Information Visualization

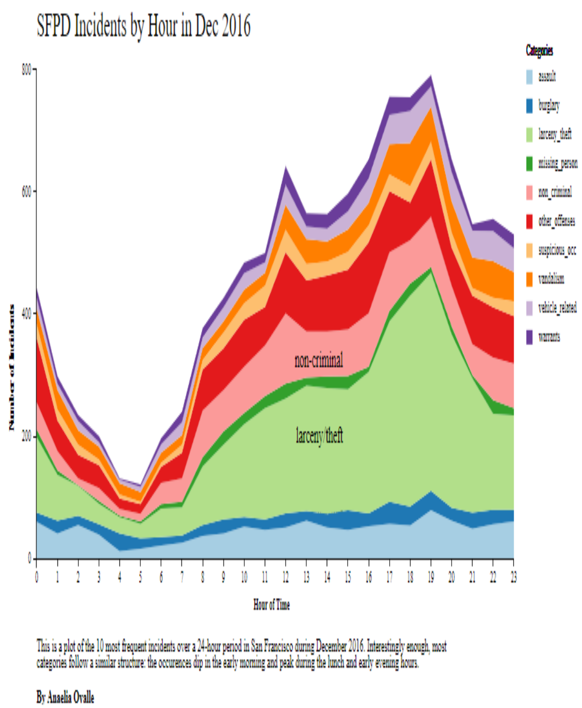
Course Module IN6221

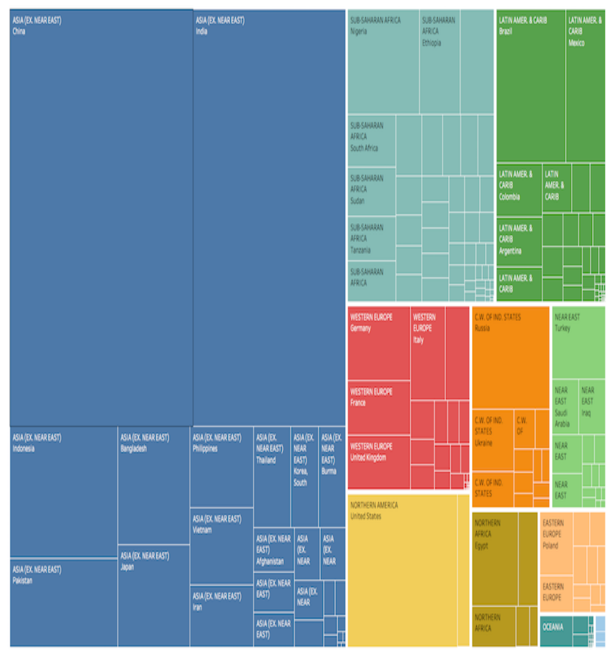
Network Graph Tools

WKW School of Communication and Information,
NTU

D3 Layouts

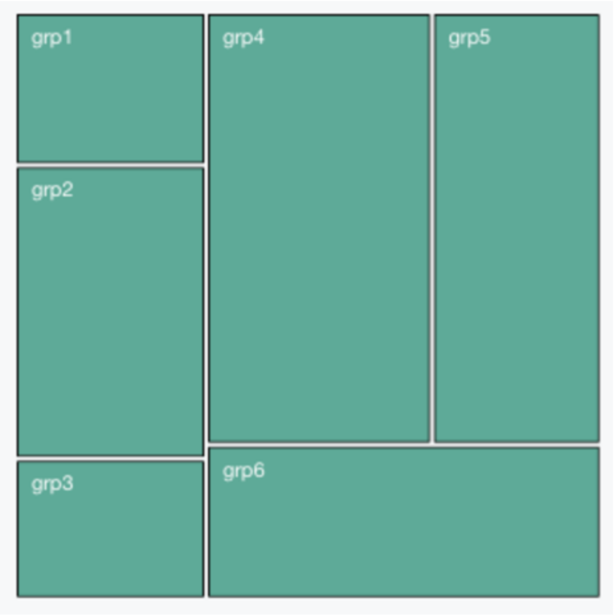
- D3 layout takes data provided and present it using **popular charting methods.**
- Common layouts: **treemap, stack,** and **force**.



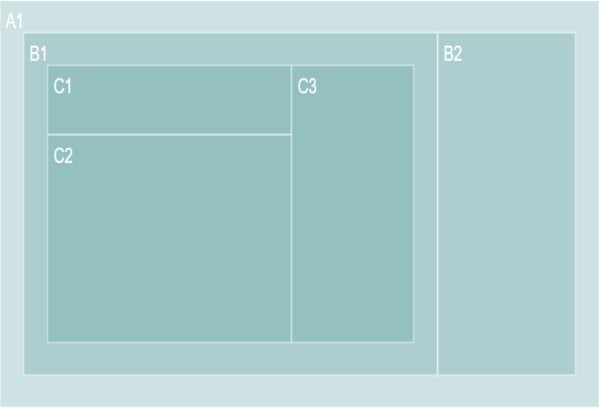


D3 Layout - Treemap

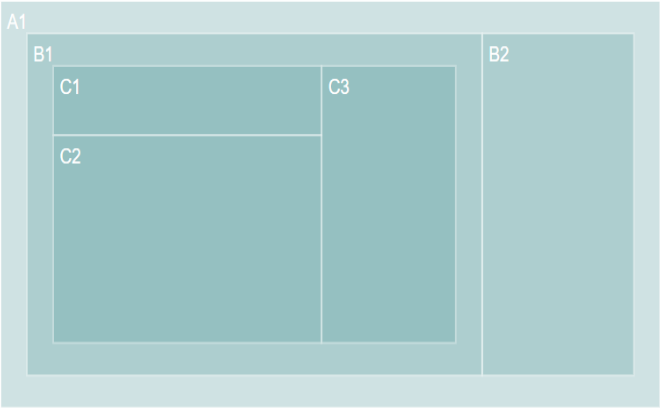
- A **Treemap** displays **hierarchical** data as a set of nested rectangles. Each **group** is represented by a rectangle, which **area** is **proportional** to its value.



- Hierarchical data maps the **parent** to **child relationships**, exists in every system: people in family trees, business org charts, and even categories like the food pyramid.



D3 Layout - Treemap



Define Treemap layout and Hierarchy mapping

```
var treemapLayout = d3.treemap()
  .size([400, 200])
  .paddingOuter(10);
```

Define **treemap** layout

```
var root = d3.hierarchy(data)
```

root => hierarchy object

Hierarchy() – define nested data structure - each node has one **parent** node (**node.parent**), except for **root**; - each node has one or more **child** nodes (**node.children**) except for **leaves**.

Hierarchical dataset

```
var data = {
  "name": "A1",
  "children": [
    {
      "name": "B1",
      "children": [
        {
          "name": "C1",
          "value": 100
        },
        {
          "name": "C2",
          "value": 300
        },
        {
          "name": "C3",
          "value": 200
        }
      ]
    },
    {
      "name": "B2",
      "value": 200
    }
```

```
root.sum(function(d) {
  return d.value;
});
```

Sum each value in data hierarchy – to give parents value e.g., B1 = 100+300+200

```
treemapLayout(root);
```

Bind hierarchy data to **treemap** layout

```
d3.select('svg g')
  .selectAll('rect')
  .data(root.descendants())
  .enter()
  .append('rect')
  .attr('x', function(d) { return d.x0; })
  .attr('y', function(d) { return d.y0; })
  .attr('width', function(d) { return d.x1 - d.x0; })
  .attr('height', function(d) { return d.y1 - d.y0; })
```

returns an array of **descendant** nodes: given node, then each child, and each child's child…

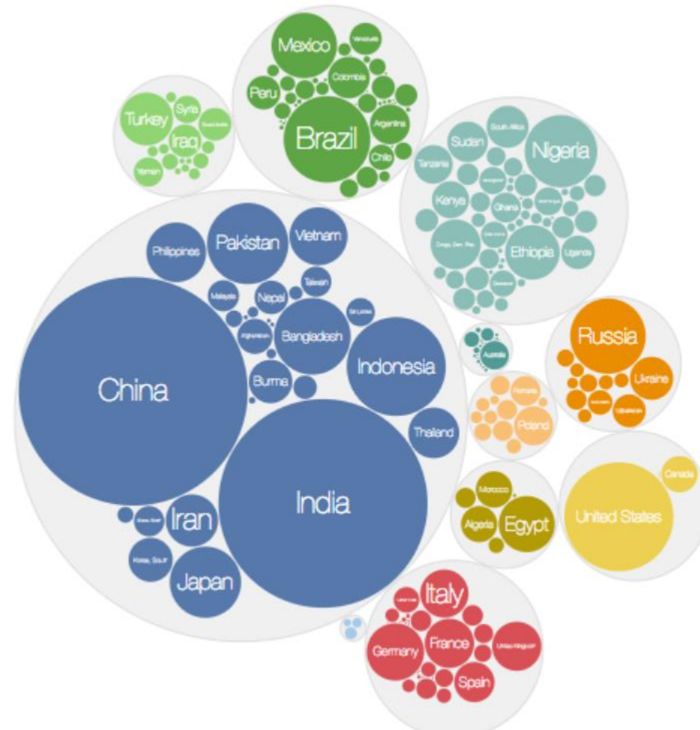
Draw rect

Treemap layout adds 4 properties x0 and y0, x1 and y1 to each node - specify dimensions of each rectangle in treemap

D3 Layout - Pack

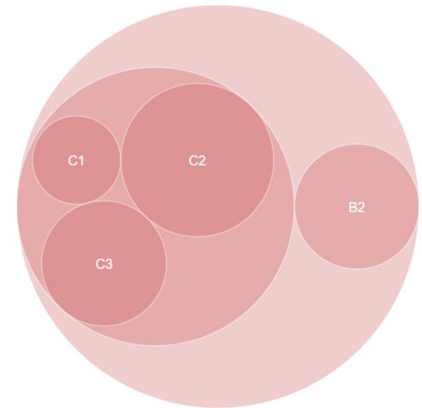
- The pack layout is similar to the tree layout **but circles instead of rectangles** are used to represent nodes.

- In the given example, the **hierarchical structure** is shown in each country represented by a circle (**sized** according to **population**) and the countries are **grouped** by **region**.



D3 Layout - Pack

Define Pack layout and Hierarchy mapping




```
var packLayout = d3.pack()
  .size([300, 300]);
```

Define **pack** layout **object**

```
var rootNode = d3.hierarchy(data)
```

Define **hierarchy** nested data structure

```
rootNode.sum(function(d) {
  return d.value;
});
```

Sum each value in data hierarchy
e.g., B1 = C1+C2+C3

```
packLayout(rootNode);
```

Bind **hierarchy** data to **pack** layout

```
var nodes = d3.select('svg g')
  .selectAll('g')
  .data(rootNode.descendants())
  .enter()
  .append('g')
  .attr('transform', function(d) {return 'translate(' + [d.x, d.y] + ')'})
```

returns an array of **descendant** nodes

Return x and y coordinates for each group circle **from pack layout**

```
nodes
  .append('circle')
  .attr('r', function(d) { return d.r; })
```

Return radius of each group circle **from pack layout**

```
nodes
  .append('text')
  .attr('dy', 4)
  .text(function(d) {
    return d.children === undefined ? d.data.name : '';
  })
```

e.g., A1 not printed as children are defined => only **print nodes without children**

If children **value** and **type** are undefined => data name, else ‘’

=== compares both data value and type, == only compares value

```
<script>
var data = {
  "name": "A1",
  "children": [
    {
      "name": "B1",
      "children": [
        {
          "name": "C1",
          "value": 100
        },
        {
          "name": "C2",
          "value": 300
        },
        {
          "name": "C3",
          "value": 200
        }
      ]
    },
    {
      "name": "B2",
      "value": 200
    }
```

Class Exercise

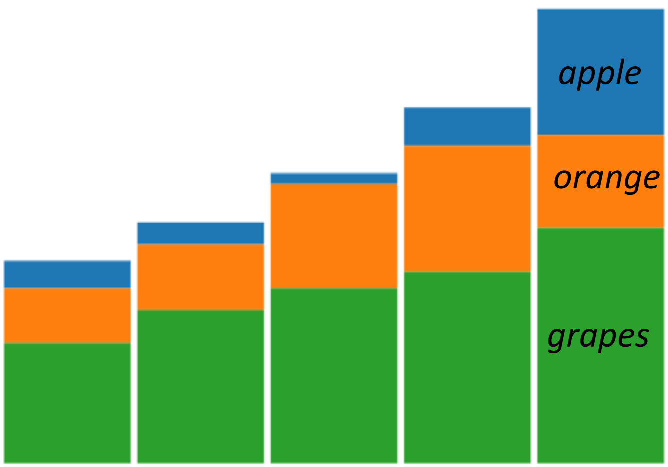
- Create a **Treemap** and a **Pack** Chart
  - Open the following files **3_D3_Layout_Treemap.html** and **3_D3_Layout_Pack.html** in the lab folder.
  - Edit the file codes to create the visualization chart.

```
//var treemapLayout = d3.treemap()
  .size([400, 200])
  .paddingOuter(16);

var rootNode = d3.hierarchy(data)

rootNode.sum(function(d) {
  return d.value;
});
```

```
var packLayout = d3.pack()
  .size([300, 300]);

//var rootNode = d3.hierarchy(data)

rootNode.sum(function(d) {
  return d.value;
});
```

Stack Layout

- **d3.stack()** returns a **stack generator** - converts **two-dimensional** data into "**stacked**" data; it calculates a **baseline** value for each datum (a **fixed starting point** of a scale or operation), so that layers of data can "stack" on top of one another.



A simple stacked bar chart (blue = apples, orange = oranges, green = grapes)

Property (key) value

Object (dictionary)

```
var dataset = [
        { apples: 5, oranges: 10, grapes: 22 },
        { apples: 4, oranges: 12, grapes: 28 },
        { apples: 2, oranges: 19, grapes: 32 },
        { apples: 7, oranges: 23, grapes: 35 },
        { apples: 23, oranges: 17, grapes: 43 }
];

//Set up stack method
var stack = d3.stack()
            .keys([ "apples", "oranges", "grapes" ]);

//Data, stacked
var series = stack(dataset);
```

Specify which **properties** (series) in the dataset to use

Bind dictionary data to **stack** layout

Stack Layout

```
<script type="text/javascript">

    //Width and height
    var w = 500;
    var h = 300;

    //Original data
    var dataset = [
        { apples: 5, oranges: 10, grapes: 22 },
        { apples: 4, oranges: 12, grapes: 28 },
        { apples: 2, oranges: 19, grapes: 32 },
        { apples: 7, oranges: 23, grapes: 35 },
        { apples: 23, oranges: 17, grapes: 43 }
    ];

    //Set up stack method
    var stack = d3.stack()
                .keys([ "apples", "oranges", "grapes" ])
                .order(d3.stackOrderDescending);  // <-- Flipped stacking order

    //Data, stacked
    var series = stack(dataset);

    //Set up scales
    var xScale = d3.scaleBand()
        .domain(d3.range(dataset.length))
        .range([0, w])
        .paddingInner(0.05);

    var yScale = d3.scaleLinear()
        .domain([0,
            d3.max(dataset, function(d) {
                return d.apples + d.oranges + d.grapes;
            })
        ])
        .range([h, 0]);  // <-- Flipped vertical scale
```

console.log(series)

stack(dataset)

Contains stack info

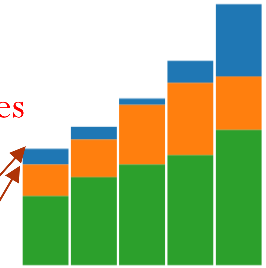
```
▼(3) [Array(5), Array(5), Array(5)]
  ▼0: Array(5)
    ▼0: Array(2)
        0: 32
        1: 37
      ▸data: {apples: 5, oranges: 10, grapes: 22}
       length: 2
      ▸__proto__: Array(0)
```

oranges+grapes

apples+oranges+grapes

5+10+22=37

10+22=32



Define stack

.order(d3.**stackOrderDescending**) => series with **greatest num** (grapes) is at **bottom** of stack

Bind **dataset** to stack which **provides stack info**

Define x and y scales

no. of records = 5

d3.range() [**not your scale range**] returns an array of evenly-spaced numbers=> d3.range(5) => [0,1,2,3,4]

width

specify size of the **gap** between bars.

Get **max of all** the (**summed values**) arrays (records in dataset)

yScale's range [h, 0] => **low values** start at the "**bottom**" of the chart and **increase "up"**.

Stack Layout

```
//Easy colors accessible via a 10-step ordinal scale
var colors = d3.scaleOrdinal(d3.schemeCategory10);

//Create SVG element
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);

// Add a group for each row of data
var groups = svg.selectAll("g")
    .data(series)
    .enter()
    .append("g")
    .style("fill", function(d, i) {
        return colors(i);
    });

// Add a rect for each data value
var rects = groups.selectAll("rect")
    .data(function(d) { return d; })
    .enter()
    .append("rect")
    .attr("x", function(d, i) {
        return xScale(i);
    })
    .attr("y", function(d) {
        return yScale(d[1]);
    })
    .attr("height", function(d) {
        return yScale(d[0]) - yScale(d[1]);
    })
    .attr("width", xScale.bandwidth());

</script>
```

Stacked dataset - contains info (value) of each stack array.

