Reducing P vs NP to Seven Statements

Qais Alassa
q@qasawa.net

November 2026

Abstract

Cook–Levin reductions, self-reducibility, and Levin's universal search U reduce $\mathsf{P} \neq \mathsf{NP}$ to a single Π_2^0 sentence about one explicit machine: for every $c \geq 1$, some satisfiable CNF formula φ forces U past $|\varphi|^c + c$ steps. We set out the reduction in six short propositions and state the open sentence as the seventh. Specializing the Gutfreund–Shaltiel–Ta-Shma refuter to the clocked searcher A_c produces, for each c and each length n at which A_c errs, one explicit formula on which A_c fails. Appendix A gives a self-contained Haskell program that executes the finite constructions on small instances.

Contents

1 Definitions **1**

2 The seven statements **2**

3 Structure of the reduction **4**

4 Companion program **4**

A Verification program (`verify.hs`) **6**

1 Definitions

Fix a universal multi-tape deterministic Turing machine $\mathcal{M}$, and let $M_1, M_2, \ldots$ list its programs in length-lexicographic order. Every multi-tape deterministic Turing machine with time bound $p(n) \geq 1$ is simulated on $\mathcal{M}$ by some program M_i within $O\big(p(n)\log(p(n)+2)\big)$ steps [9].

A formula φ is a CNF formula over variables $x_1, \ldots, x_v$, encoded as a nonempty binary string ($|\varphi| \geq 1$) with $v \leq |\varphi|$, under a fixed-width literal encoding in which substituting constants for variables leaves $|\varphi|$ unchanged and checking a v-bit assignment against φ takes $O(|\varphi|^2)$ steps. Let SAT denote the set of satisfiable formulas.

Universal search. On input φ, the machine U runs in phases $k = 1, 2, \ldots$. In phase k, for each $i = 1, \ldots, k$, it simulates M_i on φ from scratch for at most 2^{k-i} steps of $\mathcal{M}$. Whenever a simulated run halts, U checks in $O(|\varphi|^2)$ steps whether its output encodes a v-bit satisfying assignment of φ; if so, U prints that assignment and halts.

- $T(\varphi)$ is the number of Turing steps U takes on φ. Since brute-force search occurs as some M_i, $T(\varphi)$ is finite if and only if $\varphi \in \mathrm{SAT}$.

- For each integer $n \geq 1$, $t(n) = \max(\{0\} \cup \{T(\varphi) : \varphi \in \mathrm{SAT},\ |\varphi| \leq n\})$. Because at most 2^{n+1} strings have length at most n, membership in SAT is decidable by truth-table search, and U halts on every $\varphi \in \mathrm{SAT}$, the function t is total and computable.
- For each integer $c \geq 1$, the clocked algorithm A_c runs U on φ for $|\varphi|^c + c$ steps and accepts if and only if $T(\varphi) \leq |\varphi|^c + c$. Each A_c runs in time $O(|\varphi|^c)$ and accepts only when U has verified a satisfying assignment, so every error of A_c is a formula $\varphi \in \mathrm{SAT}$ that A_c rejects.

2 The seven statements

Statement 1 (Cook [6], Levin [10])**.** $\mathrm{SAT} \in \mathsf{NP}$*, and every language in* NP *is polynomial-time many-one reducible to* SAT*. Therefore* $\mathsf{P} = \mathsf{NP}$ *if and only if* $\mathrm{SAT} \in \mathsf{P}$*.*

The tableau reduction preserves witnesses: from any satisfying assignment of the output formula, a witness for the source NP verifier can be decoded in polynomial time. Full proofs of Statement 1 have been mechanized in Coq [7] and in Isabelle/HOL [3].

Statement 2 (Self-reducibility)**.** *If* $\mathrm{SAT} \in \mathsf{P}$*, then some polynomial-time algorithm outputs a satisfying assignment for every* $\varphi \in \mathrm{SAT}$*.*

Proof. Given φ on $v \leq |\varphi|$ variables, query the SAT decider on φ; if it rejects, halt without output. Otherwise set $x_1, \ldots, x_v$ in order, fixing $x_j = 1$ if the restricted formula remains satisfiable and $x_j = 0$ otherwise. Each restriction has length $|\varphi|$, so $v + 1$ calls to the decider suffice and take time polynomial in $|\varphi|$. □

Statement 3 (Universal search [10])**.** *If* M_i *on* $\varphi \in \mathrm{SAT}$ *halts within* $s \geq 1$ *steps of* $\mathcal{M}$ *with a satisfying assignment, then* U *halts on* φ *by phase* $i + \lceil \log_2 s \rceil$*, executing fewer than* $2^{i+2}s$ *steps of simulated programs and* $O_i(s + |\varphi|^2 \log^2(s+2))$ *steps of* U*. Consequently,*

$$\mathsf{P} = \mathsf{NP} \quad \Longleftrightarrow \quad \exists c \geq 1\ \forall n \geq 1 :\ t(n) \leq n^c + c.$$

