

Session 9

# Large Scale Graph Analysis

Big Data Analytics Technology, MSc in Data Science,  
Coventry University UK

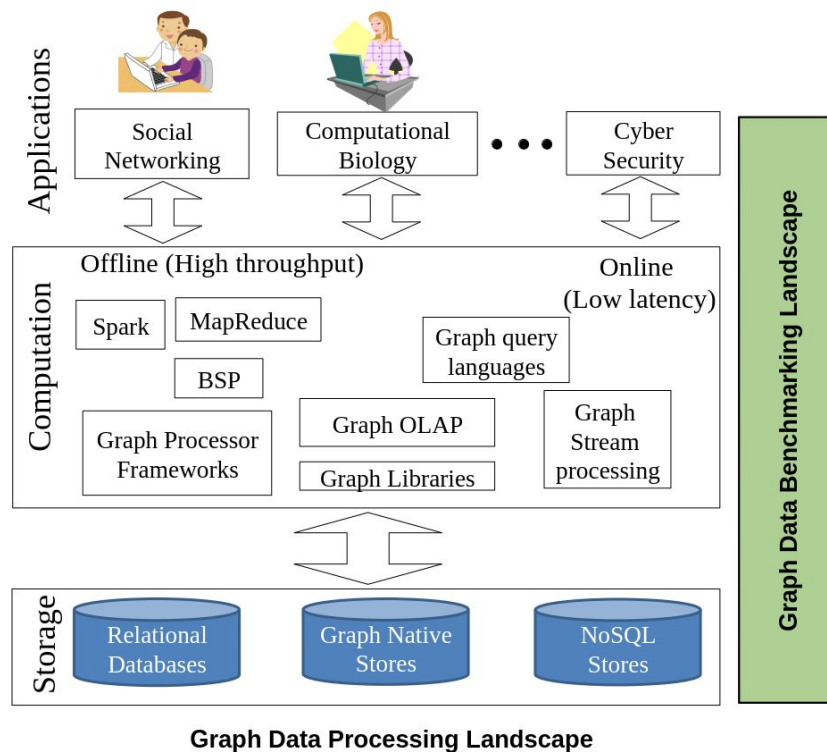
Miyuru Dayarathna

# Presentation Outline

- Introduction to Graphs
- Graph Algorithms
- Large Graph Analysis
- Conclusion



# Introduction to Graphs



# Introduction to Graphs

- A graph  $G = (V, E)$  is defined on a set of vertices  $V$ , and contains a set of edges  $E$  of ordered or unordered pairs of vertices from  $V$

# Different Types of Graphs

- Undirected vs Directed
- Weighted vs Unweighted
- Simple vs Non-simple
- Sparse vs Dense
- Cyclic vs Acyclic
- Embedded vs Topological
- Implicit vs Explicit
- Labeled vs Unlabeled

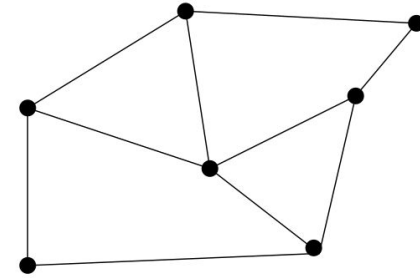
# Undirected vs Directed

A graph  $G = (V, E)$  is undirected if edge  $(x, y) \in E$  implies that  $(y, x)$  is also in  $E$ .

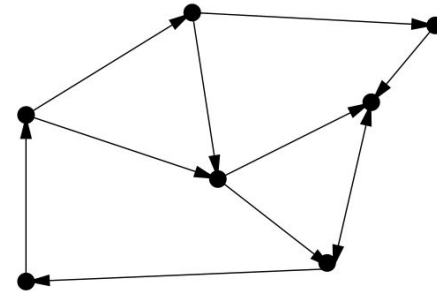
If not, we say that the graph is directed. Road networks between cities are typically undirected, since any large road has lanes going in both directions.

Street networks within cities are almost always directed, because there are at least a few one-way streets lurking somewhere.

Program-flow graphs are typically directed, because the execution flows from one line into the next and changes direction only at branches. Most graphs of graph-theoretic interest are undirected.



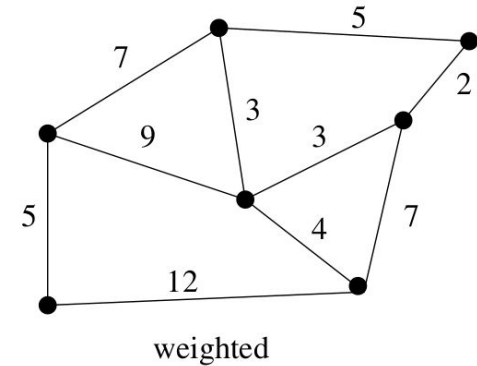
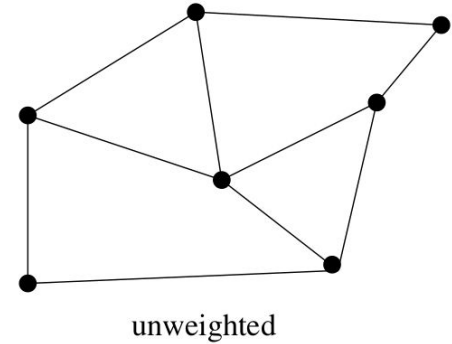
undirected



directed

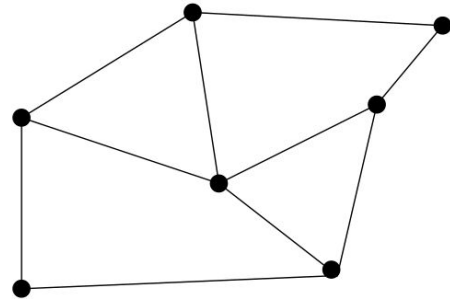
## Weighted vs. Unweighted

- Each edge (or vertex) in a weighted graph  $G$  is assigned a numerical value, or weight.
- The edges of a road network graph might be weighted with their length, drive-time, or speed limit, depending upon the application.
- In unweighted graphs, there is no cost distinction between various edges and vertices.

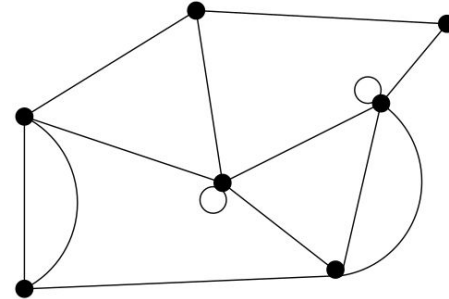


## Simple vs Non-simple

- Certain types of edges complicate the task of working with graphs.
- A self-loop is an edge  $(x, x)$  involving only one vertex.
- An edge  $(x, y)$  is a multi-edge if it occurs more than once in the graph.



simple

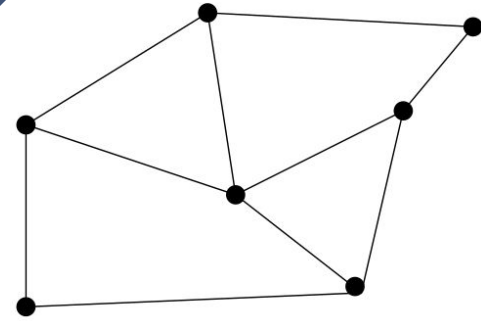


non-simple

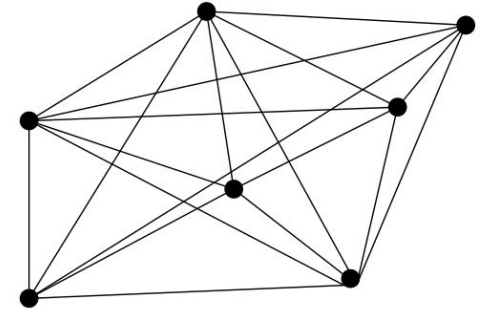


## Sparse vs Dense

- Graphs are sparse when only a small fraction of the possible vertex pairs actually have edges defined between them.
- Graphs where a large fraction of the vertex pairs define edges are called dense. There is no official boundary between what is called sparse and what is called dense, but typically dense graphs have a quadratic number of edges, while sparse graphs are linear in size.
- Sparse graphs are usually sparse for application-specific reasons. Road networks must be sparse graphs because of road junctions.



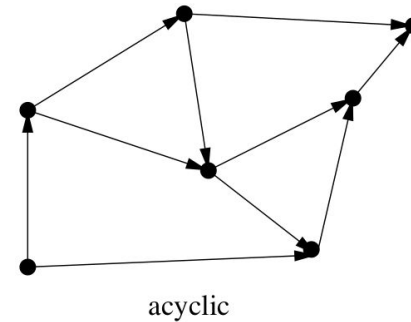
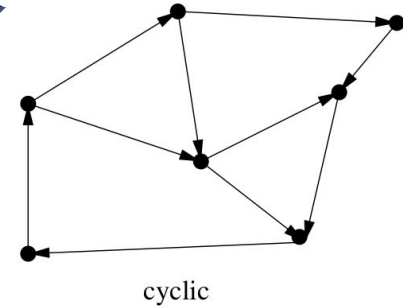
sparse



dense

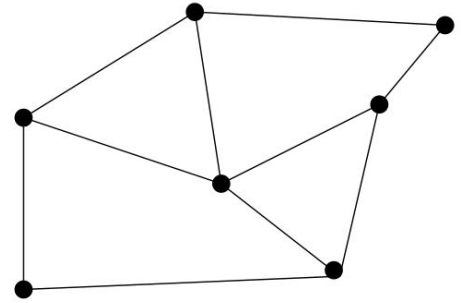
## Cyclic vs. Acyclic

- An acyclic graph does not contain any cycles. Trees are connected, acyclic undirected graphs.
- Trees are the simplest interesting graphs, and are inherently recursive structures because cutting any edge leaves two smaller trees.
- Directed acyclic graphs are called DAGs. They arise naturally in scheduling problems, where a directed edge  $(x, y)$  indicates that activity  $x$  must occur before  $y$ .
- An operation called topological sorting orders the vertices of a DAG to respect these precedence constraints.
- Topological sorting is typically the first step of any algorithm on a DAG

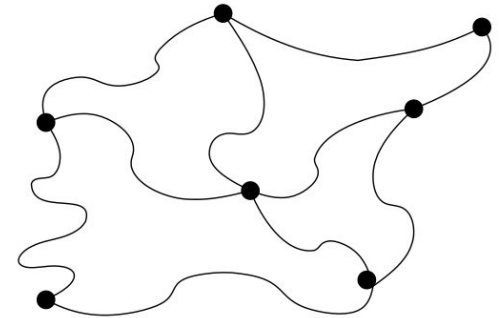


# Embedded vs Topological

- A graph is embedded if the vertices and edges are assigned geometric positions.
- Thus, any drawing of a graph is an embedding, which may or may not have algorithmic significance.



