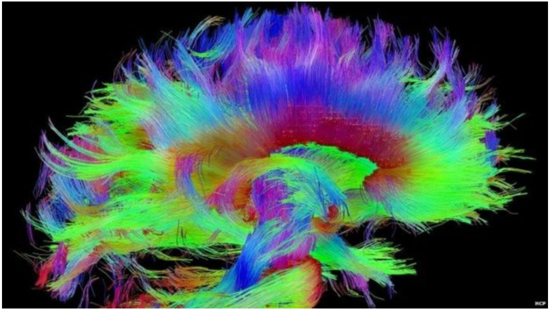
CS 4984: Connectivity Matrices and Node Degrees

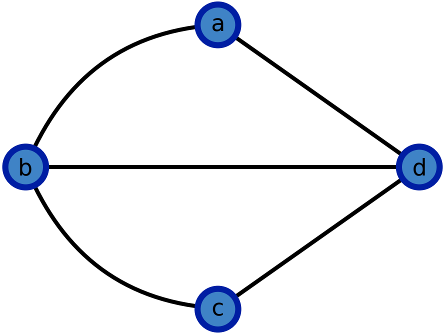
T. M. Murali

February 6, 2018



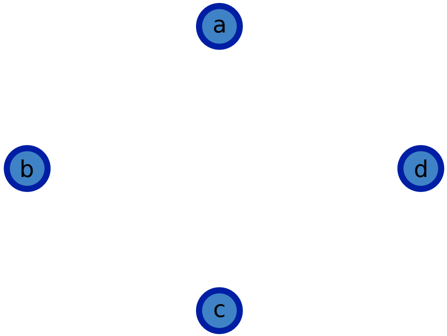
Definition of an Undirected Graph

- *Weighted, undirected graph* $G = (V, E, w)$:
  - set V of nodes.
  - set E of edges.
    - ★ Each element of E is an unordered pair of nodes.
    - ★ Exactly one edge between any pair of nodes (G is not a multigraph).
    - ★ G contains no self loops, i.e., edges of the form (u, u).
  - Each edge (u, v) in E has a weight $w(u, v) \in \mathbb{R}$
    - ★ Weight of each edge is usually positive.
    - ★ G is *unweighted* if all edges have weight 1.



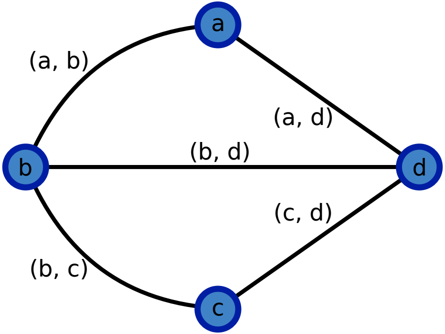
Definition of an Undirected Graph

- *Weighted, undirected graph* $G = (V, E, w)$:
  - set V of nodes.
  - set E of edges.
    - Each element of E is an unordered pair of nodes.
    - Exactly one edge between any pair of nodes (G is not a multigraph).
    - G contains no self loops, i.e., edges of the form (u, u).
  - Each edge (u, v) in E has a weight $w(u, v) \in \mathbb{R}$
    - Weight of each edge is usually positive.
    - G is *unweighted* if all edges have weight 1.



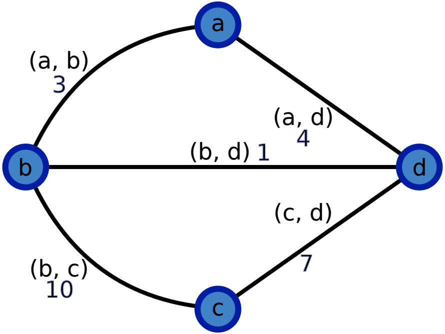
Definition of an Undirected Graph

- *Weighted, undirected graph* $G = (V, E, w)$:
  - set V of nodes.
  - set E of edges.
    - Each element of E is an unordered pair of nodes.
    - Exactly one edge between any pair of nodes (G is not a multigraph).
    - G contains no self loops, i.e., edges of the form (u, u).
  - Each edge (u, v) in E has a weight $w(u, v) \in \mathbb{R}$
    - Weight of each edge is usually positive.
    - G is *unweighted* if all edges have weight 1.



Definition of an Undirected Graph

- *Weighted, undirected graph* $G = (V, E, w)$:
  - set V of nodes.
  - set E of edges.
    - Each element of E is an unordered pair of nodes.
    - Exactly one edge between any pair of nodes (G is not a multigraph).
    - G contains no self loops, i.e., edges of the form (u, u).
  - Each edge (u, v) in E has a weight $w(u, v) \in \mathbb{R}$
    - Weight of each edge is usually positive.
    - G is *unweighted* if all edges have weight 1.



