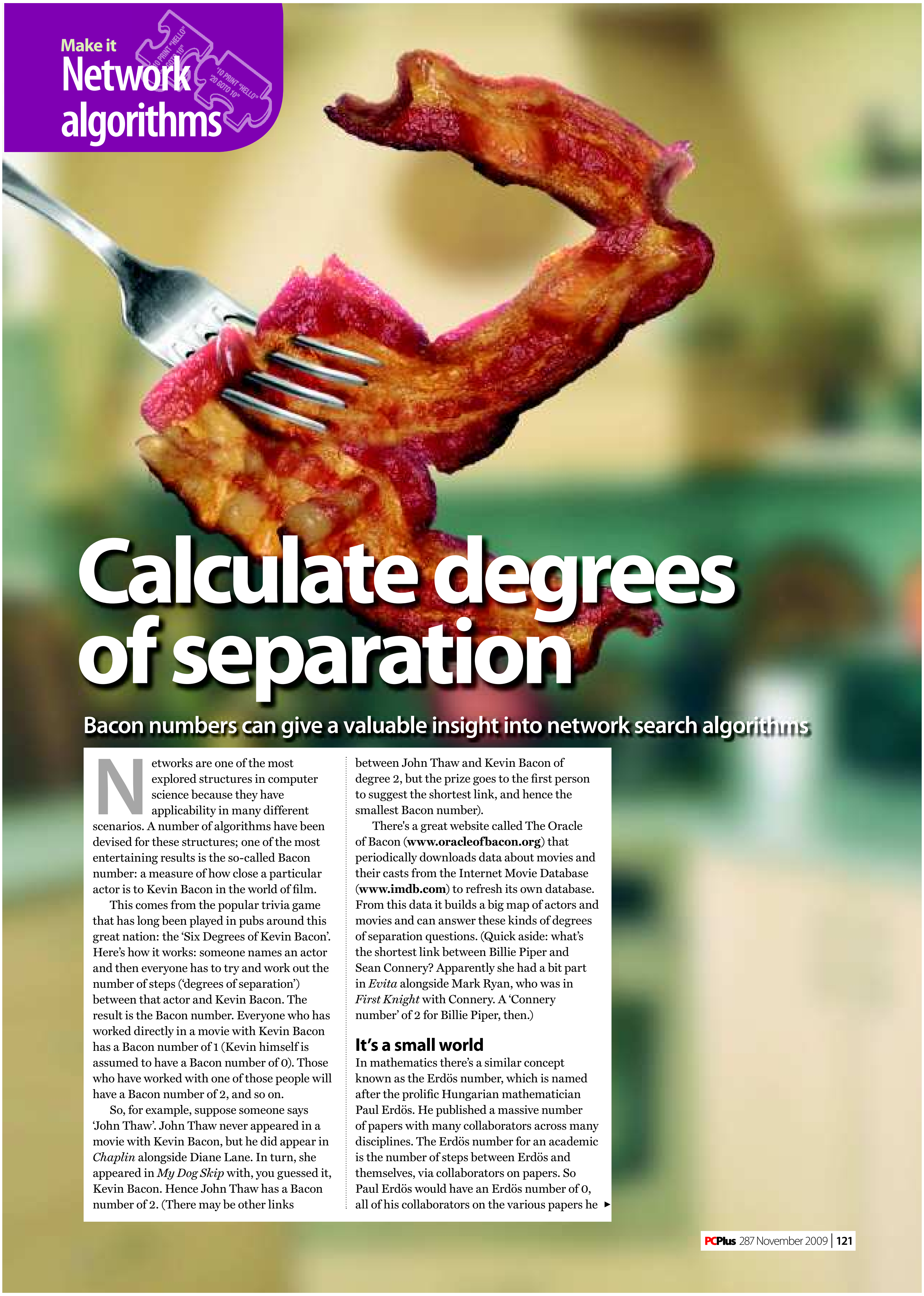


Calculate degrees of separation

Bacon numbers can give a valuable insight into network search algorithms

Networks are one of the most explored structures in computer science because they have applicability in many different scenarios. A number of algorithms have been devised for these structures; one of the most entertaining results is the so-called Bacon number: a measure of how close a particular actor is to Kevin Bacon in the world of film.

