Formal Languages, Automata and Computation

Administrative Stuff

- Textbook: *Introduction to the Theory of Computation, 3rd edition* by Michael Sipser (MIT)
- Evaluation:
  - 2 Midterm Exams
  - 1 Final Exam
  - 8 Homeworks
- See syllabus for details.

What is this course about? – Formal Languages

- An abstraction of the notion of a “problem”
- Problems are cast **either** as Languages (= sets of “Strings”)
  - ”Solutions” determine if a given “string” is in the set or not
    - e.g., Is a given integer, *n*, prime?
- **Or**, as transductions between languages
  - “Solutions” transduce/transform the input string to an output string
    - e.g., What is 3+5?

What is this course about? – Formal Languages

- So essentially all computational processes can be reduced to one of
  - Determining membership in a set (of strings)
  - Mapping between sets (of strings)
- We will formalize the concept of mechanical computation by
  - giving a precise definition of the term “algorithm”
  - characterizing problems that are or are not suitable for mechanical computation.

What is this course about? – Automata

- Automata (singular *Automaton*) are abstract mathematical devices that can
  - Determine membership in a set of strings
  - Transduce strings from one set to another
- They have all the aspects of a computer
  - input and output
  - memory
  - ability to make decisions
  - transform input to output
- Memory is crucial:
  - Finite Memory
  - Infinite Memory
    - Limited Access
    - Unlimited Access

What is this course about?– Automata

- We have different types of automata for different classes of languages.
- They differ in
  - the amount of memory then have (finite vs infinite)
  - what kind of access to the memory they allow.
- Automata can behave **non-deterministically**
  - A non-deterministic automaton can at any point, among possible next steps, pick one step and proceed
  - This gives the conceptual illusion of (infinitely) parallel computation for some classes of automata
    - All branches of a computation proceed in parallel (sort of)
  - More on this later

What is this course about?– Complexity

- How much resource does a computation consume?
  - Time and Space
- What are the implications of nondeterminism for complexity?
- How can we classify problems into classes based on their resource use?
  - Are there problems with very unreasonable resource usage (Intractable problems)?
  - How can we characterize such problems?
    - P vs. NP, PSPACE, Log Space

What is this course about?– Computability

- What is computational power?
  - Automaton 1 tells Automaton 2
    "Tell me what kinds of problems you can solve and I will tell you how powerful you are? "
- What does computational power depend on? (it turns out, not "speed")
- What does it mean for a problem to be computable ?
- Are there any uncomputable functions or unsolvable problems?
  - What does this mean?
  - Why do we care?

Applications/Relevance

- Pattern matching
  - Perl Hacking
  - Bioinformatics
  - Lexical analysis
- Design and Verification
  - Hardware
  - Software
  - Communication Protocols
- Parsing Languages
  - Compiler construction
  - XML Analysis
  - Natural language processing, Machine Translation
- Algorithm design and analysis

Decision Problems

- A decision problem is a function with a YES/NO output
- We need to specify
  - the set A of possible inputs (usually A is infinite)
  - the subset $B \subseteq A$ of YES instances (usually B is also infinite)
- The subset B should have a finite description!

Decision Problems – Examples

- A: integers
  - is_even?(x)
  - is_prime?(x)
- A: integers $\times$ integers
  - is_relatively_prime?(x,y)

Decision Problems – Examples

- A: set of all pairs (G, t)
  - G is a {finite set of triples of the sort (i, j, w)},
  - i and j are integers and w is real
  - The finite set encodes the edges of a weighted directed graph G.
  - $A = \{\ldots(\{\ldots, (3, 4, 5.6), \ldots\}, 8.0), \ldots\}$
- Each pair in A, (G, t), represents a graph G and a threshold t
- Does G have a path that goes through all nodes once with total weight $< t$?
  - Travelling Salesperson Problem
- A is the set of all TSP instances.

Encoding Sets

- Sets can be
  - Finite
  - Infinite
    - Countably Infinite: can be put in one-to-one correspondence with natural numbers (e.g., rational numbers, integers)
    - Uncountably Infinite: can NOT be put in one-to-one correspondence with natural numbers (e.g., real numbers)

Encoding Sets

- In real life, we use many different types of data: integers, reals, vectors, complex numbers, graphs, programs (your program is somebody else's data).
- These can all be encoded as strings
- So we will have only one data type: strings

Strings

- An alphabet is any finite set of distinct symbols
  - $\{0, 1\}$, $\{0,1,2,\ldots,9\}$, $\{a,b,c\}$
  - We denote a generic alphabet by Σ
- A string is any finite-length sequence of elements of Σ.
- e.g., if $\Sigma = \{a, b\}$ then *a*, *aba*, *aaaa*, ..., *abababbaab* are some strings over the alphabet Σ

Strings

- The length of a string ω is the number of symbols in ω. We denote it by $|\omega|$. $|aba| = 3$.
- The symbol ϵ denotes a special string called the empty string
  - ϵ has length 0
- String concatenation
  - If $\omega = a_1, \ldots, a_n$ and $\nu = b_1, \ldots, b_m$ then $\omega \cdot \nu$ (or $\omega\nu$) $= a_1, \ldots, a_n b_1, \ldots, b_m$
  - Concatenation is associative with ϵ as the identity element.
- If $a \in \Sigma$, we use a^n to denote a string of n a's concatenated
  - $\Sigma = \{0, 1\}, 0^5 = 00000$
  - $a^0 =_{def} \epsilon$
  - $a^{n+1} =_{def} a^n a$

Strings

- The reverse of a string ω is denoted by ω^R.
  - $\omega^R = a_n, \ldots, a_1$
- A substring y of a string ω is a string such that $\omega = xyz$ with $|x|, |y|, |z| \geq 0$ and $|x| + |y| + |z| = |\omega|$
- If $\omega = xy$ with $|x|, |y| \geq 0$ and $|x| + |y| = |\omega|$, then x is prefix of ω and y is a suffix of ω.
  - For $\omega = abaab$,
    - ϵ, a, aba, and $abaab$ are some prefixes
    - ϵ, $abaab$, aab, and $baab$ are some suffixes.

Strings

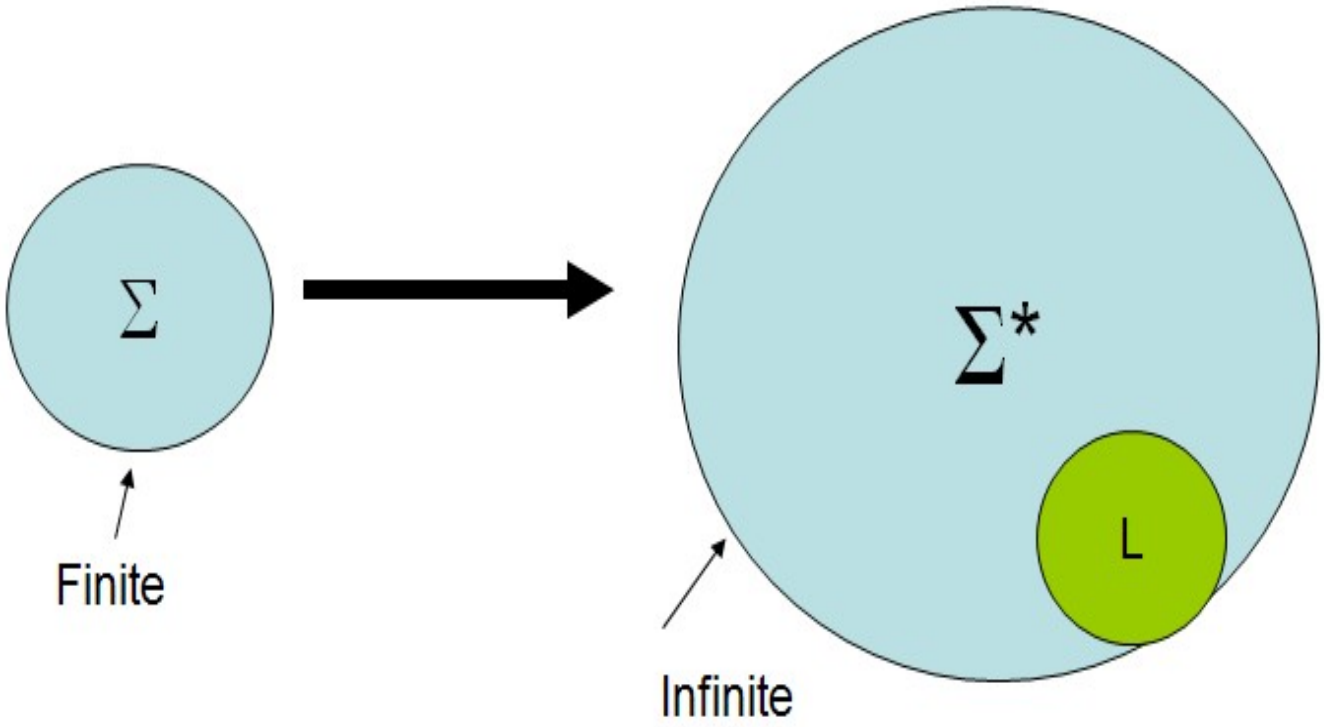
- The set of all possible strings over Σ is denoted by Σ^*.
- We define $\Sigma^0 = \{\epsilon\}$ and $\Sigma^n = \Sigma^{n-1} \cdot \Sigma$
  - with some abuse of the concatenation notation applying to sets of strings now
- So $\Sigma^n = \{\omega | \omega = xy \text{ and } x \in \Sigma^{n-1} \text{ and } y \in \Sigma\}$
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots \Sigma^n \cup \ldots = \bigcup_0^\infty \Sigma^i$
  - Alternatively, $\Sigma^* = \{x_1, \ldots, x_n | n \geq 0 \text{ and } x_i \in \Sigma \text{ for all } i\}$
- Φ denotes the empty set of strings $\Phi = \{\}$,
  - but $\Phi^* = \{\epsilon\}$

Strings

- Σ^* is a countably infinite set of finite length strings
- If x is a string, we write x^n for the string obtained by concatenating n copies of x.
  - $(aab)^3 = aabaabaab$
  - $(aab)^0 = \epsilon$

Languages

- A language L over Σ is any subset of Σ^*



- L can be finite or (countably) infinite

Some Languages

- $L = \Sigma^*$ – The mother of all languages!
- $L = \{a, ab, aab\}$ – A fine finite language.
  - Description by enumeration
- $L = \{a^n b^n : n \geq 0\} = \{\epsilon,\ \text{ab},\ \text{aabb},\ \text{aaabbb}, \ldots\}$
- $L = \{\omega | n_a(\omega) \text{ is even}\}$
  - $n_x(\omega)$ denotes the number of occurrences of x in ω
  - all strings with even number of a's.
- $L = \{\omega | \omega = \omega^R\}$
  - All strings which are the same as their reverses – palindromes.
- $L = \{\omega | \omega = xx\}$
  - All strings formed by duplicating some string once.
- $L = \{\omega | \omega$ is a syntactically correct Java program $\}$

Languages

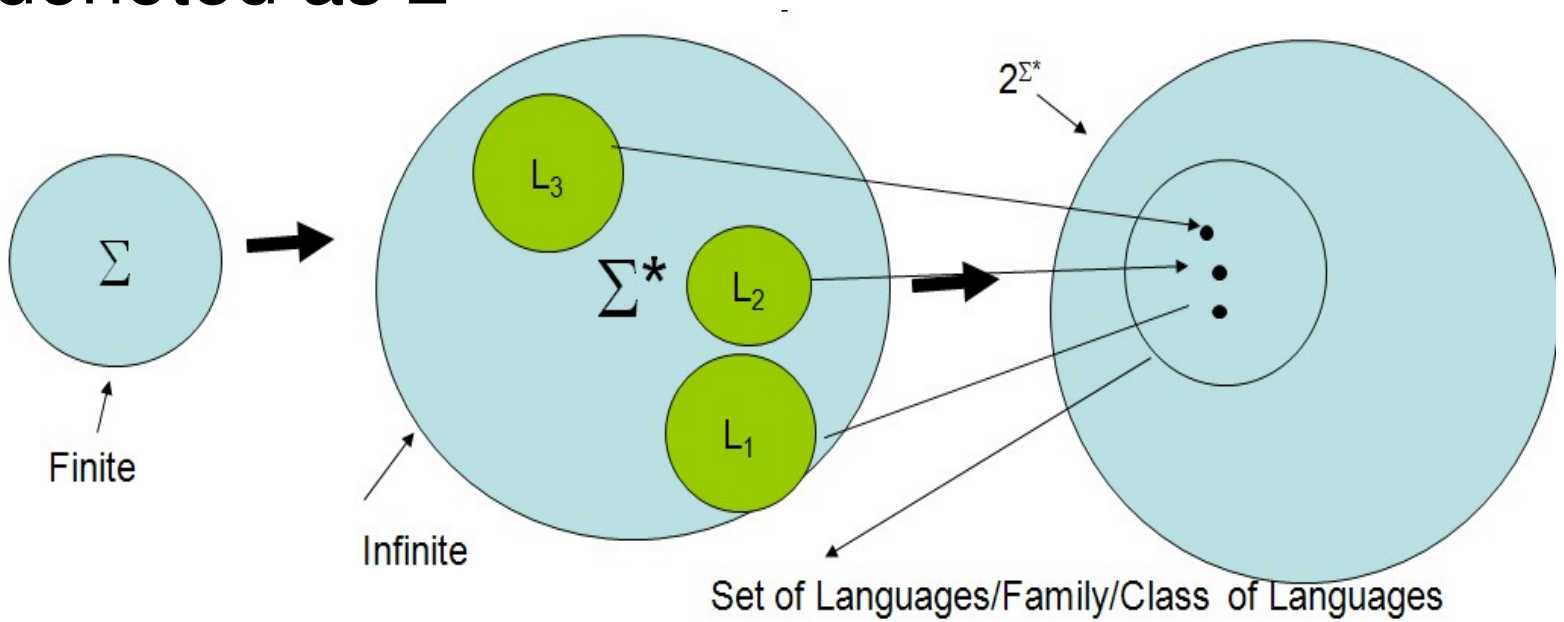
- Since languages are sets, all usual set operations such as intersection and union, etc. are defined.
- Complementation is defined with respect to the universe Σ^* : $\overline{L} = \Sigma^* - L$

Languages

- If L, L_1 and L_2 are languages:
  - $L_1 \cdot L_2 = \{xy | x \in L_1 \text{ and } y \in L_2\}$
  - $L^0 = \{\epsilon\}$ and $L^n = L^{n-1} \cdot L$
  - $L^* = \bigcup_0^\infty L^i$
  - $L^+ = \bigcup_1^\infty L^i = L^* - \{\epsilon\}$

Sets of Languages

- The power set of Σ^*, the set of all its subsets, is denoted as 2^{Σ^*}



Describing Languages

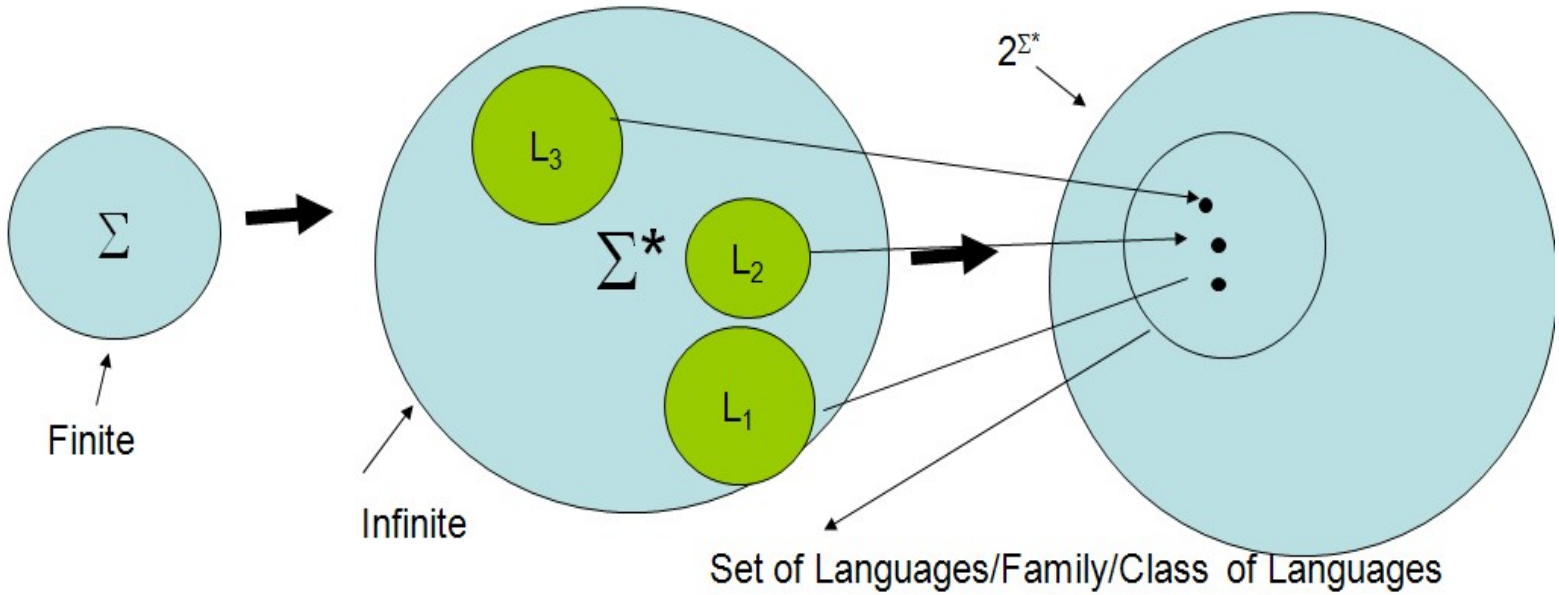
- Interesting languages are infinite
- We need finite descriptions of infinite sets
  - $L = \{a^n b^n : n \geq 0\}$ is fine but not terribly useful!
- We need to be able to use these descriptions in mechanizable procedures

Formal Languages, Automata and Computation

Finite State Machines

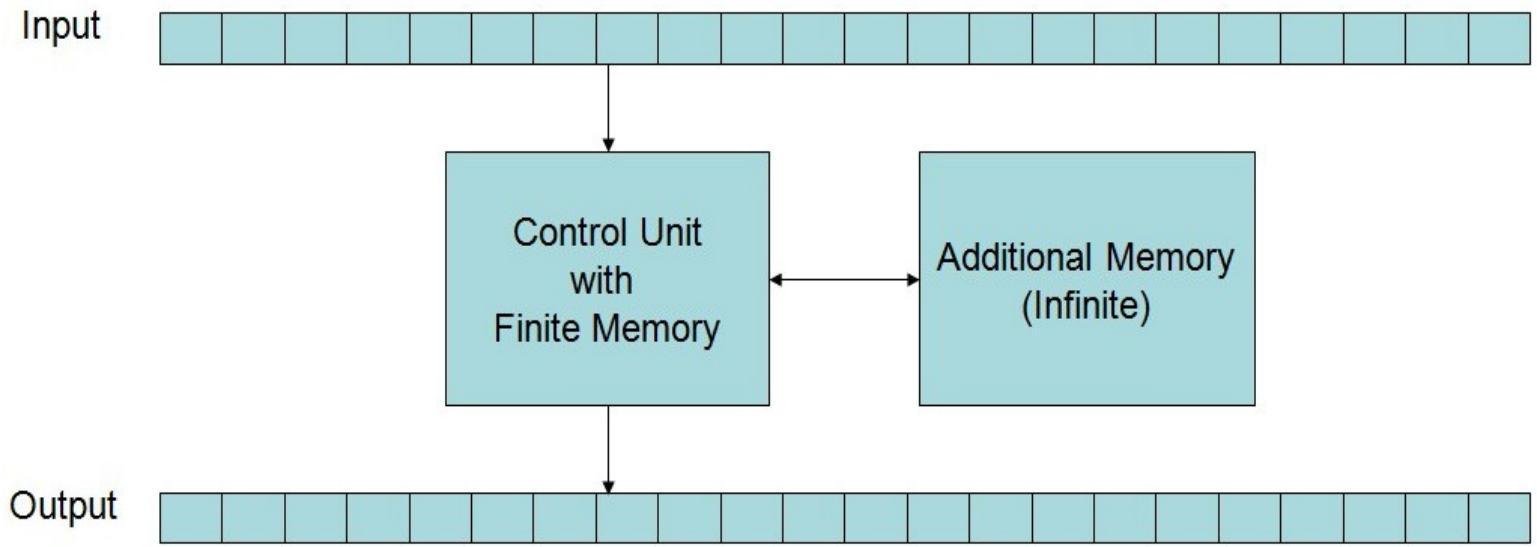
SUMMARY

- Alphabet Σ,
- Set of all Strings, Σ^*,
- Language $L \subseteq \Sigma^*$,
- Set of all languages 2^{Σ^*}



AUTOMATA

- Abstract Models of computing devices



- Each step of operation is like:
  - If the current input symbol is X and memory state is Z, then output Y, move (left/right)

AUTOMATA

- The control unit has some finite memory and it keeps track of what step to execute next.
- Additional memory (if any) is infinite - we never run out of memory!
  - Infinite but like a stack - only the top item is accessible at a given time.
  - Infinite but like a tape, any cell is (sequentially) accessible.

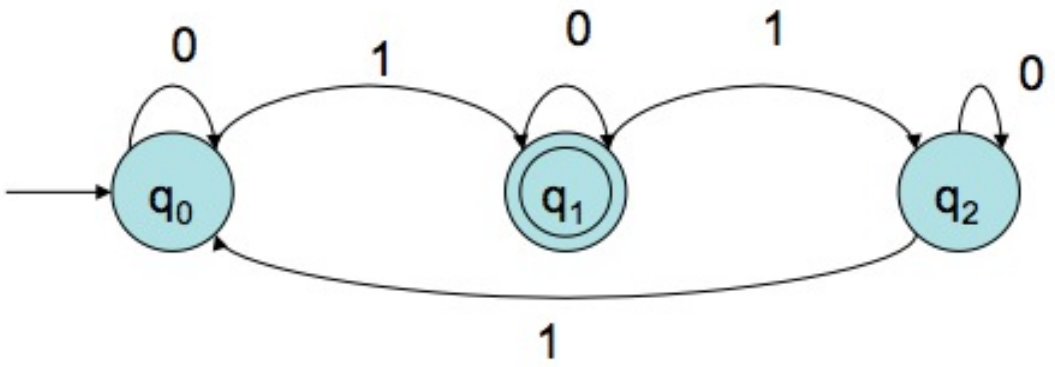
Finite State Automata

- Finite State Automata (FSA) are the simplest automata.
- Only the finite memory in the control unit is available.
- The memory can be in one of a finite number states at a given time – hence the name.
  - One can remember only a (fixed) finite number of properties of the past input.
  - Since input strings can be of arbitrary length, it is not possible to remember unbounded portions of the input string.
- It comes in Deterministic and Nondeterministic flavors.

Deterministic Finite State Automata (DFA)

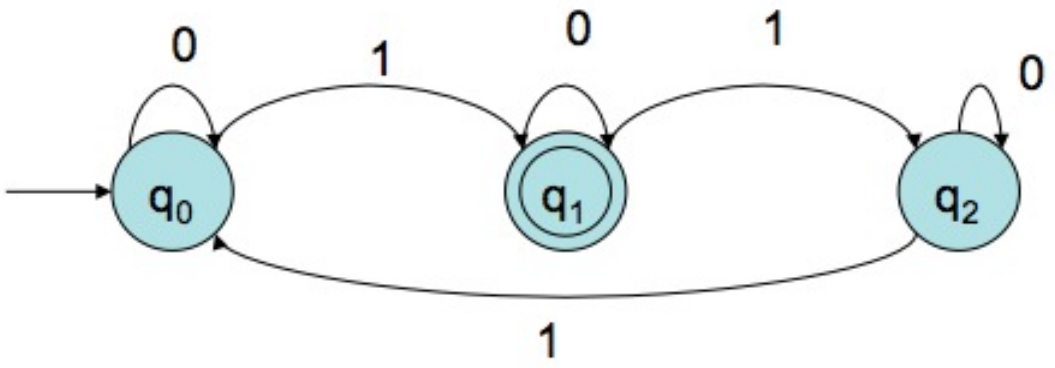
- A DFA starts in a start state and is presented with an input string.
- It moves from state to state, reading the input string one symbol at a time.
- What state the DFA moves next depends on
  - the current state,
  - current input symbol
- When the last input symbol is read, the DFA decides whether it should accept the input string

A simple DFA Example



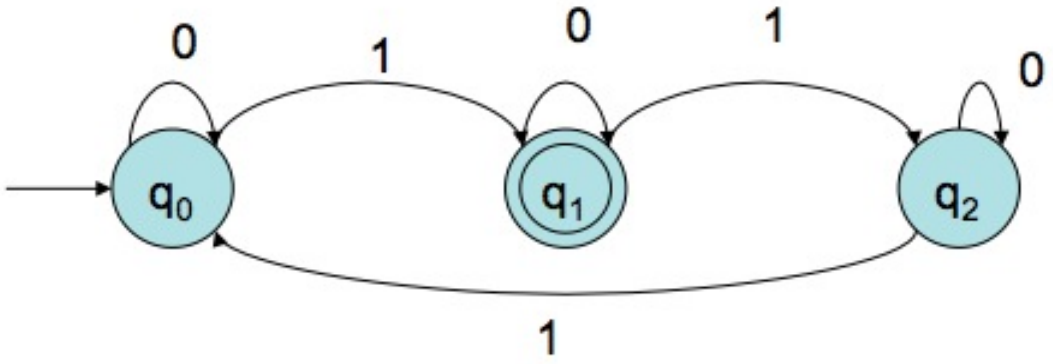
- States are shown with circles. We usually have labels on the states.
  - One designated state is the start state, (State q_0 here).
  - States with double circles denote the accepting or final states (State q_1 here)
- Directed and labeled arrows between states denote state transitions.

A SIMPLE DFA EXAMPLE



- This DFA stays in the same state when the next input symbol is a 0.
- In state q_0, an input of 1 moves the DFA to state q_1.
- In state q_1, an input of 1 moves the DFA to state q_2.
- In state q_2, an input of 1 moves the DFA back to state q_0.
- If the DFA is in state q_1 when the input is finished, the DFA accepts the input string.

A SIMPLE DFA EXAMPLE

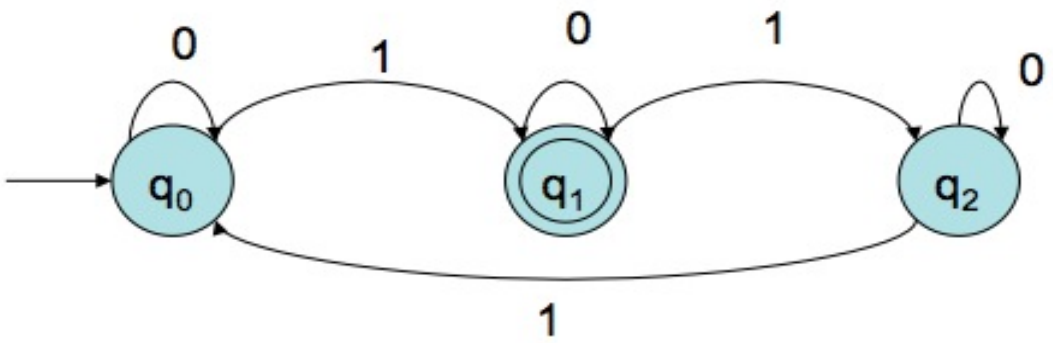


- What kinds of strings does this DFA accept?
  - It accepts $\omega = 00010000$
  - It accepts $\omega = 00010011001$
  - It accepts $\omega = 1$
  - It rejects $\omega = 1100001$
  - It rejects $\omega = 0110000$
- It accepts all strings $\omega \in \{0,1\}^*$ such that $n_1(\omega) = 1 \mod 3$

DFA – Formal Definition

- A Deterministic Finite State Acceptor (DFA) is defined as the 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where
  - Q is a finite set of states
  - Σ is a finite set of symbols – the alphabet
  - $\delta : Q \times \Sigma \to Q$ is the next-state function
  - $q_0 \in Q$ is the (label of the) start state
  - $F \subseteq Q$ is the set of final (accepting) states

Formal Description of the Example DFA



- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- δ :

δ	0	1
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_0

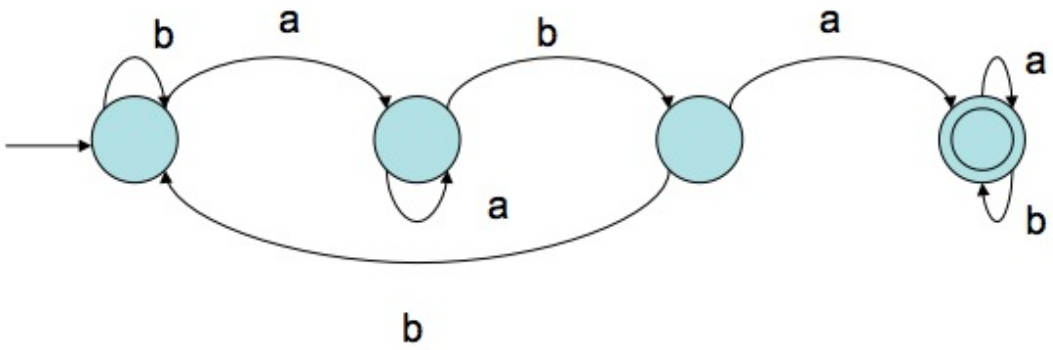
- q_0
- $F = \{q_1\}$

We will almost always use the graphical description for δ. The other components will always be implicit!

How the DFA Works

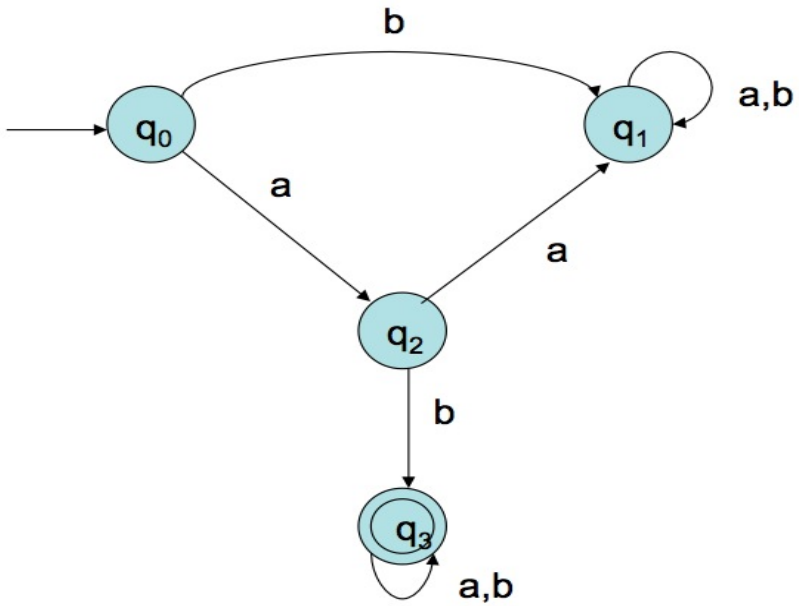
- The DFA accepts a string $\omega = x_1 x_2 \cdots x_n$ if a sequence of states $r_0 r_1 r_2 \cdots r_n$, $r_i \in Q$, exists, such that
  1. $r_0 = q_0$ (Start in the initial state)
  2. $r_i = \delta(r_{i-1}, x_i)$ for $i = 1, 2, \ldots n$
     - Move from state to state.
  3. $r_n \in F$
     - End up in a final state.
- If the DFA is NOT in an accepting state when the input string is exhausted, then the string is rejected.

DFA Example



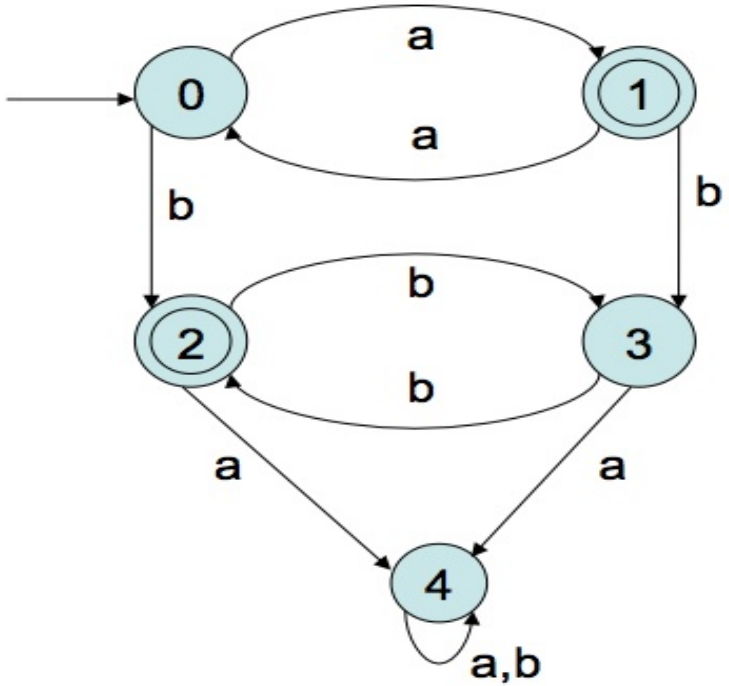
- This DFA accepts strings that have *aba* somewhere in it.
- Once the existence of *aba* is ascertained, the rest of the input is ignored!
- What do the states “remember”?

DFA Example



- This DFA accepts strings that start with *ab*
- Once the string starts with *ab* the rest is ignored!
- The state q_1 is known as a sink state.
  - Once a machine enters a sink state, there is no getting out! It is rejected.

DFA Example



- This DFA accepts strings of the sort $a^n b^m$ such that $n + m$ is odd.

A MORE INTERESTING DFA EXAMPLE

- Input is a string over $\Sigma = \{0, 1\}$
- We interpret the string as a binary number.
- We want to accept strings where the corresponding binary number is divisible by 3.
  - Accept e.g., 0, 11, 1001, 1100, 1111, 111100, ...
  - Reject e.g., 1, 10, 101, 10000, ...
- The most significant (leftmost) digit comes first!
- No obvious pattern at first sight!

A MORE INTERESTING DFA EXAMPLE

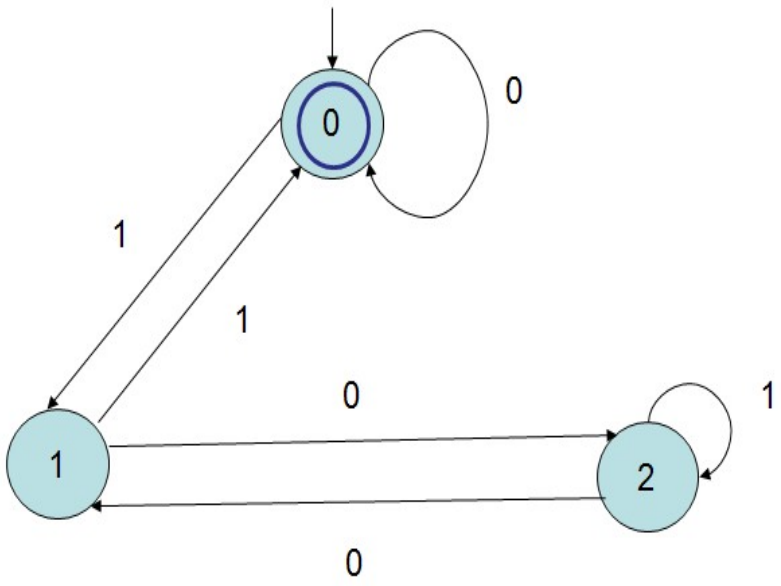
- How do we find the decimal equivalent of binary number digit-by-digit?
  1. `value=0`
  2. `repeat as long as there is another binary-digit`
     - `value=value*2+binary-digit`

- $1101_2 \rightarrow 0 \cdot 2 + 1 = 1 \rightarrow 1 \cdot 2 + 1 = 3 \rightarrow 3 \cdot 2 + 0 = 6 \rightarrow 6 \cdot 2 + 1 = 13_{10}$
- We can not compute this number with a DFA, since the number can be arbitrarily large!
- However, for our problem, we can compute a running modulo 3 with a DFA!!

Computing a Running Modulo 3 Remainder

- Consider any number $n = 3p + r$. It has remainder r when divided by 3
- Multiply by 2 and add 0
  - $r = 0 : 2n + 0 = 2(3p + 0) + 0 = 3(2p) + 0 \rightarrow$ New r is 0.
  - $r = 1 : 2n + 0 = 2(3p + 1) + 0 = 3(2p) + 2 \rightarrow$ New r is 2.
  - $r = 2 : 2n + 0 = 2(3p + 2) + 0 = 3(2p + 1) + 1 \rightarrow$ New r is 1.
- Multiply by 2 and add 1
  - $r = 0 : 2n + 1 = 2(3p + 0) + 1 = 3(2p) + 1 \rightarrow$ New r is 1.
  - $r = 1 : 2n + 1 = 2(3p + 1) + 1 = 3(2p + 1) + 0 \rightarrow$ New r is 0.
  - $r = 2 : 2n + 1 = 2(3p + 2) + 1 = 3(2p + 1) + 2 \rightarrow$ New r is 2.
- This information now defines the state transition function
  - We let each state denote the remainder. So δ maps each remainder and input digit combination, to a new remainder.

A DFA FOR BINARY NUMBERS DIVISIBLE BY 3



- Running some examples:
  - For $11_2 = 3_{10} \Rightarrow 0 \rightarrow 1 \rightarrow 0 \Rightarrow$ Accept
  - For $1100_2 = 12_{10} \Rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 1 \Rightarrow 0$ Accept
  - For $1111_2 = 15_{10} \Rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \Rightarrow 0$ Accept
  - For $1010_2 = 10_{10} \Rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \Rightarrow 2$ Reject

The Extended State Transition Function

- $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function. The input is a symbol.
- $\delta^* : Q \times \Sigma^* \rightarrow Q$ is the extended state transition function.
  - $\delta^*(q, \epsilon) = q$
  - $\delta^*(q, \omega \cdot a) = \delta(\delta^*(q, \omega), a)$, where $a \in \Sigma$ and $\omega \in \Sigma^*$
    - First, go (sort of recursively) where ω (a string) takes you, $(\delta^*(q, \omega) = q')$
    - Then, make a single transition with symbol a $(\delta(q', a))$

The Language Accepted by a DFA

- $L(M)$ denotes the language accepted by a DFA M

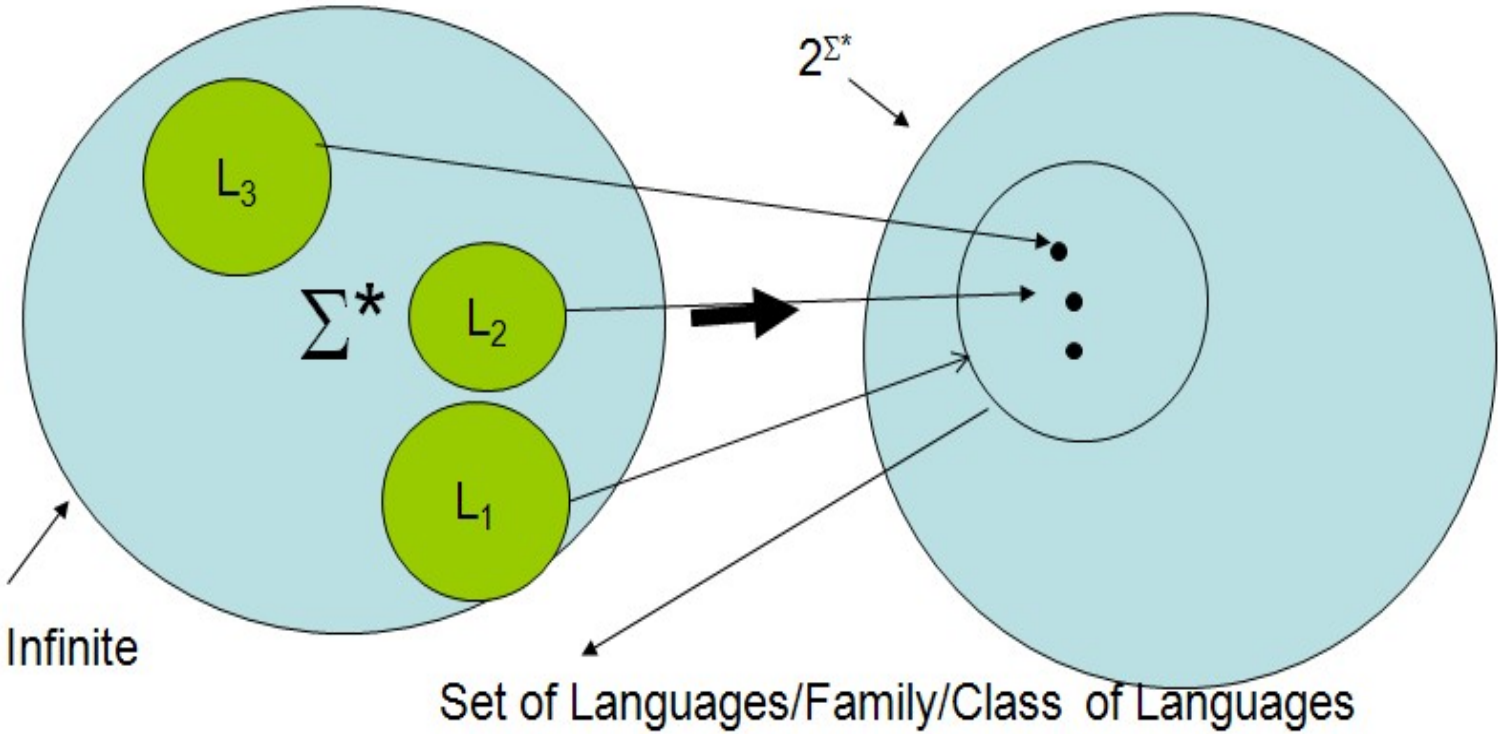
$$L(M) = \{\omega | \omega \in \Sigma^* \text{ and } \delta^*(q_0, \omega) \in F\}$$

- Similarly

$$\overline{L(M)} = \{\omega | \omega \in \Sigma^* \text{ and } \delta^*(q_0, \omega) \notin F\}$$

Regular Languages

A language L is called a regular language if and only if there exists a DFA M such that $L(M) = L$.



Sample Problems

- Design a DFA for all strings over the alphabet $\Sigma = \{a, b\}$ that contain *aba* but not *abaa* as a substring.[1]

[1] A substring is any consecutive sequence of symbols that occurs anywhere in a string. For example, *ab* and *bc* are substrings in *abc* while *cb* or *ac* are not.

SAMPLE PROBLEMS

- Design a DFA for the language
  $L = \{w \mid w \text{ contains at least one } 0 \text{ and at most one } 1\}$

SAMPLE PROBLEMS

- Design a DFA for the language
  $L = \{w \mid w \text{ does not contain } 100 \text{ as a substring}\}$

Sample Problems

- Design a DFA for all strings over the alphabet $A = \{a, b, c\}$ in which no two consecutive positions are the same symbol. (5 states should be sufficient)

Sample Problems

- Design a DFA for all strings over the alphabet $\{0, 1\}$ where the 3^{rd} symbol from the end is a 0.

SAMPLE PROBLEMS

- Design a DFA all strings over the alphabet $\{0, 1\}$ where the leftmost and the rightmost symbols are different.

Sample Problems

- Design a DFA all strings over the alphabet $\{a, b, c\}$ where only two of the symbols occur odd number of times.

Sample Problems

- Design a DFA all strings over the alphabet $\{a, b\}$ in which every substring of length four has at least two *b*'s.
- For example, *abbababbbbaabbabba* is accepted, while *abbaaabbbb* is not, because the substring *aaab* does not contain two *b*'s. (At most 8 states should suffice.)

Sample Problems

- Design a DFA all strings over $\{a, b\}$ in which every pair of adjacent 0's appears before any pair of adjacent 1's.

Formal Languages, Automata and Computation

Regular Languages

Nondeterministic Finite State Automata

SUMMARY

- Symbols, Alphabet, Strings, Σ^*, Languages, 2^{Σ^*}
- Deterministic Finite State Automata
  - States, Labels, Start State, Final States, Transitions
  - Extended State Transition Function
  - DFAs accept regular languages

Regular Languages

- Since regular languages are sets, we can combine them with the usual set operations
  - Union
  - Intersection
  - Difference

Theorem

If L_1 and L_2 are regular languages, so are $L_1 \cup L_2$, $L_1 \cap L_2$ and $L_1 - L_2$.

Proof Idea

Construct cross-product DFAs

Cross-product DFAs

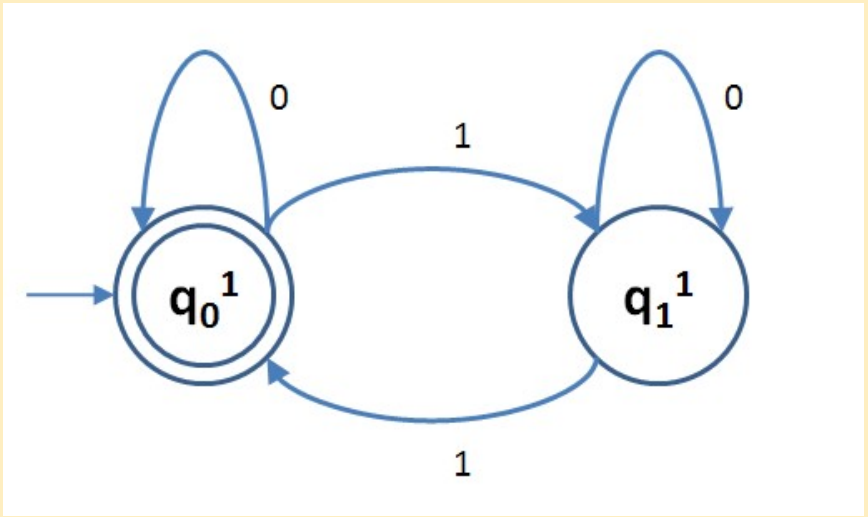
- A single DFA which simulates operation of two DFAs in parallel!
- Let the two DFAs be M_1 and M_2 accepting regular languages L_1 and L_2
  1. $M_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$
  2. $M_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$
- We want to construct DFAs $M = (Q, \Sigma, \delta, q_0, F)$ that recognize
  - $L_1 \cup L_2$
  - $L_1 \cap L_2$
  - $L_1 - L_2$

Constructing the Cross-Product DFA M

- We need to construct $M = (Q, \Sigma, \delta, q_0, F)$
- Q =pairs of states, one from M_1 and one from M_2
  $Q = \{(q_1, q_2) | q_1 \in Q_1 \text{ and } q_2 \in Q_2\}$
  $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$
- $\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$
- Union: $F = \{(q_1, q_2) | q_1 \in F_1 \text{ or } q_2 \in F_2\}$
- Intersection: $F = \{(q_1, q_2) | q_1 \in F_1 \text{ and } q_2 \in F_2\}$
- Difference: $F = \{(q_1, q_2) | q_1 \in F_1 \text{ and } q_2 \notin F_2\}$

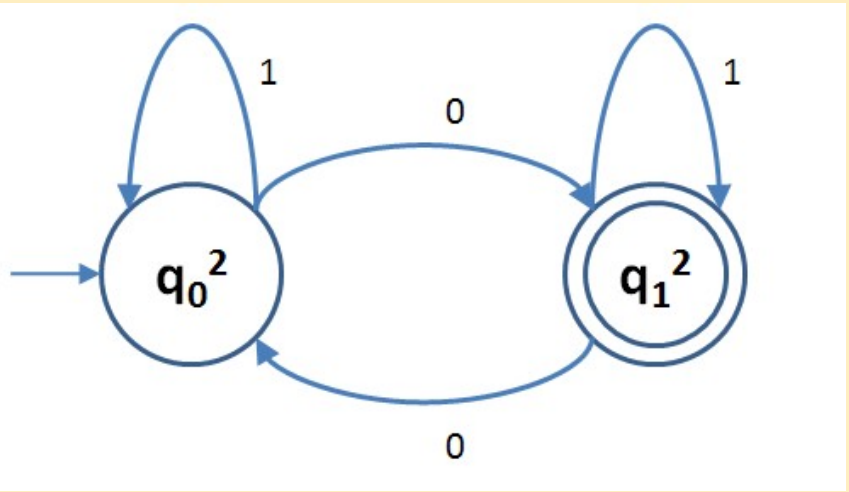
Cross-product DFA Example

Strings with even number of 1s



M_1

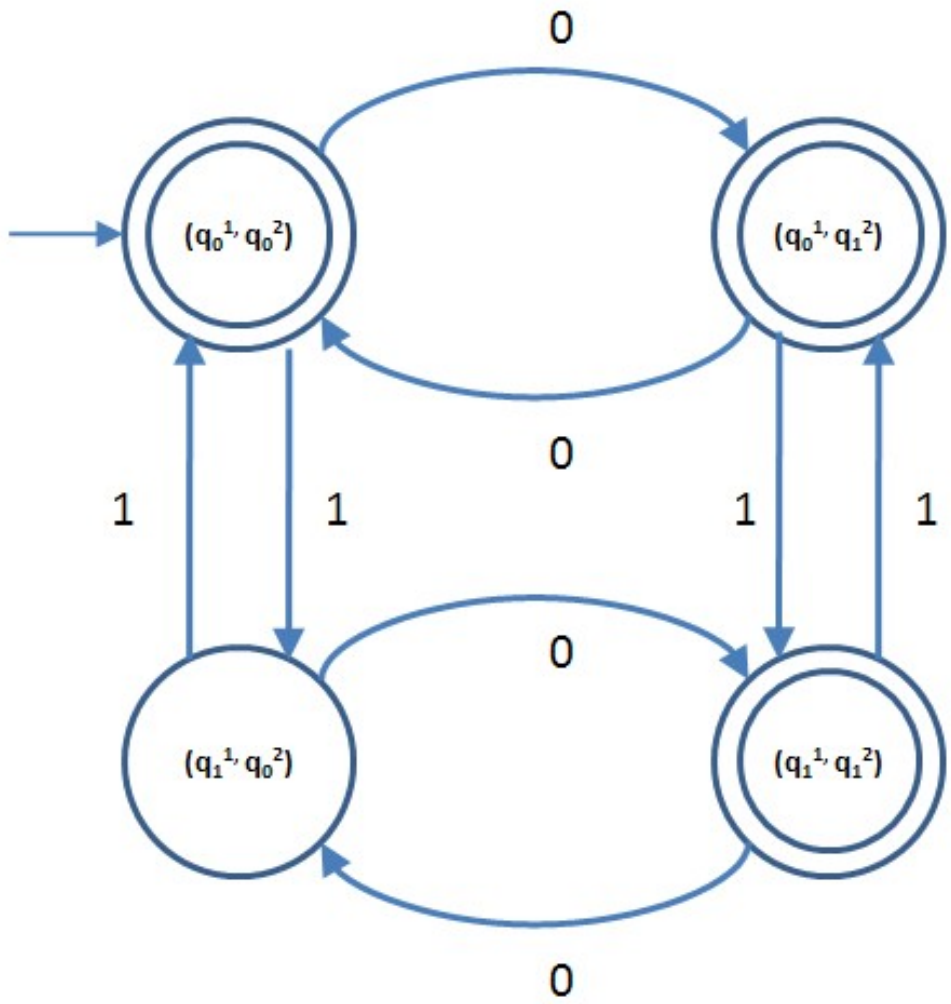
Strings with odd number of 0s



M_2

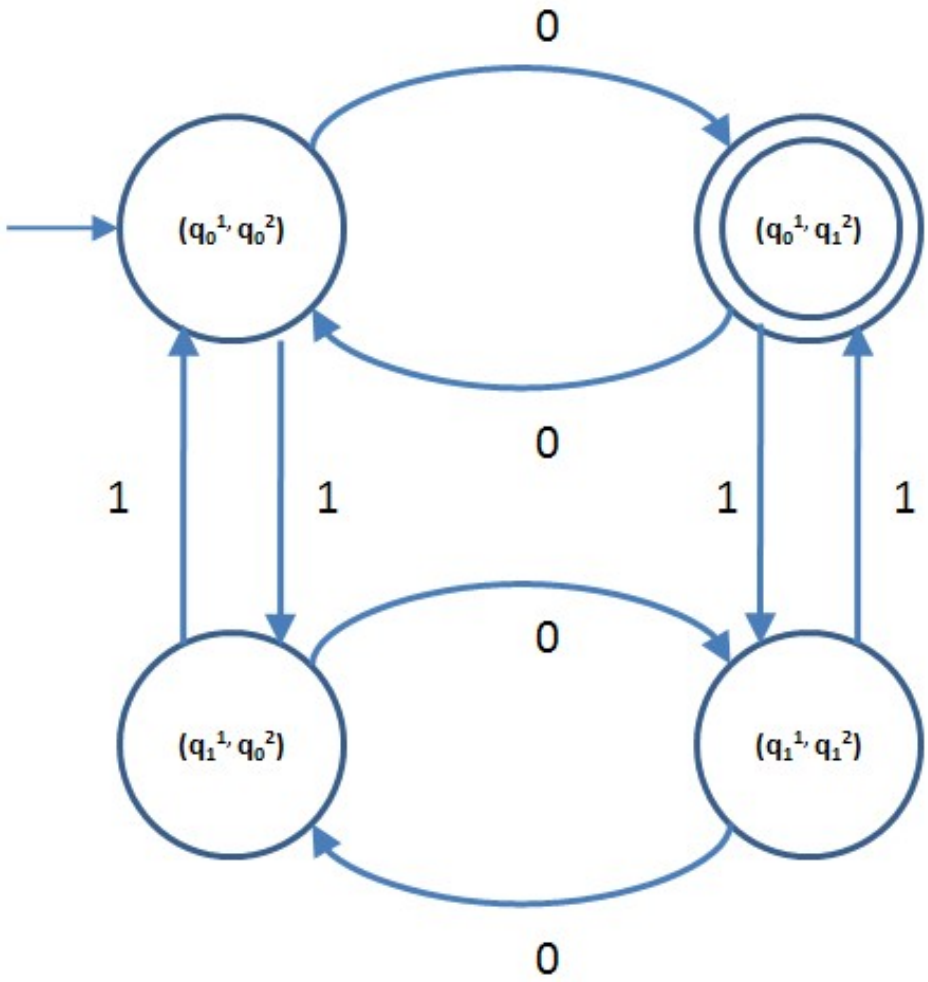
DFA FOR $L_1 \cup L_2$

- DFA for $L_1 \cup L_2$ accepts when either M_1 or M_2 accepts.



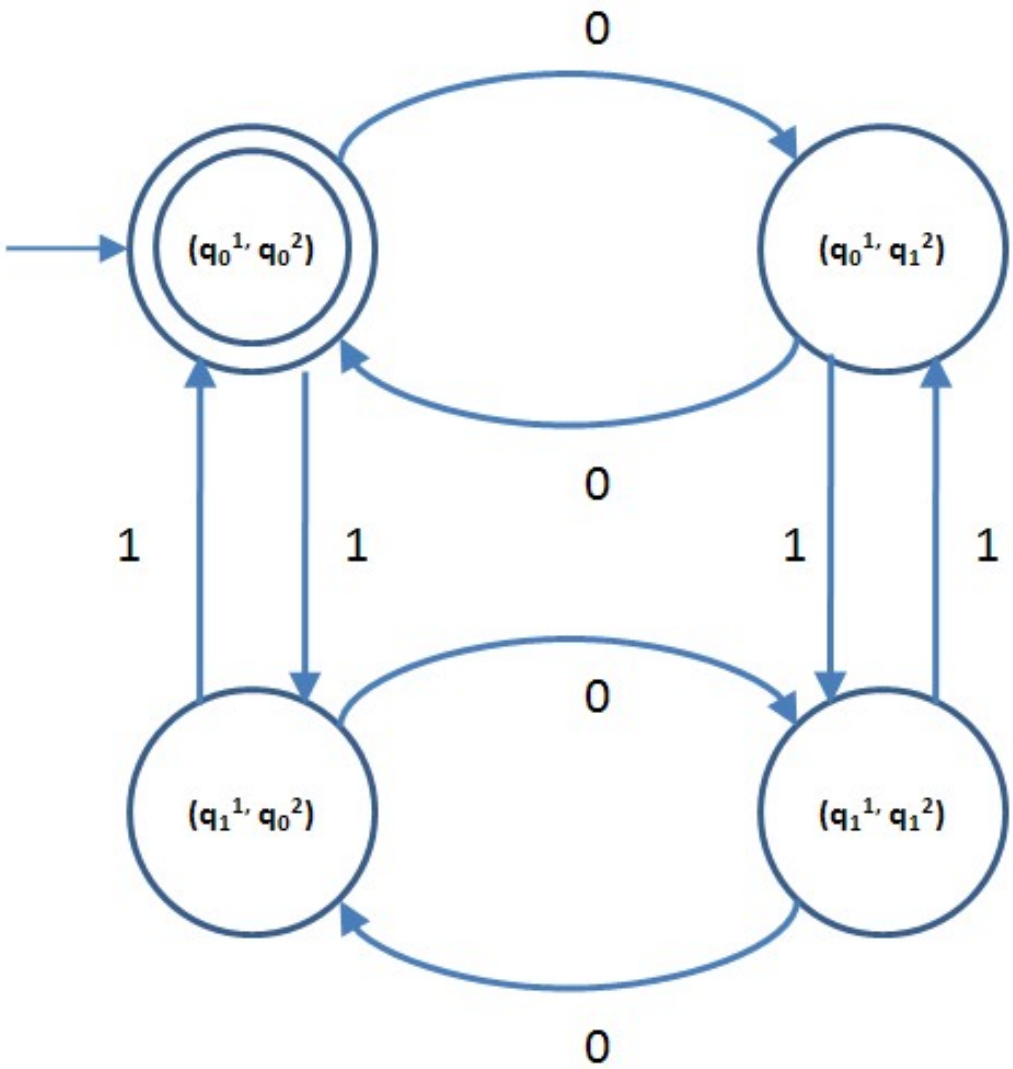
DFA FOR $L_1 \cap L_2$

- DFA for $L_1 \cap L_2$ accepts when both M_1 and M_2 accept.



DFA FOR $L_1 - L_2$

- DFA for $L_1 - L_2$ accepts when M_1 accepts and M_2 rejects.



Another example: Find the cross-product DFA for

- DFA for binary numbers divisible by 3
- DFA for binary numbers divisible by 2

Other Regular Operations

- Reverse: $L^R = \{\omega = a_1 \dots a_n | \omega^R = a_n \dots a_1 \in L\}$
- Concatenation: $L_1 \cdot L_2 = \{\omega\nu | \omega \in L_1 \text{ and } \nu \in L_2\}$
- Star Closure: $L^* = \{\omega_1\omega_2 \dots \omega_k | k \geq 0 \text{ and } \omega_i \in L\}$

The Reverse of a Regular Language

Theorem

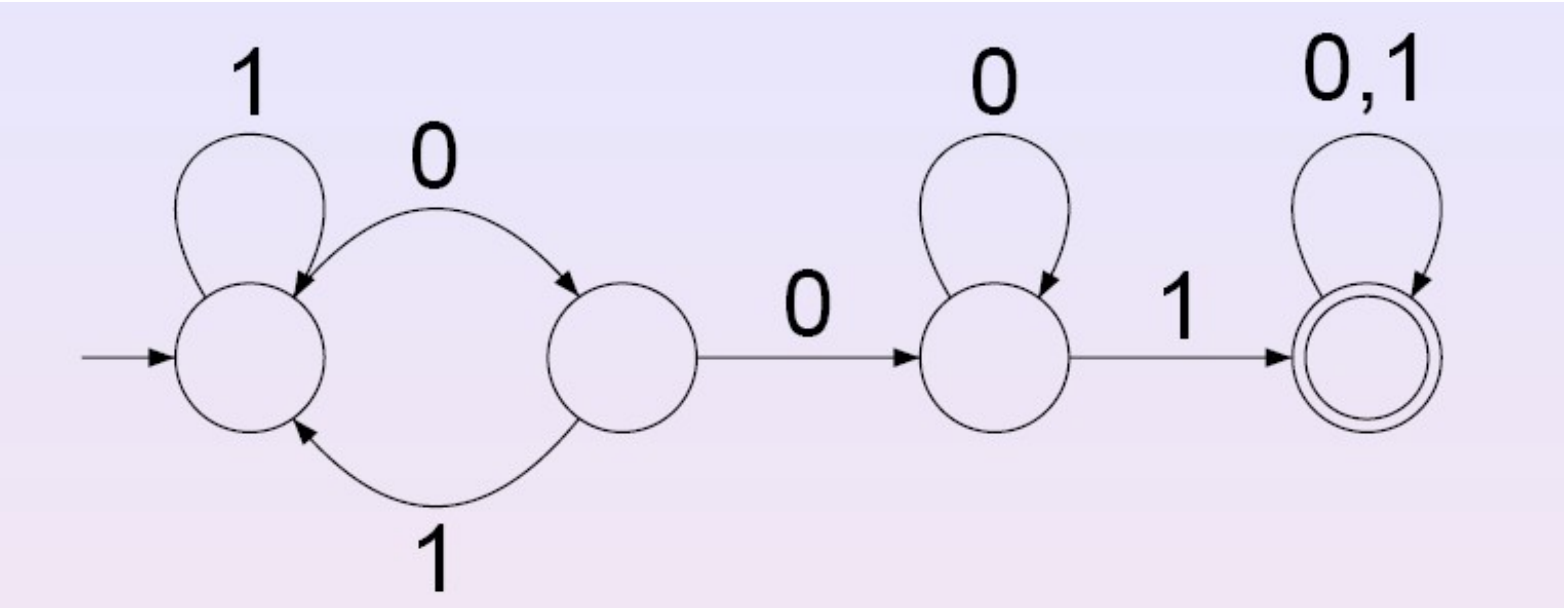
The reverse of a regular language is also a regular language.

- If a language can be recognized by a DFA that reads strings from right to left, then there is an "normal" DFA (one that reads from left to right) that accepts the same language.
- Counter-intuitive! DFAs have finite memory...

Reversing a DFA

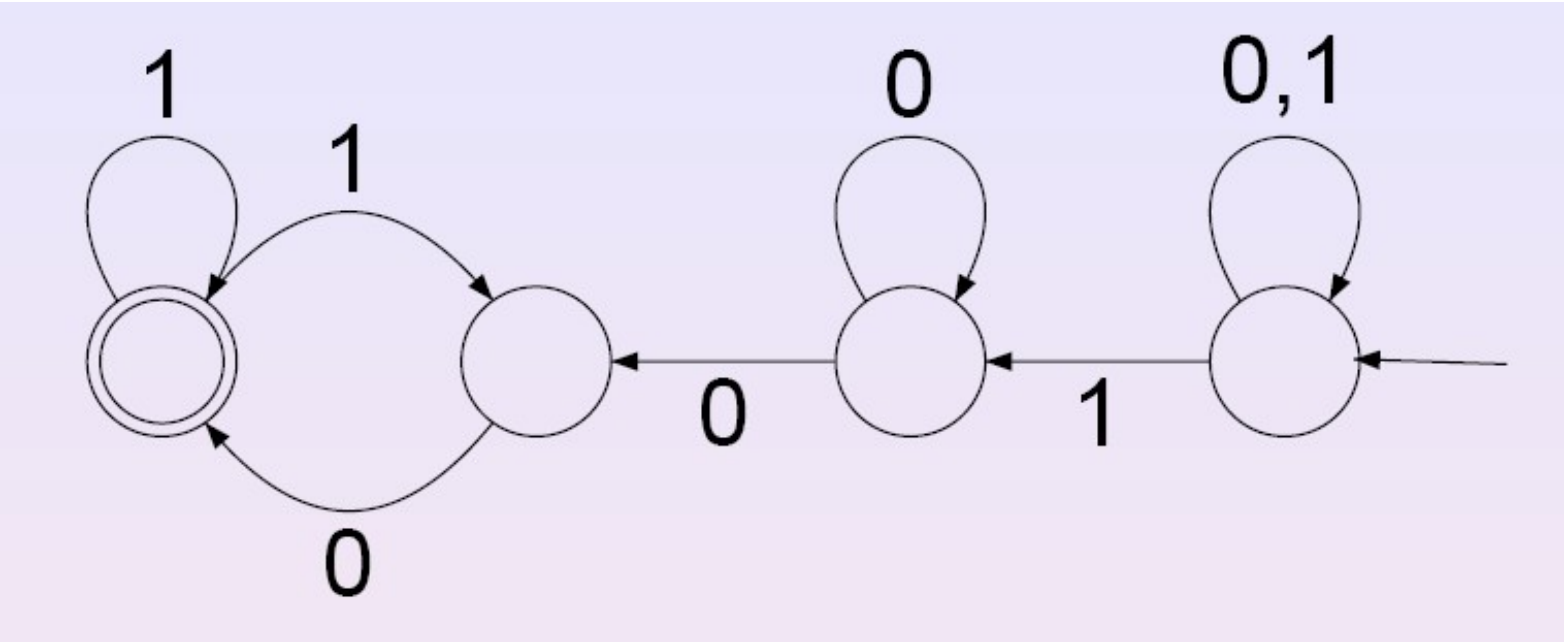
- Assume L is a regular language. Let M be a DFA that recognizes L
- We will build a machine M^R that accepts L^R
- If M accepts ω, then ω describes a directed path, in M, from the start state to a final state.
- First attempt: Try to define M^R as M as follows
  - Reverse all transitions
  - Turn the start state to a final state
  - Turn the final states to start states!
- But, as such, M^R is not always a DFA.
  - It could have many start states.
  - Some states may have too many outgoing transitions or none at all!