Definition of a Directed Graph

- *Weighted, directed graph* $G = (V, E, w)$:
  - set V of nodes.
  - set E of edges.
    - Each element of E is an ordered pair of nodes.
    - $e = (u, v)$: u is the *tail* of the edge e, v is its *head*; e is *directed from u to v*.
    - A pair of nodes $\{u, v\}$ may be connected by at most two directed edges: (u, v) and (v, u).
    - G contains no self loops.
  - Each edge (u, v) in E has a weight $w(u, v) \in \mathbb{R}$
    - Weight of each edge is usually positive.
    - G is *unweighted* if all edges have weight 1.

Definition of a Directed Graph

- *Weighted, directed graph* $G = (V, E, w)$:
  - set V of nodes.
  - set E of edges.
    - ★ Each element of E is an ordered pair of nodes.
    - ★ $e = (u, v)$: u is the *tail* of the edge e, v is its *head*; e is *directed from u to v*.
    - ★ A pair of nodes $\{u, v\}$ may be connected by at most two directed edges: (u, v) and (v, u).
    - ★ G contains no self loops.
  - Each edge (u, v) in E has a weight $w(u, v) \in \mathbb{R}$
    - ★ Weight of each edge is usually positive.
    - ★ G is *unweighted* if all edges have weight 1.

Definition of a Directed Graph

- *Weighted, directed graph* $G = (V, E, w)$:
  - set V of nodes.
  - set E of edges.
    - Each element of E is an ordered pair of nodes.
    - $e = (u, v)$: u is the *tail* of the edge e, v is its *head*; e is *directed from u to v*.
    - A pair of nodes $\{u, v\}$ may be connected by at most two directed edges: (u, v) and (v, u).
    - G contains no self loops.
  - Each edge (u, v) in E has a weight $w(u, v) \in \mathbb{R}$
    - Weight of each edge is usually positive.
    - G is *unweighted* if all edges have weight 1.

Definition of a Directed Graph

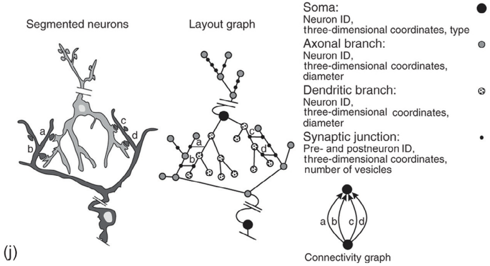
- *Weighted, directed graph* $G = (V, E, w)$:
  - set V of nodes.
  - set E of edges.
    - ★ Each element of E is an ordered pair of nodes.
    - ★ $e = (u, v)$: u is the *tail* of the edge e, v is its *head*; e is *directed from u to v*.
    - ★ A pair of nodes $\{u, v\}$ may be connected by at most two directed edges: (u, v) and (v, u).
    - ★ G contains no self loops.
  - Each edge (u, v) in E has a weight $w(u, v) \in \mathbb{R}$
    - ★ Weight of each edge is usually positive.
    - ★ G is *unweighted* if all edges have weight 1.

Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale		
Mesoscale		
Macroscale		

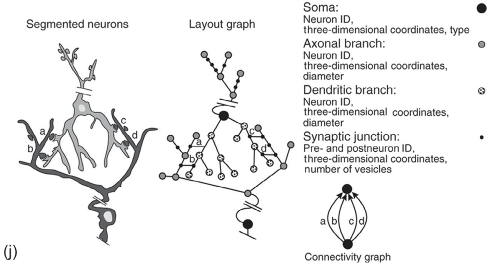
Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons	
Mesoscale		
Macroscale		



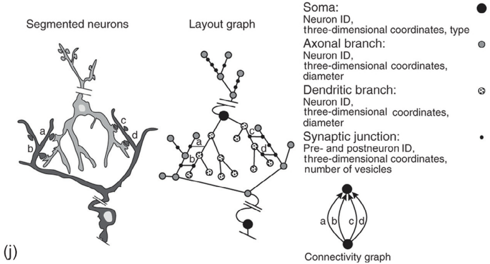
Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons Directed, weighted	
Mesoscale		
Macroscale		



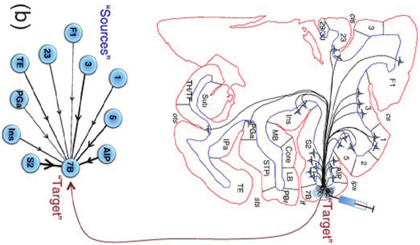
Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons Directed, weighted	Electrodes, correlations Weighted, can be negative, can be directed
Mesoscale		
Macroscale		



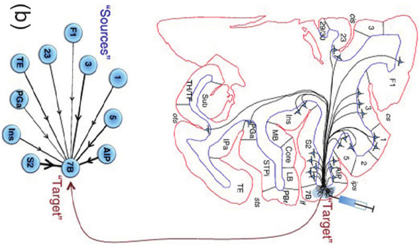
Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons Directed, weighted	Electrodes, correlations Weighted, can be negative, can be directed
Mesoscale	Invasive tract tracing	
Macroscale		



