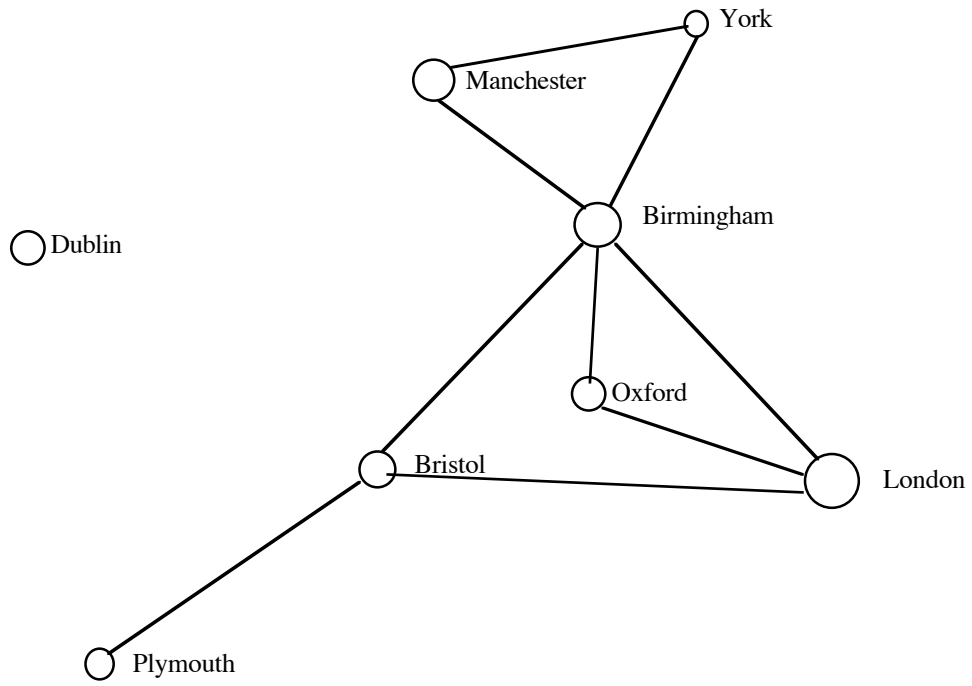
*Finding our way round maps, mazes, mathematical **graphs**.*

Earlier slides dealt with (rooted directed ordered) trees, which are a special case of a ***graph***.

Some maps are a depiction of a number of places, together with the roads/paths/railways connecting them. For example:



Such a map is an example of a mathematical ***graph***. There are lots of famous computing/mathematical problems to do with graphs, and lots of interesting algorithms.

A ‘connected’ graph is one in which all nodes are reachable from all others. Not all graphs are connected – for example, you can’t get from London to Dublin using the connections shown on the map above.

People are very good at reading maps. When they can’t see the whole map, people aren’t so good at finding their way. So to illustrate that graph-searching is a problem you can put a person down in a maze (a kind of graph) where they can’t see the whole maze.

> *“You are in a debris room filled with stuff washed in from the surface. A low wide passage with cobbles becomes plugged with mud and debris here, but an awkward canyon leads upward and west. A note on the wall says ‘magic word* xyzzy*’”*

To make the problem easier I'm going to restrict myself to *directed graphs* in which each edge (connection) goes *from* one node (place) *to* another – like a one-way street.

> *We haven't lost any generality: we can imitate undirected edges with a pair of directed edges, one out and one back.*

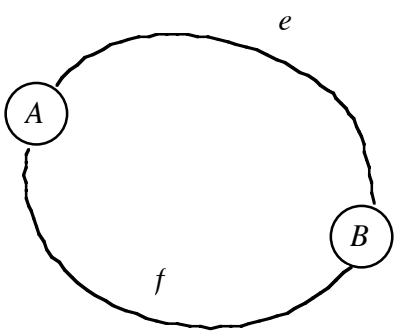
I'm going to solve a sequence of problems:

- find all nodes in a graph which are reachable from *A*;
- find a path from *A* to *B*, if there is one;
- find the shortest path from *A* to *B*.

I shall have variations on each of these problems.

Node *B* is ***reachable*** from *A* if $A = B$ or if there is a ***path*** from *A* to *B*.

A path is a sequence $N_0, E_0, N_1, E_1, ..., E_{n-2}, N_{n-1}$ of nodes interspersed with edges, such that E_i is an edge which connects N_i and N_{i+1}.



A path as a sequence of nodes won't do: given <A,B> and this graph we don't know if the path follows edge e or edge f.

A path as a sequence of edges won't do: given this graph the sequence <e,f> might be a path A to B to A or a path B to A to B.

The path $\langle A, e, B, e, A \rangle$ *is in railway terms, a return journey.*

When working with trees, a sequence of nodes will do, because there is a unique sequence of edges which connects them.

