Turing Machines

Bruce Merry

University of Cape Town

10 May 2012

Outline

Outline

What Are Turing Machines?

- Invented by Alan Turing
- Hypothetical machines
- Formalise "computation"

Alan Turing, 1912–1954

What Are Turing Machines?

Each Turing machine consists of

- A finite set of symbols, including a special blank symbol (□)

What Are Turing Machines?

Each Turing machine consists of

- A finite set of symbols, including a special blank symbol ($\Box$)
- A finite set of states, including a start state

What Are Turing Machines?

Each Turing machine consists of

- A finite set of symbols, including a special blank symbol (□)
- A finite set of states, including a start state
- A tape that is infinite in both directions, containing finitely many non-blank symbols

What Are Turing Machines?

Each Turing machine consists of

- A finite set of symbols, including a special blank symbol ($\square$)
- A finite set of states, including a start state
- A tape that is infinite in both directions, containing finitely many non-blank symbols
- A head which points at one position on the tape

What Are Turing Machines?

Each Turing machine consists of

- A finite set of symbols, including a special blank symbol (□)
- A finite set of states, including a start state
- A tape that is infinite in both directions, containing finitely many non-blank symbols
- A head which points at one position on the tape
- A set of transitions

What Are Turing Machines?

Each Turing machine consists of

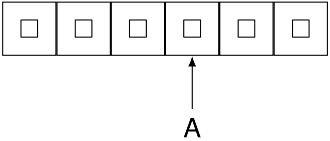
- A finite set of symbols, including a special blank symbol ($\square$)
- A finite set of states, including a start state
- A tape that is infinite in both directions, containing finitely many non-blank symbols
- A head which points at one position on the tape
- A set of transitions
    - If in state s_i and tape contains q_j, write q_k then move left/right and change to state s_m

Turing machine operation

1. Find a matching rule for the current state and tape symbol
2. If none found, halt
3. Otherwise, apply the rule and repeat

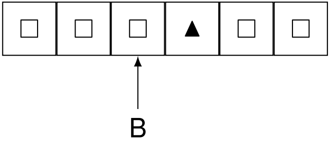
Example

- Two symbols, $\square$ and $\blacktriangle$
- Two states, A and B
- Four rules
  - A $\square$: $\blacktriangle \leftarrow$ B
  - A $\blacktriangle$: $\square \rightarrow$ B
  - B $\square$: $\blacktriangle \rightarrow$ A
  - B $\blacktriangle$: $\square \leftarrow$ A



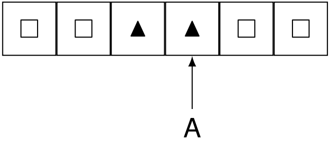
Example

- Two symbols, □ and ▲
- Two states, A and B
- Four rules
  - A □: ▲ $\leftarrow$ B
  - A ▲: □ $\rightarrow$ B
  - B □: ▲ $\rightarrow$ A
  - B ▲: □ $\leftarrow$ A



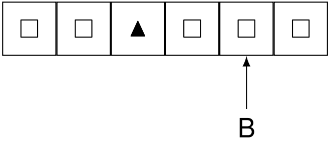
Example

- Two symbols, □ and ▲
- Two states, A and B
- Four rules
  - A □: ▲ $\leftarrow$ B
  - A ▲: □ $\rightarrow$ B
  - B □: ▲ $\rightarrow$ A
  - B ▲: □ $\leftarrow$ A



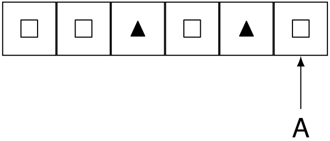
Example

- Two symbols, □ and ▲
- Two states, A and B
- Four rules
  - A □: ▲ $\leftarrow$ B
  - A ▲: □ $\rightarrow$ B
  - B □: ▲ $\rightarrow$ A
  - B ▲: □ $\leftarrow$ A



