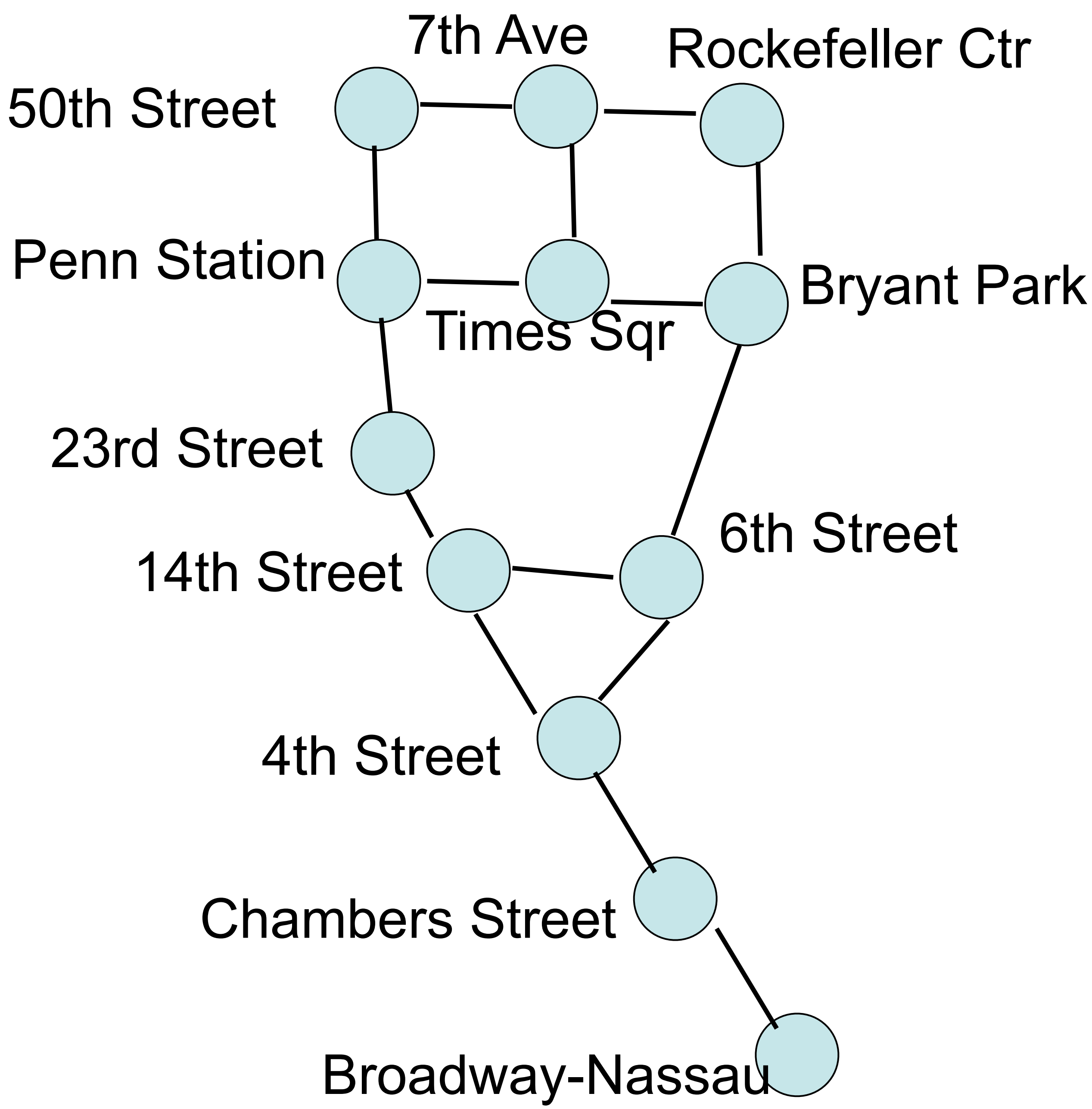
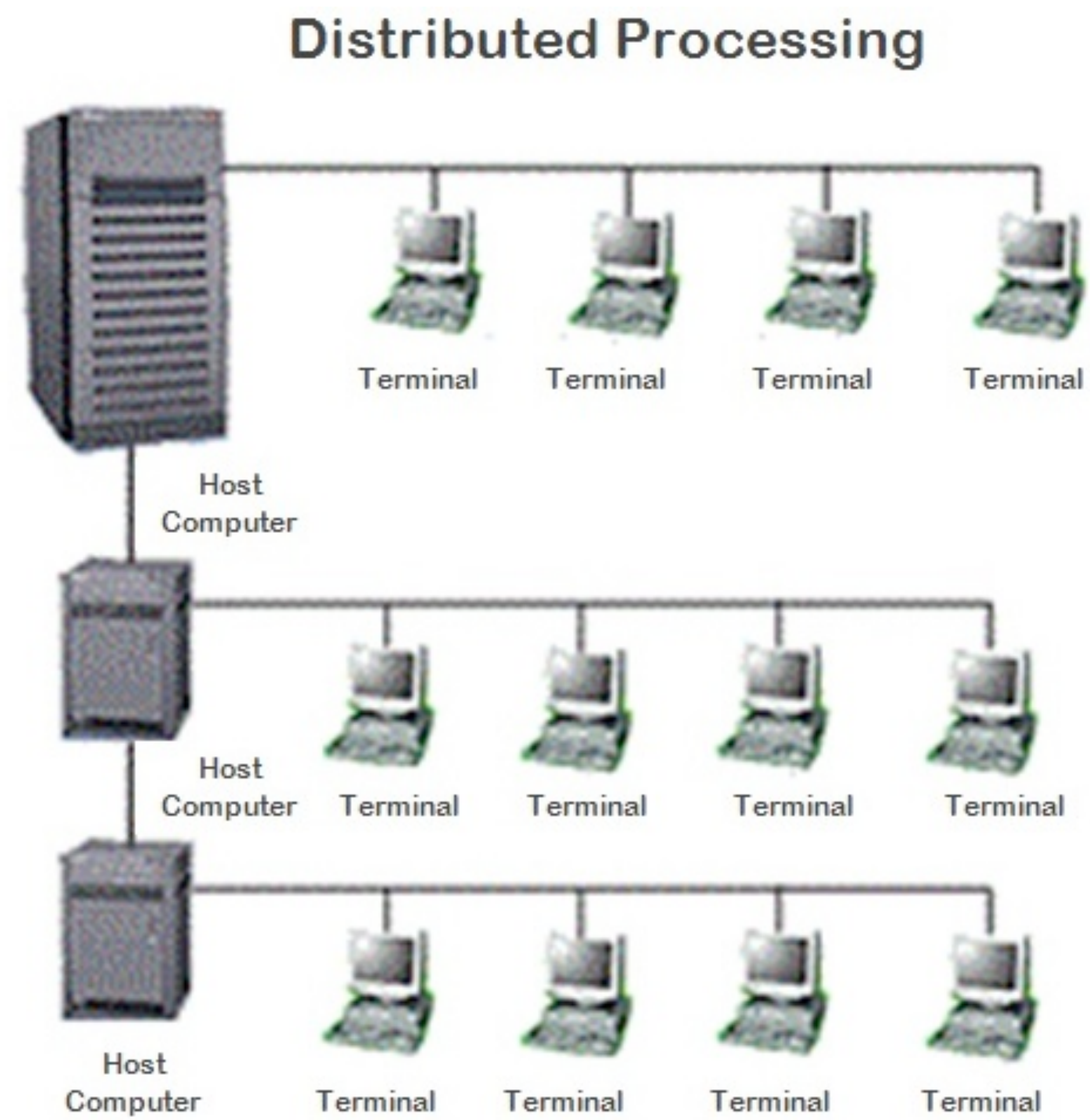
Lecture 17: Graphs and Unweighted Shortest Paths