When working with directed graphs, a sequence of edges will do, because the nodes can be deduced.

The shortest (sequence of edges) path is ⟨ ⟩,

> *it leads 'from' any node 'to' that same node, without using any edges*

the next simplest is a single edge, the next two edges, and so on.

One interesting path is a *cycle*: a non-empty path from *A* to *A* which doesn't use any edge or node more than once (except that *A* must occur at the beginning and end of the path).

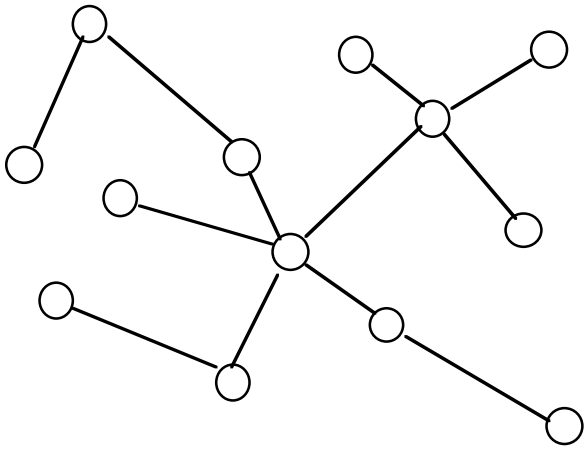
> *In the map above there is a cycle London - Oxford - Birmingham - London – and vice-versa – , but London - Oxford - London is not a cycle, and neither is the empty path from London to London.*
>
> *London - Oxford - Birmingham - York - Manchester - Birmingham - London isn't a cycle because it goes through Birmingham twice. But it can be split into two cycles.*

A form of graph which we have seen already is the *tree*.

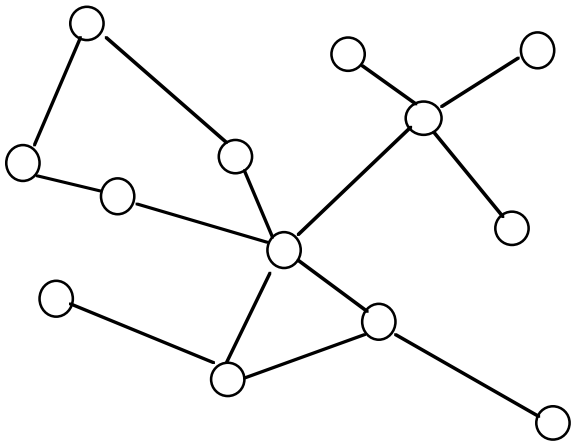
An ***undirected tree*** is a connected graph in which there are no cycles

> *alternatively: a connected graph with N nodes and N–1 edges, or a graph with no cycles, N nodes and N–1 edges.*

This is an undirected tree:

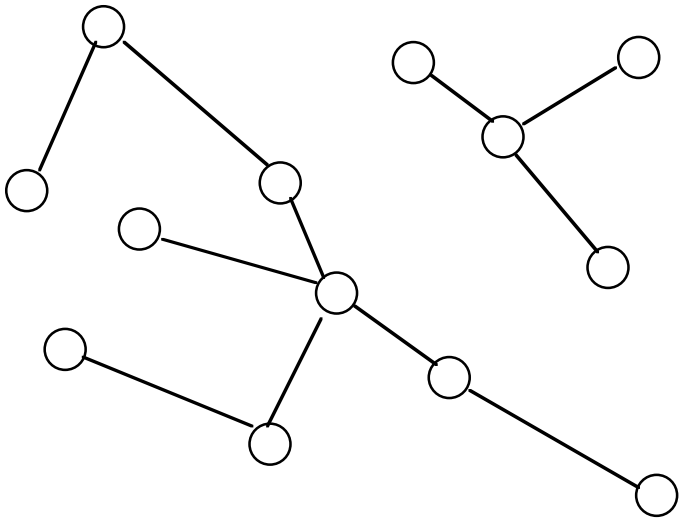


this isn't:



Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

and neither is this:

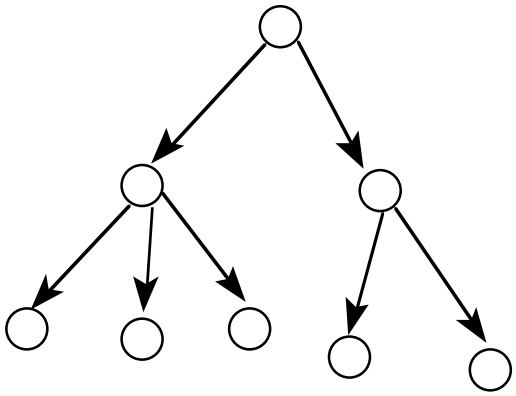


In rooted directed trees we require additionally:

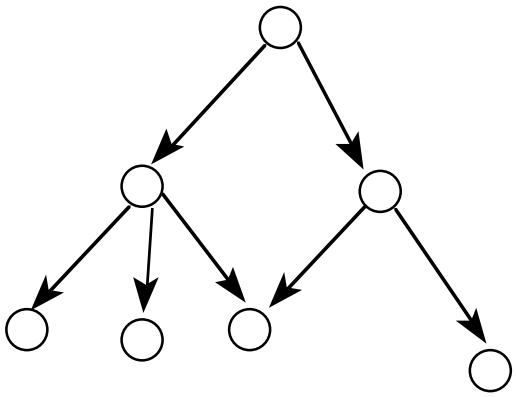
- one special node (the root);
- that each node, apart from the root, has exactly one edge pointing to it; the root has no edges pointing to it.