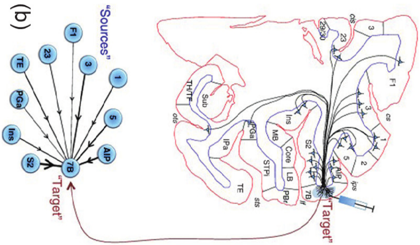
Types of Brain Graphs

	Structural connectivity	**Functional connectivity**
Microscale	SEM, Tracking neurons Directed, weighted	Electrodes, correlations Weighted, can be negative, can be directed
Mesoscale	Invasive tract tracing Directed, weighted	
Macroscale		



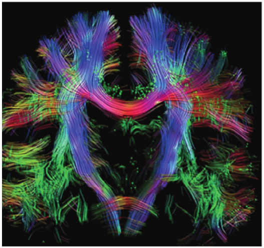
Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons Directed, weighted	Electrodes, correlations Weighted, can be negative, can be directed
Mesoscale	Invasive tract tracing Directed, weighted	Did not discuss
Macroscale		



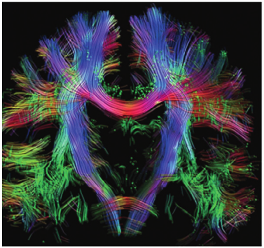
Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons Directed, weighted	Electrodes, correlations Weighted, can be negative, can be directed
Mesoscale	Invasive tract tracing Directed, weighted	Did not discuss
Macroscale	Diffusion MRI, tractography	



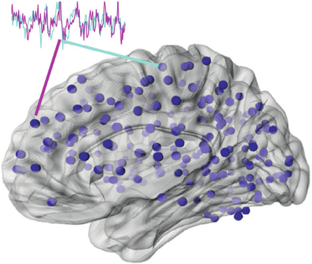
Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons Directed, weighted	Electrodes, correlations Weighted, can be negative, can be directed
Mesoscale	Invasive tract tracing Directed, weighted	Did not discuss
Macroscale	Diffusion MRI, tractography Undirected, weighted	



Types of Brain Graphs

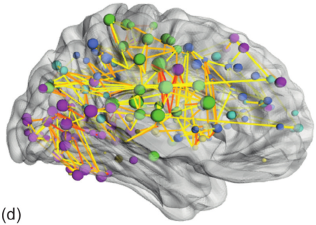
	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons Directed, weighted	Electrodes, correlations Weighted, can be negative, can be directed
Mesoscale	Invasive tract tracing Directed, weighted	Did not discuss
Macroscale	Diffusion MRI, tractography Undirected, weighted	fMRI, correlations



(c)

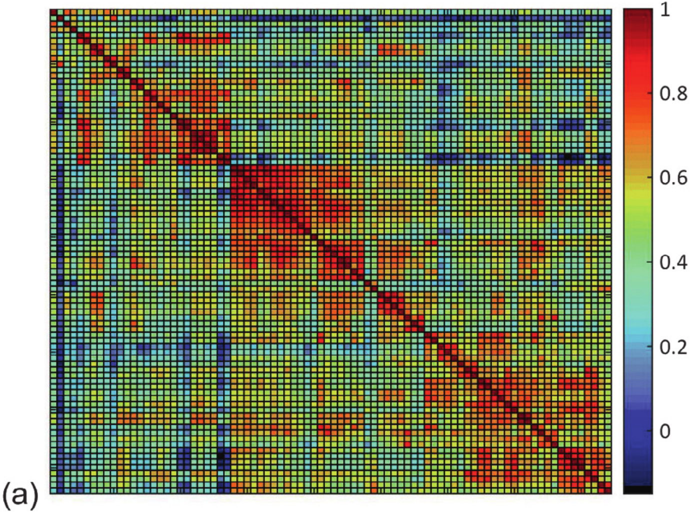
Types of Brain Graphs

	Structural connectivity	Functional connectivity
Microscale	SEM, Tracking neurons Directed, weighted	Electrodes, correlations Weighted, can be negative, can be directed
Mesoscale	Invasive tract tracing Directed, weighted	Did not discuss
Macroscale	Diffusion MRI, tractography Undirected, weighted	fMRI, correlations Weighted, can be negative can be directed



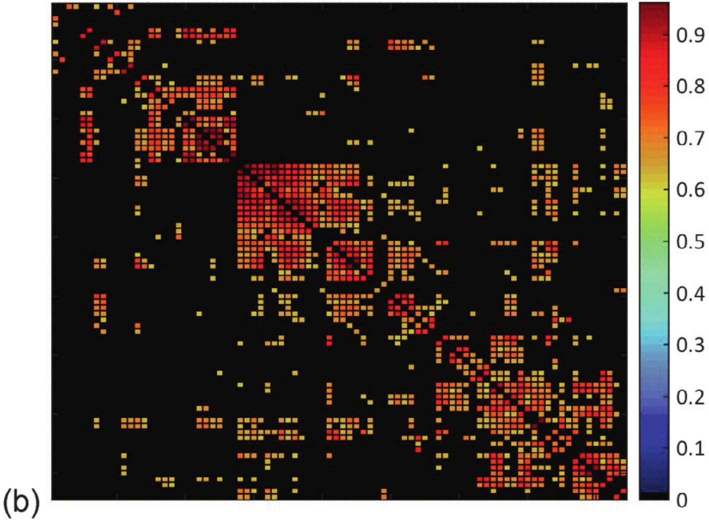
(d)

