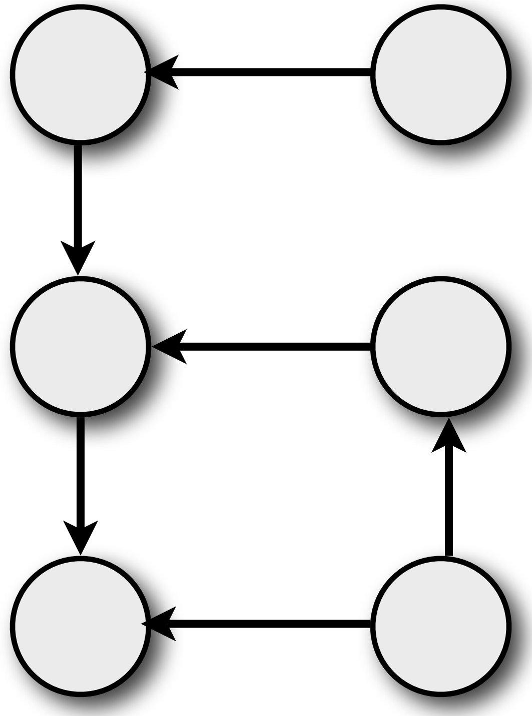
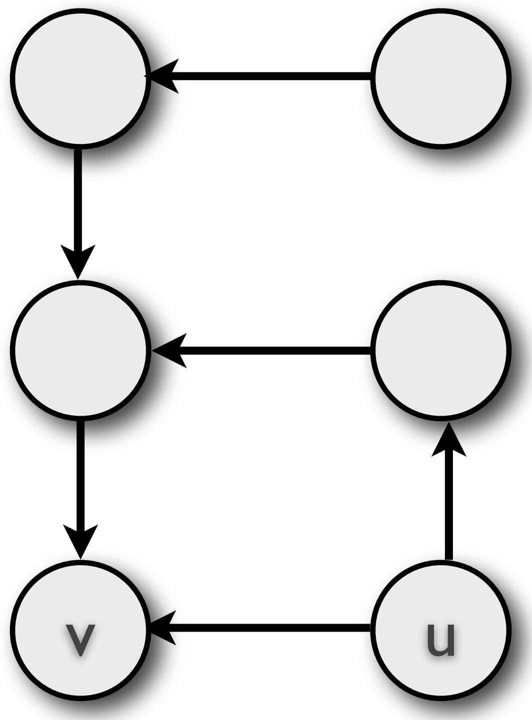
Graph

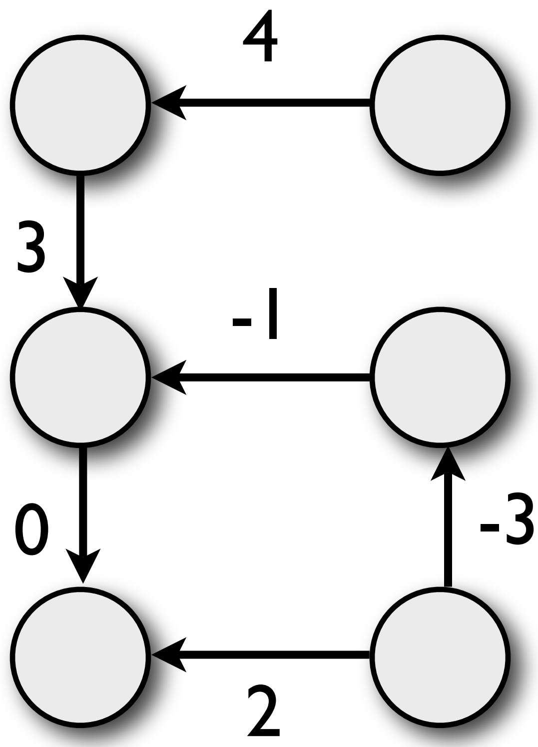
Ooi Wei Tsang
School of Computing, NUS



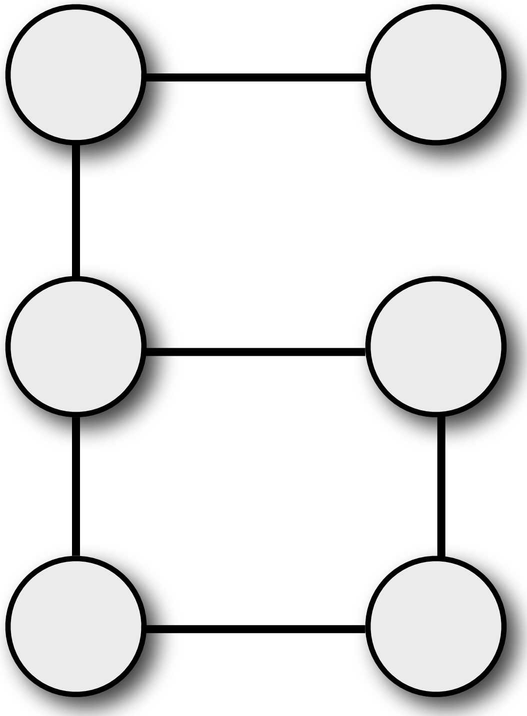
A graph consists of edges and vertices.



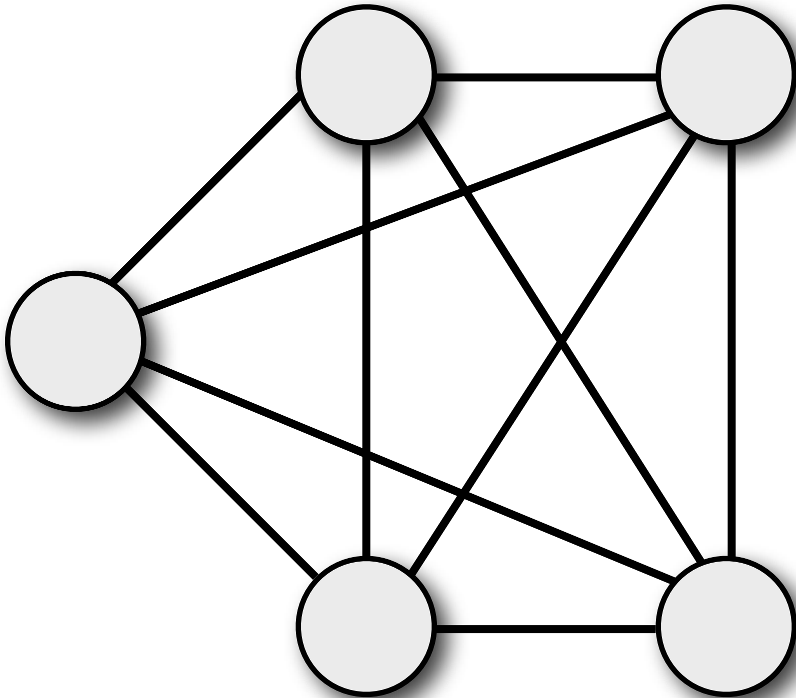
A vertex u is a neighbor of v, if there is an edge from v to u. We say u is adjacent to v. The number of neighbors of a vertex is called degree.



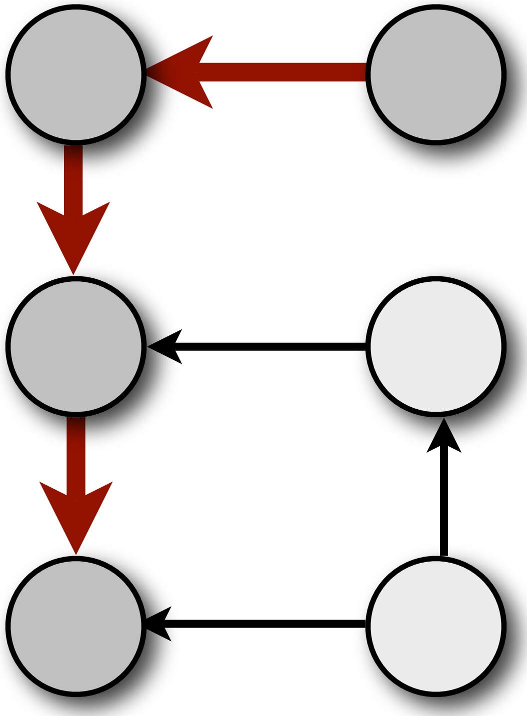
A weighted graph has a value associated with its edges.



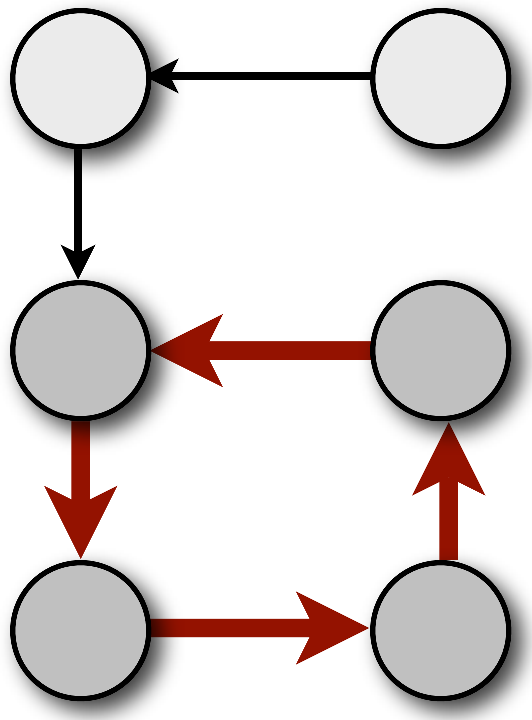
Direction of edges does not matter in a undirected graph.



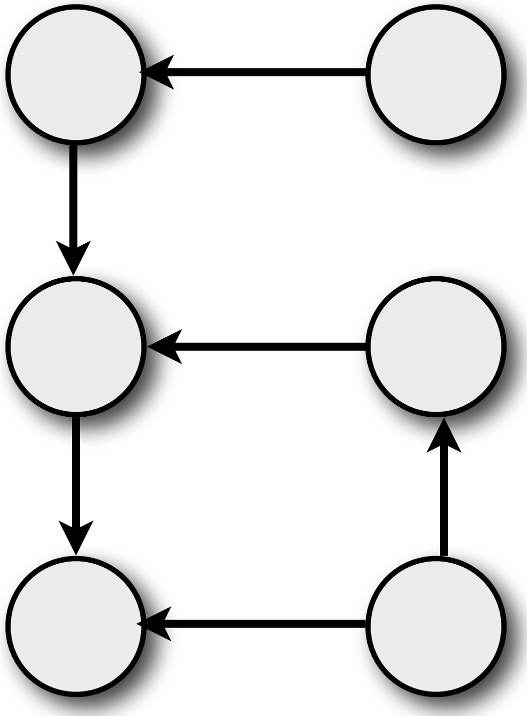
In a complete graph, every vertex is connected to every other vertices.



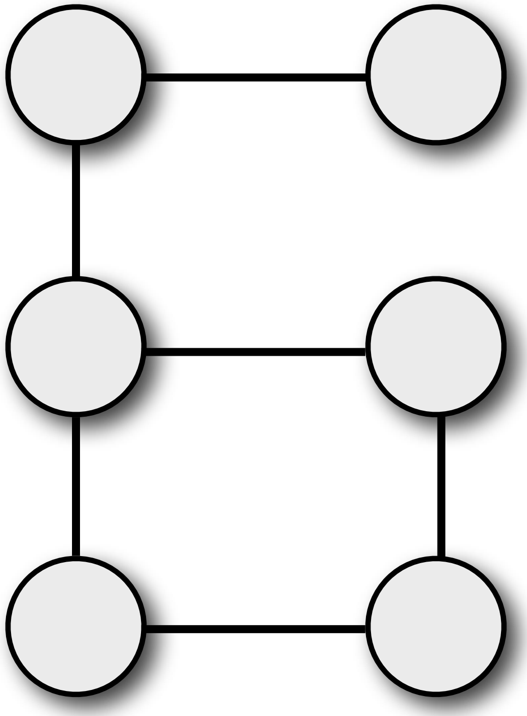
A path consists of a sequence of vertices adjacent to each other.



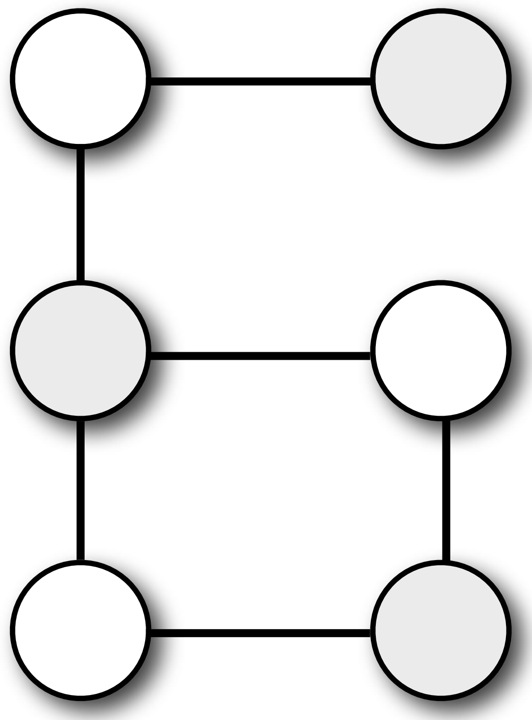
A cycle is a path that starts
and ends with the same vertex.



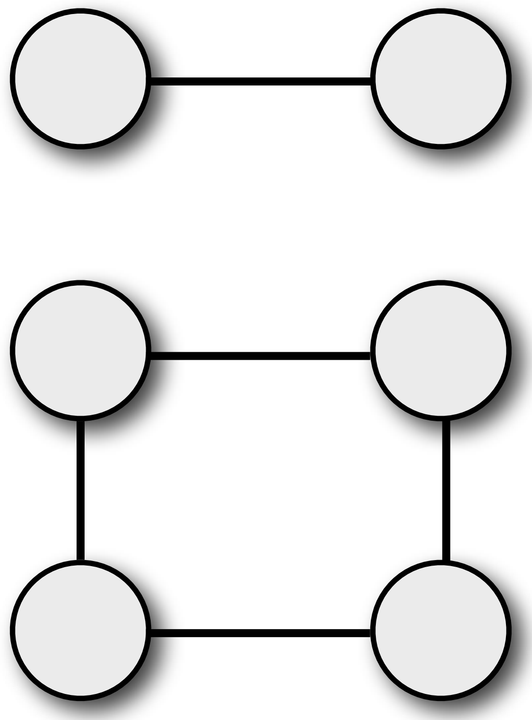
A graph is acyclic if it contains no cycle.
It is cyclic otherwise.



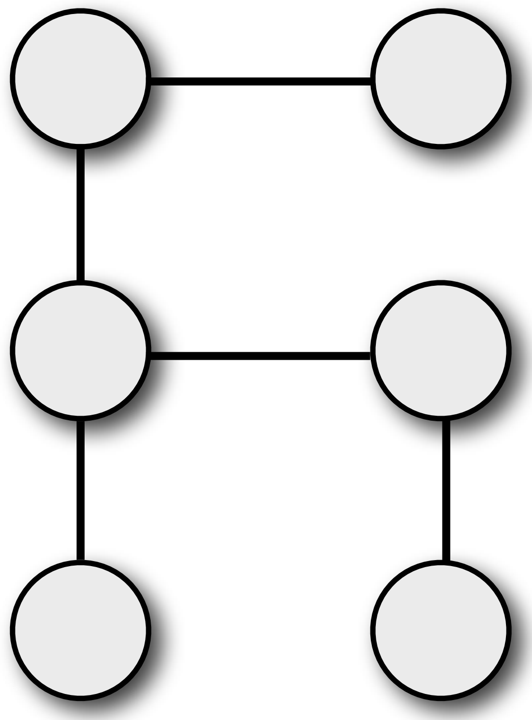
A undirected graph is connected
if there is a path between any two vertices.



A undirected graph is bipartite if we can partition the vertices into two sets and there are no edges between two vertices of the same set.



A unconnected graph consists of two connected components.



A connected, undirected, acyclic graph is called a tree.