This comes from the popular trivia game that has long been played in pubs around this great nation: the 'Six Degrees of Kevin Bacon'. Here's how it works: someone names an actor and then everyone has to try and work out the number of steps ('degrees of separation') between that actor and Kevin Bacon. The result is the Bacon number. Everyone who has worked directly in a movie with Kevin Bacon has a Bacon number of 1 (Kevin himself is assumed to have a Bacon number of 0). Those who have worked with one of those people will have a Bacon number of 2, and so on.

So, for example, suppose someone says 'John Thaw'. John Thaw never appeared in a movie with Kevin Bacon, but he did appear in *Chaplin* alongside Diane Lane. In turn, she appeared in *My Dog Skip* with, you guessed it, Kevin Bacon. Hence John Thaw has a Bacon number of 2. (There may be other links between John Thaw and Kevin Bacon of degree 2, but the prize goes to the first person to suggest the shortest link, and hence the smallest Bacon number).

There's a great website called The Oracle of Bacon (**www.oracleofbacon.org**) that periodically downloads data about movies and their casts from the Internet Movie Database (**www.imdb.com**) to refresh its own database. From this data it builds a big map of actors and movies and can answer these kinds of degrees of separation questions. (Quick aside: what's the shortest link between Billie Piper and Sean Connery? Apparently she had a bit part in *Evita* alongside Mark Ryan, who was in *First Knight* with Connery. A 'Connery number' of 2 for Billie Piper, then.)

It's a small world

In mathematics there's a similar concept known as the Erdös number, which is named after the prolific Hungarian mathematician Paul Erdös. He published a massive number of papers with many collaborators across many disciplines. The Erdös number for an academic is the number of steps between Erdös and themselves, via collaborators on papers. So Paul Erdös would have an Erdös number of 0, all of his collaborators on the various papers he ►

▸ wrote would have an Erdös number of 1, and all their collaborators on other papers would have an Erdös number of 2, and so on.

All of these degrees-of-separation-type numbers are based upon the small world phenomenon. This is the observation that the social networks of large groups of people in modern society are very interconnected. Typically, they're distinguished by short path lengths between any two nodes.

The psychologist Stanley Milgram did some research on these social networks during the '60s and found that, on average (and using the methodology he described), any person in the US could trace a path to any other person in the US using between five and six links. Hence the term 'six degrees of separation', although Milgram did not use this particular phrase (it was first widely used as the name of a play and then a film – the star of the movie was Will Smith, who has a Bacon number of 2).

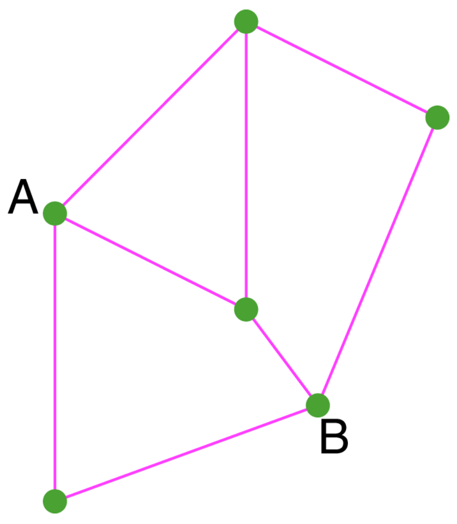
The small world phenomenon is of course widely used nowadays by the social networking sites such as Facebook and MySpace, although it's best known as part of LinkedIn, where you can view your network separation from another person directly.

