Module 10 : GRAPHS

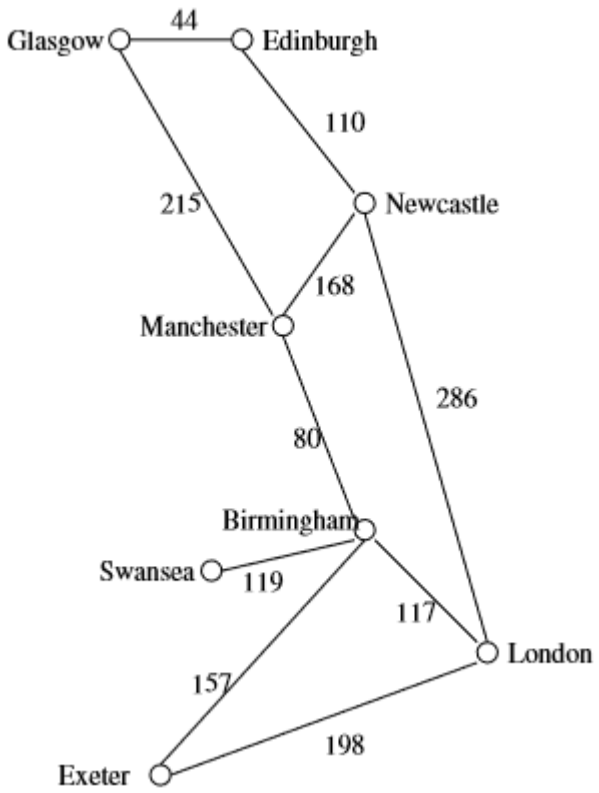
Course Learning Outcomes:

1. Familiarize graph Terminologies.
2. Learn on how to Traverse graph.
3. Know how to use and implement Graphs.

GRAPHS

Is a pictorial way of representing relationships between various quantities, parameters, or measurable variables in nature. A **graph** basically summarizes how one quantity changes if another quantity that is related to it also changes.

Often it is useful to represent information in a more general graphical form than considered so far, such as the following representation of the distances between towns:



Course Module

With similar structures (maybe leaving out the distances, or replacing them by something else), we could represent many other situations, like an underground tunnel network, or a network of pipes (where the number label might give the pipe diameters), or a railway map, or an indication of which cities are linked by flights, or ferries, or political alliances. Even if we assume it is a network of paths or roads, the numbers do not necessarily have to represent distances, they might be an indication of how long it takes to cover the distance in question on foot, so a given distance up a steep hill would take longer than on even ground.

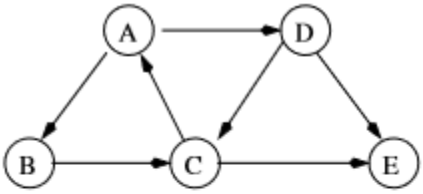
There is much more that can be done with such a picture of a situation than just reading off which place is directly connected with another place: For example, we can ask ourselves whether there is a way of getting from A to B at all, or what is the shortest path, or what would be the shortest set of pipes connecting all the locations. There is also the famous Travelling Salesman Problem which involves finding the shortest route through the structure that visits each city precisely once.

GRAPH TERMINOLOGY

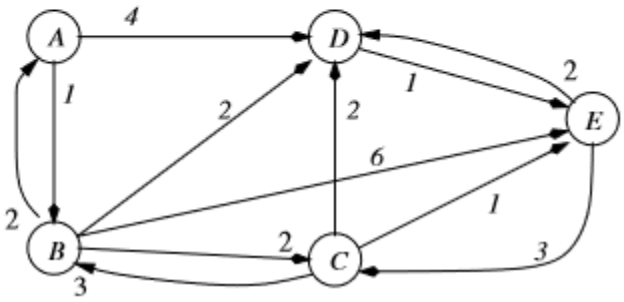
The kind of structure in the above figure is known formally as a ***graph.*** A graph consists of a series of nodes (also called vertices or points), displayed as nodes, and edges (also called lines, links or, in directed graphs, arcs), displayed as connections between the nodes. There exists quite a lot of terminology that allows us to specify graphs precisely:

A graph is said to be simple if it has no ***self-loops*** (i.e., edges connected at both ends to the same vertex) and no more than one edge connecting any pair of vertices. The remainder of this Chapter will assume that, which is sufficient for most practical applications.

If there are labels on the edges (usually non-negative real numbers), we say that the graph is weighted. We distinguish between directed and undirected graphs. In directed graphs (also called digraphs), each edge comes with one or two directions, which are usually indicated by arrows. Think of them as representing roads, where some roads may be one-way only. Or think of the associated numbers as applying to travel in one way only: such as going up a hill which takes longer than coming down. An example of an unweighted digraph is:



and an example of a *weighted* digraph, because it has labels on its edges, is:



In undirected graphs, we assume that every edge can be viewed as going both ways, that is, an edge between A and B goes from A to B as well as from B to A. The first graph given at the beginning of this chapter is weighted and undirected.

