Cutting cloth

How to minimise waste when dividing up rectangular space

Every now and then, I get an email from a reader asking for help with a particular computational problem. Said reader is usually sure there's an algorithm out there that would help them. Sometimes there is, and I'm able to give them pointers to assist.

On other occasions, after doing some digging around and a bit of research, I discover a whole branch of computer science I'd been unaware of, and then spend many happy hours investigating. Such was the pattern a couple of weeks ago, when, prompted by a reader question, I discovered the two dimensional bin-packing problem and the various strategies that can be used to solve it.

Imagine the following scenario. You have a bolt of cloth of fixed width and some indeterminate length (that is, the bolt will be long enough that it doesn't affect the solution). You are given the measurements of a set of rectangles that you have to cut from this cloth. The problem is how to arrange the rectangles on the cloth so that you minimise the waste from the cutting process.

This particular problem extends into other spheres as well. How do you arrange ads in newspaper columns? If you think of tasks as rectangles where the height represents your resources and the width your time, how do you pack them along the horizontal time axis so that completing all the tasks takes the shortest time? Suppose you're in charge of writing a memory heap engine – how do you allocate memory blocks in the most space-efficient manner to minimise overall memory usage?

Let's go back to our cloth cutting scenario. First, we assume that each rectangle will fit on the cloth (that is, the width of every rectangle is less than the width of the cloth). Also, with cloth, it's likely that there is a pattern printed on it, or that there's a distinctive weave. A rectangle cut one way will look different than the same one cut after rotating it by 90 degrees. As a result of this, we can't rotate the rectangles we have to cut in order to fit them ▸

PCP286.theory 121 30/7/09 3:35:8 pm

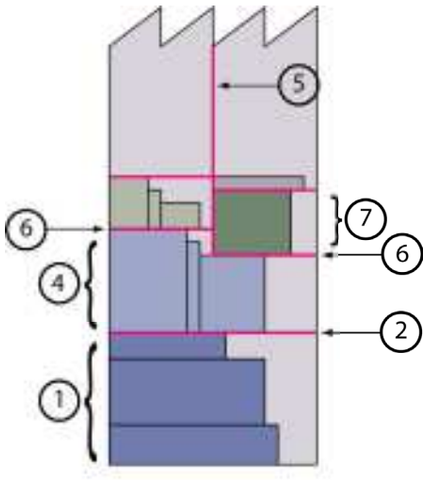
more efficiently. (For example, to illustrate the converse, suppose that we were cutting rectangles from a roll of white paper. Here we can rotate the rectangles by 90 degrees – and maybe other smaller angles – without a problem, but the algorithm becomes much harder to analyse).

So, we'll limit ourselves to fitting the rectangles as is, with no rotations allowed. Furthermore, we'll consider the scenario where we know the sizes of all the rectangles in advance, unlike a kind of Tetris situation where we're given rectangles one by one and have to fit them there and then. The former scenario is known as 'off-line' and the latter is called 'on-line', and is of particular use in the memory heap engine problem.

Sleator's algorithm

The first successful approach to solving the rectangle-cutting problem was provided by Daniel Sleator in a 1979 paper called *A 2.5 Times Optimal Algorithm for Packing in Two Dimensions*. We'll use Sleator's orientation where the cloth extends upwards (see Figure 1 for an example). It goes like this:

1. Put all the rectangles with a width greater than half the width of the cloth at the bottom, one on top of the other.
2. Draw a horizontal line across the cloth at the height of these rectangles. The remaining rectangles will all be placed above this line.
3. Sort the remaining pieces in descending order of height (not width).
4. Place rectangles in height order from left to right along the dividing line, until you run out (in which case, we're done) or the next rectangle is too wide to fit. At least two rectangles will be placed.
5. Draw a vertical line halfway between the right and left edges.
6. Draw a horizontal line segment in the left half and one in the right half – the baselines – so that they rest on top of the highest rectangle in their respective half.
7. Choose the half with the lowest baseline. Place rectangles in height order in that half along the baseline, until you run out or the next rectangle is too wide to fit. At least one rectangle will be placed.
8. Go back to step 6, and repeat.



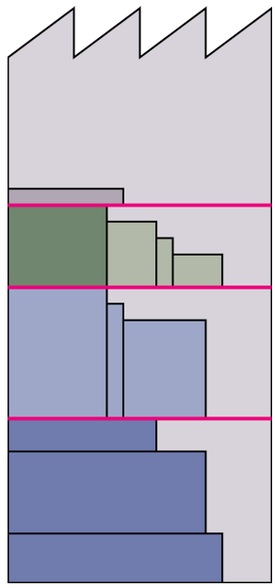
▲ **Figure 1: The results of Sleator's algorithm.**

Now, if you look at Figure 1, I'm sure you can see that the algorithm isn't perfect: there are some quite largish holes that smaller rectangles could fit into. Actually, Sleator's intent here wasn't to invent the best algorithm, but to devise one that he could derive some important mathematical results from. The most important of these was that it packed the rectangles such that the length of cloth taken was at most 2.5 times the optimal packing.

