Join The Dots

An Introduction To Graphs, Part 2

by Julian Bucknall

Before I continue with my introduction to graphs, one of my readers was kind enough to point out that I've become 'unreachable'. For the past few articles, I've neglected to add my email address to the mini-bio at the end of the article. Oops, sorry about that, it wasn't my intention to become incommunicado. You can of course reach me by sending a message to julianb@turbopower.com or to our Esteemed Editor, who will forward it if need be. And please do drop me a line; that includes compliments, brickbats, errors in text or code etc. If your message makes a valid point, I'll be sure to include it in a sidebar in a future article.

Anyway, onwards with graphs, since we have a lot to cover. You may recollect that in the October 1998 instalment of *Algorithms Alfresco*, we talked about the graph structure. We learned that a graph consists of a set of *nodes* (or *vertices*) connected together by *edges*. The edges could be either two way (if there was an edge between node A and node B then there is an edge between B and A, the same one) or the edges were *directed* (if there was an edge from A to B then it does not follow that there is an edge from B to A, we can think of the edge having an arrowhead showing the direction of the edge). The latter kind of graph is known as a *digraph* (or directed graph). Both nodes and edges could contain data (if an edge contains data it is usually known as a *weighted* graph). We provided three different ways of storing a graph in memory: as a matrix, a triangular matrix (not for digraphs though), or an array of linked lists. Finally, we looked at depth-first traversals.

Phew! After that quick résumé, let's move on (of course, if you've forgotten some of the above, please quickly go over October's column and meet us back here).

Topological Sorts

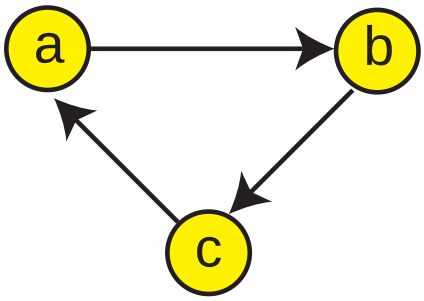
One of the things I didn't really touch on with my previous discussion was a depth-first traversal of all the nodes in unconnected graphs. The depth-first traversal we've seen so far finds all the nodes that can be visited from a given source node. It is entirely possible that some nodes in the graph would *not* be visited by such a traversal. If this is the case, we call the graph *unconnected*. With a non-directed graph it's generally simple to see if a graph is unconnected by looking at its picture: basically the graph falls into two or more sub-graphs with no edges connecting them. For a digraph, in many cases it's not obvious. If we wanted to visit absolutely every node, we'd start out from the first node and do a depth-first traversal on that, then we'd look for a node that hasn't been visited and do a depth-first traversal on that. We'd continue this process until all the nodes had been visited. Listing 1 shows `ExecuteAll`, the implementation of this algorithm.

➤ *Listing 1: Depth-first ExecuteAll method.*

```
procedure TaaDepthFirstIterator.ExecuteAll(
  var aHasCycle  : boolean; aExtraData : pointer);
var
  i : integer;
  ithHasCycle : boolean;
begin
  aHasCycle := false;
  for i := 0 to pred(dfiGraph.NodeCount) do begin
    if (PitrCounter(dfiNodes[i])^.cMarker = 0) then begin
      Execute(i, ithHasCycle, aExtraData);
      aHasCycle := aHasCycle or ithHasCycle;
    end;
  end;
end;
```

Why would we want to do this (ie, visit all nodes in a depth-first fashion)? There is a neat algorithm that falls out from such a depth-first traversal called a topological sort. Suppose we have a set of tasks that need to be done. Certain of these tasks require others to be performed first, in other words they have prerequisites. Examples of this are developing a computer program (we need to design the database before writing the code, we have to decide on the data items we're going to track before we design the database, the install program comes last of all, after all the EXEs and DLLs are finished), devising a recipe for chocolate chip cookies (we can only put the cookies in the oven after we've mixed the cookie dough), getting dressed (we have to put on our underwear first, unless we happen to be Superman) and so on. Translating this into nodes and edges in our graph vernacular: a node is a task, an edge is directed and shows the prerequisite attribute (its *from* node task must be performed before its *to* node task and the arrow points towards the *to* node).

A topological sort organizes the tasks in order so that prerequisites are done before their dependent tasks. An important restriction must be made on the graph first: it must have no cycles; in other words, it must be a directed acyclic graph, also known as a *dag*. What's a cycle? Well a cycle is when you start from a node, visit nodes along directed edges, and get right back where you started from (see Figure 1). If you think in terms of tasks and prerequisites,



➤ *Figure 1:*
A cycle in a directed graph.

this means that if the graph of the tasks is drawn up and contains a cycle, a node could be its own prerequisite. And we can't have that!