Proof. Set $k = i + \lceil \log_2 s \rceil$. Unless U has already halted with a satisfying assignment, in phase k it allots M_i $2^{k-i} \geq s$ steps, so U halts by phase k. Across phases $r = 1, \ldots, k$, the simulated steps of $\mathcal{M}$ sum to $\sum_{r=1}^{k} \sum_{j=1}^{r} 2^{r-j} < \sum_{r=1}^{k} 2^r < 2^{k+1} < 2^{i+2}s$. Up to phase k there are $k(k+1)/2$ runs; run (r, j) spends $O(k + |\varphi|)$ steps preparing M_j and φ, $O(2^{r-j})$ steps simulating $\mathcal{M}$ and resetting work tapes, and $O(|\varphi|^2)$ steps checking any halted output. Summing over $1 \leq j \leq r \leq k$ gives $T(\varphi) = O(2^i s + k^2|\varphi|^2 + k^3) = O_i(s + |\varphi|^2 \log^2(s+2))$.

If $t(n) \leq n^c + c$ for all $n \geq 1$, then A_c decides SAT in polynomial time, and Statement 1 yields $\mathsf{P} = \mathsf{NP}$. Conversely, if $\mathsf{P} = \mathsf{NP}$, Statements 1 and 2 provide a polynomial-time finder, hence a program M_i on $\mathcal{M}$ that outputs a satisfying assignment within $s(\varphi) \leq K|\varphi|^d$ steps for integers $K, d \geq 1$. The step bound for U gives integers $K', d' \geq 1$ with $t(n) \leq K' n^{d'} + K'$ for all $n \geq 1$. Setting $c = d' + 2K'$ gives $K' n^{d'} + K' \leq n^c + c$ for every $n \geq 1$. □

Thus $\mathsf{P} = \mathsf{NP}$ holds if and only if the fixed machine U runs in polynomial time on SAT. Even a non-constructive proof of $\mathsf{P} = \mathsf{NP}$ leaves only the clock exponent c unknown, with a multiplicative constant proportional to 2^i for the solver's index i.

Statement 4 (Constructive refutation; after [8])**.** *For integers* $c, n \geq 1$*, let* $\psi_{c,n}$ *be the Cook–Levin formula asserting of* (x, y) *that* x *encodes a formula of length* n*,* y *satisfies* x*, and* A_c *rejects* x*. Define* $R(c, n)$ *by running* A_c *on* $\psi_{c,n}$*: if* A_c *accepts, decode* x *from the satisfying assignment found by* U *and output* x*; otherwise output* $\psi_{c,n}$*.*

For each fixed c, $R(c,n)$ is computable in time polynomial in n. If A_c errs on some formula of length n, then A_c errs on $R(c,n)$, which has length n when A_c accepts $\psi_{c,n}$ and length $|\psi_{c,n}|$ when A_c rejects $\psi_{c,n}$. Consequently,

$$\mathsf{P} \neq \mathsf{NP} \iff \forall c \geq 1\ \exists n \geq 1 :\ R(c,n) \in \mathrm{SAT} \text{ and } A_c \text{ rejects } R(c,n).$$

Proof. Checking (x,y) runs A_c on x of length n, which for fixed c takes time polynomial in n; hence $|\psi_{c,n}|$ and the run of A_c on $\psi_{c,n}$ are polynomial in n. Suppose A_c errs on some formula of length n, so $\psi_{c,n} \in \mathrm{SAT}$. If A_c accepts $\psi_{c,n}$, the verified assignment produced by U decodes to (x,y) witnessing that $|x| = n$, y satisfies x, and A_c rejects x; thus A_c errs on $R(c,n) = x$. If A_c rejects $\psi_{c,n}$, then A_c rejects the satisfiable formula $R(c,n) = \psi_{c,n}$.

For the equivalence, if $\mathsf{P} \neq \mathsf{NP}$ and $c \geq 1$, Statement 3 yields $m \geq 1$ and $\varphi \in \mathrm{SAT}$ with $|\varphi| \leq m$ and $T(\varphi) > m^c + c \geq |\varphi|^c + c$. Then A_c errs at length $n = |\varphi|$, so A_c errs on $R(c,n)$. Conversely, if $R(c,n) \in \mathrm{SAT}$ is rejected by A_c, then $t(|R(c,n)|) > |R(c,n)|^c + c$, and Statement 3 gives $\mathsf{P} \neq \mathsf{NP}$. □

For a general $O(n^c)$-time decider with two-sided error, Gutfreund, Shaltiel, and Ta-Shma [8, Lemma 3.1] give a deterministic $n^{O(c^2)}$-time procedure printing at most three formulas of length n or $\mathrm{poly}(n^c)$ that contain an error whenever one exists at length n, from which one of six fixed procedures isolates a single length-n error [8, Remark 3.2] (the strict refuter format studied in [5]). Because A_c has one-sided error and returns a checked witness upon acceptance, $R(c,n)$ produces a single failing formula directly.

