Daniel Matar

219065161

SIT320 Module 3a: Graphs – Part 1

Contents

[Graphs 3](#_Toc78526271)

[Undirected Graph 4](#_Toc78526272)

[Directed Graph 4](#_Toc78526273)

[Graph Representation 5](#_Toc78526274)

[Adjacency Matrix 5](#_Toc78526275)

[Linked Lists 6](#_Toc78526276)

[Can you identify a unique problem that constitutes a graph? 7](#_Toc78526277)

[Identify what graph analysis would be useful? 7](#_Toc78526278)

[Linked lists representation of un-directed vs directed graphs 7](#_Toc78526279)

[Depth-First Search 9](#_Toc78526280)

[How does it work? 9](#_Toc78526281)

[Applications of DFS 10](#_Toc78526282)

[Breadth-First Search 10](#_Toc78526283)

[How does it work? 10](#_Toc78526284)

[Applications of BFS 11](#_Toc78526285)

[Bi-partite Graph 12](#_Toc78526286)

[Why is this important? 12](#_Toc78526287)

[Strongly Connected Graphs 12](#_Toc78526288)

[Why is this important? 12](#_Toc78526289)

[DFS Forest 13](#_Toc78526290)

[Design an algorithm for Topological Sorting, and test it on the following path: 15](#_Toc78526291)

[Code 16](#_Toc78526292)

[Write down the algorithm and code for determining the distance of nodes from a specified node, using BFS? 17](#_Toc78526293)

[Code 17](#_Toc78526294)

[Design an algorithm using BFS to determine if a graph is bi-partite? 19](#_Toc78526295)

[Code 19](#_Toc78526296)

[Write down the pseudo-code for determining the Strongly Connected Components in a graph 21](#_Toc78526297)

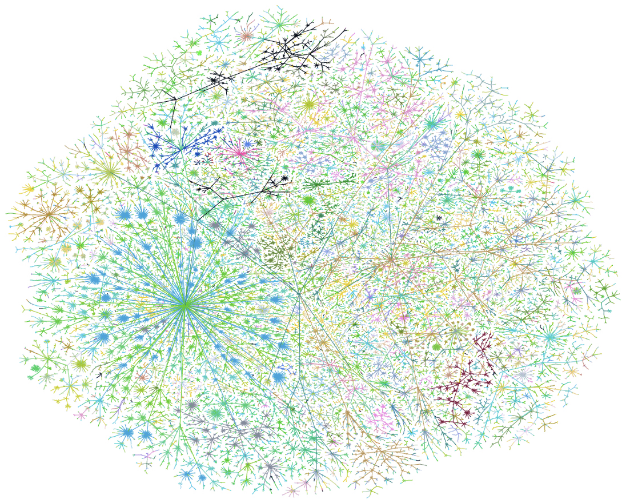
[Code 21](#_Toc78526298)

[References 22](#_Toc78526299)

