overload 84

APRIL 2008 £3

Watersheds and Waterfalls

The second part of our investigation into segmenting images into regions

The Way of the Consultant

We consult the oracle for some sound advice (or some sound bites)

The Model Student

A knotty problem...
We get tied up in more mathematical models

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 84

April 2008

ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Alistair McDonald
alistair@inrevo.com

Anthony Williams
anthony.ajw@gmail.com

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Ric Parkin
ric.parkin@ntlworld.com

Simon Farnsworth
simon@farnz.co.uk

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 85 should be submitted to the editor by 1st May 2008 and for Overload 86 by 1st July 2008.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of ACCU
For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

4 Watersheds and Waterfalls
Stuart Golodetz continues his survey of algorithms for segmenting images into regions.

9 The Model Student: A Knotty Problem, Part 1
Richard Harris explores more of the mathematics of modelling problems with computers.

16 The Way of the Consultant
Teedy Deigh offers some observations on how consultants can meet the challenge of achieving effective communication.

Copyrights and Trade Marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

After Four Years

After four years as editor of Overload it is time for a change.

Changing times

Out of curiosity I looked up the first editorial I wrote after taking over the editorship of Overload (the June 2004 issue). By a curious coincidence this editorial started by discussing change: I was optimistic that software development practices are evolving and new ideas are being put into practice. By another coincidence, the previous editorial had – like the one before this one – been written by a guest: Mark Radford, who observed a (less positive) tendency for organisations to try techniques that are known to be ineffective.

After four years one might hope that evidence would be mounting to support my view of a changing industry. I do see wider adoption of good practices – for example I rarely have to 'sell' unit tests or continuous integration. But I also see some of the same bad ideas that have been around since they were debunked in 'The Mythical Man-Month'. (Adding people to a late project? It *still* happens, and it still makes things harder!!)

Why is change so slow? Well we are talking about changes to human behaviour – and that happens very slowly.

Over a long time-scale things definitely do change in the software development world – although it has spent decades trying to model development processes into a model that assumes that correcting errors is so expensive that it justifies elaborate and costly precautions to avoid them. This may have been partially true once – in the 1970s I can remember working in environments where one got one or two attempts to compile a program each day and hand checking for code syntax errors was an essential part of making progress towards actually executing the code. In these circumstances, checking in advance was necessary.

Since then the technology supporting software development has changed. The discipline of hand checking the code became obsolete decades ago, when it became possible to run a compile and get the results faster than checking by hand. It became far more effective to throw the code at the compiler and deal with any diagnostics it produced.

The changes have gone further than that. In a typical modern development environment, syntax errors are highlighted as one types and when the file is saved the code is automatically compiled and tested: the results appear in a moment. This leads to a mode of working where the tests and code are developed in parallel and to some developers making the (unsurprising) observation that if one makes the tests easy to write the corresponding code is easy to use.

Throughout the development cycle the costs of automation have fallen: building and testing an entire system to a point where it can be deployed can be automated and run at intervals ranging from every commit to every day. Automation of deployment is also feasible with a consequent reduction in the cost of a release over extended manual checking and deployment.

Given this, it is hardly surprising that in software-for-use projects there is an increasing emphasis on delivering a partial solution as early as is feasible and regularly making incremental corrections and improvements based on user feedback. While these practices are not yet universal – and are less applicable to software-for-sale projects – the reduction in overall development costs and time to deliver are driving adoption.

Another sign of the times

In the early days of desktop computers a lot of effort went into moving data from one system to another. Even though I was working for a company whose main business was selling furniture, developing software for it involved writing device drivers, file transfer utilities and translators between different data formats and character encodings. I'm sure that mine wasn't the only company incurring costs when dealing with EBCDIC, ASCII line termination, national currency symbols and the like – not really part of the core business. And as the IBM PC became popular there were also IBM's 'extended' ASCII code pages to deal with too.

On the other hand, as PCs became popular file transfer and transformation utilities became readily available. But similar problems popped up in another area office applications (like word processing) – each supplier created its own incompatible format – and the developers of these must have expended considerable effort reverse-engineering each other's formats and writing import and export functions. These worked inconsistently, and if you didn't know what someone else used then the only reliable format was plain text (although, as noted above, there were still issues with the character encoding).

Eventually, one of these application suites (Microsoft Office) established dominance and its developers at least could relax and let the others worry about reverse engineering competitors' products. Nice for them and an extra effort for anyone else wanting to compete in the 'office' market. And that has been the case for some years now.

However, a number of other parties have reason for wanting this to change: other software developers who want to compete in this market; other OS vendors that want to supply desktop systems; customers who want to use alternative products; and organisations that have a need to ensure continued access to documents.

This conflict of interest has been focussed around Microsoft's attempts to get ISO to ratify its 'Office Open XML' standard. Ms. Geraldine Fraser-Moleketi the South Africa Minister of Public Service and Administration recently described [Idlelo] the situation as follows:

Alan Griffiths is an independent software developer who has been using "Agile Methods" since before they were called "Agile", has been using C++ since before there was a standard, has been using Java since before it went server-side and is still interested in learning new stuff. His homepage is http://www.octopull.demon.co.uk and he can be contacted at overload@accu.org

> ...The adoption of open standards by governments is a critical factor in building interoperable information systems which are open, accessible, fair and which reinforce democratic culture and good governance practices... ODF is an open standard developed by a technical committee within the OASIS consortium. The committee represents multiple vendors and Free Software community groups. OASIS submitted the standard to the International Standards Organisation in 2005 and it was adopted as an ISO standard in 2006. South Africa is amongst a growing number of National Governments who have adopted ODF over the past year.
>
> This past year has been marked by a raising in the tension between the traditional incumbent monopoly software players and the rising champions of the Free Software movement in Africa. The flashpoints of conflict have been particularly marked around the development and adoption of open standards and growing concerns about software patents...
>
> It is unfortunate that the leading vendor of office software, which enjoys considerable dominance in the market, chose not to participate and support ODF in its products, but rather to develop its own competing document standard which is now also awaiting judgement in the ISO process. If it is successful, it is difficult to see how consumers will benefit from these two overlapping ISO standards. I would like to appeal to vendors to listen to the demands of consumers as well as Free Software developers. Please work together to produce interoperable document standards. The proliferation of multiple standards in this space is confusing and costly....
>
> An issue which poses a significant threat to the growth of an African software development sector (both Free Software and proprietary) is the recent pressure by certain multinational companies to file software patents in our national and regional patent offices. Whereas open standards and Free Software are intended to be inclusive and encourage fair competition, patents are exclusive and anti-competitive in their nature. Whereas there are some industries in which the temporary monopoly granted by a patent may be justified on the grounds of encouraging innovation, there is no reason to believe that society benefits from such monopolies being granted for computer program 'inventions'. The continued growth in the quantity and quality of Free Software illustrates that such protection is not required to drive innovation in software. Indeed all of the current so-called developed countries built up their considerable software industries in the absence of patent protection for software. For those same countries to insist on patent protection for software now is simply to place protectionist barriers in front of new comers. As the economist, Ha-Joon Chang, observed: having reached the top of the pile themselves they now wish to kick away the ladder.

Between the time I'm writing this and the time you read it, the next round in this conflict will be over: toward the end of March ISO will chose whether to adopted Microsoft's OOXML as a new standard by way of its 'Fast Track' process or not.