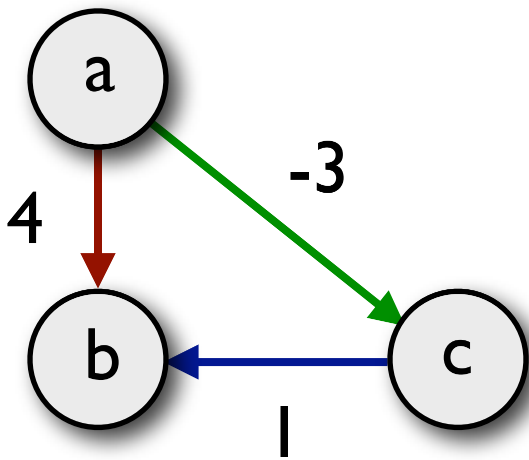
A weighted graph $G = (V, E, w)$, where

- V is the set of vertices
- E is the set of edges
- w is the weight function

$V = \{ a, b, c \}$

$E = \{ (a,b), (c,b), (a,c) \}$

$w = \{ ((a,b), 4), ((c, b), 1), ((a,c),-3) \}$

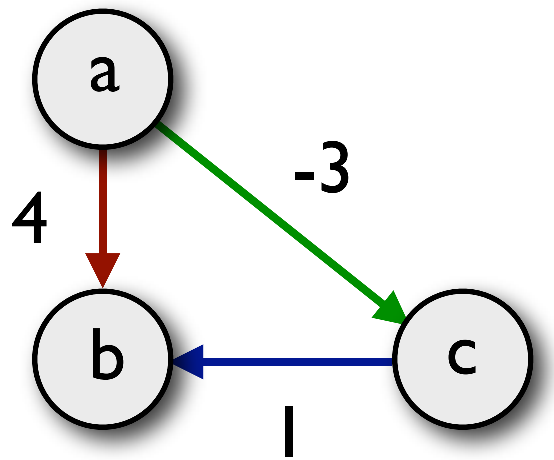


adj(*v*) : set of vertices adjacent to vertex *v*

adj(a) = {b, c}

adj(b) = { }

adj(c) = {b}



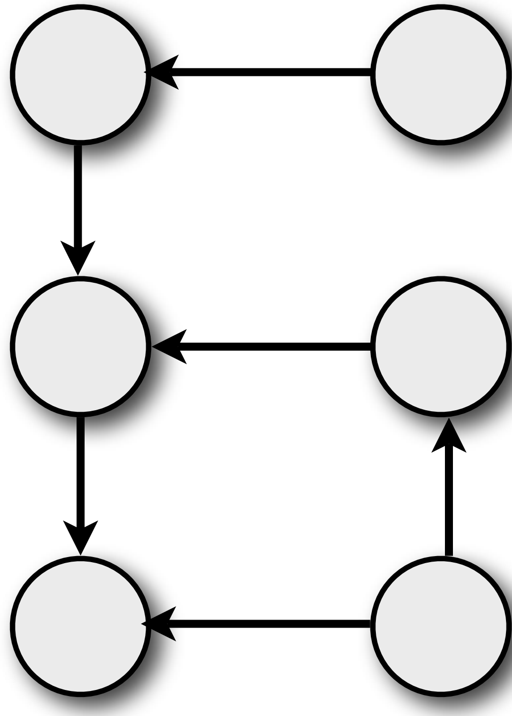
Review Questions

- How many edges are there in a undirected complete graph with N vertices?

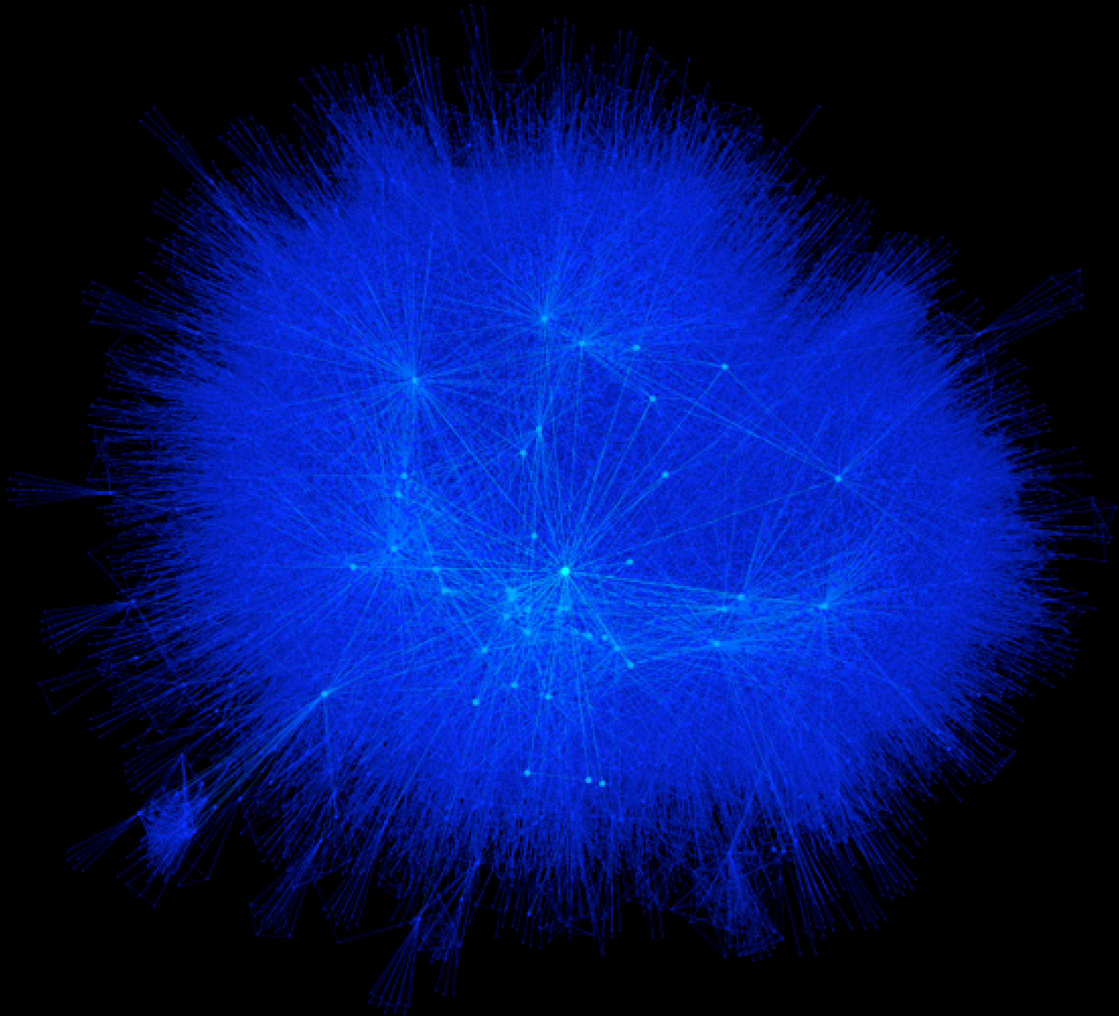
Review Questions

$$\sum_{u \in V} |adj(u)| = \ ?$$

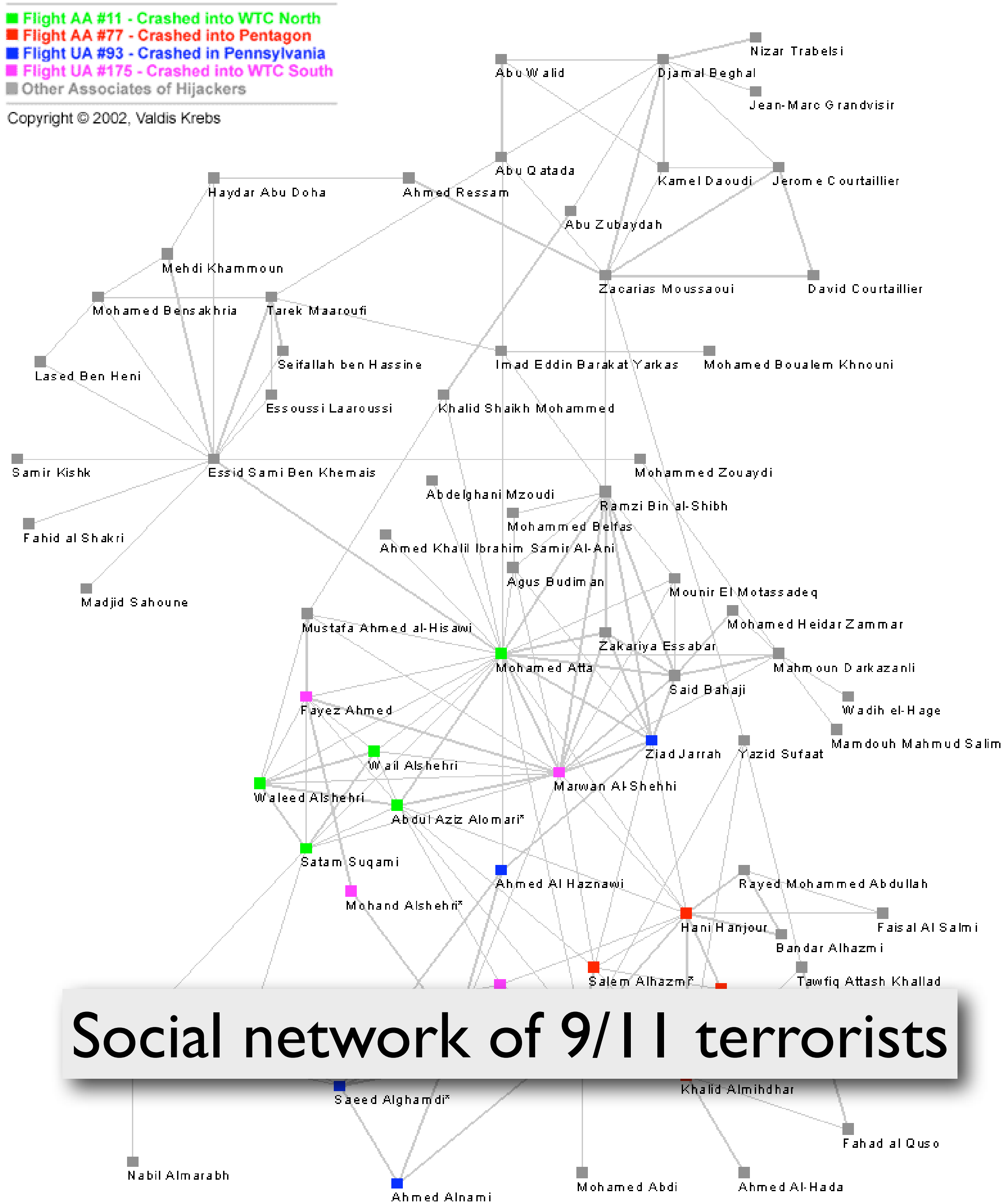
Example Applications



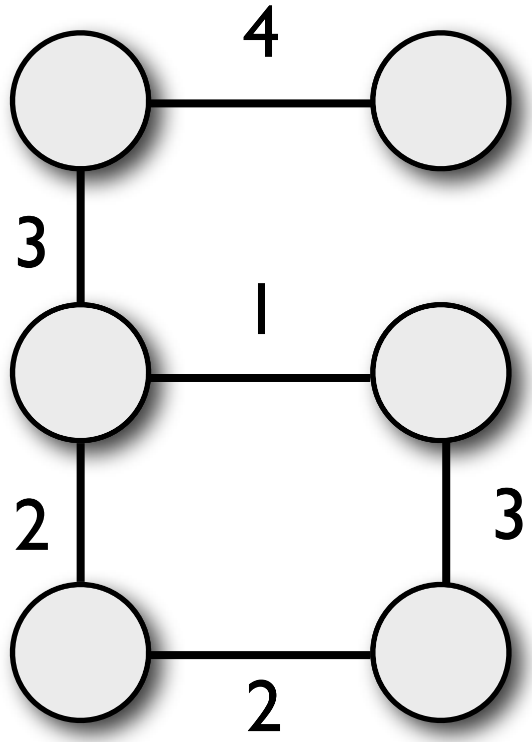
Representing a social network.
(u,v) in E if u knows v.



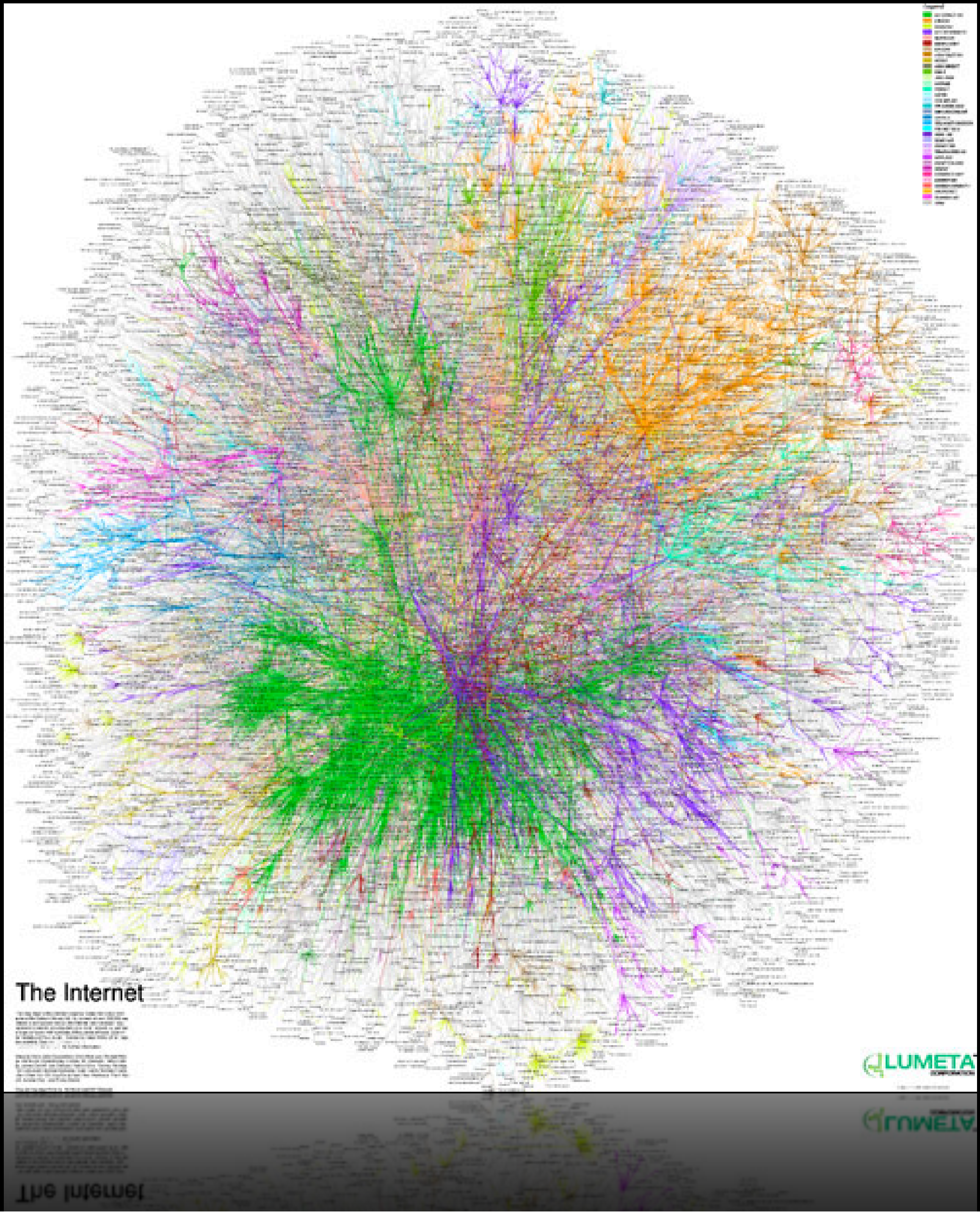
Jeffrey Heer's Social Network from Friendster
(47471 people, 432430 edges)

