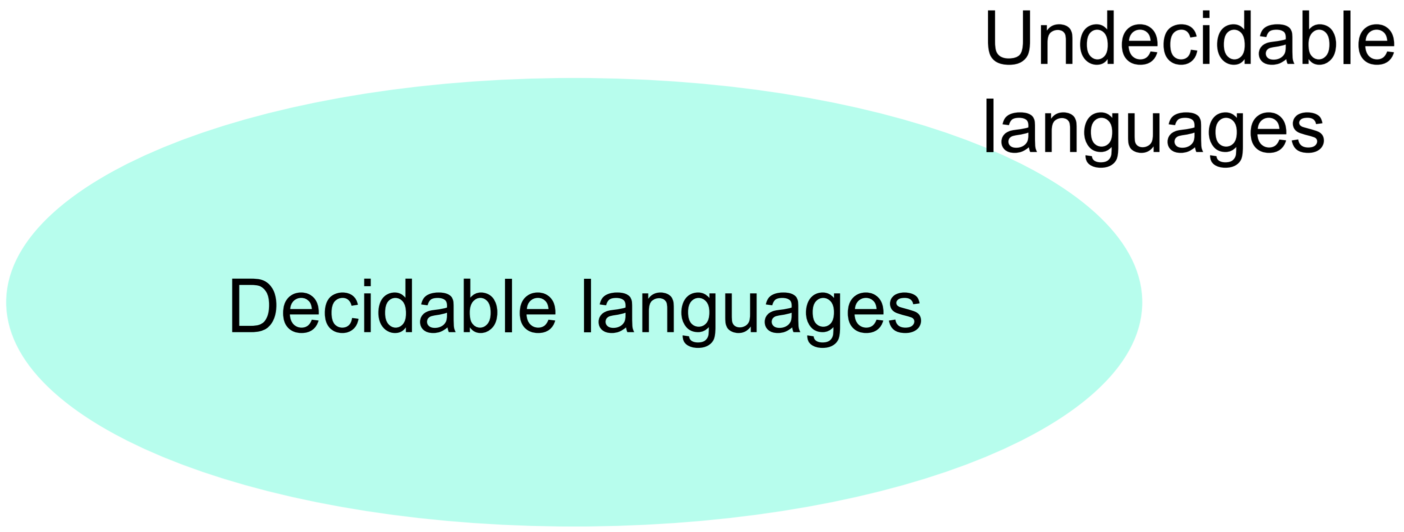
Complexity classes



- In the rest of this course, we focus on languages (decision problems) that are decidable.
- We want to know why some decidable languages are more difficult than the others.

Resources

We classify languages (decision problems) according to their requirement for resources.

Most notable resource: time

How do we measure "TIME" of a Turing machine?

Answer: the number of steps.

(If a Turing machine does not halt, the time is undefined.)

A Turing machine T is said to operate in time **t(n)** if, for <u>any</u> input x of length n, T takes <u>at most</u> **t(n)** steps to accept/reject x.

Time complexity classes

Definition: Let $TIME(n^2)$ = {L | L is a language decided by a k-tape Turing machine operating in time $O(n^2)$}.

A time complexity class

Big-O notation (Sipser p. 253): e.g., $3n^2$, $100n^2$, $13n^2 + 5n$, $13n^{1.8}$

In general, for any time function $t(n)$, define

$TIME(t(n))$ = {L | L is a language decided by a Turing machine operating in time $O(t(n))$ }.

Important complexity classes

$TIME(n)$, $TIME(n^2)$, $TIME(n^{20})$, $TIME(2^n)$, $TIME(n^{7.98})$, $TIME(2^{2^n})$, ...

A complexity class is important if it represents many real-life problems with the same degree of difficulty.

Classes to be studied in depth include:

- P = $TIME(n^{O(1)})$ (= $\bigcup_{k>0} TIME(n^k)$) --- polynomial time

Important complexity classes

TIME(n), TIME(n^2), TIME(n^{20}),TIME(2^n), TIME($n^{7.98}$), TIME(2^{2^n}), ...

A complexity class is important if it represents many real-life problems with the same degree of difficulty.

Classes to be studied in depth include:

- P = TIME ($n^{O(1)}$) (= $\bigcup_{k>0}$ TIME (n^k)) --- polynomial time

Many problems are not known to be in P:

Knapsack problem: Given positive integers (a_1, a_2 , ... , a_n, b), determine whether a subset of the a_i s has sum = b.

How to model the complexity of knapsack problem? Non-deterministic TM with polynomial time.

Deterministic Turing machines

The Turing machines we have studied so far has a <u>deterministic behavior</u>.

A Turing machine T running on an input w:

[start configuration for w]

↓

[configuration 1]

↓

[configuration 2]

↓

...

[accept/reject configuration]

At any particular time, we know exactly what is the next step of T. (Look at the transition function of T.)

Nondeterministic Turing machines

A nondeterministic Turing machine has the same definition as an ordinary (deterministic) Turing machine, except that it allows a choice of moves in each step.

Formally speaking,

1-tape TM: $\delta(q, a)$ defines one move, e.g., $\delta(q, a) = (q', b, R)$

1-tape NTM: $\delta(q, a)$ defines a constant number of moves, e.g., $\delta(q, a) = \{ (q', b, R), (q'', c, L), (q^*, d, R) \}$.

