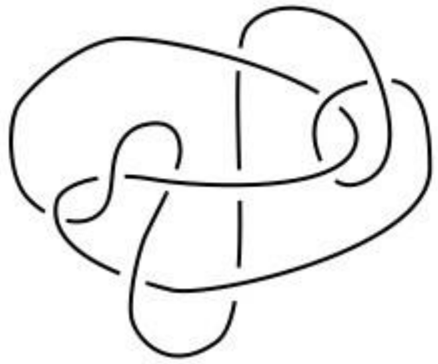
# Unknotting a PCB

**Peter Balch**

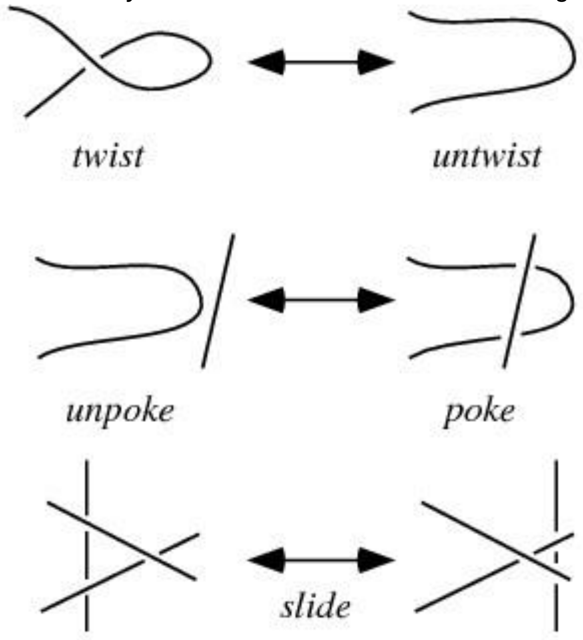
The "string" represented in this diagram forms a single loop. Is the loop knotted or not? (You will, of course, recognise this knot as The Culprit.)



In other words, can you untangle the string using "legal" moves so that it forms a simple loop with no crossings?

A move is legal if the string does not pass through itself.

Traditionally there are considered to be three legal Reidemeister moves



and all unknotted loops can be untangled using a combination of these moves.

The problem is that no-one seems know whether there is an algorithm to untangle an unknotted loop that can be performed in "polynomial time" rather than, say, exponential time. Is there an algorithm that is

$O(N^k)$
rather than
$O(k^N)$ where k is a constant and N is some measure of the "complexity" of the tangled string.

It is not even known whether a tangle can be recognised as a unknotted loop in polynomial time. Clearly, untangling is a form of recognition so if you can untangle it in polynomial time, you can recognise it in polynomial time.

If N is measured as the number of crossings then the first two of the Reidemeister moves reduce the complexity. The third move doesn't.

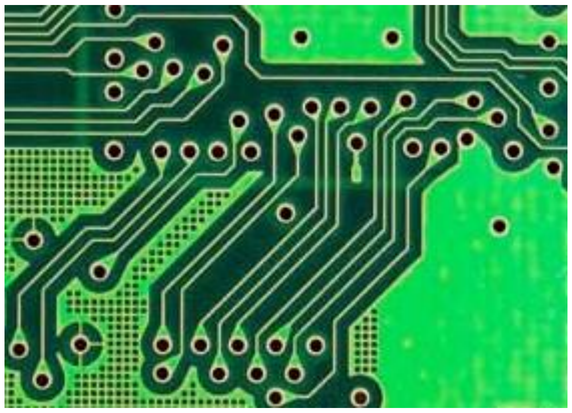
If one could find a sequence of moves that always reduces the complexity then the algorithm would run in polynomial time. Unfortunately, the first example knot (above) cannot be untangled using just the first two of the Reidemeister moves. You have to search all the possible combinations of the third move and that requires exponential time.

So, is there a representation of the knot with a better measure of complexity such that the "legal moves" always reduce the complexity?