Thresholding and Binarisation



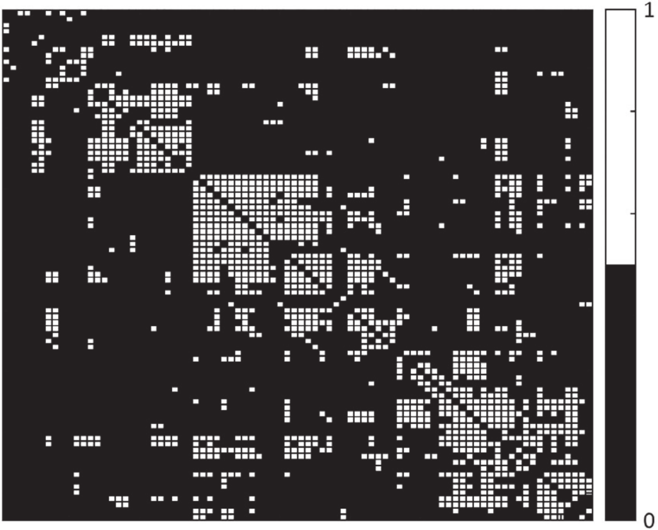
Human functional connectivity matrix from fMRI data.
Every element has a nonzero value.

Thresholding and Binarisation



Matrix after thresholding to retain only the 20% strongest weights.

Thresholding and Binarisation



(c)

Matrix after thresholding and binarisation.

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $O(n)$ time.

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $O(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $O(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $d(v) =$ the number of neighbours of node v.
  - Space used is

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $O(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $d(v) =$ the number of neighbours of node v.
  - Space used is $O\big(n + \sum_{v \in G} d(v)\big) =$

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $O(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $d(v) =$ the number of neighbours of node v.
  - Space used is $O\big(n + \sum_{v \in G} d(v)\big) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node u and node v in

Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $O(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $d(v) =$ the number of neighbours of node v.
  - Space used is $O\big(n + \sum_{v \in G} d(v)\big) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node u and node v in $O(d(u))$ time.
  - Iterate over all the edges incident on node u in

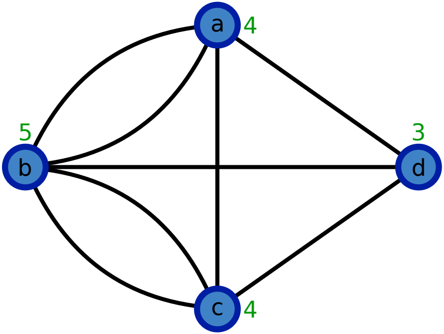
Representing an Undirected Graph

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $O(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $d(v) =$ the number of neighbours of node v.
  - Space used is $O\big(n + \sum_{v \in G} d(v)\big) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node u and node v in $O(d(u))$ time.
  - Iterate over all the edges incident on node u in $O(d(u))$ time.

Representing an Undirected Graph

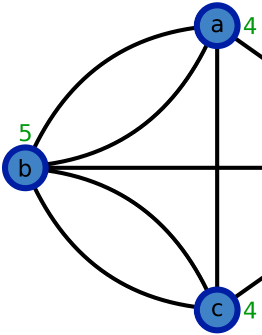
- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - We define the *size* of G to be $m + n$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix* representation: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
  - Space used is $O(n^2)$, which is optimal in the worst case.
  - Check if there is an edge between node i and node j in $O(1)$ time.
  - Iterate over all the edges incident on node i in $O(n)$ time.
- *Adjacency list* representation: array `Adj`, where `Adj`$[v]$ stores the list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.
  - $d(v) =$ the number of neighbours of node v.
  - Space used is $O\left(n + \sum_{v \in G} d(v)\right) = O(n + m)$, which is optimal for every graph.
  - Check if there is an edge between node u and node v in $O(d(u))$ time.
  - Iterate over all the edges incident on node u in $O(d(u))$ time.
- We can modify these ideas for directed graphs.

Implementing Hierholzer's Algorithm



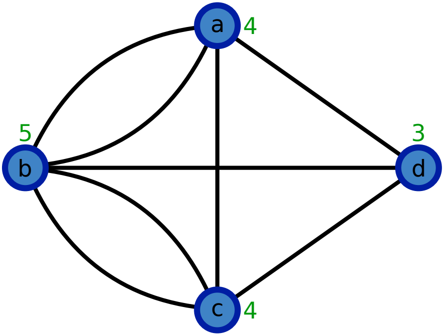
$u \leftarrow s$ #u is the current node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

Implementing Hierholzer's Algorithm



$u \leftarrow s$ $\#u$ is the current node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

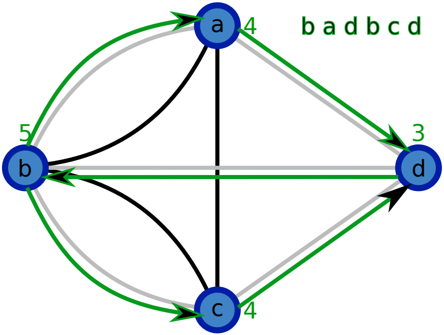
Implementing Hierholzer's Algorithm



$u \leftarrow s$ #u is the current node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