Making sure that a digraph is acyclic is easy. We just have to do a depth-first traversal and make sure that the edges we follow never end on a node that is preprocessed. How's that? Think about it. A depth-first traversal works like this: mark the current node as preprocessed, go along each edge from this node and recursively do the same with each node at the end. After we've followed all the edges from the current node, mark the node postprocessed. Suppose we *do* manage to reach a preprocessed node: this would mean that the node is one of our predecessors since a node is only marked postprocessed once all of its children and grandchildren, etc, are visited and marked as postprocessed. Hence we will have discovered a path from a node, through a child node, through a grandchild node, and so on, that reaches itself. A cycle in other words. Notice that in an undirected graph, cycles abound: there is always an edge from a child to a parent since it's the same edge as from the parent to the child! But, we're only talking about directed graphs here. The `Execute` method from the previous article has been enhanced to report whether a cycle was found. This month's disk has the new code.

So how do we do a topological sort on a dag? It's this simple: create a linked list, execute a depth-first traversal, and in the postprocess event for each node, add the node to the front of the linked list. After the traversal is over, the linked list represents the tasks organized into prerequisite order. And that's it. Listing 2 shows a topological sort. What we've done here is, rather than return a linked list with the nodes in topological order, we've created an event and the event will be fired for the nodes in sequence. This in turn means that we don't have to define a linked list external to the depth-first iterator, the implementor of the event handler can do what he likes, he'll just get them in the right order.

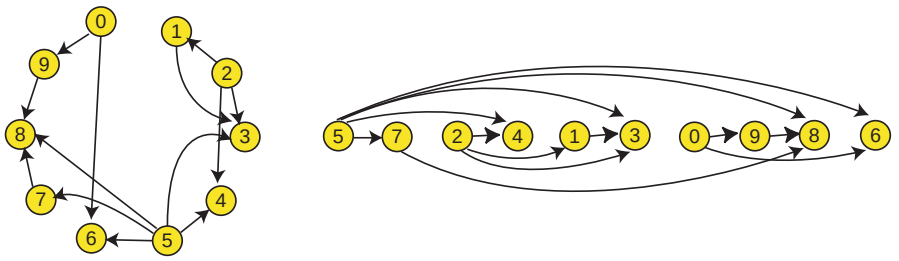
Once we have a topologically sorted dag we can redraw the graph as shown in Figure 2. The nodes (or tasks) are meant to be executed in left to right order. Notice that when we draw in all of the edges from the original dag they all point from left to right. If you think about it, that's a graphic visualization of prerequisites.

Breadth-First Traversals

Another way of walking through a graph is known as breadth-first traversal. With depth-first traversal we tried to go as far as we could from the start node until we hit a dead end and then we backtracked until we found another unvisited edge to follow. This traversal used a stack to aid in the backtracking (actually, if you look back at the depth-first iterator I wrote in the previous article, you'll notice that there isn't an explicit stack, I just used recursion and hence was implicitly using the program stack instead).

Well, with breadth-first traversals, we visit every neighboring node to the start node first, and then visit every neighboring node to those nodes and so on. It's a kind of ripple effect, moving outwards from the original node. It turns out that breadth-first traversals use a queue to store the

➤ *Figure 2:*
Topographically sorting a dag.