Meet the neighbours

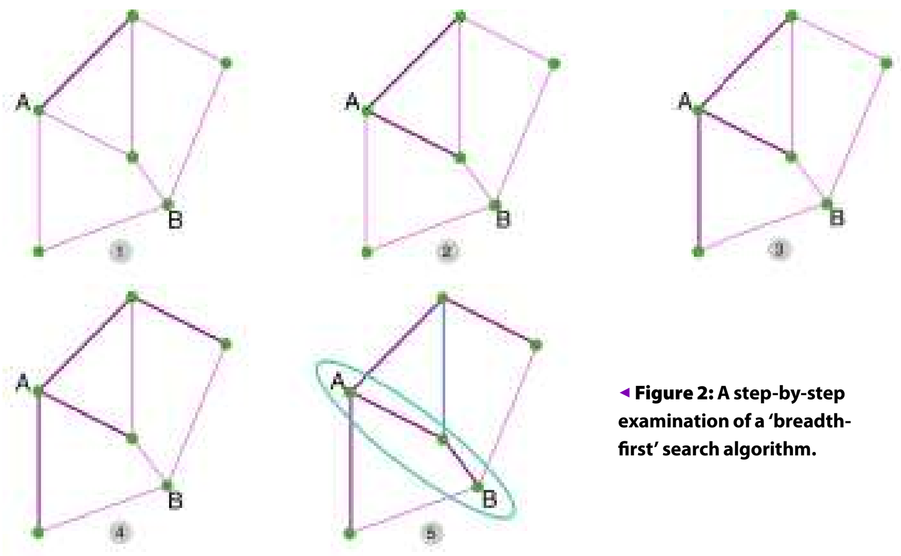
So how are these degrees of separation worked out? How does the Oracle of Bacon calculate the shortest number of links (the path) between actor A and actor B?

Graph theory holds the answer. Not graphs in the sense of those diagrams you had to draw at school to show $y=x^2$, but computer science graphs, or networks. The algorithm used is a 'shortest path' algorithm.

First, we need some terminology. I've already let slip a few pieces of jargon, so let's be more rigorous. A network is a data structure consisting of nodes (sometimes known as vertices) and the edges that connect the nodes. You can most easily represent a network as a matrix, though this is inefficient in terms of memory. Each row and column represents nodes, and the intersection of a row and column will be 1 (or 'true') if there's an edge between the nodes represented by the row and column, and 0 (or 'false') otherwise. Sometimes the edges have a 'weight' associated with them, which is the value in each cell. (An example of this is a network of towns, where the weight of the edge for two towns would be the length of the most direct route between them.)



▲ **Figure 1: A simple network. Here we need to find the shortest path from A to B.**



◂ **Figure 2: A step-by-step examination of a 'breadth-first' search algorithm.**

A path is a set of edges using which we can travel from one given node to another. The length of the path is either the number of edges we have to travel to follow the path, or the sum of the weights of the edges. The shortest path is obviously the path with the smallest 'length'.

Suppose we have a network, as shown in Figure 1. We want to find the shortest path from A to B. We'll first consider the simple case where the path length is merely the count of the number of edges we follow.

We'll need to store some information for each vertex we visit, so we should first create an array that stores references to all the nodes in the network. We'll get to the information we need to store in a moment, as we describe the algorithm. We'll also need an implementation of a 'queue' to hold nodes we're about to visit.

Our algorithm uses a technique called 'breadth-first' search. In it, we visit all the neighbours for a vertex before visiting the neighbours themselves, and so on. It's rather as if the search 'fans out' from the original node. The alternative search technique is known as 'depth-first' search, where we explore the nodes by following edges as far as we can go, and then backtrack to follow the edges that we missed.

Back to the breadth-first search. Set the first node's 'path length' to 0. That's the first value we have to store with each node in our array (obviously, the first node is at distance 0 from itself). Add the starting node to the queue. Now, in a loop, continue removing nodes from the queue until we find the target node. For each node we pick off the queue, mark it as 'visited' (to do this we'll need a second value – a true/false indicator – and all nodes should be marked as unvisited at the start). Then add all the nodes immediately reachable from that node to the queue.

However, there's a quick test here: there's no point in adding nodes that have already been visited, since if they are being visited again, the new path length must be longer than before. Also, when you add the neighbour nodes to the queue, set their path length to one more than the current node's path length. You stop when you pick off the target node from the queue. Its path length value will be the shortest distance from the source node to the target node.