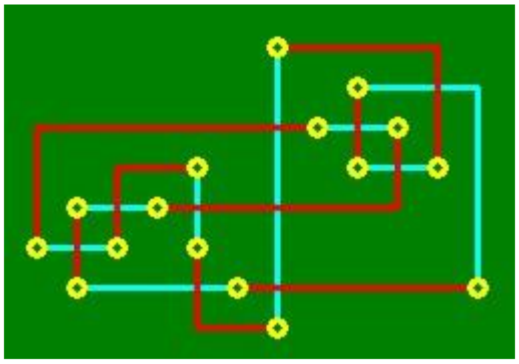
I propose one here. And I demonstrate an untangling algorithm which runs in polynomial time - maybe.

## A Double Sided PCB

The commonest form of PCB (printed circuit board) has two layers called "sides" - top and bottom. On each side there are copper tracks. The tracks on one side cannot touch or cross. Tracks on opposite sides are joined by "Vias". A Via is a hole plated with copper to form a connection



Here is The Culprit rendered as a double sided PCB:



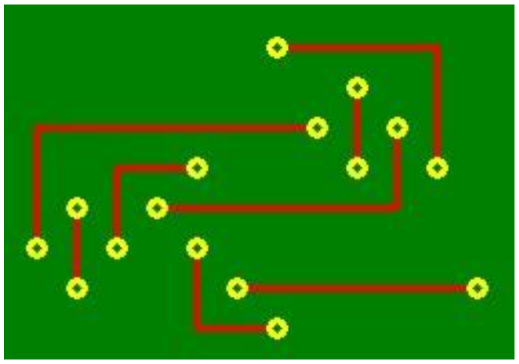
The red tracks are on the top side of the board. The blue tracks are on the bottom side of the board. The Vias are yellow. A red track always passes over a blue track.

Complexity can be measured as the number of Vias. If we can reduce the number of Vias to zero then we have untangled the knot and shown it is a simple loop.

## The Legal Moves

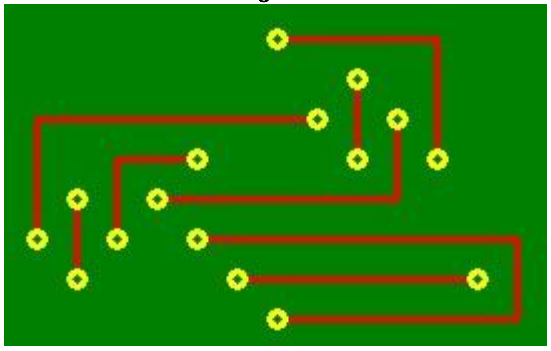
What are the legal moves?

Consider just one side of the PCB, e.g. the top side:



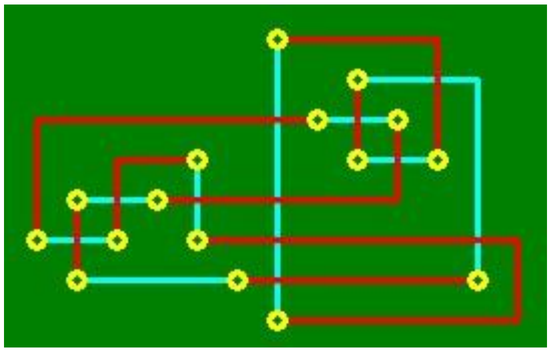
A "track segment" is the copper that runs from one Via to another.

It is legal to run any track segment anywhere so long as you don't move the Vias. In PCB design, this is called "rerouting". For instance we can reroute the lowest segment:



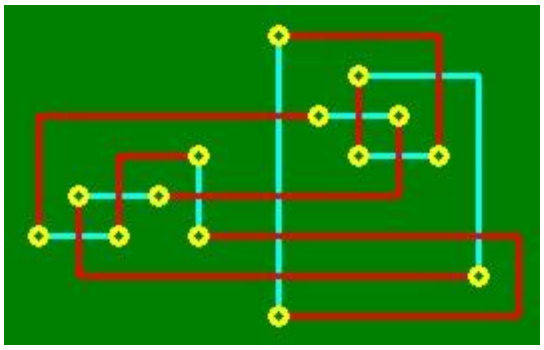
(Rerouting a single segment is legal because we can imagine the pcb as a piece of cardboard with holes in it. The surfaces of the card are slightly sticky. A string is pressed onto the card and can pass through the holes in the card. The string follows the same path as the copper on the PCB. Rerouting is equivalent to peeling a part of the string off the card and laying it back down somewhere else. The string does not pass through itself or through the card and so rerouting is legal. Rerouting several tracks at the same time is not legal, e.g. rip-up two tracks and then route both of them.)

Now lets look at both sides of the PCB:



The blue segment towards the bottom left has no crossings. It is legal to move a segment with no crossings onto the other side of the PCB. (it is legal because the "string" does not pass through itself.) I'll call that "simplifying".

So the PCB becomes:



We have reduced the complexity because the number of Vias has decreased from 16 to 14.

(Considering Vias is similar to Horst Schubert's approach of "bridges". However I cannot find that Schubert actually implemented a polynomial-time algorithm. Rerouting is similar to Raymond Robertello's "pass move" except that I only accept reroutes then reduce the number of Vias or crossings.)

## Rounds of Moves

The algorithm consists of a sequence of "rounds" each round consists of several "moves".

If I can show that

- each move executes in polynomial time
- each round consists of a polynomial number of moves

- after each round the complexity has always decreased

·the algorithm terminates when the complexity is zero then I will know that the whole algorithm executes in polynomial time.

There are three kinds of Round

- ·Easy
- ·Tricky
- ·Hard

## Easy Rounds

```
An Easy round consists of for
    every track segment
        reroute it
        if any track segment has no crossings
            simplify it
        if the complexity has not been reduced
            undo the rerouting
```

If at the end of the Easy round, there has been at least one simplification then execute another Easy round.

"Undo the rerouting" means: take a backup copy of the PCB before starting the move; to undo the rerouting, restore from the backup; otherwise discard the backup.

I now need to define "Complexity" and "Rerouting".

Complexity:

- ·PCB A is less complex than PCB B if A has fewer Vias
- ·PCB A is more complex than PCB B if A has more Vias
- ·If the number of Vias is the same
  - ·PCB A is less complex than PCB B if A has fewer crossings

So, to summarise, I'm trying to reduce the number of Vias but, if I can't do that, I try to reduce the number of crossings; if I can't do that either then I discard the changes I've just made and move onto the next segment. When I've tried every segment and haven't made any changes then the Easy rounds have terminated.

