Like A Tattoo

An Introduction To Graphs, Part 1

by Julian Bucknall

Good things come in threes, so they say (or maybe it's bad things come in threes, or maybe it was the Horsemen of the Apocalypse that came in threes until Death joined them, having been laid up with food poisoning). At any rate, the three suggestions for this month's article seemed to come together within a few weeks of each other. The first was in *The Magical Maze* by Ian Stewart, an excellent read on mathematics (at least certain parts of mathematics) which had a chapter on networks and how to solve annoying logical problems. The problems I'm thinking of are those that go like this: you have three jugs of capacity 8, 5 and 3 litres with the largest jug full, and by using all the jugs you have to split the 8 litres of water to get 4 litres in the first and second jug (and no guessing it either!). I've always done these by playing around with numbers, jotting down interim steps on a piece of paper, but Stewart showed how to do it with the Depth First Search algorithm (don't panic, I'll reveal all).

Then I got a message from someone who wanted to do some kind of Gantt chart. A Gantt chart is used in scheduling applications, where you define a bunch of tasks to be done and certain of them can only be started once other tasks have been completed. The query was not about drawing them (he could do that just fine), he wanted to find an algorithm for determining what order the tasks should be done in, given the criteria of *x* tasks, some with certain preconditions, and *y* equally able developers, and then to find the minimum time to complete all the tasks.

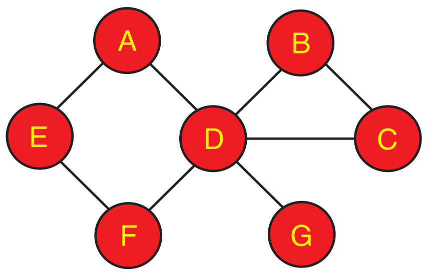
Finally came a message from someone who'd been using my EZDSL data structures library for researching the A* algorithm and wanted some changes to the priority queue to help with it. I'd never heard of the A* algorithm, and he was generous enough to point me in the right direction. And at roughly the same time someone at work started playing *Age of Empires* over the internet after work, in some kind of group death match play. It's quite a fun game (I'm more of an adventure gamer myself, things like Rama and Riven). Basically, you're in charge of a village that you have to grow into an empire. You have cute little workers that you can employ cutting wood or growing food and you manipulate them over a hexagonal map. So you don't have to keep on worrying about them and making sure they do their work, you can point them at where they're supposed to be and they go there under their own steam, as it were. Cleverly, the program ensures that they go round obstacles like rocks or trees; one of the algorithms used for this is the A* algorithm.

So what on earth do all these disparate topics have in common? The answer is *graphs* and that is the topic of this article and the next one.

Feel No Pain

Now, we'd better get one thing clear. A computer science graph is not the same thing as the drawing you do on squared paper to show trends in data, comparisons between different data and so on. In our context a graph is a network of nodes joined by lines. Classically, the nodes are known as *vertices* (we'll use the word *node* though) and the lines between the modes are known as *edges*. Figure 1 is an example graph. You can think of it as a 'network' of nodes that are interconnected. So, in Figure 1, you can go from node A to node D, and from D to A, but you can't go from A to B (at least not without going through D you can't).

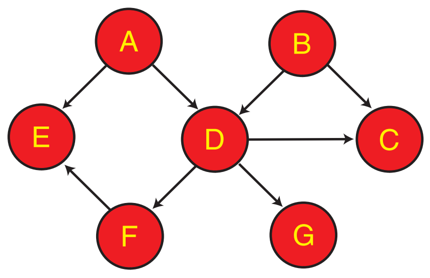
➤ *Figure 1: A simple graph.*



Sometimes the edges are given values. These values usually represent the 'cost' of going from one end of the node to another. For example, if the graph represented a set of cities and the roads between them, then the value for an edge (ie, a road) might be the distance between the cities at either end or the average time it takes to get you from one to the other. If the edges have values like this, the graphs are called *labeled graphs* or *weighted graphs*.

The edges can also be one way only, instead of the two way shown in Figure 1. You can follow a given edge along one way only. The edge has a *direction*. The graphs formed with such edges are known as *directed graphs* or *digraphs* for short. The Gantt chart example I gave above is a digraph where the nodes are tasks to be done, and the edges show what tasks have to be completed before others: obviously the edges have direction because we can't undo a task once it's completed. Anyway, Figure 2 shows a digraph; notice that the edges have arrowheads to denote the direction.