➤ *Listing 2: Topographical sort.*

```
procedure TaaDepthFirstIterator.TopologicalSort(
  aExtraData : pointer);
var
  TSList        : PbfiListItem;
  Head, Temp    : PbfiListItem;
  HasCycle      : boolean;
begin
  // create the linked list
  New(TSList);
  TSList^.liNext := nil;
  try
    // now execute the depth first traversal on all the
    // nodes: this will add all the nodes to our linked list
    Reset;
    ExecuteAll(HasCycle, TSList);
    // now trigger the event handler, cleaning up the linked
    // list as we go
    Head := TSList^.liNext;
    try
      // if there was a cycle, the topo sort is meaningless
      if HasCycle then
        raise Exception.Create('digraph is not acyclic');
      // walk the linked list
      while (Head <> nil) do begin
        // process the head node
        if Assigned(dfiProcessSortedNode) then
          dfiProcessSortedNode(
            Self, Head^.liIndex, aExtraData);
        // move down the list, dispose of the old head node
        Temp := Head;
        Head := Head^.liNext;
        Dispose(Temp);
      end;
    except
      // on error clean up the remainder of the linked list
      while (Head <> nil) do begin
        Temp := Head;
        Head := Head^.liNext;
        Dispose(Temp);
      end;
      raise;
    end;
  finally
    Dispose(TSList);
  end;
end;
```

```
procedure TaaBreadthFirstIterator.Execute(
  aFromIndex : integer; aExtraData : pointer);
var
  i          : integer;
  NewNodeInx : integer;
  Edge       : longint;
  OurLevel   : integer;
  OurIndex   : integer;
begin
  // perform preprocessing on the node
  if Assigned(bfiPreProcess) then
    bfiPreProcess(Self, aFromIndex, aExtraData);
  // mark the node as preprocessed
  with PitrCounter(bfiNodes[aFromIndex])^ do begin
    cMarker := 1;
  end;
  // push the node onto the queue
  bfiEnqueue(aFromIndex);
  // whilst there are still items in the queue...
  while not bfiQueueIsEmpty do begin
    // pop the next item off the queue
    OurIndex := bfiDequeue;
    // perform postprocessing on the node
    if Assigned(bfiPostProcess) then
      bfiPostProcess(Self, OurIndex, aExtraData);
    // mark the node as postprocessed
    with PitrCounter(bfiNodes[OurIndex])^ do begin
      cMarker := 2;
      OurLevel := cLevel;
    end;
    // iterate through the edges from this node, push
    // unvisited nodes onto the queue
    i := 0;
    while bfiGraph.GetNodeEdge(
      OurIndex, i, Edge, NewNodeInx) do begin
      with PitrCounter(bfiNodes[NewNodeInx])^ do begin
        if (cMarker = 0) then begin
          // update process information
          cParent := OurIndex;
          cLevel := succ(OurLevel);
          // perform preprocessing on the node
          if Assigned(bfiPreProcess) then
            bfiPreProcess(Self, NewNodeInx, aExtraData);
          // mark the node as preprocessed
          cMarker := 1;
          // push the node onto the queue
          bfiEnqueue(NewNodeInx);
        end;
      end;
      inc(i);
    end;
  end;
end;
```

➤ *Listing 3: The Execute method for the breadth-first iterator class.*

nodes we've visited, instead of a stack.

Again, just like in the depth-first traversal case, we'll define a separate class to perform a breadth-first traversal. This provides isolation of the iterator from the class used to store the graph and hence enables a breadth-first iterator object to work with any graph object descended from our original graph class. The design is a little more tricky than before because we have to code up a queue as well, but it's pretty easy to follow. Like before, we'll have a pre-process and a post-process event so that you have control over when you want to do some work with a node being visited. The interface for the breadth-first iterator is roughly the same as for the depth-first iterator from last time, and the code for the `Execute` method is shown in Listing 3. Figure 3 shows the steps taken in a breadth-first traversal for a sample digraph (it's the one in Figure 2 of October's article, so you can compare the depth-first and breadth-first traversals).

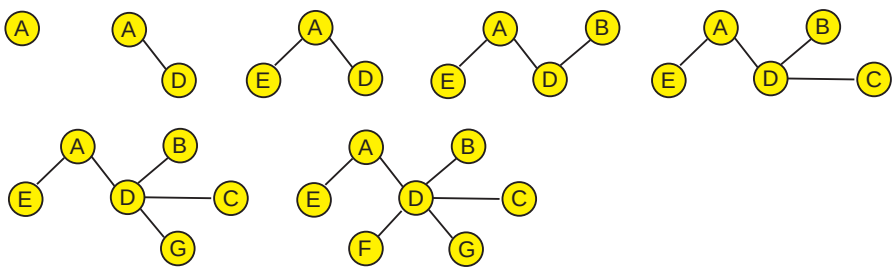
If you look at Listing 3, you'll see that the pre-process event gets fired just as the node gets added to the queue, and the post-process event when the node is popped off the queue. In other words, pre-process is when the node is first 'seen' from the vantage point of its predecessor node, and post-process when the node is actually 'visited' and the edges from it are then followed.

Shortest Path

Another thing to notice about the breadth-first traversal is that it gives you the shortest path from the source node to every other node. By shortest path, I mean the number of edges that have to be followed to get from the source node to a given node (we'll be seeing another definition of shortest path for a weighted graph in a minute). We can see this in a not-very-rigorous fashion by visualizing the breadth-first search as proceeding in 'waves' or 'ripples' from the source nodes. First all the immediate neighbor nodes are visited (their shortest paths are all of length 1), the first wave. Then all their neighbors are visited, the second wave, and their shortest path distances are of length 2. And so on, so forth, rippling outwards.