Strip-based algorithms

Sleator's algorithm is an example of a strip-based algorithm: we fill in strips across (half of) the width of the cloth. Once a strip is filled we don't go back to it, even if we could put a smaller rectangle in the 'hole' left.

Rather than using a two-column method, as Sleator did, you can use a one-column method instead, filling strips across the cloth from left to right (see Figure 2). First, you stack all the rectangles that are wider than half the width of the cloth, as in the Sleator algorithm, sort the remainder by height, and then place the rectangles in order by height across the cloth in strips. Although this algorithm will likely produce a worse bin-packing solution than Sleator's algorithm, it's nevertheless important, in that the cloth can be cut by guillotine (that is, by straight cuts across the fabric without having to worry about cutting around any corners).



▲ **Figure 2: The results of the one-column variant of Sleator's algorithm.**

Once we have the idea of a strip-based solution, there are several other algorithms we could investigate that use this type of solution. In essence, they all sort the rectangles in some order and then place them in strips across the cloth. Most of them are a little more aggressive in placing the rectangles, though, as we'll see.

The first such algorithm is based on one we've already seen. Sort all the rectangles by height and then start placing them in order, in strips, across the cloth. The difference here is that we don't place the very-wide rectangles first, and we also go through a 'first fit'-type process when the next rectangle doesn't fit.

Let's see how this algorithm works (see Figure 3). First, we sort the rectangles in height order (as before). Next, we start laying the rectangles down on the cloth. The first rectangle laid will define the height of the strip being produced. When we get to the point where the next rectangle in the series doesn't fit, instead of abandoning that strip, we look though the remaining rectangles one by one in order, seeing if there's one that will fit instead. If there is one, we place it. Now we continue the search trying to fill in the remaining space in that strip, if any. Once the strip is filled as far as we can, we start another strip with the

Exact solutions

For a small number of rectangles, deriving an exact solution within a reasonable time is possible. In essence, the proposed plan of attack is to place the rectangles in every conceivable position and select the solution that wastes the least space. The best bet is to decide on the smallest unit of measurement (say squares 1cm on a side), which we refer to as the 'granularity'. Divide the cloth and rectangles up into those 'cells' and then mark off the rectangles' cells on the cloth. You'll then more easily be able to place the rectangles (and have fewer places to check) than if you didn't use the granularity method. ■

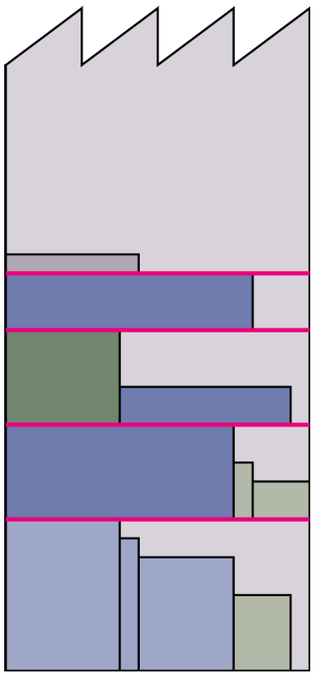
Spotlight on… Memory heaps

Modern programming runtimes, especially those that don't implement garbage collection (C++ or Delphi, say), allocate memory from the 'heap'. The allocation problem is a standard 1D bin-packing problem (unlike the 2D ones in the article).

The heap manager allocates memory from the operating system in large blocks and then parcels it out to the application in smaller chunks. When the program code requests memory, the heap manager looks at the free space in the blocks and allocates the memory from one of them (if there's no free space, it will allocate itself another block from the OS).

There are two main strategies: 'first fit' and 'best fit'. The first fit algorithm just looks through the blocks in order until it finds space that fits (in essence a chunk that's bigger or equal to the amount requested). The best fit algorithm looks through the free space, trying to find a chunk that's equal to or just bigger. The former will produce more holes (a worse packing strategy), but the latter takes more time (a better packing strategy). ■

PCP286.theory 122 30/7/09 3:35:9 pm



▲ **Figure 3: The results of a strip-based algorithm, arranged by rectangle height.**

tallest remaining rectangle and continue placing as before.

The second 'sort-then-place' strip-based algorithm sorts the rectangles into width order first, and then proceeds like the sort-by-height algorithm. The difficulty with this algorithm is that we don't know the height of a given strip until we've finished filling it as best we can. Apart from this, the algorithm proceeds pretty much as before.

There is, however, one interesting wrinkle with this second algorithm (sort-by-width). Instead of using a first fit process, we can instead use the sorted order to our advantage, and try and find a rectangle that would fit exactly. Since the rectangles are sorted by width, we can perform a binary search to find a remaining rectangle that would best fit the remaining space on the cloth.

All of these strip-based algorithms (so Sleator's two-column algorithm, the one-column variant, the sort-by-height and the sort-by-width) abandon a strip once it's been 'filled'. There's no attempt to fill the remaining space between two adjacent strips. The strips, if you like, are 'bins'. Once a bin is full, you never go back to it, and instead open a new one and start filling it instead. In essence the algorithms are 'linear' once the rectangles have been sorted: there's no choice involved. There's no goal seeking, apart from the rudimentary 'let's see if we can't find a rectangle to put here'.

