In particular, the smallest *square* cover for $\mathcal{T}$ has side $1/\sqrt{2}$ and area $1/2$.

Further questions There are many closely related interesting problems that might yield to the calculus and a little insight. Here are a few:

1. Find the smallest triangle similar to a given triangle that can cover every triangle of perimeter two, or every rectangle of perimeter two.
2. It is easy to see that every triangle of perimeter two has a circumscribed rectangle whose area is no larger than $2\sqrt{3}/9$, and since the equilateral triangle with perimeter two lies in no smaller rectangle, this constant is sharp. How *large* a circumscribed rectangle can one guarantee?
3. Find analogues of Theorem 1 and Theorem 2 in $\mathbb{R}^3$, in $\mathbb{R}^d$.

REFERENCES

1. O. Bottema, R. Ž. Djordjević, R. R. Janić, D. S. Mitrinović, and P. M. Vasić, *Geometric Inequalities*, Wolters-Noordhoff, Groningen, The Netherlands, 1969.
2. G. D. Chakerian and M. S. Klamkin, "Minimal covers for closed curves," this MAGAZINE 46 (1973), 55–63.
3. Hallard T. Croft, Kenneth J. Falconer, and Richard K. Guy, *Unsolved Problems in Geometry*, Springer-Verlag, New York, NY, 1991.
4. Steven Finch, "Moser's worm constant," URL: `www.mathsoft.com/asolve/constant/worm/`, 2000.
5. Zoltan Füredi and John E. Wetzel, "The smallest convex cover for triangles of perimeter two," to appear in *Geom. Dedicata*.
6. J. P. Jones and J. Schaer, *The Worm Problem*, University of Calgary, Department of Mathematics, Statistics, and Computing Science, Research Paper No. 100, Calgary, Alberta, Canada, 1970.
7. Jonathan Schaer and John E. Wetzel, "Boxes for curves of constant length," *Israel J. Math.* 12, 1972, 257–265.
8. John E. Wetzel, "The smallest equilateral cover for triangles of perimeter two," this MAGAZINE 70 (1997), 125–130.

Scooping the Loop Snooper

an elementary proof of the undecidability of the halting problem

No program can say what another will do.
Now, I won't just assert that, I'll prove it to you:
I will prove that although you might work till you drop,
you can't predict whether a program will stop.

Imagine we have a procedure called *P*
that will snoop in the source code of programs to see
there aren't infinite loops that go round and around;
and *P* prints the word "Fine!" if no looping is found.

You feed in your code, and the input it needs,
and then *P* takes them both and it studies and reads
and computes whether things will all end as they should
(as opposed to going loopy the way that they could).

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here's the trick I would use—and it's simple to do.
I'd define a procedure—we'll name the thing Q—
that would take any program and call P (of course!)
to tell if it looped, by reading the source;

And if so, Q would simply print "Loop!" and then stop;
but if no, Q would go right back up to the top,
and start off again, looping endlessly back,
till the universe dies and is frozen and black.

And this program called Q wouldn't stay on the shelf;
I would run it, and (fiendishly) feed it *itself*.
What behavior results when I do this with Q?
When it reads its own source code, just what will it do?

If P warns of loops, Q will print "Loop!" and quit;
yet P is supposed to speak truly of it.
So if Q's going to quit, then P should say, "Fine!"—
which will make Q go back to its very first line!

No matter what P would have done, Q will scoop it:
Q uses P's output to make P look stupid.
If P gets things right then it lies in its tooth;
and if it speaks falsely, it's telling the truth!

I've created a paradox, neat as can be—
and simply by using your putative P.
When you assumed P you stepped into a snare;
Your assumptions have led you right into my lair.

So, how to escape from this logical mess?
I don't have to tell you; I'm sure you can guess.
By *reductio*, there cannot possibly be
a procedure that acts like the mythical P.

You can never discover mechanical means
for predicting the acts of computing machines.
It's something that cannot be done. So we users
must find our own bugs; our computers are losers!

—Geoffrey K. Pullum
Stevenson College
University of California
Santa Cruz, CA 95064