Weights and measures

If you also want to output the path taken from A to B, you should record the 'parent' of each node as well as its path length when you queue the node (the parent is the node from which you're following the edge). Once you reach B you can follow the path back to A by following the parent links. (If you want the path in the right order, merely push the nodes on to a stack as you go back to A, and then you can pop them off in the forward order.)

Figure 2 shows this algorithm in action by colouring the edges visited at each step and the contents of the queue at each stage. Step 5 is the interesting one, since the edge marked in blue/grey is not followed (because the node at the end of it has already been visited).

For our application to Bacon numbers, the network may be huge (lots of nodes – sorry, actors – with each actor having many links to other actors), but this algorithm will suffice.

Julian's Bacon number

Fun trivial fact: I have a Bacon number of 2, but not through knowing or being an actor. In the late '80s and very early '90s, I owned a flat in Hammersmith. My next door neighbour on that floor was Martin Campbell, who at the time was an up-and-coming film director, most famous for *Edge of Darkness*, and yet to make a big splash in Hollywood (that would be *GoldenEye* in 1995). I'd look after his flat and car while he was away shooting and he'd thank me by giving me invites to his film premières. In 1988, he made a film called *Criminal Law*, starring Kevin Bacon and Gary Oldman, which gave me my magic number. ■

Route planning

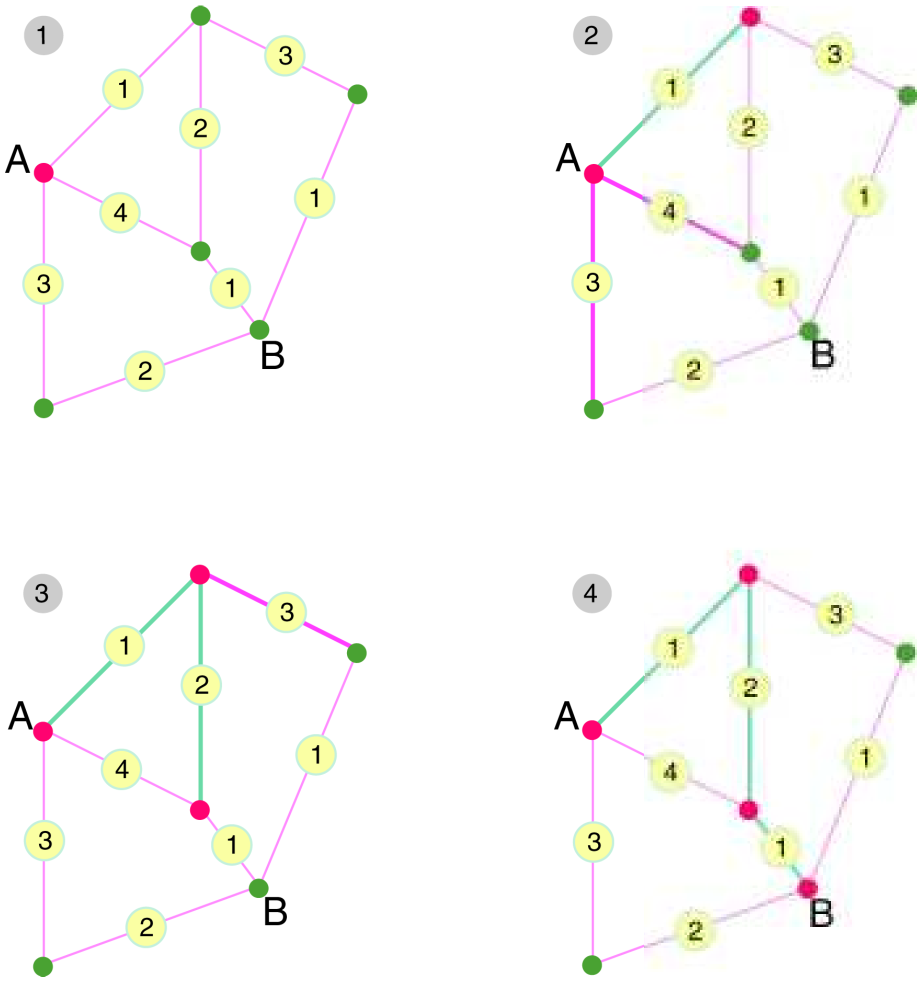
Dijkstra's algorithm is used by modern mapping applications, such as Google Maps, to calculate the best route to go from one place to another. The mapping application uses a sophisticated algorithm for determining such a route, not only the actual distance you'd travel from junction to junction (an edge), but also some qualitative measurement of the speed of travelling along that edge. So motorways, for example, would have a lower cost than A roads, which would in turn be easier to travel than B roads. Traffic lights would have an extra cost, and so on. ■