Bind **series (stacked dataset)** to group

Call colors() with data **index** and get color scheme

Groups for each data row

x coordinate of each bar [0,1,2,3,4]

With x,y coordinates draw blue rects followed by orange then green

"Draw" the rect

yScale(32) – yScale(37) => height (no.) of apples

```
▼ (3) [Array(5), Array(5), Array(5)]
  ▼ 0: Array(5)
    ▶ 0: (2) [32, 37, data: {…}]
    ▶ 1: (2) [40, 44, data: {…}]
    ▶ 2: (2) [51, 53, data: {…}]
    ▶ 3: (2) [58, 65, data: {…}]
    ▶ 4: (2) [60, 83, data: {…}]
      index: 2
      key: "apples"
      length: 5
    ▶ __proto__: Array(0)
  ▼ 1: Array(5)
    ▶ 0: (2) [22, 32, data: {…}]
    ▶ 1: (2) [28, 40, data: {…}]
    ▶ 2: (2) [32, 51, data: {…}]
    ▶ 3: (2) [35, 58, data: {…}]
    ▶ 4: (2) [43, 60, data: {…}]
      index: 1
      key: "oranges"
      length: 5
```

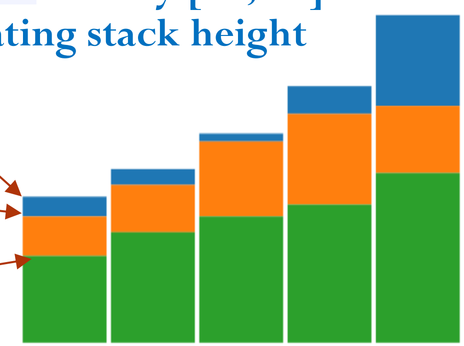
0th array (blue) d[0] = 32 **d[1] = 37**

1st array (orange) d[0] = 22 **d[1] = 32**

2nd array (green) => d[0] =0, d[1]=22

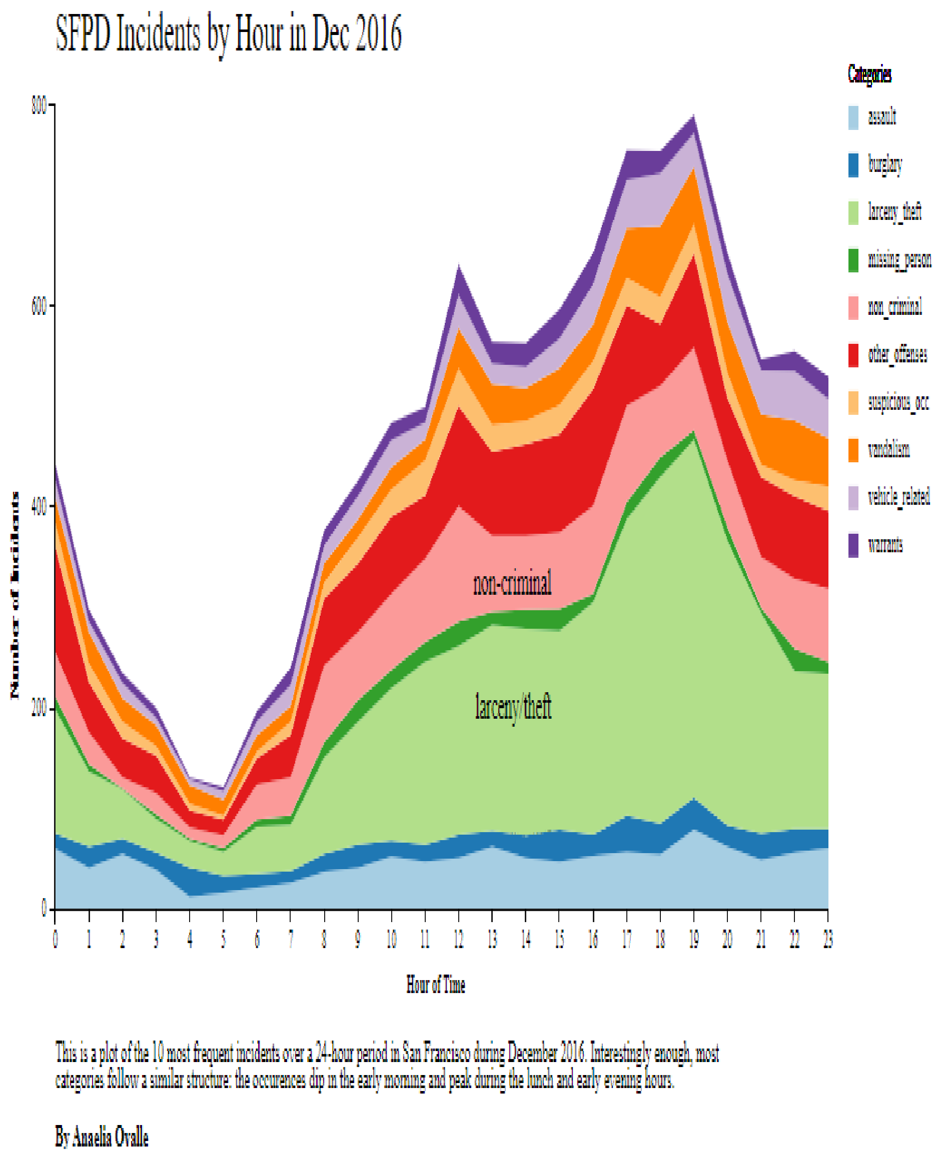
Within each data array there is another array [d0, d1] indicating stack height

[37]
[32]
[22]



Stacked Areas

- Area charts are perfect when communicating the **overall trend**, including **individual values**.
- Use a stacked area chart for **multiple data series** with **part-to-whole relationships** or for cumulative series of values.
- Helps show how **each category contributes** to the **cumulative** total.



hour	assault	burglary	larceny_t	missing_p	non_crimi	other_off	suspicious	vandalism	vehicle_re	warrants
0	60	15	124	12	45	105	23	25	18	16
1	41	21	75	7	32	50	18	32	10	12
2	55	15	49	1	11	39	16	24	15	10
3	39	17	34	4	21	38	9	21	7	10

Stacked Areas

Include the labels

```
svg.append("text")
  .attr("x", 0)
  .attr("y", -40)
  .attr("dy", "0.71em")
  .attr("fill", "#000")
  .text("SFPD Incidents by Hour in Dec 2016")
  .style("font", "23px avenir")
```

above y-origin

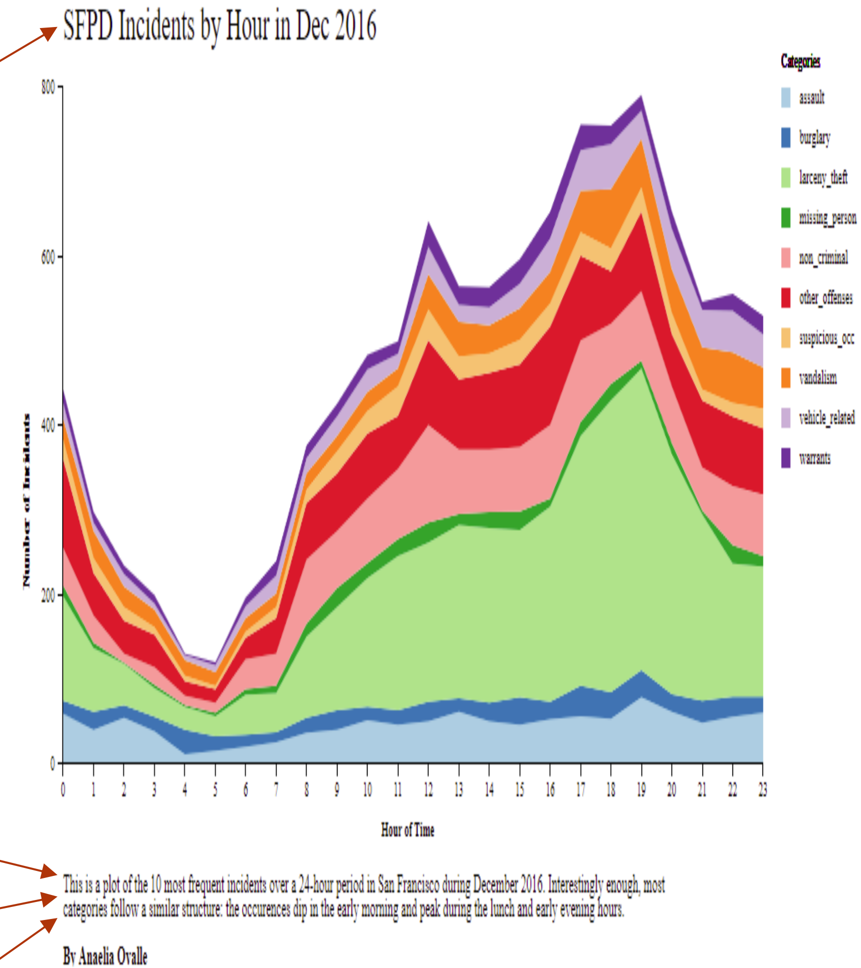
deviation of 0.71 font-size from y

black

```
  svg.append("text")
  .attr("x", 0)
  .attr("y", 402)
  .attr("dy", "0em")
  .style("font", "12px avenir")
  .style("fill", "#000000")
  .text("This is a plot of the 10 most frequent 

  svg.append("text")
  .attr("x", 0)
  .attr("y", 402)
  .attr("dy", "1em")
  .style("font", "12px avenir")
  .style("fill", "#000000")
  .text("categories follow a similar structure: 
```

deviation of 1 font-size

Continuation at 2nd line




```
svg.append("text")
.attr("x", 0)
.attr("y", 402)
.attr("dy", "3em")
.style("font", "12px avenir")
.style("fill", "#000000")
.text("By Anaelia Ovalle")
.style("font-weight", "bold");
```

Stacked Areas

```
<script>
```

Function to assign color based on category for **chart**

```
function get_colors(n) {
var colors = ["#a6cee3","#1f78b4","#b2df8a","#33a02c",
"#fb9a99","#e31a1c","#fdbf6f","#ff7f00","#cab2d6",
"#6a3d9a"];
```

Remainder operator

Return length of array

```
 return colors[ n % colors.length];}
```

=> returns [0 to 9]

e.g., 1 % 10 = 1/10 => 1

```
var margin = {top: 61, right: 140, bottom: 101, left: 50},
    width = 960 - margin.left - margin.right,
    height = 500 - margin.top - margin.bottom;

var  times = ["12am","1a", "2a", "3a", "4a", "5a", "6a",
            "7a", "8a", "9a", "10a", "11a", "12pm", "1p",
            "2p", "3p", "4p", "5p", "6p", "7p", "8p",
            "9p", "10p", "11p"];
```

Time (hour) data in string format

```
var x = d3.scale.linear()
    .range([0, width]);

var y = d3.scale.linear()
    .range([height, 0]);
```

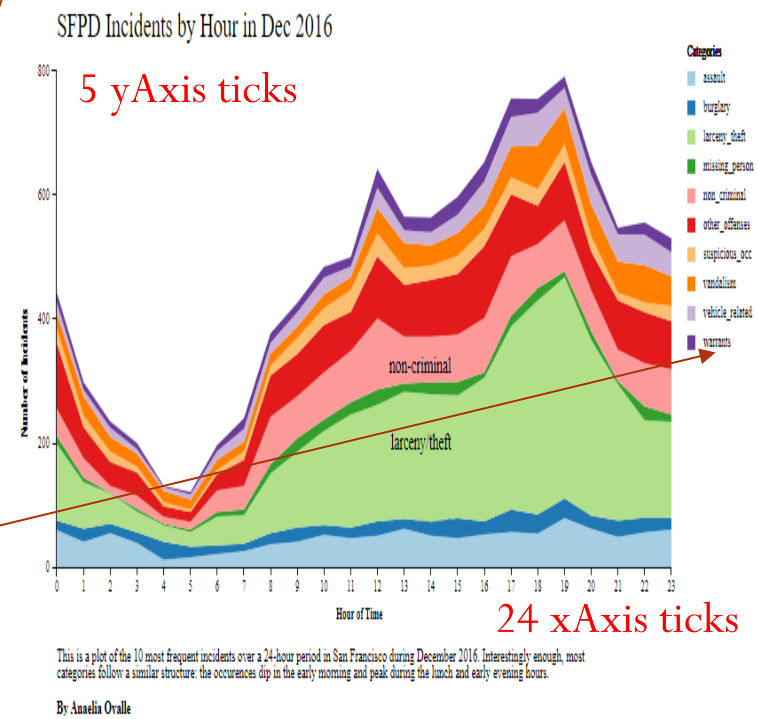
Define **x-scale** and **y-scale** range values

```
var color = d3.scale.category10();
```

Define **color** scale for **legend**

```
var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
        .ticks(24, "s");

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(5, "s");
```

Define **x-axis** and **y-axis** based on scales created

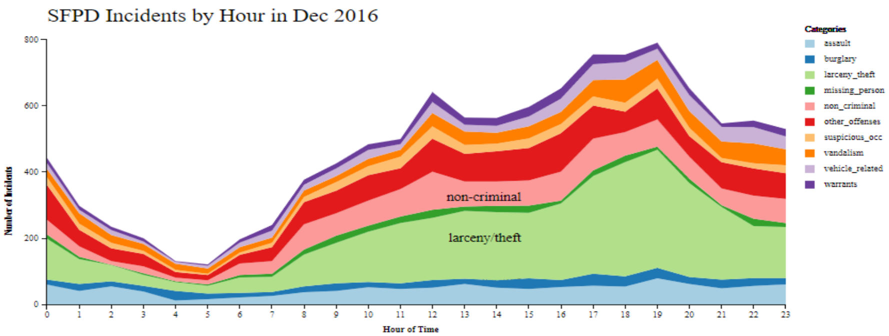


SI前缀格式类型-带有SI前缀（k）的十进制符号，例如，400000 => 400k

(But not applied in example graph)

SI-prefix format type – decimal notation with an SI prefix (k) e.g., 400000 => 400k

Stacked Areas



This is a plot of the 10 most frequent incidents over a 24-hour period in San Francisco during December 2016. Interestingly enough, most categories follow a similar structure: the occurences dip in the early morning and peak during the lunch and early evening hours.

By Anaelia Ovalle

Define Area and Stack

```
var area = d3.svg.area()
    .x(function(d) { return x(d.hour); })
    .y0(function(d) { return y(d.y0); })
    .y1(function(d) { return y(d.y0 + d.y); });
```

hour

value points

直接y(d.y1) 就好了

Difference in height

area function transforms each data point into **information that describes the shape**. Each info corresponds to an **(x)** position, a **lower** y position (y0), and an **upper** y position (y1) – <u>required for **area drawing**</u>

面积函数将每个数据点转换为描述形状的信息。每个信息对应于（x）位置、下y位置（y0）和上y位置（y1）——面积绘制所需

```
var stack = d3.layout.stack()
    .values(function(d) { return d.values; });
```

stack layout - converts **two-dimensional** data into **“stacked”** data - calculates baseline for each datum (a fixed starting point of a scale or operation), so that layers of data can “stack” on top of one another.

```
var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

Create **SVG element**, define width and height and position it.

Stacked Areas

hour	assault	burglary	larceny_th	missing_p	non_crimi	other_off	suspicious	vandalism	vehicle_re	warrants
0	60	15	124	12	45	105	23	25	18	16
1	41	21	75	7	32	50	18	32	10	12

```
d3.csv("data_stackedareachart.csv", function(error, data) {
```

Define domain values for **color scale => using categories names**

```
color.domain(d3.keys(data[0]).filter(function(key) {return key !== "hour"; }));
```

Get first row of data => **header**

Filter data to extract **category names** (excluding hour – **no need color**)

```
data.forEach(function(d) {
  d.hour = +d.hour;
  d.burglary = +d.burglary;
  d.assault= +d.assault;
  d.larceny_theft= +d.larceny_theft;
  d.vehicle_related = +d.vehicle_related;
  d.missing_person = +d.missing_person;
  d.non_criminal = +d.non_criminal;
  d.other_offenses = +d.other_offenses;
  d.suspicious_occ = +d.suspicious_occ;
  d.warrants = +d.warrants;
   });
```

Unary plus (+) convert data **string** to **numbers**

Pre-process data - Set Domain Values

```
// Set domains for axes
x.domain(d3.extent(data, function(d) { return d.hour; }));
y.domain([0, 800])
```

Set x and y scale domain

Return min and max hour

Max y value

Stacked Areas

Input **stack** and **area** info and draw chart using **path** method

Call color.domain() to extract **category names** to stack()

```
var browsers = stack(color.domain().map(function(name) {
  return {
    name: name,
    values: data.map(function(d) {
      return {hour: d.hour, y: d[name] * 1};
    })
  };
}));
```

.map() – calls function for **every array element**

Data in layers

return **hour** and **value** **for each name** based on bound data

Scale factor value (define how wide (high) the stack area should be)

Define browsers variable as **stack object** – convert hour and y value to “**stacked**” data information.

```
var browser = svg.selectAll(".browser")
    .data(browsers)
      .enter().append("g")
    .attr("class", "browser");
```

Create SVG **object** for “browser” class **element** and **bind data** using defined (**browsers) stack object**

```
browser.append("path")
    .attr("class", "area")
    .attr("d", function(d) { return area(d.values); })
    .style("fill", function(d,i) {
        return get_colors(i); });
```

Draw the path (chart)

Bind **d** (browser **stack object)** to defined **area function** => **Path** will **draw area using stack info**

Assign color fill based on **category**

Stacked Areas

Include x axis label

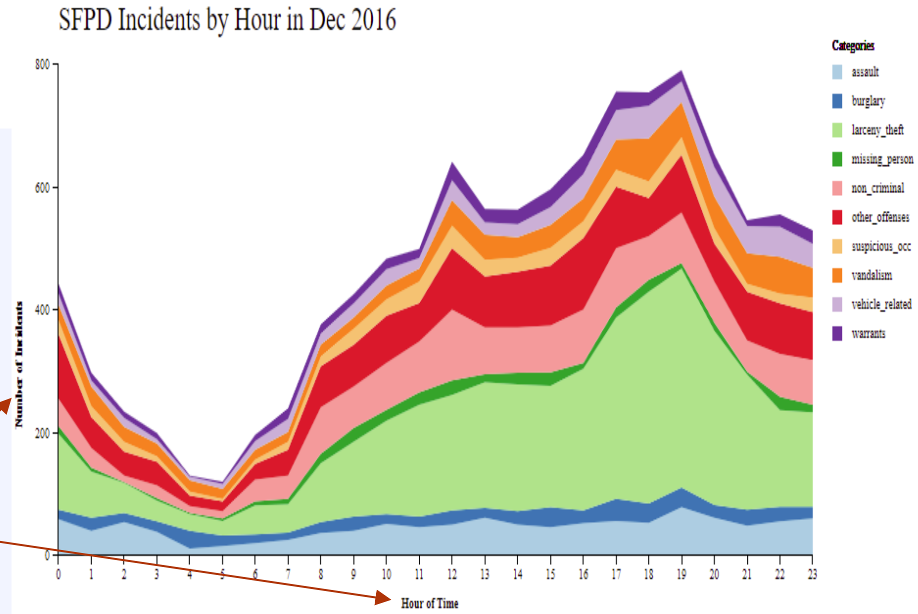
```
svg.append("g")
   .attr("class", "x axis")
   .attr("transform", "translate(0," + height + ")")
   .call(xAxis).append("text")
     .attr("x", 350)
   .attr("y", 36)
   .attr("fill", "#000")
   .text("Hour of Time")
     .style("font-weight", "bold");
```

Reference to translated height

Include y axis label

```
svg.append("g")
   .attr("class", "y axis")
   .call(yAxis)
   .append("text")
   .attr("transform", "rotate(-90)")
     .attr("x", -250)
   .attr("y", -40)
   .attr("dy", "0.3408em")
   .attr("fill", "#000")
   .text("Number of Incidents")
     .style("font-weight", "bold");
```

Reference to rotated object (**-ve** x move **down**, **-ve** y move **left**)



This is a plot of the 10 most frequent incidents over a 24-hour period in San Francisco during December 2016. Interestingly enough, most categories follow a similar structure: the occurences dip in the early morning and peak during the lunch and early evening hours.

By Anaelia Ovalle

Create Legend

d contains category names

```
var legend = svg.selectAll(".legend")
     .data(color.domain()).enter()
     .append("g")
     .attr("class","legend")
  .attr("transform", "translate(" + (width +20) + "," + 0+ ")");
```

Create legend **svg object**

Gives category names

```
legend.append("rect")
  .attr("x", 0)
  .attr("y", function(d, i) { return 20 * i; })
  .attr("width", 10)
  .attr("height", 10)
  .style("fill", function(d, i) {
    return get_colors(i);});

 legend.append("text")
  .attr("x", 20)
  .attr("dy", "0.75em")
  .attr("y", function(d, i) { return 20 * i; })
  .text(function(d) {return d});

 legend.append("text")
  .attr("x",0)
     .attr("dy", "0.75em")
  .attr("y",-10)
  .text("Categories");
```

Create rect for legend

Move space down for each category based on index i

Return category name as text

Class Exercise

- Create a Stacked Area Chart
  - Open the 3_D3_stacked_area_v2.html file in the lab folder.
  - Edit the file codes to create the visualization chart.

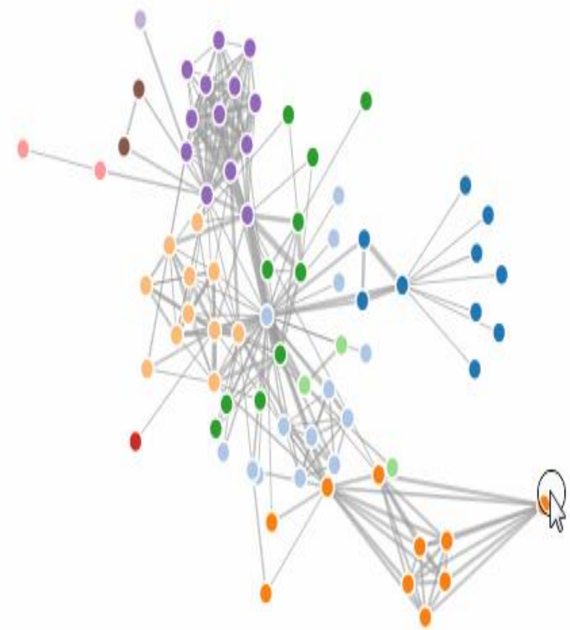
```
//var area = d3.svg.area()
    .x(function(d) { return x(d.hour); })
    .y0(function(d) { return y(d.y0); })
    .y1(function(d) { return y(d.y0 + d.y); });

var stack = d3.layout.stack()
    .values(function(d) { return d.values; });
```

Force Layout

- Force-directed layouts use **simulations** of **physical forces** to **arrange elements** on the screen.

- Force layouts are typically used with **network data (graph).** A simple graph is a list of **nodes** and **edges**. The nodes are **entities** in the dataset, and the edges are the **connections** between nodes.

- Nodes **repel** each other, yet are also **connected** by **springs**. The repelling forces **push** particles away from each other, **preventing visual overlap**, and **springs prevent** them from just **flying out of screen**.



Network Data

- D3's force layout expects us to provide **nodes** and **edges** **separately**, as **arrays** of objects.
- **Dataset object** contains two elements, **nodes** and **edges**, each of which is itself an array of objects.
- The **edges** contain two values each: a **source** ID and a **target** ID.
- These IDs **correspond** to the **preceding nodes**, node 3 is connected to 4 => Donovan is connected to Edward.