> *An alternative definition is more economical, but less illuminating: a rooted directed tree is a connected directed graph with N nodes and N–1 edges, in which each node apart from the root has exactly one edge leading to it; the root has no edge leading to it.*

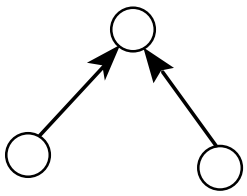
This is a rooted directed tree:



and this isn't (even though it has no cycles) – it's a *rooted directed acyclic graph* or rooted DAG:



and this is a DAG with no cycles and no multiple paths, but without a root:



Java classes for directed graphs.

```
class Node {
  public String name; public Edge[] edges;
  ...
}
class Edge {
  public String name; public Node to;
  ...
}
```

- a node has a name and a collection of edges which lead from it, an edge has a name and goes to a node.

> *Since our edges are directed, and since we access them via the node from which they lead, the* `Edge` *class doesn't have a 'from' field. Our edges have names, just as roads (M4) and railways (East Coast) have names.*

The *Node* instance method call `printchildren()` will print out the names of all the nodes reachable from a particular node, provided that the graph has no cycles:

```
public void printchildren() {
  System.out.println(name);
  for (int i=0; i<edges.length; i++)
    edges[i].to.printchildren();
}
```

1. *When r has no edges leading from it (*`edges.length==0`*)* `r.printchildren()` *prints the name of r, and doesn't do anything else.*

2. *When r does have edges leading from it* `r.printchildren()` *prints the name of r and then deals with each of the nodes reachable from r in a sequence of recursive calls.* ***If*** *each of those recursive calls prints all the names in its subgraph* ***then*** `r.printchildren()` *prints all the names in the graph reachable from r.*

3. *So* ***provided that*** *the graph reachable from* `r.edges[i].to` *is in every case a* smaller *graph than the graph reachable from r – has fewer nodes – then* `r.printchildren()` *is a valid recursive algorithm which prints the name of every node in the graph reachable from r.*

If the graph has cycles, then the argument breaks down in the last step: if there's an edge from *r* to *t* to *u* to ... to *r* then it isn't true that the graph reachable from *t* is smaller than the graph reachable from *r*.

`r.printchildren()` will print the name of each node in the graph reachable from *r* *exactly once*, if that graph happens to be a tree.

1. *When r has no edges leading from it (*`edges.length==0`*)* `r.printchildren()` *prints the name of r exactly once, and doesn't do anything else.*

2. *When r does have edges leading from it* `r.printchildren()` *prints the name of r and then deals with each of the nodes reachable from r in a sequence of recursive calls.* ***If*** *each of those recursive calls prints all the names in its subtree exactly once* ***and*** *those subtrees are disjoint,****then*** `r.printchildren()` *prints all the names in the graph reachable from r exactly once.*

3. *So* ***provided that*** *the graph reachable from* `r.edges[i].to` *is in every case a* smaller *graph than the graph rooted at r – has fewer nodes –* ***and*** *those graphs are disjoint, then* `r.printchildren()` *is a valid recursive algorithm which prints the name of every node in the graph reachable from r exactly once.*

If the graph is a DAG – that is, if there is more than one path from *r* to any node – then the argument breaks down in the second step.

> *but part of the condition that a graph is a tree is that there is only one path from r to t.*

Printchildren visits every node in the graph by traversing every edge. Actually it tries to traverse every *path*.

Printchildren does a certain amount of work at each node – $O(N)$ – and also a certain amount of work with each edge – $O(E)$ – so overall it is $O(N + E)$ in time.

Each method call uses $O(1)$ space, and the number of methods is proportional to the maximum path length: in the worst case it uses $O(E)$ space.

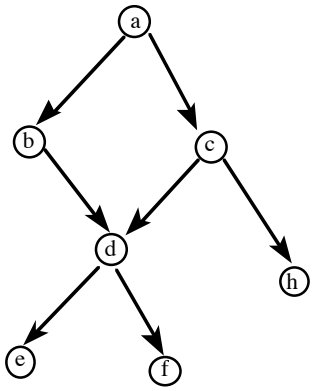
To print out all the nodes of a graph, each node once only, *printchildren* requires that the graph form a tree.

