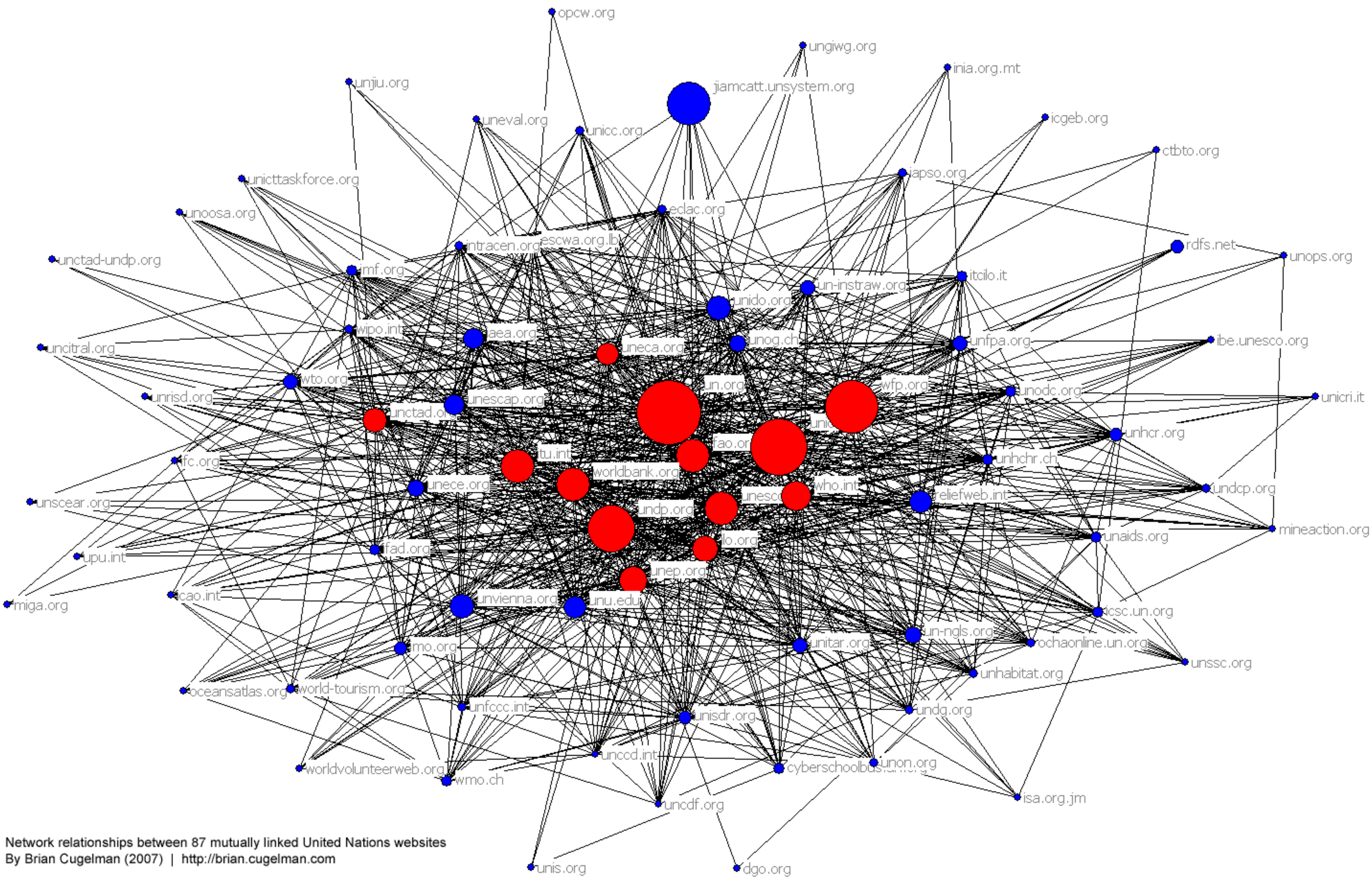
CS280-Data Structures

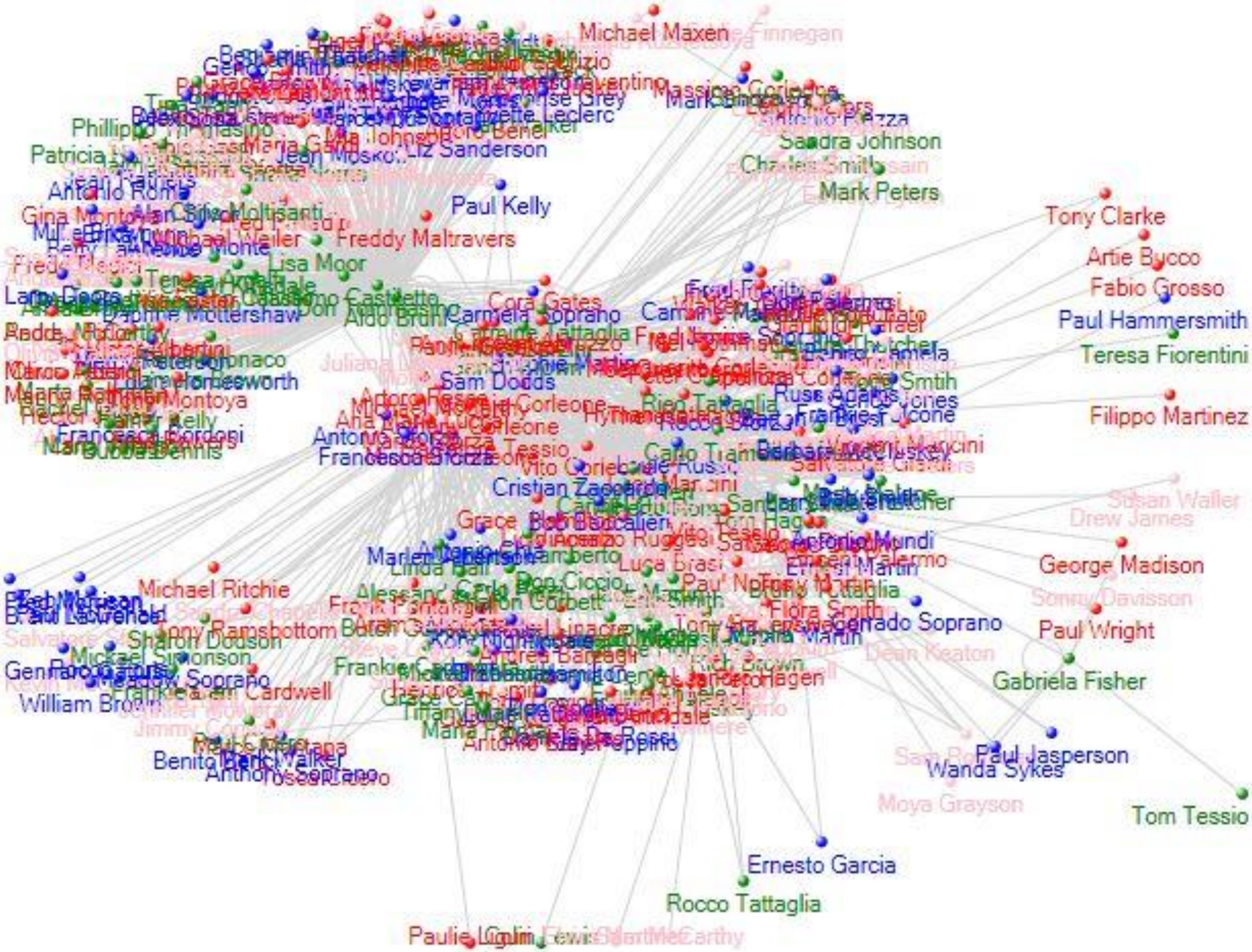
Introduction to Graphs

Internet

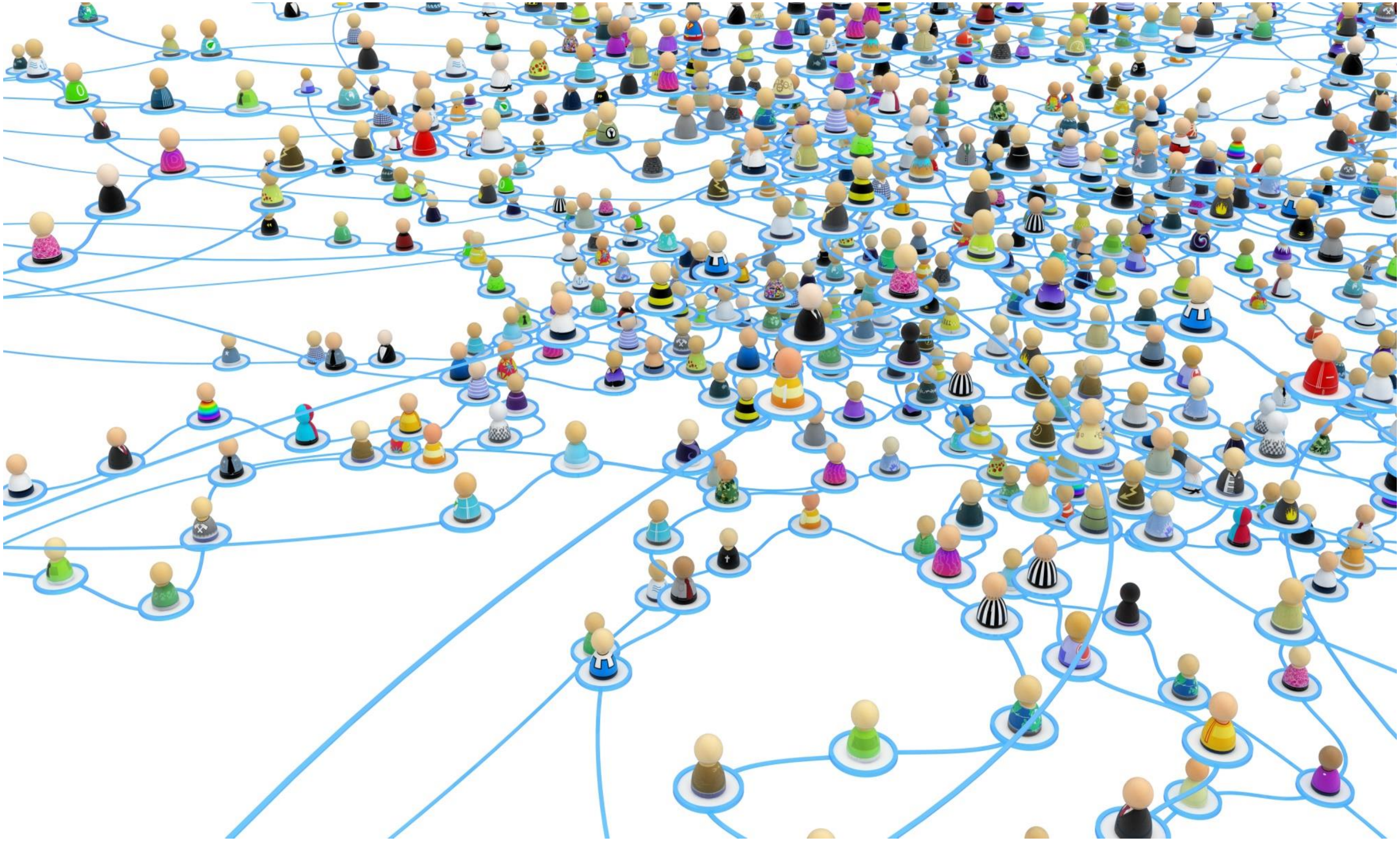


Network relationships between 87 mutually linked United Nations websites
By Brian Cugelman (2007) | http://brian.cugelman.com