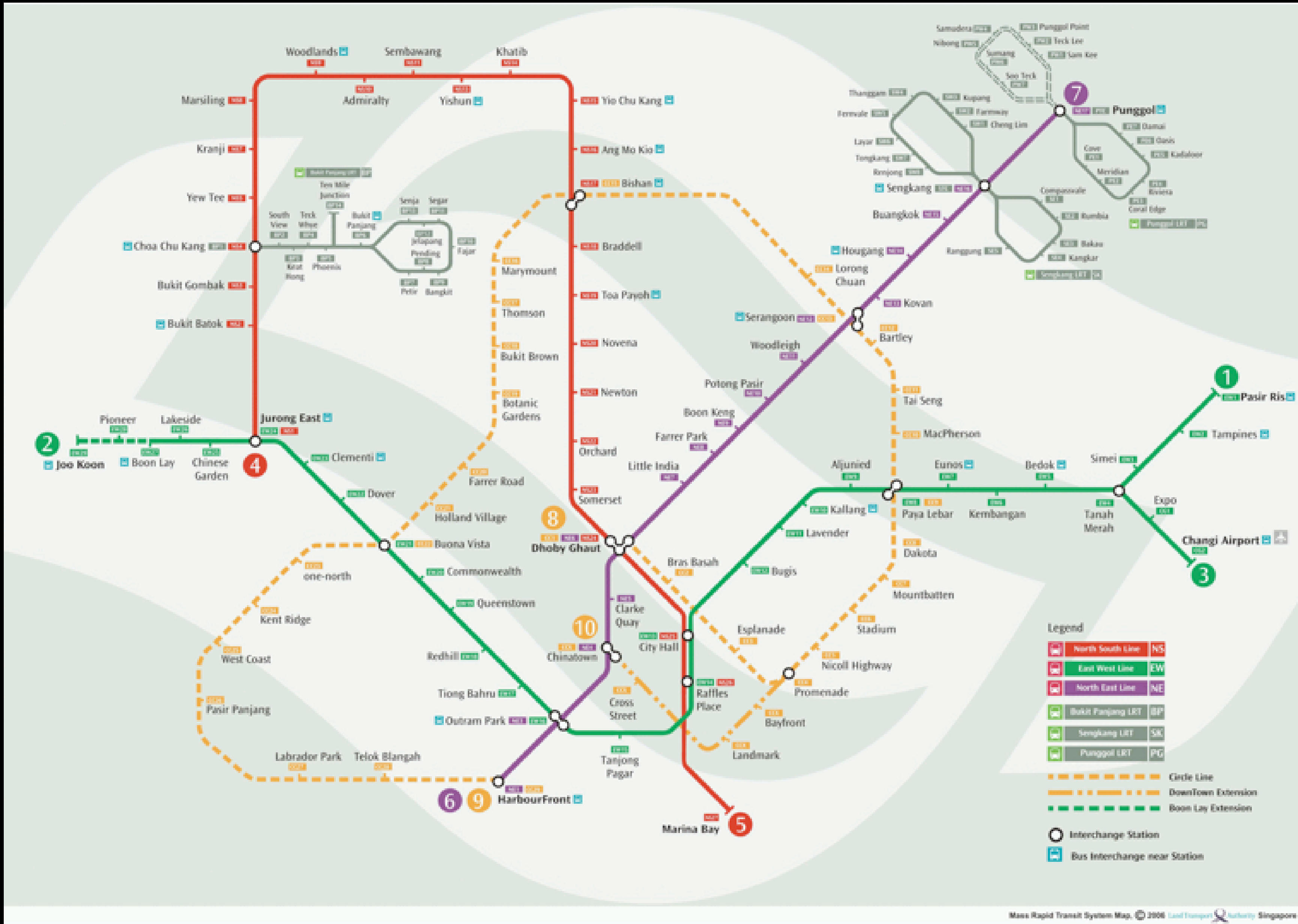


Social network of 9/11 terrorists



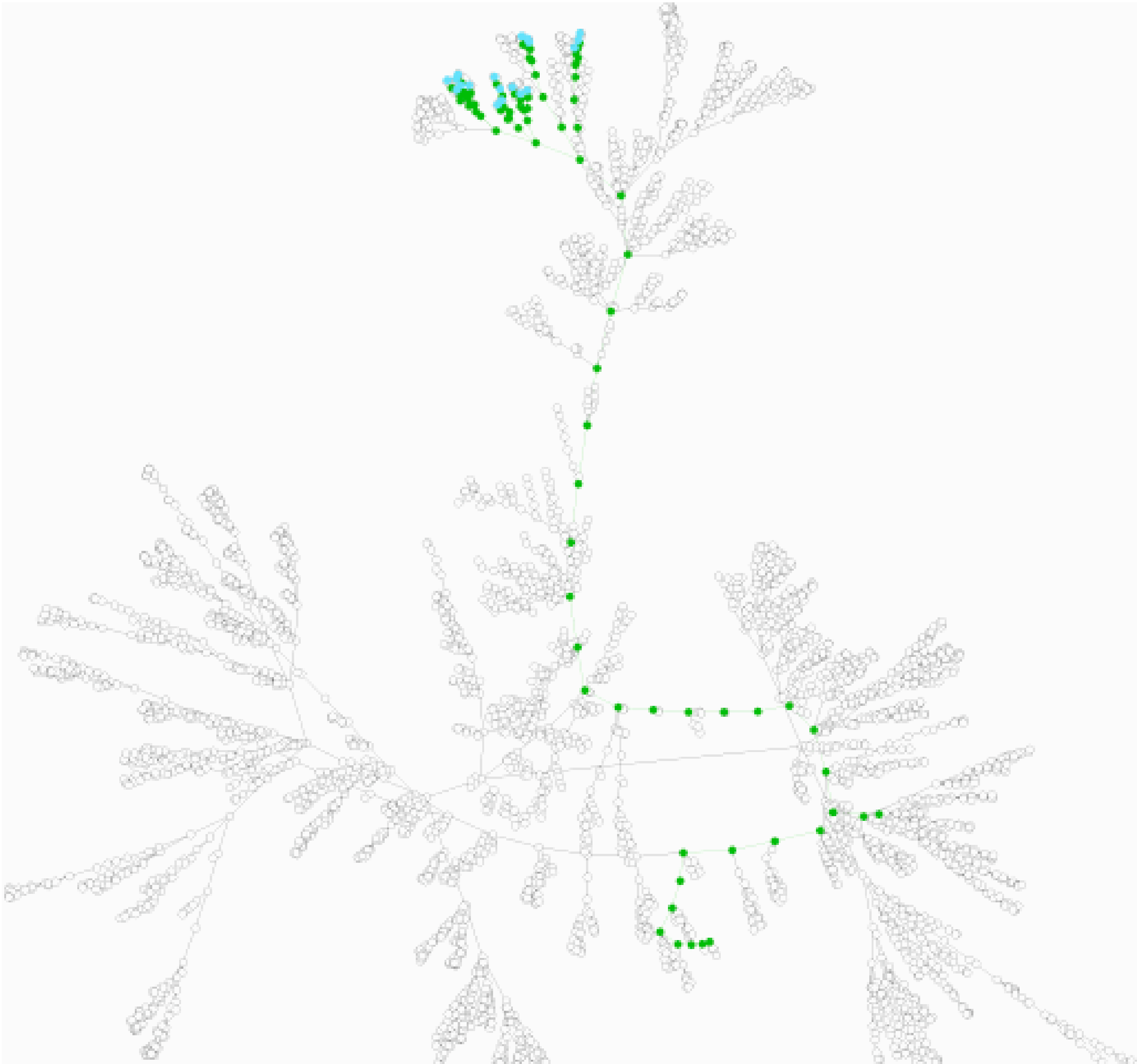
Representing places and routes. (u,v) exists if there is a direct route from u to v. Weight w(u,v) is the distance or cost. We are often interested in finding the cheapest path between between two places.


The Internet
LUMETA



Woodlands
Sembawang
Khatib
Admiralty
Yishun
Marsiling
Kranji
Yew Tee
Choa Chu Kang
Bukit Gombak
Bukit Batok
Jurong East
Pioneer
Lakeside
Joo Koon
Boon Lay
Chinese Garden
Clementi
Dover
Buona Vista
Commonwealth
Queenstown
Redhill
Tiong Bahru
Outram Park
Tanjong Pagar
Yio Chu Kang
Ang Mo Kio
Bishan
Braddell
Toa Payoh
Novena
Newton
Orchard
Somerset
Dhoby Ghaut
City Hall
Raffles Place
Marina Bay
Marymount
Thomson
Bukit Brown
Botanic Gardens
Farrer Road
Holland Village
one-north
Kent Ridge
West Coast
Pasir Panjang
Labrador Park
Telok Blangah
HarbourFront
Chinatown
Clarke Quay
Little India
Farrer Park
Boon Keng
Potong Pasir
Woodleigh
Serangoon
Kovan
Hougang
Buangkok
Sengkang
Punggol
Lorong Chuan
Bartley
Tai Seng
MacPherson
Paya Lebar
Dakota
Mountbatten
Stadium
Nicoll Highway
Promenade
Bayfront
Landmark
Esplanade
Bras Basah
Cross Street
Bugis
Lavender
Kallang
Aljunied
Eunos
Kembangan
Bedok
Tanah Merah
Simei
Tampines
Pasir Ris
Expo
Changi Airport
Legend
North South Line NS
East West Line EW
North East Line NE
Bukit Panjang LRT BP
Sengkang LRT SK
Punggol LRT PG
Circle Line
DownTown Extension
Boon Lay Extension
Interchange Station
Bus Interchange near Station

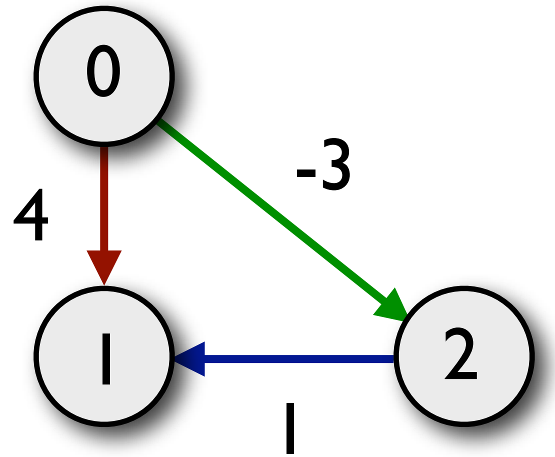
MRT ROUTE MAP



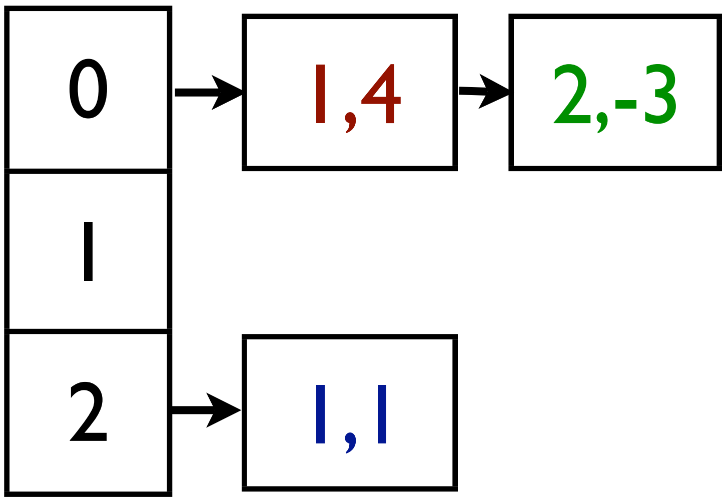
Possible moves in Rush Hour. Blue represents solutions.
Green represents the shortest paths to solving the puzzle.
(from www.aisee.com)

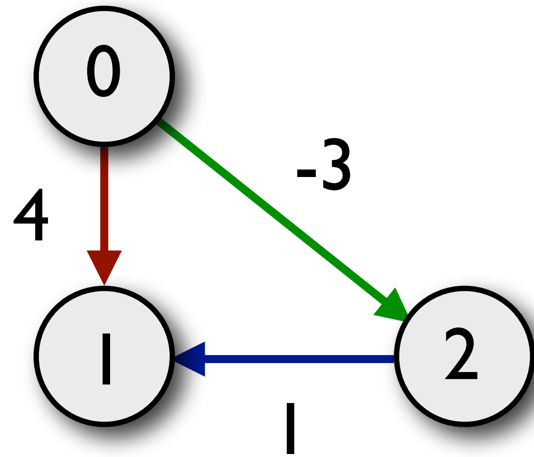
Implementation

	0	1	2
0	∞	4	-3
1	∞	∞	∞
2	∞	1	∞



Adjacency Matrix: Use a 2D array. Store w(u,v) in a[u][v] if edge (u,v) exists. Store an invalid value otherwise.





Adjacency List: Use an array of link list. a[u] stores adj(u) and the associated weight.

- How long does it take to delete an edge for

  (a) adjacency matrix ?

  (b) adjacency list ?

- How long does it take to go through all neighbors of a vertex *v* for

    (a) adjacency matrix ?

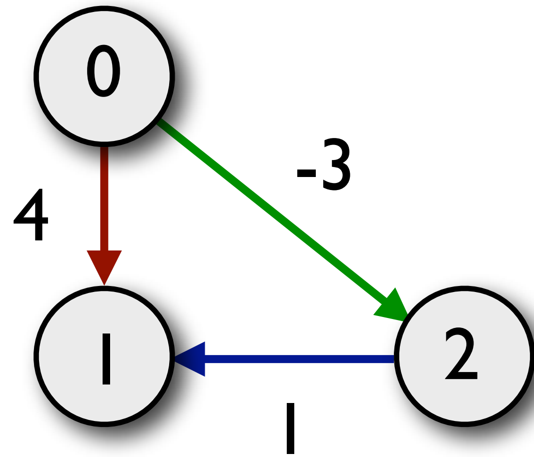
    (b) adjacency list ?

- How much space is needed to store a graph of size N if we are using

  (a) adjacency matrix ?

  (b) adjacency list ?

0	1,4	2,-3	
1			
2	1,1		



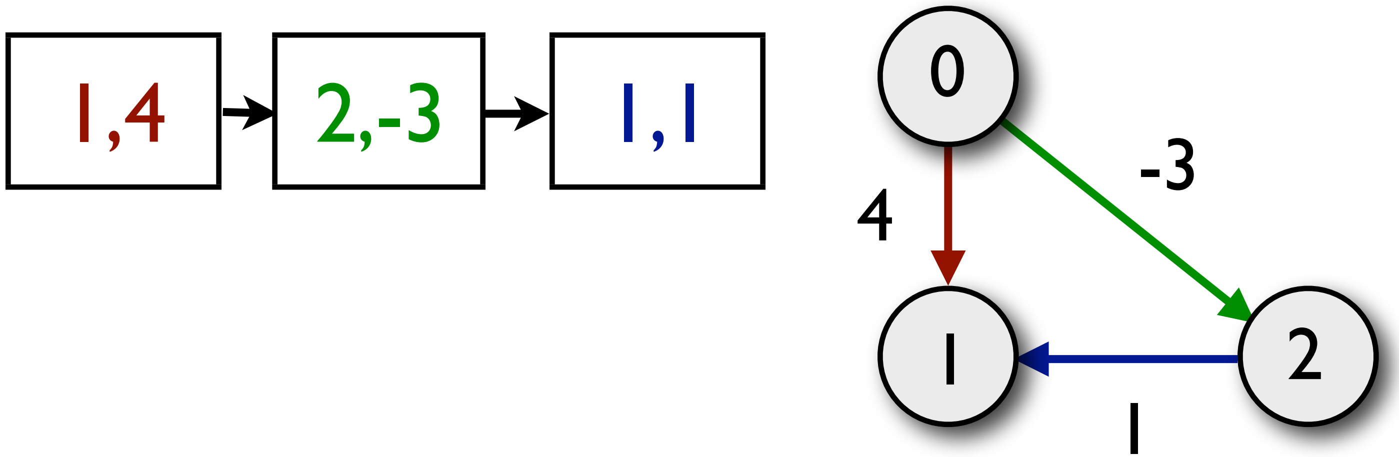
Adjacency List in Matrix: Use a 2D array. Each row is an array-representation of the adjacency list.

Avoid using pointers in competitive programming.

Most of the time, graph are static (no insert/delete after initialization).

Maximum size is often given.

```c
typedef struct neighbor {
    int id;
    int weight;
} neighbor;

// N is max num of vertices;
neighbor graph[N][N];
int num_of_vertices;
```



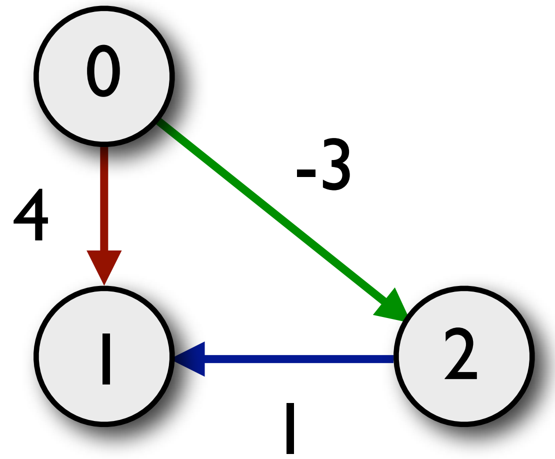
Edge List: Use a linked list of edges.

Edges

1,4	2,-3	1,1

Degree

2	0	1



Edge List: Use a array of edges.

```c
typedef struct edge {
    int from;
    int to;
    int weight;
} edge;

edge graph[MAX_NUM_OF_EDGES];
int num_of_edges;
```

Pick the simplest implementation that meets the requirements.

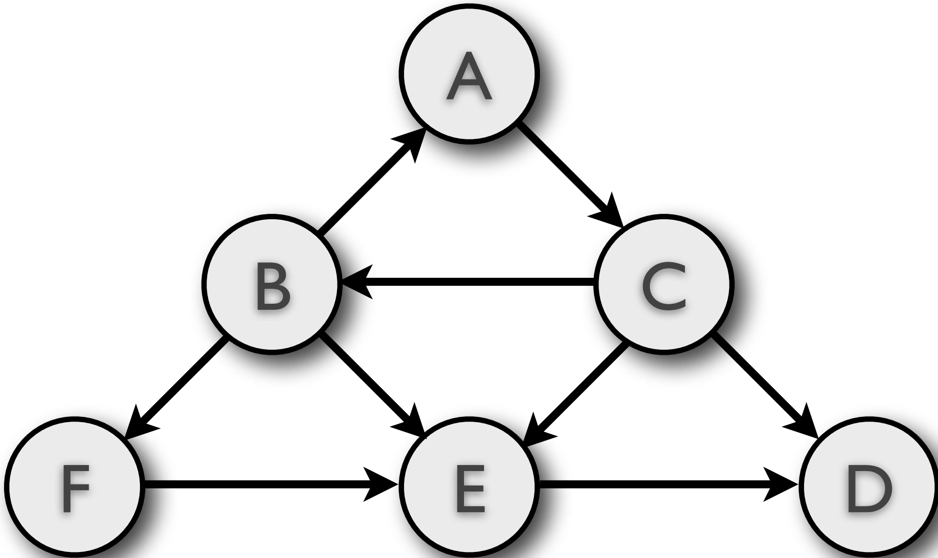
Graph Traversal

How to systematically visit the whole graph?

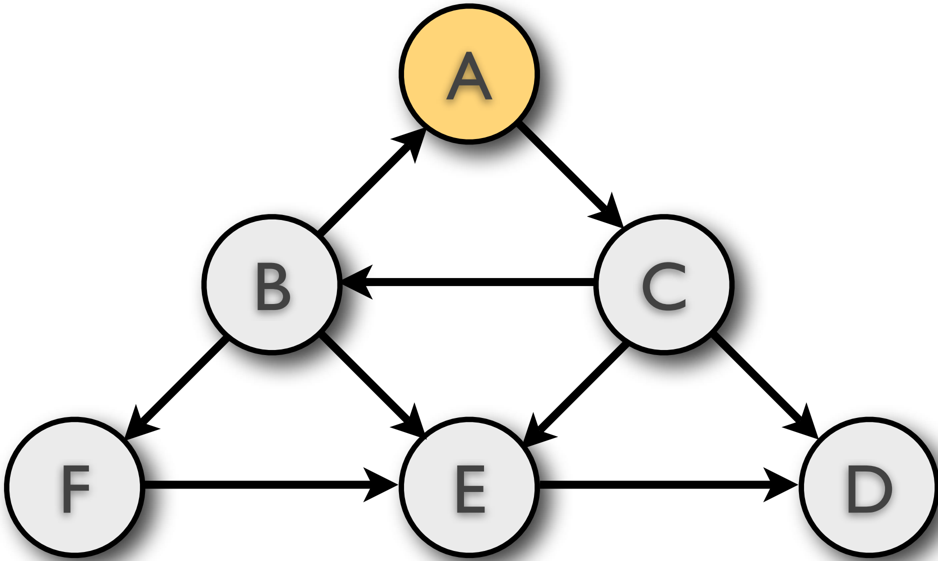
Breadth-First Search

or BFS

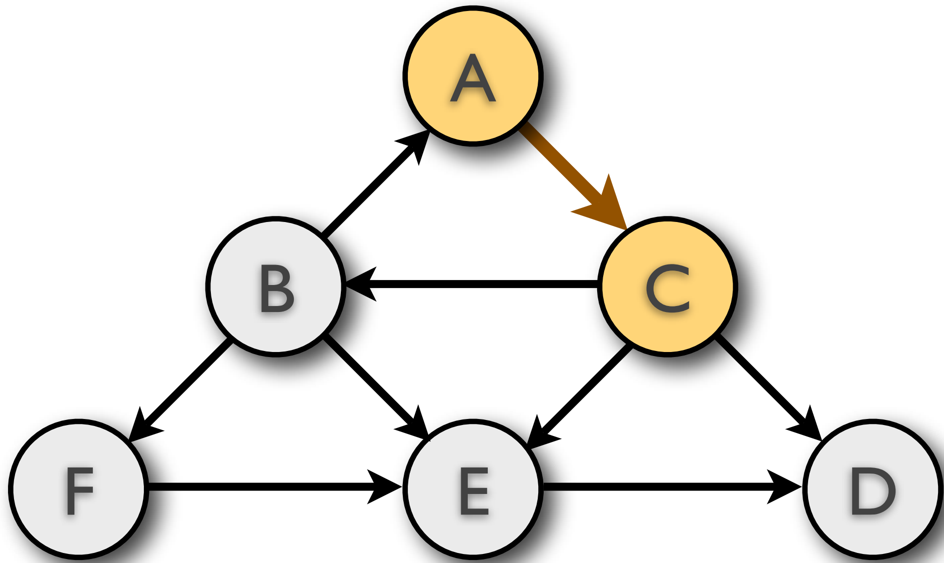
- **Basic idea**: pick a source and visit the vertices in increasing distance from the source
    - visit all vertices one hop away
    - visit all vertices two hops away etc.
- **Note**: A vertex u is k-hop away from the v if the shortest path from u to v consists of k edges.



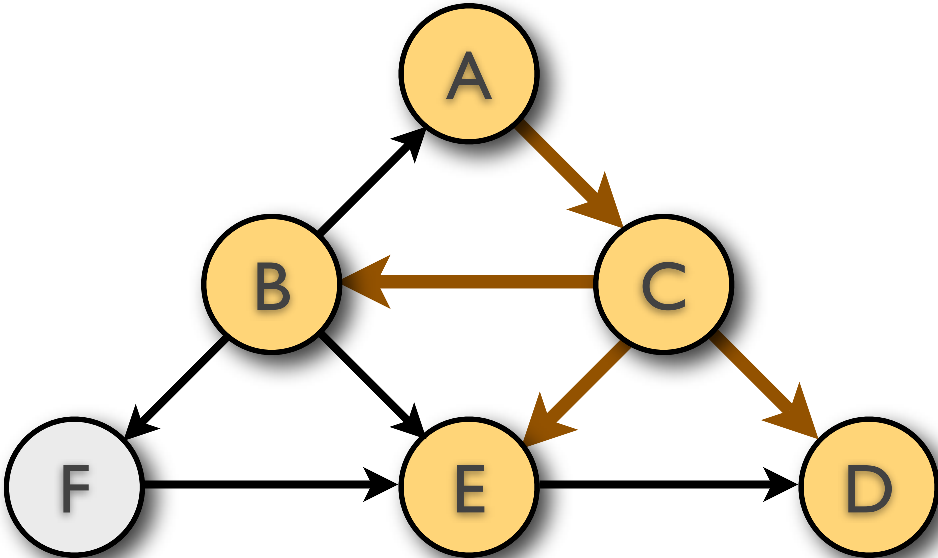
Example:
F is 3-hop away from A.
E is 2-hop away from A.



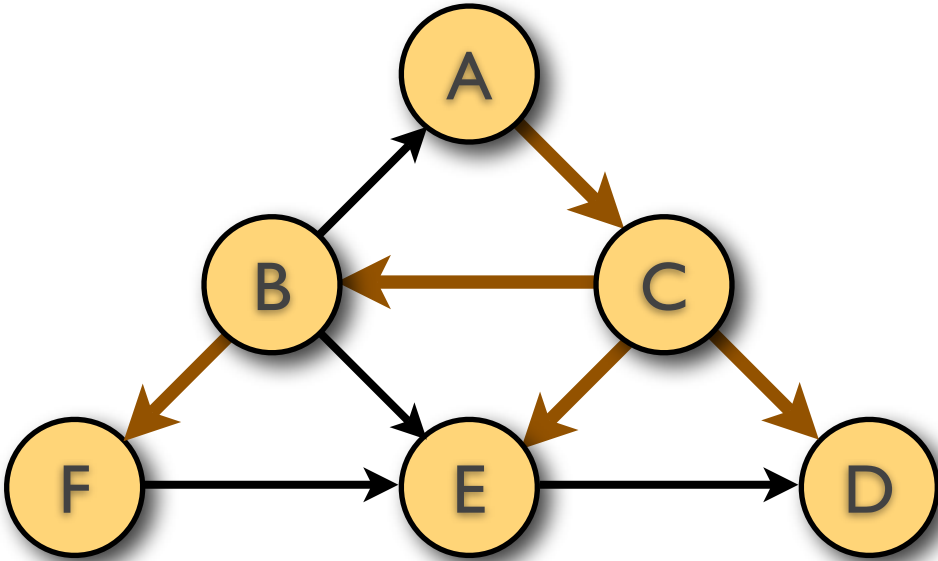
Let A be the source. We first visit the source.
I colored visited vertices yellow.



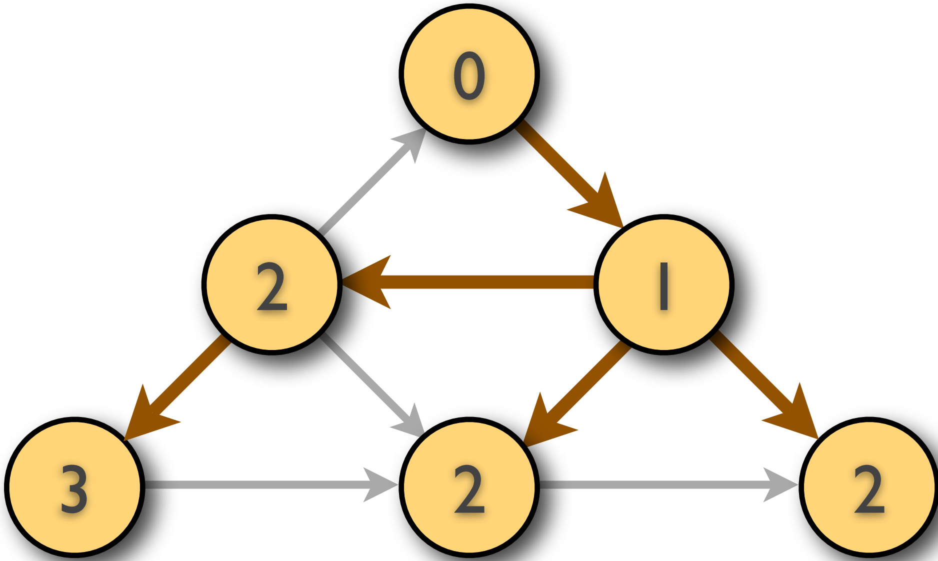
Next, visit the vertices that are one-hop away.



Next, visit the vertices that are two hops away.
(i.e, all unvisited vertices that are neighbors of one-hop neighbor of A.



Edges that lead to undiscovered node during traversal are colored brown.

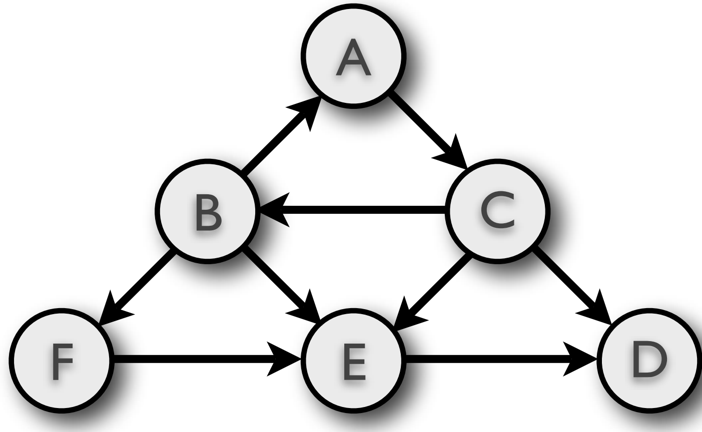


These edges form the breadth-first tree. Level of vertices in the tree is the hop distance from source.

- An implementation needs to keep track of vertices we have discovered.
- To visit the vertices in increasing order of hop distance, we need to visit the nodes the order we discover them (FIFO).

Q = **new** Queue
enqueue *source* into Q

while Q is not empty
  v = dequeue from Q
  mark v as visited
  for each neighbor u of v
    if u is not visited and not already in Q
      enqueue u into Q

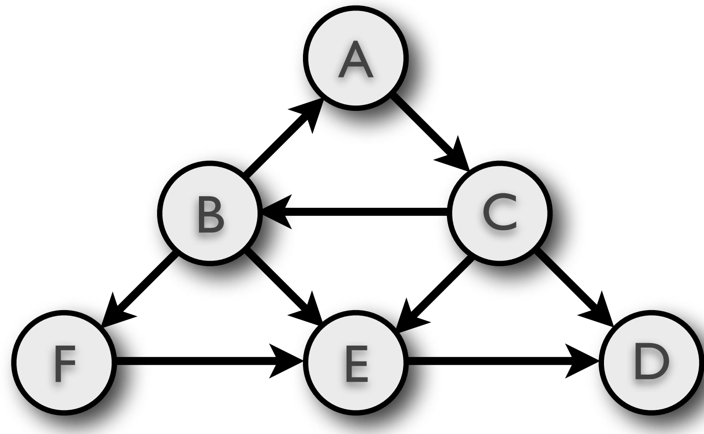


Review Questions

- Suppose we want to keep track of breadth-first tree by marking the edges in the tree as brown. How should we change the algorithm?

Q = **new** Queue
enqueue *source* into Q

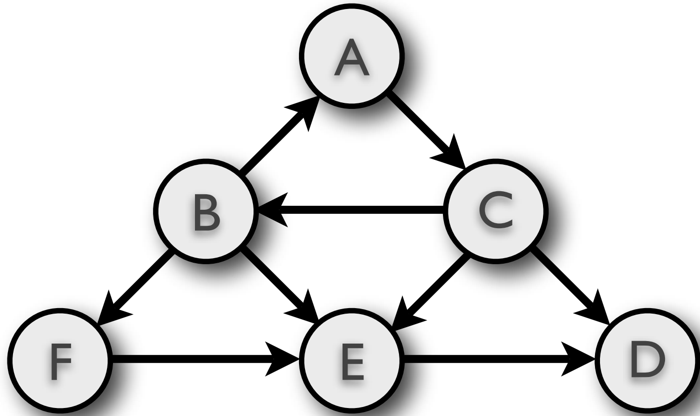
while Q is not empty
  v = dequeue from Q
  mark v as visited
  for each neighbor u of v
    if u is not visited and not already in Q
      mark (v,u) as brown
      enqueue u into Q



Review Questions

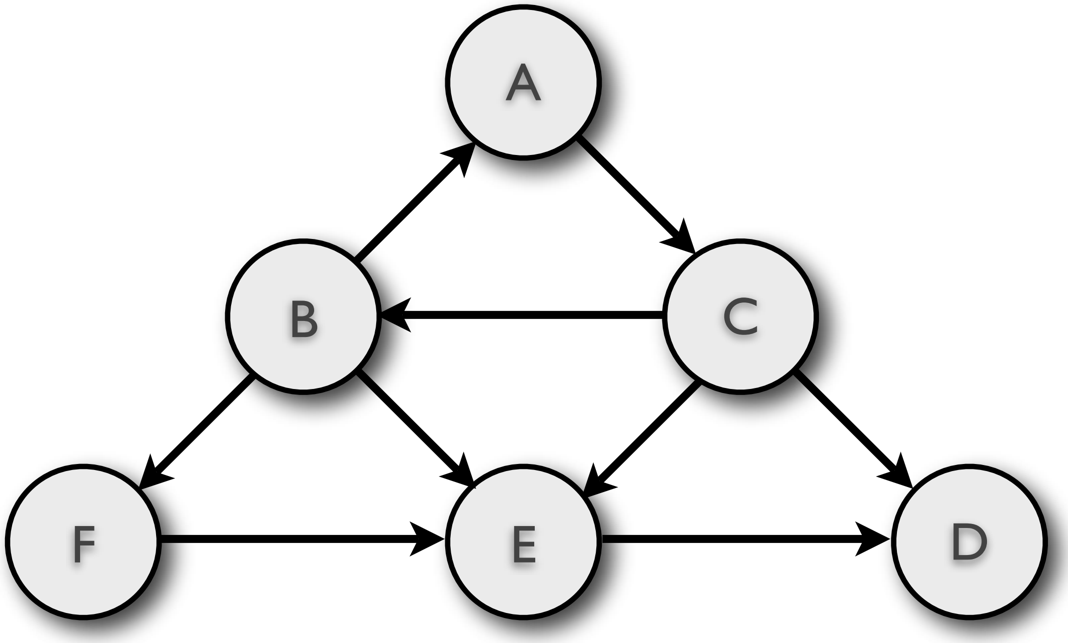
- Suppose we want to keep track of hop distance from the source. How should we change the algorithm?