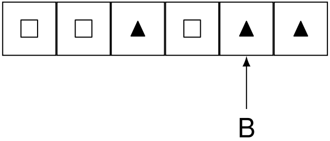
Example

- Two symbols, □ and ▲
- Two states, A and B
- Four rules
  - A □: ▲ $\leftarrow$ B
  - A ▲: □ $\rightarrow$ B
  - B □: ▲ $\rightarrow$ A
  - B ▲: □ $\leftarrow$ A



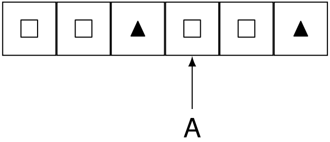
Example

- Two symbols, $\square$ and $\blacktriangle$
- Two states, A and B
- Four rules
  - A $\square$: $\blacktriangle \leftarrow$ B
  - A $\blacktriangle$: $\square \rightarrow$ B
  - B $\square$: $\blacktriangle \rightarrow$ A
  - B $\blacktriangle$: $\square \leftarrow$ A



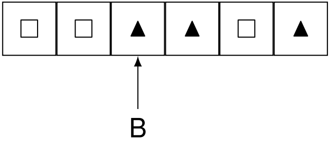
Example

- Two symbols, □ and ▲
- Two states, A and B
- Four rules
  - A □: ▲ $\leftarrow$ B
  - A ▲: □ $\rightarrow$ B
  - B □: ▲ $\rightarrow$ A
  - B ▲: □ $\leftarrow$ A



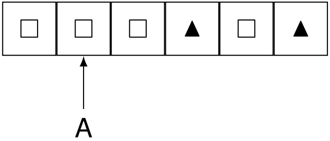
Example

- Two symbols, □ and ▲
- Two states, A and B
- Four rules
  - A □: ▲ $\leftarrow$ B
  - A ▲: □ $\rightarrow$ B
  - B □: ▲ $\rightarrow$ A
  - B ▲: □ $\leftarrow$ A

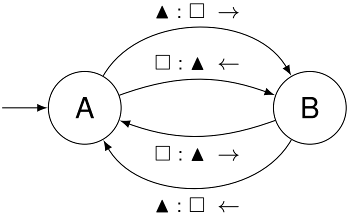


Example

- Two symbols, □ and ▲
- Two states, A and B
- Four rules
  - A □: ▲ ← B
  - A ▲: □ → B
  - B □: ▲ → A
  - B ▲: □ ← A



Graph notation



Variations

- Move *or* write
- Explicit halt state
- Only one infinite direction

Outline

Shorthand

- $q_j :\leftarrow$ Move but do not write

Shorthand

- $q_j :\leftarrow$ Move but do not write
- $q_1/q_2/q_3 : q_k \rightarrow$ Match any of q_1, q_2, q_3

Shorthand

- $q_j :\leftarrow$ Move but do not write
- $q_1/q_2/q_3 : q_k \rightarrow$ Match any of q_1, q_2, q_3
- $\neg q_j : q_k \rightarrow$ Match any except q_j

Shorthand

- $q_j :\leftarrow$ Move but do not write
- $q_1/q_2/q_3 : q_k \rightarrow$ Match any of q_1, q_2, q_3
- $\neg q_j : q_k \rightarrow$ Match any except q_j
- $* :\rightarrow$ Move right on any symbol

Shorthand

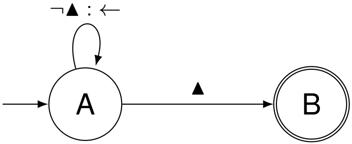
- $q_j :\leftarrow$ Move but do not write
- $q_1/q_2/q_3 : q_k \rightarrow$ Match any of q_1, q_2, q_3
- $\neg q_j : q_k \rightarrow$ Match any except q_j
- $* :\rightarrow$ Move right on any symbol
- $q_j : q_k$ Write but do not move

Shorthand

- $q_j :\leftarrow$ Move but do not write
- $q_1/q_2/q_3 : q_k \rightarrow$ Match any of q_1, q_2, q_3
- $\neg q_j : q_k \rightarrow$ Match any except q_j
- $* :\rightarrow$ Move right on any symbol
- $q_j : q_k$ Write but do not move
- q_j Change state only