For the other kind of network where each edge has a weight associated with it, the algorithm changes slightly, although it is still based on the idea of breadth-first search. Here, we're going to find the path that has the smallest cost (that is, the smallest total weight). We could find, for instance, that such a path has more edges than a strict 'shortest' path would have. This variant on the shortest path algorithm was first devised by Edsger Dijkstra (the computer scientist who published the well-known paper called 'Go To Statement Considered Harmful' in 1968 that lambasted the then common and prolific use of the Goto statement in programs).

The big assumption here is that all the weights are positive numbers. The reason for this stipulation is that if an edge has a negative weight, and that edge forms part of a cycle of negative length, you could travel round the cycle ad infinitum to produce shorter and shorter (more and more negative) paths.

Instead of an ordinary queue, Dijkstra's Algorithm makes use of a priority queue. A priority queue is usually described as a queue to hold items from which you always pick off the item with the highest priority (and is so used in operating system job queues or printer queues), but can apply to any set of items from which you want to pick off the smallest or the largest. We'll use the variant, where the next item you remove from the queue at each stage will be the smallest.

We start off as before: set the path length (or cost) to 0 for the first node, and add it to the priority queue. Now we loop over the



▲ **Figure 3: Dijkstra's algorithm, performed step by step.**

queue, picking off nodes until we find the target node (and we will pick off the smallest node at each step). For each node we remove, however, we do some extra work. As before, we have to mark nodes as visited or unvisited (and obviously mark all nodes as unvisited at the start). When we remove a node from the queue, it is marked as visited.

For the current node, we look at each of the nodes reachable from it. If a neighbour node has been visited, we ignore it, much as we did before. For the other neighbouring nodes, we calculate the total cost to reach them from this one. Say the current node has cost 10, and the edge to another node has cost 4. The cost of the next node is therefore 14, through this particular node. Check whether this is less than the current cost of the next node (for this to work we shall have to set the cost of all the nodes – except the first – at the start to some very large number, usually called infinity). If it is, update the cost of the next node with this new smaller value and store the current node as its parent. Note that this will probably put the next node in a new position in the queue.

Completing the network

Once we remove the target node from the priority queue, we'll have, as before, a path from it back to the original node, and we'll also have the cost of that node as accumulated from the source node.

Figure 3 shows an application of Dijkstra's algorithm to a simple network. Step 1 is the original network, and each of the edges is marked with its weight. Step 2, we start at A and find the smallest edge (the one with weight 1). That defines the next node to become our starting point for Step 3. Actually, for Step 4 we have two possible nodes of weight 3: the one in the middle and the very bottom one. For brevity, I chose the middle one so that the algorithm completes with Step 4 at B. The shortest path is the one marked. ■

Spotlight on… Mazes

The depth-first search technique briefly mentioned in the text is also the best way of solving mazes, like the one at Hampton Court. In a maze, the junctions where three or more paths meet are the nodes, and the path between the junctions the edges.

To solve such a maze (when you next happen to be visiting one of the royal palaces and want to show off your inner geek), you enter the maze and then follow the left turn at each junction until you reach a dead end. Backtrack to the previous junction and then take the next leftmost path. Continue like this until you reach the centre of the maze. You will always make it, because the search is exhaustive (as well as exhausting).

If you're able to smuggle in a few stones of a slightly different colour than the gravel in the maze, you'll be able to mark off the paths already explored so you don't make a mistake. You can then easily pick most of them up and palm them as you make your return from the centre directly to the outside – no doubt to the amazement of onlookers. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk