Source cannot be re-found on the web.

2.2 Undecidable problems [TILING PROBLEM, HALTING PROBLEM]

~~Unfortunately, the hierarchy of difficulty at the end of the last section didn't tell the whole story. There are some problems which are so hard that they are beyond even the 'non-elementary' class, and which are effectively undecidable (outside of the class of decision problems considered here, such algorithmically insoluble problems are referred to as 'non-computable').~~

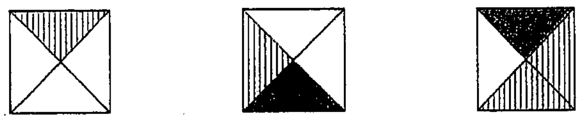
To say that a problem is 'undecidable' means that there is no way, even given unlimited resources and an infinite amount of time, that the problem can be decided by algorithmic means.

Some of these undecidable problems seem at first sight to be harmless variations on problems that whilst they may have an unreasonable time demand, are certainly decidable; ~~the situation parallels that seen in the last section, where a small-seeming variation on a tractable problem rendered it either provably to require super-polynomial time, or to be in one of the 'probably' intractable classes NP or NPC.~~

There are many well-known examples of undecidable problems: we will look at just two: **tiling problems**, and the canonical example of an undecidable problem, the **halting problem**.

Example 1: tiling problems

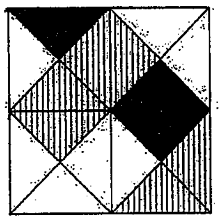
Consider the set of three tiles below:


(1) (2) (3)

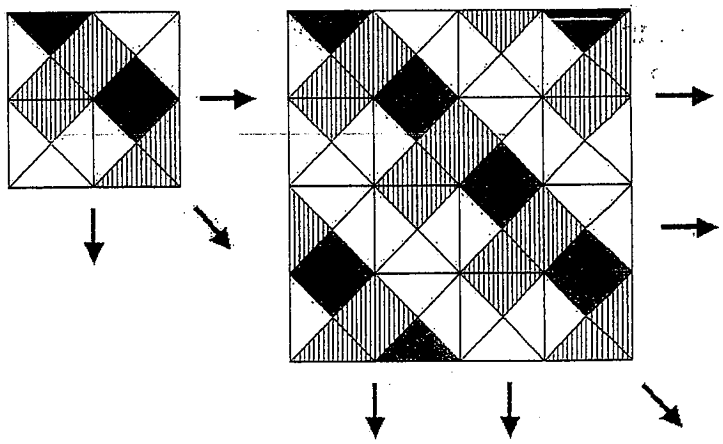
Could these be used to tile an arbitrary nxn area? The rules that have to be obeyed are:

- Only these three tile types can be used, but each can be used arbitrarily often.
- The edges of the tiles have to match up when they are used to cover an area.
- The tiles can't be rotated.

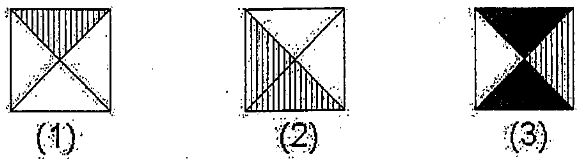
It's clear that they can when n=2:



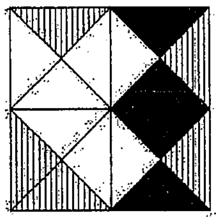
Also, in this simple case, one can see that the solution above extends easily to any nxn area.



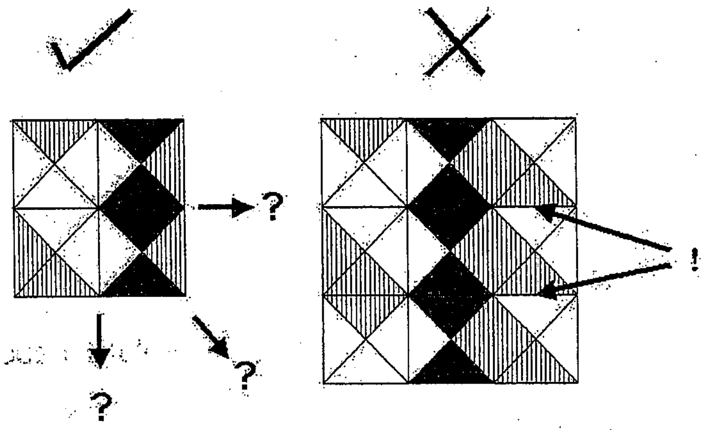
Now suppose that the bottom colours of tiles (2) and (3) are exchanged:



The new set of tiles can still cover a 2×2 area as below



but this particular solution can't be extended:



Moreover, there are no others which will work; this set can't tile any area of size greater than 2×2.

The most common forms of the 'tiling problem' are a generalisation of the problem above, with a supplied set T of tile types (which can't be added to or modified).

Bounded tiling problem:

Given a set T of tiles, can these be used to cover a specified n×n area?