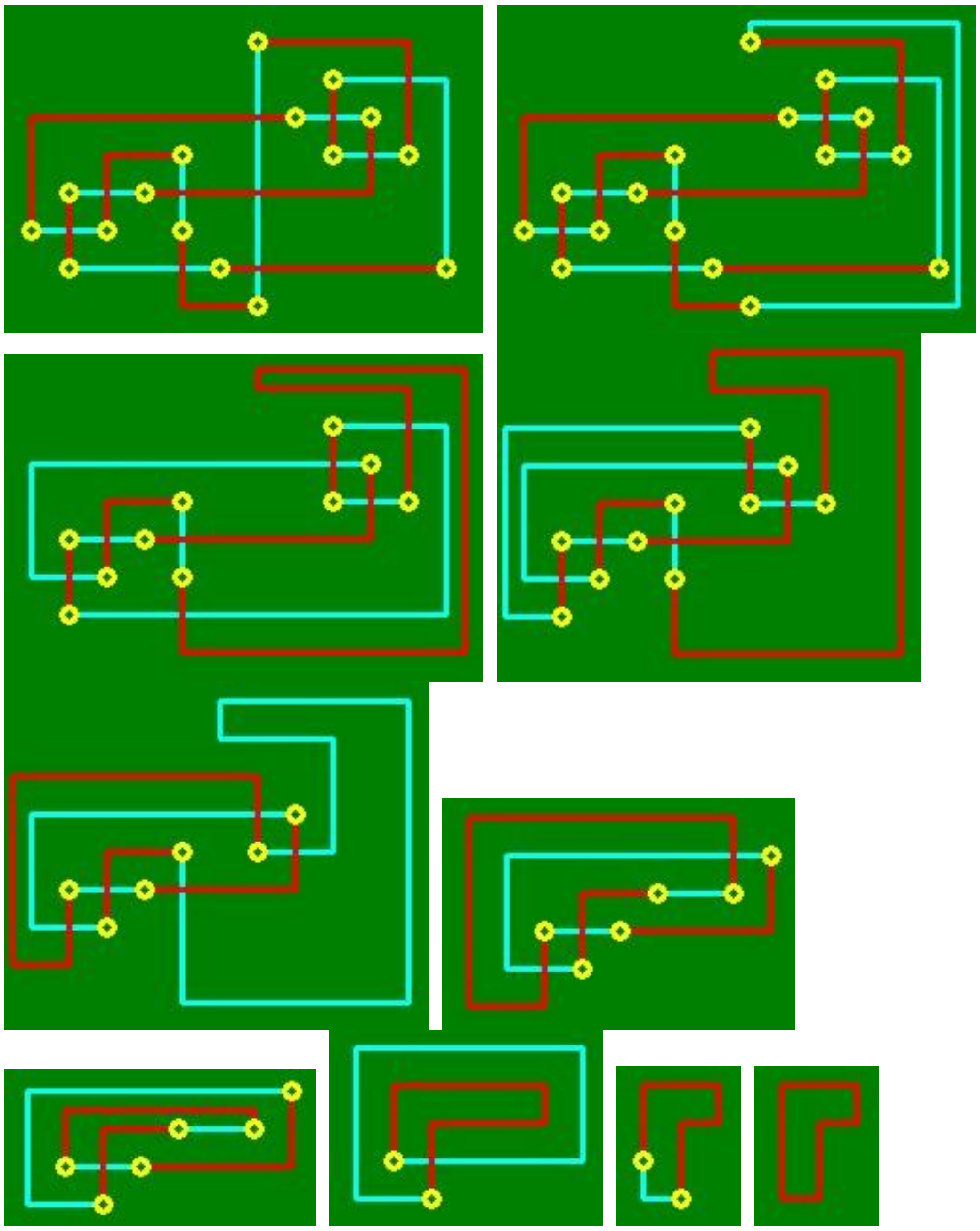
Rerouting uses the Lee algorithm. The basic Lee algorithm finds a path from the start Via to the finish Via by "wave propagation". It marks every "cell" of the PCB as empty or blocked (by a track) then marks the start Via with a 0. A cell will contain the manhattan distance from the start. The algorithm then repeatedly marks every non-blocked cell with 1 plus the minimum value (i.e. distance) of its 4 non-empty, non-blocked neighbours. When the finish Via has been marked, the route back to the start is a line of decreasing cell values. (Many descriptions of the Lee algorithm exist on the web. Refer to them for a clearer explanation.)

In the diagrams above, the tracks are two cell widths apart. That leaves enough space for a rerouted track to run between two existing tracks.

I use a modified Lee algorithm. The "distance" measure of the Lee algorithm is replaced by a "cost". As usual, the cost increases by 1 for every cell traversed but in addition the cost increases by 10,000 for every track crossed on the other side of the board. In other words, you're allowed to cross a track on the other side but I'd prefer it if you didn't.