	Adjacency matrix	**Adjacency list**
Determine $d(u)$		
Find a neighbour v of u		
Delete edge (u, v)		

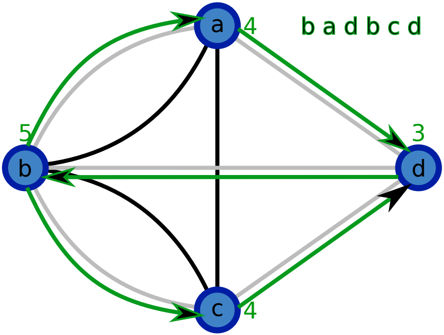
Implementing Hierholzer's Algorithm



$u \leftarrow s$ $\#u$ is the current node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

	Adjacency matrix	**Adjacency list**
Determine $d(u)$	Maintain array of node degrees. $O(1)$ time	Same idea
Find a neighbour v of u		
Delete edge (u, v)		

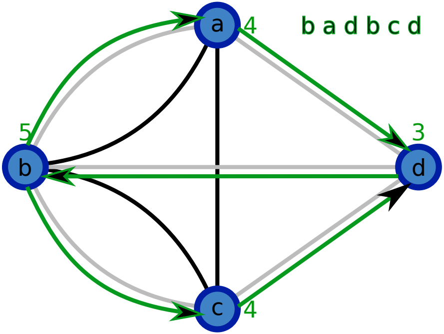
Implementing Hierholzer's Algorithm



$u \leftarrow s$ $\#u$ is the current node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

	Adjacency matrix	**Adjacency list**
Determine $d(u)$	Maintain array of node degrees. $O(1)$ time	Same idea
Find a neighbour v of u	Traverse row for u. $O(n)$ time.	v is first node in $\texttt{Adj}[u]$.
Delete edge (u, v)		

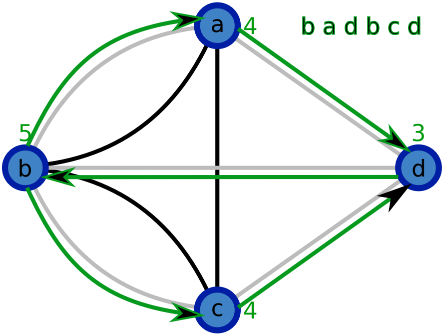
Implementing Hierholzer's Algorithm



$u \leftarrow s$ #u is the current node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

	Adjacency matrix	**Adjacency list**
Determine $d(u)$	Maintain array of node degrees. $O(1)$ time	Same idea
Find a neighbour v of u	Traverse row for u. $O(n)$ time.	v is first node in $\texttt{Adj}[u]$.
Delete edge (u, v)	Set *both* entries to 0; Update $d(v)$. $O(1)$ time.	Delete first element of $\texttt{Adj}[u]$. Update $d(v)$. $O(1)$ time.

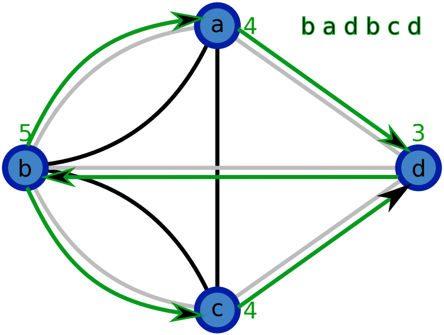
Implementing Hierholzer's Algorithm



$u \leftarrow s$ #u is the current node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

	Adjacency matrix	**Adjacency list**
Determine $d(u)$	Maintain array of node degrees. $O(1)$ time	Same idea
Find a neighbour v of u	Traverse row for u. $O(n)$ time.	v is first node in $\texttt{Adj}[u]$.
Delete edge (u, v)	Set *both* entries to 0; Update $d(v)$. $O(1)$ time.	Delete first element of $\texttt{Adj}[u]$. Update $d(v)$. $O(1)$ time. How do we delete u from $\texttt{Adj}[v]$?