What happens if the graph is not a tree? What does *printchildren* do then?

If the graph has no cycles, but is not a tree, *printchildren* will print some node name (or many node names) more than once.

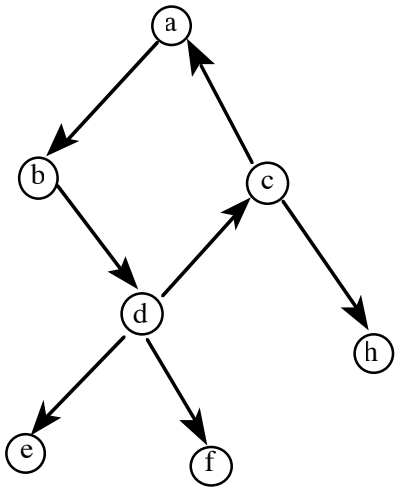
If it has a cycle, then both inductive arguments fail.

Starting from *a* in this graph, which is not a tree but is a DAG:



printchildren will visit the nodes in something like this order: a; b; d; e; f; c; d; e; f; h.

Starting from *a* in this graph, which is neither a tree nor a DAG, *printchildren* will loop a; b; d; e; f; c; h; a; b; d; e; f; ....:



Traversing a (possibly cyclic) graph.

How do we visit all the nodes of a graph – which may have cycles and/or multiple paths between nodes – in such a way that we visit each node exactly once (i.e. without looping)?

I shall write a procedure which prints the name of each node exactly once: 'visiting' will be the action of printing the name.

I give my graph-traversing method a *Vector* argument describing all the nodes that have already been visited.

`r.printreachable(v)` will print the names of all the nodes in the graph reachable from *r* except for those listed in *v*, so `r.printreachable(new Vector())` will print all the reachable nodes:

```
public void printreachable(Vector visited) {
  if (!visited.contains(this)) { // has this node been printed?
    visited.addElement(this); // don't print it again
    System.out.println(name);
    for (int i=0; i<edges.length; i++)
      edges[i].to.printreachable(visited);
}
```

I don't give an inductive proof of my claim that this procedure does what it is supposed to do. I invite you to try to make one of your own.

> *You might think it would be OK to add* `visited.removeElement(this)` *when the* for *ends and all the children nodes have been visited. But look at the DAG, and notice that in that case we would visit the d,e,f subtree twice!*

`r.printreachable(new Vector())` (invisibly) generates a ***spanning tree*** covering the nodes reachable from *r*: no edge is accepted which leads into *r*, and only one entering edge is accepted for all other reachable nodes.

Printreachable is not as fast as *printchildren*: the test `!visited.contains(this)` is probably $O(N)$, and it's executed once for each edge in the graph, so the overall time will be $O(NE)$ at least.

Printreachable needs $O(N)$ space for the *visited* vector and worst case $O(E)$ space for the method calls, so it's $O(N + E)$ in space.

Is there a path from A to B?

I extend the technique used in *printreachable*: `r.pathq(v, s)` finds if there is a path from *r* to *s* which doesn't pass through the nodes listed in *v*. So `r.pathq(new Vector(), s)` finds if there is a path between *r* and *s*:

```
boolean pathq(Vector visited, Node s) {
  if (this==s) return true; // empty path exists
  else
  if (!visited.contains(this)) { // not a cycle
    visited.addElement(this);
    for (int i=0; i<edges.length; i++)
      if (edges[i].to.pathq(visited, s)) return true;
    return false;
  }
}
```

This procedure begins as if it would visit all the nodes in the graph, but stops as soon as a path to *s* is found.

As with *printreachable*, *pathq* takes $O(NE)$ time and uses $O(N + E)$ space.

Calculate a path from A to B.

This method delivers a path – a vector of edges – which lead from *r* to *s*, if one exists. If no path exists it returns a null reference.

> *Note the distinction between a null reference – no vector, signalling no path – and an empty vector – signalling an emtpy path.*

```
public Vector pathfind(Vector visited, Node s) {
  if (this==s) return new Vector(); // the empty path
  else
  if (!visited.contains(this)) {
    visited.addElement(this);
    for (int i=0; i<edges.length; i++) {
      Vector v = edges[i].to.pathfind(visited, s);
      if (v!=null) {
        v.insertElementAt(edges[i],0); return v;
      }
    }
    return null; // no path this way
  }
}
```

> *notice how the path to the goal is built up step-by-step once the goal has been found.*

If there is a path, then *pathfind* will find a path in worst-case $O(NE)$ time, using $O(N + E)$ space. But ***not necessarily*** the shortest path. Oh no!

An aside: more efficient visiting.

Printchildren takes $O(N + E)$ time and uses $O(E)$ space (provided the graph is in fact a tree). *Printreachable*, *pathq* and *pathfind* as defined above take $O(NE)$ time and use $O(N + E)$ space.