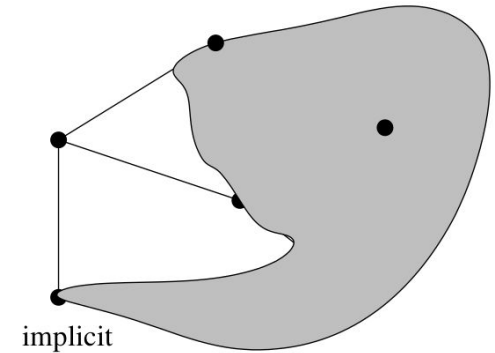
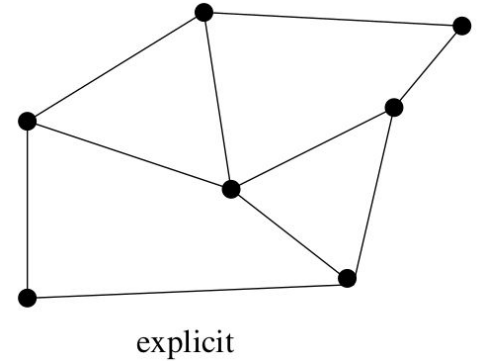
embedded



topological

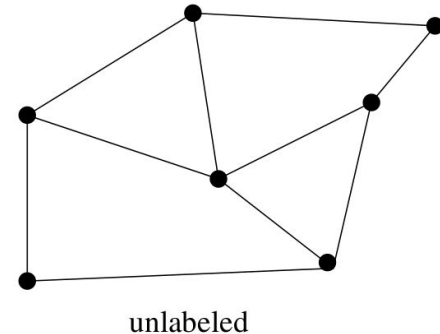
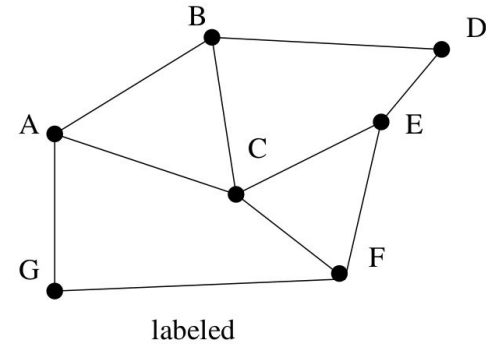
# Implicit vs Explicit

- Certain graphs are not explicitly constructed and then traversed, but built as we use them. A good example is in backtrack search.
- The vertices of this implicit search graph are the states of the search vector, while edges link pairs of states that can be directly generated from each other.
- Because you do not have to store the entire graph, it is often easier to work with an implicit graph than explicitly construct it prior to analysis.



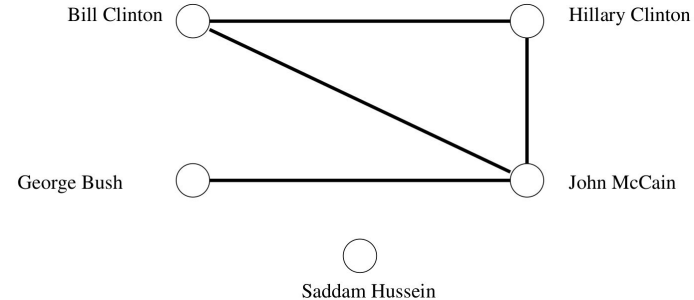
## Labeled vs Unlabeled

- Each vertex is assigned a unique name or identifier in a labeled graph to distinguish it from all other vertices.
- In unlabeled graphs, no such distinctions have been made.
- Graphs arising in applications are often naturally and meaningfully labeled, such as city names in a transportation network.
- A common problem is that of isomorphism testing—determining whether the topological structure of two graphs are identical if we ignore any labels.
- Such problems are typically solved using backtracking, by trying to assign each vertex in each graph a label such that the structures are identical.



## Example Scenario - The Friendship Graph

- Let us consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.
- Such graphs are called social networks and are well defined on any set of people
- An entire science analyzing social networks has sprung up in recent years, because many interesting aspects of people and their behavior are best understood as properties of this friendship graph.
- Most of the graphs that one encounters in real life are sparse.



## Friendship Graph - Questions

- What questions might we ask about the friendship graph?
- If I am your friend, does that mean you are my friend?
  - This question really asks whether the graph is directed. A graph is undirected if edge  $(x, y)$  always implies  $(y, x)$ . Otherwise, the graph is said to be directed.

## Friendship Graph - Questions (Contd.)

- What questions might we ask about the friendship graph?
- If I am your friend, does that mean you are my friend?
  - ▷ This question really asks whether the graph is directed. A graph is undirected if edge  $(x, y)$  always implies  $(y, x)$ . Otherwise, the graph is said to be directed.
- How close a friend are you?
  - ▷ In weighted graphs, each edge has an associated numerical attribute. We could model the strength of a friendship by associating each edge with an appropriate value, perhaps from -10 (enemies) to 10 (blood brothers).
  - ▷ The edges of a road network graph might be weighted with their length, drive-time, or speed limit, depending upon the application.
  - ▷ A graph is said to be unweighted if all edges are assumed to be of equal weight.



## Friendship Graph - Questions (Contd.)

- Am I my own friend?
  - ▷ This question addresses whether the graph is simple, meaning it contains no loops and no multiple edges. An edge of the form  $(x, x)$  is said to be a loop. Sometimes people are friends in several different ways. Perhaps  $x$  and  $y$  were college classmates and now work together at the same company. We can model such relationships using multiedges—multiple
  - ▷ edges  $(x, y)$  perhaps distinguished by different labels. Simple graphs really are often simpler to work with in practice. Therefore, we might be better off declaring that no one is their own friend.

## Friendship Graph - Questions (Contd.)

- Who has the most friends?
  - ▷ The degree of a vertex is the number of edges adjacent to it. The most popular person defines the vertex of highest degree in the friendship graph. Remote hermits are associated with degree-zero vertices. In dense graphs, most vertices have high degrees, as opposed to sparse graphs with relatively few edges. In a regular graph, each vertex has exactly the same
  - ▷ degree. A regular friendship graph is truly the ultimate in social-ism.

## Friendship Graph - Questions (Contd.)

- Do my friends live near me?
  - ▷ Many of your friends are your friends only because they happen to live near you (e.g., neighbors) or used to live near you (e.g., college roommates).
  - ▷ Thus, a full understanding of social networks requires an embedded graph, where each vertex is associated with the point on this world where they live.
  - ▷ This geographic information may not be explicitly encoded, but the fact that the graph is inherently embedded in the plane shapes our interpretation of any analysis.

## Friendship Graph - Questions (Contd.)

- Oh, you also know her?
  - ▷ Social networking services such as Myspace and LinkedIn are built on the premise of explicitly defining the links between members and their member-friends.
  - ▷ Such graphs consist of directed edges from person/vertex  $x$  professing his friendship to person/vertex  $y$ .
  - ▷ Six degrees of separation
    - ▷ argues that there is a short path linking every two people in the world but offers us no help in actually finding this path.

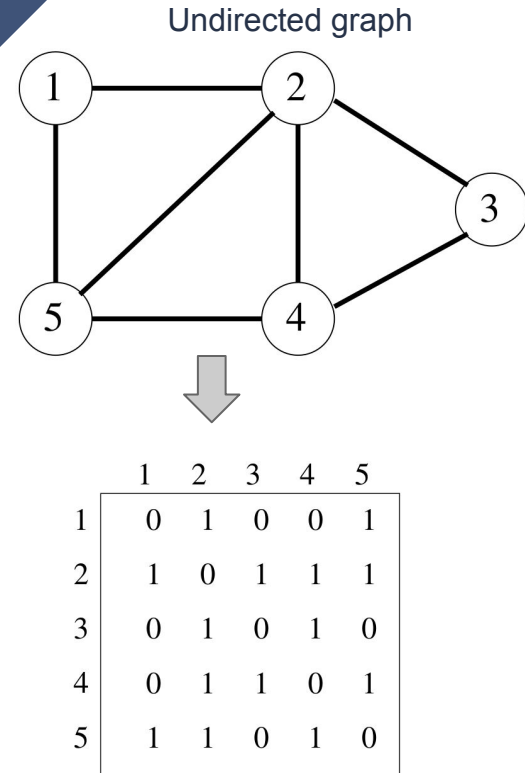
## Friendship Graph - Questions (Contd.)

- Are you truly an individual, or just one of the faceless crowd?
- This question boils down to whether the friendship graph is labeled or unlabeled. Does each vertex have a name/label which reflects its identity, and is this label important for our analysis?
- Much of the study of social networks is unconcerned with labels on graphs. Often the index number given a vertex in the graph data structure serves as its label, perhaps for convenience or the need for anonymity.
- Someone studying how an infectious disease spreads through a graph may label each vertex with whether the person is healthy or sick, it being irrelevant what their name is.

# Data Structures for Graphs

## ■ Adjacency Matrix

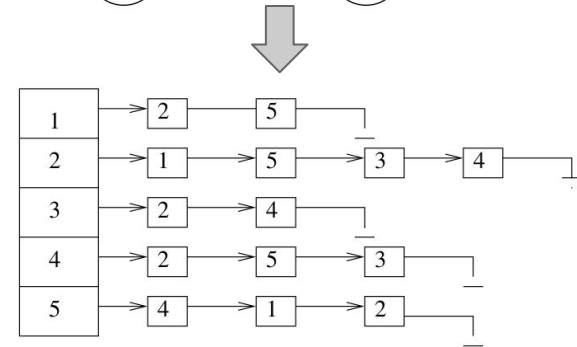
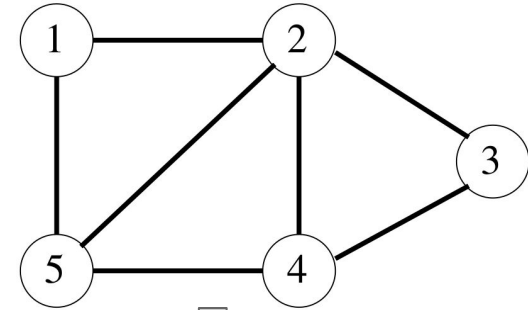
- ▷ We can represent  $G$  using an  $n \times n$  matrix  $M$ , where element  $M[i, j] = 1$  if  $(i, j)$  is an edge of  $G$ , and 0 if it isn't.
- ▷ This allows fast answers to the question “is  $(i, j)$  in  $G$ ?”, and rapid updates for edge insertion and deletion. It may use excessive space for graphs with many vertices and relatively few edges, however.