Statement 5. *For every $N \geq 1$, some $c \geq 1$ satisfies $t(n) \leq n^c + c$ for all $1 \leq n \leq N$. Changing $t(n)$ at finitely many n leaves the condition $\exists c \geq 1\ \forall n \geq 1 :\ t(n) \leq n^c + c$ unchanged.*

Proof. Take $c = \max(1, \max_{1 \leq n \leq N} t(n))$; for all $1 \leq n \leq N$, $n^c + c \geq 1 + c > c \geq t(n)$. If $t(n) \leq n^c + c$ for all $n \geq 1$ and t' agrees with t for $n > N$, then $c' = \max(c, \max_{1 \leq n \leq N} t'(n))$ satisfies $t'(n) \leq n^{c'} + c'$ for all $n \geq 1$; the converse holds by symmetry. □

Every finite prefix of t fits a polynomial bound, and any polynomial prefix can precede exponential growth. Runtime measurements on bounded inputs are therefore compatible with both $\mathsf{P} = \mathsf{NP}$ and $\mathsf{P} \neq \mathsf{NP}$.

Statement 6 (Relativization [2])**.** *There exist oracles A and B such that $\mathsf{P}^A = \mathsf{NP}^A$ and $\mathsf{P}^B \neq \mathsf{NP}^B$.*

Statements 1–3 relativize when the enumerated programs and the assignment check are given access to an oracle, so the relativized form of Statement 7 holds relative to B and fails relative to A. Proof methods whose steps remain valid relative to every oracle, such as black-box simulation and diagonalization alone, therefore cannot decide Statement 7. Two further barriers restrict lower-bound routes: Razborov and Rudich [12] exclude $\mathsf{P/poly}$-natural circuit lower bounds for $\mathsf{NP} \not\subseteq \mathsf{P/poly}$ assuming subexponentially hard pseudorandom function generators, and Aaronson and Wigderson [1] exclude algebrizing proofs.

Statement 7 (Open)**.** *For every integer $c \geq 1$ there exists $\varphi \in \mathrm{SAT}$ such that $T(\varphi) > |\varphi|^c + c$.*

By Statement 3, Statement 7 is equivalent to $\mathsf{P} \neq \mathsf{NP}$, and its negation $\exists c \geq 1\ \forall \varphi \in \mathrm{SAT} :\ T(\varphi) \leq |\varphi|^c + c$ is equivalent to $\mathsf{P} = \mathsf{NP}$. For each individual c, Statement 7 admits a finite witness: a formula φ (which by Statement 4 can be chosen as $R(c,n)$ for some n), a satisfying assignment of φ, and the $|\varphi|^c + c$-step rejecting trace of U on φ. Even if $\mathsf{P} = \mathsf{NP}$, such witnesses exist for every c below the true exponent of U, so witnesses at single values of c do not distinguish $\mathsf{P} = \mathsf{NP}$ from $\mathsf{P} \neq \mathsf{NP}$.

3 Structure of the reduction

Written out from definitions, $\mathsf{P} \neq \mathsf{NP}$ begins with quantifiers over all NP verifiers and all polynomial-time machines. Statement 1 fixes the NP language to SAT; Statement 2 replaces decision by witness search; Statement 3 replaces the quantifier over candidate algorithms by the single program U; and Statement 4 replaces the existential witness at a failing length by the explicit formula $R(c, n)$. Together, Statements 1–3 contract $\mathsf{P} \neq \mathsf{NP}$ to the Π_2^0 arithmetic sentence $\forall c \geq 1\ \exists \varphi \in \mathrm{SAT} :\ T(\varphi) > |\varphi|^c + c$, whose matrix is decidable by running U for $|\varphi|^c + c$ steps and checking a satisfying assignment. The universal quantifier over algorithms is folded into the phase schedule of U: $T(\varphi) > |\varphi|^c + c$ asserts that every program M_i scheduled within $|\varphi|^c + c$ steps of U fails to return a satisfying assignment of φ.

Unconditional lower bounds remain far below Statement 7. On multi-tape machines with space $n^{o(1)}$, SAT requires time $\Omega(n^a)$ for every $a < 2\cos(\pi/7) \approx 1.801$ [13], an exponent that is tight for current alternation-trading arguments [4]. Over the full binary basis B_2, the strongest explicit circuit size lower bound for an NP function is $(3.1 - o(1))n$ [11], whereas proving Statement 7 via $\mathsf{NP} \not\subseteq \mathsf{P/poly}$ requires a superpolynomial bound.

4 Companion program

Appendix A lists `verify.hs` in Haskell 2010 (`base` library only; compile with `ghc -O2 verify.hs`). On small instances it compares a Cook–Levin tableau encoder against direct nondeterministic simulation (Statement 1), recovers assignments from decision queries in at most $v + 1$ calls (Statement 2), checks $T_U < 2^{i+2} T_i$ and clock monotonicity on a step-monad model of U (Statement 3), runs $R(c, n)$ exhaustively across 2-CNF formulas on two and three variables (Statement 4), verifies $2^n \leq n^{101} + 101$ for $1 \leq n \leq 1000$ together with its failure at $n = 1008$ in exact integer arithmetic (Statement 5), and tests the query-tree adversary underlying $\mathsf{P}^B \neq \mathsf{NP}^B$ (Statement 6). Each routine also checks that dropping its key hypothesis (frame axioms, oracle soundness, output verification, rejection clauses, or under-querying) produces counterexamples as expected.