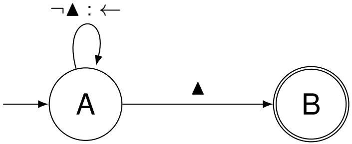
Composing Machines

Find first ▲ to the left: FL(▲)

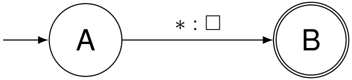


Composing Machines

Find first ▲ to the left: FL(▲)

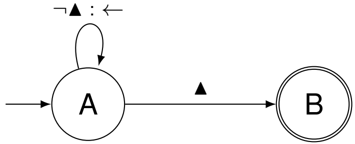


Write a blank: W(□)

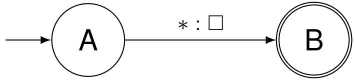


Composing Machines

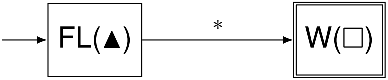
Find first ▲ to the left: FL(▲)



Write a blank: W(□)



Find first ▲to the left and blank it:



Transforming Machines

Simpler TMs can be used to simulate more general forms

- Requires a procedure for “compiling” the more complex machine
- Proves that the simpler machine is just as powerful

Example: TMs as defined can be transformed to TMs with half-infinite tape

Half-infinite Tapes

$\cdots$	q_3	q_{-2}	q_{-1}	q_0	q_1	q_2	q_3	$\cdots$

can instead be encoded as

$\blacksquare$	q_0	q_{-1}	q_1	q_{-2}	q_2	q_{-3}	q_3	$\cdots$

Half-infinite Tapes

The machine must be modified to:

- Encode the initial input

Half-infinite Tapes

The machine must be modified to:

- Encode the initial input
- Position the head on q_0

Half-infinite Tapes

The machine must be modified to:

- Encode the initial input
- Position the head on q_0
- Perform $\leftarrow$ and $\rightarrow$ correctly

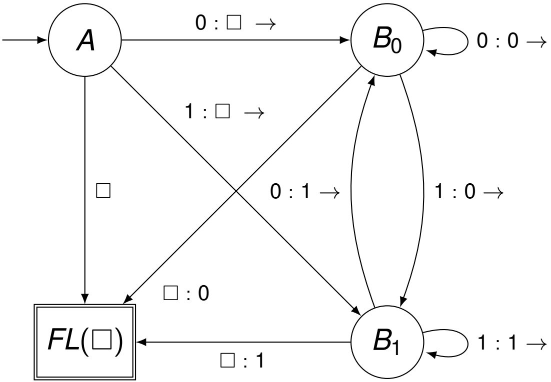
Half-infinite Tapes

The machine must be modified to:

- Encode the initial input
- Position the head on q_0
- Perform $\leftarrow$ and $\rightarrow$ correctly
- Keep track of which half it is in

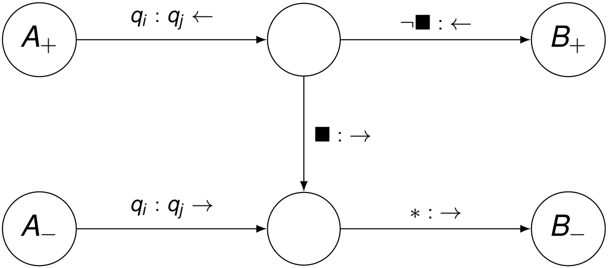
Shift To The Right

Insert a blank, shifting non-blank to the right



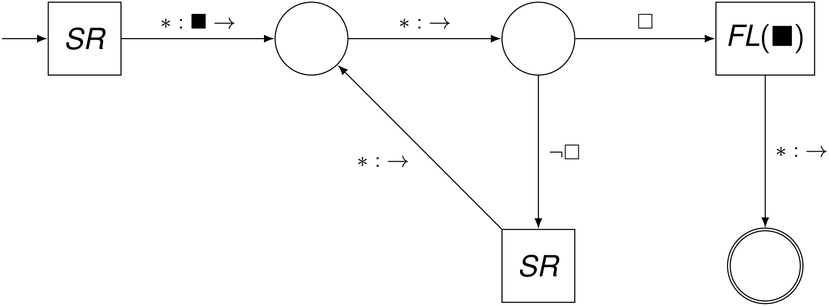
Turing Machine Transformation

- Each state A becomes two state A_+ and A_-.
- Each transition $A\ q_i : q_j \leftarrow B$ becomes



Input Preparation

Interleaves input with blanks and places ■



Multi-tape Machines

- N tapes, each with a separate head
- A current tape
- Transitions specify which tape to use next
- Input on the initial tape, others blank

Multi-tape Machines

A multi-tape machine can transformed to a single-tape one

- For each tape, add another with a head marker

$q_{0,0}$	$q_{0,1}$	$q_{0,2}$	$q_{0,3}$	$\cdots$
$\square$	$\square$	$\uparrow$	$\square$	$\cdots$
$q_{1,0}$	$q_{1,1}$	$q_{1,2}$	$q_{1,3}$	$\cdots$
$\uparrow$	$\square$	$\square$	$\square$	$\cdots$

Multi-tape Machines

A multi-tape machine can transformed to a single-tape one

- For each tape, add another with a head marker

$q_{0,0}$	$q_{0,1}$	$q_{0,2}$	$q_{0,3}$	$\cdots$
$\square$	$\square$	$\uparrow$	$\square$	$\cdots$
$q_{1,0}$	$q_{1,1}$	$q_{1,2}$	$q_{1,3}$	$\cdots$
$\uparrow$	$\square$	$\square$	$\square$	$\cdots$

- Interleave these $2N$ tapes into one

$\blacksquare$	$q_{0,0}$	$\square$	$q_{1,0}$	$\uparrow$	$q_{0,1}$	$\square$	$q_{1,1}$	$\square$	$q_{0,2}$	$\uparrow$	$q_{1,2}$	$\square$	$\cdots$

Outline

Turing Completeness

A system is Turing-complete if it can emulate any Turing Machine (ignoring finite memory limits)

- All real-world programming languages
- Many joke programming language e.g. INTERCAL, Whitespace
- Lambda calculus
- Partial recursive functions

Surprising Turing-Complete Systems

- Conway's Game of Life
- Wang Tiles
- C++ at compile time

Outline

Encoding Turing Machines

A Turing Machine T can be encoded as a string $E(T)$ in a fixed alphabet e.g.

- A $\square$: $\blacktriangle \leftarrow$ B
- A $\blacktriangle$: $\square \rightarrow$ B
- B $\square$: $\blacktriangle \rightarrow$ A
- B $\blacktriangle$: $\square \leftarrow$ A

Encoding Turing Machines

A Turing Machine T can be encoded as a string $E(T)$ in a fixed alphabet e.g.

- 1 $\square$: $\blacktriangle \leftarrow$ 01
- 1 $\blacktriangle$: $\square \rightarrow$ 01
- 01 $\square$: $\blacktriangle \rightarrow$ 1
- 01 $\blacktriangle$: $\square \leftarrow$ 1

Encoding Turing Machines

A Turing Machine T can be encoded as a string $E(T)$ in a fixed alphabet e.g.

- 1 1: 01 $\leftarrow$ 01
- 1 01: 1 $\rightarrow$ 01
- 01 1: 01 $\rightarrow$ 1
- 01 01: 1 $\leftarrow$ 1

Encoding Turing Machines

A Turing Machine T can be encoded as a string $E(T)$ in a fixed alphabet e.g.

- 1 1: 01 1 01
- 1 01: 1 01 01
- 01 1: 01 01 1
- 01 01: 1 1 1

Encoding Turing Machines

A Turing Machine T can be encoded as a string $E(T)$ in a fixed alphabet e.g.