EXAMPLE



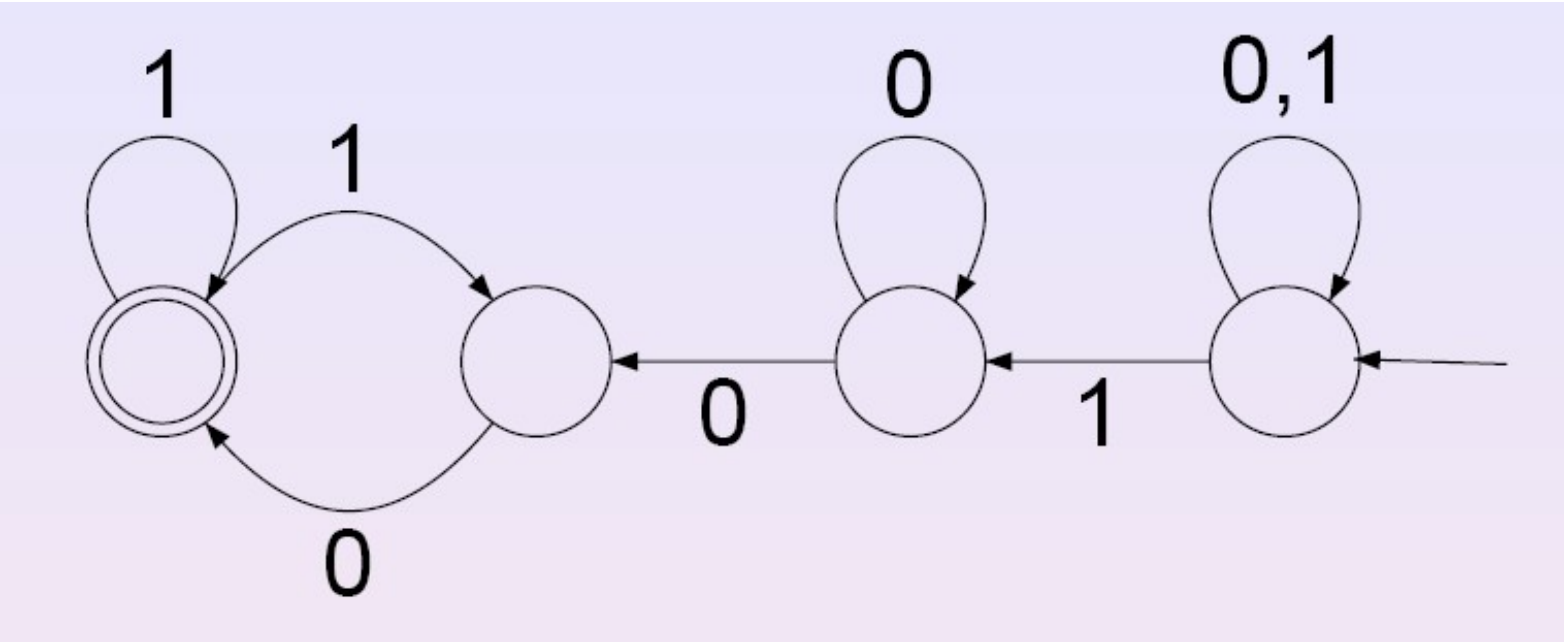
- What language does this DFA recognize?
  - All strings that contain a substring of 2 or more 0s followed by a 1.

REVERSING THE DFA



- What happens with input 100?
  - There are multiple transitions from a state labeled with the same symbol.
  - State transitions are not deterministic any more: the next state is not uniquely determined by the current state and the current input. $\rightarrow$ Nondeterminism

REVERSING THE DFA

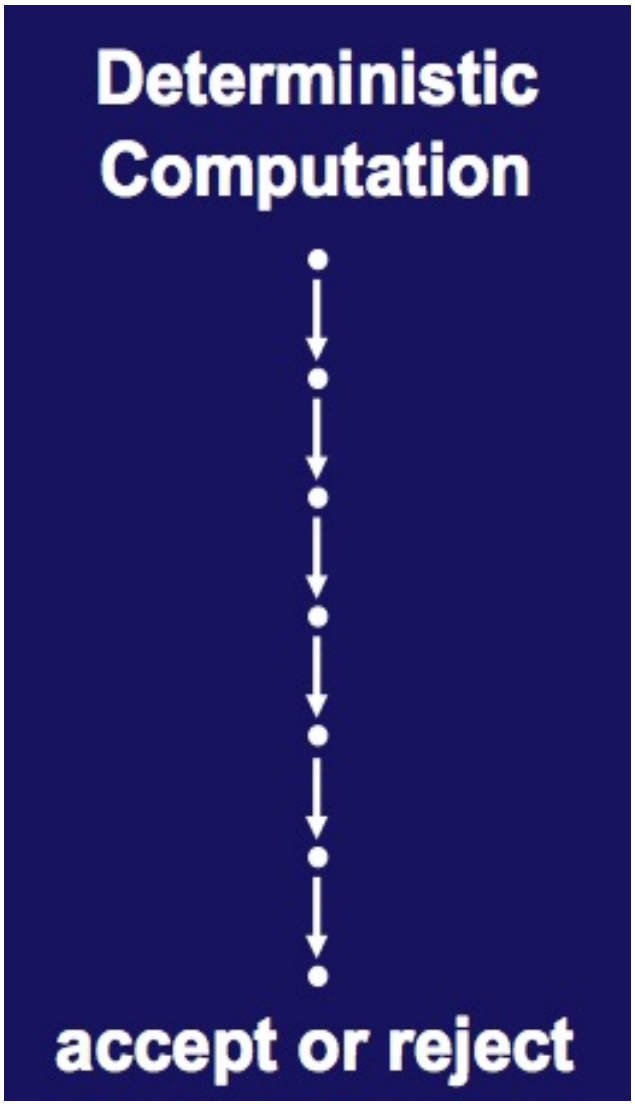


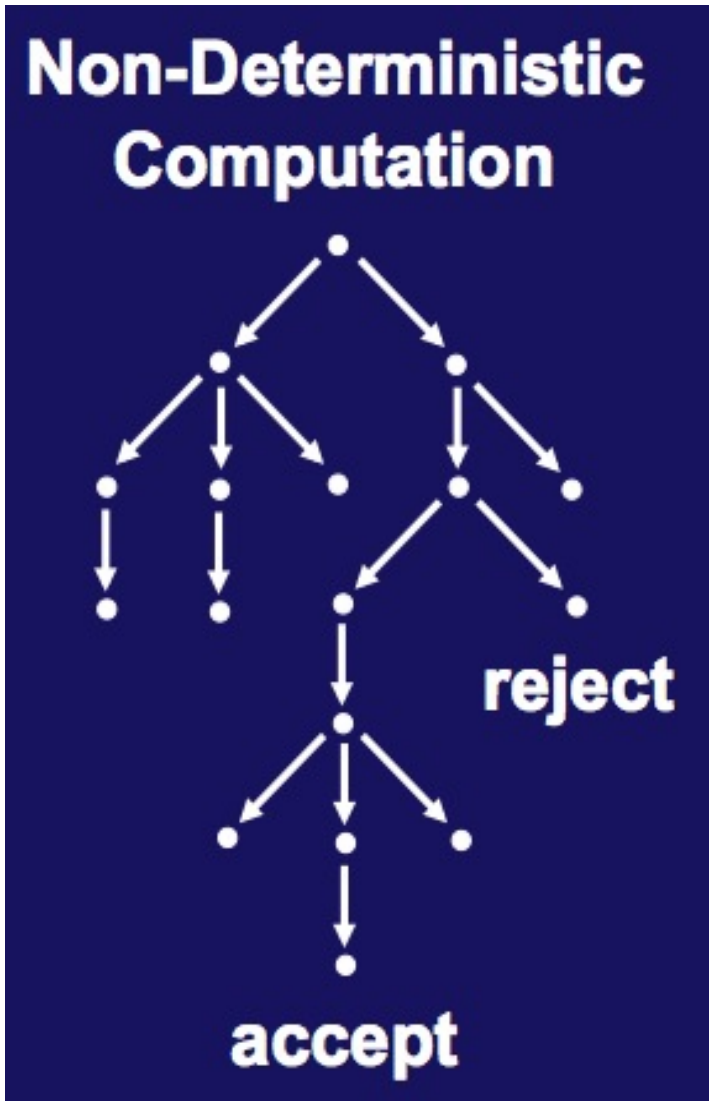
- We will say that this machine accepts a string if there is some path that reaches an accept state from a start state.

How does nondeterminism work?

- When a nondeterministic finite state automaton (NFA) reads an input symbol and there are multiple transitions labeled with that symbol
  - It splits into multiple copies of itself, and
  - follows all possibilities in parallel.

Deterministic vs Nondeterministic Computation





How does nondeterminism work?

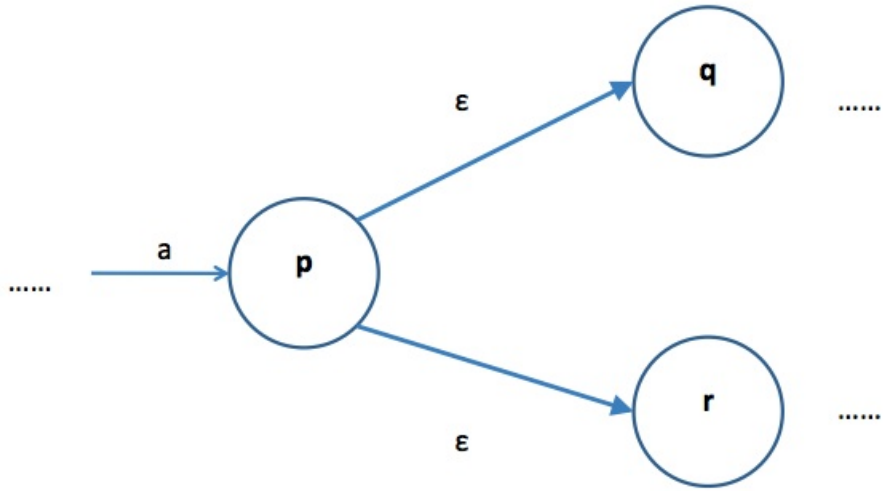
- When a nondeterministic finite state automaton (NFA) reads an input symbol and there are multiple transitions with labeled with that symbol
  - It splits into multiple copies of itself, and
  - follows all possibilities in parallel.
- Each copy of the machine takes one of the possible ways to proceed and continues as before.
- If there are subsequent choices, the machine splits again.
  - We have an unending supply of these machines that we can boot at any point to any state!

DFAs and NFAs – Other differences

- A state need not have a transition with every symbol in Σ
  - No transition with the next input symbol? $\Rightarrow$ that copy of the machine dies, along with the branch of computation associated with it.
  - If any copy of the machine is in a final state at the end of the input, the NFA accepts the input string.
- NFAs can have transitions labeled with ϵ – the empty string.

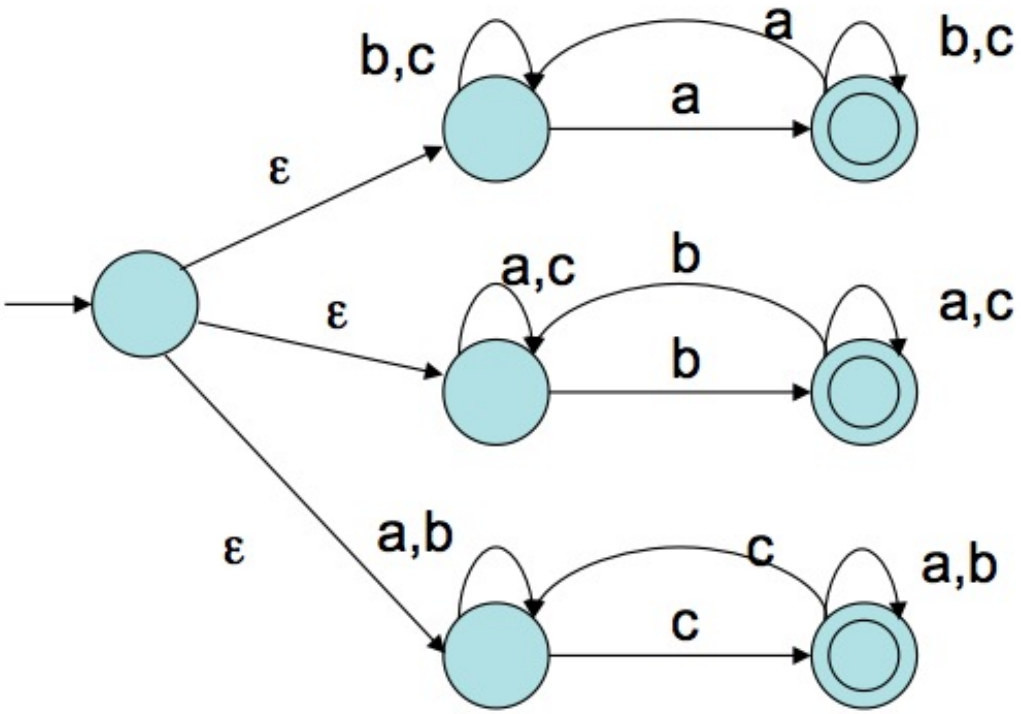
ϵ-TRANSITIONS

- If a state with only transitions with an ϵ label is encountered, something similar happens:
  - The machine does not read the next input symbol.
  - It splits into multiple copies:
    - a separate copy follows each ϵ transition
    - one stays at the current state



- What the NFA arrives at p (say after having read input *a*, it splits into 3 copies

NFA Example

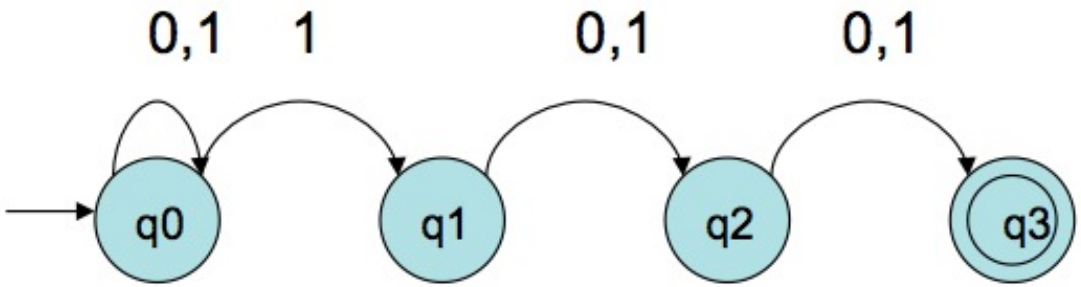


- Accepts all strings over $\Sigma = \{a, b, c\}$ with at least one of the symbols occuring an odd number of times.
- For example, the machine copy taking the upper ϵ transition guesses that there are an odd number of *a*'s and then tries to verify it.

NONDETERMINISM

- So nondeterminism can also be viewed as
  - guessing the future, and
  - then verifying it as the rest of the input is read in.
- If the machine's guess is not verifiable, it dies!

NFA Example



- Accepts all strings over $\Sigma = \{0, 1\}$ where the 3rd symbol from the end is a 1.
  - How do you know that a symbol is the 3rd symbol from the end?
- The start state guesses every 1 is the 3rd from the end, and then the rest tries to verify that it is or it is not.
  - The machine dies if you reach the final state and you get one more symbol.

NFA–Formal Definition

- A Nondeterministic Finite State Acceptor (NFA) is defined as the 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where
  - Q is a finite set of states
  - Σ is a finite set of symbols – the alphabet
  - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$, is the next-state function
    - $2^Q = \{P | P \subseteq Q\}$
  - $q_0 \in Q$ is the (label of the) start state
  - $F \subseteq Q$ is the set of final (accepting) states
- δ maps states and inputs (including ϵ) to a set of possible next states
- Similarly $\delta^* : Q \times \Sigma^* \rightarrow 2^Q$
  - $\delta^*(q, \epsilon) = \{q\}$
  - $\delta^*(q, \omega \cdot a) = \{p | \exists r \in \delta^*(q, \omega) \text{ such that } p \in \delta(r, a)\}$
    - a could be ϵ

How an NFA accepts strings

- An NFA accepts a string $\omega = x_1 x_2 \cdots x_n$ if a sequence of states $r_0 r_1 r_2 \cdots r_n, r_i \in Q$ exist such that
  1. $r_0 = q_0$ (Start in the initial state)
  2. $r_i \in \delta(r_{i-1}, x_i)$ for $i = 1, 2, \ldots n$ (Move from state to state – nondeterministically: r_i is one of the allowable next states)
  3. $r_n \in F$ (End up in a final state)

Nondeterministic vs Deterministic FA

- We know that DFAs accept regular languages.
- Are NFAs strictly more powerful than DFAs?
  - Are there languages that some NFA will accept but no DFA can accept?
- It turns out that NFAs and DFAs accept the same set of languages.
  - Q is finite $\Rightarrow |2^Q| = |\{P | P \subseteq Q\}| = 2^{|Q|}$ is also finite.

NFAs and DFAs are equivalent

Theorem

Every NFA has an equivalent DFA.

Proof Idea

- Convert the NFA to an equivalent DFA that accepts the same language.
- If the NFA has k states, then there are 2^k possible subsets (still finite)
- The states of the DFA are labeled with subsets of the states of the NFA
- Thus the DFA can have up to 2^k states.

NFAs and DFAs are equivalent

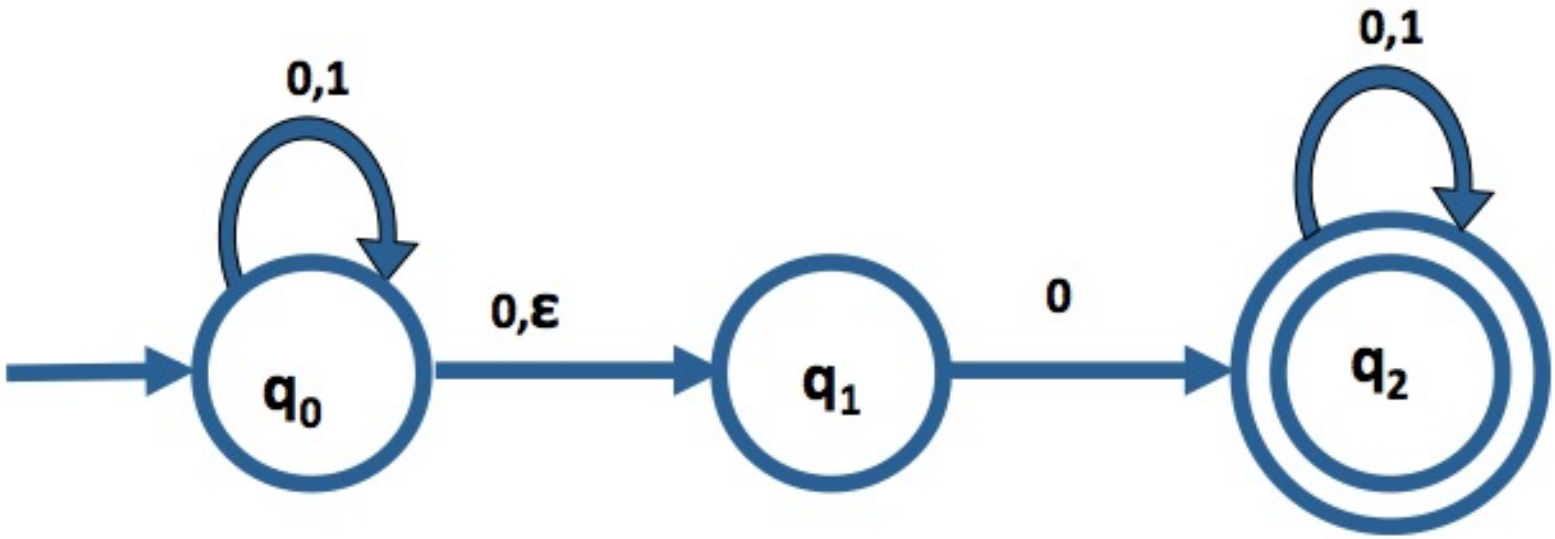
Theorem

Every NFA has an equivalent DFA.

Construction

- Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. We construct $M = (Q', \Sigma, \delta', q_0', F')$.
  1. $Q' = 2^Q$, the power set of Q
  2. For $R \in Q'$ and $a \in \Sigma$, let $\delta'(R, a) = \{q \in Q | q \in \epsilon(\delta(r, a))$ for some $r \in R\}$
     - For $R \in Q$, the ϵ-closure of R, is defined as $\epsilon(R) = \{q | q$ is reachable from some $r \in R$ by traveling along zero or more $\epsilon -$ transitions$\}$
  3. $q_0' = \epsilon(\{q_0\})$
  4. $F' = \{R \in Q' | R \cap F \neq \phi\}$: at least one of the states in R is a final state of N

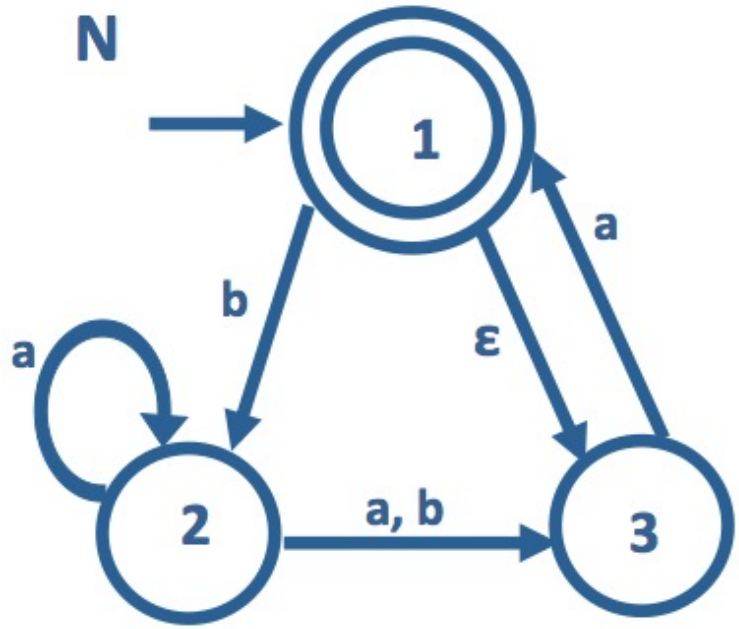
NFA Example



- Note that q_0 has an ϵ-transition
- Some states (e.g., q_1) do not have a transition for some of the symbols in Σ. Machine dies if it sees input 1 when it is in state q_1.
- $\epsilon(\{q_0\}) = \{q_0, q_1\}$

NFA to DFA Conversion Example

- Given $N = (\{1,2,3\}, \{a,b\}, \delta, 1, \{1\})$, construct $M = (Q', \Sigma, \delta', q_0', F')$.

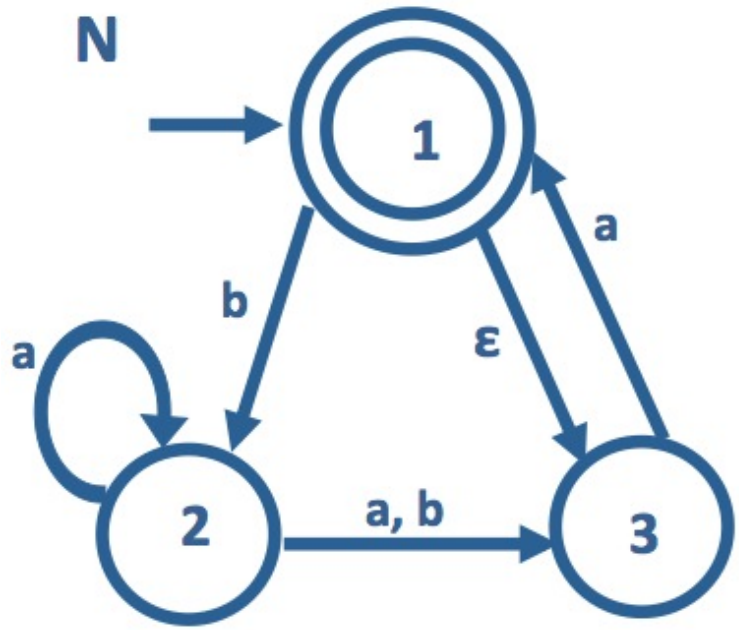


$\epsilon(\{1\}) = \{1,3\}$

	δ'	**a**	**b**
	ϕ	ϕ	ϕ
	$\{1\}$	ϕ	$\{2\}$
	$\{2\}$	$\{2,3\}$	$\{3\}$
	$\{3\}$	$\{1,3\}$	ϕ
	$\{1,2\}$	$\{2,3\}$	$\{2,3\}$
	$\{1,3\}$	$\{1,3\}$	$\{2\}$
	$\{2,3\}$	$\{1,2,3\}$	$\{3\}$
	$\{1,2,3\}$	$\{1,2,3\}$	$\{2,3\}$

NFA to DFA Conversion Example

- Given $N = (\{1,2,3\}, \{a,b\}, \delta, 1, \{1\})$, construct $M = (Q', \Sigma, \delta', q_0', F')$.



$\epsilon(\{1\}) = \{1,3\}$

$\{1,3\}$ is the start state of M

	δ'	**a**	**b**
	ϕ	ϕ	ϕ
	$\{1\}$	ϕ	$\{2\}$
	$\{2\}$	$\{2,3\}$	$\{3\}$
	$\{3\}$	$\{1,3\}$	ϕ
	$\{1,2\}$	$\{2,3\}$	$\{2,3\}$
	$\{1,3\}$	$\{1,3\}$	$\{2\}$
	$\{2,3\}$	$\{1,2,3\}$	$\{3\}$
	$\{1,2,3\}$	$\{1,2,3\}$	$\{2,3\}$

NFA to DFA Conversion Example

- Given $N = (\{1,2,3\}, \{a,b\}, \delta, 1, \{1\})$, construct $M = (Q', \Sigma, \delta', q_0', F')$.

  - States $\{1\}$ and $\{1,2\}$ do not appear as the next state in any transition! They can be removed
  - States with labels $\{1,3\}$ and $\{1,2,3\}$ are the final states of M.
  - We can now relabel the states as we wish!

	δ'	**a**	**b**
q_5	ϕ	ϕ	ϕ
q_2	$\{2\}$	$\{2,3\}$	$\{3\}$
q_1	$\{3\}$	$\{1,3\}$	ϕ
q_0	$\{1,3\}$	$\{1,3\}$	$\{2\}$
q_3	$\{2,3\}$	$\{1,2,3\}$	$\{3\}$
q_4	$\{1,2,3\}$	$\{1,2,3\}$	$\{2,3\}$

NFA to DFA Conversion Example

- Given $N = (\{1,2,3\}, \{a,b\}, \delta, 1, \{1\})$, construct $M = (Q', \Sigma, \delta', q_0', F')$.

  - States $\{1\}$ and $\{1,2\}$ do not appear as the next state in any transition! They can be removed
  - States with labels $\{1,3\}$ and $\{1,2,3\}$ are the final states of M.
  - We can now relabel the states as we wish!

δ'	**a**	**b**
q_5	q_5	q_5
q_2	q_3	q_1
q_1	q_0	q_5
$\mathbf{q_0}$	q_0	q_2
q_3	q_4	q_1
q_4	q_4	q_3

Formal Languages, Automata and Computation

Regular Expressions

Summary

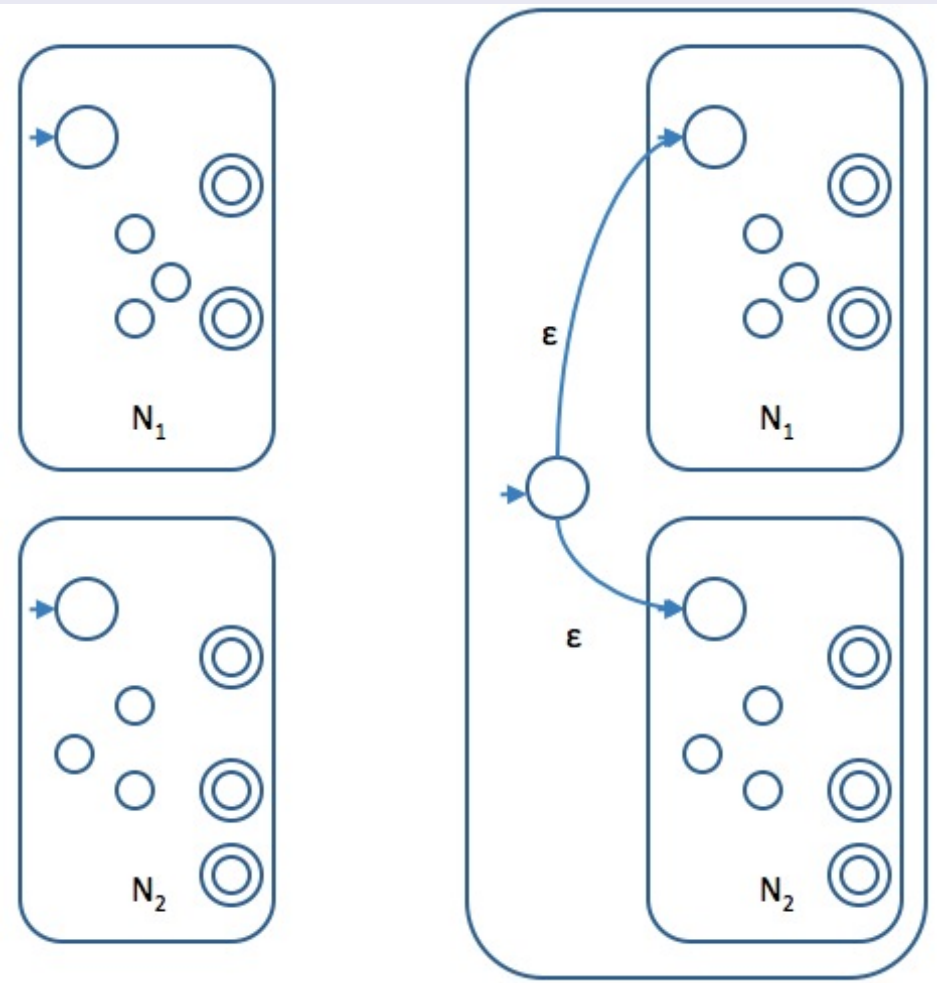
- Nondeterminism
  - Clone the FA at choice points
  - Guess and verify
- Nondeterministic FA
  - Multiple transitions from a state with the same input symbol
  - ϵ-transitions
- NFAs are equivalent to DFAs
  - Determinization procedure builds a DFA with up to 2^k states for an NFA with k states.

Closure Theorems

Theorem

The class of regular languages is closed under the union operation.

Proof Idea based on NFAs

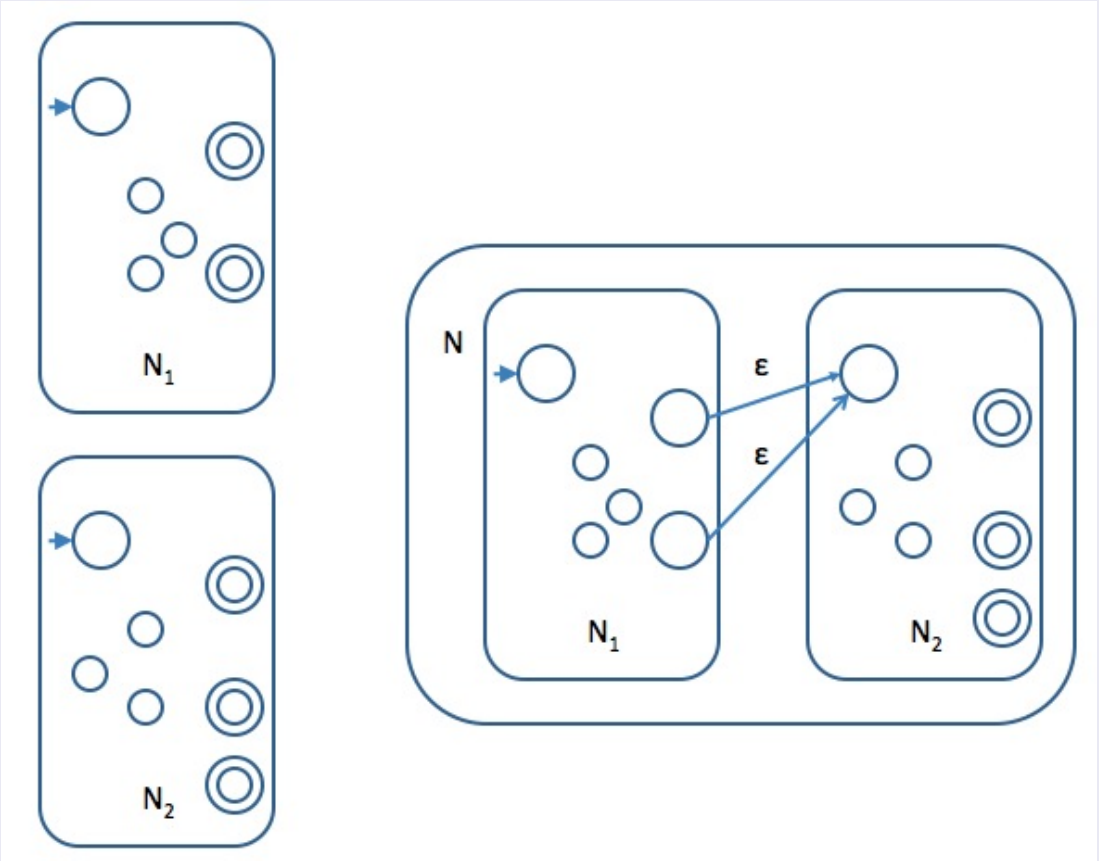


Closure Theorems

Theorem

The class of regular languages is closed under the concatenation operation.

Proof Idea based on NFAs

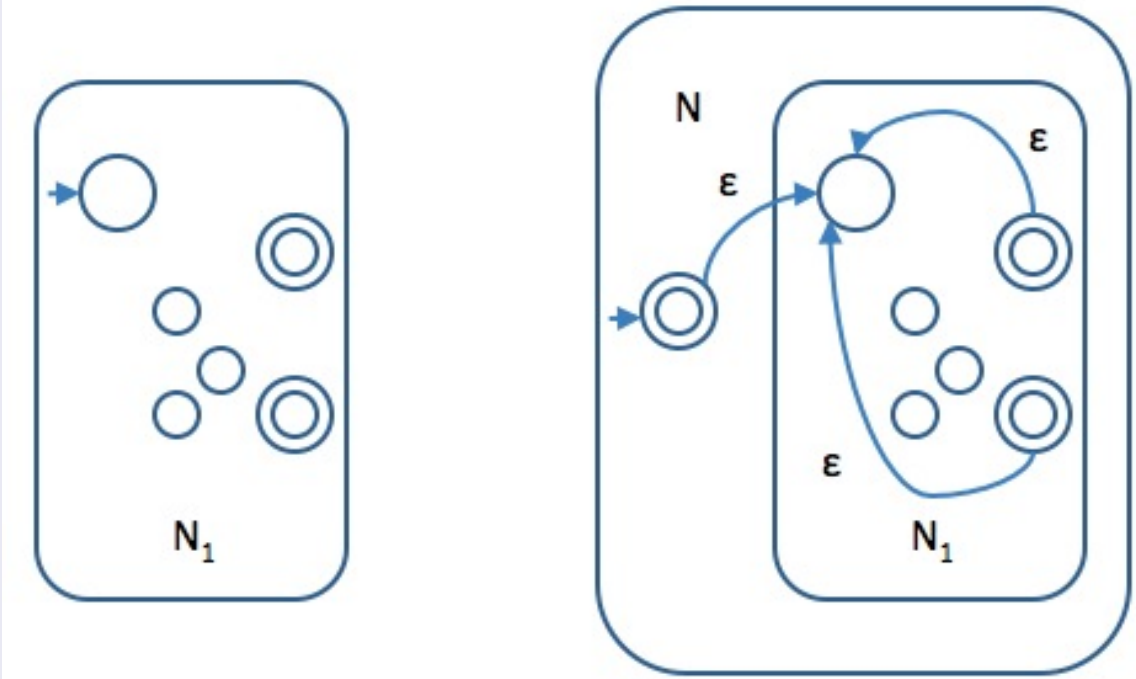


Closure Theorems

Theorem

The class of regular languages is closed under the star operation.

Proof Idea based on NFAs



Regular Expressions

- DFAs are finite descriptions of (finite or infinite) sets of strings
  - Finite number of symbols, states, transitions
- Regular Expressions provide an algebraic expression framework to describe the same class of strings
- Thus, DFAs and Regular Expressions are equivalent.

Regular Expressions

- For every regular expression, there is a corresponding regular set or language
- R, R_1, R_2 are regular expressions; $L(R)$ denotes the corresponding regular set

Regular Expression	Regular Set
ϕ	$\{\}$
a for a $\in \Sigma$	$\{a\}$
ϵ	$\{\epsilon\}$
$(R_1 \cup R_2)$	$L(R_1) \cup L(R_2)$
$(R_1 R_2)$	$L(R_1)L(R_2)$
(R^*)	$L(R)^*$

Regular Expressions– More Syntax

Regular Expression	Regular Set
ϕ	$\{\}$
a for $\in \Sigma$	$\{a\}$
ϵ	$\{\epsilon\}$
$(R_1 \cup R_2)$	$L(R_1) \cup L(R_2)$
$(R_1 R_2)$	$L(R_1)L(R_2)$
(R^*)	$L(R)^*$

- Some books may also use $R_1 + R_2$ to denote union.
- In $(\ldots)$, the parenthesis can be deleted
  - In which case, interpretation is done in the precedence order: star, concatenation and then union.
- $R^+ = RR^*$ and R^k for k-fold concatenation are useful shorthands.

Regular Expression Examples

Regular Expression		**Regular Language**
$\mathbf{0}^*\mathbf{1}\mathbf{0}^*$	$\rightarrow$	$\{\omega \mid \omega$ contains a single $\mathbf{1}\}$
$(\mathbf{0} \cup \mathbf{1})^*\mathbf{1}(\mathbf{0} \cup \mathbf{1})^*$	$\rightarrow$	$\{\omega \mid \omega$ has at least one $\mathbf{1}\}$
$\mathbf{0}(\mathbf{0} \cup \mathbf{1})^*\mathbf{0} \cup \mathbf{1}(\mathbf{0} \cup \mathbf{1})^*\mathbf{1} \cup \mathbf{0} \cup \mathbf{1}$	$\rightarrow$	$\{\omega \mid \omega$ starts and ends with the same symbol$\}$
$(\mathbf{0}^*\mathbf{1}\mathbf{0}^*\mathbf{1})^*\mathbf{0}^*$	$\rightarrow$	$\{\omega \mid n_1(\omega)$ is even$\}$

Writing Regular Expressions

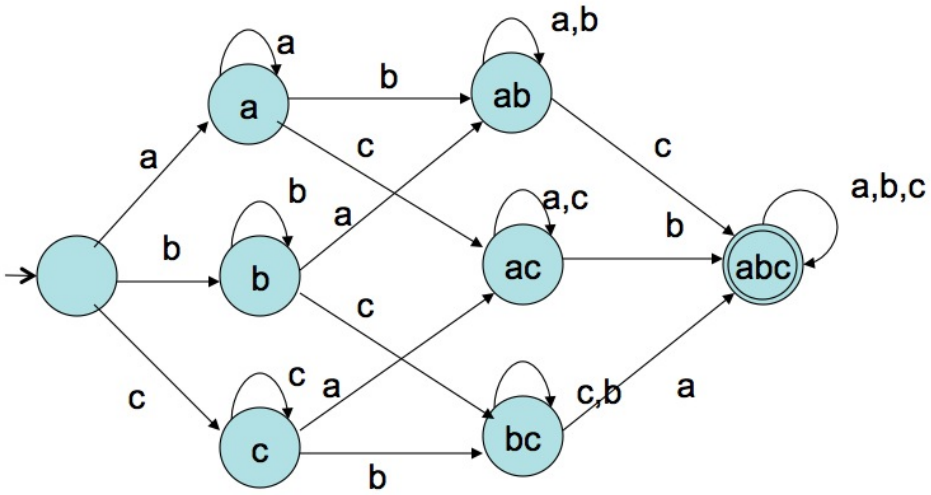
- All strings with at least one pair of consecutive 0s
  - $(\mathbf{0} \cup \mathbf{1})^*\mathbf{00}(\mathbf{0} \cup \mathbf{1})^*$
- All strings such that fourth symbol from the end is a 1
  - $(\mathbf{0} \cup \mathbf{1})^*\mathbf{1}(\mathbf{0} \cup \mathbf{1})(\mathbf{0} \cup \mathbf{1})(\mathbf{0} \cup \mathbf{1})$
- All strings with no pair of consecutive 0s
  - $(\mathbf{1}^*\mathbf{011}^*)^*(\mathbf{0} \cup \epsilon) \cup \mathbf{1}*$
  - Strings consist of repetitions of 1 or 01 or two boundary cases: $(\mathbf{1} \cup \mathbf{01})^*(\mathbf{0} \cup \epsilon)$
- All strings that do not end in 01.
  - $(\mathbf{0} \cup \mathbf{1})^*(\mathbf{00} \cup \mathbf{10} \cup \mathbf{11}) \cup \mathbf{0} \cup \mathbf{1} \cup \epsilon$

Writing Regular Expressions

- All strings over $\Sigma = \{a, b, c\}$ that contain every symbol at least once.
- $(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{a}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{b}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{c}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^* \cup$
  $(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{a}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{c}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{b}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^* \cup$
  $(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{b}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{a}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{c}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^* \cup$
  $(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{b}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{c}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{a}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^* \cup$
  $(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{c}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{a}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{b}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^* \cup$
  $(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{c}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{b}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*\mathbf{a}(\mathbf{a} \cup \mathbf{b} \cup \mathbf{c})^*$

Writing Regular Expressions

- All strings over $\Sigma = \{a, b, c\}$ that contain every symbol at least once.



- DFAs and REs may need different ways of looking at the problem.
  - For the DFA, you count symbols
  - For the RE, you enumerate all possible patterns

RE Identities

- $\mathbf{R} \cup \phi = \mathbf{R}$
- $\mathbf{R}\epsilon = \epsilon\mathbf{R} = \mathbf{R}$
- $\phi^* = \epsilon$
- Note that we do not have explicit operators for intersection or complementation!

Digression: REs in Real life

- Linux/Unix Shell, Perl, Awk, Python all have built in regular expression support for pattern matching functionality
- See `http://perldoc.perl.org/perlre.html`
- Mostly some syntactic extensions/changes to basic regular expressions with some additional functionality for remembering matches
- Substring matches in a string!
- Search for and download *Regex Coach* to learn and experiment with regular expression matching

Equivalence with Finite Automata

Theorem

A language is regular if and only if some regular expression describes it.

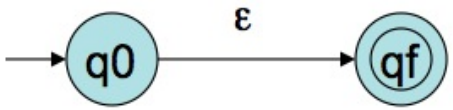
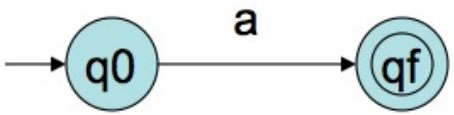
Lemma- The *if* part

If a language is described by a regular expression, then it is regular

Proof Idea

Inductively convert a given regular expression to an NFA.

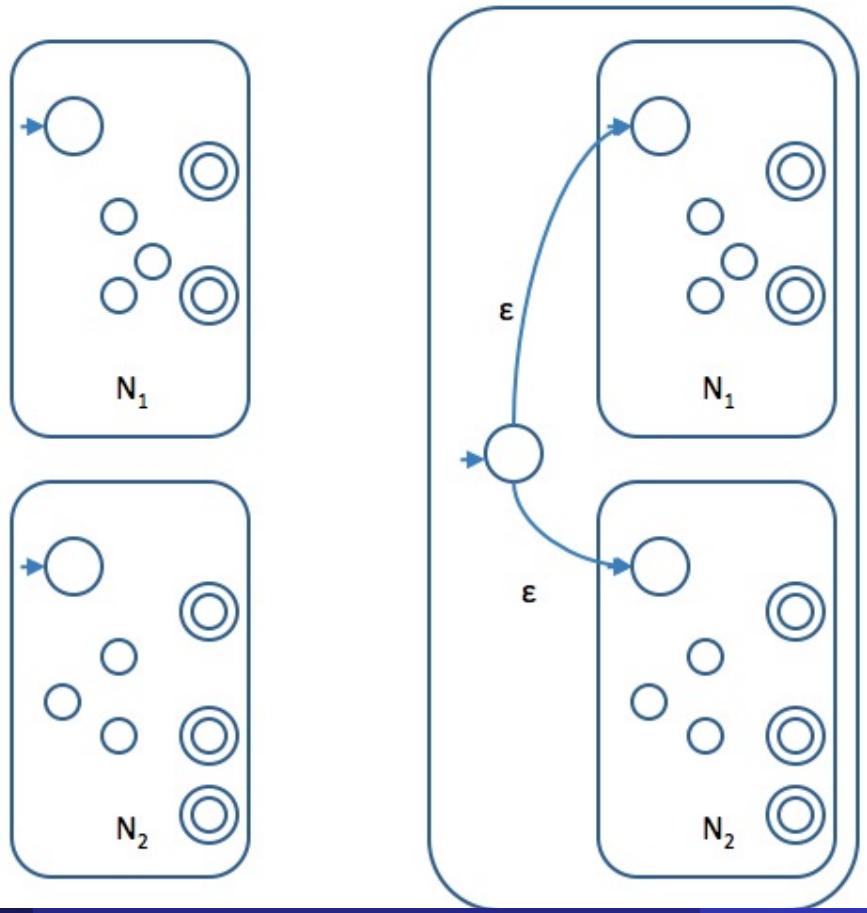
Converting REs to NFAs: Basis Cases

Regular Expression	Corresponding NFA
ϕ	
ϵ	
a for $a \in \Sigma$	

Converting REs to NFAs

Union

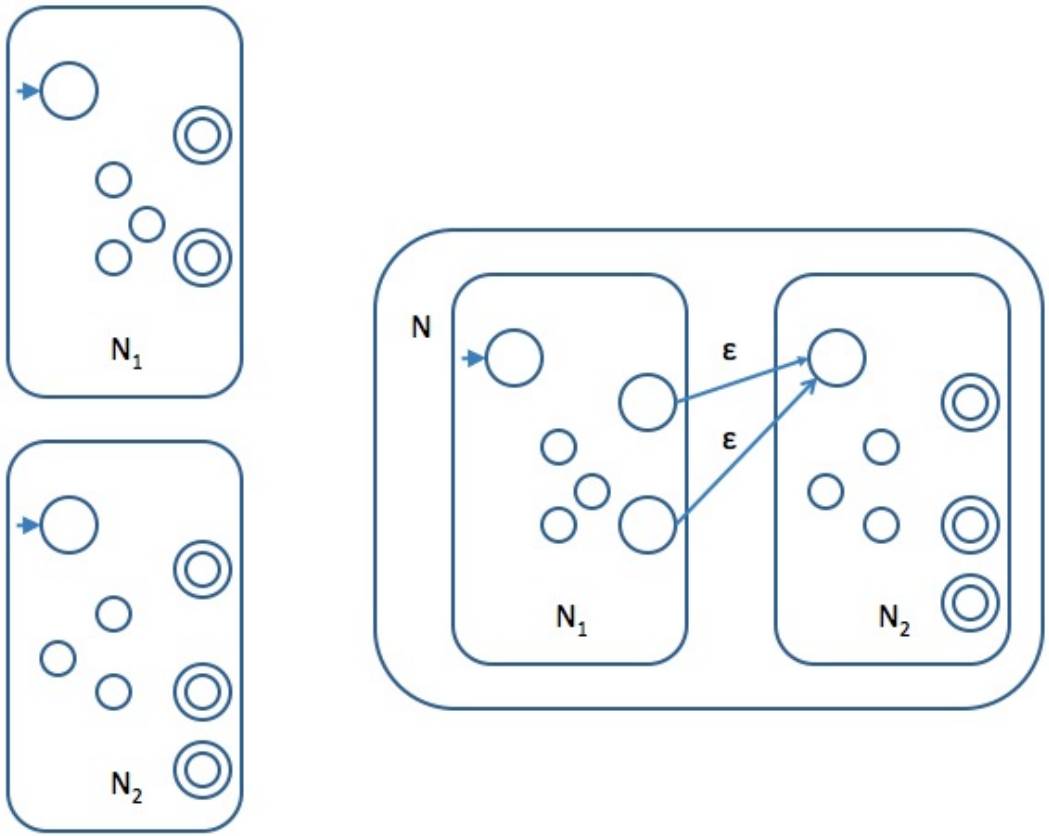
- Let N_1 and N_2 be NFAs for R_1 and R_2 respectively. Then the NFA for $\mathbf{R_1 \cup R_2}$ is



Converting REs to NFAs

Concatenation

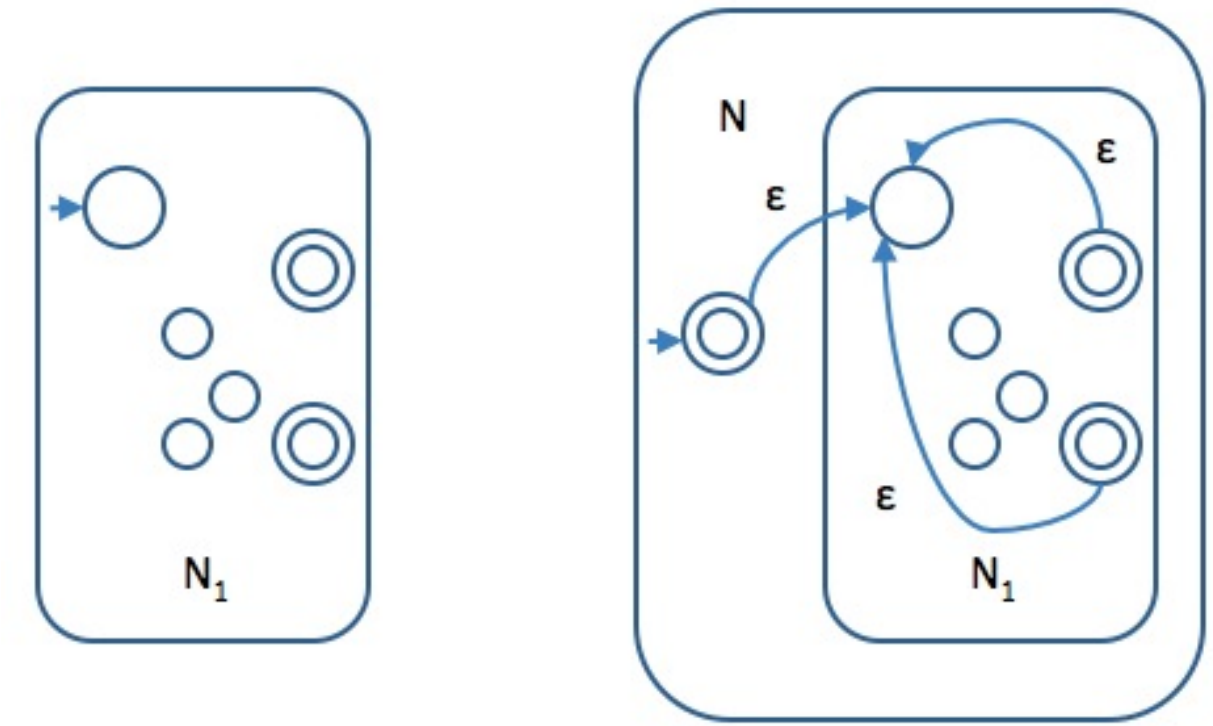
- Let N_1 and N_2 be NFAs for R_1 and R_2 respectively. Then the NFA for $\mathbf{R_1R_2}$ is



Converting REs to NFAs: Star

Star

- Let *N* be NFAs for *R*. Then the NFA for $\mathbf{R}^*$ is



RE TO NFA CONVERSION EXAMPLE

- Let's convert $(\mathbf{a} \cup \mathbf{b})^*\mathbf{aba}$ to an NFA.

RE TO NFA TO DFA

- Regular Expression $\rightarrow$ NFA (possibly with ϵ-transitions)
- NFA $\rightarrow$ DFA via determinization

Equivalence with Finite Automata

Theorem

A language is regular if and only if some regular expression describes it.

Lemma – The *only if* part

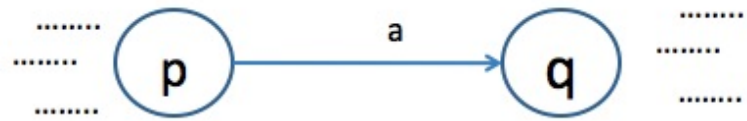
If a language is regular then it is described by a regular expression

Proof Idea

- Generalized transitions: label transitions with regular expressions
- Generalized NFAs (GNFA)
- Iteratively eliminate states of the GNFA one by one, until only two states and a single generalized transition is left.

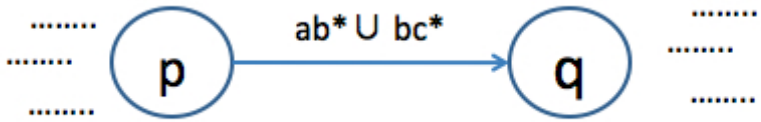
Generalized Transitions

- DFAs have single symbols as transition labels



  - If you are in state *p* and the next input symbol matches *a*, go to state *q*

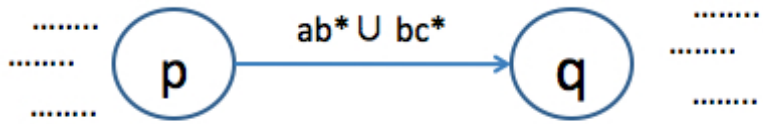
- Now consider



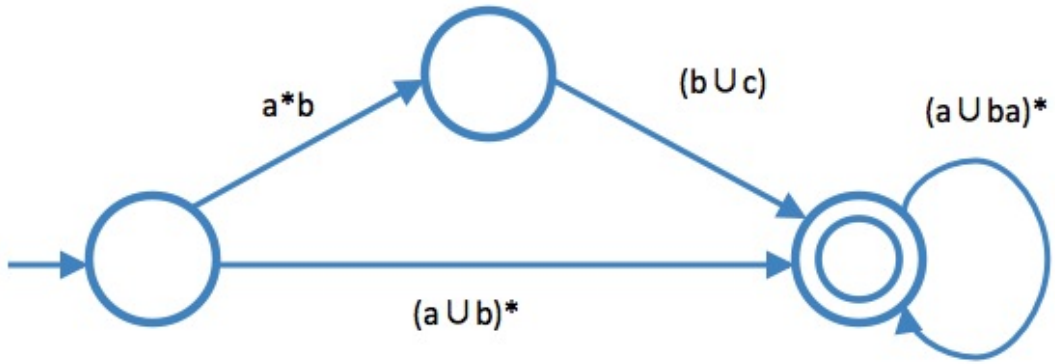
  - If you are in state *p* and a prefix of the remaining input matches the regular expression $\mathbf{ab}^* \cup \mathbf{bc}^*$ then go to state *q*

Generalized Transitions and NFA

- A generalized transition is a transition whose label is a regular expression



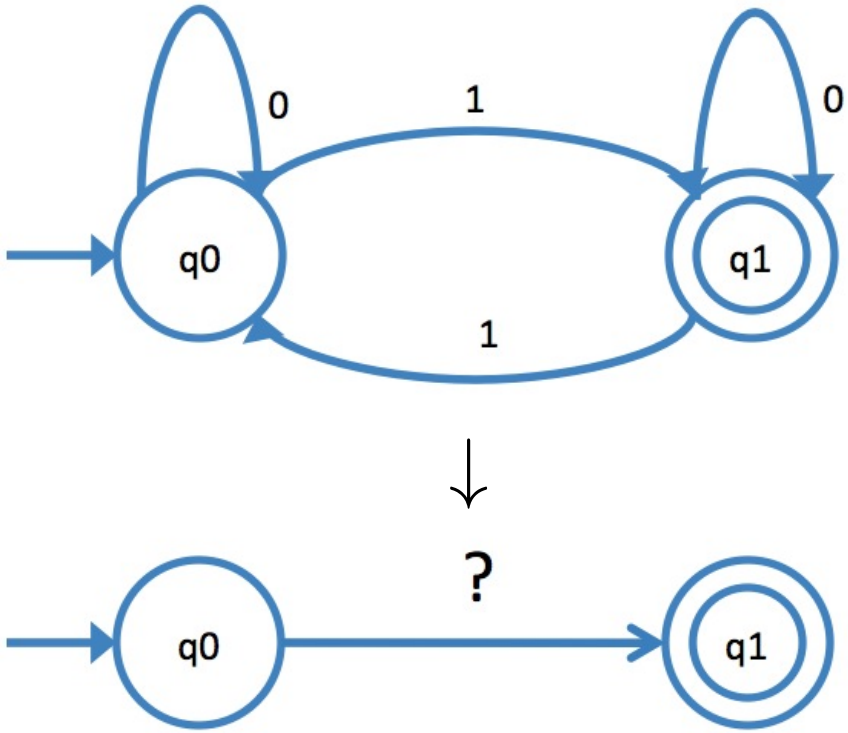
- A Generalized NFA is an NFA with generalized transitions.



  - In fact, all standard DFA transitions are generalized transitions with regular expressions of a single symbol!

Generalized Transitions

- Consider the 2-state DFA



  - $\mathbf{0}^*\mathbf{1}$ takes the DFA from state q_0 to q_1
  - $(\mathbf{0} \cup \mathbf{10}^*\mathbf{1})^*$ takes the machine from q_1 back to q_1
  - So $\mathbf{?} = \mathbf{0}^*\mathbf{1}(\mathbf{0} \cup \mathbf{10}^*\mathbf{1})^*$ represents all strings that take the DFA from state q_0 to q_1

Generalized NFAs

- Take any NFA and transform it into a GNFA
  - with only two states: one start and one accept
  - with one generalized transition
- then we can "read" the regular expression from the label of the generalized transition (as in the example above)

Formal Languages, Automata and Computation

DFAs to Regular Expressions

DFA Minimization – Closure Properties

Summary

- Regular Expression (RE) define regular sets
- RE $\Rightarrow$ NFA $\Rightarrow$ DFA

Equivalence of REs to Finite Automata

Theorem

A language is regular if and only if some regular expression describes it.

Lemma – The *only if* part

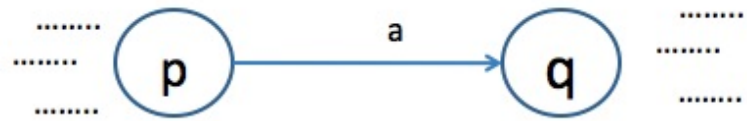
If a language is regular then it is described by a regular expression

Proof Idea

- Generalized transitions: label transitions with regular expressions
- Generalized NFAs (GNFA)
- Iteratively eliminate states of the GNFA one by one, until only two states and a single generalized transition is left.

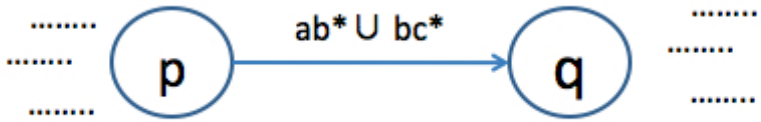
Generalized Transitions

- DFAs have single symbols as transition labels



  - If you are in state *p* and the next input symbol matches *a*, go to state *q*

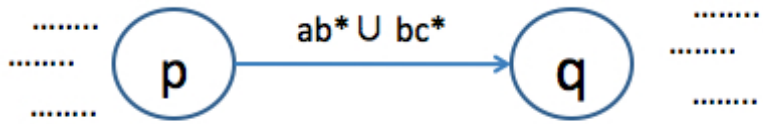
- Now consider



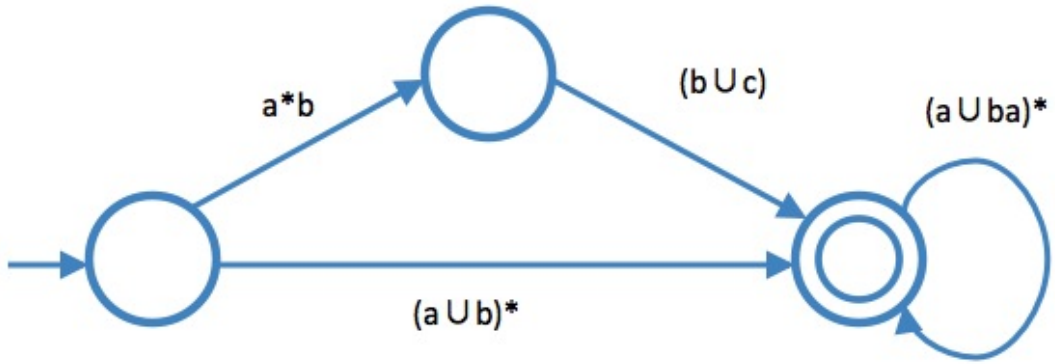
  - If you are in state *p* and a prefix of the remaining input matches the regular expression $\mathbf{ab}^* \cup \mathbf{bc}^*$ then go to state *q*

Generalized Transitions and NFA

- A generalized transition is a transition whose label is a regular expression



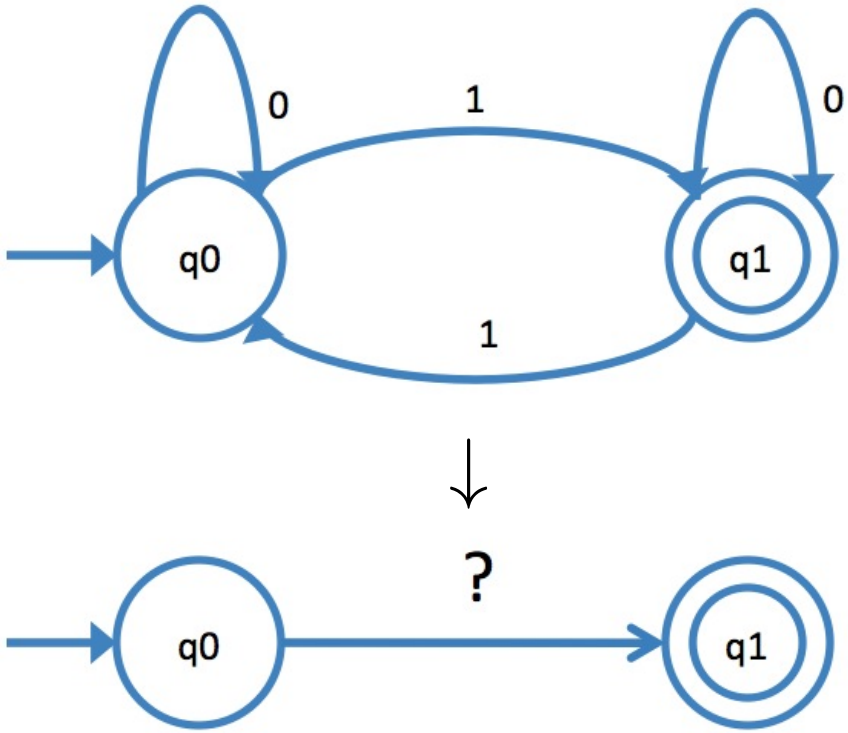
- A Generalized NFA is an NFA with generalized transitions.



  - In fact, all standard DFA transitions are generalized transitions with regular expressions of a single symbol!