So long as we're just drawing pictures, it all seems pretty easy so far. What we need to do is to get the pretty figures into some kind of



➤ *Figure 2: A digraph.*

	A	B	C	D	E	F	G
A	.	.	.	1	1	.	.
B	.	.	1	1	.	.	.
C	.	1	.	1	.	.	.
D	1	1	1	.	.	1	1
E	1	.	.	.	.	1	.
F	.	.	.	1	1	.	.
G	.	.	.	1	.	.	.

➤ *Table 1: Matrix representation of Figure 1.*

	A	B	C	D	E	F	G
A	.	.	.	1	1	.	.
B	.	.	1	1	.	.	.
C	.	.	.	.	.	.	.
D	.	.	1	.	.	1	1
E	.	.	.	.	.	.	.
F	.	.	.	.	1	.	.
G	.	.	.	.	.	.	.

➤ *Table 2: Matrix representation of Figure 2.*

data structure so that we can start to do things like traversing the graph (ie visiting every node by moving along the edges, and indeed working out whether we can), finding the smallest distance to visit all the nodes (known as the minimum spanning tree), and other important algorithms. Only then will we be able to solve the three examples given at the start.

Paradise

All right then. We have a set of nodes and a set of edges. The simplest data structure we could use (and also the most easily visualized) is a matrix or two-dimensional array. We have a column and a row for every node, and the intersection of the *i*th row and the *j*th column has a value that represents the edge between the *i*th and *j*th nodes. If the graph is simple, the value would be 0 if no edge existed between the nodes and 1 if there was an edge. I'm sure that you can see the matrix would be symmetrical about its leading diagonal in this case: if there is an edge between nodes *i* and *j*, then there's equally an edge between nodes *j* and *i*, it's the same one! For a labeled graph the value at the intersection of a row and a column would be the cost of the edge. For a digraph, the matrix is no longer symmetrical: edges now have direction and if there is an edge between two nodes one way, it doesn't mean that there is one the other way.

In Tables 1 and 2 I've given the matrices for the graphs in Figures 1 and 2. Notice that the matrix in Table 1 is symmetrical and that of Table 2 is not. I've assumed that if there is no edge between nodes, the row-column intersection has the value 0: for some labeled graphs the value 0 may have significance, and we'll have to choose another value.

Before we go any further, it would make sense for us to recognize that we're going to need a class implementation of a graph. That way, if we find a 'better' representation of a graph we can fiddle around with the internals of the class to implement it, but leave the external interface of the class the same. Even better, we create an ancestor class, and then create descendants of it. Things we'll need to do with a graph (for now anyway, we'll come across others in a minute) are as follows. First, create an empty graph for a certain number of nodes. Then possibly set some data in a node. We need to be able to set an edge value and finally, we need to return the edge value between two nodes.

Listing 1 contains a graph class definition that implements the above functionality. I won't deal with expanding a graph by adding more nodes in this article, it's generally fairly obvious how to implement it and it's either trivial and easy, or trivial and long-winded. It would be also convenient to be able to store a graph to a stream and reload it, but that goes beyond the scope of this article.