```
var dataset = {
    nodes: [
      0   { name: "Adam" },
      1   { name: "Bob" },
      2   { name: "Carrie" },
      3   { name: "Donovan" },
      4   { name: "Edward" },
          { name: "Felicity" },
          { name: "George" },
          { name: "Hannah" },
          { name: "Iris" },
          { name: "Jerry" }
    ],
    edges: [
          { source: 0, target: 1 },
          { source: 0, target: 2 },
          { source: 0, target: 3 },
          { source: 0, target: 4 },
          { source: 1, target: 5 },
          { source: 2, target: 5 },
          { source: 2, target: 5 },
          { source: 3, target: 4 },
          { source: 5, target: 8 },
          { source: 5, target: 9 },
          { source: 6, target: 7 },
          { source: 7, target: 8 },
          { source: 8, target: 9 }
    ]
};
```

Defining the Force Simulation

Call **d3.forceSimulation()** passing in a reference to the **nodes** => generate a new **simulator** and automatically start running it.

To create forces, call **.force()** as many times, each time specifying an **arbitrary name** for each force (to reference it later) and the name of a **force function**.

```
//Initialize a simple force layout, using the nodes and edges in dataset
var force = d3.forceSimulation(dataset.nodes)
            .force("charge", d3.forceManyBody())
            .force("link", d3.forceLink(dataset.edges))
            .force("center", d3.forceCenter().x(w/2).y(h/2));
```

=> Generate simulator

Call **force()** **arbitrary name** **force functions**

The specific **force functions** applied **determine how the network looks**: Are nodes spread out across space, or clustered together.

Types of Forces

- d3.**forceManyBody()** - Creates a "many-body" force that **acts on all nodes** => either **attract all nodes** to each other or **repel all nodes** from each other. Apply different **strength() values** => **Positive** values **attract**; **negative** values **repel**. Default strength() is **–30** => slight repelling force.

```
var force = d3.forceSimulation(dataset.nodes)
                .force("charge", d3.forceManyBody().strength(-15))
                .force("link", d3.forceLink(dataset.edges).distance(30))
                .force("center", d3.forceCenter().x(w/2).y(h/2));
```

- d3.**forceLink(dataset.edges)** - nodes are **connected** by edges with application of this force. Specify a target **distance()** (the default is 30 pixels), and this force will **struggle against any competing forces - to achieve that distance**. Smaller numbers result in shorter edges, larger values in longer edges.

- d3.**forceCenter(x,y)** – translates all nodes to **center** specified with x() and y(). Specify force amount using **forceCenter(x,y).strength(1)**

Types of Forces

- d3.**forceCollide(radius)** – treats **nodes as circles** with **given radius** rather than points, **preventing nodes from overlapping**. Nodes a and b are separated so that **distance between is at least radius(a) + radius(b)**

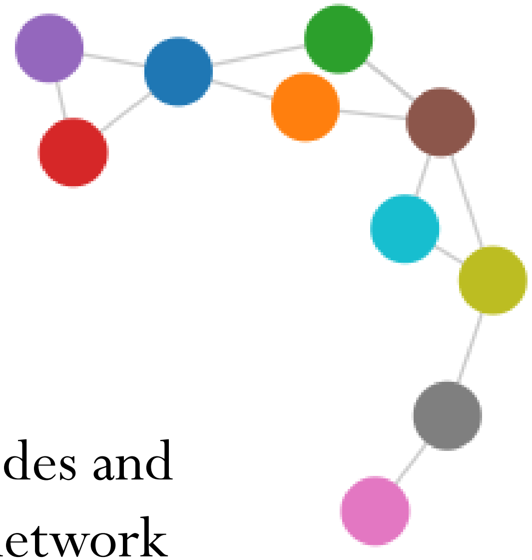
- d3.**forceX(x)** – creates a **positioning force** along the x-axis **towards the given position x** (default is 0)

- d3.**forceY(y)** – creates a **positioning force** along the y-axis **towards the given position y** (default is 0)

Comprehensive list of forces https://github.com/d3/d3-force/blob/master/README.md#forces

Forces

```
<script type="text/javascript">

    //Width and height
    var w = 500;
    var h = 300;

    //Original data
    var dataset = {
        nodes: [
            { name: "Adam" },
            { name: "Bob" },
            { name: "Carrie" },
            { name: "Donovan" },
            { name: "Edward" },
            { name: "Felicity" },
            { name: "George" },
            { name: "Hannah" },
            { name: "Iris" },
            { name: "Jerry" }
        ],
        edges: [
            { source: 0, target: 1 },
            { source: 0, target: 2 },
            { source: 0, target: 3 },
            { source: 0, target: 4 },
            { source: 1, target: 5 },
            { source: 2, target: 5 },
            { source: 2, target: 5 },
            { source: 3, target: 4 },
            { source: 5, target: 8 },
            { source: 5, target: 9 },
            { source: 6, target: 7 },
            { source: 7, target: 8 },
            { source: 8, target: 9 }
        ]
    };
```



Define the nodes and edges in the network

Forces

```
//Initialize a simple force layout, using the nodes and edges in dataset
var force = d3.forceSimulation(dataset.nodes)
                .force("charge", d3.forceManyBody().strength(-15))
                .force("link", d3.forceLink(dataset.edges).distance(30))
                .force("center", d3.forceCenter().x(w/2).y(h/2));

var colors = d3.scaleOrdinal(d3.schemeCategory10);

//Create SVG element
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);

//Create edges as lines
var edges = svg.selectAll("line")
    .data(dataset.edges)
    .enter()
    .append("line")
    .style("stroke", "#ccc")
    .style("stroke-width", 1);

//Create nodes as circles
var nodes = svg.selectAll("circle")
    .data(dataset.nodes)
    .enter()
    .append("circle")
    .attr("r", 10)
    .style("fill", function(d, i) {
        return colors(i);
    })
```

Define **Force Simulation** object – bind data **nodes** to force

Repel (-) all nodes from each other with strength 15

Attracts nodes towards **specified center** point

Force (default 1) pulling on nodes to **keep distance between them at 30px**

Create **edges** as **line elements**

Create **nodes** as **circle elements**

Circles set the same radius, but each gets a different **color** fill

Forces

```
//Every time the simulation "ticks", this will be called
force.on("tick", function() {
    edges.attr("x1", function(d) { return d.source.x; })
         .attr("y1", function(d) { return d.source.y; })
         .attr("x2", function(d) { return d.target.x; })
         .attr("y2", function(d) { return d.target.y; });

    nodes.attr("cx", function(d) { return d.x; })
         .attr("cy", function(d) { return d.y; });
```

Defined **forceSimulation** object

forceSimulation.on(“**tick**”) - runs the force layout simulation **one step**

In a d3 force simulation the **position of each node is updated on every tick**. The tick fires repeatedly throughout the simulation to keep the nodes position updated => fast enough to appear to animate the nodes movement.

“**tick**” refers to the **passage of some amount of time (iterations)** => with each tick, the simulation **adjusts the position** for each node and edge based on values from the callback function

alpha [0,1] defines how far the simulation has progressed. When a simulation **starts**, alpha is set to **1** and this **value decays** based on **alphaDecay** rate until it reaches less than **alphaTarget** => simulation stops.

Default alphaDecay = 0.0228, alphaTarget = 0 **=> default** number of ticks = **300** => adjust **alphaTarget** or **alphaDecay** to adjust simulation duration => **high values shorter simulation**

Draggable Nodes

drag() sets **event listeners** for the **three** (named) drag-related **events** and specifies **functions** to trigger whenever one of those events occurs.

```
//Create nodes as circles
var nodes = svg.selectAll("circle")
    .data(dataset.nodes)
    .enter()
    .append("circle")
    .attr("r", 10)
    .style("fill", function(d, i) {
        return colors(i);
    })
    .call(d3.drag()  //Define what to do on drag events
        .on("start", dragStarted)
        .on("drag", dragging)
        .on("end", dragEnded));
```

Previously defined

sets force layout's **cooling parameter, alpha to 0.3 => decay from 0.3 to 0**

The drag functions are customized as:

dragStarted - when user starts dragging a node, force simulation triggered and **make the node follow the mouse position** (**dragging**).

```
//Define drag event functions
function dragStarted(event, d) {
    if (!event.active) force.alpha(0.3).restart();
    event.fx = event.x;
    event.fy = event.y;
}
```

(get mouse position)

if not active **call alpha** to enable the **tick timer** and restart force layout.

```
function dragging(event, d) {
    d.fx = event.x;
    d.fy = event.y;
}
```

(get mouse position)

dragEnded - once the user lets go, **let simulation resume positioning** in response to forces."

```
function dragEnded(event, d) {
    if (!event.active) force.alphaTarget(0);
    event.fx = null;
    event.fy = null;
}
```

alpha will decay to **alphaTarget value** and stop force simulation.

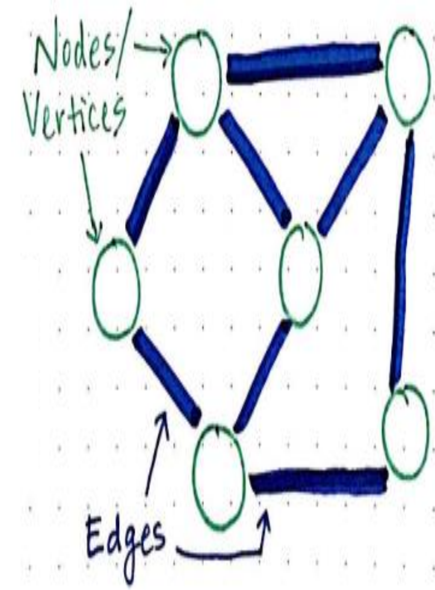
Class Exercise

- Open the 3_D3_force_draggable.html file and edit it to make it work.

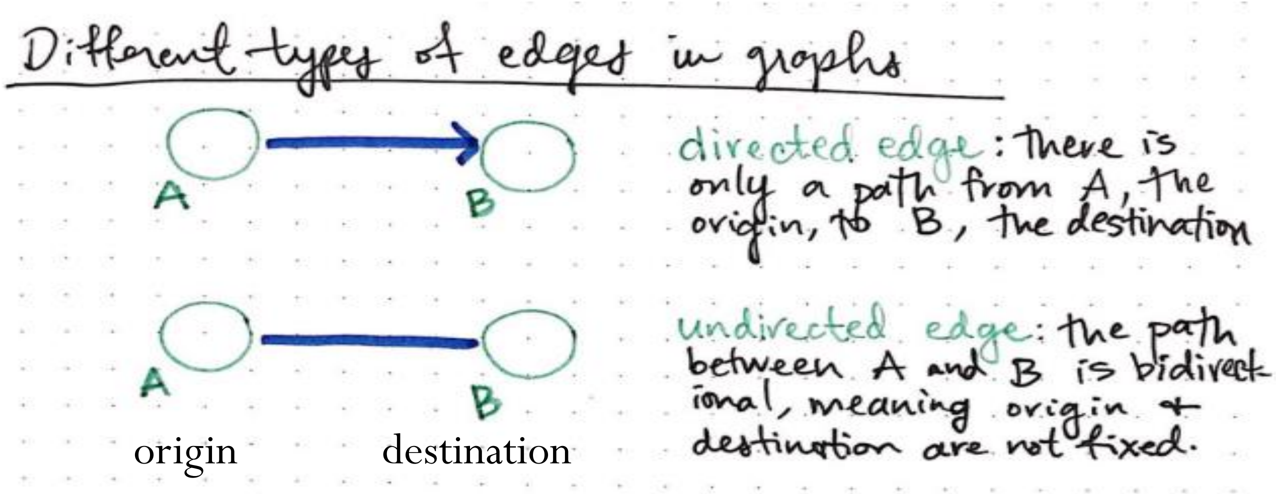
```
//Initialize a simple force layout, using the nodes and edges in dataset
//var force = d3.forceSimulation(dataset.nodes)
                .force("charge", d3.forceManyBody().strength(-100))
                .force("link", d3.forceLink(dataset.edges).distance(30))
                .force("center", d3.forceCenter().x(w/2).y(h/2));

