Putting the Pieces Together

Andrew Glassner

Secrets are sensitive things. The more people you tell, the less your secret is secure. So usually we try to control the spread of secrets as much as possible. In the November/December 2001 issue of *IEEE Computer Graphics and Applications*, I discussed a radically new way to keep secrets using the techniques of quantum cryptography. Although it's a fascinating theory, we don't have products yet that implement those principles. In the meantime, we have to manage with more traditional techniques.

Of all the ways to share a secret, perhaps the most dangerous is to write it down on paper in plain language. That has the advantages of permanence and some reduction in the need to repeat yourself verbally, but it runs a big risk: anyone who can get the paper can read your secret, even years later. If you think someone is going to see your document and use it against you, you've got to destroy the paper before they get to it.

Then, however, you have another problem, which is how best to destroy the paper. If the secret is important enough that you need to protect it, probably the best course is to burn the page, and then if you're really worried, scatter the ashes. If for some reason you can't burn it, instead you could try ripping it up into tiny pieces. But that doesn't seem very secure.

In recent years, manufacturers have begun offering devices known as document shredders. Two major categories of these machines exist: strip cutters and cross cutters (also known as confetti cutters). Both look like tall garbage cans with a mechanical unit on the top into which you feed sheets of paper.

The strip cutters slice pages into parallel strips, usually ranging from 0.25 to 0.125 inch wide. Because the pages are usually fed in short side first, an American legal-sized page results in 34 to 68 strips, each 11 inches long. The cross cutters add another step by cutting each of these strips into shorter segments, typically between 1.25 and 1.5 inches long. These devices have become very popular in recent years for destroying documents. (Enron, anyone?)

It seems unlikely to me that those strips couldn't be reassembled. I thought in this column I'd investigate how to assemble strip-cut documents back together.

The tools that solve this problem can also address a more general problem. Given a collection of pieces that are arbitrarily sized, have images on them, and are presumed to fit together to make one or more larger images, can we assemble the pieces into these larger images? In the case of shredded documents, the pieces are vertical strips of paper containing pages of text. Other applications include repairing a broken object that shattered or assembling a jigsaw puzzle. Let's see how we might go about reconstructing the text.

Getting started

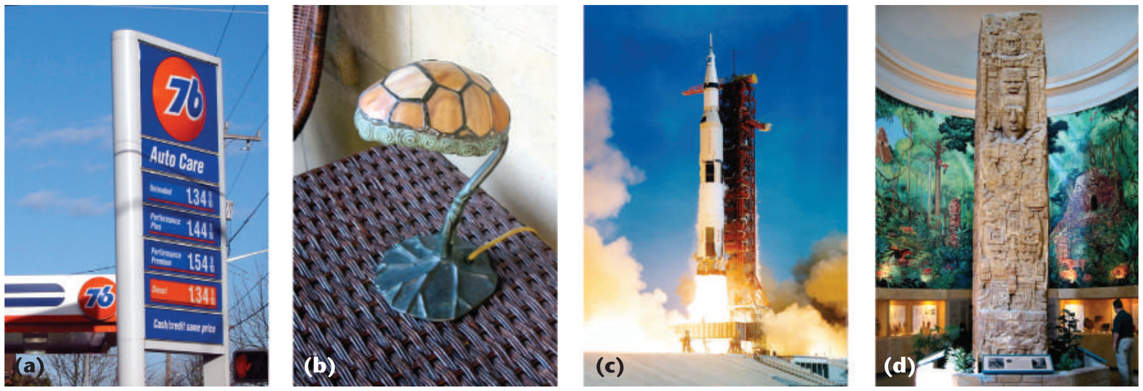
It's hard to assemble a lot of pieces at once, so I put the pieces together two at a time. In other words, I'll grow the reconstruction by first combining two pieces that belong together, then combining two more, and so on.

We should ask two questions any time we test objects pairwise. First, do we have two objects that are likely to be matches? Second, do they indeed match? The first question is important for reasons of efficiency, while the second is necessary to get a good assembly.

To get a handle on typical numbers, suppose you're recovering a bag of strip-cut documents. Each 8.5-inch wide page will have been sliced into 68 0.125-inch strips. In practice, the left and right margins of each page are generally about 1.25 inches. If we eliminate these all-white strips, then we have 6 inches of printed width, or 48 strips per page. So if someone shredded 2,000 pages, there would be about 96,000 strips, which often all go into a single bag below the shredder and get thrown away en masse. In contrast, jigsaw puzzles seem to max out at about 1,000 pieces.

So if we were to compare every jigsaw piece to every other piece (each piece generally has four sides), that's $4(N^2)$ tests, which for $N = 1{,}000$ is about 4 million tests. The shredder problem has only two edges to test per strip, so we'd require $2(N^2)$ tests, which for $N = 96{,}000$ strips is somewhat more than 18 billion. That's going to be a tough order, even on supercomputers. Hence, the efficiency question is important so that we're not wasting time comparing pieces that have no chance of matching.

A good matching test is important to ensure that pieces correctly combine. It needs to be efficient because this is where the program will spend most of its time. It also needs to look for a good match, but be tolerant of near matches. When a razor slices a page, or a jigsaw or other cutter makes puzzle pieces, the material directly under the blade is naturally lost, so the remaining two sides won't fit perfectly. Furthermore, borders and edges



1 Test images for the thin figure set. (a) Gas sign, (b) turtle lamp, (c) rocket (NASA), and (d) Mayan monument.

2 Test images for the wide figure set. (a) Earth (NASA), (b) garden, (c) musical corner, and (d) outdoor concert.

are common features, and they might align with the piece edges, making features even more difficult to detect. This is often the case for text. Consider a letter like a capital I, or lowercase l. These both have most of their information in a vertical line, aligned to the long side of the page. If the shredding machine cuts just to the left of one of these letters, there will be nothing shared between the two strips over this border. In fact, much printed text is this way: two strips that started out adjacent have frequent regions where corresponding regions of the two strips are opposite in color—one edge is black and the other is white. Thus, our matching test must pick up on what similarities are available and not overly penalize mismatches. These are contradictory desires, of course, which is what makes matching functions so interesting.

For simplicity, I'll first stick to the document reconstruction problem where the pieces are strips. Later we'll see that the ideas generalize to jigsaw and other shapes.

Although my original inspiration was to reassemble text documents, I quickly found that visually checking the reconstructions was tedious, and it was hard to spot where a misplaced strip should have gone.

To make things easier, I used two sets of photographs to develop my reconstruction software. Figure 1 is what I call the thin set, made of four images each that are about two-thirds as wide as they are tall. Figure 2 show the wide set, where each image is about twice as wide as the thin images. I'll use these images for the following discussions.

Got a match?

To judge the fit between two strips, I run them through a *fitness function* to compute a *score*. I coded things so that the score indicates the degree of mismatch. Thus, the higher the score, the worse the match is between the pieces. The ideal matching function would evaluate to zero for any two strips that were supposed to be adjacent and to infinity for all other pairs.

The principle at work here is *coherence*. In this context, coherence says that we're betting that any given column of pixels in an image is going to be a lot like the columns immediately to its left and right. After all, if the images were random noise (that is, just black and white dots with no features), then matching up strips would be hopeless because statistically no pair of strips would be any better than any other pair.