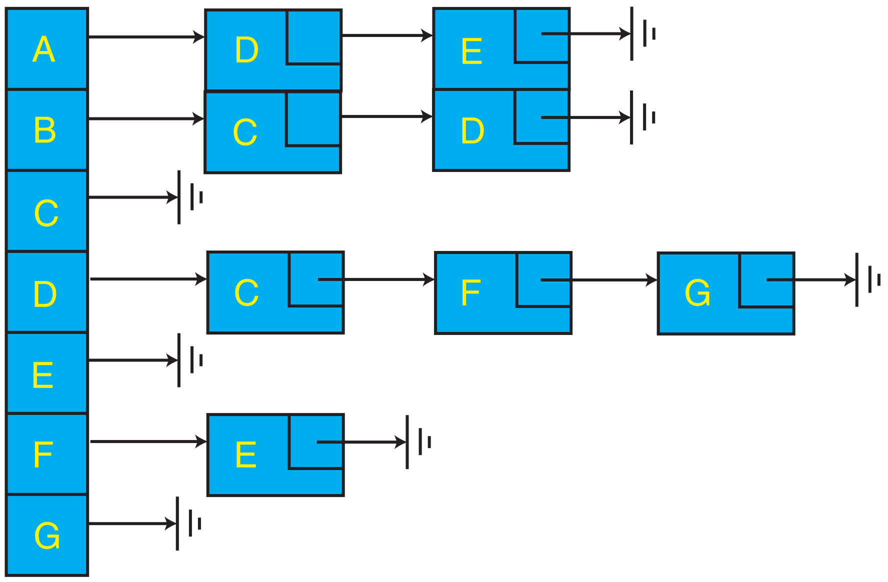
Smooth Operator

Let's consider the first matrix. I'm sure you'll agree that the matrix has a *lot* of redundant information: half the matrix has exactly the same information as the other half, with the line of symmetry being the leading diagonal (from top left to bottom right). We could use a triangular matrix instead: either the top half from the leading diagonal or, just as easily, if not more so, the bottom half.

How do we represent the triangular matrix in memory? I'm sure you can easily code up a rectangular matrix: store the matrix in a linear array and to access the element at row *i* and column *j* you'd actually access array element (*i* * ElementsPerRow + *j*) assuming that rows and columns are counted from 0. This is how the compiler compiles 2-dimensional arrays anyway. The code on this month's disk has a rectangular matrix graph class.

➤ *Listing 1: Ancestor graph class.*

```
TaaGraph = class
  private
    ...
  protected
    function gGetEdge(aFromIndex, aToIndex : integer) : pointer;
      virtual; abstract;
    function gGetNode(aIndex : integer) : pointer; virtual; abstract;
    procedure gSetEdge(aFromIndex, aToIndex : integer;
      aValue : pointer); virtual; abstract;
    procedure gSetNode(aIndex : integer; aValue : pointer); virtual; abstract;
  public
    constructor Create(aNodeCount : integer);
    property Edges[aFromIndex, aToIndex : integer] : pointer
      read gGetEdge write gSetEdge;
    property NodeCount : integer read gNodeCount;
    property Nodes[aIndex : integer] : pointer read gGetNode write gSetNode;
end;
```



➤ *Figure 3: Array of linked lists representation of Figure 2.*

But triangular matrices? Well, we use the same trick, but just make the conversion calculation a little more complex (but not that much so). Let's use the bottom half of the matrix. The first row (row 0) has one element, that is at (0,0), and this is mapped to element 0 of the linear array. The second row has two elements, at (1,0) and (1,1), and these are mapped to elements 1 and 2. The third row has three, at (2,0), (2,1), (2, 2), being mapped to elements 3, 4 and 5. And so on. The formula for the mapping makes use of triangular numbers, 1, 3, 6, 10, 15... (the number of cells in a triangular matrix is a triangular number). The formula for the *n*th triangular number is $n*(n+1)/2$, so for the mapping formula we'll need to calculate the triangular number for the previous row and then add the column number to that.

Remembering that we count from 0 and not 1, the mapping formula to convert (i, j) to an array element is $(i * (i+1) / 2) + j$. In Delphi that converts to:

```
ElementNumber :=
  ((i * succ(i)) div 2) + j;
```

The compiler will convert the `div 2` into a shift-right operation instead, so the expression compiles very neatly. Listing 2 shows a simple implementation of the triangular matrix representation of a graph, with edges represented as a pointer (a four byte entity), we could typecast these pointers to longint or `TObjects`, for example. Notice that we use the symmetrical aspect of the matrix to access (j, i) with $j < i$ by swapping the indexes over and accessing (i, j) instead.

A point to make is that we can save some space with this implementation if we know that there will never be an edge from a node to itself, ie (i, i) is always empty or 0. We can code this special case (row = column) into the access routines and change the mapping formula to:

```
ElementNumber :=
  ((i * pred(i)) div 2) + j;
```

But, in general, saving the extra space is not worth it for the loss of possible functionality in a general class, especially when we consider how empty the matrix is.