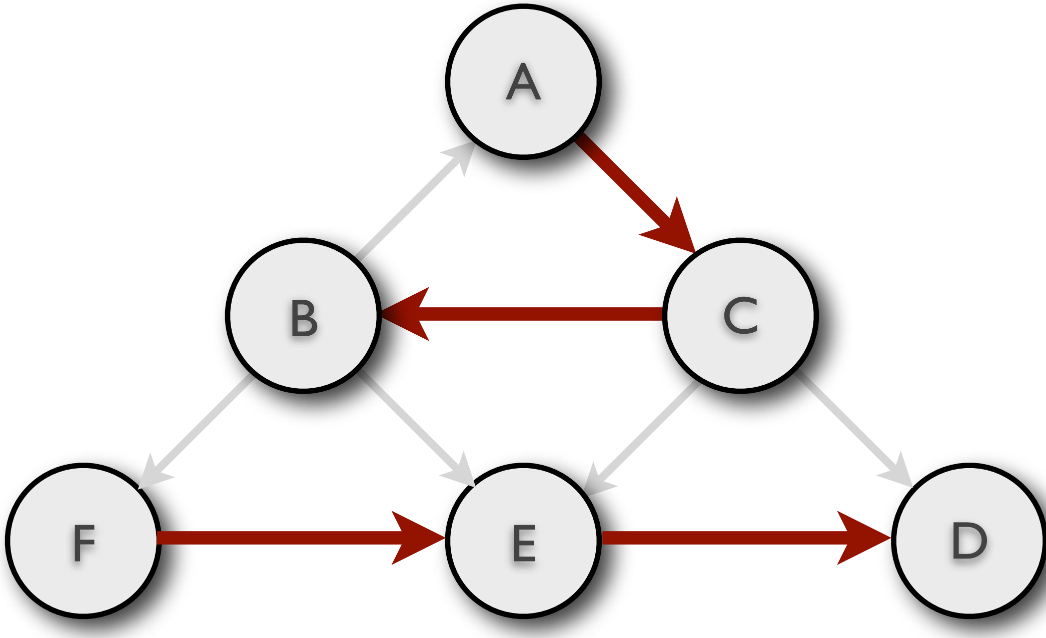
Q = **new** Queue
enqueue *source* into *Q*
level[*source*] = 0
while *Q* is not empty
  v = dequeue from *Q*
  mark *v* as visited
  for each neighbor *u* of *v*
    if *u* is not visited and not already in *Q*
      level[u] = level[v] + 1
      enqueue *u* into *Q*



Review Questions

- Can we always visit every vertex using the previous algorithm?


A
B
C
F
E
D

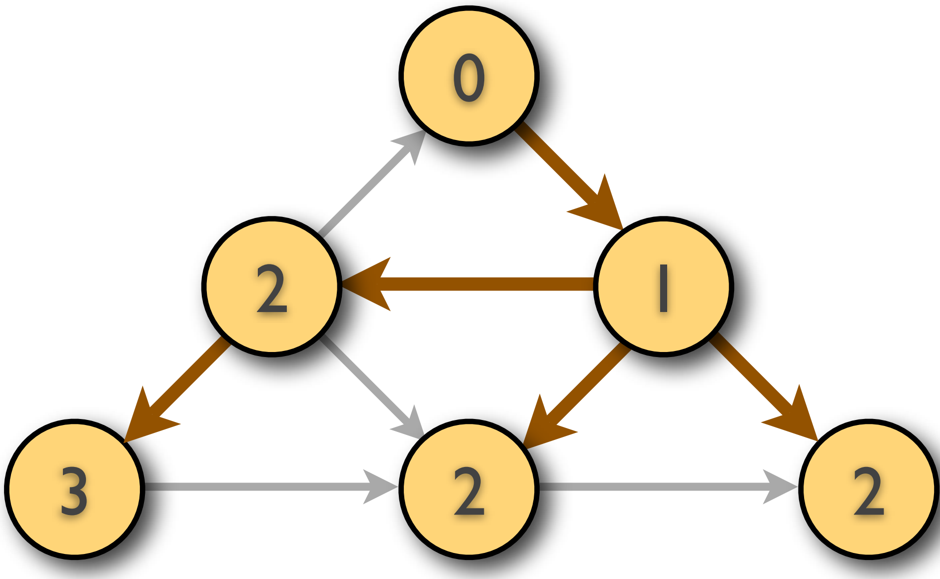


If we pick F as the source, then we can't visit A, B, and C, and need to visit them through another source.

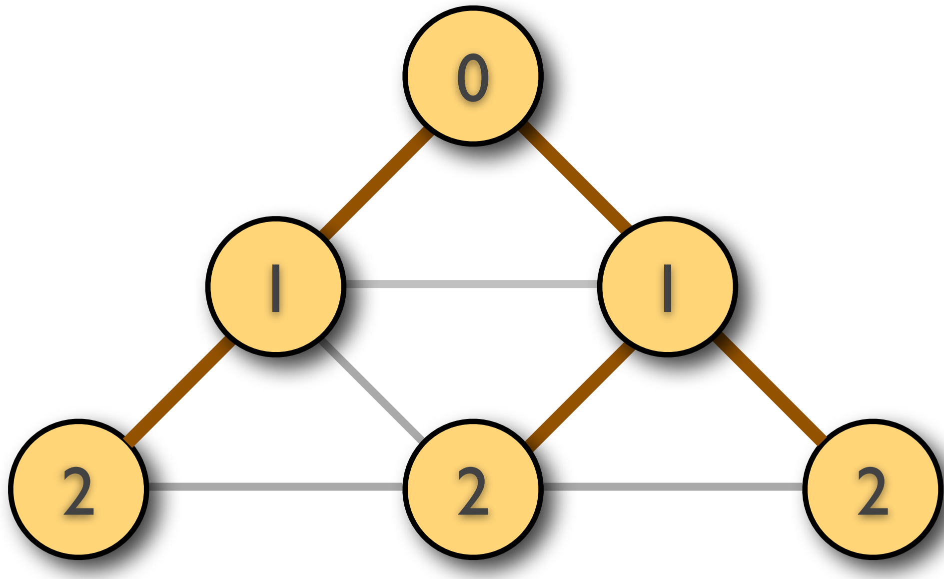
Mark all vertices as unvisited

for each vertex *v*
    if *v* is not visited
        use *v* as *source* and run BFS

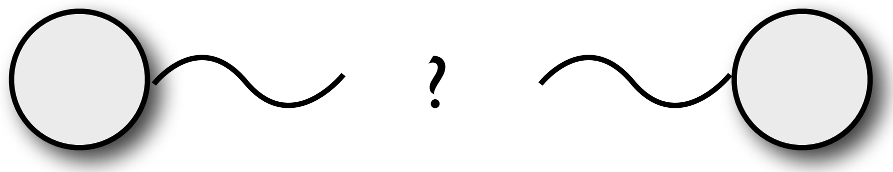
Applications of BFS



On an unweighted graph, the breadth-first tree tells us the shortest path from source to all the other vertices.



The algorithm works for undirected graph too.



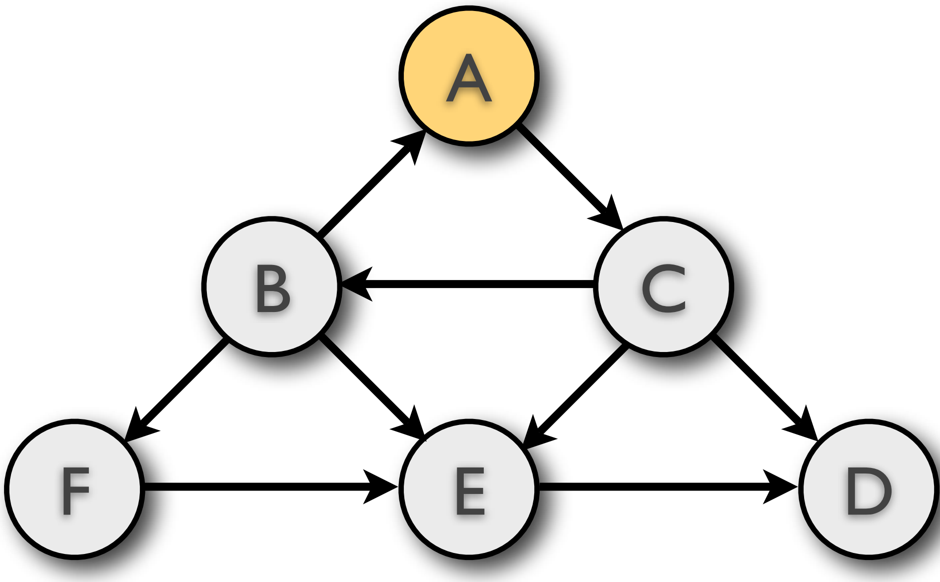
We can check if two vertices are connected using BFS.

Depth-First Search

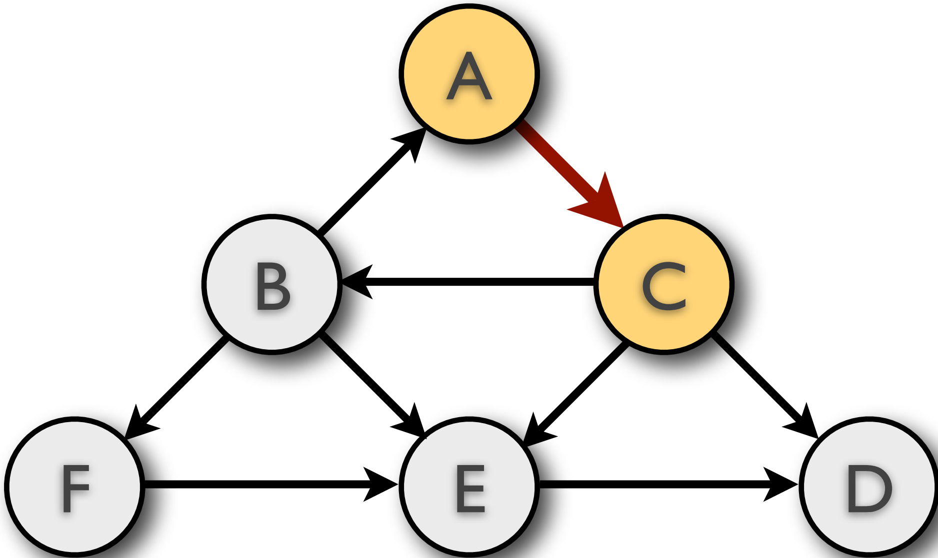
or, DFS

- **Basic idea**: Starting from a source, repeatedly visit a neighbor of the current vertex until we hit a dead-end (no unvisited neighbors), then backtrack.

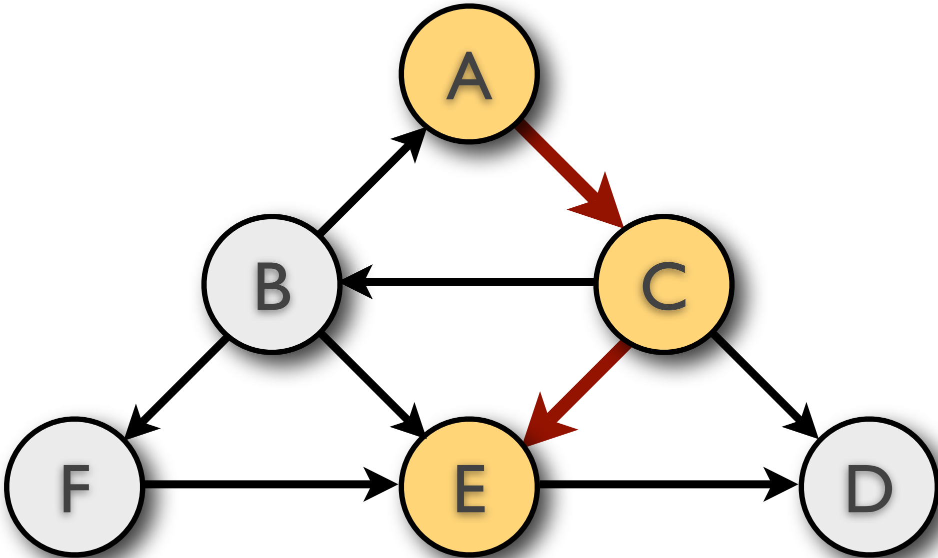
- After we visit a vertex v, we visit all vertices reachable from v.



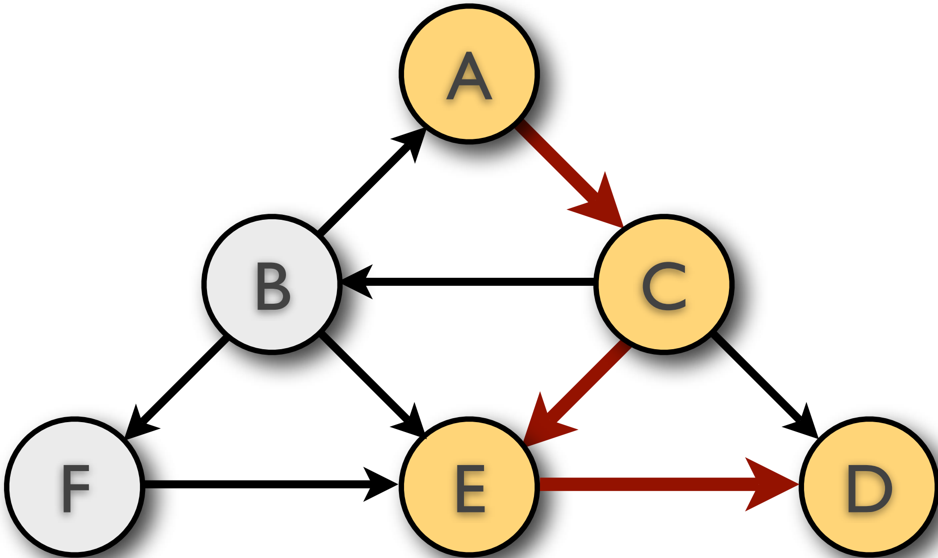
Let A be the source.



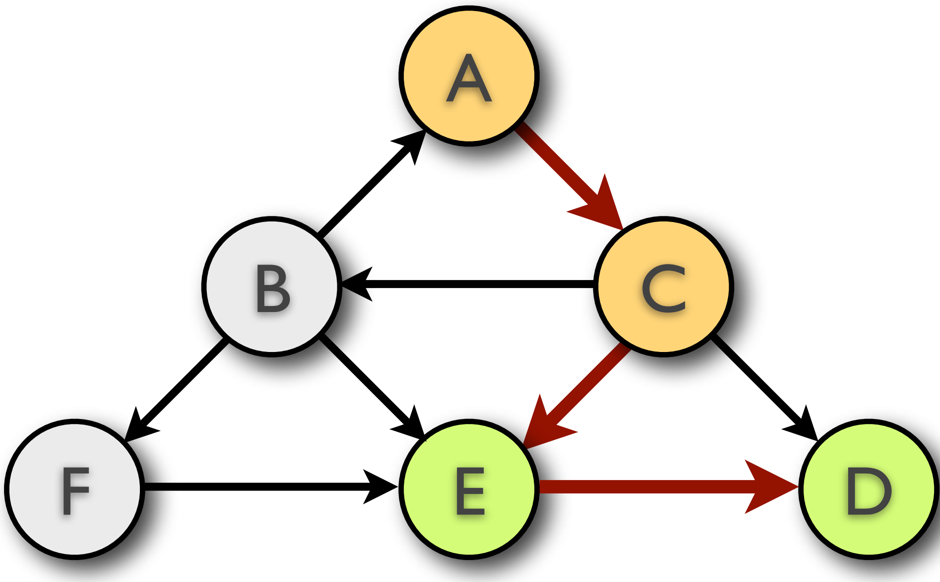
Visit a neighbor of A (say, C).



