Graphs and Graph Algorithms

In this part of the course unit, we shall

- learn about one of the most widely applicable structures in computing: graphs,
- consider how to use them in applications,
- understand how to represent graphs in programming languages, and
- consider some of the wide variety of algorithms that are available for computing with graphs.

This is an introduction to the material in Part II (Chapters 6, 7 and 8) of the course textbook .

Graphs

Graphs consist of a set of nodes and a set of edges which link some of the nodes.

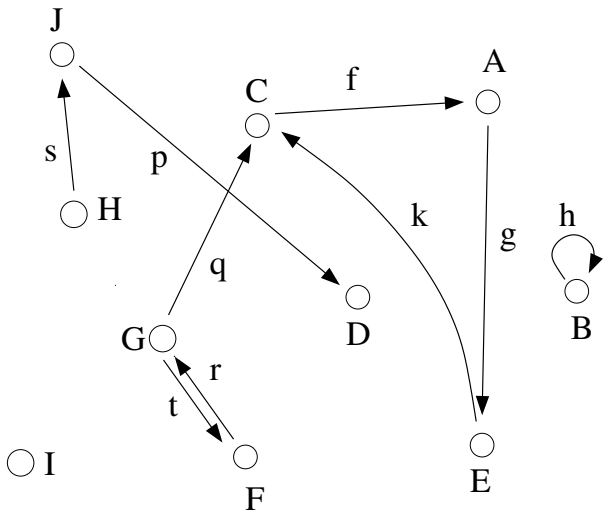


Figure : A directed graph (digraph).

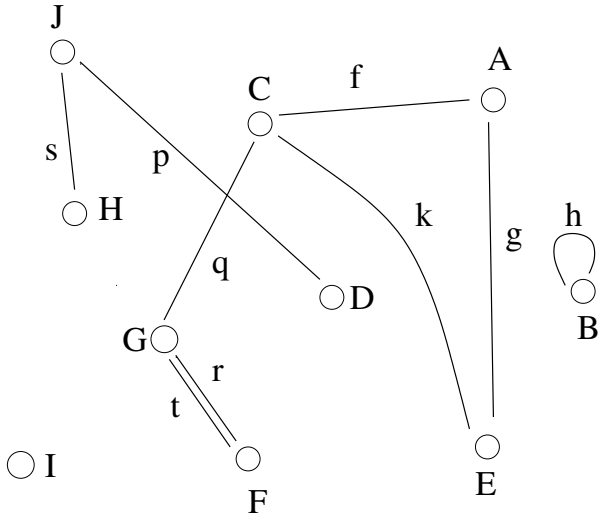


Figure : An undirected graph (a multigraph - more than one edge possible between pairs of nodes).

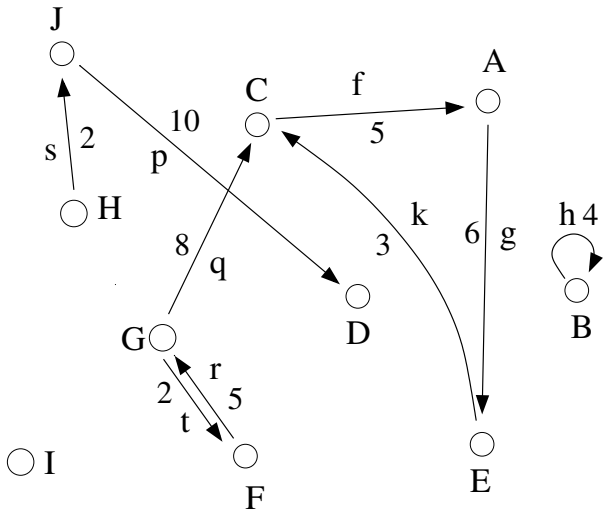


Figure : An edge-weighted directed graph.

Applications

Application	Nodes	Edges	Type of graph
Wired Networks	Computers	Cables	Undirected multigraph
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....

Terminology

Nodes - sometimes called vertices, points, etc.
Edges - sometimes called arcs, lines, etc.

Two nodes linked by an edge are adjacent or, simply linked.

For directed graphs, an edge e from node A to node B, is said to have A as its source node (or start or origin node) and B as its target node (or end or finish or destination node).

An edge is incident on a node if it has the node as source or target (directed) or if it links the node to another (undirected).

For an undirected graphs, the number of edges incident at a node is the degree of the node. For directed graphs, the number of edges whose source is a node, is the out-degree of the node, likewise targets and in-degree .

More terminology: Paths

A path is a sequence (possibly empty) of adjacent nodes and the linking edges, in the case of undirected graphs.