References

[1] S. Aaronson and A. Wigderson. Algebrization: a new barrier in complexity theory. *ACM Transactions on Computation Theory*, 1(1), Article 2, 2009.

[2] T. Baker, J. Gill, and R. Solovay. Relativizations of the $\mathsf{P} =?\ \mathsf{NP}$ question. *SIAM Journal on Computing*, 4(4):431–442, 1975.

[3] F. J. Balbach. The Cook–Levin theorem. *Archive of Formal Proofs*, January 2023. `https://isa-afp.org/entries/Cook_Levin.html`

[4] S. Buss and R. Williams. Limits on alternation-trading proofs for time-space lower bounds. *Computational Complexity*, 24(3), 2015.

[5] L. Chen, C. Jin, R. Santhanam, and R. Williams. Constructive separations and their consequences. *TheoretiCS*, 3, Article 3, 2024.

[6] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC)*, pp. 151–158, 1971.

[7] L. Gäher and F. Kunze. Mechanising complexity theory: the Cook–Levin theorem in Coq. In *Interactive Theorem Proving (ITP)*, pp. 20:1–20:18, 2021.

[8] D. Gutfreund, R. Shaltiel, and A. Ta-Shma. If NP languages are hard on the worst-case, then it is easy to find their hard instances. *Computational Complexity*, 16(4):412–441, 2007.

[9] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, 1966.

[10] L. A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973. English translation in *Problems of Information Transmission*, 9(3):265–266, 1973.

[11] J. Li and T. Yang. $3.1n - o(n)$ circuit lower bounds for explicit functions. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pp. 1180–1193, 2022.

[12] A. A. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1), 1997.

[13] R. Williams. Alternation-trading proofs, linear programming, and lower bounds. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 2010.

