Complexity and Computability

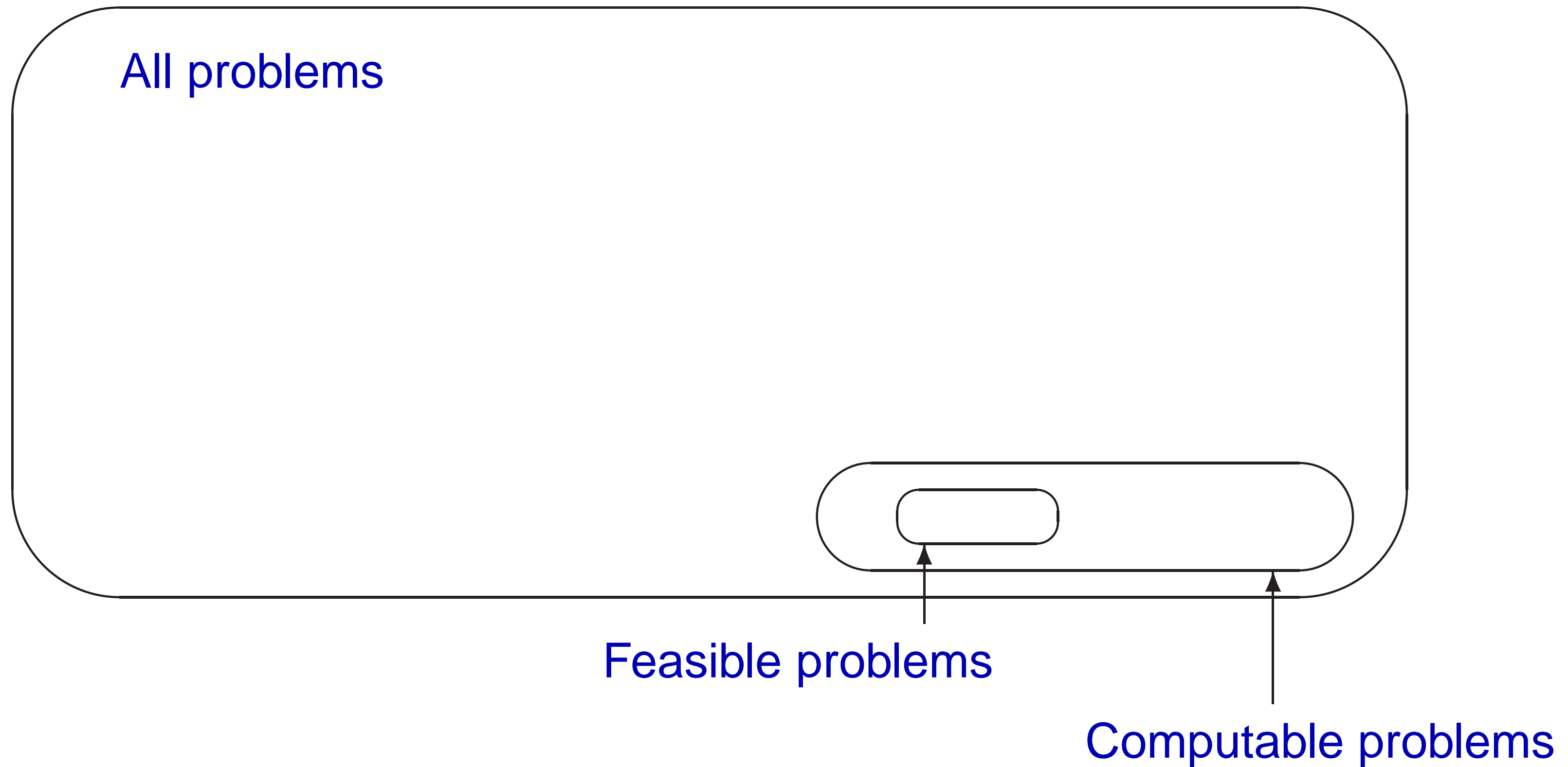
COMP2600 — Formal Methods for Software Engineering

Jeremy Dawson

Australian National University

Semester 2, 2011

The Universe of Problems



- Each of the inner sets is a tiny proportion of the set that contains it.

Jargon of Computability

- We talk of problems being *solvable*, functions being *computable* and questions being *decidable*.