In the case of directed graphs, a path is a sequence (possibly empty) of adjacent nodes and linking edges, but with all edges in the same direction.

If there is a path from node A to node B, we say B is reachable from A.

A path from a node to itself is a cycle.

A loop is an edge from a node to itself.

More terminology: Connected graphs

Two nodes in an undirected graph are connected if there is a path between them.

For directed graphs, the notion of 'connected' is different (warning: the textbook doesn't make this clear):

Two nodes in a directed graph are connected if there is a sequence of adjacent nodes between them. (Notice that the edges can be in any direction between nodes in the sequence.)

A graph is connected if all pairs of nodes are connected.

More terminology: Components

A subgraph of a graph G, is a graph whose nodes and edges are a subset of those of G and whose edges have the same source and target as those of G.

A connected component (sometimes, just a 'component') of a graph G, is a largest connected subgraph (i.e. one that cannot be expanded with additional nodes without becoming disconnected).

Every graph can be partitioned into disjoint connected components.

Question: How do we compute the components of a graph efficiently? Is there a quadratic algorithm $O(N^2)$ or even a linear algorithm $O(N)$, where N is the number of nodes - what about the number of edges?

More about graphs

More about graphs:

- representing graphs in programming languages,
- traversal techniques for trees and graphs.

This is material from Chapter 6 of the course textbook.

Representing graphs

How do we represent graphs using the data types commonly provided in programming languages: arrays, linked lists, vectors, 2-D arrays etc?

There are several representations in widespread use. We deal with the two commonest.

Which to choose depends on

- properties of the graph (e.g. 'sparseness'),
- the algorithms we wish to implement,
- the programming language and how data structures are implemented, and
- application of the code.

Representing graphs: Adjacency lists

An adjacency list representation of a graph consists of a list of all nodes, and with each node n a list of all adjacent nodes (for directed graphs, these are the nodes that are the target of edges with source n).

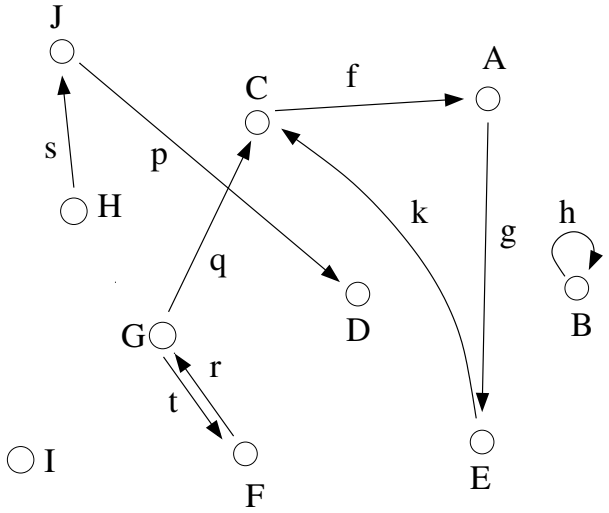


Figure : A directed graph.

Node	Adjacent nodes
A	E
B	B
C	A
D	
E	C
F	G
G	F, C
H	J
I	
J	D

Figure : Adjacency list representation of above graph

Representing graphs: Adjacency matrices

An adjacency matrix representation of a graph consists of a 2-dimensional array (or matrix), each dimension indexed by the nodes of the graph. The entries in the matrix are:

- 1 at index (m, n) if there is an edge from m to n,
- 0 at index (m, n) if there is no edge from m to n.

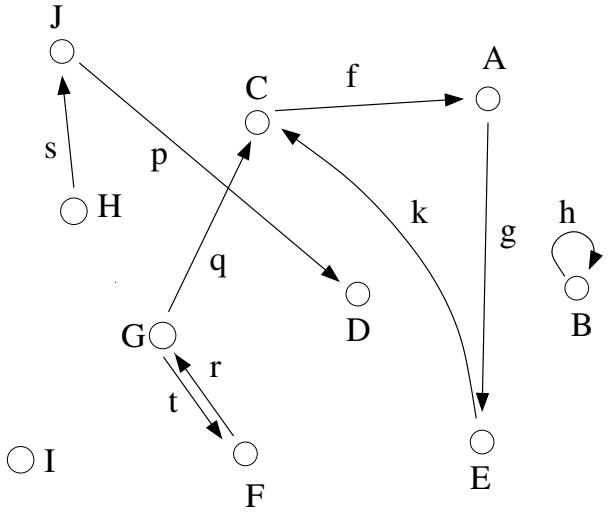


Figure : A directed graph.