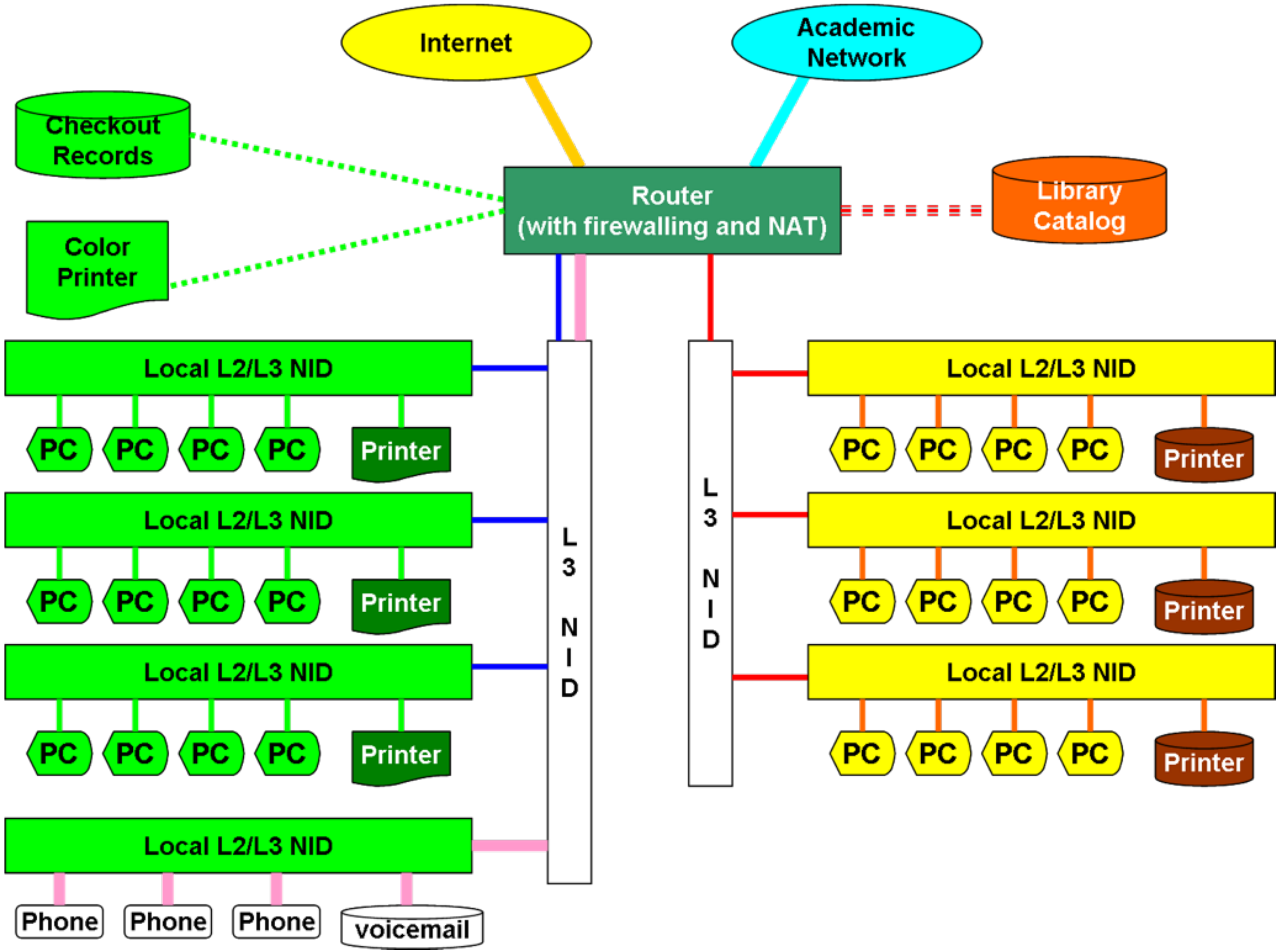
Graphs show the relationship between objects/events

routes for mass transit





http://en.wikipedia.org/wiki/File:Distributed_Processing.jpg

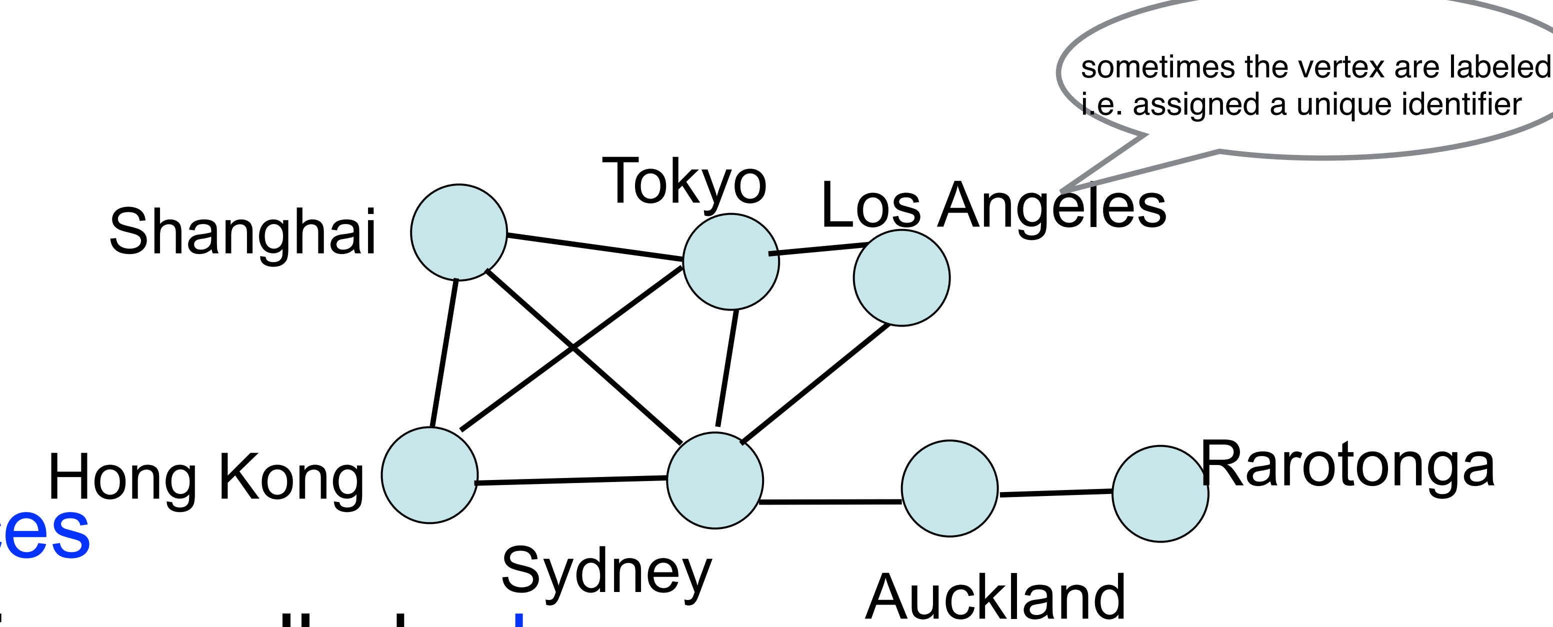


http://en.wikipedia.org/wiki/File:NETWORK-Library-LAN.png

Graphs

- Graphs in computer science are not like the graphs you draw in physics lab.
- Mathematically, a graph $G = (V,E)$ consists of
  - a set V of vertices
  - a set E of edges
- An edge of the graph is a pair (v,w), where v and w are vertices in V.
- Vertices are also often called nodes.
- It's often convenient to draw a picture of a graph.

Graph



- G = (V,E)
  - V is a set of nodes/vertices
  - E is a set of pairs of vertices, called edges
- Example:
  - A vertex represent an airport

    V = {Shanghai, Tokyo, Los Angeles, Hong Kong, Sydney, Auckland, Rarotonga}
  - An edge represents a flight route between two airports

    E = {(Shanghai, Tokyo), (Shanghai, Hong Kong), (Shanghai, Sydney),
    (Tokyo, Hong Kong), (Tokyo, Sydney), (Tokyo, Los Angeles),
    (Hong Kong, Sydney), (Sydney, Auckland), (AuckLand, Rarotonga),
    (Los Angeles, Sydney)}