A Verification program (`verify.hs`)

```
-- Companion checks for "Reducing P vs NP to Seven Statements" (Qais Alassa).
-- Pure Haskell 2010 (base only). Build and run: ghc -O2 verify.hs && ./verify
module Main (main) where

import Control.Applicative ((<|>))
import Control.Monad (ap, replicateM, unless)
import Data.Bits (bit, setBit, shiftL, shiftR, testBit, xor, (.&.), (.|.))
import Data.List (minimumBy, nub, tails)
import Data.Maybe (fromMaybe, isJust, isNothing)
import Data.Ord (comparing)
import Data.Word (Word64)
import System.Exit (die)

-- SplitMix64 generator in a minimal state monad.

newtype Rand a = Rand { runRand :: Word64 -> (a, Word64) }

instance Functor Rand where
  fmap f (Rand m) = Rand $ \s -> let (x, s') = m s in (f x, s')

instance Applicative Rand where
  pure x = Rand $ \s -> (x, s)
  (<*>)  = ap

instance Monad Rand where
  Rand m >>= k = Rand $ \s -> let (x, s') = m s in runRand (k x) s'

randWord :: Rand Word64
randWord = Rand $ \s ->
  let s' = s + 0x9e3779b97f4a7c15
      z1 = (s' `xor` shiftR s' 30) * 0xbf58476d1ce4e5b9
      z2 = (z1 `xor` shiftR z1 27) * 0x94d049bb133111eb
  in (z2 `xor` shiftR z2 31, s')

randInt :: Int -> Rand Int
randInt k = (\w -> fromIntegral (w `mod` fromIntegral k)) <$> randWord

randRange :: Int -> Int -> Rand Int
randRange lo hi = (lo +) <$> randInt (hi - lo + 1)

pick :: Int -> [a] -> Rand [a]
pick 0 _  = pure []
pick _ [] = pure []
pick k xs = do
  i <- randInt (length xs)
  case splitAt i xs of
    (as, b : bs) -> (b :) <$> pick (k - 1) (as ++ bs)
    _            -> pure []

check :: String -> Bool -> String -> IO ()
check tag ok msg = do
  unless ok $ die ("failed (" ++ tag ++ "): " ++ msg)
  putStrLn (tag ++ ": " ++ msg)

-- CNF representation and DPLL solver with Integer bitmask propagation.

type Lit    = Int
type Clause = [Lit]
type CNF    = [Clause]
type Env    = [(Int, Bool)]

evalLit :: Env -> Lit -> Bool
evalLit env l = maybe False (== (l > 0)) (lookup (abs l) env)

satisfies :: CNF -> Env -> Bool
satisfies cnf env = all (any (evalLit env)) cnf

propagate :: CNF -> Env -> Maybe (CNF, Env)
propagate cnf env = case [l | [l] <- cnf] of
  [] -> Just (cnf, env)
```

```
  us ->
    let posM  = foldl setBit (0 :: Integer) [u | u <- us, u > 0]
        negM  = foldl setBit (0 :: Integer) [-u | u <- us, u < 0]
        inU l = testBit (if l > 0 then posM else negM) (abs l)
        env'  = [(abs u, u > 0) | u <- nub us] ++ env
        step [] acc = Just (reverse acc)
        step (c : cs) acc
          | any inU c = step cs acc
          | otherwise = case filter (not . inU . negate) c of
              [] -> Nothing
              c' -> step cs (c' : acc)
    in if (posM .&. negM) /= 0 then Nothing
       else step cnf [] >>= \cnf' -> propagate cnf' env'

solve :: CNF -> Maybe Env
solve cnf
  | any null cnf = Nothing
  | otherwise    = go cnf []
  where
    go cs env = do
      (cs', env') <- propagate cs env
      case cs' of
        [] -> Just env'
        _  -> case minimumBy (comparing length) cs' of
          (l : _) -> go ([l] : cs') env' <|> go ([-l] : cs') env'
          []      -> Nothing

randomCnf :: Int -> Int -> Int -> Rand CNF
randomCnf nVars nClauses width = replicateM nClauses $ do
  vs <- pick (min width nVars) [1 .. nVars]
  mapM (\v -> (\b -> if b == 0 then v else -v) <$> randInt 2) vs

-- 1. Cook-Levin tableau reduction vs direct nondeterministic simulation.

type Trans = [((Int, Int), [(Int, Int, Int)])]

qStart, qAccept, symBlank :: Int
qStart   = 0
qAccept  = 1
symBlank = 2

setTape :: Int -> Int -> [(Int, Int)] -> [(Int, Int)]
setTape p a [] = [(p, a) | a /= symBlank]
setTape p a ((q, b) : rest)
  | p < q     = [(p, a) | a /= symBlank] ++ ((q, b) : rest)
  | p == q    = [(p, a) | a /= symBlank] ++ rest
  | otherwise = (q, b) : setTape p a rest

ntmAccepts :: Trans -> [Int] -> Int -> Bool
ntmAccepts delta x tMax = step 0 [(qStart, 0, initTape)]
  where
    initTape = [(i, b) | (i, b) <- zip [0 ..] x, b /= symBlank]
    step t front
      | any (\(q, _, _) -> q == qAccept) front = True
      | t >= tMax = False
      | otherwise = step (t + 1) $ nub
          [ (q2, p + m, setTape p a2 tp)
          | (q, p, tp)  <- front
          , let cur = fromMaybe symBlank (lookup p tp)
          , (q2, a2, m) <- fromMaybe [] (lookup (q, cur) delta) ]

cookLevin :: Bool -> Trans -> Int -> [Int] -> Int -> CNF
cookLevin useFrame delta nStates x tMax =
  wellFormed ++ initCfg ++ transitions ++ [[vS tMax qAccept]]
  where
    (pLo, pHi) = (-tMax, max (length x) 1 + tMax)
    cells      = [pLo .. pHi]
    nC         = length cells
    b1         = (tMax + 1) * nStates
    b2         = b1 + (tMax + 1) * nC
    b3         = b2 + (tMax + 1) * nC * 3