  All such formulations of computability are equivalent.

- A question is *totally decidable* if an algorithm exists which will (ultimately) decide whether it is true or false.

  That algorithm is called a total decision procedure.

- A question is *partially decidable* if an algorithm exists which will ultimately say so if the answer is true but which may not terminate otherwise.

  That algorithm is called a partial decision procedure.

The Halting Problem is Undecidable

- Given a particular Turing Machine, and a description of the data on the tape, will that machine ever reach the Halt state?
- This is equivalent asking "will this program terminate", provided the program is written in a Turing Complete language.

- Proof of termination is *sometimes* possible.
- Proof of non-termination is *sometimes* possible.

- There is no total decision procedure that can decide whether an arbitrary Turing Machine, let loose on some data, will ever terminate.

Proof that the Halting Problem is Undecidable

- Uses proof by contradiction ($\neg$-introduction).

- Assume someone solved the Halting Problem.
- What we would have is a function, call it $halts$, that says whether program P when applied to data D:

$$halts(P,D) \equiv \begin{cases} yes & \text{if } P(D) \text{ halts} \\ no & \text{otherwise} \end{cases}$$

- Define a new function $paradox$ which makes uses of $halts$:

$$paradox(P) = \textbf{if } halts(P,P) \textbf{ then } \text{loopForever...} \textbf{ else } \text{halt}$$

- Does $paradox(paradox)$ halt?

  That is, does $halts(P,P)$ return yes, where $P = paradox$?

  There are 2 cases:

  yes: If so, then $halts(P,P)$ must have returned no – contradiction!

  no: If so, then $halts(P,P)$ must have returned yes – contradiction!

- Conclusion: the Halting Problem has no solution.

Features of the Halting Problem Proof

Two aspects commonly found in proofs of this sort
(eg also Russell's paradox)

- Self-application: involves applying P (as program) to P (as data)
- Diagonalization: we create $paradox$ which differs from *every* program.
  - Why?
    $paradox$ differs from P as it gives a different result on *some* input.
  - Which input?
    Program $paradox$ gives a different result from *program* P on *input* P
  - When $paradox$ is a program, it *can't* be *different* from *every* program

Non-Computable Problems

We showed the halting problem is non-computable.

It is also impossible to have an algorithm to say whether a function over the natural numbers is total (ie, halts on every input)

- Assume the contrary, ie, that we have a program $total$, where $total(P)$ returns yes if P halts on *every* input.
- Then, given P, D, define program P_D as follows: $P_D(I)$ ignores its input I and runs just like $P(D)$
- Then $halts(P,D) = total(P_D)$
- Thus we can write a program $halts$: given P, D it must construct P_D and then run $total(P_D)$
- As there is no such program $halts$, there can be no such program $total$

Partially Decidable Problems

- Of course, we could always just run the program to see if it halts.
- Simulating a TM is a partial decision procedure for the question: "Does that Turing Machine halt (on given input)".
- If it halts, we will know that it halted..
- A more complex question: "Does a given Turing Machine terminate on *some* input".

  This is also partially decidable: you have to run n steps of the computation for all inputs of length $\leq n$, for each n in turn, until one computation terminates.

Other Partially Decidable Problems

- Finding a natural deduction proof for some theorem. It is partially decidable because we can systematically generate all possible proofs.
- Determining whether a given string belongs to the language generated by a given context free grammar.
- Finding a deterministic PDA which accepts the same language as an arbitrary non-deterministic one.
- Hilbert's 10th Problem: (solve the Diophantine equations)
  "If polynomial $p(x_1, x_2, .., x_n)$ has integer coefficients, does the equation $p(x_1, x_2, .., x_n) = 0$ have a solution in the integers?"
- Equivalence of grammars.
- Equivalence of functions.