Generalized Transitions

- Consider the 2-state DFA



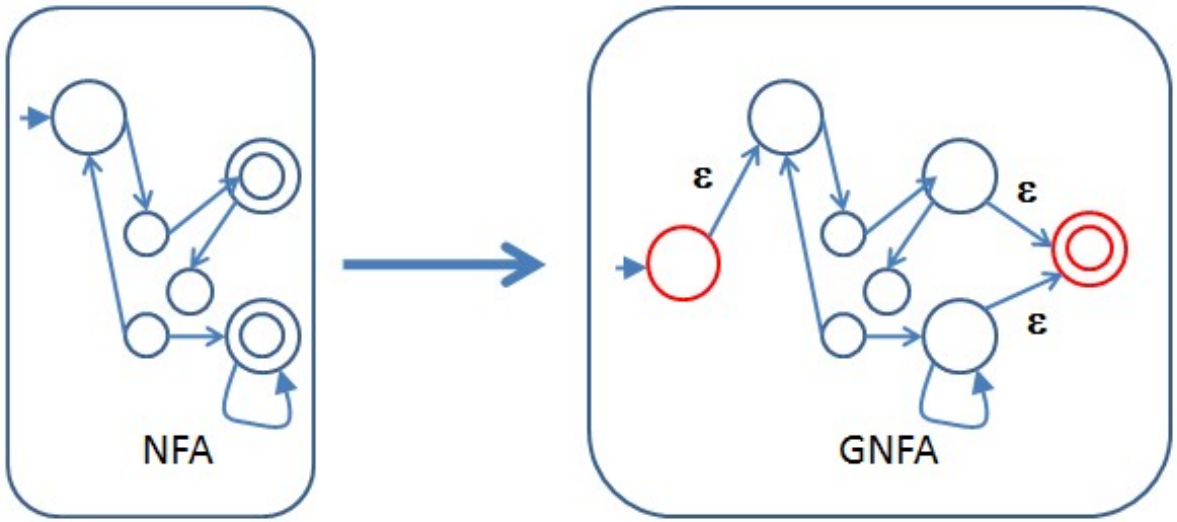
  - $\mathbf{0}^*\mathbf{1}$ takes the DFA from state q_0 to q_1
  - $(\mathbf{0} \cup \mathbf{10}^*\mathbf{1})^*$ takes the machine from q_1 back to q_1
  - So $\mathbf{?} = \mathbf{0}^*\mathbf{1}(\mathbf{0} \cup \mathbf{10}^*\mathbf{1})^*$ represents all strings that take the DFA from state q_0 to q_1

Generalized NFAs

- Take any DFA and transform it into a GNFA
  - with only two states: one start and one accept
  - with one generalized transition
- then we can "read" the regular expression from the label of the generalized transition (as in the example above)

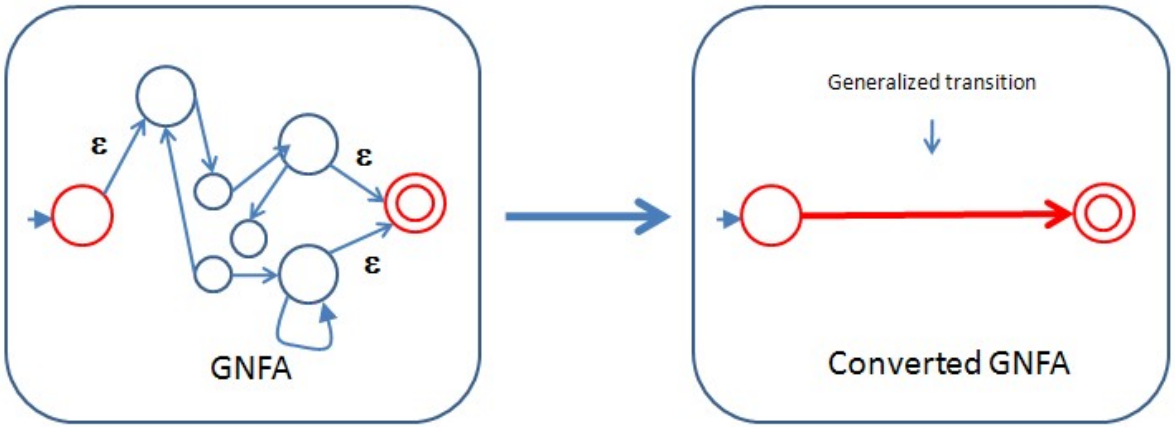
DFA to GNFA

- We will add two new states to a DFA:
  - A new start state with an ϵ-transition to the original start state, but with no transitions from any other state
  - A new final state with an ϵ-transition from all the original final states, but with no transitions to any other state
- The previous start and final states are no longer!



Reducing a GNFA

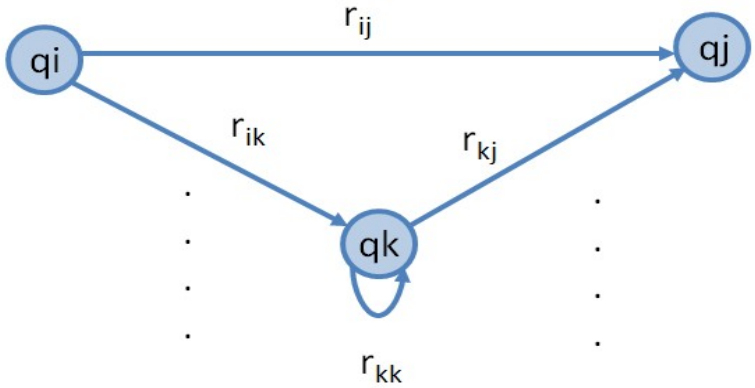
- We eliminate all states of the GNFA one-by-one leaving only the start state and the final state.



- When the GNFA is fully converted, the label of the only generalized transition is the regular expression for the language accepted by the original DFA.

Eliminating States

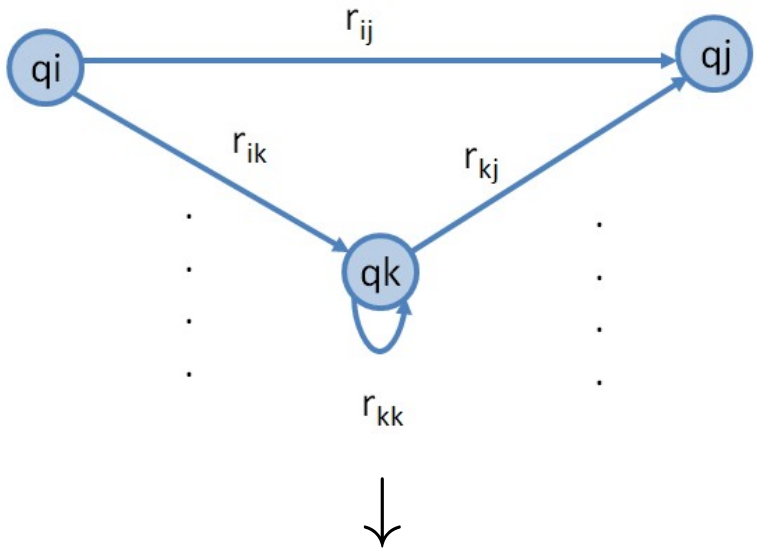
- Suppose we want to eliminate state q_k, and q_i and q_j are two of the remaining states ($i = j$ is possible).

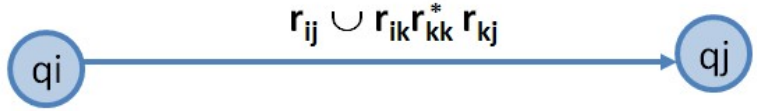


- How can we modify the transition label between q_i and q_j to reflect the fact that q_k will no longer be there?
  - There are two paths between q_i and q_j
    - Direct path with regular expression $\mathbf{r_{ij}}$
    - Path via q_k with the regular expression $\mathbf{r_{ik}r_{kk}^{*}r_{kj}}$

Eliminating States

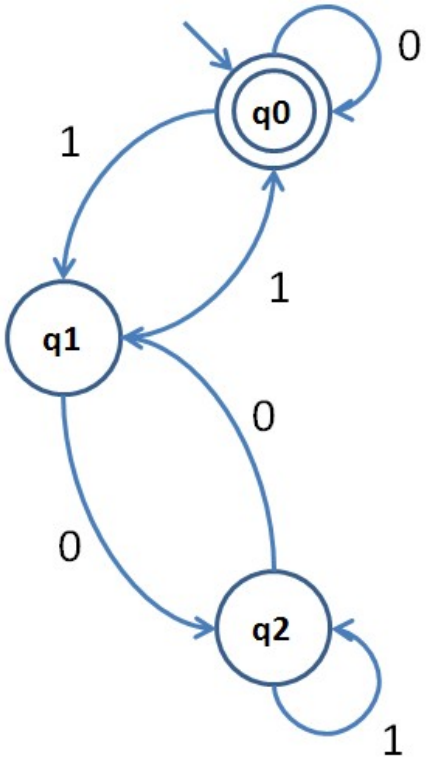
- There are two paths between q_i and q_j
  - Direct path with regular expression $\mathbf{r_{ij}}$
  - Path via q_k with the regular expression $\mathbf{r_{ik}r_{kk}^*r_{kj}}$
- After removing q_k, the new label would be $\mathbf{r'_{ij} = r_{ij} \cup r_{ik}r_{kk}^*r_{kj}}$



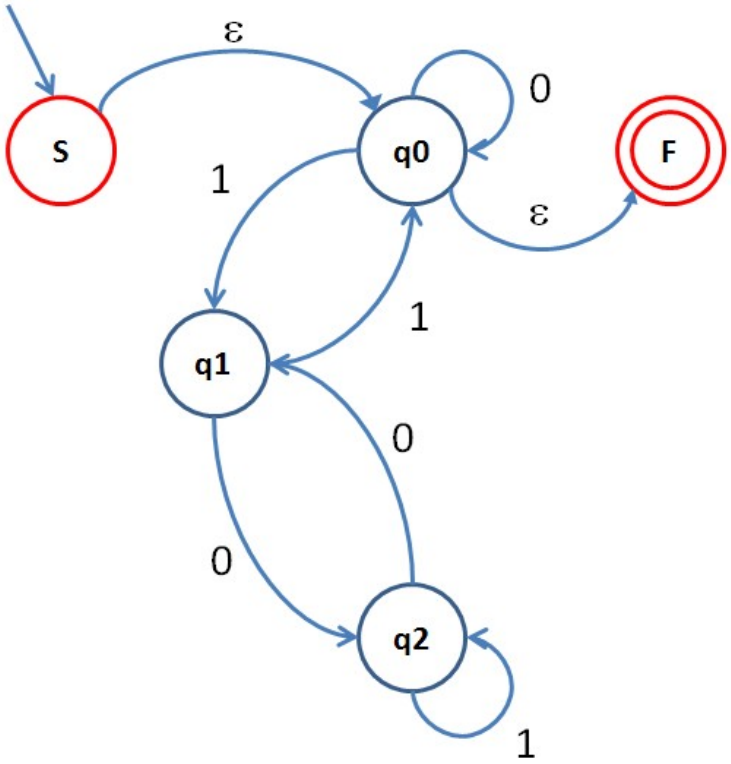


DFA-TO-GNFA-RE CONVERSION EXAMPLE

- DFA for binary numbers divisible by 3

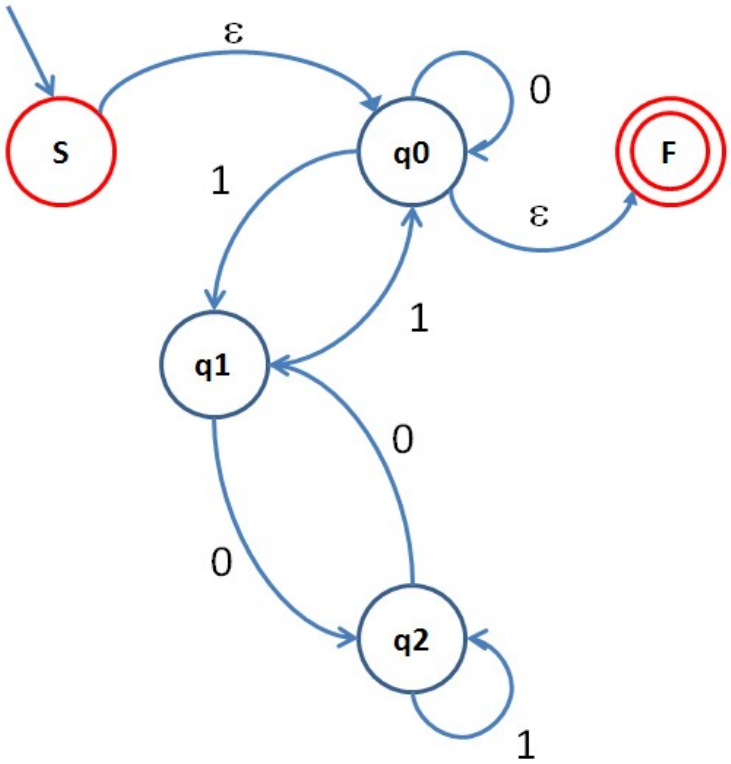


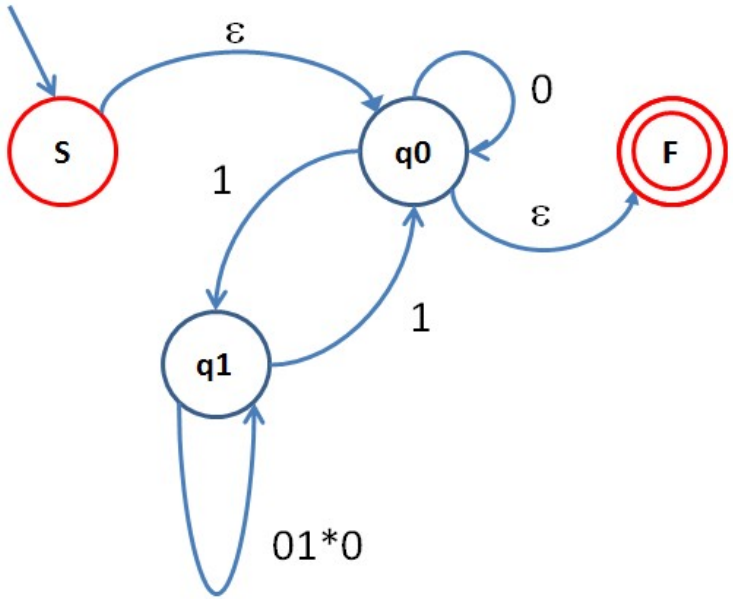
- Initial GNFA



DFA-TO-GNFA-RE CONVERSION EXAMPLE

- Let's eliminate q_2

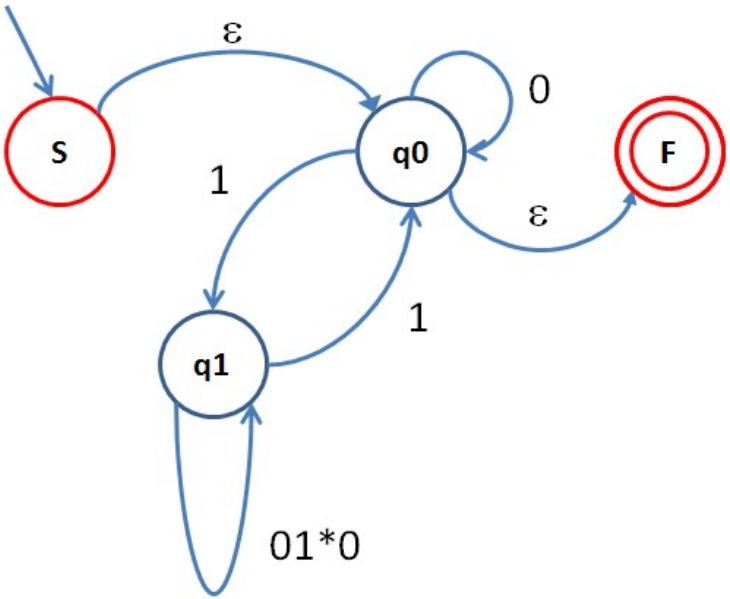


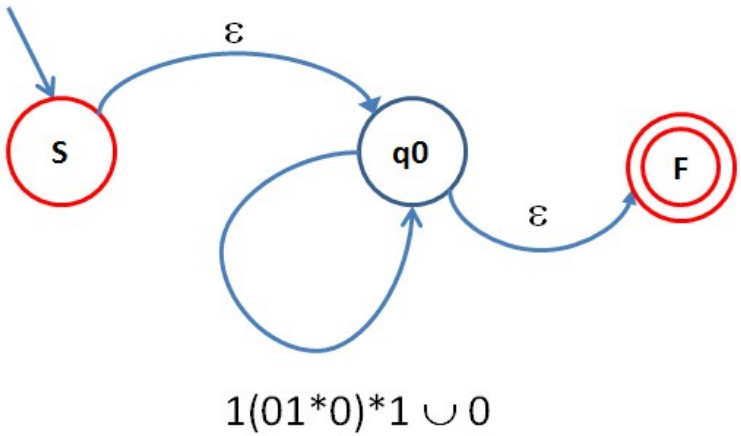


$q_i = q_1, q_j = q_1, q_k = q_2$

DFA-TO-GNFA-RE CONVERSION EXAMPLE

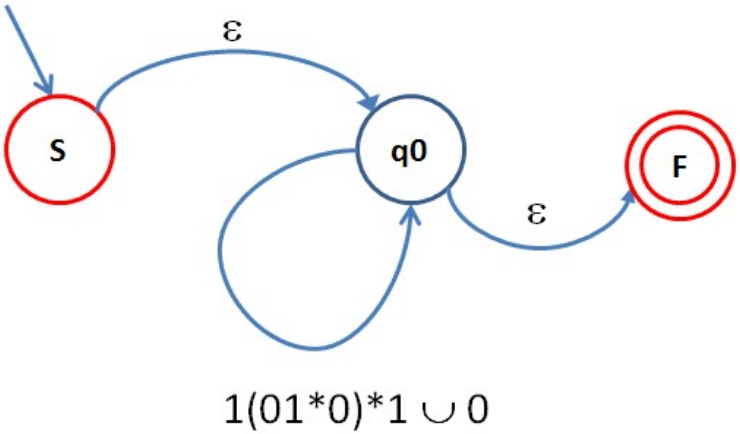
- Let's eliminate q_1

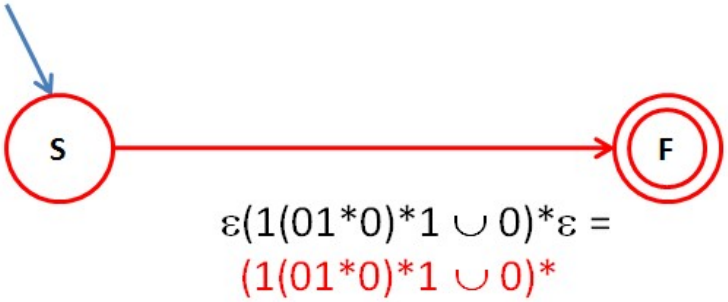




DFA-TO-GNFA-RE CONVERSION EXAMPLE

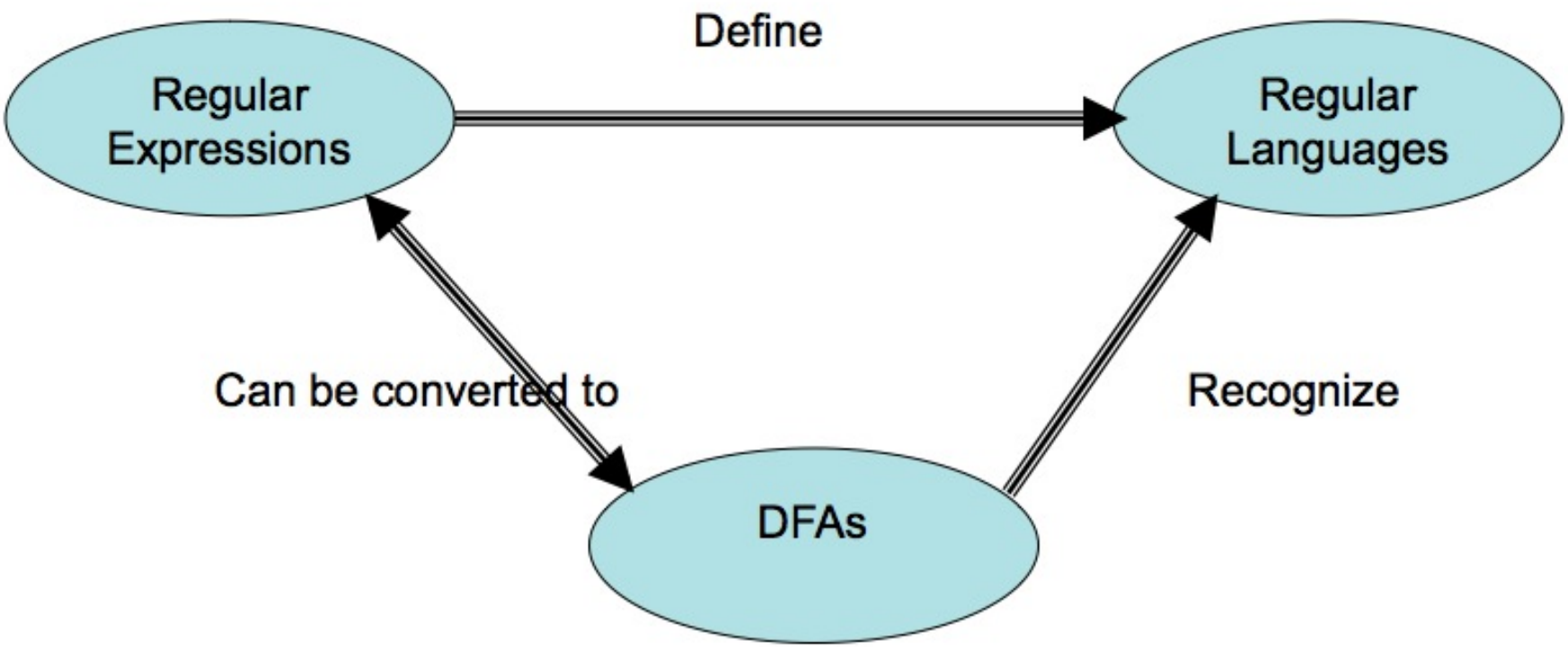
- Let's eliminate q_0



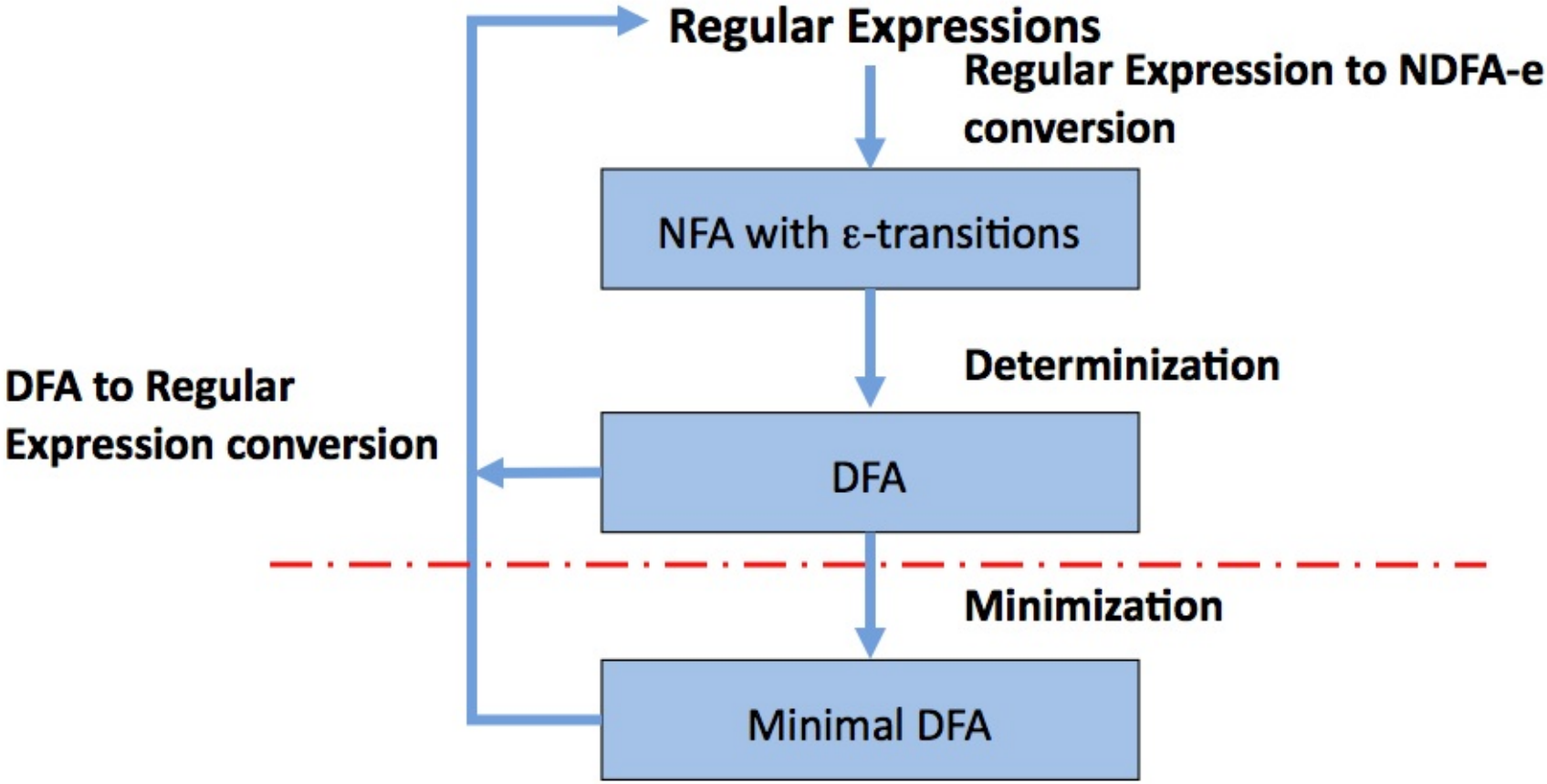


- So the regular expression we are looking for is $(\mathbf{1}(\mathbf{01}^*\mathbf{0})^*\mathbf{1} \cup \mathbf{0})^*$

The Story So Far

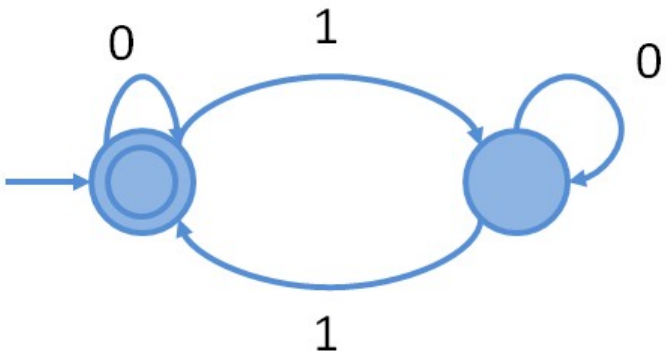


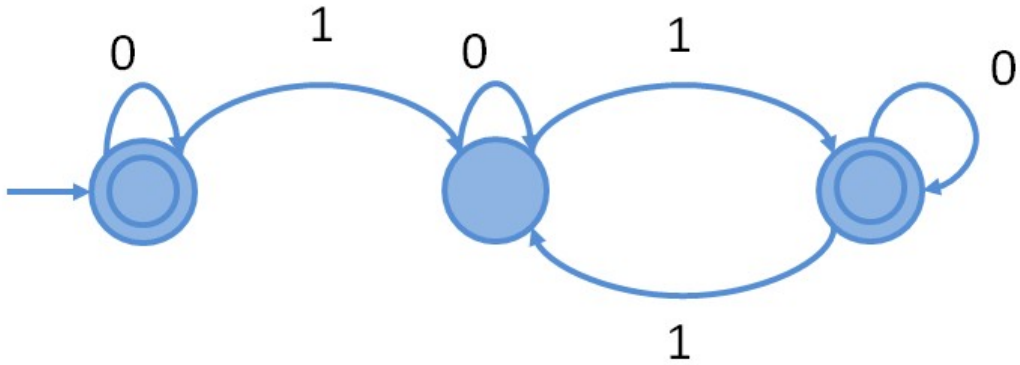
The Story So Far



DFA Minimization

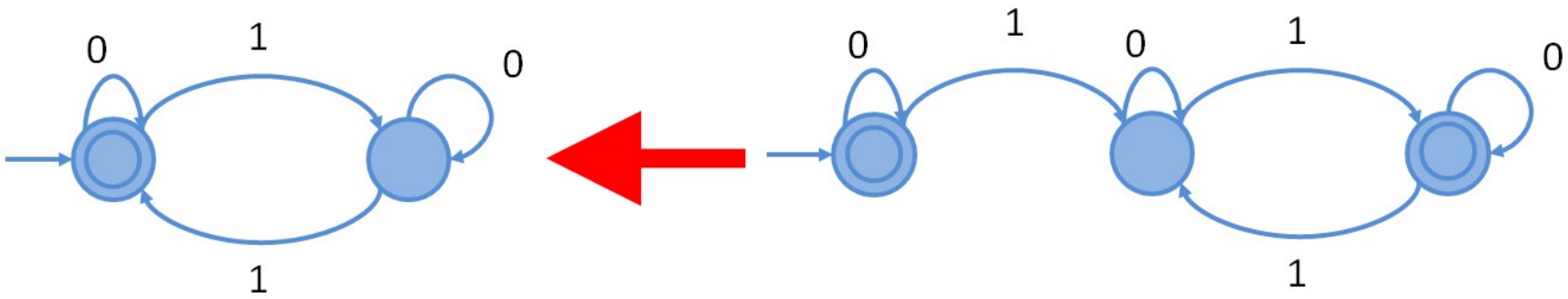
- Every DFA defines a unique language
- But in general, there may be many DFAs for a given language.
- These DFAs accept the same language.





DFA Minimization

- In practice, we are interested in the DFA with the minimal number of states
  - Use less memory
  - Use less hardware (flip-flops)



Indisguishable States

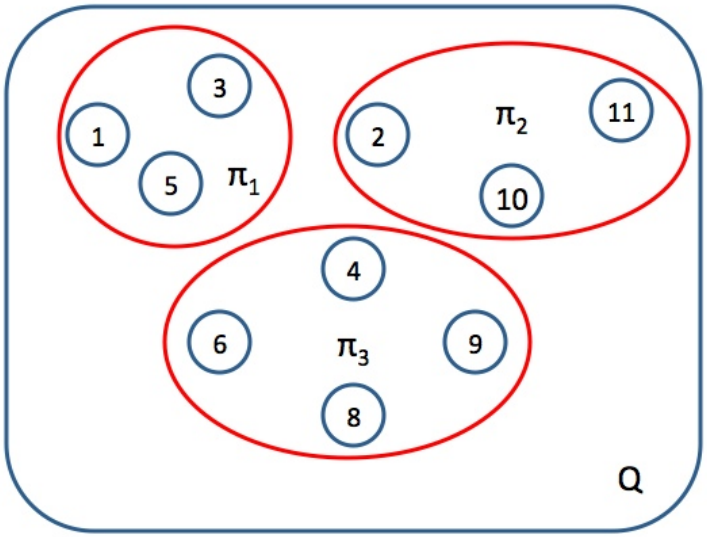
- Two states p and q of a DFA are called indistinguishable if for all $\omega \in \Sigma^*$,
  - $\delta^*(p, \omega) \in F \Leftrightarrow \delta^*(q, \omega) \in F$, and
  - $\delta^*(p, \omega) \notin F \Leftrightarrow \delta^*(q, \omega) \notin F$,
- Basically, these two states behave the same for all possible strings!
- Hence, a state p is distinguishable from state q
  - If there is at least one string ω such that either $\delta^*(p, \omega) \in F$ or $\delta^*(q, \omega) \in F$ and the other is not

INDISTINGUISHABILITY

- Indistinguishable states behave the same for all possible strings!
- So why have indistinguishable states? All but one can be eliminated!
- Indistinguishability is an equivalence relation
  - Reflexive: Each state is indistinguishable from itself
  - Symmetric: If p is indistinguishable from q, then q is indistinguishable from p
  - Transitive: If p is indistinguishable from q, and q is indistinguishable from r, then p is indistinguishable from r.

Indistinguishability and Partitions

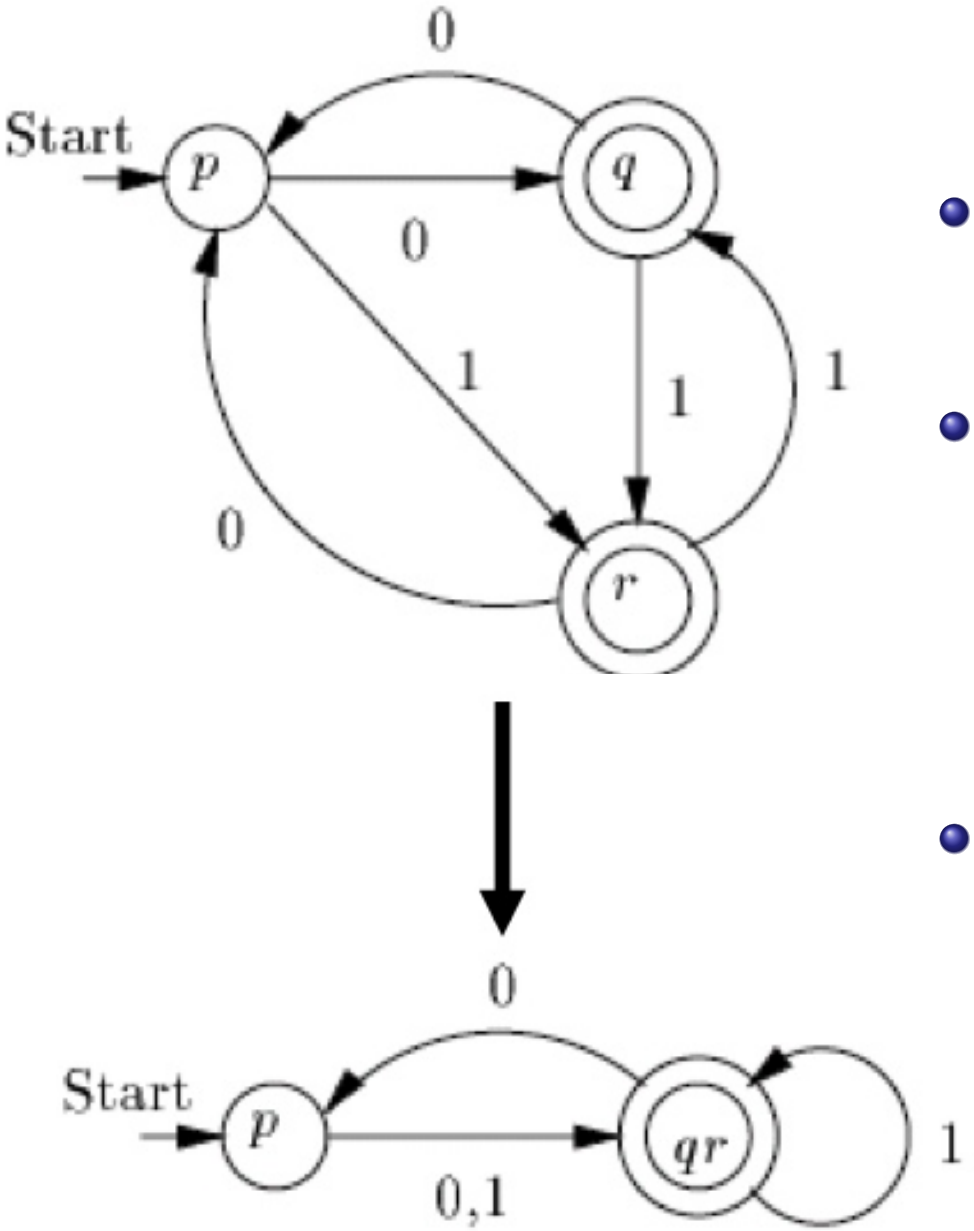
- Indistinguishability is an equivalence relation
  - Reflexive:Each state is indistinguishable from itself
  - Symmetric: If p is indistinguishable from q, then q is indistinguishable from p
  - Transitive: If p is indistinguishable from q, and q is indistinguishable from r, then p is indistinguishable from r.
- An equivalence relation on a set Q induces a partitioning $\pi = \{\pi_1, \pi_2, \cdots, \pi_k\}$ such that
  - For all i and j, $\pi_i \cap \pi_j = \Phi$,
  - $\bigcup_i \pi_i = Q$



Identifying Distinguishable States

- Basis: Any nonaccepting state is distinguishable from any accepting state ($\omega = \epsilon$).
- Induction: States p and q are distinguishable if there is some input symbol a such that $\delta(p, a)$ is distinguishable from $\delta(q, a)$.
- All other pairs of states are indistinguishable, and can be merged appropriately

Identifying Distinguishable States



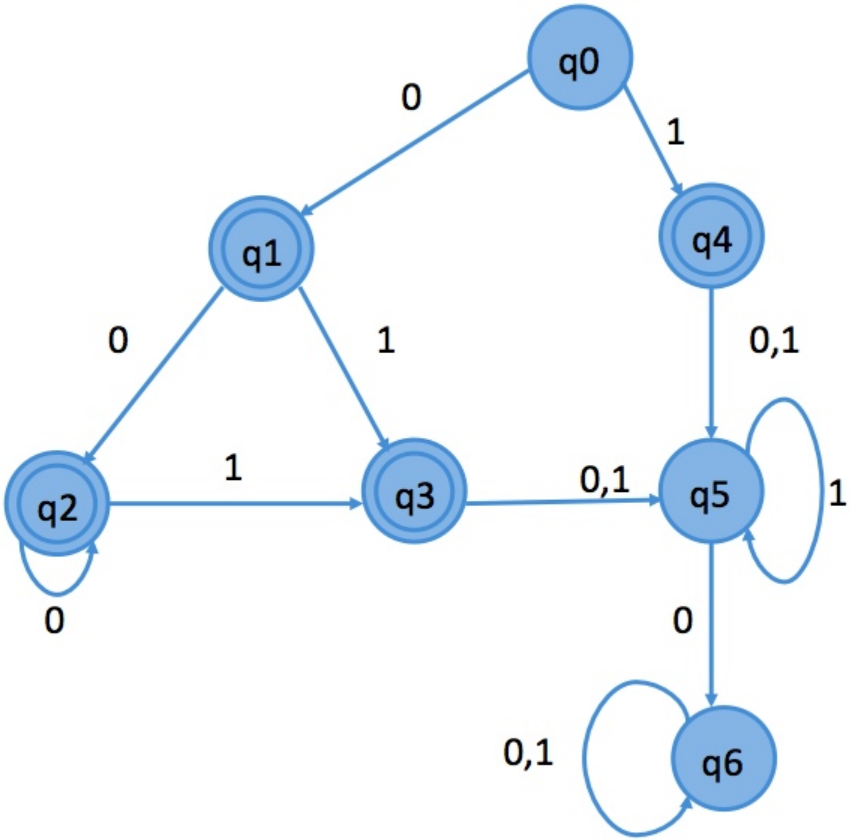
- p is distinguishable from q and r by basis
- Both q and r go to p with 0, so no string beginning with 0 will distinguish them
- Starting in either q and r, an input of 1 takes us to either, so they are indistinguishable.

Identifying Distinguishable States

The Procedure MARK

1. Remove all inaccessible states
2. Consider all pairs of states (p, q)
   - if $p \in F$ and $q \notin F$ or $p \notin F$ and $q \in F$, mark (p, q) as distinguishable
3. Repeat the following until no previously unmarked pairs are marked
   - $\forall p, q \in Q$ and $\forall a \in \Sigma$, find $\delta(p, a) = p'$ and $\delta(q, a) = q'$,
   - if (p', q') is marked distinguishable then mark (p, q) distinguishable.

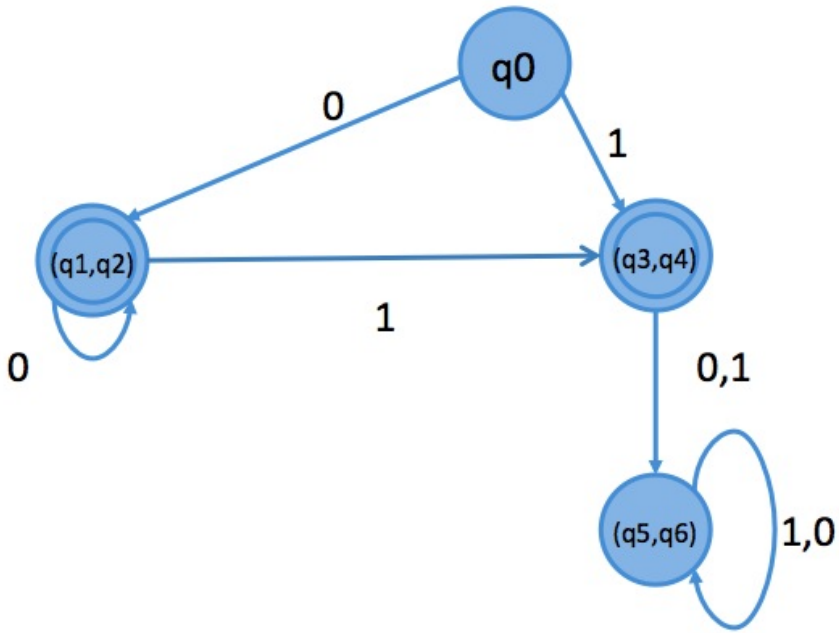
Minimization Example



- q_1 and q_2 are equivalent
- q_3 and q_4 are equivalent
- q_5 and q_6 are equivalent

q_1	×	■	■	■	■	■
q_2	×	✓	■	■	■	■
q_3	×	×	×	■	■	■
q_4	×	×	×	✓	■	■
q_5	×	×	×	×	×	■
q_6	×	×	×	×	×	✓
■	q_0	q_1	q_2	q_3	q_4	q_5

The Minimized DFA



- q_1 and q_2 are equivalent
- q_3 and q_4 are equivalent
- q_5 and q_6 are equivalent

q_1	×	■	■	■	■	■
q_2	×	✓	■	■	■	■
q_3	×	×	×	■	■	■
q_4	×	×	×	✓	■	■
q_5	×	×	×	×	×	■
q_6	×	×	×	×	×	✓
■	q_0	q_1	q_2	q_3	q_4	q_5

Is the minimized DFA really minimal?

- Let M be the DFA found by the previous procedure (with states $P = \{p_0, p_1, \ldots, p_m\}$)
- Suppose there is an equivalent DFA M_1 with δ_1 but with fewer states ($Q = \{q_0, q_1 \ldots, q_n\} n < m$).
- Since all states of M are distinguishable, there must be distinct strings, $\omega_1, \omega_2, \ldots, \omega_m$ such that $\delta^*(p_0, \omega_i) = p_i$ for all i.

Is the minimized DFA really minimal?

- Since M_1 has fewer states than M, then there must be strings ω_k and ω_l (among the previous $\omega_i's$) such that $\delta_1^*(q_0, \omega_k) = \delta_1^*(q_0, \omega_l)$ (Pigeonhole principle-see later)
- Since p_k and p_l are distinguishable, there must be some string x such that
  - $\delta^*(p_0, \omega_k \cdot x) = \delta^*(p_k, x)$ is a final state and $\delta^*(p_0, \omega_l \cdot x) = \delta^*(p_l, x)$ is NOT a final state, or vice versa. So $\omega_k \cdot x$ is accepted and $\omega_l \cdot x$ is not (or vice versa)

Is the minimized DFA really minimal?

- But

$$\begin{aligned}\delta_1^*(q_0, \omega_k \cdot x) &= \delta_1^*(\delta_1^*(q_0, \omega_k), x) \\ &= \delta_1^*(\delta_1^*(q_0, \omega_l), x) \\ &= \delta_1^*(q_0, \omega_l \cdot x)\end{aligned}$$

- So M_1 either accepts both $\omega_k \cdot x$ and $\omega_l \cdot x$ or rejects both. So M_1 and M can not be equivalent.
- So M_1 can not exist.

More on DFA Minimization

- DFA minimization is not covered in the textbook.
- See
  - `en.wikipedia.org/wiki/DFA_minimization`
  - Introduction to Automata Theory, Languages and Computation, by Hopcroft, Motwani and Ullman, Addison Wesley, 3rd edition, Section 4.4

  for more formal details.

Closure Properties of Regular Languages

- Regular languages are closed under
  - Union
  - Intersection
  - Difference
  - Concatenation
  - Star Closure
  - Complementation
  - Reversal

  operations

HOMOMORPHISM

- Suppose Σ and Γ are alphabets, the function $h : \Sigma \rightarrow \Gamma^*$ is called a homomorphism
- It is a substitution in which a single symbol $a \in \Sigma$ is replaced by a string $x \in \Gamma^*$, that is. $h(a) = x$
- Extend to strings: $h(\omega) = h(a_1) \ldots, h(a_n)$ where $\omega \in \Sigma^*$ and $a_i \in \Sigma$
- Extend to languages $h(L) = \{h(\omega) | \omega \in L\}$
  - $h(L)$ is called the homomorphic image of L.

Homomorphism Example

- Let $\Sigma = \{a, b\}$ and $\Gamma = \{a, b, c\}$
  - $h(a) = ab$ and $h(b) = bbc$
  - $h(aba) = abbbcab$

Theorem

Let h be a homomorphism. If L is regular then $h(L)$ is also regular.

Proof

Obvious: Modify the DFA transitions

Decision Properties of Regular Languages

Theorem

Given a standard representation (DFA, NFA, RE) of any regular language L on Σ and any ω in Σ^*, there exists an algorithm to determine if ω is in L or not.

Proof.

Represent the language with a DFA and test if ω is accepted or not $\square$

Decision Properties of Regular Languages

Theorem

There exist algorithms for determining whether a regular language in standard representation is empty or not.

Proof.

Represent the language with a DFA. If there is a path from the start state to some final state, the language is not empty. □

Decision Properties of Regular Languages

Theorem

There exist algorithms for determining whether a regular language in standard representation is finite or infinite.

Proof.

Find all states that form a cycle. If any of these are on path from the start state to a final state, then the language is infinite. □

Proof.

If DFA with n states accepts some string of length between n and $2n - 1$ then it accepts an infinite set of strings.(needs Pumping Lemma) □

Decision Properties of Regular Languages

Theorem

Given standard representations of two regular languages L_1 and L_2, there exists an algorithm to determine if $L_1 = L_2$.

Proof.

Compute $L_3 = (L_1 - L_2) \cup (L_2 - L_1)$ which has to be regular. If $L_3 = \Phi$ then $L_1 = L_2$. □

More Decision Problems

- To decide if $L_1 \subseteq L_2$, check if $L_1 - L_2 = \Phi$
- To decide if $\epsilon \in L$, check if $q_0 \in F$
- To decide if L contains ω such that $\omega = \omega^R$
  - Let M be the DFA for L. Construct M^R.
  - Construct $M \cap M^R$ using the cross-product construction
  - Check if $L(M \cap M^R) \neq \Phi$.

Formal Languages, Automata and Computation

Identifying Nonregular languages

Pumping Lemma

SUMMARY

- DFAs to Regular Expressions

Summary

- DFAs to Regular Expressions
- Minimizing DFA's

Summary

- DFAs to Regular Expressions
- Minimizing DFA's
- Closure Properties

Summary

- DFAs to Regular Expressions
- Minimizing DFA's
- Closure Properties
- Decision Properties

IDENTIFYING NONREGULAR LANGUAGES

- Given language L how can we check if it is not a regular language ?

IDENTIFYING NONREGULAR LANGUAGES

- Given language L how can we check if it is not a regular language ?
  - The answer is not obvious.

IDENTIFYING NONREGULAR LANGUAGES

- Given language L how can we check if it is not a regular language ?
  - The answer is not obvious.
  - Not being able to design a DFA does not constitute a proof!

The Pigeonhole Principle

- If there are n pigeons and m holes and $n > m$, then at least one hole has > 1 pigeons.

The Pigeonhole Principle

- If there are n pigeons and m holes and $n > m$, then at least one hole has > 1 pigeons.

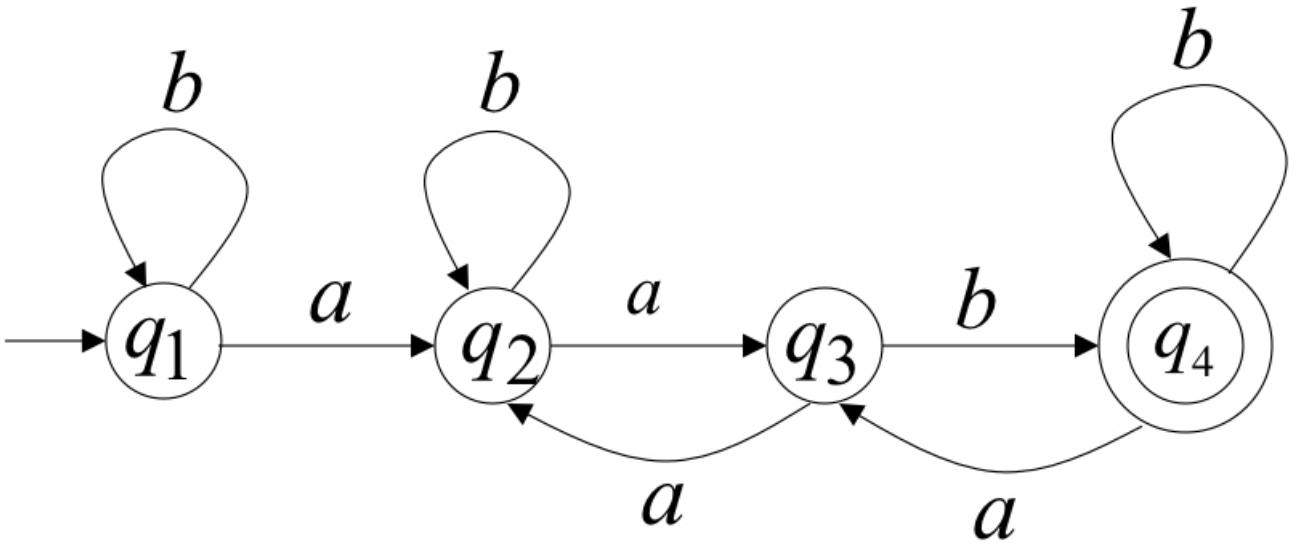
The Pigeonhole Principle

- If there are n pigeons and m holes and $n > m$, then at least one hole has > 1 pigeons.

- What do pigeons have to do with regular languages?

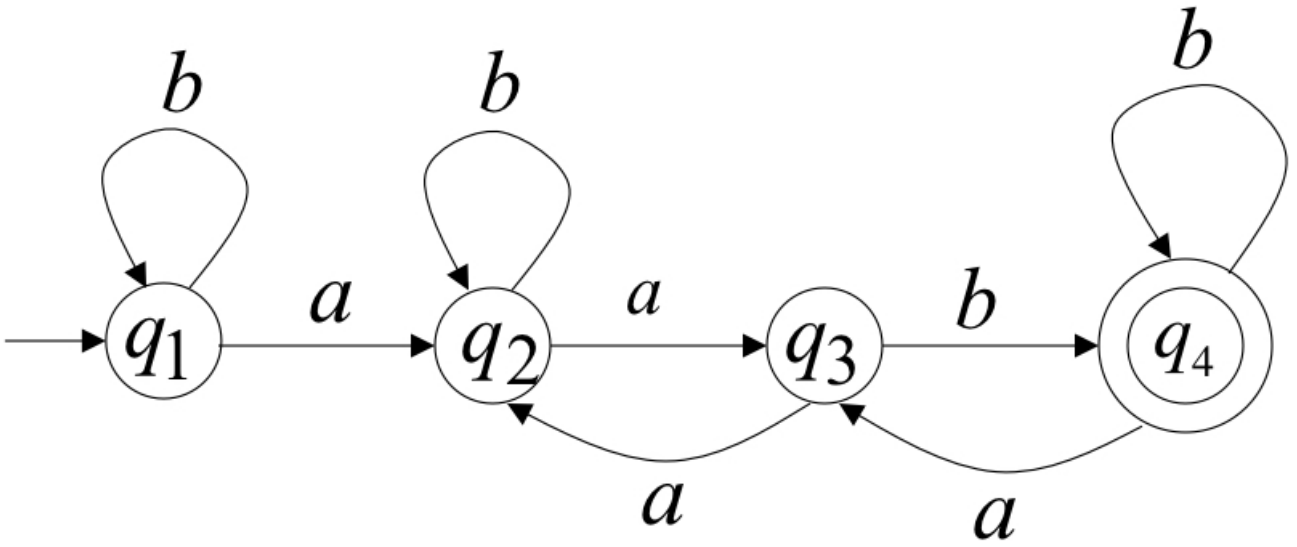
THE PIGEONHOLE PRINCIPLE

- Consider the DFA



THE PIGEONHOLE PRINCIPLE

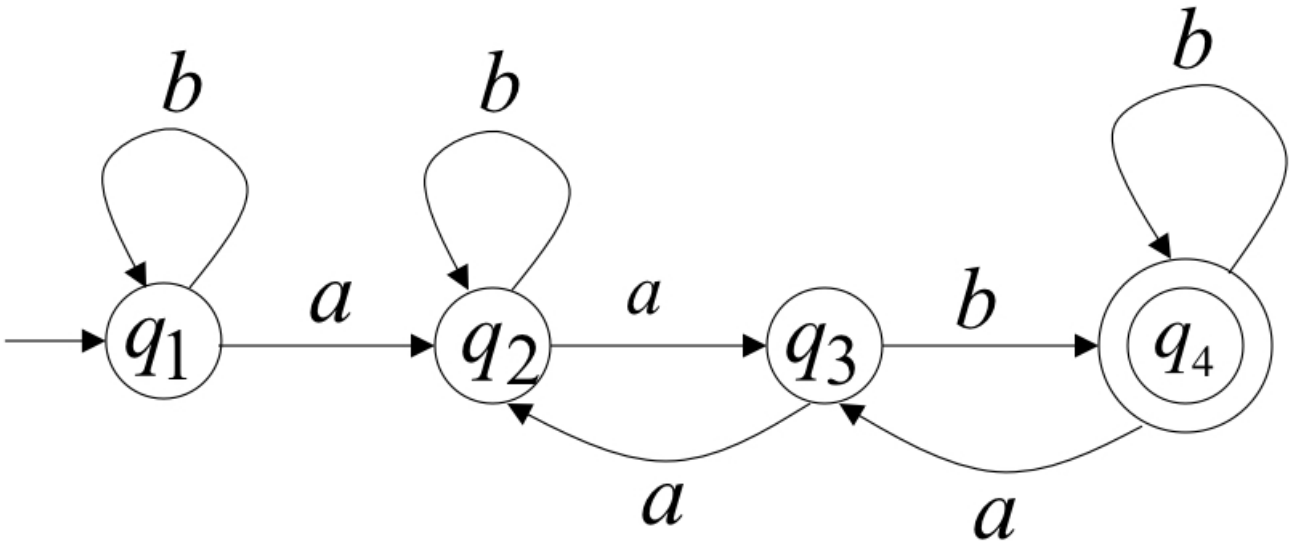
- Consider the DFA



- With strings *a*, *aa* or *aab*, no state is repeated

THE PIGEONHOLE PRINCIPLE

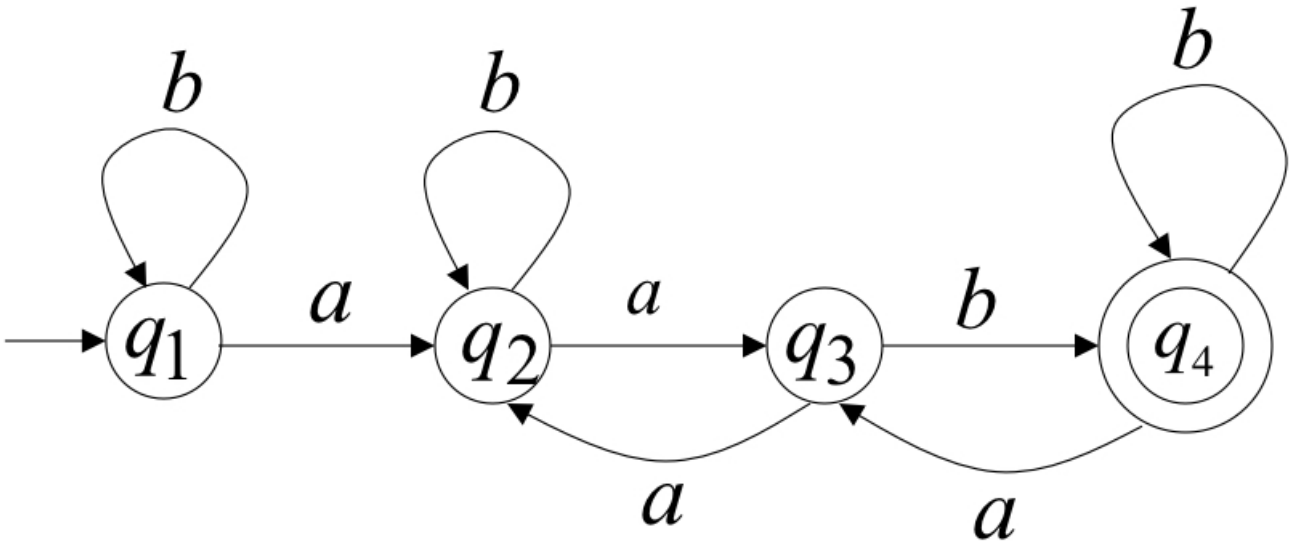
- Consider the DFA



- With strings *a*, *aa* or *aab*, no state is repeated
- With strings *aabb*, *bbaa*, *abbabb* or *abbbabbabb*, a state is repeated

THE PIGEONHOLE PRINCIPLE

- Consider the DFA



- With strings *a*, *aa* or *aab*, no state is repeated
- With strings *aabb*, *bbaa*, *abbabb* or *abbbabbabb*, a state is repeated
- In fact, for any ω where $|\omega| \geq 4$, some state has to repeat? Why?

The Pigeonhole Principle

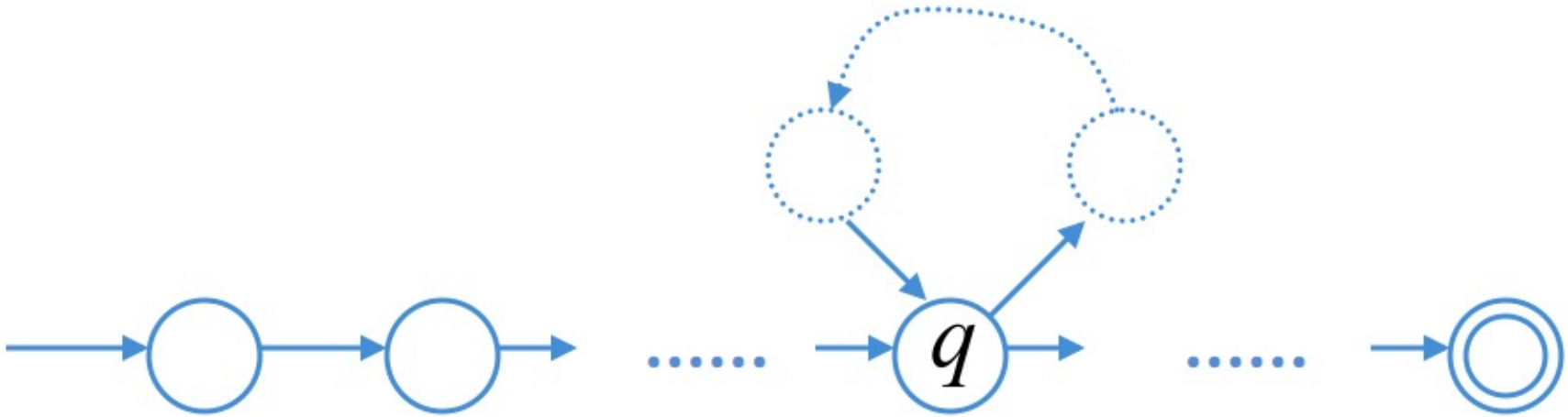
- When traversing the DFA with the string ω, if the number of transitions $\geq$ number of states, some state q has to repeat!

The Pigeonhole Principle

- When traversing the DFA with the string ω, if the number of transitions $\geq$ number of states, some state q has to repeat!
- Transitions are pigeons, states are holes.

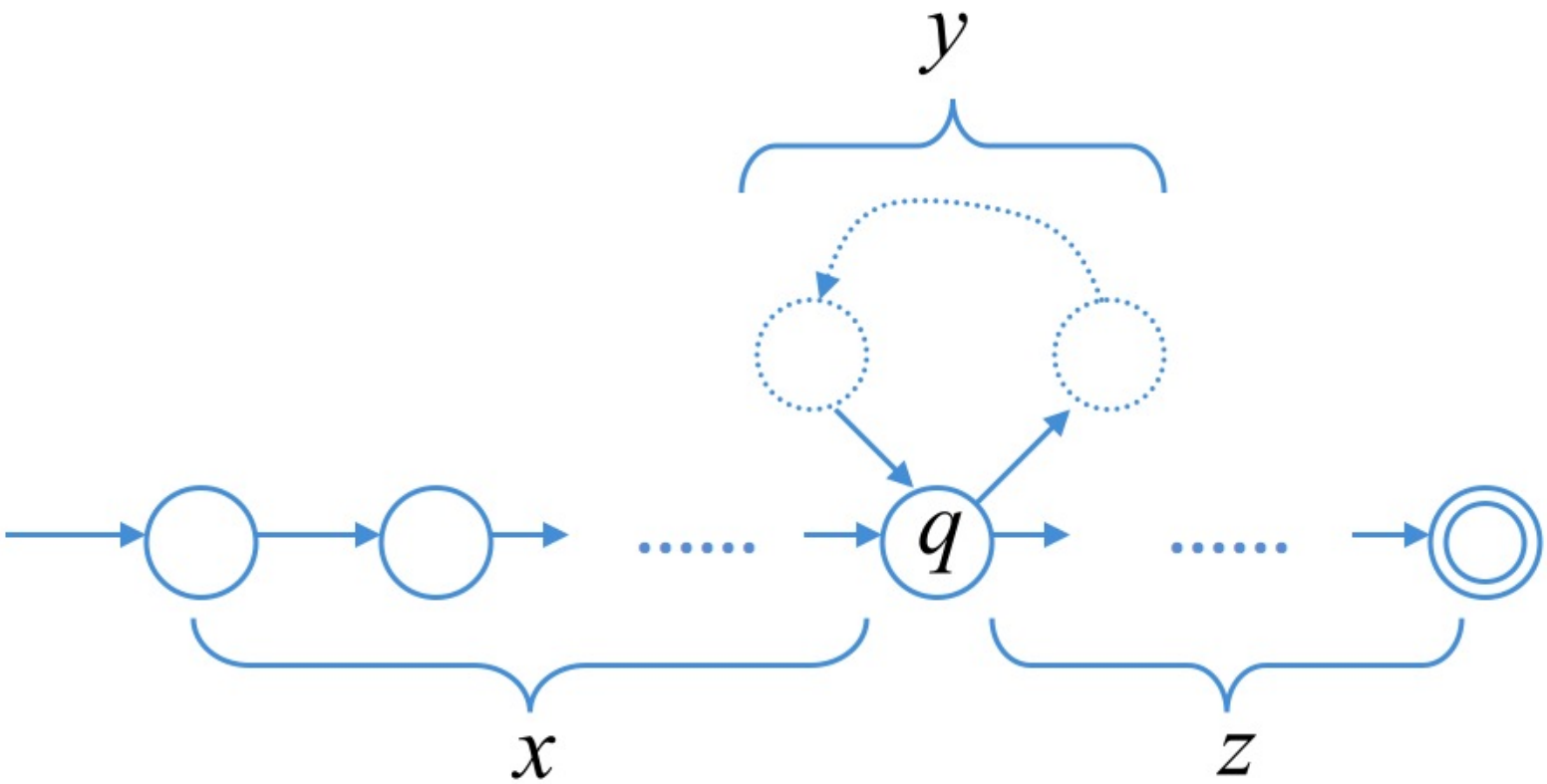
THE PIGEONHOLE PRINCIPLE

- When traversing the DFA with the string ω, if the number of transitions $\geq$ number of states, some state q has to repeat!
- Transitions are pigeons, states are holes.



