COSC 2P95 – Lab 8 – Section 01/02 – Graphs

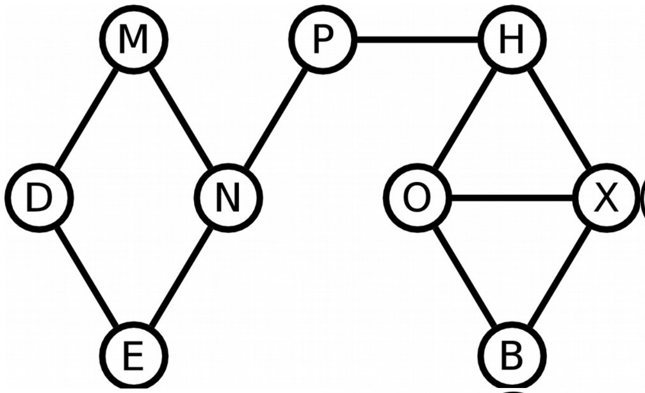
Thus far, our labs have been about learning new techniques in C++, demonstrated via hypothetically interesting topics. However, we haven't really done the opposite yet: starting with a task, and then writing a program to meet the requirements of that task.

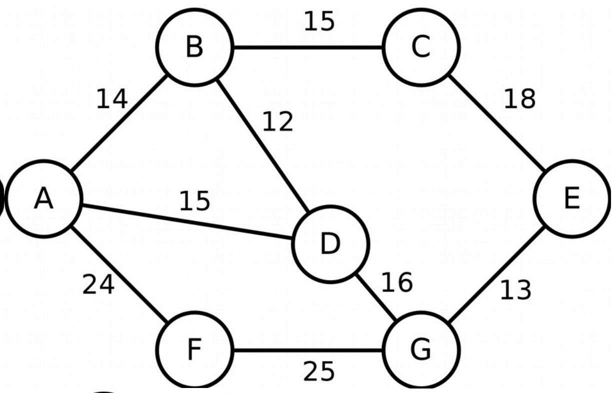
For this lab, we'll be learning to represent *graphs*, and apply a couple basic algorithms to the graph.

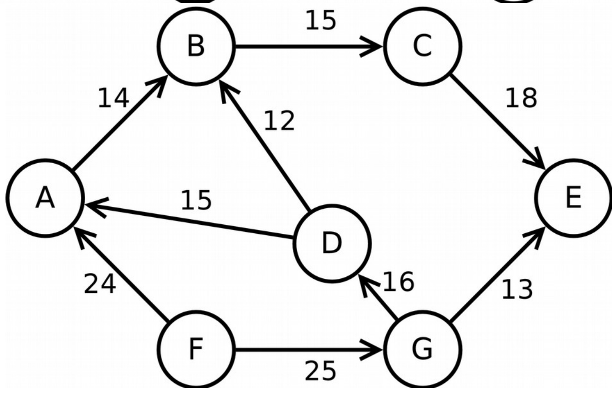
Graphs

In *graph theory*, a graph is a special collection of elements (*vertices*), where much of the structure is defined by which elements are connected to each other (via *edges*).

For simplicity, consider the following examples:







The first is just a simple *unweighted, undirected* (or bidirectional) graph.
The second is a weighted, but still undirected, graph.
The third is both weighted, and also directional.

e.g. for the first, M and N are connected by an edge (sometimes labeled as MN).
For the second, edge BC has a *weight* (sometimes a *cost*) of 15, and the edge may be treated as BC or CB.
For the third, BC has a weight of 15, but may only be treated as BC (i.e. CB is not included).

There are many algorithms we can perform on graphs, including minimal spanning trees (useful for broadcasting/multicasting on networks), and shortest-path algorithms (for vehicle routing), but one of the most basic features is traversal.

For this lab, we'll be starting at some defined vertex, and then listing those nodes reachable from the starting vertex, according to some traversal strategy. e.g. for the third graph above, F is only reachable if it's the start.

