];
      validateVertex(s);
      for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;
      // relax vertices in order of distance from s
      pq = new IndexMinPQ<Double>(G.V());
      pq.insert(s, distTo[s]);
      while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (DirectedEdge e : G.adj(v))
          relax(e);
      }
       // check optimality conditions
      assert check(G, s);
    }
```