- 1 1: 01 1 01
- 1 01: 1 01 01
- 01 1: 01 01 1
- 01 01: 1 1 1

1101101■10110101■01101011■0101111

Universal Turing Machines

There exists a Universal Turing Machine U

- Take a machine T and an input I
- Run machine U on the tape $E(T)\blacklozenge I$
- The result will be the same as running T on I

U operates like a stored-program computer

Outline

Languages In Computability

A language is a set of strings in an alphabet

- Each string in a language is finite
- A language can be infinite

Examples of Languages

- The set of all Bulgarian words

Examples of Languages

- The set of all Bulgarian words
- The set of all English sentences

Examples of Languages

- The set of all Bulgarian words
- The set of all English sentences
- The set of all valid C++ programs

Examples of Languages

- The set of all Bulgarian words
- The set of all English sentences
- The set of all valid C++ programs
- The set of all prime numbers

Examples of Languages

- The set of all Bulgarian words
- The set of all English sentences
- The set of all valid C++ programs
- The set of all prime numbers
- The set of all encodings of Turing machines that halt

Examples of Languages

- The set of all Bulgarian words
- The set of all English sentences
- The set of all valid C++ programs
- The set of all prime numbers
- The set of all encodings of Turing machines that halt
- The set of all formal proofs

Languages and Turing Machines

Turing Machines can classify strings with three outcomes

- Halt in an accept state
- Halt in a reject state
- Run forever

Recursive Languages

L is recursive or Turing-decidable if there is a TM T such that

- T always halts (either accepts or rejects)
- T accepts exactly the strings in L

Recursively Enumerable Languages

L is recursively enumerable if there is a TM T such that

- T accepts every string in L
- T does not halt given a string not in L

Outline

1. Basics
   - Definition
   - Building programs
   - Turing Completeness
2. Computability
   - Universal Machines
   - Languages
   - The Halting Problem
3. Complexity
   - Non-determinism
   - Complexity classes
   - Satisfiability

The Halting Problem

For a specific Turing Machine T

- Does T halt given a blank tape?
- Does $E(T)$ belong to the language of Turing machines that halt?

More generally:

- Is the language recursively-enumerable?
- Is the language Turing-decidable?

The Halting Problem

Suppose H is a Turing Machine that takes $E(T)$ as input and decides whether T halts on blank input.

The Halting Problem

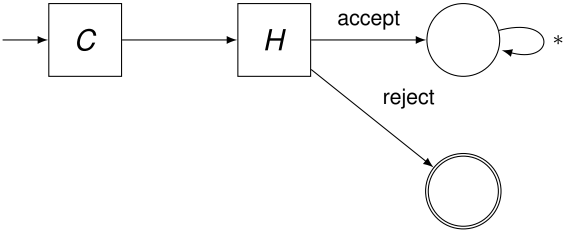
Suppose H is a Turing Machine that takes $E(T)$ as input and decides whether T halts on blank input. Let

- C transform $E(T)$ to $E(T')$, where T' first writes $E(T)$ to the tape then executes T

The Halting Problem

Suppose H is a Turing Machine that takes $E(T)$ as input and decides whether T halts on blank input. Let

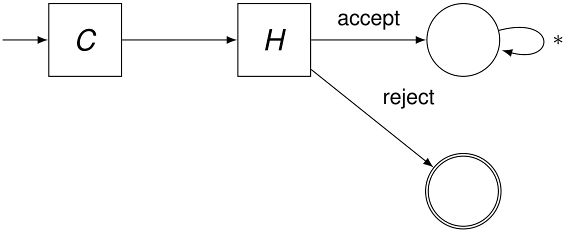
- C transform $E(T)$ to $E(T')$, where T' first writes $E(T)$ to the tape then executes T
- F be the machine



The Halting Problem

Suppose H is a Turing Machine that takes $E(T)$ as input and decides whether T halts on blank input. Let

- C transform $E(T)$ to $E(T')$, where T' first writes $E(T)$ to the tape then executes T
- F be the machine

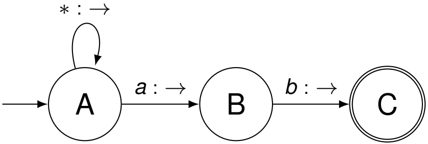


Then F run on $E(F)$ halts iff it does not.

Outline

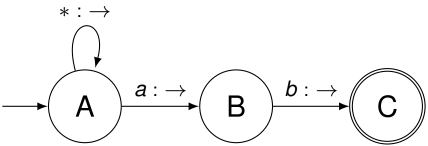
Non-deterministic Turing Machines

What if transitions are ambiguous?