Keep Looking

Having got our first main graph class representation, let's sit back and consider the matrices it will be encoding. The first thing to recognize is that a vast majority of matrix elements will be empty: networks we encounter in real life are not that dense. There will not be an edge from every node to every other node (even in our brains' networks, each neuron is 'only' connected to about 10,000 others, compared to the billions that are present). So we are 'wasting' a lot of space in our triangular matrix representation. However, the tradeoff is that it's very fast to access an element. Imagine a 1,000 node graph. The memory required for the linear array is about 2Mb (there'll be 500,500 elements, now you see why including the diagonal makes little difference in practice), and if we propose that every node is attached to two others on the average, there'll be 1,000 edges in use, or 0.2% of the elements. Plus, the triangular matrix representation doesn't help us with digraphs. Time to investigate another representation that will be slower in use but much more space efficient.

Ideally, what we want to do is to just encode the edges that are in use. That's all. No 'empty' cells to allocate memory for or to track. Just the facts, man.

The best way of doing this is to have an array of linked lists. Each element in the array represents a single node, and the linked list from that array element has one element per edge. The edge linked list items hold a node index (ie, to which node is this edge defined?) and an edge value. Figure 3 shows the 'array of linked lists' representation of our original digraph in Figure 2. Note that this representation allows simple graphs, weighted graphs and directed graphs, whereas the triangular matrix version was only good for graphs that aren't directed.

Thinking of implementation details now, it would make sense to have the linked list sorted in node order: this makes it much easier and quicker to determine whether a node was in a list or not. Also, we haven't really talked about the nodes themselves yet, we've been concentrating so hard on the representation of the edges. Presumably we'd have an array of nodes anyway (each of them an object instance, at a guess) and this implementation neatly kills two birds with one stone: each element in the array of nodes has a node instance and a linked list of edges that radiate out from that node.

Listing 3 shows the class that implements the array of linked lists representation of a graph. There are a couple of little clever things to point out in the code. To make the insertion and search for items in each node's linked list of edges easier to code, we use a linked list with a head node and a footer node. The head node uses a node index of -1, which is less than any node index (these start at zero); the footer node uses a node index of `$7FFFFFFF`, which is greater than any node index (this value is 2 billion-odd, and we just can't have that many items in a linked list due to memory restrictions). This means that every edge we add will fit in between these two extremes, making the insertion code much easier to write (and the search for an edge easier too). This kind of trick is usual with linked lists, trading off some extra allocations for simpler and more efficient code.

The next trick we use is to recognize that the header node holds no edge value (it doesn't represent an edge at all, it's just there for our coding convenience) and so we can store the node's value in there (we use a variant record declaration to make the code legible). Apart from these two tricks, the code is fairly simple.

Haunt Me

We've now seen three different representations for a graph: a rectangular matrix (very fast, but large memory requirements), the triangular matrix (very fast, smaller memory requirements, but no good for digraphs) and the linked list approach (slower, much smaller memory requirements, good for any graph). Now it's time to think about what we can do with graphs.

The first thing we might want to try and do is to visit all the nodes, starting at a given node. Before you say 'Well, duh! Julian, you've defined your abstract graph with an array of nodes, so just use that with a for loop' think on a little. Look at Figure 4. This is a graph that consists of two separate sub-graphs. There's no way to go from node A to node Z, the graphs each reside in are not joined together. Figure 2 is another