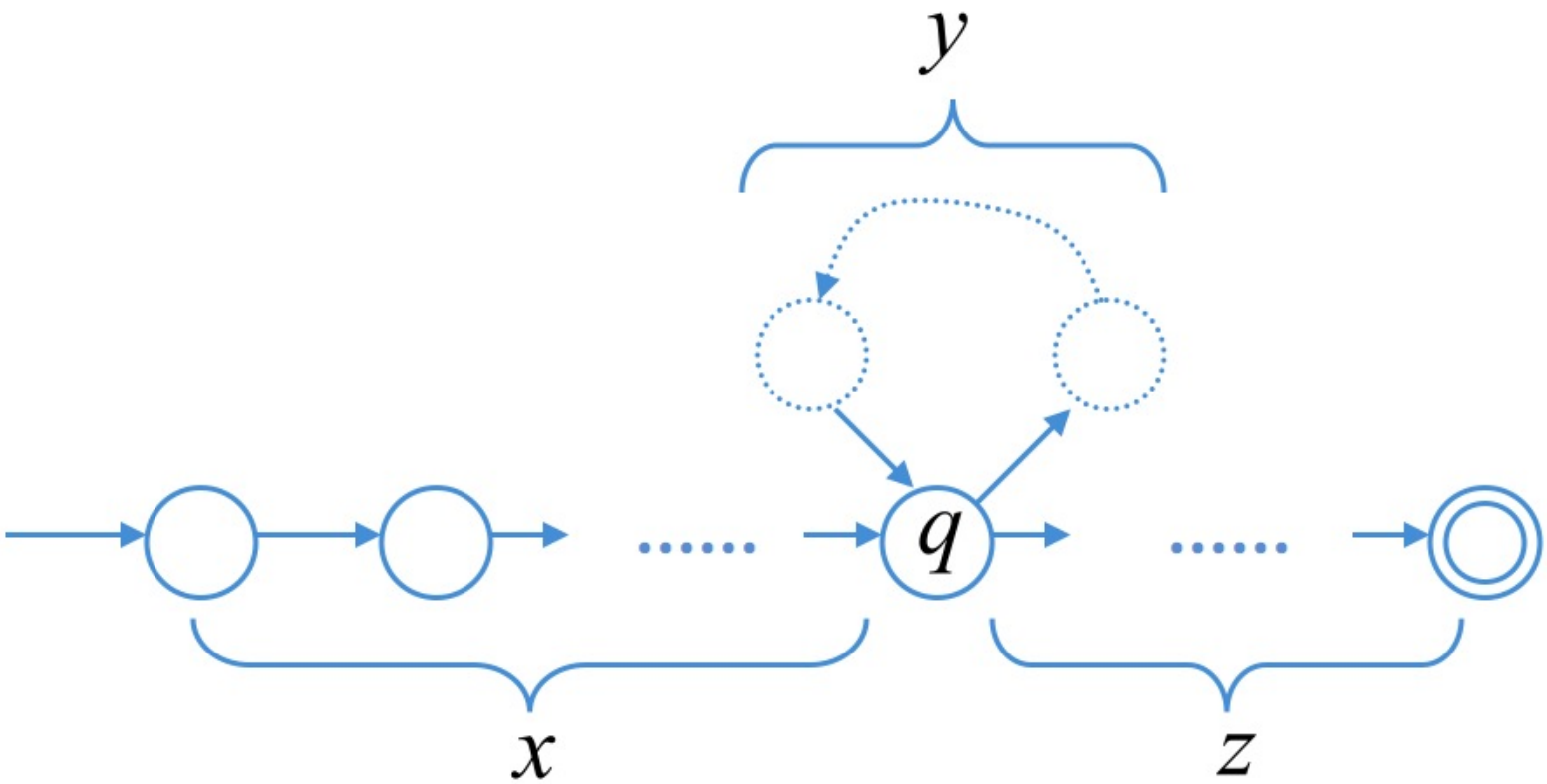
PUMPING A STRING

- Consider a string $\omega = xyz$



Pumping a String

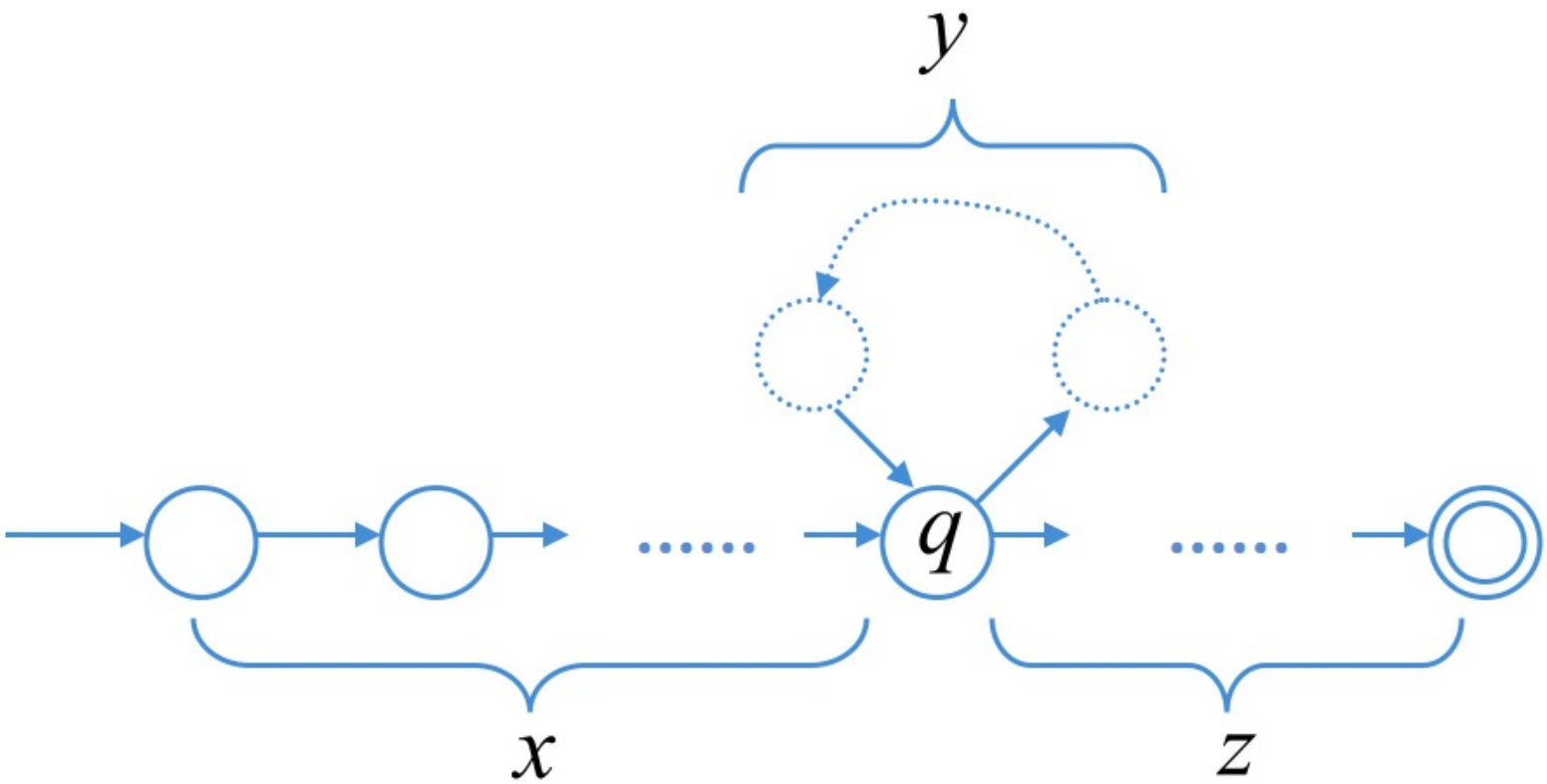
- Consider a string $\omega = xyz$



- $|y| \geq 1$

Pumping a String

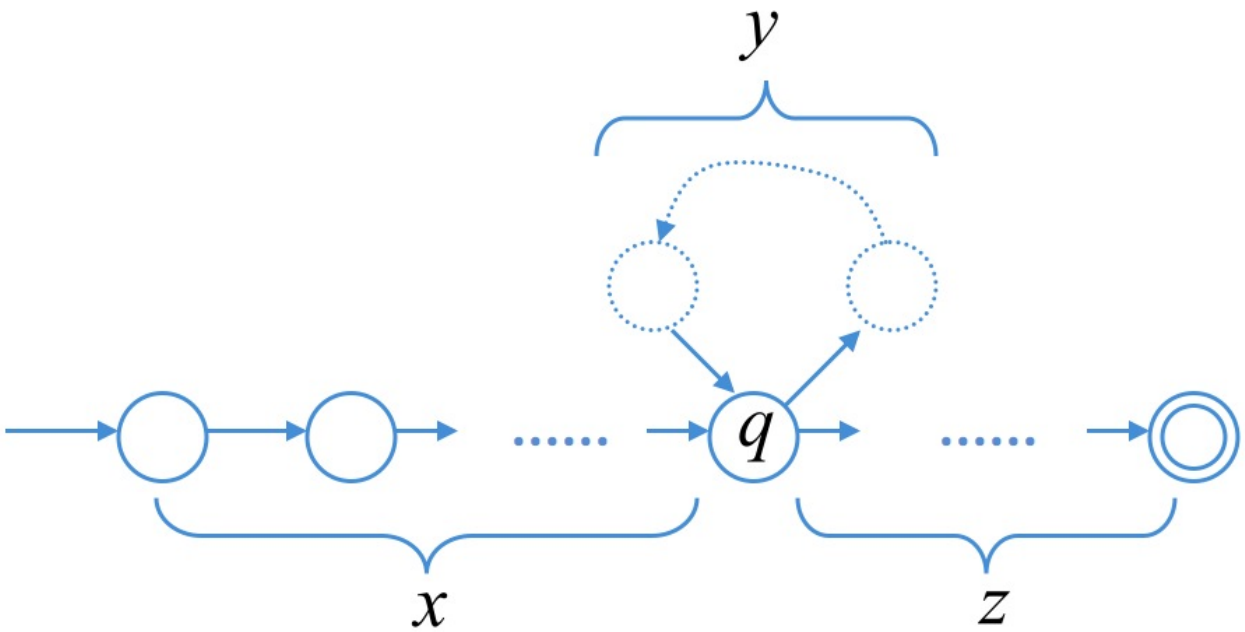
- Consider a string $\omega = xyz$



- $|y| \geq 1$
- $|xy| \leq m$ (m the number of states)

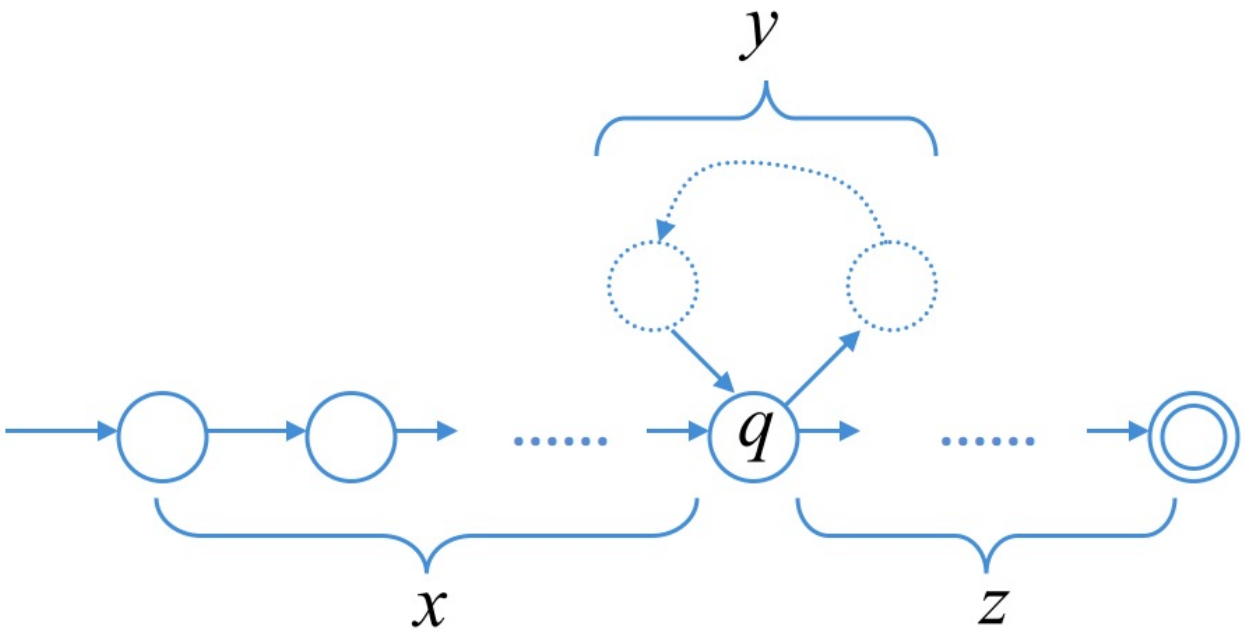
PUMPING A STRING

- Consider a string $\omega = xyz$



Pumping a String

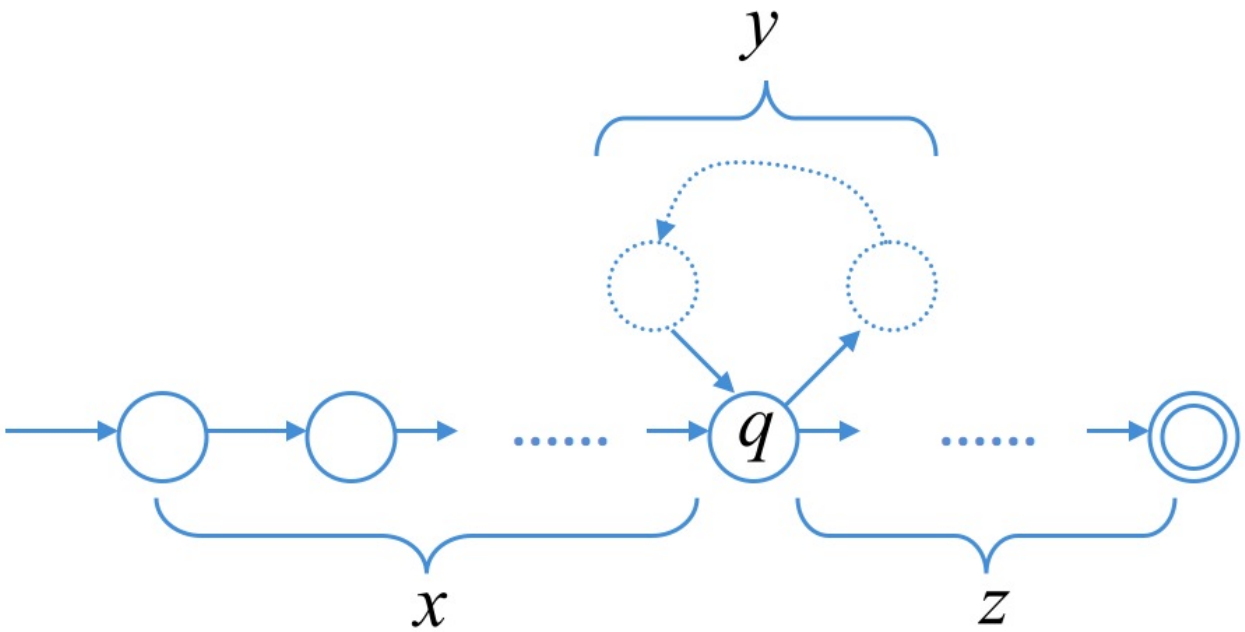
- Consider a string $\omega = xyz$



- If $\omega = xyz \in L$ that so are xy^iz for all $i \geq 0$

PUMPING A STRING

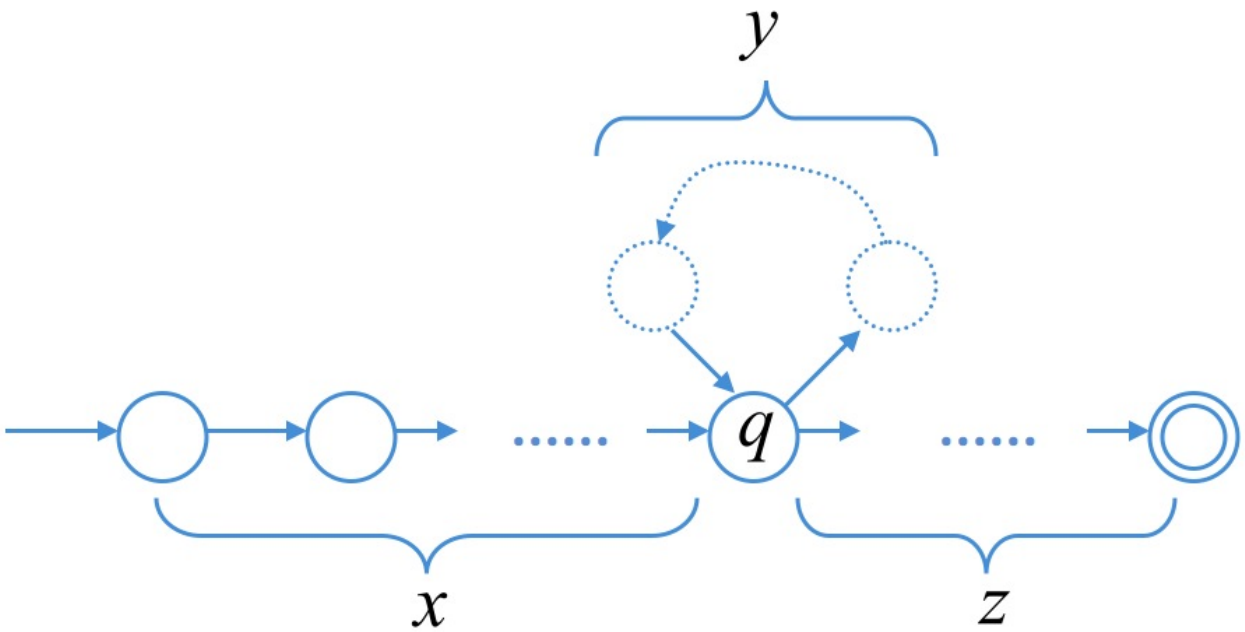
- Consider a string $\omega = xyz$



- If $\omega = xyz \in L$ that so are xy^iz for all $i \geq 0$
- The substring y can be pumped.

Pumping a String

- Consider a string $\omega = xyz$



- If $\omega = xyz \in L$ that so are xy^iz for all $i \geq 0$
- The substring y can be pumped.
- So if a DFA accepts a sufficiently long string, then it accepts an infinite number of strings!

A Nonregular Language

- Consider the language $L = \{a^n b^n | n \geq 0\}$

A Nonregular Language

- Consider the language $L = \{a^n b^n | n \geq 0\}$
- Suppose L is regular and a DFA with p states accepts L

A Nonregular Language

- Consider the language $L = \{a^n b^n | n \geq 0\}$
- Suppose L is regular and a DFA with p states accepts L
- Consider $\delta^*(q_0, a^i)$ for $i = 0, 1, 2, \ldots$

A Nonregular Language

- Consider the language $L = \{a^n b^n | n \geq 0\}$
- Suppose L is regular and a DFA with p states accepts L
- Consider $\delta^*(q_0, a^i)$ for $i = 0, 1, 2, \ldots$
- Since there are infinite i's, but a finite number states, the Pigeonhole Principle tells us that there is some state q such that
  - $\delta^*(q_0, a^n) = q$ and $\delta^*(q_0, a^m) = q$, but $n \neq m$

A Nonregular Language

- Consider the language $L = \{a^n b^n | n \geq 0\}$
- Suppose L is regular and a DFA with p states accepts L
- Consider $\delta^*(q_0, a^i)$ for $i = 0, 1, 2, \ldots$
- Since there are infinite i's, but a finite number states, the Pigeonhole Principle tells us that there is some state q such that
  - $\delta^*(q_0, a^n) = q$ and $\delta^*(q_0, a^m) = q$, but $n \neq m$
  - Thus if M accepts $a^n b^n$ it must also accept $a^m b^n$, since in state q is does not "remember" if there were n or m a's.

A Nonregular Language

- Consider the language $L = \{a^n b^n | n \geq 0\}$
- Suppose L is regular and a DFA with p states accepts L
- Consider $\delta^*(q_0, a^i)$ for $i = 0, 1, 2, \ldots$
- Since there are infinite i's, but a finite number states, the Pigeonhole Principle tells us that there is some state q such that
  - $\delta^*(q_0, a^n) = q$ and $\delta^*(q_0, a^m) = q$, but $n \neq m$
  - Thus if M accepts $a^n b^n$ it must also accept $a^m b^n$, since in state q is does not "remember" if there were n or m a's.
- Thus M can not exist and L is not regular.

The Pumping Lemma

Lemma

Given an infinite regular language L

The Pumping Lemma

Lemma

Given an infinite regular language L

1. *There exists an integer m such that*

The Pumping Lemma

Lemma

Given an infinite regular language L

1. *There exists an integer m such that*
2. *for any string $\omega \in L$ with length $|\omega| \geq m$,*

The Pumping Lemma

Lemma

Given an infinite regular language L

1. *There exists an integer m such that*
2. *for any string $\omega \in L$ with length $|\omega| \geq m$,*
3. *we can write $\omega = xyz$ with $|y| \geq 1$ and $|xy| \leq m$,*

The Pumping Lemma

Lemma

Given an infinite regular language L

1. *There exists an integer m such that*
2. *for any string $\omega \in L$ with length $|\omega| \geq m$,*
3. *we can write $\omega = xyz$ with $|y| \geq 1$ and $|xy| \leq m$,*
4. *such that the strings $xy^i z$ for $i = 0, 1, 2 \ldots$ are also in L*

The Pumping Lemma

Lemma

Given an infinite regular language L

1. *There exists an integer m such that*
2. *for any string $\omega \in L$ with length $|\omega| \geq m$,*
3. *we can write $\omega = xyz$ with $|y| \geq 1$ and $|xy| \leq m$,*
4. *such that the strings $xy^i z$ for $i = 0, 1, 2 \ldots$ are also in L*

Thus any sufficiently long string can be "pumped."

The Pumping Lemma

Lemma

Given an infinite regular language L

1. *There exists an integer m such that*
2. *for any string $\omega \in L$ with length $|\omega| \geq m$,*
3. *we can write $\omega = xyz$ with $|y| \geq 1$ and $|xy| \leq m$,*
4. *such that the strings $xy^i z$ for $i = 0, 1, 2 \ldots$ are also in L*

Thus any sufficiently long string can be "pumped."

Proof Idea

We already have some hints.

The Pumping Lemma

Proof.

- If L is regular then M with p states recognizes L. Take a string $s = s_1 s_2 \cdots s_n \in L$ with $n \geq p$.

The Pumping Lemma

Proof.

- If L is regular then M with p states recognizes L. Take a string $s = s_1 s_2 \cdots s_n \in L$ with $n \geq p$.
- Let $r_1 r_2 \cdots r_{n+1}$ be the sequence of $n+1 (\geq p+1)$ states M enters while processing s ($r_{i+1} = \delta(r_i, s_i)$)

The Pumping Lemma

Proof.

- If L is regular then M with p states recognizes L. Take a string $s = s_1 s_2 \cdots s_n \in L$ with $n \geq p$.
- Let $r_1 r_2 \cdots r_{n+1}$ be the sequence of $n+1 (\geq p+1)$ states M enters while processing s ($r_{i+1} = \delta(r_i, s_i)$)
- r_j and r_l (for some j and l ($j < l \leq p+1$) should be the same state (Pigeons!)

The Pumping Lemma

Proof.

- If L is regular then M with p states recognizes L. Take a string $s = s_1 s_2 \cdots s_n \in L$ with $n \geq p$.
- Let $r_1 r_2 \cdots r_{n+1}$ be the sequence of $n+1 (\geq p+1)$ states M enters while processing s ($r_{i+1} = \delta(r_i, s_i)$)
- r_j and r_l (for some j and l ($j < l \leq p+1$) should be the same state (Pigeons!)
- Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, and $z = s_l \cdots s_n$.

The Pumping Lemma

Proof.

- If L is regular then M with p states recognizes L. Take a string $s = s_1 s_2 \cdots s_n \in L$ with $n \geq p$.
- Let $r_1 r_2 \cdots r_{n+1}$ be the sequence of $n+1 (\geq p+1)$ states M enters while processing s ($r_{i+1} = \delta(r_i, s_i)$)
- r_j and r_l (for some j and l ($j < l \leq p+1$) should be the same state (Pigeons!)
- Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, and $z = s_l \cdots s_n$.
- x takes M from r_1 to r_j, y takes M from r_j to r_j, and z takes M from r_j to r_{n+1}, which is an accepting state. So M must also accept $xy^i z$ for $i \geq 0$.

The Pumping Lemma

Proof.

- If L is regular then M with p states recognizes L. Take a string $s = s_1 s_2 \cdots s_n \in L$ with $n \geq p$.
- Let $r_1 r_2 \cdots r_{n+1}$ be the sequence of $n+1 (\geq p+1)$ states M enters while processing s ($r_{i+1} = \delta(r_i, s_i)$)
- r_j and r_l (for some j and l ($j < l \leq p+1$) should be the same state (Pigeons!)
- Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, and $z = s_l \cdots s_n$.
- x takes M from r_1 to r_j, y takes M from r_j to r_j, and z takes M from r_j to r_{n+1}, which is an accepting state. So M must also accept $xy^i z$ for $i \geq 0$.
- We know $j \neq l$, so $|y| > 0$ and $l \leq p+1$ so $|xy| \leq p$

Using the Pumping Lemma

- If a language violates the pumping lemma, then it can not be regular.

Using the Pumping Lemma

- If a language violates the pumping lemma, then it can not be regular.
- Two Player Proof Strategy:

Using the Pumping Lemma

- If a language violates the pumping lemma, then it can not be regular.
- Two Player Proof Strategy:
  - Opponent picks m

Using the Pumping Lemma

- If a language violates the pumping lemma, then it can not be regular.
- Two Player Proof Strategy:
  - Opponent picks m
  - Given m, we pick ω in L such that $|\omega| \geq m$. We are free to choose ω as we please, as long as those conditions are satisfied.

Using the Pumping Lemma

- If a language violates the pumping lemma, then it can not be regular.
- Two Player Proof Strategy:
  - Opponent picks m
  - Given m, we pick ω in L such that $|\omega| \geq m$. We are free to choose ω as we please, as long as those conditions are satisfied.
  - Opponent picks $\omega = xyz$ - the decomposition subject to $|xy| \leq m$ and $|y| \geq 1$.

Using the Pumping Lemma

- If a language violates the pumping lemma, then it can not be regular.
- Two Player Proof Strategy:
  - Opponent picks m
  - Given m, we pick ω in L such that $|\omega| \geq m$. We are free to choose ω as we please, as long as those conditions are satisfied.
  - Opponent picks $\omega = xyz$ - the decomposition subject to $|xy| \leq m$ and $|y| \geq 1$.
  - We try to pick an i such that $xy^i z \notin L$

Using the Pumping Lemma

- If a language violates the pumping lemma, then it can not be regular.
- Two Player Proof Strategy:
  - Opponent picks m
  - Given m, we pick ω in L such that $|\omega| \geq m$. We are free to choose ω as we please, as long as those conditions are satisfied.
  - Opponent picks $\omega = xyz$ - the decomposition subject to $|xy| \leq m$ and $|y| \geq 1$.
  - We try to pick an i such that $xy^i z \notin L$
  - If for all possible decompositions the opponent can pick, we can find an i, then L is not regular.

Using the Pumping Lemma

Consider $L = \{a^n b^n | n \geq 0\}$

Using the Pumping Lemma

Consider $L = \{a^n b^n | n \geq 0\}$

1. Opponent picks m

Using the Pumping Lemma

Consider $L = \{a^n b^n | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = a^m b^m$. Clearly $|\omega| \geq m$.

Using the Pumping Lemma

Consider $L = \{a^n b^n | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = a^m b^m$. Clearly $|\omega| \geq m$.
3. Since the first m symbols are all a's, the opponent is forced to pick $x = a^j$, $y = a^k$ and $z = a^l b^m$, with $j + k \leq m$ and $l \geq 0$ and $j + k + l = m$

$$\omega = \underbrace{a \cdots a}_{x} \underbrace{a \cdots a}_{y} \underbrace{a \ldots a b \cdots b}_{z}$$

Using the Pumping Lemma

Consider $L = \{a^n b^n | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = a^m b^m$. Clearly $|\omega| \geq m$.
3. Since the first m symbols are all a's, the opponent is forced to pick $x = a^j$, $y = a^k$ and $z = a^l b^m$, with $j + k \leq m$ and $l \geq 0$ and $j + k + l = m$

$$\omega = \underbrace{a \cdots a}_{x} \underbrace{a \cdots a}_{y} \underbrace{a \ldots a b \cdots b}_{z}$$

4. We choose $i = 2$ which means $a^j a^k a^k a^l b^m = a^{m+k} b^m \in L$ but it can not be!

Using the Pumping Lemma

Consider $L = \{a^n b^n | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = a^m b^m$. Clearly $|\omega| \geq m$.
3. Since the first m symbols are all a's, the opponent is forced to pick $x = a^j$, $y = a^k$ and $z = a^l b^m$, with $j + k \leq m$ and $l \geq 0$ and $j + k + l = m$

$$\omega = \underbrace{a \cdots a}_{x} \underbrace{a \cdots a}_{y} \underbrace{a \ldots a b \cdots b}_{z}$$

4. We choose $i = 2$ which means $a^j a^k a^k a^l b^m = a^{m+k} b^m \in L$ but it can not be!
5. The opponent does not have any other way of partitioning ω, so L is not regular.

Using the Pumping Lemma

Consider $L = \{\omega | n_a(\omega) < n_b(\omega)\}$

Using the Pumping Lemma

Consider $L = \{\omega | n_a(\omega) < n_b(\omega)\}$

1. Opponent picks m

Using the Pumping Lemma

Consider $L = \{\omega | n_a(\omega) < n_b(\omega)\}$

1. Opponent picks m
2. We pick $a^m b^{m+1}$. Clearly $|\omega| \geq m$.

Using the Pumping Lemma

Consider $L = \{\omega | n_a(\omega) < n_b(\omega)\}$

1. Opponent picks m
2. We pick $a^m b^{m+1}$. Clearly $|\omega| \geq m$.
3. Opponent is forced to pick $y = a^k$ for some $1 \leq k \leq m$

Using the Pumping Lemma

Consider $L = \{\omega | n_a(\omega) < n_b(\omega)\}$

1. Opponent picks m
2. We pick $a^m b^{m+1}$. Clearly $|\omega| \geq m$.
3. Opponent is forced to pick $y = a^k$ for some $1 \leq k \leq m$
4. We pick $i = 2$ which means $a^{m+k} b^{m+1} \in L$ but it can not be!

Using the Pumping Lemma

Consider $L = \{\omega | n_a(\omega) < n_b(\omega)\}$

1. Opponent picks m
2. We pick $a^m b^{m+1}$. Clearly $|\omega| \geq m$.
3. Opponent is forced to pick $y = a^k$ for some $1 \leq k \leq m$
4. We pick $i = 2$ which means $a^{m+k} b^{m+1} \in L$ but it can not be!
5. The opponent does not have any other way of partitioning ω, so L is not regular.

Using the Pumping Lemma

Consider $L = \{1^{n^2} | n \geq 0\}$

Using the Pumping Lemma

Consider $L = \{1^{n^2} | n \geq 0\}$

1. Opponent picks m

Using the Pumping Lemma

Consider $L = \{1^{n^2} | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = 1^{m^2}$. Clearly $|\omega| \geq m$.

Using the Pumping Lemma

Consider $L = \{1^{n^2} | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = 1^{m^2}$. Clearly $|\omega| \geq m$.
3. Opponent chooses any partitioning of $\omega = xyz = 1^j 1^k 1^l$ with $1 \leq k \leq m$ and $j + k \leq m$

Using the Pumping Lemma

Consider $L = \{1^{n^2} | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = 1^{m^2}$. Clearly $|\omega| \geq m$.
3. Opponent chooses any partitioning of $\omega = xyz = 1^j 1^k 1^l$ with $1 \leq k \leq m$ and $j + k \leq m$
4. With $|xyz| = m^2$ and $i = 2$, $m^2 < |xyyz| \leq m^2 + m$.

Using the Pumping Lemma

Consider $L = \{1^{n^2} | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = 1^{m^2}$. Clearly $|\omega| \geq m$.
3. Opponent chooses any partitioning of $\omega = xyz = 1^j 1^k 1^l$ with $1 \leq k \leq m$ and $j + k \leq m$
4. With $|xyz| = m^2$ and $i = 2$, $m^2 < |xyyz| \leq m^2 + m$. But $m^2 < m^2 + m < m^2 + 2m + 1 = (m+1)^2$

Using the Pumping Lemma

Consider $L = \{1^{n^2} | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = 1^{m^2}$. Clearly $|\omega| \geq m$.
3. Opponent chooses any partitioning of $\omega = xyz = 1^j 1^k 1^l$ with $1 \leq k \leq m$ and $j + k \leq m$
4. With $|xyz| = m^2$ and $i = 2$, $m^2 < |xyyz| \leq m^2 + m$. But $m^2 < m^2 + m < m^2 + 2m + 1 = (m+1)^2$
5. $|xyyz|$ lies between two consecutive perfect squares. So $xyyz \notin L$.

Using the Pumping Lemma

Consider $L = \{1^{n^2} | n \geq 0\}$

1. Opponent picks m
2. We pick $\omega = 1^{m^2}$. Clearly $|\omega| \geq m$.
3. Opponent chooses any partitioning of $\omega = xyz = 1^j 1^k 1^l$ with $1 \leq k \leq m$ and $j + k \leq m$
4. With $|xyz| = m^2$ and $i = 2$, $m^2 < |xyyz| \leq m^2 + m$. But $m^2 < m^2 + m < m^2 + 2m + 1 = (m+1)^2$
5. $|xyyz|$ lies between two consecutive perfect squares. So $xyyz \notin L$.
6. L can not be regular.

Summary

- Symbols, Strings, Languages, Set of all Languages

Summary

- Symbols, Strings, Languages, Set of all Languages
- DFAs, Regular Languages, NFAs, Regular Expressions

Summary

- Symbols, Strings, Languages, Set of all Languages
- DFAs, Regular Languages, NFAs, Regular Expressions
- DFA $\Leftrightarrow$ REs

Summary

- Symbols, Strings, Languages, Set of all Languages
- DFAs, Regular Languages, NFAs, Regular Expressions
- DFA $\Leftrightarrow$ REs
- Minimal DFAs

SUMMARY

- Symbols, Strings, Languages, Set of all Languages
- DFAs, Regular Languages, NFAs, Regular Expressions
- DFA $\Leftrightarrow$ REs
- Minimal DFAs
- Closure properties, Decision properties

Summary

- Symbols, Strings, Languages, Set of all Languages
- DFAs, Regular Languages, NFAs, Regular Expressions
- DFA $\Leftrightarrow$ REs
- Minimal DFAs
- Closure properties, Decision properties
- Nonregular Languages, Pumping Lemma

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L_1 is not regular and L_2 is regular then $L = L_1 L_2 = \{xy : x \in L_1 \text{and } y \in L_2\}$ is not regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L_1 is not regular and L_2 is regular then $L = L_1 L_2 = \{xy : x \in L_1 \text{ and } y \in L_2\}$ is not regular.
2. $L = \{a^i b^j a^k : i + k < 10 \text{ and } j > 10\}$ is not regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L_1 is not regular and L_2 is regular then $L = L_1 L_2 = \{xy : x \in L_1 \text{ and } y \in L_2\}$ is not regular.
2. $L = \{a^i b^j a^k : i + k < 10 \text{ and } j > 10\}$ is not regular.
3. $L = \{w \in \{a, b\}^* : n_a(w) \times n_b(w) = 0 \bmod 2\}$ is regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L_1 is not regular and L_2 is regular then $L = L_1 L_2 = \{xy : x \in L_1 \text{ and } y \in L_2\}$ is not regular.
2. $L = \{a^i b^j a^k : i + k < 10 \text{ and } j > 10\}$ is not regular.
3. $L = \{w \in \{a, b\}^* : n_a(w) \times n_b(w) = 0 \bmod 2\}$ is regular.
4. $L = \{a^i b^j : i + j \geq 10\}$ is not regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L_1 is not regular and L_2 is regular then $L = L_1 L_2 = \{xy : x \in L_1 \text{ and } y \in L_2\}$ is not regular.
2. $L = \{a^i b^j a^k : i + k < 10 \text{ and } j > 10\}$ is not regular.
3. $L = \{w \in \{a, b\}^* : n_a(w) \times n_b(w) = 0 \bmod 2\}$ is regular.
4. $L = \{a^i b^j : i + j \geq 10\}$ is not regular.
5. $L = \{a^i b^j : i - j > 10\}$ is not regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L_1 is not regular and L_2 is regular then $L = L_1 L_2 = \{xy : x \in L_1 \text{ and } y \in L_2\}$ is not regular.
2. $L = \{a^i b^j a^k : i + k < 10 \text{ and } j > 10\}$ is not regular.
3. $L = \{w \in \{a, b\}^* : n_a(w) \times n_b(w) = 0 \bmod 2\}$ is regular.
4. $L = \{a^i b^j : i + j \geq 10\}$ is not regular.
5. $L = \{a^i b^j : i - j > 10\}$ is not regular.
6. $L = \{a^i a^j : i/j = 5\}$ is not regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L_1 is not regular and L_2 is regular then $L = L_1 L_2 = \{xy : x \in L_1 \text{ and } y \in L_2\}$ is not regular.
2. $L = \{a^i b^j a^k : i + k < 10 \text{ and } j > 10\}$ is not regular.
3. $L = \{w \in \{a, b\}^* : n_a(w) \times n_b(w) = 0 \bmod 2\}$ is regular.
4. $L = \{a^i b^j : i + j \geq 10\}$ is not regular.
5. $L = \{a^i b^j : i - j > 10\}$ is not regular.
6. $L = \{a^i a^j : i/j = 5\}$ is not regular.
7. If $L_1 \cap L_2$ is regular then L_1 and L_2 are regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L_1 is not regular and L_2 is regular then $L = L_1 L_2 = \{xy : x \in L_1 \text{ and } y \in L_2\}$ is not regular.
2. $L = \{a^i b^j a^k : i + k < 10 \text{ and } j > 10\}$ is not regular.
3. $L = \{w \in \{a, b\}^* : n_a(w) \times n_b(w) = 0 \bmod 2\}$ is regular.
4. $L = \{a^i b^j : i + j \geq 10\}$ is not regular.
5. $L = \{a^i b^j : i - j > 10\}$ is not regular.
6. $L = \{a^i a^j : i/j = 5\}$ is not regular.
7. If $L_1 \cap L_2$ is regular then L_1 and L_2 are regular.
8. If $L_1 \subseteq L_2$ and L_2 is regular, then L_1 must be regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. There are subsets of a regular language which are not regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. There are subsets of a regular language which are not regular.
2. If L_1 and L_2 are nonregular, then $L_1 \cup L_2$ must be nonregular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. There are subsets of a regular language which are not regular.
2. If L_1 and L_2 are nonregular, then $L_1 \cup L_2$ must be nonregular.
3. If F is a finite language and L is some language, and $L - F$ is a regular language, then L must be a regular language.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. There are subsets of a regular language which are not regular.
2. If L_1 and L_2 are nonregular, then $L_1 \cup L_2$ must be nonregular.
3. If F is a finite language and L is some language, and $L - F$ is a regular language, then L must be a regular language.
4. $L = \{w \in \{a, b\}$: the number a's times the number of b's in w is greater than 1333$\}$ is not regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. There are subsets of a regular language which are not regular.
2. If L_1 and L_2 are nonregular, then $L_1 \cup L_2$ must be nonregular.
3. If F is a finite language and L is some language, and $L - F$ is a regular language, then L must be a regular language.
4. $L = \{w \in \{a, b\}$: the number a's times the number of b's in w is greater than 1333$\}$ is not regular.
5. If the start state of a DFA has a self-loop, then the language accepted by that DFA is infinite.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. There are subsets of a regular language which are not regular.
2. If L_1 and L_2 are nonregular, then $L_1 \cup L_2$ must be nonregular.
3. If F is a finite language and L is some language, and $L - F$ is a regular language, then L must be a regular language.
4. $L = \{w \in \{a, b\}$: the number a's times the number of b's in w is greater than 1333$\}$ is not regular.
5. If the start state of a DFA has a self-loop, then the language accepted by that DFA is infinite.
6. The set of strings of 0's, 1's, and 2's with at least 100 of each of the three symbols is a regular language.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. There are subsets of a regular language which are not regular.
2. If L_1 and L_2 are nonregular, then $L_1 \cup L_2$ must be nonregular.
3. If F is a finite language and L is some language, and $L - F$ is a regular language, then L must be a regular language.
4. $L = \{w \in \{a, b\}$: the number a's times the number of b's in w is greater than 1333$\}$ is not regular.
5. If the start state of a DFA has a self-loop, then the language accepted by that DFA is infinite.
6. The set of strings of 0's, 1's, and 2's with at least 100 of each of the three symbols is a regular language.
7. The union of a countable number of regular languages is regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L is nonregular then $\overline{L}$ is nonregular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L is nonregular then $\overline{L}$ is nonregular.
2. If $L_1 \cap L_2$ is finite then L_1 and L_2 are regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L is nonregular then $\overline{L}$ is nonregular.
2. If $L_1 \cap L_2$ is finite then L_1 and L_2 are regular.
3. The family of regular languages is closed under *nor* operation, $nor(L_1, L_2) = \{w \ : \ w \notin L_1 \text{ and } w \notin L_2\}$

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L is nonregular then $\overline{L}$ is nonregular.
2. If $L_1 \cap L_2$ is finite then L_1 and L_2 are regular.
3. The family of regular languages is closed under *nor* operation, $nor(L_1, L_2) = \{w \ : \ w \notin L_1 \text{ and } w \notin L_2\}$
4. If L is a regular language, then so is $\{xy\text{: } x \in L \text{ and } y \notin L\}$

LET'S SEE IF WE CAN TIE THINGS TOGETHER

True or False?

1. If L is nonregular then $\overline{L}$ is nonregular.
2. If $L_1 \cap L_2$ is finite then L_1 and L_2 are regular.
3. The family of regular languages is closed under *nor* operation, $nor(L_1, L_2) = \{w \ : \ w \notin L_1 \text{ and } w \notin L_2\}$
4. If L is a regular language, then so is $\{xy$: $x \in L$ and $y \notin L\}$
5. Let L be a regular language over $\Sigma = \{a, b, c\}$. Let us define $SINGLE(L) = \{w \in L : \text{all symbols in } w \text{ are the same}\}$. $SINGLE(L)$ is regular.

LET'S SEE IF WE CAN TIE THINGS TOGETHER

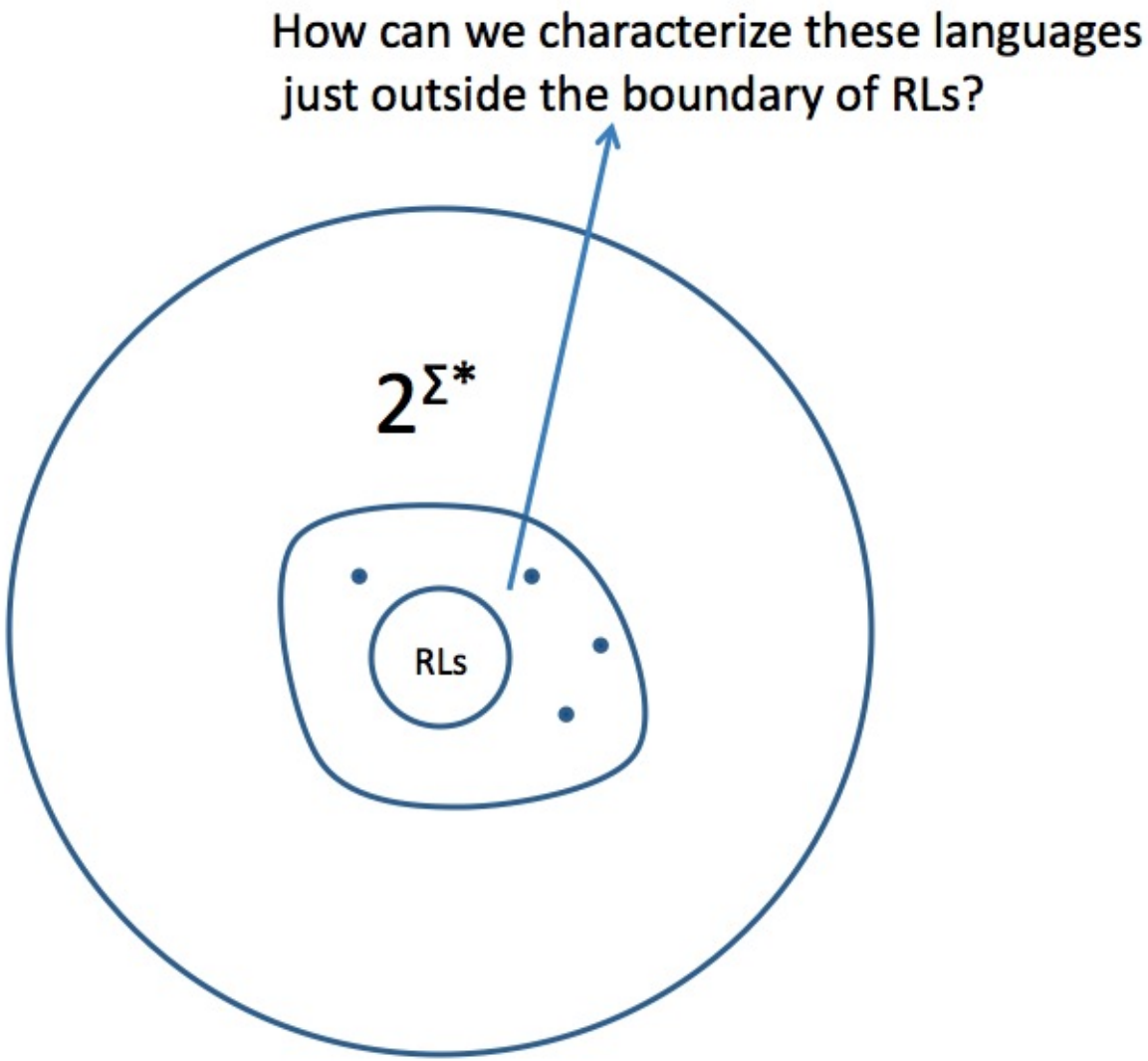
Let $\Sigma = \{a\}$ and let M be a *deterministic finite state acceptor* that accepts a regular language $L \subseteq \Sigma^*$.

A) Describe with very simple diagrams, possible structures of the state graph of M, if M has only a single final state. Show any relevant parameters that you feel are necessary.

B) Describe with a regular expression the language accepted by M, if M has a single final state. If necessary, use any parameters you showed in part *a)*.

C) Describe *mathematically* the language accepted by M, if M *has more than one final state*.

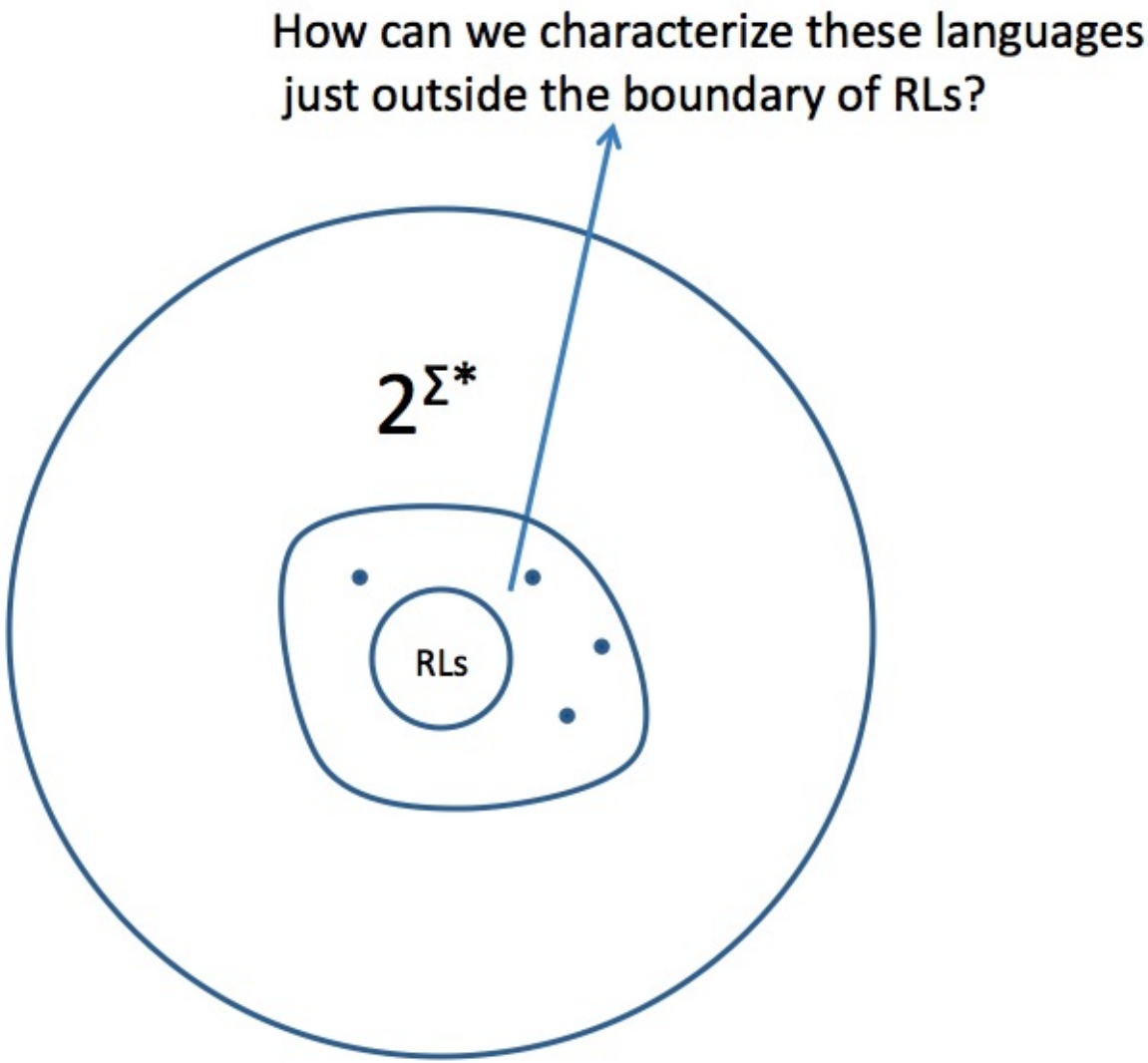
WHERE DO WE GO FROM HERE?



Formal Languages, Automata and Computation

Context Free Languages

WHERE ARE WE?


How can we characterize these languages
just outside the boundary of RLs?
2^{Σ^*}
RLs

A Nonregular Language

- We showed that $L = \{a^n b^n \mid n \geq 0\}$ was not regular.
  - No DFA
  - No Regular Expression
- How can we describe such languages?
- Remember: the description has to be finite!

A Nonregular Language

- Consider $L = \{a^n b^n \mid n \geq 0\}$ again.
- How can we generate such strings?
  - Remember DFAs did recognition, not generation.
- Consider the following inductive way to generate elements of L
  - Basis: ϵ is in the language ($n = 0$)
  - Recursion: If the string w (for some n) is in the language, then so is the string awb (for $n + 1$).
- $\epsilon \rightarrow ab \rightarrow aabb \cdots \rightarrow a^{55}b^{55} \cdots$
- Looks like we have simple and finite length process to generate all the strings in L
- How can we generalize this kind of description?

Another Nonregular Language

- Consider $L = \{w \mid n_a(w) = n_b(w)\}$.
- Now consider the following inductive way to generate elements of L
  - Basis: ϵ is in the language
  - Recursion 1: If the string w is in the language, then so are awb and bwa
  - Recursion 2: If the strings w and v are in the language, so is wv.
- The first recursion rules makes sure that the a's and b's are generated in the same number (regardless of order)
- The second recursion takes any two strings each with equal number of a's and b's and generates a new such string by concatenating them.

GRAMMARS

- Grammars provide the generative mechanism to generate all strings in a language.
- A grammar is essentially a collection of substitution rules, called productions
- Each production rule has a left-hand-side and a right-hand-side.

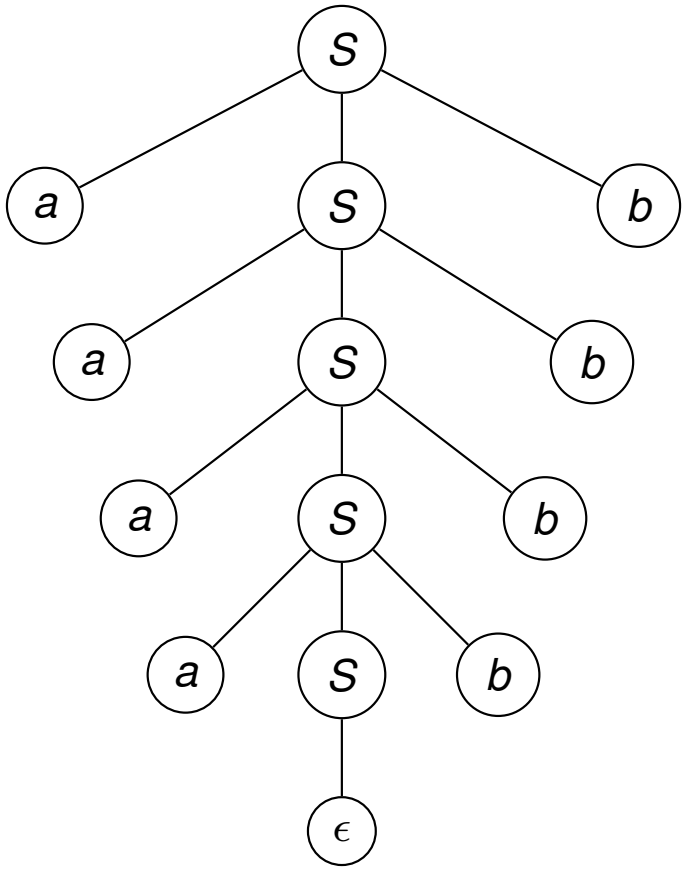
Grammars - An Example

- Consider once again $L = \{a^n b^n \mid n \geq 0\}$
- Basis: ϵ is in the language
  - Production: $S \rightarrow \epsilon$
- Recursion: If w is in the language, then so is the string awb.
  - Production: $S \rightarrow aSb$
- S is called a variable or a nonterminal symbol
- a, b etc., are called terminal symbols
- One variable is designated as the start variable or start symbol.

How does a grammar work?

- Consider the set of rules $R = \{S \to \epsilon, S \to aSb\}$
- Start with the start variable S
- Apply the following until all remaining symbols are terminal.
  - Choose a production in R whose left-hand sides matches one of the variables.
  - Replace the variable with the rule's right hand side.
- $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$
- The string $aaaabbbb$ is in the language L
- The sequence of rule applications above is called a derivation.

Parse Trees



The terminals concatenated from left to right give us the string.

- Derivations can also be represented with a parse tree.
- The leaves constitute the yield of the tree.
- Terminal symbols can occur only at the leaves.
- Variables can occur only at the internal nodes.

Language of a Grammar

- All strings generated this way starting with the start variable constitute the language of the grammar.
- We write $L(G)$ for the language of the grammar G.

A Grammar for a Fragment of English

$$
\begin{array}{rcl}
S & \rightarrow & NP\ VP \\
NP & \rightarrow & CN \mid CN\ PP \\
VP & \rightarrow & CV \mid CV\ PP \\
PP & \rightarrow & P\ NP \\
CN & \rightarrow & DT\ N \\
CV & \rightarrow & V \mid V\ NP \\
DT & \rightarrow & \text{a} \mid \text{the} \\
N & \rightarrow & \text{boy} \mid \text{girl} \mid \text{flower} \mid \\
 & & \text{telescope} \\
V & \rightarrow & \text{touches} \mid \text{likes} \mid \\
 & & \text{sees} \mid \text{gives} \\
P & \rightarrow & \text{with} \mid \text{to}
\end{array}
$$

Nomenclature:

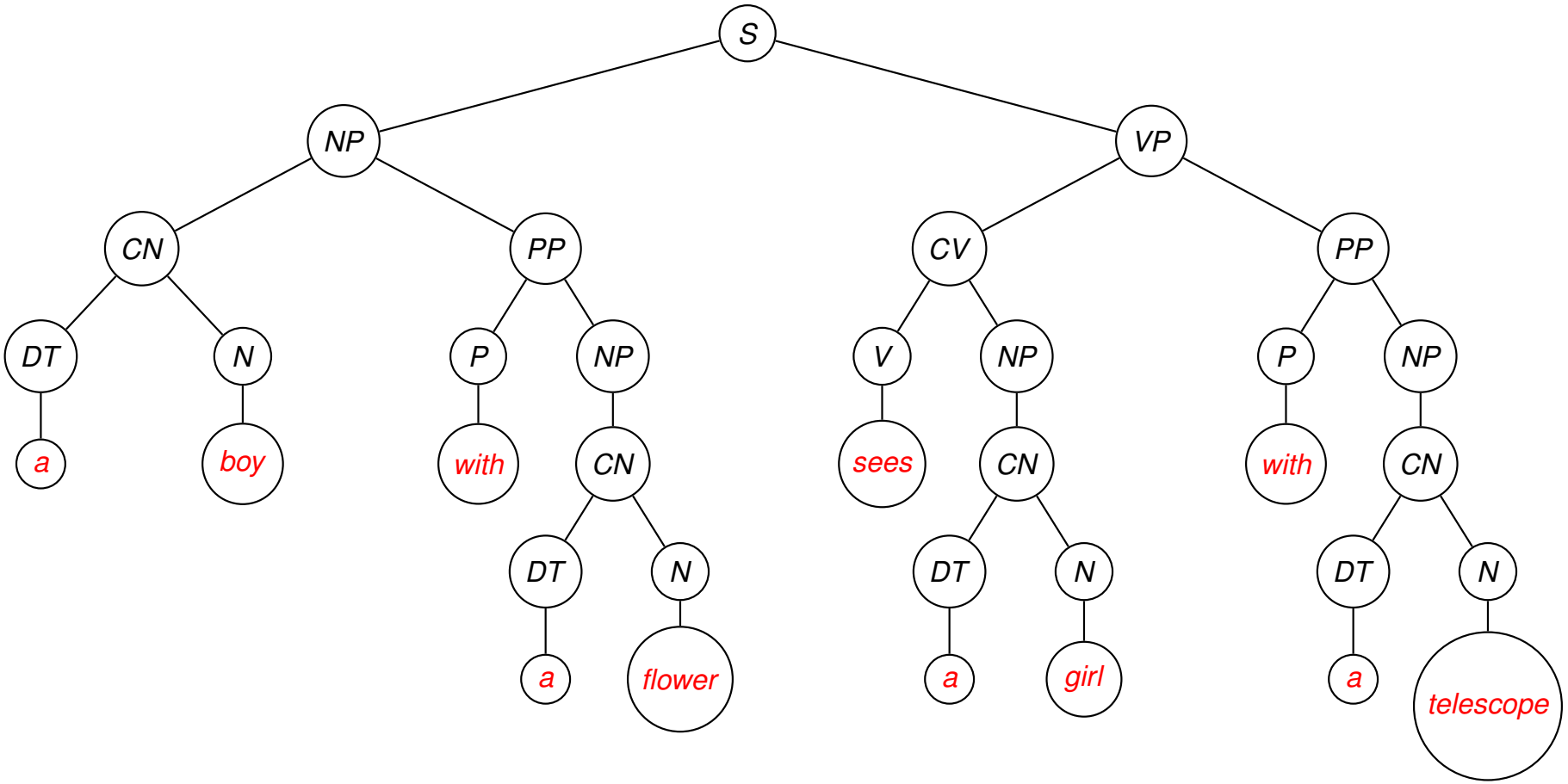
- *S*: Sentence
- *NP*: Noun Phrase
- *CN*: Complex Noun
- *PP*: Prepositional Phrase
- *VP*: Verb Phrase
- *CV*: Complex Verb
- *P*: Preposition
- *DT*: Determiner

A Grammar for a Fragment of English

$$
\begin{array}{rcl}
S & \rightarrow & NP\ VP \\
NP & \rightarrow & CN \mid CN\ PP \\
VP & \rightarrow & CV \mid CV\ PP \\
PP & \rightarrow & P\ NP \\
CN & \rightarrow & DT\ N \\
CV & \rightarrow & V \mid V\ NP \\
DT & \rightarrow & \text{a} \mid \text{the} \\
N & \rightarrow & \text{boy} \mid \text{girl} \mid \text{flower} \mid \\
 & & \text{telescope} \\
V & \rightarrow & \text{touches} \mid \text{likes} \mid \\
 & & \text{sees} \mid \text{gives} \\
P & \rightarrow & \text{with} \mid \text{to}
\end{array}
$$

$$
\begin{array}{rcl}
S & \Rightarrow & NP\ VP \\
 & \Rightarrow & CN\ PP\ VP \\
 & \Rightarrow & DT\ N\ PP\ VP \\
 & \Rightarrow & \text{a}\ N\ PP\ VP \\
 & \Rightarrow & \ldots \\
 & \Rightarrow & \text{a boy with a flower}\ VP \\
 & \Rightarrow & \text{a boy with a flower}\ CV\ PP \\
 & \Rightarrow & \ldots \\
 & \Rightarrow & \text{a boy with a flower sees a girl} \\
 & & \text{with a telescope}
\end{array}
$$

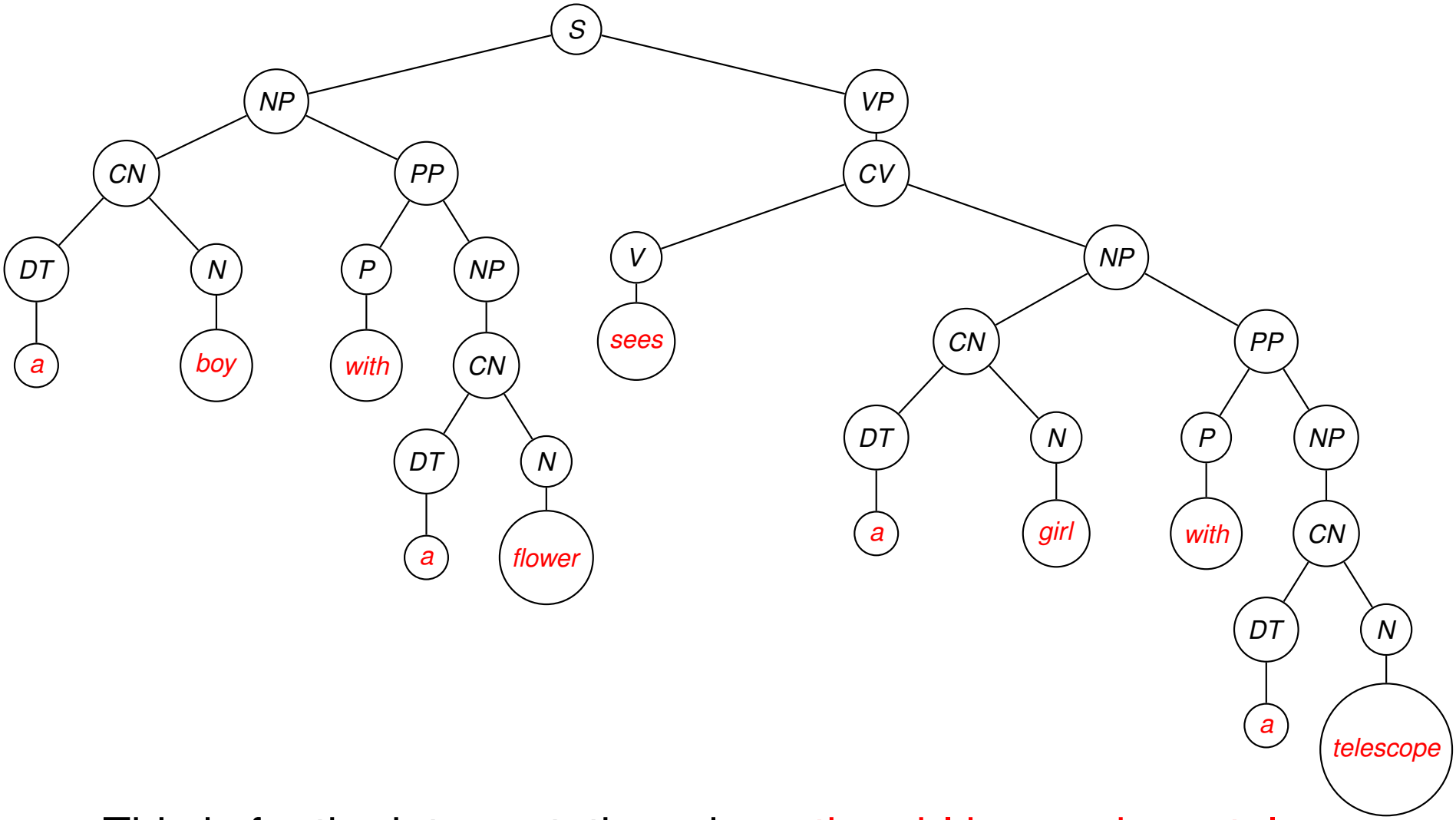
English Parse Tree



- This structure is for the interpretation where the boy is seeing with the telescope!