As you will know from past editorials, I'm of the opinion that OOXML is not currently fit to be a standard – and my opinion hasn't been changed by the changes voted at the recent Ballot Resolution Meeting. As an illustration of why I feel this way I refer you to Rob Wier's study 'How many defects remain in OOXML?' [Weir]. His conclusion (my emphasis):

> That's as far as I've gone. But this doesn't look good, does it? Not only am I finding numerous errors, these errors appear to be new ones, ones not detected by the NB 5-month review, and as such were not addressed in Geneva. Since I have not come across any error that actually was fixed at the BRM, the current estimate of the defect removal effectiveness of the Fast Track process is < 1/64 or 1.5%. That is the upper bounds. (Confidence interval? I'll need to check on this, but I'm thinking this would be based on standard error of a proportion, where SE=sqrt((p*(1-p))/N)), making our confidence interval 1.5% ± 3%) Of course, this value will need to be adjusted as my study continues. *However, it is starting to look like the Fast Track review was very shallow and that detected only a small percentage of the errors in the DIS.*

The number of errors isn't really surprising given the speed with which this draft standard has been produced – we all know how hard it is to produce accurate technical information. And a standard of this size (it is bigger than SQL) needs a couple of years worth of review – not the few months allowed by the Fast Track process.

Closer to home

The reason I was looking back to my first editorial during this term as editor is that I'm giving up the role again. I've enjoyed my time with the magazine, but I'm no longer moving it forward and it is time that someone else has the opportunity to do something with it.

That someone is Ric Parkin who has been on the editorial team for some time now and, while I disappeared on holiday, edited the October issue last year (so he does know what he's getting into). The team that has been supporting me for the last year remains in place so I'm sure that it is in good hands.

Good luck Ric!

References

[Idlelo] http://www.raffee.co.za/post/29079077

[Weir] http://www.robweir.com/blog/2008/03/how-many-defects-remain-in-ooxml.html

Watersheds and Waterfalls (Part 2)

Stuart Golodetz continues his survey of algorithms for segmenting images into regions.

In my last article [Golodetz], I described a way of segmenting images using the watershed transform and commented that the biggest problem with the results was one of oversegmentation: the image gets divided into too many regions because a region is generated for every regional minimum in the image, regardless of whether it's of any interest to us. The waterfall algorithm [Marcotegui], the subject of this article, is a hierarchical approach which attempts to solve this problem. (Readers may wish to consult the original paper for a more detailed justification of some of the methods involved.).

Gaussian blurring

Before we start looking at the algorithm in detail, though, it's worth observing that there are useful pre-processing steps we can take to reduce the initial number of regional minima before even applying the watershed transform. In particular, it's well worth our time to apply a Gaussian blur to the original image before taking its gradient. (There are other useful things we should do here as well, but this is an area I'm still looking into.)

Gaussian blurring is essentially a form of weighted pixel averaging based on a discrete approximation to the 2D version of the normal distribution. Many of you will doubtless be familiar with the 1D Gaussian from statistics:

$$g_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-x^2}{2\sigma^2}\right)$$

Its 2D version can be obtained by multiplying a 1D Gaussian in the x direction with one in the y direction:

$$G_\sigma(x, y) = g_\sigma(x) \times g_\sigma(y) = \frac{1}{2\pi\sigma^2} exp\left(\frac{-(x^2+y^2)}{2\sigma^2}\right)$$

In 1D, its graph is the familiar bell-shaped curve; in 2D, we get a bell-shaped surface (see Figure 1).

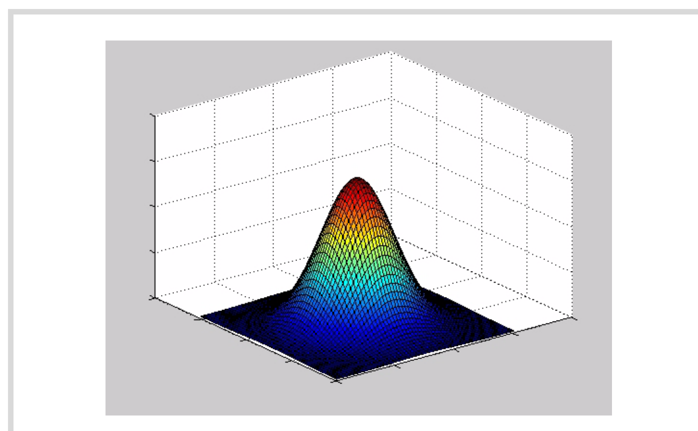

Figure 1

To use this for image blurring, we form a symmetric mask (see Figure 2) from the values of $G_\sigma(x, y)$ at discrete points in a grid centred at the origin (e.g. for a 3x3 mask, we calculate values at (-1,-1), (0,-1), (1,-1), …, (0,0), …, (1,1)). We then normalize the mask by dividing by the sum of all the values in it (this is done to ensure that regions of uniform intensity in the image will be unaffected by smoothing). This procedure can be used to generate masks of other sizes as well.

0.0585	0.0965	0.0585
0.0965	0.1592	0.0965
0.0585	0.0965	0.0585

⟶

0.0751	0.1238	0.0751
0.1238	0.2042	0.1238
0.0751	0.1238	0.0751

Figure 2

As an example (Figure 2), we'll calculate a 3x3 mask for the 2D Gaussian $G_1(x, y)$ (i.e. the Gaussian with standard deviation $\sigma = 1$). First we calculate the values of $G_1(x, y)$ at the grid points (i.e. we calculate $G_1(-1, -1)$, …, $G_1(1, 1)$ to give us the unnormalized mask (left); then, we normalize it by dividing through by the sum of all the values in the mask to give the final result (right).

The actual blurring is done by what is known as convolving the image with the mask. This basically means overlaying the mask on each pixel of the image in turn, multiplying the value of each pixel in the mask by the value of the pixel beneath it, summing the results and using the value thus obtained as the value of the centre pixel in the blurred image. For the 3x3 mask with $\sigma = 1$, this means that if $I(x, y)$ is the source image, $M(x, y)$ is the mask and $I'(x, y)$ is the blurred image, then:

$$\begin{aligned} I'(x,y) = {} & 0.0751 \times (I(x-1,y-1) + I(x+1,y-1) + I(x-1,y+1) + I(x+1,y+1)) + \\ & 0.1238 \times (I(x,y-1) + I(x,y+1) + I(x-1,y) + I(x+1,y)) + \\ & 0.2042 \times I(x,y) \end{aligned}$$

Introducing the Waterfall

Having talked about pre-processing, we can now turn our attentions to the actual waterfall algorithm. The basic idea is to take the result of the watershed transform on the gradient of the original image and use it to produce a sequence of images by merging some adjacent regions (see Figure 3). The waterfall algorithm produces a hierarchical sequence of segmentations, starting from the original watershed result (far left). The final image will eventually be a single region (not shown).

The algorithm described in [Marcotegui] works on the region adjacency graph (RAG) of the watershed result. This is a graph with one vertex for

Stuart Golodetz has been programming for 13 years and is studying for a computing doctorate at Oxford University. His current work is on the automatic segmentation of abdominal CT scans. He can be contacted at stuart.golodetz@comlab.ox.ac.uk

the weights on the edges essentially determine the order of region merging

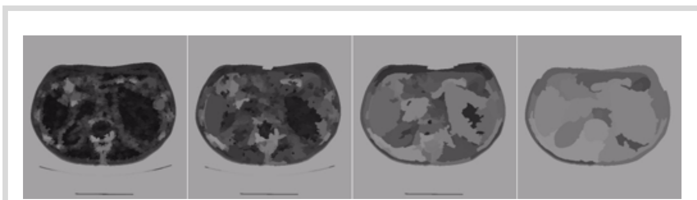

Figure 3

each region in the watershed, and weighted edges joining adjacent regions (see Figure 4). As we will see, the weights on the edges essentially determine the order of region merging, and we have a number of different options when calculating them. For now, we'll assume that we already have a suitably weighted graph, and focus on how to use it to iterate from one stage in the waterfall sequence to the next.