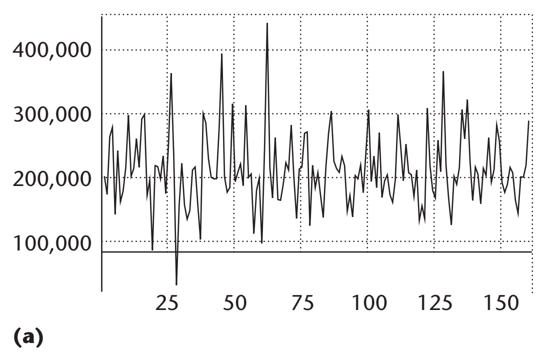
The most obvious place to start is to compare the RGB values of adjacent pixels over the edge and add them up. If we have two pixels P_0 and P_1 with colors (r_0, g_0, b_0) and (r_1, g_1, b_1), then we can find their simple difference (which I'll call D_0) by summing up the absolute values of their differences:

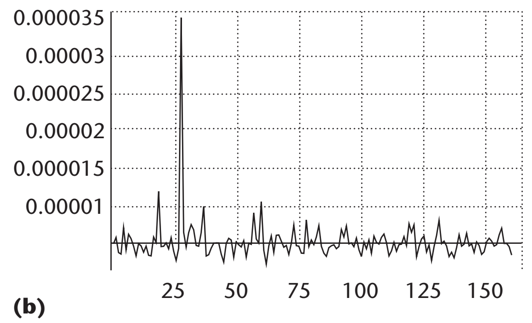
$$D_0 = \Delta r + \Delta g + \Delta b$$

where

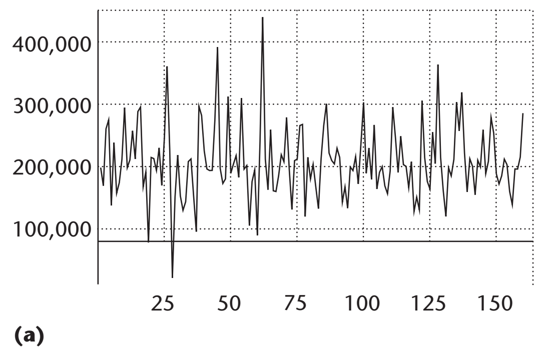
$$\Delta r = |r_0 – r_1|, \quad \Delta g = |g_0 – g_1|, \quad \Delta b = |b_0 – b_1|$$

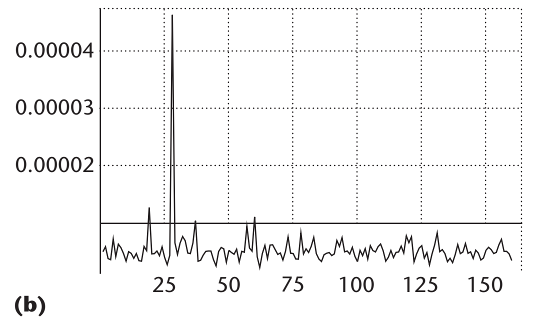
Let's see how well this metric works. I'll take the four thin images, cut them into 40 strips each, and randomly scramble them. Now let's try to find the piece that best fits the right-hand side of the left-most strip from Mayan monument image. Looking through the scrambled pieces, I find that the left-most strip of this image was assigned strip number 69 and that the strip that was originally to its right became strip 27. So I'll place each strip to the right of strip 69, compute its score using D_0, and see if strip 27 has the lowest score.





3 Looking for strip 27 using metric D_0. (a) The scores using D_0 and (b) the inverse of Figure 3a. The ratio of the spike to the minimum value is about 16.





4 Looking for strip 27 using metric D_1. (a) The scores using D_1 and (b) the inverse of Figure 4a. The ratio of the spike to the minimum value is about 20.

Figure 3a shows the result. The scores range from a low of 28,516 for strip 27 to a high score of 442,850. Number 27 seems to stand out, but perhaps we can improve the data. In Figure 3b, I've plotted the inverse of Figure 3a (that is, each value x is replaced by $1/x$). This looks great, and strip 27 sure stands out, so we're off to a good start. The ratio between the highest value in the inverse plot to the lowest value is about 16—let's see if we can improve that.

Consider the real world for a moment. Paper has variations, and scanners are noisy. We'd like to avoid penalizing little imperfections caused by dust and scratches and printing technology, where one dot might be just slightly darker or lighter than another. So I'll add another step to the test function: a *threshold*. If the difference between two color values is less than the threshold T, I'll set it to zero:

$$\Delta r^T = \begin{cases} \text{if } \Delta r < T_r & 0 \\ \text{else} & \Delta r \end{cases}$$

and similarly for Δg^T and Δb^T. The result is a new, thresholded metric D_1:

$$D_1 = \Delta r^T + \Delta g^T + \Delta b^T$$

Figure 4a shows the result. The scores range from a low of 21,621 for strip 27, to a high score of 442,648. Figure 4b shows the inverse function. Here I used the same value of 10 for the three thresholds T_r, T_g, and T_b. The noise seems to have settled down a little in the inverse plot, which makes the spike at 27 even easier to find. The ratio between that highest spike to the smallest value in the plot is about 20, which is a nice improvement.

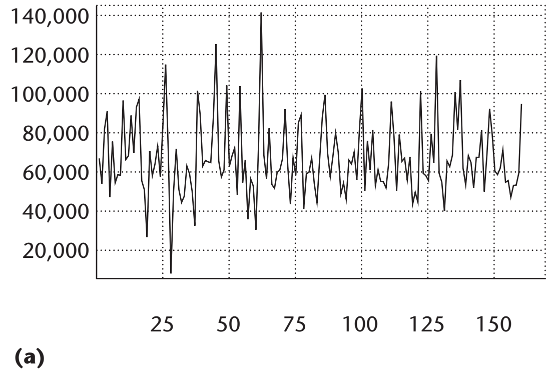
So far I've been treating the three color components equally. In the case of a black-and-white document, like type on paper, that's pretty reasonable. But when it comes to images, we know that we're more sensitive to some colors than others. This perceptual quirk won't generally help us put together photographs of the natural world, but I think it might help us with man-made environments and rendered creations like paintings and computer graphics. My hypothesis is that because artists are using the same visual system as their audience, they tend to create works tailored to that system. For example, we're more sensitive to variations in greens than in blues, so there might be more information packed into the green part of a painting than the blue parts.

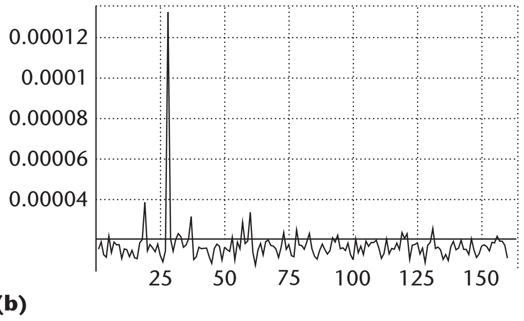
I encoded this observation into a metric D_2 that weights the three color components individually. Including the thresholded colors as before, metric D_2 is given by

$$D_2 = w_r \, \Delta r^T + w_g \, \Delta g^T + w_b \, \Delta b^T$$

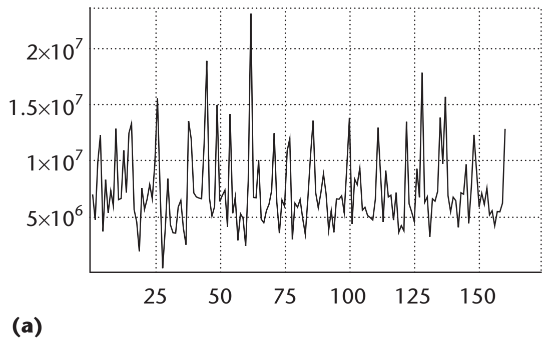
where I used weights from the standard luminance function: w_r=0.3, w_g=0.59, w_b=0.11.