We might use a binary balanced tree, instead of a vector, and reduce the time to $O(E \lg N)$; we might reduce it to something like $O(N + E)$ by using a hash table.

But in fact there is a much simpler solution: record visits in the Node data-structure itself! The cost is an additional *cleanup* method call to remove the marks inserted by a traversal of the tree.

```
class Node {
  private String name; private Edge[] edges;
  private boolean visited;
  public void printreachable() ...
  public boolean pathq(Node s) ...
  public Vector pathfind(Node s) ...
  public void cleanup() ...
  ...
}
```

Here is the fast ($O(N + E)$) version of *printreachable*. Note that it tests the *visited* variable where previously a vector was searched, and sets the *visited* variable where previously an element was added to a vector:

```
public void printreachable() { // fast version
  if (!visited) {
    visited = true;
    System.out.println(name);
    for (int i=0; i<edges.length; i++)
      edges[i].to.printreachable();
  }
}
```

But once all the nodes have been visited, and marked as visited, we need a way of cleaning off the marks:

```
public void cleanup() {
  if (visited) {
    visited = false;
    for (int i=0; i<edges.length; i++)
      edges[i].to.cleanup();
  }
}
```

What the fast version of *printreachable* does is to print all the nodes reachable from *r* ***except*** those which are only reachable through a node whose *visited* field is set to *true*.

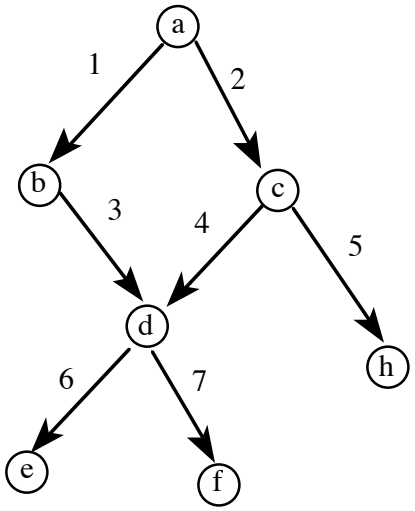
> *It follows that* printreachable *won't print the entire graph if there are any* visited *fields already set to* true.

The additional cost of *cleanup* is small – $O(N + E)$ in time and worst-case $O(N + E)$ in space. In return we get a fast, simple graph-traversal/search mechanism.

```
boolean pathq(Node s) { // fast version
  if (this==s) return true; // empty path exists
  else
  if (!visited) { // not a cycle
    visited = true;
    for (int i=0; i<edges.length; i++)
      if (edges[i].to.pathq(s)) return true;
    return false;
  }
}

Vector pathfind(Node s) { // fast version
  if (this==s) return new Vector(); // the empty path
  else
  if (!visited) { // not a cycle
    visited = true;
    for (int i=0; i<edges.length; i++) {
      Vector v = edges[i].to.pathfind(s);
      if (v!=null) {
        v.insertElementAt(edges[i],0); return v;
      }
    }
    return null; // no path this way
  }
}
```

Finding the shortest path: queues rather than stacks.



Consider the DAG above, and consider the problem of finding a path from *a* to *h* in that graph.

The *findpath* procedure extends its search – through the spanning tree – in a 'stack-like' way. Starting from *a* it has to consider edges 1 and 2; it considers edge 1 first, leaving 2 for later. Edge 1 leads to *b*; edge 3 leads from *b* to *d*; that in turn leads it to consider edges 6 and 7 to *e* and *f*. Only when all those searches have failed does it consider edge 2, which is part of the answer.

This is called ***depth-first*** search; it extends the spanning tree by taking the deepest so-far unexpanded node and expanding it.

Another way of looking at it is that it keeps a 'stack' of nodes (or edges - it makes no difference) and it extends the search by taking (popping) the node from the top of the stack and replacing it with (pushing) the nodes which are accessible from that node:

<a> // starting position
<b,c> // expand a using edges 1 & 2
<d,c> // expand b, using edge 3
<e,f,c> // expand d, using edges 6 & 7
<f,c> // node e has no children
<c> // neither does f
<d,h> // node d is the left-most child of c
<h> // but we've already visited d, so we don't expand it
<> // bingo! node h is our goal.

The node on the front of the list is always the unexpanded node which is farthest from the starting point.

We can consider this an *inefficient* search: it searches regions far away from the starting point even though the destination may be close. In the example above it looks at all the rest of the graph before it gets round to *h*.

If the graph leading from node *d* was very large, it would be a long time before the algorithm looked at node *c* and thus found *h*.

Too often 'depth-first' (stack-wise) search wanders off into the distance in the wrong direction.

We can do better.

Suppose we extend the search queue-wise: when we reach an unexplored node, we add its edges not to the *front* of the list of things to look at, but to the *back*.