Figure 4 shows a set of regions (left) and their region adjacency graph (right) – note that edges to the surrounding region are not shown to make things clearer.

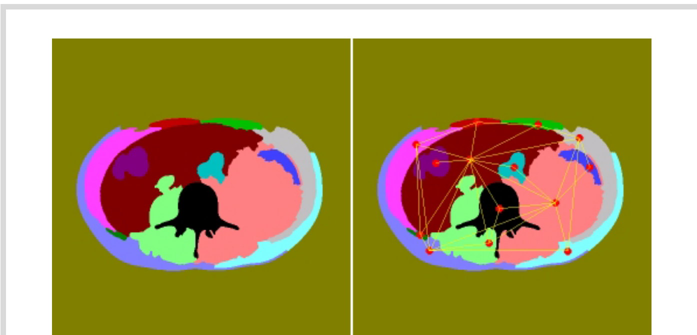

Figure 4

The basic idea of a waterfall iteration involves doing something very like a watershed algorithm on the RAG. First of all, we have to find the regional minima of the graph, which in this case means its regional minimum edges (we'll define what we mean by this shortly). We then mark each such edge with a different label and carefully propagate the labels to the rest of the edges of the graph (this is an implementation of the watershed-from-markers algorithm, where the regional minimum edges are the markers). This induces a new labelling of the various regions, resulting in some of the adjacent regions being merged.

In practice, we don't run the algorithm on the RAG itself; for reasons that are fully explained in the referenced paper, the minimum spanning tree (MST) of the graph contains sufficient information that we can simply run the algorithm on that, with a corresponding gain in efficiency. To briefly recap for those who are unfamiliar with MSTs, they can be defined as follows.

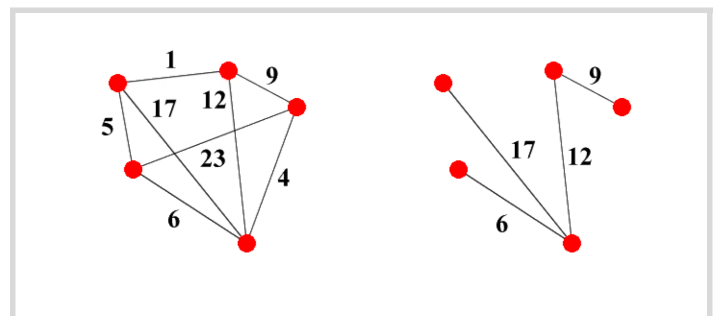

Figure 5

Given a graph $G = (V,E,w)$ with vertex set V, edge set E and weight function $w: E \rightarrow Z^+$, the set of spanning trees $ST(G)$ of G is the set of subgraphs of G which are both trees (i.e. they're acyclic) and which span G (i.e. they contain every vertex in V): see Figure 5 for an example, which shows a graph and one possible spanning tree for it.

A minimum spanning tree is then simply one with a minimum total cost, i.e. a spanning tree $T = (V, E', w) \in ST(G)$ such that

$$\forall (V, E'', w) \in ST(G) \circ \sum_{e' \in E'} w(e') \leq \sum_{e'' \in E''} w(e'')$$

Constructing a minimum spanning tree can be done straightforwardly using Kruskal's algorithm. This involves sorting the edges in the graph into ascending order by weight, then adding the edges in ascending order to the minimum spanning tree provided they wouldn't create a cycle and invalidate the tree.

Data structures

To implement the waterfall efficiently, we're going to have to learn a bit about data structures. One of the key things we need to know about is Tarjan's data structure for disjoint set forests (I mentioned this briefly last time). This structure, designed for maintaining a collection of (mutually) disjoint sets that change over time, is widely useful and not merely restricted to our current purposes. Indeed it's the sort of thing that crops up in Computer Science degree courses [Worrell]! In the previous article and this one alone, this structure gets used during the fletching stage of the watershed algorithm and as part of Kruskal's algorithm, and that's before we've even mentioned its usage in maintaining the regions for the actual waterfall algorithm itself.

The idea, then, is to represent each disjoint set as a rooted tree. Finding which set an element is in is as simple as walking up the tree to the root. Unioning two sets involves finding the roots of two separate trees, and making one tree root a child of the other. For efficiency reasons, it makes sense to keep the paths to the roots of the trees as small as possible. Two tricks used to accomplish this are *union-by-rank* and *path compression*. Without dwelling on the details, the first of these tries to ensure that we're making the root of the smaller tree a child of that of the larger one, and the second changes all the parent pointers on a path to the root to point directly

In the waterfall algorithm, we use such a disjoint set forest to store which regions are connected to each other

```
MAKE-SET(x)
    parent[x] ? x
    rank[x] ? 0

FIND-SET(x)
    if x ? parent[x] then
        parent[x] ? FIND-SET(parent[x])
    return parent[x]

LINK(x,y)
    if rank[x] > rank[y] then
        parent[y] ? x
    else
        parent[x] ? y
        if rank[x] = rank[y] then
            rank[y] = rank[y] + 1

UNION(x,y)
    LINK(FIND-SET(x), FIND-SET(y))
```

Listing 1

to the root when a 'find root' call is made: this ensures that subsequent calls on any element on the path will be constant time. The code to implement all this is shown in Listing 1, which shows Tarjan's Disjoint Set Forest.

In the waterfall algorithm, we use such a disjoint set forest to store which regions are connected to each other. Initially, we have a tree for each region in the watershed result; as we merge regions (see Edge Elision below), we then union their respective trees. This makes region merging a very fast process, since we don't have to update the region indices for all the individual pixels in the regions. Instead, each pixel maintains the label it was originally given by the watershed transform: this can then be used to look up the correct region value in the disjoint set forest associated with any given level of the waterfall. The space savings are also noticeable: instead of storing a full image of labels for each waterfall iteration, we need only store the results of the watershed and a disjoint set forest for each level of the hierarchy.

The other data structures we'll use are for maintaining edges. The layout is as shown in Figure 6. We maintain an array (in practice, a **`std::vector`**) which stores all the edges we'll be referring to (these can either be all the edges in the RAG, or all the edges in the initial MST if we want to be particularly space-efficient). The MST is represented as a list of edge pointers sorted in ascending order of edge weight. Finally, we store an edge adjacency table, which stores lists of pointers to edges which are adjacent to each of the various regions.

Figure 6 shows the data structures used for the waterfall algorithm: some of the pointers aren't shown for reasons of clarity.

Step 1: Finding the Regional Minimum Edges

A regional minimum edge (RME) of a graph *G* is a connected subgraph of *G* whose own edges have equal weight and whose adjacent edges in *G* have strictly higher weights (see Figure 7).

Figure 7 shows an example graph and its RMEs (drawn as triple edges). Note that the two edges with a weight of 1 are part of the same RME.