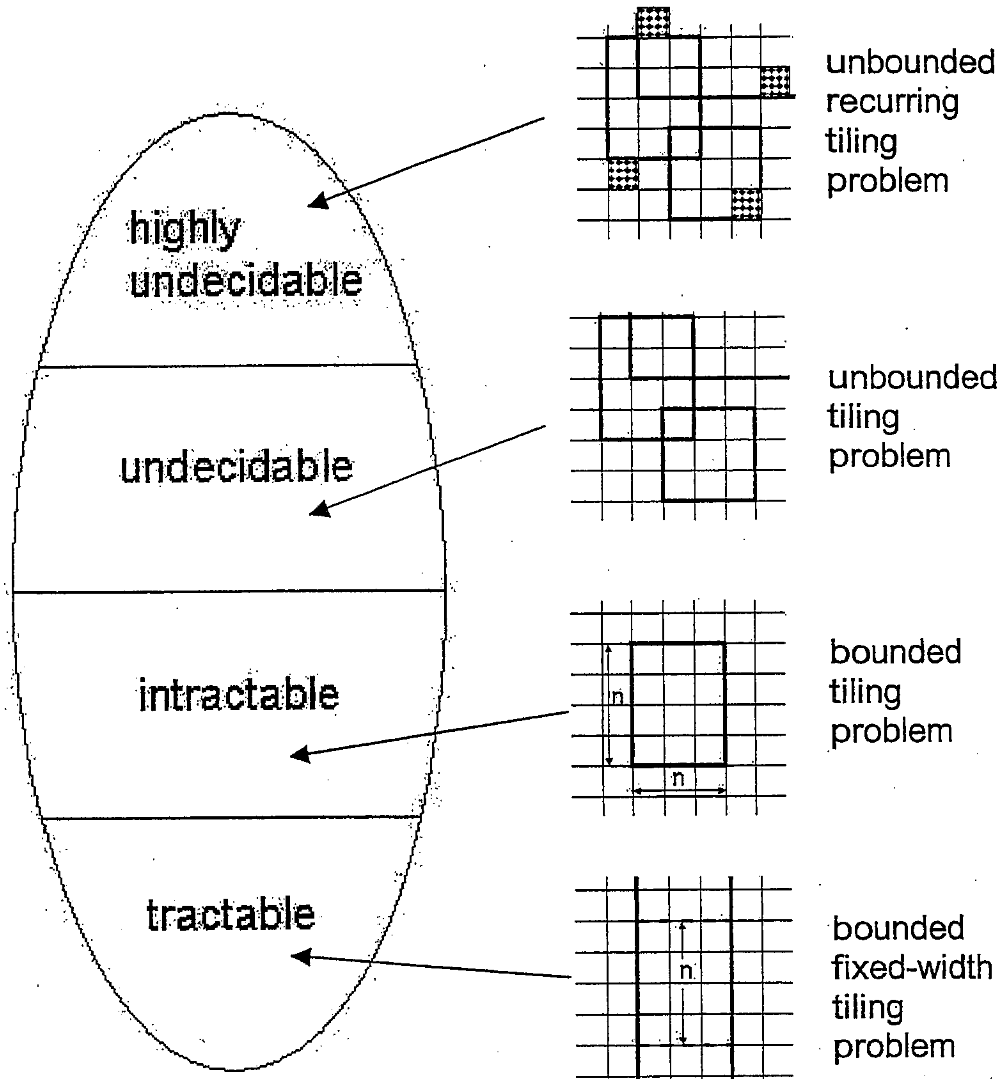
~~This problem is in fact in the set NPC: it has a short certificate (a putative tiling of the nxn area can be checked (to see that the 'matching edges' and 'no rotations' rules have been obeyed) in time in $O(n^2)$, so is in NP; the problem also can be shown to be such that $B \leq_p$ (bounded tiling) for some $B \in NPC$, so bounded tiling is NP-complete.~~ There appears to be no algorithm that can do significantly better than just checking through all possible arrangements of the tiles, something that takes an exponential amount of time.

Unbounded tiling problem:

Given a set T of tiles, can these be used to cover <u>any</u> n×n area? (Or equivalently, can they be used to tile the <u>entire integer grid</u>?)

This problem is much worse than bounded tiling; it is in fact undecidable, as the display of no finite tiled area can prove – in general, we aren't just talking about especially easy tile sets such as the one first considered above – that *any* area can be tiled.

However it would be wrong to assume that the essence of the difficulty was that the number of things that might need to be checked (all possible nxn areas) was infinite. It might have been that there was some rule that could look at a set of tiles and say "Yes, these have Property X, and so they can tile any area".


highly undecidable
undecidable
intractable
tractable
n
n
n
unbounded recurring tiling problem
unbounded tiling problem
bounded tiling problem
bounded fixed-width tiling problem

Example 2: the halting problem

Consider the program

```
while $x \neq 1$
    {
     x := x - 2
    }
```

For odd x (let's suppose for simplicity that the program's inputs are positive integers) this decrements by 2 until 1 is reached, when the program terminates. However, when x is even, decrementing by 2 each time 'misses' 1, and the program continues to decrease the original value forever.