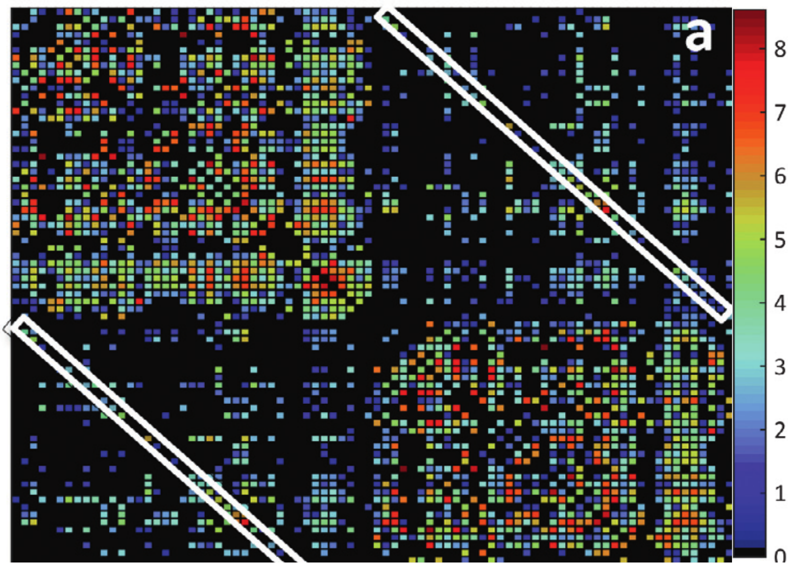
Implementing Hierholzer's Algorithm



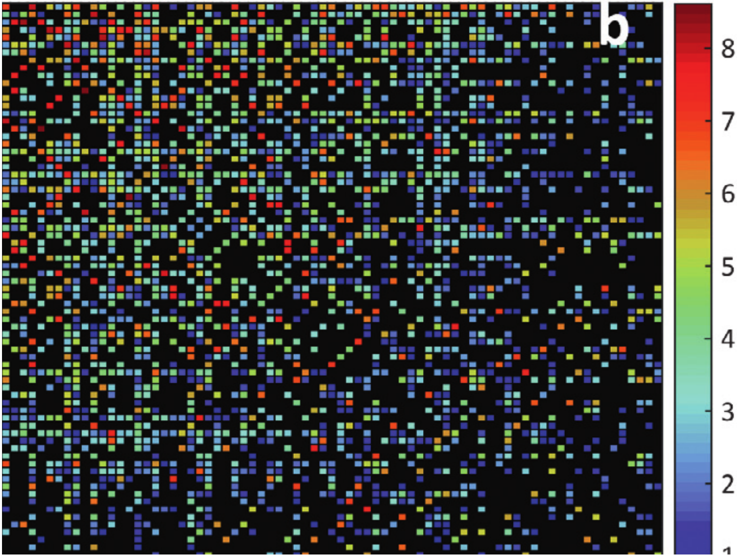
$u \leftarrow s$ #u is the current node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

	Adjacency matrix $O(n^2)$ time.	**Adjacency list** $O(n)$ time.
Determine $d(u)$	Maintain array of node degrees. $O(1)$ time	Same idea
Find a neighbour v of u	Traverse row for u. $O(n)$ time.	v is first node in `Adj`[u].
Delete edge (u, v)	Set *both* entries to 0; Update $d(v)$. $O(1)$ time.	Delete first element of `Adj`[u]. Update $d(v)$. $O(1)$ time. How do we delete u from `Adj`[v]?