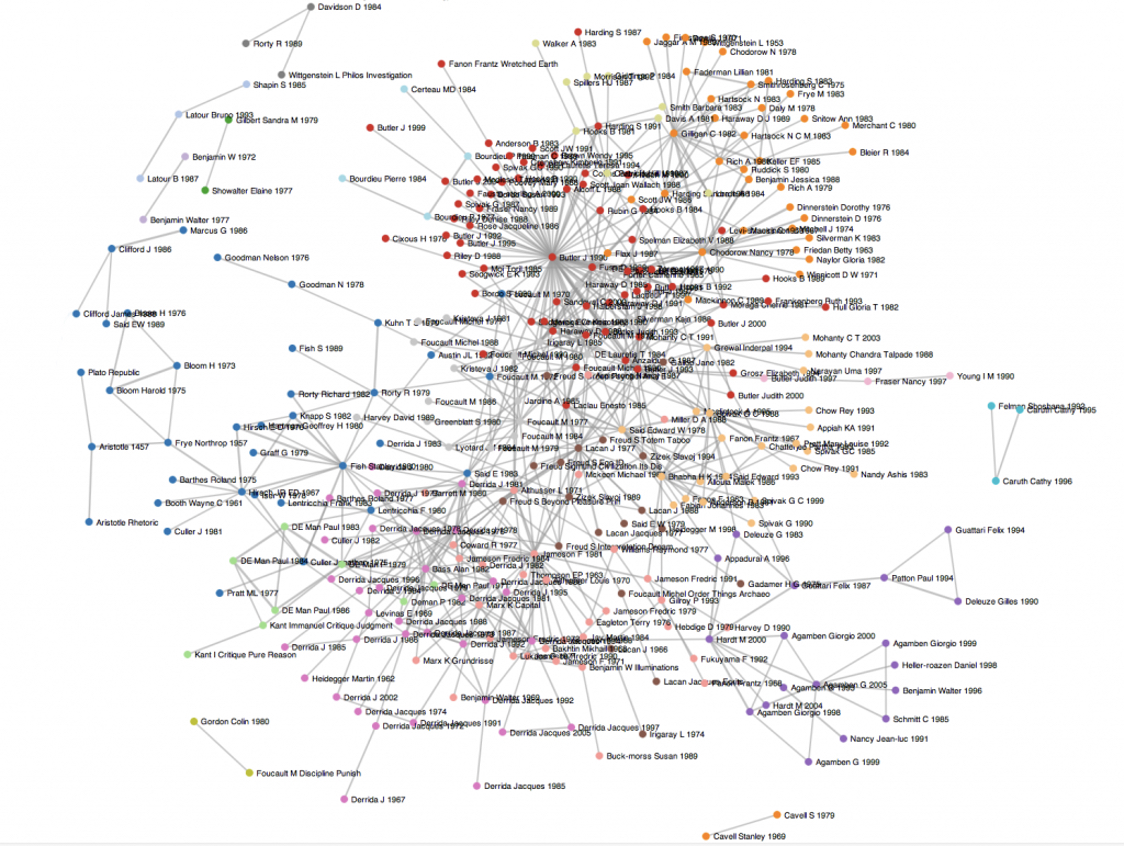
# Graphs

Graphs in the world of computing are a ubiquitous software. We will be going through Breadth-first Search (BFS), Depth-first Search (DFS), finding bi-partite graphs, and strongly connected components.

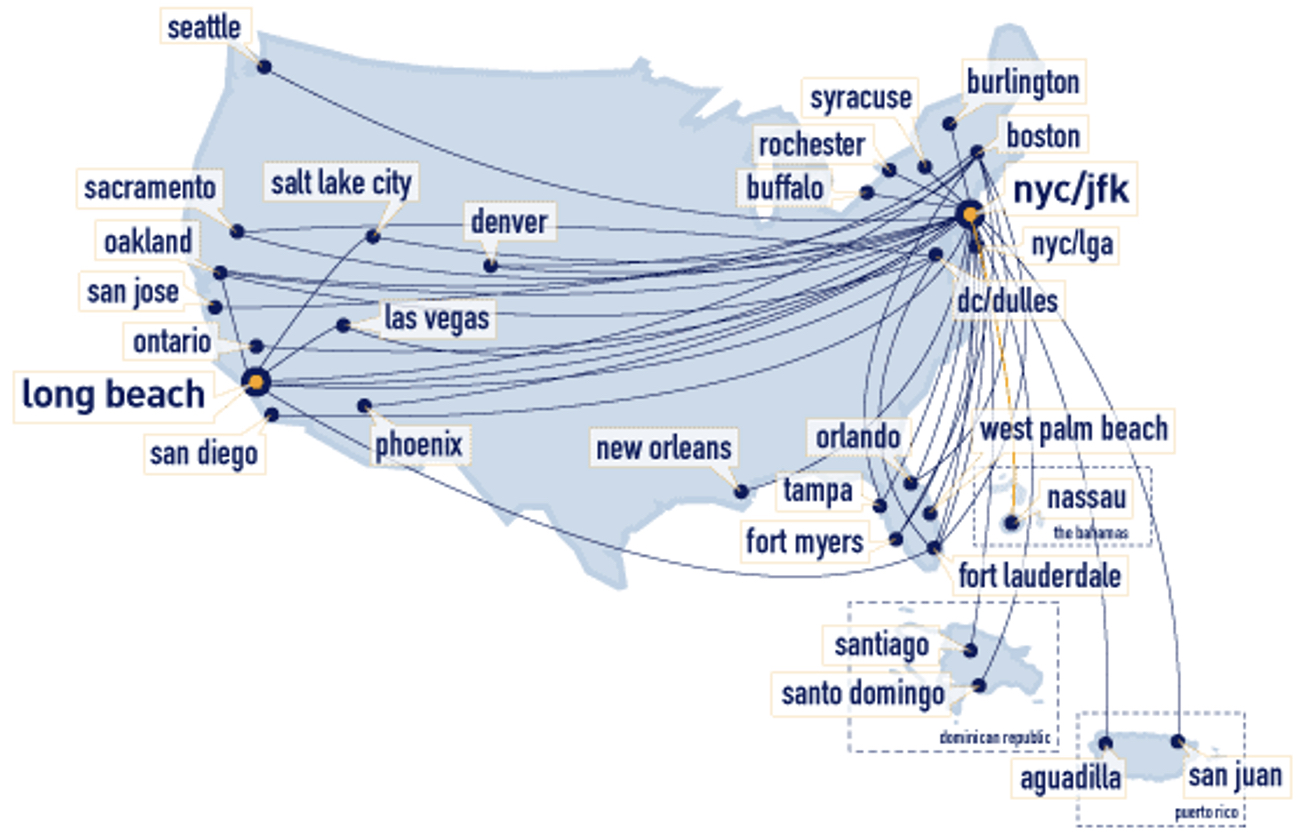
A few examples of graphs include:



[1] Figure Graph of the internet (1999)



[1] Figure Citation graph of Literary Papers



[1] Figure JetBlue Flights in US

The number of graphs available in our world is enormous. Many websites exist that provide graph datasets, here a few to list.