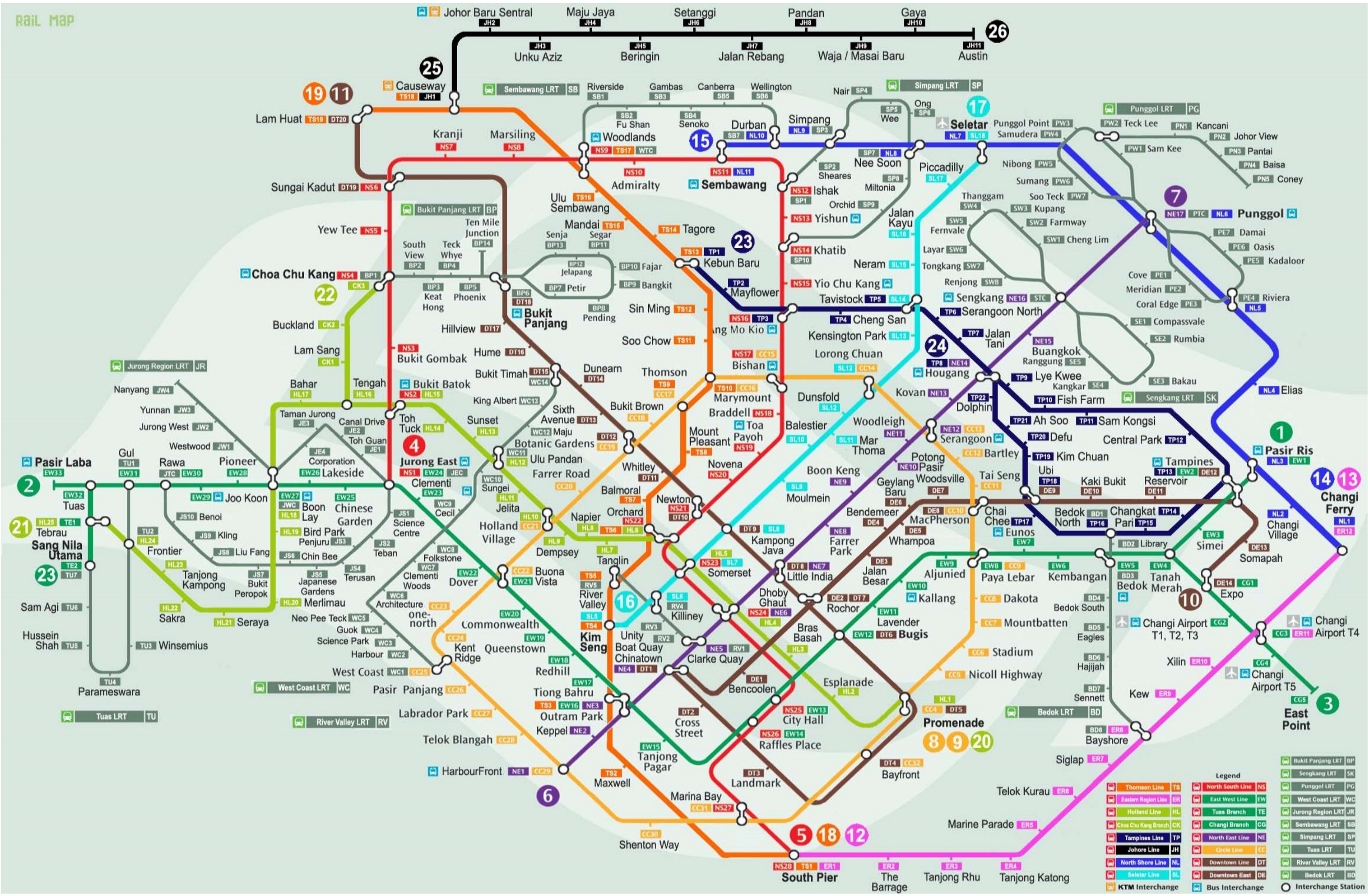
Email Networks



Social Networks

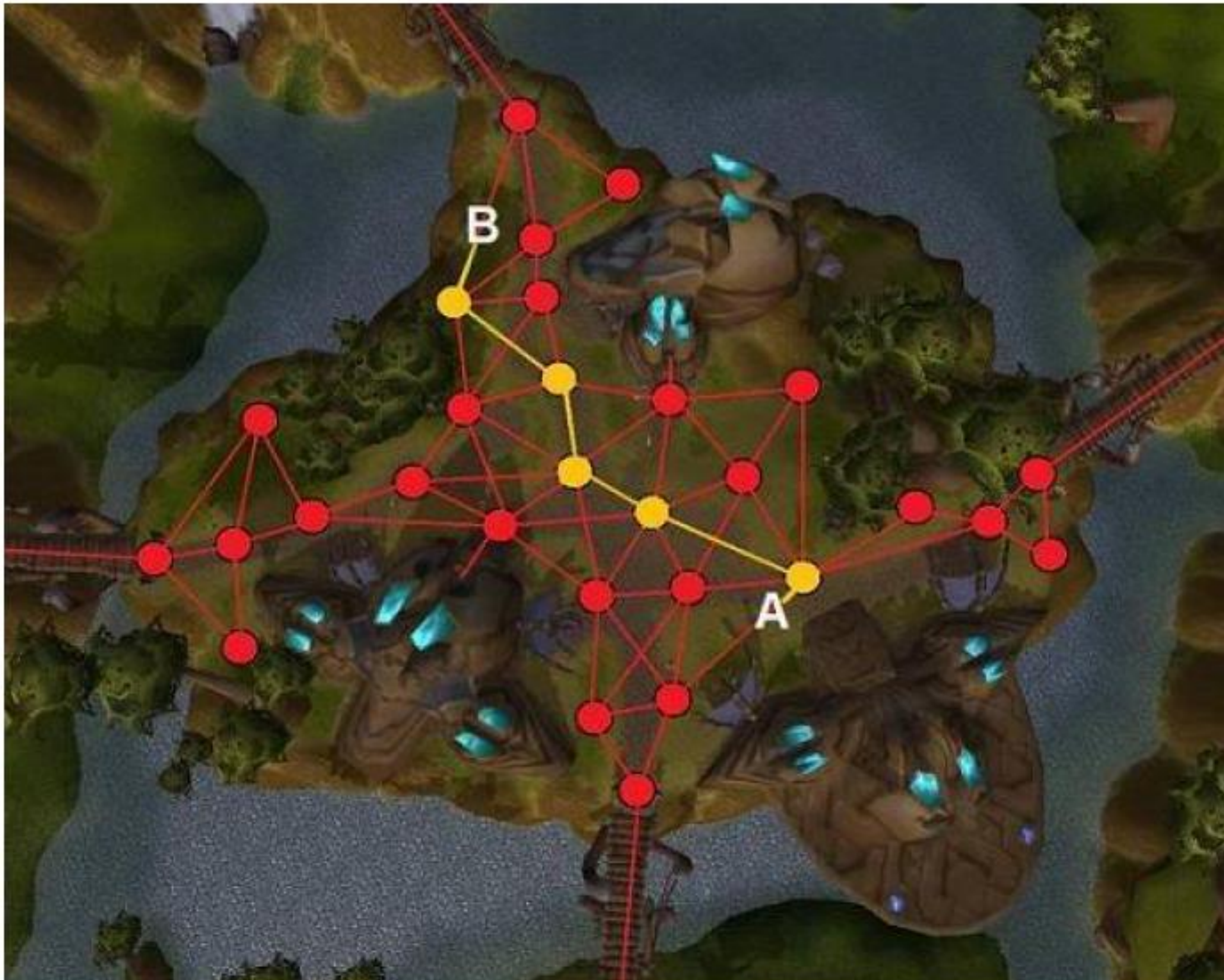


Public Transport



Video Games

Video Games



Overview

- Introduction & Terminology
- Representing Graphs
- Graph Traversals
- Spanning Trees
- Shortest Path Algorithms

Introduction & Terminology

Introduction

- One of the most useful data structures (A very large topic).
- Related to trees in that a tree is a special kind of graph (Trees are much simpler).
- Graphs are more general and have a wider range of use. (Generality trades simplicity).
- Represent problems involving interconnected (dependent) objects.
- Graph algorithms are complex. Need to account for cycles; trees have only one path between nodes.

Introduction

- One of the most useful data structures (A very large topic).
- Tree is a special kind of graph (Trees are much simpler).
- Graphs are more general and have a wider range of use. (Generality trades simplicity).
- Represent problems involving interconnected (dependent) objects.
- Graph algorithms are complex. Need to account for cycles; trees have only one path between nodes.

Introduction

- One of the most useful data structures (A very large topic).
- Related to trees in that a tree is a special kind of graph (Trees are much simpler).
- Graphs are more general and have a wider range of use. (Generality trades simplicity).
- Represent problems involving interconnected (dependent) objects.
- Graph algorithms are complex. Need to account for cycles; trees have only one path between nodes.

Introduction

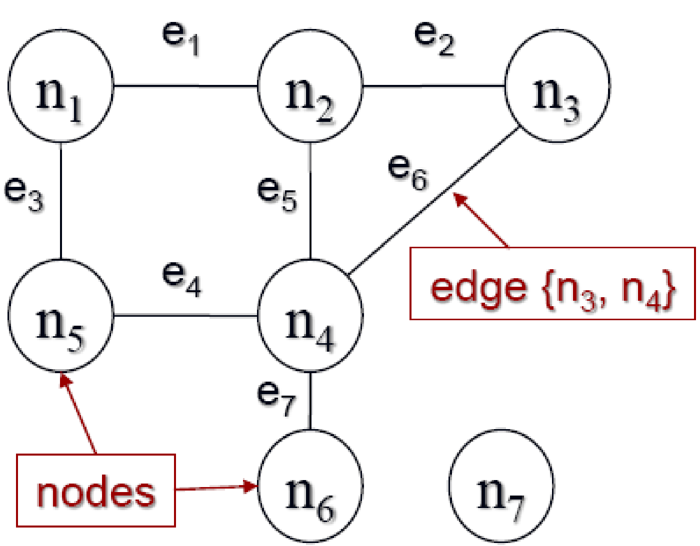
- One of the most useful data structures (A very large topic).
- Related to trees in that a tree is a special kind of graph (Trees are much simpler).
- Graphs are more general and have a wider range of use. (Generality trades simplicity).
- Represent problems involving interconnected (dependent) objects.
- Graph algorithms are complex. Need to account for cycles; trees have only one path between nodes.

Introduction

- One of the most useful data structures (A very large topic).
- Related to trees in that a tree is a special kind of graph (Trees are much simpler).
- Graphs are more general and have a wider range of use. (Generality trades simplicity).
- Represent problems involving interconnected (dependent) objects.
- Graph algorithms are complex. Need to account for cycles; trees have only one path between nodes.

Terminology

- A graph is essentially a collection of points connected by line segments.
- The points are referred to as nodes or vertices; the segments are called edges.