force.alphaDecay(0.1);
force.alphaTarget(1);
```

Introduction to Graph Theory

- Graph theory is the study of graphs, which are mathematical structures used to **model pairwise relations between objects**. A **graph** in this context is made up of **vertices** (aka **nodes** or points) which are connected by **edges** (aka **links** or lines)



- Graphs can have two types of **edges**:
  - an edge that has **a direction** or flow (**directed**), and
  - an edge that has **no direction** or flow (**undirected**).

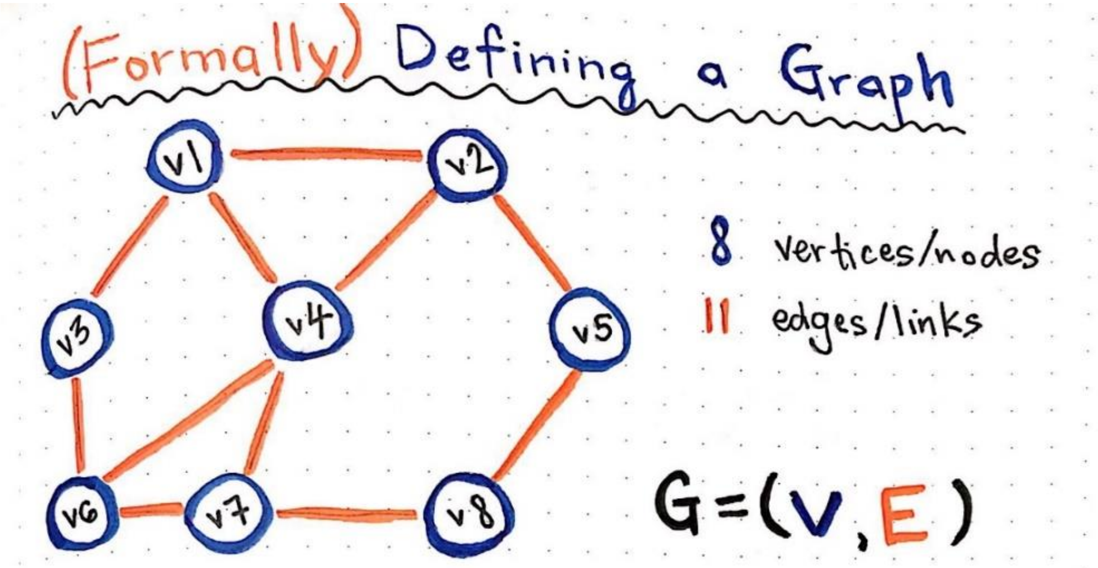


Graph Definition

- Notations of a graph are: **V**, for **vertices**, and **E**, for its **edges**. The formal, mathematical definition for a graph is given as:

$$G = (V, E)$$

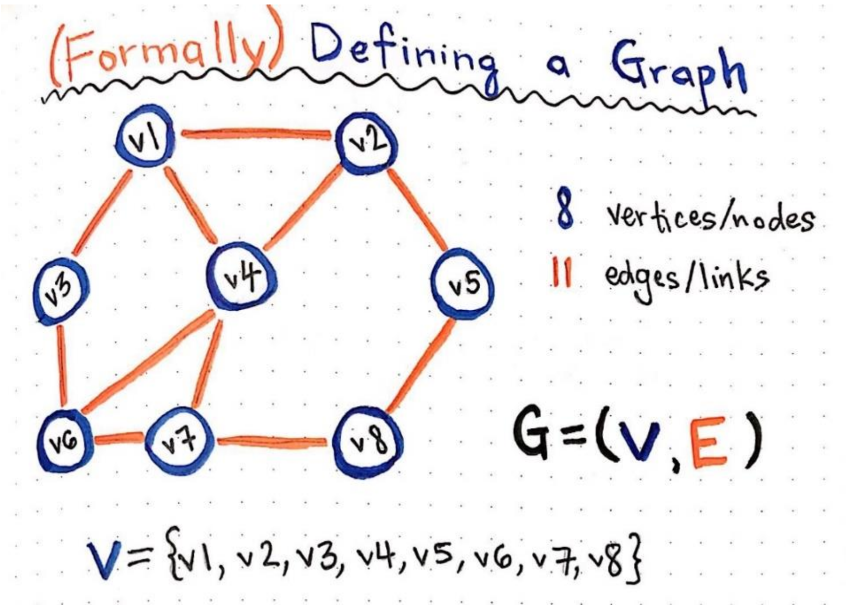
- (V, E) — is made up of **two objects**: a **set of vertices**, and a **set of edges**.

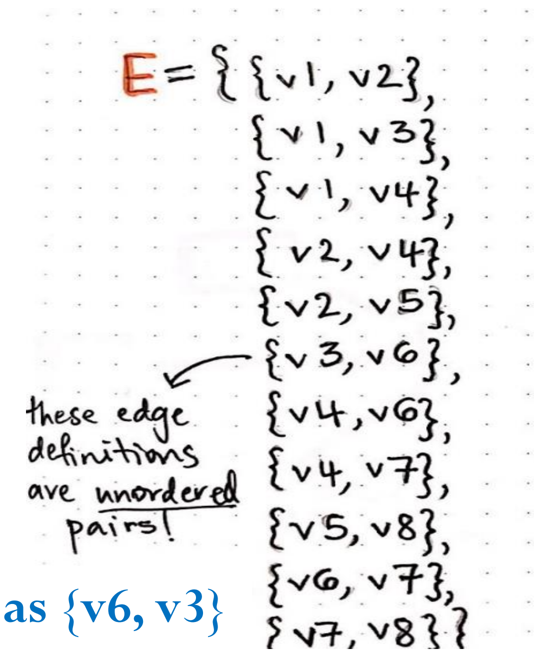


Undirected Graph

- E.g., **V** is defined as an **unordered** set of references to the 8 vertices. The nodes are "unordered" as there is **no hierarchy of nodes** (order doesn't matter)

- This is an **undirected** graph, which means that the edges are **bidirectional** and the **origin** node and **destination** node are **not** fixed. So, each of the edge objects are also **unordered pairs**.

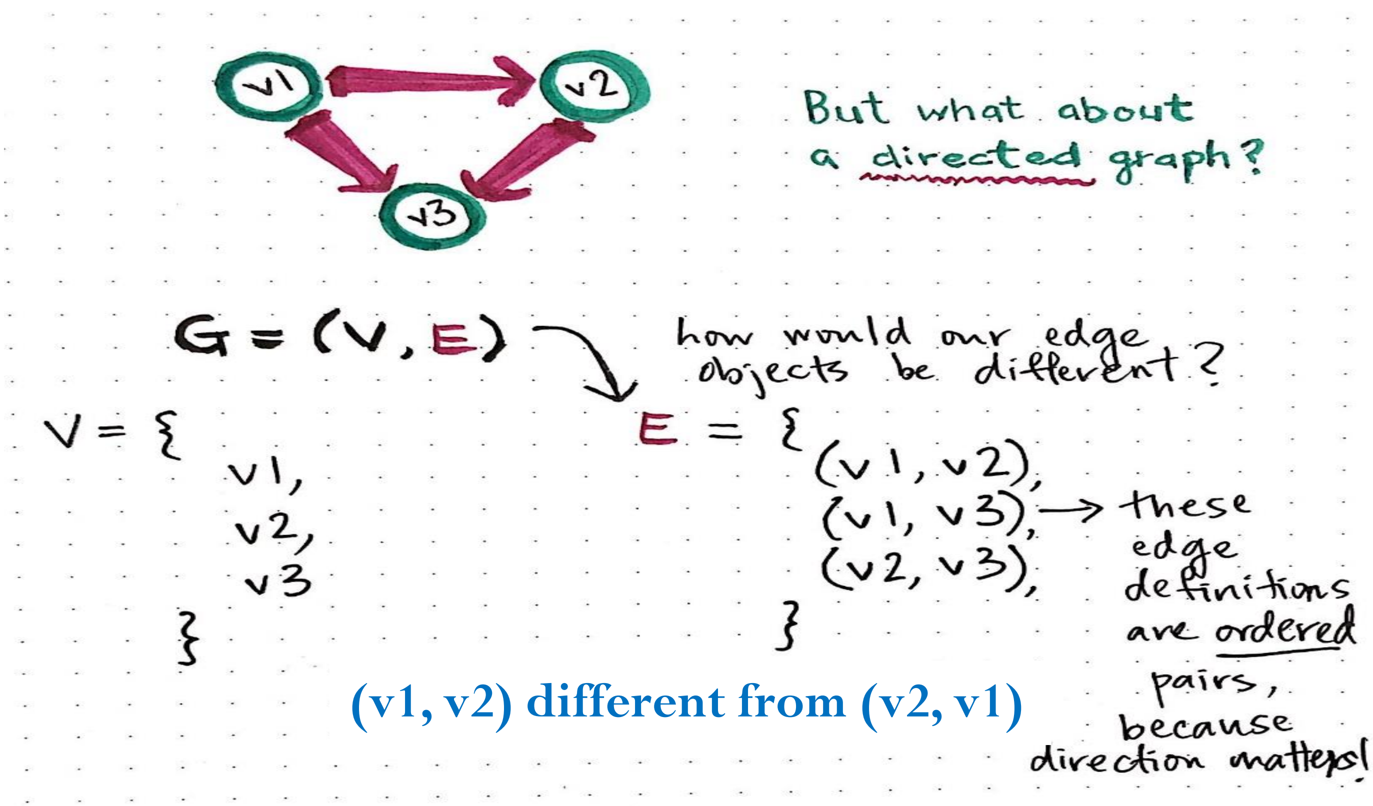




{v3, v6} same as {v6, v3}

Directed Graph

- The edge objects in the **directed** graph are **ordered** pairs, because **direction matters** in this case. Direction is only from the origin node to the destination node => edges ordered, such that the **origin** node is the **first**, and **destination** node the **second**.



(v1, v2) different from (v2, v1)

How to differentiate between ordered and unordered pairs?

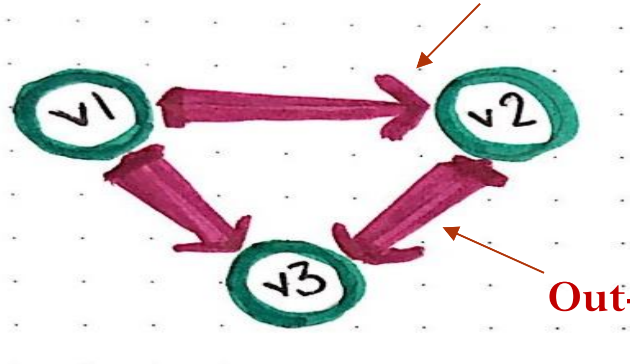
Degree of a node

- In a **non-directed graph**, **degree of a node** is defined as the **number of direct connections** a node has with other nodes.

- In a **directed graph** (each edge has a **direction**), degree of a node is further divided into **In-degree** and **Out-degree**. In-degree refers to the number of edges/connections **incident on it** and Out-degree refers to the number of edges/connections **from it to other nodes**

In-degree of V2 = 1

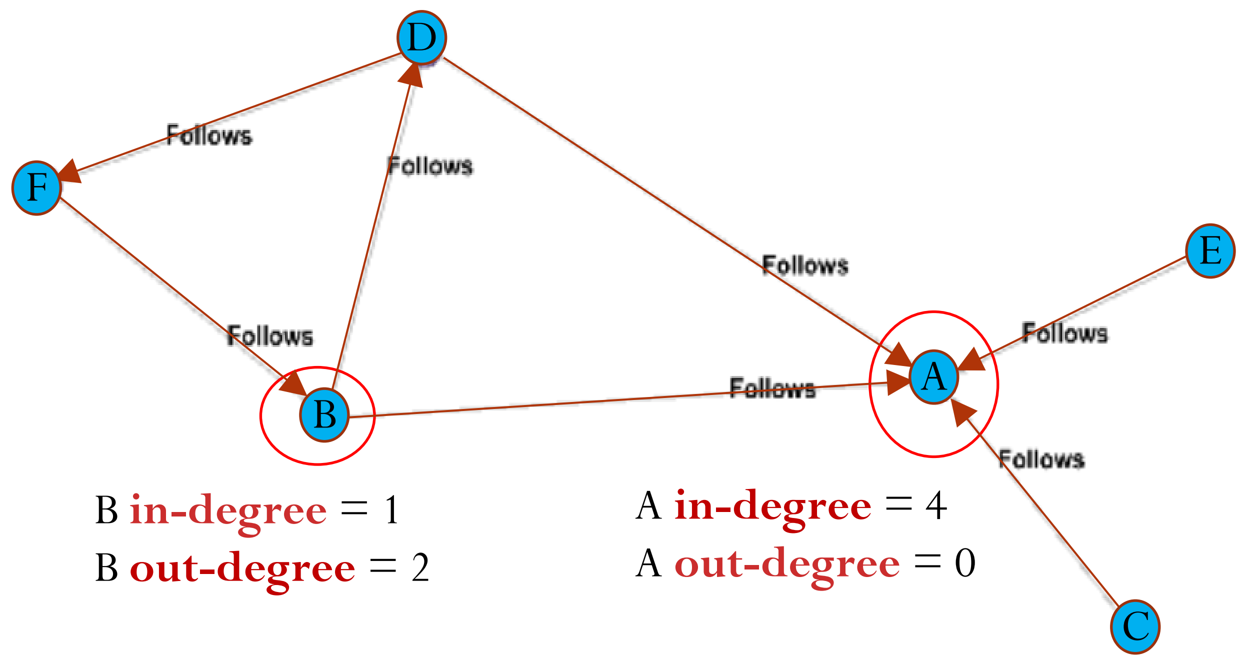
Out-degree of V1 = 2



Out-degree of V2 = 1

Degree of a node

- Nodes **E,C,D and B** have an **outgoing** edge towards node **A** and hence **follow** node A. Thus, the **in-degree of node A is 4** as it has 4 edges incident on it. Node B follows both node D and node A, hence it's **out-degree is 2**.



B **in-degree** = 1
B **out-degree** = 2

A **in-degree** = 4
A **out-degree** = 0

Centrality

- In graph analytics, **Centrality** is a very important concept in **identifying important nodes** in a graph.
- It is used to **measure the importance** (or "centrality" as in **how "central" a node is** in the graph) of various nodes in a graph.
- Centrality comes in **different flavors** and each flavor or a metric defines importance of a node from a **different perspective** and further provides relevant analytical information about the graph and its nodes.
  - **Degree** Centrality
  - **Closeness** Centrality
  - **Betweenness** Centrality
  - **Eigen Vector** Centrality

Degree Centrality

- **Degree Centrality** metric defines importance of a node in a graph as being measured based on its degree i.e the **higher the degree** of a node, the **more important** it is in a graph.

- Mathematically, **Degree Centrality** is defined as **D(i)** for a node "i" as below:

Degree Centrality for **undirected** graph

$$D(i) = \sum_j m(i,j)\ , where\ m(i,j) = 1$$

if there is a **link (m)** **between node i** and **node j**

Degree Centrality for **directed** graph

In-degree Centrality

$$D(\boldsymbol{i}) = \sum_j m(i,j)\ , where\ m(\boldsymbol{j},\boldsymbol{i}) = 1$$

if there is a **link (m)** from node **j** to node **i**

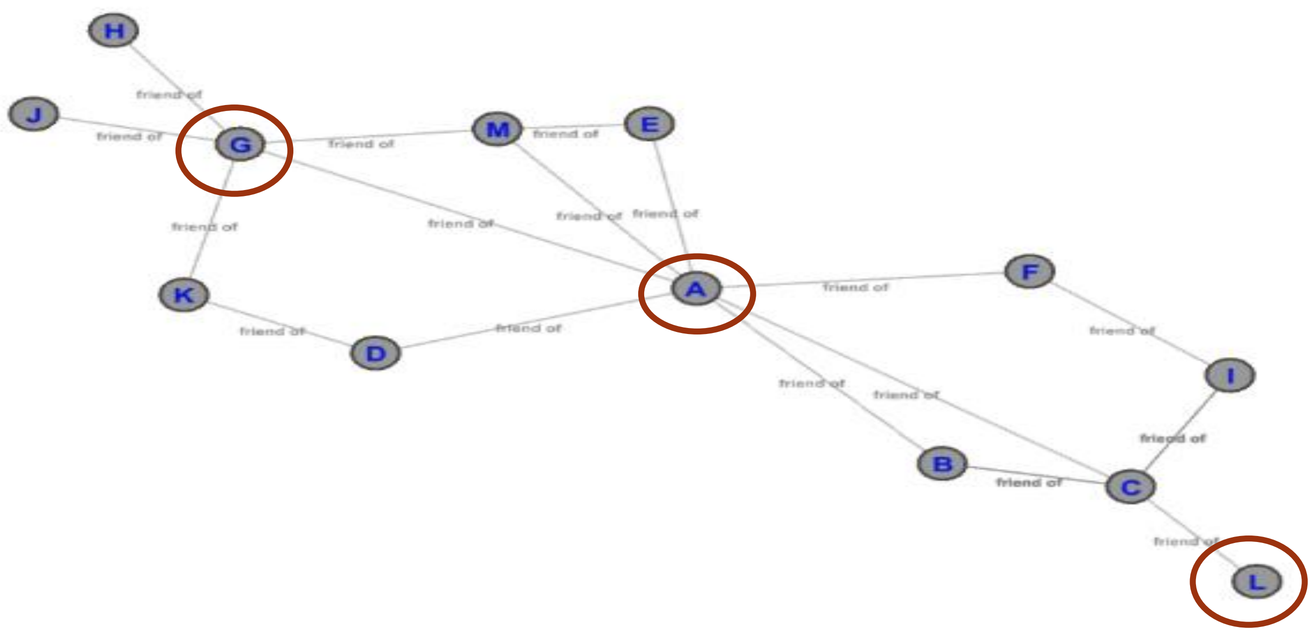
Out-degree Centrality

$$D(\boldsymbol{i}) = \sum_j m(i,j)\ , where\ m(\boldsymbol{i},\boldsymbol{j}) = 1$$

if there is a **link (m)** from node **i** to node **j**

Degree Centrality

- Node **A** and **G**, have **high degree centrality** (**7** and **5** respectively) – possible to **propagate information** to the network quickly as compared to node **L** (degree centrality **1**)

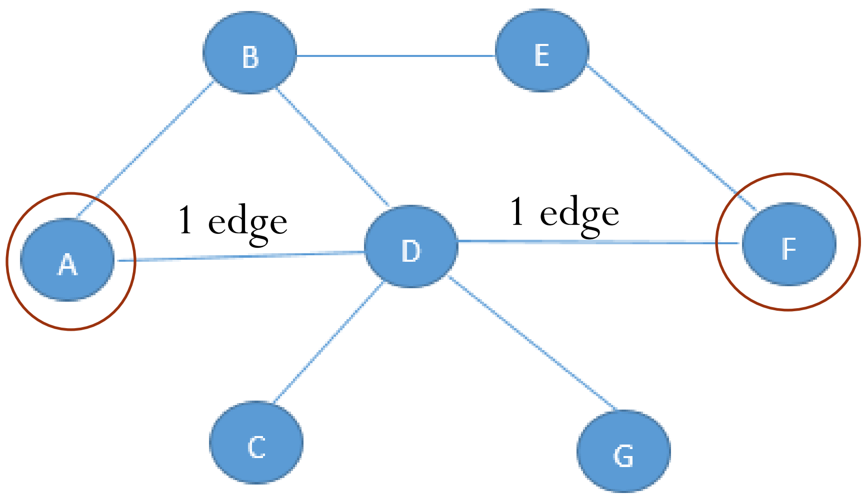


- **Marketing** or an **influencing strategy** of a new product/idea/thought in the network. Focus on nodes such as A,G etc. with **high degree centrality** to market product or ideas in the network to ensure higher reach-ability among nodes.

Closeness Centrality

- The **Geodesic** (**shortest** possible) **distance** d between two nodes is defined as the **number of edges/links** between these two nodes on the **shortest path** (path with **minimum number of edges**) between them => **measures how close the node is to all other nodes in the network.**

- We can reach F from A by going through B and E (3 edges) or by going through D. However, the **shortest path** from A to F is through D (2 edges), hence the **geodesic distance d(A,F)** is defined as **2** as there are 2 edges between A and F.



Closeness Centrality

- Mathematically, **Geodesic distance** is defined as:
  - **d(*a* , *b*)** = **No. of edges between *a* and *b*** on the **shortest path** from *a* to *b*, if a path exists from *a* to *b*
  - **d(*a* , *b*)** = 0, if **a = *b*** **(same node)**
  - **d(*a* , *b*)** = ∞ (Infinity) , if **no path** exists from *a* to *b*

- **Closeness centrality** metric defines the importance of a node in a graph as being measured by **how close (connected) it is to all other nodes** in the graph. For a node, it is defined as the **sum of the geodesic distance** between that **node to all other nodes** in the network.

Mathematically, **Closeness Centrality** C(i) of a **node *i*** in a graph is defined as:

$$\boxed{C(i) = \frac{1}{\sum_j d(i,j)}}$$

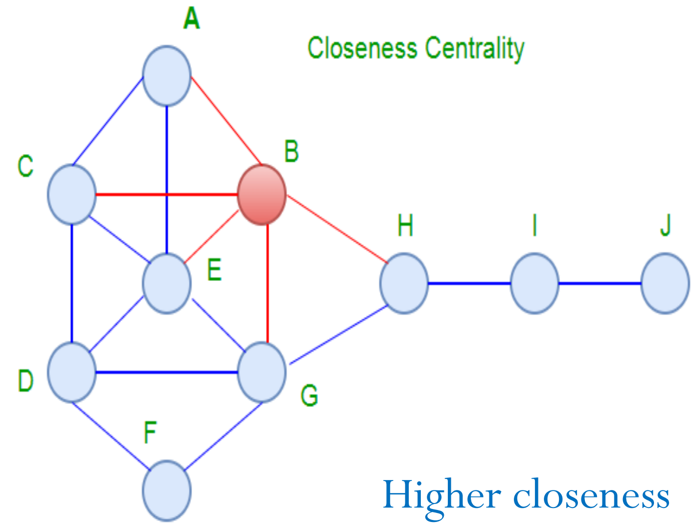
Reciprocal of the sum of the geodesic distance

Shorter distance between nodes => **higher** the **Closeness** Centrality

Application of Closeness Centrality

- Suppose that in the graph, each link/edge had a weight (attribute) of 1 minute associated with it, i.e it would take 1 minute to transmit information from a node to its neighboring node.
- Suppose we want to **send information** to each node of the graph and to **select a node in the graph that can transmit it quickly** to all the nodes in the network.
  - Calculate **Closeness Centrality** measure for all the nodes in the network

Higher the Closeness Centrality measure => **more central** the node



Higher closeness centrality score

C(A) = 1/19

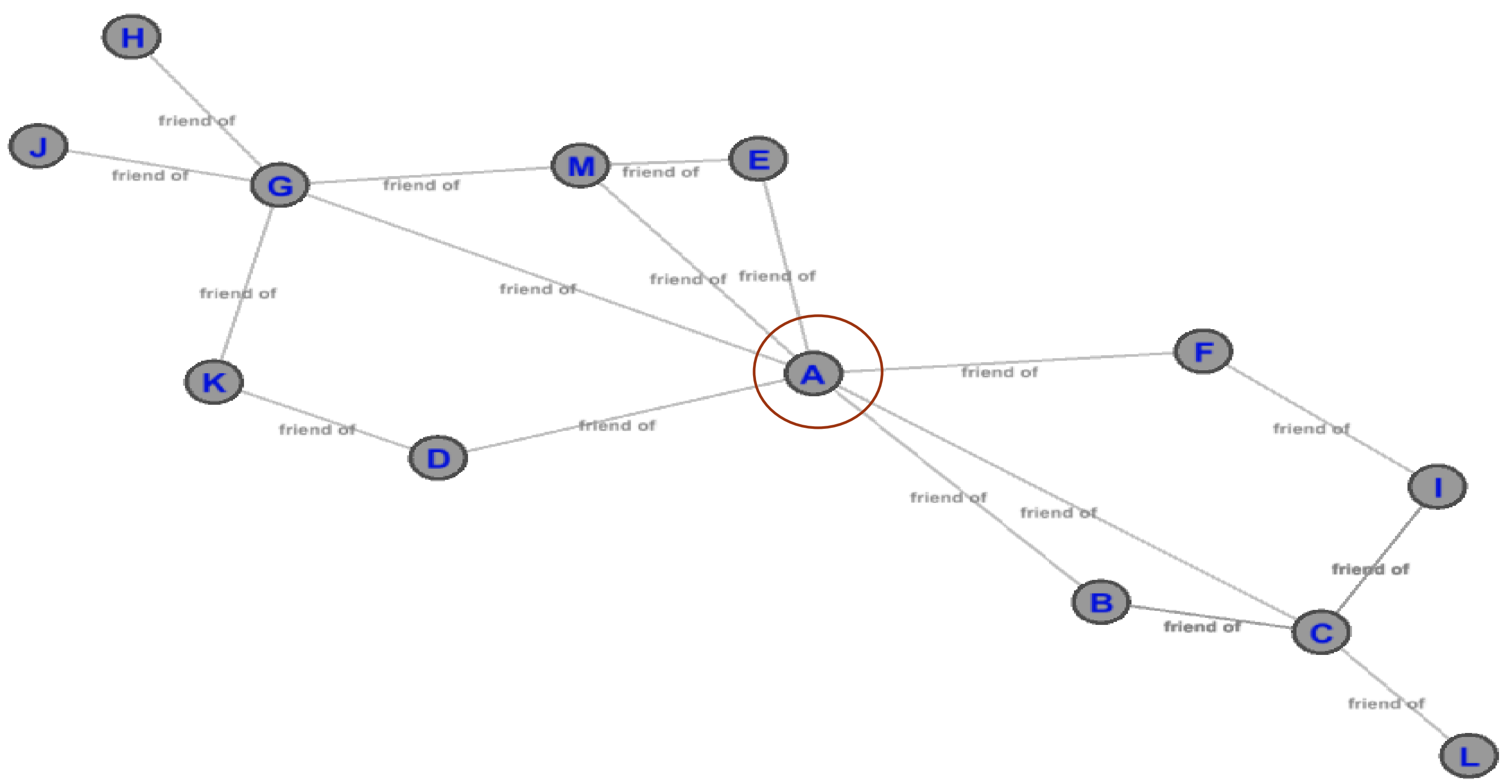
d(A,B) = 1
d(A,C) = 1
d(A,D) = 2
d(A,E) = 1
d(A,F) = 3
d(A,G) = 2
d(A,H) = 2
d(A,I) = 3
d(A,J) = 4

C(B) = 1/14

d(B,A) = 1
d(B,C) = 1
d(B,D) = 2
d(B,E) = 1
d(B,F) = 2
d(B,G) = 1
d(B,H) = 1
d(B,I) = 2
d(B,J) = 3

Betweenness Centrality

- "**Betweenness Centrality**" (BC) defines and measures the **importance** of a node in a network based upon **how many times it occurs in the shortest path between all pairs** of nodes in a graph => **measures the influence a node has over the flow of information in a graph**



Betweenness Centrality

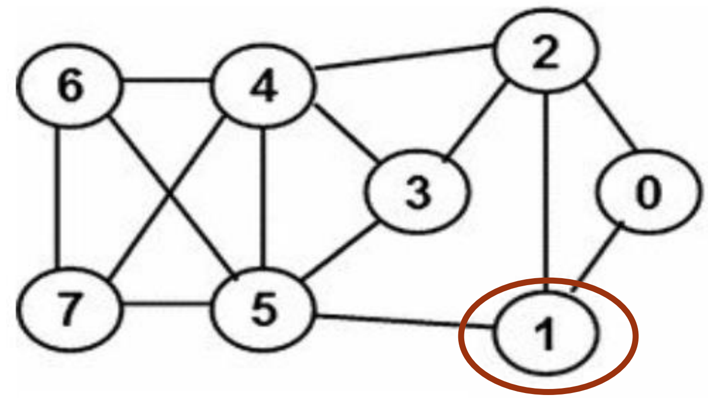
Betweenness Centrality B(i) of a node *i* in a graph is defined as:

$$B(i) = \sum_{a,b} \frac{g_{aib}}{g_{ab}}$$

Shortest paths between **a** and **b** **passing through i**

Shortest paths between **a** and **b** **(possible to have more than one shortest path)**

a,*b* is any pair of nodes in the graph



g_{aib} is the number of shortest paths from node **a to b** **passing through i**

Geodesic distance

g_{ab} is the number of shortest paths from node **a to b**