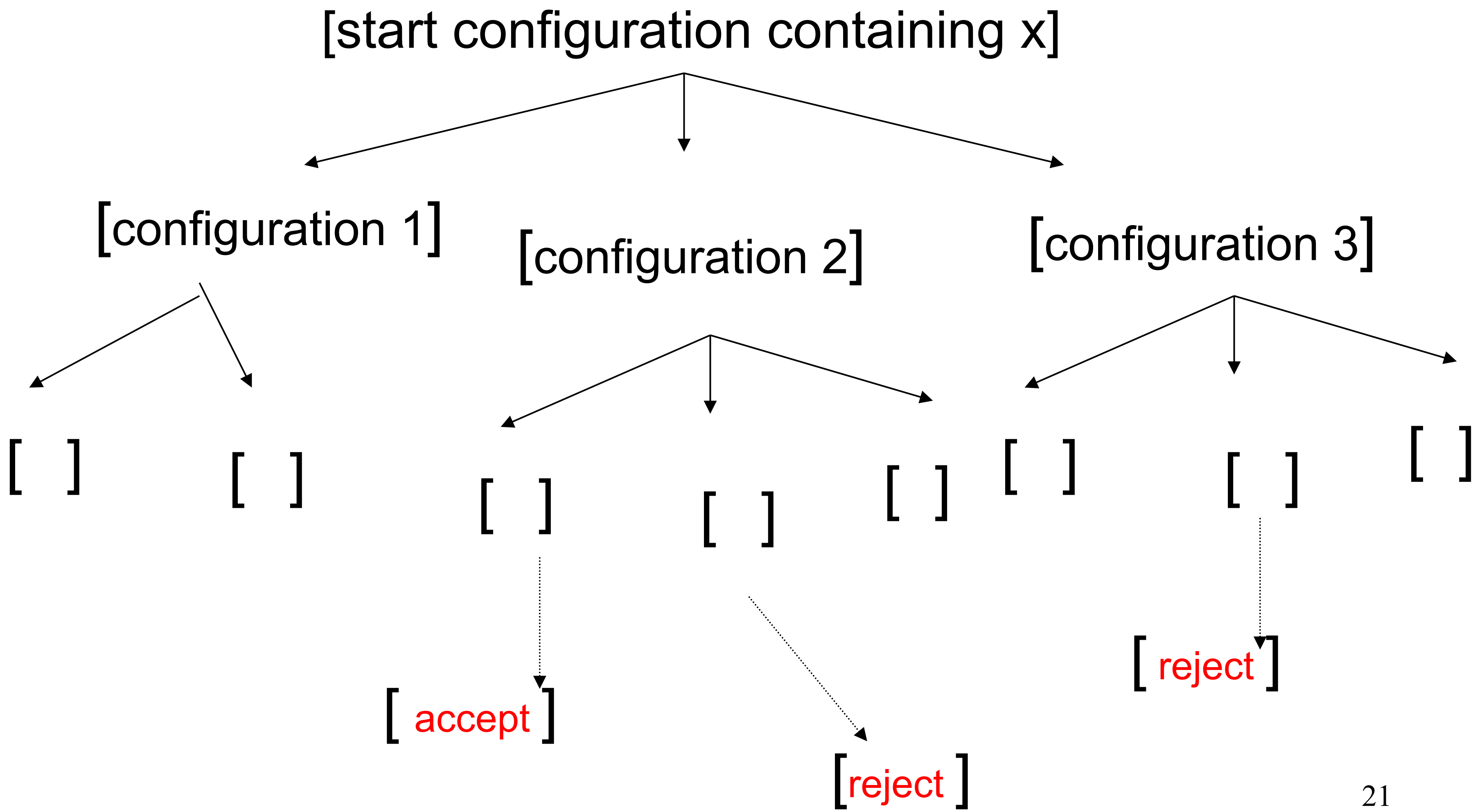
In general,

k-tape TM: $\delta(q, a_1, a_2, \ldots, a_k)$ is an ***element*** in $Q \times (\Gamma \times \{L,R\})^k$

k-tape NTM: $\delta(q, a_1, a_2, \ldots, a_k)$ is a ***subset*** of $Q \times (\Gamma \times \{L,R\})^k$

Computation of NTM

With respect to an input x, the computation of an NTM T defines a tree:



Nondeterministic computation

How does an NTM accept/reject an input x?

Consider the computation tree of x. If there is a path ending at an accepting configuration, x is said to be accepted.

NB. We say that x is rejected when all paths end at rejecting configurations.

Definitions

An NTM T is said to decide a language L if for all $x \in \Sigma^*$, if $x \in L$, then T accepts x; otherwise, T rejects x.

An NTM T is said to operate in time t(n) if, for any input x of length n, each path in the computation tree takes at most t(n) steps to accept/reject x.

Define NTIME(t(n)) = { L | L is a language decided by an NTM operating in time O(t(n))}.

Fact: TIME(t(n)) $\subseteq$ NTIME (t(n)).

Example of NTM

Let L = {1^n | n is not a prime number }.

(1^n is the unary representation of the integer n).

Claim: $L \in$ NTIME(n)

Note that if n is not a prime number, then n = pq for some integers p,q in the range [2, n-1].

