# Data Structures
# Introduction to Graphs

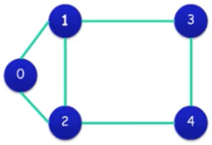
Cain Susko

Queen's University
School of Computing

March 22, 2022

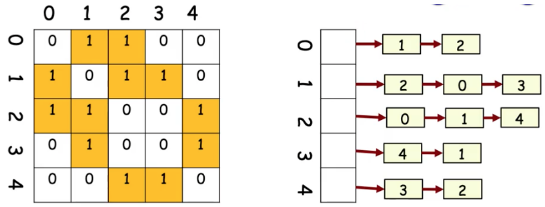
# Graphs

Grpahs are made up of nodes which can be anythinf from data, to a person, to another network). The nodes are connected via edges which define with nodes are related to which.



# Graph Reoresentation

A graph can be represented as either an adjacency matrix or edge list:



|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 2 | 1 | 1 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 1 | 0 |

0 → 1 → 2
1 → 2 → 0 → 3
2 → 0 → 1 → 4
3 → 4 → 1
4 → 3 → 2

Because of the flexibility of the edge list, which is a linked list of arrays, it is a better option to implement and represent a graph in code. However, if you know your graph will be quite full, this flexibility isn't used and the speed from pre-allocating an array (adjacency matrix) will be more helpful.

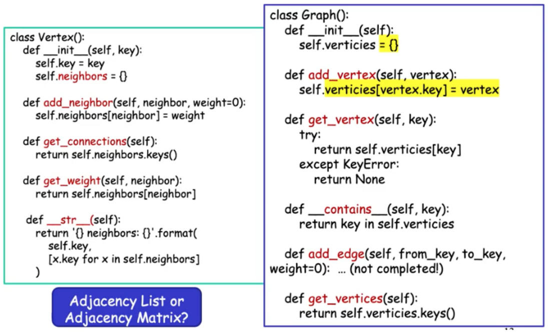
Graphs can be both dense or sparse, an example of a sparse graph is a subway map like so:

1 Vaughan Metropolitan Centre
Highway 407
Pioneer Village
York University
Finch West
Downsview Park
Sheppard West
Wilson
Yorkdale
Lawrence West
Glencairn
Eglinton West
St Clair West
Dupont
Spadina
St George
1 Finch
North York Centre
4 Sheppard-Yonge
Bayview
Bessarion
Leslie
Don Mills 4
York Mills
Lawrence
Eglinton
Davisville
St Clair
Summerhill
Rosedale
Pearson Airport
192 Airport Rocket
2 Kipling
Islington
Royal York
Old Mill
Jane
Runnymede
High Park
Keele
Dundas West
Lansdowne
Dufferin
Ossington
Christie
Bathurst
Bay
Bloor-Yonge
Sherbourne
Castle Frank
Broadview
Chester
Pape
Donlands
Greenwood
Coxwell
Woodbine
Main Street
Victoria Park
Warden
Kennedy 2 3
Lawrence East
Ellesmere
Midland
Scarborough Centre
McCowan 3
Museum
Queen's Park
St Patrick
Osgoode
St Andrew
Union
Wellesley
College
Dundas
Queen
King
N

## Graph as ADT

A conceptual Graph would have the following operations:

- Test if graph is empty
- Get number of nodes in graph
- Get number of edges in graph
- See whether edge exists between 2 nodes
- Insert node into graph
- Insert edge between nodes in graph
- Remove node from graph and any edges to said node
- Remove edge between two nodes in graph



```
class Vertex():
    def __init__(self, key):
        self.key = key
        self.neighbors = {}

    def add_neighbor(self, neighbor, weight=0):
        self.neighbors[neighbor] = weight

    def get_connections(self):
        return self.neighbors.keys()

    def get_weight(self, neighbor):
        return self.neighbors[neighbor]

     def __str__(self):
        return '{} neighbors: {}'.format(
            self.key,
            [x.key for x in self.neighbors]
        )
```

Adjacency List or Adjacency Matrix?

```
class Graph():
    def __init__(self):
        self.verticies = {}

    def add_vertex(self, vertex):
        self.verticies[vertex.key] = vertex

    def get_vertex(self, key):
        try:
            return self.verticies[key]
        except KeyError:
            return None

    def __contains__(self, key):
        return key in self.verticies

    def add_edge(self, from_key, to_key,
weight=0):  ... (not completed!)

    def get_vertices(self):
        return self.verticies.keys()
```

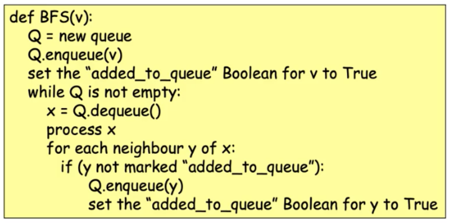
## Traversal

like with a tree (which is almost a kind of graph) there are may way to traverse a graph:

**Breadth first Search** This way explores the graph in **growing concentric circles**, exploring a node's first edge, then their second, etc...Iterating over Graph

**Depth first Search** A depth first search keeps moving forward until it hits a dead end or a previously-visited node. It then backtracks and tries another path.

## Breadth First Implementation



```
def BFS(v):
  Q = new queue
  Q.enqueue(v)
  set the "added_to_queue" Boolean for v to True
  while Q is not empty:
    x = Q.dequeue()
    process x
    for each neighbour y of x:
      if (y not marked "added_to_queue"):
          Q.enqueue(y)
          set the "added_to_queue" Boolean for y to True
```