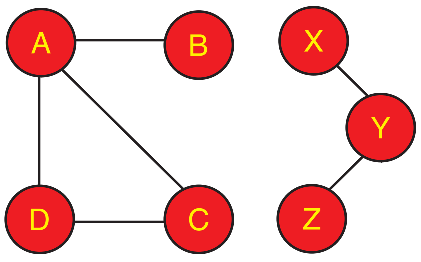
```
constructor TaaTriMatrixGraph.Create(aNodeCount : integer);
begin
  inherited Create(aNodeCount);
  mgNodes := TList.Create;
  mgNodes.Count := aNodeCount;
  mgEdges := TList.Create;
  mgEdges.Count := (aNodeCount * succ(aNodeCount)) div 2;
end;
destructor TaaTriMatrixGraph.Destroy;
begin
  mgEdges.Free;
  mgNodes.Free;
  inherited Destroy;
end;
function TaaTriMatrixGraph.gGetEdge(
  aFromIndex, aToIndex : integer) : pointer;
var Temp : integer;
begin
  {..validation of indexes..}
  if (aFromIndex < aToIndex) then begin
    Temp := aFromIndex;
    aFromIndex := aToIndex;
    aToIndex := Temp;
  end;
  Result :=
    mgEdges[(aFromIndex*succ(aFromIndex)) div 2 + aToIndex];
end;
function TaaTriMatrixGraph.gGetNode(aIndex : integer) :
  pointer;
begin
  {..validation of index..}
  Result := mgNodes[aIndex];
end;
procedure TaaTriMatrixGraph.gSetEdge(aFromIndex, aToIndex :
  integer; aValue : pointer);
var Temp : integer;
begin
  ..validation of indexes..
  if (aFromIndex < aToIndex) then begin
    Temp := aFromIndex;
    aFromIndex := aToIndex;
    aToIndex := Temp;
  end;
  mgEdges[(aFromIndex*succ(aFromIndex)) div 2 + aToIndex] :=
    aValue;
end;
procedure TaaTriMatrixGraph.gSetNode(aIndex : integer;
  aValue : pointer);
begin
  ..validation of index..
  mgNodes[aIndex] := aValue;
end;
```

➤ *Listing 2: Triangular matrix representation.*

example: if you look carefully, you can't get to node B from node A and vice-versa. So the question is a valid one. If, for every node, there is a way to get to every other node, the graph is called *connected*. From the visual representation of a simple graph you can usually tell quickly if it is connected; a digraph is a completely different animal, sometimes it's just not obvious at all. So, we want to be able to start a given node and then visit all of the nodes that are connected to it.

We'll describe the *depth-first traversal* first. This is a systematic (and recursive) way of traversing a graph: you begin at the starting node, 'visit' it (ie do something with it), select an edge from that node and follow it. You continue this process until you reach a dead end (either a node with only one edge, the one you came in on, or a node from which all edges lead to nodes you've already visited). You then back up your path until you find a node with another edge that leads to an unvisited node and continue the process. Eventually, you will visit all nodes (if the graph is connected) or you'll have exhausted all the nodes you can visit, leaving some unvisited (the graph is not connected).

Depth-first traversals are equivalent to preorder traversals of trees. To see this, let's look at the depth-first traversal of the graph in Figure 1. Figure 5 has the details. We can rearrange the graph according to its depth-first traversal into a multiway tree as shown in Figure 6. This tree is known as a *spanning tree*. Preorder traversal means: visit the current node, then recursively preorder traverse the child subtrees, from left to right.

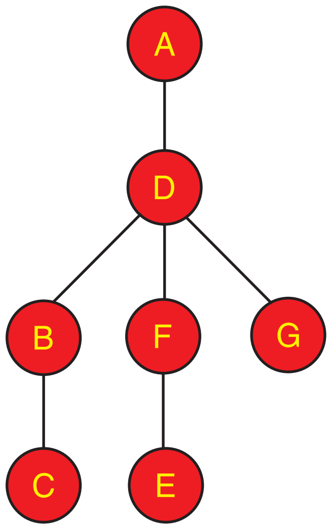


➤ *Figure 4: A graph that is not connected.*

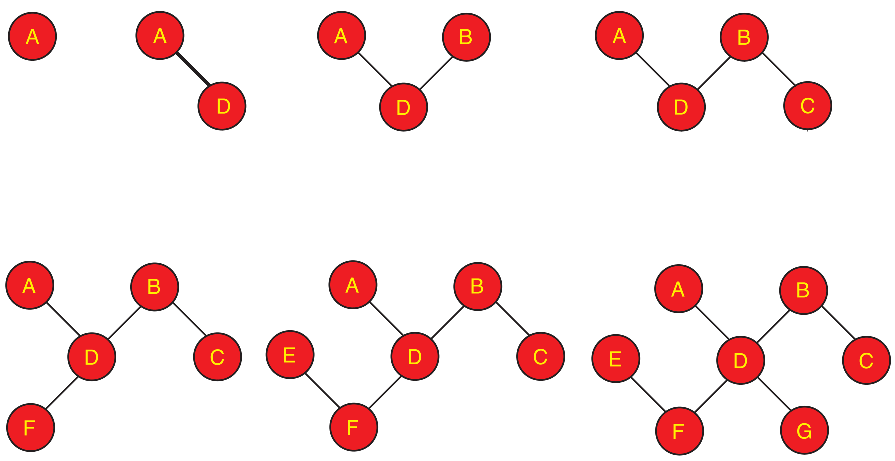
Cherry Pie

Enough of the appetizer theory, let's get to the entrée coding. We can write the depth-first traversal algorithm as pseudo-code first, see Listing 4. From this simple pseudo-code, you see that we have a few extra things to record and take care of before writing the