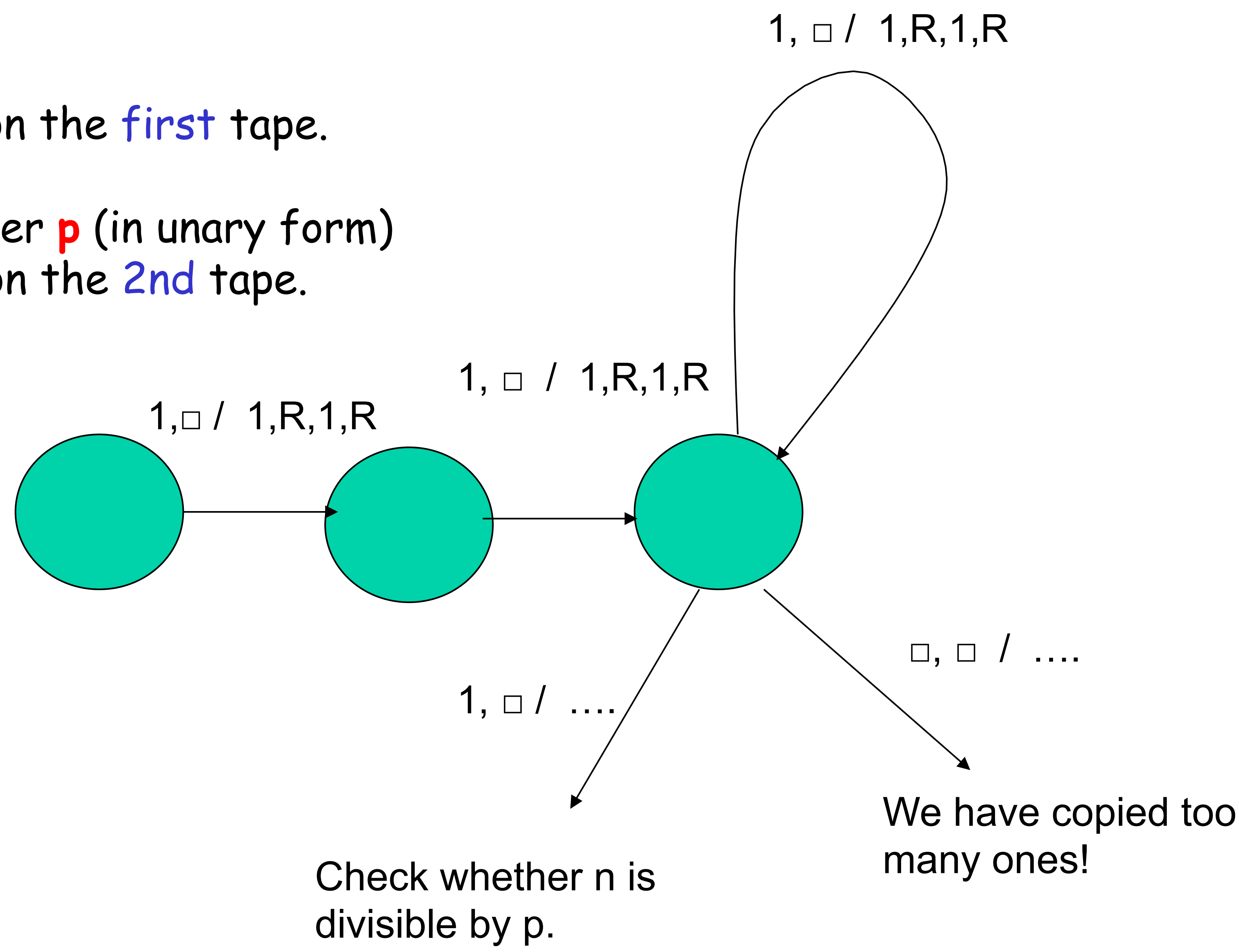
Design an NTM T to decide L as follows:

- Guess p (in unary): write two 1's, or three 1's , or …, or n-1 1's on tape 2.
- If n is divisible by p, then accepts, else rejects.

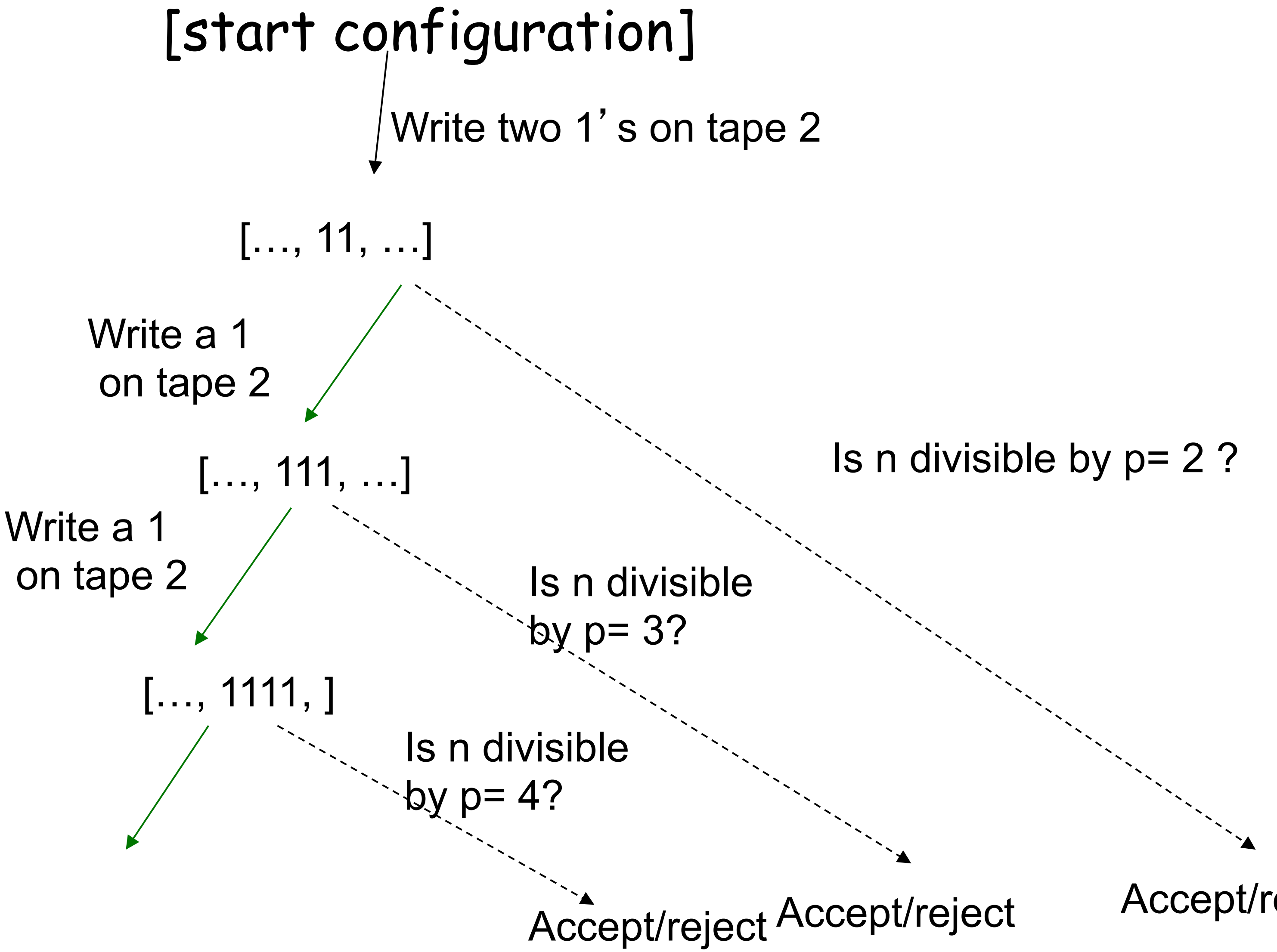
If x is in L, then at least one path in the computation tree of T ends with an accepting configuration.
If x is not in L, then all paths end with a rejecting configuration.