Graph Representation
There are many different ways to represent graphs; what's reasonable will depend largely on how one intends to use them. This will include multiple design decisions:

- Does each vertex need to have multiple pieces of data contained within it? If so, it might warrant using a class/struct for a Vertex
    - Examples include path cost and a reference to the parent node for path-finding, or even just a boolean for traversal (to determine which have already been processed)
    - One *could* also include a list of adjacent vertices, but this isn't commonly necessary
- Even when paths/costs are necessary, could one get away with simply using arrays?
    - Absolutely, yes. You can simply use multiple arrays to store parallel data:
        - One 2D array contains the adjacencies (neighbour vertices connected via edges)
        - One array could contain parent indices
        - One array could contain path costs
        - Simply use the same row index for each array, and you've matched up the components of a vertex with each other
            - Of course, that's terrible for encapsulation, but it's pretty fast (and easy)
- Are the edges themselves important?
    - Do we need to have an explicit entry for an edge, just because it has a weight?
    - Do we need to keep track of which vertices connect to a given edge?
    - Generally, we don't need to worry about this
- Do we need to represent a Graph as a completely separate class, complete with methods and proper information hiding, or can we just have raw data exposed to the code using it?
    - More generally, there's no harm in creating a Graph class. However, graphs aren't *usually* a sufficiently general data structure to warrant a custom graph library. You're writing a graph for a specific task; not for the sake of unique data representation
        - Short answer: more often than not, we can just access and manipulate the data directly, but if we encounter an exception, we've learned enough techniques to make it easy
        - For this lab, we should probably skip making a Graph class
- Do we care about efficient storage, including for cases like *sparse graphs*?
    - A sparse graph is one which contains several vertices, but relatively few connections (edges) between them; depending on the representation, that could mean quite a bit of wasted memory

Adjacency Matrix
Consider the first graph above. How could we list the connections?

	B0	D1	E2	H3	M4	N5	O6	P7	X8
B0							T		T
D1			T		T				
E2		T				T			
H3							T	T	T
M4		T				T			
N5			T		T			T	
O6	T			T					T
P7				T		T			
X8	T			T			T		

(T indicates an edge; Falses are omitted for readability, but are found in every blank cell)

In the matrix above, one would typically read it as *from* in the row indices on the left, and *to* in the column indices along the top (though it doesn't matter which is which in this case, since the matrix is symmetrical).

An alternate representation is an Adjacency List, with each *from* vertex containing a *linked list* to its neighbours. It's more efficient for sparse graphs (as the space used is proportional to the actual number of edges in the graph), but that level of memory efficiency isn't really useful here, so it's advisable to stick with the adjacency matrix.

One remaining question is how to easily handle the actual indexing itself.
The labels for each vertex could be consecutive numbers, letters, or arbitrary labels.
It's probably best to use a standard (e.g. 0-based) numbering for the vertices, and then have a separate look-up to match those numbers to their corresponding labels. (e.g. for the graph above, the E vertex would be treated as 2, and the “E” label could be looked up via that index)

For the sake of this lab, feel free to assume that no graph will ever exceed 20 vertices. In doing so, you can avoid needing to deal with the hassle of either allocating and using variable-length 2D arrays, or using a 1D array with a mapping function. (We've already had our lab on arrays, so let's keep it simple)

Note that the graph above is symmetrical, only because the graph is undirected. The third graph in the trio of diagrams on the first page would not be symmetrical (as edge XY doesn't imply edge YX).

Important: This version only allowed for unweighted graphs. If you want to support weighted graphs, then you need to store those weights for the edges.

- A popular option is to use an integer instead of a boolean; instead of storing true/false, store the weight
  - If there's no connection there, so long as you don't need negative edge weights (which *are* a thing), you can just enter -1
  - If negative edge weights also need to be supported, the easiest solution is to have a boolean matrix for the presence of edges, and an integer matrix to store all of the actual weights

Before we get to a task, we'll need to actually populate a graph.

Graph Data

The first data file (`undirecteddata.txt`) you've been provided has a pretty simple format[*]:

- A single integer, stating the number of vertices in the graph
  - As stated, you'll never need to deal with more than 20 vertices
- A single line of vertex labels, tab-delimited
  - For the sample file, there are 9 vertices, so there are 9 vertex letter labels
- A single integer, stating the number of edges in the graph
  - For undirected graphs, expect to see edges listed twice
    - e.g. if MN is an edge, then NM is *another* edge, to allow for directed graphs
- For each edge, a line containing two (tab-separated) values:
  - The *numeric* index of the *from* vertex
  - The *numeric* index of the *to* vertex
  - e.g. for BO above, you'll see a line with 0	6, and one with 6	0

[*]actually, it's artificially simple. Text-processing isn't part of this week's topic, so it's been kept simple. However, if you ever end up doing this 'for real' (e.g. for research, number-crunching, network-design, etc.), expect this to be at least slightly less palatable.

When loading the graph data file, you may assume that the file is entirely legal. This doesn't mean that all vertices will be reachable from every other vertex; only that you don't need to worry about typos.

