SAT is Not Solvable in Constant Time

Daniel Shea

May 8, 2026

Abstract

This entry establishes that the Boolean Satisfiability Problem (SAT) cannot be decided by a deterministic Turing machine in constant time ($O(1)$).

The core of the argument rests on an indistinguishability invariant: a Turing machine bounded by constant time C can only ever read the first $C + 1$ cells of its input tape. Consequently, any language decided in constant time must be a prefix language. We then show that SAT is not uniquely determined by any finite prefix, yielding the impossibility result.

Contents

theory *SAT-Not-Const-Time*
imports
Main
Cook-Levin.Reduction-TM
begin

1 Constant-Time Lower Bounds

This theory establishes that the Boolean Satisfiability Problem (SAT) cannot be decided by a deterministic Turing machine in constant time ($O(1)$).

The core of the argument rests on an indistinguishability invariant: a Turing machine bounded by constant time C can only ever read the first $C+1$ cells of its input tape. Consequently, any language decided in constant time must be a prefix language. We then show that SAT is not uniquely determined by any finite prefix, yielding the impossibility result.

Configuration records and the step function are inherited from the Cook-Levin formalization [1].

1.1 Head-Movement Bounds for Turing Machines

We first establish that a Turing machine's head can move at most one cell per execution step. Configuration records and the step function are inherited from *Reduction-TM*. We isolate the position of the input-tape head (tape 0).

definition *hd-span* :: *config* ⇒ *nat* **where**
hd-span cfg ≡ *cfg* <#> *0*

lemma *le-imp-le-Suc*: *n* ≤ *m* ⟹ *n* ≤ *Suc m*
⟨*proof*⟩

lemma *contents-nil* [*simp*]:
⌊[]⌋ = (λ*i*::*nat*. *if i* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *start-config-tape1* [*simp*]:
assumes *2* ≤ *k*
shows (*start-config k xs* <:> *1*) = ⌊[]⌋
⟨*proof*⟩

lemma *start-config-string-tape1*[*simp*]:
assumes *2* ≤ *k*
shows *snd* (*start-config-string k w*) ::: *1* = ⌊[]⌋
⟨*proof*⟩

lemma *config-tape-snd*[*simp*]:
(*snd cfg* ::: *j*) = (*cfg* <:> *j*)
⟨*proof*⟩

lemma *fst-start-config-string* [*simp*]:
fst (*start-config-string k w*) = *0*
⟨*proof*⟩

lemma *act-move-bound*:
fixes *tp tp′* :: *tape* **and** *a* :: *action*
assumes *tp′* = *act a tp*
shows *hd-span* (*q*, [*tp′*] @ *tps*) ≤ *Suc* (*hd-span* (*q*, [*tp*] @ *tps*))
⟨*proof*⟩

lemma *nth-map2*:
assumes *length xs* = *length ys* *i* < *length xs*
shows (*map2 f xs ys*) ! *i* = *f* (*xs* ! *i*) (*ys* ! *i*)
⟨*proof*⟩

lemma *map2-act-nth0*:
assumes *length as* = *length ts* *ts* ≠ []
shows *map2 act as ts* ! *0* = *act* (*as* ! *0*) (*ts* ! *0*)
⟨*proof*⟩

lemma *map2-act-nth0-rewrite*:
assumes *as* ≠ [] *ts* ≠ []
shows *map2 act as ts* ! *0* = *act* (*as* ! *0*) (*ts* ! *0*)
⟨*proof*⟩

lemma *tapes-pos-by-no-unfold* [*simp*]:
(*tps* :: *tape list*) :#: *j* = *snd* (*tps* ! *j*)
⟨*proof*⟩

The head position on the input tape is bounded by the number of steps taken.