The input is on the first tape.

Guess a number p (in unary form) < n; p is put on the 2nd tape.



Computation tree



Example

Knapsack (subset sum) problem: Given n+1 positive integers (a_1, a_2 , … , a_n, b), determine whether a subset of the a_i s has the sum exactly equal to b.

NTM:

- guess S: write a possible bit vector V of length n (i.e., repeatedly write 0 or 1 n times);
- verify: A bit vector V defines a subset S; accept if the numbers specified by S has a sum equal to b; reject otherwise.

???

- NTM: Guessing a bit vector & verify …
- (D)TM: For every possible bit vector, verify …

P versus NP

P = $\cup_{i>0}$ TIME (n^i) (including TIME(n), TIME(n^2), etc.)
NP = $\cup_{i>0}$ NTIME (n^i) (including NTIME(n), NTIME(n^2), etc.)

Fact: P $\subseteq$ NP.

P captures all decision problems (languages) that can be solved (decided, resp.) efficiently on our computers.
NB. Time complexity classes like TIME (2^n) is too time consuming.

NP includes the languages that we can't solve efficiently on our computers (which is deterministic in nature), but which can be verified efficiently.

Polynomial time verifiable languages

L is polynomial time verifiable if there is a (deterministic) TM M running in $O(n^c)$ time such that for any input x of length n,

- if $x \in L$, there exists a string y such that M with x and y as inputs will accept.
- If $x \notin L$, then for all strings y, M with x and y as inputs will reject.

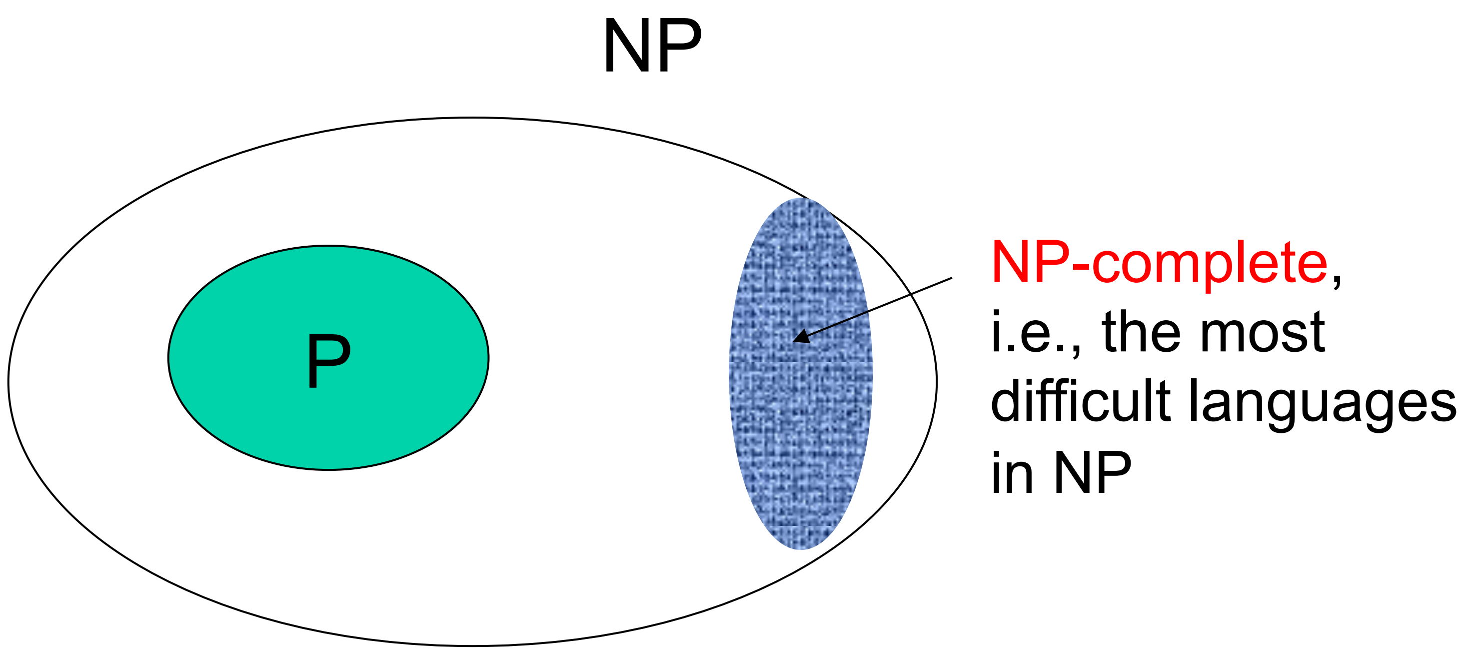
Fact. If a language L that is polynomial time verifiable, then $L \in$ NTIME (n^c) for some constant c.

A NTM can accept an input $x \in L$ by guessing the right proof y and then simulating M on x & y. An accepting path is of length $O(n^c)$.

The belief after 40+ years

We believe (but unable to prove at this point) that P is a proper subset of NP, i.e., P ≠ NP.

In particular, we believe that the most difficult problems in NP are not in P.



Polynomial-time reduction

A notion to compare the "hardness" of problems in NP.

Consider any languages $L_1, L_2 \subseteq \Sigma^*$.