Non-deterministic Turing Machines

What if transitions are ambiguous?



A non-deterministic Turing Machine (NDTM) halts if there is *any* choice of transitions that would lead to the halt state.

This machine is equivalent to the regex `.*ab`

Non-deterministic Computing Power

Anything computable with a NDTM is also computable with a TM

- A TM can simulate all possible states of a NDTM
- This could be far “slower” than the NDTM

Outline

Polynomial Time

A Turing Machine runs in polynomial time if given an input of size N it halts within $f(N)$ steps, for some polynomial f.

A language is in P if a polynomial-time Turing Machine can decide it.

Polynomial Time

A Turing Machine runs in polynomial time if given an input of size N it halts within $f(N)$ steps, for some polynomial f.

A language is in P if a polynomial-time Turing Machine can decide it.

Exercise: There is a language L and a machine T which halts in polynomial time given a string from L, and never terminates given a string not from L. Prove that L is in P.

Non-deterministic Polynomial Time

A language is in NP if a non-deterministic Turing Machine can accept it in polynomial time.

Every member of such a language has a certificate that can be validated in polynomial time on a normal Turing Machine.

Reductions

Consider two languages:

- A, which has the alphabet Σ_1 ($A \subseteq \Sigma_1^*$)
- B, which has the alphabet Σ_2 ($B \subseteq \Sigma_2^*$)

A reduction from A to B is a computable function

$$f : \Sigma_1^* \to \Sigma_2^*$$

such that

$$a \in A \iff f(a) \in B.$$

Reductions

If A can be reduced to B, then an algorithm for deciding A is:

- Compute $f(a)$
- Test whether $f(a) \in B$, using an algorithm for B

Thus, B is at least as "hard" as A.

Reduction example

A N pilots are available to fly N planes. Each pilot is only qualified to fly some of the planes. Is it possible to assign each plane a different qualified pilot? (*Bipartite Matching*)

Reduction example

A N pilots are available to fly N planes. Each pilot is only qualified to fly some of the planes. Is it possible to assign each plane a different qualified pilot? (*Bipartite Matching*)

B There are E one-way network connections between V computers, each of which has a capacity. Is it possible for computer P to send information to computer Q at a rate of at least R? (*Network flow*)

Bipartite matching can be reduced to network flow.

NP-Complete

A problem (language) L is in *NPC* if

- it is in NP; and
- *any* problem in NP can be reduced to L in polynomial time.

NP-Complete

A problem (language) L is in *NPC* if

- it is in NP; and
- *any* problem in NP can be reduced to L in polynomial time.

If any problem in NPC can be solved in polynomial time, then $P = NP$.

NP-Hard

A problem is NP-Hard if some problem from *NP* can be reduced to it

- Can include non-decision problems e.g. Travelling Salesman
- *At least* as hard as any problem in NPC

Outline

Boolean Satisfiability

Given a boolean expression in N variables, can values for the variables be found to make it true? e.g.

$$(a \lor b \lor \neg c) \land (\neg b \lor c \lor \neg d) \land (\neg a \lor b \lor d)$$

SAT is in NP

- This is trivial: a non-determinisitic Turing Machine can "guess" a solution and verify it in polynomial time.

SAT is in NP

- This is trivial: a non-determinisitic Turing Machine can "guess" a solution and verify it in polynomial time.
- Equivalently, any assignment that satisfies the condition forms a certificate.

SAT is in NPC

Proof Outline

- Take a language L in NP

SAT is in NPC

Proof Outline

- Take a language L in NP
- Take the NDTM T that accepts L

SAT is in NPC

Proof Outline

- Take a language L in NP
- Take the NDTM T that accepts L
- Construct a boolean expression that can be satisfied iff T terminates

SAT is in NPC

Proof Outline

- Take a language L in NP
- Take the NDTM T that accepts L
- Construct a boolean expression that can be satisfied iff T terminates
- L has now been reduced to SAT

SAT is in NPC

Variables

Values of variables correspond to *one* possible execution trace

- $Q_{t,i,q}$ After t steps, the symbol i to the right of the head is q (left if $i < 0$)
- $S_{t,s}$ After t steps, the machine is in state s
- $M_{t,k}$ After t steps, the next transition is via rule k

SAT is in NPC

How many variables?