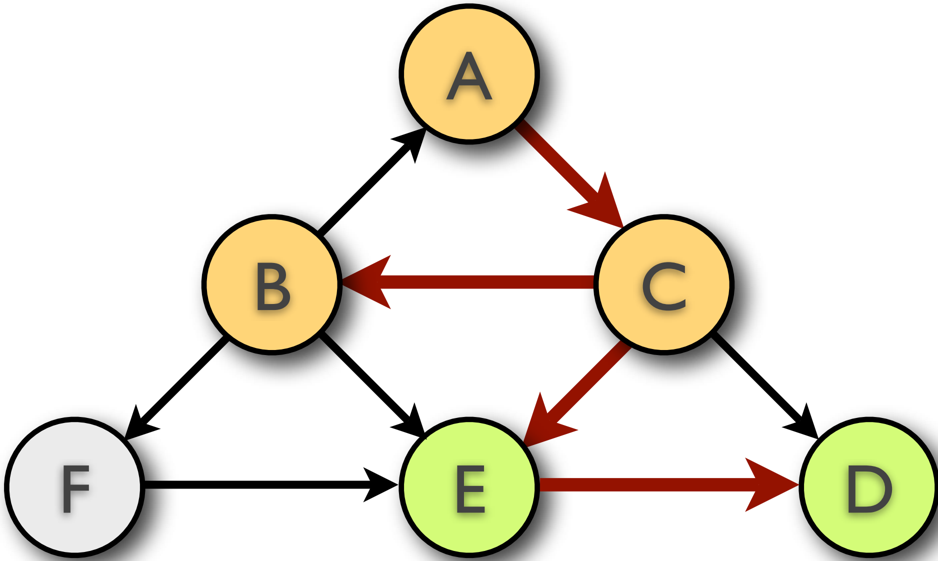
Visit a neighbor of C (say, E).



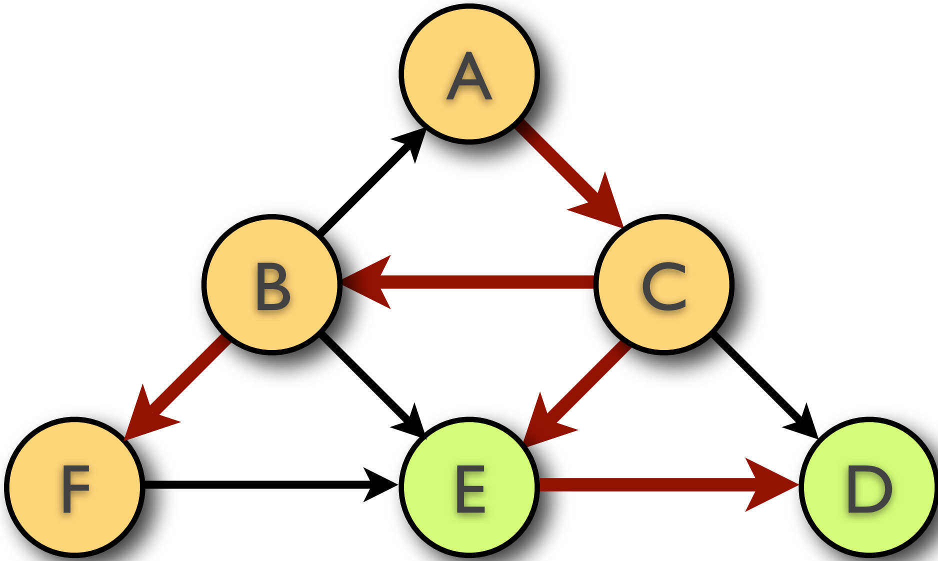
Visit a neighbor of E (say, D).



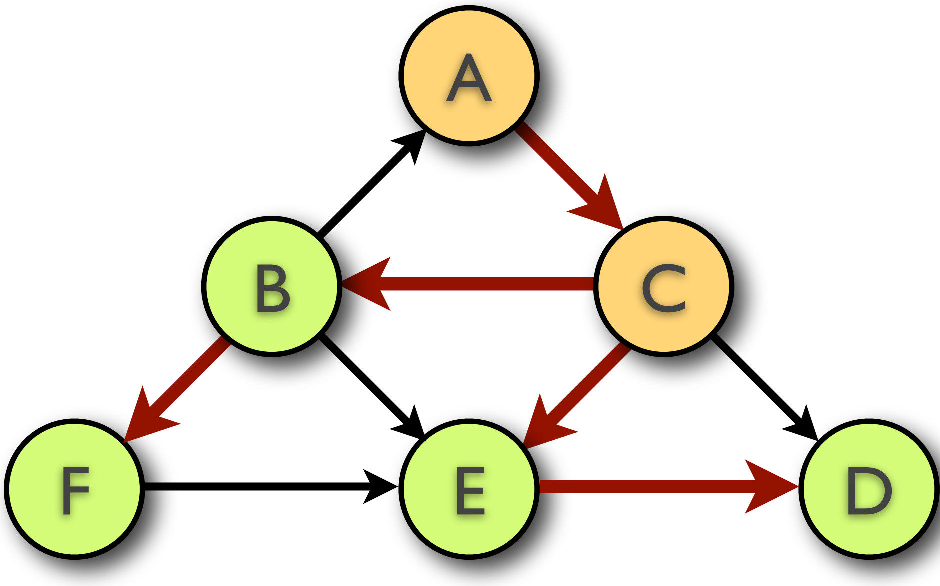
D has no neighbor. Back to E.
E has no unvisited neighbor. Back to C.



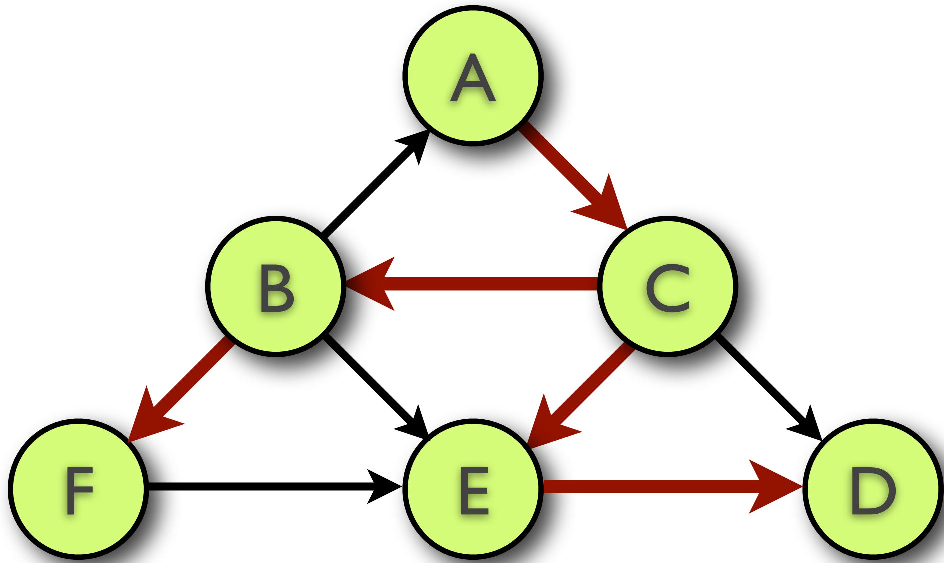
Visit B.



Visit F.



F has no unvisited neighbor. Back to B.
B has no unvisited neighbor. Back to C.



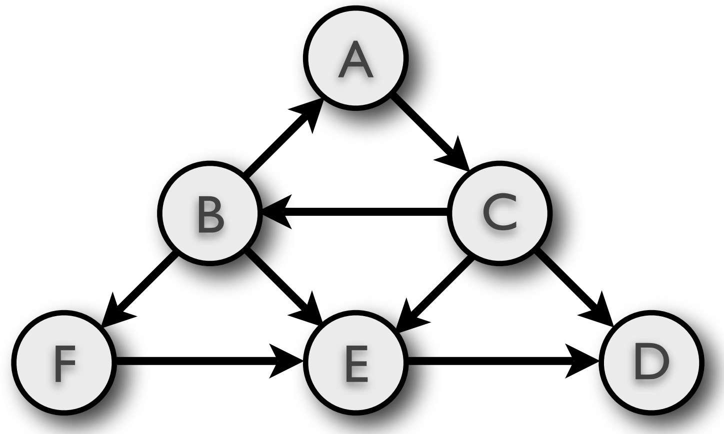
C has no unvisited neighbor. Back to A.

A, the source, has no unvisited neighbor. Done!

- An implementation needs to keep track of vertices we have discovered.
- When backtrack, we need to go back to the last vertex we visited. (LIFO).

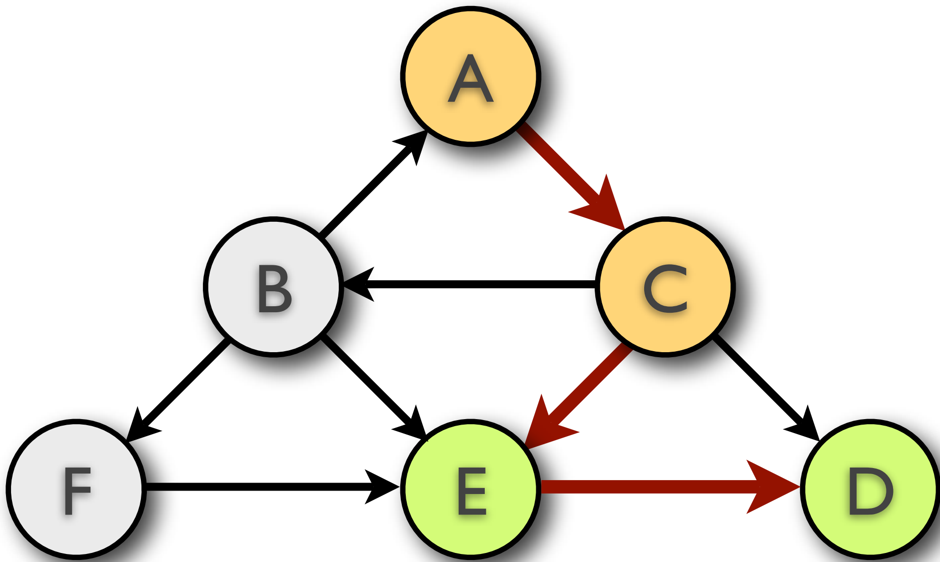
S = **new** Stack
push *source* onto *S*

while *S* is not empty
  v = top of *S*
  if *v* has a unvisited neighbor *u*
    mark *u* as visited
    push *u* onto *S*
  else
    pop *v* from *S*



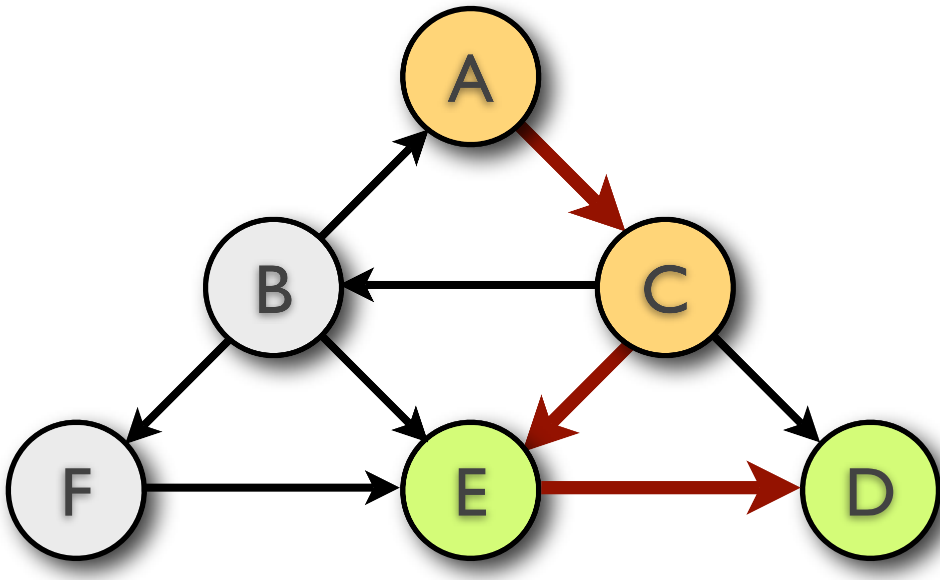
Mark all vertices as unvisited

for each vertex *v*
    if *v* is not visited
        use *v* as *source* and run DFS



D has no neighbor. Back to E. E has no unvisited neighbor. Back to C.

- What is the color of a vertex:
  (a) before it is inserted into the stack ?
  (b) while it is inside the stack ?
  (c) after it is pop from the stack ?



D has no neighbor. Back to E. E has no unvisited neighbor. Back to C.

A vertex can be in three states: unvisited, visiting, visited.

S = **new** Stack

push *source* onto S

while S is not empty

  v = top of S

  if v has a unvisited neighbor u

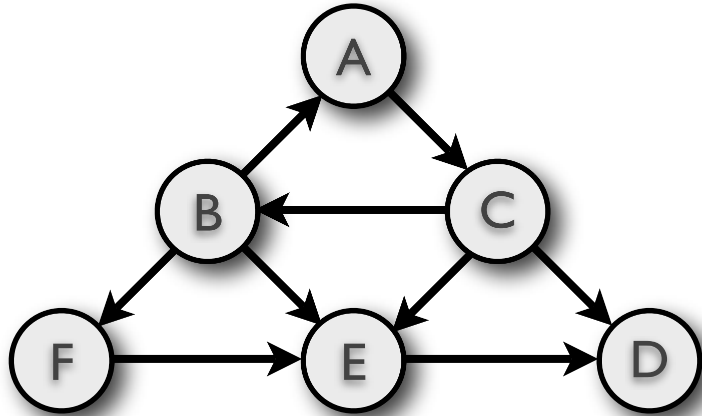
    mark u as "visiting"

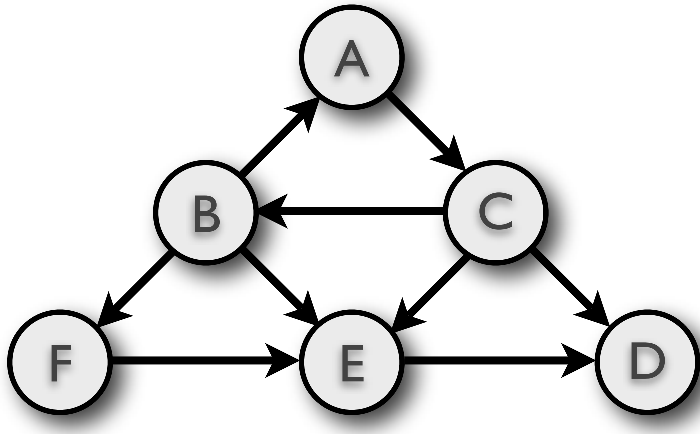
    push u onto S

  else

    pop v from S

    mark u as "visited"





proc DFS(u):

// recursive version of DFS

mark *u* as "visiting"

for each unvisited neighbor v of u

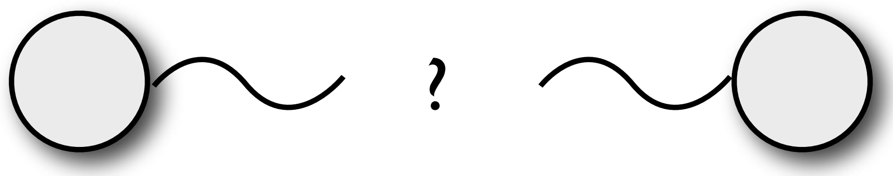
    DFS(v)

mark *u* as "visited"

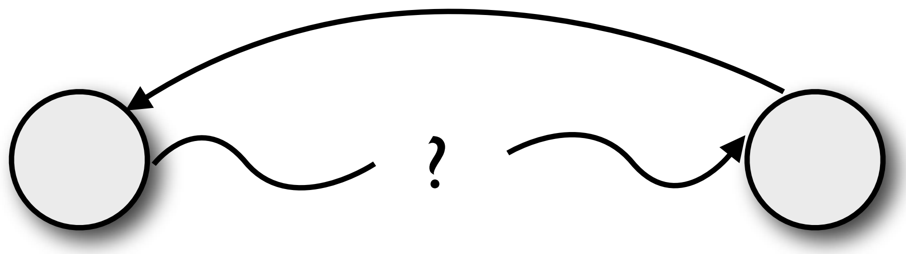
Review Questions

- True/False? : There is always a path from the vertices in the stack to the vertex at the top of the stack.
- (Alternatively: There is always a path from a vertex marked “visiting” to the current vertex.)

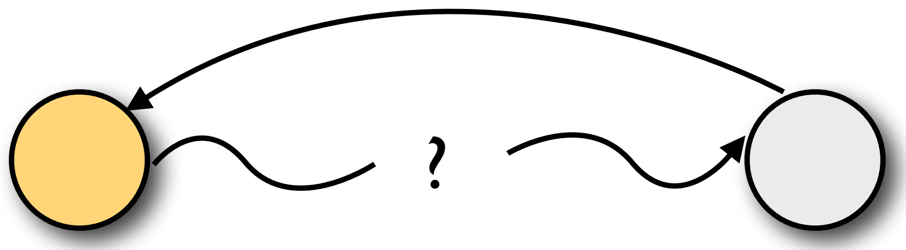
Applications of DFS



We can check if two vertices are connected using DFS.



We can check if a graph is acyclic/cyclic using DFS.