English Parse Tree

Alternate Structure



- This is for the interpretation where the girl is carrying a telescope.

Structural Ambiguity

- A set of rules can assign multiple structures to the same string.
- Which rule one chooses determines the eventual structure.
  - $VP \rightarrow CV \mid CV\ PP$
  - $CV \rightarrow V \mid V\ NP$
  - $NP \rightarrow CN \mid CN\ PP$
  - $\cdots$ [$_{VP}$ [$_{CV}$ sees [$_{NP}$ a girl] [$_{PP}$ with a telescope]].
  - $\cdots$ [$_{VP}$ [$_{CV}$ sees] [$_{NP}$ [$_{CN}$ a girl] [$_{PP}$ with a telescope]].
    - (Not all brackets are shown!)

Other Examples of Grammar Applications

- Programming Languages
  - Users need to know how to “generate” correct programs.
  - Compilers need to know how to “check” and “translate” programs.
- XML Documents
  - Documents need to have a structure defined by a DTD grammar.
- Natural Language Processing, Machine Translation

Formal Definition of a Grammar

- A Grammar is a 4-tuple $G = (V, \Sigma, R, S)$ where
  - V is a finite set of variables
  - Σ is a finite set of terminals, disjoint from V.
  - R is a set of rules of the $X \to Y$
  - $S \in V$ is the start variable
- In general $X \in (V \cup \Sigma)^+$ and $Y \in (V \cup \Sigma)^*$
- A context-free grammar is a grammar where all rules have $X \in V$ (remember $V \subset (V \cup \Sigma)^+$)
  - The substitution is independent of the context V appears in.
- The right hand side of the rules can be any combination of variables and terminals, including ϵ (hence $Y \in (V \cup \Sigma)^*$).

Formal Definition of a Grammar

- If u, v and w are strings of variables and terminals and $A \to w$ is a rule of the grammar, we say uAv yields uwv, notated as $uAv \Rightarrow uwv$
- We say u derives v, notated as, $u \overset{*}{\Rightarrow} v$, if either
  - $u = v$, or
  - a sequence $u_1, u_2, \ldots, u_k$, $k \geq 0$ exists such that $u \Rightarrow u_1 \Rightarrow u_2, \cdots, \Rightarrow u_k \Rightarrow v$.
  - We call u, v, and all u_i as sentential forms.
- The language of the grammar is $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$

Designing Context Free Grammars

- Consider once again the language $L = \{w \mid n_a(w) = n_b(w)\}$.
- The grammar for this language is $G = (\{S\}, \{a, b\}, R, S)$ with R as follows:
  1. $S \to aSb$
  2. $S \to bSa$
  3. $S \to SS$
  4. $S \to \epsilon$
- From now we will only list the productions, the others will be implicit.
- We will also combine productions with the same left-hand side using | symbol.
- $S \to aSb \mid bSa \mid SS \mid \epsilon$

Designing Context Free Grammars

- $L = \{w \mid n_a(w) = n_b(w)\}$.
- $S \rightarrow aSb \mid bSa \mid SS \mid \epsilon$

- Clearly the strings generated by G have equal number of a's and b's. (Obvious from the rules!)
- We also have to show that all strings in L can be generated with this grammar.

Designing Context Free Grammars

Assertion

Grammar G with $R = \{S \rightarrow aSb \mid bSa \mid SS \mid \epsilon\}$ generates $L = \{w \mid n_a(w) = n_b(w)\}$.

Proof (by Induction)

- The grammar generates the basis strings of ϵ, *ab* and *ba*.
- All other strings in L have even length and can be in one of the 4 possible forms ($w \in \Sigma^*$)
  1. *awb*
  2. *bwa*
  3. *awa*
  4. *bwb*

Designing Context Free Grammars

Proof (continued)

- Assume that G generates all strings of equal number of a's and b's of (even) length n.
- Consider a string like awb of length $n+2$.
- awb will be generated from w by using the rule $S \rightarrow aSb$ provided $S \stackrel{*}{\Rightarrow} w$.
- But w is of length n, so $S \stackrel{*}{\Rightarrow} w$ by the induction hypothesis.
- There is a symmetric argument for strings like bwa.

Designing Context Free Grammars

Proof (continued)

- Consider a string like *awa*. Clearly $w \notin L$. Consider (symbols of) this string annotated as follows

$$_0a_1 \cdots _{-1}a_0$$

where the subscripts after a prefix v of *awa* denotes $n_a(v) - n_b(v)$.

- Think of this as count starting as 0, each a adding one and each b subtracting 1. We should end with 0 at the end.
- Note that right after the first symbol we have 1 and right before the last a we must have -1.
- Somewhere along the string (in w) the counter crosses 0.

Designing Context Free Grammars

Proof (continued)

- Somewhere along the string (in w) the counter crosses 0.

$$0 \overbrace{a_1 \cdots x}^{u} 0 \underbrace{y \cdots_{-1} a}_{v} 0 \qquad x, y \in \{a, b\}$$

- So u and v have equal numbers of a's and b's and are shorter.
- $u, v \in L$ by the induction hypothesis and the rule $S \to SS$ generates $awa = uv$, given $S \overset{*}{\Rightarrow} u$ and $S \overset{*}{\Rightarrow} v$
- There is a symmetric argument for strings like bwb.

Formal Languages, Automata and Computation

Context Free Languages

Summary

- Describing nonregular languages
- Grammars as finite descriptions of infinite sets
- Context-free Grammars and context-free languages
- Derivations and parse trees
- Ambiguity
- Writing grammars

Grammar Examples

- Consider $L = \{a^n b^n \mid n \geq 0\}$
- $S \rightarrow aSb$
  - a's and b's are generated in the right order and in equal numbers
- $S \rightarrow \epsilon$
  - get rid of any remaining S at the end.

Grammar Examples

- Consider $L = \{a^n b^m \mid m > n \geq 0\}$
- $S \rightarrow AB$
- $A \rightarrow aAb \mid \epsilon$
  - a's and b's are generated in the right order and in equal numbers, followed by B
- $B \rightarrow bB \mid b$
  - Generate 1 or more (additional) b's

Grammar Examples

- $L = \{a^n b^{2n} \mid n \geq 0\}$
  - $S \rightarrow aSbb \mid \epsilon$
- $L = \{a^{n+2} b^n \mid n \geq 1\}$
  - $S \rightarrow aaA$,
  - $A \rightarrow aAb \mid ab$

Grammar for Arithmetic Expressions

- $L \rightarrow a \mid b \mid \cdots \mid z$ (letters)
- $D \rightarrow 0 \mid \cdots \mid 9$ (digits)
- $V \rightarrow L \mid V\ L \mid V\ D$ (variables)
- $N \rightarrow D \mid N\ D$ (positive numbers)
- $F \rightarrow V \mid N \mid (E)$ (factors)
- $T \rightarrow F \mid T * F \mid T/F$ (terms)
- $E \rightarrow T \mid E + T \mid E - T$ (expressions)
- E is the start symbol.

Let us generate $(v23 + 456) * k23/(a - b * 34)$ as an exercise.

Ambiguity

- Remember a boy with a flower sees a girl with a telescope?
- We say that a grammar generates a string ambiguously, if the string has two different parse trees (not just two different derivations)
- A derivation of a string w in a grammar G is a leftmost derivation if at every step, the leftmost remaining variable is the one replaced.

Ambiguity

Definition

A string w is derived ambiguously in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.

- Sometimes an ambiguous grammar can be transformed into an unambiguous grammar for the same language.
- Some context-free grammars can be generated only by ambiguous grammars. These are known as inherently ambiguous languages.
  - $L = \{a^i b^j c^k \mid i = j \text{ or } j = k\}$

Grammar Transformations

- Some types of productions cause problems in some uses of grammars.
- ϵ-productions: $A \rightarrow \epsilon$
  - Intermediate sentential forms in a derivation get shorter and this has computational implications.
- Unit productions: $A \rightarrow B$.
  - Such a rule does not achieve much except for lengthening the derivation sequence.
  - There may be inadvertent "infinite loops": e.g., if $A \stackrel{*}{\Rightarrow} A$

Removing ϵ-productions

- If $\epsilon \in L$, then we can not do much. $S \to \epsilon$ is needed for this.
- For all rules of the type $A \to \epsilon$ and A is not the start symbol, we proceed as follows:
- For occurrence of an A on the right-hand side of a rule, we add a rule with that occurence deleted.
  - For a rule like $R \to uAv$, we add the rule $R \to uv$ (either u or v not ϵ)
  - For a rule like $R \to A$, we add $R \to \epsilon$, unless we removed $R \to \epsilon$ earlier.
  - For a rule with multiple occurences of A, we add one rule for each combination. $R \to uAvAw$ would add $R \to uvAw$, $R \to uAvw$, and $R \to uvw$.

REMOVING ϵ-PRODUCTIONS

- Consider

$$\begin{aligned} S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \epsilon \end{aligned}$$

- Add a new start symbol S_0

$$\begin{aligned} \mathbf{S_0} &\rightarrow \mathbf{S} \\ S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \epsilon \end{aligned}$$

- Remove $B \rightarrow \epsilon$

$$\begin{aligned} \mathbf{S_0} &\rightarrow \mathbf{S} \\ S &\rightarrow ASA \mid aB \mid \mathbf{a} \\ A &\rightarrow B \mid S \mid \epsilon \\ B &\rightarrow b \end{aligned}$$

- Remove $A \rightarrow \epsilon$

$$\begin{aligned} \mathbf{S_0} &\rightarrow \mathbf{S} \\ S &\rightarrow ASA \mid aB \mid a \mid \\ &\quad\ \mathbf{SA} \mid \mathbf{AS} \mid \mathbf{S} \\ A &\rightarrow B \mid S \\ B &\rightarrow b \end{aligned}$$

Removing Unit Productions

- To remove a unit rule like $A \to B$,
  - We first add to the grammar a rule $A \to u$ whenever $B \to u$ is in the grammar, unless this is a unit rule previously removed.
  - We then delete $A \to B$, from the grammar.
- We repeat these until we eliminate all unit rules.

Removing Unit Productions

- After ϵ-rule removal

$$
\begin{aligned}
S_0 &\rightarrow S \\
S &\rightarrow ASA \mid aB \mid a \mid SA \mid AS \mid S \\
A &\rightarrow B \mid S \\
B &\rightarrow b
\end{aligned}
$$

- Remove $S \rightarrow S$

$$
\begin{aligned}
S_0 &\rightarrow S \\
S &\rightarrow ASA \mid aB \mid a \mid SA \mid AS \\
A &\rightarrow B \mid S \\
B &\rightarrow b
\end{aligned}
$$

- Remove $S_0 \rightarrow S$

$$
\begin{aligned}
S_0 &\rightarrow ASA \mid aB \mid a \mid SA \mid AS \\
S &\rightarrow ASA \mid aB \mid a \mid SA \mid AS \\
A &\rightarrow B \mid S \\
B &\rightarrow b
\end{aligned}
$$

Removing Unit Productions

- After $S_0 \to S$ removal

$$
\begin{aligned}
S_0 &\to ASA \mid aB \mid a \mid SA \mid AS \\
S &\to ASA \mid aB \mid a \mid SA \mid AS \\
A &\to B \mid S \\
B &\to b
\end{aligned}
$$

- Remove $A \to B$

$$
\begin{aligned}
S_0 &\to ASA \mid aB \mid a \mid SA \mid AS \\
S &\to ASA \mid aB \mid a \mid SA \mid AS \\
A &\to b \mid S \\
B &\to b
\end{aligned}
$$

- Remove $A \to S$

$$
\begin{aligned}
S_0 &\to ASA \mid aB \mid a \mid SA \mid AS \\
S &\to ASA \mid aB \mid a \mid SA \mid AS \\
A &\to b \mid ASA \mid aB \mid a \mid SA \mid AS \\
B &\to b
\end{aligned}
$$

Chomsky Normal Form

- CFGs in certain standard forms are quite useful for some computational problems.

Chomsky Normal Form

A context-free grammar is in Chomsky normal form(CNF) if every rule is either of the form

$$A \to BC \text{ or } A \to a$$

where a is a terminal and A, B, C are variables – except B and C may not be the start variable. In addition, we allow the rule $S \to \epsilon$ if necessary.

Chomsky Normal Form

Theorem

Every context-free language can be generated by a context-free grammar in Chomksy normal form.

Proof Idea

- Add a new start variable and the production $S_0 \to S$.
- Remove all ϵ-productions
- Remove all unit productions.
- Add new variables and rules so that all rules have the right forms.

Chomsky Normal Form

Proof

u_i below is either a terminal or a variable.

- Replace each rule like $A \to u_1 u_2 \cdots u_k$ where $k \geq 3$, with rules $A \to u_1 A_1$, $A_1 \to u_2 A_2$, $\cdots$ $A_{k-2} \to u_{k-1} u_k$
- After this stage, all rules have right-hand side of length either 2 or 1
- For each rule like $A \to u_1 u_2$ where either or both u_i is a terminal, replace u_i with the new variable U_i and add the rule $U_i \to u_i$ to the grammar.

Conversion to Chomsky Normal Form

- Grammar after ϵ and unit production removal

$$
\begin{aligned}
S_0 &\rightarrow ASA \mid aB \mid a \mid SA \mid AS \\
S &\rightarrow ASA \mid aB \mid a \mid SA \mid AS \\
A &\rightarrow b \mid ASA \mid aB \mid a \mid SA \mid AS \\
B &\rightarrow b
\end{aligned}
$$

- Remove $S_0 \rightarrow ASA$ and add $S_0 \rightarrow AA_1$ and $A_1 \rightarrow SA$
- Remove $S \rightarrow ASA$ and add $S \rightarrow AA_1$ ($A_1 \rightarrow SA$ already added)
- Remove $A \rightarrow ASA$ and add $A \rightarrow AA_1$ ($A_1 \rightarrow SA$ already added)
- Replace $S_0 \rightarrow aB$ with $S_0 \rightarrow UB$ and $U \rightarrow a$
- Replace $S \rightarrow aB$ with $S \rightarrow UB$ ($U \rightarrow a$ already added)
- Replace $A \rightarrow aB$ with $A \rightarrow UB$ ($U \rightarrow a$ already added)

Conversion to Chomsky Normal Form

- Final grammar in Chomsky normal form

$$
\begin{aligned}
S_0 &\rightarrow AA_1 \mid UB \mid a \mid SA \mid AS \\
S &\rightarrow AA_1 \mid UB \mid a \mid SA \mid AS \\
A &\rightarrow b \mid AA_1 \mid UB \mid a \mid SA \mid AS \\
A_1 &\rightarrow SA \\
U &\rightarrow a \\
B &\rightarrow b
\end{aligned}
$$

Another Example

- Let's convert $R = \{S \to SS, S \to aSb, S \to bSa, S \to \epsilon\}$ to Chomsky Normal Form.

Other Interesting Forms for Grammars

- If all productions of a grammar are like $A \rightarrow bB$ or $A \rightarrow b$ where b is a terminal and B is a variable, then it is called a right-linear grammar.
- If all productions of a grammar are like $A \rightarrow Bb$ or $A \rightarrow b$ where b is a terminal and B is a variable, then it is called a left-linear grammar.
- Right-linear grammars generate regular languages.
- Left-linear grammars generate regular languages.

THE RECOGNITION PROBLEM FOR CFL'S

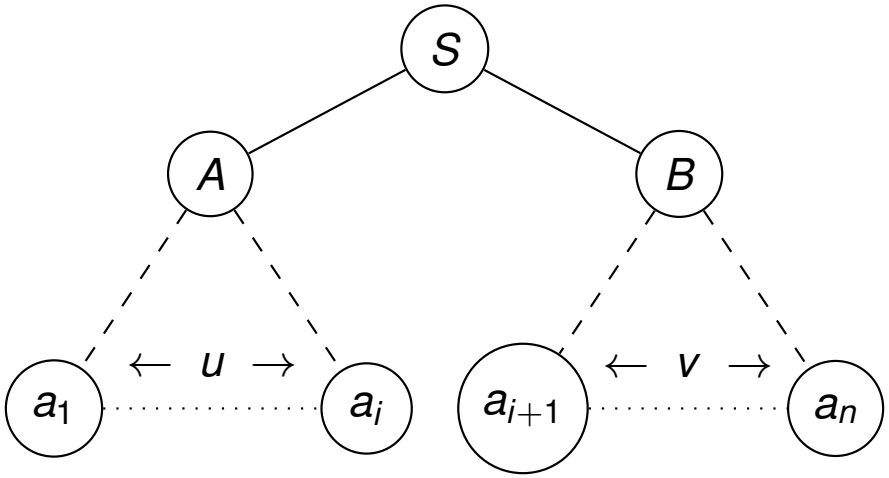
- Given a context-free grammar G and a string $w \in \Sigma^*$ how can we tell if $w \in L(G)$?
- If $w \in L(G)$, what are the possible structures assigned to w by G?
- Different grammars for the same language
  - will answer the first question the same, but
  - will assign possibly different structures to strings in the language.
  - Consider original and Chomsky Normal Form of some example grammars earlier!

The Cocke-Younger-Kasami (CYK) algorithm

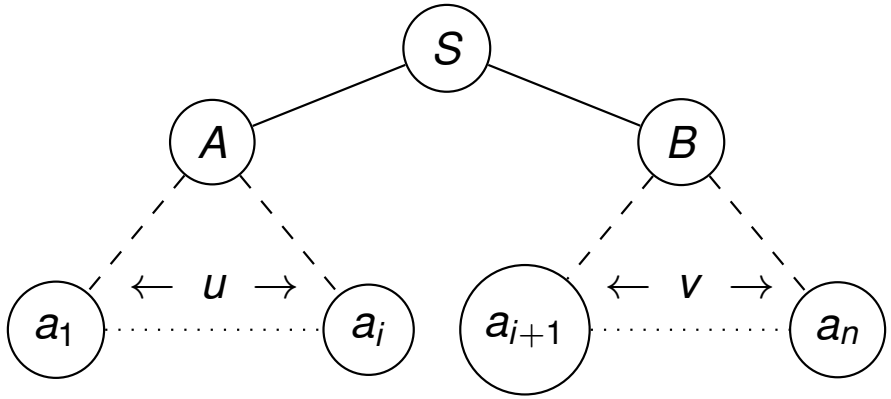
- The CYK parsing algorithm determines if $w \in L(G)$ for a grammar G in Chomsky Normal Form
  - with some extensions, it can also determine possible structures.
  - Assume $w \neq \epsilon$ (if so, check if the grammar has the rule $S \rightarrow \epsilon$)

The CYK Algorithm

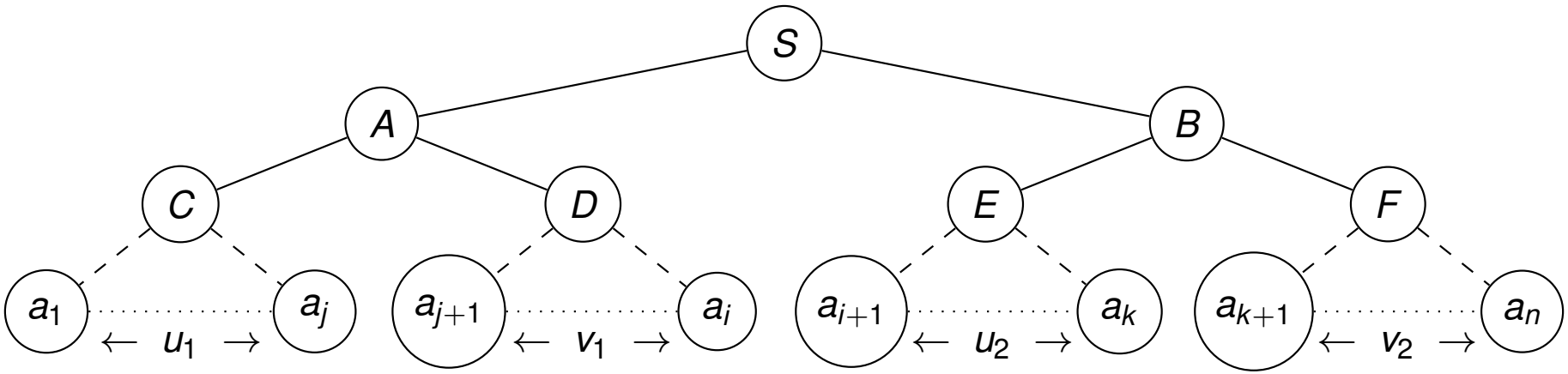
- Consider $w = a_1 a_2 \cdots a_n$, $a_i \in \Sigma$
- Suppose we could cut up the string into two parts $u = a_1 a_2 .. a_i$ and $v = a_{i+1} a_{i+2} \cdots a_n$
- Now suppose $A \stackrel{*}{\Rightarrow} u$ and $B \stackrel{*}{\Rightarrow} v$ and that $S \rightarrow AB$ is a rule.



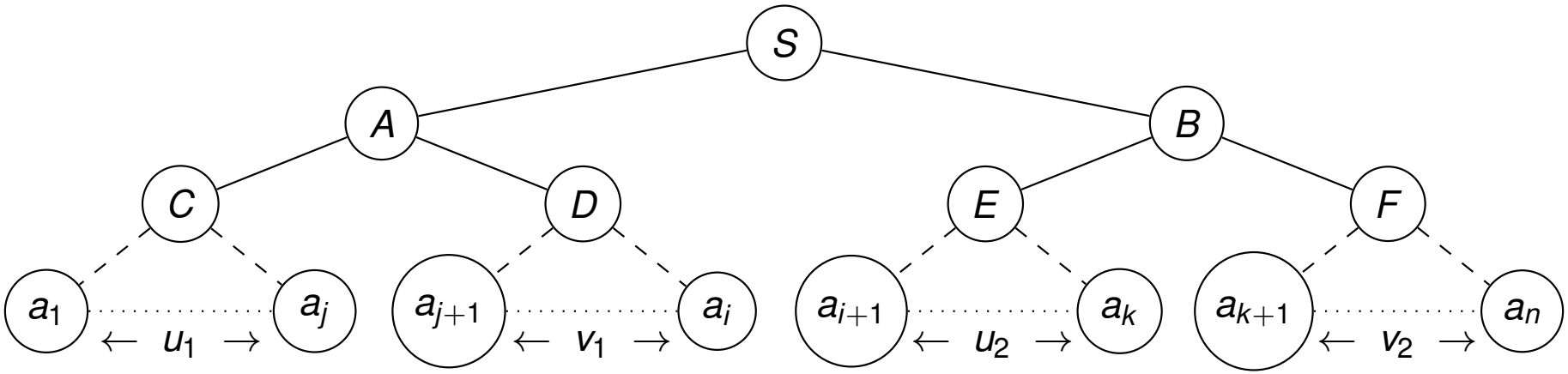
The CYK Algorithm



- Now we apply the same idea to A and B recursively.



The CYK Algorithm



- What is the problem here?
- We do not know what i, j and k are!
- No Problem! We can try all possible i's, j's and $k's$.
- Dynamic programming to the rescue.

DIGRESSION - DYNAMIC PROGRAMMING

- An algorithmic paradigm
- Essentially like divide-and-conquer but subproblems overlap!
- Results of subproblem solutions are reusable.
- Subproblem results are computed once and then memoized
- Used in solutions to many problems
  - Length of longest common subsequence
  - Knapsack
  - Optimal matrix chain multiplication
  - Shortest paths in graphs with negative weights (Bellman-Ford Alg.)

(Back to) The CYK Algorithm

- Let $w = a_1 a_2 \cdots a_n$.
- We define
  - $w_{i,j} = a_i \cdots a_j$ (substring between positions i and j)
  - $V_{i,j} = \{A \in V \mid A \stackrel{*}{\Rightarrow} w_{i,j}\} (j \geq i)$ (all variables which derive w_{ij})
- $w \in L(G)$ iff $S \in V_{1,n}$
- How do we compute $V_{i,j} (j \geq i)$?

The CYK Algorithm

- How do we compute $V_{i,j}$?
- Observe that $A \in V_{i,i}$ if $A \to a_i$ is a rule.
  - So V_{ii} can easily be computed for $1 \leq i \leq n$ by an inspection of w and the grammar.
- $A \stackrel{*}{\Rightarrow} w_{ij}$ if
  - There is a production $A \to BC$, and
  - $B \stackrel{*}{\Rightarrow} w_{i,k}$ and $C \stackrel{*}{\Rightarrow} w_{k+1,j}$ for some k, $i \leq k < j$.
- So

$$V_{i,j} = \bigcup_{i \leq k < j} \{A :| A \to BC \text{ and } B \in V_{i,k} \text{ and } C \in V_{k+1,j}\}$$

The CYK Algorithm

$$V_{i,j} = \bigcup_{i \leq k < j} \{A : A \to BC \text{ and } B \in V_{i,k} \text{ and } C \in V_{k+1,j}\}$$

- Compute in the following order:

$$\begin{array}{cccccccc} & \to & & & & & & \\ \downarrow & V_{1,1} & V_{2,2} & V_{3,3} & \cdots & \cdots & \cdots & V_{n,n} \\ & V_{1,2} & V_{2,3} & V_{3,4} & \cdots & \cdots & V_{n-1,n} & \\ & V_{1,3} & V_{2,4} & V_{3,5} & \cdots & V_{n-2,n} & & \\ & \cdots & & & & & & \\ & V_{1,n-1} & V_{2,n} & & & & & \\ & V_{1,n} & & & & & & \end{array}$$

- For example to compute $V_{2,4}$ one needs $V_{2,2}$ and $V_{3,4}$, and then $V_{2,3}$ and $V_{4,4}$ all of which are computed earlier!

The CYK Algorithm

```
1)  for i=1 to n do // Initialization
2)    V_{i,i} = {A | A → a is a rule and w_{i,i} = a]
3)  for j=2 to n do
4)    for i=1 to n-j+1 do
5)      begin
6)        V_{i,j} = {}; // Set V_{i,j} to empty set
7)        for k=1 to j-1 do
8)          V_{i,j} = V_{i,j} ∪ {A :| A → BC is a rule and
                    B ∈ V_{i,k} and C ∈ V_{k+1,j}}
```

- This algorithm has 3 nested loops with the bound for each being $O(n)$. So the overall time/work is $O(n^3)$.
- The size of the grammar factors in as a constant factor as it is independent of n – the length of the string.
- Certain special CFGs have subcubic recognition algorithms.

The CYK Algorithm in Action

- Consider the following grammar in CNF

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow BB \mid a \\ B &\rightarrow AB \mid b \end{aligned}$$

- The input string is $w = aabbb$

$i \rightarrow$	1	2	3	4	5
	a	a	b	b	b
	$\{A\}$	$\{A\}$	$\{B\}$	$\{B\}$	$\{B\}$
	$\{\}$	$\{S, B\}$	$\{A\}$	$\{A\}$	
	$\{S, B\}$	$\{A\}$	$\{S, B\}$		
	$\{A\}$	$\{S, B\}$			
	$\{S, B\}$				

- Since $S \in V_{1,5}$, this string is in $L(G)$.

The CYK Algorithm in Action

- Consider the following grammar in CNF

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow BB \mid a \\ B &\rightarrow AB \mid b \end{aligned}$$

- Let us see how we compute $V_{2,4}$
  - We need to look at $V_{2,2}$ and $V_{3,4}$
  - We need to look at $V_{2,3}$ and $V_{4,4}$

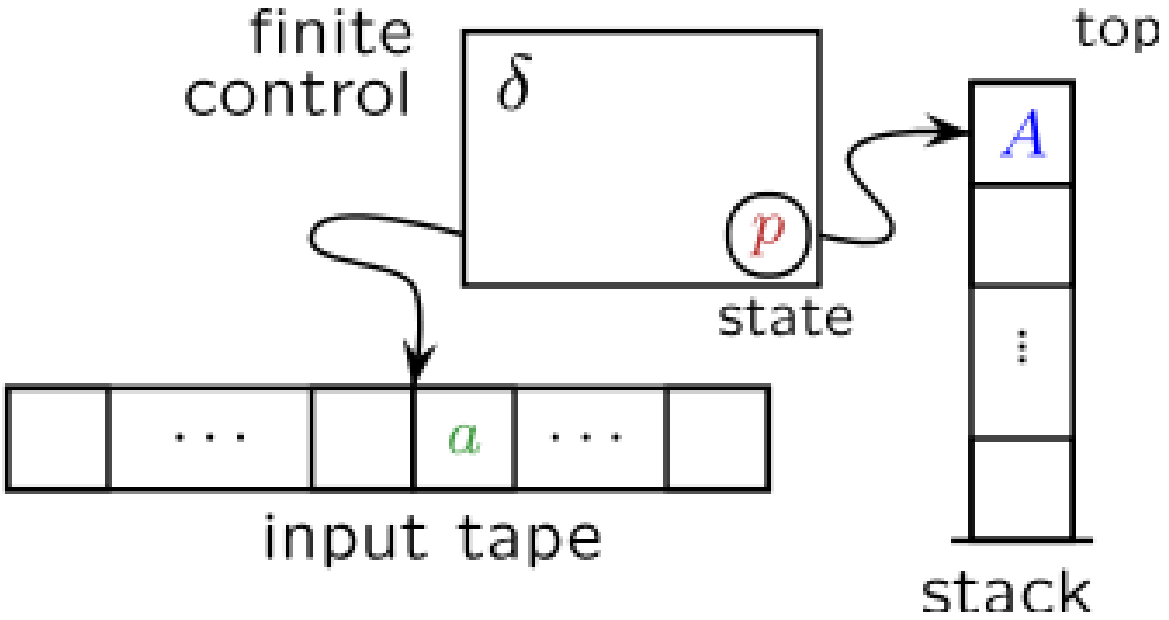
$i \rightarrow$	1	2	3	4	5
	a	a	b	b	b
	$\{A\}$	$\{A\}$	$\{B\}$	$\{B\}$	$\{B\}$
	$\{\}$	$\{S, B\}$	$\{A\}$	$\{A\}$	
	$\{S, B\}$	$\{A\}$	$\{S, B\}$		
	$\{A\}$	$\{S, B\}$			
	$\{S, B\}$				

Formal Languages, Automata and Computation

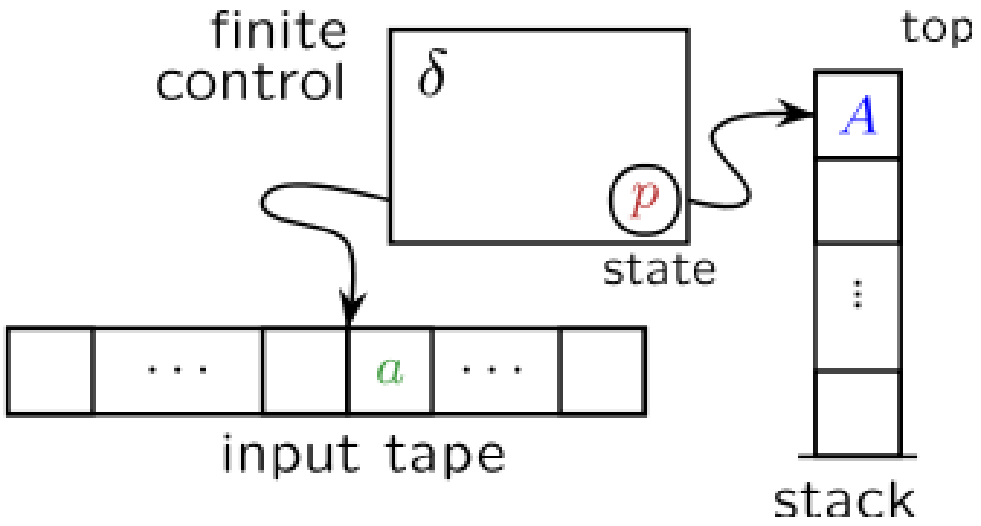
Pushdown Automata

Pushdown Automata

- Pushdown automata (PDA) are abstract automata that accept all context-free languages.
- PDAs are essentially NFAs with an additional infinite stack memory.
  - (Or NFAs are PDAs with no additional memory!)



Pushdown Automata



- Input is read left-to-right
- Control has finite memory (NFA)
- State transition depends on input and top of stack
- Control can push and pop symbols to/from the infinite stack

Pushdown Automata – Informal

- Let's look at $L = \{a^n b^n \mid n \geq 0\}$
- How can we use a stack to recognize $w \in L$?
  1. Push a special bottom of stack symbol \$ to the stack
  2. As long as you are seeing *a*'s in the input, push an *a* onto the stack.
  3. While there are *b*'s in the input AND there is a corresponding *a* on the top of the stack, pop *a* from the stack
  4. If at any point there is no *a* on the stack (hence you encounter \$), you should reject the string – not enough *a*'s!
  5. If at the end of *w*, the top of the stack is NOT \$ reject the string – not enough *b*'s.
  6. Otherwise accept the string.

Pushdown Automata – Informal

- How can we use a PDA to recognize $L = \{w \mid n_a(w) = n_b(w)\}$
- Remember how we argued that the grammar generates such strings
  - Keep track of the difference of counts
- We do something similar but using the stack.
  - Push a special bottom of stack symbol \$ to the stack
  - An *a* in the input "cancels" a *b* on the top of the stack, otherwise pushes an *a*
  - A *b* in the input "cancels" an *a* on the top of the stack, otherwise pushes a *b*
  - At the end nothing should be left on the stack except for the \$, if not reject.

Pushdown Automata –Formal Definition

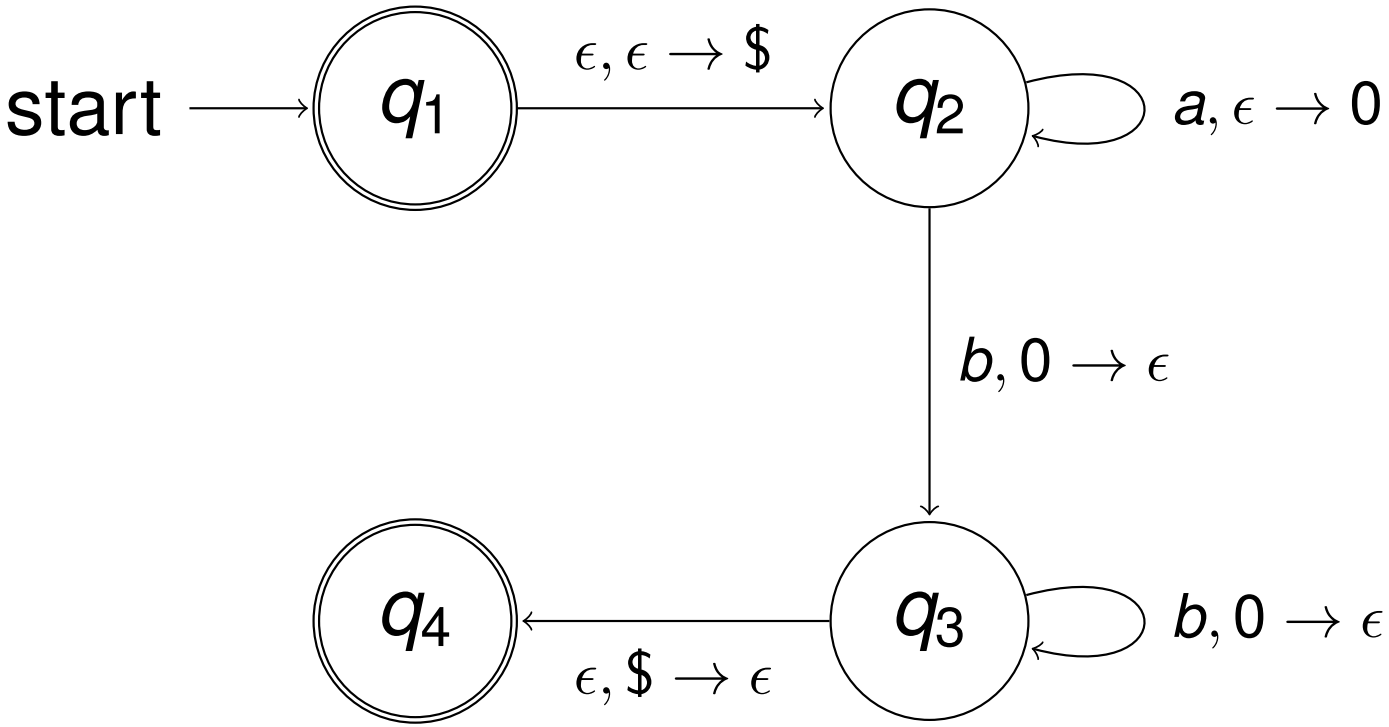
- We have two alphabets Σ for symbols of the input string and Γ for symbols for the stack. They need not be disjoint.
- Define $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$
- A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ, and F are finite sets, and
  - Q is the set of states
  - Σ is the input alphabet, Γ is the stack alphabet
  - $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the state transition function. ($\mathcal{P}(S)$is the power set of S. Earlier we used 2^S.)
  - $q_0 \in Q$ is the start state, and
  - $F \subseteq Q$ is the set of final or accepting states.

Computation on a PDA

- A PDA computes as follows:
  - Input w can be written as $w = w_1 w_2 \cdots w_m$ where each $w_i \in \Sigma_\epsilon$. So some w_i can be ϵ.
  - There is a sequence of states $r_0, r_1, \cdots, r_m$, $r_i \in Q$.
  - There is a sequence of strings $s_0, s_1, \cdots, s_m$, $s_i \in \Gamma^*$. These represent sequences of stack contents along an accepting branch of M's computation.
- $r_0 = q_0$ and $s_0 = \epsilon$.
- $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a), i = 0, 1, \cdots, m-1$
- a is popped, b is pushed, t is the rest of the stack.
  - $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$
- $r_m \in F$

Example PDA

- PDA for $L = \{a^n b^n \mid n \geq 0\}$
  - $\Sigma = \{a, b\}, \Gamma = \{0, \$\}$
  - \$ keeps track of the "bottom" of the stack

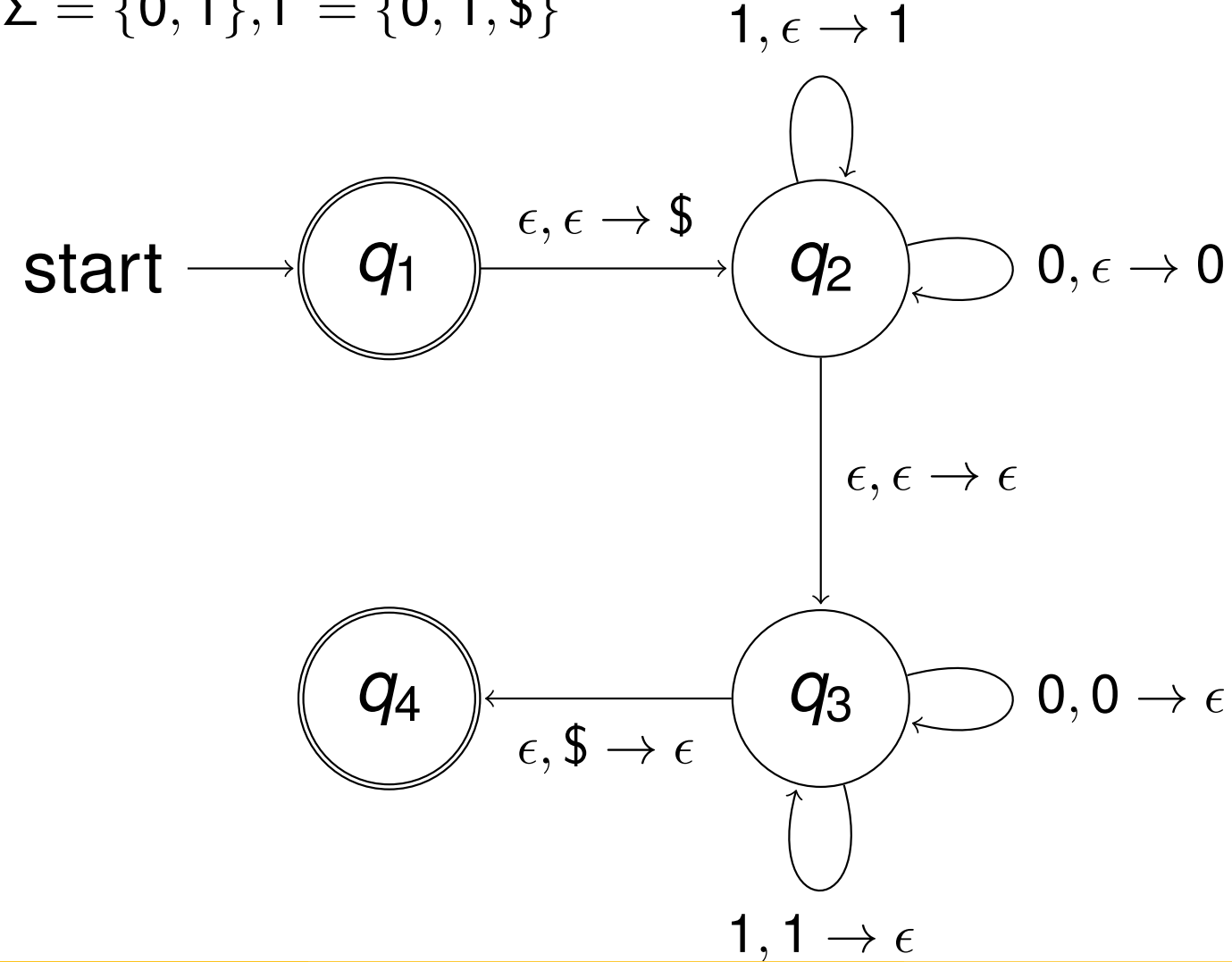


Example PDA

- PDA for $L = \{ww^R \mid w \in \{0,1\}^*\}$
- Palindromes: See `http://norvig.com/palindrome.html` for interesting examples:
- A 17,826 word palindrome starts and ends as:
  - *A man, a plan, a cameo, Zena, Bird, Mocha, Prowel, a rave, Uganda, Wait, a lobola, Argo, Goto, Koser, Ihab, Udall, a revocation, Ebarta, Muscat, eyes, Rehm, a cession, Udella, E-boat, OAS, a mirage, IPBM, a caress, Etam, . . ., a lobo, Lati, a wadna, Guevara, Lew, Orpah, Comdr, Ibanez, OEM, a canal, Panama*

Example PDA

- PDA for $L = \{ww^R \mid w \in \{0,1\}^*\}$
  - $\Sigma = \{0,1\}, \Gamma = \{0,1,\$\}$

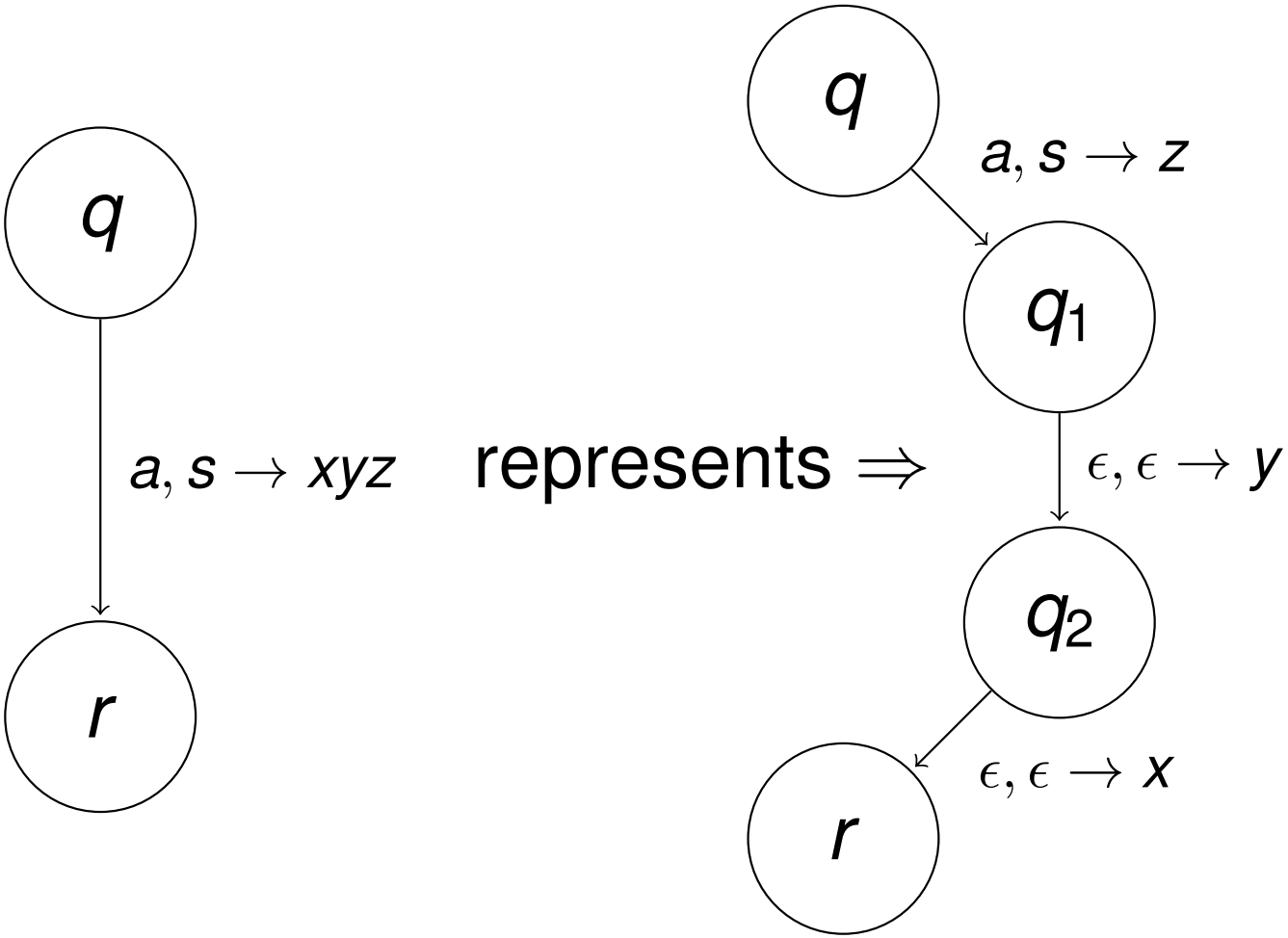


Example PDA

- Let's construct a PDA for $L = \{w \mid n_a(w) = n_b(w)\}$

PDA Shorthands

- It is usually better and more succinct to represent a series of PDA transitions using a shorthand



PDAs and CFGs

- PDAs and CFGs are equivalent in power: they both describe context-free languages.

Theorem

A language is context free if and only if some pushdown automaton recognizes it.

PDAs and CFGs

Lemma

If a language is context free, then some pushdown automaton recognizes it.

Proof Idea

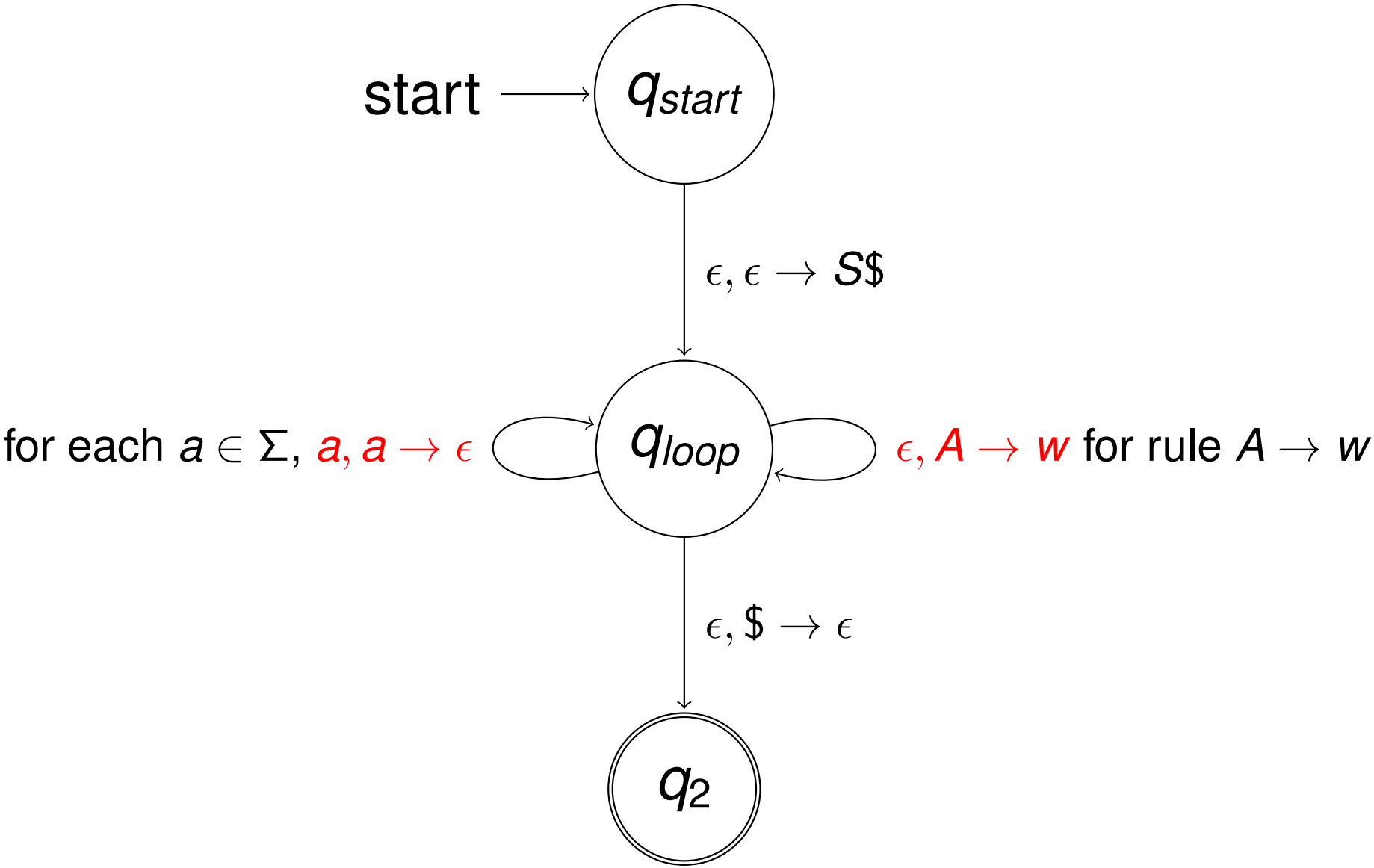
If A is a CFL, then it has a CFG G for generating it. Convert the CFG to an equivalent PDA.

- Each rule maps to a transition.

CFGs to PDAs

- We simulate the leftmost derivation of a string using a 3-state PDA with $Q = \{q_{start}, q_{loop}, q_{accept}\}$
- One transition from q_{start} pushes the start symbol S onto the stack (along with \$).
- Transitions from q_{loop} simulate either a rule expansion, or matching an input symbol.
  - $\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, w) \mid A \to w \text{ is a production in } G\}$
    - If the top of the stack is A, nondeterministically expand it in all possible ways.
  - $\delta(q_{loop}, a, a) = \{(q_{loop}, \epsilon)\}$, for all $a \in \Sigma$.
    - If the input symbol matches the top of the stack, consume the input and pop the stack.
- One transition takes the PDA from q_{loop} to q_{accept} when \$ is seen on the stack.

CFGs to PDAs


start
q_{start}
$\epsilon, \epsilon \to S\$$
for each $a \in \Sigma$, $a, a \to \epsilon$
q_{loop}
$\epsilon, A \to w$ for rule $A \to w$
$\epsilon, \$ \to \epsilon$
q_2

CFG to PDA Example

- Let's convert the following grammar for $L = \{w \mid n_a(w) = n_b(w)\}$.
  - $S \rightarrow aSb$
  - $S \rightarrow bSa$
  - $S \rightarrow SS$
  - $S \rightarrow \epsilon$

Formal Languages, Automata and Computation

Pushdown Automata

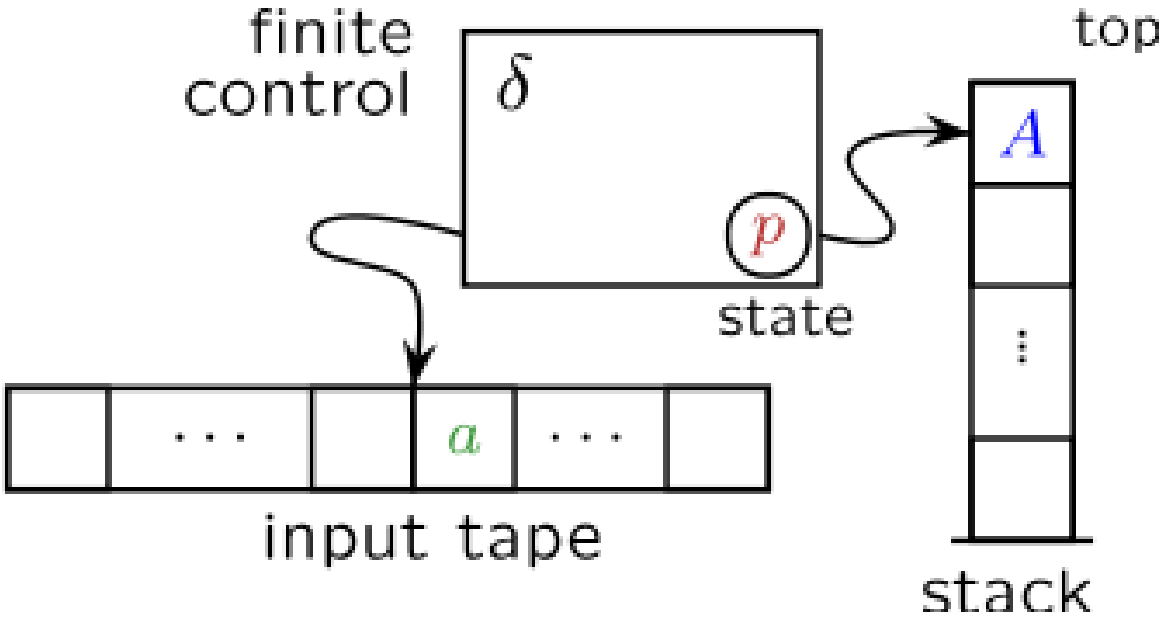
Properties of CFLs

Pushdown Automata-Summary

- Pushdown automata (PDA) are abstract automata that accept all context-free languages.

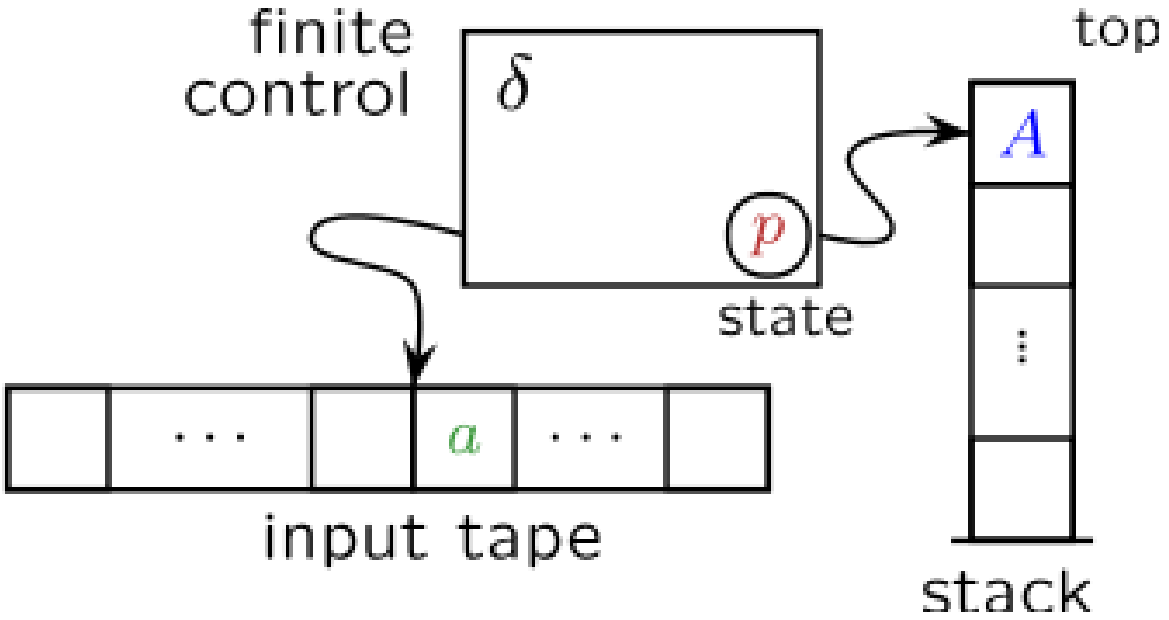
PUSHDOWN AUTOMATA-SUMMARY

- Pushdown automata (PDA) are abstract automata that accept all context-free languages.
- PDAs are essentially NFAs with an additional infinite stack memory.



Pushdown Automata-Summary

- Pushdown automata (PDA) are abstract automata that accept all context-free languages.
- PDAs are essentially NFAs with an additional infinite stack memory.
  - (Or NFAs are PDAs with no additional memory!)



PDA to CFG

Lemma

If a PDA recognizes some language, then it is context free.

Proof Idea

Create from P a CFG G that generates all strings that P accepts, i.e., G generates a string if that string takes PDA from the start state to some accepting state.

PDA TO CFG–PRELIMINARIES

Let us modify the PDA P slightly

- The PDA has a single accept state q_{accept}

PDA TO CFG–PRELIMINARIES

Let us modify the PDA P slightly

- The PDA has a single accept state q_{accept}
  - Easy – use additional $\epsilon, \epsilon \rightarrow \epsilon$ transitions.

PDA TO CFG–PRELIMINARIES

Let us modify the PDA P slightly

- The PDA has a single accept state q_{accept}
  - Easy – use additional $\epsilon, \epsilon \rightarrow \epsilon$ transitions.
- The PDA empties its stack before accepting.

PDA to CFG–Preliminaries

Let us modify the PDA P slightly

- The PDA has a single accept state q_{accept}
  - Easy – use additional $\epsilon, \epsilon \rightarrow \epsilon$ transitions.
- The PDA empties its stack before accepting.
  - Easy – add an additional loop to flush the stack.

PDA TO CFG-PRELIMINARIES

More modifications to the PDA P:

- Each transition either pushes a symbol to the stack or pops a symbol from the stack, but not both!.

PDA TO CFG-PRELIMINARIES

More modifications to the PDA P:

- Each transition either pushes a symbol to the stack or pops a symbol from the stack, but not both!.
  1. Replace each transition with a pop-push, with a two-transition sequence.

PDA TO CFG-PRELIMINARIES

More modifications to the PDA P:

- Each transition either pushes a symbol to the stack or pops a symbol from the stack, but not both!.
  1. Replace each transition with a pop-push, with a two-transition sequence.
     - For example replace $a, b \rightarrow c$ with $a, b \rightarrow \epsilon$ followed by $\epsilon, \epsilon \rightarrow c$, using an intermediate state.

PDA TO CFG-PRELIMINARIES

More modifications to the PDA P:

- Each transition either pushes a symbol to the stack or pops a symbol from the stack, but not both!.
  1. Replace each transition with a pop-push, with a two-transition sequence.
     - For example replace $a, b \rightarrow c$ with $a, b \rightarrow \epsilon$ followed by $\epsilon, \epsilon \rightarrow c$, using an intermediate state.
  2. Replace each transition with no pop-push, with a transition that pops and pushes a random symbol.

PDA to CFG-Preliminaries

More modifications to the PDA P:

- Each transition either pushes a symbol to the stack or pops a symbol from the stack, but not both!.
  1. Replace each transition with a pop-push, with a two-transition sequence.
     - For example replace $a, b \to c$ with $a, b \to \epsilon$ followed by $\epsilon, \epsilon \to c$, using an intermediate state.
  2. Replace each transition with no pop-push, with a transition that pops and pushes a random symbol.
     - For example, replace $a, \epsilon \to \epsilon$ with $a, \epsilon \to x$ followed by $\epsilon, x \to \epsilon$, using an intermediate state.

PDA TO CFG–PRELIMINARIES

- For each pair of states p and q in P, the grammar with have a variable A_{pq}.

PDA TO CFG–PRELIMINARIES

- For each pair of states p and q in P, the grammar with have a variable A_{pq}.
  - A_{pq} generates all strings that take P from p with an empty stack, to q, leaving the stack empty.

PDA TO CFG–PRELIMINARIES

- For each pair of states p and q in P, the grammar with have a variable A_{pq}.
  - A_{pq} generates all strings that take P from p with an empty stack, to q, leaving the stack empty.
  - A_{pq} also takes P from p to q, leaving the stack as it was before p!

PDA TO CFG–PRELIMINARIES

- Let x be a string that takes P from p to q with an empty stack.

PDA TO CFG–PRELIMINARIES

- Let x be a string that takes P from p to q with an empty stack.
  - First move of the PDA should involve a push! (Why?)

PDA TO CFG–PRELIMINARIES

- Let x be a string that takes P from p to q with an empty stack.
  - First move of the PDA should involve a push! (Why?)
  - Last move of the PDA should involve a pop! (Why?)

PDA TO CFG–PRELIMINARIES

- There are two cases:

PDA TO CFG–PRELIMINARIES

- There are two cases:
  1. Symbol pushed after p, is the same symbol popped just before q

PDA TO CFG–PRELIMINARIES

- There are two cases:
  1. Symbol pushed after p, is the same symbol popped just before q
  2. If not, that symbol should be popped at some point before! (Why?)

PDA TO CFG–PRELIMINARIES

- There are two cases:
  1. Symbol pushed after p, is the same symbol popped just before q
  2. If not, that symbol should be popped at some point before! (Why?)
- First case can be simulated by rule $A_{pq} \rightarrow aA_{rs}b$

PDA to CFG–Preliminaries

- There are two cases:
  1. Symbol pushed after p, is the same symbol popped just before q
  2. If not, that symbol should be popped at some point before! (Why?)
- First case can be simulated by rule $A_{pq} \rightarrow aA_{rs}b$
  - Read a, go to state r, then transit to state s somehow, and then read b.

PDA to CFG–Preliminaries

- There are two cases:
  1. Symbol pushed after p, is the same symbol popped just before q
  2. If not, that symbol should be popped at some point before! (Why?)
- First case can be simulated by rule $A_{pq} \rightarrow aA_{rs}b$
  - Read a, go to state r, then transit to state s somehow, and then read b.
- Second case can be simulated by rule $A_{pq} \rightarrow A_{pr}A_{rq}$

PDA TO CFG–PRELIMINARIES

- There are two cases:
  1. Symbol pushed after p, is the same symbol popped just before q
  2. If not, that symbol should be popped at some point before! (Why?)
- First case can be simulated by rule $A_{pq} \rightarrow aA_{rs}b$
  - Read a, go to state r, then transit to state s somehow, and then read b.
- Second case can be simulated by rule $A_{pq} \rightarrow A_{pr}A_{rq}$
  - r is the state the stack becomes empty on the way from p to q

PDA to CFG – Proof

- Assume $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$.

PDA to CFG – Proof

- Assume $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$.
- The variables of G are $\{A_{pq} \mid p, q \in Q\}$

PDA to CFG – Proof

- Assume $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$.
- The variables of G are $\{A_{pq} \mid p, q \in Q\}$
- The start variable is $A_{q_0, q_{accept}}$

PDA to CFG – Proof

- Assume $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$.
- The variables of G are $\{A_{pq} \mid p, q \in Q\}$
- The start variable is $A_{q_0, q_{accept}}$
- The rules of G are as follows:

PDA to CFG – Proof

- Assume $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$.
- The variables of G are $\{A_{pq} \mid p, q \in Q\}$
- The start variable is $A_{q_0, q_{accept}}$
- The rules of G are as follows:
  - For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if
    - $\delta(p, a, \epsilon)$ contains (r, t) and
    - $\delta(s, b, t)$ contains (q, ϵ)

    Add rule $A_{pq} \rightarrow aA_{rs}b$ to G.