## Data Structures for Graphs (Contd.)

### Adjacency Lists

- ▶ Sparse graphs can be more efficiently represented by using linked lists to store the neighbours adjacent to each vertex
- ▶ Adjacency lists require pointers (i.e., variable references)
- ▶ Adjacency lists make it harder to verify whether a given edge  $(i, j)$  is in  $G$ , since we must search through the appropriate list to find the edge.
- ▶ However, it is easy to design graph algorithms that avoid any need for such queries
- ▶ Typically, we sweep through all the edges of the graph in one pass via a breadth-first or depth-first traversal, and update the implications of the current edge as we visit it



## Adjacency List vs Adjacency Matrix

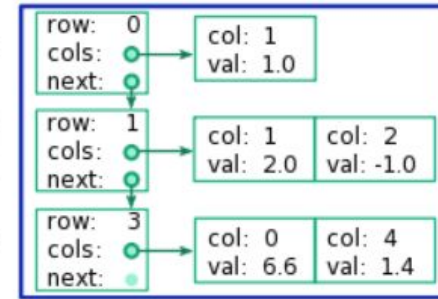
Comparison	Winner
Faster to test if $(x, y)$ is in graph?	adjacency matrices
Faster to find the degree of a vertex?	adjacency lists
Less memory on small graphs?	adjacency lists $(m + n)$ vs. $(n^2)$
Less memory on big graphs?	adjacency matrices (a small win)
Edge insertion or deletion?	adjacency matrices $O(1)$ vs. $O(d)$
Faster to traverse the graph?	adjacency lists $\Theta(m + n)$ vs. $\Theta(n^2)$
Better for most problems?	adjacency lists



# Sparse Matrix Formats - List of Lists (LIL/LOL)

- LIL is one of the simplest sparse matrix formats
- Each non-zero matrix row is represented by an element in a linked list
- Each element in the linked list records the row number, and the column data for the matrix entries in that row.
- The column data consists of a list, where each list element corresponds to a non-zero matrix element, and stores information about (i) the column number, and (ii) the value of the matrix element.

$$\begin{bmatrix} 0 & 1.0 & 0 & 0 & 0 \\ 0 & 2.0 & -1.0 & 0 & 0 \\ 6.6 & 0 & 0 & 0 & 1.4 \end{bmatrix}$$



## Sparse Matrix Formats - List of Lists (LIL/LOL) - Advantages

- Memory efficiency
  - ▷ The list of rows only needs to be as long as the number of non-zero matrix rows; the rest are omitted. Hence this format is very memory efficient.
  - ▷ Each list of column data only needs to be as long as the number of non-zero elements on that row.
  - ▷ The total amount of memory required is proportional to the number of non-zero elements, regardless of the size of the matrix itself.

## Sparse Matrix Formats - List of Lists (LIL/LOL) - Advantages (Contd.)

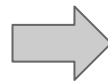
- It is relatively easy to alter the "sparsity structure" of the matrix
- To add a new non-zero element, one simply has to step through the row list, and either,
  - ▷ insert a new element into the linked list if the row was not previously on the list (this insertion takes  $O(1)$  time), or
  - ▷ modify the column list (which is usually very short if the matrix is very sparse)
  - ▷ The LIL format is preferred if one needs to construct a sparse matrix where the non-zero elements are not distributed in any useful pattern

## Sparse Matrix Formats - List of Lists (LIL/LOL) - Disadvantages

- Compared to other sparse matrix formats accessing an individual matrix element in LIL format is relatively slow
  - ▷ Looking up a given matrix index  $(i,j)$  requires stepping through the row list to find an element with row index  $i$ ; and if one is found, stepping through the column row to find index  $j$
  - ▷ Looking up an element in a diagonal  $N \times N$  matrix in the LIL format takes  $O(N)$  time!
  - ▷ Matrix arithmetic in the LIL format is very inefficient
  - ▷ The CSR and CSC formats are much more efficient at element access

## LIL Example

```
1  from scipy import *
2  import scipy.sparse as sp
3
4  A = sp.lil_matrix((4,5))      # Create empty 4x5 LIL matrix
5  A[0,1] = 1.0
6  A[1,1] = 2.0
7  A[1,2] = -1.0
8  A[3,0] = 6.6
9  A[3,4] = 1.4
10
11  ## Verify the matrix contents by printing it
12  print(A)
```

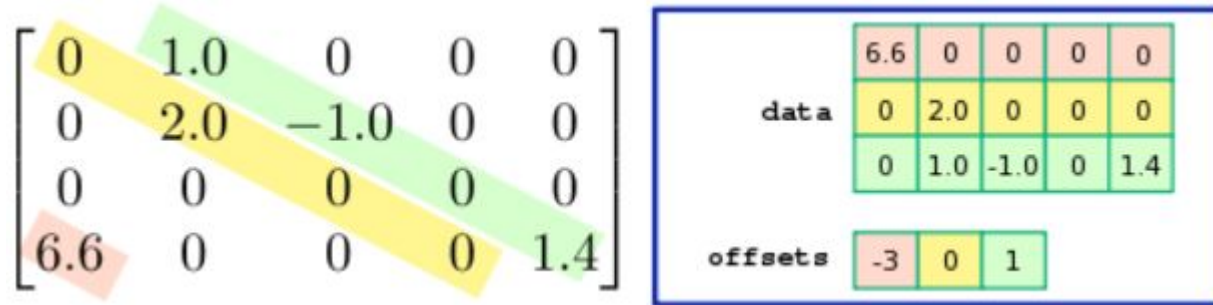


(0, 1)	1.0
(1, 1)	2.0
(1, 2)	-1.0
(3, 0)	6.6
(3, 4)	1.4

Output

## Sparse Matrix Formats - Diagonal Storage (DIA)

- The Diagonal Storage (DIA) format stores the contents of a sparse matrix along its diagonals. It makes use of a 2D array, which we denote by data, and a 1D integer array, which we denote by offsets.



## Sparse Matrix Formats - Diagonal Storage (DIA) - Contd.

- Each row of the data array stores one of the diagonals of the matrix, and `offsets[i]` records which diagonal that row of the data corresponds to, with "offset 0" corresponding to the main diagonal.
- In the above example, row 0 of data contains the entries `[6.6,0,0,0,0]`, and `offsets[0]` contains the value `-3`, indicating that the entry 6.6 occurs along the `-3` subdiagonal, in column 0. (The extra elements in that row of data lie outside the bounds of the matrix, and are ignored.)
- Diagonals containing only zero are omitted.

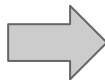
## Sparse Matrix Formats - Diagonal Storage (DIA) - Contd.

- For sparse matrices with very few non-zero diagonals, such as diagonal or tridiagonal matrices, the DIA format allows for very quick arithmetic operations.
- Its main limitation is that looking up each matrix element requires performing a blind search through the offsets array.
- That's fine if there are very few non-zero diagonals, as offsets will be small.
- But if the number of non-zero diagonals becomes large, performance becomes very poor.
- In the worst-case scenario of an anti-diagonal matrix, element lookup takes  $O(N)$  time!



## Diagonal Storage (DIA) Example

```
1  from scipy import *
2  from numpy import *
3  import scipy.sparse as sp
4
5  N = 6  # Matrix size
6
7  diag0 = -2 * ones(N)
8  diag1 = ones(N)
9
10 A = sp.dia_matrix(([diag1, diag0, diag1], [-1,0,1]), shape=(N,N))
11
12 ## Verify the matrix contents by printing it
13 print(A.toarray())
```



```
[[-2.  1.  0.  0.  0.  0.]
 [ 1. -2.  1.  0.  0.  0.]
 [ 0.  1. -2.  1.  0.  0.]
 [ 0.  0.  1. -2.  1.  0.]
 [ 0.  0.  0.  1. -2.  1.]
 [ 0.  0.  0.  0.  1. -2.]]
```

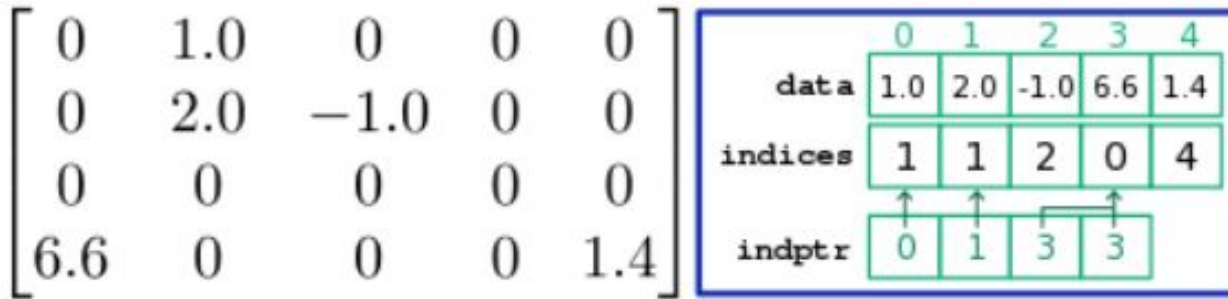
Output

## Diagonal Storage (DIA) Example - Contd.

- Here, the first input to `dia_matrix` is a tuple of the form (data, offsets), where data and offsets are arrays of the sort described above.
- This returns a sparse matrix in the DIA format, with the specified contents (the elements in data which lie outside the bounds of the matrix are ignored).
- In this example, the matrix is tridiagonal with -2 along the main diagonal and 1 along the +1 and -1 diagonals.
- Another way to create a DIA matrix is to first create a matrix in another format (e.g. a conventional 2D array), and provide that as the input to `dia_matrix`. This returns a sparse matrix with the same contents, in DIA format.

## Sparse Matrix Formats - Compressed Sparse Row (CSR)

- The Compressed Sparse Row (CSR) format represents a sparse matrix using three arrays, which we denote by *data*, *indices*, and *indptr*



## Sparse Matrix Formats - Compressed Sparse Row (CSR) - Contd.

- data
  - ▷ Stores the values of the non-zero elements of the matrix, in sequential order from left to right along each row, then from the top row to the bottom.
  - ▷ The array denoted indices records the column index for each of these elements.
  - ▷ In the above example, data[3] stores a value of 6.6, and indices[3] has a value of 0, indicating that a matrix element with value 6.6 occurs in column 0.
  - ▷ These two arrays have the same length, equal to the number of non-zero elements in the sparse matrix.

## Sparse Matrix Formats - Compressed Sparse Row (CSR) - Contd.

- indptr (stands for "index pointer")
  - ▷ provides an association between the row indices and the matrix elements, but in an indirect manner.
  - ▷ Its length is equal to the number of matrix rows (including zero rows).
  - ▷ For each row  $i$ , if the row is non-zero,  $\text{indptr}[i]$  records the index in the data and indices arrays corresponding to the first non-zero element on row  $i$ . (For a zero row,  $\text{indptr}$  records the index of the next non-zero element occurring in the matrix.)

## Sparse Matrix Formats - Compressed Sparse Row (CSR) - Data lookup example

$$\begin{bmatrix} 0 & 1.0 & 0 & 0 & 0 \\ 0 & 2.0 & -1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 6.6 & 0 & 0 & 0 & 1.4 \end{bmatrix}$$

	0	1	2	3	4
data	1.0	2.0	-1.0	6.6	1.4
indices	1	1	2	0	4
indptr	0	1	3	3	

- Consider looking up index (1,2) in the above figure.
- The row index is 1, therefore we examine indptr[1] (whose value is 1) and indptr[2] (whose value is 3).
- This means that the non-zero elements for matrix row 1 correspond to indices  $1 \leq n < 3$  of the data and indices arrays.
- We search indices[1] and indices[2], looking for a column index of 2. This is found in indices[2], therefore we look in data[2] for the value of the matrix element, which is -1.0.

## Compressed Sparse Row (CSR) - Advantages

- Looking up an individual matrix element is very efficient
- Unlike the LIL format, where we need to step through a linked list, in the CSR format the indptr array allows to jump straight to the data for the relevant row.
- For the same reason, the CSR format is efficient for row slicing operations (e.g.,  $A[4,:]$ ), and for matrix-vector products like  $A\vec{x}$  (which involves taking the product of each matrix row with the vector  $\vec{x}$  ).

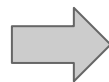
## Compressed Sparse Row (CSR) - Disadvantages

- Column slicing (e.g.  $A[:,4]$ ) is inefficient, since it requires searching through all elements of the indices array for the relevant column index.
- Changes to the sparsity structure (e.g., inserting new elements) are also very inefficient, since all three arrays need to be re-arranged.



## Compressed Sparse Row (CSR) Example

```
1  from scipy import *
2  import scipy.sparse as sp
3
4  data = [1.0, 2.0, -1.0, 6.6, 1.4]
5  rows = [0, 1, 1, 3, 3]
6  cols = [1, 1, 2, 0, 4]
7
8  A = sp.csr_matrix((data, [rows, cols]), shape=(4,5))
9  print(A)
```



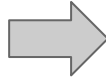
(0, 1)	1.0
(1, 1)	2.0
(1, 2)	-1.0
(3, 0)	6.6
(3, 4)	1.4

Output

The first input to `csr_matrix` is a tuple of the form  $(data, idx)$ , where `data` is a 1D array specifying the non-zero matrix elements, `idx[0,:]` specifies the row indices, and `idx[1,:]` specifies the column indices.

## Compressed Sparse Row (CSR) Example

```
print(A.data)  
print(A.indices)  
print(A.indptr)
```



```
[ 1.  2. -1.  6.6  1.4]  
[1 1 2 0 4]  
[0 1 3 3 5]
```

## Sparse Matrix Formats - Compressed Sparse Column (CSC)

- The Compressed Sparse Column (CSC) format is very similar to the CSR format, except that the role of rows and columns is swapped.

$$\begin{bmatrix} 0 & 1.0 & 0 & 0 & 0 \\ 0 & 2.0 & -1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 6.6 & 0 & 0 & 0 & 1.4 \end{bmatrix}$$

	0	1	2	3	4
data	6.6	1.0	2.0	-1.0	1.4
indices	3	0	1	1	3
indptr	0	1	3	4	4

## Sparse Matrix Formats - Compressed Sparse Column (CSC) (Contd.)

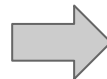
- data
  - ▷ This array stores non-zero matrix elements in sequential order from top to bottom along each column, then from the left-most column to the right-most.
- indices
  - ▷ This array stores row indices
- indptr
  - ▷ This array corresponds to one column of the matrix

## Sparse Matrix Formats - Compressed Sparse Column (CSC) (Contd.)

- The CSC format is efficient at matrix lookup, column slicing operations (e.g.,  $A[:,4]$ ), and vector-matrix products like  $\vec{x}^T \mathbf{A}$  (which involves taking the product of the vector  $\vec{x}$  with each matrix column).
- However, it is inefficient for row slicing (e.g.  $A[4,:]$ ), and for changes to the sparsity structure.

## Compressed Sparse Column (CSC) Example

```
1  from scipy import *
2  import scipy.sparse as sp
3
4  data = [1.0, 2.0, -1.0, 6.6, 1.4]
5  rows = [0, 1, 1, 3, 3]
6  cols = [1, 1, 2, 0, 4]
7
8  A = sp.csc_matrix((data, [rows, cols]), shape=(4,5))
9  print(A)
```



(3, 0)	6.6
(0, 1)	1.0
(1, 1)	2.0
(1, 2)	-1.0
(3, 4)	1.4

# Graph Traversal

- The most fundamental graph problem is to visit every edge and vertex in a graph in a systematic way.
- Basic bookkeeping operations on graphs (e.g., printing, copying graphs, converting between alternate representations) are applications of graph traversal
- The key idea behind graph traversal is to mark each vertex when we first visit it and keep track of what we have not yet completely explored.

## Graph Traversal (Contd.)

- Each vertex will exist in one of the three states
  - ▷ **undiscovered** - The vertex is in its initial state
  - ▷ **discovered** - the vertex has been found, but we have not yet checked out all its incident edges.
  - ▷ **processed** - the vertex after we have visited all its incident edges
- A vertex cannot be processed until after we discover it, therefore, the state of each vertex progresses over the course of the traversal from undiscovered to discovered to processed.



## Graph Traversal (Contd.)

- Need to maintain a structure containing the vertices that have been discovered but not yet completely processed.
- Initially, only the single start vertex is considered to be discovered.
- To completely explore a vertex  $v$ , we must evaluate each edge leaving  $v$ . If an edge goes to an undiscovered vertex  $x$ , we mark  $x$  discovered and add it to the list of work to do.
- We ignore an edge that goes to a processed vertex, because further contemplation will tell us nothing new about the graph.
- We can also ignore any edge going to a discovered but not processed vertex, because the destination already resides on the list of vertices to process.

## Graph Traversal (Contd.)

- Each undirected edge will be considered exactly twice, once when each of its endpoints is explored.
- Directed edges will be considered only once, when exploring the source vertex.
- Every edge and vertex in the connected component must eventually be visited.
- Why? Suppose that there exists a vertex  $u$  that remains unvisited, whose neighbor  $v$  was visited.
- This neighbor  $v$  will eventually be explored, after which we will certainly visit  $u$ .
- Thus, we must find everything that is there to be found.

## Graph Traversal (Contd.)

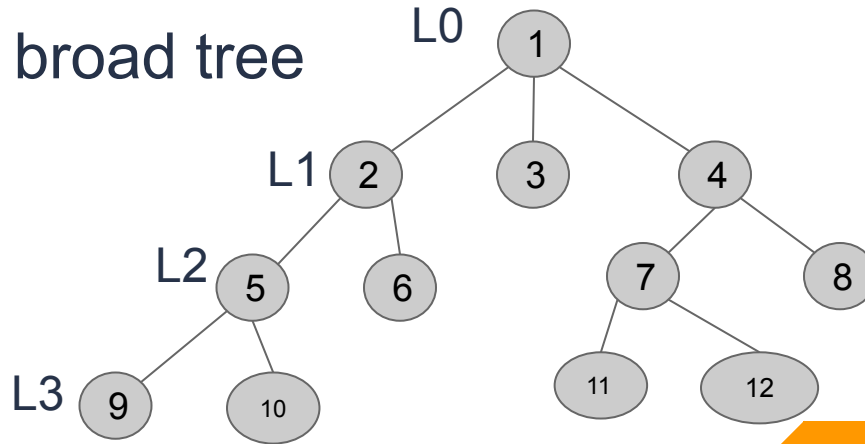
- There are two primary graph traversal algorithms
  - ▷ Breadth-first search (BFS)
  - ▷ Depth-first search (DFS).
- For certain problems, it makes absolutely no difference which you use, but in others the distinction is crucial.
- The difference between BFS and DFS results is in the order in which they explore vertices.

## Graph Traversal (Contd.)

- This order depends completely upon the container data structure used to store the discovered but not processed vertices.
- Queue
  - ▷ By storing the vertices in a first-in, first-out (FIFO) queue, we explore the oldest unexplored vertices first. Thus our explorations radiate out slowly from the starting vertex, defining a breadth-first search.
- Stack
  - ▷ By storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by lurching along a path, visiting a new neighbor if one is available, and backing up only when we are surrounded by previously discovered vertices. Thus, our explorations quickly wander away from our starting point, defining a depth-first search.

# Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph.
- A BFS traversal of a graph returns the nodes of the graph level-by-level
- BFS produces short, broad tree



# Breadth-First Search - The basic algorithm

BFS( $G, s$ )

for each vertex  $u \in V[G] - \{s\}$  do

$state[u] = \text{"undiscovered"}$

$p[u] = nil$ , i.e. no parent is in the BFS tree

$state[s] = \text{"discovered"}$

$p[s] = nil$

$Q = \{s\}$

while  $Q \neq \emptyset$  do

$u = \text{dequeue}[Q]$

    process vertex  $u$  as desired

    for each  $v \in Adj[u]$  do

        process edge  $(u, v)$  as desired

        if  $state[v] = \text{"undiscovered"}$  then

$state[v] = \text{"discovered"}$

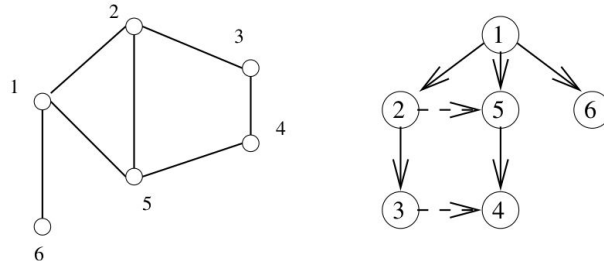
$p[v] = u$

            enqueue[ $Q, v$ ]

$state[u] = \text{"processed"}$

## Breadth-First Search - The basic algorithm (Contd.)

- At some point during the course of a traversal, every node in the graph changes state from undiscovered to discovered.
- In a breadth-first search of an undirected graph, we assign a direction to each edge, from the discoverer  $u$  to the discovered  $v$ . We thus denote  $u$  to be the parent of  $v$ .
- Since each node has exactly one parent, except for the root, this defines a tree on the vertices of the graph. This tree, (shown below), defines a shortest path from the root to every other node in the tree.
- This property makes breadth-first search very useful in shortest path problems.