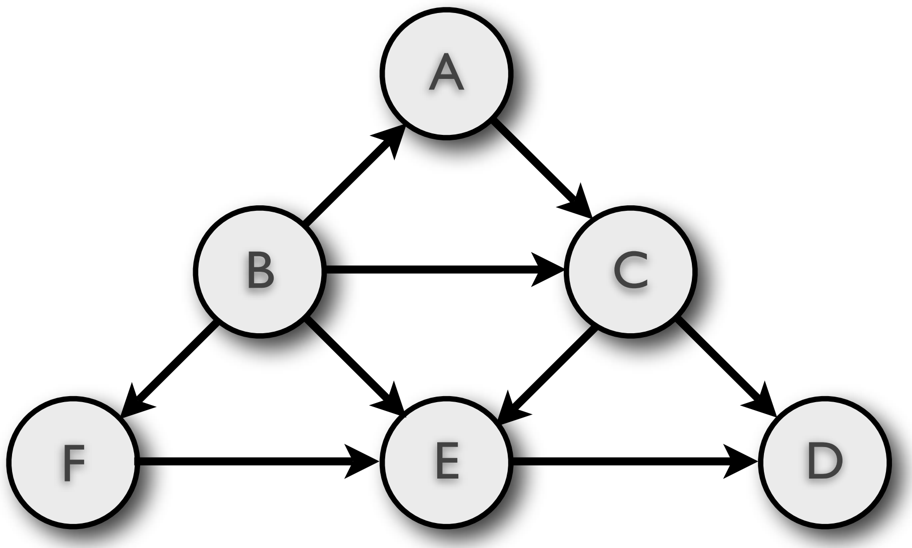
There is a cycle iff we found an edge from current vertex to a visiting vertex (called backward edge)

```
proc DFS(u):

mark u as "visiting"
for each neighbor v of u
  if v is marked as "visiting"
    we found a cycle!
  else if v is marked as "unvisited"
    DFS(v)
mark u as "visited"
```

Topological Sort

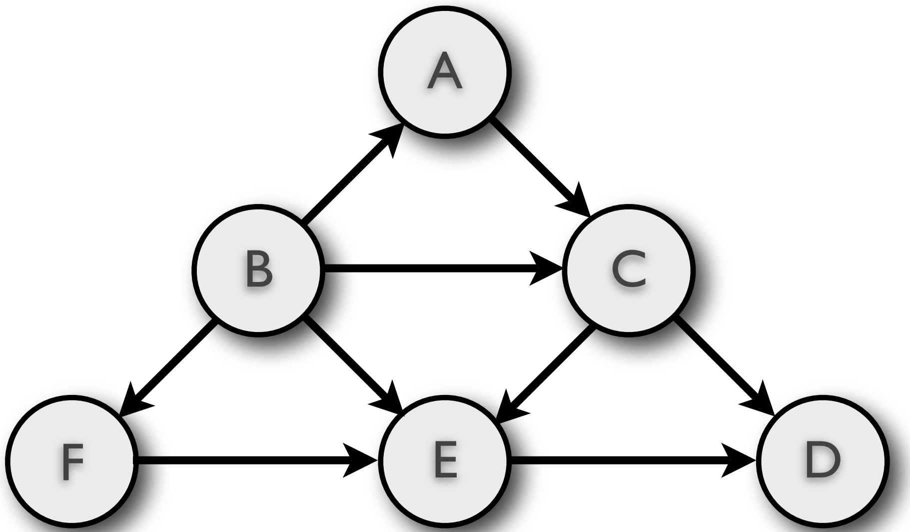
Goal: Given a directed acyclic graph, order the vertices such that if there is a path from *u* to *v*, then *u* appears before *v* in the output.

Goal: Given a directed acyclic graph, order the vertices such that if there is a path from *u* to *v*, then *u* appears before *v* in the output.

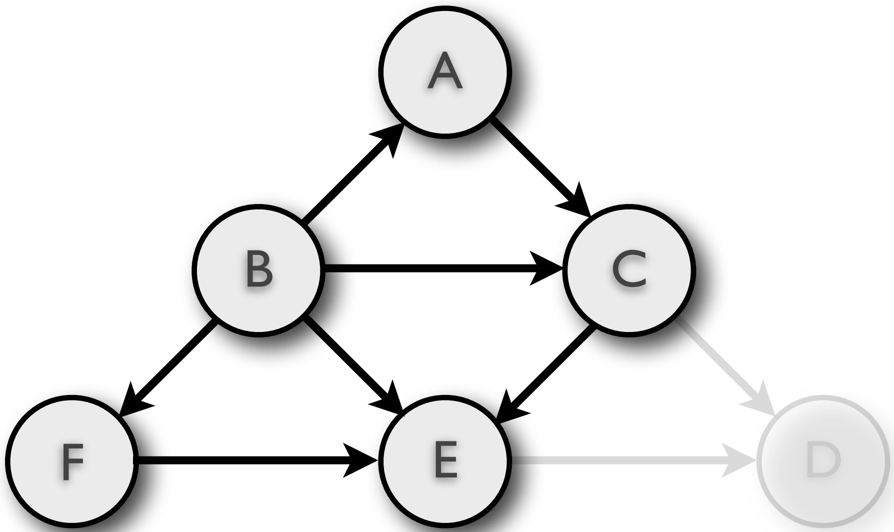


BACFED?
BCAFED?
BFACED?

Idea: The first vertex marked "visited" can appear last in the topological order.



Now, we remove that vertex from consideration, and repeat -- the next vertex marked as visited can appear last in the topological sort order.



proc DFS(u):

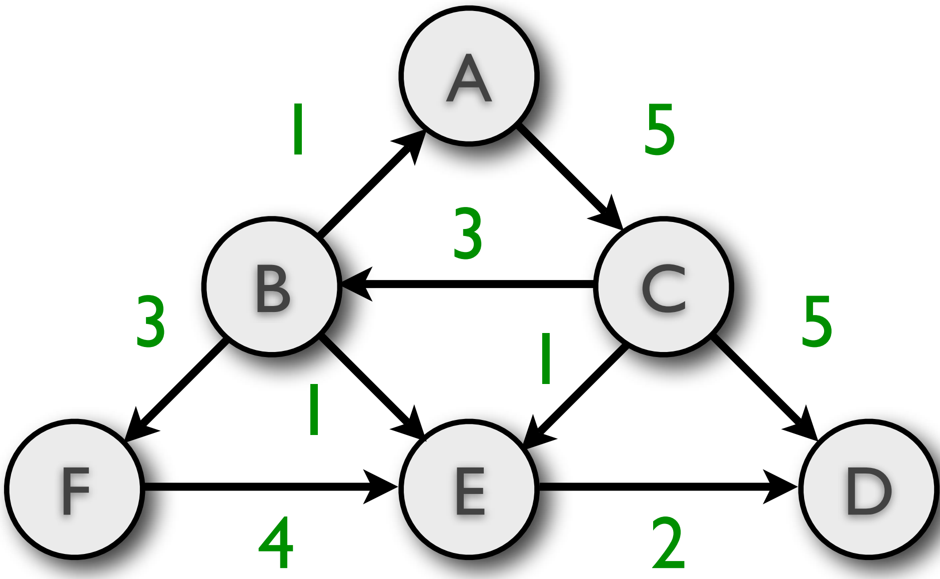
for each unvisited neighbor v of u
    DFS(v)
push u onto a stack

To output in topological sort order, pop from stack and print after completing DFS.

Dijkstra’s Algorithm

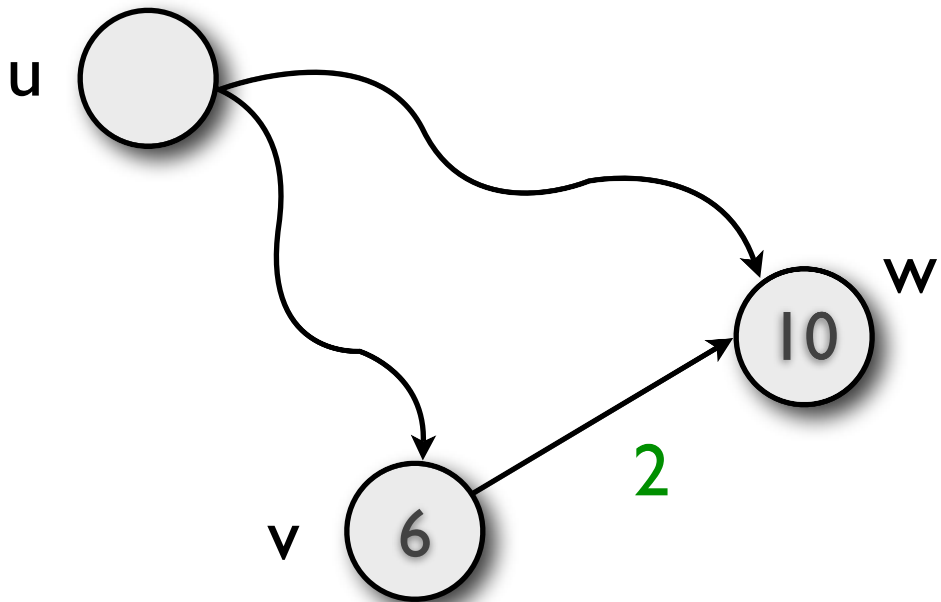
Single-Source Shortest Path

- **Problem**: Given a weighted graph G and a vertex v in G, find the shortest (or least cost) path from v to all other vertices.
- Restrict ourselves to **positive** weight.



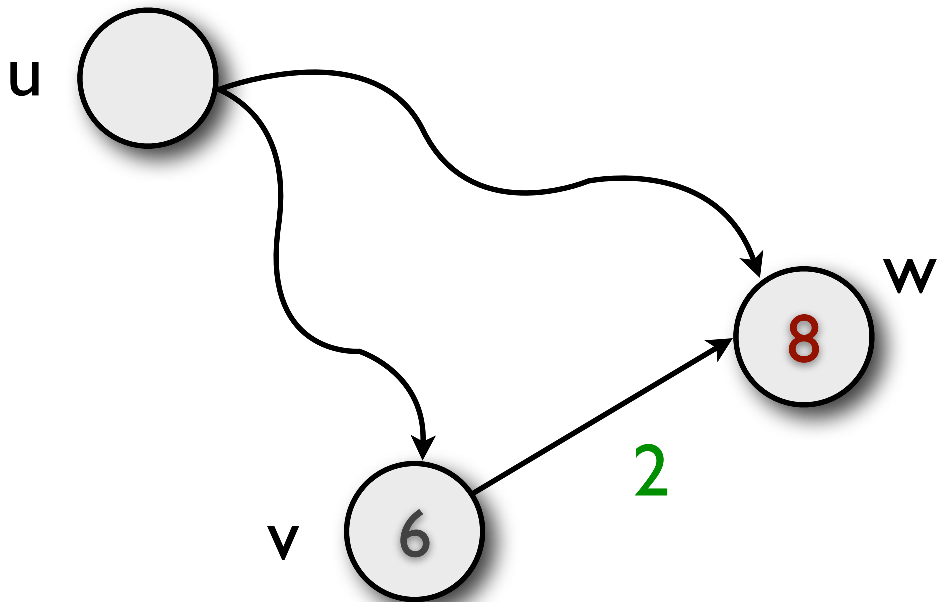
Shortest Path from A to D = A-C-E-D (Cost = 8)

- Must keep track of smallest distance so far.
- If we found a new, shorter path, update the distance.



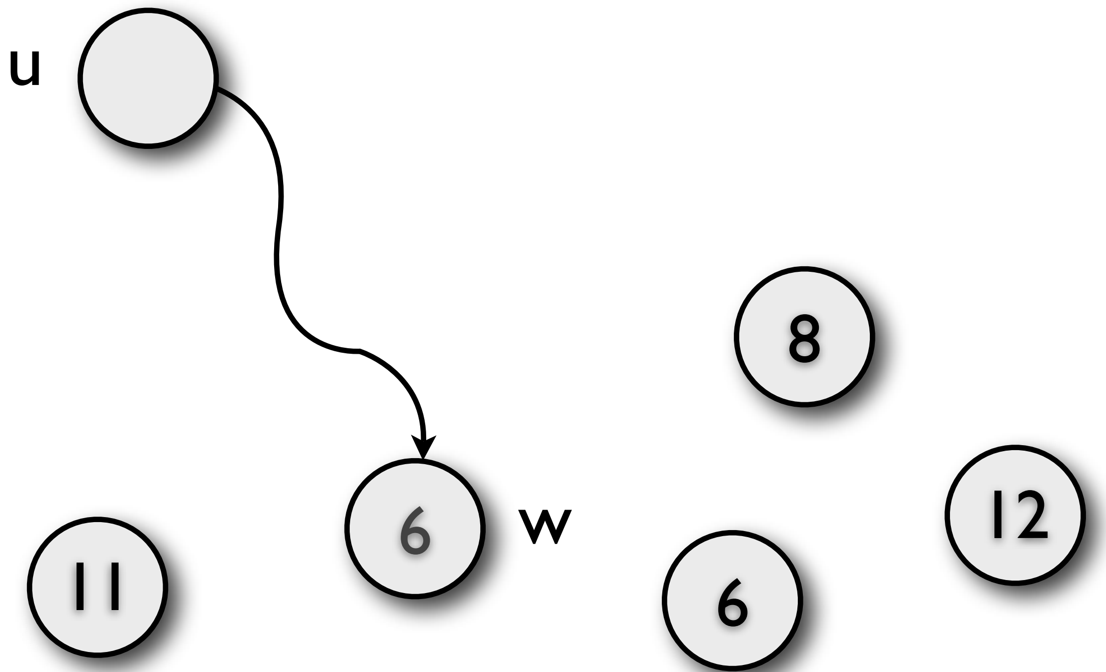
Let d[v] be the current known shortest distance from u to v.

d[v] = 6, d[w] = 10

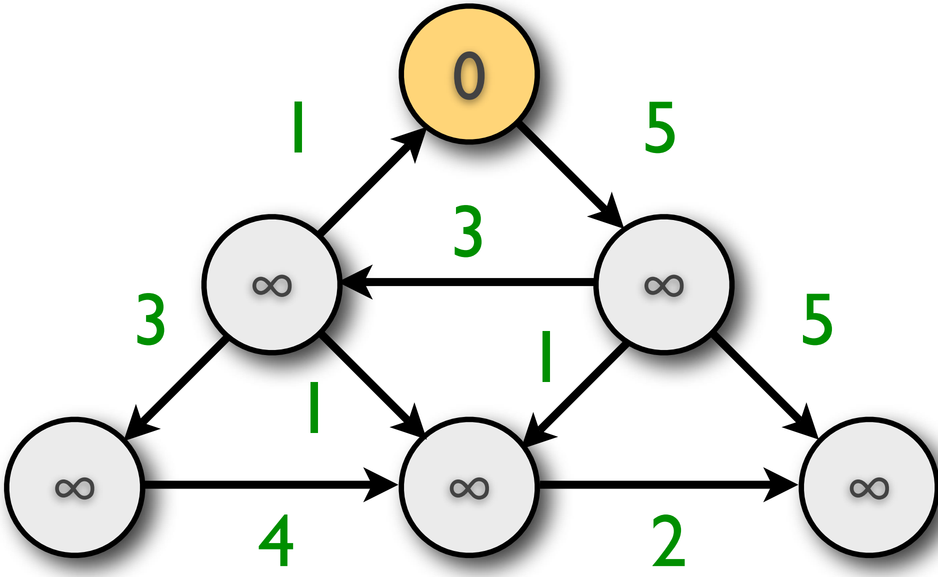


We just found a shorter path from u to w.
Update d[w] = d[v] + cost(v,w).
We call this step relax(v,w).