> *The nodes at the back of the queue will now be the ones farthest from the starting point; by expanding the node at the front of the queue we will be exploring the region of the graph nearest to the starting point.*

We will construct our search 'breadth-first', and – almost by accident! – the first time we come across the goal we will necessarily have reached it by a shortest path.

> *a shortest path, not necessarily the shortest path.*
>
> *I'm measuring length of path by number of edges, but I shall refine that later.*

Here's a breadth-first search, using the same DAG and the same problem as before:

<a> // starting position
<b,c> // expand a using edges 1 & 2
<c,d> // expand b, using edge 3
<d,d,h> // expand c, using edges 4 & 5
<d,h,e,f> // expand d, using edges 6 & 7
<h,e,f> // we've already expanded d (it's been painted)
<e,f> // bingo! node h is our goal.

Even in this very small example, there are many fewer steps in this expansion; far-away parts of the graph are simply not examined.

I can use a vector as a primitive kind of queue. Given that, here's a breadth-first version of *pathq*:

```
public boolean breadthfirstpathq(Node s) {
  Vector q = new Vector();
  q.addElement(this);
  while (q.size()!=0) { // as long as there is a queue ...
    Node r = (Node)q.elementAt(0);
    q.removeElementAt(0);
    if (r==s) return true; // there is a path!
    else
    if (!r.visited) { // not already dealt with
      r.visited=true;
      for (int i=0; i<r.edges.length; i++)
        q.addElement(r.edges[i].to);
    }
  }
  return false; // search failed, no path
}
```

Unfortunately the work done at each node will be dominated by `q.removeElementAt(0)` – almost certainly worst-case $O(N)$. So it is worth looking at a faster implementation of a queue.

Aside: qeues are easier than you might think.

A queue is a data structure which supports insertion at the *back* and removal at the *front*. By contrast, a list (stack) provides insertion and removal at the front.

One easy way to build a queue is to use an array Q and a couple of variables h and t: the first element in the queue is $Q[h]$, and the last element is $Q[t-1]$.

When t gets too big we can wrap around to the beginning ... we can make more space if the queue gets full ... (see Weiss pp 416-421).

```
class Queue { // array version
  private Object[] Q; private int h, t, count;
  private final int inc = 10; // smallest queue
  public Queue() { Q = new Object(inc); h=0; t=0; count=0; }

  public boolean isempty() { return count==0; }

  public void insert(Object o) {
    if (count==Q.length) { // stretch the queue
      Object[] R = new Object(Q.length+inc);
      for (int i=0, c=count; i<c; i++) R[i]=remove();
      count=c; h=0; t=count; Q=R;
    }
    Q[t]=o; t=(t+1)%Q.length; count++;
  }

  public Object remove() { // assume queue is not empty
    Object o = Q[h];
    h=(h+1)%Q.length; count--; return o;
  }
}
```

Remove is constant time. *Insert* sometimes stretches the queue-holding array, but if the array is made large enough to start with, it's constant time.

Another way to build a queue is to make a list which eats its head!

Use a null pointer as the value describing an empty queue.

For a non-empty queue make a list as usual, but make the *last* element of the list contain not a null reference but a reference to the first element in the list (that is, make a circle).

Then keep a note of the *last* element in the queue. From the pointer in the last element you can find the first element!

> *It needs a bit of care when you insert into an empty queue, and when you remove the last element in a queue (thus making the queue empty).*

```
class Queue { // of Objects
  private static class Qnode {
    public Object entry; public Qnode next;
    public Qnode(Object o) { entry=o; }
  }
  private Qnode q; // initialised to null automatically

  public boolean isempty() { return q==null; }

  public void insert(Object o) {
    Qnode last = new Qnode(o);
    if (q==null) // insert into empty queue
      last.next=last; // first and last!
    else { last.next=q.next; q.next=last; }
    q=last;
  }

  public Object remove() {
    Qnode first = q.next;  // exception if q==null
    Object o = first.entry;
    if (first==q) q=null; // now queue is empty
    else q.next=first.next; // delete first
  }
}
```

Remove is constant time, but *insert* uses `new`, and that causes garbage collection, and that's unpredictable.

End of aside on queues.

So we have a procedure which will find if there is a path from *r* to *s* without spending time rooting round the far edges of the graph.

```
public boolean breadthfirstpathq(Node s) {
  Queue q = new Queue();
  q.insert(this);
  while (!q.isempty()) { // as long as there is a queue ...
    Node r = (Node)q.remove();
    if (r==s) return true; // there is a path!
    else
    if (!r.visited) { // not already dealt with
      r.visited=true;
      for (int i=0; i<r.edges.length; i++)
        q.insert(r.edges[i].to);
    }
  }
  return false; // search failed, no path
}
```

An analogy which may help: think of the graph as a collections of points in 3-D space; then `breadthfirstpathq` *expands a sphere around* r *until the sphere includes* s.