[The value of 10,000 is sufficiently large that it is greater than any neighbouring distance the algorithm is likely to encounter. It would be better to use a larger value if very much more complex knots are to be untangled but the algorithm must run on a computer with a finite integer size. If necessary, I could use 64-bit integers or the cost could consist of two parts: distance and crossings.]

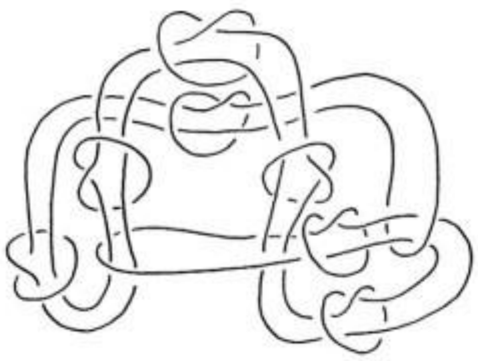
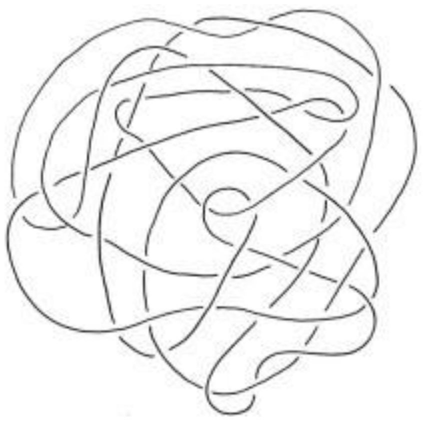
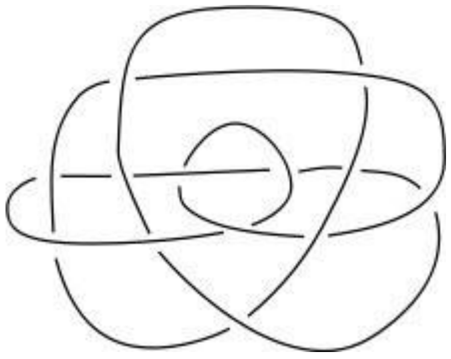
Here is a sequence of moves that untangle The Culprit. A segment is rerouted, this exposes other segments that are then Simplified. The distance between the tracks is then "Normalised" so that subsequent rerouting can occur. Normalisation consists of shifting the tracks left/right or up/down so they are at least 2 cell widths apart; any big redundant gaps are closed.



Easy rounds are sufficient to solve many of the knots I've found:

## Tricky Rounds

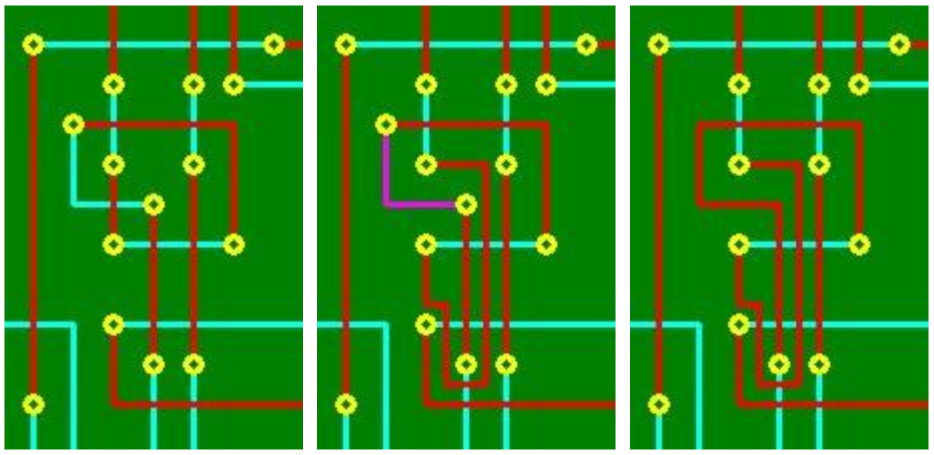
Unfortunately, Easy rounds are not sufficient for some knots:

```
A Tricky round consists of for
    every track segment A
        if segment A crosses exactly one segment (B)
            mark segment A as Keep Out
            reroute segment B
            if the reroute was successful
                simplify segment A
            else
                undo the rerouting
```

So a Tricky move tries to force a segment that has exactly one crossing to have no crossings. If it succeeds then the segment can be simplified - i.e. the Vias at either end can be removed.