A need for speed?

The simplest algorithms we've shown here (the various strip-based methods) are very fast, but the packing is not optimal. Provided that the material being cut is inexpensive and the cutting mechanism fast (for example, cloth), these may be good-enough solutions. The other goal-seeking algorithms, especially the optimisation strategies, will take a much longer time, but using one may be worth the effort if the material is more expensive, or takes longer to cut (for example, thick sheet steel). As ever, select the algorithm based upon the scenario you're working with. ■

Thinking about boxes

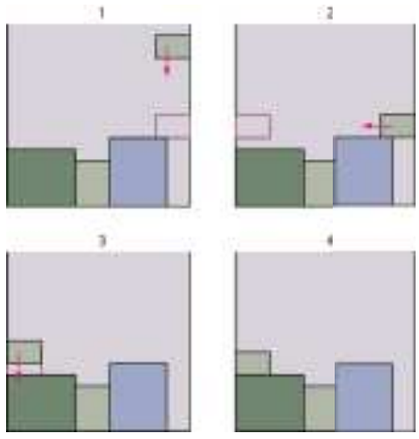
An interesting variant on this is packing boxes on a lorry so that the boxes do not have to be partially unloaded during delivery. That is, the boxes for the customer who is first on a delivery route are placed last on the lorry, and those for the customer last on the route are placed first. We can assume that the problem is a 2D one (that is, we won't places boxes on top of one another, so assume they're large TVs or something). How would you change the algorithms you've seen to pack the lorry most efficiently? ■

Bottom-left algorithms

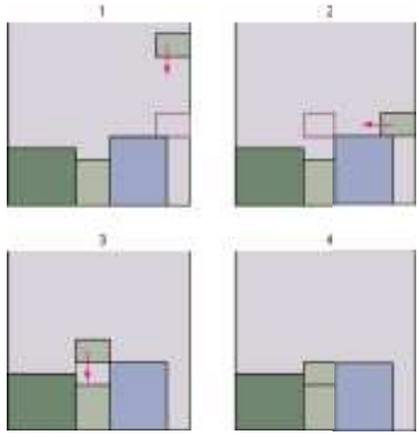
Let's look at some other types of algorithm – those that don't use strips at all. In other words, we'll remove the restriction of using guillotine cuts. What we're left with starts to look a bit like Tetris.

The first class of algorithms is known as 'bottom-left' (BL) algorithms. A successful one (at least for a while) was created by S Jacobs in 1996. This was an optimisation algorithm: a simple algorithm on the surface, but one that was designed to be run over and over again under the control of some optimisation strategy. The one Jacobs used was a genetic strategy, but the underlying algorithm can be used by other optimisation heuristics as well.

The algorithm worked like this (see Figure 4). For some ordering of the rectangles (this would be provided by the optimisation strategy), each rectangle is placed on the cloth at the top-right corner. (Since the cloth is assumed to have infinite length, we basically say 'high enough'). Now we move the rectangle down and left in a series of steps. We first slide it down until it hits another rectangle or the bottom of the cloth. Then we slide the rectangle to the left until it hits an already positioned rectangle, or the left side of the cloth.



▲ **Figure 4: The results of Jacob's algorithm.**



▲ **Figure 5: The results of Lui and Teng's algorithm.**

We continue this down-slide, left-slide, until we can no longer move the rectangle. It's now placed, and we select the next rectangle and do the same thing. Essentially we will slide a given rectangle in a series of steps until it is as far bottom-left as it can go.

A variant of this simple algorithm was suggested by D Lui and H Teng in 1999 (see Figure 5). The variation was that downward slides take precedence over leftward slides. In other words, the rectangles only slide left until there was a possible slide downwards. If such a downwards slide was found, you should then slide the rectangle downwards as far as possible, and then try sliding left again.

The second class of algorithms builds on the bottom-left algorithms. They're known, imaginatively enough, as 'bottom-left fill' algorithms. Again, they're algorithms based upon an optimisation strategy.

Very broadly the algorithm goes like this. You maintain a set of points where the bottom-left corner of a rectangle could be placed, sorted by lowest and then leftmost. At the start, there is but one such point: the bottom-left corner of the cloth. So the first rectangle is easy: there's only one point where you can place it. After you've placed it at the bottom-left corner of the cloth, you remove that point from your list, but you now have two more points to add to it: the top-left corner and the bottom-right corner of the just-placed rectangle. From now on, for each rectangle, you try and place it at the first point. If it won't fit, you try the next and so on, until you can fit it. You remove that point from your list and work out the new points to add, and then add them in sorted order. Continue until you've placed all your rectangles.

These types of algorithms – the non-strip-based algorithms – will in general provide a better packing of the rectangles than the simpler ones. But this better packing comes at a cost: time. It takes much longer to find a packing than in the simpler cases, but that time is offset by the reduced waste. ■

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

PCP286.theory 123 30/7/09 3:35:11 pm