Limitations of static code analysis

- Analyse this function and tell me whether it halts or not:

$$simple\ () = \textbf{if}\ \ 2+3 > 15\ \ \textbf{then}\ \ loop\ ()\ \ \textbf{else}\ \ halt\ ()$$

- What about this one:

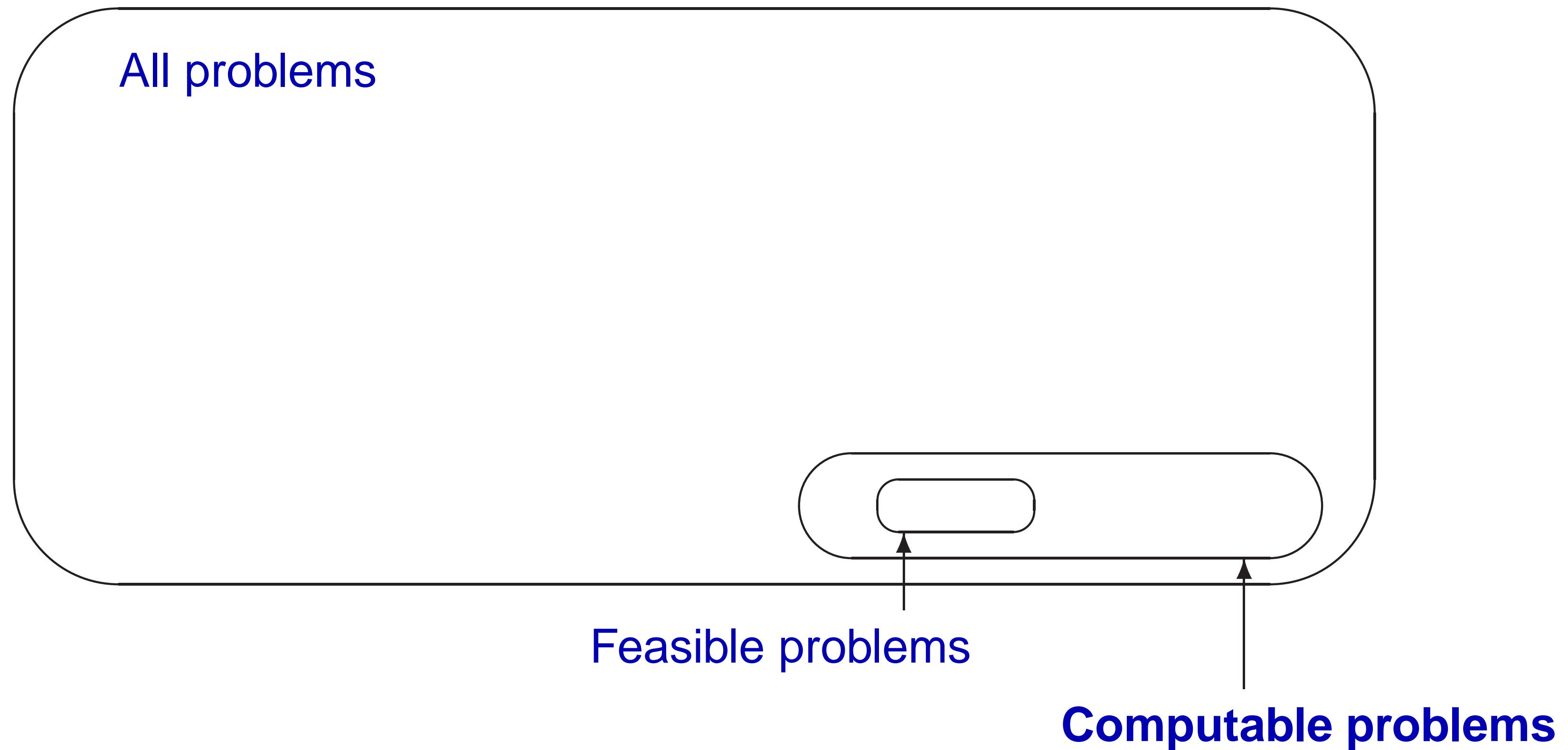
$$notSoSimple\ () = \textbf{if}\ \ fac\ 4 > 20\ \ \textbf{then}\ \ loop\ ()\ \ \textbf{else}\ \ halt\ ()$$

- Good luck with this one:

$$goodLuck\ () = \textbf{if}\ primes\ !!\ (fac\ 100)\ `rem`\ 10\ >\ 5$$
$$\textbf{then}\ \ loop()\ \ \textbf{else}\ \ halt()$$

- At a certain point, the only way to determine whether a function will halt or not is simply to evaluate the function, which may or may not halt.

The Universe of Problems



- Just because a problem is computable, doesn't mean we expect to ever have the technology to compute it.

Time Complexity

- Negating a boolean value takes constant time.