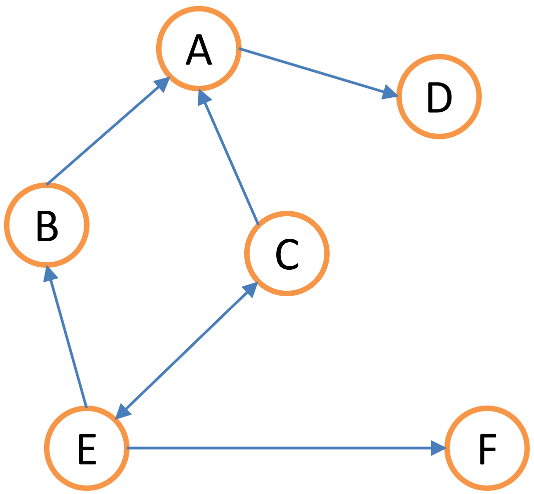


$V = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$
$= \{(n_1, n_2), (n_2, n_3), (n_1, n_5), (n_4, n_5), (n_2, n_4), (n_3, n_4), (n_4, n_6)\}$

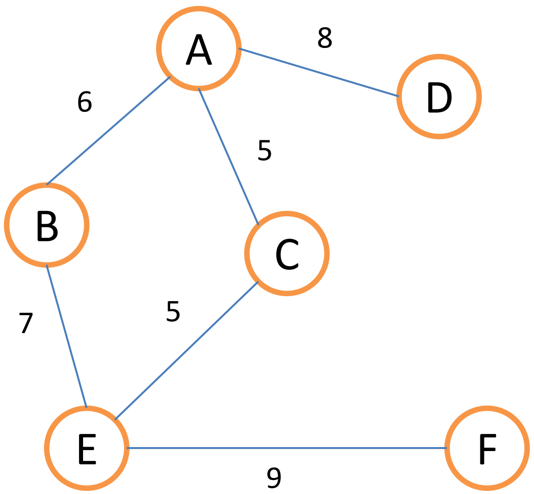
Terminology

- If the edges have a direction (arrowheads in a diagram), the graph is a directed graph, or digraph.



Terminology

- If the graph has values (weights or costs) assigned to edges is called a weighted graph.



Notation

- A graph, G, consists of a set of vertices, V and edges, E, where the edges are constructed from pairs of distinct vertices: $G(V,E)$

- In an undirected graph, each edge is an unordered pair: $e=(v_1,v_2)$

- In a directed graph, each edge is an ordered pair: $e=(v_1,v_2)$
  - v_1 is the origin (source) and v_2 is the terminus (destination).

Notation

- A graph, G, consists of a set of vertices, V and edges, E, where the edges are constructed from pairs of distinct vertices: $G(V,E)$

- In an undirected graph, each edge is an unordered pair: $e=(v_1,v_2)$

- In a directed graph, each edge is an ordered pair: $e=(v_1,v_2)$
  - v_1 is the origin (source) and v_2 is the terminus (destination).

Notation

- A graph, G, consists of a set of vertices, V and edges, E, where the edges are constructed from pairs of distinct vertices: $G(V,E)$

- In an undirected graph, each edge is an unordered pair: $e=(v_1,v_2)$

- In a directed graph, each edge is an ordered pair: $e=(v_1,v_2)$
  - v_1 is the origin (source) and v_2 is the terminus (destination).

Notation

- Two vertices, x and y are said to be adjacent if there is an edge connecting them.

- We use the notation sGd to mean that *s* is adjacent to d. With a digraph, sGd implies direction. (xGy is not the same as yGx).

- The set of nodes adjacent to s is called the adjacency set of s or neighbors of s.
  - This set is fundamental to many graph algorithms.

Notation

- Two vertices, x and y are said to be adjacent if there is an edge connecting them.

- We use the notation sGd to mean that s is adjacent to d. With a digraph, sGd implies direction. (xGy is not the same as yGx).

- The set of nodes adjacent to s is called the adjacency set of s or neighbors of s.
  - This set is fundamental to many graph algorithms.

Notation

- Two vertices, x and y are said to be adjacent if there is an edge connecting them.

- We use the notation sGd to mean that s is adjacent to d. With a digraph, sGd implies direction. (xGy is not the same as yGx).

- The set of nodes adjacent to s is called the adjacency set of s or neighbors of s.
  - This set is fundamental to many graph algorithms.