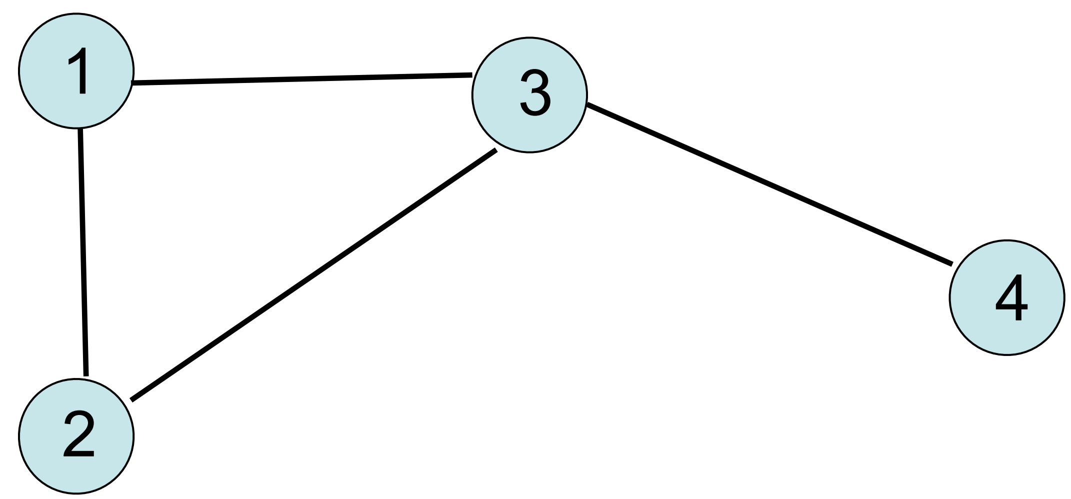
    |V| = 7

    |E| = 10

    A path: Los Angeles,Tokyo, Shanghai, Sydney, Auckland

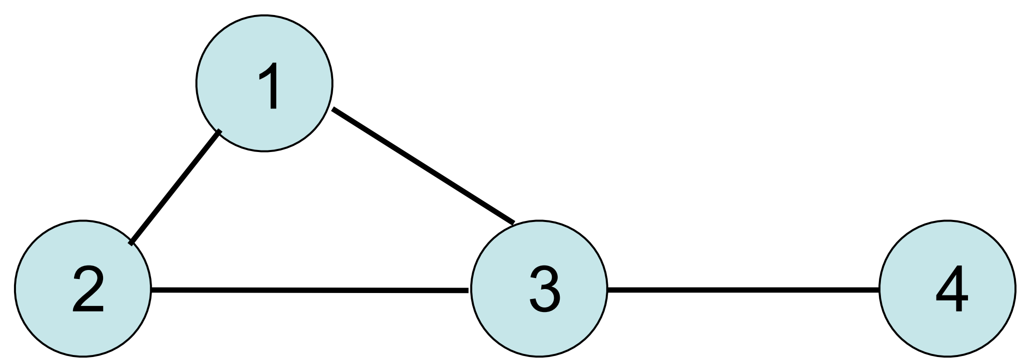
    This path has length 4. Unweighted Path Length

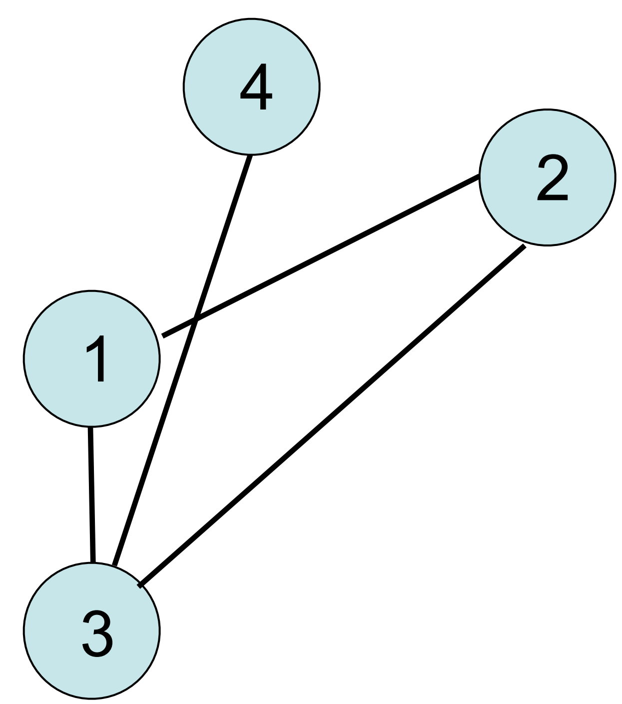
Graph Example



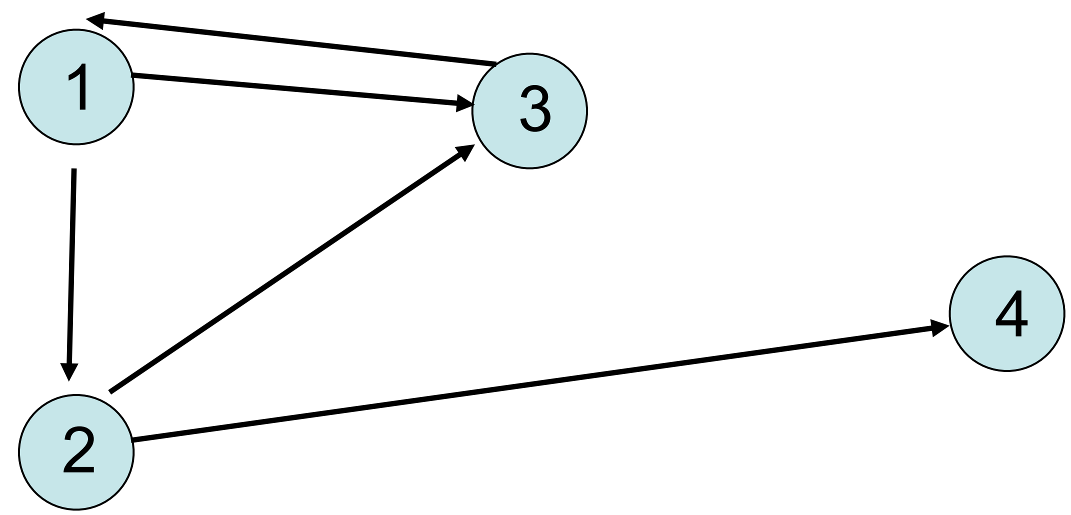
- Undirected graph with 4 vertices and 4 edges
- (1,3) and (3,1) designate the same edge

The same graph can be drawn many different ways

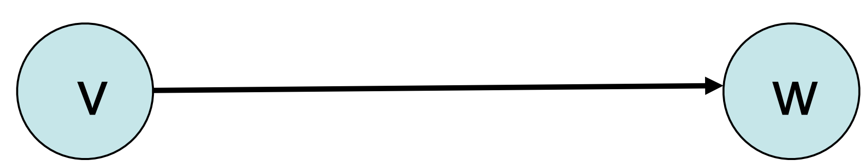




Directed Graph Example



Each edge has a direction, shown by an arrow

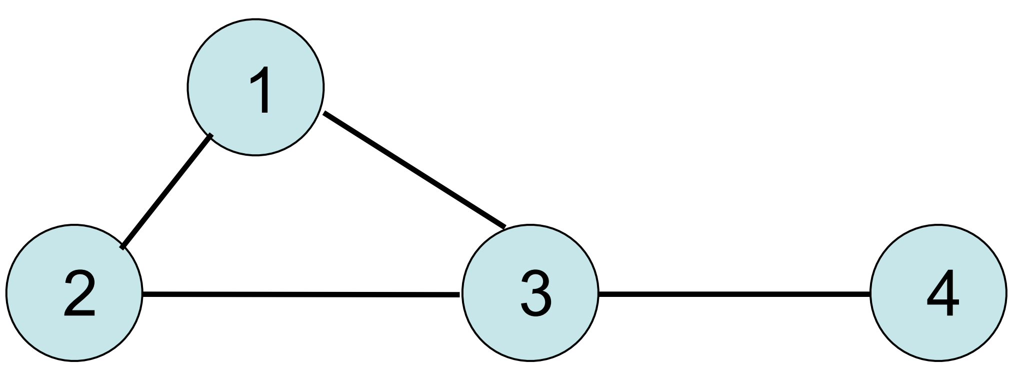


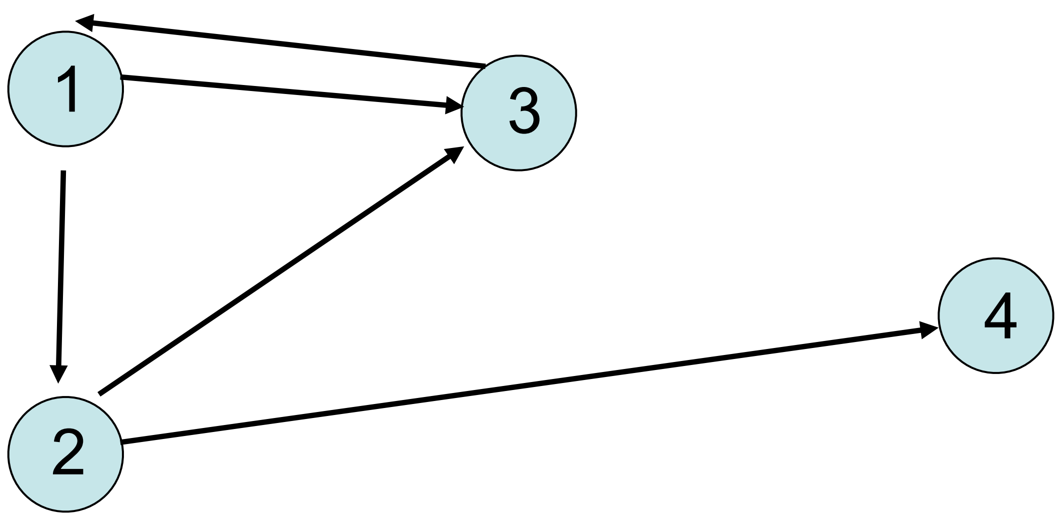
Edge (v,w) goes FROM vertex v, TO vertex w