PDA to CFG – Proof

- Assume $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$.
- The variables of G are $\{A_{pq} \mid p, q \in Q\}$
- The start variable is $A_{q_0, q_{accept}}$
- The rules of G are as follows:
  - For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if
    - $\delta(p, a, \epsilon)$ contains (r, t) and
    - $\delta(s, b, t)$ contains (q, ϵ)

    Add rule $A_{pq} \to aA_{rs}b$ to G.
  - For each $p, q, r \in Q$, add rule $A_{pq} \to A_{pr}A_{rq}$ to G.

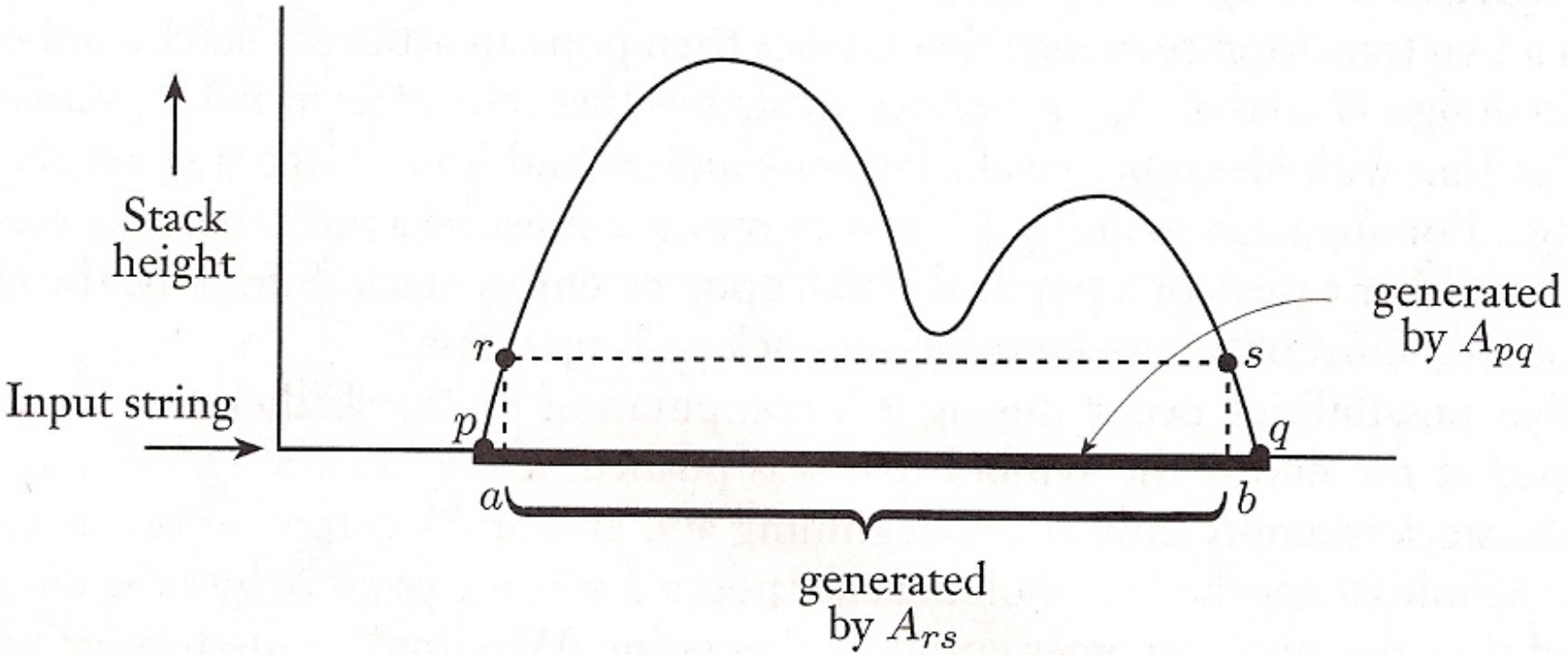
PDA to CFG – Proof

- Assume $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$.
- The variables of G are $\{A_{pq} \mid p, q \in Q\}$
- The start variable is $A_{q_0, q_{accept}}$
- The rules of G are as follows:
  - For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if
    - $\delta(p, a, \epsilon)$ contains (r, t) and
    - $\delta(s, b, t)$ contains (q, ϵ)

    Add rule $A_{pq} \rightarrow aA_{rs}b$ to G.
  - For each $p, q, r \in Q$, add rule $A_{pq} \rightarrow A_{pr}A_{rq}$ to G.
  - For each, $p \in Q$, add the rule $A_{pp} \rightarrow \epsilon$ to G.

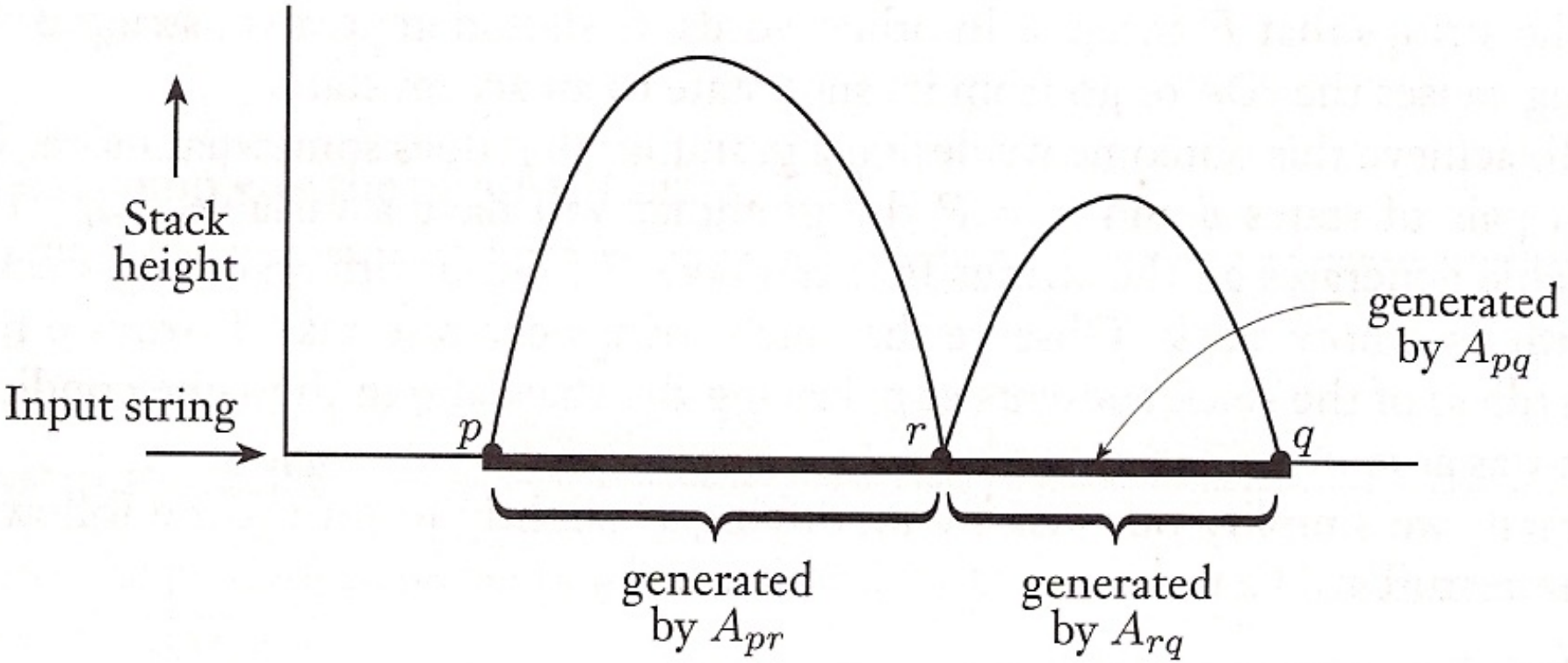
PDA to CFG Intuition

- PDA computation for $A_{pq} \rightarrow aA_{rs}b$



PDA to CFG Intuition

- PDA computation for $A_{pq} \rightarrow A_{pr}A_{rq}$



PDA to CFG Proof (cont'd)

Claim

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

PDA to CFG Proof (cont'd)

Claim

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

- Basis Case: Derivation has 1 step.

PDA to CFG Proof (cont'd)

Claim

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

- Basis Case: Derivation has 1 step.
  - This can only be possible with a production of the sort $A_{pp} \rightarrow \epsilon$. We have such a rule!

PDA to CFG Proof (cont'd)

Claim

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

- Basis Case: Derivation has 1 step.
  - This can only be possible with a production of the sort $A_{pp} \rightarrow \epsilon$. We have such a rule!
- Assume true for derivations of length at most k, $k \geq 1$

PDA to CFG Proof (cont'd)

Claim

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

- Basis Case: Derivation has 1 step.
  - This can only be possible with a production of the sort $A_{pp} \rightarrow \epsilon$. We have such a rule!
- Assume true for derivations of length at most k, $k \geq 1$
  - Suppose that $A_{pq} \stackrel{*}{\Rightarrow} x$ with $k+1$ steps. The first step in this derivation would either be $A_{pq} \rightarrow aA_{rs}b$ or $A_{pq} \rightarrow A_{pr}A_{rq}$

PDA to CFG Proof (cont'd)

Claim

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

- Basis Case: Derivation has 1 step.
  - This can only be possible with a production of the sort $A_{pp} \rightarrow \epsilon$. We have such a rule!
- Assume true for derivations of length at most k, $k \geq 1$
  - Suppose that $A_{pq} \stackrel{*}{\Rightarrow} x$ with $k+1$ steps. The first step in this derivation would either be $A_{pq} \rightarrow aA_{rs}b$ or $A_{pq} \rightarrow A_{pr}A_{rq}$
- We handle these cases separately.

PDA to CFG Proof (cont'd)

Case $A_{pq} \rightarrow aA_{rs}b$:

PDA to CFG Proof (cont'd)

Case $A_{pq} \rightarrow aA_{rs}b$:

- $A_{rs} \stackrel{*}{\Rightarrow} y$ in k steps where $x = ayb$ and by induction hypothesis, P can go from r to s with an empty stack.

PDA to CFG Proof (cont'd)

Case $A_{pq} \rightarrow aA_{rs}b$:

- $A_{rs} \stackrel{*}{\Rightarrow} y$ in k steps where $x = ayb$ and by induction hypothesis, P can go from r to s with an empty stack.
- If P pushes t onto the stack after p, after processing y it will leave t back on stack.

PDA to CFG Proof (cont'd)

Case $A_{pq} \rightarrow aA_{rs}b$:

- $A_{rs} \stackrel{*}{\Rightarrow} y$ in k steps where $x = ayb$ and by induction hypothesis, P can go from r to s with an empty stack.
- If P pushes t onto the stack after p, after processing y it will leave t back on stack.
- Reading b will have to pop the t to leave an empty stack.

PDA to CFG Proof (cont'd)

Case $A_{pq} \rightarrow aA_{rs}b$:

- $A_{rs} \stackrel{*}{\Rightarrow} y$ in k steps where $x = ayb$ and by induction hypothesis, P can go from r to s with an empty stack.
- If P pushes t onto the stack after p, after processing y it will leave t back on stack.
- Reading b will have to pop the t to leave an empty stack.
- Thus, x can bring P from p to q with an empty stack.

PDA to CFG Proof (cont'd)

Case $A_{pq} \rightarrow A_{pr}A_{rq}$

PDA to CFG Proof (cont'd)

Case $A_{pq} \rightarrow A_{pr}A_{rq}$

- Suppose $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$, where $x = yz$.

PDA to CFG Proof (cont'd)

Case $A_{pq} \to A_{pr}A_{rq}$

- Suppose $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$, where $x = yz$.
- Since these derivations are at most k steps, before p and after r we have empty stacks, and thus also after q.

PDA to CFG Proof (cont'd)

Case $A_{pq} \to A_{pr}A_{rq}$

- Suppose $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$, where $x = yz$.
- Since these derivations are at most k steps, before p and after r we have empty stacks, and thus also after q.
- Thus x can bring P from p to q with an empty stack.

PDA to CFG Proof (cont'd)

Claim

If x can bring P from p to q with empty stack, $A_{pq} \stackrel{*}{\Rightarrow} x$.

PDA to CFG Proof (cont'd)

Claim

If x can bring P from p to q with empty stack, $A_{pq} \stackrel{*}{\Rightarrow} x$.

- Basis Case: Suppose PDA takes 0 steps.

PDA to CFG Proof (cont'd)

Claim

If x can bring P from p to q with empty stack, $A_{pq} \stackrel{*}{\Rightarrow} x$.

- Basis Case: Suppose PDA takes 0 steps.
  - It should stay in the same state. Since we have a rule in the grammar $A_{pp} \rightarrow \epsilon$, $A_{pp} \stackrel{*}{\Rightarrow} \epsilon$.

PDA to CFG Proof (cont'd)

Claim

If x can bring P from p to q with empty stack, $A_{pq} \stackrel{*}{\Rightarrow} x$.

- Basis Case: Suppose PDA takes 0 steps.
  - It should stay in the same state. Since we have a rule in the grammar $A_{pp} \rightarrow \epsilon$, $A_{pp} \stackrel{*}{\Rightarrow} \epsilon$.
- Assume true for all computations of P of length at most k, $k \geq 0$.

PDA to CFG Proof (cont'd)

Claim

If x can bring P from p to q with empty stack, $A_{pq} \stackrel{*}{\Rightarrow} x$.

- Basis Case: Suppose PDA takes 0 steps.
  - It should stay in the same state. Since we have a rule in the grammar $A_{pp} \rightarrow \epsilon$, $A_{pp} \stackrel{*}{\Rightarrow} \epsilon$.
- Assume true for all computations of P of length at most k, $k \geq 0$.
  - Suppose with x, P can go from p to q with an empty stack. Either the stack is empty only at the beginning and at the end, or it becomes empty elsewhere, too.

PDA to CFG Proof (cont'd)

Claim

If x can bring P from p to q with empty stack, $A_{pq} \stackrel{*}{\Rightarrow} x$.

- Basis Case: Suppose PDA takes 0 steps.
  - It should stay in the same state. Since we have a rule in the grammar $A_{pp} \rightarrow \epsilon$, $A_{pp} \stackrel{*}{\Rightarrow} \epsilon$.
- Assume true for all computations of P of length at most k, $k \geq 0$.
  - Suppose with x, P can go from p to q with an empty stack. Either the stack is empty only at the beginning and at the end, or it becomes empty elsewhere, too.
- We handle these two cases separately.

PDA to CFG Proof (cont'd)

Case: Stack is empty only at the beginning and at the end of a derivation of length $k + 1$

PDA to CFG Proof (cont'd)

Case: Stack is empty only at the beginning and at the end of a derivation of length $k + 1$

- Suppose $x = ayb$. a and b are consumed at the beginning and at the end of the computation (with t being pushed and popped).

PDA to CFG Proof (cont'd)

Case: Stack is empty only at the beginning and at the end of a derivation of length $k + 1$

- Suppose $x = ayb$. a and b are consumed at the beginning and at the end of the computation (with t being pushed and popped).
- P takes $k - 2$ steps on y.

PDA to CFG Proof (cont'd)

Case: Stack is empty only at the beginning and at the end of a derivation of length $k + 1$

- Suppose $x = ayb$. a and b are consumed at the beginning and at the end of the computation (with t being pushed and popped).
- P takes $k - 2$ steps on y.
- By hypothesis, $A_{rs} \stackrel{*}{\Rightarrow} y$ where $(r, t) \in \delta(q, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$.

PDA to CFG Proof (cont'd)

Case: Stack is empty only at the beginning and at the end of a derivation of length $k + 1$

- Suppose $x = ayb$. a and b are consumed at the beginning and at the end of the computation (with t being pushed and popped).
- P takes $k - 2$ steps on y.
- By hypothesis, $A_{rs} \stackrel{*}{\Rightarrow} y$ where $(r, t) \in \delta(q, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$.
- Thus, using rule $A_{pq} \rightarrow aA_{rs}b$, $A_{pq} \stackrel{*}{\Rightarrow} x$.

PDA to CFG Proof (cont'd)

Case: Stack becomes empty at some intermediate stage in the computation of x

PDA to CFG Proof (cont'd)

Case: Stack becomes empty at some intermediate stage in the computation of x

- Suppose $x = yz$, such that P has the stack empty after consuming y.

PDA to CFG Proof (cont'd)

Case: Stack becomes empty at some intermediate stage in the computation of x

- Suppose $x = yz$, such that P has the stack empty after consuming y.
- By induction hypothesis $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$ since P takes at most k steps on y and z.

PDA to CFG Proof (cont'd)

Case: Stack becomes empty at some intermediate stage in the computation of x

- Suppose $x = yz$, such that P has the stack empty after consuming y.
- By induction hypothesis $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$ since P takes at most k steps on y and z.
- Since rule $A_{pq} \rightarrow A_{pr}A_{rq}$ is in the grammar, $A_{pq} \stackrel{*}{\Rightarrow} x$.

Regular Languages are Context Free

Corollary

Every regular language is context free.

Regular Languages are Context Free

Corollary

Every regular language is context free.

Proof.

Since a regular language L is recognized by a DFA and every DFA is a PDA that ignores it stack, there is a CFG for L □

Regular Languages are Context Free

Corollary

Every regular language is context free.

Proof.

Since a regular language L is recognized by a DFA and every DFA is a PDA that ignores it stack, there is a CFG for L □

- Right-linear grammars
- Left-linear grammars

Non-Context-free Languages

- There are non-context-free languages.

Non-Context-free Languages

- There are non-context-free languages.
- For example $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.

Non-Context-free Languages

- There are non-context-free languages.
- For example $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.
  - Intuitively, once the PDA reads the *a*'s and then matches the *b*'s, it "forgets" what the *n* was, so can not properly check the *c*'s.

Non-Context-free Languages

- There are non-context-free languages.
- For example $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.
  - Intuitively, once the PDA reads the *a*'s and then matches the *b*'s, it "forgets" what the *n* was, so can not properly check the *c*'s.
- There is an analogue of the Pumping Lemma we studied earlier for regular languages.

Non-Context-free Languages

- There are non-context-free languages.
- For example $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.
  - Intuitively, once the PDA reads the *a*'s and then matches the *b*'s, it "forgets" what the *n* was, so can not properly check the *c*'s.
- There is an analogue of the Pumping Lemma we studied earlier for regular languages.
  - It states that there is a pumping length, such that all longer strings can be pumped.

Non-Context-free Languages

- There are non-context-free languages.
- For example $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.
  - Intuitively, once the PDA reads the *a*'s and then matches the *b*'s, it "forgets" what the *n* was, so can not properly check the *c*'s.
- There is an analogue of the Pumping Lemma we studied earlier for regular languages.
  - It states that there is a pumping length, such that all longer strings can be pumped.
  - For regular languages, we related the pumping length to the number of states of the DFA.

Non-Context-free Languages

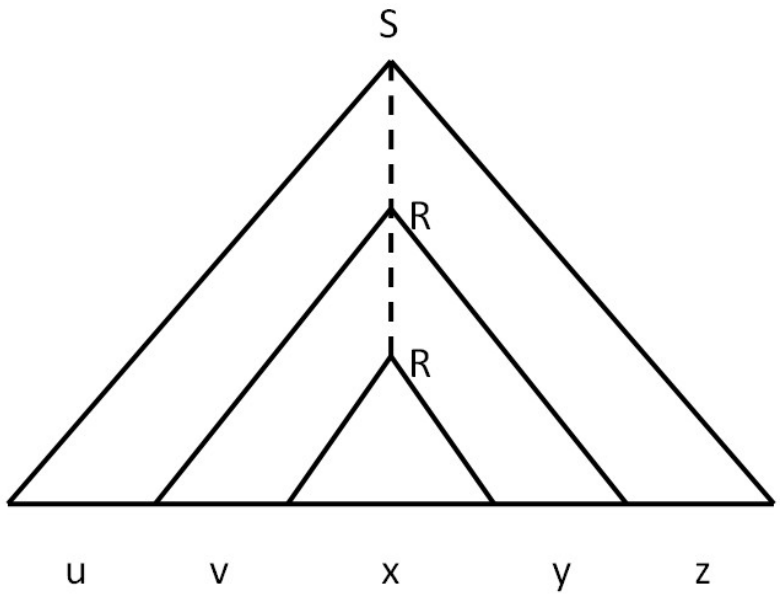
- There are non-context-free languages.
- For example $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.
  - Intuitively, once the PDA reads the *a*'s and then matches the *b*'s, it "forgets" what the *n* was, so can not properly check the *c*'s.
- There is an analogue of the Pumping Lemma we studied earlier for regular languages.
  - It states that there is a pumping length, such that all longer strings can be pumped.
  - For regular languages, we related the pumping length to the number of states of the DFA.
  - For CFLs, we relate the pumping length to the properties of the grammar!.

PUMPING LEMMA FOR CFLS - INTUITION

- Let s be a "sufficiently long" string in L.

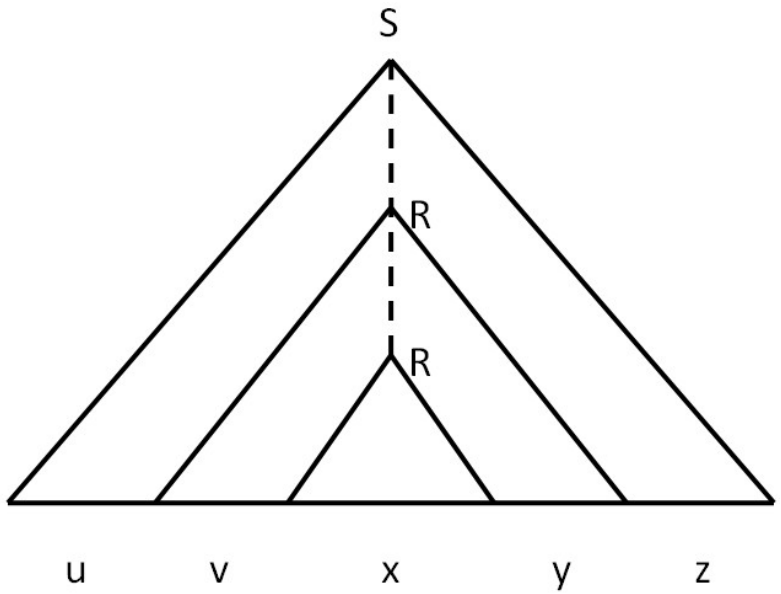
Pumping Lemma for CFLs - Intuition

- Let s be a "sufficiently long" string in L.
- $s = uvxyz$ should have a parse tree of the following sort:



Pumping Lemma for CFLs - Intuition

- Let s be a "sufficiently long" string in L.
- $s = uvxyz$ should have a parse tree of the following sort:



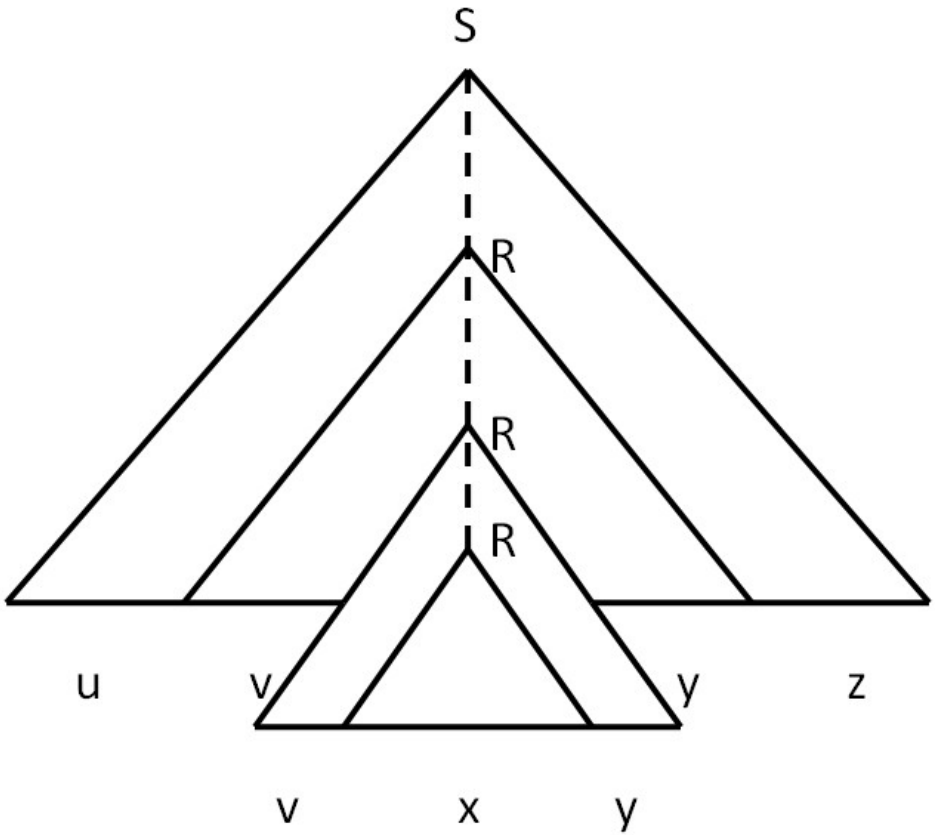
- Some variable R must repeat somewhere on the path from S to some leaf. (Why?)

Pumping Lemma for CFLs - Intuition

- Then the string $s' = uvvxyyz$, should also be in the language.

PUMPING LEMMA FOR CFLS - INTUITION

- Then the string $s' = uvvxyyz$, should also be in the language.

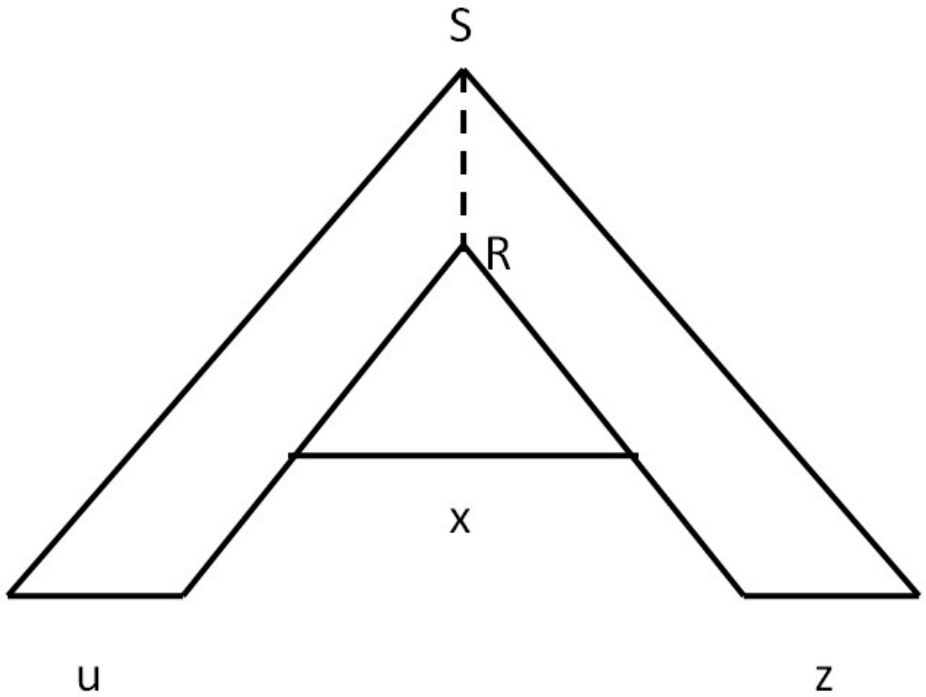


Pumping Lemma for CFLs - Intuition

- Also the string $s'' = uxz$, should also be in the language.

PUMPING LEMMA FOR CFLS - INTUITION

- Also the string $s'' = uxz$, should also be in the language.



Pumping Lemma for CFLs

Lemma

If L is a CFL, then

- *there is a number p (the pumping length) such that*
- *if s is any string in L of length at least p,*
- *then s can be divided into 5 pieces $s = uvxyz$ satisfying the conditions:*

Pumping Lemma for CFLs

Lemma

If L is a CFL, then

- *there is a number p (the pumping length) such that*
- *if s is any string in L of length at least p,*
- *then s can be divided into 5 pieces $s = uvxyz$ satisfying the conditions:*
  1. $|vy| > 0$

Pumping Lemma for CFLs

Lemma

If L is a CFL, then

- *there is a number p (the pumping length) such that*
- *if s is any string in L of length at least p,*
- *then s can be divided into 5 pieces $s = uvxyz$ satisfying the conditions:*
  1. $|vy| > 0$
  2. $|vxy| \leq p$

Pumping Lemma for CFLs

Lemma

If L is a CFL, then

- *there is a number p (the pumping length) such that*
- *if s is any string in L of length at least p,*
- *then s can be divided into 5 pieces $s = uvxyz$ satisfying the conditions:*
  1. $|vy| > 0$
  2. $|vxy| \leq p$
  3. *for each* $i \geq 0$, $uv^i xy^i z \in L$

Pumping Lemma for CFLs

Lemma

If L is a CFL, then

- *there is a number p (the pumping length) such that*
- *if s is any string in L of length at least p,*
- *then s can be divided into 5 pieces $s = uvxyz$ satisfying the conditions:*
  1. $|vy| > 0$
  2. $|vxy| \leq p$
  3. *for each* $i \geq 0$, $uv^i xy^i z \in L$

- Either v or y is not ϵ otherwise, it would be trivially true.

Proof – The Pumping Length

- Let G be the grammar for L. Let b be the maximum number of symbols of any rule in G.

Proof – The Pumping Length

- Let G be the grammar for L. Let b be the maximum number of symbols of any rule in G.
  - Assume b is at least 2, that is every grammar has some rule with at least 2 symbols on the RHS.

Proof – The Pumping Length

- Let G be the grammar for L. Let b be the maximum number of symbols of any rule in G.
  - Assume b is at least 2, that is every grammar has some rule with at least 2 symbols on the RHS.
- In any parse tree, a node can have at most b children.

Proof – The Pumping Length

- Let G be the grammar for L. Let b be the maximum number of symbols of any rule in G.
  - Assume b is at least 2, that is every grammar has some rule with at least 2 symbols on the RHS.
- In any parse tree, a node can have at most b children.
  - At most b^h leaves are within h steps of the start variable.

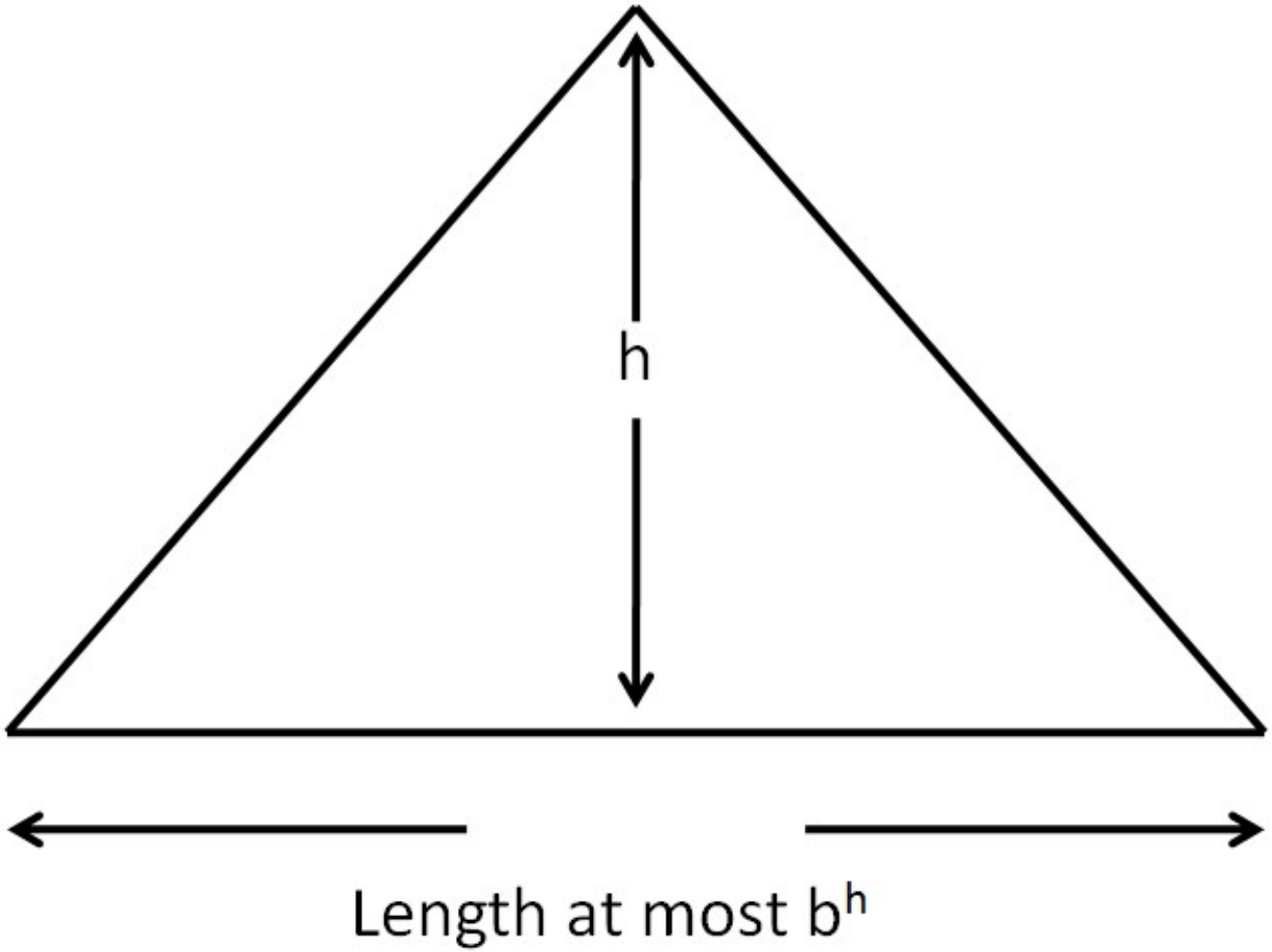
Proof – The Pumping Length

- Let G be the grammar for L. Let b be the maximum number of symbols of any rule in G.
  - Assume b is at least 2, that is every grammar has some rule with at least 2 symbols on the RHS.
- In any parse tree, a node can have at most b children.
  - At most b^h leaves are within h steps of the start variable.
- If the parse tree has height h, the length of the string generated is at most b^h.

Proof – The Pumping Length

- Let G be the grammar for L. Let b be the maximum number of symbols of any rule in G.
  - Assume b is at least 2, that is every grammar has some rule with at least 2 symbols on the RHS.
- In any parse tree, a node can have at most b children.
  - At most b^h leaves are within h steps of the start variable.
- If the parse tree has height h, the length of the string generated is at most b^h.
- Conversely, if the string is at least $b^h + 1$ long, each of its parse trees must be at least $h + 1$ high.

Proof – The Pumping Length



Proof - The Pumping Length

- Let $|V|$ be the number of variables in G.

Proof - The Pumping Length

- Let $|V|$ be the number of variables in G.
- We set the pumping length $p = b^{|V|+1}$.

Proof - The Pumping Length

- Let $|V|$ be the number of variables in G.
- We set the pumping length $p = b^{|V|+1}$.
- If s is a string in L and $|s| \geq p$, its parse tree must be at least $|V| + 1$ high.

Proof - The Pumping Length

- Let $|V|$ be the number of variables in G.
- We set the pumping length $p = b^{|V|+1}$.
- If s is a string in L and $|s| \geq p$, its parse tree must be at least $|V| + 1$ high.
  - $b^{|V|+1} \geq b^{|V|} + 1$

Proof - How to Pump a String

- Let τ be the parse tree of s that has the smallest number of nodes. τ must be at least $|V| + 1$ high.

Proof - How to Pump a String

- Let τ be the parse tree of s that has the smallest number of nodes. τ must be at least $|V| + 1$ high.
- This means some path from the root to some leaf has length at least $|V| + 1$.

Proof - How to Pump a String

- Let τ be the parse tree of s that has the smallest number of nodes. τ must be at least $|V| + 1$ high.
- This means some path from the root to some leaf has length at least $|V| + 1$.
- So the path has at least $|V| + 2$ nodes: 1 terminal and at least $|V| + 1$ variables.

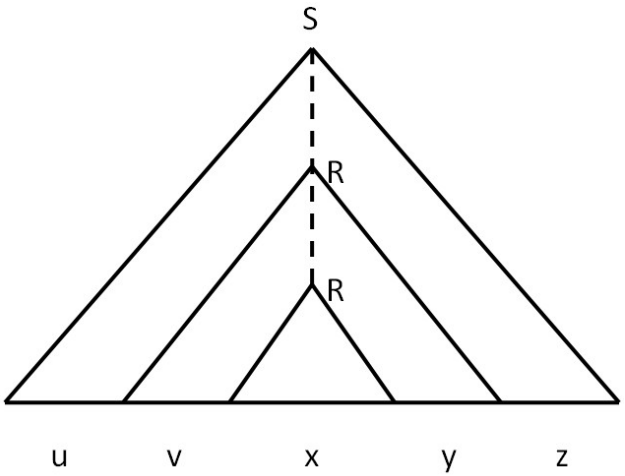
Proof - How to Pump a String

- Let τ be the parse tree of s that has the smallest number of nodes. τ must be at least $|V| + 1$ high.
- This means some path from the root to some leaf has length at least $|V| + 1$.
- So the path has at least $|V| + 2$ nodes: 1 terminal and at least $|V| + 1$ variables.
- Some variable R must appear more than once on that path (Pigeons!)

Proof - How to Pump a String

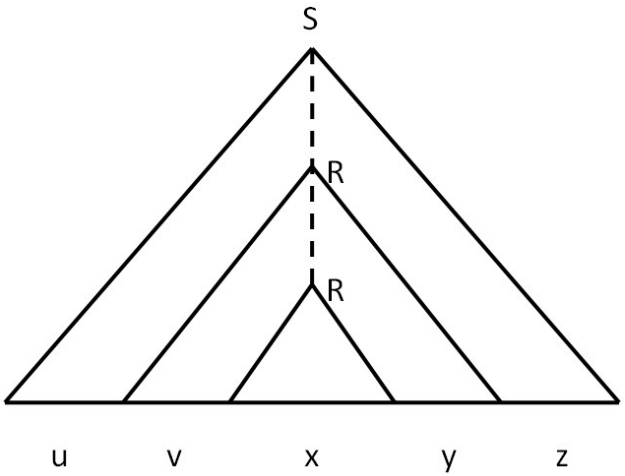
- Let τ be the parse tree of s that has the smallest number of nodes. τ must be at least $|V| + 1$ high.
- This means some path from the root to some leaf has length at least $|V| + 1$.
- So the path has at least $|V| + 2$ nodes: 1 terminal and at least $|V| + 1$ variables.
- Some variable R must appear more than once on that path (Pigeons!)
  - Choose R as the variable that repeats among the lowest $|V| + 1$ variables on this path.

Proof - How to Choose a String



- We divide *s* into *uvxyz* according to this figure.

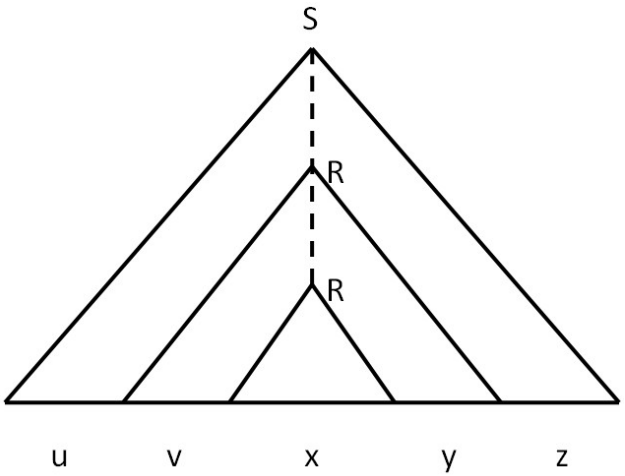
Proof - How to Choose a String



- We divide *s* into *uvxyz* according to this figure.

- Upper *R* generates *vxy* while the lower *R* generates *x*.

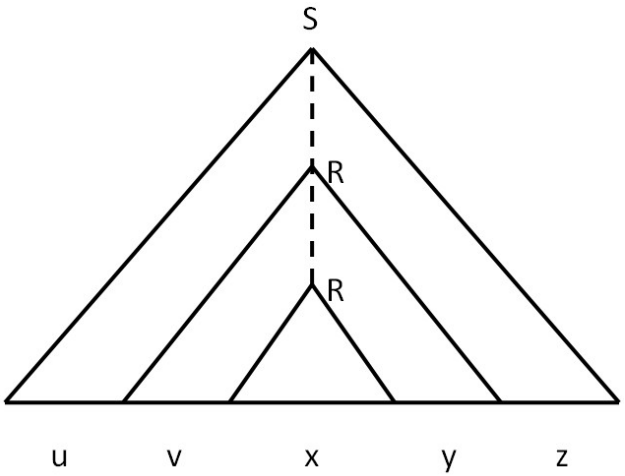
Proof - How to Choose a String



- We divide *s* into *uvxyz* according to this figure.

- Upper *R* generates *vxy* while the lower *R* generates *x*.
- Since the same variable generates both subtrees, they are interchangeable!

Proof - How to Choose a String



- We divide s into $uvxyz$ according to this figure.

- Upper R generates vxy while the lower R generates x.
- Since the same variable generates both subtrees, they are interchangeable!
- So all strings of the form uv^ixy^iz should also be in the language for $i \geq 0$.

Proof – How to Choose a String

- We must make sure both v and y are not both ϵ.

Proof – How to Choose a String

- We must make sure both v and y are not both ϵ.
- If they were, then τ would not be smallest tree for s.

Proof – How to Choose a String

- We must make sure both v and y are not both ϵ.
- If they were, then τ would not be smallest tree for s.
  - We could get a smaller tree for s by substituting the smaller tree!

Proof – How to Choose a String

- We must make sure both v and y are not both ϵ.
- If they were, then τ would not be smallest tree for s.
  - We could get a smaller tree for s by substituting the smaller tree!
- $R \stackrel{*}{\Rightarrow} vxy$.

Proof – How to Choose a String

- We must make sure both v and y are not both ϵ.
- If they were, then τ would not be smallest tree for s.
  - We could get a smaller tree for s by substituting the smaller tree!
- $R \stackrel{*}{\Rightarrow} vxy$.
- We chose R so that both its occurrences were within the last $|V| + 1$ variables on the path.

Proof – How to Choose a String

- We must make sure both v and y are not both ϵ.
- If they were, then τ would not be smallest tree for s.
  - We could get a smaller tree for s by substituting the smaller tree!
- $R \stackrel{*}{\Rightarrow} vxy$.
- We chose R so that both its occurrences were within the last $|V| + 1$ variables on the path.
- We chose the longest path in the tree, so the subtree for $R \stackrel{*}{\Rightarrow} vxy$ is at most $|V| + 1$ high.

Proof – How to Choose a String

- We must make sure both v and y are not both ϵ.
- If they were, then τ would not be smallest tree for s.
  - We could get a smaller tree for s by substituting the smaller tree!
- $R \stackrel{*}{\Rightarrow} vxy$.
- We chose R so that both its occurrences were within the last $|V| + 1$ variables on the path.
- We chose the longest path in the tree, so the subtree for $R \stackrel{*}{\Rightarrow} vxy$ is at most $|V| + 1$ high.
- A tree of this height can generate a string of length at most $b^{|V|+1} = p$.

Formal Languages, Automata and Computation

Pumping Lemma

Properties of CFLs

Summary

- Context-free Languages and Context-free Grammars
- Pushdown Automata
- PDAs accept all languages CFGs generate.
- CFGs generate all languages that PDAs accept.
- There are languages which are NOT context free.

Pumping Lemma for CFLs

Lemma

If L is a CFL, then there is a number p (the pumping length) such that if s is any string in L of length at least p, then s can be divided into 5 pieces $s = uvxyz$ satisfying the conditions:

1. $|vy| > 0$
2. $|vxy| \leq p$
3. *for each* $i \geq 0$, $uv^i xy^i z \in L$

- The pumping length is determined by the number of variables the grammar for *L* has.

Application of the Pumping Lemma

- Just as for regular languages we employ the pumping lemma in a two-player game setting.
- If a language violates the CFL pumping lemma, then it can not be a CFL.
- Two Player Proof Strategy:
  - Opponent picks p, the pumping length
  - Given p, we pick s in L such that $|s| \geq p$. We are free to choose s as we please, as long as those conditions are satisfied.
  - Opponent picks $s = uvxyz$ - the decomposition subject to $|vxy| \leq p$ and $|vy| \geq 1$.
  - We try to pick an i such that $uv^i xy^i z \notin L$
  - If for all possible decompositions the opponent can pick, we can find an i, then L is not context-free.

Using Pumping Lemma – Example-1

- Consider the language $L = \{a^n b^n c^n \mid n \geq 0\}$
- Opponent picks p.
- We pick $s = a^p b^p c^p$. Clearly $|s| \geq p$.
- Opponent may pick the string partitioning in a number of ways.
- Let's look at each of these possibilities:

Using Pumping Lemma–Example 1

- Cases 1,2 and 3: vxy contains symbols of only one kind
  1. Only a's: $\underbrace{a\cdots a}_{u}\underbrace{a\cdots a}_{vxy}\underbrace{a\cdots ab\cdots bc\cdots c}_{z}$
  2. Only b's: $\underbrace{a\cdots ab}_{u}\underbrace{b\cdots b}_{vxy}\underbrace{b\cdots bc\cdots c}_{z}$
  3. Only c's: $\underbrace{a\cdots ab\cdots bc}_{u}\underbrace{c\cdots c}_{vxy}\underbrace{c\cdots c}_{z}$
- Pumping v and y will introduce more symbols of one type into the string.
- The resulting strings will not be in the language.

Using Pumping Lemma–Example 1

- Cases 4 and 5: *vxy* contains two symbols – crosses symbol boundaries.
  1. Only *a*'s and *b*'s: $\underbrace{a \cdots a}_{u}\underbrace{a \cdots ab \cdots b}_{vxy}\underbrace{b \cdots bc \cdots c}_{z}$
  2. Only *b*'s and *c*s: $\underbrace{a \cdots ab \cdots b}_{u}\underbrace{b \cdots c}_{vxy}\underbrace{c \cdots c}_{z}$
- Note that *vxy* has length at most *p* so can not have 3 different symbols.
- Pumping *v* and *y* will both upset the symbol counts and the symbol patterns.
- The resulting strings will not be in the language.

Using Pumping Lemma–Example 2

- Consider the language $L = \{a^n \mid n \text{ is prime}\}$
- Opponent picks (prime) p.
- We pick $s = a^p$. Clearly $|s| \geq p$.
- Opponent may pick any partitioning $s = uvxyz$.
  - Let $m = |uxz|$ for the partitioning selected, that is, the length of everything else but v and y.
  - Any pumped string $uv^i xy^i z$ will have length $m + i(p - m)$.
  - We choose $i = p + 1$.
  - The pumped string has length $m + (p + 1)(p - m)$. But:

$$\begin{aligned} m + (p+1)(p-m) &= m + p^2 - pm + p - m \\ &= p^2 + p - pm \\ &= p(p - m + 1) \end{aligned}$$

which is not prime since both p and $p - m + 1$ are greater than 1. (Note $0 \leq m \leq p - 1$)

Closure Properties of Context-free Languages

- Context-free languages are closed under
  - Union
  - Concatenation
  - Star Closure
  - Intersection with a regular language
- We will provide very informal arguments for these.

Closure Properties of CFLs-Union

- Let G_1 and G_2 be the grammars with start variables S_1 and S_2, variables V_1 and V_2, and rules R_1 and R_2.
- Rename the variables in V_2 if they are also used in V_1
- The grammar G for $L = L(G_1) \cup L(G_2)$ has
  - $V = V_1 \cup V_2 \cup \{S\}$ (S is the new start symbol $S \notin V_1$ and $S \notin V_2$
  - $R = R_1 \cup R_2 \cup \{S \to S_1 \mid S_2\}$

Closure Properties of CFLs – Concatenation

- Let G_1 and G_2 be the grammars with start variables S_1 and S_2, variables V_1 and V_2, and rules R_1 and R_2.
- Rename the variables in V_2 if they are also used in V_1
- The grammar G for $L = \{wv \mid w \in L(G_1), v \in L(G_2)\}$ has
  - $V = V_1 \cup V_2 \cup \{S\}$ (S is the new start symbol $S \notin V_1$ and $S \notin V_2$
  - $R = R_1 \cup R_2 \cup \{S \to S_1 S_2\}$

Closure Properties of CFLs – Star Closure

- Let G_1 be the grammar with start variable S_1, variables V_1, rules R_1.
- The grammar G for $L = \{w \mid w \in L(G_1)^*\}$ has
  - $V = V_1 \cup \{S\}$ (S is the new start symbol $S \notin V_1$).
  - $R = R_1 \cup \{S \to S_1 S \mid \epsilon\}$

Closure Properties of CFLs – Intersection with a Regular Language

- Let P be the PDA for the CFL L_{cfl} and M be the *DFA* for the regular language $L_{regular}$
- We have a procedure for building the cross-product PDA from P and M.
  - Very similar to the cross-product construction for DFAs.
  - Details are not terribly interesting. (Perhaps later.)

Closure Properties of CFLs

- CFLs are NOT closed under intersection.
  - $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ is a CFL.
  - $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$ is a CFL.
  - $L = L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ is NOT a CFL.
- CFLs are not closed under complementation.
  - $L = \{ww \mid w \in \Sigma^*\}$ is NOT a CFL (Prove it using pumping lemma!)
  - $\overline{L}$ is actually a CFL and $L = L_1 \cup L_2$
    - $\overline{L}$ has all strings of odd length (L_1)
    - $\overline{L}$ has all strings where at least one pair of symbols $n/2$ apart are different (n length of the string!) (L_2)
    - $S \to aA \mid bA \mid a \mid b$
      $A \to aS \mid bS$
      generates L_1

      $S \to AB \mid BA$
      $A \to ZAZ \mid a$
      $B \to ZBZ \mid b$
      $Z \to a \mid b$
      generates L_2

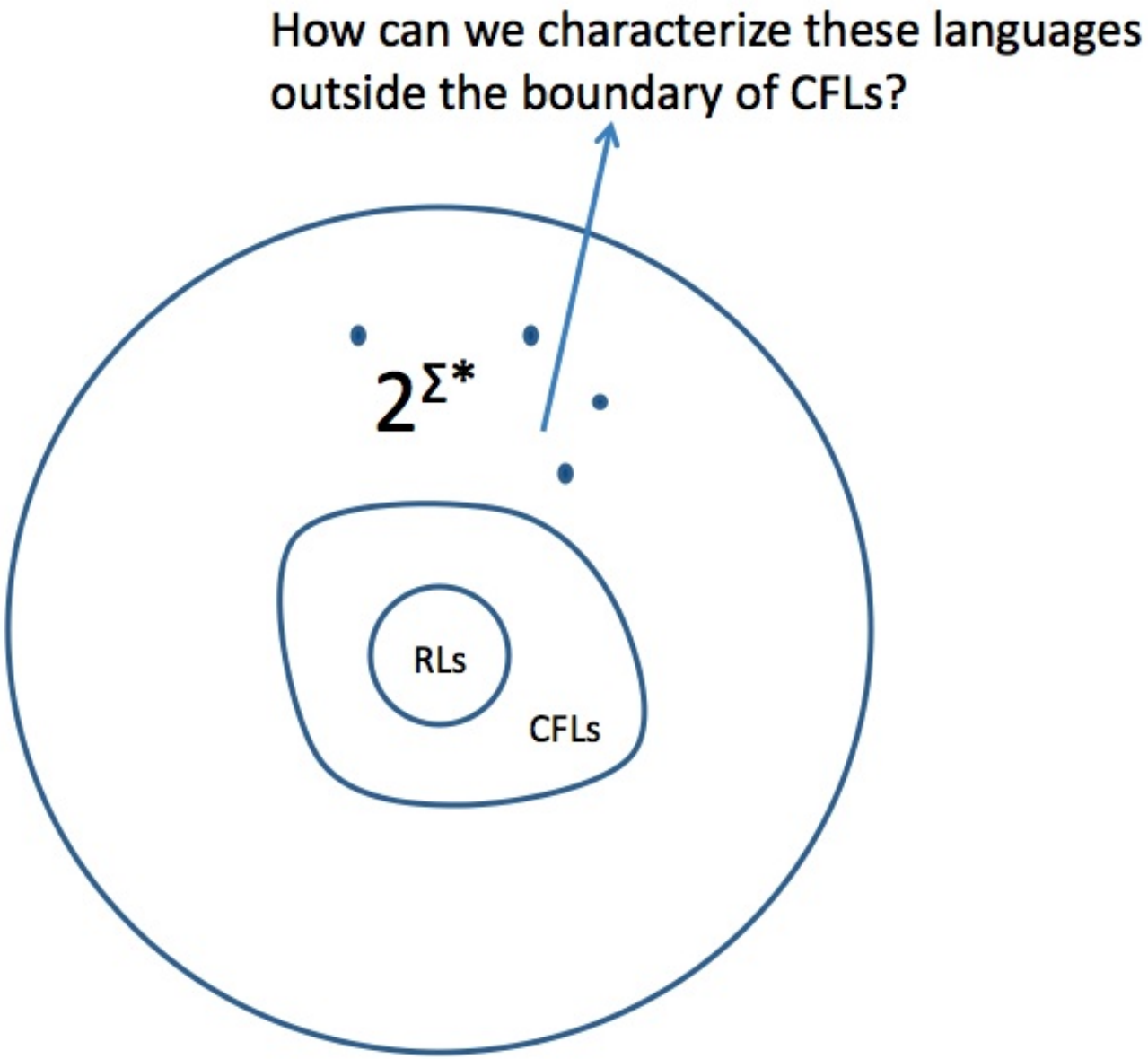
CFL Closure Properties in Action

- Is $L = \{a^n b^n \mid n \geq 0, n \neq 100\}$ a CFL?
  - $L = \underbrace{\{a^n b^n \mid n \geq 0\}}_{CFL} \cap \underbrace{(L(a^* b^*) - \{a^{100} b^{100}\})}_{RL}$
  - The intersection of a CFL and a RL is a CFL!

- Is $L = \{w \mid w \in \{a, b, c\}^* \text{ and } n_a(w) = n_b(w) = n_c(w)\}$ a CFL?
  - $\underbrace{L}_{CFL?} \cap \underbrace{L(a^* b^* c^*)}_{RL} = \underbrace{\{a^n b^n c^n \mid n \geq 0\}}_{Not\ CFL}$
  - Thus L is NOT a CFL.

Moving beyond the Milky Way

What other kinds of languages are out there?



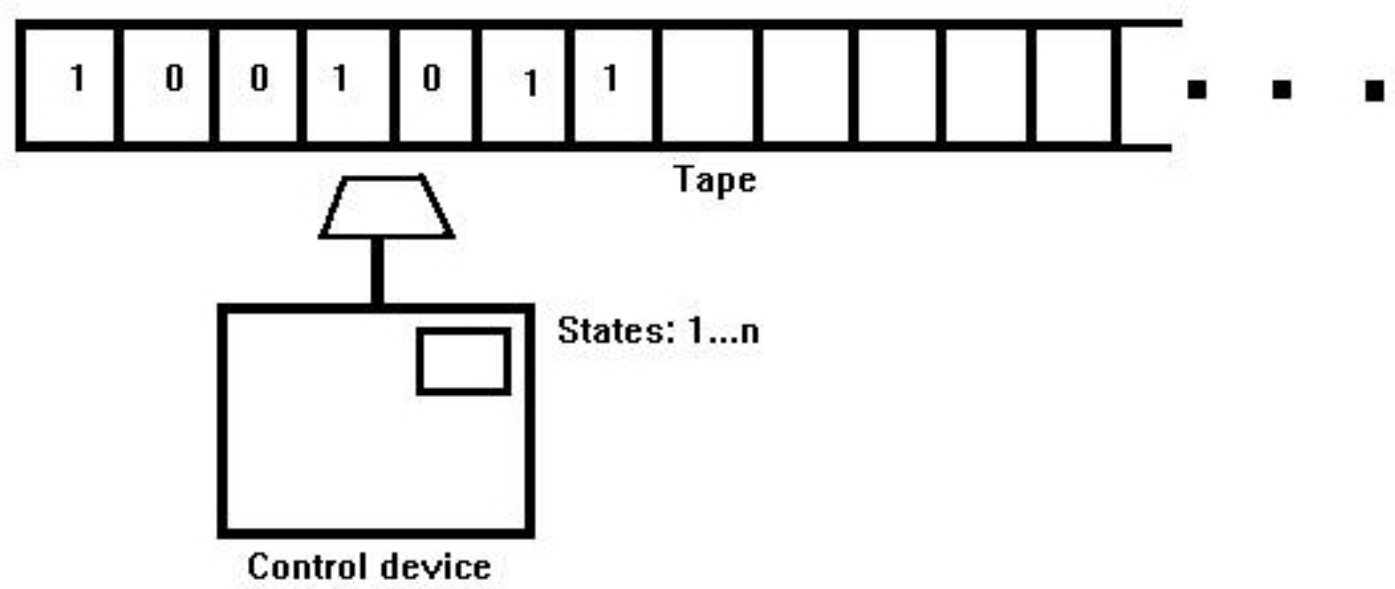
Formal Languages, Automata and Computation

Turing Machines

TURING MACHINES-SYNOPSIS

- We now turn to a much more powerful model of computation called Turing Machines (TM).
- TMs are similar to a finite automaton, but a TM has an unlimited and unrestricted memory.
- A TM is a much more accurate model of a general purpose computer.
- Bad News: Even a TM can not solve certain problems.
- Such problems are beyond theoretical limits of computation.

TURING MACHINES



Turing Machines vs Finite Automata

- A TM can both read from the tape and write on the tape.
- The read-write head can move both to the left (L) and to the right (R).
- The tape is infinite (to the right).
- The states for rejecting and accepting take effect immediately (not at the end of input.)

How does a TM compute?

- Consider $B = \{w\#w \mid w \in \{0,1\}^*\}$.
- The TM starts with the input on the tape.

$\underline{0}\ 1\ 1\ 0\ 0\ 0\ \#\ 0\ 1\ 1\ 0\ 0\ 0\ \sqcup\ \sqcup\ \sqcup\sqcup$

$X\ \underline{1}\ 1\ 0\ 0\ 0\ \#\ 0\ 1\ 1\ 0\ 0\ 0\ \sqcup\ \sqcup\ \sqcup\sqcup$

$\rightarrow\rightarrow\cdots$

$X\ 1\ 1\ 0\ 0\ 0\ \#\ \underline{X}\ 1\ 1\ 0\ 0\ 0\ \sqcup\ \sqcup\ \sqcup\sqcup$

$\leftarrow\leftarrow\cdots$

$\underline{X}\ 1\ 1\ 0\ 0\ 0\ \#\ X\ 1\ 1\ 0\ 0\ 0\ \sqcup\ \sqcup\ \sqcup\sqcup$

$X\ \underline{X}\ 1\ 0\ 0\ 0\ \#\ X\ 1\ 1\ 0\ 0\ 0\ \sqcup\ \sqcup\ \sqcup\sqcup$

$\rightarrow\rightarrow\cdots$

$X\ X\ X\ X\ X\ X\ \#\ X\ X\ X\ X\ X\ X\ \underline{\sqcup}\ \sqcup\ \sqcup\sqcup$ **ACCEPT**

Formal Definition of a Turing Machine

A TM is 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are all finite sets.

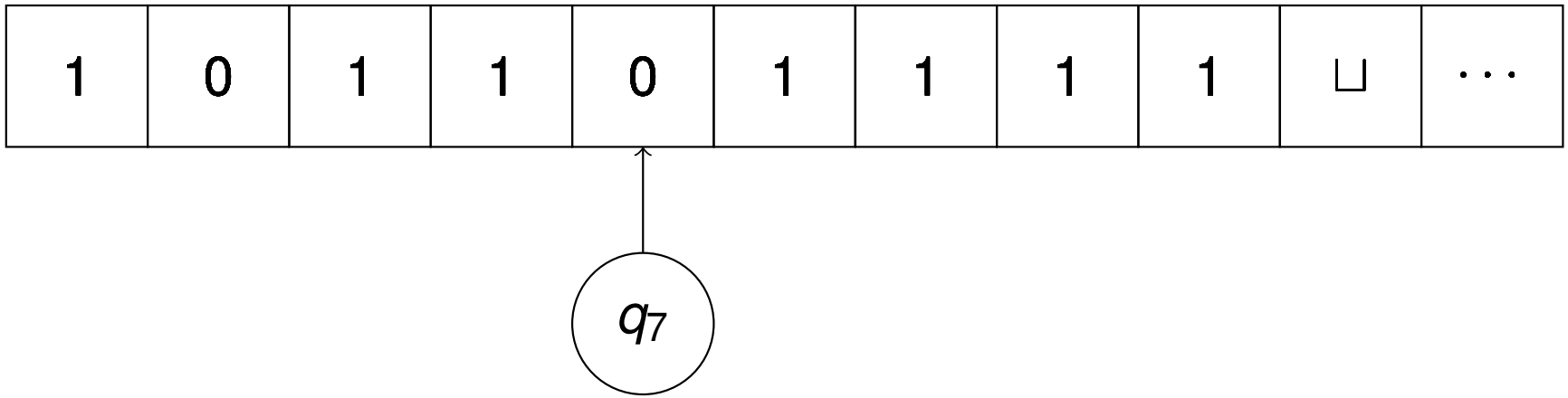
1. Q is the set of states,
2. Σ is the input alphabet (**blank symbol** $\sqcup \notin \Sigma$),
3. Γ is the tape alphabet ($\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$),
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the state transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state,
7. $q_{reject} \in Q$ is the reject state and $q_{reject} \neq q_{accept}$

How does a TM Compute?

- M receives its input $w = w_1 w_2 \cdots w_n$ on the leftmost n squares on the tape. The rest of the tape is blank.
- The head starts on the leftmost square on the tape.
- The first blank symbol on the tape marks the end of the input.
- The computation proceeds according to δ.
- The head of M never moves left of the beginning of the tape (stays there!)
- The computation proceeds until M enters either q_{accept} or q_{reject}, when it halts.
- M may go on forever, never halting!

Configuration of a TM

- As a TM proceeds with its computation, the state changes, the tape changes, the head moves.
- We capture each step of a TM computation, by the notion of a configuration.



- The machine is in state q_7, $u = 1011$ is to the left of the head, $v = 01111$ is under and to the right of the head. Tape has $uv = 1011101111$ on it.
- We represent the configuration by $1011q_701111$.

CONFIGURATIONS

- Configuration C_1 yields ($\Rightarrow$)configuration C_2 if TM can legally go from C_1 to C_2.
- $ua\ q_i\ bv \Rightarrow u\ q_j\ acv$ if $\delta(q_i, b) = (q_j, c, L)$
- $ua\ q_i\ bv \Rightarrow uac\ q_j\ v$ if $\delta(q_i, b) = (q_j, c, R)$
- If the head is at the left end, $q_i bv \Rightarrow q_j cv$ if the transition is left-moving.
- If the head is at the left end, $q_i bv \Rightarrow c q_j v$ if the transition is right-moving.
- Think of a configuration as the contents of memory and a transition as an instruction.