A path is a sequence of nodes or vertices v1, v2, ..., vn such that vi and vi+1 are connected by an edge for all $1 \leq i \leq n-1$. Note that in a directed graph, the edge from vi to vi+1 is the one which has the corresponding direction. A circle is a non-empty path whose first vertex is the same as its last vertex. A path is simple if no vertex appears on it twice (with the exception of a circle, where the first and last vertex may be the same – this is because we have to 'cut open' the circle at some point to get a path, so this is inevitable).

An undirected graph is connected if every pair of vertices has a path connecting them. For directed graphs, the notion of connectedness has two distinct versions: We say that a digraph is weakly connected if for every two vertices A and B there is either a path from A to B or a path from B to A. We say it is strongly connected if there are paths leading both ways. So, in a weakly connected digraph, there may be two vertices i and j such that there exists no path from i to j.

A graph clearly has many properties similar to a tree. In fact, any tree can be viewed as a simple graph of a particular kind, namely one that is connected and contains no circles. Because a graph, unlike a tree, does not come with a natural 'starting point' from which there is a unique path to each vertex, it does not make sense to speak of parents and children in a graph. Instead, if two vertices A and B are connected by an edge e, we say that they are neighbours, and the edge connecting them is said to be ***adjacent.***

IMPLEMENTING GRAPHS

All the data structures we have considered so far were designed to hold certain information, and we wanted to perform certain actions on them which mostly centred around inserting new items, deleting particular items, searching for particular items, and sorting the collection. At no time was there ever a connection between all the items represented, apart from the order in which their keys appeared. Moreover, that connection was never something that was inherent in the structure and that we therefore tried to represent somehow – it was just a property that we used to store the items in a way which made sorting and searching quicker. Now, on the other hand, it is the connections that are the crucial information we need to encode in the data structure. We are given a structure which comes with specified connections, and we need to design an implementation that efficiently keeps track of them.

Array-based implementation. The first underlying idea for *array*-based implementations is that we can conveniently rename the vertices of the graph so that they are labelled by non-negative integer indices, say from 0 to $n-1$, if they do not have these labels already. However, this only works if the graph is given *explicitly*, that is, if we know in advance how many vertices there will be, and which pairs will have edges between them. Then we only need to keep track of which vertex has an edge to which other vertex, and, for weighted graphs, what the weights on the edges are. For unweighted graphs, we can do this quite easily in an $n \times n$ two-dimensional binary array `adj`, also called a *matrix*, the so-called *adjacency matrix*. In the case of weighted graphs, we instead have an $n \times n$ *weight matrix* `weights`. The array/matrix representations for the two example graphs shown above are then:

		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	0	1	0
B	1	0	0	1	0	0
C	2	1	0	0	0	1
D	3	0	0	1	0	1
E	4	0	0	0	0	0

		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	∞	4	∞
B	1	2	0	2	2	6
C	2	∞	3	0	2	1
D	3	∞	∞	∞	0	1
E	4	∞	∞	3	2	0

In the first case, for the unweighted graph, a '0' in position `adj[i][j]` reads as `false`, that is, there is no edge from the vertex i to the vertex j. A '1', on the other hand, reads as `true`, indicating that there is an edge. It is often useful to use boolean values here, rather than the numbers 0 and 1, because it allows us to carry out operations on the booleans. In the second case, we have a weighted graph, and we have the real-valued weights in the matrix instead, using the infinity symbol ∞ to indicate when there is no edge.

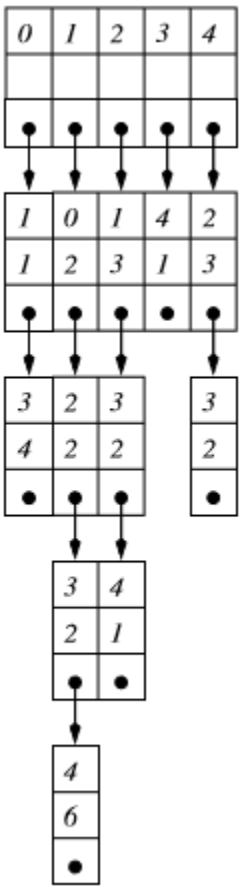
For an undirected graph, if there is a 1 in the ith column and the jth row, we know that there is an edge from vertex i to the vertex with the number j, which means there is also an edge from vertex j to vertex i. This means that `adj[i][j] == adj[j][i]` will hold for all i and j from 0 to $n-1$, so there is some redundant information here. We say that such a matrix is *symmetric* – it equals its mirror image along the main diagonal.