There does NOT exist an edge (w,v)

Neighbor of/Adjacent to

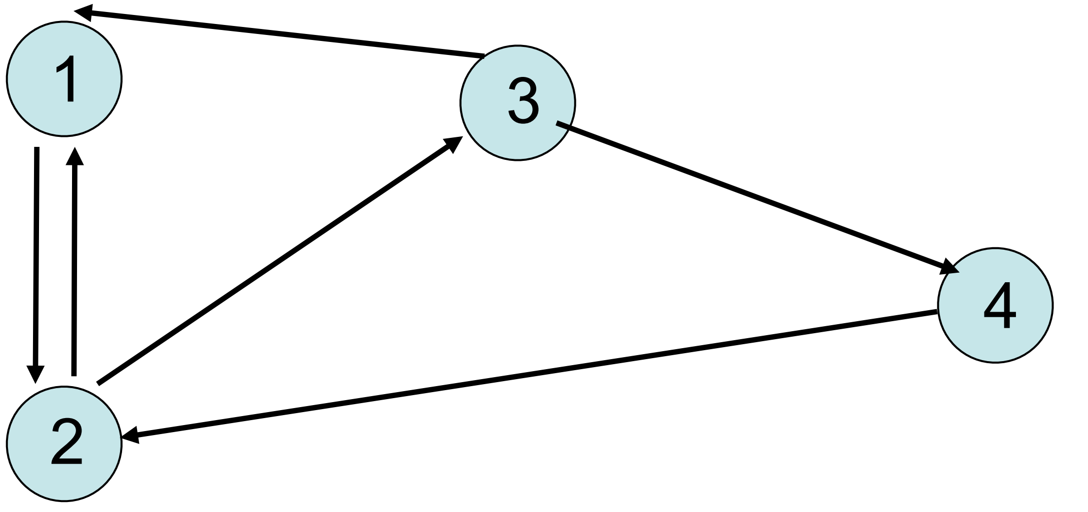
- In an undirected graph, w is said to be a neighbor of v if there is an edge joining v and w. If v is a neighbor of w, then w is a neighbor of v
- In a directed graph, w is said to be a neighbor of v if there is a (directed) edge FROM v TO w
- "is a neighbor of" same as "is adjacent to"



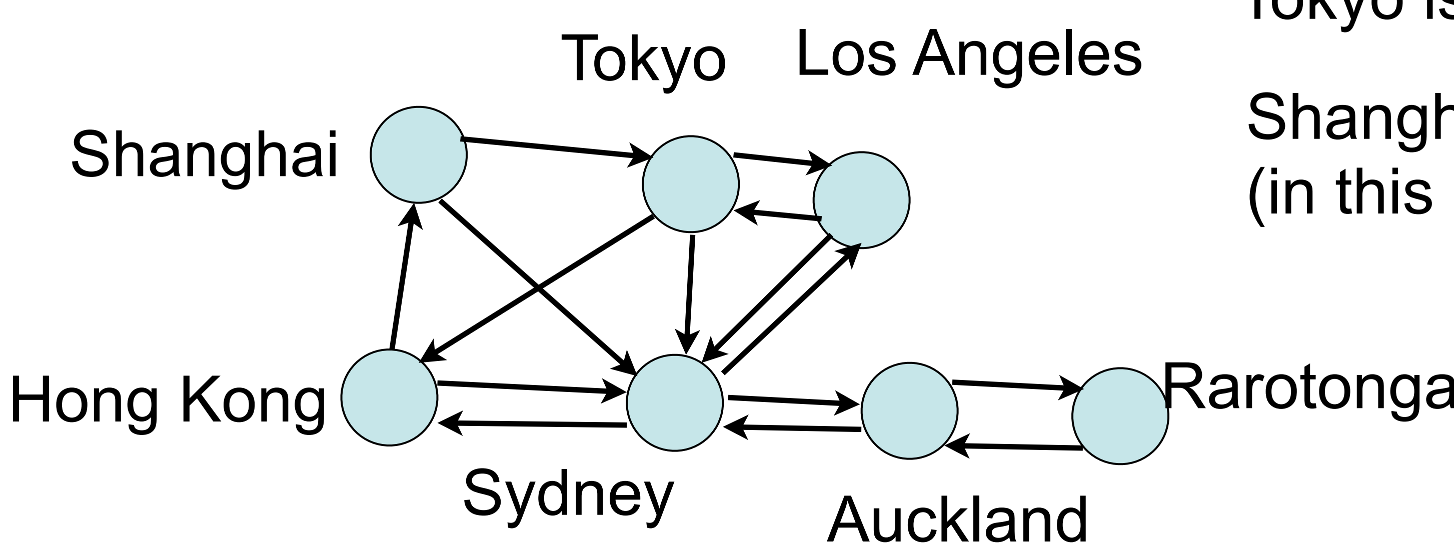


Paths

- A path is a sequence of vertices, connected by edges.
- The length of the path is the number of edges.
- Example: path 1,2,3 has length 2
- Note that there may be zero, one, or more than one paths connecting two vertices.
- We often want to find the shortest path connecting two vertices.



Directed Graph:



Tokyo is adjacent to Shanghai

Shanghai is NOT adjacent to Tokyo
(in this graph)

G = (V, E),

V = {Shanghai, Tokyo, Los Angeles, Hong Kong, Sydney, Auckland, Rarotonga}

E = {(Shanghai, Tokyo), (Shanghai, Sydney), (Los Angeles, Tokyo)
(Tokyo, Hong Kong), (Tokyo, Sydney), (Tokyo, Los Angeles),
(Hong Kong, Sydney), (Sydney, Auckland), (Sydney, Hong Kong),
(Sydney, Los Angeles),(Hong Kong, Shanghai),(Auckland, Sydney),
(Auckland, Rarotonga), (Rarotonga, Auckland), (Los Angeles, Sydney)}

|V| = 7 |E| = 15

There does NOT exist a path: Los Angeles,Tokyo, Shanghai, Sydney, Auckland

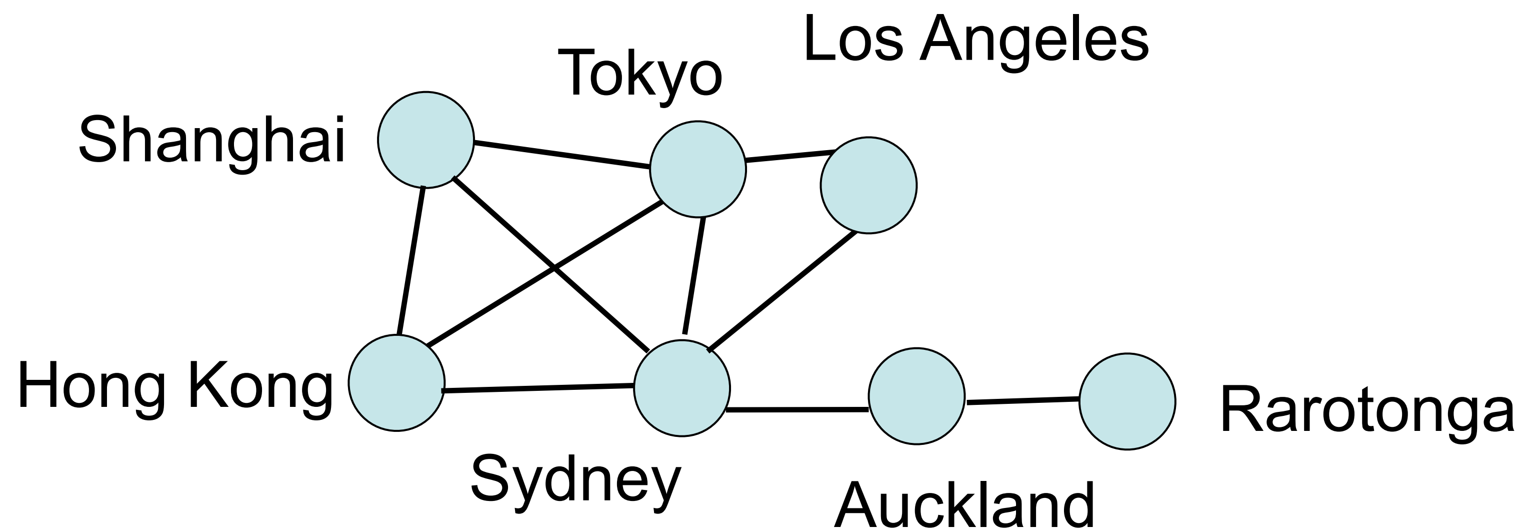
There is a path: Los Angeles,Tokyo, Sydney, Auckland. This path has length 3.