L1 is said to be polynomial-time reducible to L_2, denoted $L_1 \leq_p L_2$, if there exists a function **f** such that

- for all $x \in \Sigma^*$, $x \in L_1$ **if and only if** **f**$(x) \in L_2$; and
- **f** is *computable* in polynomial time, i.e.,

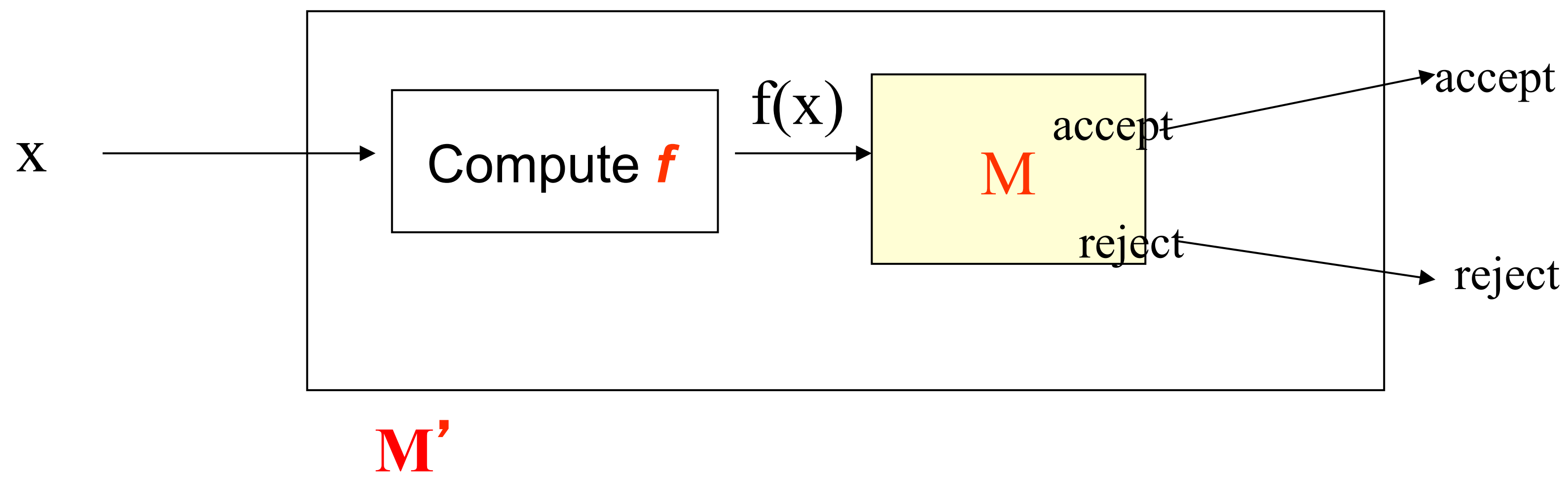
  for any $x \in \Sigma^*$ of length n, f(x) can be computed by a (deterministic) Turing machine in $O(n^c)$ time for some constant c.

Lemma. If $L_1 \leq_p L_2$ and L_2 is in P, then L_1 is in P.

Proof:

- Let M be a polynomial time Turing machine deciding L_2.
- Suppose f is a polynomial time computable function such that for all $x \in \Sigma^*$, $x \in L_1 \Leftrightarrow f(x) \in L_2$.
- Then the following Turing machine **M'** decides L_1

  (i.e., for all $x \in \Sigma^*$, if $x \in L_1$, **M'** accepts; otherwise, **M'** rejects.)



How much time does M' take

Given an input x of length n,
computing f(x) takes $O(n^c)$ time, where c is a constant.

How big is f(x)? Of length $O(n^c)$.

M operates in polynomial time, that means, for any input y, M takes $O(|y|^{c'})$ time, where c' is another constant.

Thus, M on input f(x) takes $O(|f(x)|^{c'})$ time.

$$n^{c\,c'} = |x|^{cc'}$$

In conclusion, M' on any input x of length n uses $O(n^c + n^{cc'})$ time, and $L_1 \in P$.

NP-completeness: definition

A language L is said to be NP-complete if

- L is in NP; and
- for **all** languages L' in NP, L' $\leq_p$ L.

We don't know how to do it at this moment!

Fact. For any NP-complete language L, if we can show that L is in P, then all languages in NP are in P (i.e., NP = P).

In other words, if you know how to solve a NP-complete problem in polynomial time on your PC, you can solve all problems in NP.

Examples of NP-complete problems

- Knapsack problem, partition (subset sum) problem.
- Classic problems: formula satisfiability , clique, vertex cover, travelling salesman problem

The 1st problem known to be NP-complete

- Database problems: minimum cardinality key, conjunctive Boolean query, safety of database transaction systems, consistency of database frequency tables
- Network design: …
- Scheduling: …
- Games, Number theory: ….

Referenecs (Wikipedia);
Garey & Johnson;
Paul E. Dunne.

NP-Completeness

Why are we interested in NP-Completeness?

The past few decades have witnessed many open problems (in different applications) for which no practical algorithms have been devised.

The theory of NP-completeness shows that most of these problems actually fall into the same class (NP-complete).

At present we believe that $P \neq NP$ and therefore NP-complete problems don't have (deterministic) polynomial-time solution.

This gives researchers an "excuse" to stop finding polynomial time algorithm for these problems.

Practical implication

When you realize a problem is too tough to solve, you should try to prove it to be NP-complete.

Formula

Let x_1, x_2 , x_3 , … be Boolean variables. (I.e., each x_i has value equal to **true(1)** or **false(0)**.)

A formula is a Boolean expression composed of Boolean variables and operators (and $\wedge$, or $\vee$, not ~).

E.g., $x_1 \vee x_2$; $\sim((x_1 \vee x_2) \wedge (\sim x_2))$; $x_2 \wedge (\sim x_2)$

Given a formula F, an *assignment* of the values to its variables determines the (truth) value of F.

