ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

G r a p h s

Goodrich et al. chapter 12

Koffman chapter 12

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

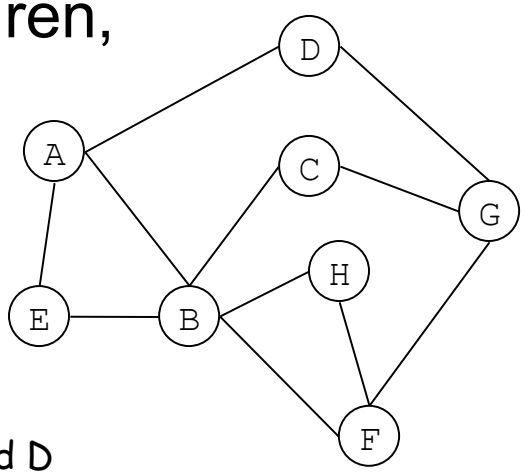
The Graph Data-Structure

- Graphs show objects (nodes), and the connections between them
- They enable us to solve problems such as:
  - Finding routes between cities: the objects could be towns, and the connections could be road/rail links.
  - Planning a project: the objects are tasks, and connections show relationships between tasks.
  - Deciding on a move in a game: the objects are board states, and the connections correspond to possible moves.
- Graph algorithms allow us, for example
  - *search* a graph for a path between two given nodes
  - find the *shortest path* between two nodes
  - *order* the vertices in the graph is a particular way.
- These very general algorithms can be used to solve problems of the kind mentioned above
- For example, search algorithms can be used find a possible winning move in a game; shortest path algorithms can be used to find the shortest route between two cities

See alison cawleys course: http://www.macs.hw.ac.uk/~alison/ds98/node1.html

What is a graph structure?

- A collection of nodes, some of which are connected.
- A graph is something like a tree, where every node can have many parents as well as many children.
  - So it is a more general inter-connected set of nodes
- The nodes of a graph are often called vertices, and the connections between them are referred to as edges of the graph.
- Rather than talk about parents and children, we refer to the 'neighbours' of a node or its 'adjacent nodes'
  - all the nodes which are connected to it by an edge of the graph



neighbours of A are E, B and D
neighbours of B are A, E, H, F and C

Some Features of a graph

- Weighted or not weighted
  - A weighted graph is one where each edge has an associated weight.
  - For example, it may represent distance or time where the vertices represent different locations.
- Directed or undirected
  - A directed graph is one where each of the edges has a direction associated with it
  - It could represent a one way road between 2 locations.
- Connected or not connected
  - A connected graph is one where a path exists between any 2 vertices which can be chosen.
  - In the case of a directed graph, it is strongly connected if a path exists *in each direction* between any 2 vertices which can be chosen.

A Graph ADT

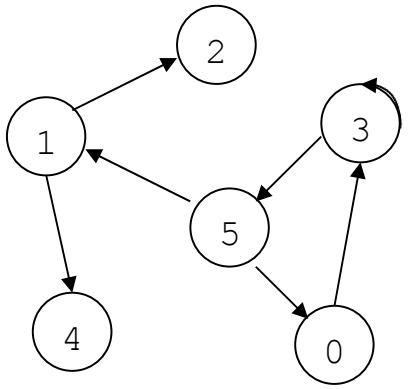
- An abstract data type to represent a graph should provide operations for:
- constructing a graph (adding and removing edges and vertices)
- checking connections in a graph.
- If we allow labelled nodes then there will be additional operations to add, access and remove labels.
- If the graph is weighted, then it must be possible to add a weight to an edge.
- Assuming that we have suitable type declarations for label and graph, the following is a reasonable minimal set of operations, where nodes are just indicated by integers.

```
Graph(int n) // Creates and initialises a graph of given size
void AddEdge(int n1, int n2, Weight w) // Adds edge from n1 to n2
                                          of Weight w
void RemoveEdge(int n1, int n2) //Removes an edge between n1 and n2
bool EdgeExists(int n1, int n2)
Weight getWeight(int n1, int n2)
void setWeight(Weight w, int n1, int n2);
void SetLabel(GraphLabel l, int n) // Adds a label l to node n
GraphLabel GetLabel(int n) // Returns the label of a node
```

- We might also want to add a search of the graph, and a traversal of all the nodes in the graph.