    maxDeg     = maximum (1 : map (length . snd) delta)
```

```
    vS t q     = 1 + t * nStates + q
    vH t pc    = b1 + 1 + t * nC + pc
    vC t pc a  = b2 + 1 + (t * nC + pc) * 3 + a
    vE t pc a q i = b3 + 1 + (((t * nC + pc) * 3 + a) * nStates + q) * maxDeg + i
    one vs = vs : [[-a, -b] | (a : bs) <- tails vs, b <- bs]
    wellFormed = concat
      [ one [vS t q | q <- [0 .. nStates - 1]]
        ++ one [vH t pc | pc <- [0 .. nC - 1]]
        ++ concat [one [vC t pc a | a <- [0 .. 2]] | pc <- [0 .. nC - 1]]
      | t <- [0 .. tMax] ]
    initCfg = [vS 0 qStart] : [vH 0 (0 - pLo)] :
      [ [vC 0 pc (if 0 <= p && p < length x then x !! p else symBlank)]
      | (pc, p) <- zip [0 ..] cells ]
    transitions = concat
      [ [[vH t pc, -vC t pc a, vC (t + 1) pc a] | useFrame] ++
        concat
          [ let raw = if q == qAccept then [(qAccept, a, 0)]
                      else fromMaybe [] (lookup (q, a) delta)
                mvs = [ (q2, a2, m) | (q2, a2, m) <- raw
                      , p + m >= pLo, p + m <= pHi ]
                es  = [vE t pc a q i | i <- [0 .. length mvs - 1]]
            in (-vS t q : -vH t pc : -vC t pc a : es) :
               concat [ [ [-e, vS (t + 1) q2]
                        , [-e, vH (t + 1) (pc + m)]
                        , [-e, vC (t + 1) pc a2] ]
                      | (e, (q2, a2, m)) <- zip es mvs ]
          | q <- [0 .. nStates - 1] ]
      | t <- [0 .. tMax - 1], (pc, p) <- zip [0 ..] cells, a <- [0 .. 2] ]

randomNtm :: Int -> Rand Trans
randomNtm nStates = mapM mkEntry keys
  where
    keys = [(q, a) | q <- [0 .. nStates - 1], q /= qAccept, a <- [0 .. 2]]
    mkEntry qa = do
      d <- ([0, 1, 2, 2] !!) <$> randInt 4
      mvs <- replicateM d $
        (,,) <$> randInt nStates <*> randInt 2 <*> ((+ (-1)) <$> randInt 3)
      pure (qa, mvs)

checkS1 :: Rand (IO ())
checkS1 = do
  let (nS, tMax) = (4, 4)
      inputs = concatMap (\n -> sequence (replicate n [0, 1])) [0 .. 3]
  deltas <- replicateM 60 (randomNtm nS)
  let runs = [ (truth, isJust (solve (cookLevin True d nS x tMax)) == truth)
             | d <- deltas, x <- inputs, let truth = ntmAccepts d x tMax ]
      acc  = length (filter fst runs)
      away = [((0, 0), [(2, 0, 1)]), ((2, 2), [(3, 2, -1)]), ((3, 1), [(1, 1, 0)])]
      frameNeeded = not (ntmAccepts away [0] tMax)
                 && isNothing (solve (cookLevin True away nS [0] tMax))
                 && isJust (solve (cookLevin False away nS [0] tMax))
  pure $ check "S1" (all snd runs && acc > 0 && acc < length runs && frameNeeded)
    (show (length runs) ++ " NTM/CNF runs agree (" ++ show acc
     ++ " accepting); frame-axiom omission caught")

-- 2. Self-reducibility (witness recovery from decision queries).

searchSat :: (CNF -> Bool) -> CNF -> Int -> (Maybe Env, Int)
searchSat dec cnf nVars
  | not (dec cnf) = (Nothing, 1)
  | otherwise     = (Just [(abs l, l > 0) | [l] <- fixed], 1 + nVars)
  where
    fixed = foldl (\acc v -> acc ++ [[if dec (cnf ++ acc ++ [[v]]) then v else -v]])
                  [] [1 .. nVars]

checkS2 :: Rand (IO ())
checkS2 = do
  trials <- replicateM 400 $ do
    n   <- randRange 3 12
    m   <- randRange 2 (5 * n)
    cnf <- randomCnf n m 3
    let truth = isJust (solve cnf)
```

```
        (ans, calls) = searchSat (isJust . solve) cnf n
        sound = calls <= n + 1 && maybe (not truth) (\a -> truth && satisfies cnf a) ans
        bad   = truth && maybe True (not . satisfies cnf)
                              (fst (searchSat (const True) cnf n))
    pure (sound, truth, bad)
  let satCnt = length [() | (_, True, _) <- trials]
      caught = length [() | (_, _, True) <- trials]
  pure $ check "S2" (all (\(ok, _, _) -> ok) trials && satCnt > 0 && caught > 0)
    ("400 trials solved in <= v+1 queries (" ++ show satCnt
     ++ " SAT); dishonest oracle rejected " ++ show caught ++ " times")

-- 3. Levin universal search via the step monad Run.

data Run a = Halt a | Step (Run a)

instance Functor Run where
  fmap f (Halt x) = Halt (f x)
  fmap f (Step r) = Step (fmap f r)

instance Applicative Run where
  pure  = Halt
  (<*>) = ap

instance Monad Run where
  Halt x >>= k = k x
  Step r >>= k = Step (r >>= k)

tick :: Run ()
tick = Step (Halt ())

delay :: Int -> a -> Run a
delay k x | k <= 0    = Halt x
          | otherwise = Step (delay (k - 1) x)

type Prog = CNF -> Int -> Run (Maybe Env)

pConst :: Bool -> Prog
pConst val _ n = tick >> pure (Just [(v, val) | v <- [1 .. n]])

pLoop :: Prog
pLoop _ _ = let spin = Step spin in spin

pMajority :: Prog
pMajority cnf n = go (concat cnf) [(v, 0 :: Int) | v <- [1 .. n]]
  where
    go [] sc = pure (Just [(v, s >= 0) | (v, s) <- sc])