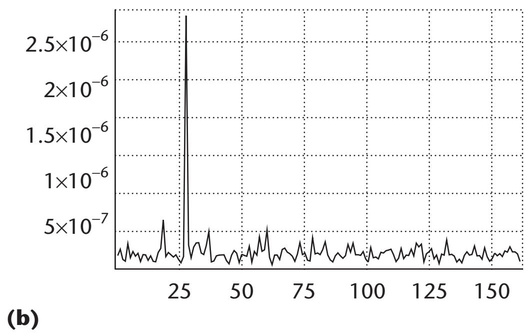
Figure 5a shows the result. The scores range from a low of 6,217 for strip 27 to a high score of 142,072. Figure 5b shows the inverse function. This modification hasn't made a big change in this data, but we've had a slight improvement on the ratio to about 23. In my experience, this color weighting can lead to a slight improvement for color images.



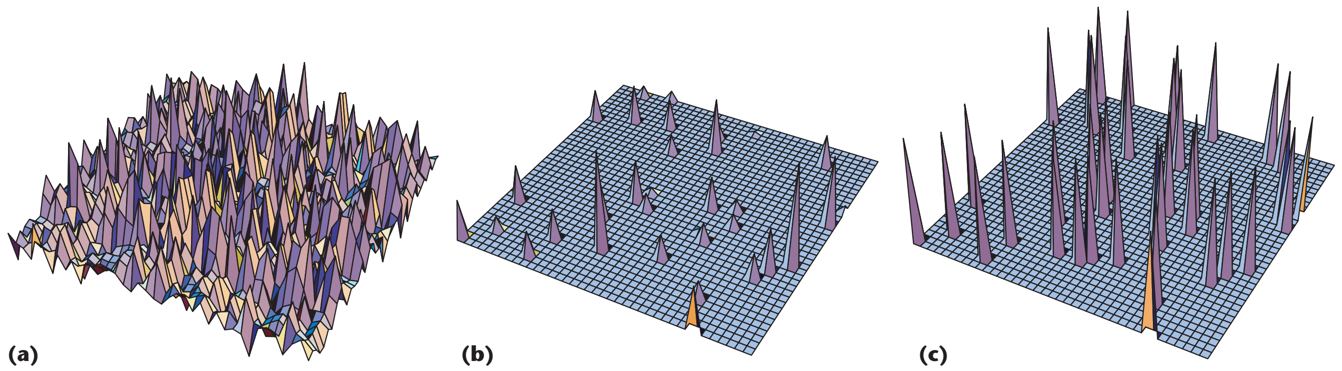


5 Looking for strip 27 using metric D_2. (a) The scores using D_2 and (b) the inverse of Figure 5a. The ratio of the spike to the minimum value is about 23.





6 Looking for strip 27 using metric D_3. (a) The scores using D_3 and (b) the inverse of Figure 6a. The ratio of the spike to the minimum value is about 64.



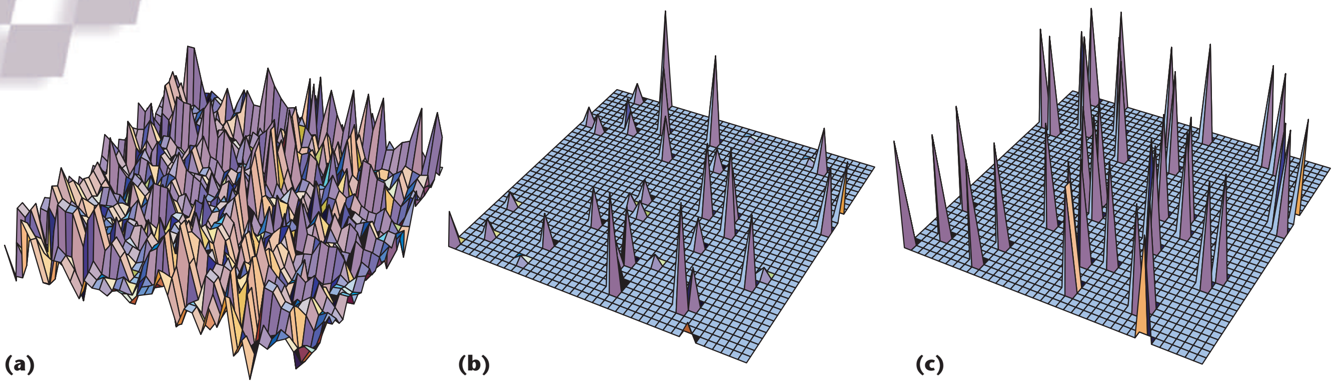
7 Comparing the four thin images after being cut into 10 strips each, using metric D_3. (a) The output of the metric, (b) the inverse of Figure 7a, and (c) a thresholded and normalized version of Figure 7b.

Remember that our goal is to make the spike for slice 27 as unambiguous as possible. A useful way to crank up differences in data is through exponentiation. This leads to a third metric, D_3:

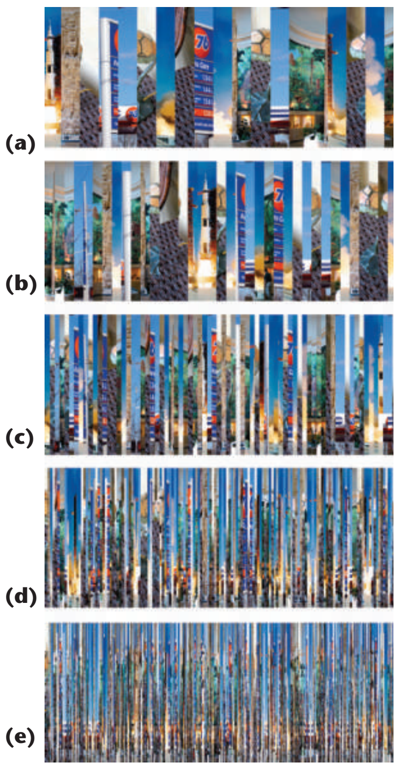
$$D_3 = (w_r \, \Delta r^T)^{n_r} + (w_g \, \Delta g^T)^{n_g} + (w_b \, \Delta b^T)^{n_b}$$

Figure 6a shows the results when I used the same value of 2 for all three exponents: $n_r = n_g = n_b = 2$. The scores range from a low of 35,000 for strip 27 to a high score of about 23 million. Figure 6b shows the inverse function. The spike now is nearly impossible to miss. The ratio of the spike's value to the minimum value in the plot is about 64. Experiments with text and color images have led me to settle on D_3 for my testing function. On each run, though, I have the opportunity to tweak all the variables, weights, thresholds, and exponents to accommodate the peculiarities of different printers, scanners, paper types, and so forth.