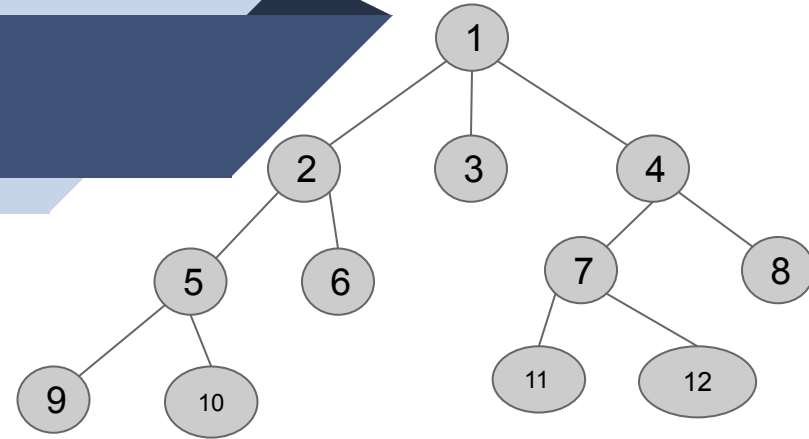


# Breadth-First Search - Illustration

Item Dequeued

1	[2,3,4]
2	[3,4,5,6]
3	[4,5,6]
4	[5,6,7,8]
5	[6,7,8,9,10]
6	[7,8,9,10]
7	[8,9,10,11,12]
8	[9,10,11,12]
9	[10,11,12]
10	[11,12]
11	[12]
12	[]

Queue



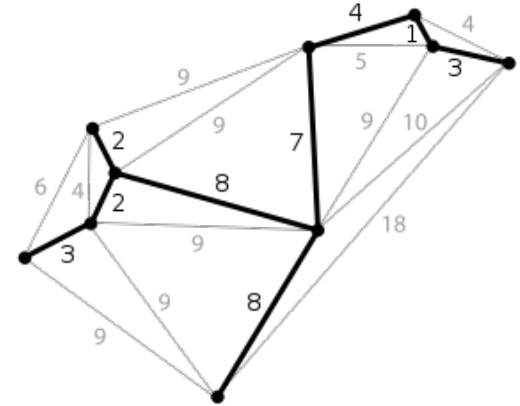


## Applications of BFS

- Shortest path and minimum spanning tree for unweighted graph
  - ▷ In an unweighted graph, the shortest path is the path with the least number of edges.
  - ▷ With Breadth First Search, we always reach a vertex from a given source using the minimum number of edges.
  - ▷ Furthermore, in the case of unweighted graphs, any spanning tree is *Minimum Spanning Tree* and we can use either Depth or Breadth first traversal for finding a spanning tree.
- Minimum Spanning Tree for weighted graphs (weight should be non-negative and the same for each pair of vertices)

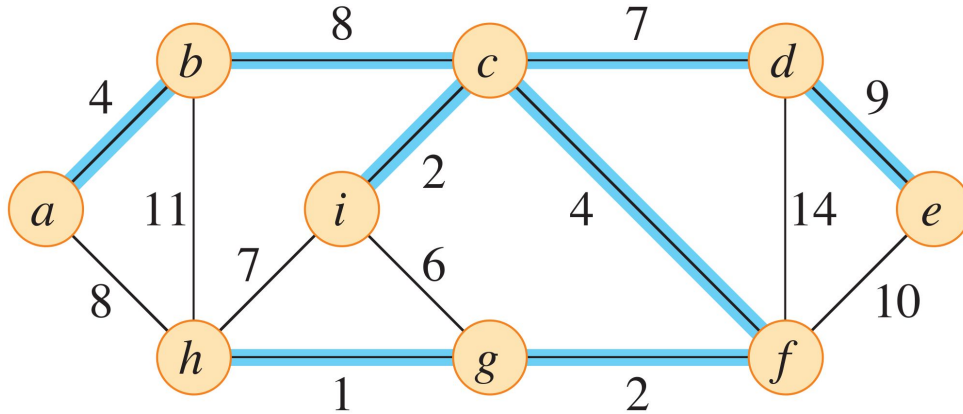
# Minimum Spanning Tree (MST)

- A subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
- This is a spanning tree whose sum of edge weights is as small as possible
- Any edge-weighted undirected graph has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components



A planar graph and its minimum spanning tree. Each edge is labeled with its weight, which here is roughly proportional to its length.

## Minimum Spanning Tree (MST) (Contd.)



A minimum spanning tree for a connected graph. The weights on edges are shown, and the blue edges form a minimum spanning tree. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (*b*,*c*) and replacing it with the edge (*a*,*h*) yields another spanning tree with weight 37

## Applications of BFS (Contd.)

- Peer-to-peer Networks
  - ▷ In Peer-to-Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
- Crawlers in Search Engines
  - ▷ Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same.
  - ▷ Depth First Traversal can also be used for crawlers, but the advantage of Breadth First Traversal is, the depth or levels of the built tree can be limited.
- Social Networking Websites
  - ▷ In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

## Applications of BFS (Contd.)

- GPS Navigation systems
  - Breadth First Search is used to find all neighboring locations.
- Broadcasting in Network
  - In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- Finding all nodes within one connected component
  - We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.
- Path Finding
  - We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
- Artificial Intelligence (AI) applications
  - In AI, BFS is used in traversing a game tree to find the best move.

## Applications of BFS (Contd.)

- Network Security
  - In the field of network security, BFS is used in traversing a network to find all the devices connected to it.
- Connected Component
  - BFS is used to find all connected components in an undirected graph
- Topological sorting
  - BFS can be used to find a topological ordering of the nodes in a directed acyclic graph (DAG).
  - Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers
- Recommender systems
  - BFS can be used to find similar items in a large dataset by traversing the items' connections in a similarity graph.

## Advantages of Breadth First Search

- BFS will never get trapped exploring the useful path forever.
- If there is a solution, BFS will definitely find it.
- If there is more than one solution then BFS can find the minimal one that requires less number of steps.
- Low storage requirement – linear with depth.
- Easily programmable.

## Disadvantages of Breadth First Search

- The main drawback of BFS is its memory requirement.
- Since each level of the graph must be saved in order to generate the next level and the amount of memory is proportional to the number of nodes stored the space complexity of BFS is  $O(b^d)$ , where  $b$  is the branching factor (the number of children at each node, the outdegree) and  $d$  is the depth.
- As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.



## Depth-First Search (DFS)

- A DFS traversal of a graph returns the nodes of the graph by traveling deep through one path until hitting a dead end and then retracing the steps
  - ▷ DFS produces a deep, narrow tree
  - ▷ We start by pushing the root (start) node onto the stack
  - ▷ Each time we're going to pop a node (say X) off the stack
  - ▷ We will push all of the new neighbors of node X onto the stack

# Depth-First Search - A basic algorithm

```
DFS( $G, u$ )  
   $state[u] = \text{"discovered"}$   
  process vertex  $u$  if desired  
  
     $time = time + 1$   
     $entry[u] = time$   
    for each  $v \in Adj[u]$  do  
      process edge  $(u, v)$  if desired  
      if  $state[v] = \text{"undiscovered"}$  then  
         $p[v] = u$   
        DFS( $G, v$ )  
  
   $state[u] = \text{"processed"}$   
   $exit[u] = time$   
   $time = time + 1$ 
```

A depth-first search can be thought of as a breadth-first search with a stack instead of a queue.

The beauty of implementing dfs recursively is that recursion eliminates the need to keep an explicit stack

## Depth-First Search - A basic algorithm (Contd.)

- The implementation of dfs shown in the previous slide maintains a notion of traversal time for each vertex.
- The time clock ticks each time we enter or exit any vertex. We keep track of the entry and exit times for each vertex.
- The time intervals have interesting and useful properties with respect to depth-first search
  - ▷ Who is an ancestor?
    - ▷ Suppose that  $x$  is an ancestor of  $y$  in the DFS tree. This implies that we must enter  $x$  before  $y$ , since there is no way we can be born before our own father or grandfather!
    - ▷ We also must exit  $y$  before we exit  $x$ , because the mechanics of DFS ensure we cannot exit  $x$  until after we have backed up from the search of all its descendants. Thus the time interval of  $y$  must be properly nested within ancestor  $x$ .

## Depth-First Search - A basic algorithm (Contd.)

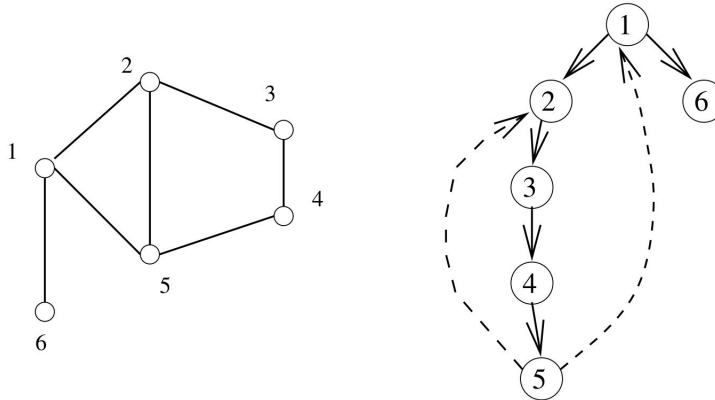
- How many descendants?
  - ▷ The difference between the exit and entry times for  $v$  tells us how many descendants  $v$  has in the DFS tree.
  - ▷ The clock gets incremented on each vertex entry and vertex exit, so half the time difference denotes the number of descendants of  $v$ .
- The entry and exit times are useful in several applications of DFS
  - ▷ E.g., topological sorting and biconnected/strongly-connected components

## Depth-First Search - A basic algorithm (Contd.)

- The other important property of a depth-first search is that it partitions the edges of an undirected graph into exactly two classes
  - ▷ tree edges
    - ▷ The tree edges discover new vertices, and are those encoded in the parent relation.
  - ▷ back edges.
    - ▷ Back edges are those whose other endpoint is an ancestor of the vertex being expanded, therefore they point back into the tree.

## Depth-First Search - A basic algorithm (Contd.)

- An amazing property of depth-first search is that all edges fall into these two classes. Why can't an edge go to a brother or cousin node instead of an ancestor?
- All nodes reachable from a given vertex  $v$  are expanded before we finish with the traversal from  $v$ , so such topologies are impossible for undirected graphs. This edge classification proves fundamental to the correctness of DFS-based algorithms.

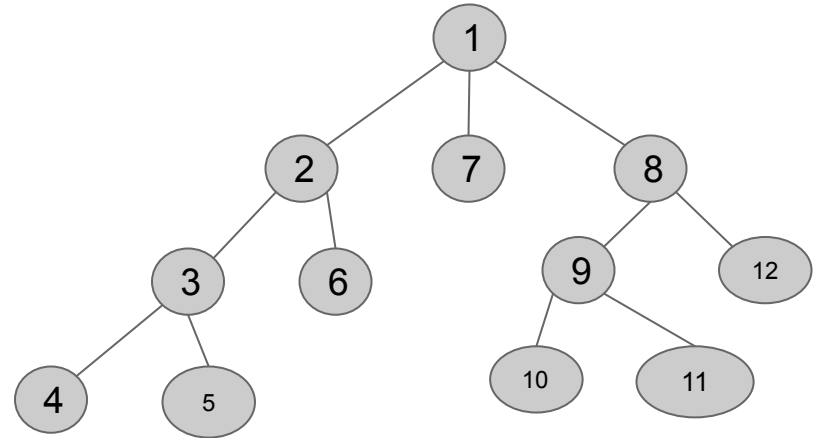


An undirected graph and its depth-first search tree

# Depth-First Search - Illustration

Item popped out

-	[1]
1	[2,7,8]
2	[3,6,7,8]
3	[4,5,6,7,8]
4	[5,6,7,8]
5	[6,7,8]
6	[7,8]
7	[8]
8	[9,12]
9	[10,11,12]
10	[11,12]
12	[]



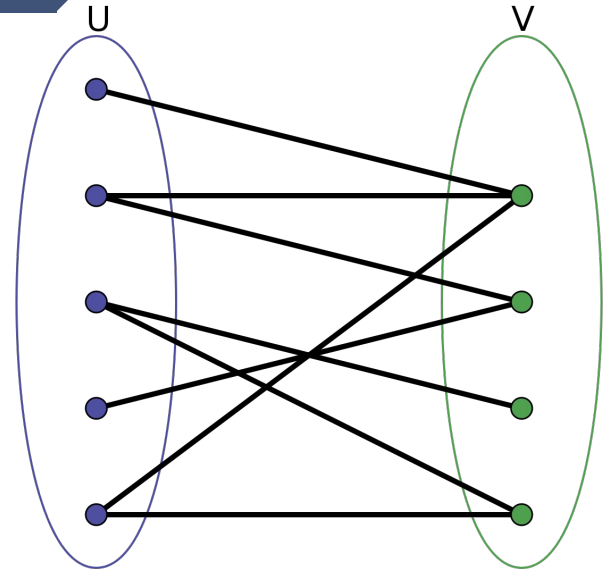
## Applications of DFS

- Detecting cycle in a graph: A graph has a cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.
- Path Finding: We can specialize the DFS algorithm to find a path between two given vertices  $u$  and  $z$ .
- Topological Sorting
- Test if a graph is bipartite
- Finding Strongly Connected Components of a graph



# Bipartite Graph

- Graphs ( $G = (V, E)$ ) in which the vertex set can be partitioned into  $V = L \cup R$ , where  $L$  and  $R$  are disjoint and all edges in  $E$  go between  $L$  and  $R$ . We further assume that every vertex in  $V$  has at least one incident edge



## Applications of DFS (Contd.)



- Solving puzzles with only one solution such as mazes.
  - ▷ A maze is a type of puzzle involving a collection of paths, usually where a player has to find a route from start to finish.
  - ▷ DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.
- Web crawlers
  - ▷ Depth-first search can be used in the implementation of web crawlers to explore the links on a website
- Maze generation
  - ▷ Depth-first search can be used to generate random mazes.
- Model checking
  - ▷ Depth-first search can be used in model checking, which is the process of checking that a model of a system meets a certain set of properties.