Paths and Connectivity

- A (contiguous) sequence of edges is a path.
- If there is a path from x to y, y is reachable from x.
- The length of a path is the number of edges on the path.
- Two vertices are connected if there is a path from one to the other.
- A connected component is a subset, S, of vertices that are all connected.

Paths and Connectivity

- A (contiguous) sequence of edges is a path.
- If there is a path from x to y, y is reachable from x.
- The length of a path is the number of edges on the path.
- Two vertices are connected if there is a path from one to the other.
- A connected component is a subset, S, of vertices that are all connected.

Paths and Connectivity

- A (contiguous) sequence of edges is a path.
- If there is a path from x to y, y is reachable from x.
- The length of a path is the number of edges on the path.
- Two vertices are connected if there is a path from one to the other.
- A connected component is a subset, S, of vertices that are all connected.

Paths and Connectivity

- A (contiguous) sequence of edges is a path.
- If there is a path from x to y, y is reachable from x.
- The length of a path is the number of edges on the path.
- Two vertices are connected if there is a path from one to the other.
- A connected component is a subset, S, of vertices that are all connected.

Connectedness

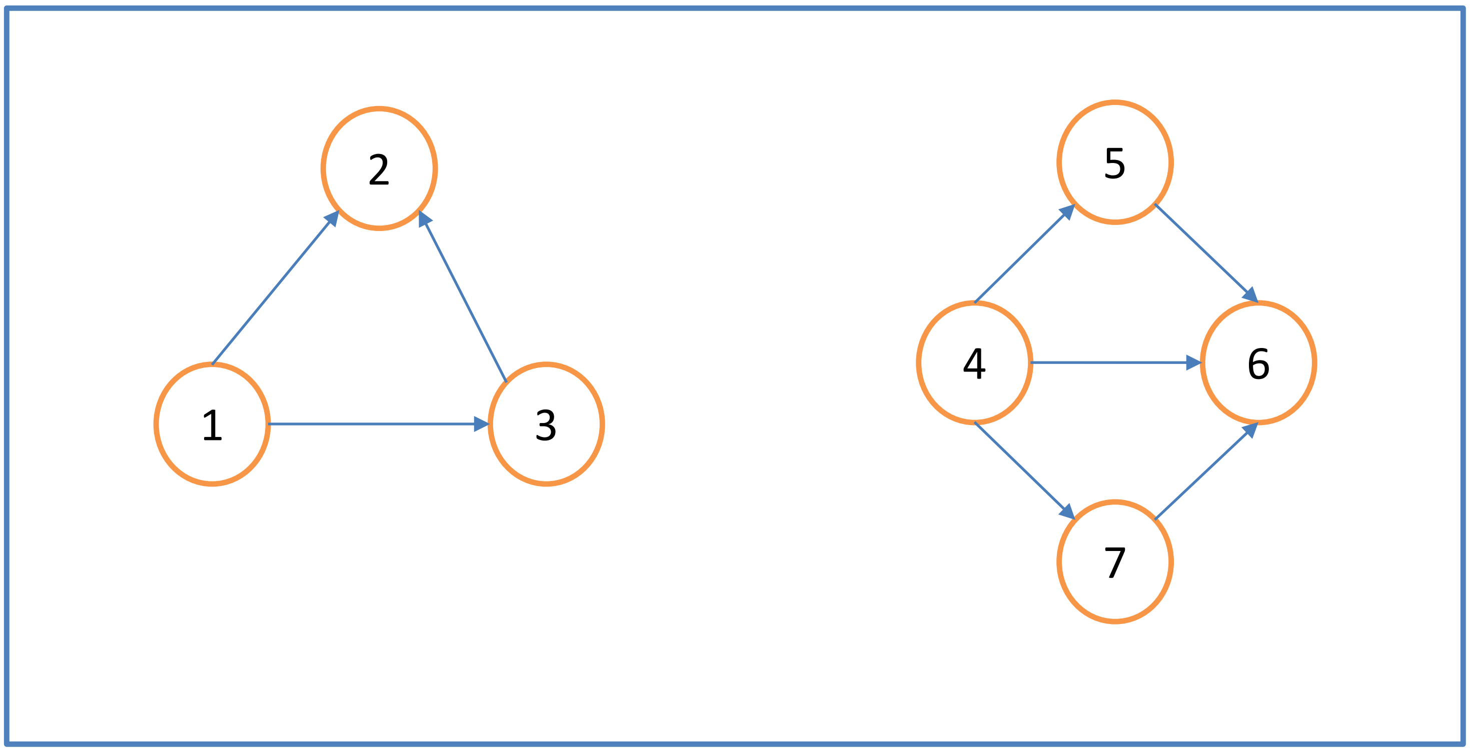
- Connectedness is an equivalence relation on the node set of a graph
  - Reflexive: every node is in a path of length 0 with itself
  - Symmetric: if $(n_i, n_j) \in$ path, then $(n_j, n_i) \in$ path
  - Transitive: if $(n_i, n_j) \in$ path and $(n_j, n_w) \in$ path, then $(n_i, n_w) \in$ path.

Paths and Connectivity

- A (contiguous) sequence of edges is a path.
- If there is a path from x to y, y is reachable from x.
- The length of a path is the number of edges on the path.
- Two vertices are connected if there is a path from one to the other.
- A connected component is a subset, S, of vertices that are all connected.

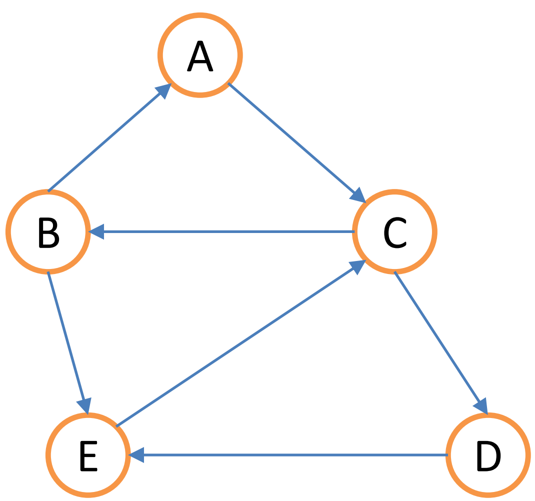
Connected Component: Example

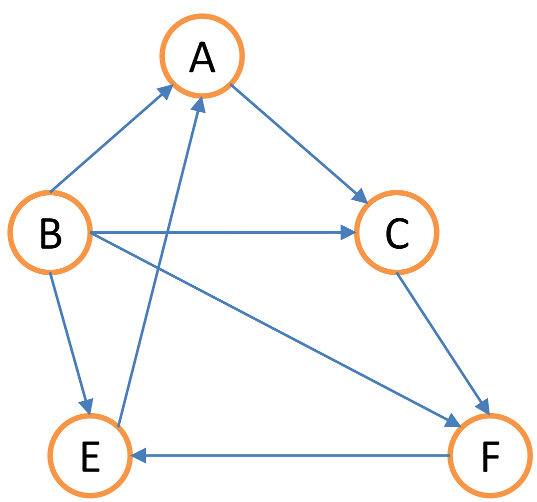
- A single directed graph with two components:



Connection Types

- ohs can be strongly connected or weakly cted.
  - ngly connected: There is a path from every node to ry other node
  - akly connected: There is **NOT** a path from each node to ry other node (see Node C)





Cycles

- A cycle is a path whose source and destination node are the same.
- A cycle is simple if all nodes on the path are distinct (with the exception of the first and last). A simple cycle must include at least 3 vertices.
- Another way of describing a simple cycle: When you travel around a loop in a simple cycle, you must visit at least three different vertices and you must visit each vertex only once.
- Think of a "Figure 8" as being a non-simple cycle. (The middle vertex is visited twice.)
- If a graph has no cycles, it is acyclic. A directed acyclic graph is called a DAG.

Cycles

- A cycle is a path whose source and destination node are the same.
- A cycle is simple if all nodes on the path are distinct (with the exception of the first and last). A simple cycle must include at least 3 vertices.
- Another way of describing a simple cycle: When you travel around a loop in a simple cycle, you must visit at least three different vertices and you must visit each vertex only once.
- Think of a "Figure 8" as being a non-simple cycle. (The middle vertex is visited twice.)
- If a graph has no cycles, it is acyclic. A directed acyclic graph is called a DAG.

Cycles

- A cycle is a path whose source and destination node are the same.
- A cycle is simple if all nodes on the path are distinct (with the exception of the first and last). A simple cycle must include at least 3 vertices.
- Another way of describing a simple cycle: When you travel around a loop in a simple cycle, you must visit at least three different vertices and you must visit each vertex only once.
- Think of a “Figure 8” as being a non-simple cycle. (The middle vertex is visited twice.)
- If a graph has no cycles, it is acyclic. A directed acyclic graph is called a DAG.