To find all the regional edges in the MST, we run through all the edges in the MST and flood outwards from each one to determine (a) whether it's part of an RME and (b) the extent of the RME if so (see Listing 2, the flooding algorithm for finding RMEs). We do this by maintaining two things: an equal edges list, which holds all the edges that may form part of the current hypothetical RME, and an adjacent edges queue, which holds

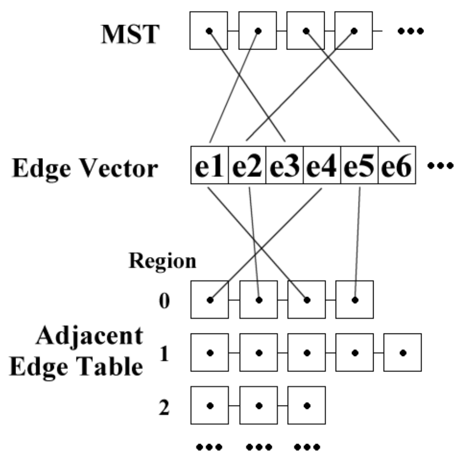

Figure 6

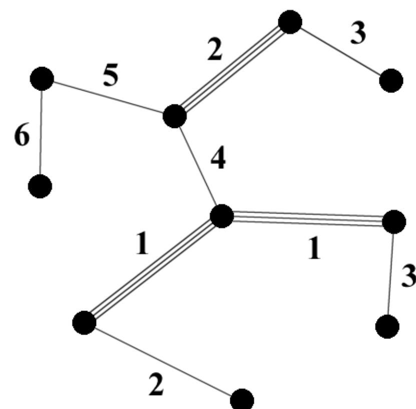

Figure 7

all unprocessed edges which are adjacent to one of the aforementioned equal edges. We process all adjacent edges one at a time. If we see a lower edge, this isn't an RME. If we see an equal edge, we add it to the list of edges which may be in the RME and add its adjacent edges to the queue. If we see a higher edge, we ignore it. If we empty the adjacent edges queue without seeing a lower edge, we've found an RME, so we add it to the list, mark all the edges in it as being part of an existing minimum (to avoid later duplication) and continue from the top with the next MST edge.

Edge elision

In my implementation of the waterfall, no actual relabelling of the regions is done. Instead, the same effect is achieved by merging regions via the mechanism of eliding (i.e. removing) edges in the RAG. The process of edge elision is slightly intricate because all the relevant data structures need to be updated. We need to union the disjoint set forest trees associated with the regions at either end of the edge, we need to splice the edge adjacency lists together (thus adding the list of all the edges adjacent to one of the regions to that of the other) and we need to do various bits of additional housekeeping (see Listing 3, edge elision). One important step is to mark the edge as elided, so that it can be removed from the MST in Step 4 of the algorithm (see below).

Step 2: Eliding the RMEs

Having found the RMEs in Step 1 (above), the next part of the actual algorithm is to elide them. This is done as an alternative to assigning them all the same label (as per the original waterfall description).

Step 3: Marker propagation

Once we've 'labelled' the marker regions by eliding all the RMEs connecting them, the next step is to propagate the markers to the rest of the MST using a flooding process, at each stage processing an adjacent edge with lowest cost. The ideal data structure for this is a priority queue. The algorithm (see Listing 4, marker propagation) works as follows: we initialise the queue with any edges which are adjacent to RMEs and mark the regions joined by the RMEs as already processed. We then repeatedly pop the lowest cost edge from the queue, merging the regions it connects if one of them is unmarked, and ignoring it otherwise. If an edge is elided, its own adjacent edges are also added to the queue.

```
Vector<EdgeList> rmeArray;

foreach(Edge startEdge   mst)
  // The edge is already part of a regional
  // minimum edge. If we processed it, we'd
  // end up duplicating that minimum edge,
  // so skip over it.
  if(startEdge->minimum) continue;

  // Maintain a list of equally-valued edges
  // which may form part of the same RME.
  EdgeList equalEdges;
  EdgeQueue adjacentEdges;

  // Maintain a queue of pending edges so they
  // can be unmarked again later.
  EdgeQueue pendingEdges;

  // Mark the initial edge to ensure it isn't
  // wrongly identified as being adjacent to
  // itself.
  startEdge->pending = true;
  pendingEdges.push(startEdge);

  add_adjacent_edges(startEdge,
                     adjacentEdges,
                     pendingEdges);

  bool isMinimum = true;
  while(!adjacentEdges.empty())
    Edge adjacent = adjacentEdges.pop();
    if(adjacent->value < startEdge->value)
      // If its value is less than the start
      // edge, then the start edge is not a
      // minimum.
      isMinimum = false;
      break;
    elseif(adjacent->value == startEdge->value)
      equalEdges.push_back(adjacent);
      add_adjacent_edges(adjacent,
                         adjacentEdges,
                         pendingEdges);

  // Unmark all the pending edges.
  while(!pendingEdges.empty())
    Edge e = pendingEdges.pop();
    e->pending = false;

  if(isMinimum)
    // Add the start edge to the list now that
    // we know we've found a minimum.
    equalEdges.push_front(startEdge);
    for(Edge e   equalEdges)
      e->minimum = true;
    rmeArray.push_back(equalEdges);
```

Listing 2

```
ELIDE-EDGE(e)

unsigned int u = e->u;
unsigned int v = e->v;

unsigned int setU = forest.find_set(u);
unsigned int setV = forest.find_set(v);

forest.union_nodes(u, v);

// Mark the edge as elided for when we come to
// later rebuild the MST.
e->value = -1;

// forest.find_set(u) == forest.find_set(v)
unsigned int parent = forest.find_set(u);

// Add all the edges adjacent to the child
// regions to the parent region in the
// adjacency table.
EdgeList parentList = adjacencyTable[parent];
if(setU != parent)
  parentList.splice(parentList.end(),
                    adjacencyTable[setU]);
if(setV != parent)
  parentList.splice(parentList.end(),
                    adjacencyTable[setV]);

// Remove the elided edge from the parent list.
parentList.remove(e);

// Update the region values.
Value parentValue = forest.value_of(parent);
Value uValue = forest.value_of(setU);
Value vValue = forest.value_of(setV);
if(uValue > parentValue) parentValue = uValue;
if(vValue > parentValue) parentValue = vValue;
```

Listing 3

Step 4: Rebuilding the MST

The final step of the algorithm is to rebuild the MST so that it's ready for the next waterfall iteration. This turns out to be an almost trivial process, since we just have to remove any elided edges from the tree. Since we carefully marked them all with a value of -1 when they were elided, all we have to do is run through the list representing the MST and remove any edges whose value is -1 (in C++, this can be handled extremely simply using a `remove_if` call).

Edge valuations

One issue we haven't yet touched on is how to generate suitable weights for the edges of the region adjacency graph. Since these weights determine the order of region merging, they have a substantial effect on the output of the whole algorithm, so it's important to choose them carefully.

There are a number of different options available. The simplest approach, known as lowest pass, valuates each edge with the height of the lowest pass point on the border between the regions it joins. One advantage of this method is that it's easy to calculate: you just run through all the pixels in the watershed result, find any border pixels (pixels which have at least one neighbour with a different label) and update the lowest pass between any two regions as necessary.

Lowest pass isn't always the best method to use, however, and several other sensible valuations have been proposed. These generally focus on the idea of dynamics, which involves thinking about either the height (contrast dynamics), the surface area (area dynamics) or the volume (volume dynamics) of the water in the catchment basins of the adjacent regions at the point when they would meet during the flooding process (i.e. the point where a watershed is built to keep them apart). These all produce different results and some experimentation is needed to see which may be the most appropriate in a given situation. Another interesting valuation can be found in [Climent], where the authors use some knowledge about the human perception of shapes to define a dynamic which (on their test images at any rate) produces more visually-pleasing results than (in particular) the volume dynamic, with which they contrast it.

My own work is currently using lowest pass, but I plan to experiment further with the other valuations in due course.

Conclusion

From my own experiments with the waterfall algorithm, I can attest to the fact that the results of waterfall segmentation seem to be much more useful than the original watershed result. It's not that any particular image in the waterfall sequence gives us the ideal segmentation: that would be far too easy. What the waterfall does give us is a lot of connected regions (in the various different images in the sequence) to work with further. With these results, we can go on to use region analysis and classification strategies to identify which regions in the various waterfall iterations correspond to features of interest in our original image. Waterfall thus takes us one step closer to our original goal of automatic segmentation. How to do the actual region analysis and classification is still a research problem, but one I hope to continue working on in the near future.