In fact, if we make note of the parent (or predecessor) node of each node as we traverse the graph, which you'll notice that the iterator does, we can easily print out the shortest path by walking backwards following these backward links. We can use recursion or an explicit stack to print out the path in the correct order (every time we move backwards from a given node we push the parent node onto a stack until we reach the source node; now we pop the nodes off the stack and print them out to give the shortest path). Listing 4 gives the details where we are using the program stack implicitly through recursion. Notice how we first traverse the whole graph from the given index, and then we print out the shortest path to the given index by walking backwards (since nodes are untyped we require an extra routine to print a node: this routine could print the name or identifier of the node, for example). It could very well be that there is no path from the source node to the target node (in which case we'll hit a dead end on a node without a predecessor in our backwards walk), so we ought to report that fact as well. The implementation makes the shortest path routine a

➤ *Figure 3: Breadth-first traversal, step-by-step.*



```
function TaaBreadthFirstIterator.bfiShortPathPrim(
  aFromIndex : integer; aToIndex : integer;
  aExtraData : pointer) : boolean;
var
  Parent : integer;
begin
  if (aFromIndex = aToIndex) then begin
    { we reached source node, print it & return success }
    if Assigned(bfiPrintNode) then
      bfiPrintNode(Self, aToIndex, aExtraData);
    Result := true;
  end else begin
    Parent := PitrCounter(bfiNodes[aToIndex])^.cParent;
    if (Parent = -1) then begin
      {we've hit a dead end-no path back to the source node}
      Result := false;
    end else begin
      {recurse to the parent, if successful print this node}
      if bfiShortPathPrim(aFromIndex, Parent, aExtraData)
        then begin
        if Assigned(bfiPrintNode) then
          bfiPrintNode(Self, aToIndex, aExtraData);
        Result := true;
      end else
        Result := false;
    end;
  end;
end;
function TaaBreadthFirstIterator.ShortestPath(
  aFromIndex : integer; aToIndex : integer;
  aExtraData : pointer) : boolean;
begin
  // first execute the breadth first traversal: this sets up
  // our internal data structure
  Reset;
  Execute(aFromIndex, nil);
  // now traverse from the ToIndex node back to the
  // FromIndex node pushing visited nodes on the stack:
  // we'll then unwind the stack to print the shortest path
  Result :=
    bfiShortPathPrim(aFromIndex, aToIndex, aExtraData);
end;
```

➤ *Listing 4: The unweighted shortest path.*

function: it returns `True` if a path was found, `False` if not.

All right! Time for a breather. Up to this point we've discussed (in depth) graph representations, depth-first traversals and breadth-first traversals. We really ought to move onto some heavyweight algorithms, otherwise our Esteemed Editor will be wondering whether I'm losing touch!

Weighted Graph Algorithms

Let's now consider weighted graphs. If you recall, a weighted graph is a graph where each edge has a weight or a cost associated with it. An obvious example is a map where the cities and towns are the nodes and the roads between them are the edges of a graph. If we denote the 'cost' of an edge as being the average time it would take to travel along that road, we can then start asking questions like which journey between city X and city Y would take the smallest time and through which other cities must we pass? Or, which tour that starts and ends at city X and visits each and every city once would take the shortest time?

If, instead, the cost of an edge was the distance along the road, and we wished to lay our high bandwidth high cost optical cable to connect primary hubs in all of the cities, along which roads should we lay it to minimize the amount of cable?

These types of problems are solvable by using weighted graphs, a couple of important algorithms and the priority queue class from November's *Algorithms Alfresco*.

Before we start we must make a couple of important assumptions. In my original graph class I allowed an edge to be weighted with a typeless pointer. Since our code up to now hasn't had to worry about what data structure these typeless pointers point to, we haven't worried about it too much. Well, now is the time to worry about it. For the algorithms that follow we need to lay down a couple of rules. Rule 1 is that we need to be able to take two edge weights and state whether the first is less than, equal to, or greater than the second. The weights have to be sortable, in other words. Rule 2 is even more strict: given two edge weights, we need to be able to add them together. In other words we need to calculate the total cost of going from node X to node Y and then on to node Z. These two rules pretty well force us to assume that the weight of an edge is numeric. We shall assume that edge weights are of type longint from now on (the code on the disk has been changed to reflect this).

The next assumption is less strict. One of the algorithms we shall meet in a minute (Dijkstra's algorithm) only works if the weights are never negative. We'll see why in a moment, but for now we'll embrace this assumption as well.

So, edge weights are non-negative `longint` values.

Priority-First Traversals

The two traversals we've seen so far are depth-first and breadth-first. The main traversal method for solving weighted graph algorithms is priority-first. The previous two used a stack and a queue respectively; the priority-first traversal uses a priority queue (seems obvious, no?). The traversal works by taking the edge with the largest priority (however that may be defined). Note that I didn't say *cost* or *weight*, but *priority*. We'll see a couple of different ways of calculating the priority in a moment for different algorithms.

What happens in a priority-first traversal is this:

1. Mark the starting node as preprocessed.
2. Compute the priority of the starting node.
3. Insert it into the priority queue.
4. Remove the node with the maximum priority from the queue, mark it as postprocessed. Follow each edge from that node.
5. If an edge leads to a node that hasn't been visited yet, mark the node as preprocessed, calculate its priority and insert it into the priority queue.
6. If the edge leads to a preprocessed node, the node is in the priority queue already, so calculate the new priority of the node (it might have changed because of the different path we've taken), find the node in the priority queue and change the priority if the new value is larger than the old. Update the priority queue, if so.
7. If the priority queue is not empty, return to step 4.