	A	B	C	D	E	F	G	H	I	J
A	0	0	0	0	1	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0
E	0	0	1	0	0	0	0	0	0	0
F	0	0	0	0	0	0	1	0	0	0
G	0	0	1	0	0	1	0	0	0	0
H	0	0	0	0	0	0	0	0	0	1
I	0	0	0	0	0	0	0	0	0	0
J	0	0	0	1	0	0	0	0	0	0

Figure : Adjacency matrix representation of above graph (source nodes on the left)

Representing graphs: Notes

- These are both tabular representations. The exact choice of types to represent them depends on the application.
  For example, an adjacency list may be an array of linked lists, if we wish to have fast (random) access to the lists of adjacent nodes, but to iterate through these lists.
- Notice how sparse the adjacency matrix is: most entries are zero as there are few edges in the graph.
- The representations need modification to include various data. For example, for a multigraph, we may record the edges as well as the adjacent nodes in a list representation, or the number of edges in a matrix representation. Also for edge-weighted graphs, we include the weights in the representation.
- Notice that for undirected graphs, the matrix is symmetrical.

- Adjacency matrices allow us to do arithmetic! We may add and multiply matrices, take determinants etc. These determine transformations of graphs and values derived from graphs.
- Different representations for different tasks: We choose which representation to use by considering the efficiency of an algorithm for the task.
  Example: Consider the task of finding whether there is a path of length 2 from a given node S to another node T. For adjacency lists, we consider the list of nodes adjacent to S - in the worst case this is of length E (the number of edges of the graph). For each node in this list we see whether T is in its list. Thus the worst-case time complexity is $E \times E$. Using adjacency matrices, this complexity is N (the number of nodes) - why?
  In general, for path finding algorithms, adjacency lists are sometimes more efficient, adjacency matrices for other algorithms.

Traversals: Trees

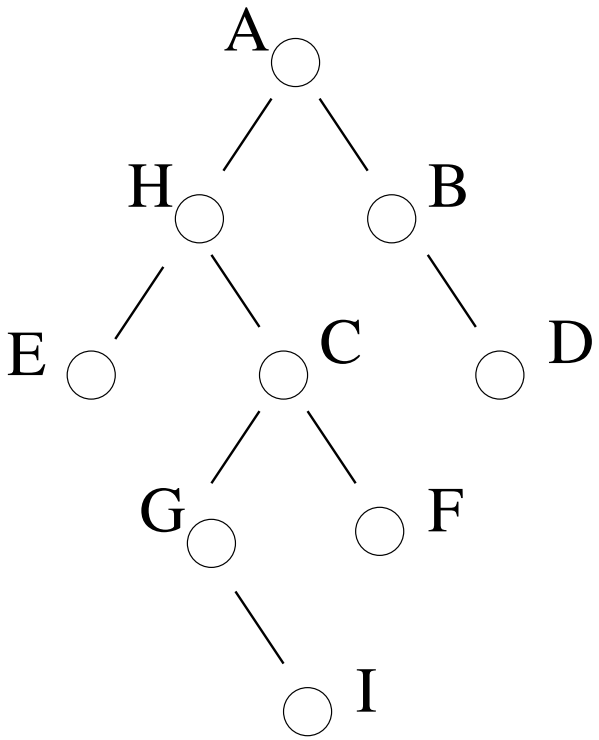
A traversal of a tree or graph is a means of visiting the nodes using the edges and revisits of nodes. Usually, there are rules to determine possible next nodes to visit or revisit.

There are many techniques, the most widely used are Depth-First Search and Breadth-First Search.

For trees, we start at the root and:

- For Depth-First Search (DFS), visit all descendants of a node, before visiting sibling nodes;
- For Breadth-First Search (BFS), visit all children of a node, then all grandchildren, etc;

Traversals of trees: Examples



A Depth-First Search from the root A (in terms of order of visiting nodes): A,H,C,F,G,I,E,B,D.

Another DFS (left-to-right): A,H,E,C,G,I,F,B,D.

For DFS, the revisiting of nodes takes place through backtracking.

A Breadth-First Search (right-to-left): A,B,H,D,C,E,F,G,I.

Priority search

Now let trees have a numerical priority assigned to each node.