L is in NP, so an input of length n can be accepted in at most $P(n)$ steps:

- t need only range from 0 to $P(n)$
- i need only range from $-P(n)$ to $P(n)$

SAT is in NPC

Constraints

- Initial state: $Q_{0,i,q}$ iff the tape at i initially contains q

SAT is in NPC

Constraints

- Initial state: $Q_{0,i,q}$ iff the tape at i initially contains q
- Single symbol: $\neg(Q_{t,i,q} \wedge Q_{t,i,q'})$ for $q \neq q'$
- Single transition: $\neg(M_{t,k} \wedge M_{t,k'})$ for $k \neq k'$
- Single state: $\neg(S_{t,s} \wedge S_{t,s'})$ for $s \neq s'$

SAT is in NPC

Constraints

- Initial state: $Q_{0,i,q}$ iff the tape at i initially contains q
- Single symbol: $\neg(Q_{t,i,q} \wedge Q_{t,i,q'})$ for $q \neq q'$
- Single transition: $\neg(M_{t,k} \wedge M_{t,k'})$ for $k \neq k'$
- Single state: $\neg(S_{t,s} \wedge S_{t,s'})$ for $s \neq s'$
- Transition: if state $s \neq H$, symbol q allows transitions $k_1, \ldots, k_m$ then $(S_{t,s} \wedge Q_{t,0,q}) \implies (M_{t,k_1} \vee \ldots \vee M_{t,k_m})$

SAT is in NPC

Constraints

- Initial state: $Q_{0,i,q}$ iff the tape at i initially contains q
- Single symbol: $\neg(Q_{t,i,q} \wedge Q_{t,i,q'})$ for $q \neq q'$
- Single transition: $\neg(M_{t,k} \wedge M_{t,k'})$ for $k \neq k'$
- Single state: $\neg(S_{t,s} \wedge S_{t,s'})$ for $s \neq s'$
- Transition: if state $s \neq H$, symbol q allows transitions $k_1, \ldots, k_m$ then $(S_{t,s} \wedge Q_{t,0,q}) \implies (M_{t,k_1} \vee \ldots \vee M_{t,k_m})$
- Timestep: if transition k is $(q' \leftarrow s')$ then
  - $(M_{t,k} \wedge Q_{t,i,q}) \implies Q_{t+1,i+1,q}$ for $i \neq 0$
  - $M_{t,k} \implies Q_{t+1,1,q'}$
  - $M_{t,k} \implies S_{t+1,s'}$

SAT is in NPC

Constraints

- Initial state: $Q_{0,i,q}$ iff the tape at i initially contains q
- Single symbol: $\neg(Q_{t,i,q} \wedge Q_{t,i,q'})$ for $q \neq q'$
- Single transition: $\neg(M_{t,k} \wedge M_{t,k'})$ for $k \neq k'$
- Single state: $\neg(S_{t,s} \wedge S_{t,s'})$ for $s \neq s'$
- Transition: if state $s \neq H$, symbol q allows transitions $k_1, \dots, k_m$ then $(S_{t,s} \wedge Q_{t,0,q}) \implies (M_{t,k_1} \vee \dots \vee M_{t,k_m})$
- Timestep: if transition k is $(q' \leftarrow s')$ then
  - $(M_{t,k} \wedge Q_{t,i,q}) \implies Q_{t+1,i+1,q}$ for $i \neq 0$
  - $M_{t,k} \implies Q_{t+1,1,q'}$
  - $M_{t,k} \implies S_{t+1,s'}$
- Halt: $S_{0,H} \vee S_{1,H} \vee \dots \vee S_{P(n),H}$

SAT is in NPC

Putting it all together

Final expression E is $\wedge$ of all the constraints

- If T can reach the halt state then E can be satisfied
- If E can be satisfied then $M_{t,k}$ gives a way for T to halt
- Therefore we've reduced L to satisfiability of E

Questions

?