Graph Representation: Adjacency Matrix

- An Adjacency Matrix is one of the 2 main ways of constructing a graph data structure :
- It consists of a (possibly sparse) 2-D array where each row and each column corresponds to one of the nodes
- Here is an example of a directed graph and its adjacency matrix. The convention followed here is that an adjacent edge counts 1 in array index position [1, 2] in the array matrix for the directed graph if there is an edge from 1 to 2. (X,Y coordinates are 0-5)



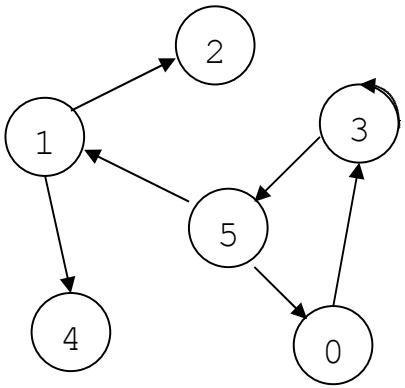
	0	1	2	3	4	5
0				1		
1			1		1	
2						
3				1		1
4						
5	1	1				

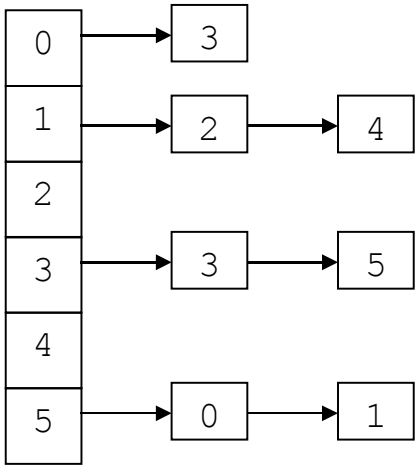
If the graph were undirected, then it would be symmetric: if node 1 were connected to node 4, then node 4 would also be connected to node 1

If it were weighted, then the entries in the matrix would show the weight

Graph Representation: Edge List

- An edge list representation will list, for each node, all the nodes which are connected to it via an edge.
- This could be implemented as a 1 dimensional array, where each entry corresponds to a node, and holds a linked list of the connected nodes.





If the graph were weighted, then each linked list element would contain the node of the graph and the weight of the connection to it.

Graph Representation: which to choose

- If the graph is ‘sparse’ (i.e a lot of nodes with not that many edges) then the adjacency matrix will also be sparse, and not space-efficient.
- If the graph is quite dense, and an edge list is used, then the linked lists at each node will be long, and a search of it may have significant overhead.
- Certain operations, such as `EdgeExists` are likely to be much more efficient using adjacency matrices
- So final choice will depend on the likely use of the graph, and the constraints on the system.

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Searching a graph

- In a *breadth first* search or traversal of a graph starting at a particular node (say, n1), then all nodes reached by 1 edge from n1 are visited, followed by all nodes reached by 2 edges etc.
- In a *depth first* search or traversal, when a node reached by 1 edge from n1 is visited, then all nodes connected to that node are visited before returning to the next node connected to n1 by one edge.
- Thinking of this as a family tree:
  - breadth first search looks at all the siblings before looking at any of their descendants.
  - Depth first search looks at all descendants of a sibling before considering the next sibling.
- For any traversal of the graph, it will be necessary to mark a node as visited so that it is not visited again.
  - If the graph has ‘cycles’ (paths through the graph which return to the starting node) we would otherwise have an infinite loop.
  - Even without cycles, there could be a lot of redundant visits to the same node.

Example of an application of graphs: Garbage Collection

Introduction

- Languages such as C++ mandate that the programmer is responsible for the memory allocation and de-allocation in a program. In the hands of a weak programmer, this can be a source of errors.
- The Java language does not give this responsibility to the programmer, instead a garbage collection mechanism is used.
- At intervals, all objects allocated from the memory heap will be checked to see if they are still 'live' (reachable from other live objects in the program). Such objects must NOT be de-allocated, but all others will be.