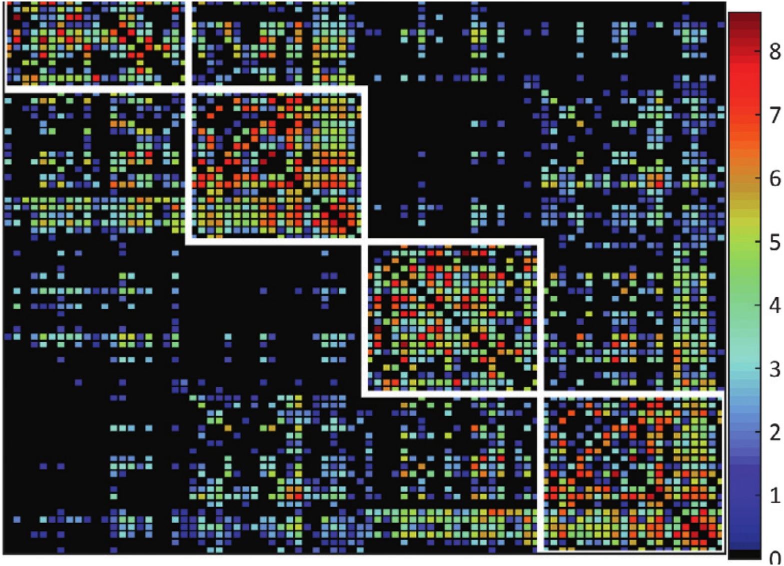
Visualising Matrices



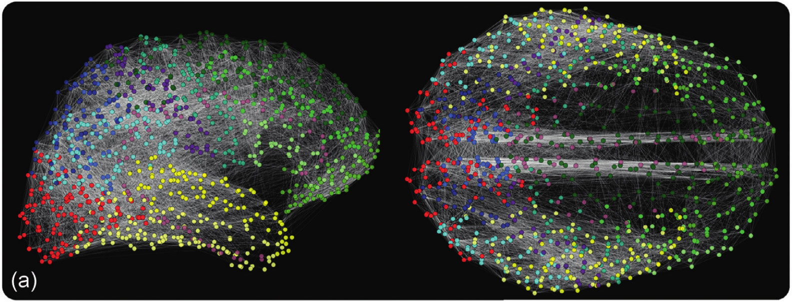
Visualising Matrices



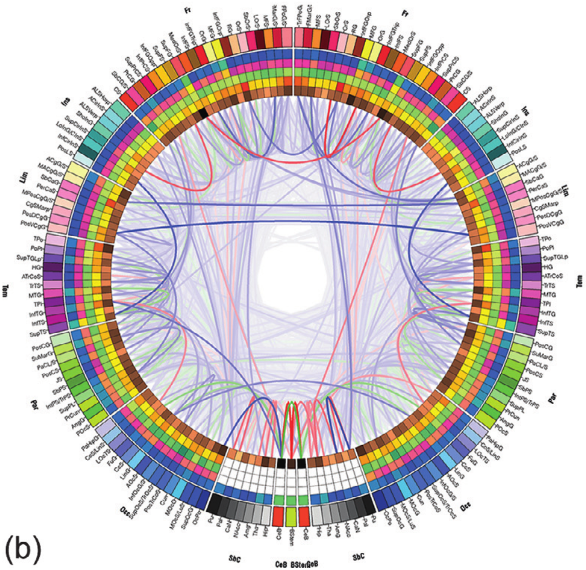
Visualising Matrices



Anatomical Projection

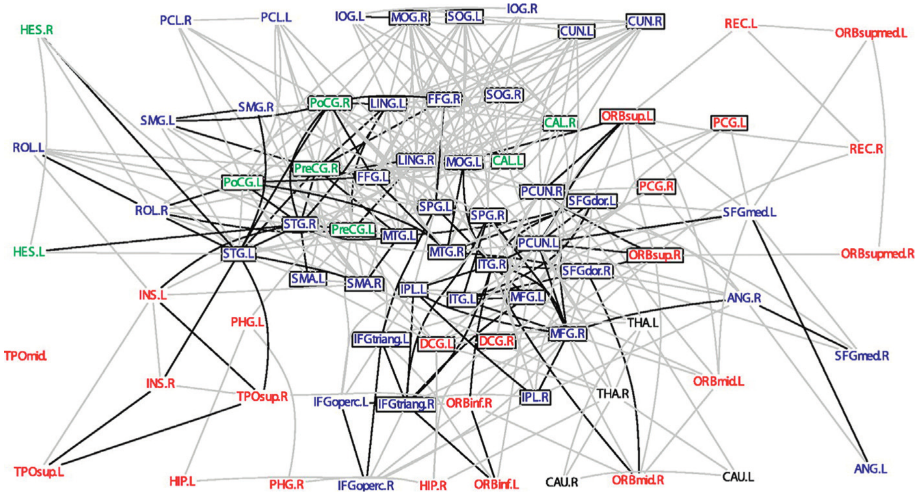


Circular Layout

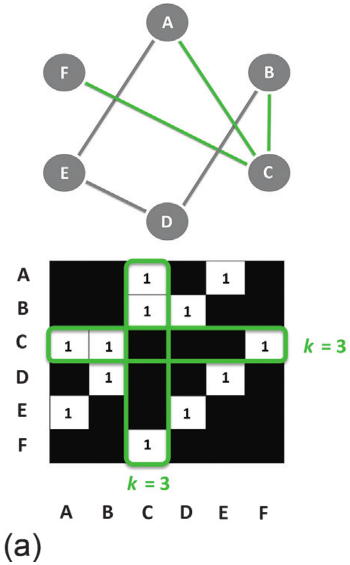


Force-Directed Layout

Spring-Embedded Layout

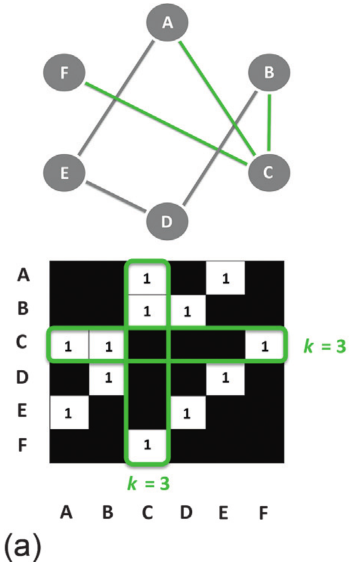


Node Degree



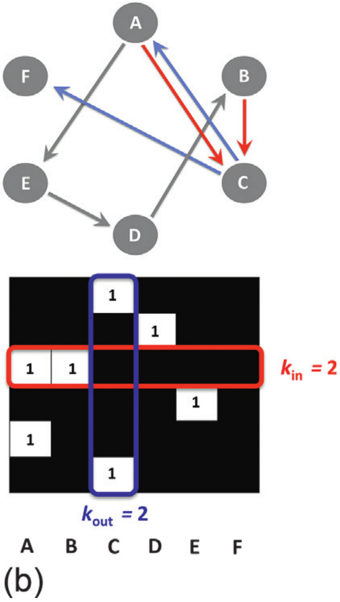
- Undirected graph $G = (V, E)$: *degree* $d(v)$ of a node v is the number of edges in E that are incident on v.