- Multiplication of integers takes an amount of time proportional to the square of the total number of digits.

  If we double number of digits, then the algorithm will take *four times* as long to complete.

  If we set n to be the total number of digits, we can say that multiplication has a complexity of order n^2.

- Smart matrix multiplication is order $n^{2.376}$
- Some theorem-proving algorithms are order 2^{2^n}
- Inserting an element into a binary tree is order $log\ n$

Ackermann's Function...

```
ack 0 n  = n + 1
ack n 0  = ack (n - 1) 1
ack n m  = ack (n - 1) (ack n (m - 1))
```

We don't have the language to describe the complexity measure of computing $ack\ n\ m$.

If we fix `n` to a low value, the results are:

$$
\begin{aligned}
ack\ 0\ y &= y + 1 \\
ack\ 1\ y &= y + 2 \\
ack\ 2\ y &= 2y + 3 \\
ack\ 3\ y &= 2^{y+3} - 3
\end{aligned}
$$

... can take a very long time to evaluate

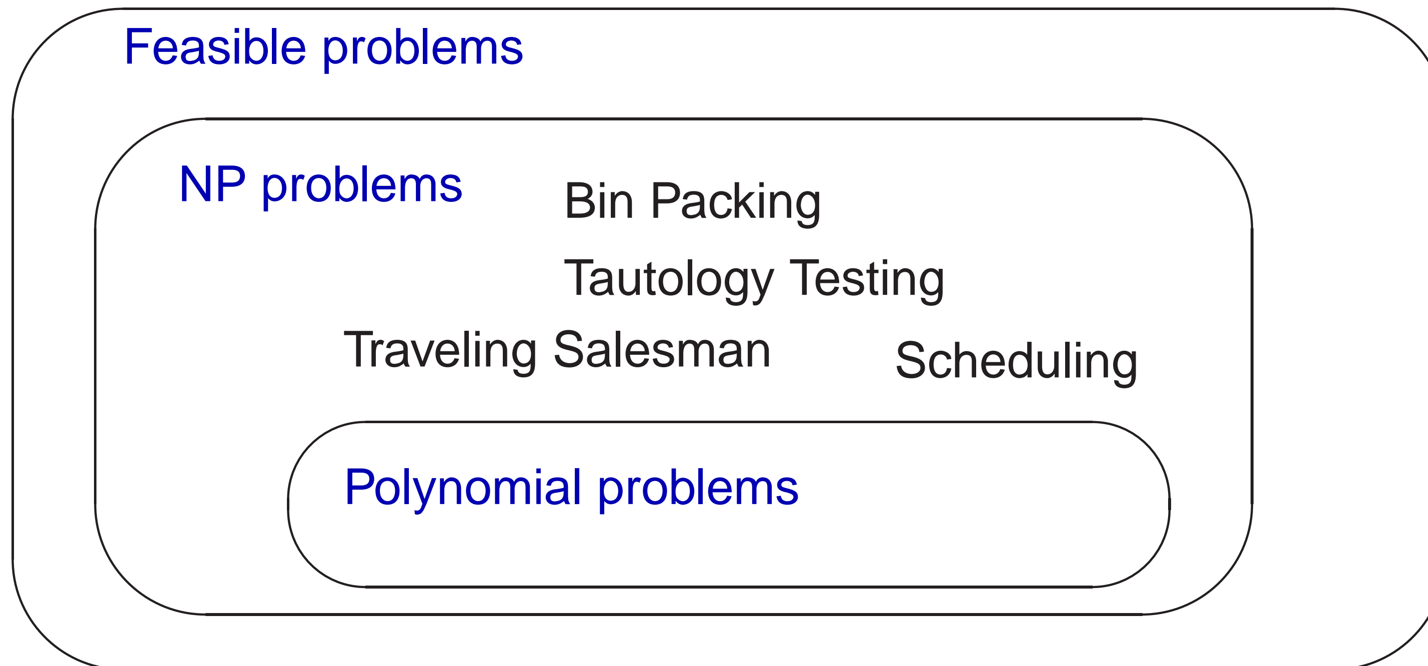
$$\begin{aligned} ack\ 4\ 1 &= 2^{2^{2^2}} - 3 \\ ack\ 4\ 2 &= 2^{2^{2^{2^2}}} - 3 \\ ack\ 4\ 3 &= 2^{2^{2^{2^{2^2}}}} - 3 \\ ack\ 4\ n &= \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3\ \text{twos}} - 3 \end{aligned}$$

Notice that the definition of `ack` does not use the multiply operator. The result must be computed via addition.