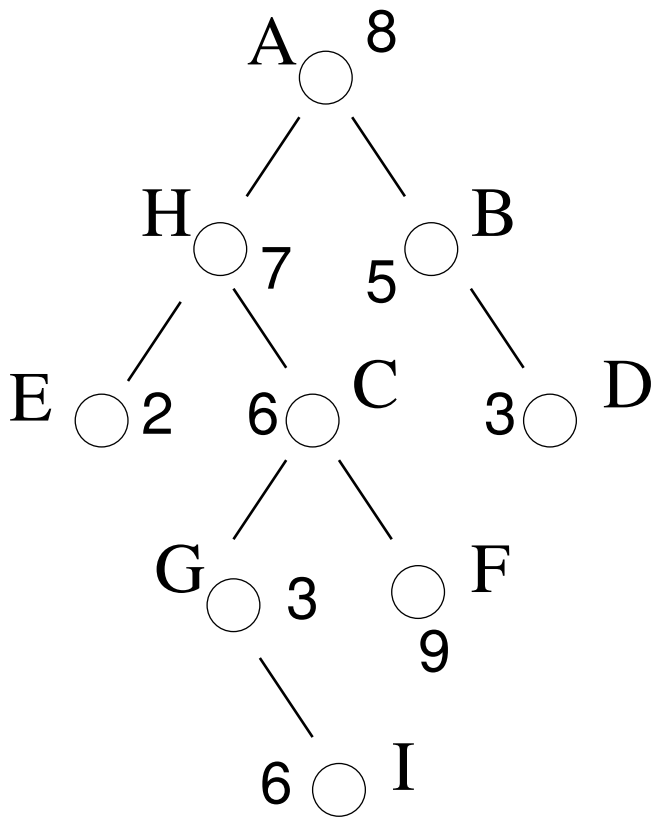
A priority search is:

1. Visit the root.
2. At each step, visit a node that has highest priority amongst unvisited children of visited nodes.

Priority searches are often used to implement heuristic search methods where the priority is calculated at each node to provide an indication of whether a route through this node is likely to reach a required goal.

Priority search: Example

As an example consider the tree:



One priority search of this tree is: A, H, C, F, B, G, I, D, E.

Generic search routine for trees

We now show how we can code all the above search techniques (traversals) of tree with the same program.

The program uses an auxiliary date structure with operations push, pop, top, empty.

- When the data structure is a stack the traversal is DFS,
- When the data structure is a queue the traversal is BFS,
- When the data structure is a priority queue the traversal is a priority search.

A priority queue has the operations of a queue (push, pop, top, empty), but top returns an item of highest priority in the queue and pop removes this item.

Generic search routine for trees - program

Idea:

1. Start by pushing root node of the tree onto the structure.
2. Then visit the top element on the structure, pop it and push its children onto the structure.

The structure therefore stores the unvisited children of visited nodes in the order in which they are encountered.

Generic search routine for trees - pseudocode

The pseudocode labels each node u with search-num(u) giving the order the nodes are encountered:

```
u <- rootnode;
s <- empty;
i <-  0;
push(u,s);
while s not empty do
 { i <-  i+1;
   u <- top(s);
   search-num(u) <- i;
   pop(s);
   forall v children of u
     push(v,s) }
```

Exercise: Hand run examples above.

Traversals: From trees to graphs

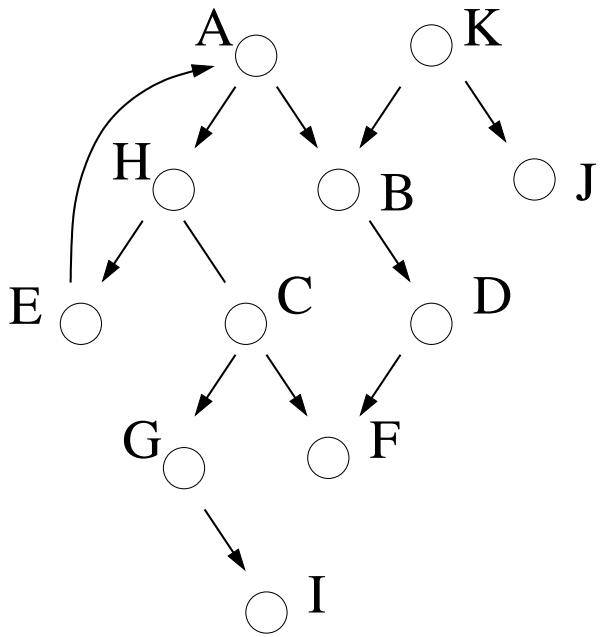
A tree is a directed graph such that:

> *There is a distinguished node (the root) such that there is a unique path from the root to any node in the graph.*

To modify tree traversal for graphs:

1. We may revisit nodes: so mark nodes as visited/unvisited and only continue traversal from unvisited nodes.
2. There may not be one node from which all others are reachable: so choose node, perform traversal, then start traversal again from any unvisited nodes.