Till next time... ■

References

[Climent] 'Visually Significant Dynamics for Watershed Segmentation', Joan Climent and Alberto Sanfeliu. In *Proceedings of the 18th International Conference on Pattern Recognition* (ICPR '06), 2006.

[Golodetz] 'Watersheds and Waterfalls (Part 1)', Stuart Golodetz, *Overload* 83, February 2008.

[Marcotegui] 'Fast Implementation of Waterfall Based on Graphs', Beatriz Marcotegui and Serge Beucher. In *Mathematical Morphology: 40 Years On*, Springer Netherlands, 2005.

[Worrell] 'Data Structures for Disjoint Sets', James Worrell. Advanced Data Structures and Algorithms course notes, Oxford University, 2006.


```
// Flags indicating whether a region has been
// marked or not.
Vector<char> markedRegion(forest.node_count());

PriorityQueue<Edge> pq;
EdgeQueue pendingEdges;

// Add all the edges adjacent to RMEs to the
// priority queue.
foreach(EdgeList rme   rmeArray)
  foreach(Edge e   rme)
    add_adjacent_edges(e, pq, pendingEdges);
    int setU = forest.find_set(e->u);
    int setV = forest.find_set(e->v);
    markedRegion[setU] = 1;
    markedRegion[setV] = 1;

// Process each edge in ascending order of
// value, adding adjacent edges to the priority
// queue each time.
while(!pq.empty())
  Edge e = pq.pop();

  int setU = forest.find_set(e->u);
  int setV = forest.find_set(e->v);

  // If this region connects a marked region to
  // an unmarked one, it needs processing.
  // Otherwise it should be ignored. Note that
  // because of the way the propagation works,
  // any edge will either connect a marked
  // region to an unmarked one, or it will
  // connect two marked regions.
  if(!markedRegion[setU] || !markedRegion[setV])
    ELIDE-EDGE(e);
    markedRegion[setU] = 1;
    markedRegion[setV] = 1;
    add_adjacent_edges(e, pq, pendingEdges);

// Unmark all the pending edges.
while(!pendingEdges.empty())
  Edge e = pendingEdges.pop();
  e->pending = false;
```

Listing 4

The Model Student: A Knotty Problem, Part 1

Richard Harris explores more of the mathematics of modelling problems with computers.

If there's one thing that's guaranteed to irritate me, it's headphones. I don't mean the continual tinny noise pollution that thoughtless public transport patrons inevitably inflict upon their unfortunate fellow passengers; I'm far too busy inflicting my own tinny noise pollution on them to pay it any heed. No, I refer instead to their annoying tendency to tie themselves up in knots at every conceivable opportunity. As foolish as it is to anthropomorphise, I can't help but suspect that they are possessed of a demonic nature; that they will not rest until they have damned all of humanity to an eternal tortured state of minor inconvenience.

It's not just headphones either. If anything, fairy lights are even *more* belligerent. It seems that no matter how carefully I pack them away with the other Christmas decorations they will, 11 months or so later, have contrived to rearrange themselves into a tangle of Gordian complexity. Thus far I have resisted the temptation to assume the mantle of Alexander and take a pair of scissors to the blasted things; I suppose that it's only a matter of time before I succumb.

I am not alone in my frustration. Jerome K. Jerome [Jerome89] made the same observation of tow-lines as long ago as 1889:

> There may be tow-lines that are a credit to their profession – conscientious, respectable tow-lines – tow-lines that do not imagine that they are crotchet-work, and try to knit themselves up into antimacassars the instant they are left to themselves. I say there may be such tow-lines; I sincerely hope there are. But I have not met with them.

So are I and my illustrious forebears suffering from an overactive imagination or do we really live in a universe in which strings and cables spontaneously tie themselves into knots?

Believe it or not, there is an entire field of mathematics dedicated to the study and classification of knots that does, in fact, have something to say on the matter (Brian Hayes [Hayes] has an interesting discussion on the subject of random closed knots). Unfortunately, it's an extremely difficult subject. So much so that even determining whether or not two knots are equivalent is still an unsolved problem.

Whilst knot theory would certainly shed a great deal of light upon the subject, I'm afraid I might sprain something trying to understand it. Instead, I shall propose a simpler model which, with luck, won't cause me any permanent injury.

That model is a random walk.

Generally speaking, a random walk is a sequence generated by a series of random steps from some given starting position. In this case, we will model a string as a chain of finitely sized inflexible links. We can divide the two dimensional surface onto which we lower it, a table for example, into a discrete grid. We can then hypothesise that each link will occupy the location of the previous link or of one of its eight neighbours with equal probability, as illustrated in figure 1 (which shows the candidate steps in a two dimensional random walk). Figure 2 illustrates one such walk (a 9-step random walk).

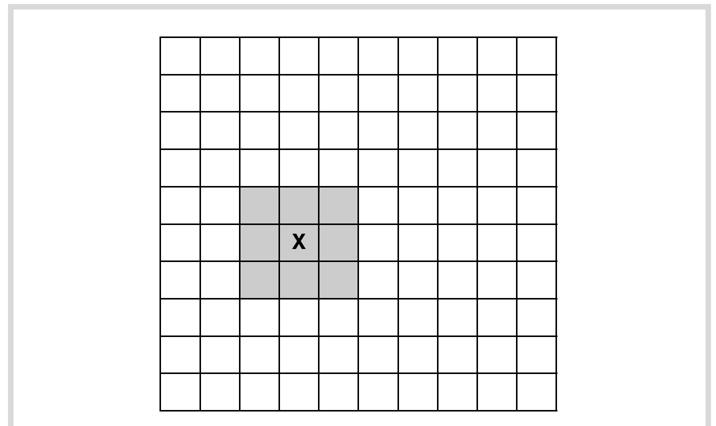

Figure 1

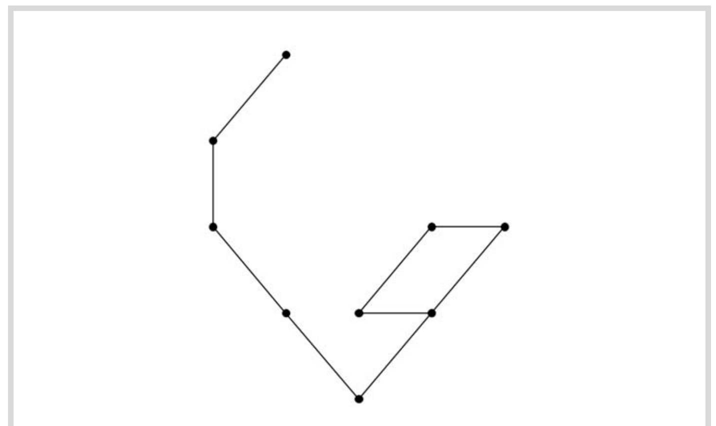

Figure 2

The properties of random walks have been studied for a great many years since they are, in the continuous limit, the mathematical model for Brownian motion. This is the random motion of tiny particles suspended in liquid or gas named after Robert Brown who observed it for pollen grains floating in water in 1827 [Brown28]. Some fifty years later, the mathematics of Brownian motion was described by Thorvald Thiele [Thiele80]. Since then it has been a subject of great interest to mathematicians and physicists, and more recently financiers. The latter has grown into a lucrative field since there are strong theoretical arguments that share prices exhibit a form of Brownian motion [Bachelier00, Hull05].