Q. How long will `ack (fac 100) (fac 100)` take to evaluate?

A. Longer than the age of the universe, many, many times over.

The Feasible Problems

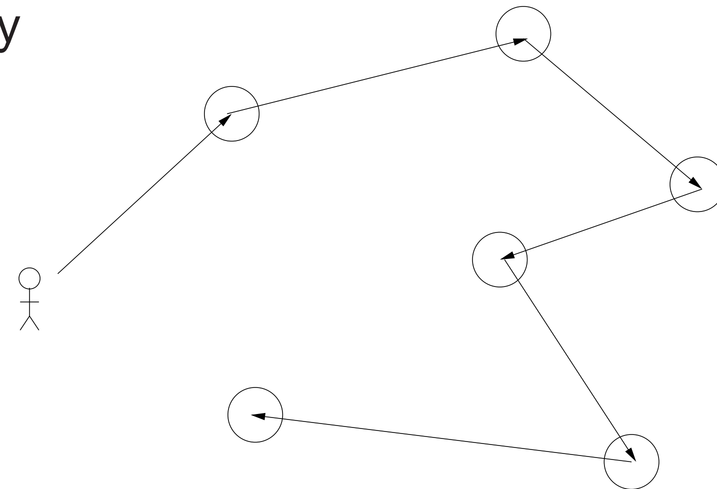


- Polynomial problems are the ones who's time complexity is bounded by some polynomial of the input size.
- Many useful, feasible algorithms have polynomial complexity.

The complexity class NP

- The complexity class of NP is the class of languages L that can be verified in polynomial time by a non-deterministic Turing machine.
- For a given state and symbol on the tape, a non-deterministic Turing machine can choose the “best” action among several options.

Traveling Salesman Non-deterministically *guess* the best route for the salesman to follow, then calculate whether it satisfies the length bound.



The NP Question

- Many NP problems are equivalent, and *very hard.*
- These are the NP-complete problems.
- If someone comes up with a polynomial algorithm for one of them then that algorithm will make all these problems polynomial.
- This is because a solution to one can be use to solve the others. Translation between these problems also takes polynomial time.
- If someone shows one is exponential, then that proof shows they all are.
- Whether $P = NP$ is a celebrated open problem.

Representation, Propagation, Interaction

Surprisingly simple systems are capable of universal computation. We have seen two so far:

- The Lambda Calculus
- Turing Machines

In theory, we could write all our programs in the raw lambda calculus, but no one does.

It would be too hard, and the programs would run too slowly.

There is a gap between what is *possible* and what is *practical*.

Photo: Buchanan-Hermit (attribution)

"Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy."

– Alan Perlis

The SKI Combinator system is Turing Complete

The SKI system contains three combinators and the following rules:

$$\begin{array}{llll} \mathrm{I}\,x & \longrightarrow & x & \text{(identity)} \\ \mathrm{K}\,x\,y & \longrightarrow & x & \text{(discard second argument)} \\ \mathrm{S}\,x\,y\,z & \longrightarrow & x\,z\,(y\,z) & \text{(substitute)} \end{array}$$

Any λ-calculus expression can be expressed using S, K, I.

Eg, function composition, given by $\lambda f\,g\,x.\ f\,(g\,x)$, is equal to $S\,(K\,S)\,K$

You can check that $S\,(K\,S)\,K\,f\,g\,x = f\,(g\,x)$ using the rules above

We can use Church style encoding to build integers, etc, from S, K, I.

Here is the diverging expression $(\lambda x.\ x\,x)\ (\lambda x.\ x\,x)$ expressed using S, K, I

$$\mathrm{SII(SII)} \longrightarrow \mathrm{I(SII)(I(SII))} \longrightarrow \mathrm{I(SII)(SII)} \longrightarrow \mathrm{SII(SII)}$$

The SK combinator system is Turing Complete

You don't actually need all three combinators.

$$\mathrm{I} \equiv \mathrm{S\,K\,K}$$

For example:

$$\begin{aligned} & \mathrm{S\,K\,K}\,x \\ \longrightarrow\ & \mathrm{K}\,x\,(\mathrm{K}\,x) \\ \longrightarrow\ & x \end{aligned}$$

The SK combinator system can encode the SKI system,
so it is still *Turing Complete*.