Determining live objects

- All the local variables of a program will be stored on the run-time stack, and these are called root objects
- All objects on the memory heap which are referenced by the root objects (ie data members of the root objects) are also live objects
- All objects referenced by these additional live objects are also live, and so on …
- So all 'live' objects are connected as if in a graph – starting with the objects which are local variables, then connecting each with every other object that it references, and each of those with …

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Garbage collection cont: the mark-sweep algorithm

- There are many different algorithms for garbage collection, one of the most common is the **'mark-sweep algorithm'.**
- each object will have an associated mark-bit which we need to set to true if the object is live, and false otherwise.
- We start with the local variables, and traverse the graph of associated (and therefore live) objects

  - **Mark** all objects on the memory heap as non-live to start:
  - **Sweep** through the live objects by doing a depth first search from each of the objects which are local variables, and set the mark-bit to true for each object 'visited' in the search.
  - **Delete** all those objects on the heap which have not had their mark-bit set to true: they cannot be reached from any variable in the program.

Shortest path algorithms

- The need to find the shortest path through a graph is a common problem
  - E.g. In route finding software
- There are many shortest path algorithms.
- Dijkstra's algorithm is the most well known
- Dijkstra's is a form of 'greedy' algorithm
  - In general, greedy algorithms always choose the best possible immediate, or local, solution while finding an answer, and live with the consequences!
  - Greedy algorithms find the overall optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems.
  - Recall that the huffman algorithm was also a greedy algorithm
- In Dijkstra algorithm, the greedy step is to say "take the end node on the shortest path I've found from the beginning to *anywhere* so far, and then look next at all paths that go *via* this node"

ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Dijkstra's algorithm to find a shortest path through a graph from a to b

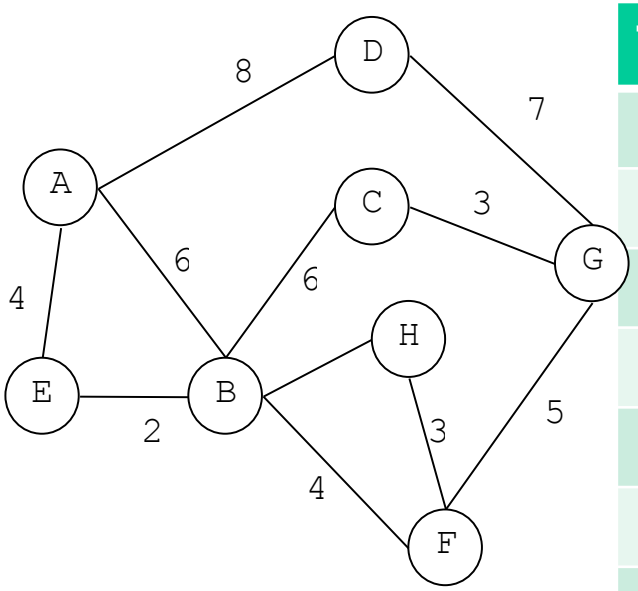
- For each node in the graph, store the shortest distance we have found *so far* from the source a to this node. And the route taken to achieve this.
- Let V be the set of all vertices to which we know the shortest *possible* path from the source a.
- Compute the distance:
  - from the source a to each node adjacent to a
  - For each node n in V, from the source a *via n* to all nodes adjacent to n.
- The shortest distance computed (to a node not already in V), is the shortest path that can be found from the source to this node, so add it to the set V.
- The distances computed to each other node (for which we do not already know the shortest possible path) should be compared with the shortest distance we hold so far to that node, and that distance and route updated if necessary.
- At each iteration we only have to 'visit' the latest node to have been added to V, as the distances via all the other nodes in V have already been calculated.
- We can start the ball rolling by saying distance from a to itself is 0, and all other distances are infinite, So a goes into the set V, and shortest-so-far distance to each other node can be set to INT_MAX.