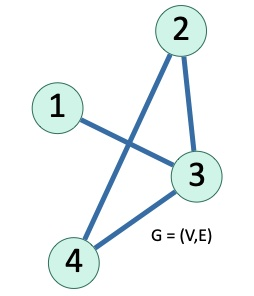
1. <https://snap.stanford.edu/data/>
2. <https://www.kaggle.com/startupsci/awesome-datasets-graph>
3. <http://networkrepository.com/additional-resources.php>

Graphs consist of two parts, Nodes and Vertices (or Edges), and can be directed or undirected.

## Undirected Graph

Undirected graphs are represented as **G = (V,E)**:

Where the graph is, a set of vertices **V**, and a set of edges **E**. An example of this is shown below:



[1] Figure Undirected Graph Example

In **figure 4** above, we have the following:

The vertices **V = (1,2,3,4)**

and

The edges **E = ((1,3), (2,4), (3,4), (2,3))**

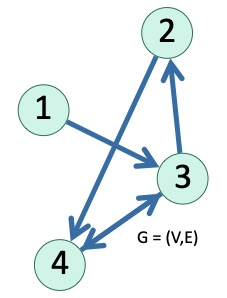
The **degree of a node** is defined as the **number of vertices connected to it**. An example of this using the **figure 4**, the vertex 4 has a **degree of** **2.**

Vertex 4 has **2** edges coming out and its neighbours are **2 and 3.**

## Directed Graph

Directed graphs are also represented as **G = (V,E).**

Where the graph is, a set of vertices **V**, and a set of **directed edges** **E**. An example of this is shown below:



[1] Figure Directed Graph Example

In **figure 5** above, we have the following:

The vertices **V = (1,2,3,4)**

and

The directed edges **E = ((1,3), (2,4), (3,4), (4,3), (3,2))**

The degree is defined as either **in-degree or out-degree.**

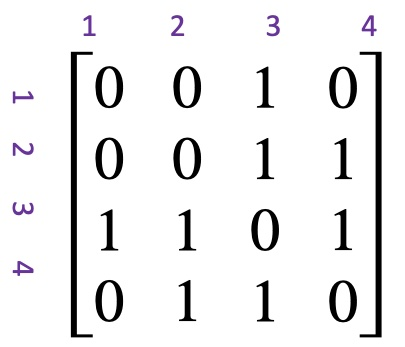
The **in-degree of vertex 4 is 2**. The **out-degree of vertex 4 is 1.**

Vertex 4’s incoming neighbours are **2 and 3, and its outgoing neighbour is 3.**

## Graph Representation

## Adjacency Matrix

Typically, a matrix or list is used to represent a graph. The matrix is more known as the adjacency matrix, an example of this is shown below:



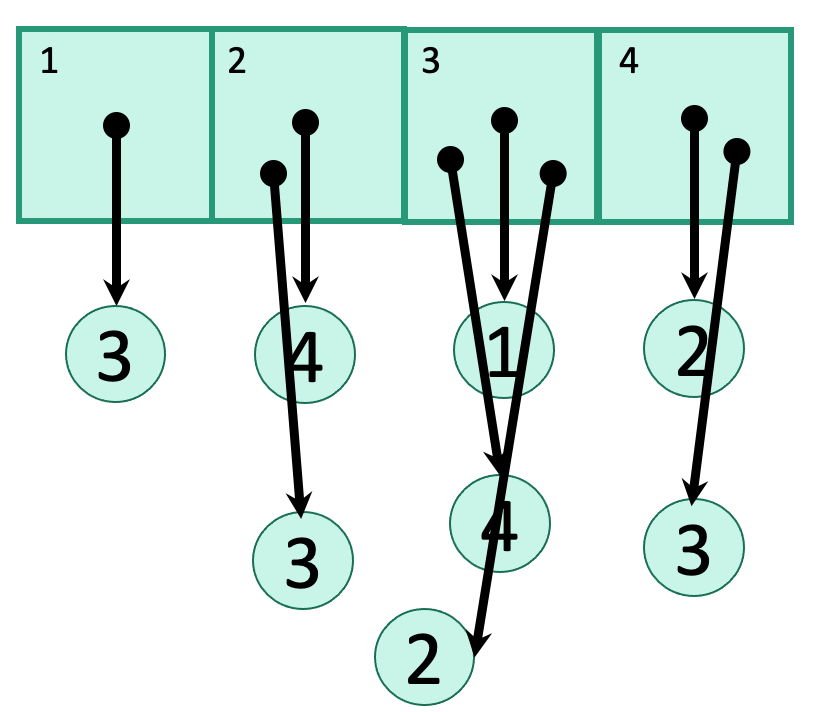
[1] Figure Adjacency Matrix Un-Directed Graph Example

In figure 6 we can see that this matrix is symmetric. Since the graph is un-directed a link from **2 to 3** would also imply there is a link from **3 to 2** as well. If our adjacency matrix is symmetric, we can assume that our graph is un-directed.