Here is a part of one of the knots shown above.



The L-shaped blue segment has exactly one crossing. It is marked as Keep Out (purple). The vertical red segment that crosses it is rerouted. The L-shaped blue segment can now be Simplified. (The PCB would then be Normalised to open-up space for more rerouting.)

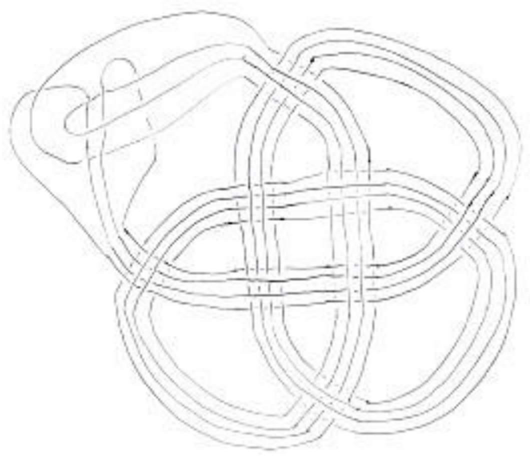
So the modified Lee algorithm must not only avoid existing tracks on the same side of the PCB, it must avoid Keep Out tracks on the <u>other</u> side of the PCB.

(It's called "Keep Out" because PCB design software allows you to designate keep-out area which automatic track routing and area-fill must avoid.)

Easy and Tricky rounds alone are sufficient to untangle almost every example knot I've attempted. However, Tricky rounds are slower than Easy rounds so I try Easy rounds first then a Tricky round, then Easy rounds, etc. until there are no Vias left or a round fails.