Summary of steps when 'visiting' node n

- At any stage we will hold a set of nodes for which we have a '*shortest-so-far*' distance from a, and a set of nodes for which we have a '*shortest-possible*' distance from a.
  - We must also store the route associated with this distance.
- if n is the latest node to which we have found the 'shortest-possible' distance from the source a
- 'visit' node n as follows:
  - Find the distance from a *via n* to each node which is adjacent to n

    This will be found by adding the shortest distance from a to n and the distance from n to each adjacent node.
  - if we have found distance from a to any node for which which improves its 'shortest-so-far' distance, update.
  - Mark node n as visited: it will never need to be visited again.
- Now take the node m with the shortest distance in our set of 'shortest so far' distances to each node.
- This distance will definitely represent the shortest possible path from a to m, so we add m to the set V and continue by visiting node m

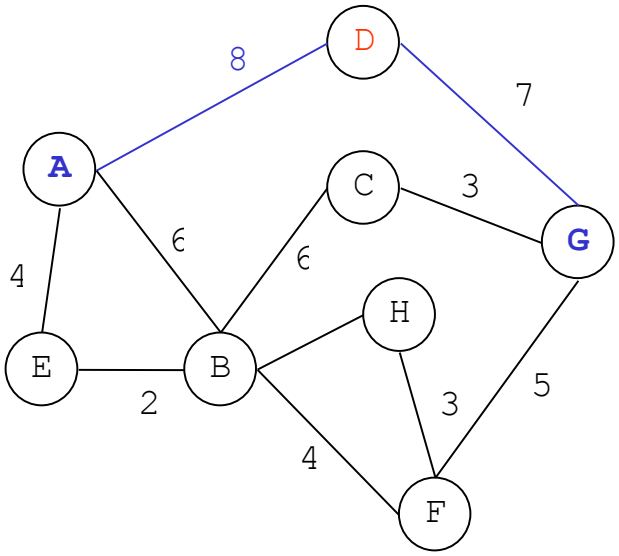
- Exercise: step through this algorithm with the graph shown to find the shortest distance from D to H.



To:	A	B	C	D	E	F	G	H
From D:	8	∞	∞	0	∞	∞	7	∞
From G:	8	∞	10	0	∞	12	7	∞
From A:	8	14	10	0	12	12	7	∞
From C:	8	14	10	0	12	12	7	∞
From E:	8	14	10	0	12	12	7	∞
From F:	8	14	10	0	12	12	7	15
From B:	8	14	10	0	12	12	7	15

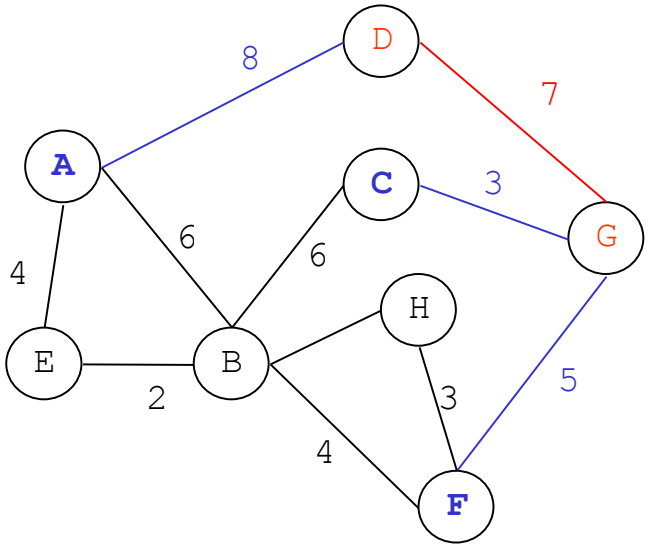
- This table shows, as each node is visited, the shortest-so-far distance from a to each node
- Coloured blue whenever the shortest-so-far is updated
- Coloured red when it is determined to be the 'shortest-possible' distance (will then be the next node visited)
- Shortest distance from D to H is 15: path is D->G->F->H or D->A->B->H or D->A->E->B->H

More graphics to explain the steps of Dijkstra's algorithm: finding shortest path from D->H