Many applications

- Communications:
  - Nodes : computers
  - Edges: direct connections between computers
- Transportation:
  - Nodes: cities
  - Edges: roads, air routes, train tracks, etc.
- Social
  - Nodes: people
  - Edges: friendships, common interests, etc.

How would we store the graph ADT?

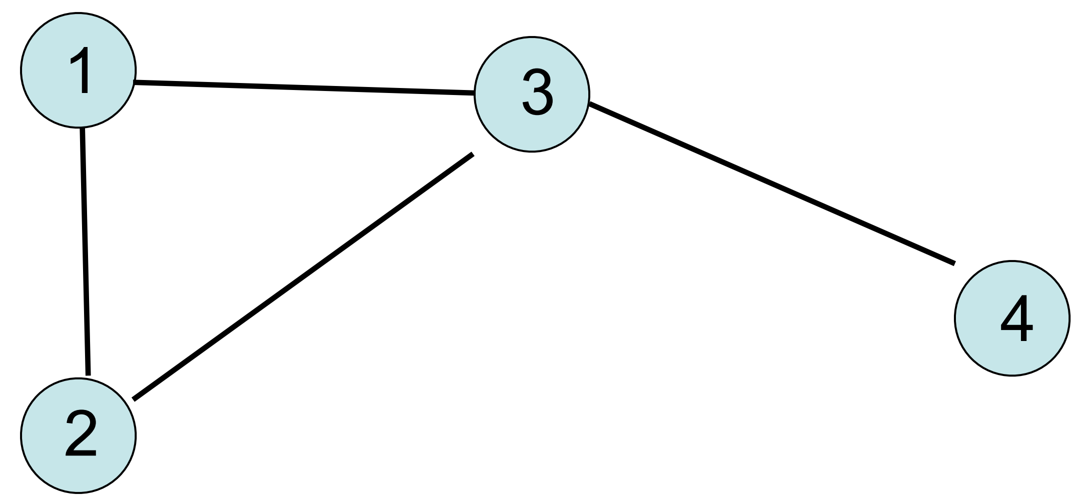
Implementation?



Adjacency Matrix:

	Shanghai	Hong Kong	Sydney	Tokyo	Auckland	Los Angeles	Rarotonga
Shanghai		X	X	X			
Hong Kong	X		X	X			
Sydney	X	X		X	X	X	
Tokyo	X	X	X			X	
Auckland			X				X
Los Angeles			X	X			
Rarotonga					X		

Adjacency List Representation



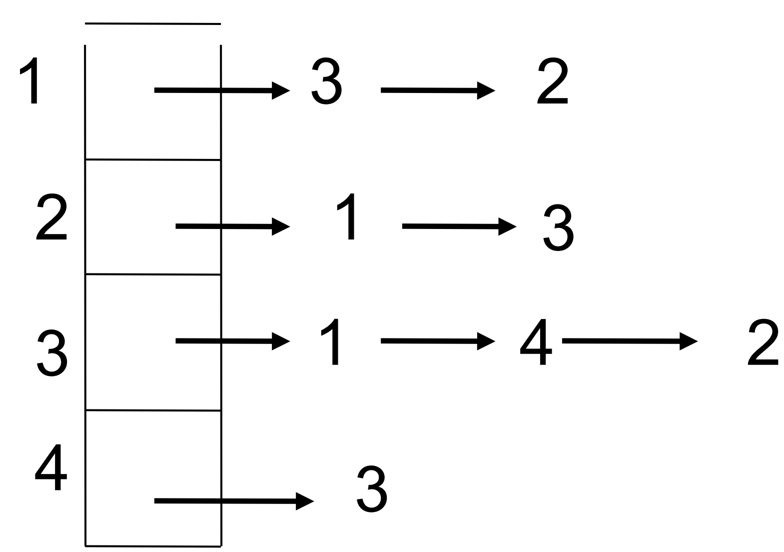
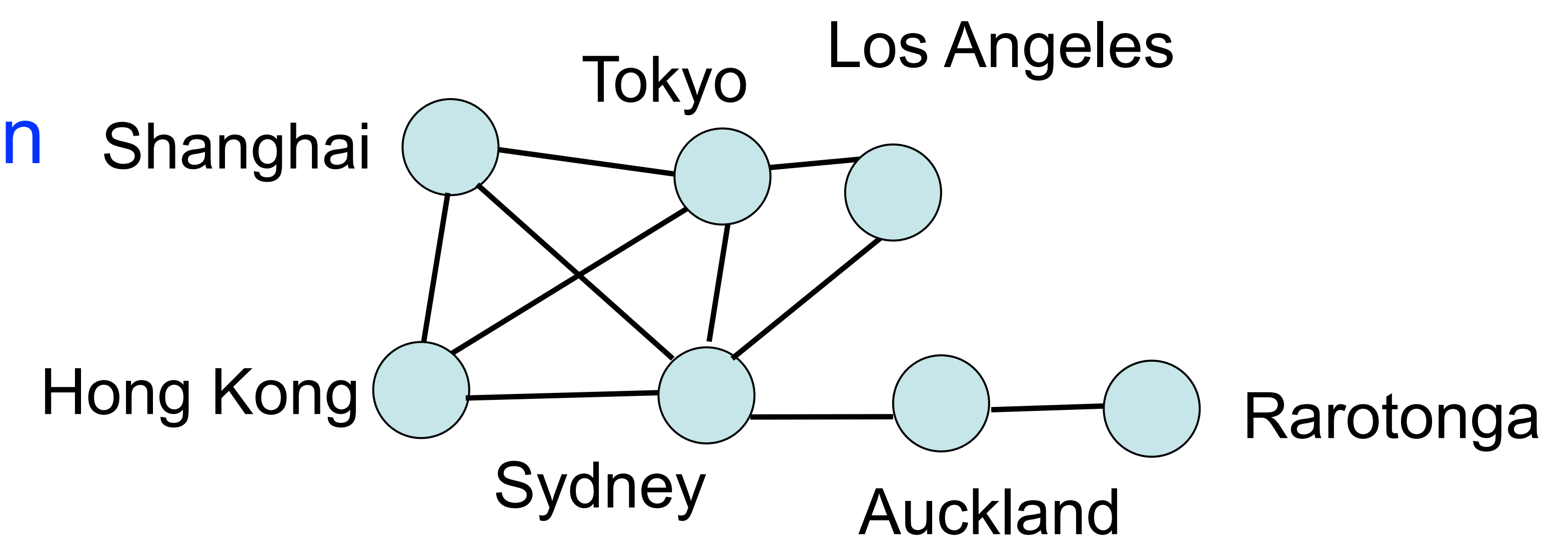


Table in position j contains list of vertices adjacent to vertex j

Adjacency List Implementation



G = (V, E),

Adjacency List:

Shanghai → Tokyo → Hong Kong → Sydney

Tokyo → Hong Kong → Sydney → Los Angeles → Shanghai

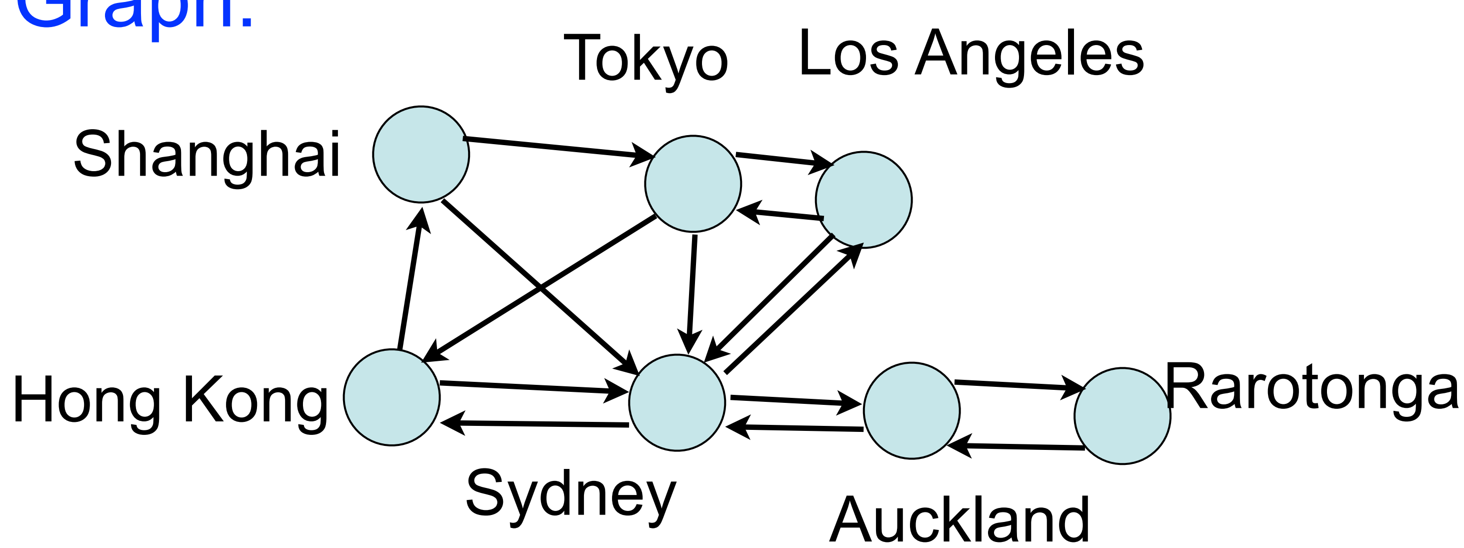
Hong Kong → Sydney → Tokyo → Shanghai

...

