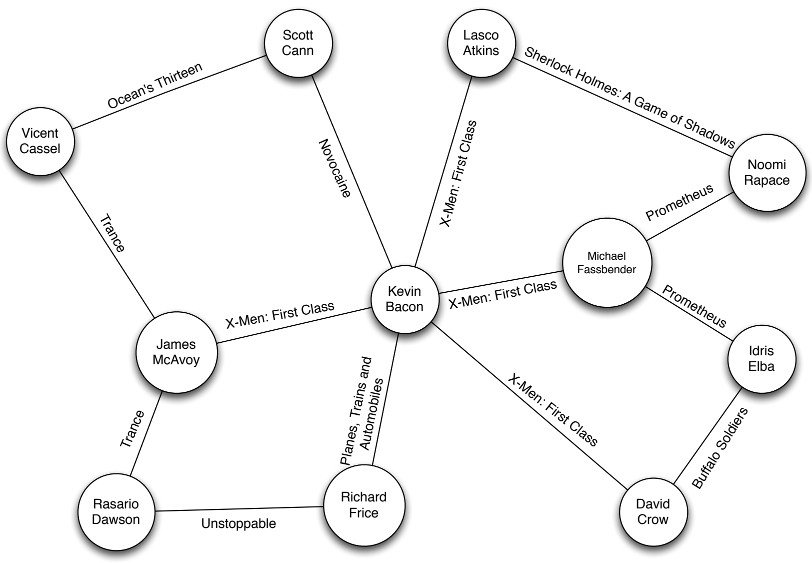
Graphs

Graphs are used to represent complex relationships. They use abstraction to remove unnecessary detail from a given problem in order to provide a representation that is easy to interpret.

A graph consists of **vertices** that are connected together by **edges.** These edges can be labelled to provide information about the relationship.



Relationships

The relationships that each vertex has can be summarised using the terms **neighbour** and **degree**.

- **Neighbour** - if two vertices are connected by an edge they are neighbours.
- **Degree** - the number of neighbours for a particular vertex.

The graph to the right shows part of the **Hollywood** graph. This graph represents which actors have worked together.

We can use a tool called **Network X** to create graphs in Python.

Exercise One

a) Recreate the section of the Hollywood graph provided above in Python using the Network X tool.

b) Get the **neighbours** and **degree** for the actor Kevin Bacon.

c) Investigate the Oracle of Bacon to find out more about the Kevin Bacon game and Bacon numbers.

Python and Network X

To make use of Network X in Python you must import it into your current module:

```
import networkx as nx
```

We use the **as** syntax to shorten the name of the module so that it is not so cumbersome to use in our code.

A basic graph is created with the following code:

```
my_graph = nx.Graph()
```

A **node** can be added to the graph quite easily:

```
my_graph.add_node(“Sarah”)
```

An **edge** can be described:

```
my_graph.add_edge(“Sarah”,”John”)
```

When creating an edge existing nodes are used where possible but if you specify an edge with a non-existent node then it will be created for you.

You can see which nodes and edges are part of a graph by calling the following methods:

```
my_graph.nodes()
my_graph.edges()
```

The neighbours and degree for a particular node can be accessed by using the methods:

```
nx.neighbors(my_graph,“Sarah”)
nx.degree(my_graph,“Sarah”)
```

The Tube

The London tube map is one of the most well know examples of a graph. The tube map provides a clear and easy to understand representation of the tube network because it abstracts away unnecessary detail and leave only the relationships between the stations and the different lines that are needed to navigate underground.

We are going to develop an application that will provide directions to users. They will be able to specify a start and end station and the program will work out the route (including any line changes) that are needed to get from A to B.

The class to represent the Tube Network is provided below:

TubeMap
map file_name
get_stations generate_edge_colours create_graph_plot display_full_map display_travel_map get_directions

Exercise Two

a) The data for the Tube map is provided as a CSV file. Make sure you have a copy of this file.

b) Complete the **get_stations** method - this will generate the tube map from the CSV file.

There is a method called **add_path** (see side bar) that will enable you to add all of the stations that are connected in one line to the graph in a single step.