The relationship between random walks and knots has not escaped the attention of mathematicians. Nechaev [Nechaev98] provides a thorough review of the current state of the art, a state which I must admit is sufficiently advanced that I am reticent to tackle it.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

each time the cable crosses itself it is presented with an opportunity to tangle

So how should an amateur like me relate random walks to the question at hand? Well, each time the cable crosses itself it is presented with an opportunity to tangle. In my experience cables are loathe to pass up such opportunities lightly, so the number of self crossings should hopefully give some insight into their tendency to become knotted.

The obvious way to measure the number of self crossings is to generate every possible walk of a given length and simply count the number of times each of them crossed itself, or other words returns to a point it has already visited. Equally obvious is that this will be extremely computationally expensive. Since each step has nine possible outcomes, a walk of n steps will have a total of 9^n distinct outcomes.

Nevertheless, it's worth pursuing since it leads to some interesting C++.

Before we start we'll need some classes to manage the random walks and keep track of the number of crossings. Let's start with a class to represent a position on the planar grid. Listing 1 shows a class to represent a position in a random walk.

The important features of this class are that it supports moving from one position to another with the **move** member function and that it defines a strict weak ordering with **operator<**, making it possible to use it as a key in a **std::set** or **std::map**.

```
namespace knots
{
  class position
  {
  public:
    position();
    position(long x, long y);
    position move(long dx, long dy) const;
    bool     operator<(const position &rhs) const;
    bool     operator==(const position &rhs) const;
  private:
    long x_;
    long y_;
  };
}
```

Listing 1

```
knots::position
knots::position::move(long dx, long dy) const
{
  return position(x_+dx, y_+dy);
}
bool
knots::position::operator<(
   const position &rhs) const
{
  return x_<rhs.x_ || (x_==rhs.x_ && y_<rhs.y_);
}
```

Listing 2

Their implementation is fairly simple, the only trap being ensuring that **operator<** is a strict weak ordering. We can ensure this by making it a lexicographical comparison. Note that this has no real mathematical meaning and serves only to make **position** compatible with associative containers. Listing 2 shows an implementation of **move** and comparison.

Once again, we'll need a histogram to keep track of the number of self crossings of the random walks. The histogram shall assume that every self crossing results in a knot, thus measuring the worst possible outcome. Apart from some of the names, this is pretty much identical to the one we used for the travelling salesman problem.

Listing 3 shows a class to maintain a histogram of knots. Again, most of the member functions are straightforward. The constructors are amongst

```
namespace knots
{
  class knot_histogram
  {
    public:
    struct value_type
    {
      double knots;
      size_t count;
      value_type();
      value_type(double k, size_t c);
    };
    typedef std::vector<value_type>
       histogram_type;
    typedef histogram_type::const_iterator
       const_iterator;
    typedef const value_type &
       const_reference;
    typedef histogram_type::size_type
       size_type;
    knot_histogram();
    explicit knot_histogram(size_t length);
    knot_histogram(size_type buckets,
                   size_t knots_per_bucket);
    bool      empty() const;
    size_type size() const;
    size_type walk_length() const;
    const_iterator begin() const;
    const_iterator end() const;
    const_reference operator[](size_type i) const;
    const_reference at(size_type i) const;
    void add(size_t knots);
  private:
    void init();
    size_type      knots_per_bucket_;
    histogram_type histogram_;
  };
}
```

Listing 3

We need only divide the number of knots by the knots per bucket to identify the correct bucket

```
knots::knot_histogram::knot_histogram()
{
}

knots::knot_histogram::knot_histogram(
   size_t length) : knots_per_bucket_(1),
                    histogram_(length)
{
  init();
}
knots::knot_histogram::knot_histogram(
   size_type buckets, size_t knots_per_bucket) :
   knots_per_bucket_(knots_per_bucket),
                     histogram_(buckets)
{
  init();
}
```

Listing 4

those that require some care. Listing 4 shows construction of the knot histogram.

By default we use one bucket per step of the walk since we know that the maximum possible number of self crossings results from every step staying in the same cell. As before, the **init** member function initialises the buckets (Listing 5).

This time it's slightly simpler since we've forced the user to pass in the number of knots per bucket in the histogram. Note that we're using the proportion of the walk that's knotted, rather than the absolute number of

```
void
knots::knot_histogram::init()
{
  if(empty()) throw std::invalid_argument("");

  histogram_type::iterator first =
     histogram_.begin();
  histogram_type::iterator last  =
     histogram_.end();
  size_t knots = 0;
  --last;

  while(first!=last)
  {
    knots += knots_per_bucket_;
    *first++ = value_type(
       double(knots)/double(walk_length()), 0);
  }

  *first = value_type(1.0, 0);
}
```

Listing 5

```
void
knots::knot_histogram::add(size_t knots)
{
  knots /= knots_per_bucket_;
  if(knots<size())   histogram_[knots].count += 1;
}
```

Listing 6

knots, as the value for the bucket. The value represents the upper bound for the bucket in question, with the lower bound being the value of the previous bucket, or zero for the first.

Once again we'll be exploiting the fact that the buckets are evenly distributed over the unit range to simplify adding a knot count to the histogram. We need only divide the number of knots by the knots per bucket to identify the correct bucket.

If we were to do this alone, however, we would have a problem with the walk in which each of the *n* steps is to remain in the same location. This would have *n* crossings which would lead us to attempt to access the bucket after the last in the histogram. We therefore make a slight approximation and ignore that walk. Naturally this introduces an error, but since it is but one walk out of 9^n it shouldn't have a significant impact on the results.

Listing 6 shows adding a walk to the histogram. Note that we can recover the length of the walk by simply multiplying the number of buckets by the number of knots per bucket:

```
knots::knot_histogram::size_type
knots::knot_histogram::walk_length() const
{
  return size() * knots_per_bucket_;
}
```

Finally, we need a way to represent a random walk and to count the number of self crossings. Observing that each step has nine possible outcomes of equal probability, we can trivially represent a walk with a **std::vector<unsigned char>** element of which contains a value in the range zero up to and including eight.

In addition to a **typedef** formalising this definition of a random walk, the following includes the declaration for a function to count the number of crossings:

```
namespace knots
{
  typedef std::vector<unsigned char> walk;
  size_t crossings(const walk &w);
}
```

The implementation of **crossings** needs to iterate through each **position** in the walk and check whether or not it has already been visited. Since we went out of our way to make **position** compatible with **std::set**, it seems to be a natural choice to keep track of the visited **position**s.

Whilst recursion can greatly simplify the expression of these kinds of operations, it can, in some situations at least, be less efficient than iteration

```
size_t
knots::crossings(const walk &w)
{
  size_t knots = 0;
  position p(0, 0);
  std::set<position> visited;

  walk::const_iterator first = w.begin();
  walk::const_iterator last  = w.end();

  visited.insert(p);

  while(first!=last)
  {
    if(*first>8) throw std::invalid_argument("");

    long dx = long(*first)%3 - 1;
    long dy = long(*first)/3 - 1;

    p = p.move(dx, dy);
    if(!visited.insert(p).second) ++knots;
    ++first;
  }

  return knots;
}
```

Listing 7

Listing 7 shows counting the self crossings.

The principal trick we're exploiting to update the **`position`** as we iterate through the **`walk`** is the use of integer division and modulus to generate the nine distinct steps. At the risk of labouring the point, figure 3 (mapping the step id to the offsets) illustrates how this works.

step	step%3-1	step/3-1
0	-1	-1
1	0	-1
2	1	-1
3	-1	0
4	0	0
5	1	0
6	-1	1
7	0	1
8	1	1

Figure 3

```
void
knots::full_crossings(
   walk &w, knot_histogram &h, size_t n)
{
  if(n==w.size())
  {
    h.add(crossings(w));
  }
  else
  {
    for(size_t i=0;i!=9;++i)
    {
      w[n] = i;
      full_crossings(w, h, n+1);
    }
  }
}

void
knots::full_crossings(knot_histogram &h)
{
  walk w(h.walk_length());
  full_crossings(w, h, 0);
}
```

Listing 8

Ideally, instead of throwing an exception when a step in a walk is too large we should create a class to represent a limited range integer and use it instead of **`unsigned char`**. That would be a little too much work for this article however, so I'm not going to bother.