O(|E|) space (assuming that every vertex is in some edge - or the number of edges is at least O(|V|)). Linear in the size of the graph

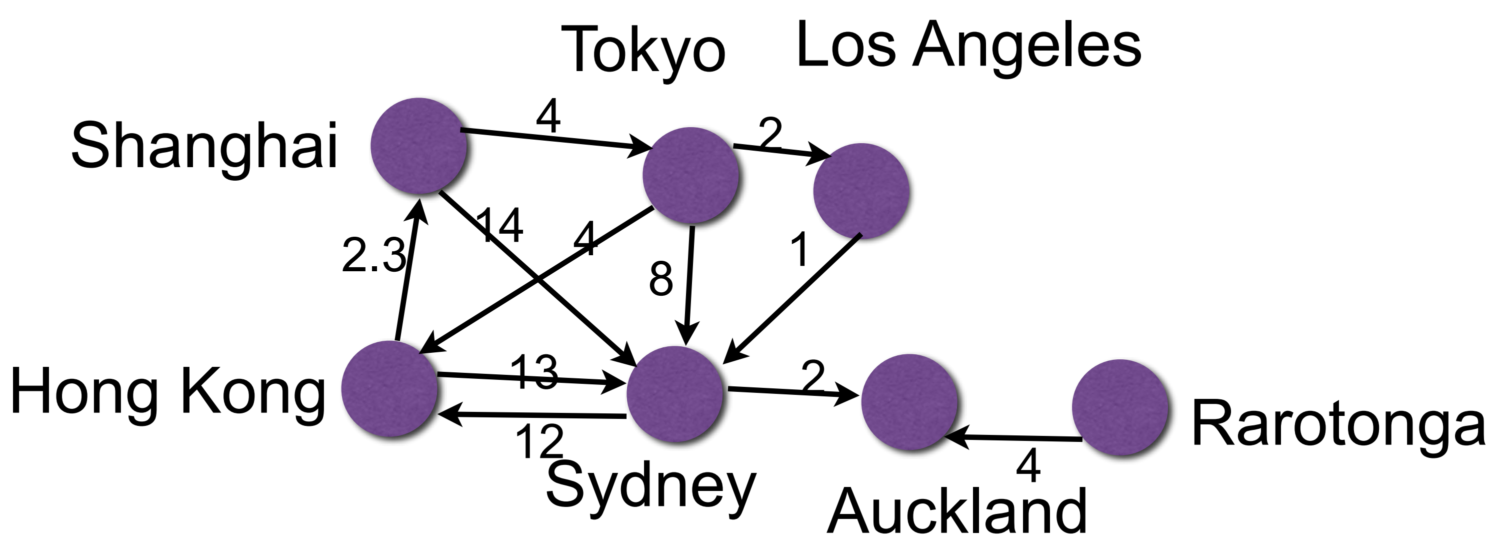
O(|E|) time to construct from a list of the edges.

Directed Graph:



Adjacency List:

Directed Graph:



Adjacency List:

Shanghai → Tokyo (4) → Sydney (14)

Tokyo → Hong Kong (4) → Sydney (10) → Los Angeles (10)

Hong Kong → Sydney (13) → Shanghai (2.3)

...

Which to Choose?

- space

  Adjacency List: $O(V + E)$

  Adjacency Matrix: $O(V^2)$

- computational efficiency

  Adjacency List: Easy to determine all nodes adjacent to a vertex.
  Easy to add an adjacent edge.

  Adjacency Matrix: Easy to determine if an edge is in the graph $O(1)$ time
  When not storing weights, only one bit needed per entry

- flexibility

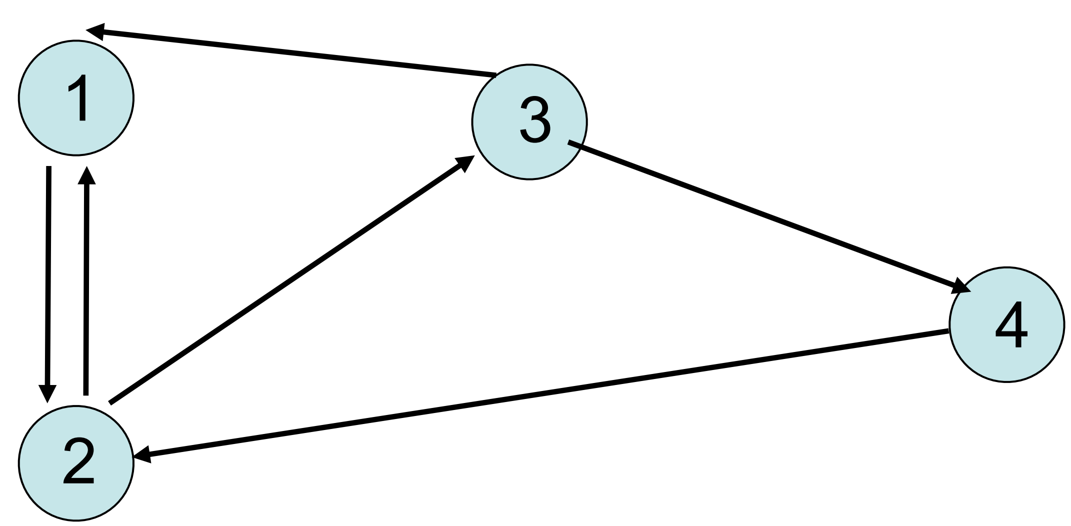
  Adjacency List: Cumbersome to vary the edges incident with a vertex

  Adjacency Matrix: Easy to vary the edges adjacent to a vertex

Unweighted Shortest Path Problem

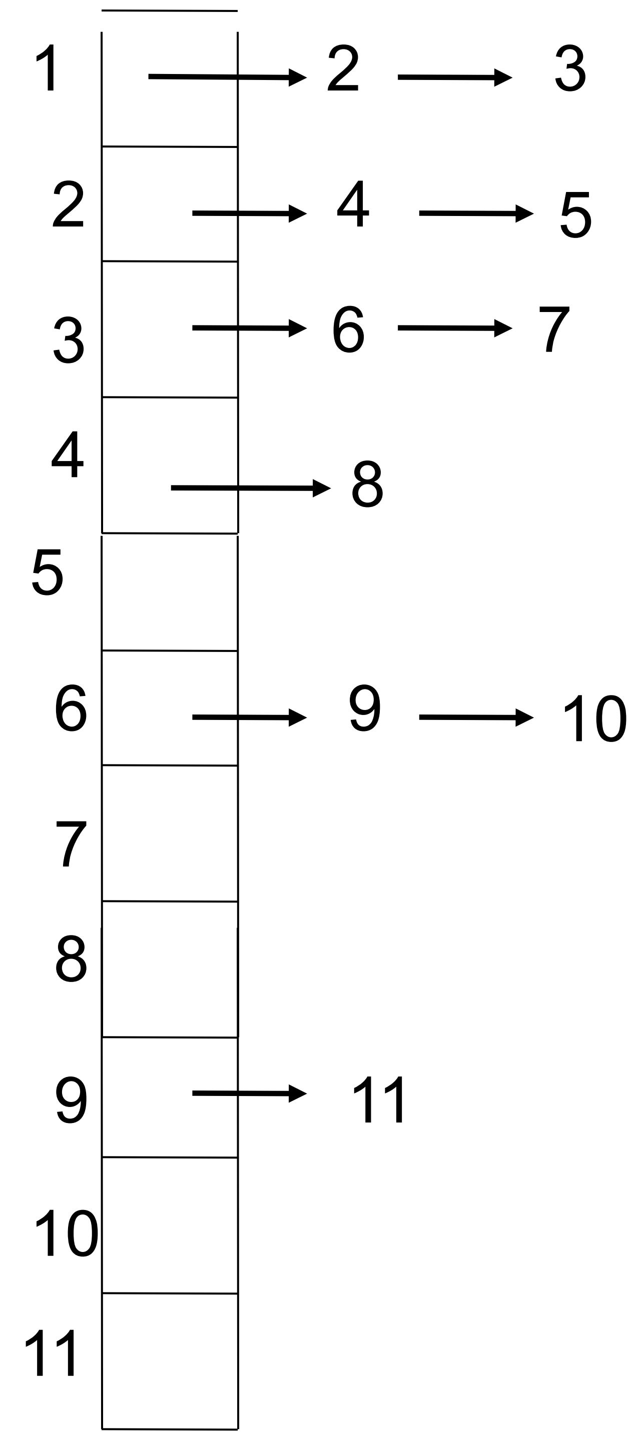
- Problem: Given graph G and source vertex s, find length of shortest path from s to every vertex in G.
  - Length of path = number of edges on the path
  - Distance of vertex v from s = length of shortest path from s to v

Enumerating paths and comparing their lengths would be extremely expensive: can it be done efficiently?