Traversals of Graphs: Examples



A Depth-First Search: A,H,E,C,F,G,I,B,D,K,J.

A Breadth-First Search: A,H,B,E,C,D,G,F,I,K,J.

Depth-First Search for Graphs: Recursive Algorithm

Allocates a number dfsnum(u) to each node u in a graph, giving the order of encountering nodes in a depth-first search on a graph. Notice how we use this numbering to determine whether or not we have visited a node before (Zero = No, Non-zero = Yes).

```
forall nodes u do dfsnum(u) <- 0 end;
i <- 0;

visit(u) =
   { i <- i+1;
     dfsnum(u) <- i;
     forall nodes v adjacent to u do
       if dfsnum(v) = 0 then visit(v) end };

forall nodes u do
   if dfsnum(u) = 0 then visit(u) end;
```

The complexity of DFS

For a graph with N nodes and E edges:

- For the adjacency list representation, the complexity is linear $O(N + E)$,
- For the adjacency matrix representation, the complexity is quadratic $O(N^2)$.

Why?

Traversals of graphs: Notes

- The actual order determined by a depth-first search is that of a stack-based discipline, that is when we backtrack we visit the 'most recent' unvisited branch first, before backtracking to more remote ancestors.
- Breadth-First Search does not have such a recursive formulation.
- The generic code for trees above extends to graphs to provide DFS, BFS and priority search for graphs using the same auxiliary data structures.

Survey of graph algorithms

Many graph algorithms are based on traversals.

There are efficient (usually linear) algorithms based on Depth-First Search for:

- finding the connected components of a graph (how? - consider the case of undirected graphs),
- detecting cycles in a graph,
- finding 'strong components' of a graph,
- planarity testing and embedding (see below),
- articulation points and blocks of undirected graphs, 'orientability' and 'reducibility' of graphs, etc.

Survey of graph algorithms: Path finding

There are numerous path-finding problems in graphs and a variety of algorithms:

- To find all paths between all pairs of nodes ('transitive closure'), or
- To find all paths between a fixed pair of nodes, or
- To find all paths from one node to all others (the 'single source problem').

When the edges are labelled with numerical values, then we can ask for shortest paths, by which we mean a path of minimum length, where the length of a path is the sum of its edge labels.

Each problem above yields a shortest path problem - an example of an optimization problem.

In fact, the second and third problems are equivalent.

Survey of graph algorithms: Path finding (continued)

There is an optimization property concerning shortest paths:

> *If p is a shortest path from node u to node v via node w, then the portions of p from u to w and from w to v are both shortest paths.*

Proof: Evident!

Such optimization properties mean that there are direct methods of computing shortest paths: To accumulate shortest paths we need combine only other shortest paths (and not consider any other paths).

This is the basis of both Floyd's Algorithm and Dijkstra's Algorithm.

Survey of graph algorithms: Planarity

Planarity is about depicting graphs in 2-D space: how do we 'draw' graphs and can we do so without edges crossing.

Definition: An embedding of a (directed or undirected) graph in the plane is an allocation of distinct points in the plane to the nodes and distinct continuous lines (not necessarily straight - that is another problem) to the edges, so that no two lines intersect.

There may be more than one 'way' of embedding a graph in the plane. Can all graphs be embedded in the plane? No!

A graph that can be embedded in the plane is called a planar graph.

Hopcroft and Tarjan introduced a linear-time planarity algorithm (1974) based on Depth-First Search

Survey of graph algorithms: Graph colouring

A colouring of a graph with k colours is an allocation of the colours to the nodes of the graph, such that each node has just one colour and nodes linked by an edge have different colours.

To determine whether a graph can be coloured with k ($k \geq 3$) colours is an NP-complete problem.

Thus the only algorithms that exist in general for colourability are exhaustive unlimited back-tracking algorithms, and hence are exponential-time.

Survey of graph algorithms: Finale

The 4-colouring of planar graphs... is a celebrated problem.

Proposition Every planar graph can be coloured with just 4 colours.

- Discussed as a possibility in the 1850s,
- 1879: Kempe publishes a 'proof' of the 4-colour theorem,
- 1890: Heawood finds error in Kempe's proof, but shows Kempe's proof establishes the 5-colourability of planar graphs,
- Since then finding a proof of 4-colourability has been a catalyst for much combinatorial mathematics,
- 1977: Computer-aided proof of 4-colourability: reduced the graphs required to be coloured to a finite number (over 1000) and then used a computer to generate colourings.