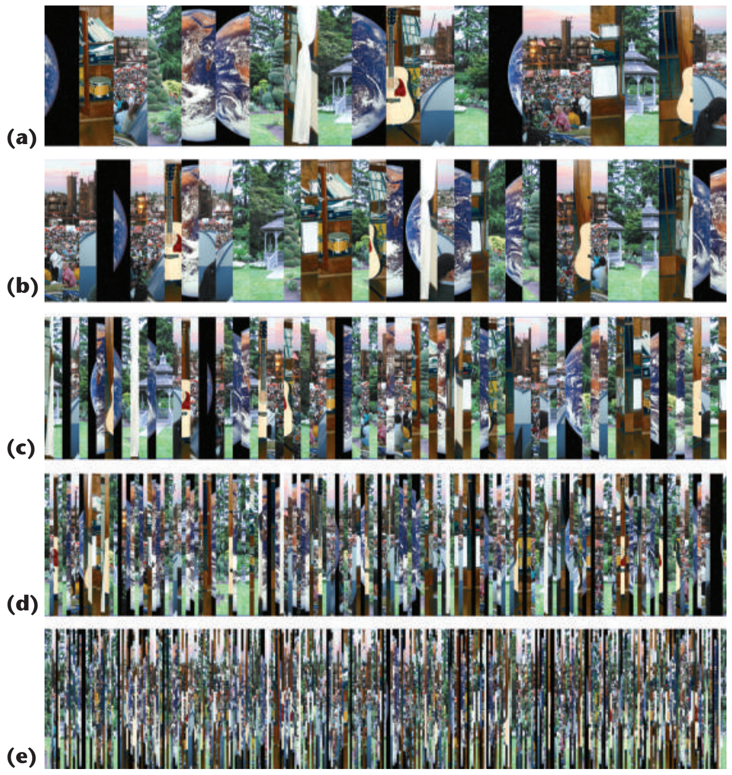
To see how this metric performs on a larger problem, Figure 7a shows the scores resulting from slicing all four images into 10 strips each and then comparing every pair. Figure 7b contains the inverse plot, which shows the spikes pretty well. Those spikes are where the metric says the two pieces match up well. In Figure 7c, I processed the inverse data so that any values below a threshold set halfway between the average and the largest value are set to zero and the others are set to one.



8 Comparing the four wide images after being cut into 10 strips each, using metric D_3. (a) The output of the metric, (b) the inverse of Figure 8a, and (c) a thresholded and normalized version of Figure 8b.



9 The unorganized strips associated with the thin data set. (a) Five strips per image, (b) 10 strips per image, (c) 20 strips per image, (d) 40 strips per image, and (e) 80 strips per image.



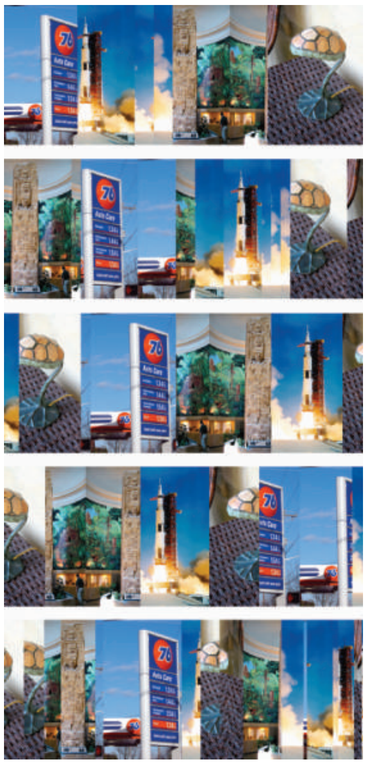
10 The unorganized strips associated with the wide data set. (a) Five strips per image, (b) 10 strips per image, (c) 20 strips per image, (d) 40 strips per image, and (e) 80 strips per image.

This basically pushes all the noise to zero and leaves just the spikes.

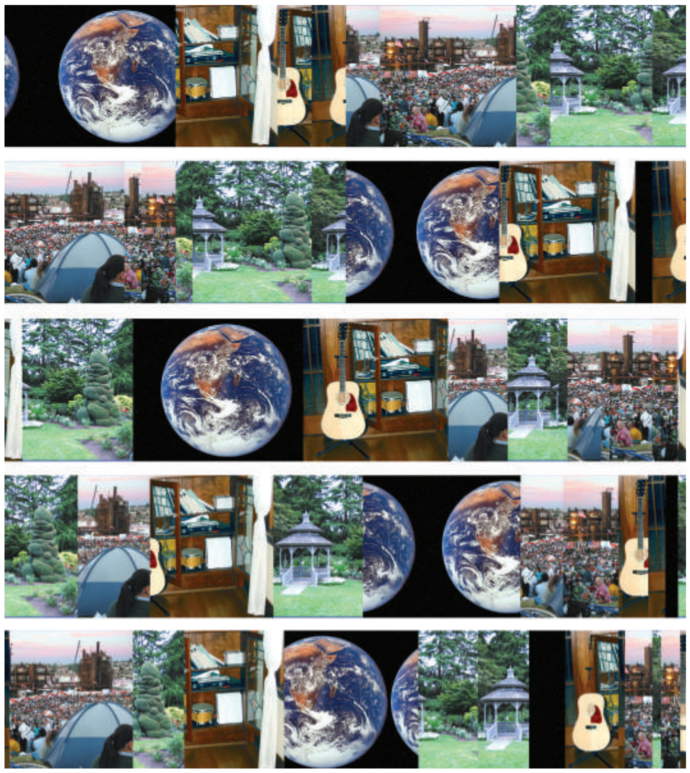
Figure 8 shows the same data for the wide data set. Some of the spikes are in the same place as in Figure 7 because I used the same random number seed for shuffling the strips in both figures.

If we could afford to run this metric on all the database objects, it would suggest an easy way to get a good initial match. Compute all the pairings and then threshold the result until the correct number of matches remains. For example, when we have four images of 10 strips each, we can find 36 good pairings. We need only 36 rather than 40 because four of these pairs just connect up the edges of unconnected pages, and because their order doesn't matter, we don't have to worry about them.

Just thresholding the pairing data and connecting the strips wouldn't always solve the problem perfectly because that process ignores important issues. For example, we must ensure that no strip is a neighbor to itself and that we don't accidentally make cycles of strips. I'll talk about these issues more in a bit.



11 The result of starting with a random strip and building to the right for the thin data set. (a) Five strips per image, (b) 10 strips per image, (c) 20 strips per image, (d) 40 strips per image, and (e) 80 strips per image.



12 The result of starting with a random strip and building to the right for the wide data set. (a) Five strips per image, (b) 10 strips per image. (c) 20 strips per image, (d) 40 strips per image, and (e) 80 strips per image.

An orderly development

Let's suppose that we had access to the entire grid of all pairings, like that of Figures 7 and 8. How might we use this data to assemble the pieces?

In this section I'll describe my first approach to assembly, which does a good, but not great, job. It points the way to a better solution, though.

Let's begin by looking at the input. Figure 9 shows the thin image set after it has been sliced into several strips. Note that there's no obvious correlation between the strips—they look as though they'd been fished out of wastebasket and scanned in randomly. One thing we do know is which way is up. (I'll talk about handling unknown orientation later.) Figure 10 shows the wide data set, in the same initial condition.

Let's randomly pick one strip, and then try to assemble it by adding to its right side using a greedy technique. So given this strip, we'll consult the database of scores, find the strip that is the best right-hand neighbor, and place that down. Then we'll get the best right neighbor for that strip, working our way from left to right, until we've finally placed them all.

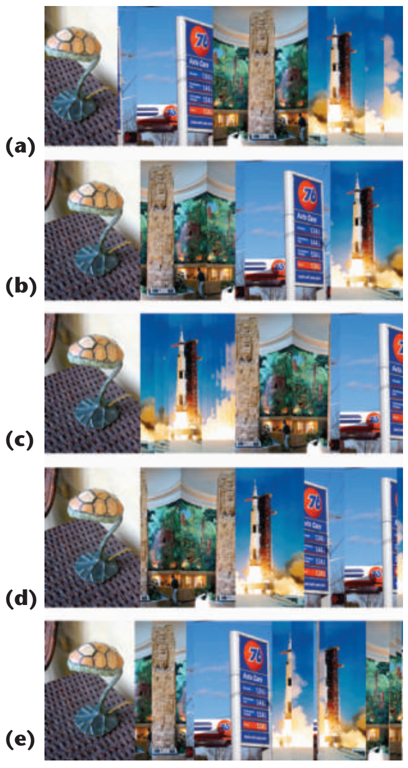
Figure 11 shows the result for several different strips in the thin data set, and Figure 12 shows the results for the wide data set. The good news is that we've reduced the chaos of unrelated strips into a few large chunks (and some smaller ones). But the images aren't quite assembled the way we'd like. Curiously, the number of strips involved changes the result but not the general feel of the errors. What went wrong?