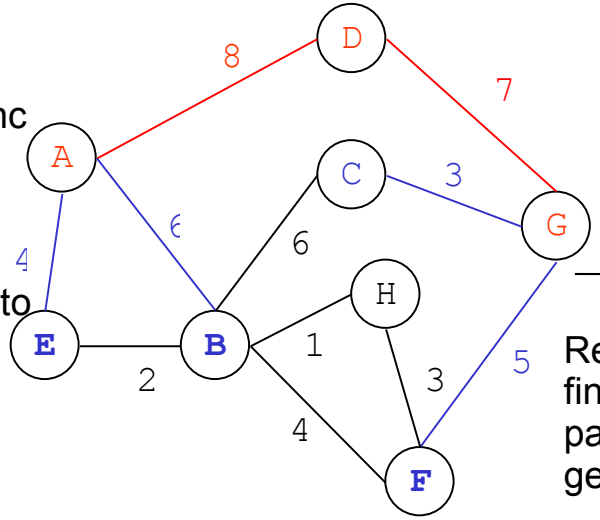
Start by finding 'shortest paths so far' from D to adjacent nodes (colour in blue)

The shortest of these must be the shortest path we can ever find to its endpoint, colour it red, and 'visit' its end-point

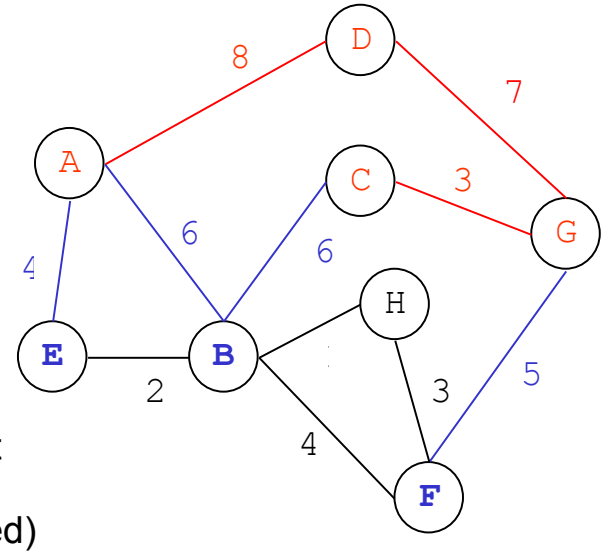


Check which is the 'shortest paths so far' from D to any other node not yet visited (inc paths that go via G)

The shortest of these must be the shortest path we can ever find to its endpoint, colour it red, and 'visit' its end-point



Repeat until we find the shortest path to H (i.e. H gets coloured red)



… could visit either E or F next, there are routes of length 12 to each of them

Implementation tips for Dijkstra's algorithm: Possible data structures to use

Can see many ways to implement the algorithm, here are some suggested data structures that might be useful in its implementation.

- We will need to keep track of which nodes have already been visited
  - This can be done by developing a node struct or class, and including a data member which is a 'visited' flag.
  - Or using C++, in a non-object solution, we could just throw all nodes which have been visited into a set (STL), and check if a node has been visited by querying the set.
- We might also store 'routes from a' in a priority queue. Each route will store information about
  - The end-node on this route
  - The path from a to the end-node via this route (e.g as a queue or linked list of nodes)
  - The distance from a via this route
- the highest priority route will be the shortest one stored
  - As each route is removed from the PQ, its end node is visited (if it has not been visited already)
  - If the route to any node adjacent to the end node on this route gives us a shorter path to the next node, then update its shortest path on the map, and add the route to it to our priority queue.
  - Nodes may appear more than once with different routes to them, but if the route is not the shortest to this node, the node will already have been visited before this route is removed from the PQ

More possible implementation structures

- If we have a node class, maybe a map which stores the 'label' of a node as the key, and the node itself as the value.
  - This would enable us to move quickly from the label to the node
- Or identify each node by a number, and store the node in an array with the number as index position.
  - This might be useful if we store the graph as an adjacency matrix – the numbers will identify the index position within the matrix
- The graph itself should really be implemented as a class
  - Data members which stores the graph and its size
  - Methods to add nodes and edges etc which will update the graph (as discussed earlier)
  - Shortest path algorithm would be a method of the class