Cycles

- A cycle is a path whose source and destination node are the same.
- A cycle is simple if all nodes on the path are distinct (with the exception of the first and last). A simple cycle must include at least 3 vertices.
- Another way of describing a simple cycle: When you travel around a loop in a simple cycle, you must visit at least three different vertices and you must visit each vertex only once.
- Think of a “Figure 8” as being a non-simple cycle. (The middle vertex is visited twice.)
- If a graph has no cycles, it is acyclic. A directed acyclic graph is called a DAG.

Cycles

- A cycle is a path whose source and destination node are the same.
- A cycle is simple if all nodes on the path are distinct (with the exception of the first and last). A simple cycle must include at least 3 vertices.
- Another way of describing a simple cycle: When you travel around a loop in a simple cycle, you must visit at least three different vertices and you must visit each vertex only once.
- Think of a “Figure 8” as being a non-simple cycle. (The middle vertex is visited twice.)
- If a graph has no cycles, it is acyclic. A directed acyclic graph is called a DAG.

Degree

- For an undirected graph, the degree is the number of edges connecting a node.

- For directed graph:
  - In-degree: Is the number of incoming edges into a node(node is a destination).
  - Out-degree: Is the number of outgoing edges from a node (node is a source).

Degree

- For an undirected graph, the degree is the number of edges connecting a node.

- For directed graph:
  - In-degree: Is the number of incoming edges into a node(node is a destination).
  - Out-degree: Is the number of outgoing edges from a node (node is a source).

Representing Graphs

Tree v.s. Graphs

- A tree is a collection of nodes. Each node can be accessed from the root.
- A graph has no “root” node so there is no logical “beginning”.
- Each node in a graph can be used as a starting point for traversals.
- With a tree, we are guaranteed to reach every node by starting from the root.
- With a graph, there is no guarantee that we will reach any other nodes from any particular node.
- Because of these differences, the data structures representing the structures are quite different.

Tree v.s. Graphs

- A tree is a collection of nodes. Each node can be accessed from the root.
- A graph has no "root" node so there is no logical "beginning".
- Each node in a graph can be used as a starting point for traversals.
- With a tree, we are guaranteed to reach every node by starting from the root.
- With a graph, there is no guarantee that we will reach any other nodes from any particular node.
- Because of these differences, the data structures representing the structures are quite different.

Tree v.s. Graphs

- A tree is a collection of nodes. Each node can be accessed from the root.
- A graph has no "root" node so there is no logical "beginning".
- Each node in a graph can be used as a starting point for traversals.
- With a tree, we are guaranteed to reach every node by starting from the root.
- With a graph, there is no guarantee that we will reach any other nodes from any particular node.
- Because of these differences, the data structures representing the structures are quite different.

Tree v.s. Graphs

- A tree is a collection of nodes. Each node can be accessed from the root.
- A graph has no "root" node so there is no logical "beginning".
- Each node in a graph can be used as a starting point for traversals.
- With a tree, we are guaranteed to reach every node by starting from the root.
- With a graph, there is no guarantee that we will reach any other nodes from any particular node.
- Because of these differences, the data structures representing the structures are quite different.

Tree v.s. Graphs

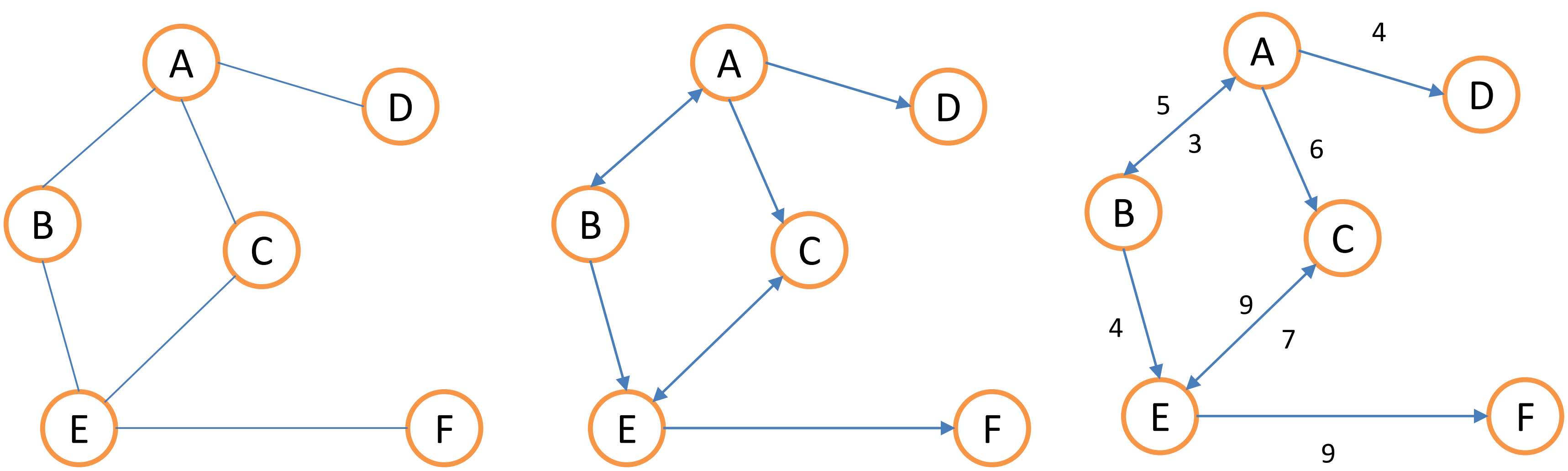
- A tree is a collection of nodes. Each node can be accessed from the root.
- A graph has no “root” node so there is no logical “beginning”.
- Each node in a graph can be used as a starting point for traversals.
- With a tree, we are guaranteed to reach every node by starting from the root.
- With a graph, there is no guarantee that we will reach any other nodes from any particular node.
- Because of these differences, the data structures representing the structures are quite different.

Tree v.s. Graphs

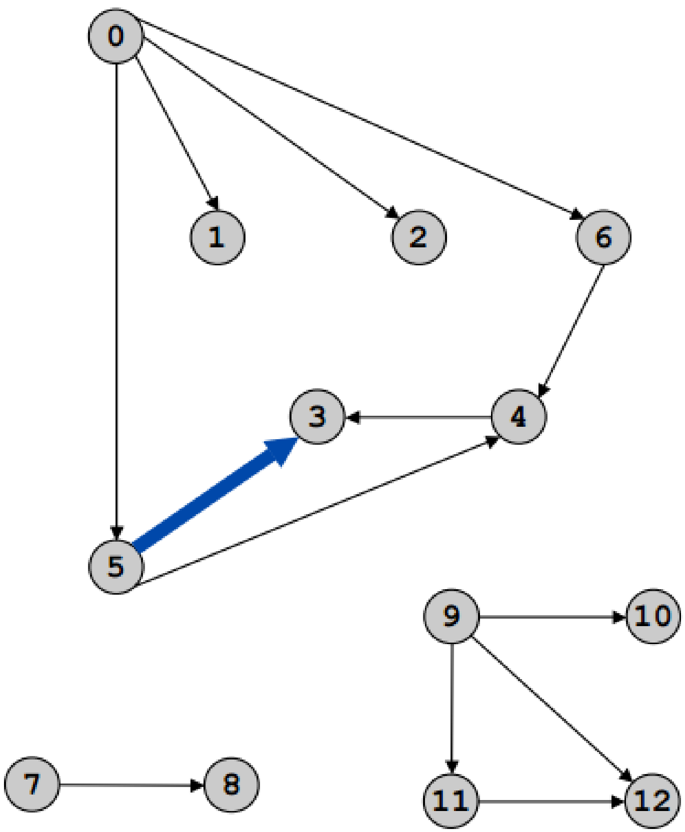
- A tree is a collection of nodes. Each node can be accessed from the root.
- A graph has no "root" node so there is no logical "beginning".
- Each node in a graph can be used as a starting point for traversals.
- With a tree, we are guaranteed to reach every node by starting from the root.
- With a graph, there is no guarantee that we will reach any other nodes from any particular node.
- Because of these differences, the data structures representing the structures are quite different.

Adjacency Matrix

- A graph G with N nodes represented by an $\mathrm{N} \times \mathrm{N}$ boolean array (matrix).
- For each x and y, $\mathrm{G(x,y)} = \mathrm{TRUE}$ if xGy, otherwise false.