## Advantages of DFS

- Memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
- The time complexity of a depth-first Search to depth  $d$  and branching factor  $b$  (the number of children at each node, the outdegree) is  $O(bd)$  since it generates the same set of nodes as breadth-first search, but simply in a different order.
  - ▷ Thus practically depth-first search is time-limited rather than space-limited.

## Advantages of DFS (Contd.)

- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.
- DFS requires less memory since only the nodes on the current path are stored. By chance DFS may find a solution without examining much of the search space at all.

## Disadvantages of DFS

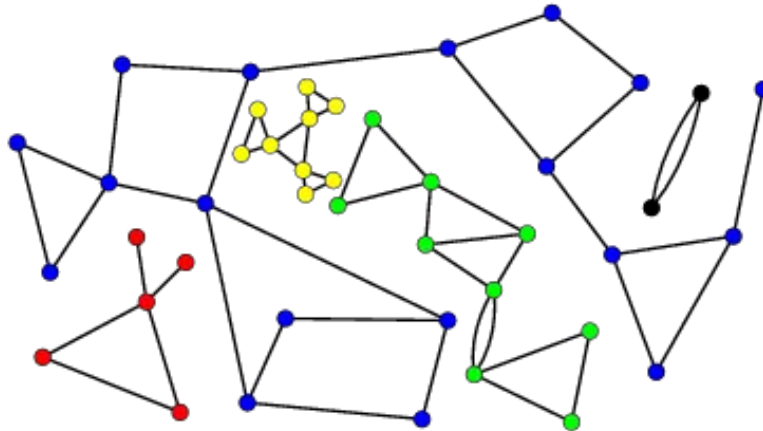
- The disadvantage of Depth-First Search is that there is a possibility that it may down the left-most path forever. Even a finite graph can generate an infinite solution to this problem is to impose a cutoff depth on the search. Although ideal cutoff is the solution depth  $d$  and this value is rarely known in advance of actually solving the problem.
  - ▷ If the chosen cutoff depth is less than  $d$ , the algorithm will fail to find a solution, whereas if the cutoff depth is greater than  $d$ , a large price is paid in execution time, and the first solution found may not be an optimal one.
- Depth-First Search is not guaranteed to find the solution.
- There is no guarantee to find a minimal solution, if more than one solution.

# Important Graph Algorithms

- Connected Components
- Graph Coloring
- Triangle Counting
- Centrality measures

# Connected Components

- The “six degrees of separation” theory argues that there is always a short path linking every two people in the world.
- A graph is connected if there is a path between any two vertices.
- If the theory is true, it means the friendship graph must be connected



## Connected Components (Contd.)

- A connected component of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices.
- The components are separate “pieces” of the graph such that there is no connection between the pieces.
- If we envision tribes in remote parts of the world that have yet not been encountered, each such tribe would form a separate connected component in the friendship graph.
- A remote hermit, or extremely unpleasant fellow, would represent a connected component of one vertex.
- An amazing number of seemingly complicated problems reduce to finding or counting connected components.



## Connected Components (Contd.)

- Connected components can be found using breadth-first search, since the vertex order does not matter.
- We start from the first vertex. Anything we discover during this search must be part of the same connected component.
- We then repeat the search from any undiscovered vertex (if one exists) to define the next component, and so on until all vertices have been found.

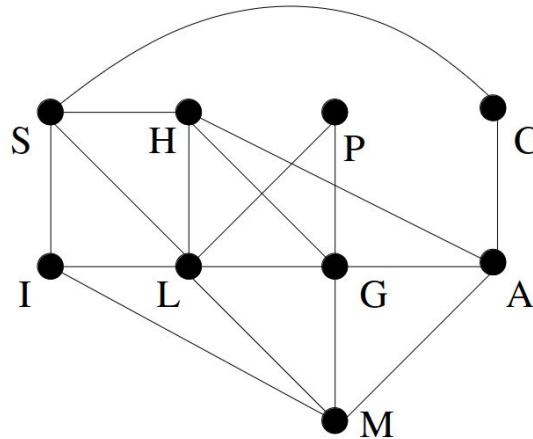
# Graph Coloring

- Suppose that you are responsible for scheduling times for lectures in a university.
- You want to make sure that any two lectures with a common student occur at different times to avoid a conflict.
- We could put the various lectures on a chart and mark with an “X” any pair that has students in common.

Lecture	A	C	G	H	I	L	M	P	S
Astronomy		X	X	X			X		
Chemistry	X								X
Greek	X			X		X	X	X	
History	X		X			X			X
Italian						X	X		X
Latin			X	X	X		X	X	X
Music	X		X		X	X			
Philosophy			X			X			
Spanish		X		X	X	X			

## Graph Coloring (Contd.)

- A more convenient representation of this information is a graph with one vertex for each lecture and in which two vertices are joined if there is a conflict between them
- Now, we cannot schedule two lectures at the same time if there is a conflict, but we would like to use as few separate times as possible, subject to this constraint. How many different times are necessary?

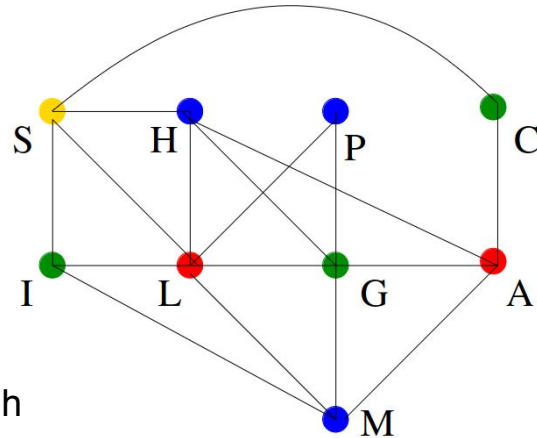


## Graph Coloring (Contd.)

- We can code each time with a color, for example 11:00-12:00 might be given the color green, and those lectures that meet at this time will be colored green.
- The no-conflict rule then means that we need to color the vertices of our graph in such a way that no two adjacent vertices (representing courses which conflict with each other) have the same color.

## Proper Coloring, k-Coloring, k-Colorable

- A proper coloring is an assignment of colors to the vertices of a graph so that no two adjacent vertices have the same color.
- A k-coloring of a graph is a proper coloring involving a total of k colors. A graph that has a k-coloring is said to be k-colorable.



4-coloring of the graph

## The Greedy Algorithm For Coloring Vertices

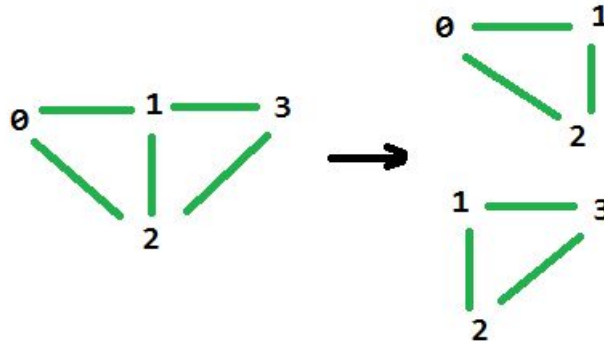
- The algorithm is called greedy because it is a rather short-sighted way of trying to make a proper coloring with as few colors as possible.
- It does not always succeed in finding the minimum number (the chromatic number), but at least provides some proper coloring.
- The procedure requires us to number consecutively the colors that we use, so each time we introduce a new color, we number it also.
  - a. Color a vertex with color 1.
  - b. Pick an uncolored vertex  $v$ . Color it with the lowest-numbered color that has not been used on any previously-colored vertices adjacent to  $v$ .
  - c. Repeat the previous step until all vertices are colored

## The Greedy Algorithm For Coloring Vertices (Contd.)

- Clearly, this produces a proper coloring, since we are careful to avoid conflicts each time we color a new vertex.
- How many colors will be used?
  - ▷ It is hard to say in advance, and it depends on what order we choose to color the vertices.

# Graph Triangle Counting

- Given an undirected graph  $G = (V, E)$ , the triangle counting problem asks for the number of triangles in this graph.
- The density of triangles in the graph is called the global clustering coefficient of a network



Graph with 2 triangles



# Graph Triangle Counting (Contd.)

---

**Algorithm 1:** The Naïve Algorithm for Counting Triangles

---

```
1.1  $C \leftarrow 0$ 
1.2 for  $u = 1 \dots n$  do
1.3   for  $v = i \dots n$  do
1.4     for  $w = j \dots n$  do
1.5       if  $u, v, w$  is a triangle then
1.6          $C \leftarrow C + 1$ 
1.7 return  $C/6$ .
```

---

A first naïve algorithm is to enumerate over all triples  $\{u, v, w\}$  of the vertex set  $V$  and count how many of these form a triangle. (We divide by 6 to compensate for the overcounting we count each triangle  $3! = 6$  times.)