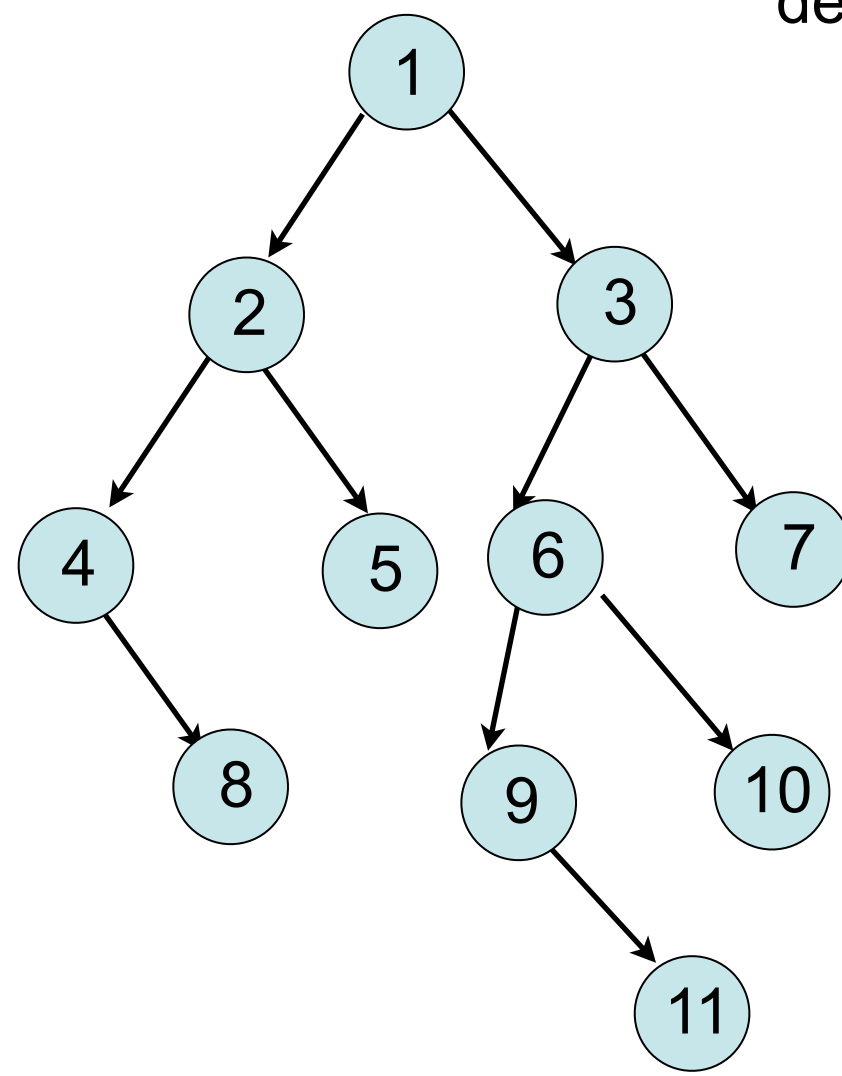


Finding the depth of all the nodes

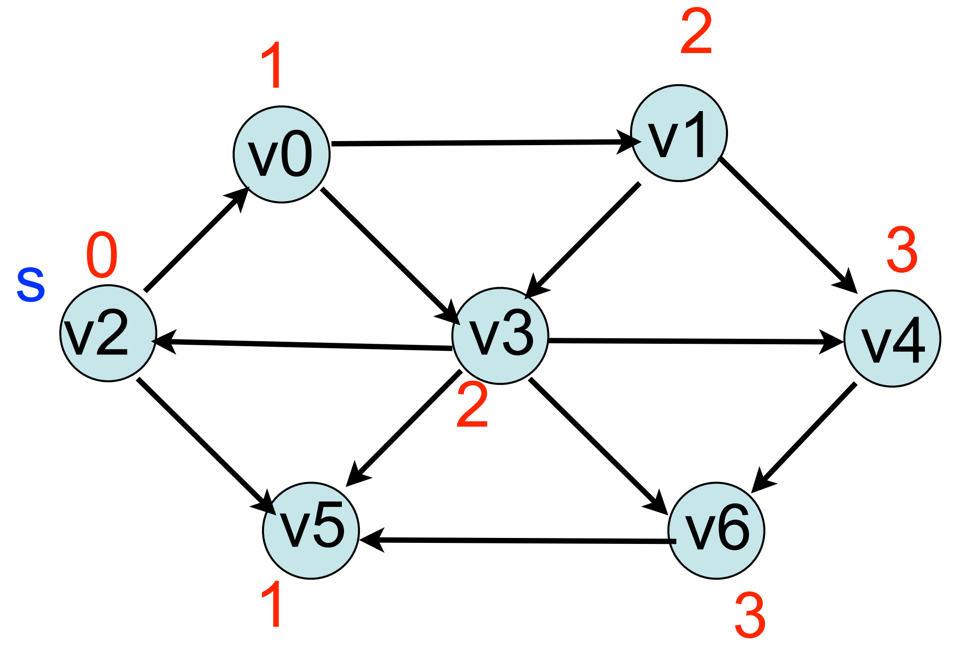
Adjacency List



Use level order search to find the depth of each node



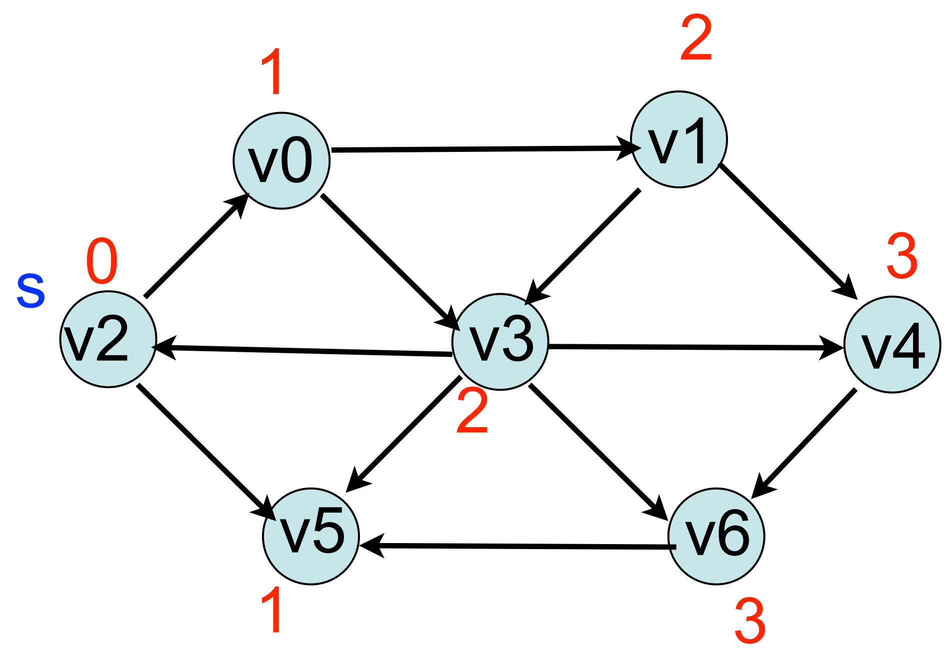
Unweighted Shortest-path



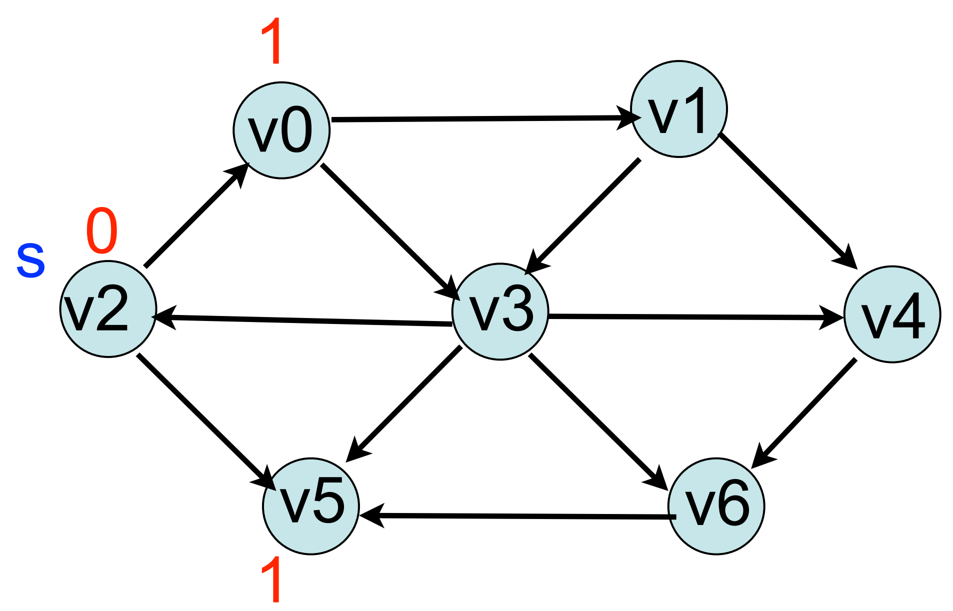
Breadth First Search (BFS)