Adjacency Matrix - Example



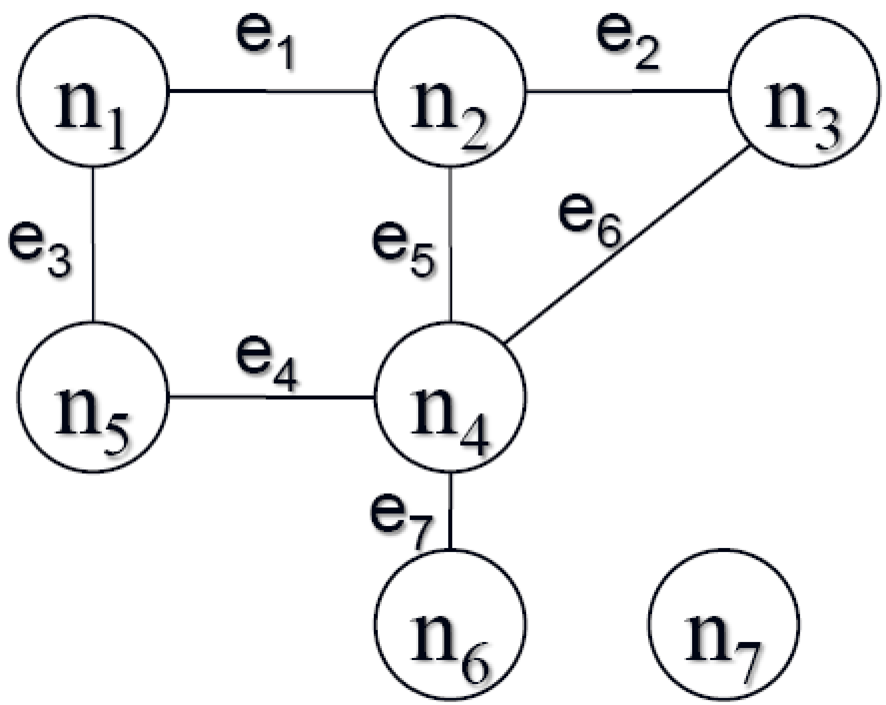
to

from

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	0	0	0

Practice

- Represent the below graph in an adjacency matrix.



Adjacency Matrix

- Space required is $O(N^2)$
  - A sparse graph has few edges
    - Sparse graphs will have many matrix entries of 0.
  - a dense graph has many edges.
    - Dense graphs will have many matrix entries of 1.
- Determining if two nodes are adjacent is $O(1)$
- The size of the matrix is **independent** of the number of edges.
- An adjacency matrix may be a more desirable representation for dense graphs.

Adjacency Matrix

- Space required is $O(N^2)$
  - A sparse graph has few edges
    - Sparse graphs will have many matrix entries of 0.
  - a dense graph has many edges.
    - Dense graphs will have many matrix entries of 1.
- Determining if two nodes are adjacent is $O(1)$
- The size of the matrix is **independent** of the number of edges.
- An adjacency matrix may be a more desirable representation for dense graphs.

Adjacency Matrix

- Space required is $O(N^2)$
  - A sparse graph has few edges
    - Sparse graphs will have many matrix entries of 0.
  - a dense graph has many edges.
    - Dense graphs will have many matrix entries of 1.
- Determining if two nodes are adjacent is $O(1)$
- The size of the matrix is **independent** of the number of edges.
- An adjacency matrix may be a more desirable representation for dense graphs.

Adjacency Matrix

- Space required is $O(N^2)$
  - A sparse graph has few edges
    - Sparse graphs will have many matrix entries of 0.
  - a dense graph has many edges.
    - Dense graphs will have many matrix entries of 1.
- Determining if two nodes are adjacent is $O(1)$
- The size of the matrix is **independent** of the number of edges.
- An adjacency matrix may be a more desirable representation for dense graphs.

Adjacency Matrix

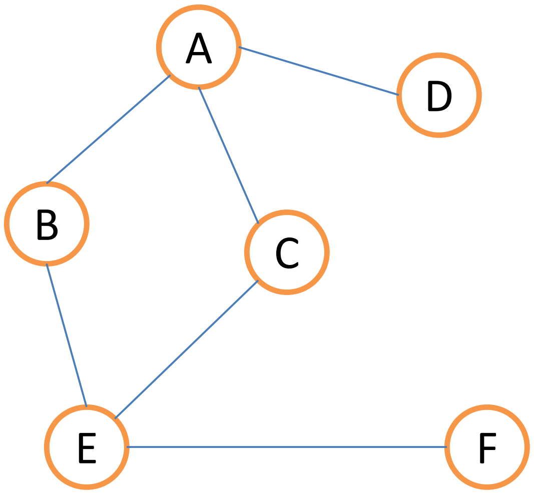
- Space required is $O(N^2)$
  - A sparse graph has few edges
    - Sparse graphs will have many matrix entries of 0.
  - a dense graph has many edges.
    - Dense graphs will have many matrix entries of 1.
- Determining if two nodes are adjacent is $O(1)$
- The size of the matrix is **independent** of the number of edges.
- An adjacency matrix may be a more desirable representation for dense graphs.

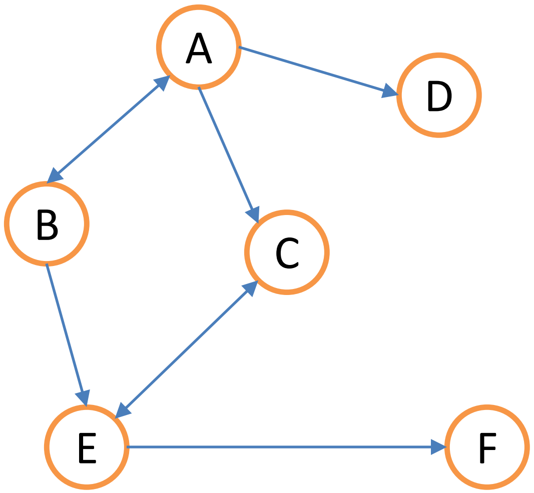
Adjacency Matrix

- Space required is $O(N^2)$
  - A sparse graph has few edges
    - Sparse graphs will have many matrix entries of 0.
  - a dense graph has many edges.
    - Dense graphs will have many matrix entries of 1.
- Determining if two nodes are adjacent is $O(1)$
- The size of the matrix is **independent** of the number of edges.
- An adjacency matrix may be a more desirable representation for dense graphs.

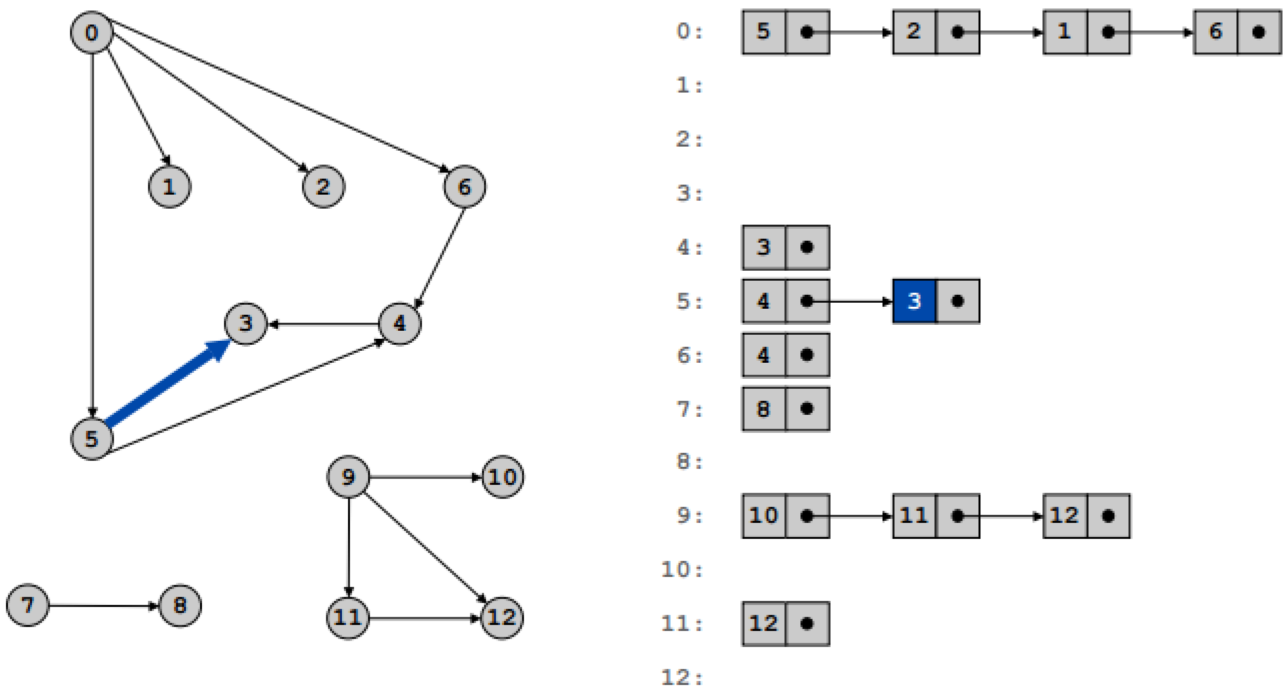
Adjacency Lists

- A graph G with N nodes represented by an array of N linked lists.
- For each x and y, if xGy is $TRUE$, y is on x's list.