```
**************************************
betweeness for node 1
Pair <5,0> --->1 / 1
Pair <6,0> --->1 / 2
Pair <7,0> --->1 / 2
Pair <5,2> --->1 / 3
Pair <0,5> --->1 / 1
Pair <2,5> --->1 / 3
Pair <0,6> --->1 / 2
Pair <0,7> --->1 / 2
Betweenness of 1 : 4.66666666666667
**************************************
```

Shortest paths Btw 7 and 0

7,5,1,0

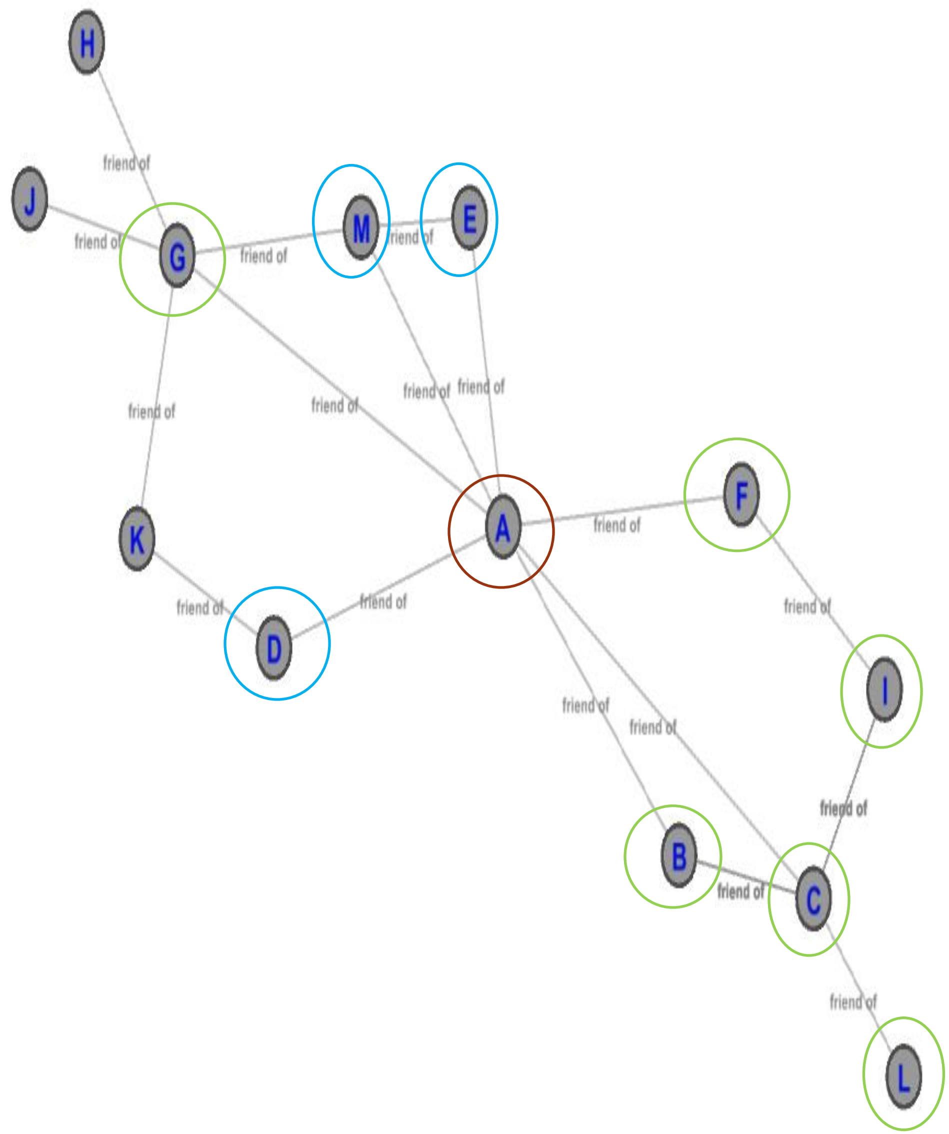
7,4,2,0

Betweenness Centrality

- Node A lies on the **shortest path** between the pair of nodes: (D,E), (D,M), (D,G), (G,B), (G,C), (G,F), (G,I), (G, L), etc. and has the highest Betweenness Centrality among all other nodes in the graph.

- If node A was **removed**, it would lead to **huge disruption** in the network as there would be no way for nodes **{J,H,G,M,K,E,D}** to communicate with nodes **{F,B,C,I,L}** and vice versa, ending up with two isolated sub graphs.

- This shows the importance of nodes with high Betweenness Centrality.



Eigen Vector Centrality 特征向量中心性

- This metric measures the importance of a node in a graph as **a function of the importance of its neighbors**.
  该度量测量图中节点的重要性，作为其邻居重要性的函数。

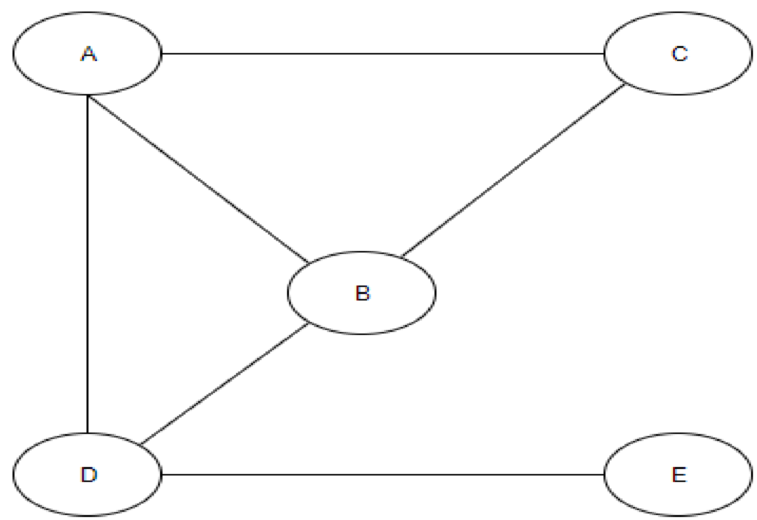
- If a node is **connected to highly important nodes**, it will have a **higher Eigen Vector Centrality** score as compared to a node which is connected to lesser important nodes.
  如果一个节点连接到非常重要的节点，那么与连接到不太重要的节点的节点相比，它将具有更高的特征向量中心性得分。

- The **adjacency matrix A** of the graph is shown below:
  图表的邻接矩阵A如下所示：

Indicates whether nodes are linked to each other
指示节点是否相互连接

$$
\begin{array}{c|c}
 & \begin{array}{ccccc} A & B & C & D & E \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} & \begin{bmatrix} - & 1 & 1 & 1 & 0 \\ 1 & - & 1 & 1 & 0 \\ 1 & 1 & - & 0 & 0 \\ 1 & 1 & 0 & - & 1 \\ 0 & 0 & 0 & 1 & - \end{bmatrix}
\end{array}
$$

AE has no link



Eigen Vector Centrality

Node	Degree
A	3
B	3
C	2
D	3
E	1

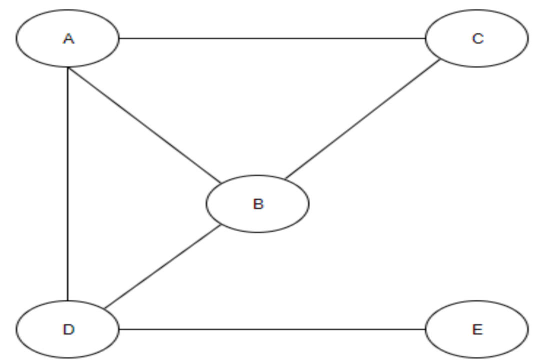
- Assume the importance of each node is measured by its degree, such that the **higher the degree** of a node, the **more important** it is in the graph.

  假设每个节点的重要性都是通过其度来衡量的，这样节点的度越高，它在图中就越重要

- The above can also be represented as a **degree matrix vector V** :

  上述也可以表示为度矩阵向量V：

$$\begin{bmatrix} 3 \\ 3 \\ 2 \\ 3 \\ 1 \end{bmatrix}$$



- The **Eigen Vector Centrality** is calculated as a **dot product** shown below:

  特征向量中心性计算为如下所示的点积：

$$A \times V = \begin{bmatrix} - & 1 & 1 & 1 & 0 \\ 1 & - & 1 & 1 & 0 \\ 1 & 1 & - & 0 & 0 \\ 1 & 1 & 0 & - & 1 \\ 0 & 0 & 0 & 1 & - \end{bmatrix} \begin{bmatrix} 3 \\ 3 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0\times3+1\times3+1\times2+1\times3+0\times1 \\ 1\times3+0\times3+1\times2+1\times3+0\times1 \\ 1\times3+1\times3+0\times2+0\times3+0\times1 \\ 1\times3+1\times3+0\times2+0\times3+1\times1 \\ 0\times3+0\times3+0\times2+1\times3+0\times1 \end{bmatrix} = \begin{bmatrix} 8 \\ 8 \\ 6 \\ 7 \\ 3 \end{bmatrix}$$

Adjacency matrix **A** of graph

Degree matrix vector **V**

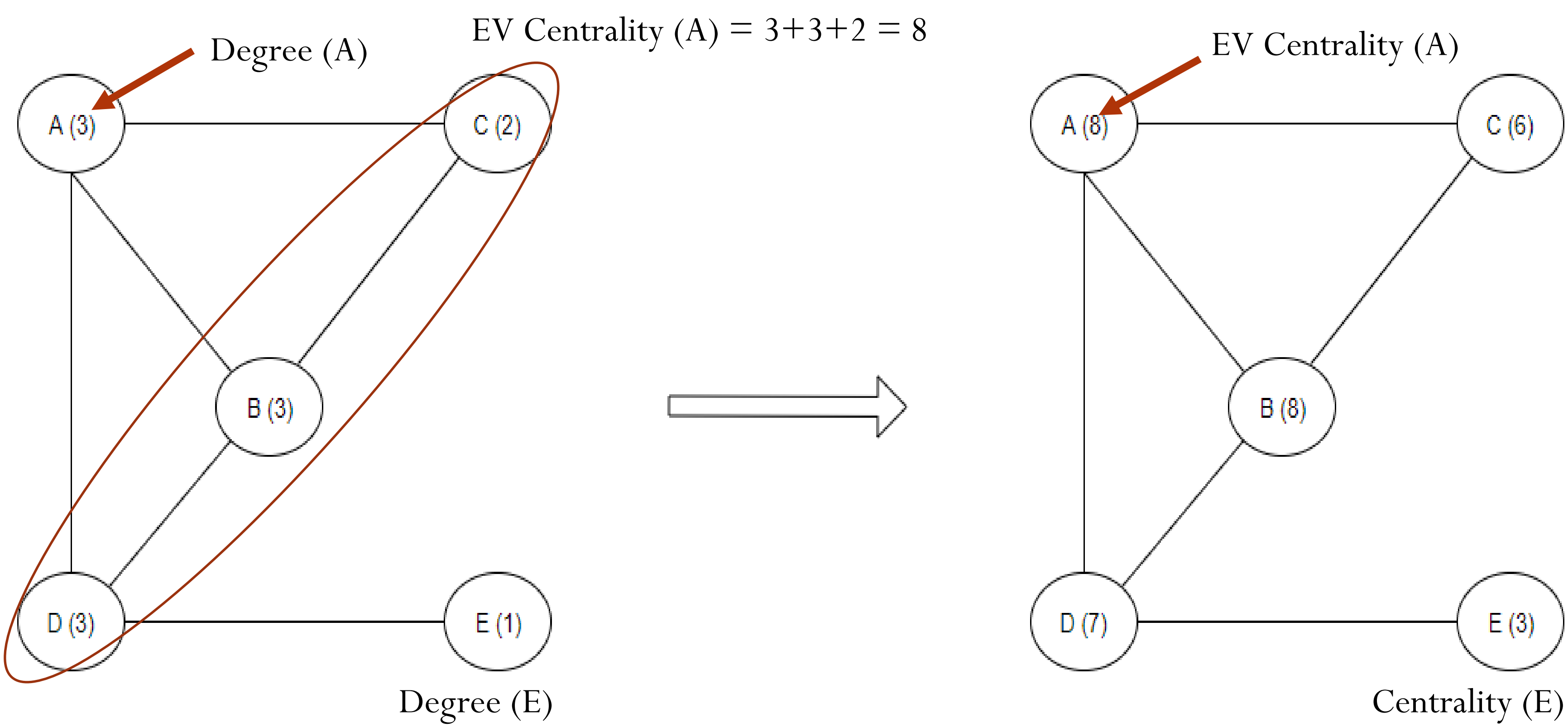
How to interpret results?

Eigen Vector Centrality

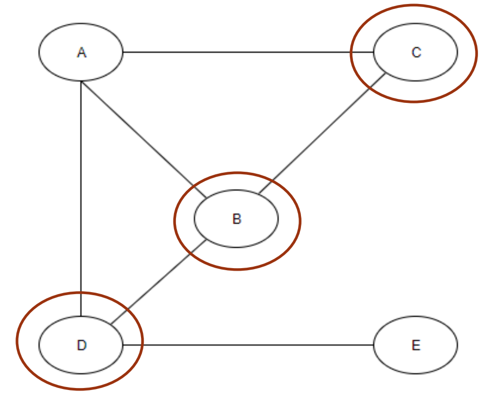
- Node **A** and **B** both have a high score of **8** since both are connected to **multiple nodes** with **high degrees** (importance) while node **E** has a score of **3** since its only connected to a **single node** of degree 3.

节点A和B的高分为8，因为两者都连接到多个高度（重要性）的节点，而节点E的得分为3，因为它只连接到3度的单个节点。

$$\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} \begin{bmatrix} 8 \\ 8 \\ 6 \\ 7 \\ 3 \end{bmatrix}$$



Eigen Vector Centrality



$$\begin{bmatrix}3\\3\\2\\3\\1\end{bmatrix}$$

The EVC score value for each node in the resultant vector is nothing but the **sum of degrees of its neighboring nodes**.

- E.g., **EVC** score for node **A** = **degree(B)** [3] + **degree(C)** [2] + **degree(D)** [3] = 8

- Now if the **resultant** EVC vector above in the equation is **again multiplied** by the adjacency matrix A, we will get **bigger values** for EVC score for each node in the graph, 现在，如果方程中上述结果的EVC向量再次乘以邻接矩阵A，我们将获得图中每个节点的EVC得分值，

$$A\times V = \begin{bmatrix} - & 1 & 1 & 1 & 0 \\ 1 & - & 1 & 1 & 0 \\ 1 & 1 & - & 0 & 0 \\ 1 & 1 & 0 & - & 1 \\ 0 & 0 & 0 & 1 & - \end{bmatrix} \begin{bmatrix} 8\\8\\6\\7\\3 \end{bmatrix} = \begin{bmatrix} 0\times 8 + 1\times 8 + 1\times 6 + 1\times 7 + 0\times 3 \\ 1\times 8 + 0\times 8 + 1\times 6 + 1\times 7 + 0\times 3 \\ 1\times 8 + 1\times 8 + 0\times 6 + 0\times 7 + 0\times 3 \\ 1\times 8 + 1\times 8 + 0\times 6 + 0\times 7 + 1\times 3 \\ 0\times 8 + 0\times 8 + 0\times 6 + 1\times 7 + 0\times 3 \end{bmatrix} = \begin{bmatrix} 21\\21\\16\\19\\7 \end{bmatrix}$$

Adjacency matrix **A**

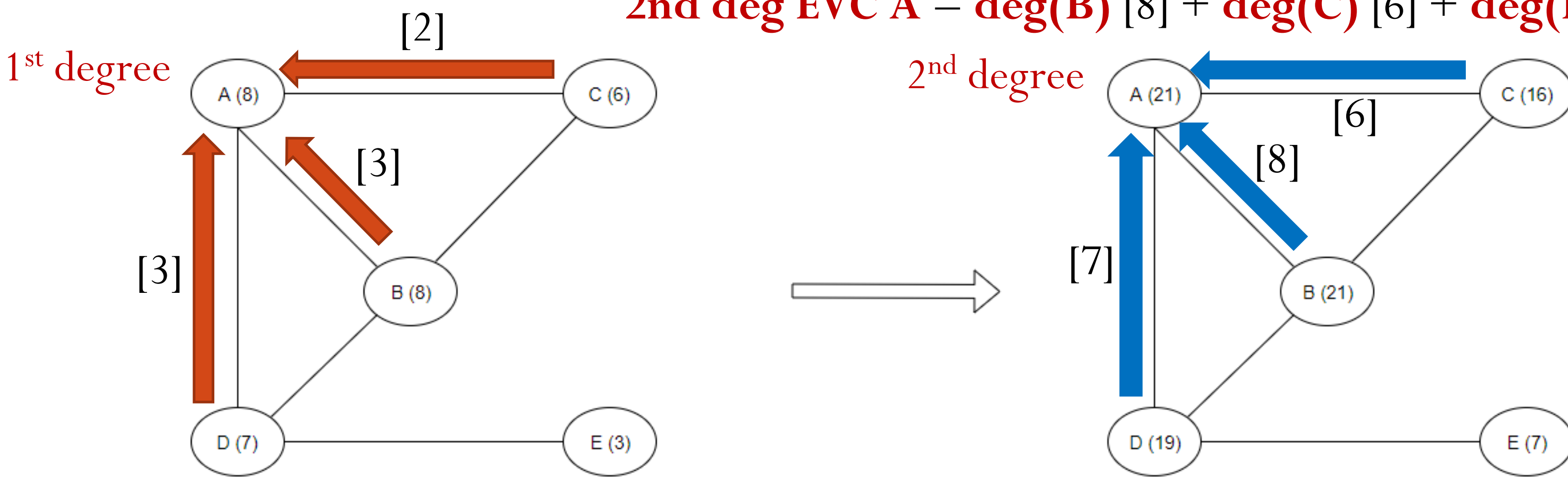
Resultant EVC vector

Eigen Vector Centrality

After the first iteration (**1st degree**) of multiplication, each node gets it's EVC score from its **direct neighbors**.

In the second iteration, when we multiply the **resultant vector** again with the adjacency matrix, each node **again** gets it's EVC score from its **direct neighbors**:

2nd deg EVC A = **deg(B)** [8] + **deg(C)** [6] + **deg(D)** [7] = 21



2nd deg EVC for A includes consideration for E due to E being a neighbor of D

In the second iteration the **scores of the direct neighbors have already been impacted by their own direct (1st degree) neighbors previously** (from the first iteration of multiplication) which eventually helps the EVC score of any node to be a **function of its 2nd degree neighboring nodes as well.**

Eigen Vector Centrality

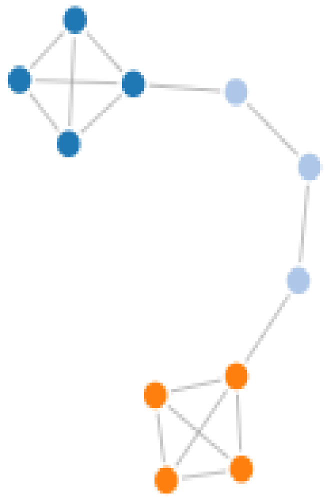
- In subsequent iterations of multiplication, the EVC score of graph nodes keeps getting updated by getting impacted by EVC scores from neighboring nodes of **farther degree** (3rd, 4th and so on).

- Usually the process of multiplying the EVC vector with the adjacency matrix is repeated until the **EVC values** for nodes in the graph **reach an equilibrium.**

- Multiplying the resultant vector again with the adjacency matrix of the graph **spreads the EVC score** in the graph to get a more **global** EVC score vs a **localized** EVC score for each node in the graph.

Introduction to NetworkX

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.



Features

- Data structures for graphs, directed graphs, and multigraphs
- Many standard graph algorithms
- Network structure and analysis measures
- Generators for classic graphs, random graphs and synthetic networks

NetworkX Installation

- **NetworkX** requires Python 3.6, 3.7, or 3.8.
- Download **Anaconda** from https://www.anaconda.com/download

Installing networkx

$ pip install networkx

Or in Jupyter Notebook:
!pip install networkx

Create Undirected Graph

Creating an **empty graph object** with no nodes and no edges