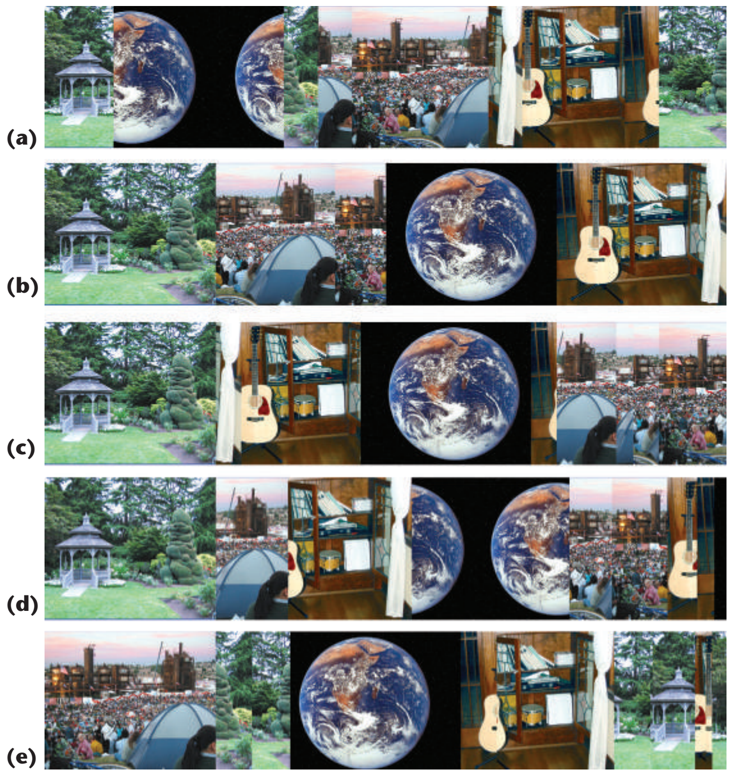
As long as we're working our way to the right in a given image, generally the pieces go together well. But then when we hit the right edge of the picture, the best remaining strip to its right is essentially a random choice. From that point on we continue to the right again, until we either hit the right side of the image, or need a strip that's already been used. Then we start again. Thus we end up getting chunks of images that get smaller as we work to the right.

Let's see what happens if we start off with a better choice. Before we put down the first piece, we'll look through all the strips and find the one that has the worse match on its left side. The idea is that whatever strip has no good left neighbors is probably the left edge of an image.

Figure 13 (next page) shows the result of this change for the thin test cases. This is encouraging but still not quite right. Figure 14 (next page) shows the result for the wide test cases. We still sometimes start at the wrong place, because the piece with the worst left neighbor, as determined by the scoring function D_3, isn't always at the image's left edge. Sometimes there's another strip somewhere in the database whose right edge just happens to match the left edge of an image pretty well, so the piece we end up with isn't really what we wanted.

As I thought about it more, I realized that this left-





13 **The result of starting with the strip with the worst left neighbor and then building to the right for the thin data set. (a) Five strips per image, (b) 10 strips per image, (c) 20 strips per image, (d) 40 strips per image, and (e) 80 strips per image.**

14 **The result of starting with the strip with the worst left neighbor and then building to the right for the wide data set. (a) Five strips per image, (b) 10 strips per image, (c) 20 strips per image, (d) 40 strips per image, and (e) 80 strips per image.**

to-right growing approach could be endlessly tweaked, but was probably doomed in the long run. One big problem is that it requires access to all the pairing data, which for a large data set will be prohibitive. But even with this information, there's still an essential problem: as we march along, looking for the best match to a given piece, we might end up selecting a piece that has a really terrific match somewhere else and then that piece is over. For example, suppose that by coincidence some strip in the middle of the gas sign image nicely matched a strip in the middle of the rocket image. When we place the gas sign strip down in the middle of the rocket image, we've just ruined our chance to get either of those images right.

Faced with these difficulties, I decided to shift gears and try a clustering approach.

Clustering

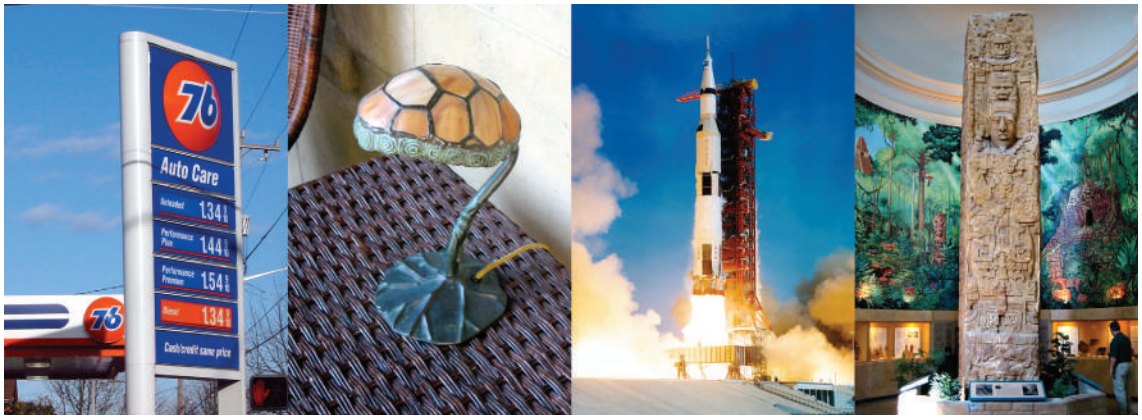
To cut down on errors, then, I created a clustering algorithm. Suppose we have four pieces, which I'll call *A*, *B*, *C*, and *D*, and when correctly assembled they're in order as *ABCD*. Now suppose that the match between *C* and *D* is excellent but the match between *A* and *B* isn't. When we've placed *A*, we look through the remaining pieces. We might find that piece *D* is the best match for the right side of *A*. However, best in this case might mean nothing more than the least worst. Of course, the computer has no way of knowing that, so it picks *D* and places it next to *A*. The shame here is that this now removes the possibility of ever getting the *CD* pairing, which had a very low score.

This is a common problem with all greedy algorithms and philosophies. By doing what's most expeditious at the moment, we can eliminate the opportunity to do something better later.

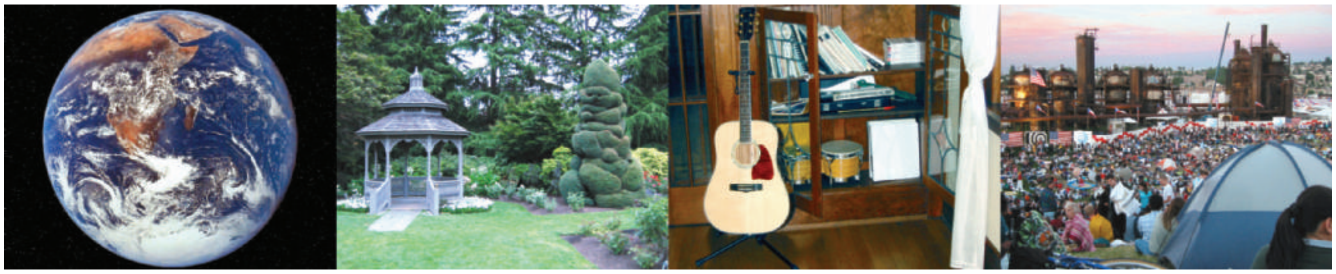
So instead, let's take a clustering approach. Here's the basic idea behind clustering: if *C* and *D* have a really low score, let's put them together first. Then we'll deal with the other pieces.