E.g.,

With respect to the assignment x_1 = **false** , x_2 = **true**,

- $x_1 \vee x_2$ is **true**;
- $\sim((x_1 \vee x_2) \wedge (\sim x_2))$ is **true**;
- $x_2 \wedge (\sim x_2)$ is **false**

With respect to the assignment x_1 = **false** , x_2 = **false**,

- $x_1 \vee x_2$ is **false**;
- $\sim((x_1 \vee x_2) \wedge (\sim x_2))$ is **true**;
- $x_2 \wedge (\sim x_2)$ is **false**

Satisfiability

A formula F is *satisfiable* if there exists an assignment to its Boolean variables such that F becomes **true**.

E.g., $x_1 \vee x_2$ is satisfiable;

$(x_1 \vee x_2) \wedge (\sim x_2)$ is satisfiable;

$x_2 \wedge (\sim x_2)$ is not satisfiable.

The Satisfiability problem (SAT): Given a formula F, determine whether F is satisfiable.

To prove: SAT is NP-complete.

Memory refresh

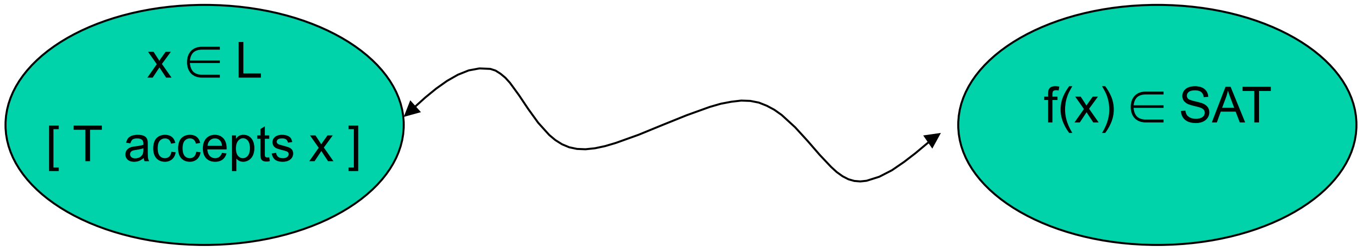
A language L is said to be NP-complete if

- L is in NP; and
- for **all** languages L' in NP, $L' \leq_p L$.

$L_1 \leq_p L_2$

- There exists a function **f** such that
  - for all $x \in \Sigma^*$, $x \in L_1$ **if and only if** **f**$(x) \in L_2$; and
  - **f** is *computable* in polynomial time

Lemma. For any language $L \in NP$, $L \leq_p SAT$.



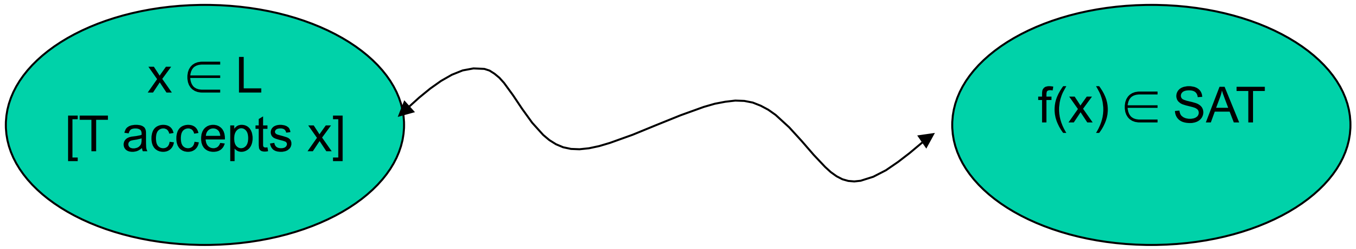
Since $L \in NP$, there is a one-tape NTM T deciding L in $p(n)$ time for some polynomial $p(n) \geq n$.

Notations for T:

- States: $Q = \{q_0, q_1, q_2, \ldots, q_h, q_{accept}, q_{reject}\}$
- Tape alphabet: $\Sigma = \{a_1, a_2, \ldots, a_c\}$
- Transition function: δ

Let $\Pi = Q \cup \Sigma \cup \{\$\}$, where \$ is a new symbol.

Lemma. For any language $L \in NP$, $L \leq_p SAT$.



Since $L \in NP$, there is a one-tape NTM T deciding L in $p(n)$ time for some polynomial $p(n) \geq n$.

Notations for T:

- States: $Q = \{q_0, q_1, q_2, \ldots, q_h, q_{accept}, q_{reject}\}$
- Tape alphabet: $\Sigma = \{a_1, a_2, \ldots, a_c\}$
- Transition function: δ

To remember:
Given T & x, what should be f(x) ?

Let $\Pi = Q \cup \Sigma \cup \{\$\}$, where $\$$ is a new symbol.

Configurations

At any time, the **configuration** of T is characterized by

- the current state;
- the position of the tape head; and
- the content of the tape.

Let u and v be strings in Σ^*.

The string uqv defines the following configuration:

- current state = q
- tape content = uv followed by blanks
- tape head position = at the first symbol of v.

Acceptance of T

Consider any input $x \in L$. Let $n = |x|$.

T accepts x using at most $p(n)$ steps, or equivalently,
T admits a sequence of $\ell \leq p(n) + 1$ configurations
$C_1, C_2, \ldots, C_\ell$ such that

- C_1 contains $q_0 x$,
- C_ℓ contains q_{accept}, and
- $C_i \Rightarrow C_{i+1}$.

More notations: Let $x = x_1 x_2 \cdots x_n$, and $t = p(n)$.

Each C_i contains at most t non-blank symbols on the tape.

A table of configurations
(t+1) rows & (t+3) columns

t + 1

C_1: $ q_0 x_1 x_2 … x_n □ □ □ □ … $

C_2: $ y q_1 x_2 … x_n □ □ □ □ … $

C_3: $ ….. $

.

.

.

C_l $ … q_{accept} … $

.

Repeat the accepting configuration C_ℓ

.

C_{t+1} $ … q_{accept} … $

t + 1

C_i and C_{i+1} are very similar

How similar are two consecutive configurations?

- They differ by at most 3 tape squares.

C_i: .... q d ....

C_{i+1}: .... b q' ....