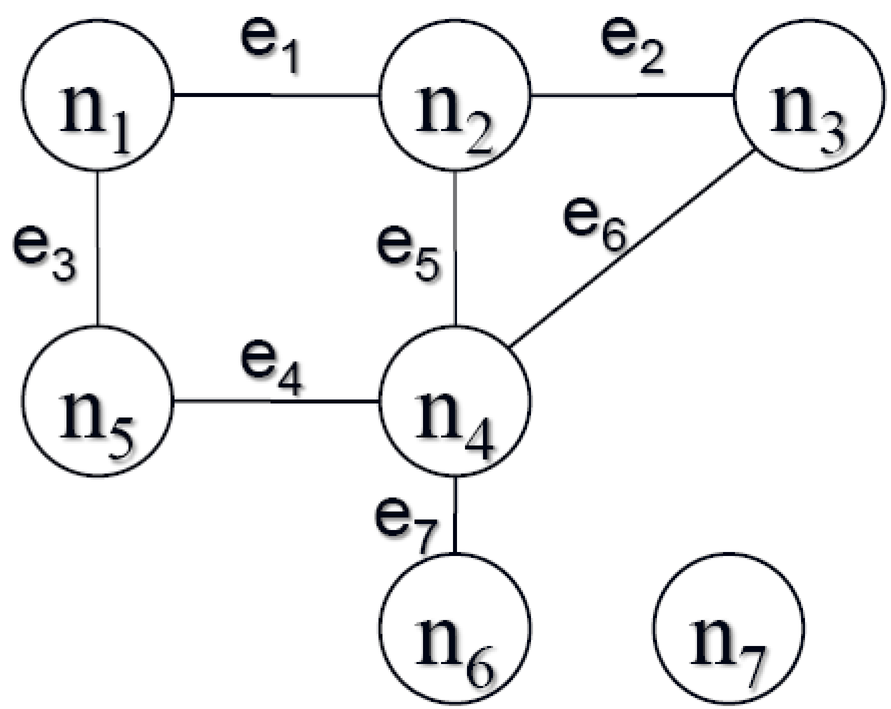


Adjacency Lists - Example



Practice

- Represent the below graph in an adjacency list.



Adjacency Lists

- Space required is $O(N^2)$
- Density affects the lists:
  - Sparse graphs will have shorter lists.
  - Dense graphs will have longer lists.
- The order of the nodes in a list may be arbitrary.
  - A weighted graph may order them by weight
- Determining if two nodes are adjacent is O(N) in the worst case. Could be much less if there are few edges.
- The number of nodes in the lists is **dependent** on the number of edges.
- An adjacency list may be a more desirable representation for sparse graphs.

Adjacency Lists

- Space required is $O(N^2)$
- Density affects the lists:
  - Sparse graphs will have shorter lists.
  - Dense graphs will have longer lists.
- The order of the nodes in a list may be arbitrary.
  - A weighted graph may order them by weight
- Determining if two nodes are adjacent is $O(N)$ in the worst case. Could be much less if there are few edges.
- The number of nodes in the lists is **dependent** on the number of edges.
- An adjacency list may be a more desirable representation for sparse graphs.

Adjacency Lists

- Space required is $O(N^2)$
- Density affects the lists:
  - Sparse graphs will have shorter lists.
  - Dense graphs will have longer lists.
- The order of the nodes in a list may be arbitrary.
  - A weighted graph may order them by weight
- Determining if two nodes are adjacent is O(N) in the worst case. Could be much less if there are few edges.
- The number of nodes in the lists is **dependent** on the number of edges.
- An adjacency list may be a more desirable representation for sparse graphs.

Adjacency Lists

- Space required is $O(N^2)$
- Density affects the lists:
  - Sparse graphs will have shorter lists.
  - Dense graphs will have longer lists.
- The order of the nodes in a list may be arbitrary.
  - A weighted graph may order them by weight
- Determining if two nodes are adjacent is $O(N)$ in the worst case. Could be much less if there are few edges.
- The number of nodes in the lists is **dependent** on the number of edges.
- An adjacency list may be a more desirable representation for sparse graphs.

Adjacency Lists

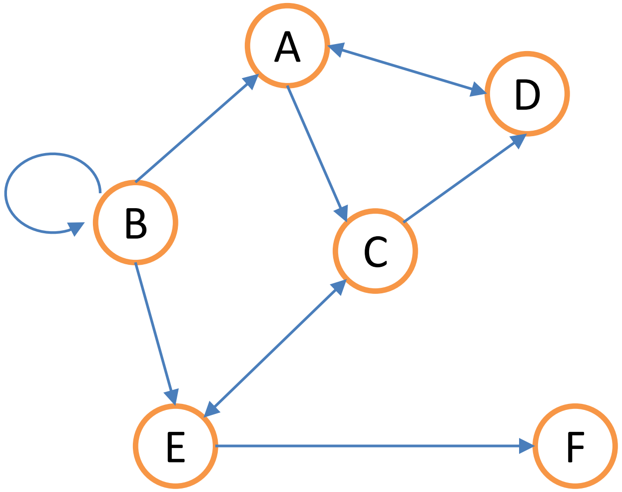
- Space required is $O(N^2)$
- Density affects the lists:
  - Sparse graphs will have shorter lists.
  - Dense graphs will have longer lists.
- The order of the nodes in a list may be arbitrary.
  - A weighted graph may order them by weight
- Determining if two nodes are adjacent is O(N) in the worst case. Could be much less if there are few edges.
- The number of nodes in the lists is **dependent** on the number of edges.
- An adjacency list may be a more desirable representation for sparse graphs.

Adjacency Lists

- Space required is $O(N^2)$
- Density affects the lists:
  - Sparse graphs will have shorter lists.
  - Dense graphs will have longer lists.
- The order of the nodes in a list may be arbitrary.
  - A weighted graph may order them by weight
- Determining if two nodes are adjacent is $O(N)$ in the worst case. Could be much less if there are few edges.
- The number of nodes in the lists is **dependent** on the number of edges.
- An adjacency list may be a more desirable representation for sparse graphs.

Exercises

- Draw the adjacency matrix and adjacency list for the following digraph:



Graph Traversals

Graph Traversals

- Unlike tree traversals, there is no "starting" (i.e. root) node in a graph.

- Choosing an arbitrary starting node will not guarantee that all nodes are visited.

- The search must systematically traverse all of the edges in order to discover all of the vertices.

- Although it sounds like a lot of redundant work, it can be accomplished in $O(N)$ time, where N is the number of vertices.

Graph Traversals

- Unlike tree traversals, there is no "starting" (i.e. root) node in a graph.

- Choosing an arbitrary starting node will not guarantee that all nodes are visited.

- The search must systematically traverse all of the edges in order to discover all of the vertices.

- Although it sounds like a lot of redundant work, it can be accomplished in $O(N)$ time, where N is the number of vertices.

Graph Traversals

- Unlike tree traversals, there is no "starting" (i.e. root) node in a graph.
- Choosing an arbitrary starting node will not guarantee that all nodes are visited.
- The search must systematically traverse all of the edges in order to discover all of the vertices.
- Although it sounds like a lot of redundant work, it can be accomplished in $O(N)$ time, where N is the number of vertices.

Graph Traversals

- Unlike tree traversals, there is no "starting" (i.e. root) node in a graph.

- Choosing an arbitrary starting node will not guarantee that all nodes are visited.

- The search must systematically traverse all of the edges in order to discover all of the vertices.

- Although it sounds like a lot of redundant work, it can be accomplished in $\mathrm{O}(\mathrm{N})$ time, where N is the number of vertices.

Graph Traversals

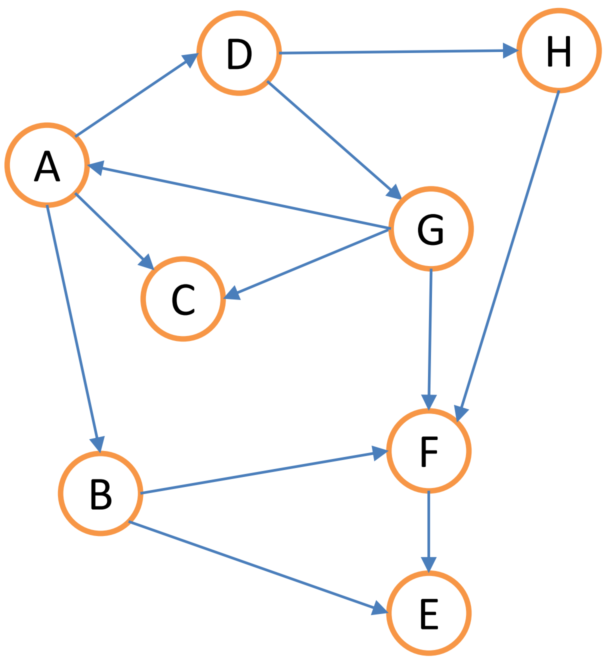
- Breath-first traversal
- Depth-first traversal

Pseudo-code for Graphs Traversals

```
GraphSearch(G is the graph to search, v is the starting vertex){
   Put v into container C;
   while (container C is not empty){
     Remove a vertex, x, from container C;
     if (x has not been visited){
       Visit x;
       Set x.visited to TRUE;
       for (each vertex, w, adjacent to x){
         if (w has not been visited)
           Put w into container C;
       }//end for
     }//end if
   }//end while
} //end GraphSearch
```

Example

- Given this graph, determine the sequence of nodes that are visited from different starting nodes. Starting at A/G, using Stack/Queue