Node Degree



- Undirected graph $G = (V, E)$: *degree* $d(v)$ of a node v is the number of edges in E that are incident on v.
  $d(v) = |\{u \text{ such that } (u, v) \in E\}|$
- Directed graph $G = (V, E)$:

Node Degree



- Undirected graph $G = (V, E)$: *degree* $d(v)$ of a node v is the number of edges in E that are incident on v.

$$d(v) = |\{u \text{ such that } (u, v) \in E\}|$$

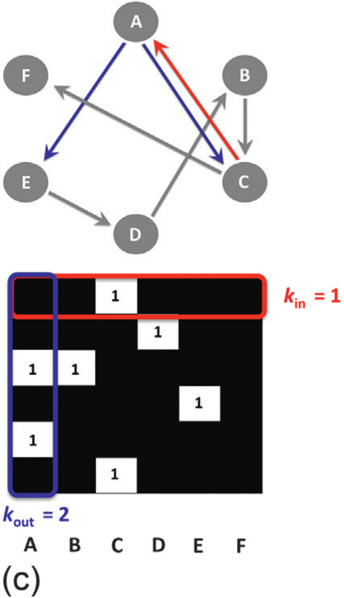
- Directed graph $G = (V, E)$:
  - *in-degree* $d_{in}(v)$ of node v is the number of edges with v as the head.
  - *out-degree* $d_{out}(v)$ of node v is the number of edges with v as the tail.

$$d_{in}(v) = |\{u \text{ such that } (u, v) \in E\}|$$

$$d_{out}(v) = |\{u \text{ such that } (v, u) \in E\}|$$

- Textbook also defines *strength* of a node: total weight of edges incident on that node.

Node Degree



- Undirected graph $G = (V, E)$: *degree* $d(v)$ of a node v is the number of edges in E that are incident on v.

$$d(v) = |\{u \text{ such that } (u, v) \in E\}|$$

- Directed graph $G = (V, E)$:
  - *in-degree* $d_{in}(v)$ of node v is the number of edges with v as the head.
  - *out-degree* $d_{out}(v)$ of node v is the number of edges with v as the tail.

$$d_{in}(v) = |\{u \text{ such that } (u, v) \in E\}|$$

$$d_{out}(v) = |\{u \text{ such that } (v, u) \in E\}|$$

- Textbook also defines *strength* of a node: total weight of edges incident on that node.

Degree Distribution

- A way to summarize information about a graph.

Degree Distribution

- A way to summarize information about a graph.
- *Degree distribution* of an undirected graph G: for every integer $k \geq 0$, the fraction $p(k)$ of nodes in G whose degree is k.

Degree Distribution

- A way to summarize information about a graph.
- *Degree distribution* of an undirected graph G: for every integer $k \geq 0$, the fraction $p(k)$ of nodes in G whose degree is k.
- *Cumulative degree distribution* of G: for every integer $k \geq 0$, the fraction $P(k)$ of nodes in G whose degree is at most k.

Degree Distribution

- A way to summarize information about a graph.
- *Degree distribution* of an undirected graph G: for every integer $k \geq 0$, the fraction $p(k)$ of nodes in G whose degree is k.
- *Cumulative degree distribution* of G: for every integer $k \geq 0$, the fraction $P(k)$ of nodes in G whose degree is at most k.
- Plotting the cumulative degree distribution can offer interesting insights into a graph.

Degree Distribution

- A way to summarize information about a graph.
- *Degree distribution* of an undirected graph G: for every integer $k \geq 0$, the fraction $p(k)$ of nodes in G whose degree is k.
- *Cumulative degree distribution* of G: for every integer $k \geq 0$, the fraction $P(k)$ of nodes in G whose degree is at most k.
- Plotting the cumulative degree distribution can offer interesting insights into a graph.
- What is the value of $\sum_k kp(k)$?

Degree Distribution

- A way to summarize information about a graph.
- *Degree distribution* of an undirected graph G: for every integer $k \geq 0$, the fraction $p(k)$ of nodes in G whose degree is k.
- *Cumulative degree distribution* of G: for every integer $k \geq 0$, the fraction $P(k)$ of nodes in G whose degree is at most k.
- Plotting the cumulative degree distribution can offer interesting insights into a graph.
- What is the value of $\sum_k kp(k)$?
- Define $n(k) = np(k)$, the number of nodes with degree k.

$$\sum_{k \geq 0} kp(k) = \frac{1}{n} \sum_{k \geq 0} kn(k)$$

Degree Distribution

- A way to summarize information about a graph.
- *Degree distribution* of an undirected graph G: for every integer $k \geq 0$, the fraction $p(k)$ of nodes in G whose degree is k.
- *Cumulative degree distribution* of G: for every integer $k \geq 0$, the fraction $P(k)$ of nodes in G whose degree is at most k.
- Plotting the cumulative degree distribution can offer interesting insights into a graph.
- What is the value of $\sum_k kp(k)$?
- Define $n(k) = np(k)$, the number of nodes with degree k.

$$\sum_{k \geq 0} kp(k) = \frac{1}{n} \sum_{k \geq 0} kn(k) = \frac{1}{n} \sum_{v \in V} d(v) = \frac{2m}{n}$$