So assuming again for the moment that we have access to the entire database of scores for every pair of pieces, we could first put together the two pieces with the lowest score, then the next two, and so on, until every piece has been placed.

This algorithm works beautifully once you take care of an important gotcha—you can't make cycles. As an example, I'll reuse our *ABCD*, but to make the discussion clearer, I'll indicate each pairing with subscripts. Thus we have one score for when the right side of *A* abuts the left side of *B*, which I'll write as $A_R\,B_L$ and another score for when the right side of *B* abuts the left side of *A*, which I'll write as $B_R\,A_L$. Suppose that the four lowest scores in the database are (in ascending order) $B_R\,D_L$, $D_R\,A_L$, and $A_R\,B_L$. This would create



15 The result of the clustering algorithm for the thin database.



16 The result of the clustering algorithm for the wide database.

the cyclic sequence *BDA* where the right side of *A* is linked to the left side of *B*. Where is *C* to go? It's left out in the cold.

Even worse, if we're not careful we might find that a single strip best matches itself. For example, $A_R A_L$ might have a really low score if *A* is just a solid color. Obviously we can handle that as a special case, but if we prevent cycles of any size from forming, then single-piece cycles are prevented as well.

I prevent the formation of cycles with a brute-force technique. Whenever I want to add a piece at either end of a cluster, I check the other end to see if that piece is already there. If so, I skip adding that one in and move on to the next best pairing. The only exception to this rule is when all the pieces have been placed—the very last one will make a single big cycle that contains the whole database.

To recap, I search the database for the lowest-scored pair and put those two pieces together. Then I search for the next lowest pair, assemble those, and so on, until all the pieces have been placed. Along the way, I make sure that I don't create any cycles except when placing the last piece.

This algorithm works great. Figure 15 shows the result for the thin data set, and Figure 16 shows the wide data set. Because the whole reconstruction creates a cycle, it doesn't matter which piece you choose as the left-most one when printing them out—in effect the printout is a tube. But because it's nicer to start on a boundary, I search the entire chain, after it's been completely assembled, for the worst score between two neighboring pieces, on the assumption that this spot is likely to be a border. When I create the output image, I use the strip to the right of that border as my starting point.

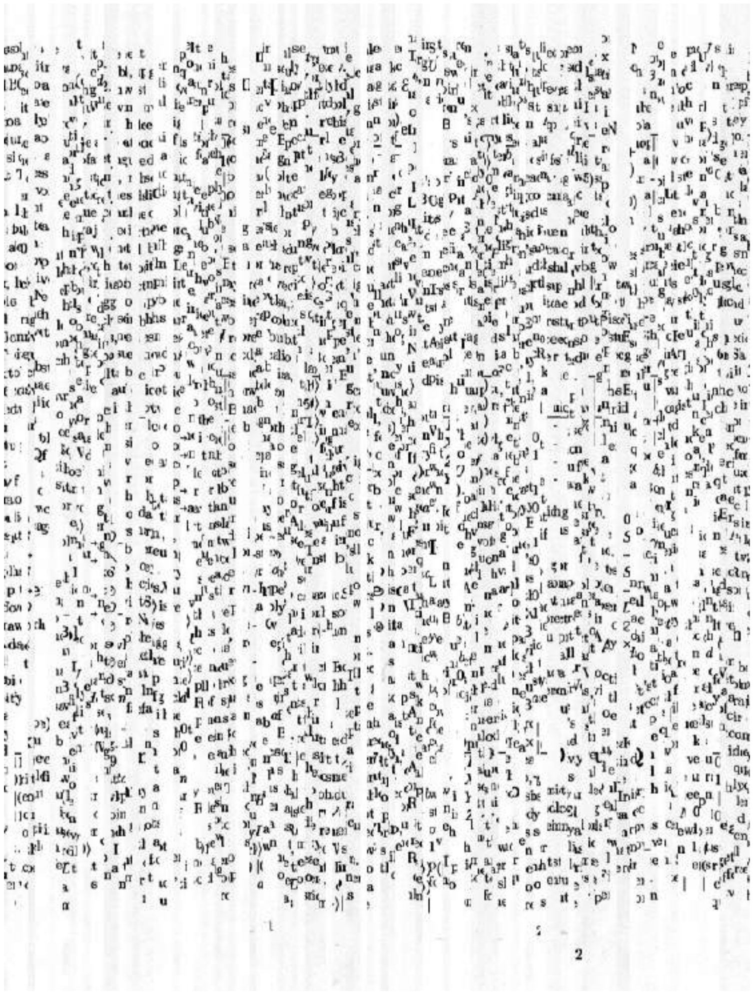
Sorting it out

The data sets in the last examples had at most a few hundred strips. Suppose that we had a few thousand strips, or even a few more orders of magnitude? As I discussed earlier, computing and storing all the pairwise scores demands resources proportional to the square of the number of pieces, and that's a number that grows quickly.

To work with large data sets I scanned in the PDF pages of text from my column on quantum computing. I didn't use the published pages with the figures on them because I wanted to simulate the kind of text-only documents people shred in offices. There were 51 pages in all, with text in the middle six inches of the page, giving me 306 inches of text. Cutting those pages into 0.125-inch wide strips gave me 2,448 strips to reassemble, which I call the text data set. Figure 17 (next page) shows a piece of this data set in its original, random order.

All the pairwise scores for this data set would mean creating and storing about 6 million scores. The time it takes to compute D_3 for any given pair of strips depends on the strips' size and the processor speed, but when you get into these large numbers all sorts of practical issues like those involving finite computer memory—which didn't matter much before—can become significant. Although we can imagine handling 6 million scores, if we had 250 pages to reassemble, that would be 289 million scores and things would be getting out of hand by that point.

My approach was to sort the pieces that had a reasonable likelihood of belonging together into smaller bins. This is a simple version of a general technique known as multiresolution processing. Once the pieces were placed into bins, I could apply the clustering algo-



17 Some of the 2,448 strips making up the text-only text data set.

rithm of the last section just to the contents of the bins. For example, using just 10 bins for the text data set results in about 244 strips per bin, which is even smaller than the image-based data sets I used (when each image was cut into 80 strips, the algorithm had 320 strips to assemble). Those data sets assembled in about a second on my little home PC using unoptimized code. If I had 250 pages (or about 12,000 strips) then using 60 bins would let me assemble the entire database in about a minute.

Note that when a piece goes into a bin, we indicate whether it's the left or right edge of the piece that has landed it there. A piece might appear in more than one bin if its left side matches one group and its right side another. Keeping this detail in mind, I'll just speak of strips and bins from now on.

The trick to this approach is to ensure that we get the right pieces into the bins. This is a question of heuristics, or approaches that seem to make sense but might not have a deep theoretical foundation. Basically I thought about what tests might be good at picking up features between strips and tried them.

I experimented with several different sorting heuristics, using all three data sets. My first, and simplest, approach was to average all the colors along the strip's side and take that color's luminance. I set up as many bins as I wanted, each holding a range of luminance from 0 to 1. Because many of the text pieces were near mid-gray in average intensity, I used big bins for the brightest and darkest ends and smaller ones in the middle. For example, one bin might hold all sides with luminance in the range (0.0, 0.15) while one closer to the middle of the range might hold the range (0.45, 0.46). I call this sorter *L*, for luminance.