Now consider

```
while $x \neq 1$
    {
     if even(x) then
         x := x/2
    else
         x := 2x + 1
    }
```

Again, there is no problem seeing for which inputs this terminates (this time, only for x's that are powers of 2), and for which it runs forever.

For the variation below, however

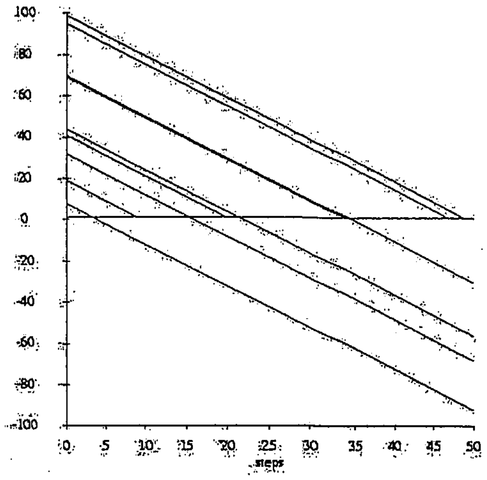
```
while $x \neq 1$
    {
     if even(x) then
         x := x/2
    else
         x := 3x + 1
    }
```

the situation is very different. Although in practice it does seem that this program terminates for all positive integers x – although sometimes reaching very high and unpredictable values before it does so – no-one knows why. Consequently it's impossible to say *with certainty* before this program is run on a new input whether it will terminate or not (and if it *appeared* for some input not to, how would we know whether it simply hadn't been given enough time?).

The examples below take random integers from {0,...,100} and run the three programs above for 50 steps, showing at each step the value reached.

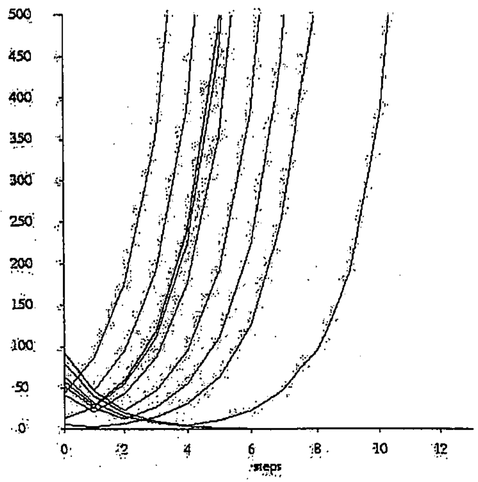
```
while x ≠ 1
    {
     x := x - 2
    }
```

(terminates for *odd x*)



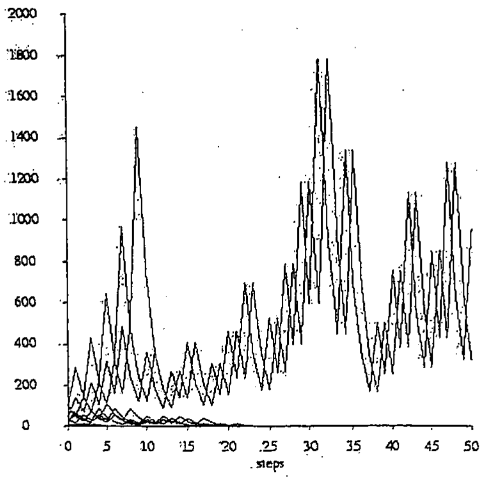
```
while x ≠ 1
    {
     if even(x) then
          x := x/2
     else
          x := 2x + 1
    }
```

(terminates when *x is a power of 2*)



```
while x ≠ 1
    {
     if even(x) then
          x := x/2
     else
          x := 3x + 1
    }
```

(terminates when ???)



(Note that all three of the programs have an infinite number of possible inputs (all positive integers), but for the first two cases we can say when the program will terminate without checking through all possible cases (in the first case 'Property X' is 'being an odd number', in the second it's 'being a power of 2'). So this is more evidence that the presence of an unbounded, potentially infinite input set, with a seeming necessity to check all cases, is not the fundamental factor that determines decidability.)

***The halting problem** is defined to be the problem, for any program R run on a **legal input X** (for example in the cases above that would mean X had to be a positive integer), of **deciding if R halts (terminates) on X**.*

Note that this is about *any* program, and not just some specific one (in the same way that the tiling problem was about *any* set of tiles T, not just some specific, and possibly easy to decide, set), but that the halting problem is not, once R has been chosen, asking whether R terminates on *all* inputs, just a given one, X.

The halting problem is the best known example of a problem that is not algorithmically decidable. Morever it has a special status because it's possible to prove its undecidability without reference to another undecidable problem – it plays the same role with respect to undecidability that PSAT plays with respect to NP-completeness.

Fortunately, it's a lot easier to explain why the halting problem is undecidable that it is to explain why PSAT is in NPC.