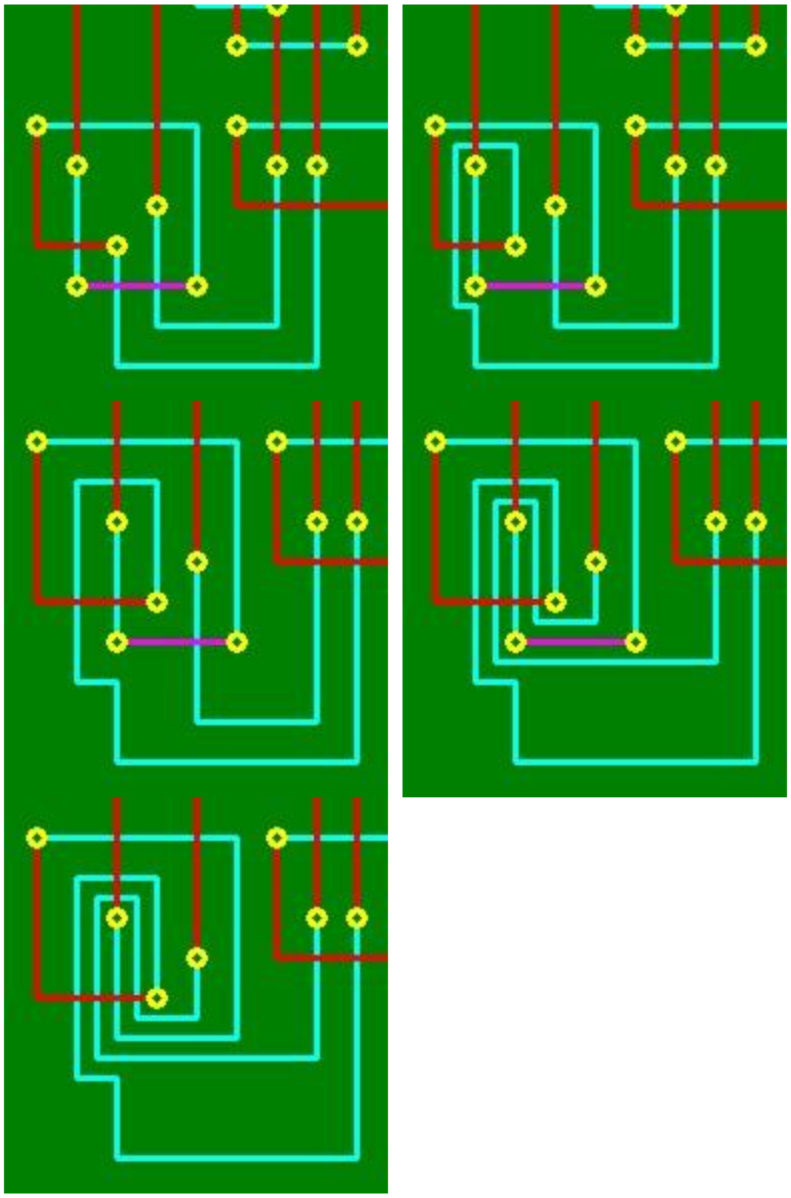
## Hard Rounds

This knot cannot be solved with just Easy and Tricky moves:



It requires a further kind of move called a Hard move.

A Hard round is like a Tricky round except that the Keep Out segment can have several crossings. All the tracks that cross the Keep Out segment must be rerouted. If any of them cannot be rerouted then the move has failed and the PCB is restored to how it was at the start of the move.



## Moves and Rounds

The algorithm consists of a sequence of "rounds" each round consists of several "moves". The Rounds repeat until no further changes have been made. If no Easy, Tricky or Hard moves can be made then the algorithm is unable to untangle the knot.

That leaves the question of what is the most efficient order in which to call the different moves.

Easy moves are fastest, Tricky moves are slower, Hard moves are slower still. So I try Easy then Tricky then Hard rounds. If any round succeeds, we go back to Easy moves.

The program attempts an Easy move on every segment in turn, trying the segments again and again until no further changes can be made. If the knot has not been untangled then Tricky moves are tried.

The program attempts a Tricky move on every segment in turn, continuing round and round the segments until no further changes can be made. If any change has been made, it goes back to Easy moves. If the knot has not been untangled then Hard moves are tried.

The program attempts a Hard move on every segment in turn. If it succeeds on any segment, it then tries a round of Easy moves, as before, then Tricky, etc. If it can make no Hard moves then the knot cannot be untangled.

So the program doesn't try all possible Hard moves before returning to Easy moves. It has been found that stopping after a single Hard move tends to solve tangles in around 60% of the time of trying all possible Hard moves. (That's what's done in Ver 1.2 and later versions of the program.) It doesn't enable the algorithm to solve different knots - it just solves them faster.

I have yet to find an unknotted loop knot that cannot be untangled with Easy, Tricky and Hard rounds. The algorithm correctly spots that trefoil, figure-of-eight, etc. knots cannot be untangled; after all, there is no combination of legal moves that can untangle a trefoil.

## The Order of the Algorithm

A round consists of visiting every track segment once - that's called a Move. A track segment is bounded by Vias. Complexity can be measured by the number of Vias. Therefore a round consists of O(N) number of moves.

Rounds can be repeated but, at the end of each round, the complexity must have been reduced (otherwise the algorithm has failed. Therefore there are at most O(N) rounds - i.e. $O(N^2)$ moves.

Tricky and Hard moves are looking for crossings. For every segment, you consider every other segment to see if it crosses. Complexity can be measured by the number of crosses. Therefore a Tricky and Hard move is O(N3).

A simplification consists of visiting every track segment once. For every segment, you consider every other segment to see if it crosses. Therefore a simplification consists of $O(N^2)$ number of comparisons.