So now we're ready to start generating random walks and taking some measurements. The temptation to use a recursive algorithm to generate the full set of walks of a given length is strong; the approach is a natural fit for this kind of problem. At each step we can iterate over every possible move and then recursively generate the remainder of the walk. Each time we reach the end of a walk we add the number of crossings to the histogram to record the results. Listing 8, which calculates the histogram of self crossings, illustrates how we might implement this.

Whilst recursion can greatly simplify the expression of these kinds of operations, it can, in some situations at least, be less efficient than iteration.

Is it possible to express this operation succinctly with an iterative approach?

Well, **`std::next_permutation`** provides us with a clue as to how to go about it. It takes an iterator range and transforms it to the lexicographically next greatest permutation of the contained values. What we need is a **`next_state`** function that transforms the values in an iterator range to the lexicographically next greatest set of values.

To make it as general as possible, we shouldn't assume that the values are integer types. We must assume a method for transforming a value to the next greatest, however. We have a likely candidate in **`operator++`**; it

the complexity arises from setting up the digits and printing out the states rather than from iterating through them

```
template<class BidIt, class T>
bool
rotate_state(BidIt it, const T &lb, const T &ub)
{
  bool last = ++*it==ub;
  if(last) *it=lb;
  return last;
}
```

Listing 9

```
template<class BidIt, class T>
bool
next_state(BidIt first, BidIt last,
           const T &ub, const T &lb = T())

  BidIt it = last;
  while(it!=first && rotate_state(--it, lb, ub));
  return first!=last && (it!=first || *it!=lb);
}
```

Listing 10

```
std::vector<int> state(3, 0);
std::ostream_iterator<int> out(std::cout);

do
{
  std::copy(state.begin(), state.end(), out);
  std::cout << std::endl;
}
while(next_state(state.begin(), state.end(), 2));
```

Listing 11

already does what we need for integer types and can be overloaded for user defined classes. If overloading is not appropriate, perhaps the state transformation is too computationally expensive or our state is formed from classes that we cannot modify, we could instead use iterators into a container of states.

So how should the algorithm operate? Well, we need simply observe that iterating through a set of states is equivalent to counting through a set of integers. Each time a digit takes its maximum value, or upper bound, we set it to its minimum value, or lower bound, and perform a carry; i.e. increment the next most significant digit.

We begin with a helper function to increment a digit and indicate whether or not we need to carry. Listing 9 shows rotating a digit of the state.

Given this we need only take care that the carry operation ripples through the iterator range correctly. Listing 10 shows generating the next state.

All of the work is done in the `while` statement. We iterate backwards through the range, rotating the state for as long as we need to carry. If we have reached the first digit and it has been rotated back to the lower bound, we have reached the state in which every digit has the minimum value. Assuming that this was the starting state, have exhausted all possible states and return `false` to indicate this.

Note that we pass in the lower and upper bounds in decreasing order. This is because there is a reasonable value for the lower bound, namely the default constructed value. Note that built in types will be zero initialised which is very likely to be what we want for integer valued states.

```
000
001
010
011
100
101
110
111
```

Figure 4

Listing 11 illustrates the use of `next_state` for states with integer valued digits and Figure 4 illustrates the output from this code snippet.

```
typedef std::vector<std::string> digits_type;
typedef std::vector<
   digits_type::const_iterator> state_type;

digits_type digits(3);
digits[0] = " ";
digits[1] = "o";
digits[2] = "x";

state_type state(2, digits.begin());
std::cout << "12" << std::endl;

do
{
  state_type::const_iterator first =
state.begin();
  state_type::const_iterator last  = state.end();

  while(first!=last) std::cout << **first++;
  std::cout << std::endl;
}
while(next_state(state.begin(), state.end(),
                 digits.end(), digits.begin()));
```

Listing 12

Listing 12 illustrates the more complex task of iterating through states with a compound type for its digits, in this case `std::string`. As discussed earlier, if we store the set of digits in a container we can use iterators to represent the current value of a digit. The output of Listing 12 is shown in Figure 5.

```
12
  
 o
 x
o 
oo
ox
x 
xo
xx
```

Figure 5

As can be seen, the complexity arises from setting up the digits and printing out the states rather than from iterating through them, which remains as simple as it was in the integer case.

Rewriting `full_crossings` to exploit `next_state` is relatively straightforward. If anything, it's even simpler now than it was with a recursive implementation.

I shall continue to use the iterative approach because of the surprising fact that it actually simplifies the code

```
void
knots::full_crossings(knot_histogram &h)
{
  if(h.walk_length())
  {
    walk w(h.walk_length(), 0);
    h.add(crossings(w));

    while(next_state(w.begin(), w.end(), 9))
    {
      h.add(crossings(w));
    }
  }
}
```

Listing 13

Listing 13 shows calculating the histogram of self crossings.

But is it any more efficient?

Sadly, not very much so, as Figure 6, which shows the computational expense of generating knot histograms, clearly illustrates.

So why not? Could it be that the cost of counting the number of self crossings vastly outweighs the cost of recursively constructing the walk; the use of `std::set` to keep track of the visited locations results in O(n ln n) complexity for n crossings. Or perhaps that my compiler and the hardware I'm using are able to mitigate the cost of recursion that I recall from my youth.

Actually, no.

Had I profiled the code in the first place as did one of the reviewers (thanks Roger), I'd have noticed that most of the time is spent inserting and erasing elements from `std::set`. Unfortunately I've not given myself enough time to replace `std::set` with something more efficient.

However, I shall continue to use the iterative approach because of the surprising fact that it actually simplifies the code.

So what do the histograms look like? Figure 7 shows the results for exhaustive enumeration of walks of lengths from six to nine steps. Before looking at them, it's worth pointing out again that our histogram records the number of knots as a proportion of the length of the walk, so the histograms will record counts from 0.0 to 1.0.

n	recursed time (seconds	iterated time (seconds)
5	0.17	0.17
6	1.80	1.69
7	17.80	16.89
8	175.56	167.51
9	1701.56	1650.41

Figure 6

So how are the knots distributed? Is there a standard statistical distribution that describes them? Well, I have my suspicions, but before we can check them we need to know the average, or mean, number of knots for a walk of a given length.

We can calculate the observed mean number of crossings directly from these histograms. Firstly we should note that the walks recorded in each bucket are those with lengths less than the label, but greater than or equal to that of the previous bucket. For these short walks, we have one knot count per bucket, so the number of knots must be the previous bucket's value and zero for the first bucket.

To calculate the mean, we simply add up the proportion of the walks in each bucket multiplied by the number of knots it represents. Recall that the mean is defined as the expected number of knots and that we use *E(x)* to denote the expected value of *x*.

$$\frac{E\,(knots)}{n} = \sum_{i\,=\,1}^{n} \frac{i-1}{n} \times \frac{count(i)}{9^n - 1}$$

As pointed out above, we're ignoring the walk with the maximum number of knots, so there are only 9^n-1 walks.

The results of this calculation are given in Figure 8, which shows the expected proportional number of knots for walks of length 6, 7, 8 and 9.

So can we deduce an explicit, or closed form, formula for the expected number of knots? Well, we shall have a go at it next time. ■

Acknowledgements

With thanks to Andrey Biryuk and Chris Morris for proof reading this article.