If you think back to November's *Algorithms Alfresco* column for a moment, I deliberately introduced

a priority queue towards the end of the article that allows step 6 to be performed efficiently. Foresight or what?!

Anyway, before I get too self-congratulatory, it seems I'm forgetting something important: *what on earth is the priority value?* Answer: it depends on which algorithm we're talking about. Before you start thinking that your author has finally become way too esoteric and academic and self-referential and maybe it's time to read *One Last Compile*, let's look at the first weighted graph algorithm. This is the one to determine a *minimum spanning tree* (or, How to Visit All the Nodes and Minimize the Cost). This is known as Prim's Algorithm. In our couple of examples a few paragraphs back, this will be the solution to the optical fibre one.

Prim's Algorithm

A spanning tree is a representation of the graph as a multi-way tree, with every node in the graph somewhere in the tree (ie, we make the assumption that the graph is connected), and the links between parents and children in the tree are edges in the graph. There can be several spanning trees for a given graph, it all depends where we start and which edges we travel down first. A minimum spanning tree is the one of these many spanning trees such that the cost of traversing the tree is minimal for the given graph. In other words, if we sum the cost of the edges in the various spanning trees, then the minimum spanning tree is the one whose total cost is the smallest.

Before we look at how to implement Prim's algorithm with a priority-first traversal, and hence define how to calculate the priority, let's see how it works.

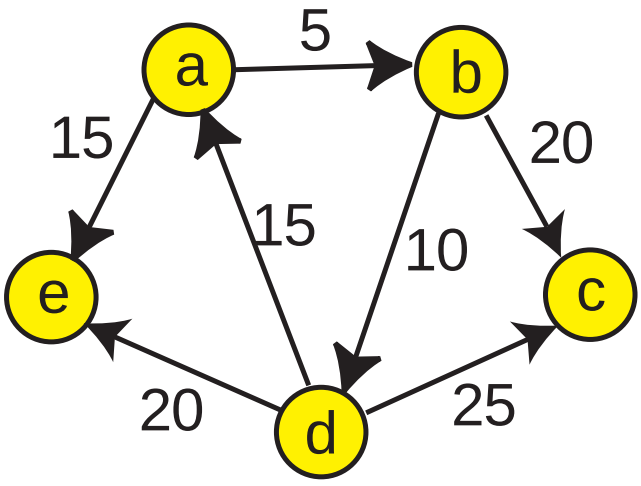
Suppose we have the graph in Figure 4. We want to find the minimum spanning tree starting at node *a* (or, in the vernacular, rooted at *a*). Visit *a*. Select the edge with the smallest cost going from *a* and visit the node at the end. This node is *b* and the cost was 5 units. Now select the edge with the smallest cost going from either *a* or *b*, and visit the node at the end. In our case this is the edge from *b* to *d* with cost 10. Repeat the process. Selecting the next edge reveals a choice of two: from *a* to *e*, or from *d* to *a*. We reject the second alternative because *a* has already been visited, so we visit *e*. Again, selecting the next edge gives us a choice of two: *d* to *e*, and *b* to *c*. We reject the first choice (*e* has been visited before) so we use the second and visit *c*. At this point we can stop since we've visited all the nodes. Figure 5 shows the steps to produce the final minimum spanning tree.

So, to return to our priority-first traversal, what is the priority value of a node? Well, I hope that you can see that it's inversely proportional to the edge cost to visit that node: the smaller the cost to get to the node, the larger the priority of the node. We could use a formula like *VeryLargeValue minus EdgeCost* to calculate the priority. However, a cleverer plan would be to change the priority queue to act on the edge cost directly and return the smallest (that is, to use a min-heap). In fact this is what we will do: we will change the priority queue to a min-heap and, in step 5 above, we'll replace the priority of an already preprocessed node if the new value is *smaller* than the old.