lemma *sem-head-move-bound*:
fixes *cmd* :: *command* **and** *cfg cfg′* :: *config*
assumes *pc* : *proper-command* ||*cfg*|| *cmd*
and *step*: *cfg′* = *sem cmd cfg*
shows *hd-span cfg′* ≤ *Suc* (*hd-span cfg*)
⟨*proof*⟩

lemma *exe-head-move-bound*:
fixes *cfg cfg′* :: *config*
assumes *pm* : *proper-machine* ||*cfg*|| *M*
and *step*: *cfg′* = *exe M cfg*
shows *hd-span cfg′* ≤ *Suc* (*hd-span cfg*)
⟨*proof*⟩

lemma *execute-head-pos-le-time*:
fixes *M* :: *machine* **and** *cfg cfg′* :: *config*
assumes *pm* : *proper-machine* ||*cfg*|| *M*
and *step*: *cfg′* = *execute M cfg t*
shows *hd-span cfg′* ≤ *hd-span cfg* + *t*
⟨*proof*⟩

1.2 Configuration Indistinguishability

Two machine configurations are completely indistinguishable to the execution semantics for at least one step if their control states match, their non-input tapes are identical, and their input tapes agree up to the strict bounds of where the read head can currently be located.

definition *indist* :: *nat* ⇒ *config* ⇒ *config* ⇒ *bool* **where**
indist C cfg1 cfg2 ≡
||*cfg1*|| = ||*cfg2*|| ∧
fst cfg1 = *fst cfg2* ∧
(∀ *j*>*0*. *snd cfg1* ! *j* = *snd cfg2* ! *j*) ∧

$(\forall\, i{\leq}C.\ (cfg1 <:> 0)\ i = (cfg2 <:> 0)\ i) \wedge$
$hd\text{-}span\ cfg1 = hd\text{-}span\ cfg2 \wedge hd\text{-}span\ cfg1 \leq C$

lemma *start-config-string-conv* [*simp*]:
start-config-string k w = start-config k (*string-to-symbols w*)
⟨*proof*⟩

lemma *big-oh-constE*:
assumes *big-oh T* (λ*n. Suc 0*)
obtains *C* **where** ∀ *n. T n* ≤ *C*
⟨*proof*⟩

lemma *contents-eq-on-prefix*:
assumes *EQ*: *take* (*Suc C*) *w* = *take* (*Suc C*) *v*
shows ∀ *i*≤*Suc C*.
⌊*map* (λ*b. if b then 3 else 2*) *w*⌋ *i* =
⌊*map* (λ*b. if b then 3 else 2*) *v*⌋ *i*
⟨*proof*⟩

lemma *DTIME1-const-time*:
assumes *L* ∈ *DTIME* (λ*n.1*)
obtains *k G M C* **where**
turing-machine k G M
computes-in-time k M (*characteristic L*) (λ-. *C*)
⟨*proof*⟩

lemma *start-input-prefix-eq*:
assumes *take* (*Suc C*) *w* = *take* (*Suc C*) *v*
and *i* ≤ *C*
shows (*start-config-string k w* <:> *0*) *i* =
(*start-config-string k v* <:> *0*) *i*
⟨*proof*⟩

lemma *indist-start*:
assumes *EQ*: *take* (*Suc C*) *w* = *take* (*Suc C*) *v*
and *k*: *2* ≤ *k*
shows *indist C* (*start-config-string k w*) (*start-config-string k v*)
⟨*proof*⟩

lemma *config-read-eq-indist*:
assumes *indist C cfg1 cfg2*
shows *config-read cfg1* = *config-read cfg2*
⟨*proof*⟩

lemma *nth-map-act-zip*:
assumes *j*: *j* < *length as* **and** *len*: *length as* = *length tps*
shows *map* (λ(*a*,*tp*). *act a tp*) (*zip as tps*) ! *j* = *act* (*as* ! *j*) (*tps* ! *j*)
⟨*proof*⟩

lemma *nth-Nil* [*simp*]: ([] :: *'a list*) ! *n* = *undefined n*
⟨*proof*⟩

lemma *nth-overflow* [*simp*]:
length xs ≤ *n* ⟹ *xs* ! *n* = *undefined* (*n* − *length xs*)
⟨*proof*⟩