We can also add a data object to the path that we can use to label each edge - again see the side bar.

Further Network X methods

There are many additional methods that Network X provides that you will need to be familiar with.

We can create an entire **path** - a sequence of edges that begin at one vertex and travels to another vertex along the edges of the graph:

```
my_graph.add_path(node_list)
```

Notice that the parameter passed to the method is a **list**. The **add_path** method will add each node in the list to the graph and generate edges between nodes that are adjacent in the list.

We can also **label** each edge of the path by providing an additional object to the **add_path** method as its **data** parameter:

```
my_graph.add_path(node_list,dat
a=”label_dict”)
```

Normally we use a **dictionary** as the data object.

A dictionary is useful as it allows you to associate values with particular keys - this makes it easy to retrieve the data later.

If you are unsure about dictionaries speak to your teacher.

Exercise Three

a) The **generate_edge_colours** method should **return** a list containing the colour to use to represent each edge on the map.

   In the previous task you added a data object to each edge that contains the necessary data.

   It is possible to use the **get_edge_data** method get this data object for each edge - see the side bar for further details.

Exercise Four

a) The **create_graph_plot** makes use of another Python tool called **matplotlib**.

   The tool enable us to create graphs and charts very easily - see the side bar for further details.

   This method takes the current graph and then uses it to:

   - Generate the position for each node
   - Get the colours for each edge
   - Plot the nodes of the graph
   - Plot the edges of the graph
   - Plot the labels on the graph

More Network X methods

It is possible to return the data object that can be attached to an edge:

```
my_graph.get_edge_data(“John”,”Sarah”)
```

Calling this method will return a dictionary that will contain the data object that you added previously.

Matplotlib and Network X

To use some of the basic functionality of matplotlib you must add the following import statements to your program:

```
import matplotlib.pyplot as plt

import numpy
```

To produce a graph you must first of all generate the positions for each node:

```
pos = nx.spring_layout(my_graph,iterations=1000)
```

Don’t worry too much about the iteration parameter, it just impacts on the spacing of the nodes.

Once the positions have been generated the figure can be plotted by calling:

```
plt.figure()
```

After plotting the blank figure we can then get Network X to draw the different components of the graph to the figure:

```
nx.draw_networkx_nodes(my_graph,pos,node_size=100,node_color=’w’)

nx.draw_networkx_edges(my_graph,pos,edge_color=current_edge_colours,width=5.0)

nx.draw_networkx_labels(my_graph,pos)
```

Finally, once Network X has drawn all of the components to the figure it is shown to the user with:

```
plt.show()
```

Directions

Now that we have a representation of the tube it is time to think about generating directions between stations.

The user should be able to specify:

- A starting station
- A destination station

The **get_directions** method should then work out a route to from the starting station to the destination and **return** a list of directions showing:

- Which station and line to start from
- Any transfers between lines
- Where to get off

In order to complete this task you will need to be familiar with some additional Network X functions and Python methods - see the side bar for further details.

Exercise Five

a) Complete the **get_directions** method which will take the start and destination stations and return a list of written directions.

Finally we must complete the **display_full_map** and **display_travel_map** methods.

The **display_full_map** method will show the user a graph of the entire tube network (given by our file) and the **display_travel_map** method will show a map of the route between the starting and destination stations.

Network X provides a method to create a sub graph when it is provided with a set of nodes:

```
route_graph = my_graph.subgraph(shortest_path)
```

Exercise Six

a) Complete the **display_full_map** and **display_travel_map** methods.

Shortest Path

Network X provides a method to generate the shortest path between two given nodes:

```
nx.shortest_path(my_graph,
node1,node2)
```

This method will return a list of stations starting with node1 and ending with node2.

Combining Lists

Sometime you will want to be able to combine list together. For example you could have the following lists:

```
capitals = ['London',
Edinburgh']
```

```
countries =
["England","Scotland"]
```

We can create country **tuples** by making use of the **zip function**:

```
data =
zip(capitals,countries)
```

This will return a zip iterator which we can convert to a list of tuples:

```
data = list(data)
```

Printing the data out would result in:

```
[("London","England"),
("Edinburgh","Scotland")]
```