    go (l : ls) sc = tick >> go ls
      [(v, if v == abs l then s + (if l > 0 then 1 else -1) else s) | (v, s) <- sc]

pWalk :: Prog
pWalk cnf n =
  let s0 = fromIntegral (length cnf * 1000 + n + 17)
      (a0, s1) = runRand (mapM (\v -> (\b -> (v, b == 1)) <$> randInt 2) [1 .. n]) s0
  in walk (40 * n) a0 s1
  where
    walk 0 _ _ = pure Nothing
    walk k a s = do
      tick
      case [c | c <- cnf, not (any (evalLit a) c)] of
        [] -> pure (Just a)
        bs -> let ((ci, li), s') = runRand ((,) <$> randInt (length bs) <*> randWord) s
                  cl  = bs !! ci
                  tgt = abs (cl !! fromIntegral (li `mod` fromIntegral (length cl)))
              in walk (k - 1) [(v, if v == tgt then not b else b) | (v, b) <- a] s'

pBrute :: Prog
pBrute cnf n = go (sequence (replicate n [False, True]))
  where
    go [] = pure Nothing
    go (bs : rest) = tick >>
      let a = zip [1 .. n] bs
      in if satisfies cnf a then pure (Just a) else go rest
```

```
programs :: [Prog]
programs = [pConst False, pLoop, pConst True, pMajority, pWalk, pBrute]

checked :: Bool -> Prog -> CNF -> Int -> Run (Maybe Env)
checked ver prog cnf n = do
  out <- prog cnf n
  case out of
    Nothing -> pure Nothing
    Just a  -> delay (length cnf) $
                 if not ver || satisfies cnf a then Just a else Nothing

runAlone :: Int -> Run (Maybe Env) -> (Maybe Env, Int)
runAlone cap = go 0
  where
    go s _ | s >= cap = (Nothing, s)
    go s (Halt out)   = (out, s + 1)
    go s (Step r)     = go (s + 1) r

universal :: Bool -> Maybe Int -> CNF -> Int -> (Maybe Env, Int)
universal ver mClock cnf n = runPhase 1 1 0
  where
    runPhase k i s
      | i > min k (length programs) = runPhase (k + 1) 1 s
      | otherwise = slice (bit (k - i) :: Int)
                          (checked ver (programs !! (i - 1)) cnf n) s
      where
        slice 0 _ st = runPhase k (i + 1) st
        slice b r st
          | maybe False (st >=) mClock = (Nothing, st)
          | otherwise = case r of
              Halt (Just a) -> (Just a, st + 1)
              Halt Nothing  -> runPhase k (i + 1) (st + 1)
              Step r'       -> slice (b - 1) r' (st + 1)

checkS3 :: Rand (IO ())
checkS3 = do
  let gather comps okAll | comps >= 1500 = pure (comps, okAll)
      gather comps okAll = do
        n   <- randRange 3 8
        m   <- randRange 2 (3 * n)
        cnf <- randomCnf n m 3
        if isNothing (solve cnf) then gather comps okAll
        else do
          let (mOut, tU) = universal True Nothing cnf n
              valid = maybe False (satisfies cnf) mOut
              runs  = [ (i, tI) | (i, p) <- zip [1 ..] programs
                      , let (res, tI) = runAlone 2000 (checked True p cnf n)
                      , isJust res ]
              holds = all (\(i, tI) -> valid && tU < bit (i + 2) * tI) runs
          gather (comps + length runs) (okAll && holds)
  (comps, boundOk) <- gather 0 True
  clocked <- replicateM 120 $ do
    m   <- randRange 6 16
    cnf <- randomCnf 4 m 2
    let truth = isJust (solve cnf)
        big   = bit (length programs + 2) * (bit (4 :: Int) + 2 * length cnf + 2)
        acc   = [isJust (fst (universal True (Just b) cnf 4)) | b <- [8, 64, 512, big]]
        sound = (truth || not (or acc))
             && and [not x || y | (x, y) <- zip acc (drop 1 acc)]
             && (acc !! 3 == truth)
        unverifiedAccept = not truth && isJust (fst (universal False (Just 64) cnf 4))
    pure (sound, not truth, unverifiedAccept)
  let unsat = length [() | (_, True, _) <- clocked]
      falseAcc = length [() | (_, _, True) <- clocked]
  pure $ check "S3"
    (boundOk && all (\(s, _, _) -> s) clocked && unsat > 0 && falseAcc > 0)
    ("T_U < 2^(i+2) T_i across " ++ show comps ++ " runs; 120 clocked 2-CNFs ("
     ++ show unsat ++ " UNSAT); " ++ show falseAcc ++ " unverified false accepts")

-- 4. Constructive refutation R(c,n) on 2-CNF over v in {2,3}.
```

```
clauseUniv :: Int -> [Clause]
clauseUniv v =
  [[s * x] | x <- [1 .. v], s <- [1, -1]] ++
  [ [s1 * x, s2 * y]
  | x <- [1 .. v], y <- [x + 1 .. v], s1 <- [1, -1], s2 <- [1, -1] ]

lowWeight :: Int -> Int -> [Env]
lowWeight v k =
  [ zip [1 .. v] bs
  | bs <- sequence (replicate v [False, True]), length (filter id bs) <= k ]

candidate :: CNF -> Int -> Int -> Maybe Env
candidate cnf v k = case [a | a <- lowWeight v k, satisfies cnf a] of
  (a : _) -> Just a
  []      -> Nothing

buildPsi :: Bool -> Int -> Int -> [Clause] -> CNF
buildPsi withRej v k univ =
  [(- (v + 1 + j)) : c | (j, c) <- zip [0 ..] univ] ++
  [ [v + 1 + j | (j, c) <- zip [0 ..] univ, not (satisfies [c] a)]
  | withRej, a <- lowWeight v k ]