```
import networkx as nx

# Define non-directional graph object
G = nx.Graph()
```

Adding **nodes** to the graph

```
# Add nodes
G.add_node(1)
```

or

```
G.add_nodes_from([
    (2, {"color": "red"}),
    (3, {"color": "green"}),
    (4, {"color": "blue"}),
    (5, {"color": "yellow"})
])
```

Node attributes

Adding nodes with attributes using

(node, {node_attribute_dict})

Adding **edges** to the graph

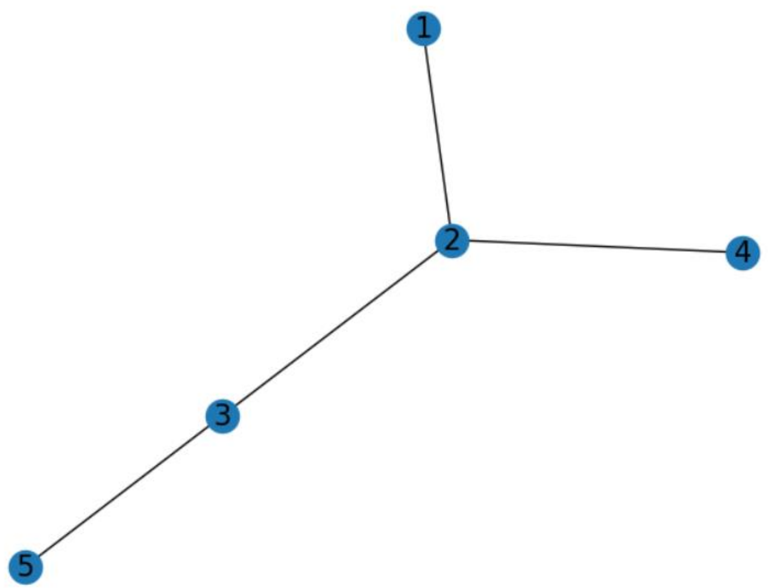
```
# Add edges
G.add_edge(1, 2)
G.add_edge(2, 3)
G.add_edge(2, 4)
G.add_edge(3, 5)
```

Draw the Undirected Graph

```
# Get nodes and edges from graph
print("Nodes:", G.nodes())
print("Edges:", G.edges())
print()
```

```
Nodes: [1, 2, 3, 4, 5]
Edges: [(1, 2), (2, 3), (2, 4), (3, 5)]
```

```
#Draw the graph
nx.draw(G, with_labels=True, font_size=10, font_family='sans-serif')
```



Create Directed Graph

- Construct a **directed** graph using **nx.DiGraph( )**
- Add **directional** edges using **add_edge** method, giving a graph with two nodes and one edge between them.
- **Edges are represented by tuple (X,Y)** (ordered set of values) since both nodes that it connects are uniquely identified themselves.

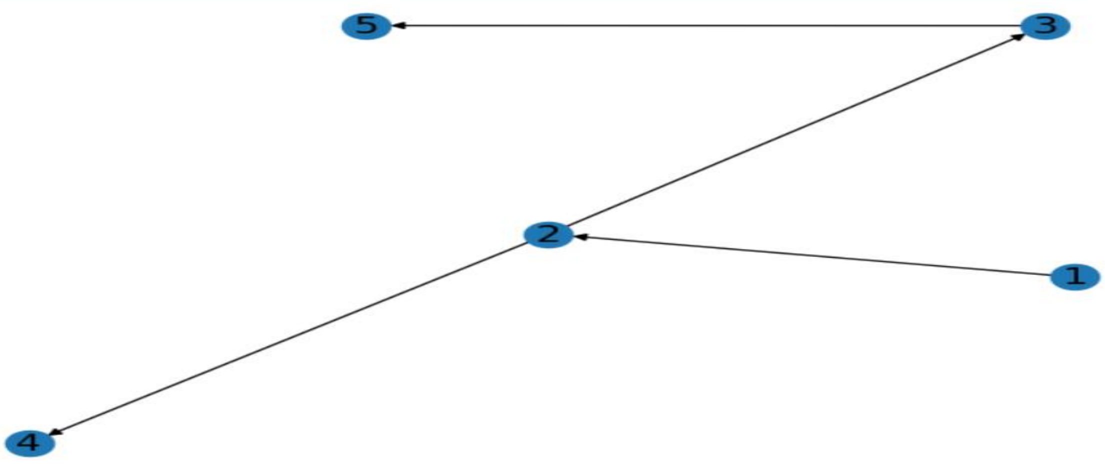
```python
import networkx as nx

# Define directional graph object
G = nx.DiGraph()

# Add nodes
G.add_nodes_from([
    (1, {"color": "grey"}),
    (2, {"color": "red"}),
    (3, {"color": "green"}),
    (4, {"color": "blue"}),
    (5, {"color": "yellow"})
])

# Add edges
G.add_edge(1, 2)
G.add_edge(2, 3)
G.add_edge(2, 4)
G.add_edge(3, 5)
```

Draw the Directed Graph

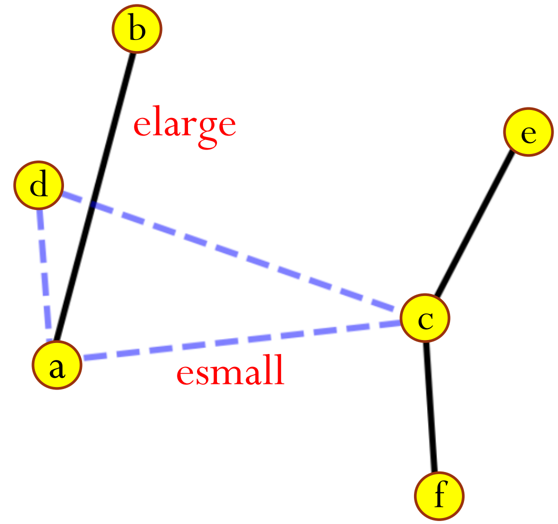
```
# Get nodes and edges from graph
print("Number of nodes:", G.number_of_nodes())
print("Nodes:", G.nodes())
print("Number of edges:", G.number_of_edges())
print("Edges:", G.edges())
```

```
Number of nodes: 5
Nodes: [1, 2, 3, 4, 5]
Number of edges: 4
Edges: [(1, 2), (2, 3), (2, 4), (3, 5)]
```

```
#Draw the graph
nx.draw(G, with_labels=True, font_size=15, font_family='sans-serif')
```



NetworkX – Weighted Graph




```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()
```

Define **undirected** graph object

```
G.add_edge('a', 'b', weight=0.6)
```

Define **weight** of edge

```
G.add_edge('a', 'c', weight=0.2)
G.add_edge('c', 'd', weight=0.1)
G.add_edge('c', 'e', weight=0.7)
G.add_edge('c', 'f', weight=0.9)
G.add_edge('a', 'd', weight=0.3)

elarge = [(u, v) for (u, v, d) in G.edges(data=True) if d['weight'] > 0.5]
esmall = [(u, v) for (u, v, d) in G.edges(data=True) if d['weight'] <= 0.5]

pos = nx.spring_layout(G)  # positions for all nodes

# nodes
nx.draw_networkx_nodes(G, pos, node_size=700)
```

Specify how big the node is

```
# edges
nx.draw_networkx_edges(G, pos, edgelist=elarge,
                       width=6)
nx.draw_networkx_edges(G, pos, edgelist=esmall,
                       width=6, alpha=0.5, edge_color='b', style='dashed')
```

Edge transparency

Blue

```
# labels
nx.draw_networkx_labels(G, pos, font_size=20, font_family='sans-serif')

plt.axis('off')
plt.show()
```

Add node “letter” label

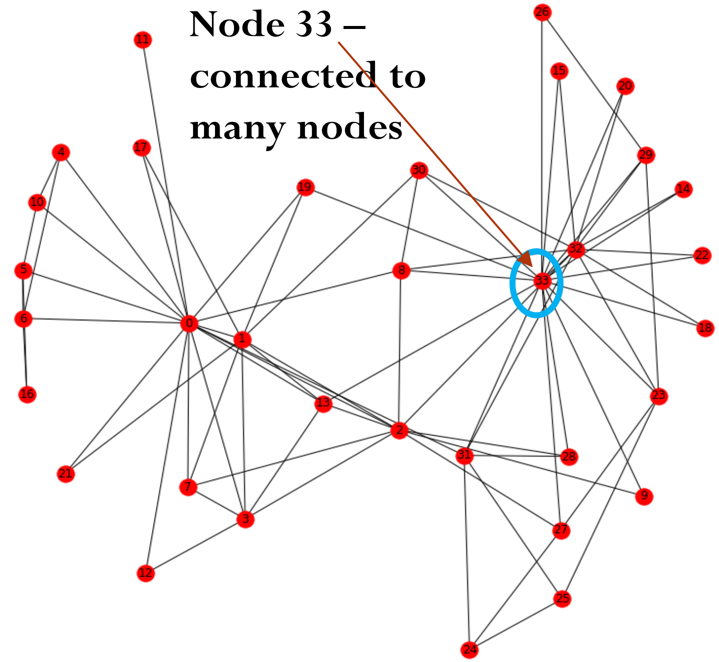
NetworkX - nx.layout options

bipartite_layout(G, nodes[, align, scale, …])	Position nodes in two straight lines.
circular_layout(G[, scale, center, dim])	Position nodes on a circle.
kamada_kawai_layout(G[, dist, pos, weight, …])	Position nodes using Kamada-Kawai path-length cost-function.
planar_layout(G[, scale, center, dim])	Position nodes without edge intersections.
random_layout(G[, center, dim, seed])	Position nodes uniformly at random in the unit square.
rescale_layout(pos[, scale])	Returns scaled position array to (-scale, scale) in all axes.
rescale_layout_dict(pos[, scale])	Return a dictionary of scaled positions keyed by node
shell_layout(G[, nlist, rotate, scale, …])	Position nodes in concentric circles.
spring_layout(G[, k, pos, fixed, …])	Position nodes using Fruchterman-Reingold force-directed algorithm.
spectral_layout(G[, weight, scale, center, dim])	Position nodes using the eigenvectors of the graph Laplacian.
spiral_layout(G[, scale, center, dim, …])	Position nodes in a spiral layout.
multipartite_layout(G[, subset_key, align, …])	Position nodes in layers of straight lines.

https://networkx.github.io/documentation/stable/reference/drawing.html?highlight=layout#module-networkx.drawing.layout

Centrality Measures using NetworkX

- **Centrality Measures** pinpoint the most **important nodes** of a Graph.
  - **(Degree) Influential nodes** in a Social Network.
  - **(Closeness)** Nodes that **disseminate information to many nodes**
  - **(Betweeness) Hubs** in a transportation network
  - **(Eigen Vector)** Nodes **linked to influential nodes**




```
# Centrality Measures

import matplotlib.pyplot as plt
import networkx as nx

G = nx.karate_club_graph()

plt.figure(figsize=(15,15))
nx.draw_networkx(G, with_labels=True)

# Degree Centrality
deg_centrality = nx.degree_centrality(G)
print("Degree Centrality: "+str(deg_centrality))
```

Pattern split nodes into groups (based on members joining 2 opposing karate clubs)

Returns a **dictionary** of size equal to the number of nodes in Graph G, where the ith element is the degree centrality measure of the ith node.

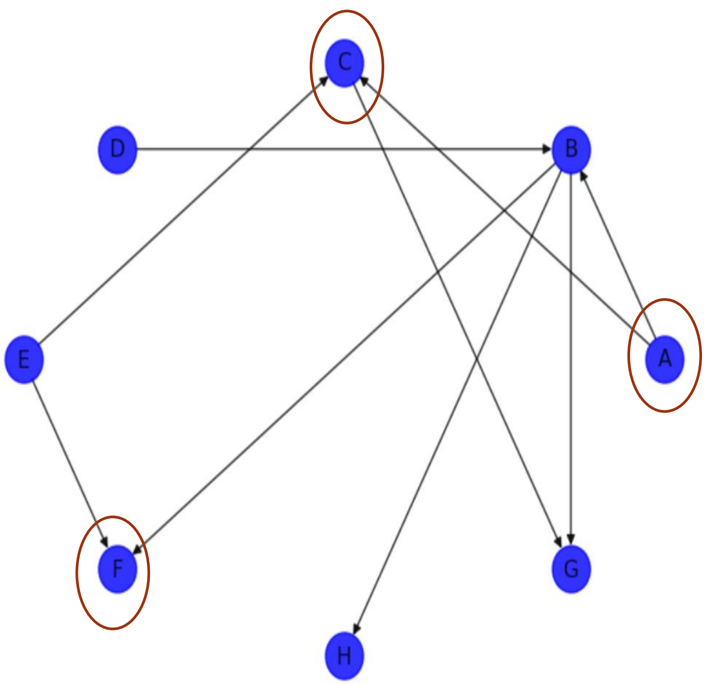
```
Degree Centrality: {0: 0.48484848484848486, 1: 0.2727272727272727, 2: 0.30303030303030304, 3: 0.18181818181818182, 4: 0.090909
09090909091, 5: 0.12121212121212122, 6: 0.12121212121212122, 7: 0.12121212121212122, 8: 0.15151515151515152, 9: 0.060606060606
06061, 10: 0.09090909090909091, 11: 0.030303030303030304, 12: 0.06060606060606061, 13: 0.15151515151515152, 14: 0.060606060606
06061, 15: 0.06060606060606061, 16: 0.06060606060606061, 17: 0.06060606060606061, 18: 0.06060606060606061, 19: 0.0909090909090
9091, 20: 0.06060606060606061, 21: 0.06060606060606061, 22: 0.06060606060606061, 23: 0.15151515151515152, 24: 0.09090909090909
091, 25: 0.09090909090909091, 26: 0.06060606060606061, 27: 0.12121212121212122, 28: 0.09090909090909091, 29: 0.121212121212121
22, 30: 0.12121212121212122, 31: 0.18181818181818182, 32: 0.36363636363636365, 33: 0.5151515151515151}
```

Centrality Measures - Degree

- **Degree Centrality** - based on assumption that important nodes have **many connections**.

$$Centrality_{degree}(v) = d_v / (\underbrace{|N| - 1}_{\text{normalization}})$$

Where d_v is the **degree of node v** and **N is the set of all nodes** of the Graph