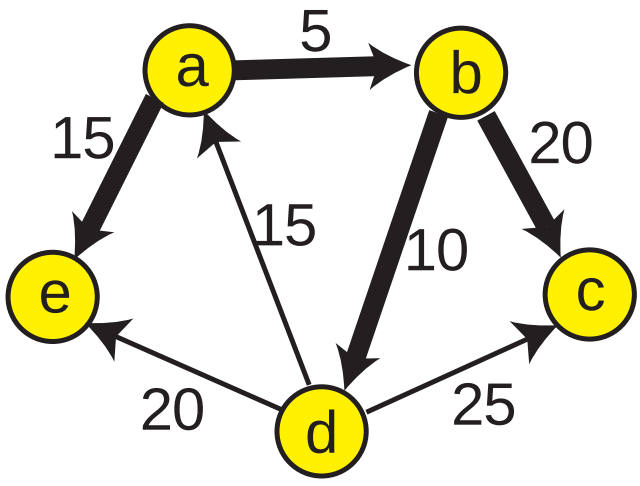
We can now take all of this and produce a priority-first traversal iterator, and then a specialized routine for calculating the minimum spanning tree. The `Execute` method of the iterator is shown in Listing 5. Note that when we store the node in the priority queue, we actually store the internal counter record (the one that takes account of a node's parent, its priority and so on). This enables the priority queue to compare priorities of items (or rather, enables us to easily write a comparison function for the priority queue). I'm sure you can follow the code (supplemented by the files on the diskette) and compare it to the seven-step algorithm above.

So how do we code Prim's algorithm using this iterator? Really simply as Listing 6 shows. Notice the routine to evaluate the priority based on Prim's Algorithm: all we

➤ *Figure 4: A digraph.*



➤ *Figure 5: The minimum spanning tree for the digraph in Figure 4.*



```
procedure TaaPriorityFirstIterator.Execute(
  aEvalPriority : TaaEvalPriority; aFromIndex : integer;
  aExtraData : pointer);
var
  i          : integer;
  NewNodeInx : integer;
  Edge       : longint;
  OurLevel   : integer;
  OurIndex   : integer;
  NewPriority: longint;
begin
  // perform preprocessing on the node
  if Assigned(pfiPreProcess) then
    pfiPreProcess(Self, aFromIndex, aExtraData);
  // mark the node as preprocessed
  with PitrCounter(pfiNodes[aFromIndex])^ do begin
    cMarker := 1;
    cPriority := 0;
    // push the node onto the queue
    cHandle := pfiQueue.Add(pfiNodes[aFromIndex]);
  end;
  // whilst there are still items in the queue...
  while (pfiQueue.Count <> 0) do begin
    // pop the next item off the queue
    OurIndex := PitrCounter(pfiQueue.Remove)^.cIndex;
    // perform postprocessing on the node
    if Assigned(pfiPostProcess) then
      pfiPostProcess(Self, OurIndex, aExtraData);
    // mark the node as postprocessed
    with PitrCounter(pfiNodes[OurIndex])^ do begin
      cMarker := 2;
      OurLevel := cLevel;
    end;
    // iterate through the edges from this node, push
    // unvisited nodes onto the queue
    i := 0;
    while pfiGraph.GetNodeEdge(OurIndex, i, Edge,
      NewNodeInx) do begin
      with PitrCounter(pfiNodes[NewNodeInx])^ do begin
        {totally unvisited before}
        if (cMarker = 0) then begin
          // update process information
          cParent := OurIndex;
          cLevel := succ(OurLevel);
          // perform preprocessing on the node
          if Assigned(pfiPreProcess) then
            pfiPreProcess(Self, NewNodeInx, aExtraData);
          // mark the node as preprocessed
          cMarker := 1;
          // calculate the priority
          cPriority := aEvalPriority(Self, OurIndex, Edge,
            NewNodeInx);
          // push the node onto the queue
          cHandle := pfiQueue.Add(pfiNodes[NewNodeInx]);
        end else if (cMarker = 1) then begin
        {already preprocessed}
          // calculate the new priority
          NewPriority := aEvalPriority(Self, OurIndex, Edge,
            NewNodeInx);
          // if it is less than the current one, update the
          // node and reheapify the queue
          if (NewPriority < cPriority) then begin
            cParent := OurIndex;
            cLevel := succ(OurLevel);
            cPriority := NewPriority;
            pfiQueue.Replace(cHandle, pfiNodes[NewNodeInx]);
          end;
        end;
      end;
      inc(i);
    end;
  end;
end;
```

➤ *Listing 5: The Execute method for the priority-first iterator class.*

do is to return the cost of the edge leading to the node.

Travelling Salesmen Problems

What is so special about Prim's algorithm and minimum spanning trees? Well, they're used in travelling salesmen type problems.