In order to be able to find the path that leads from *r* to *s*, I need to queue more information: not only a node, but also the path to it.

```
private class PathInfo {
  public Node r; public Vector path;
}

public void enqueue(Queue q, Node r, Vector path) {
  PathInfo pi = new PathInfo();
  pi.r=r; pi.path=path; q.insert(pi);
}

public Vector breadthfirstpathfind(Node s) {
  Queue q = new Queue;
  enqueue(q,this,new Vector());
  while (!q.isempty()) {
    PathInfo pi=(PathInfo)q.remove();
    Node r = pi.r; Vector path = pi.path;
    if (r==s) return path;
    else
    if (!r.visited) {
      r.visited=true;
      for (int i=0; i<r.edges.length; i++) {
        Vector path2 = path.clone();
        path2.addElement(r.edges[i]);
        enqueue(q,r.edges[i].to,path2);
      }
    }
  }
  return null;
}
```

This will find a shortest path in a directed graph, whether or not the graph has cycles, whether or not there is more than one path between some nodes.

This method is hugely inefficient, because it clones a vector each time it traverses an edge: $O(NE)$, even if the queue operations are constant-time.

If we use lists we can share information between all the descendants of a node:

```
class EdgeList { public Edge hd; public EdgeList tl; }

private class PathInfo {
  public Node r; public EdgeList path;
}

public void enqueue(Queue q, Node r, EdgeList path) {
  PathInfo pi = new PathInfo();
  pi.r=r; pi.path=path; q.insert(pi);
}

public EdgeList breadthfirstpathfind(Node s) {
  Queue q = new Queue;
  enqueue(q,this,null);
  while (!q.isempty()) {
    PathInfo pi=(PathInfo)q.remove();
    Node r = pi.r; EdgeList path = pi.path;
    if (r==s) return path;
    else
    if (!r.visited) {
      r.visited=true;
      for (int i=0; i<r.edges.length; i++) {
        Edgelist path2 = new EdgeList();
        path2.hd = r.edges[i]; path2.tl = path;
        enqueue(q,r.edges[i].to,path2);
      }
    }
  }
  return null;
}
```

This method builds paths in reverse order (the element at the *hd* of the path is always the last edge in the path). The result can be reversed if required.

If `new` were a constant-time operation, this method would be $O(N + E)$ in time, $O(NE)$ in space (because it constructs worst-case one path per node, each worst-case of length proportional to E).

We could make it adhere to these bounds more precisely by using arrays and not queues or lists. But that is another topic, and won't be covered in this course.

In practice, on maps and in lots of other sorts of graphs, edges have *weight*: a value which says how long, or wide, or important, or desirable, that edge is.

In looking for shortest paths I'm going to interpret the weight of an edge as the 'length' of the edge, and the length of a path will therefore be the sum of the weights of its edges.

I shall presume that the weight is an integer, and I shall assume that it is never negative.

```
class Edge {
  public String name; public Node to; public int weight;
};
```

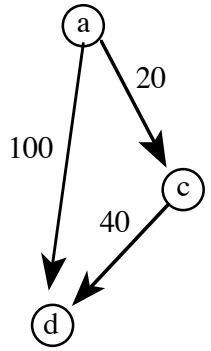
In order to find the shortest path, I shall need to keep the queue of nodes in path-distance order. That requires a structure called a *priority queue*.

```
interface PriorityQueue {
  public boolean isempty();
  public Object insert(Object o, int priority);
  public Object remove();
}
```

We shall deal with priority queues in a later set of slides.

If I keep the queue in distance order (least distance at the front of the queue), the first node in the queue will always be the one closest to the origin.

It may be the case that an entry farther back in the list is superseded as a result of expanding the node at the head of the list. For example, consider:



The route a-d has length 100; the route a-c-d has length 60. Here is how the search might go:

<(a,0,[])> // starting position
<(c,20,[a-c]),(d,100,[a-d])> // expand a
<(d,60[c-d,a-c]),(d,100[a-d])> // expand c

At this point we have two entries for d. The first is the one we want, and our `visited` *mechanism will ensure that we ignore the second one if we reach it.*

The Dijkstra algorithm.

The pathfinding procedure is *exactly* the same mechanism as before, with the exception that now I queue distances as well:

```
class EdgeList { public Edge hd; public EdgeList tl; }

private class PathInfo {
  public Node r; public EdgeList path; public int len;
}

public void enqueue(PriorityQueue q, Node r, EdgeList path,
                    int len) {
  PathInfo pi = new PathInfo();
  pi.r=r; pi.path=path; pi.len=len; q.insert(pi,len);
}

public EdgeList Dijkstrafind(Node s) {
  Queue q = new Queue;
  enqueue(q,this,null,0);
  while (!q.isempty()) {
    PathInfo pi=(PathInfo)q.remove();
    Node r = pi.r; EdgeList path = pi.path; int len=pi.len;
    if (r==s) return path;
    else
    if (!r.visited) {
      r.visited=true;
      for (int i=0; i<r.edges.length; i++) {
        Edgelist path2 = new EdgeList();
        path2.hd = r.edges[i]; path2.tl = path;
        enqueue(q,r.edges[i].to,path2,len+r.edges[i].weight);
      }
    }
  }
  return null;
}
```

This is the “Dijkstra algorithm” for finding shortest paths in a graph.