Rerouting uses the Lee algorithm. The order of the Lee algorithm is proportional to the a area of the PCB measured in cells. Normalisation ensures that the track segments are approximately 2 cells apart. The area of the PCB is therefore proportional to the number of segments and is approximately O(N) or, at worst, O(N2).

Normalisation consists of visiting every track segment once and making a list of which rows and columns of cells are in use. Complexity can be measured by the number of straight tracks. Normalisation is therefore proportional to the number of segments and is approximately O(N) or, at worst, O(N2).

So the entire algorithm executes in polynomial time. There are at worst $O(N^2)$ moves and each move is at worst O(N3) so the overall algorithm executes at worst with O(N5).

## Can It Untangle Them All?

Can the algorithm untangle all tangled unknotted loops?

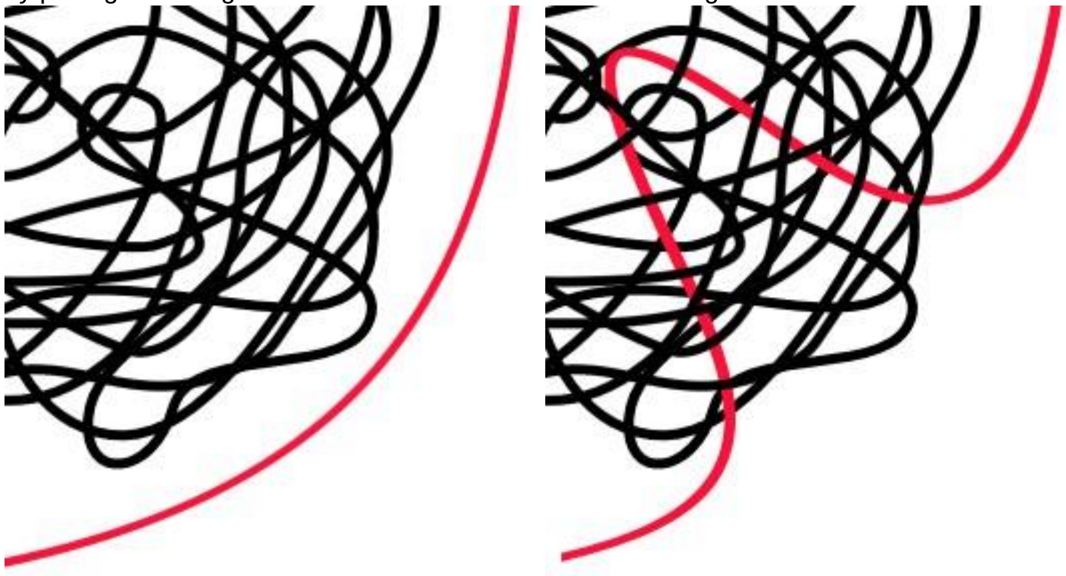
I don't know.

I cannot find a counter-example.

Consider starting with a simple loop and applying Reidemeister moves in order to tangle it. Can the algorithm untangle it? What I would like is to show that the algorithm will always be able to undo every combination of Reidemeister moves.

But I haven't worked out how to prove that yet.

In my imagination, a proof would show that the only way to increase the complexity of a tangle is by poking the string above and or below the rest of the string:



For instance, in the example above, the red string is moved by a combination of Reidemeister pokes and slides. All we need to do to untangle it is to move the black strings aside.

And that can be done by marking the red string as Keep Out and re-routing the black strings.

We don't need to move them all aside at once. The red string got tangled quite slowly by doing one Reidemeister move at a time so we ought to be able to untangle it one Reidemeister move at a time.

But it's not obvious that can be done with a combination of Easy, Tricky and Hard operations that always decrease the complexity at the end of each round.

## Windows program

A Windows program is available that allows you to draw knots and manually or automatically untangle them. Email me for a copy.

Peter Balch EH9 1DX, UK
peterbalch@btinternet.co
m