Suppose that a salesman has a set of stores that he must visit in order to demonstrate and sell his company's latest widget. It makes sense for him to minimize the amount of time it takes him to travel between these stores so that he can take the rest of the day off (and indeed to make sure that he has some time to take off!). Using his knowledge of the traffic in his city, where these stores are, the one-way systems, etc, he can draw up a graph with the stores as nodes and the cost of the edge between two nodes the time it would take to go from one store to the other. It is assumed that he can get from any store to any other without passing through a third (the graph representing the map of the stores and roads is said to be *complete*). How does he calculate a tour of the stores that will take the minimum time?

```
function EvalPrimsPriority(aSender : TObject; aFromIndex : integer;
  aEdgeCost : longint; aToIndex : integer) : longint;
begin
  Result := aEdgeCost;
end;
procedure MinSpanningTree(aGraph : TaaGraph; aProcessNode : TaaProcessNode;
  aExtraData : pointer);
var
  Iter : TaaPriorityFirstIterator;
begin
  Iter := TaaPriorityFirstIterator.Create(aGraph);
  Iter.OnPostProcess := aProcessNode;
  try
    Iter.Execute(EvalPrimsPriority, 0, aExtraData);
  finally
    Iter.Free;
  end;
end;
```

➤ *Listing 6: Prim's algorithm.*

Believe it or not, solving this type of problem generically is difficult in the extreme. Actually, let's rephrase that. Although the solution is simple to state (list all the possible tours and select the one with the smallest cost), the problem that arises is the number of possible tours increases exponentially with the number of nodes and edges. In our salesman problem, suppose that there were 5 possible stores and that he could go from any store to any other. He starts off at the factory, has a choice of 5 stores to visit first, then a choice of 4 stores to visit next, and so on until he's visited the last store and then returns to the factory. There are 5! different possible tours he could make, and that's 120. Calculating the cost of each tour is pretty simple and it wouldn't take too long to calculate them all and find the minimum. Suppose now that there are 10 different stores to visit. This time he'd have to find the minimal tour in a set of 10! or 3,628,800 possibilities. This is getting a little unwieldy, but if his boss had given him a PC that could calculate the cost of 100,000 tours a second it'd take about 40 seconds to find the minimum one. If there were 15 stores then there would be 1,307,674,368,000 different tours in all, which would take about 151 days to calculate the minimal one on his little PC. This is just not feasible.

The travelling salesman problem is one of a set of *NP-complete* problems (where NP means non-deterministic polynomial-time). The main characteristic of all these problems is that the running

time of an algorithm to solve the problem increases exponentially with the number of items in the problem. Compare this with sequential search, for example. This simple algorithm for finding an item in an array is linear in terms of execution speed: the time to find a item in 1,000 of them will take about 10 times longer than finding one in 100 of them. Another example: the binary search algorithm's execution time increases proportionally to the log of the number of items, so if it takes 5 units of time to find an item in 32 of them, it'll take 10 units of time to find an item in 1,024 of them (ie, twice as long for the square of the number of items).

When faced with an NP-complete problem, the plan is to try and find a reasonably good approximation to the solution in a reasonable amount of time. In other words, we acknowledge the intractability of finding the real solution and instead settle for a 'pretty good' answer that we can calculate reasonably quickly. In the travelling salesman problem, it turns out that we can get a pretty good minimal tour from a minimum spanning tree. Since Prim's algorithm for obtaining the minimum spanning tree is $O(E * logn)$, where E is the number of edges and n the number of nodes, we can obtain a pretty good answer in much, much less time than getting the correct one. (In fact, without going into the mathematics, we can derive a tour from a minimum spanning tree that costs not more than twice the cost of the minimal tour and in practice, much less.) The derivation of a tour from a minimum spanning tree will have to wait for another time, though.

Dijkstra's Algorithm

Now let's move on to our second algorithm with weighted graphs: finding the path with the lowest cost between two nodes in a graph, also known as Dijkstra's Algorithm.

Often we need to calculate the smallest cost to move from one node (the start) in a graph to another (the destination or target). Suppose our graph represents an airline network between a set of cities. Each link (or edge) between two cities represents a flight, and the 'cost' of the flight is the time taken to get from one city to the other. How can we calculate the minimal time it would take to go from city A to city B?

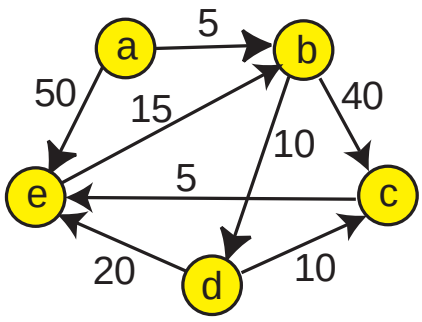
Enter Dijkstra's Algorithm. Like Prim's Algorithm, this uses a priority-first traversal. Actually Dijkstra's Algorithm gives us the shortest path from the source node to every other node (note that in this section by *shortest path* we mean *the path with the overall lowest cost*), and we'll deal with the shortest path between two given nodes in a moment. The algorithm works like this. We walk the graph one node at a time, starting from the start node. As we visit each node, we calculate the minimum path from the start to that node. This is an incremental procedure as we'll see, we don't actually try and determine the shortest path from first principles every time. So which nodes do we choose to visit?