➤ *Figure 6: Depth-first traversal as spanning tree.*



➤ *Figure 5: Depth-first traversal of a graph.*



```
type
  PllNode = ^TllNode;
  TllNode = packed record
    llnNext    : PllNode; // next node
    llnNodeInx : integer; // node index
    case boolean of
      false : (llnEdge : pointer); // edge value
      true  : (llnNode : pointer); // node value
  end;
constructor TaaLinkListGraph.Create(aNodeCount : integer;
  aIsDigraph : boolean);
var i : integer;
begin
  inherited Create(aNodeCount);
  lgNodes := TList.Create;
  lgNodes.Count := aNodeCount;
  for i := 0 to pred(aNodeCount) do
    lgCreateEmptyLinkedList(i);
  gIsDigraph := aIsDigraph;
end;
destructor TaaLinkListGraph.Destroy;
var i : integer;
begin
  for i := 0 to pred(NodeCount) do
    lgDestroyLinkedList(i);
  lgNodes.Free;
  inherited Destroy;
end;
function TaaLinkListGraph.gGetEdge(aFromIndex, aToIndex :
  integer) : pointer;
var
  WalkNode : PllNode;
begin
  {..validation of index..}
  Result := nil;
  WalkNode := lgNodes[aFromIndex];
  while (WalkNode^.llnNodeInx < aToIndex) do
    WalkNode := WalkNode^.llnNext;
  if (WalkNode^.llnNodeInx = aToIndex) then
    Result := WalkNode^.llnEdge;
end;
function TaaLinkListGraph.gGetNode(aIndex : integer) :
  pointer;
begin
  {..validation of index..}
  Result := PllNode(lgNodes[aIndex])^.llnNode;
end;
procedure TaaLinkListGraph.gSetEdge(aFromIndex, aToIndex :
  integer;
  aValue : pointer);
begin
  {..validation of index..}
  lgSetEdgePrim(aFromIndex, aToIndex, aValue);
  if (not IsDigraph) and (aFromIndex <> aToIndex) then
    lgSetEdgePrim(aToIndex, aFromIndex, aValue);
end;
procedure TaaLinkListGraph.gSetNode(aIndex : integer;
  aValue : pointer);
begin
  {..validation of index..}
  PllNode(lgNodes[aIndex])^.llnNode := aValue;
end;
procedure TaaLinkListGraph.lgCreateEmptyLinkedList(
  aAtIndex : integer);
var
  FirstNode : PllNode;
  LastNode : PllNode;
begin
  New(LastNode);
  with LastNode^ do begin
    llnNext := nil;
    llnEdge := nil;
    llnNodeInx := $7FFFFFFF; // greater than any node index
  end;
  New(FirstNode);
  with FirstNode^ do begin
    llnNext := LastNode;
    llnNode := nil;
    llnNodeInx := -1; // less than any node index
  end;
  lgNodes[aAtIndex] := FirstNode;
end;
procedure TaaLinkListGraph.lgDestroyLinkedList(aAtIndex :
  integer);
var Dad, Son : PllNode;
begin
  Son := lgNodes[aAtIndex];
  while (Son <> nil) do begin
    Dad := Son;
    Son := Dad^.llnNext;
    Dispose(Dad);
  end;
end;
procedure TaaLinkListGraph.lgSetEdgePrim(aFromIndex,
  aToIndex : integer; aValue : pointer);
var DadNode, WalkNode, NewNode : PllNode;
begin
  DadNode := nil;
  WalkNode := lgNodes[aFromIndex];
  while (WalkNode^.llnNodeInx < aToIndex) do begin
    DadNode := WalkNode;
    WalkNode := DadNode^.llnNext;
  end;
  if (WalkNode^.llnNodeInx = aToIndex) then
    WalkNode^.llnEdge := aValue
  else begin
    New(NewNode);
    with NewNode^ do begin
      llnNext := WalkNode;
      llnEdge := aValue;
      llnNodeInx := aToIndex;
    end;
    DadNode^.llnNext := NewNode;
  end;
end;
```

➤ *Listing 3: Array of linked lists representation.*

traversal code itself. The first is obvious: we have to somehow mark a node as being unvisited and visited. The next is that we need to expand our abstract graph class definition to include a method that returns the *n*th edge from a given node.