Matrix representation only works with if the graph we have is dense (meaning it has many edges), though if the graph’s edges are sparse, this would not output a good representation of our graph. In this case we would use linked lists.

## Linked Lists

Linked list representation of our un-directed graph would look like the below:

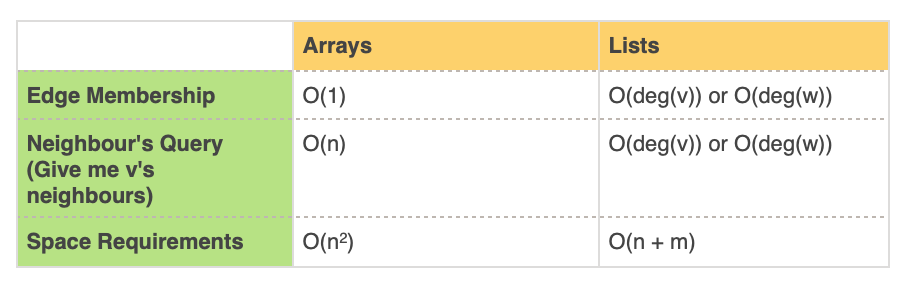


[1] Figure Linked List Un-Directed Graph Example

This representation is quite like the BST representation in the previous module.

In figure 7 we can see **1 links to 3, 2 links to 4 and 3, and so on.**

The efficiency of lists and adjacency matrix are shown below:



[1] Figure Linked List vs Matrix Efficiency

The **Edge Membership** operation checks if **e = (v,w) in E?**

And the **Neighbour’s Query**, gives **v’s neighbours**.

Where **n** is the number of nodes and **m** is the number of edges.

## Can you identify a unique problem that constitutes a graph?

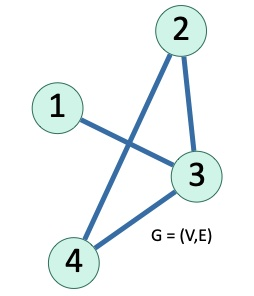
The problem to where the solution would constitute a graph would be where the data you have is connected, has a trend that can be seen. For example, what the average rainfall is in a suburb in NSW compared to its neighbouring suburbs.

### Identify what graph analysis would be useful?

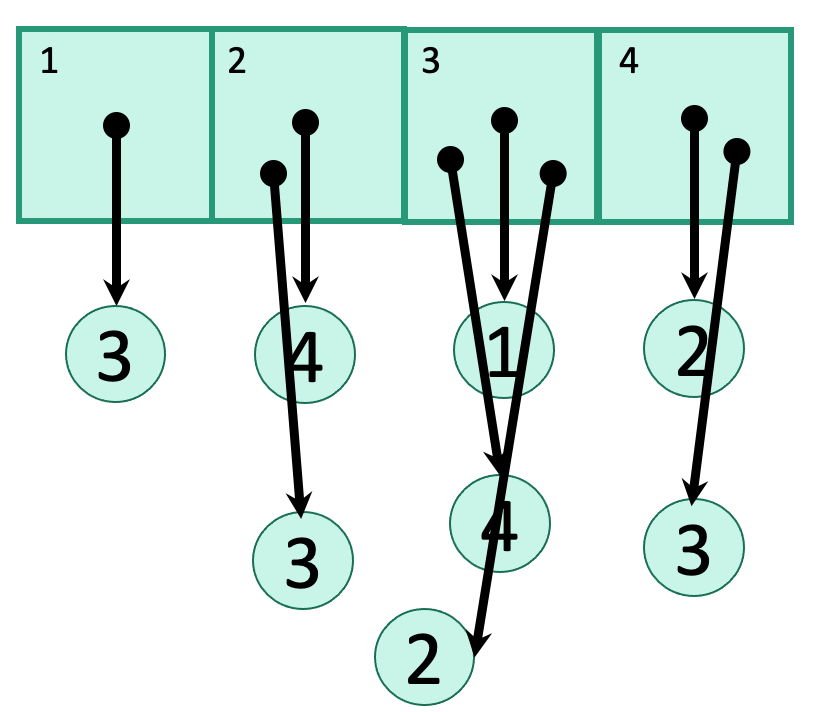
For this analysis a **Line Graph** would be best used for this scenario. As the changes between our data would be relatively small and we can use this to compare changes over the same period for more than one group.

## Linked lists representation of un-directed vs directed graphs

As seen beforehand in figure 7, shown again below, our example of an un-directed graph using figure 5 as the graph itself:

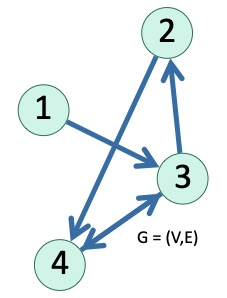


[1] Figure Undirected Graph Example



[1] Figure Linked List Un-Directed Graph Example

While the representation for a directed graph would look more like this:



[1] Figure Directed Graph Example

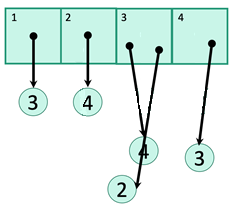


Figure 9 Linked List Directed Graph Example

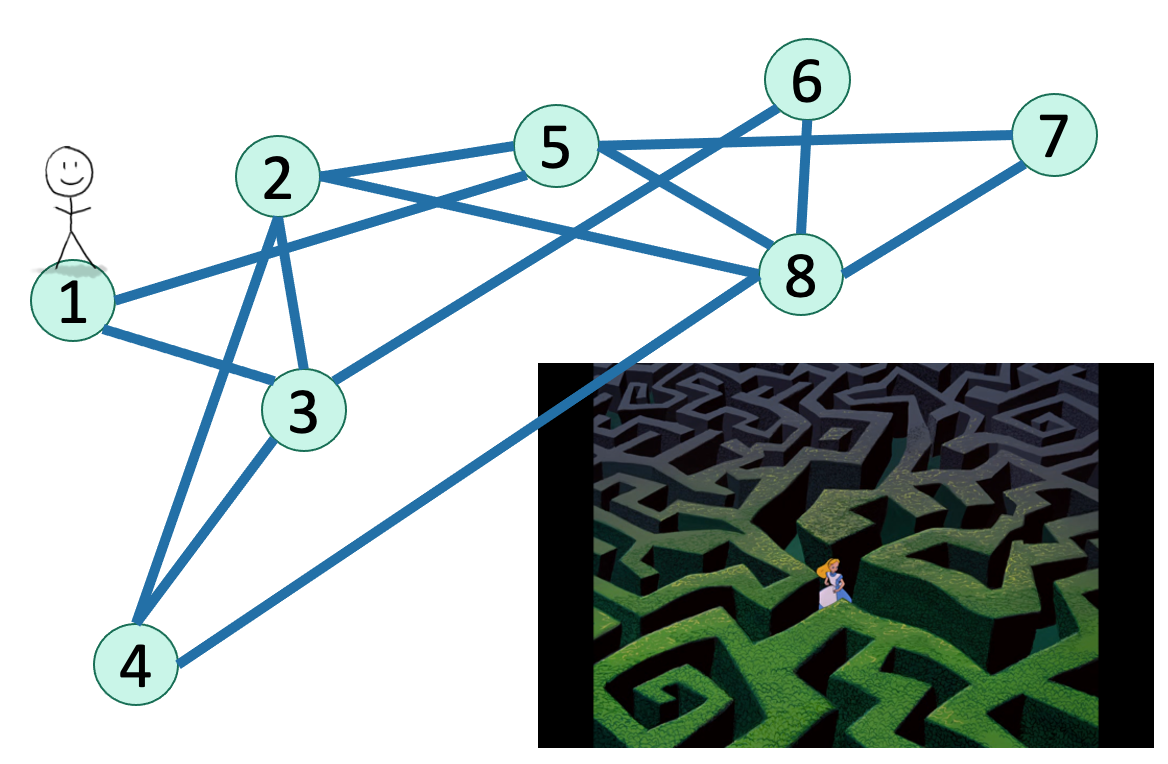
As can be seen by these representations the directed linked list removes links between edges where it is not bi-directional links.

## Depth-First Search

There are various ways to explore graphs. Depth-first search (DFS) being the easiest of them.

### How does it work?

Depending which node, we start at, there are many options for the first move, based on the number of neighbours. This exploration can also be like how we would explore a maze, depending on our current options, what is something we haven’t visited. Like the example shown below:



[2] Figure 10 Graph Traverse vs Maze Example

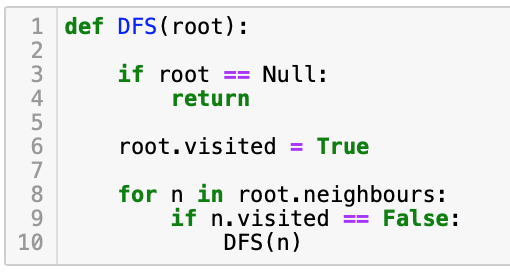
DFS simply works with few things to keep track of:

* Unvisited node
* In progress, and
* All Done

Also with each vertex we keep track of:

* The time we enter the vertex, marking it ‘In Progress’
* The time we finished with the vertex, marking it ‘All Done’

An example of this as an algorithm is shown below:



[2] Figure 11 DFS Recursive Algorithm Example

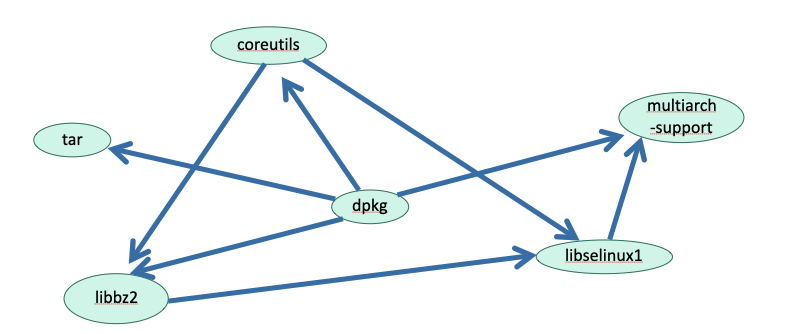
DFS can find all the nodes that are reachable from the starting point of a graph and is quite simple. Though if the graph is not entirely connected it would need to be re-run to find all nodes.

DFS can also work on trees, as they are a form of graphs.

DFS complexity for a single connected graph is **O(m) where m is the number of edges**. For multiple connected graphs the complexity increases as we would have to run our DFS multiple times resulting in a complexity of **O(n+m), where n is the number of connected graphs**.

### Applications of DFS

One application of DFS is known as **Topological Sorting**, which is a way to sort nodes in a way that dependencies are respected. An example of this is shown below:



[2] Figure Topological Sorting Example

**Topological Sorting** can help us understand in which order do we install these packages.

Another application of DFS is **Graph Traversal**. Which is quite simple and can be done in the form of:

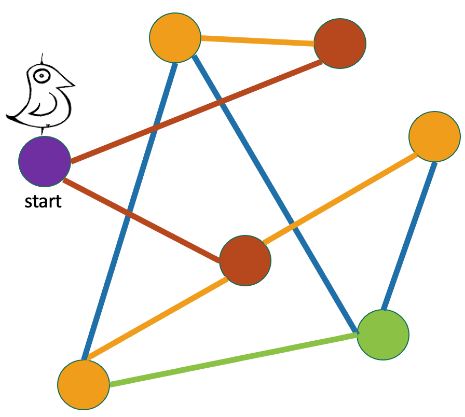
* In-order
* Post-order, or
* Pre-order

## Breadth-First Search

Another way to explore graphs is, Breadth-first Search (BFS), this is a common alternative to DFS.

BFS is less intuitive (due to it not being recursive) and implements the queue data structure to search.

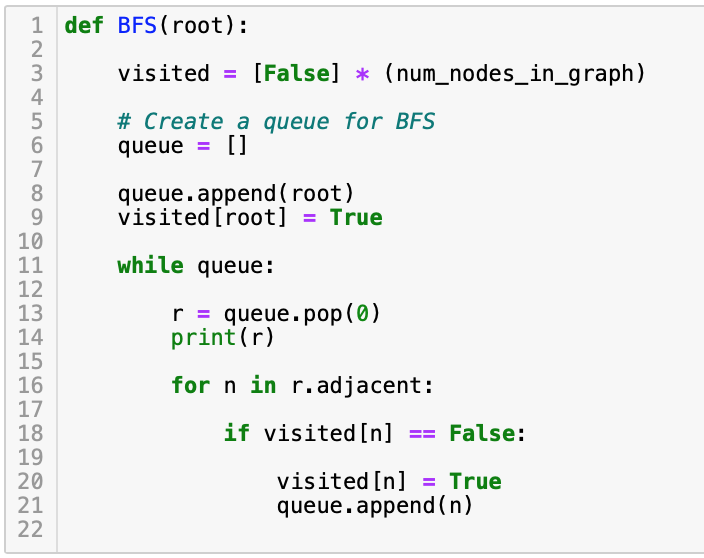
### How does it work?

BFS explores graphs with a bird-eyes view, as shown in the below figure: 

[3] Figure Breadth-First Search Example

If our starting node is where the bird is in figure 13, we would colour-code the nodes depending on how far they are to us. Each colour representing the amount of steps to get to that specified node.

The algorithm for BFS is still simple, though still more complex than BFS. An example is shown below:



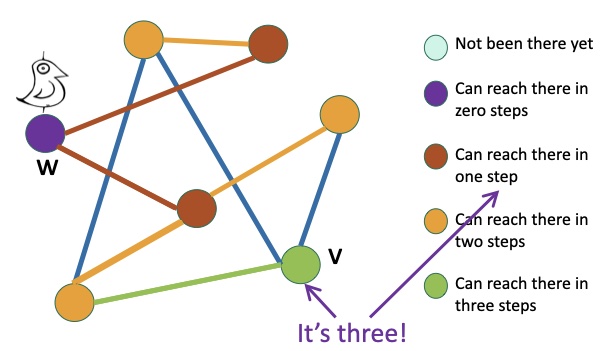
[3] Figure Breadth-first Search Algorithm Example

From this algorithm we can see it is not recursive like DFS. In BFS, visiting a node, will lead to visiting all its neighbours before visiting any other neighbours. This can be though of as level-by-level clearing.

Like DFS, BFS can also find all the nodes reachable from a starting point. The complexity is like DFS, being, **O(n + m)**, where **n is the number** **of connected graphs** and **m is the number of edges**.

### Applications of BFS

An application of BFS, is to find the shortest path. This is used is a variety of places, most commonly on maps, an example of this is shown below:

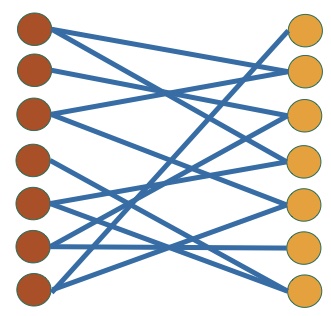


[3] Figure Shortest Path Using BFS Example

If we want to find the shortest path between node **w and v**, we can see it would only take us as little as **3 steps** to reach **v**.

## Bi-partite Graph

Bi-partite graphs look like the below:



[4] Figure Bi-Partite Graph Example

Bi-partite, checks if it can find two sets of nodes, such that there is no other connection to the nodes in the same set. The edges only exist among nodes of two different sets. In figure 14, we have two sets, red and orange. Connections do not exist between the sets themselves only between the two.

### Why is this important?

This is important for graphs to see correlations between different sets of data and separate things that are not.

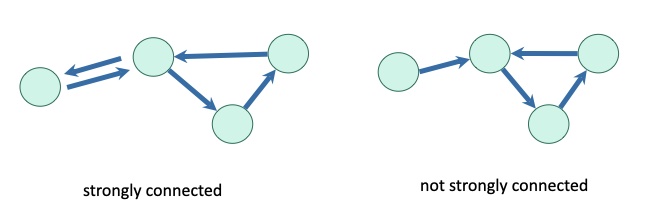
The algorithm for this uses BFS, with one change. We alter the colour of the node at each ‘level’. And if we find two nodes coloured the same connected to each other, we have a bi-partite graph, else we don’t.

## Strongly Connected Graphs

A directed graph is strongly connected if:

* For all **v, w** in **V:**
  + There is a path from **v to w** and
  + There is a path from **w to v**

An example of this is shown below:



[5] Figure Strongly Connected Graph Example

### Why is this important?

This is quite useful, as it lends itself to community or cluster detection which is quite handy for a variety of purposes.

A simple way to determine Strongly Connected Components (SCCs) in a graph is to use a DFS.

With a few steps we can do this:

* For each pair (**u, v**), we use DFS to find if there is a path from **u to v, and v to u**
* Collect information about all the pairs of nodes and aggregate.

Though this approach is quite brute force and can be quite tedious, meaning the complexity of our algorithm would be **O()**.

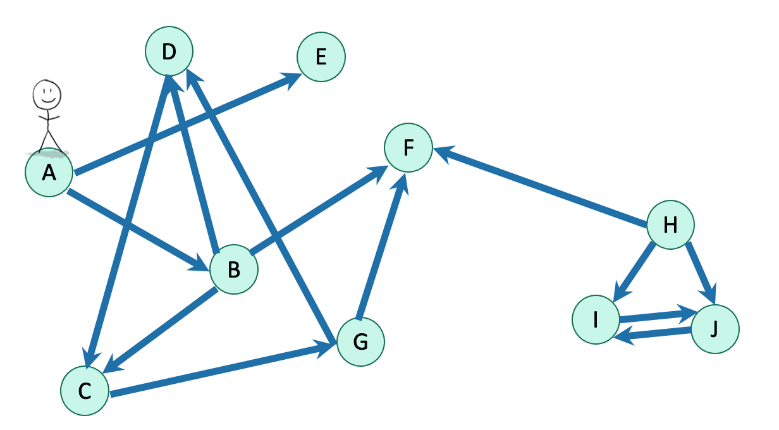
This is something we would like to avoid and in such there is a much better way to approach this.

### DFS Forest

When we continuously run DFS, we result in clustered trees of all the connected nodes.

This can be shown by the example below:

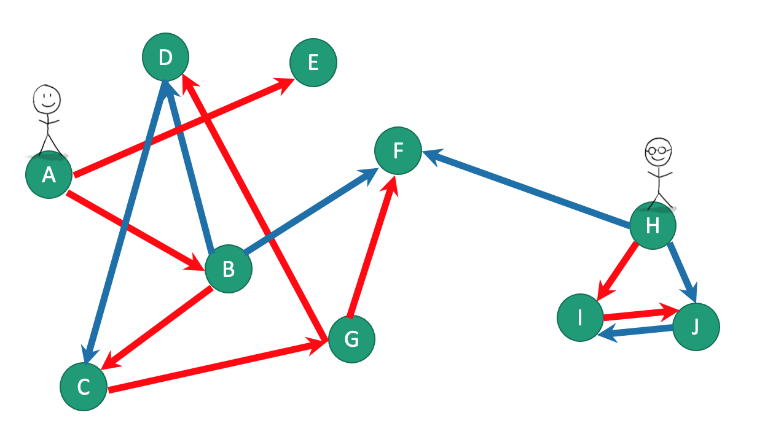
We start at node **A.**



[5] Figure SCCs using DFS Example

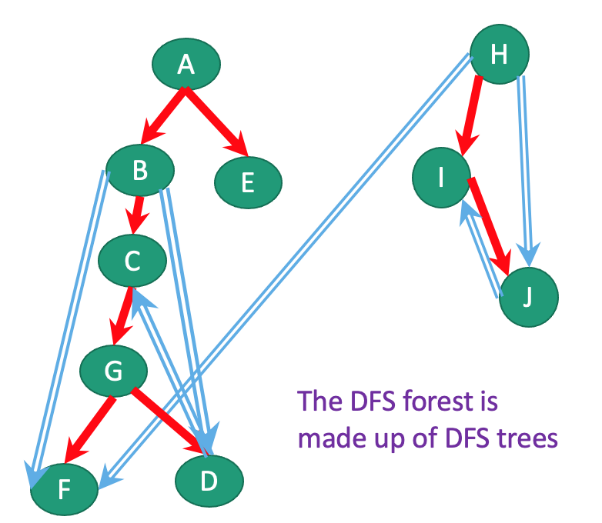
From this we can explore the nodes **A, B, C, D, E, F and G.**

Though **H, I and J** do not get explored, thus we run another DFS starting from one of these nodes.



[5] Figure SCCs using DFS Example 2

This repeated DFS results in two **Trees** comprising a **depth-first forest**. Which can be shown in the form of trees:

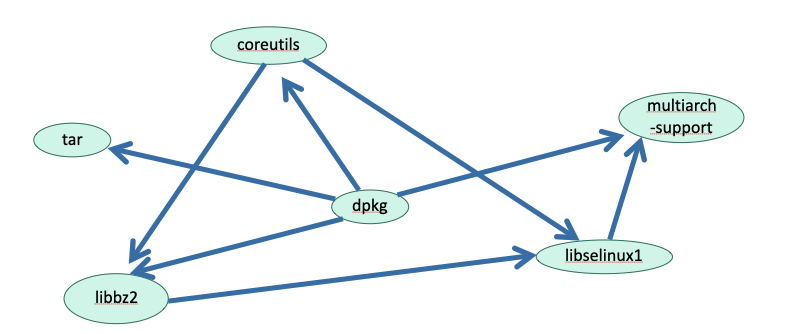


[5] Figure Depth-first Forest Representation as Trees

From this we can use DFS and Depth-first Forests to establish SCCs by doing these steps:

* Do repeated DFS to create a DFS Forest, while:
  + We keep track of finishing times of each node
* Once this is done, we reverse all the edges in the graph
* And we do another repeated DFS to create another DFS Forest
  + This time, we order the nodes in reverse order of their finishing times that they had from the first run
* The SCCs are the different trees in the second DFS Forest

# Design an algorithm for Topological Sorting, and test it on the following path:



### Code



## Write down the algorithm and code for determining the distance of nodes from a specified node, using BFS?

### Code



## Design an algorithm using BFS to determine if a graph is bi-partite?

### Code



## Write down the pseudo-code for determining the Strongly Connected Components in a graph

### Code



# References

|  |  |
| --- | --- |
| [1] | N. Zaidi, “Deakin Sync Module 3a Graphs Part 1,” [Online]. Available: https://d2l.deakin.edu.au/d2l/le/content/1031061/viewContent/5741932/View. [Accessed 29 07 2021]. |
| [2] | N. Zaidi, “Deakin Sync Module 3a Graphs Part 1 - DFS,” [Online]. Available: https://d2l.deakin.edu.au/d2l/le/content/1031061/viewContent/5742250/View. [Accessed 29 July 2021]. |
| [3] | N. Zaidi, “Deakin Sync Module 3a Graphs Part 1 - BFS,” [Online]. Available: https://d2l.deakin.edu.au/d2l/le/content/1031061/viewContent/5742259/View. [Accessed 29 July 2021]. |
| [4] | N. Zaidi, “Deakin Sync Module 3a Graphs Part 1 - Bi-Partite Graphs,” [Online]. Available: https://d2l.deakin.edu.au/d2l/le/content/1031061/viewContent/5742395/View. [Accessed 29 July 2021]. |
| [5] | N. Zaidi, “Deakin Sync Module 3a Graphs Part 1 - Strongly Connected Graphs,” [Online]. Available: https://d2l.deakin.edu.au/d2l/le/content/1031061/viewContent/5742396/View. [Accessed 29 July 2021]. |
| [6] | NCES, “NSES - Graphing Tutorial,” [Online]. Available: https://nces.ed.gov/nceskids/help/user\_guide/graph/whentouse.asp. [Accessed 29 July 2021]. |
| [7] | Javatpoint.com, “Javatpoint,” [Online]. Available: https://www.javatpoint.com/graph-representation. [Accessed 29 July 2021]. |
| [8] | N. Yadav, “Geeks For Geeks - SCCs,” [Online]. Available: https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/. [Accessed 29 July 2021]. |
| [9] | D. Mehta, “Geeks For Geeks - Bi-partite Graph,” [Online]. Available: https://www.geeksforgeeks.org/bipartite-graph/. [Accessed 29 July 2021]. |