	A	B	C	D	E	F	G	H
A	0	3	9	7	0	0	0	0
B	0	0	0	0	6	5	0	0
C	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	4	2
E	0	0	0	0	0	0	0	0
F	0	0	0	0	8	0	0	0
G	5	0	1	0	0	4	0	0
H	0	0	0	0	0	8	0	0

Adjacency Matrix

Graph Traversals

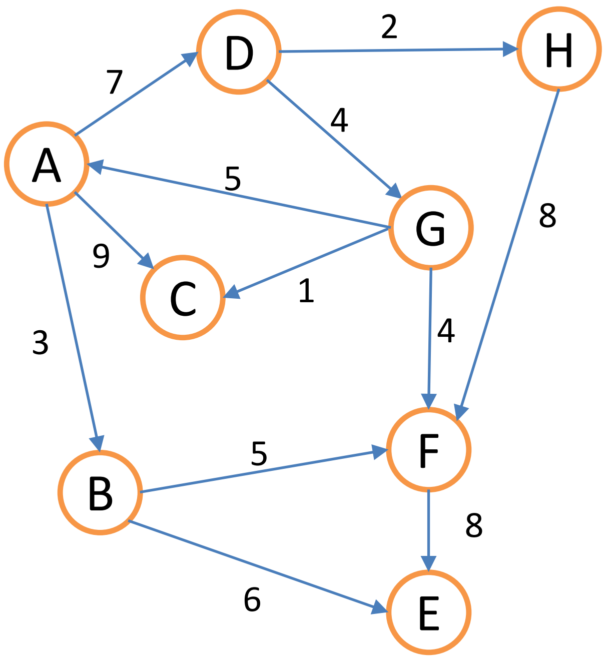
- Breath-first traversal
- Depth-first traversal
- **Example 1: Starting at A**
  - If *C is a Stack, one order of traversal is:*
    - *A, D, H, F, E, G, C, B*
    - Another traversal is: A, B, E, F, C, D, G, H
  - If *C is a Queue, one order of traversal is:*
    - *A, B, C, D, E, F, G, H*
    - Another traversal is: A, D, C, B, H, G, F, E

Graph Traversals

- Exercise: **Starting at G**
  - If *C is a Stack, one order of traversal is:*
    - *G, F, E, C, A, D, H, B*
    - Another traversal is: G, A, D, H, F, E, C, B
  - If *C is a Queue, one order of traversal is:*
    - *G, A, C, F, B, D, E, H*
    - Another traversal is: G, F, C, A, E, D, B, H

Weighted Digraph Starting at A

- Now we can sort the edges.



	A	B	C	D	E	F	G	H
A	0	3	9	7	0	0	0	0
B	0	0	0	0	6	5	0	0
C	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	4	2
E	0	0	0	0	0	0	0	0
F	0	0	0	0	8	0	0	0
G	5	0	1	0	0	4	0	0
H	0	0	0	0	0	8	0	0

Adjacency Matrix

Weighted Digraph Traversal

- **Example 3: Starting at A and sorting the adjacency set (maybe with a priority queue):**
- Performing a breadth-first traversal, the order is: A, C, D, B, G, H, E, F
- Performing a depth-first traversal, the order is: A, C, D, G, F, E, H, B

Notes

- Depth-first: Descendants are visited before siblings.
  - To traverse depth-first, use a Stack.
- Breadth-first: siblings are visited before descendants.
  - To traverse breadth-first, use a Queue.
- For all vertices to be visited from any node, the graph must be strongly connected.
- For weakly connected graphs, you'd need to exhaustively traverse from every vertex.

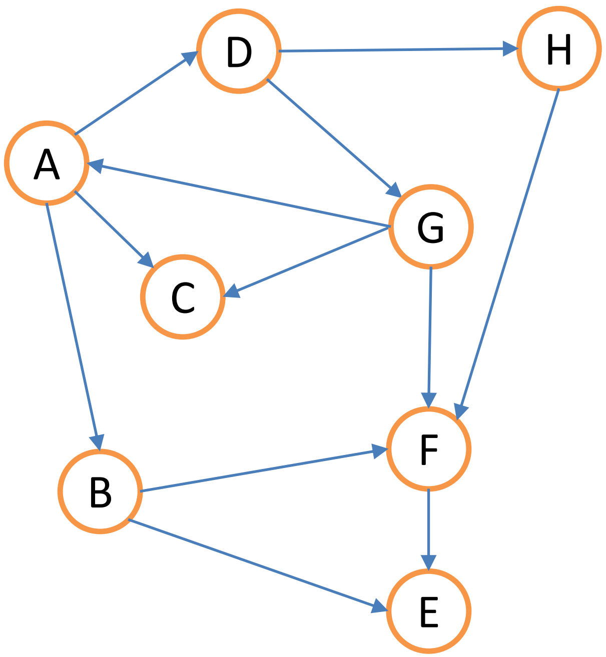
```
for each vertex, v, in graph, G
   GraphSearch(G, v)
```

A Simple Implementation

```
const int SIZE = 8;
typedef bool Graph[SIZE][SIZE];
Graph G = { // Adjacency matrix            Adjacency list
            {0, 1, 1, 1, 0, 0, 0, 0},       // A-->B-->C-->D
            {0, 0, 0, 0, 1, 1, 0, 0},       // B-->E-->F
            {0, 0, 0, 0, 0, 0, 0, 0},       // C
            {0, 0, 0, 0, 0, 0, 1, 1},       // D-->G-->H
            {0, 0, 0, 0, 0, 0, 0, 0},       // E
            {0, 0, 0, 0, 1, 0, 0, 0},       // F-->E
            {1, 0, 1, 0, 0, 1, 0, 0},       // G-->A-->C-->F
            {0, 0, 0, 0, 0, 1, 0, 0}        // H-->F
          };

struct Vertex
{
  char label;       // For displaying
  bool visited;     // Visited flag
  bool *neighbors; // Adjacency "list"
};

Vertex Vertices[SIZE] = {
                          {'A', false, G[0]},
                          {'B', false, G[1]},
                          {'C', false, G[2]},
                          {'D', false, G[3]},
                          {'E', false, G[4]},
                          {'F', false, G[5]},
                          {'G', false, G[6]},
                          {'H', false, G[7]}
                        };
```



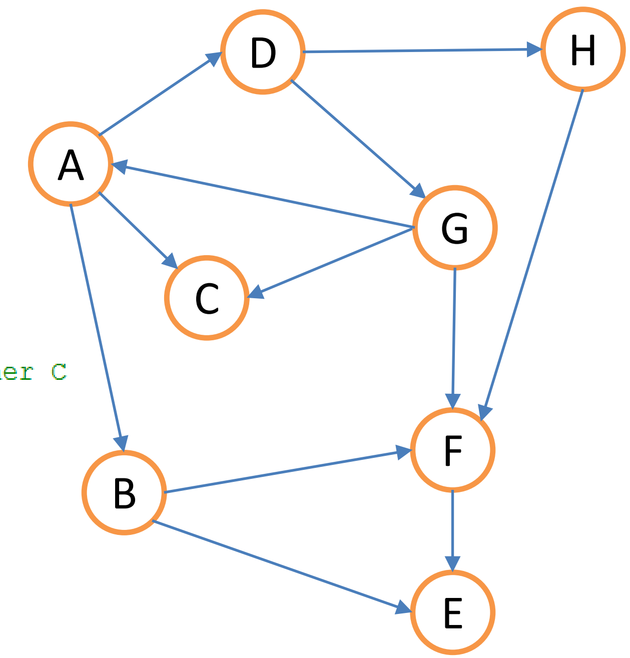
A Simple Implementation

```
void Visit(Vertex &v)
{
  cout << v.label << " ";
}

void GraphSearchStack1(Vertex *v, Vertex Vertices[])
{
  stack<Vertex *> C;

  C.push(v);                       //Put v into container C.
  while (!C.empty())               //While (container C is not empty)
  {
    Vertex *x = C.top();           //Remove a vertex, x, from container C
    C.pop();
    if (!x->visited)               //If (x has not been visited)
    {
      Visit(*x);                   //Visit x
      x->visited = true;           //Set x.visited to TRUE
      for (int i = 0; i < SIZE; i++)  //For each vertex, w,
      {
        if ((x->neighbors[i]) &&   // (adjacent to x) and
            (!Vertices[i].visited)) //   (has not been visited)
          C.push(&Vertices[i]);    //Put w into container C
      }
    }
  }
}

void main(void)
{
  GraphSearchStack1(&Vertices[0], Vertices);
}
```



Changing the **for** loop causes the alternative ordering

```
for (int i = SIZE-1; i>=0; --i)
```

Interview Question: clone a graph

```cpp
/**
 * Definition for undirected graph.
 * struct UndirectedGraphNode {
 *     int label;
 *     vector<UndirectedGraphNode *> neighbors;
 *     UndirectedGraphNode(int x) : label(x) {};
 * };
 */
class Solution {
public:
    UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {

    }
};
```