It is important that in the while loop we ***don’t look*** to see if an edge leads to the goal. Success only comes when the goal reaches the front of the queue.

That’s almost all there is to say about finding shortest paths, but there is one final twist. We can do better if we sniff the air to guess which way to go!

The A* algorithm.

Given an estimate of how far it is from where we are to where we are going, we can expand the nodes that seem to be nearer to the goal.

The technique is almost the same as the Dijkstra algorithm, but we keep the queue in 'estimated first to finish' order; that means we look at the best-seeming nodes first.

The estimates ***must*** be optimistic – strictly, they ***must not*** be pessimistic and they ***must not*** be negative.

> *notice: 'must be optimistic' is* **not** *the opposite of 'must not be pessimistic'.*
>
> *If we are searching a 2-dimensional map, the estimate can be 'Euclidean distance',* $\sqrt{easting^2 + northing^2}$.

If an estimate was pessimistic, then its entry might be ignored for too long.

If a pessimistic estimate was longer than an actual path to the goal, then the path labelled with that estimate would be completely ignored; that might mean that a shortest path could be missed.

This is the new *PathInfo* structure. *Distance* is the length of *path* (which leads from the starting point to node *r*); *estimate* is the estimated distance from *r* to the destination. We shall keep the queue in '*distance+estimate*' order:

```
class PathInfo {
  public Node r; public int distance, estimate;
  public Edgelist path;
}
public void enqueue(PriorityQueue q, Node r, EdgeList path,
                    int dist, int est) {
  PathInfo pi = new PathInfo();
  pi.r=r; pi.path=path;
  pi.distance=dist; pi.estimate=est;
  q.insert(pi,dist+est);
}
```

When the goal (Node s) reaches the front of the priority queue, its estimate must be exact (anything else would be pessimistic or negative!), and no other remaining path, however optimistically estimated, can get there faster, so in that case we shall have found the shortest path.

Now a complication: we expand the priority queue in *distance+estimate* order, so we can't be sure that when we first visit a node, we have found the shortest path to it. The `visited` boolean that has served so far will no longer do the job.

I add an element to the Node structure to cope with this: it records the shortest distance yet found to this node.

```
class Node {
  public String name; public Edge[] edges;
  public boolean visited; public int distfromorigin;
};
```

When I look at a node I may find it expanded, but it may have been expanded (due to over-optimism) using a longer path from the source than the one I'm exploring.

And then I have to expand it again; oh dear. The new expansion will eventually win over the remains of the old expansion, because the estimates will be the same and the distances will be less.

Re-expansion means that it is hard to decide just what the complexity of this algorithm is. Never mind.

```
public EdgeList Astarfind(node s) {
  Queue q = new Queue;
  enqueue(q,this,null,0,something);
  while (!q.isempty()) {
    PathInfo pi=(PathInfo)q.remove();
    Node r = pi.r; EdgeList path = pi.path;
    int dist=pi.dist; int est=pi.est
    if (r==s) return path;
    else
    if (!r.visited || r.distfromorigin>dist) {
      r.visited=true; r.distfromorigin=dist;
      for (int i=0; i<r.edges.length; i++) {
        Edgelist path2 = new EdgeList();
        path2.hd = r.edges[i]; path2.tl = path;
        enqueue(q,r.edges[i].to,path2,
                dist+r.edges[i].weight,estimate);
      }
    }
  }
  return null;
}
```

It is claimed that, despite the multiple expansions of some nodes which might be caused by poor estimation, this algorithm is faster in practice than the Dijkstra version.

I invite you to experiment.

For those of you who have come this far, a puzzle.

London Transport provide (or used to provide at Waterloo) machines which tourists could use to find their way round the Underground.

Londoners know lots of clever ways to use the Underground. For example, I know that the fastest way from Finsbury Park to Paddington is: Victoria Line to Oxford Circus, over the little bridge, Bakerloo to Paddington. And the fastest way from Finsbury Park to Waterloo is Victoria Line to Oxford Circus, cross the platform, Bakerloo to Waterloo.

But LT's machine told me to go to Waterloo via Euston and the Northern Line, and it told me to go to Paddington via Euston Square and the Circle!!!

Can you write a search algorithm which takes account of journey times, change times, frequency of trains on different lines, reliability of lines, ... and which would perform as well as you and I can when planning routes through the Tube?