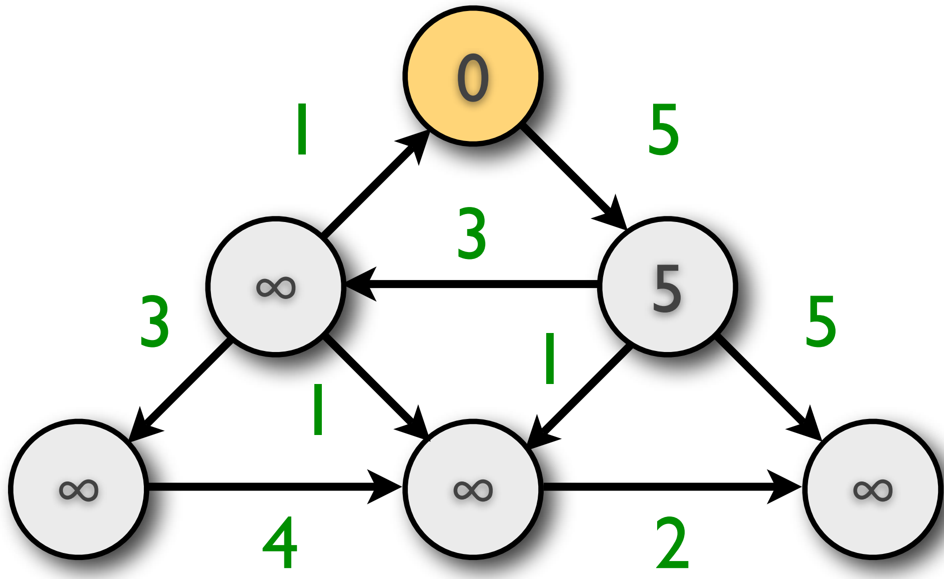
```
proc relax (v,w):

   Let d = d[v] + cost(v,w)
   if  d[w] > d
       d[w] = d
```



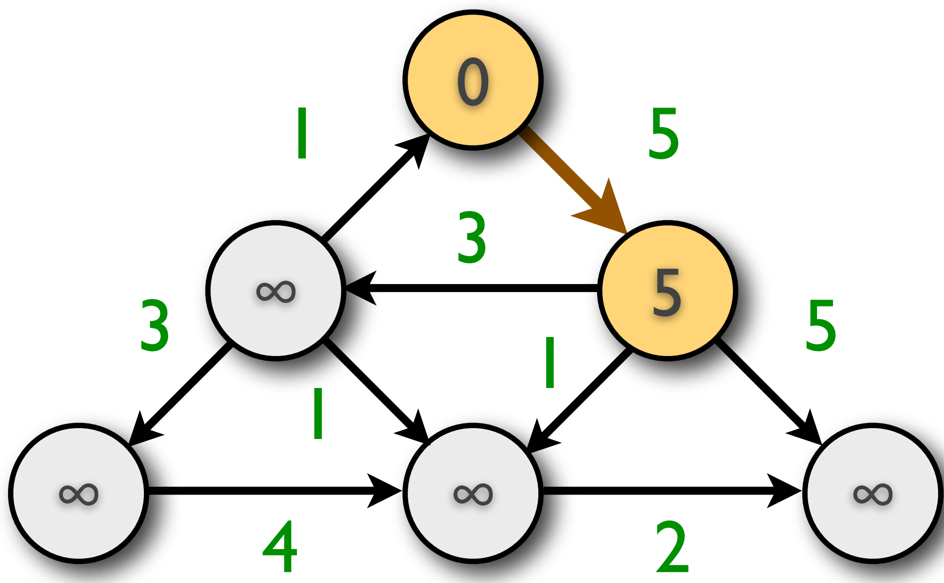
If d[w] is the smallest among the “remaining” vertices, then d[w] is the smallest possible (can’t be relaxed further)



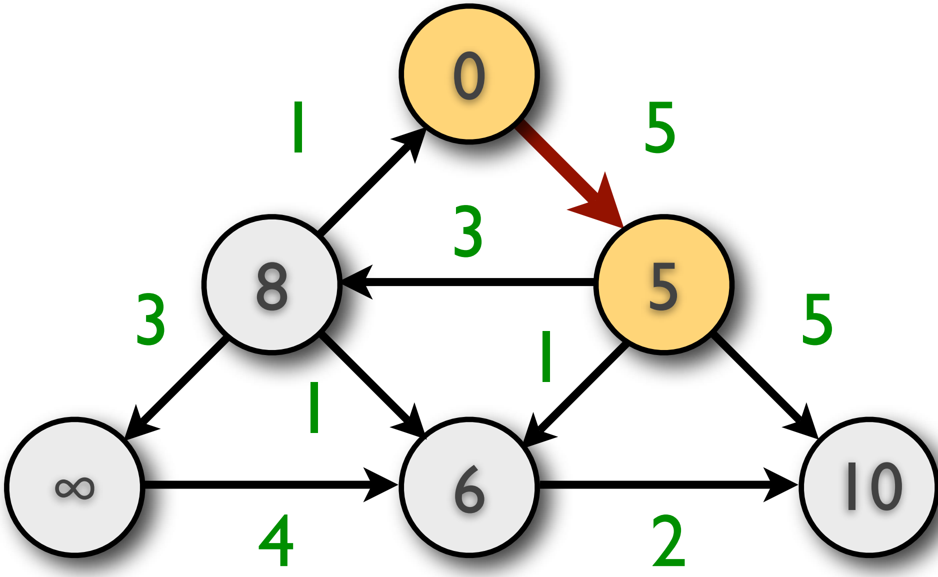
At the beginning, we know d[A]. But the rest is unknown and is set to infinity.



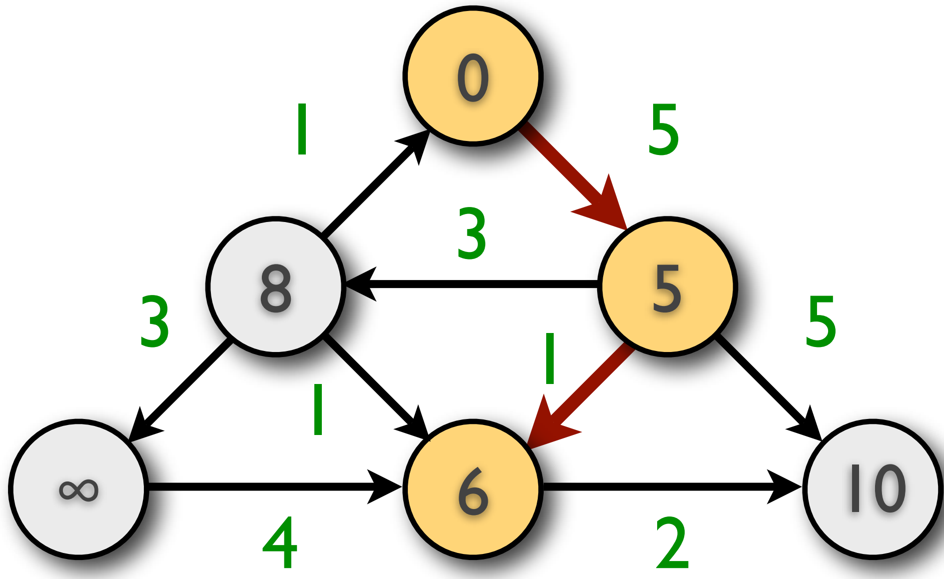
Relax all neighbors of A.



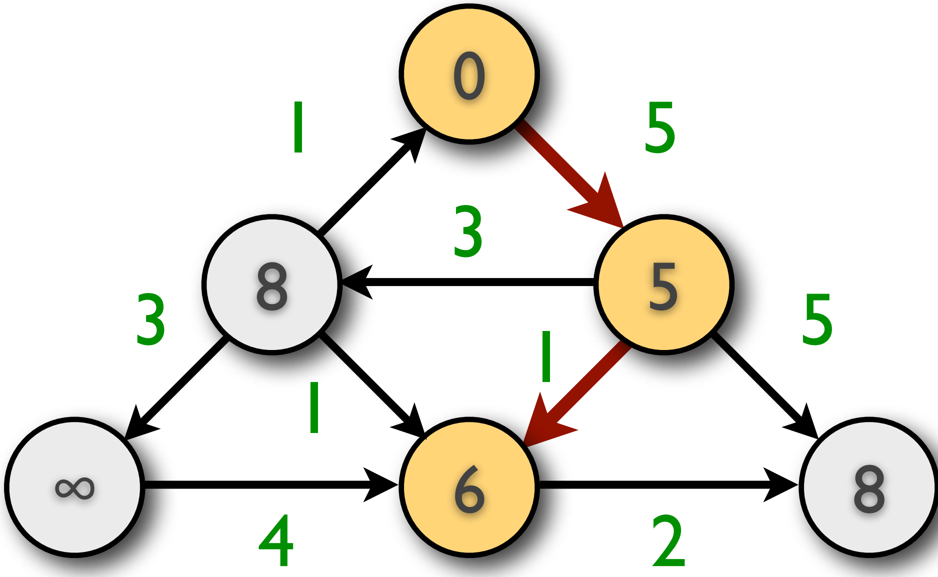
Pick a white vertex with smallest d[ ]. Color it yellow.



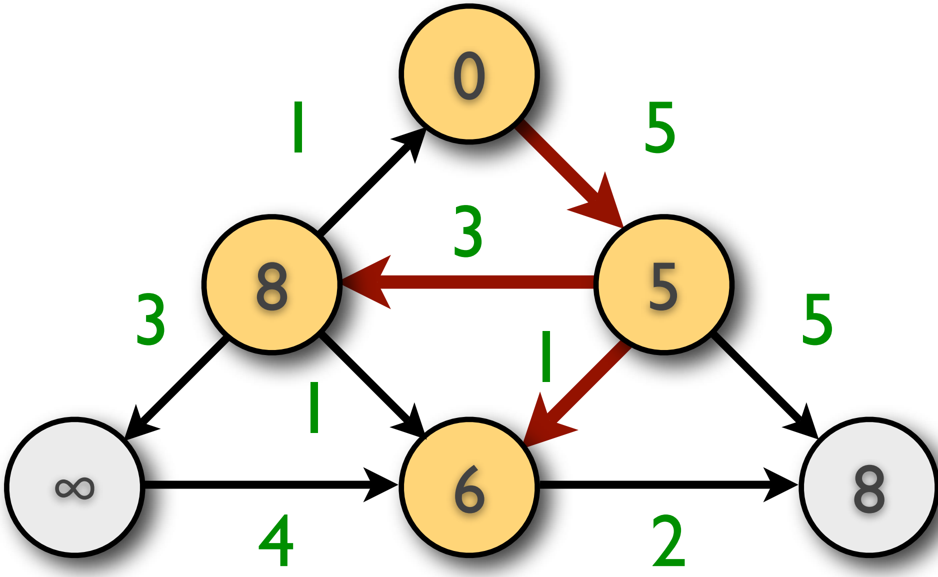
Relax all neighbors of this vertex.

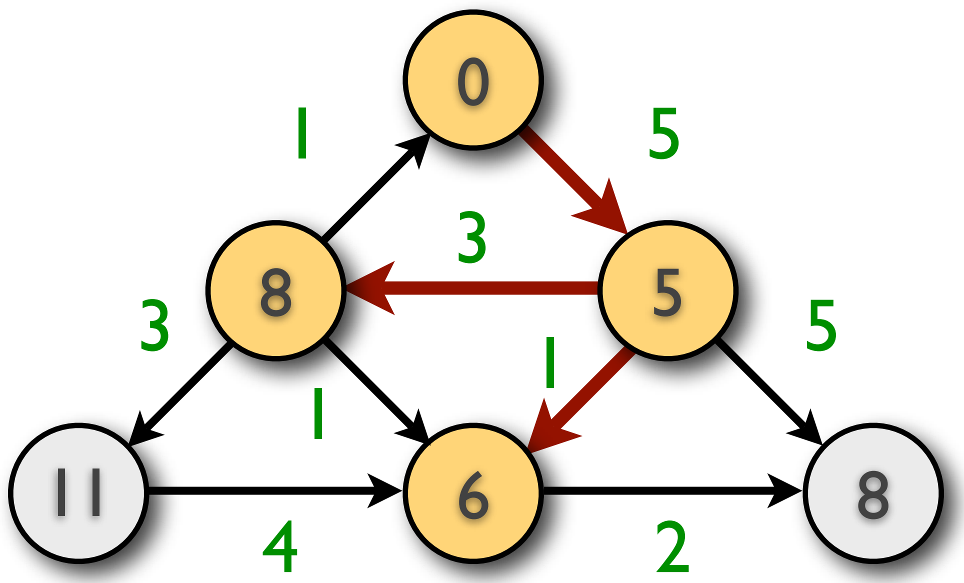


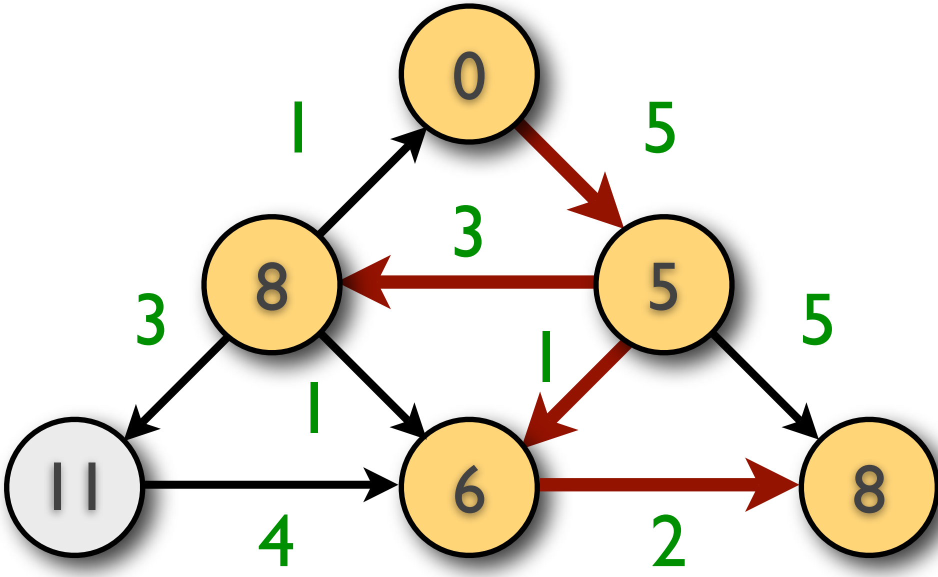
Repeat: pick a white vertex with smallest d[ ].

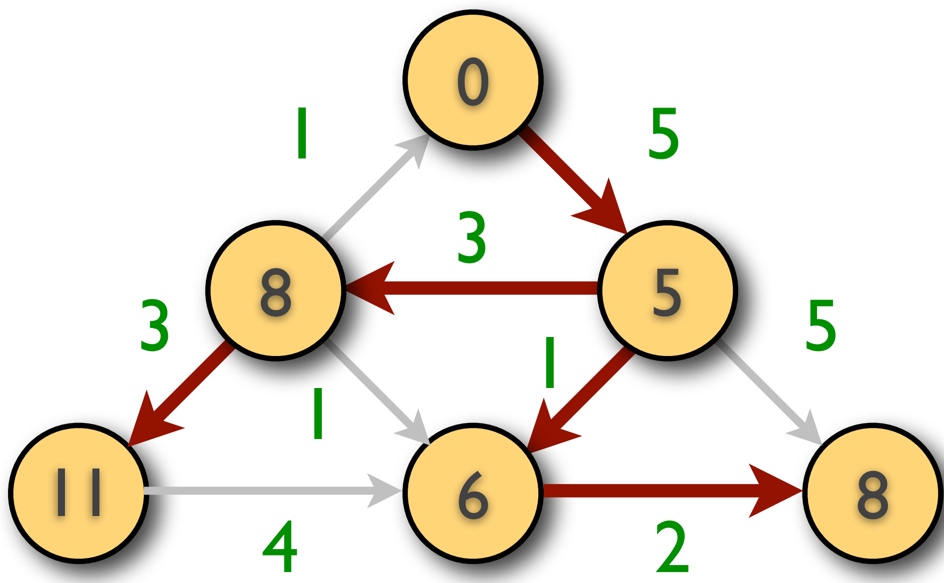


Relax its neighbors


0
8
5
∞
6
8
1
5
3
3
5
1
1
4
2


0
1
5
3
8
5
3
5
1
1
11
6
8
4
2


0
1
5
3
8
5
3
5
1
1
11
6
8
4
2



Everyone is yellow. Done!

```
proc Dijkstra(s):

for each vertex v in G
    d[w] = infinity
    color[w] = white

d[s] = 0
```

```
while there exists a white vertex

    let u be a white vertex with smallest d
    color[u] = yellow
    for each neighbor v of u
        relax(u,v)
```

Array Implementation

```
while there exists a white vertex

    min = infinity
    for each vertex v
        if color[v] is white and d[v] < min
            min = d[v]
            u = v

    color[u] = yellow
    for each neighbor v of u
```

Priority Queue Implementation

```
while there exists a white vertex
    u = q.getMin()
    color[u] = yellow
    for each neighbor v of u
        relax(u,v)
```

Priority Queue Implementation

```
proc relax (v,w):

  Let d = d[v] + cost(v,w)
  if  d[w] > d
      d[w] = d
      q.decreaseCost(w, d)
```

Summary: Graph

- Basic terms
- Representations
- Applications
- BFS
  - find shortest path in unweighted path
  - finding connected component

Summary: Graph

- DFS
  - finding connected component
  - check for cycles
  - topological sort
- Dijkstra algorithm
  - finding shortest path from a single source in a weighted graph with positive weights.