Traversals

We *traverse* a graph when we wish to perform the same task on each vertex (or edge). Tasks could include complicated analysis, or simply printing. In this case, we're just looking to print those values accessible from some arbitrary starting point.

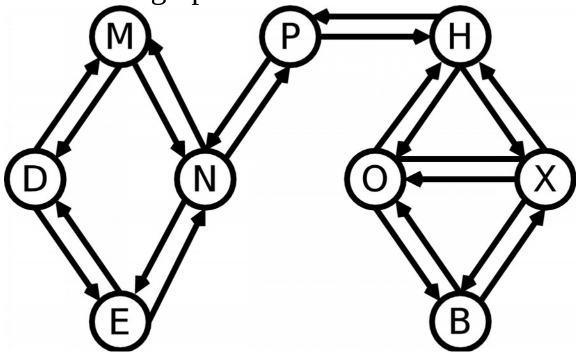
Strictly speaking, we actually won't necessarily be traversing the entire graph.
If an undirected graph is *connected* (the graph contains sufficient edges that every vertex is reachable from every other vertex), then a full traversal is possible; otherwise, you can only reach those vertices within the *connected component* that includes the starting vertex.
Similarly, a directed graph must be *strongly connected* (all vertices are reachable from all over vertices, through *some* route) to guarantee reaching all vertices.

However, for this task, we only care about printing those vertices reachable from (and including) the designated starting vertex, so the distinction is a moot point.

There are lots of potential traversals, but two obvious ones: *depth-first* traversal, and *breadth-first* traversal.

Depth-first: A depth-first traversal is exactly what it sounds like: from a given vertex, it tries proceeding as far as it can, processing/printing every vertex it encounters. Once that path has been exhausted, it returns to the most recent point where it had a choice of paths, and then tries to continue from there (and this is repeated until all other avenues have been exhausted).

For example, let's look at the first undirected graph:



In this case, since our program will be using *directed graphs*, it's been redrawn as the equivalent directed graph.
Suppose our starting point is E.

1. Before we do anything, we'll mark all vertices as not having been *visited*
2. We start at E, and process (in this case print) it
   - We should also mark E as having been *visited*
   - We have two neighbours: D and N
     - *It does not matter* which of those two we choose
     - We'll arbitrarily choose N
3. We're now at N, and should process it
   - We'll also mark it as *visited*
   - We have three neighbours, E, M and P
     - We will *not* consider E, as it is already *visited*
     - Let's choose P
4. We're at P, and process it
   - We mark it as *visited*
   - We have two neighbours, N and H
     - But N is already visited
     - We must choose H

5. We're at H; process and mark it visited
    - Of our three neighbours (P, O, and X), only O and X may be considered
    - We'll choose O
6. We're at O; process and mark it visited
    - Of H, X, and B, X and B are unvisited
    - We'll choose X
7. We're at X; process and mark it visited
    - We can't choose H (as it's been visted); we can't choose O (ditto); we can choose B
    - So we'll choose B
8. We're at B; process and mark it visited
    - We have no unvisited neighbours, so we're done with this path
7. We're back at X; we have no other unvisited neighbours, so we're done here
6. We're back at O; all of our neighbours have been visited now (B's taken care of now); we're done here
5. We're back at H; we have no unvisited neighbours; we're done here
4. We're at P; we have no unvisited neighbours; we're done here
3. We're at N; this is interesting, because M is still unvisited
    - So let's visit M
9. We're at M; process and mark it visited
    - N has already been visited, but D hasn't
    - So we choose D
10. We're at D; process and mark it visited
    - We have no unvisited neighbours, so we're done with this path
9. We're back at M; we have no more unvisited neighbours, so we're done here
3. We're back at N (again); now we have no unvisited neighbours, so we're done here
2. We're back at E; both of our neighbours have been visited, so we're done here
1. We're now right where we started, so we've traversed as much as we can from this starting point
    - That is, we're done!

The reason for marking vertices as *visited* is simple: this is a *cyclic graph*; we could easily go from D→M→N→E→D→... infinitely. By only considering unvisited neighbours, no vertex may be visited more than once.

Of course, if we were to repeat the algorithm (whether from the same starting point or a different one), we'd need to first reset all of the *visited* flags to false.