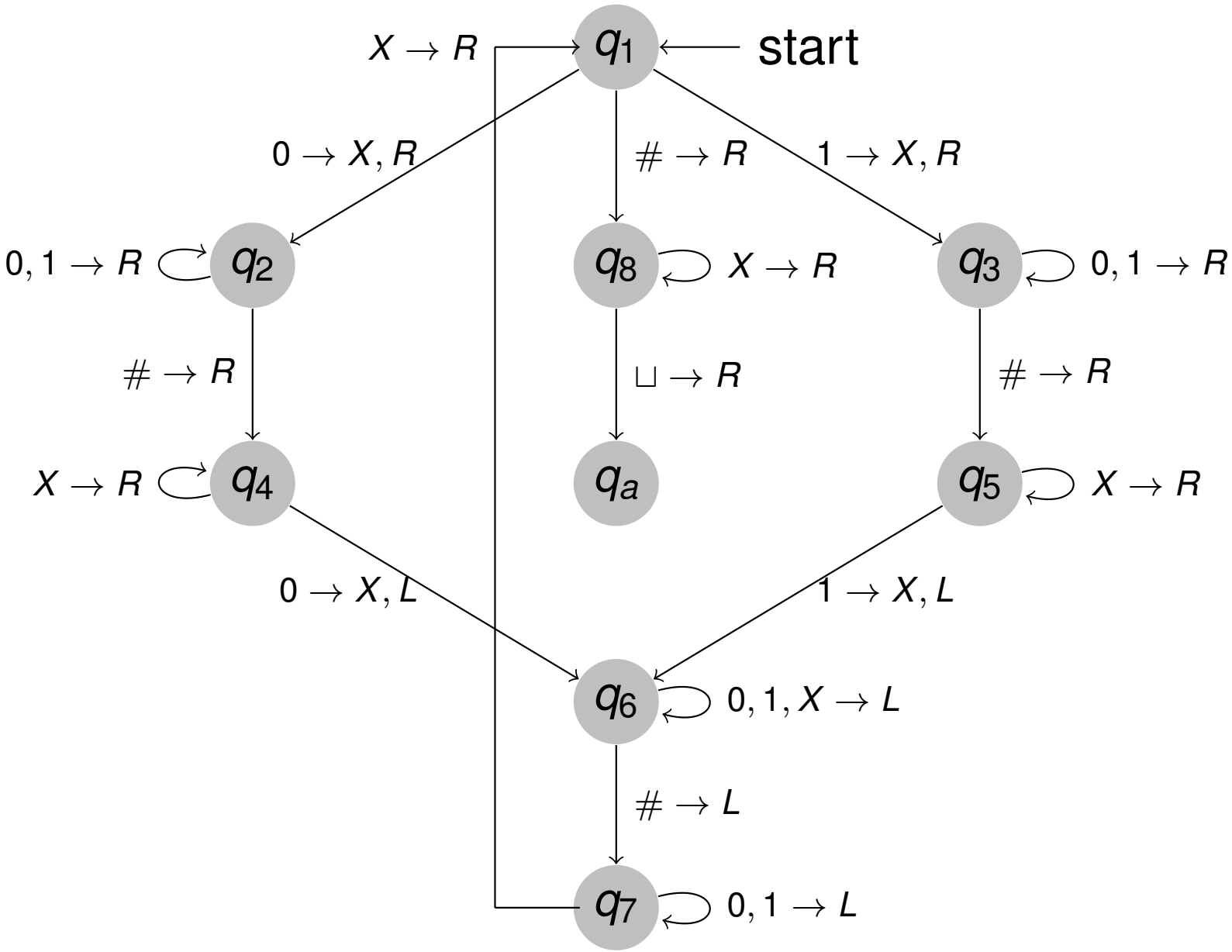
CONFIGURATIONS

- The start configuration is $q_0 w$.
- $uq_{accept}v$ is an accepting configuration,
- $uq_{reject}v$ is a rejecting configuration.
- Accepting and rejecting configurations are halting configurations.

Accepting Computation

- A TM M accepts input w if a sequence of configurations $C_1, C_2, \cdots, C_k$ exists, where
  1. C_1 is the start configuration of M in input w,
  2. $C_i \Rightarrow C_{i+1}$, and
  3. C_k is an accepting configuration.
- $L(M)$ is the set of strings w recognized by M.
- A language L is Turing-recognizable if some TM recognizes it (also called Recursively enumerable)
- A TM is called a decider if it halts on all inputs.
- A language is Turing-decidable if some TM decides it (also called Recursive)
- Every decidable language is Turing recognizable!

EXAMPLE TM-1


$X \to R$
q_1
start
$0 \to X, R$
$\# \to R$
$1 \to X, R$
$0, 1 \to R$
q_2
q_8
$X \to R$
q_3
$0, 1 \to R$
$\# \to R$
$\sqcup \to R$
$\# \to R$
$X \to R$
q_4
q_a
q_5
$X \to R$
$0 \to X, L$
$1 \to X, L$
q_6
$0, 1, X \to L$
$\# \to L$
q_7
$0, 1 \to L$

EXAMPLE TM-1

- Let us see how this TM operates on input 001101#001101

Formal Languages, Automata and Computation

Turing Machines

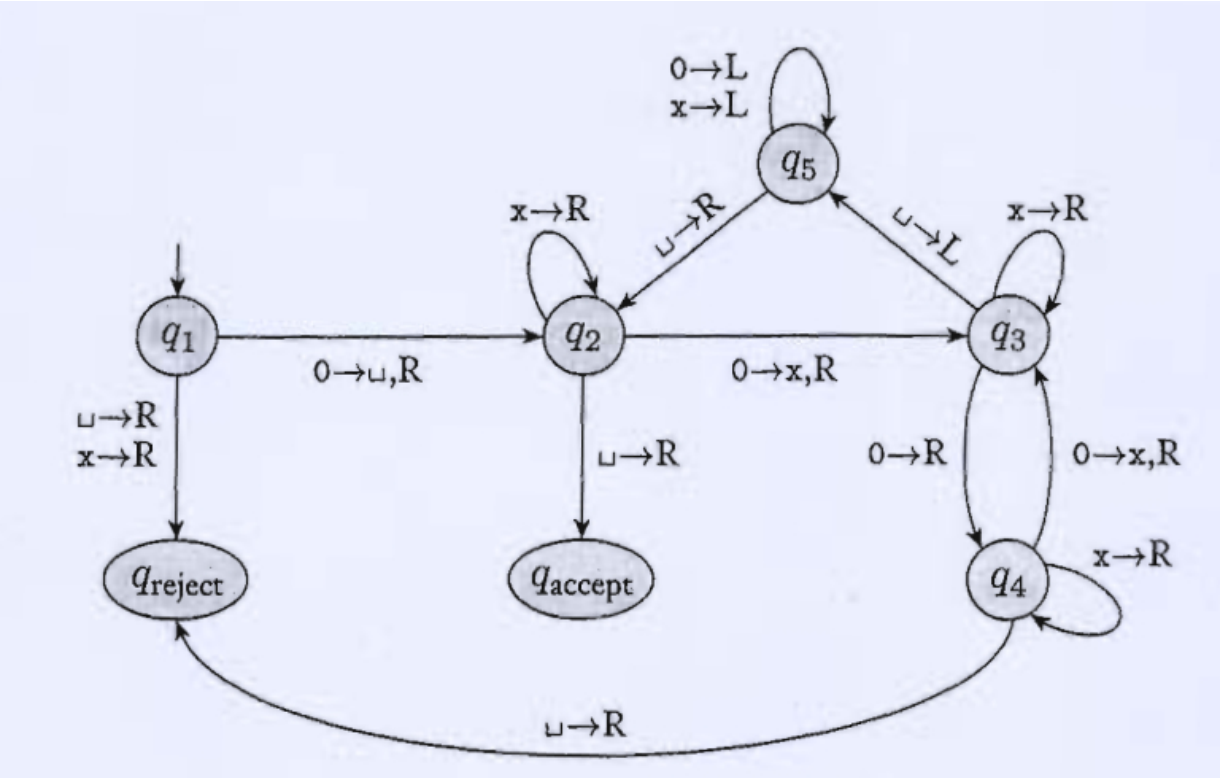
Turing Machines-Synopsis

- The most general model of computation
- Computations of a TM are described by a sequence of configurations.
  - Accepting Configuration
  - Rejecting Configuration
- Turing-recognizable languages
  - TM halts in an accepting configuration if w is in the language.
  - TM may halt in a rejecting configuration or go on indefinitely if w is not in the language.
- Turing-decidable languages
  - TM halts in an accepting configuration if w is in the language.
  - TM halts in a rejecting configuration if w is not in the language.

EXAMPLE TM-2

- A Turing machine that decides $A = \{0^{2^n} \mid n \geq 0\}$
- $M =$ "On Input string w
  1. Sweep left-to-right across the tape, crossing off every other 0.
  2. If in 1) that tape has one 0 left, *accept* (Why?)
  3. If in 1) tape has more than one 0, and the number of 0's is odd, *reject*. (Why?)
  4. Return the head to the left end of the tape.
  5. Go to 1)"
- Basically every sweep cuts the number of 0's by two.
- At the end only 1 should remain and if so the original number of zeroes was a power of 2.'

EXAMPLE TM-2



Configurations for input 0000.

1. $q_1 0000␣$
2. $␣q_2 000␣$
3. $␣xq_3 00␣$
4. $␣x0q_4 0␣$
5. $␣x0xq_3␣$
6. $␣x0q_5 x␣$
7. $␣xq_5 0x␣$
8. $␣q_5 x0x␣$
9. $q_5 ␣ x0x␣$
10. $␣q_2 x0x␣$
11. $␣xq_2 0x␣$
12. $␣xxq_3 x␣$
13. $␣xxxq_3␣$
14. $␣xxq_5 x␣$
15. $␣xq_5 xx␣$
16. $␣q_5 xxx␣$
17. $q_5 ␣ xxx␣$
18. $␣q_2 xxx␣$
19. $␣xq_2 xx␣$
20. $␣xxq_2 x␣$
21. $␣xxxq_2␣$
22. $␣xxx ␣ q_{accept}$

EXAMPLE TM-3

- A TM to add 1 to a binary number (with a 0 in front)
- $M =$ "On input w
  1. Go to the right end of the input string
  2. Move left as long as a 1 is seen, changing it to a 0.
  3. Change the 0 to a 1, and halt."
- For example, to add 1 to $w = 0110011$
  - Change all the ending 1's to 0's $\Rightarrow$ 0110000
  - Change the next 0 to a 1 $\Rightarrow$ 0110100
- Now let's design a TM for this problem.

Variants of TMs

- We defined the basic Turing Machine
  - Single tape (infinite in one direction)
  - Deterministic state transitions
- We could have defined many other variants:
  - Ordinary TMs which need not move after every move.
  - Multiple tapes – each with its own independent head
  - Nondeterministic state transitions
  - Single tape infinite in both directions
  - Multiple tapes but with a single head
  - Multidimensional tape (move up/down/left/right)

Equivalence of Power

- A computational model is robust if the class of languages it accepts does not change under variants.
  - We have seen that DFA's are robust for nondeterminism.
  - But not PDAs!
- The robustness of Turing Machines is by far greater than the robustness of DFAs and PDAs.
- We introduce several variants on Turing machines and show that all these variants have equal computational power.
- When we prove that a TM exists with some properties, we do not deal with questions like
  - How large is the TM? or
  - How complex is it to "program" that TM?
- At this point we only seek existential proofs.

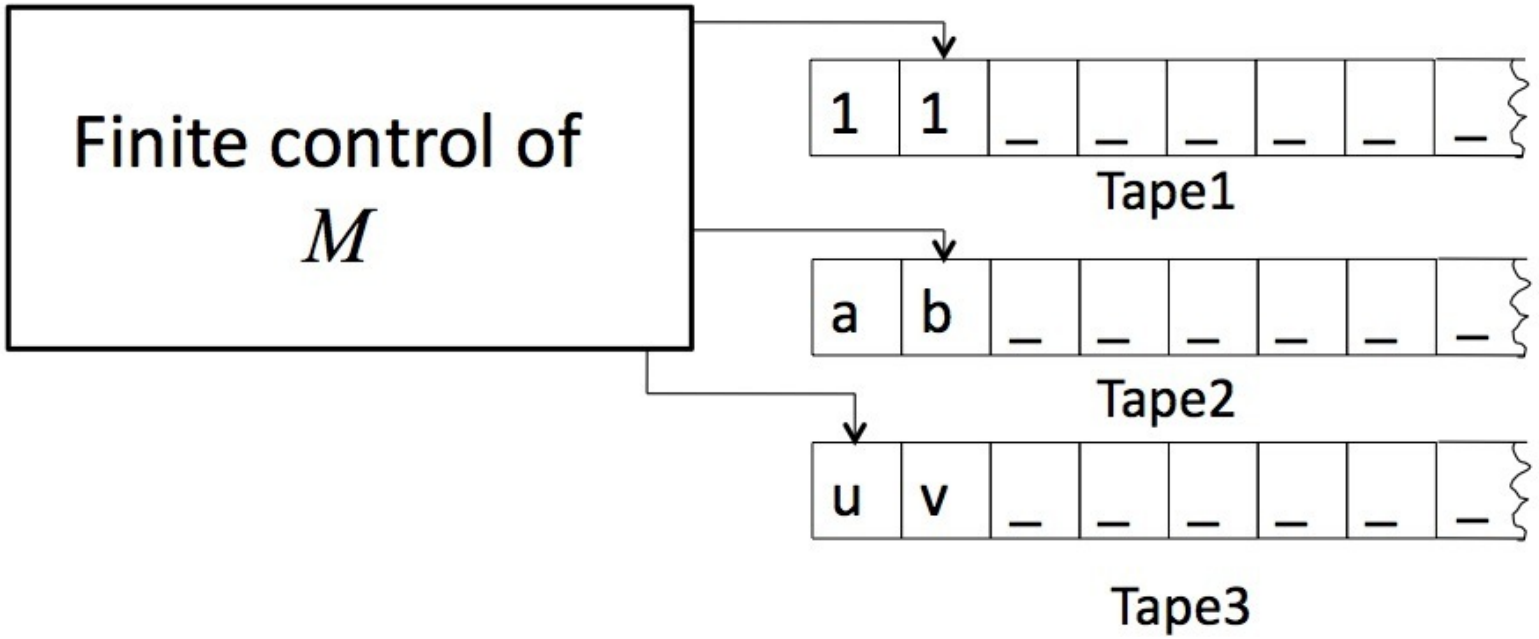
Turing Machines with the Stay Option

- Suppose in addition moving Left or Right, we give the option to the TM to stay (S) on the current cell, that is:

$$\delta : Q \times \Gamma = Q \times \Gamma \times \{L, R, S\}$$

- Such a TM can easily simulate an ordinary TM: just do not use the S option in any move.
- An ordinary TM can easily simulate a TM with the stay option.
  - For each transition with the S option, introduce a new state, and two transitions
    - One transition moves the head right, and transits to the new state.
    - The next transition moves the head back to left, and transits to the previous state.

Multitape Turing Machines



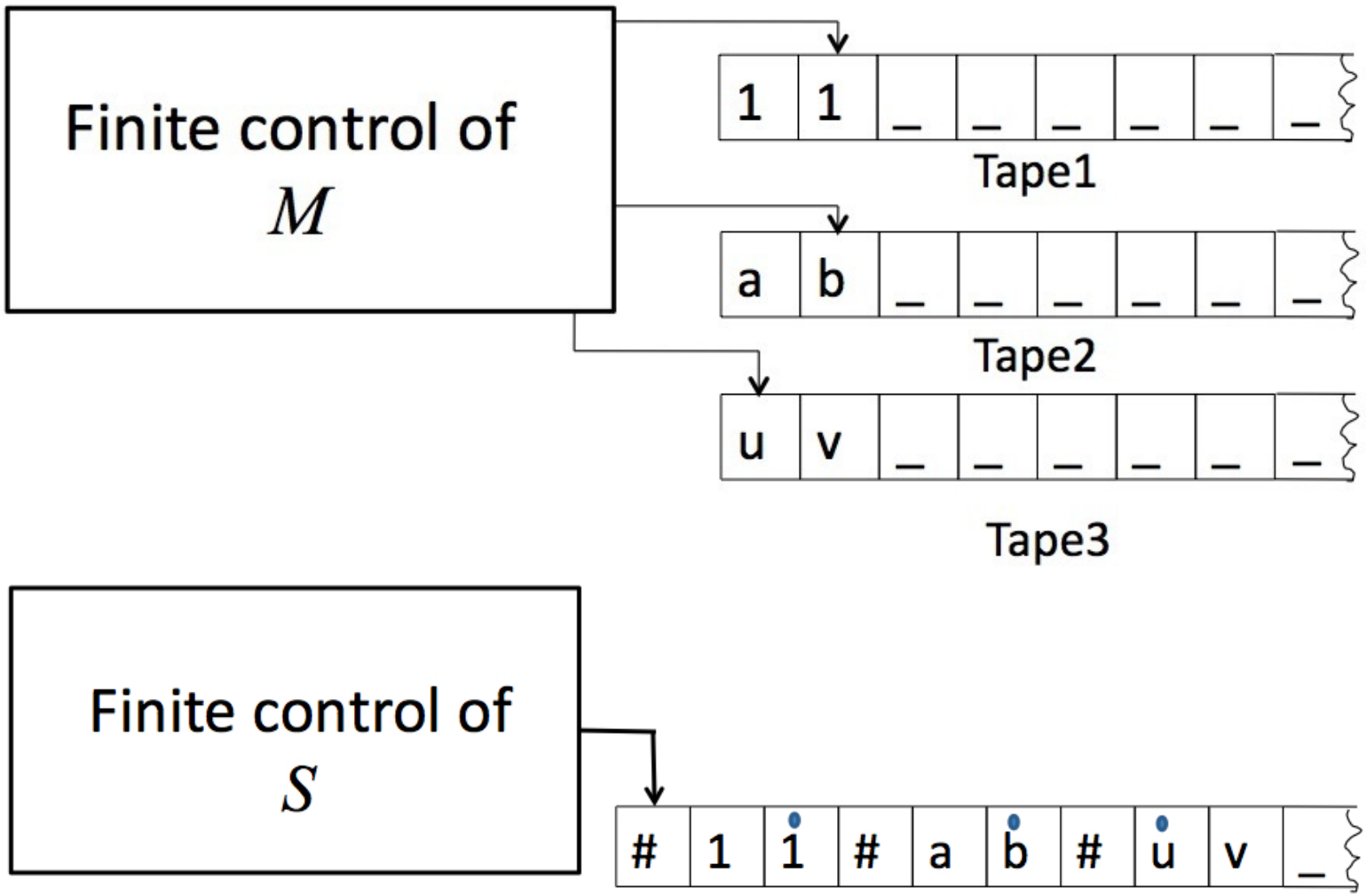
Multitape Turing Machines

- A multitape Turing Machine is like an ordinary TM
  - There are k tapes
  - Each tape has its own independent read/write head.
- The only fundamental difference from the ordinary TM is δ – the state transition function.

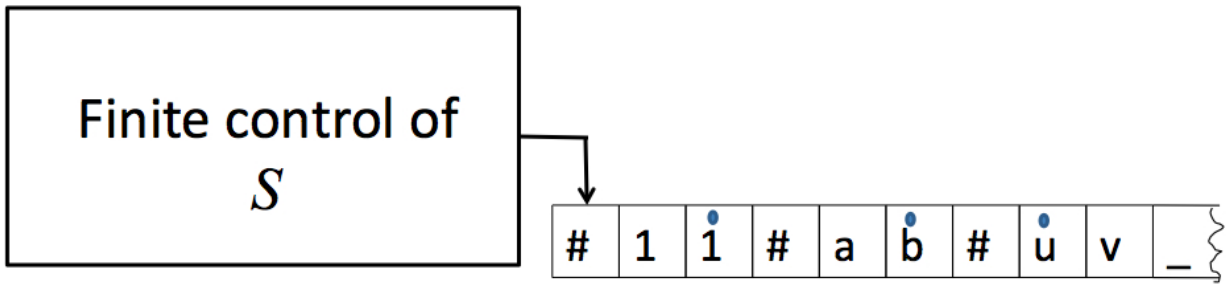
$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

- The δ entry $\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, L, \ldots L)$ reads as :
  - If the TM is in state q_i and
  - the heads are reading symbols a_1 through a_k,
  - Then the machine goes to state q_j, and
  - the heads write symbols b_1 through b_k, and
  - Move in the specified directions.

Simulating a Multitape TM with an Ordinary TM



Simulating a Multitape TM with an Ordinary TM



- We use # as a delimiter to separate out the different tape contents.
- To keep track of the location of heads, we use additional symbols
  - Each symbol in Γ has a "dotted" version.
  - A dotted symbol indicates that the head is on that symbol.
  - Between any two #'s there is only one symbol that is dotted.
- Thus we have 1 real tape with k "virtual' tapes, and
- 1 real read/write head with k "virtual" heads.

Simulating a Multitape TM with an Ordinary TM

- Given input $w = w_1 \cdots w_n$, S puts its tape into the format that represents all k tapes of M

$$\# \dot{w}_1 \; w_2 \cdots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$$

- To simulate a single move of M, S starts at the leftmost $\#$ and scans the tape to the rightmost $\#$.
  - It determines the symbols under the "virtual" heads.
  - This is remembered in the finite state control of S. (How many states are needed?)
- S makes a second pass to update the tapes according to M.
- If one of the virtual heads, moves right to a $\#$, the rest of tape to the right is shifted to "open up" space for that "virtual tape". If it moves left to a $\#$, it just moves right again.

Simulating a Multitape TM with an Ordinary TM

- Thus from now on, whenever needed or convenient we will use multiple tapes in our constructions.
- You can assume that these can always be converted to a single tape standard TM.

Nondeterministic Turing Machines

- We defined the state transition of the ordinary TM as

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

- A nondeterministic TM would proceed computation with multiple next cnfigurations. δ for a nondeterministic TM would be

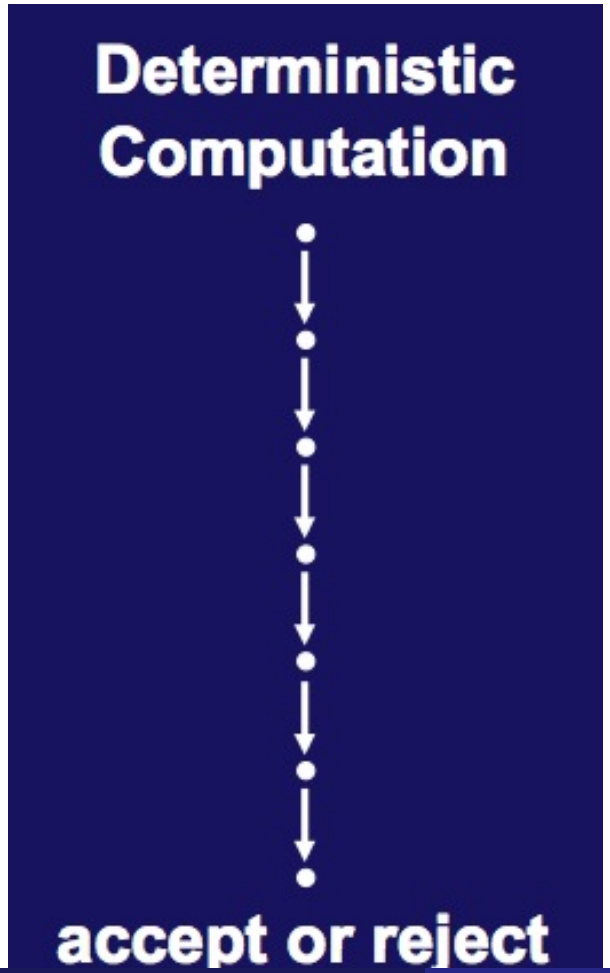
$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

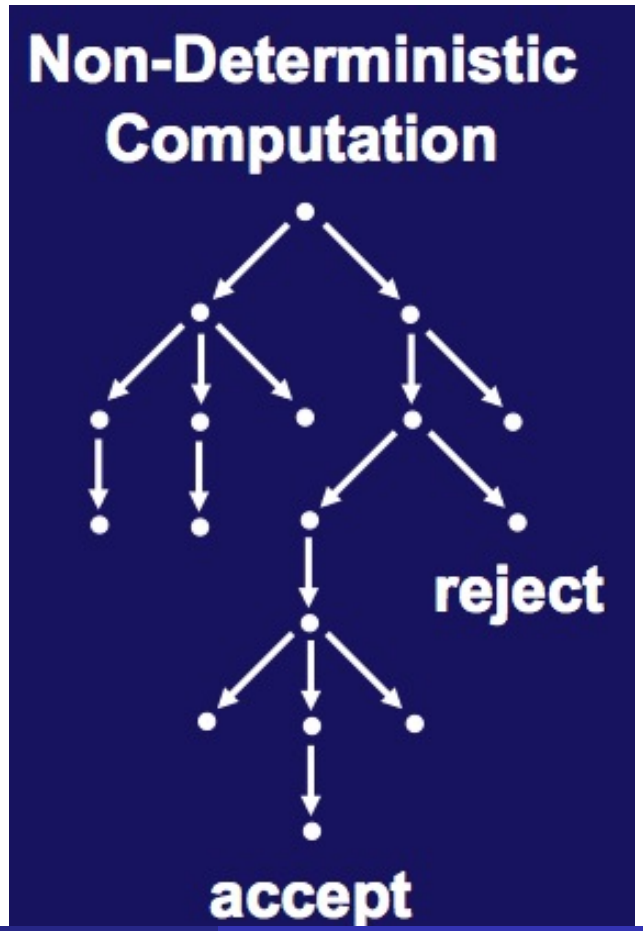
($\mathcal{P}(S)$ is the power set of S.)

- This definition is analogous to NFAs and PDAs.

Nondeterministic Turing Machines

- A computation of a Nondeterministic TM is a tree, where each branch of the tree is looks like a computation of an ordinary TM.





Nondeterministic Turing Machines

- If a single branch reaches the accepting state, the Nondeterministic TM accepts, even if other branches reach the rejecting state.
- What is the power of Nondeterministic TMs?
  - Is there a language that a Nondeterministic TM can accept but no deterministic TM can accept?

Nondeterministic Turing Machines

Theorem

Every nondeterministic Turing machine has an equivalent deterministic Turing Machine.

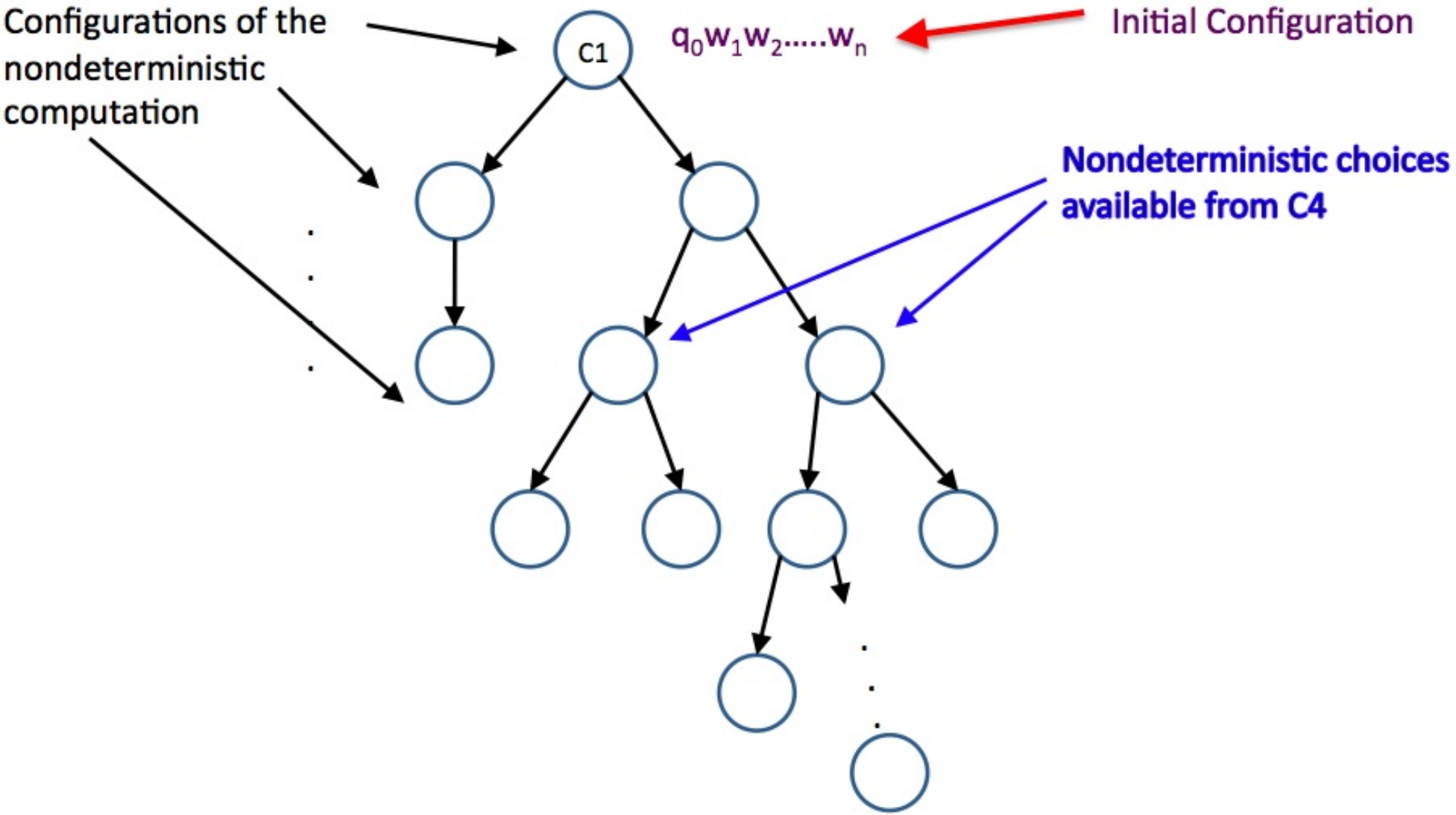
Proof Idea

- Timeshare a deterministic TM to different branches of the nondeterministic computation!
- Try out all branches of the nondeterministic computation until an accepting configuration is reached on one branch.
- Otherwise the TM goes on forever.

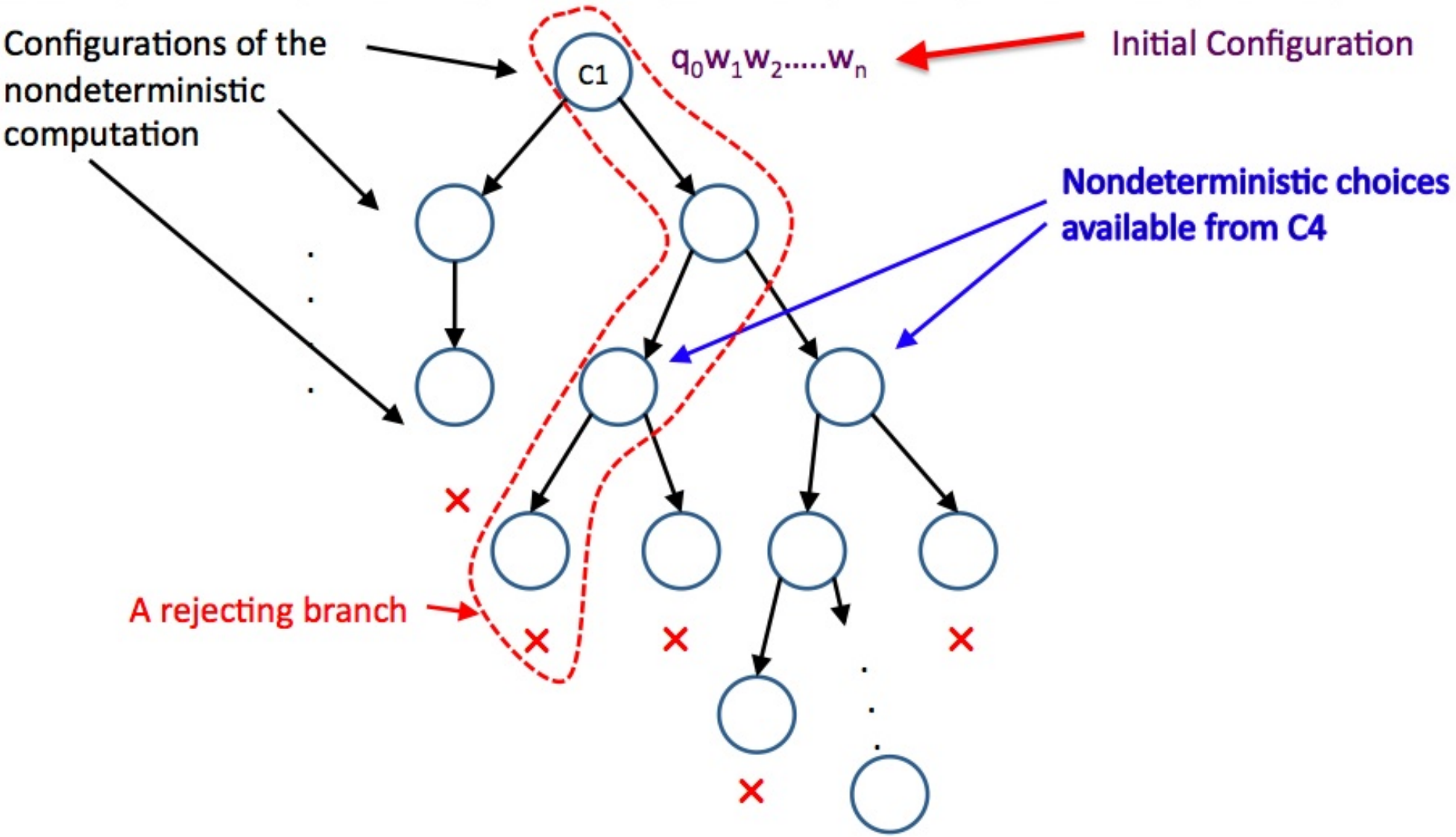
Nondeterministic Turing Machines

- Deterministic TM *D* simulates the Nondeterministic TM *N*.
- Some of branches of the *N*'s computations may be infinite, hence its computation tree has some infinite branches.
- If *D* starts its simulation by following an infinite branch, *D* may loop forever even though *N*'s computation may have a different branch on which it accepts.
- This is a very similar problem to processor scheduling in operating systems.
  - If you give the CPU to a (buggy) process in an infinite loop, other processes "starve".
- In order to avoid this unwanted situation, we want *D* to execute all of *N*'s computations concurrently.

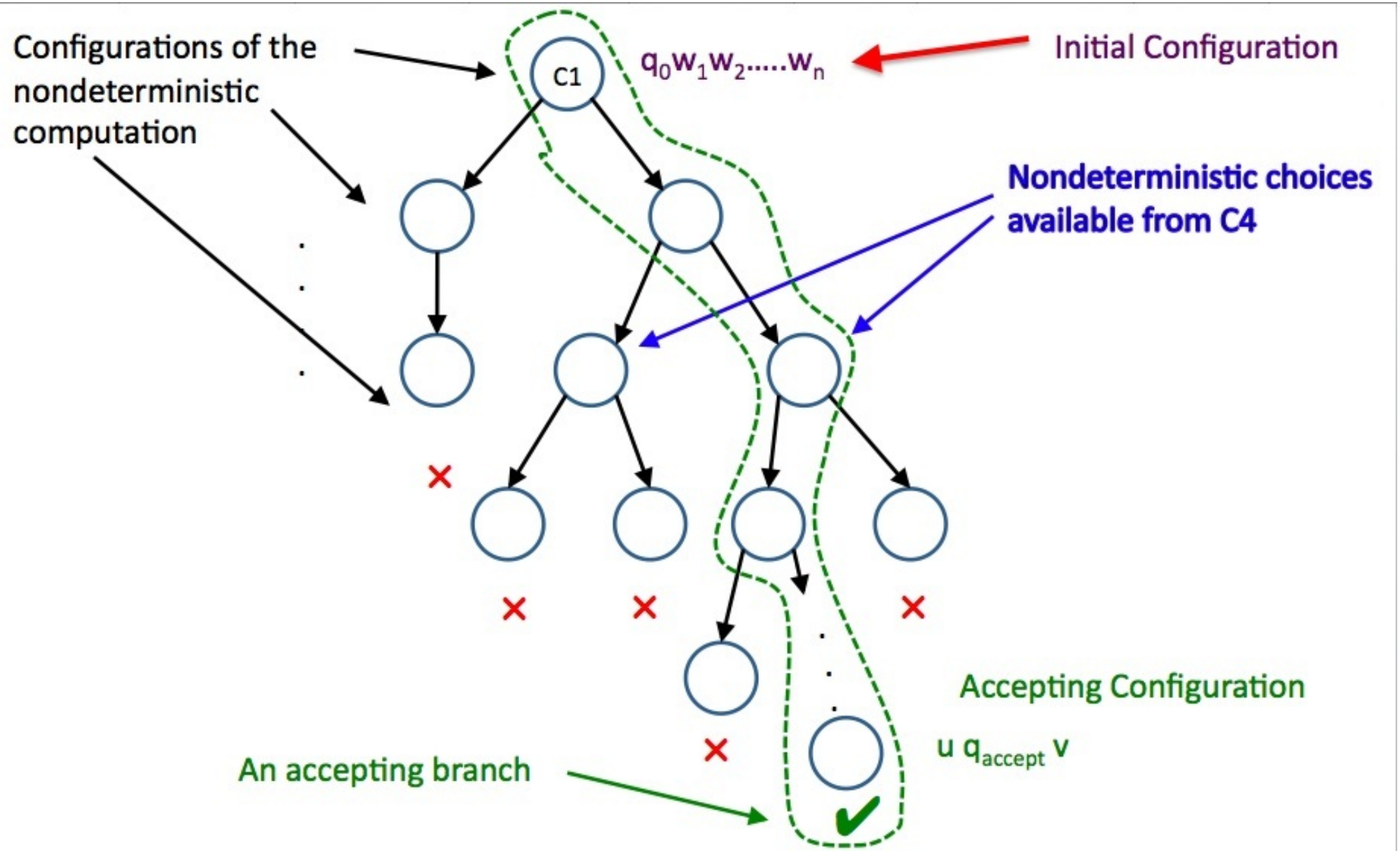
NONDETERMINISTIC COMPUTATION



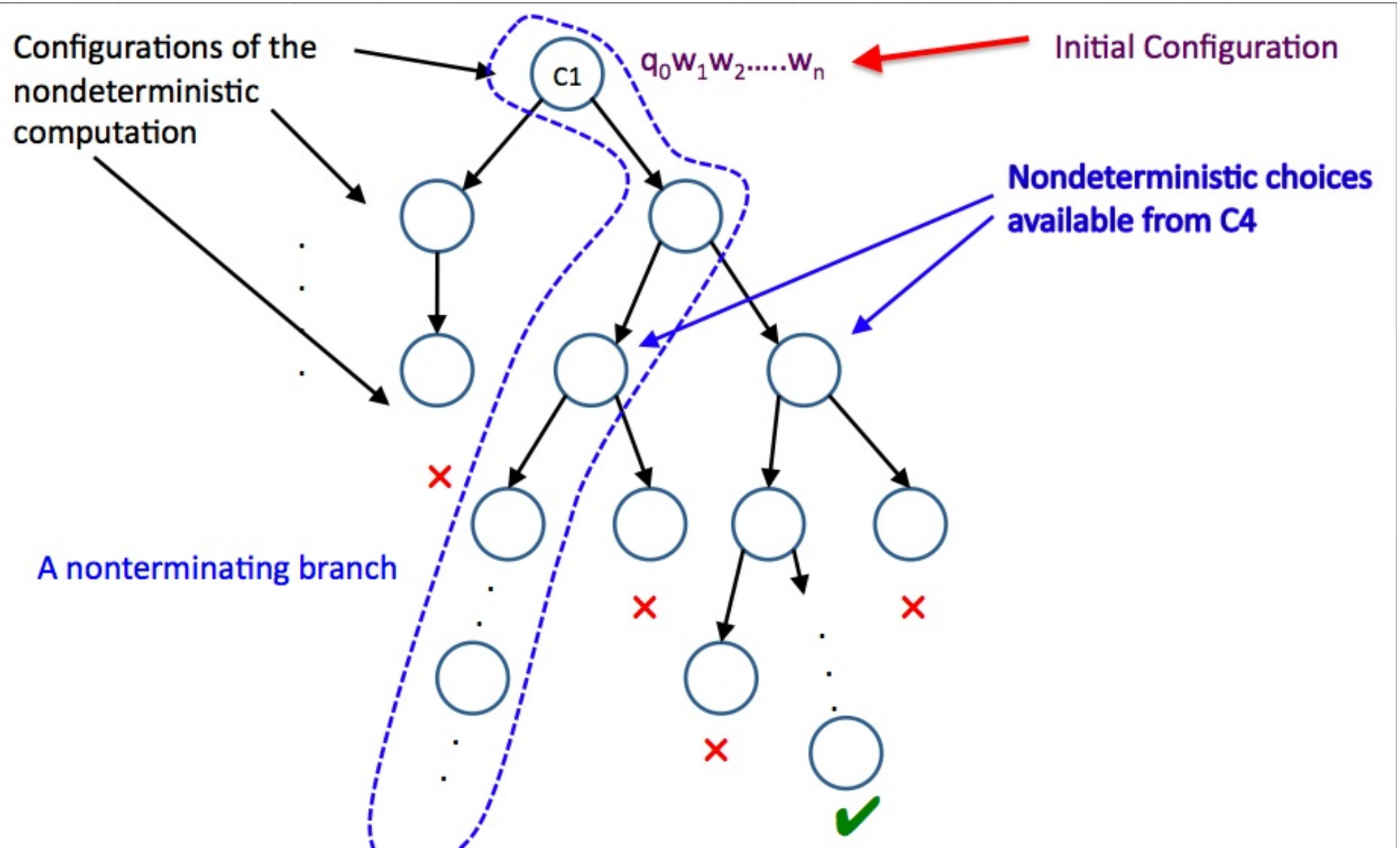
NONDETERMINISTIC COMPUTATION



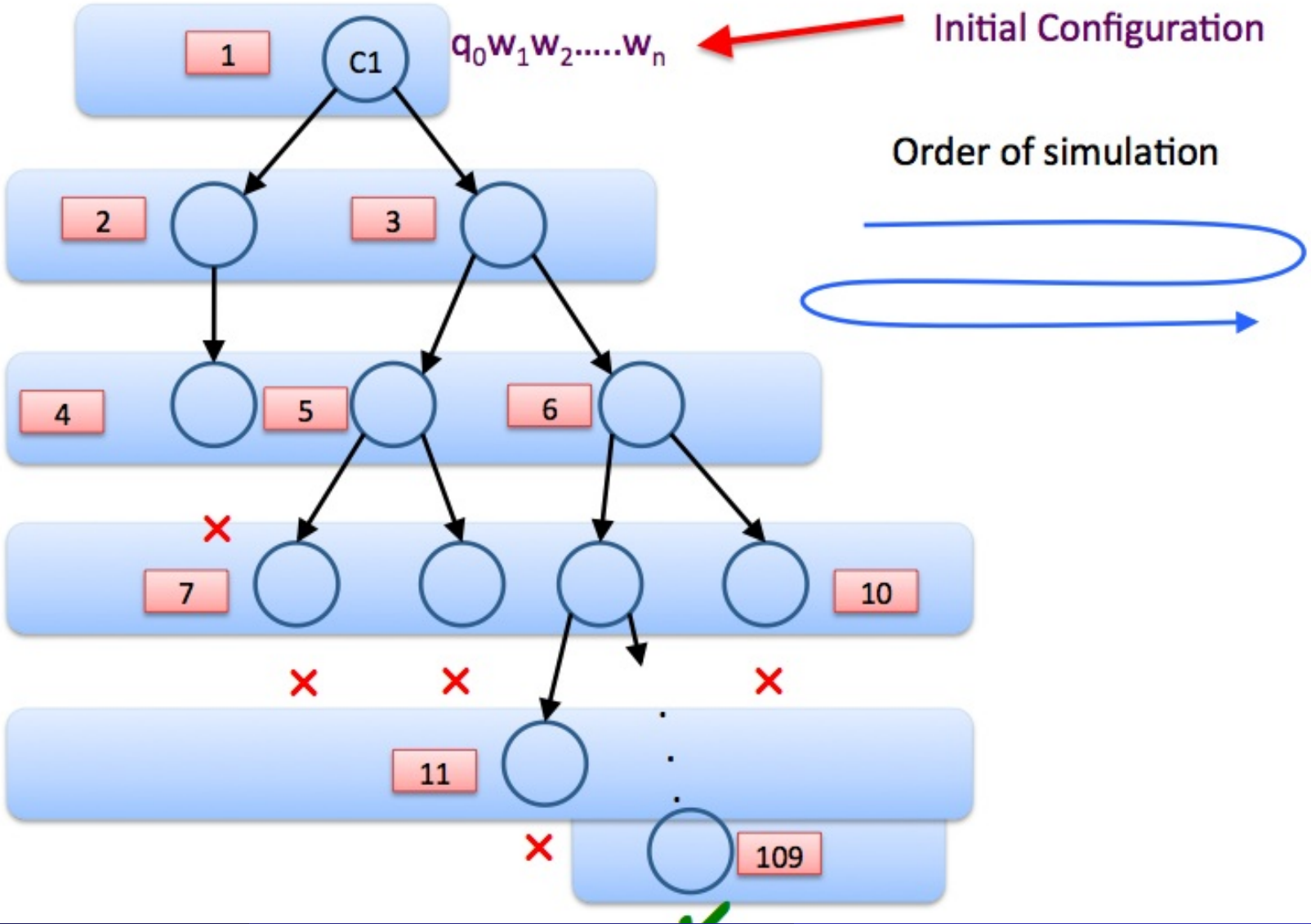
NONDETERMINISTIC COMPUTATION



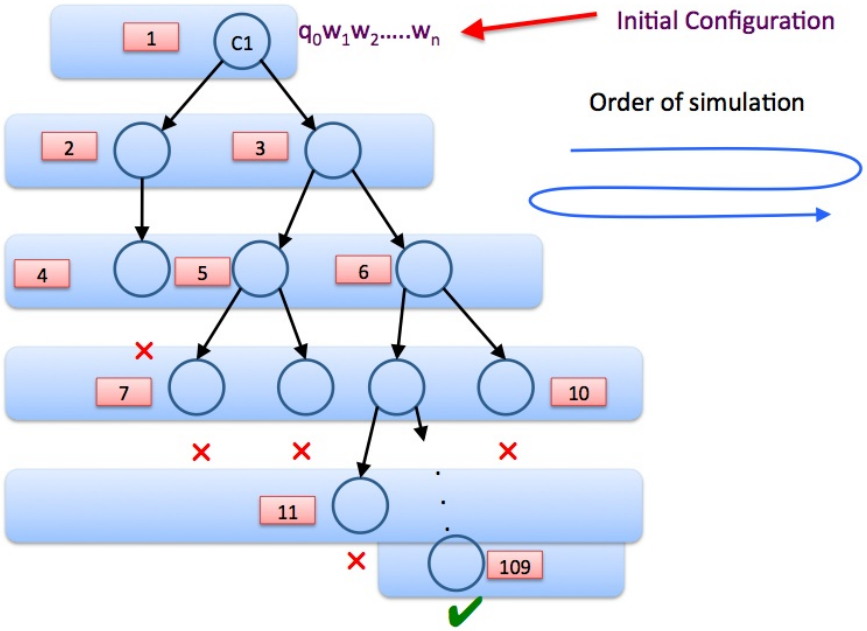
NONDETERMINISTIC COMPUTATION



Simulating Nondeterministic Computation



Simulating Nondeterministic Computation

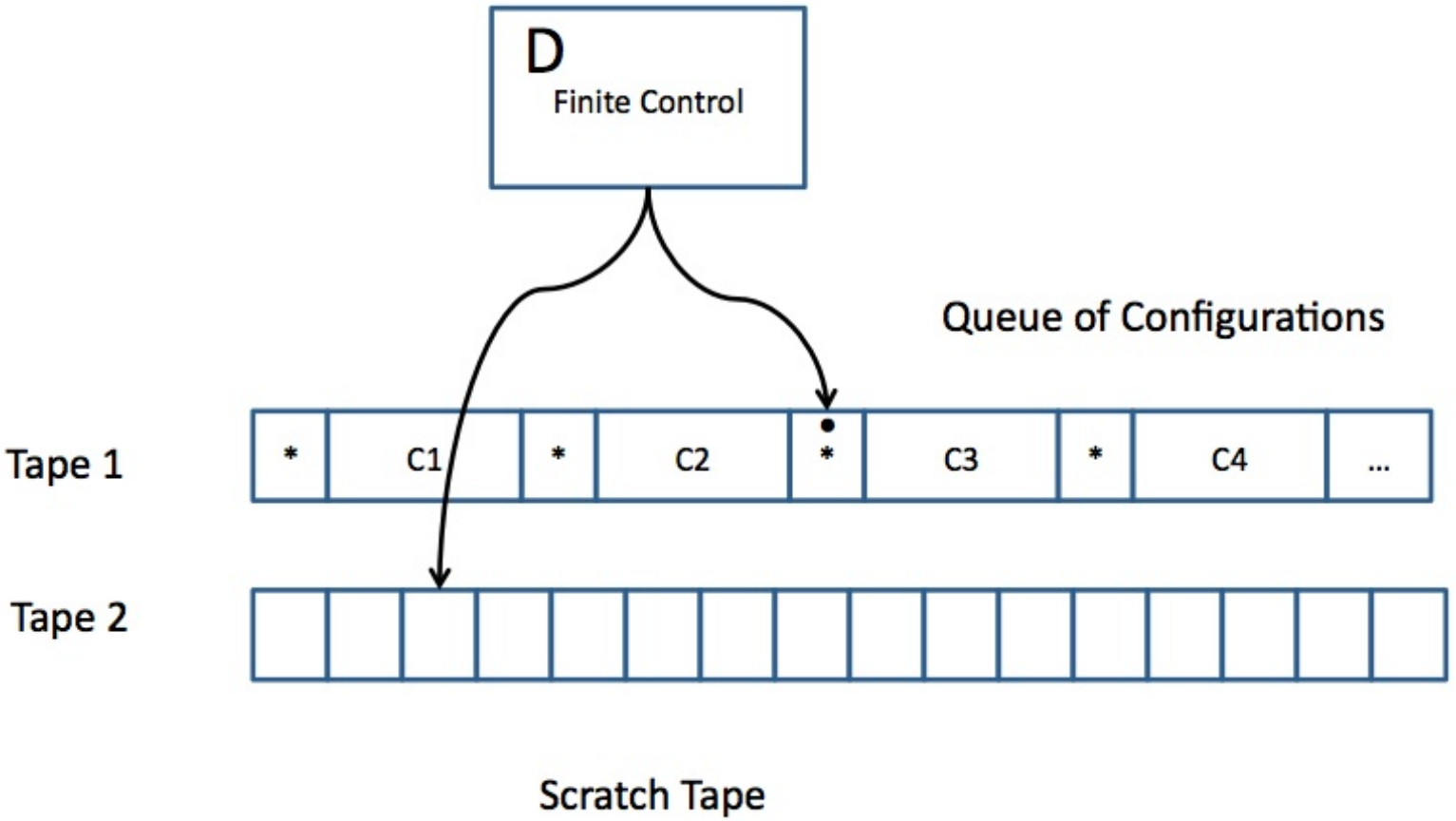


- During simulation, *D* processes the configurations of *N* in a breadth-first fashion.
- Thus *D* needs to maintain a queue of *N*'s configurations (Remember

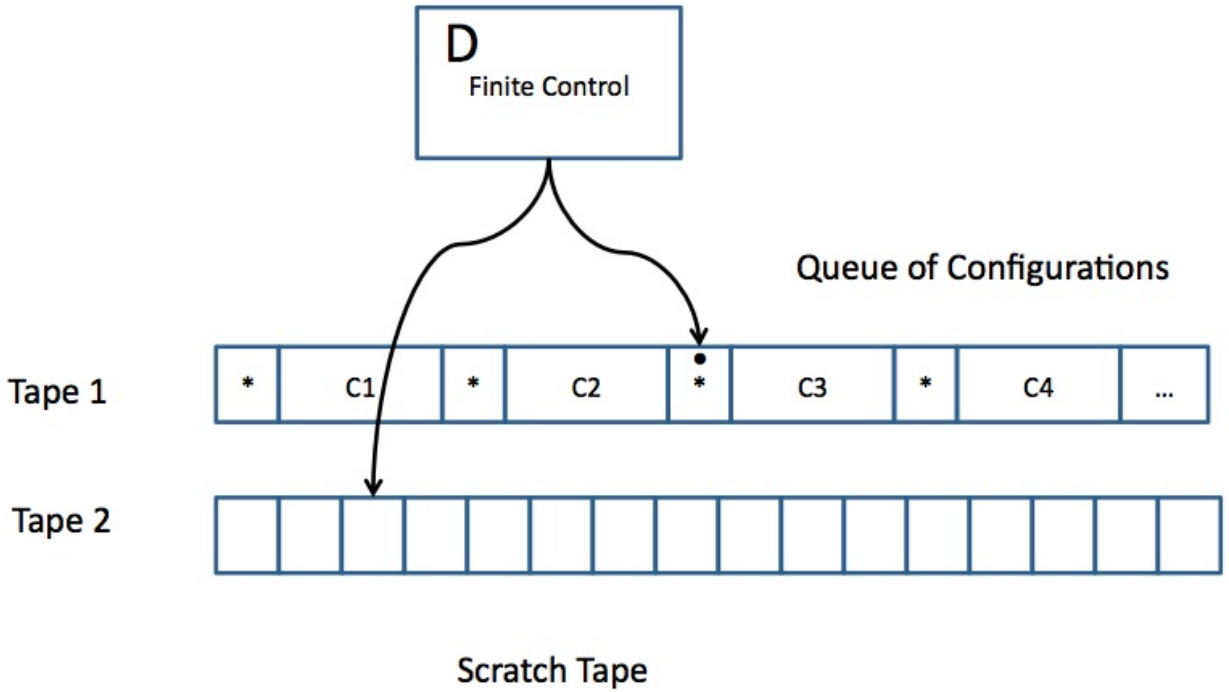
- *D* gets the next configuration from the head of the queue.
- *D* creates copies of this configuration (as many as needed)
- On each copy, *D* simulates one of the nondeterministic moves of *N*.
- *D* places the resulting configurations to the back of the queue.

Structure of the Simulating DTM

- *N* is simulated with 2-tape DTM, *D*
  - Note that this is different from the construction in the book!

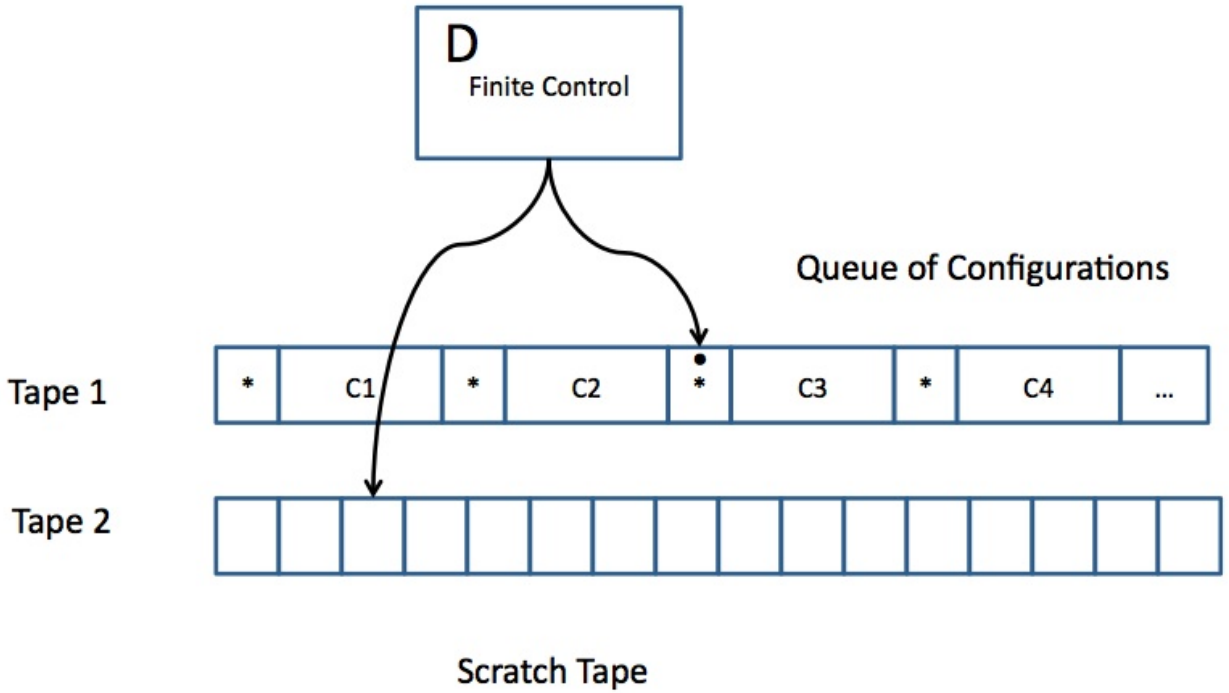


How *D* Simulates *N*



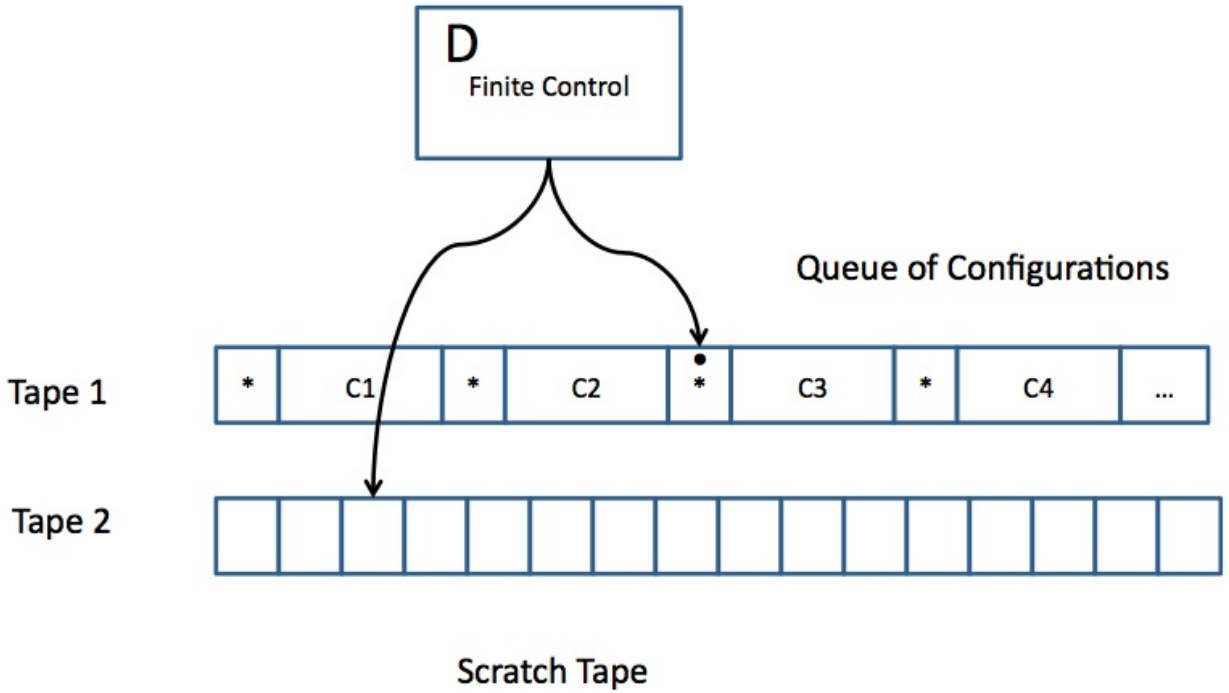
- Built into the finite control of *D* is the knowledge of what choices of moves *N* has for each state and input.

How *D* Simulates *N*



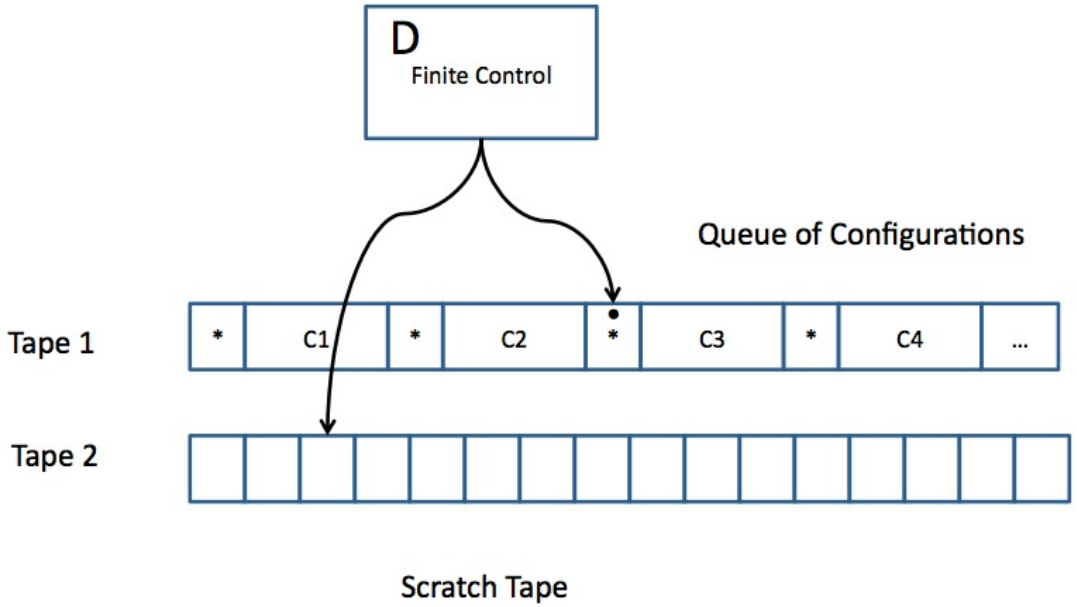
1. *D* examines the state and the input symbol of the current configuration (right after the dotted separator)
2. If the state of the current configuration is the accept state of *N*, then *D* accepts the input and stops simulating *N*.

How *D* Simulates *N*



1. *D* copies *k* copies of the current configuration to the scratch tape.
2. *D* then applies one nondeterministic move of *N* to each copy.

How *D* Simulates *N*



3. *D* then copies the new configurations from the scratch tape, back to the end of tape 1 (so they go to the back of the queue), and then clears the scratch tape.
4. *D* then returns to the marked current configuration, and "erases" the mark, and "marks" the next configuration.
5. *D* returns to step 1), if there is a next configuration. Otherwise rejects.

How D Simulates N

- Let m be the maximum number of choices N has for any of its states.
- Then, after n steps, N can reach at most $1 + m + m^2 + \cdots + m^n$ configurations (which is at most nm^n)
- Thus D has to process at most this many configurations to simulate n steps of N.
- Thus the simulation can take exponentially more time than the nondeterministic TM.
- It is not known whether or not this exponential slowdown is necessary.

Implications

Corollary

A language is Turing-recognizable if and only if some nondeterministic TM recognizes it.

Corollary

A language is decidable if and only of some nondeterministic TM decides it.

Formal Languages, Automata and Computation

Turing Machines

TURING MACHINES-SYNOPSIS

- The most general model of computation
- Computations of a TM are described by a sequence of configurations.
  - Accepting Configuration
  - Rejecting Configuration
- Turing-recognizable languages
  - TM halts in an accepting configuration if w is in the language.
  - TM may halt in a rejecting configuration or go on indefinitely if w is not in the language.
- Turing-decidable languages
  - TM halts in an accepting configuration if w is in the language.
  - TM halts in a rejecting configuration if w is not in the language.