Mixed implementation. There is a potential problem with the adjacency/weight matrix representation: If the graph has very many vertices, the associated array will be extremely large (e.g., 10,000 entries are needed if the graph has just 100 vertices). Then, if the graph is *sparse* (i.e., has relatively few edges), the adjacency matrix contains many 0s and only a few 1s, and it is a waste of space to reserve so much memory for so little information.

A solution to this problem is to number all the vertices as before, but, rather than using a two-dimensional array, use a one-dimensional array that points to a linked list of neighbours for each vertex. For example, the above weighted graph can be represented as follows, with each triple consisting of a vertex name, connection weight and pointer to the next triple:

Mixed implementation. There is a potential problem with the adjacency/weight matrix representation: If the graph has very many vertices, the associated array will be extremely large (e.g., 10,000 entries are needed if the graph has just 100 vertices). Then, if the graph is *sparse* (i.e., has relatively few edges), the adjacency matrix contains many 0s and only a few 1s, and it is a waste of space to reserve so much memory for so little information.

A solution to this problem is to number all the vertices as before, but, rather than using a two-dimensional array, use a one-dimensional array that points to a linked list of neighbours for each vertex. For example, the above weighted graph can be represented as follows, with each triple consisting of a vertex name, connection weight and pointer to the next triple:



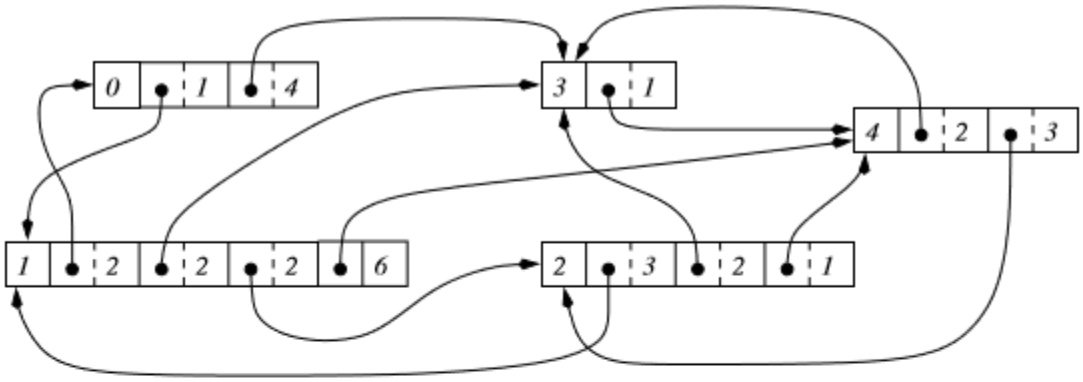
If there are very few edges, we will have very short lists at each entry of the array, thus saving space over the adjacency/weight matrix representation. This implementation is using so-called *adjacency lists*. Note that if we are considering undirected graphs, there is still a certain amount of redundancy in this representation, since every edge is represented twice, once in each list corresponding to the two vertices it connects. In *Java*, this could be accomplished with something like:

```
class Graph {
   Vertex[] heads;
   private class Vertex {
      int name;
      double weight;
      Vertex next;
      ...//methods for vertices
   }
   ...//methods for graphs
}
```

Pointer-based implementation. The standard *pointer*-based implementation of binary trees, which is essentially a generalization of linked lists, can be generalized for graphs. In a language such as Java, a class `Graph` might have the following as an internal class:

```
class Vertex {
   string name;
   Vertex[] neighbours;
   double[] weights;
}
```

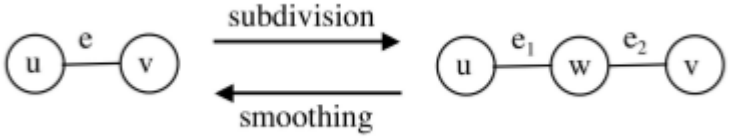
When each vertex is created, an array `neighbours` big enough to accommodate (pointers to) all its neighbours is allocated, with (for weighted graphs) an equal sized array `weights` to accommodate the associated weights. We then place the neighbours of each vertex into those arrays in some arbitrary order. Any entries in the `neighbours` array that are not needed will hold a `null` pointer as usual. For example, the above weighted graph would be represented as follows, with each weight shown following the associated pointer:



11.3 Relations between graphs

Many important theorems about graphs rely on formal definitions of the relations between them, so we now define the main relevant concepts. Two graphs are said to be *isomorphic* if they contain the same number of vertices with the same pattern of adjacency, i.e. there is a *bijection* between their vertices which preserves the adjacency relations. A *subgraph* of a graph G is defined as any graph that has a vertex set which is a subset of that of G, with adjacency relations which are a subset of those of G. Conversely, a *supergraph* of a graph G is defined as any graph which has G as a subgraph. Finally, a graph G is said to *contain* another graph H if there exists a subgraph of G that is either H or isomorphic to H.