decodePsi :: Env -> Int -> [Clause] -> (CNF, Env)
decodePsi sol v univ =
  ( [c | (j, c) <- zip [0 ..] univ, fromMaybe False (lookup (v + 1 + j) sol)]
  , [(i, fromMaybe False (lookup i sol)) | i <- [1 .. v]] )

isError :: CNF -> Int -> Int -> Bool
isError x v k = isJust (solve x) && isNothing (candidate x v k)

refuter :: Int -> Int -> [Clause] -> (CNF, Int)
refuter v k univ =
  let psi   = buildPsi True v k univ
      nVars = v + length univ
  in case candidate psi nVars k of
       Just sol -> (fst (decodePsi sol v univ), v)
       Nothing  -> (psi, nVars)

checkS4 :: Rand (IO ())
checkS4 = do
  let pairs = [(v, k) | v <- [2, 3], k <- [0 .. v]]
  stats <- mapM testPair pairs
  let scanned = sum [sc | (sc, _, _, _) <- stats]
      refuted = sum [rf | (_, rf, _, _) <- stats]
      decoded = sum [dc | (_, _, dc, _) <- stats]
  pure $ check "S4" (all (\(_, _, _, ok) -> ok) stats && refuted > 0 && decoded > 0)
    (show scanned ++ " formulas scanned across 7 (v,k) cases: "
     ++ show refuted ++ " refuted, " ++ show decoded ++ " witnesses decoded")
  where
    testPair (v, k) = do
      let univ    = clauseUniv v
          allAsg  = lowWeight v v
          fullM   = (1 `shiftL` length allAsg) - (1 :: Int)
          cMasks  = [ foldl (.|.) 0
                            [bit i | (i, a) <- zip [0 ..] allAsg, satisfies [c] a]
                    | c <- univ ]
          satSets = foldl (\acc m -> acc ++ map (.&. m) acc) [fullM] cMasks
          lowM    = foldl (.|.) 0
                      [ bit i | (i, a) <- zip [0 ..] allAsg
                      , length [() | (_, True) <- a] <= k ]
          hasErr  = any (\s -> s /= 0 && (s .&. lowM) == 0) satSets
          psi     = buildPsi True v k univ
          refOk   = not hasErr || let (tgt, nv) = refuter v k univ
                                  in isJust (solve tgt) && isNothing (candidate tgt nv k)
          noRejBad = maybe False (\(x, _) -> not (isError x v k))
                       (decodePsi <$> solve (buildPsi False v k univ) <*> pure v <*> pure univ)
          lits = [s * p | p <- [1 .. v + length univ], s <- [1, -1]]
      samples <- replicateM (if hasErr then 200 else 0) $ do
        blk <- pick 3 lits
        pure $ case solve (psi ++ [[l] | l <- blk]) of
          Nothing  -> (0 :: Int, True)
          Just sol -> let (x, y) = decodePsi sol v univ
```

```
                    in (1, satisfies x y && isError x v k)
      pure ( length satSets, if hasErr then 1 else 0 :: Int, sum (map fst samples)
           , (isJust (solve psi) == hasErr) && refOk && noRejBad && all snd samples )

-- 5. Finite prefixes vs asymptotic growth (exact Integer powers).

checkS5 :: IO ()
checkS5 = do
  let lim   = 1000 :: Integer
      fit e = all (\n -> (2 :: Integer) ^ n <= n ^ e + e) [1 .. lim]
      c     = case [e | e <- [1 :: Integer .. 200], fit e] of (e : _) -> e; [] -> 0
      brk   = case [n | n <- [lim .. 2000], (2 :: Integer) ^ n > n ^ c + c] of
                (n : _) -> n
                []      -> 0
  check "S5" (c == 101 && fit c && brk == 1008)
    ("2^n <= n^101 + 101 holds for 1 <= n <= 1000 and first fails at n = " ++ show brk)

-- 6. Baker-Gill-Solovay decision-tree diagonalization step.

data Tree = Ans Bool | Ask Int Tree Tree

runTree :: Tree -> [Int] -> (Bool, [Int])
runTree (Ans b) _ = (b, [])
runTree (Ask q noT yesT) orc =
  let (ans, qs) = runTree (if q `elem` orc then yesT else noT) orc
  in (ans, q : qs)

fooled :: Int -> Tree -> Bool
fooled uSize tree =
  let (ans0, qs) = runTree tree []
  in if ans0 then True
     else case [s | s <- [0 .. uSize - 1], s `notElem` qs] of
            []      -> False
            (s : _) -> not (fst (runTree tree [s]))

allTrees :: Int -> Int -> [Tree]
allTrees _ 0  = [Ans True, Ans False]
allTrees sz d =
  let subs = allTrees sz (d - 1)
  in Ans True : Ans False :
     [Ask q noT yesT | q <- [0 .. sz - 1], noT <- subs, yesT <- subs]

randTree :: Int -> Int -> Rand Tree
randTree _ 0  = Ans . (== 1) <$> randInt 2
randTree sz d = do
  stop <- randInt 10
  if stop == 0 then Ans . (== 1) <$> randInt 2
  else Ask <$> randInt sz <*> randTree sz (d - 1) <*> randTree sz (d - 1)

checkS6 :: Rand (IO ())
checkS6 = do
  let exh  = allTrees 3 2
      full = foldr (\s rest -> Ask s rest (Ans True)) (Ans False) [0, 1, 2]
  smp <- replicateM 2000 (randTree 16 15)
  pure $ check "S6" (all (fooled 3) exh && all (fooled 16) smp && not (fooled 3 full))
    ("adversary fooled all " ++ show (length exh) ++ " depth-2 and "
     ++ show (length smp) ++ " depth-15 trees; full-query tree escaped")

main :: IO ()
main = do
  let suite = sequence [checkS1, checkS2, checkS3, checkS4, pure checkS5, checkS6]
  sequence_ (fst (runRand suite 20261127))
```