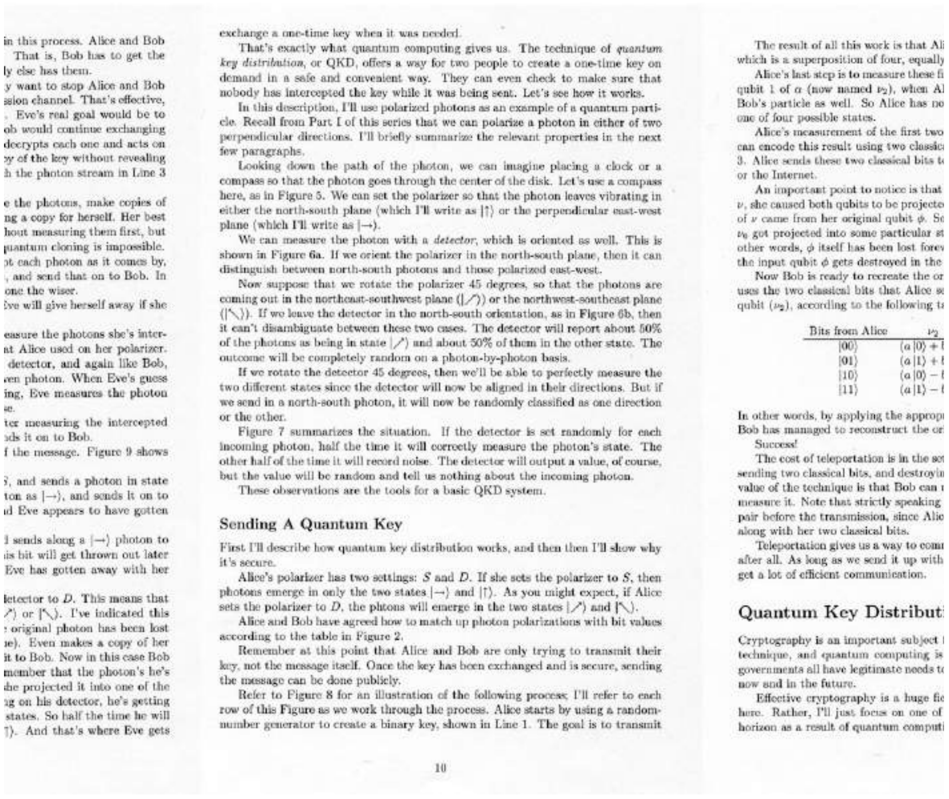
That test works well, but the middle bins fill up quickly and sometimes close neighbors fall into different bins. That means that pieces that should match never get the chance to see each other. I therefore opened up the range of each bin so that they overlap. For example, the first two bins might now cover (0.0, 0.2) and (0.1, 0.3) while a couple of bins near the middle could hold (0.445, 0.465) and (0.455, 0.475). A factor *f* that the user can set determines how much the bins overlap. I call this overlapping luminance sorter L_f.

The sorters *L* and L_f group text-based pages based on the density of black ink to white paper, but they don't take into account how the patterns fall on the page. I implemented a run-length metric that tries to get at some of this information. I start at the top of the page and count the lengths of continuous sequences of the same color. For full-color images such as those in my previous data sets, this isn't a useful metric, but it's well suited to text. I save the length and color of the longest run and the arithmetic average of all the runs. I use these values in different ways to create a few different sorters. One useful value comes from multiplying the longest-run color by its length, taking its luminance, and dividing by the average run length. As before, I have a variety of overlapping bins for the result. I call this overlapping run-length sorter R_f. In another sorter I use just the luminance of the color in the longest run, giving me sorter S_f.

These sorters all have different strengths, and I'm sure you could cook up any number of additional sorters that are sensitive to different aspects of the data that you might be interested in. In practice, I use them all. So each strip gets run through each sorter and placed into a bin from that sorter. Then I score all the strips in each bin for all the sorters.

With that data in hand, I assemble the clusters by looking first for the lowest scored pair among all the pairs in all the bins, then the next lowest-scored pair, and so on. That way if one of the sorters misses out on connecting two pieces that really belong together, one of the other sorters has a chance to catch it.

Between these four metrics and the clustering step, the narrow and wide image sets assembled perfectly. The text data set went together almost perfectly—out



in this process. Alice and Bob
That is, Bob has to get the
ly else has them.
y want to stop Alice and Bob
ssion channel. That's effective,
. Eve's real goal would be to
ob would continue exchanging
decrypts each one and acts on
oy of the key without revealing
h the photon stream in Line 3

e the photons, make copies of
ng a copy for herself. Her best
hout measuring them first, but
quantum cloning is impossible.
ot each photon as it comes by,
, and send that on to Bob. In
one the wiser.
Eve will give herself away if she

easure the photons she's inter-
at Alice used on her polarizer.
detector, and again like Bob,
sen photon. When Eve's guess
ing, Eve measures the photon
se.
ter measuring the intercepted
nds it on to Bob.
f the message. Figure 9 shows

S, and sends a photon in state
ton as $|\rightarrow\rangle$, and sends it on to
id Eve appears to have gotten

d sends along a $|\rightarrow\rangle$ photon to
is bit will get thrown out later
Eve has gotten away with her

detector to D. This means that
$\nearrow\rangle$ or $|\nwarrow\rangle$. I've indicated this
e original photon has been lost
ne). Even makes a copy of her
it to Bob. Now in this case Bob
member that the photon's he's
she projected it into one of the
ng on his detector, he's getting
states. So half the time he will
$\uparrow\rangle$). And that's where Eve gets

exchange a one-time key when it was needed.

That's exactly what quantum computing gives us. The technique of *quantum key distribution*, or QKD, offers a way for two people to create a one-time key on demand in a safe and convenient way. They can even check to make sure that nobody has intercepted the key while it was being sent. Let's see how it works.

In this description, I'll use polarized photons as an example of a quantum particle. Recall from Part I of this series that we can polarize a photon in either of two perpendicular directions. I'll briefly summarize the relevant properties in the next few paragraphs.

Looking down the path of the photon, we can imagine placing a clock or a compass so that the photon goes through the center of the disk. Let's use a compass here, as in Figure 5. We can set the polarizer so that the photon leaves vibrating in either the north-south plane (which I'll write as $|\uparrow\rangle$) or the perpendicular east-west plane (which I'll write as $|\rightarrow\rangle$).

We can measure the photon with a *detector*, which is oriented as well. This is shown in Figure 6a. If we orient the polarizer in the north-south plane, then it can distinguish between north-south photons and those polarized east-west.

Now suppose that we rotate the polarizer 45 degrees, so that the photons are coming out in the northeast-southwest plane ($|\nearrow\rangle$) or the northwest-southeast plane ($|\nwarrow\rangle$). If we leave the detector in the north-south orientation, as in Figure 6b, then it can't disambiguate between these two cases. The detector will report about 50% of the photons as being in state $|\nearrow\rangle$ and about 50% of them in the other state. The outcome will be completely random on a photon-by-photon basis.

If we rotate the detector 45 degrees, then we'll be able to perfectly measure the two different states since the detector will now be aligned in their directions. But if we send in a north-south photon, it will now be randomly classified as one direction or the other.

Figure 7 summarizes the situation. If the detector is set randomly for each incoming photon, half the time it will correctly measure the photon's state. The other half of the time it will record noise. The detector will output a value, of course, but the value will be random and tell us nothing about the incoming photon.

These observations are the tools for a basic QKD system.

Sending A Quantum Key

First I'll describe how quantum key distribution works, and then then I'll show why it's secure.