The Iota combinator system is Turing Complete

.. and you don't actually need all two combinators.

Define:

$$\iota\, x = x\, \mathrm{S}\, \mathrm{K}$$

Using ι we can reconstruct S and K as:

$$\begin{aligned} \mathrm{S} &\equiv \iota(\iota(\iota(\iota\iota))) \\ \mathrm{K} &\equiv \iota(\iota(\iota\iota)) \end{aligned}$$

Useful? ... doubtful.

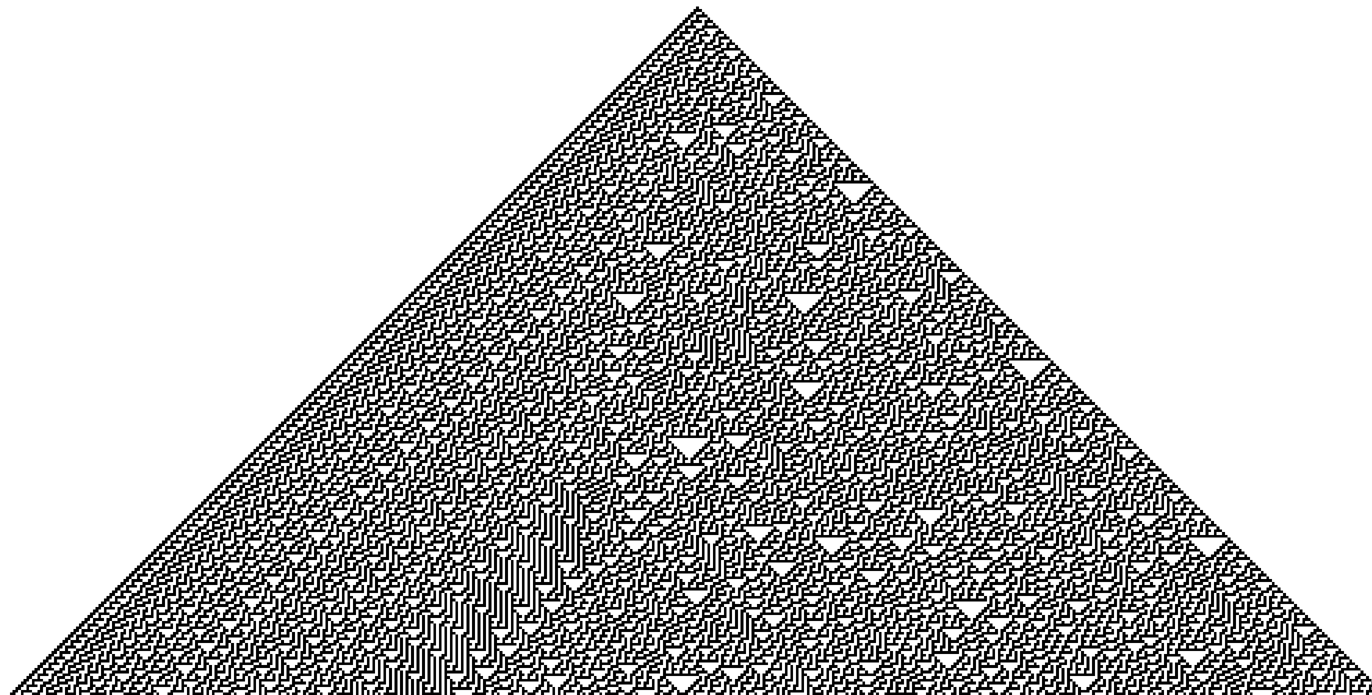
Interesting? ... perhaps.