```python
### Degree Centrality - Directed Graph ###
G = nx.DiGraph()
G.add_edges_from(
    [('A', 'B'), ('A', 'C'), ('D', 'B'), ('E', 'C'), ('E', 'F'),
     ('B', 'H'), ('B', 'G'), ('B', 'F'), ('C', 'G')])
pos = nx.circular_layout(G)
nx.draw_networkx(G, pos, alpha=0.2, node_color='b', node_size=500)
plt.show()

in_deg_centrality = nx.in_degree_centrality(G)
out_deg_centrality = nx.out_degree_centrality(G)

print(in_deg_centrality)
print(out_deg_centrality)
```

No inbound edges

2/(8-1)=0.2857

```
{'A': 0.0, 'B': 0.2857142857142857, 'C': 0.2857142857142857, 'D': 0.0, 'E': 0.0, '|
4285, 'G': 0.2857142857142857}
{'A': 0.2857142857142857, 'B': 0.42857142857142855, 'C': 0.14285714285714285, 'D':
7, 'F': 0.0, 'H': 0.0, 'G': 0.0}
```

No outbound edges

Centrality Measures - Closeness

- **Closeness Centrality** - based on the assumption that important nodes are linked to other nodes. Calculated as the **sum of the geodesic path (shortest lengths)** from the **given node** to **all other** nodes.

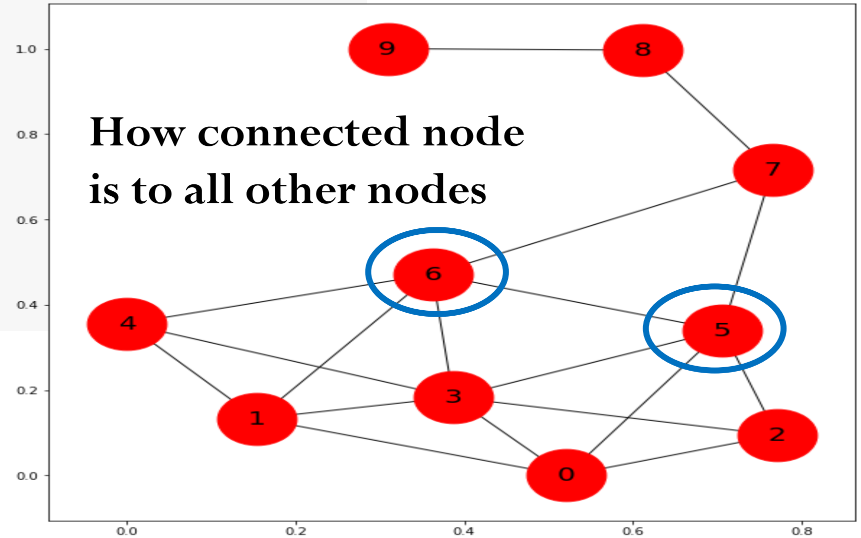
```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.krackhardt_kite_graph()

plt.figure(figsize =(10, 10))
nx.draw_networkx(G, node_size = 3000, with_labels = True, font_size = 20)
close_centrality = nx.closeness_centrality(G)

# G is the kite Graph
#print(close_centrality)
for meas in close_centrality:
    print(str(meas)+':'+str(close_centrality[meas]))
```

in inches

in pixels

Default=12

```
0:0.5294117647058824
1:0.5294117647058824
2:0.5
3:0.6
4:0.5
5:0.6428571428571429
6:0.6428571428571429
7:0.6
8:0.42857142857142855
9:0.3103448275862069
```



Centrality Measures - Betweenness

- Betweenness assumes that **important nodes connect other nodes**.

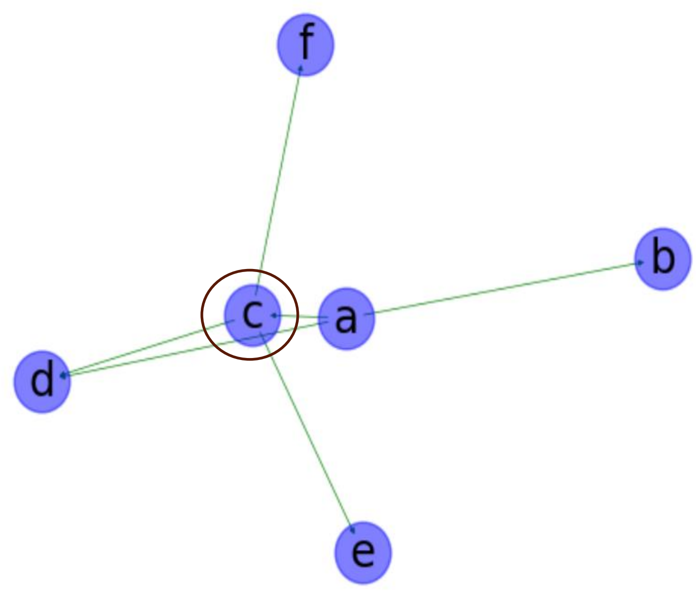
$$Centrality_{betweenness}(v) = \sum_{s,t \epsilon N} \sigma_{s,t}(v)/\sigma_{s,t}$$

where $\sigma_{s,t}$(v) is the **number of shortest paths** between nodes s and t that **pass through v**, $\sigma_{s,t}$ is the **number of shortest paths** between nodes s and t.

```
Betweenness Centrality: {'a': 0.0, 'b': 0.0, 'c': 0.1, 'd': 0.0, 'e': 0.0, 'f': 0.0}
```

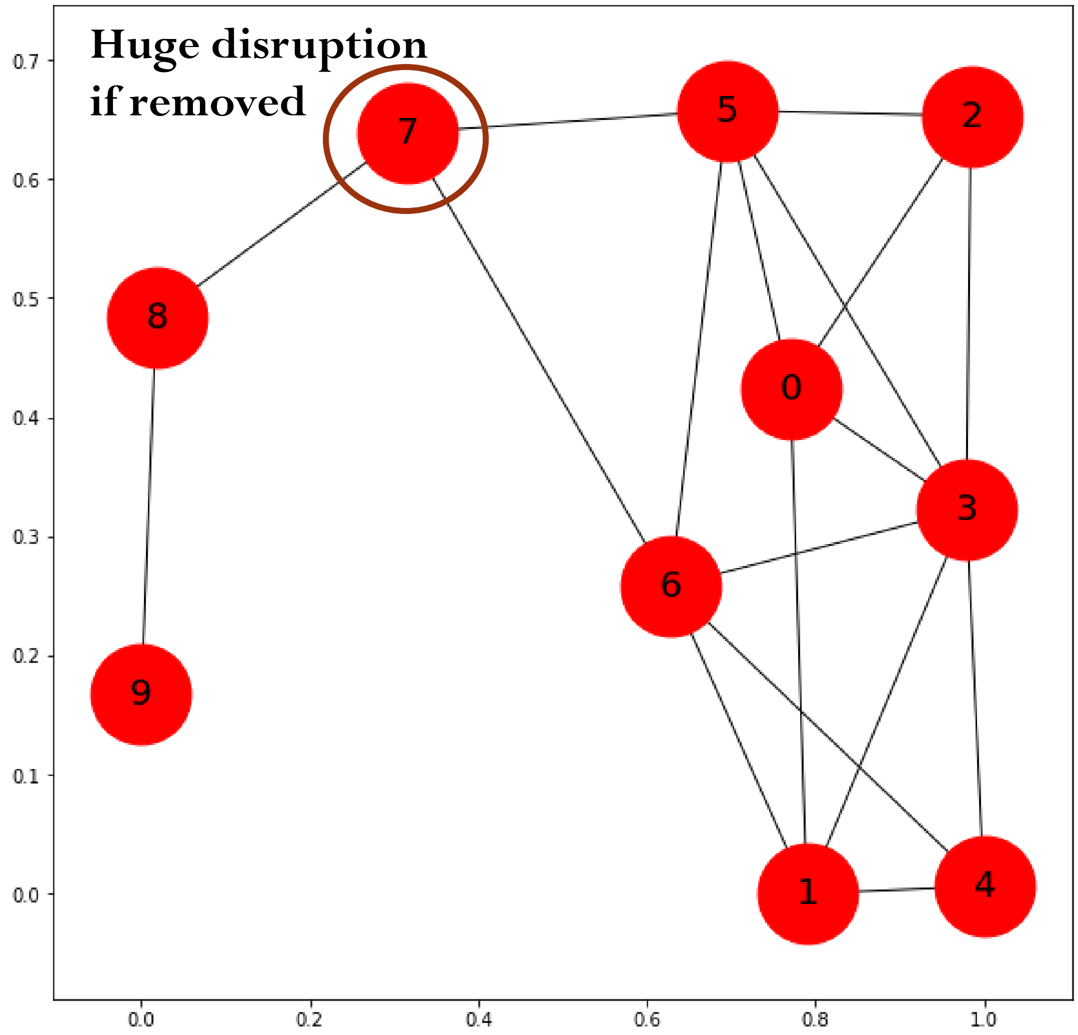
```python
import matplotlib.pyplot as plt
import networkx as nx

G = nx.krackhardt_kite_graph()

plt.figure(figsize =(10, 10))
nx.draw_networkx(G, node_size = 3000, with_labels = True, font_size = 20)
```

```python
# Betweeenness Centrality
between_centrality = nx.betweenness_centrality(G, normalized=True)
print("Betweenness Centrality: "+str(between_centrality))
```

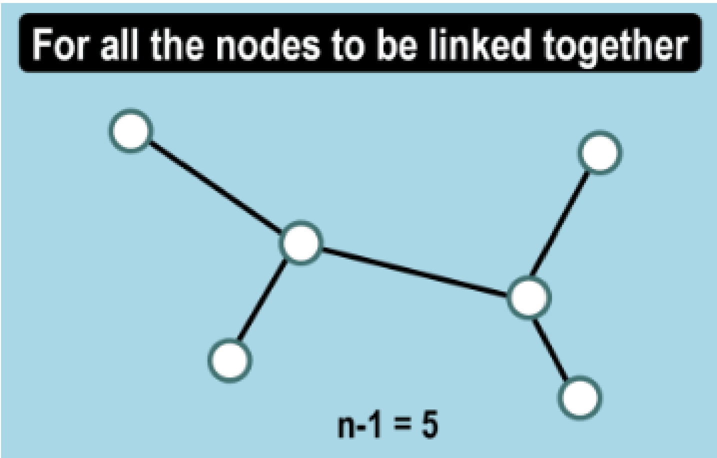


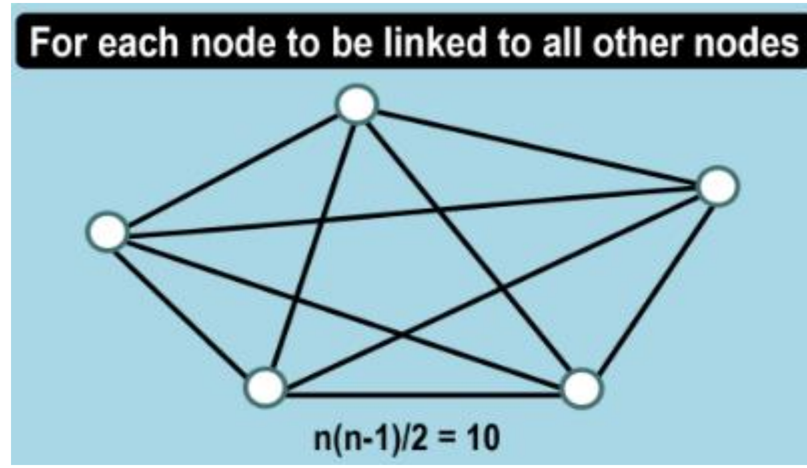
Centrality Measures - Betweenness

```
0:0.023148148148148143
1:0.023148148148148143
2:0.0
3:0.10185185185185183
4:0.0
5:0.23148148148148148
6:0.23148148148148148
7:0.38888888888888884
8:0.2222222222222222
9:0.0
```



Normalization





For large graphs, the betweenness centrality will be high. **Normalize** the centrality value by dividing with **number of node pairs**

For **Directed** Graphs, the number of **node pairs** (**exclude current node**) is **(|N-1|)*(|N|-2)**

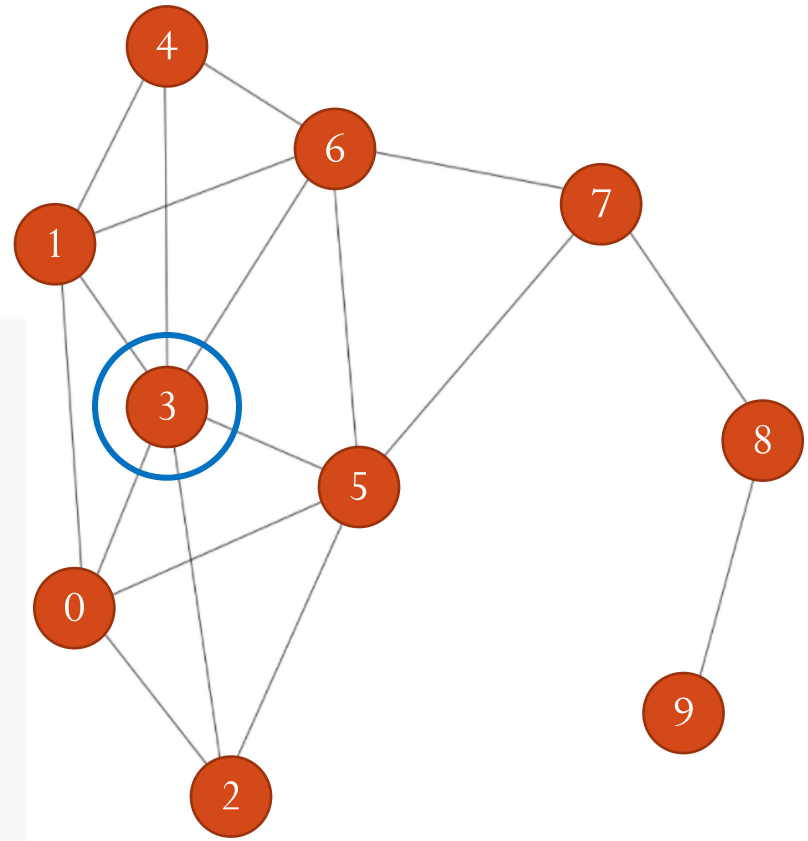
For **Undirected** Graphs, the number of node pairs is **(1/2)*(|N-1|)*(|N|-2)**.

halved

Centrality Measures - Eigenvector

If a node is **connected to highly connected nodes**, it will have a higher Eigen Vector Centrality score - **a function of the importance of its neighbors**.

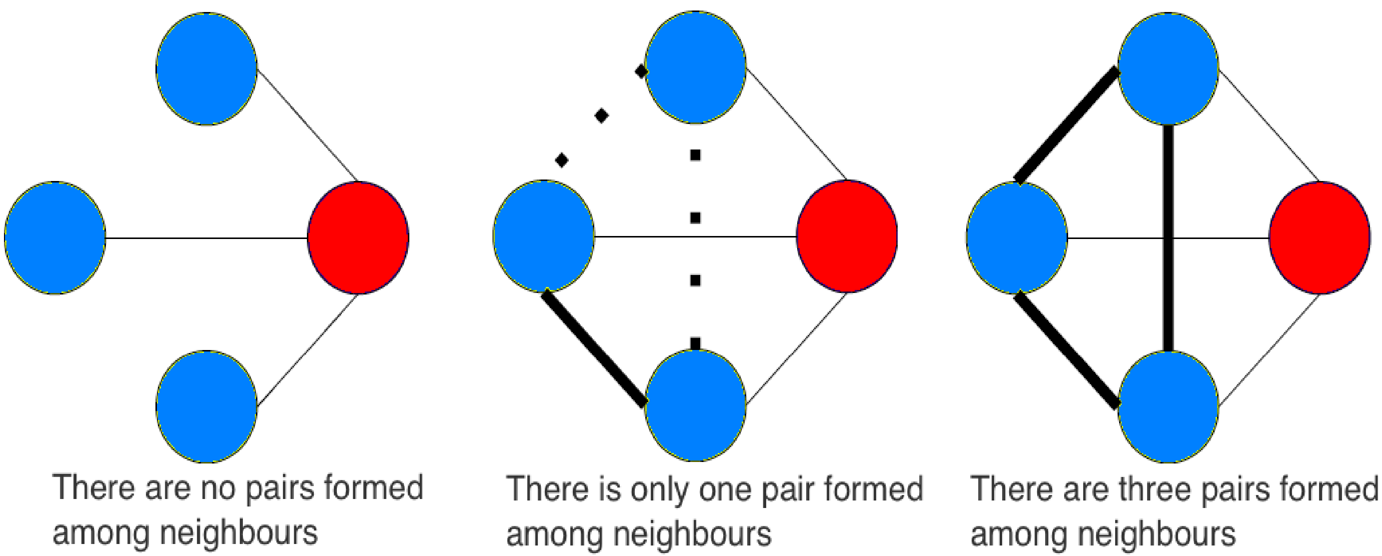
```
# Centrality Measures

import matplotlib.pyplot as plt
import networkx as nx

G = nx.krackhardt_kite_graph()

plt.figure(figsize=(15,15))
nx.draw_networkx(G, with_labels=True)

# Eigenvector Centrality
eigen_centrality = nx.eigenvector_centrality(G)
print("Eigenvector Centrality: "+str(eigen_centrality))
print()
```