$\delta(q,d) = \{ \ldots (q',b,R) \ldots \}$

C_i: ....c q d ....

C_{i+1}: ....q' c b..............

$\delta(q,d) = \{ \ldots (q',b,L) \ldots \}$

The **relation** between C_i and C_{i+1} can be represented by a function **w**: $(\Pi = Q \cup \Sigma \cup \{\$\})^6 \rightarrow \{true, false\}$.

A sliding window

C_i: ……………….c a d ……………

C_{i+1}: ……………….e b f……………….

Define a Boolean function **w**(c,a,d,e,b,f) = **true** if

- $a \in Q$, $(f, b, R) \in \delta(a,d)$, and $c = e$; or
- $a \in Q$, $(e, f, L) \in \delta(a,d)$, and $b = c$; or
  - $a \in Q$, $c = e = \$$, $(b, f, L) \in \delta(a,d)$; or
- $a = q_{accept}$, $c = e$, $a = b$, and $d = f$; or
- $a \notin Q$, and $c, d \notin Q$, and $a = b$; or
- $a \notin Q$, and ($c \in Q$ or $d \in Q$)

Note that **w** doesn't depend on the input.

True or False

The number of combinations of (a, b, c, d, e, f) that can make **w**(c,a,d,e,b,f) true is infinite.

The number of combinations of (a, b, c, d, e, f) that can make **w**(c,a,d,e,b,f) true depends on the input x.

A trivial fact

Let $A = a_1a_2\ldots a_{t+3}$ and $B = b_1b_2\ldots b_{t+3}$ be two configurations, where $a_1 = a_{t+3} = b_1 = b_{t+3} = \$$.

Fact.

$A \Rightarrow B$ if and only if

$w(a_1\, a_2\, a_3\, b_1\, b_2\, b_3\,) =$ true; and

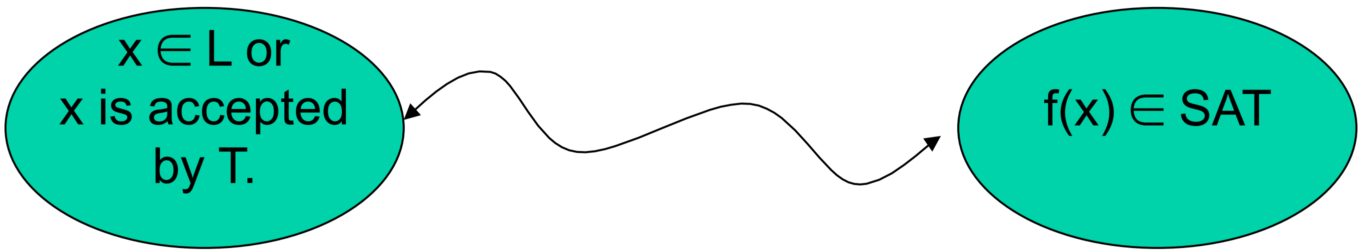
$w(a_2\, a_3\, a_4\, b_2\, b_3\, b_4\,) =$ true; and

$w(a_3\, a_4\, a_5\, b_3\, b_4\, b_5) =$ true; and

…

$w(a_{t+1}\, a_{t+2}\, a_{t+3}\, b_{t+1}\, b_{t+2}\, b_{t+3}) =$ true.

Lemma. For any language $L \in NP$, $L \leq_p SAT$.



$|x| = n$

Goal: Find a reduction function f that transforms an input x to T (that decides L) to a formula such that

an input x is accepted by T in p(n) steps

⇔ there exist p(n) +1 configurations in which every two consecutive configurations satisfy the function w

⇔ there is a formula f(x) that is satisfiable

Boolean Variables

To remember:
Given T, w & x, what should be f(x) ?

For all $1 \leq i \leq p(n)+1$,
$1 \leq j \leq p(n)+3$, and
$a \in \Pi$ (= $Q \cup \Sigma \cup \{\$\}$),
create a Boolean variable $C_{i,j,a}$.

An arbitrary assignment of $C_{i,j,a}$ may not form a sequence of p(n)+1 configurations that accepts x.

f(x) is a formula asserting that

- C_1 specifies the start configuration for x,
- $C_{p(n)+1}$ is an accepting configuration, and
- Every C_i and C_{i+1} satisfy the function w.

Notations

$\vee$ = OR; $\wedge$ = AND

$\vee$ p[i] where $0 < i < n+1$ is meant to be p[1] or p[2] or … or p[n].

$\wedge$ p[i] where $0 < i < n+1$ is meant to be p[1] and p[2] and … and p[n].

Exactly one symbol defined for one position

Let $t = p(n)$. Consider any $1 \le i \le t+1$, $1 \le j \le t+3$. Denote $\Pi = \{a_1, a_2, \ldots. a_d\}$, where d is a constant.

Define $F1_{ij} =$

$C_{i,j,a1} \wedge (\sim C_{i,j,a2} \wedge \sim C_{i,j,a3} \wedge \sim C_{i,j,a4} \wedge \ldots \wedge \sim C_{i,j,ad})$

or

$C_{i,j,a2} \wedge (\sim C_{i,j,a1} \wedge \sim C_{i,j,a3} \wedge \sim C_{i,j,a4} \wedge \ldots \wedge \sim C_{i,j,ad})$

…

or

$C_{i,j,ad} \wedge (\wedge \sim C_{i,j,b}$ where $b \ne a_d)$

Define $F1 = \wedge F1_{i,j}$ where $1 \le i \le t+1$, $1 \le j \le t+3$

Left & right borders are "$"

For all $1 \leq i \leq t+1$,

define $F2_i = C_{i,1,\$}$ and $C_{i,t+3,\$}$

Top row = start configuration with input x

Define F3 =

$C_{1,2,q0} \wedge C_{1,3,x_1} \wedge C_{1,4,x_2} \wedge \ldots \wedge C_{1,n+2,x_n}$

$\wedge C_{1,n+3,U} \ldots \wedge C_{1,t+2,U}$

Last row is an accepting configuration

Define F4 =

C_i, C_{i+1} and w

Consider all i,j such that $1 \le i \le t$, $2 \le j \le t+2$.