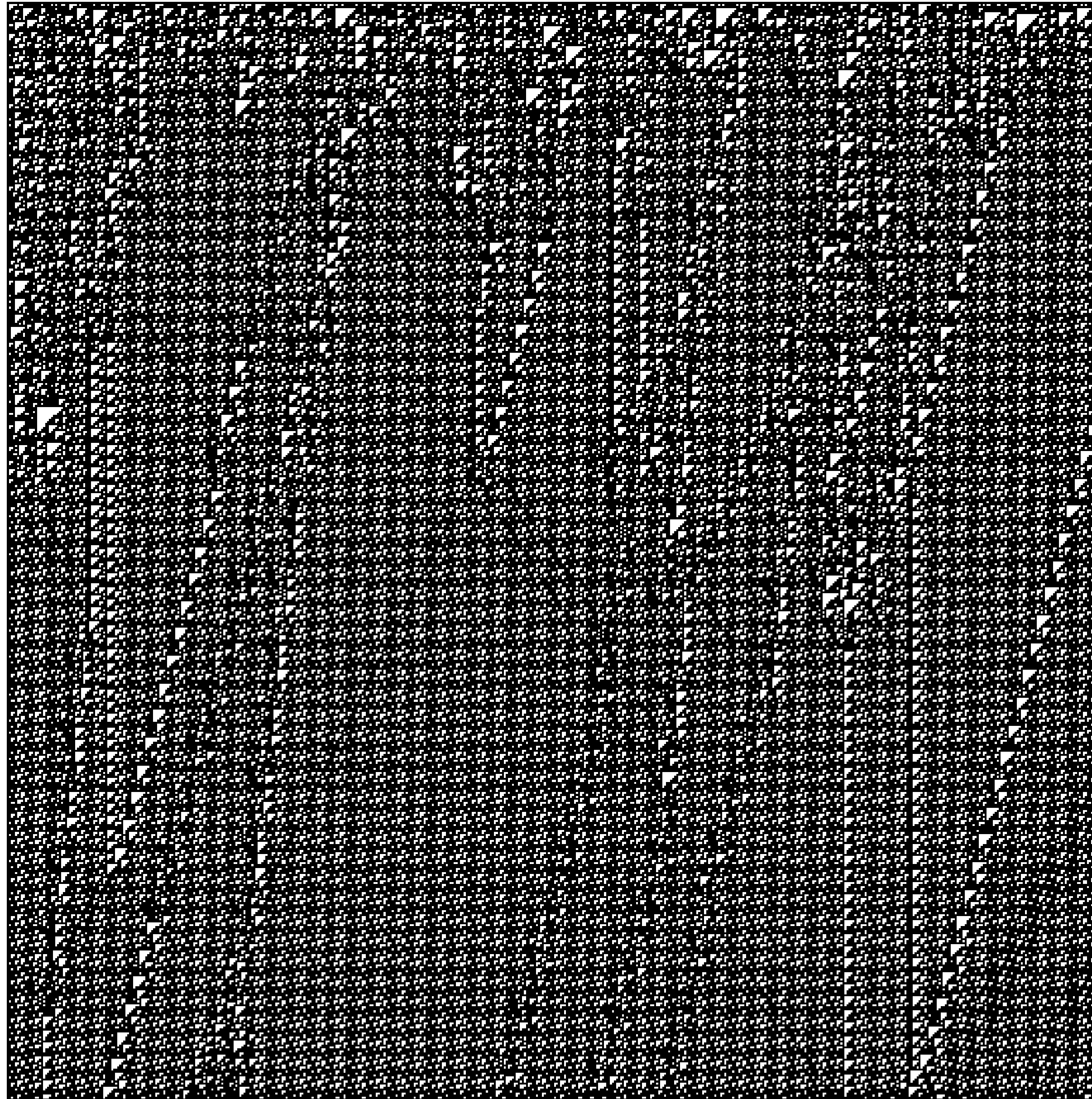
John Conway’s Game of Life is Turing Complete

A Turing machine constructed by Paul Rendell, April 2000.

The 1D Cellular Automata “Rule 30” is Turing Complete

prev line	111	110	101	100	011	010	001	000
next line	0	0	0	1	1	1	1	0





Conus textile

Photo: Richard Ling <richard@research.canon.com.au>

References

- Artificial Intelligence and Computability
  - Ray Kurzweil, *The age of intelligent machines*, 1990.
- Computability and Turing Machines
  - James Hein, *Discrete Structures, Logic and Computability*, 1994.
  - Goldschlager & Lister, *Computer Science: a Modern Introduction*, 1982.
  - Lewis & Papadimitriou, *Elements of the theory of computation*, 1981.
- Complexity
  - Goldschlager & Lister, *Computer Science: a Modern Introduction*, 1982.