```
Eigenvector Centrality: {0: 0.35220918419838565, 1: 0.35220918419838565, 2: 0.28583482369644964, 3: 0.481020669200118, 4: 0.28583482369644964, 5: 0.3976909028137205, 6: 0.3976909028137205, 7: 0.19586101425312444, 8: 0.04807425308073236, 9: 0.011163556091491361}
```

The clustering coefficient of graphs

- The **clustering coefficient** of a node or a vertex in a graph measures the **degree** to which **nodes in a graph tend to cluster together**.
- Compute based on **how close the neighbors are** so that they form a clique (or a small complete graph), as shown in the following diagram:



The clustering coefficient metric **differs from measures of centrality**. It is more akin to the **density metric** for whole networks.

Clustering coefficient

- A graph G= (V,E) consists of a set of vertices V and a set of edges E between them. An **edge** e_{ij} connects **vertex** v_i to **vertex** v_j
- **Neighbourhood Ni** for a vertex vi is defined as its **immediately connected neighbours**

Connected neighbours   logical OR

$$N_i = \{v_j : \underbrace{e_{ij} \in E \vee e_{ji} \in E}_{\text{If there's an edge (link) from node i to node j or node j to node i}}\}.$$

- k_i is the **number of vertices (nodes)** in the **neighbourhood Ni** of a vertex.
- **Local clustering coefficient Ci** for a vertex vi => **proportion of links between the neighbouring vertices** divided by the **number of links that could possibly exist between them**.
- For a **directed** graph, e_{ij} is distinct from e_{ji} => there are $\mathbf{k_i(k_i\text{-}1)}$ possible links among neighboring vertices.

Local clustering coefficient for directed graphs

$$C_i = \frac{\overbrace{|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}^{\text{Number of links between vertices in neighbourhood}}}{k_i(k_i - 1)}.$$

Clustering Coefficient

- In **undirected** graph, e_{ij} and e_{ji} are identical => $\frac{k_i(k_i - 1)}{2}$ edges could exist among vertices within neighbourhood

- **Local clustering coefficient for undirected graphs**

$$C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)}.$$

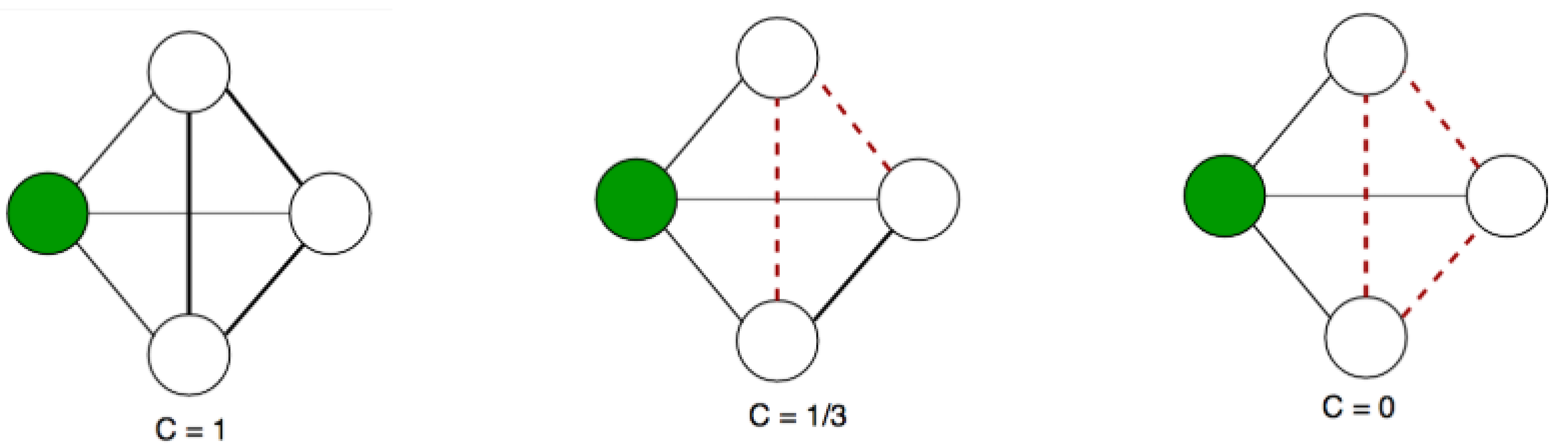
$$\text{Ci} = \frac{2 \text{ x } (\textbf{number of links between the vertices within its neighbourhood})}{\boldsymbol{ki(ki-1)}}$$

Clustering Coefficient

In the figures, the green node has three neighbours (**k_i=3**), which can have a **maximum of 3 connections among them**.

- In the left figure all three possible connections are realised (thick black segments) => local clustering coefficient of 1.
- Middle figure one connection is realised (thick black line), 2 connections are missing (dotted red lines) => local cluster coefficient = 1/3.
- Right figure no connections among the neighbours of the green node => local clustering coefficient value of 0.

$$C_i = \frac{2 \times (\textbf{number of links between the vertices within its neighbourhood})}{\boldsymbol{ki(ki-1)}}$$

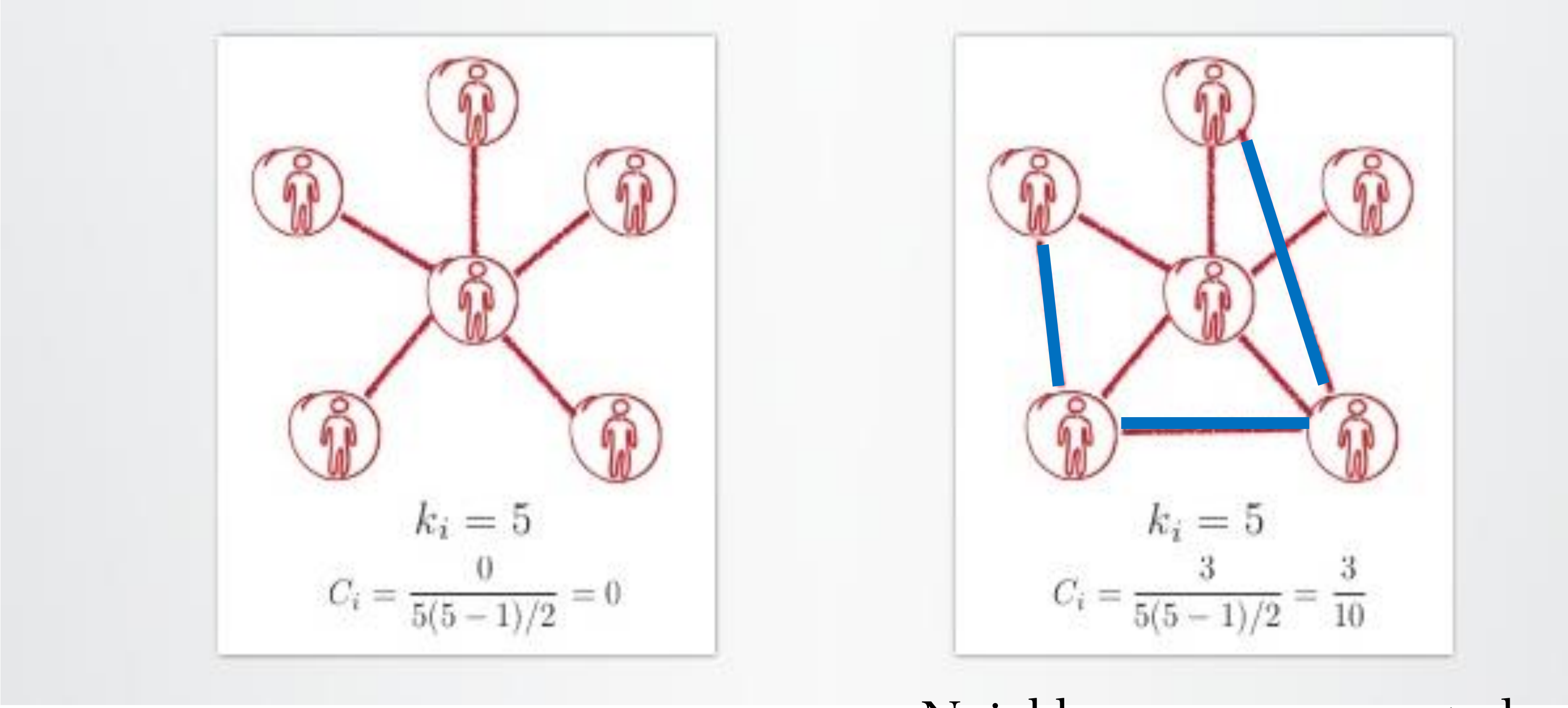


High Clustering Coefficient => highly connected neighbours

Degree and Clustering Coefficient

The influence of $\mathbf{k_i}$ (degree – number of links) and $\mathbf{C_i}$ (clustering coefficient) need to be distinguished carefully.

E.g., there are two people (centre person in each diagram) with 5 friends, but the number of links within the neighbors (of centre person) is different in both diagrams



Neighbours are connected

Clustering Coefficient

Local Clustering Coefficient is the fraction of pairs of the node's neighbors that are connected with each other. To determine the local clustering coefficient, use **nx.clustering(Graph, Node)** function.

Local Clustering Coefficient of **all nodes**

Local Clustering Coefficient of **node 'a'**

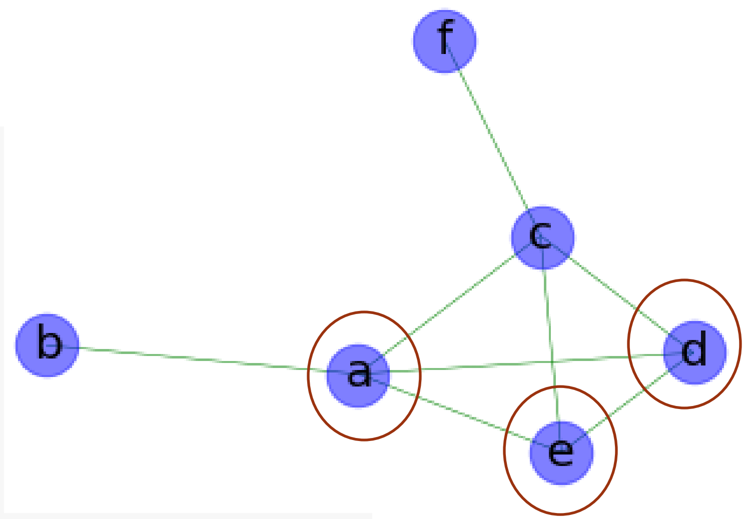
```
# Clustering
local_clustering_node = nx.clustering(G)
local_clustering_node_a = nx.clustering(G, 'a')
print("Local Clustering: " + str(local_clustering_node))
print("Local Clustering for node a: " + str(local_clustering_node_a))
avg_clustering = nx.average_clustering(G)
print("Average Clustering: " + str(avg_clustering))
```

```
Local Clustering: {'a': 0.5, 'b': 0.0, 'c': 0.5, 'e': 1.0, 'd': 1.0, 'f': 0.0}
Local Clustering for node a: 0.5
Average Clustering: 0.5
```

The **average clustering coefficient** (**sum of all the local clustering coefficients divided by the number of nodes**). To determine the average clustering coefficient, use **nx.average_clustering(Graph, Node)** function.

```
nx.average_clustering(G)
```

Clustering Coefficient




```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()

G.add_edge('a', 'b')
G.add_edge('a', 'c')
G.add_edge('a', 'e')
G.add_edge('c', 'd')
G.add_edge('c', 'e')
G.add_edge('c', 'f')
G.add_edge('a', 'd')
G.add_edge('d', 'e')

# Clustering
local_clustering_node = nx.clustering(G)
local_clustering_node_a = nx.clustering(G, 'a')
print("Local Clustering: " + str(local_clustering_node))
print("Local Clustering for node a: " + str(local_clustering_node_a))
avg_clustering = nx.average_clustering(G)
print("Average Clustering: " + str(avg_clustering))
```

```
Local Clustering: {'a': 0.5, 'b': 0.0, 'c': 0.5, 'e': 1.0, 'd': 1.0, 'f': 0.0}
Local Clustering for node a: 0.5
Average Clustering: 0.5
```

Class Exercise

Installing Anaconda and Python tools

Anaconda is a free and open-source distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment.

Download Anaconda Individual Edition from (https://www.anaconda.com/products/individual)

–

Install Anaconda on the lab machines.
Run Windows 64-bit Graphical Installer: **Anaconda3-x.x.x-Windows-x86_64.exe** with the following options:

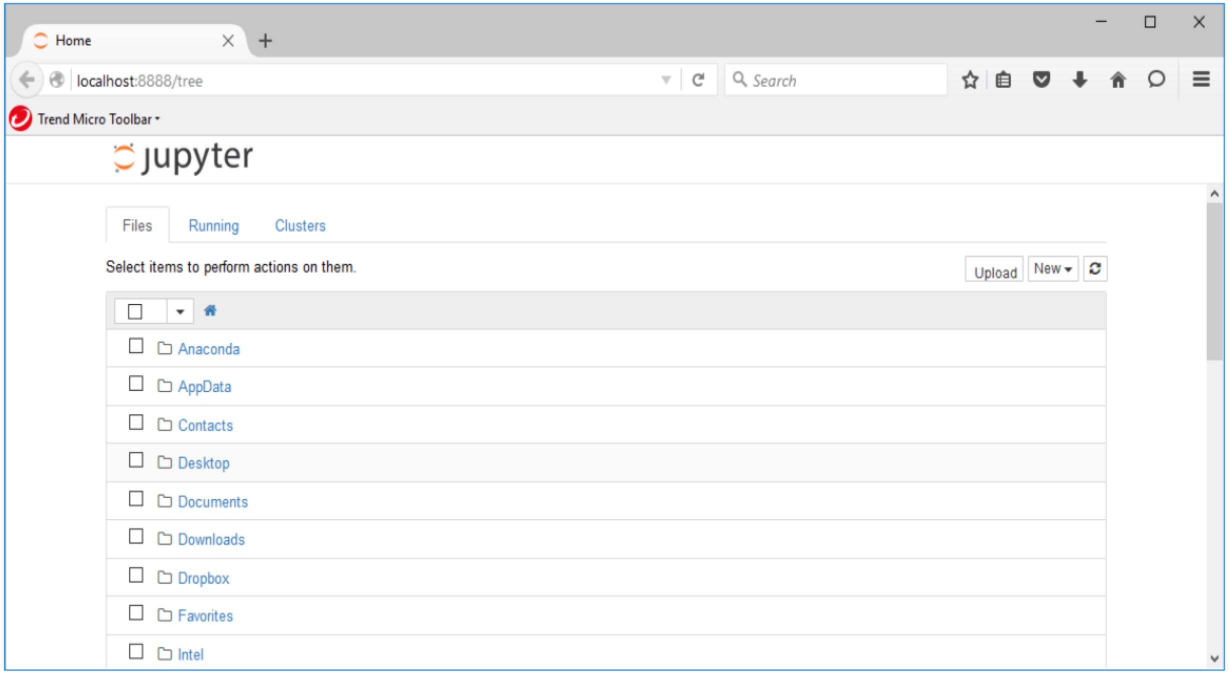
- Just Me (recommended)
- Destination folder: c:\Anaconda3

Installing Anaconda and Python tools

Launching *Jupyter Notebook App*

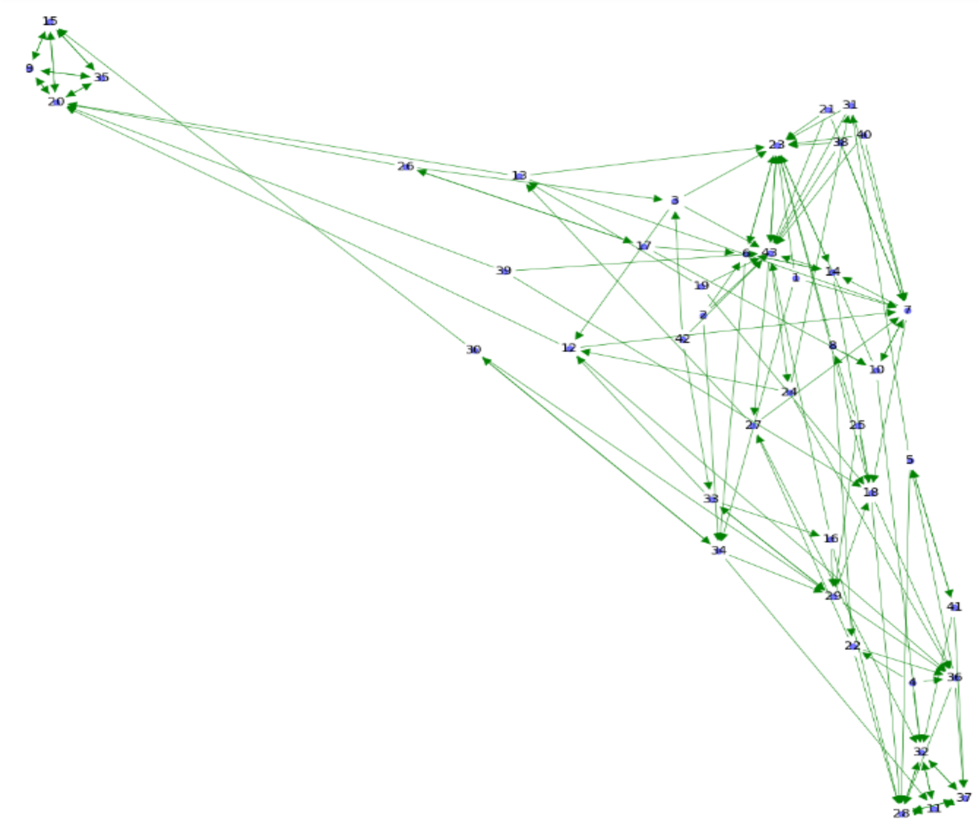
The Jupyter Notebook App can be launched by clicking on the ***Jupyter Notebook*** icon on Start (or type in "**Jupyter Notebook**" in Window Menu) or by typing in a terminal (*cmd* on Windows): **jupyter notebook**

This will launch a new browser window (or a new tab) showing the Notebook Dashboard, a control panel that allows the selection of which notebook to open.



Class Exercise

Open the **Week10_NetworkVisual_Lab.ipynb** file from the **Jupyter Notebook UI**



NANYANG TECHNOLOGICAL UNIVERSITY

SEMESTER 1 AY2024-2025 – CLASS ASSESSMENT

IN6221 – INFORMATION VISUALIZATION

15 NOVEMBER 2024 Time Allowed: **2 hours**

INSTRUCTIONS

1. This paper contains **FOUR (4)** questions.
2. Answer **ALL FOUR (4)** questions. (40 marks)
3. This is an **OPEN BOOK** test.
4. Type your answer directly below each question.
5. Answers are to be in Times New Roman, 12 point font.
6. Do not alter the margins or any other document setting.
7. Each question and respective answer **MUST** begin on a new page.

Full Name (as in your matriculation card)

Matriculation Card Number

Instructions

Attendance will be taken.

6:55 PM - Students will be given 5 mins to download the question file and prepare themselves. The file is located at the **Content** Section (where your lecture notes are) under the **Class Test** folder.

The test will start at **7:00 PM** and end at **9:00 PM.**

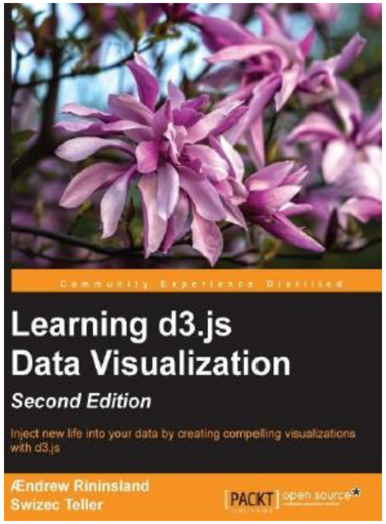
9:00 PM to **9:10 PM** – submit the **answer file** and **codes** in a zip file through **Turnitin** for Class Test found in the same **Assignment** Section.

Write your answers (including **screen shots** of your **codes** and **visualization**) in the word document. Remember to periodically save the file to avoid losing your work.

Save your answer file according to following naming convention:
ClassTest**_Name_MatricNumber.docx**

References

- Rininsland, A. (2016). Learning d3.js Data Visualization 2nd Ed. Packt Publishing.



- Murray, S.(2017) Interactive Data Visualization for the Web, 2nd Ed. O'Reilly.