Alice's polarizer has two settings: S and D. If she sets the polarizer to S, then photons emerge in only the two states $|\rightarrow\rangle$ and $|\uparrow\rangle$. As you might expect, if Alice sets the polarizer to D, the phtons will emerge in the two states $|\nearrow\rangle$ and $|\nwarrow\rangle$.

Alice and Bob have agreed how to match up photon polarizations with bit values according to the table in Figure 2.

Remember at this point that Alice and Bob are only trying to transmit their key, not the message itself. Once the key has been exchanged and is secure, sending the message can be done publicly.

Refer to Figure 8 for an illustration of the following process; I'll refer to each row of this Figure as we work through the process. Alice starts by using a random-number generator to create a binary key, shown in Line 1. The goal is to transmit

10

The result of all this work is that Al
which is a superposition of four, equally
Alice's last step is to measure these f
qubit 1 of α (now named ν_2), when Al
Bob's particle as well. So Alice has no
one of four possible states.
Alice's measurement of the first two
can encode this result using two classic
3. Alice sends these two classical bits t
or the Internet.
An important point to notice is that
ν, she caused both qubits to be project
of ν came from her original qubit ϕ. So
ν_0 got projected into some particular st
other words, ϕ itself has been lost forev
the input qubit ϕ gets destroyed in the
Now Bob is ready to recreate the or
uses the two classical bits that Alice s
qubit (ν_2), according to the following t

Bits from Alice	ν_2
$\lvert 00\rangle$	$(a\lvert 0\rangle +$
$\lvert 01\rangle$	$(a\lvert 1\rangle +$
$\lvert 10\rangle$	$(a\lvert 0\rangle -$
$\lvert 11\rangle$	$(a\lvert 1\rangle -$

In other words, by applying the approp
Bob has managed to reconstruct the or
Success!
The cost of teleportation is in the se
sending two classical bits, and destroyin
value of the technique is that Bob can
measure it. Note that strictly speaking
pair before the transmission, since Alic
along with her two classical bits.
Teleportation gives us a way to com
after all. As long as we send it up with
get a lot of efficient communication.

Quantum Key Distribut

Cryptography is an important subject
technique, and quantum computing is
governments all have legitimate needs t
now and in the future.
Effective cryptography is a huge fi
here. Rather, I'll just focus on one of
horizon as a result of quantum comput

18 Results of running the multiple-bin sorters on the text data set, followed by clustering.

of 2,480 strips, only 15 strips weren't properly matched up on one side or the other. Figure 18 shows a piece of the results. The complete output just looks like a bunch of pages of text side-by-side, as it should.

Heuristics

A few other heuristics could be added to this system. For example, we might know that we were scanning in pages of text on standard American-sized letter paper. Printers generally need at least a half-inch margin on both sides of the text. If we found that we were making a cluster that was more than about 7.5-inches wide, we could suspect an error and try to find a place to split it apart.

We could also try to detect, and weight, specific image features. For example, many office memos, particularly involving numbers, have horizontal and vertical lines and borders. We could look for such features and give them extra weight in the sorting step.

I mentioned earlier that I assumed that all the strips were scanned in so that they're in the proper orientation. In the case of text, we could precede the matching process with a step of optical character recognition (OCR) on the strip in both its original orientation and after rotating it 180 degrees. The OCR process tries to recognize letters and is likely to find better matches in one orientation rather than the other.

In the case of images, the problem is a little harder. I haven't implemented it, but I think that you could get pretty far by scoring all the strips in both orientations and then using whichever orientation produced lower scores overall.

Crosscuts and jigsaw puzzles

So far I've only spoken about strips, but the general approach also works for irregularly shaped pieces.

As I mentioned, some shredding machines cut the strips into smaller pieces. At first, this may seem to be a straightforward problem. Just build a few more tables and match all four sides of the rectangular strips rather than just the left and right sides.

There's a problem, though, that results from the fact that there's a lot of white space on a page of text. The top and bottom of many of these crosscut strips will be white (since that edge will fall between lines of text). Because these edges are all completely white, they will have near-perfect scores of zero (or almost zero). Of course, we can't just line up these pieces vertically.

My approach is to look for edges that are the same color, within some threshold. If an edge is a single color (or almost a single color) then it doesn't take part in the matching process. That edge is essentially ignored and doesn't influence the scoring, sorting, or clustering processes.

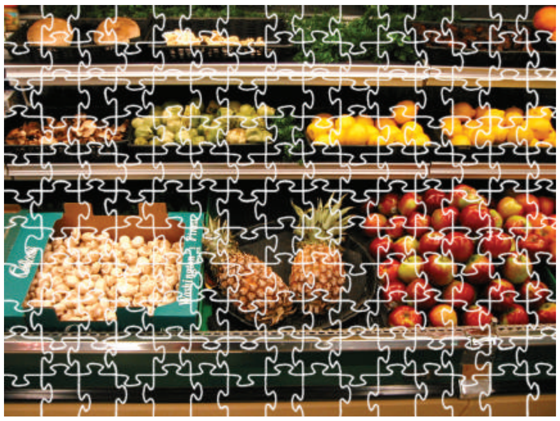
Using this test, I matched my text data using strip lengths of 1.25 inches (I simulated cutting each 11-inch strip into 8 pieces of 1.25 inches each and one piece of 1.5 inches), resulting in 22,032 pieces. I applied the sorting and clustering techniques and got good, but not perfect, results. I think a better sorting metric would go a long way here.

The sorting technique is amenable to the type of multiresolution methods I mentioned earlier, where you match things quickly at a coarse level, apply a more expensive test to refine the collection, then an even more expensive test, and so on, until you finally apply the most complete and accurate test only to a few candidates with a high likelihood of matching. We could certainly use a sequence of sorting steps, where the first few are crude and fast and only become more accurate (and slower) as the number of elements in the ever-smaller bins becomes tractable.

A fun application of this technique is to solve jigsaw puzzles. To test this out, I took a photograph and drew jigsaw-shaped pieces on it in Adobe Photoshop. I then wrote a program that broke the pieces apart and scattered them. Figure 19 (next page) shows some of the pieces.

To reconstruct the image, I used my assembly algorithm. I replaced the idea of a strip's side with the color values running around the perimeter of one side of the piece. This 165-piece puzzle, shown in Figure 20 (next page), assembled perfectly in just a few seconds. This really surprised me, since I was looking forward to writ-

19 Some pieces of a jigsaw puzzle.


20 The assembled 165-piece jigsaw puzzle.

ing some code to account for the pieces' shape information. Finding a nice characterization of the shapes so that the algorithm could match them quickly on a purely geometric basis would be an interesting project.

Putting the pieces together

In this column I've overlooked a few practical problems. When reconstructing a document, sometimes strips will be missing. Luckily, the clustering algorithm doesn't care about that and will just match up pieces as well as possible. Missing crosscut pieces are more troublesome and will result in irregularly shaped chunks of the original images.

I've also assumed that the pieces fit together without gaps or overlaps. If a vase shatters on the floor, you often get tiny little pieces and some dust. The big shards will fit together, but the joins will be imperfect because of the lost material. Archaeologists and historians often have to cope with these kinds of shattered, and possibly incomplete, artifacts. A great example of this problem is the Forma Urbia Romae, a shattered marble map of ancient Rome (see http://graphics.stanford.edu/projects/forma-urbis). It would be interesting to find algorithms to reconstruct these objects.

If you're putting a vase back together again, one thing that helps is that the vase has some thickness. The shapes go together not just based on a 2D outline, like a jigsaw puzzle piece, but also in terms of how the edges are scalloped and shaped. It would be great to find ways to take advantage of that information, too. ■

Readers may contact Andrew Glassner by email at andrew@glassner.com.