Define $F5_{i,j}$ to be

$$\bigvee\left(C_{i,j-1,c} \wedge C_{i,j,a} \wedge C_{i,j+1,d} \wedge C_{i+1,j-1,e} \wedge C_{i+1,j,b} \wedge C_{i+1,j+1,f} \right)$$

where **w**(c,a,d,e,b,f) = true.

C_i: ....c a d..............

C_{i+1}: ....e b f................

j-1 j j+1

Define F5 = $\bigwedge$ $F5_{i,j}$ where $1 \le i \le t$, $2 \le j \le t+2$

What is f(x)

In summary, f(x) is the formula F1 ∧ F2 ∧ F3 ∧ F4 ∧ F5.

Claim: T accepts x if and only if there exists an assignment to Q_{ija} such that f(x) is true.

Claim: The length of f(x) is O($p^2(n)$).

Things (for you) to verify:

- f(x) is computable in O($p^2(n)$) time, and
- x in L (x is accepted by T) if and only if f(x) is satisfiable.

Reading

- Sipser: Chapter 7
- Hopcroft et al.: Chapter 10
- NP-completeness chapter in any algorithms book

Questions

Do nondeterministic TMs decide more languages than deterministic TMs?

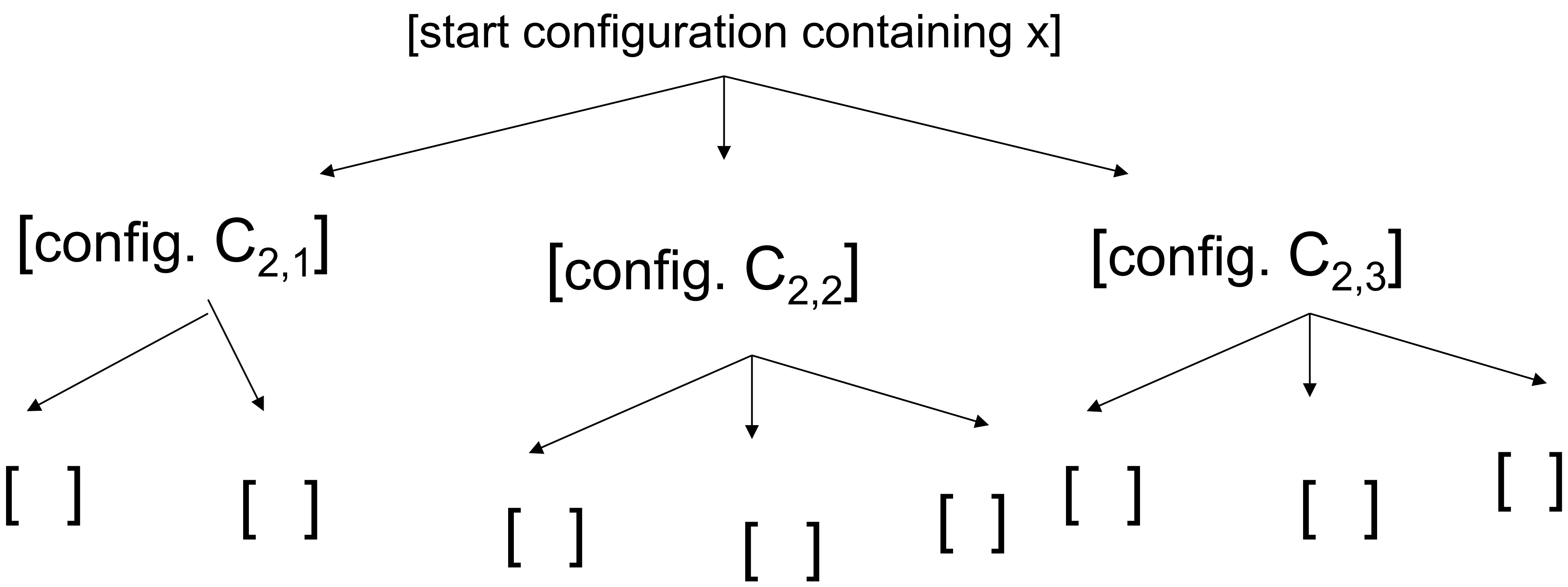
NO.

Theorem. If a language L is decided by an NTM M operating in time $t(n)$, then L can be decided by a (D)TM T operating in time $2^{O(t(n))}$.

Corollary: $\mathrm{NTIME}(t(n)) \subseteq \mathrm{TIME}(2^{O(t(n))})$.

Given any input x (on tape 1), T simulates M as follows. The idea is to perform a breadth first search of M's computation tree.

1. T writes the start configuration C_1 of M on tape 2.
2. T, based on the transition function of M, determines the possible next moves of M and write all the resulting configurations $C_{2,1}$ $C_{2,2}$,…, $C_{2,k}$ on tape 3. Erase tape 2.



Tape 2 | [Start configuration]

Tape 3 | [configuration 1] [configuration 2] [configuration 3]

Simulation of an NTM

3. For each configuration C on tape 3,
- if C accepts, T accepts;
- if C rejects, T ignores C;
- Otherwise, T determines the next moves from C and write the resulting configurations on tape 2.

4. Repeat Step 3 with the role of tapes 2 & 3 exchanged.

Tape 2	[config. 1][config. 2] ….
Tape 3	[config.1] [config. 2] [config. 3]

Time complexity

If M accepts an input x in time $t(n)$, the computation tree of M(x) contains an accepting configuration which is within $t(n)$ steps from the start configuration.

The computation tree contains $\leq d^{t(n)+1}$ configurations, where d is a constant (the max number of branches in a move).

Each configuration of M has length $O(t(n))$.

T takes $O(t(n))$ time to construct a "next" configuration.

Therefore, T takes at most $O(t(n)\ d^{t(n)+1}) = 2^{O(t(n))}$ time to find the accepting configuration.