Depth-first implementation:
This algorithm can be implemented pretty easily. After resetting the visited flags as false, a high-level algorithm might go as follows:

```
DFT(v):
        process v
        mark v.visited as true
        for each neighbour, w, of v:
                if v is not visited:
                        DFT(w)
```

Aaand... that's it.
You might notice this is a *recursive algorithm*. If you aren't keen on recursion, you can alternatively do an iterative version pretty easily using a stack, but that's up toyou.

Important Note; no, seriously, read this:

Your submission task for this lab is simply to write a program that can:

- Ask the user for a filename (or accept the filename as a command-line parameter, per your preference)
- Load the file into a graph
    - Again, you're allowed to assume no larger than a 20-vertex graph
    - As hinted at in the file description above, we're using only unweighted graphs
        - i.e. we're only listing which vertices directly connect to which
- The user may then choose a starting vertex
    - At which point, your program will perform a depth-first traversal as described above
- The user may then either display another traversal, or choose to quit (e.g. `-1`)
- When the user is choosing the starting vertex, it's up to you how they specify it (e.g. in the `undirecteddata.txt` sample, the user could specify B as the starting point either by entering `B`, or by entering `0`, per your own preference)

Here's why it's important to read: this task (the depth-first traversal) is the submission task. This is what you need to do to complete the lab. However, as a graph example, we really should know more about graphs, so we'll at least discuss one more traversal, for the sake of learning.
*But this is as far as you are **required** to go.*

Breadth-first: A common alternative to depth-first searches (and traversals) is breadth-first.
To do a *breadth-first* search or traversal means that, after the initial starting point, you first explore *all* neighbours at a distance of **1** from that point. You then consider all neighbours at a distance of **2**. Then **3**, then **4**, etc.
You don't necessarily need to keep track of which depth you're currently on (though this is easy to do by simply adding an extra depth/cost property to each vertex).
Fun fact: If you do add that depth-tracking, and also keep track of the vertex that got you to each new vertex you're visiting, then you've actually implemented an algorithm for finding the shortest-paths from a starting vertex. For things like orthogonal maps (i.e. the simple puzzle), this is an easy way to solve them.

That said, there's a very easy way to do a breadth-first traversal without needing to actually measure distances: simply use a queue (which... we're pretty comfortable writing/using by now, right?).

Breadth-first algorithm:
Because this isn't recursive, we can initialize right inside the algorithm.

```
BFT(v):
    Reset all vertices to not visited
    Set vertex v to visited
    Process/print v
    Place v onto an empty queue
    While the queue is not empty:
        Remove the first vertex, w, from the queue
        For each unvisited neighbour, u, of w:
            Set u to visited
            Process/print u
            Add u to the queue
```

Of course, there are lots of small variations. For example, you can process/print when you add it to the queue, or when you take it off the queue (so long as you're consistent).

Feel free to try implementing this. It's not required for this lab, but it's good practice.

Submission Task
As indicated above, you need to allow for depth-first traversals, to print the values within the graph.
The user needs to be able to choose the data file.
Remember that, depending on the graph, not all vertices may be reachable. You've also been given an *acyclic directed graph* (equivalent to the third graph shown on the first page, except unweighted), for which reaching all vertices will *only* be possible if you happen to start on F.

Submission
When you're finished, simply demonstrate your sample program.
You'll be using the `directeddata.txt` sample file for your lab demonstrator, who will verify that the number of reachable vertices will depend on the starting point.

Two such possible executions are shown below:

(Undirected Example)

```
Graph filename: undirecteddata.txt
Using undirecteddata.txt
File loaded.
Loaded graph.

Vertices:
[0:B], [1:D], [2:E], [3:H], [4:M], [5:N], [6:O], [7:P], [8:X]

Edges:
B -> O,X
D -> E,M
E -> D,N
H -> O,P,X
M -> D,N
N -> E,M,P
O -> B,H,X
P -> H,N
X -> B,H,O
Starting vertex number for DFT: 2
 E D M N P H O B X

Starting vertex number for DFT: 5
 N E D M P H O B X

Starting vertex number for DFT: 0
 B O H P N E D M X

Starting vertex number for DFT: -1
```

(Directed Example)

```
Graph filename: directeddata.txt
Using directeddata.txt
File loaded.
Loaded graph.

Vertices:
[0:A], [1:B], [2:C], [3:D], [4:E], [5:F], [6:G]

Edges:
A -> B
B -> C
C -> E
D -> A,B
E ->
F -> A,G
G -> D,E
Starting vertex number for DFT: 0
 A B C E

Starting vertex number for DFT: 4
 E

Starting vertex number for DFT: 5
 F A B C E G D

Starting vertex number for DFT: -1
```