Look at Figure 6. This a weighted digraph and we are going to find out the shortest path from node *a* to node *e*. Remember that by 'shortest path' I don't mean the path with the smallest number of hops, for that would be [*a*, *e*] and this section of the article would be very short. No, I mean the path with the smallest cost. For example, the cost of the path [*a*, *e*] is 50 units; can we do better?

Dijkstra's Algorithm goes like this. Associate a value for each node to represent the cost of getting there from node *a*. Let's store this in an array called `DV` for Dijkstra's value (or maybe distance value). We initialize the array: `DV[a]` is zero, `DV[b]` is 5, `DV[e]` is 50 and the values for nodes *c* and *d* are unknown as yet and so we set them to a large number. (I know we can just look at the graph and work it out, but we're trying to illustrate an algorithm here!) We choose the next node to visit by selecting a node that can be visited from the ones we already have visited, and we select the one with the smallest Dijkstra's value. Well, from the initial setup, we have only visited node *a* (it's where we are starting from after all), and we can only visit *b* or *e*. We choose *b* since it has the smallest value. We now update our `DV` array. From *b* we can visit either *c* or *d*. The distance value for *c*, going through *b*, would be 45 (ie, 5 + 40). If this is less than the current value in `DV[c]` then we replace it (it is, and so we do). The same goes for *d*. At this point the array `DV` contains `[0, 5, 45, 15, 50]`. Start the process over again. We can visit any of the unvisited nodes *c*, *d*, *e* at this point, and we choose the edge that is least costly: the one to *d*. We visit *d* and then update the `DV` array again. This time it'll equal `[0, 5, 25, 15, 35]`; notice how we've gotten 'closer' to both *c* and *e* by this point. We still haven't visited *e* yet and so we continue. The 'nearest' unvisited node this time is *c* so we take that edge, and update `DV` again to `[0, 5, 25, 15, 30]`. The nearest unvisited node is now *e* and so we finally visit it. The 'cost' of getting to *e* from *a* is 30 units, and if we'd saved the shortest paths at each step we'd have noticed that we took the path [*a*, *b*, *d*, *c*, *e*] to get there.

This then is Dijkstra's Algorithm. It can be shown that this 'greedy' algorithm does produce the shortest path from start to target node. In fact, if we let it visit every single node in the graph we can find out the shortest path from a given node to every other. (Notice that the `DV` array in our example stores the cost of going from *a* to every other node since we managed to visit every node in the process.) Although our example doesn't show this, it is entirely possible for the algorithm to hit a dead end and have to backtrack. It doesn't matter though: if the graph is connected we will eventually get to the target node.

Let's cast this in terms of our priority-first traversal. Simple, right? Well, yes. The priority of a node is now its total distance from the start node (or rather the total cost of getting to the node from the start node). All we have to do is alter the priority evaluation function and we're done. Which is what Listing 7



➤ *Figure 6: Work out the minimum cost to go from a to e.*

➤ *Listing 7: Dijkstra's Algorithm.*

```
function EvalDijkstrasPriority(aSender : TObject; aFromIndex : integer;
  aEdgeCost : longint; aToIndex : integer) : longint;
begin
  with (aSender as TaaPriorityFirstIterator) do
    Result := GetPriority(aFromIndex) + aEdgeCost;
end;
procedure SmallestCostPath(aGraph : TaaGraph; aFromIndex : integer; aToIndex :
  integer; aProcessNode : TaaProcessNode; aExtraData : pointer);
var
  Iter : TaaPriorityFirstIterator;
begin
  Iter := TaaPriorityFirstIterator.Create(aGraph);
  Iter.OnPrintNode := aProcessNode;
  try
    Iter.Execute(EvalDijkstrasPriority, aFromIndex, nil);
    Iter.TracePath(aFromIndex, aToIndex, aExtraData);
  finally
    Iter.Free;
  end;
end;
```

does. We do the same trick as we did with the shortest path algorithm with the breadth-first traversal (ie, work out the path from A to B by working backwards from B, pushing the parent nodes onto a stack until we get to A, and then popping them off in order).

All Done For Now

After all that, take a huge pat on your back. We've just gone through some pretty chunky concepts, structures, algorithms and coding. It's taken three articles to do it as well, so if you've managed to stay with me through it all, well done. This has been a learning exercise for me as well, although I vaguely knew what graphs were and had read about the various algorithms we've covered, I'd never tried to code them before.

If you want to know more about graphs, I used three books as research: *Introduction to Algorithms* by Cormen, Leiserson and Rivest; *Practical Algorithms in C++* by Bryan Flamig; and *Data Structures, Algorithms, and Performance* by Derick Wood.

Next time, we'll take it a little easier. Promise *[Thanks! Ed]*.

Julian Bucknall can be pretty graphic, if a little edgy. He can be contacted via email at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.