A *subdivision* of an edge e with endpoints u and v is simply the pair of edges e_1, with endpoints u and w, and e_2, with endpoints w and v, for some new vertex w. The reverse operation of *smoothing* removes a vertex w with exactly two edges e_1 and e_2, leaving an edge e connecting the two adjacent vertices u and v:



A *subdivision* of a graph G can be defined as a graph resulting from the subdivision of edges in G. Two graphs G and H can then be defined as being *homeomorphic* if there is a graph isomorphism from some subdivision of G to some subdivision of H.

An *edge contraction* removes an edge from a graph and merges the two vertices previously connected by it. This can lead to multiple edges between a pair of vertices, or *self-loops* connecting a vertex to itself. These are not allowed in *simple graphs*, in which case some edges may need to be deleted. Then an undirected graph H is said to be a *minor* of another undirected graph G if a graph isomorphic to H can be obtained from G by contracting some edges, deleting some edges, and deleting some isolated vertices.

TRAVERSALS – SYSTEMATICALLY VISITING ALL VERTICES

In order to *traverse* a graph, i.e. systematically visit all its vertices, we clearly need a strategy for exploring graphs which guarantees that we do not miss any edges or vertices. Because, unlike trees, graphs do not have a root vertex, there is no natural place to start a traversal, and therefore we assume that we are given, or randomly pick, a starting vertex i. There are two strategies for performing graph traversal.

The first is known as *breadth first traversal*: We start with the given vertex i. Then we visit its neighbours one by one (which must be possible no matter which implementation we use), placing them in an initially empty *queue*. We then remove the first vertex from the queue and one by one put its neighbours at the end of the queue. We then visit the next vertex in the queue and again put its neighbours at the end of the queue. We do this until the queue is empty.

However, there is no reason why this basic algorithm should ever terminate. If there is a circle in the graph, like A, B, C in the first unweighted graph above, we would revisit a vertex *we have already visited*, and thus we would run into an infinite loop (visiting A's neighbours puts B onto the queue, visiting that (eventually) gives us C, and once we reach C in the queue, we get A again). To avoid this we create a second array `done` of booleans, where `done[j]` is `true` if we have already visited the vertex with number j, and it is `false` otherwise. In the above algorithm, we only add a vertex j to the queue if `done[j]` is `false`. Then we mark it as done by setting `done[j] = true`. This way, we will not visit any vertex more than once, and for a finite graph, our algorithm is bound to terminate. In the example we are discussing, breadth first search starting at A might yield: A, B, D, C, E.

To see why this is called breadth first search, we can imagine a tree being built up in this way, where the starting vertex is the root, and the children of each vertex are its neighbours (that haven't already been visited). We would then first follow all the edges emanating from the root, leading to all the vertices on level 1, then find all the vertices on the level below, and so on, until we find all the vertices on the 'lowest' level.

The second traversal strategy is known as *depth first traversal*: Given a vertex i to start from, we now put it on a *stack* rather than a queue (recall that in a stack, the next item to be removed at any time is the last one that was put on the stack). Then we take it from the stack, mark it as done as for breadth first traversal, look up its neighbours one after the other, and put them onto the stack. We then repeatedly pop the next vertex from the stack, mark it as done, and put its neighbours on the stack, provided they have not been marked as done, just as we did for breadth first traversal. For the example discussed above, we might (starting from A) get: A, B, C, E, D. Again, we can see why this is called depth first by

formulating the traversal as a search tree and looking at the order in which the items are added and processed.

Note that with both breadth first and depth first, the order of the vertices depends on the implementation. There is no reason why A's neighbour B should be visited before D in the example. So it is better to speak of *a* result of depth first or breadth first traversal, rather than of *the* result. Note also that the only vertices that will be listed are those in the same *connected component* as A. If we have to ensure that all vertices are visited, we may need to start the traversal process with a number of different starting vertices, each time choosing one that has not been marked as done when the previous traversal terminated.

Exercises: Write algorithms, in pseudocode, to (1) visit *all* nodes of a graph, and (2) decide whether a given graph is connected or not. For (2) you will actually need two algorithms, one for the strong notion of connectedness, and another for the weak notion.

References and Supplementary Materials

Books and Journals

1. John Bullinaria ; March 2019 ; Data Structures and Algorithms ; Birmingham, UK;
2. Michael T Goodrich ; Roberto Tamassia ; David Mount ; Second Edition ; Data Structures and Algorithm in C++
3. www.google.com
4. Merriam Webster Dictionary