References and further reading

[Bachelier00] Bachelier, L., *Théorie de la Spéculation*, PhD thesis, 1900.

[Brown28] Brown, R., 'A brief account of microscopical observations made in the months of June, July and August, 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies', *Edinburgh New Philosophical Journal*, July-September, pp. 358-371, 1828.

[Feller68] Feller, W., *An Introduction to Probability Theory and its Applications,* vol. 1, 3rd ed., Wiley, 1968.

[Hayes97] Hayes, B., 'Square Knots', *American Scientist*, vol. 85, num. 6, pp. 506-510, 1997

[Hull05] Hull, J., *Options, Futures and Other Derivatives*, 6th ed., Prentice Hall, 2005.

[Jerome89] Jerome, J. K., *Three Men in a Boat*, J. W. Arrowsmith, 1889.

[Nechaev98] Nechaev, S., 'Statistics of Knots and Entangled Random Walks', arXiv:cond-mat/9812205 v1, www.arxiv.org, 1998.

[Press92] Press et al., *Numerical Recipes in C*, 2nd ed., Cambridge, 1992

[Robbins55] Robbins, H., 'A Remark on Stirling's Formula', *American Mathematical Monthly*, vol. 62, pp. 26-29, 1955.

[Thiele80] Theile, T. N., 'Om Anvendelse af mindste Kvadraters Methode i nogle Tilfælde, hvor en Komplikation af visse Slags uensartede tilfældige Fejlkilder giver Fejlene en 'systematisk' Karakter', *Vidensk. Selsk. Skr. 5. Rk., naturvid. og mat. Afd.*, vol. 12, pp. 381-408, 1880.

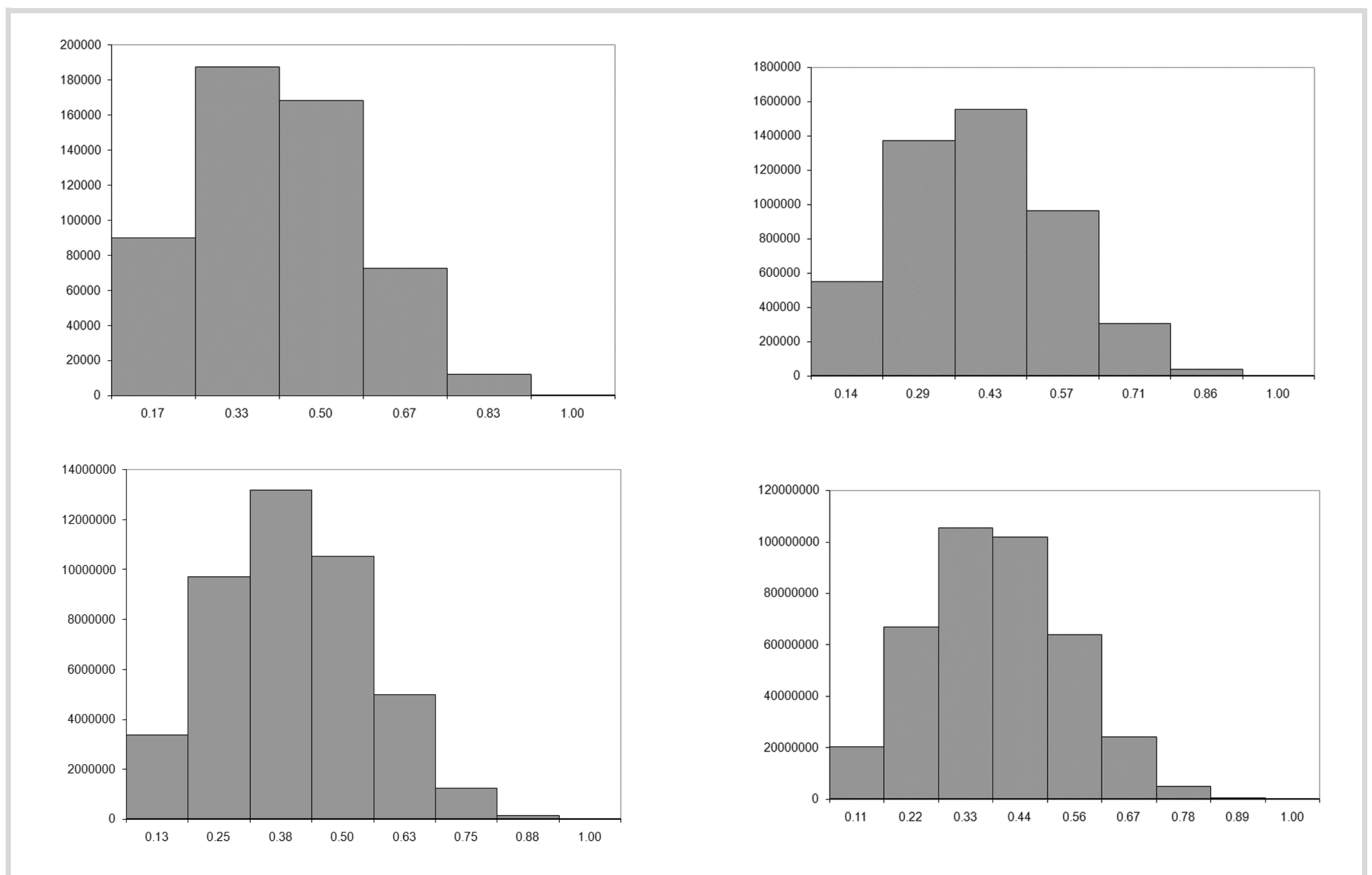

Figure 7

n	E(knots)/n
6	0.2492
7	0.2623
8	0.2737
9	0.2837

Figure 8

The Way of the Consultant

Effective communication is a challenging reponsibility of the communicator – Teedy Deigh offers some observations on how consultants can meet this challenge.

As with many other skills in life, there are different levels of mastery and achievement in reviewing things such as code, architecture or development process. At the pinnacle of achievement is the master level of consultant, where the meaning of any statement is shrouded in deep mystery, opportunistic ambiguity, tenebrous circumlocution and consultancy fees.

But how can one distinguish between the different levels? Consider the following response to a questionable system implementation:

> What do I think? This code sucks!

Although brief and honest, it is clearly the response of someone who has not even been initiated on the path. The following, however, shows some promise, but no more than that of an infant Padawan:

> What do I think? Well… it's not *all* bad! Nothing that some aggressive, merciless and inconsiderate refactoring couldn't solve.

By contrast the following response demonstrates a superior command of the panoply of techniques involved in consultancy:

> What do I think? Although there are aspects of the system's design that are sound, the solution as a whole may be better aligned with the needs of the business by leveraging the synergies of complementary solution paths. The resulting amelioration of quality will be further enhanced by the displacement of vestigial solution components extant from the status quo.

Only when the other person looks completely perplexed, appears hypnotised, has fallen asleep or has wandered off, can one know that they have ascended to an introductory level of mastery. The principal consultant, however, is one who has passed through the gate without even bothering to shut it once through. A true Jedi master of consultancy is able to present a dazzling array of possibilities with the swiftness of a light sabre, the sharpness of an Overload reviewer's feedback and the simplicity of a US President:

> What do I think? It depends.

Here ends the lessen (sic). ■

Following the valuable insights and success of her April 2007 article, A Practical Form of OO Layering, **Teedy Deigh** has found herself increasingly involved in the rarefied atmosphere of software development consultancy. She is only too happy to pass off her subjective whims as sound advice, and her opinions as grounded in objective reality. It is almost as much fun as programming Singletons!

Learn to write better code
Take steps to improve your skills
Release your talents

ACCU
PROFESSIONALISM IN PROGRAMMING
JOIN : IN