- Idea:
  - Find all vertices at distance:
    - 0 from s
    - then all vertices at distance 1
    - then all vertices at distance 2
    - etc
  - Once you’ve found all vertices at distance k, then the vertices at distance k+1 are those that haven’t been found yet, and that are adjacent to vertices at distance k.
  - Can implement using a queue.

Unweighted Shortest-path



Unweighted Shortest-path



Node Visiting: v2

Queue:

v0	v5

Pseudocode for Unweighted Shortest Paths

Input: Graph G, source vertex s

```
Let d[v] denote distance from s to v
For all vertices v in G, initialize d[v] to sentinel value (-1)
d[s] = 0
Insert s into initially empty queue Q
```

preprocessing

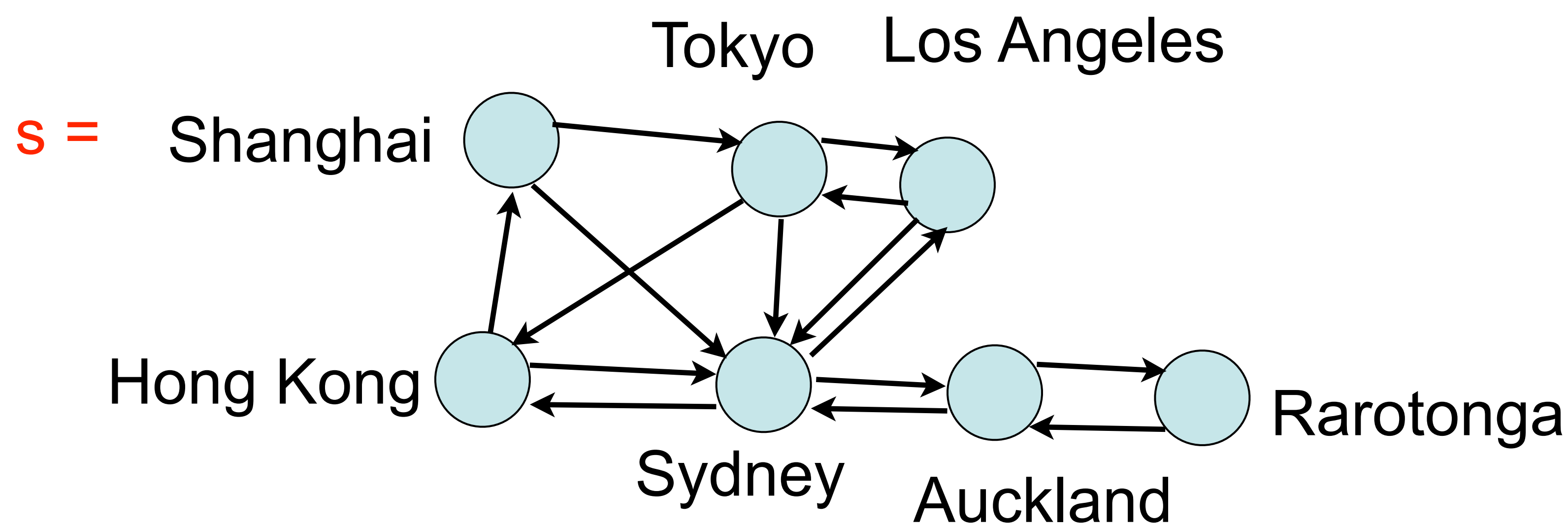
```
While Q is not empty {
   Delete a vertex v from front of Q (dequeue)
   For each neighbor w of v  {
        If d[w] == sentinel {
             d[w] = d[v] + 1
             insert w at the back of Q (enqueue)
        }
}
```

breadth first search

Modification of the Pseudocode for Unweighted Shortest Paths

- Often want to know the shortest path, not just its length
  - Can modify shortest path code to also store predecessor on the path
  - Update when w is “discovered” as a neighbor of v.
  - At end, use these to work backwards from each target to get shortest path from source to target.
- A simple implementation is given later in the slides

Unweighted Shortest Path From Shanghai:



phase	distance/predecessor							visiting	queue
	Sh	T	L	H	Sy	A	R		(front at queue on the left)
init	0/-								Sh
1	0/-							Sh	

Analysis of Breadth-First Search

Therefore BFS is O(V + E) time - linear in the size of the adjacency-list representation of G

- It takes O(V) time for queue operations
  - Every vertex/node put in queue exactly one time
  - Every vertex/node removed from the queue exactly one time
  - O(1) time to push/pop/get value from a queue
- O(V) time for initialization
- The adjacency list is scanned exactly one time - thus O(E) time in total scanning adjacency list

// Data structures

```
typedef vector<list<int> > Graph;
 // The graph is given in an adjacency list.
 // Vertices are indexed from 0 to V-1
// The indices of the vertices adjacent to vertex i
// are in the list Graph[i].
// Graph can be directed or undirected.
```

```
struct vertexInf // Stores information for a vertex
{
    int dist;  // distance to vertex from the source
    int prev;  // previous node in BFS tree
};
```

```
//                  1
//                  *
//          0     2 / \
//          *-----*---*3
//                |   |
//                |   |
//              5*---*4
//
//
void main()
{
     Graph g(6);

     g[0].push_back(2);
     g[1].push_back(3);
     g[1].push_back(2);
     g[2].push_back(3);
     g[2].push_back(5);
     g[2].push_back(1);
     g[2].push_back(0);
     g[3].push_back(1);
     g[3].push_back(2);
     g[3].push_back(4);
     g[5].push_back(4);
     g[4].push_back(5);
     g[4].push_back(3);
     g[5].push_back(2);


}
```

<u>If using a singly linked list,</u> of course, push_front is better - unless you are the instructor doing this on the whiteboard...

// Preprocessing

```
const int DEFAULT_VAL =  -1;   // must be less than 0

// Breadth First Search
// The unweighted shortest path algorithm on the graph g, with vertex
// i as the source
// Prints the length (number of edges) of the shortest path from the source
// to every vertex in the graph

void shortestpaths(const Graph & g, int s)
{
    queue<int> q;                // q is the queue of vertex numbers
    vector<vertexInf> vertices(g.size());// stores BFS info for the vertices
                                         // info for vertex j is in position

   for (int j=0; j < vertices.size(); ++j)// Initialize distances and prev values
       { vertices[j].dist = DEFAULT_VAL; vertices[j].prev = DEFAULT_VAL; }

   vertices[s].dist = 0;



   //rest of code on the next slide



}
```

```
    //continuation of code from previous slide
    q.push(s);
    while  (!q.empty() )
    {

        int v = q.front();
        q.pop();
        for (list<int>::const_iterator w = g[v].begin(); w != g[v].end(); w++)
        {

            if (vertices[*w].dist == DEFAULT_VAL)
             //distance of *w from source not determined yet
            {
                vertices[*w].dist = vertices[v].dist+1;
                vertices[*w].prev = v;
                q.push(*w);
            }
        }
    }
   for (int j = 0; j < vertices.size(); j++)// print distances from source and paths
   {
          cout << "vertex " << j << endl;
          cout << "distance: " << vertices[j].dist << endl;
          cout << "shortest path: ";
          printpath(j,vertices);
          cout << endl;
   }
}
```

```
void printpath(int j, const vector<vertexInf> & vinfo)
{
    stack<int> t;

    int current = j;
    while (current != DEFAULT_VAL)
    {
       t.push(current);
       current = vinfo[current].prev;
    }
    while (!t.empty())
    {
       cout << t.top() << " ";
       t.pop();
    }
}
```