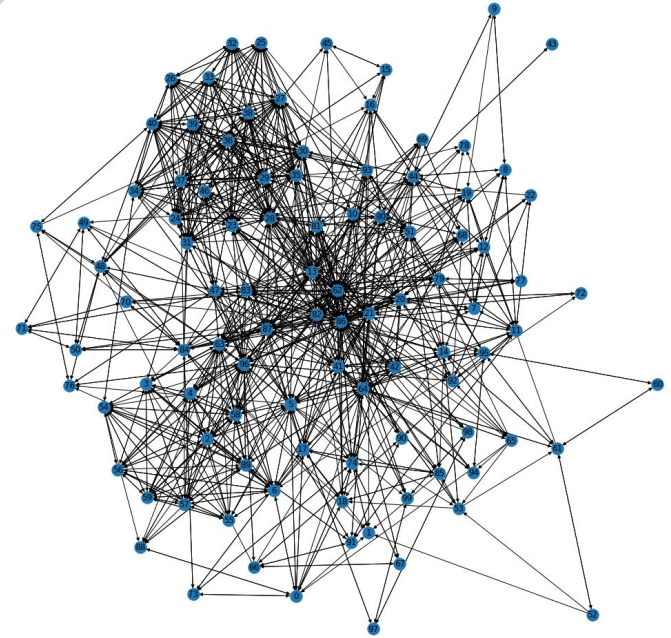
There are  $\binom{n}{3}$  such triples, and if we assume that checking the presence of an edge can be done in constant time, this takes  $O(n^3)$  time

## Graph Centrality Measures

- Centrality is a crucial concept in graph analytics that deals with distinguishing important nodes in a graph.
- Simply put, it recognizes nodes that are important or central among the whole list of other nodes in a graph.
- The different perspectives of a particular node are studied under different indices, which are collectively known as centrality measures

## Graph Centrality Measures (Contd.)

- Betweenness Centrality
- Closeness Centrality
- Degree Centrality
- Closeness Centrality
- Harmonic Centrality
- Eigenvector Centrality
- PageRank



<https://www.harshaash.com/Python/Network%20centrality/>

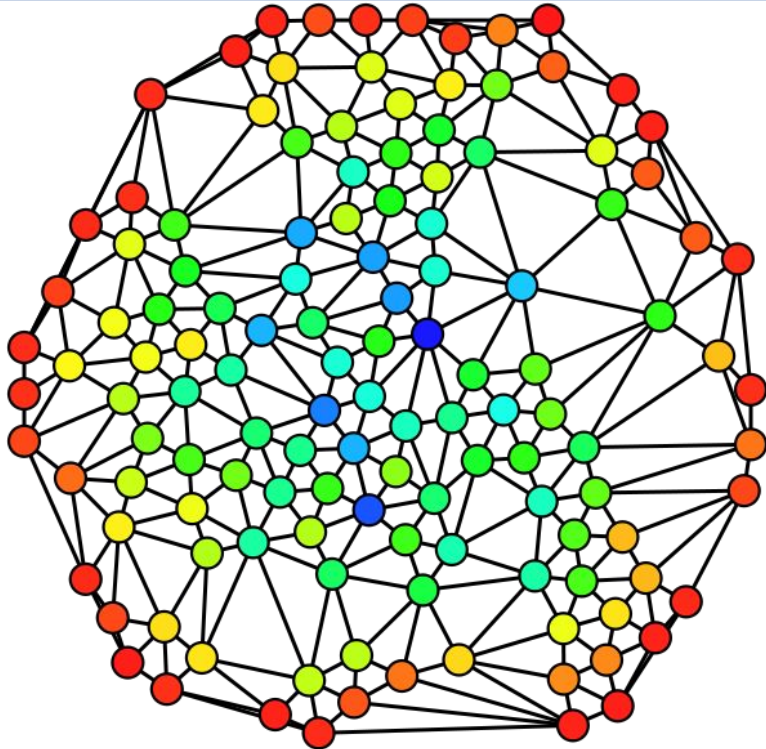
# Betweenness centrality

- Betweenness centrality defines the importance of any node based on the number of times it occurs in the shortest paths between other nodes.
- It measures the percent of the shortest path in a network and where a particular node lies in it.
- A node with high betweenness centrality is considered the most influential one over other nodes in the network. This is because this measure can provide insights into the most critical path as disrupting them will disrupt the network.



The red nodes have high betweenness centrality

## Betweenness centrality (Contd.)



Betweenness: red is minimum; dark blue is maximum

## Betweenness centrality (Contd.)

Directed graph  $G = \langle V, E \rangle$

$\sigma(s, t)$ : number of shortest paths between nodes  $s$  and  $t$

$\sigma(s, t|v)$ : number of shortest paths between nodes  $s$  and  $t$  that pass through  $v$ .

$C_B(v)$ , the betweenness centrality of  $v$ :

$$C_B(v) = \sum_{s, t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}$$

If  $s = t$ , then  $\sigma(s, t) = 1$

If  $v \notin s, t$ , then  $\sigma(s, t|v) = 0$

## Betweenness centrality - Applications

- Betweenness centrality is used to analyze global terrorism networks.
  - ▷ Anti-terrorism agencies utilize the information from these measures to detect and eliminate possible threats.
- It's also used to measure the network flow in telecommunication networks or e-commerce package delivery processes.
- In addition, microbloggers use this centrality to enhance their reach on Twitter with the assistance of a recommendation engine.
  - ▷ It shows who they should connect with to form a bigger communication network.

## Closeness Centrality

- Closeness centrality identifies a node's importance based on how close it is to all the other nodes in the graph.
- The closeness is also known as geodesic distance (GD), which is the number of links included in the shortest path between two nodes.
- To calculate that closeness or GD for a node, sum up all the GD amidst that and all the other nodes in the network on the graph.



## Closeness Centrality - Applications

- Closeness centrality finds application in identifying individuals that are in a position to influence the entire network in the fastest way possible.
- It's also used to identify influences outside a highly connected network.
- Individuals with a high score of this centrality can acquire and control crucial information within the organization.
- A third application is to predict the importance of words in a particular document on the basis of a graph-based keyphrase extraction process.

## Harmonic centrality

- This centrality is a type of closeness centrality.
- GD is measured between nodes.
- The harmonic centrality measures give a more accurate measure of closeness in a case when some of the nodes are outside the perimeter of reach.

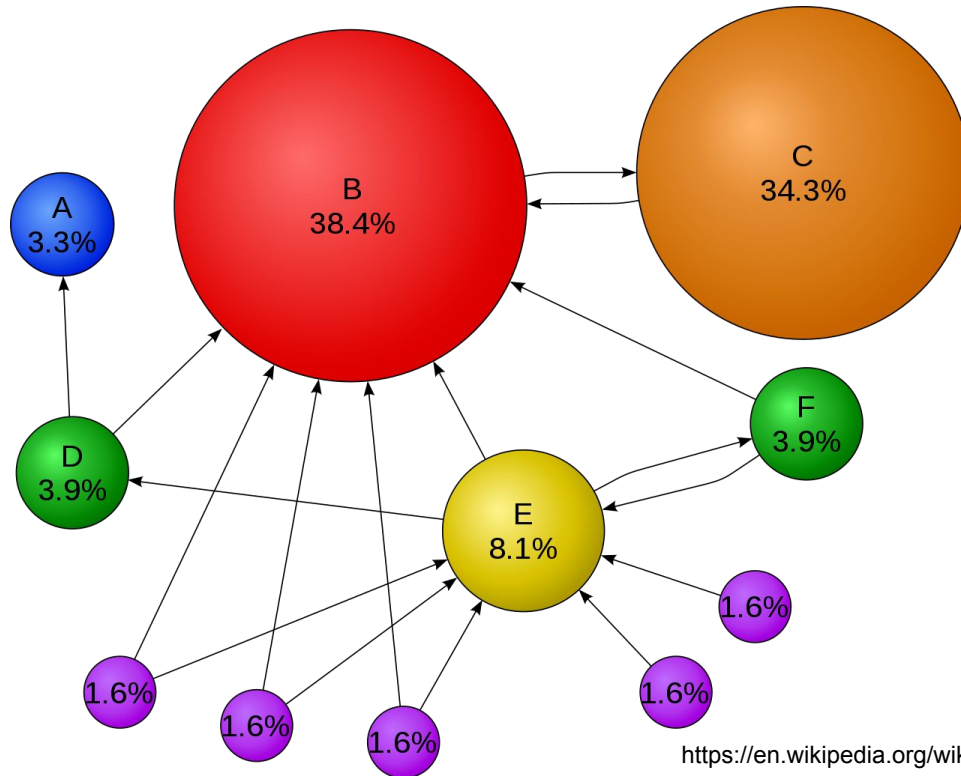
# Eigenvector centrality

- Eigenvector centrality defines a node's importance based on the function of its neighboring nodes. For instance, consider a node in a network. Check all the nodes it's connected to. If a node is linked to or surrounded by highly important nodes in a network, it ought to have a high eigenvector centrality score. It's what makes it an important part of the whole network.
- Relationships with high scoring nodes have more contribution to the score of a node than connections to nodes with low eigenvector centrality scores.
- This centrality identifies nodes that influence the whole network, not only those that are linked. A high score of a node will automatically mean that its neighboring connected nodes have high scores too. It has a wide range of applications, one of which is the calculation of PageRank used by Google. It also finds use in understanding human social networks, malware propagation, etc.

# PageRank

- A subvariant in the eigenvector centrality is PageRank. It's defined as the measure of directional influence of nodes and, thus, is most suited for directed graphs.
- Since eigenvector centrality is suitable for undirected graphs, there wasn't one for directed graphs before PageRank entered the picture. It has applications in online platforms like Twitter, which uses this centrality to offer recommendations of other accounts that a user can follow.
- PageRank is also used to detect flaws in the fraud detection system utilized in the insurance and healthcare industries. The algorithm even predicts traffic workflows in public spaces and streets by running over a graph that includes road intersections.

## PageRank (Contd.)



The percentage shows the perceived importance, and the arrows represent hyperlinks.

# Graph Partitioning

- Graph partitioning addresses the following problem,

Given a problem that can be represented as a graph with nodes and edges, how can we efficiently exploit the concurrency in this problem and map it onto parallel processing elements to guarantee efficient and load balanced execution?

- Many applications can be represented as a graph with a set of nodes connected by edges.
- The nodes and edges usually represent computation and communication.

# Graph Partitioning

- Each node and edge can have a weight that represents a particular cost of executing the computation or communication associated with it.
- In order to efficiently exploit parallelism in such problems, we need to decompose them among processing elements.
- To efficiently execute this application on a parallel platform, the computation must be load-balanced and the communication must be minimized.
- Graph partitioning is used to accomplish this task.

# Graph Partitioning Algorithms

## ■ BFS

- ▷ The well-known BFS (Breadth-First-Search) algorithm can also be used for graph partitioning.
- ▷ BFS algorithm traverses the graph level by level and marks each vertex with the level in which it was visited.
- ▷ After completion of the traversal, the set of vertices of the graph is portioned into two parts  $V1$  and  $V2$  by putting all vertices with level less than or equal to a pre-determined threshold  $L$  in the set  $V1$  and putting the remaining vertices (with level greater than  $L$ ) in the set  $V2$ .
- ▷  $L$  is so chosen that  $|V1|$  is close to  $|V2|$ .