lemma *act-prefix-eq*:
assumes *cont*: ∀ *i*≤*C*. *fst* (*tp1* :: *tape*) *i* = *fst tp2 i*
and *head*: *snd tp1* = *snd tp2*
shows ∀ *i*≤*C*. *fst* (*act a tp1*) *i* = *fst* (*act a tp2*) *i*
⟨*proof*⟩

lemma *act-head-eq*:
assumes *snd* (*tp1*:: *tape*) = *snd tp2*
shows *snd* (*act a tp1*) = *snd* (*act a tp2*)
⟨*proof*⟩

lemma *indist-step*:
assumes *inv* : *indist C cfg1 cfg2*
and *pc* : *proper-command* ||*cfg1*|| *cmd*
and *bounds*: *hd-span cfg1* < *C*
and *s1* : *cfg1′* = *sem cmd cfg1*
and *s2* : *cfg2′* = *sem cmd cfg2*
shows *indist C cfg1′ cfg2′*
⟨*proof*⟩

lemma *indist-execute*:
assumes *pm*: *proper-machine* ||*cfg*|| *M*
and *inv0*: *indist C cfg1 cfg2*
and *len*: ||*cfg1*|| = ||*cfg*||
and *hd-bound*: *hd-span cfg1* + *t* ≤ *C*
shows *indist C* (*execute M cfg1 t*) (*execute M cfg2 t*)
⟨*proof*⟩

1.3 Constant-Time Dependence on Prefix

By applying the indist_execute invariant up to step C, we can easily show that two executions starting with the same prefix must deposit the exact same output onto tape 1.

lemma *exec-same-output-C*:
fixes *k C t* :: *nat* **and** *w v* :: *string* **and** *G* :: *nat* **and** *M* :: *machine*
defines *cfgw* ≡ *start-config-string k w*
and *cfgv* ≡ *start-config-string k v*
assumes *TM* : *turing-machine k G M*
and *EQ* : *take* (*Suc C*) *w* = *take* (*Suc C*) *v*
and *LE* : *t* ≤ *C*
shows (*execute M cfgw t* <:> *1*) = (*execute M cfgv t* <:> *1*)
⟨*proof*⟩

lemma *string-to-contents-inj*:
assumes *string-to-contents xs* = *string-to-contents ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *constant-time-depends-on-prefix*:
fixes *f* :: *string* ⇒ *string*
assumes *TM*: *turing-machine k G M*
and *CT*: *computes-in-time k M f* (λ-. *C*)
and *EQ*: *take* (*Suc C*) *w* = *take* (*Suc C*) *v*
shows *f w* = *f v*
⟨*proof*⟩

lemma *DTIME1-depends-on-prefix*:
fixes *L* :: *language*
assumes *L* ∈ *DTIME* (λ*n*. *1*)
obtains *C H* **where** ∀ *w*. *w* ∈ *L* ⟷ *take* (*Suc C*) *w* ∈ *H*
⟨*proof*⟩

1.4 Main Theorem: SAT ∉ DTIME(1)

The core diagonalization-style argument works by constructing two strings: v, which is properly formatted but trivially unsatisfiable, and w, which shares the same $O(1)$ prefix with v but is engineered via an odd length to be syntactically invalid (and thus conditionally belongs to SAT due to the library's inverted validity mapping for the default case). The hypothesized $O(1)$ decider contradicts itself on this shared prefix.

lemma *SAT-not-prefix-language*:
shows ¬ (∃ *C H*. ∀ *w*. *w* ∈ *SAT* ⟷ *take* (*Suc C*) *w* ∈ *H*)
⟨*proof*⟩

theorem *SAT-not-const-time*:
shows *SAT* ∉ *DTIME* (λ*n*. *1*)
⟨*proof*⟩

end

References

[1] F. J. Balbach. The cook-levin theorem. *Archive of Formal Proofs*, January 2023. `https://isa-afp.org/entries/Cook_Levin.html`, Formal proof development.