Let's take the last one first. For the rectangular matrix representation, all we need to do is step along a row in the matrix (the row for the node we're interested in) and count off the edges that exist, ignoring the ones that don't, until we get to the *n*th one. Pretty easy. Similarly for the linked list representation, which is, if anything, even easier: after all there are no non-existent edges in the linked list. (The only gotcha is making sure we don't run off the end of the linked list: to make this easier we could keep a count of edges for each node.) The code on the diskette has the details.

The triangular matrix representation, on the other hand, is awkward. It's got a similar feel to the rectangular matrix but with a virtual half. We could implement a for loop that calculated the linear array index for a given node for each other node and then access the edge value, but that's a little long-winded. Let's be a little cleverer. Consider the indexes we are calculating for node *n*: for all nodes $m <= n$ we calculate $n(n+1)/2 + m$, and for $m > n$ we calculate $m(m+1)/2 + n$. Diagrammatically, we step along a horizontal row until we hit the leading diagonal and then we step down a vertical column. Stepping along the row is simple: we continue adding one to the linear array index until we hit the point on the diagonal ($n = m$), obviously we have to increment *m* to know when we do hit the diagonal. To step down the column, we continue adding $m + 1$ to the linear array index, remembering again to increment *m*. To see why we use

➤ *Listing 4: Depth-first traversal as pseudo-code.*

```
procedure DepthFirst(aNode)
  Visit aNode
  Mark aNode as visited
  for all edges from aNode
    if node linked through edge has not been visited
      DepthFirst(linked node)
```

```
function TaaTriMatrixGraph.GetNodeEdge(
  aFromIndex : integer; aNthEdge : integer;
  var aEdge : pointer; aToIndex : integer) : boolean;
var
  ArrayInx : integer;
  ToIndex  : integer;
begin
  Result := false;
  if (aFromIndex < 0) or
     (aFromIndex >= mgNodes.Count) or
     (aNthEdge < 0) then
    Exit;
  ArrayInx := (aFromIndex * succ(aFromIndex)) div 2;
  ToIndex := 0;
  {first go along horizontally along a row}
  while (ToIndex <= aFromIndex) do begin
    if (mgEdges[ArrayInx] <> nil) then begin
      if (aNthEdge = 0) then begin
        Result := true;
        aEdge := mgEdges[ArrayInx];
        aToIndex := ToIndex;
        Exit;
      end;
      dec(aNthEdge);
    end;
    inc(ToIndex);
    inc(ArrayInx);
  end;
  {then go vertically down a column}
  inc(ArrayInx, pred(ToIndex));
  while (ToIndex < NodeCount) do begin
    if (mgEdges[ArrayInx] <> nil) then begin
      if (aNthEdge = 0) then begin
        Result := true;
        aEdge := mgEdges[ArrayInx];
        aToIndex := ToIndex;
        Exit;
      end;
      dec(aNthEdge);
    end;
    inc(ToIndex);
    inc(ArrayInx, ToIndex);
  end;
end;
```

➤ *Listing 5: Getting the* n*th edge for a node in a triangular matrix.*

m+1, calculate the difference in linear array index between (*n*, *m*) and (*n*, *m*+1) for *m* > *n*. Listing 5 has the GetNodeEdge method for the triangular matrix representation.

Having sorted out the method to get the *n*th edge for a given node, we now have to think about the other problem the pseudo-code offers us: marking nodes that have been visited. We could alter the graph classes again, but this isn't a very attractive proposition. Every time we need a more specialized traversal we'll find that we're going to change our graph representation class to accommodate it, some of these traversals might require information to be gathered during the traversal that's mutually incompatible with other traversals. We'll end up with a graph class for traversal method X, and one for method Y, and so on. Even worse, we'd have to repeat the same code in all our descendants if we're not too careful. And what happens if we want to have a couple of traversals active on the same graph at the same time?