## Graph Partitioning Algorithms (Contd.)

- Kernighan-Lin Algorithm
  - ▷ The Kernighan-Lin algorithm (KL algorithm hereafter) is one of the oldest heuristic graph partitioning algorithms proposed in 1970
  - ▷ In the simplest possible setting, KL algorithm takes an edge-weighted graph  $G = (V, E, c)$  with  $2n$  vertices and an initial bi-partition  $(V_1, V_2)$  of the vertex set  $V$  where  $|V_1| = |V_2| = n$  and produces a new partition  $(V_1', V_2')$  such that  $|V_1'| = |V_2'| = n$  and the total cost of the new partition is lower than (or equal to) the cost of the original partition.
  - ▷ Note that KL algorithm is a balanced partitioning algorithm i.e. the two parts produced by KL algorithm have the same (or almost same, in a more general setting) number of vertices.
  - ▷ It iteratively swaps pairs of vertices until it reaches a locally optimal partition and runs in time  $O(N^3)$  where  $N$  is the number of vertices in  $G$  (for example,  $N = 2n$  when we assume that we started with a graph with  $2n$  vertices).

## Graph Partitioning Algorithms (Contd.)

- Fiduccia-Mattheyses Partitioning Algorithm
  - ▷ Fiduccia-Mattheyses algorithm (FM algorithm hereafter) is another heuristic partitioning algorithm which generalizes the concept of swapping of nodes introduced in KL algorithm.
  - ▷ To contrast FM algorithm with KL algorithm, FM algorithm is designed to work on hypergraphs and instead of swapping a pair of nodes as was happening in KL algorithm, FM algorithm swaps a single node in each iteration.
  - ▷ The basic essence of FM algorithm is the same as KL algorithm – we define gains for each vertex of the (hyper)graph, select one node according to some criterion, remove it from its present partition and put it to the other partition, lock that vertex, update gains of all other unlocked vertices and iterate these steps until we reach a local optimum configuration.
  - ▷ The tool hMETIS implements an augmented version of FM algorithm

## Graph Partitioning Algorithms (Contd.)


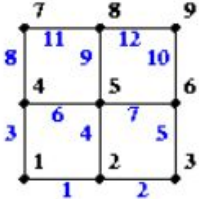
- Spectral Bisection

- ▶ The theory of spectral bisection was developed by Fiedler in 1970 and it was popularized by Pothier, Simon and Liou in 1990.
- ▶ It is based on eigen-vector computation of the 'Laplacian matrix' of the graph under consideration.

## Graph Partitioning Algorithms (Contd.)

- For a graph  $G$ , we define its Laplacian matrix  $L(G)$  in the following way:
  - ▷ The Laplacian matrix  $L(G)$  of a graph  $G = (V, E)$  is a  $|V| \times |V|$  symmetric matrix with one row and one column for each vertex and the entries of the matrix is defined as follows:
    - ▷  $L(G)(i, i) = \text{degree of node } i \text{ (number of incident edges)}$
    - ▷  $L(G)(i, j) = -1$  if  $i \neq j$  and there is an edge  $(i, j)$
    - ▷  $L(G)(i, j) = 0$ , otherwise

# Graph Partitioning Algorithms (Contd.)

Graph G	Laplacian Matrix $L(G)$
	$  \begin{matrix}  & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\  \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix}  1 & -1 & & & \\  -1 & 2 & -1 & & \\  & -1 & 2 & -1 & \\  & & -1 & 2 & -1 \\  & & & -1 & 1  \end{bmatrix}  \end{matrix}  $
	$  \begin{matrix}  & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{matrix} \\  \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix} & \begin{bmatrix}  2 & -1 & & & & & & \\  -1 & 3 & -1 & & & & & \\  & -1 & 2 & & & -1 & & \\  -1 & & & 3 & -1 & & -1 & \\  & -1 & & -1 & 4 & -1 & & -1 \\  & & -1 & & -1 & 3 & & -1 \\  & & & -1 & & & 2 & -1 \\  & & & & -1 & & -1 & 3 & -1 \\  & & & & & -1 & & -1 & 2  \end{bmatrix}  \end{matrix}  $

Two very simple mesh graphs and their Laplacian matrices for the purpose of illustration.

The black numbers (on the vertices) represent the vertex identifies and the blue numbers (on the edges) represent the edge identifiers.

## Graph Partitioning Algorithms (Contd.)

- Spectral Bisection Algorithm

- ▷ Construct the Laplacian matrix  $L(G)$  for the input graph  $G = (V, E)$
- ▷ Compute the eigenvector  $v_2$  corresponding to the second eigenvalue  $\lambda_2$  of the Laplacian matrix  $L(G)$
- ▷ For each vertex  $i \in V$ ,
  - ▷ if  $v_2[i] < 0$  put vertex  $i$  in partition  $V(-)$
  - ▷ else put vertex  $i$  in partition  $V(+)$
- ▷ The key computational steps of this algorithm is determination of  $\lambda_2$  and  $v_2$  of  $L(G)$ . In practice, it is performed using Lanczos algorithm.

## Graph Partitioning Algorithms (Contd.)

- k-way partitioning
  - ▷ K-way partitioning method can be used to directly partition a graph into k sets.
  - ▷ The k-way hypergraph partitioning problem is defined as follows:  
Given a hypergraph  $G=(V, E)$  (where  $V$  is the set of vertices and  $E$  is the set of hyperedges) and an overall load imbalance tolerance  $c$  such that  $c \geq 1.0$ , the goal is to partition the set  $V$  into  $k$  disjoint subsets,  $V_1, V_2, \dots, V_k$  such that the number of vertices in each set  $V_i$  is bounded by  $|V|/(ck) \leq |V_i| \leq c|V|/k$ , and a function defined over the hyperedges is optimized

## Graph Partitioning Algorithms (Contd.)

- k-way partitioning
  - ▷ The requirement that the size of each partition is bounded is referred to as the partitioning constraint, and the requirement that a particular function is optimized is referred to as the partitioning objective.
  - ▷ Some examples of objective functions include minimizing the hyperedge-cut (the total number of hyperedges that span the partition) or to minimize the sum of external degrees (the number of partitions all the hyperedges that cross the partitioning boundary span).



## Popular Tools for Graph Partitioning

- METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering
  - ▷ METIS is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices.
  - ▷ The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes
  - ▷ Provides High Quality Partitions
    - ▷ Experiments on a large number of graphs arising in various domains including finite element methods, linear programming, VLSI, and transportation show that METIS produces partitions that are consistently better than those produced by other widely used algorithms. The partitions produced by METIS are consistently 10% to 50% better than those produced by spectral partitioning algorithms.

# Popular Tools for Graph Partitioning (Contd.)

- METIS

- ▷ Is extremely fast
  - ▷ Experiments on a wide range of graphs has shown that METIS is one to two orders of magnitude faster than other widely used partitioning algorithms. Graphs with several millions of vertices can be partitioned in 256 parts in a few seconds on current generation workstations and PCs.

- ParMETIS

- ▷ ParMETIS [2] is an MPI-based parallel library that includes algorithms for partitioning unstructured meshes and graphs and for computing fill-reducing orderings of sparse matrices.
- ▷ It is designed especially for large numerical simulation codes. The algorithms it implements are based on k-way multilevel graph-partitioning and adaptive repartitioning.
- ▷ It performs graph partitioning quickly, taking advantage of geometry information. It also computes graph repartitioning and refinement and optimizes both the number of vertices that are moved and the edge-cut of the resulting problem.

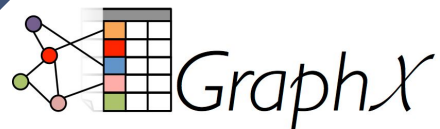
## Popular Tools for Graph Partitioning (Contd.)

- hMETIS
  - ▷ hMETIS is a set of programs developed for partitioning hypergraphs.
  - ▷ This is useful in the VLSI circuit design using multilevel hypergraph partitioning schemes. The algorithms in hMETIS are very fast and are of high quality.
- Zoltan
  - ▷ A tool developed by the Sandia National Laboratories. Zoltan is a collection of data management software for parallel, unstructured, adaptive and dynamic applications.
- Jostle
  - ▷ Software package that is designed to partition unstructured meshes for use on distributed memory parallel computers.
  - ▷ It uses an undirected graph model to represent the mesh and then uses graph partition to partition the mesh across computing nodes.

# Large Graph Processing Platforms

- Pegasus
- Giraph
- Google Pregel
- Apache Spark GraphX
- Parallel Boost Graph Library

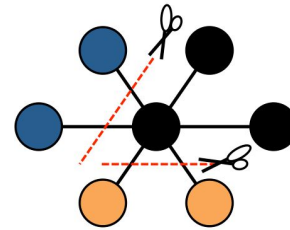
# GraphX



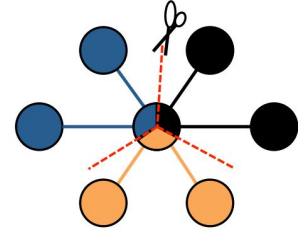
- GraphX is a new component in Spark for graphs and graph-parallel computation.
- At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge.
- To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API.
- In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

## Edge Cut vs Vertex Cut

- GraphX adopts a vertex-cut approach to distributed graph partitioning
- Rather than splitting graphs along edges, GraphX partitions the graph along vertices which can reduce both the communication and storage overhead.
- Logically, this corresponds to assigning edges to machines and allowing vertices to span multiple machines.



Edge Cut



Vertex Cut

## Large Graph Processing with Apache Spark

Download the following compressed file

<https://www.apache.org/dyn/closer.lua/spark/spark-3.5.0/spark-3.5.0-bin-hadoop3.tgz>

A red starburst graphic with multiple points, containing the text "Practical Session".

Practical Session

# Large Graph Processing with Apache Spark (Contd.)

<https://snap.stanford.edu/data/email-Enron.html>

By Jure Leskovec

STANFORD  
UNIVERSITY



## Enron email network

### Dataset information

Enron email communication network covers all the email communication within a dataset of around half million emails. This data was originally made public, and posted to the web, by the Federal Energy Regulatory Commission during its investigation. Nodes of the network are email addresses and if an address  $i$  sent at least one email to address  $j$ , the graph contains an undirected edge from  $i$  to  $j$ . Note that non-Enron email addresses act as sinks and sources in the network as we only observe their communication with the Enron email addresses.

The [Enron email data](#) was originally released by William Cohen at CMU.

### Dataset statistics

Nodes	36692
Edges	183831
Nodes in largest WCC	33696 (0.918)
Edges in largest WCC	180811 (0.984)
Nodes in largest SCC	33696 (0.918)
Edges in largest SCC	180811 (0.984)
Average clustering coefficient	0.4970
Number of triangles	727044
Fraction of closed triangles	0.03015
Diameter (longest shortest path)	11
90-percentile effective diameter	4.8

### Source (citation)

- J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney. [Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters](#). Internet Mathematics 6(1) 29–123, 2009.
- B. Klimmt, Y. Yang. [Introducing the Enron corpus](#). CEAS conference, 2004.

File	Description
<a href="#">email-Enron.txt.gz</a>	Enron email network
<a href="#">Enron email data</a>	Complete Enron email dataset (includes full email message text and attachments)

Download the following file

<https://snap.stanford.edu/data/email-Enron.html>



## Large Graph Processing with Apache Spark (Contd)

```
scala> val graph = GraphLoader.edgeListFile(sc, "Email-Enron.txt")
graph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@67b4b072

scala> graph.vertices.count()
res8: Long = 36692

scala> graph.edges.count()
res9: Long = 367662

scala>
```

**Thank you!**