Bullet Proof Soul

No, we need to rethink this design. What we'll do is have a separate class that will perform the depth-first traversal. This class will store a list of nodes already visited and will call a routine we define that will perform the action for each node we visit. The class will store enough state information to efficiently move through the nodes in the graph, *no matter what graph representation we use*. This is a very powerful concept: isolation of the data structure we iterate through from the actual mechanics of the iteration.

The class has a very simple structure: it has a couple of events that will fire for each node visited, a list containing information about each node visited, and an Execute method that will perform the entire depth-first traversal. Two events for each node? Why? Because it provides a more generic solution to our problem. The first

➤ *Listing 6: The depth-first iterator class.*

```
TaaDepthFirstIterator = class
  private
  ...
  protected
  ...
  public
    constructor Create(aGraph : TaaGraph);
    destructor Destroy; override;
    procedure Execute(aFromIndex : integer);
    procedure Reset;
    property OnPreProcess : TaaProcessNode
      read dfiPreProcess write dfiPreProcess;
    property OnPostProcess : TaaProcessNode
      read dfiPostProcess write dfiPostProcess;
end;
```

➤ *Listing 7*

```
procedure TaaDepthFirstIterator.Execute(aFromIndex : integer);
var
  i          : integer;
  NewNodeInx : integer;
  Edge       : pointer;
  OurLevel   : integer;
begin
  // perform preprocessing on the node
  if Assigned(dfiPreProcess) then
    dfiPreProcess(Self, aFromIndex);
  // mark the node as preprocessed
  with PdfiCounter(dfiNodes[aFromIndex])^ do begin
    cMarker := 1;
    OurLevel := cLevel;
  end;
  // iterate through the edges from this node
  i := 0;
  while dfiGraph.GetNodeEdge(aFromIndex, i, Edge, NewNodeInx) do begin
    with PdfiCounter(dfiNodes[NewNodeInx])^ do begin
      if (cMarker = 0) then begin
        cParent := aFromIndex;
        cLevel := succ(OurLevel);
        Execute(NewNodeInx);
      end;
    end;
    inc(i);
  end;
  // perform postprocessing on the node
  if Assigned(dfiPostProcess) then
    dfiPostProcess(Self, aFromIndex);
  // mark the node as postprocessed
  with PdfiCounter(dfiNodes[aFromIndex])^ do begin
    cMarker := 2;
  end;
end;
```

event, the pre-process event, is fired in preorder sequence (that is, before any of the node's edges are followed) and the second, the post-process event, is fired in postorder sequence (that is, after all the node's edges are followed). Usually you would use the pre-process event for a depth-first traversal.

The iterator class (its interface is in Listing 6) stores the following information for each node: whether it's been visited or not, its 'parent' (ie, the node from which it was first visited), the level of the node (the starting node is at level 0, nodes visited from that node are at level 1, nodes from those nodes are at level 2, and so on, this gives us an appreciation of the 'depth' of the traversal).

And the `Execute` method, shown in Listing 7, ties it all together in a recursive manner. The current node is visited for the pre-process phase, marked as such, then all edges are followed and the `Execute` method is called recursively for all unvisited nodes at their ends. Finally the current node is visited for the post-process phase.

As with all recursive routines, we must worry about the depth of recursion. Because graphs are such a freeform kind of data structure, we must consider the worst case: the nodes are strung together like beads on a string. In this case, the depth of recursion would be the same as the number of nodes. This is a little too unbounded for safety's sake (it's safe enough for 10 nodes, but for 1,000 it's getting a little too dodgy for my tastes), so an enhancement we should consider making is to remove the recursion by using an external stack, as I discussed in my *Algorithms Alfresco* column in the July issue.

When Am I Going To Make A Living?

At this point, I must break off and continue with discussions of graphs and graph algorithms next month. The reason? I get married in three days time and if I don't do some wedding preparations soon, my wife-to-be just won't turn up!

Julian Bucknall can be graphic and is fully networked, but is somewhat shady. The code that accompanies this article is freeware and can be used as-is in your own applications.