NONDETERMINISTIC TURING MACHINES

- We defined the state transition of the ordinary TM as

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

- A nondeterministic TM would proceed computation with multiple next cnfigurations. δ for a nondeterministic TM would be

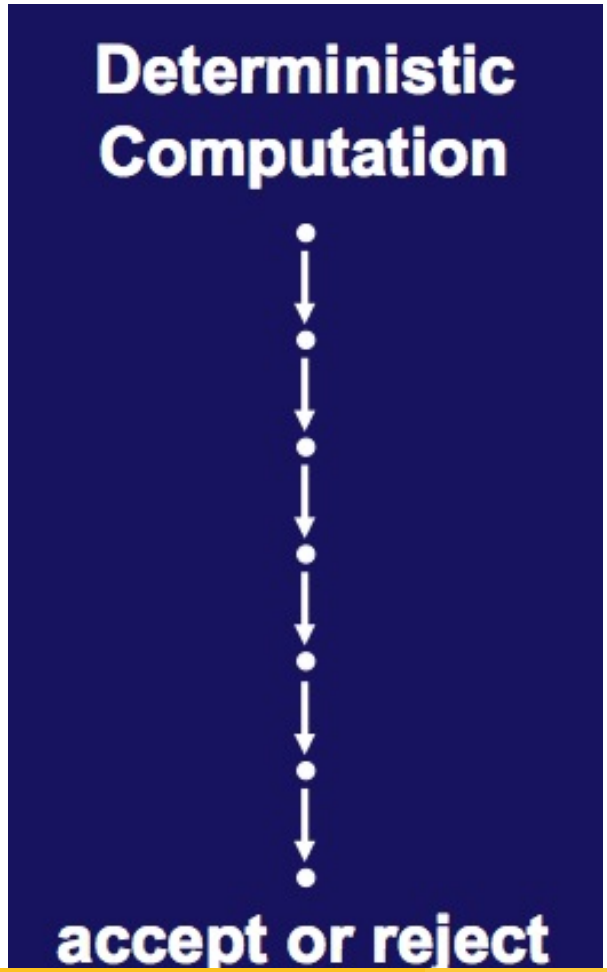
$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

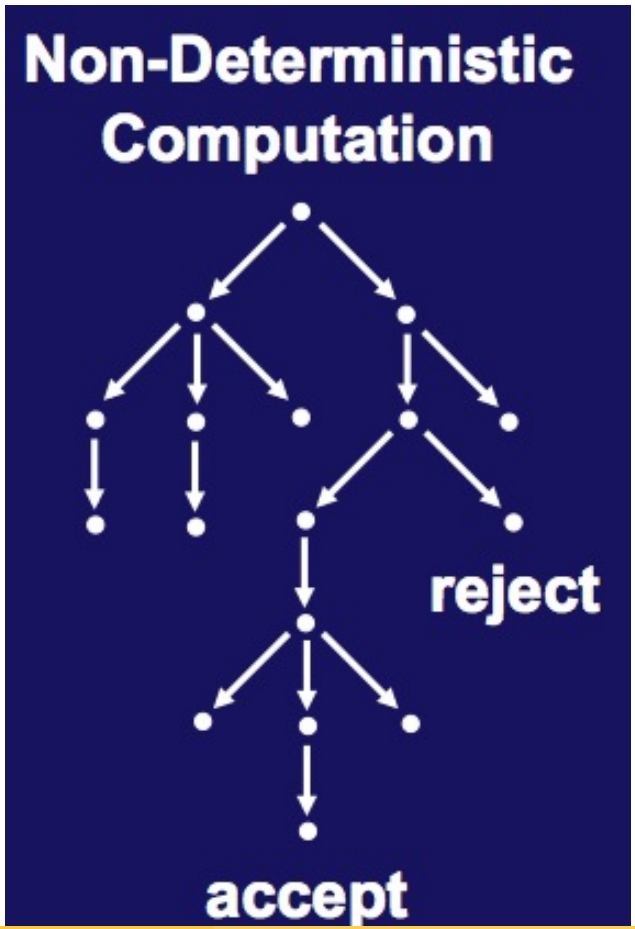
($\mathcal{P}(S)$ is the power set of S.)

- This definition is analogous to NFAs and PDAs.

NONDETERMINISTIC TURING MACHINES

- A computation of a Nondeterministic TM is a tree, where each branch of the tree is looks like a computation of an ordinary TM.





Nondeterministic Turing Machines

- If a single branch reaches the accepting state, the Nondeterministic TM accepts, even if other branches reach the rejecting state.
- What is the power of Nondeterministic TMs?
  - Is there a language that a Nondeterministic TM can accept but no deterministic TM can accept?

Nondeterministic Turing Machines

Theorem

Every nondeterministic Turing machine has an equivalent deterministic Turing Machine.

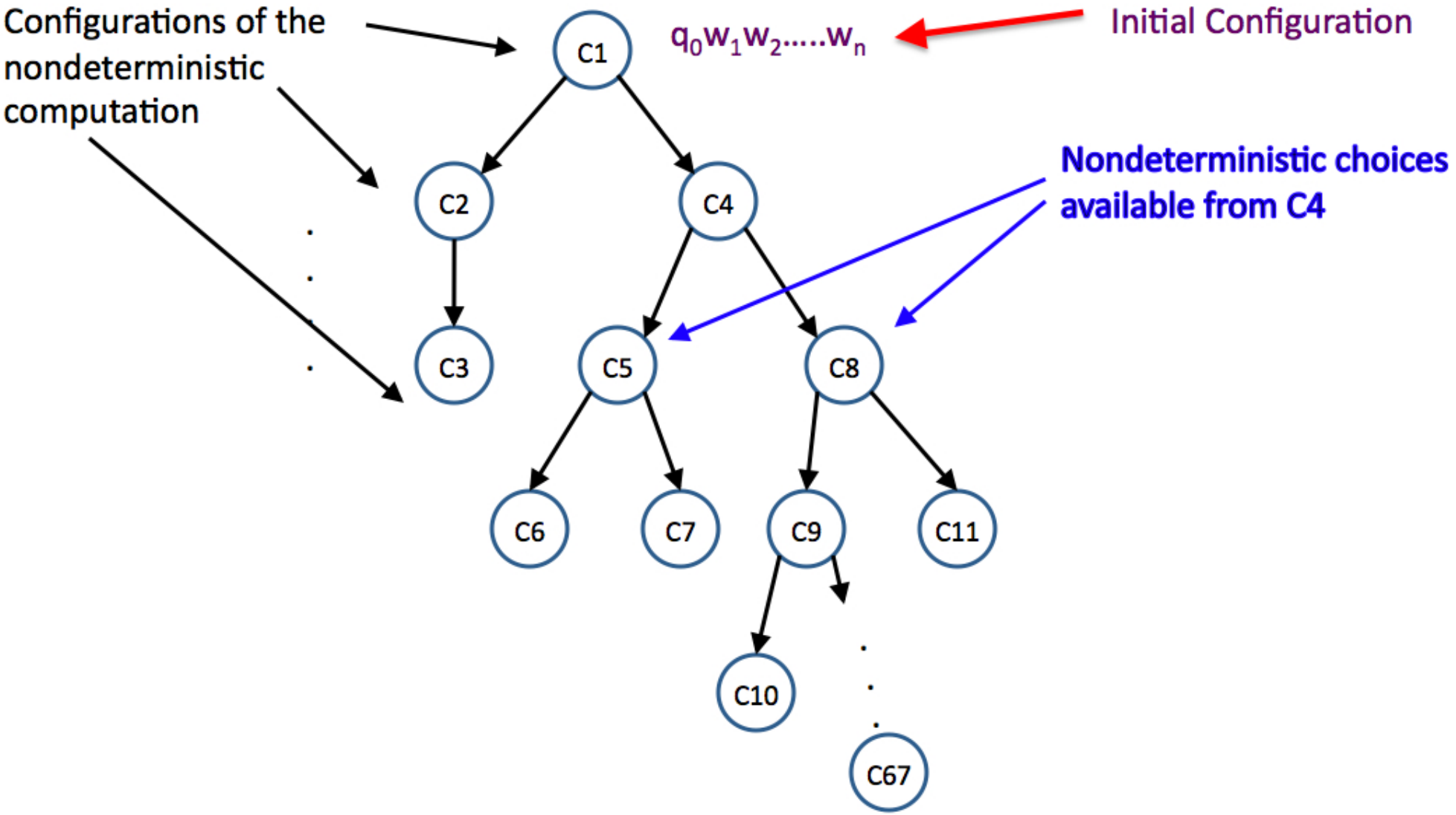
Proof Idea

- Timeshare a deterministic TM to different branches of the nondeterministic computation!
- Try out all branches of the nondeterministic computation until an accepting configuration is reached on one branch.
- Otherwise the TM goes on forever.

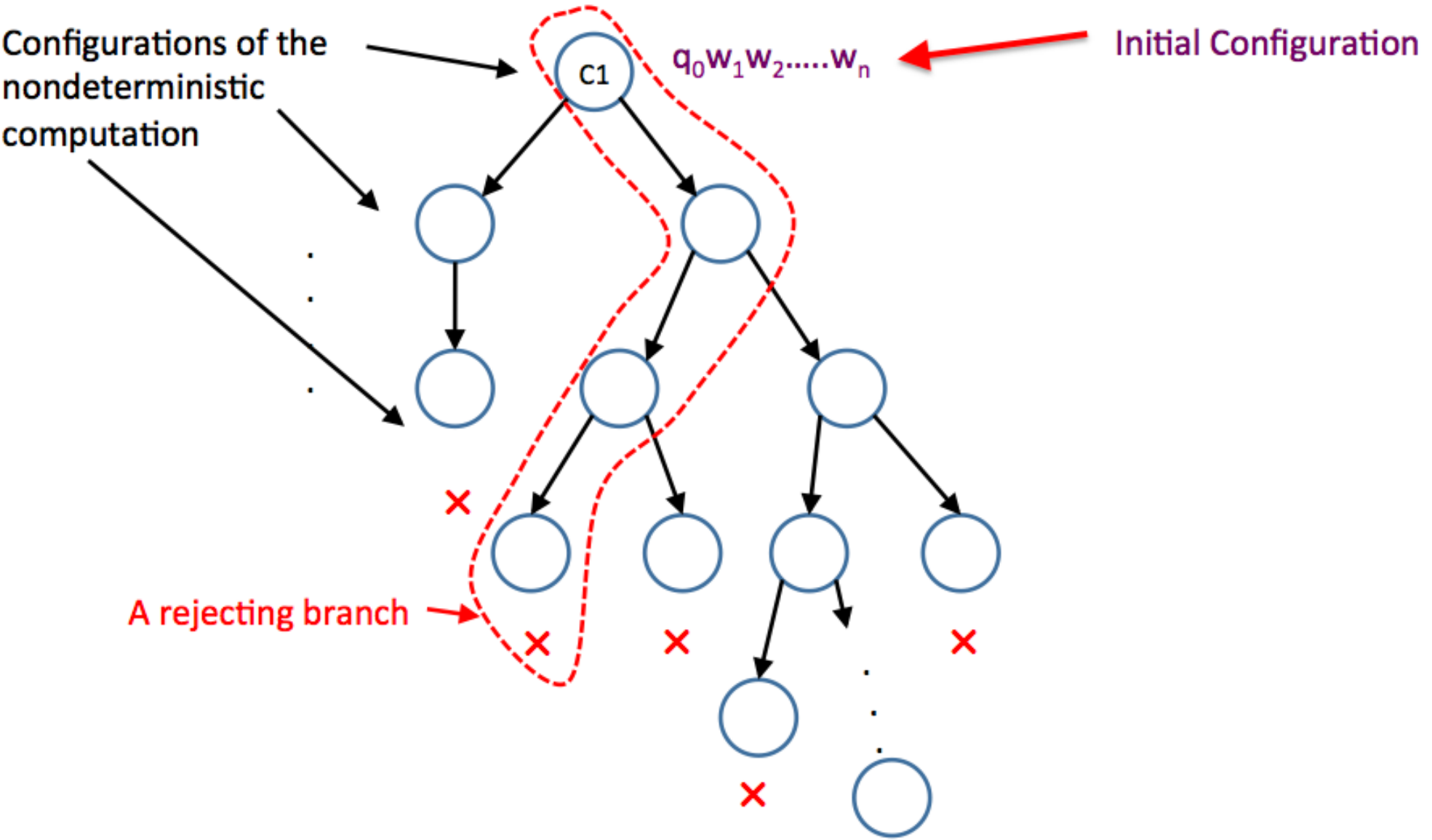
NONDETERMINISTIC TURING MACHINES

- Deterministic TM *D* simulates the Nondeterministic TM *N*.
- Some of branches of the *N*'s computations may be infinite, hence its computation tree has some infinite branches.
- If *D* starts its simulation by following an infinite branch, *D* may loop forever even though *N*'s computation may have a different branch on which it accepts.
- This is a very similar problem to processor scheduling in operating systems.
  - If you give the CPU to a (buggy) process in an infinite loop, other processes "starve".
- In order to avoid this unwanted situation, we want *D* to execute all of *N*'s computations concurrently.

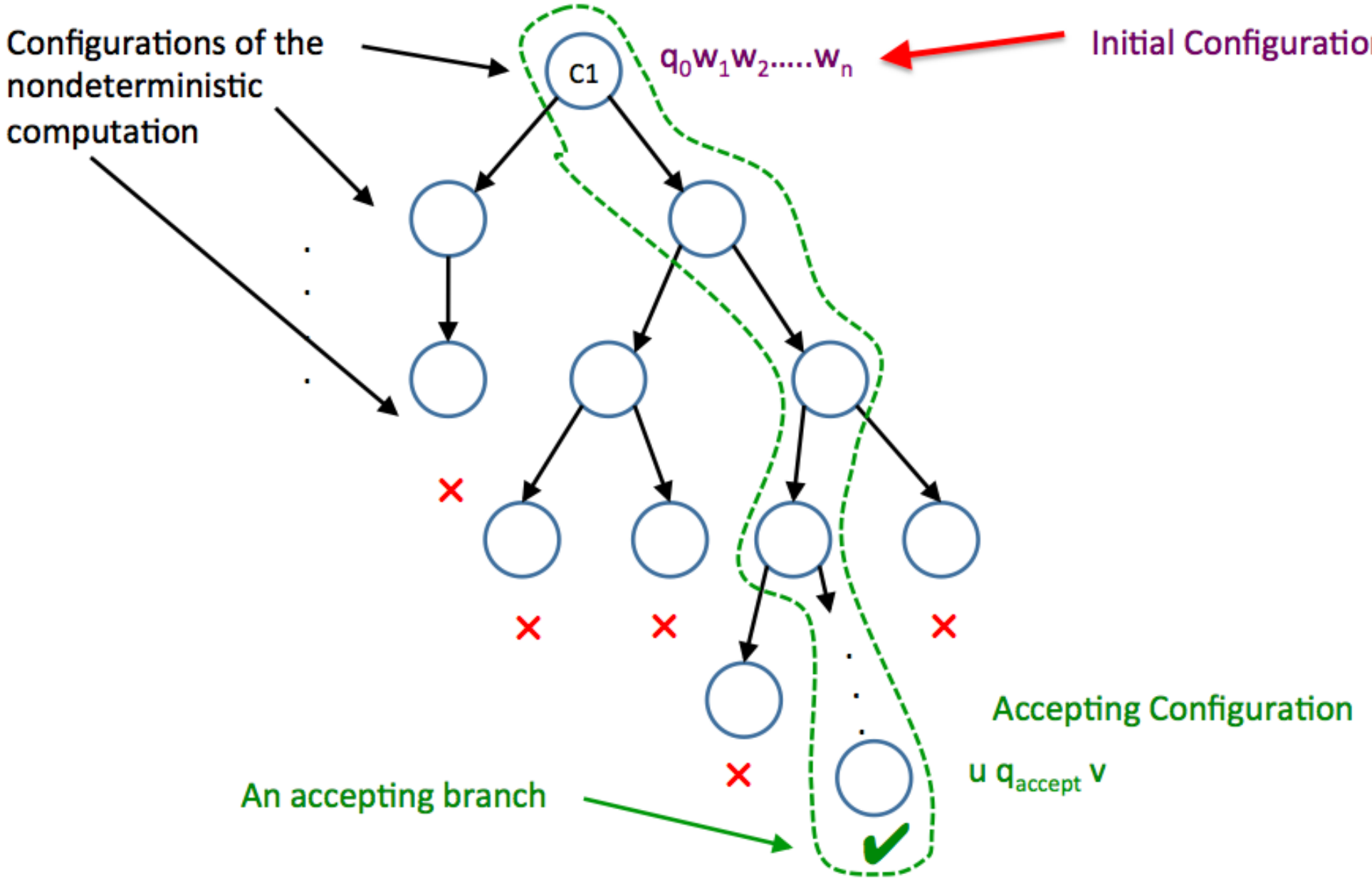
NONDETERMINISTIC COMPUTATION



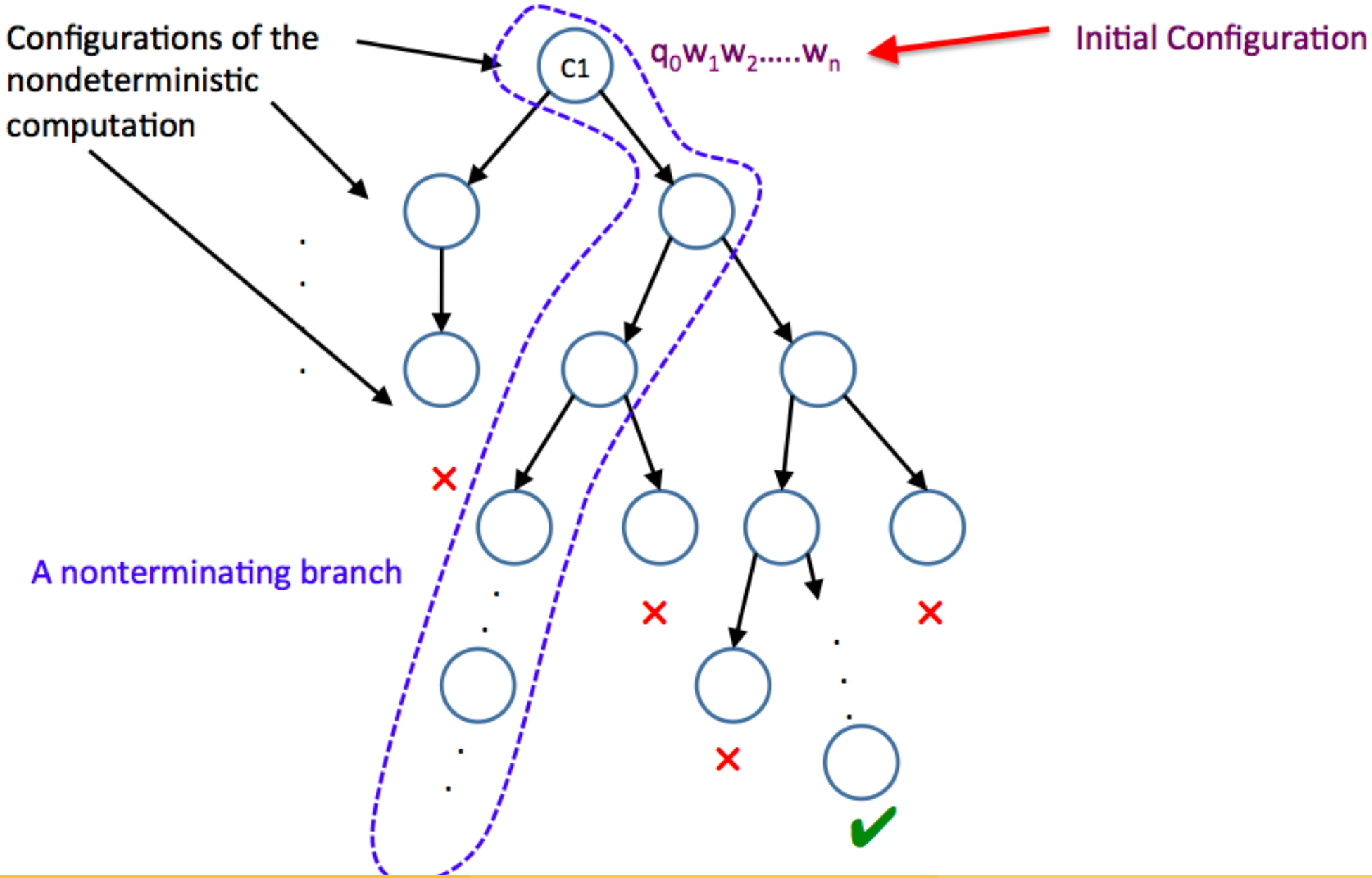
NONDETERMINISTIC COMPUTATION



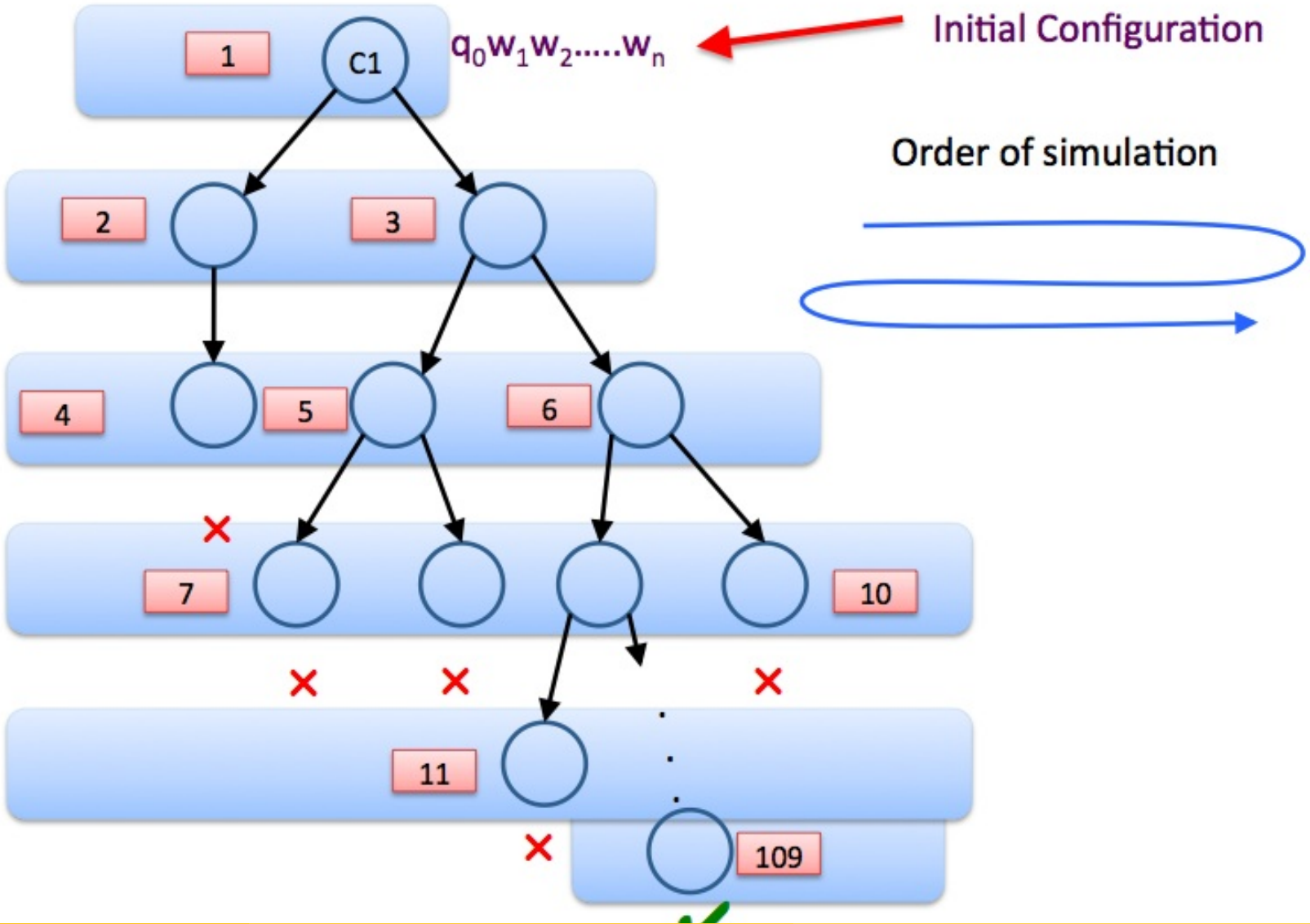
NONDETERMINISTIC COMPUTATION



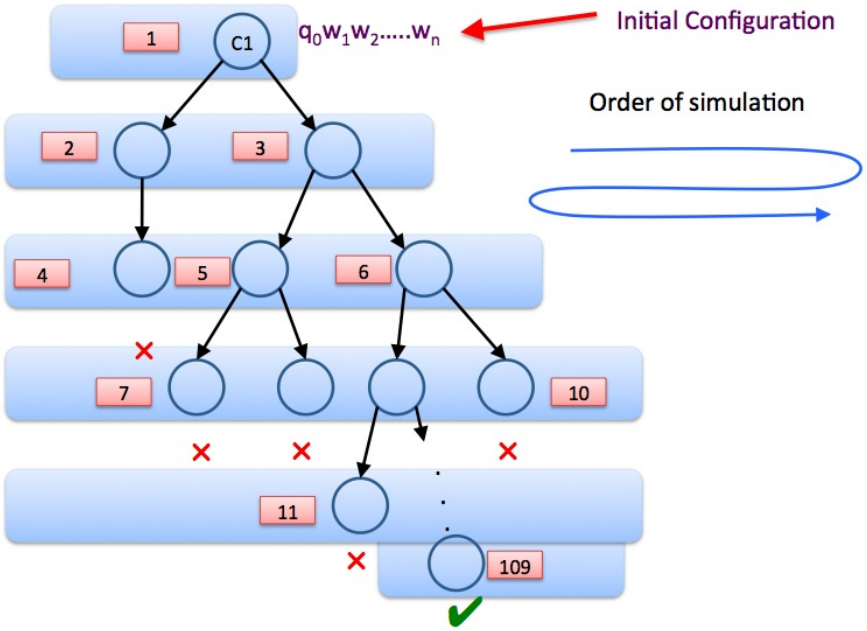
NONDETERMINISTIC COMPUTATION



Simulating Nondeterministic Computation



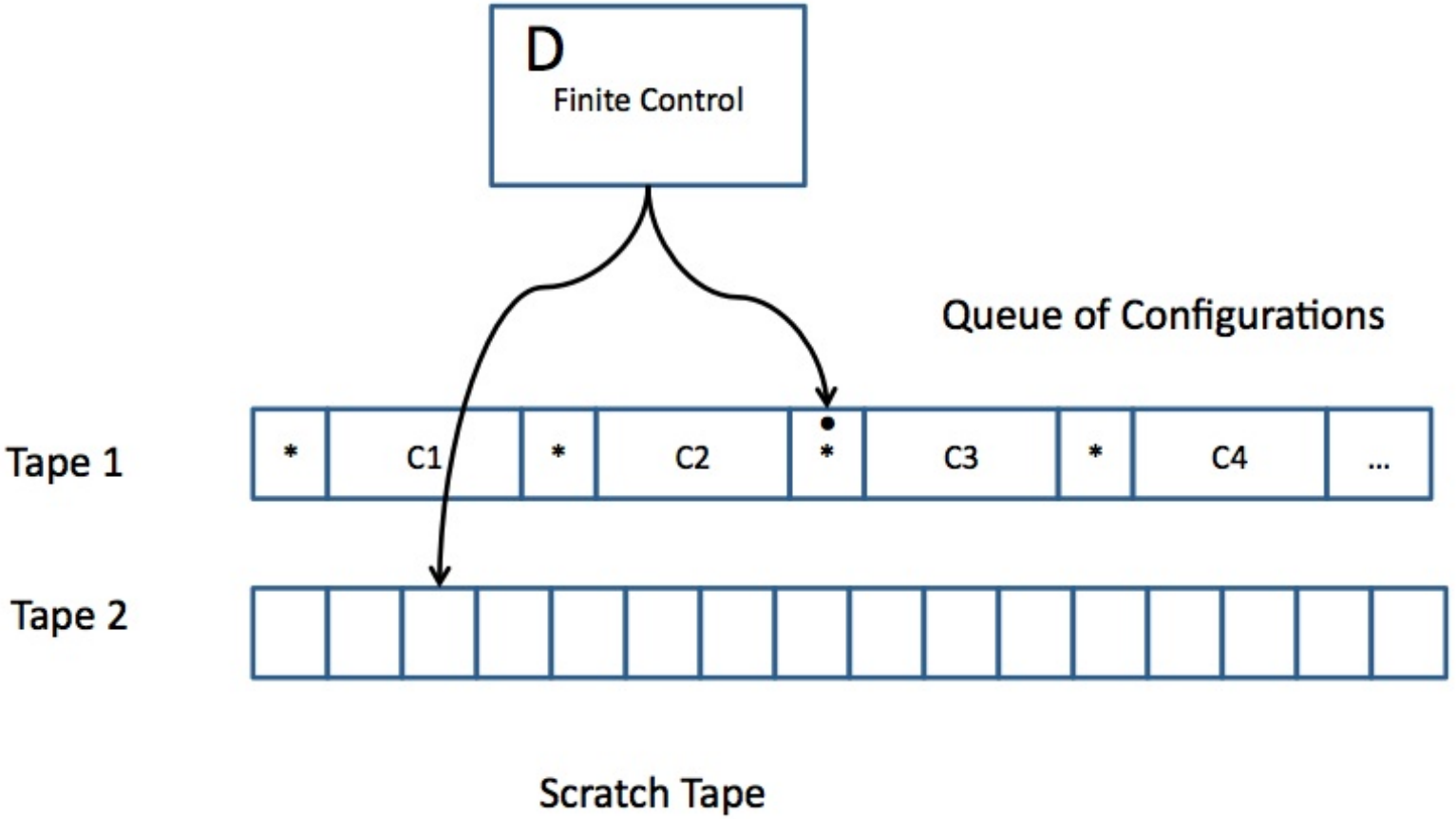
Simulating Nondeterministic Computation



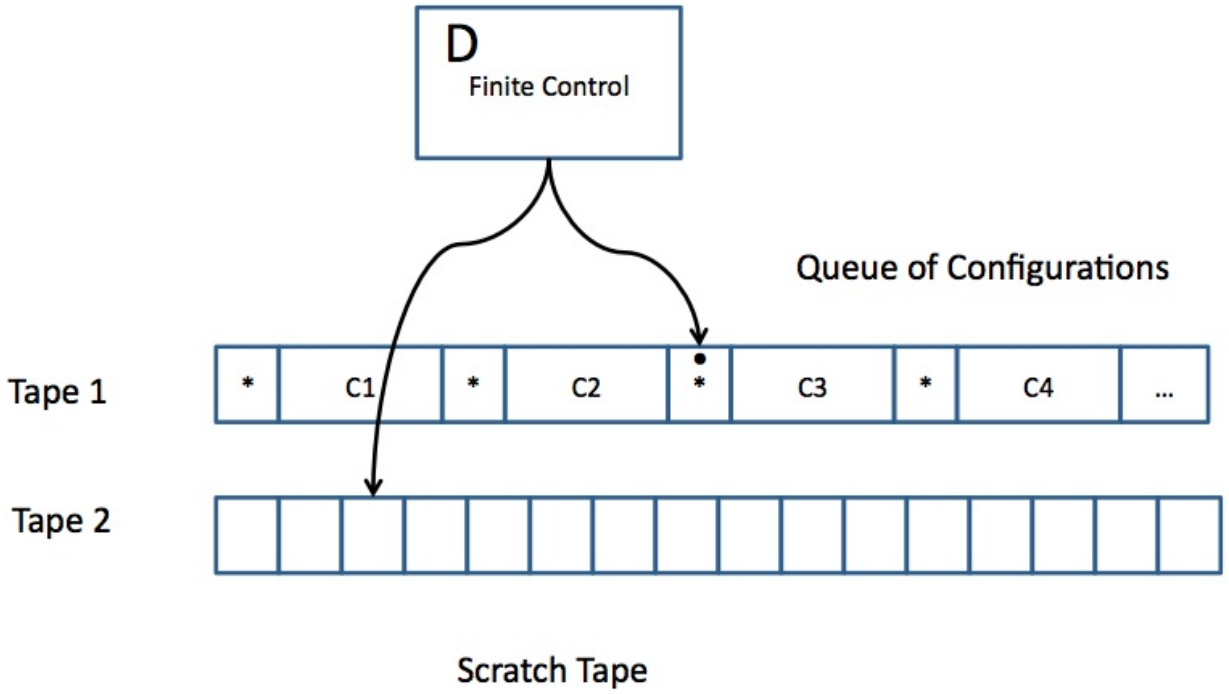
- During simulation, *D* processes the configurations of *N* in a breadth-first fashion.
- Thus *D* needs to maintain a queue of *N*'s configurations (Remember queues?)
- *D* gets the next configuration from the head of the queue.
- *D* creates copies of this configuration (as many as needed)
- On each copy, *D* simulates one of the nondeterministic moves of *N*.
- *D* places the resulting configurations to the back of the queue.

Structure of the Simulating DTM

- *N* is simulated with 2-tape DTM, *D*
  - Note that this is different from the construction in the book!

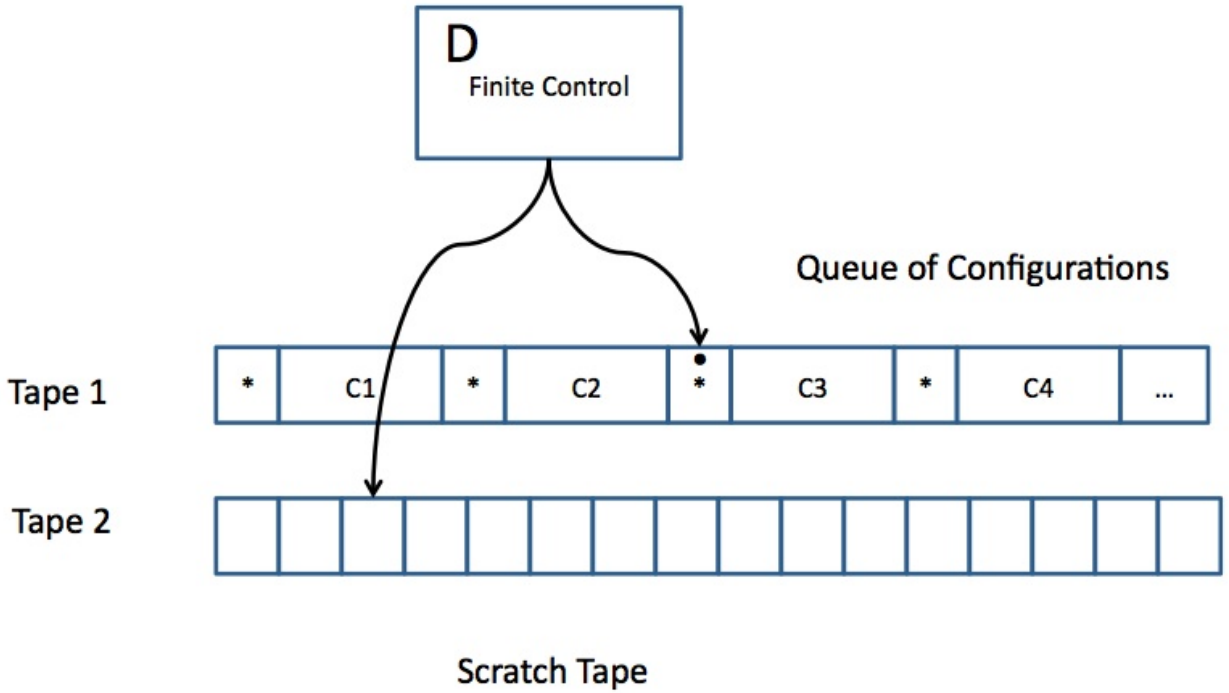


How *D* Simulates *N*



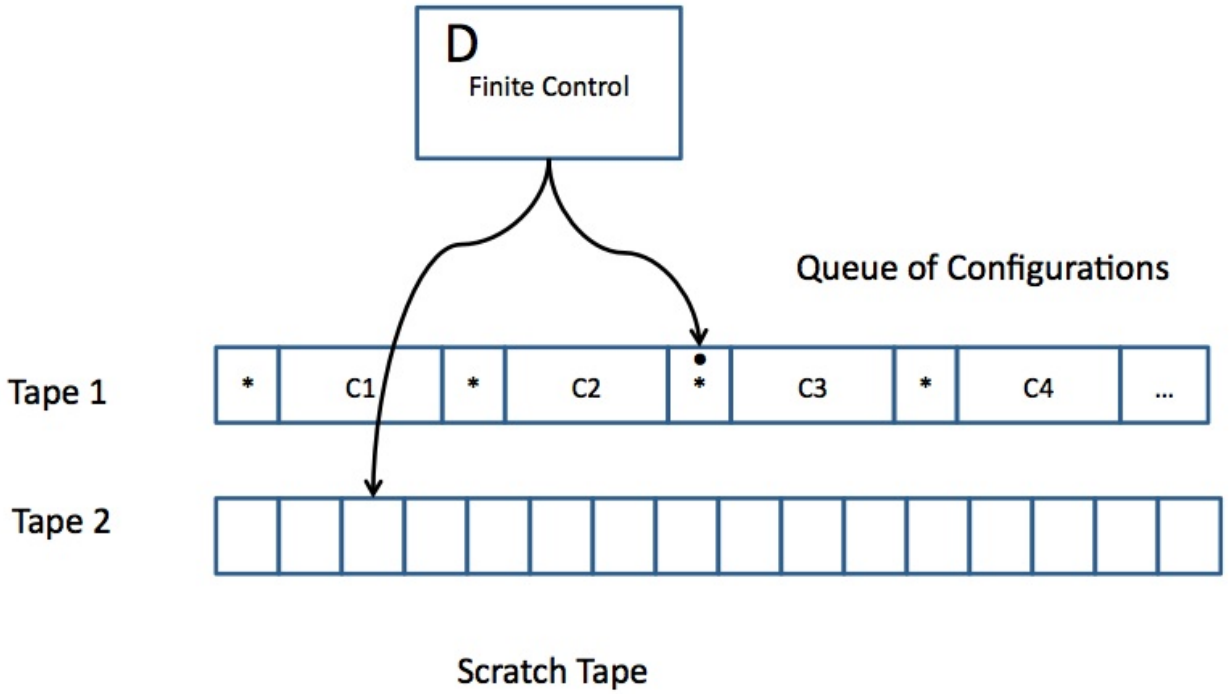
- Built into the finite control of *D* is the knowledge of what choices of moves *N* has for each state and input.

How *D* Simulates *N*



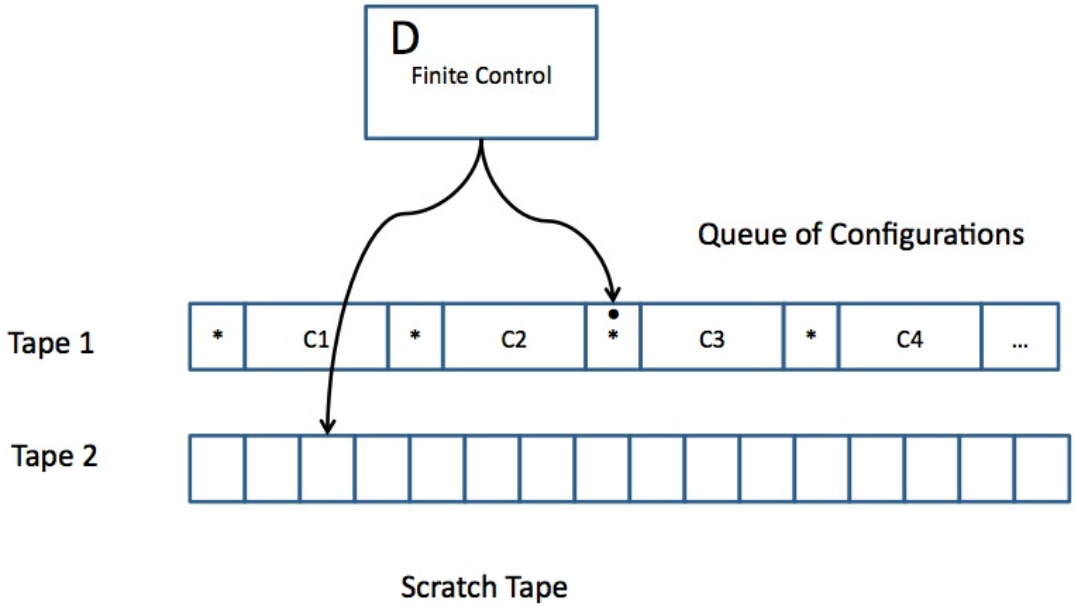
1. *D* examines the state and the input symbol of the current configuration (right after the dotted separator)
2. If the state of the current configuration is the accept state of *N*, then *D* accepts the input and stops simulating *N*.

How *D* Simulates *N*



1. *D* copies *k* copies of the current configuration to the scratch tape.
2. *D* then applies one nondeterministic move of *N* to each copy.

How *D* Simulates *N*



3. *D* then copies the new configurations from the scratch tape, back to the end of tape 1 (so they go to the back of the queue), and then clears the scratch tape.
4. *D* then returns to the marked current configuration, and "erases" the mark, and "marks" the next configuration.
5. *D* returns to step 1), if there is a next configuration. Otherwise rejects.

How *D* Simulates *N*

- Let m be the maximum number of choices N has for any of its states.
- Then, after n steps, N can reach at most $1 + m + m^2 + \cdots + m^n$ configurations (which is at most nm^n)
- Thus D has to process at most this many configurations to simulate n steps of N.
- Thus the simulation can take exponentially more time than the nondeterministic TM.
- It is not known whether or not this exponential slowdown is necessary.

Implications

Corollary

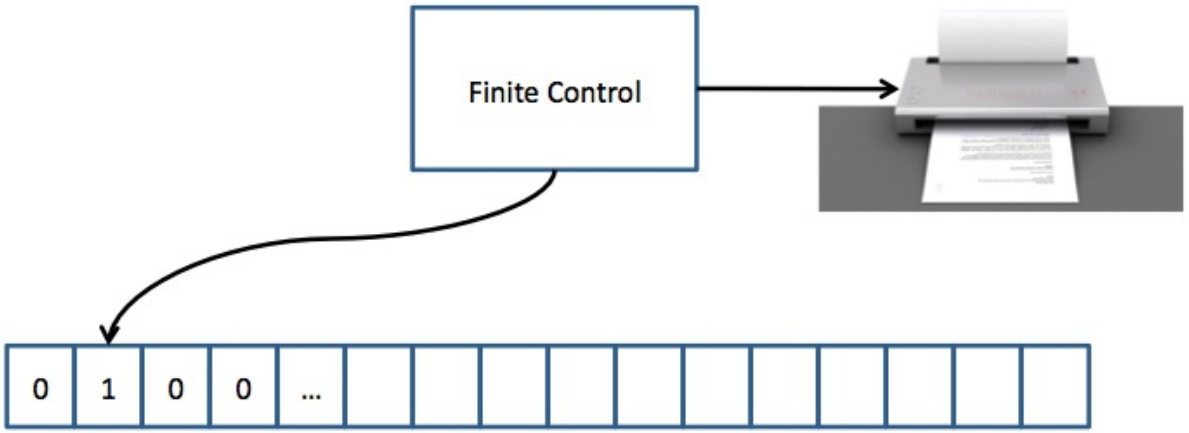
A language is Turing-recognizable if and only if some nondeterministic TM recognizes it.

Corollary

A language is decidable if and only of some nondeterministic TM decides it.

ENUMERATORS

- Remember we noted that some books used the term recursively enumerable for Turing-recognizable.
- This term arises from a variant of a TM called an enumerator.



- TM generates strings one by one.
- Everytime the TM wants to add a string to the list, it sends it to the printer.

ENUMERATORS

- The enumerator E starts with a blank input tape.
- If it does not halt, it may print an infinite list of strings.
- The strings can be enumerated in any order; repetitions are possible.
- The language of the enumerator is the collection of strings it eventually prints out.

ENUMERATORS

THEOREM

A language is Turing recognizable if and only if some enumerator enumerates it.

PROOF.

The If-part: If an enumerator E enumerates the language A then a TM M recognizes A.

$M =$ "On input w

1. Run E. Everytime E outputs a string, compare it with w.
2. If w ever appears in the output of E, *accept.*"

Clearly M accepts only those strings that appear on E's list.

□

ENUMERATORS

THEOREM

A language is Turing recognizable if and only if some enumerator enumerates it.

PROOF.

The Only-If-part: If a TM M recognizes a language A, we can construct the following enumerator for A. Assume $s_1, s_2, s_3, \ldots$ is a list of possible strings in Σ^*.

$E =$ "Ignore the input

1. Repeat the following for $i = 1, 2, 3, \ldots$
2. Run M for i steps on each input $s_1, s_2, s_3, \ldots s_i$.
3. If any computations accept, print out corresponding s_j."

If M accepts a particular string, it will appear on the list generated by E (in fact infinitely many times)

THE DEFINITION OF ALGORITHM - HISTORY

- in 1900, Hilbert posed the following problem:

  > *"Given a polynomial of several variables with integer coefficients, does it have an integer root – an assignment of integers to variables, that make the polynomial evaluate to 0"*

- For example, $6x^3yz^2 + 3xy^2 - x^3 - 10$ has a root at $x = 5, y = 3, z = 0$.
- Hilbert explicitly asked that an algorithm/procedure to be "devised". He assumed it existed; somebody needed to find it!
- 70 years later it was shown that no algorithm exists.
- The intuitive notion of an algorithm may be adequate for giving algorithms for certain tasks, but was useless for showing no algorithm exists for a particular task.

The Definition of Algorithm - History

- In early 20^{th} century, there was no formal definition of an algorithm.
- In 1936, Alonzo Church and Alan Turing came up with formalisms to define algorithms. These were shown to be equivalent, leading to the

Church-Turing Thesis

Intutitive notion of algorithms $\equiv$ Turing Machine Algorithms

-

THE DEFINITION OF AN ALGORITHM

- Let $D = \{p \mid p \text{ is a polynomial with integral roots}\}$
- Hilbert's 10^{th} problem in TM terminology is "Is D decidable?" (No!)
- However D is Turing-recognizable!
- Consider a simpler version
  $D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with integral roots}\}$
- $M_1 =$ "The input is polynomial p over x.
  1. Evaluate p with x successively set to 0, 1, -1, 2, -2, 3, -3, ...
  2. If at any point, p evaluates to 0, *accept*."
- D_1 is actually decidable since only a finite number of x values need to be tested (math!)
- D is also recognizable: just try systematically all integer combinations for all variables.

Describing Turing Machines and Their Inputs

- For the rest of the course we will have a rather standard way of describing TMs and their inputs.
- The input to TMs have to be strings.
- Every object O that enters a computation will be represented with an string $\langle O \rangle$, encoding the object.
- For example if G is a 4 node undirected graph with 4 edges $\langle O \rangle =$ `(1,2,3,4)((1,2),(2,3),(3,1),(1,4))`
- Then we can define problems over graphs,e.g., as:

$$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$$

Describing Turing Machines and Their Inputs

- A TM for this problem can be given as:
- $M =$ "On input $\langle G \rangle$, the encoding of a graph G:
  1. Select the first node of G and mark it.
  2. Repeat 3) until no new nodes are marked
  3. For each node in G, mark it, if there is edge attaching it to an already marked node.
  4. Scan all the nodes in G. If all are marked, the *accept*, else *reject*"

Other object encodings

- DFAs: Represent as a graph with 4 components, q_0, F, δ as a list of labeled edges.
- TMs: Represent as a string encoding δ with blocks of 5 components, e.g., q_i, a, q_j, b, L. Assume that q_0 is always the start state and q_1 is the final state.
  - Individual symbols can even be encoded using only two symbols e.g. just $\{0, 1\}$.

Formal Languages, Automata and Computation

Decidability

TURING MACHINES-SYNOPSIS

- The most general model of computation
- Computations of a TM are described by a sequence of configurations. (Accepting Configuration, Rejecting Configuration)
- Turing-recognizable languages
  - TM halts in an accepting configuration if w is in the language.
  - TM may halt in a rejecting configuration or go on indefinitely if w is not in the language.
- Turing-decidable languages
  - TM halts in an accepting configuration if w is in the language.
  - TM halts in a rejecting configuration if w is not in the language.
- Nondeterministic TMs are equivalent to Deterministic TMs.

Describing Turing Machines and Their Inputs

- For the rest of the course we will have a rather standard way of describing TMs and their inputs.
- The inputs to TMs have to be strings.
- Every object O that enters a computation will be represented with a string $\langle O \rangle$, encoding the object.
- For example if G is a 4 node undirected graph with 4 edges $\langle G \rangle = \texttt{(1,2,3,4)((1,2),(2,3),(3,1),(1,4))}$
- Then we can define problems over graphs,e.g., as:

$$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$$

DECIDABILITY

- We investigate the power of algorithms to solve problems.
- We discuss certain problems that can be solved algorithmically and others that can not be.
- Why discuss unsolvability?
- Knowing a problem is unsolvable is useful because
  - you realize it must be simplified or altered before you find an algorithmic solution.
  - you gain a better perspective on computation and its limitations.

Overview

- Decidable Languages
- Diagonalization
- Halting Problem as a undecidable problem
- Turing-unrecognizable languages.

Decidable Languages

Some notational details

- $\langle B \rangle$ represents the encoding of the description of an automaton (DFA/NFA).
- We need to encode Q, Σ, δ and F.

Encoding Finite Automata as Strings

- Here is one possible encoding scheme:
- Encode Q using unary encoding:
  - For $Q = \{q_0, q_1, \ldots q_{n-1}\}$, encode q_i using $i + 1$ 0's, i.e., using the string 0^{i+1}.
  - We assume that q_0 is always the start state.
- Encode Σ using unary encoding:
  - For $\Sigma = \{a_1, a_2, \ldots a_m\}$, encode a_i using i 0's, i.e., using the string 0^i.
- With these conventions, all we need to encode is δ and F!
- Each entry of δ, e.g., $\delta(q_i, a_j) = q_k$ is encoded as

$$\underbrace{0^{i+1}}_{q_i} 1 \underbrace{0^{j}}_{a_j} 1 \underbrace{0^{k+1}}_{q_k}$$

Encoding Finite Automata as Strings

- The whole δ can now be encoded as

$$\underbrace{00100001000}_{transition_1}1\underbrace{000001001000000}_{transition_2}\cdots 1\underbrace{000000100000010}_{transition_t}$$

- F can be encoded just as a list of the encodings of all the final states. For example, if states 2 and 4 are the final states, F could be encoded as

$$\underbrace{000}_{q_2}1\underbrace{00000}_{q_4}$$

- The whole DFA would be encoded by

$$11\underbrace{00100010000100000\cdots 0}_{\text{encoding of the transitions}}11\underbrace{000000001000000}_{\text{encoding of the final states}}11$$

Encoding Finite Automata as Strings

- $\langle B \rangle$ representing the encoding of the description of an automaton (DFA/NFA) would be something like

$$\langle B \rangle = 11\underbrace{00100010000100000\cdots0}_{\textit{encoding of the transitions}}11\underbrace{0000000010000000}_{\textit{encoding of the final states}}11$$

- In fact, the description of all DFAs could be described by a regular expression like

$$11(0^+10^+10^+1)^*1(0^+1)^+1$$

- Similarly strings over Σ can be encoded with (the same convention)

$$\langle w \rangle = \underbrace{0000}_{a_4}1\underbrace{000000}_{a_6}1\cdots\underbrace{0}_{a_1}$$

Encoding Finite Automata as Strings

- $\langle B, w \rangle$ represents the encoding of a machine followed by an input string, in the manner above (with a suitable separator between $\langle B \rangle$ and $\langle w \rangle$.
- Now we can describe our problems over languages and automata as problems over strings (representing automata and languages).

Decidable Problems

Regular Languages

- Does B accept w?
- Is $L(B)$ empty?
- Is $L(A) = L(B)$?

The Acceptance Problem for DFAs

Theorem 4.1

$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$ is a decidable language.

Proof

- Simulate with a two-tape TM.
  - One tape has $\langle B, w\rangle$
  - The other tape is a work tape that keeps track of which state of B the simulation is in.
- $M =$ "On input $\langle B, w\rangle$
  1. Simulate B on input w
  2. If the simulation ends in an accept state of B, *accept*; if it ends in a nonaccepting state, *reject*."

The Acceptance Problem for NFAs

Theorem 4.2

$A_{NFA} = \{\langle B, w\rangle \mid B \text{ is a NFA that accepts input string } w\}$ is a decidable language.

Proof

- Convert NFA to DFA and use Theorem 4.1
- $N =$ "On input $\langle B, w\rangle$ where B is an NFA
  1. Convert NFA B to an equivalent DFA C, using the determinization procedure.
  2. Run TM M in Thm 4.1 on input $\langle C, w\rangle$
  3. If M accepts *accept*; otherwise *reject*."

The Generation Problem for Regular Expressions

Theorem 4.3

$A_{REX} = \{\langle R, w\rangle \mid R \text{ is a regular exp. that generates string } w\}$ is a decidable language.

Proof

- Note R is already a string!!
- Convert R to an NFA and use Theorem 4.2
- $P =$ "On input $\langle R, w\rangle$ where R is a regular expression
  1. Convert R to an equivalent NFA A, using the Regular Expression-to-NFA procedure
  2. Run TM N in Thm 4.2 on input $\langle A, w\rangle$
  3. If N accepts *accept*; otherwise *reject*."

The Emptiness Problem for DFAs

Theorem 4.4

$E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \Phi\}$ is a decidable language.

Proof

- Use the DFS algorithm to mark the states of DFA
- $T =$ "On input $\langle A \rangle$ where A is a DFA.
  1. Mark the start state of A
  2. Repeat until no new states get marked.
     - Mark any state that has a transition coming into it from any state already marked.
  3. If no final state is marked, *accept*; otherwise *reject*."

The Equivalence Problem for DFAs

Theorem 4.5

$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ is a decidable language.

Proof

- Construct the machine for $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$ and check if $L(C) = \Phi$.
- $T =$ "On input $\langle A, B\rangle$ where A and B are DFAs.
  1. Construct the DFA for $L(C)$ as described above.
  2. Run TM T of Theorem 4.4 on input $\langle C\rangle$.
  3. If T accepts, *accept*; otherwise *reject*."

Decidable Problems

Context-free Languages

- Does grammar G generate w?
- Is $L(G)$ empty?

The Generation Problem for CFGs

Theorem 4.7

$A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$ is a decidable language.

Proof

- Convert G to Chomsky Normal Form and use the CYK algorithm.
- $C =$ "On input $\langle G, w\rangle$ where G is a CFG
  1. Convert G to an equivalent grammar in CNF
  2. Run CYK algorithm on w of length n
  3. If $S \in V_{i,n}$ *accept*; otherwise *reject*."

The Generation Problem for CFGs

Alternative Proof

- Convert G to Chomsky Normal Form and check all derivations up to a certain length (Why!)
- $S =$ "On input $\langle G, w \rangle$ where G is a CFG
  1. Convert G to an equivalent grammar in CNF
  2. List all derivations with $2n - 1$ steps where n is the length of w. If $n = 0$ list all derivations of length 1.
  3. If any of these strings generated is equal to w, *accept*; otherwise *reject*."

- This works because every derivation using a CFG in CNF either increase the length of the sentential form by 1 (using a rule like $A \to BC$ or leaves it the same (using a rule like $A \to a$)
- Obviously this is not very efficient as there may be exponentially many strings of length up to $2n - 1$.

The Emptiness Problem for CFGs

Theorem 4.8

$E_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \Phi\}$ is a decidable language.

Proof

- Mark variables of G systematically if they can generate terminal strings, and check if S is unmarked.
- $R =$ "On input $\langle G \rangle$ where G is a CFG.
  1. Mark all terminal symbols G
  2. Repeat until no new variable get marked.
     - Mark any variable A such that G has a rule $A \to U_1 U_2 \cdots U_k$ and $U_1, U_2, \ldots U_k$ are already marked.
  3. If start symbol is NOT marked, *accept*; otherwise *reject*."

The Equivalence Problem for CFGs

$EQ_{CFG} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$

- It turns out that EQ_{DFA} is not a decidable language.
- The construction for DFAs does not work because CFLs are NOT closed under intersection and complementation.
- Proof comes later.

Decidability of CFLs

Theorem 4.9

Every context free language is decidable.

Proof

- Design a TM M_G that has G built into it and use the result of A_{CFG}.
- $M_G =$ "On input w
  1. Run TM S (from Theorem 4.7) on input $\langle G, w \rangle$
  2. If S accepts, *accept*, otherwise *reject*.

Acceptance Problem for TMs

Theorem 4.11

$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable.

- Note that A_{TM} is Turing-recognizable. Thus this theorem when proved, shows that recognizers are more powerful than deciders.
- We can encode TMs with strings just like we did for DFA's (How?)

ACCEPTANCE PROBLEM FOR TMS

- The TM U recognizes A_{TM}
- $U =$ "On input $\langle M, w \rangle$ where M is a TM and w is a string:
  1. Simulate M on w
  2. If M ever enters its accepts state, *accept*; if M ever enters its reject state, *reject*.
- Note that if M loops on w, then U loops on $\langle M, w \rangle$, which is why it is NOT a decider!
- U can not detect that M halts on w.
- A_{TM} is also known as the Halting Problem
- U is known as the Universal Turing Machine because it can simulate every TM (including itself!)

The Diagonalization Method

Some basic definitions

- Let A and B be any two sets (not necessarily finite) and f be a function from A to B.
- f is one-to-one if $f(a) \neq f(b)$ whenever $a \neq b$.
- f is onto if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$.
- We say A and B are the same size if there is a one-to-one and onto function $f : A \longrightarrow B$.
- Such a function is called a correspondence for pairing A and B.
  - Every element of A maps to a unique element of B
  - Each element of B has a unique element of A mapping to it.

The Diagonalization Method

- Let $\mathcal{N}$ be the set of natural numbers $\{1, 2, \ldots\}$ and let $\mathcal{E}$ be the set of even numbers $\{2, 4, \ldots\}$.
- $f(n) = 2n$ is a correspondence between $\mathcal{N}$ and $\mathcal{E}$.
- Hence, $\mathcal{N}$ and $\mathcal{E}$ have the same size (though $\mathcal{E} \subset \mathcal{N}$).
- A set A is countable if it is either finite or has the same size as $\mathcal{N}$.
- $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathcal{N}\}$ is countable!
- Z the set of integers is countable:

$$f(n) = \begin{cases} \frac{n}{2} & n \text{ even} \\ -\frac{n+1}{2} & n \text{ odd} \end{cases}$$

The Diagonalization Method

Theorem

$\mathcal{R}$ is uncountable

Proof.

- Assume f exists and every number in $\mathcal{R}$ is listed.
- Assume $x \in \mathcal{R}$ is a real number such that x differs from the j^{th} number in the j^{th} decimal digit.
- If x is listed at some position k, then it differs from itself at k^{th} position; otherwise the premise does not hold
- f does not exist

n	$f(n)$
1	3.14159...
2	55.77777...
3	0.12345...
4	0.50000...
⋮	⋮

$x = .4527\ldots$ defined as such, can not be on this list.

□

Diagonalization over Languages

Corollary

Some languages are not Turing-recognizable.

Proof

- For any alphabet Σ, Σ^* is countable. Order strings in Σ^* by length and then alphanumerically, so $\Sigma^* = \{s_1, s_2, \ldots, s_i, \ldots\}$
- The set of all TMs is a countable language.
  - Each TM M corresponds to a string $\langle M \rangle$.
  - Generate a list of strings and remove any strings that do not represent a TM to get a list of TMs.

Diagonalization over Languages

Proof (continued)

- The set of infinite binary sequences, $\mathcal{B}$, is uncountable. (Exactly the same proof we gave for uncountability of $\mathcal{R}$)
- Let $\mathcal{L}$ be the set of all languages over Σ.
- For each language $A \in \mathcal{L}$ there is unique infinite binary sequence χ_A
  - The i^{th} bit in χ_A is 1 if $s_i \in A$, 0 otherwise.

$\Sigma^*=\{$	ϵ,	0,	1,	00,	01,	10,	11,	000,	001,	$\cdots$ $\}$
$A=\{$		0,		00,	01,			000,	001,	$\cdots$ $\}$
$\chi_A=\{$	0	1	0	1	1	0	0	1	1	$\cdots$ $\}$

Diagonalization over Languages

Proof (continued)

- The function $f : \mathcal{L} \longrightarrow \mathcal{B}$ is a correspondence. Thus $\mathcal{L}$ is uncountable.
- So, there are languages that can not be recognized by some TM. There are not enough TMs to go around.

The Halting Problem is Undecidable

Theorem

$A_{TM} = \{\langle M, w\rangle \mid M$ is a TM and M accepts $w\}$, is undecidable.

Proof

- We assume A_{TM} is decidable and obtain a contradiction.
- Suppose H decides A_{TM}

$$H(\langle M, w\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ accepts } w \\ \textit{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

The Halting Problem is Undecidable

Proof (continued)

- We now construct a new TM D
  $D =$ "On input $\langle M \rangle$, where M is a TM
  1. Run H on input $\langle M, \langle M \rangle \rangle$.
  2. If H accepts, *reject*, if H rejects, *accept*"
- So

$$D(\langle M \rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ does not accept } \langle M \rangle \\ \textit{reject} & \text{if } M \text{ accepts } \langle M \rangle \end{cases}$$

- When D runs on itself we get

$$D(\langle D \rangle) = \begin{cases} \textit{accept} & \text{if } D \text{ does not accept } \langle D \rangle \\ \textit{reject} & \text{if } D \text{ accepts } \langle D \rangle \end{cases}$$

- Neither D nor H can exist.

What Happened to Diagonalization?

Consider the behaviour of all possible deciders:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$ $\langle M_j \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	$\cdots$	*accept*	$\cdots$
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*	$\cdots$	*reject*	$\cdots$
M_4	*accept*	*accept*	*reject*	*reject*	$\cdots$	*accept*	$\cdots$
$\vdots$	$\vdots$	$\vdots$			$\ddots$		
$D = M_j$	*reject*	*reject*	*accept*	*accept*	$\cdots$	?	$\cdots$
$\vdots$	$\vdots$	$\vdots$					$\ddots$

- D computes the opposite of the diagonal entries!

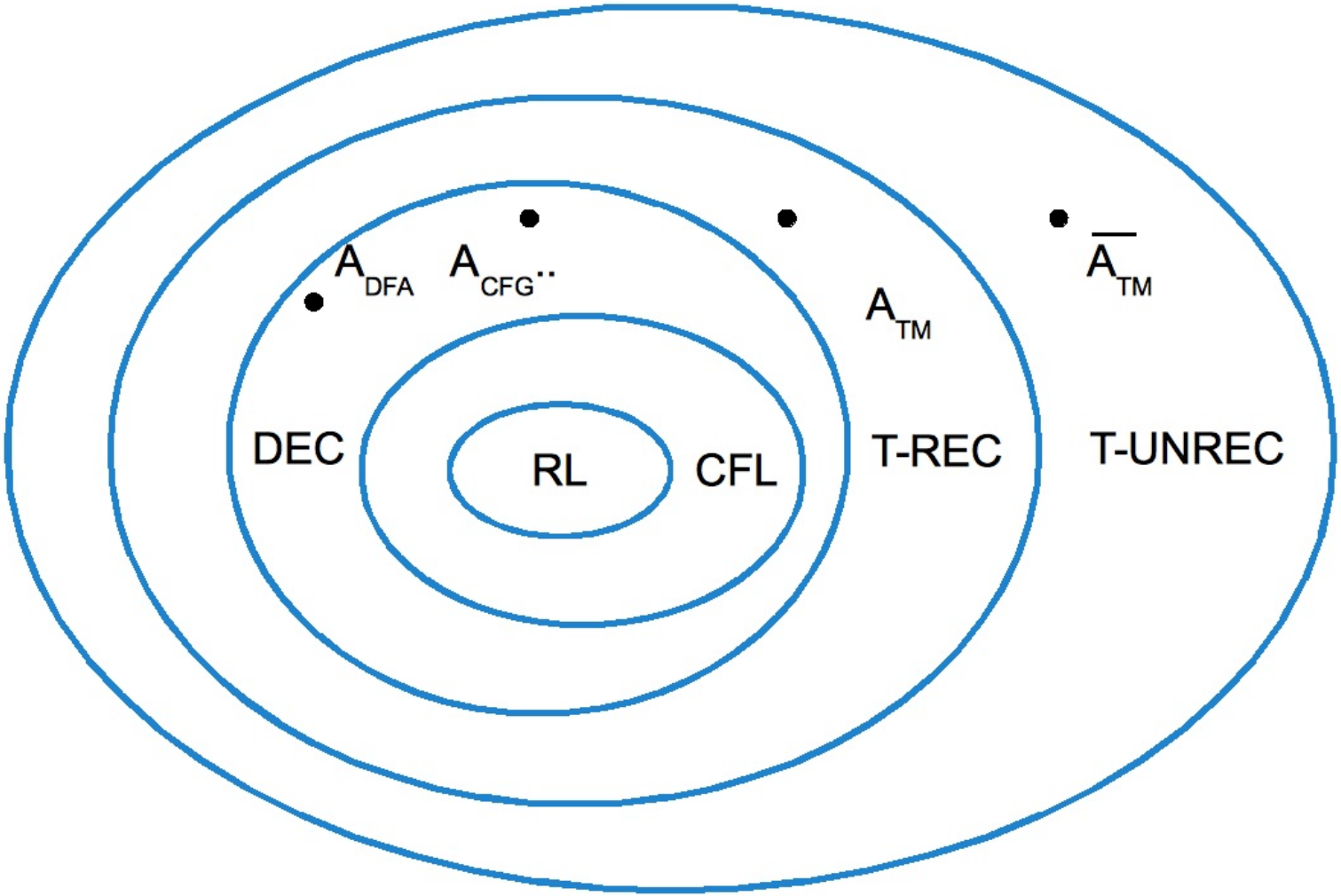
A Turing Unrecognizable Language

- A language is co-Turing-recognizable if it is the complement of a Turing-recognizable language.
- A language is decidable if it is Turing-recognizable and co-Turing-recognizable.
- $\overline{A_{TM}}$ is not Turing recognizable.
  - We know A_{TM} is Turing-recognizable.
  - If $\overline{A_{TM}}$ were also Turing-recognizable, A_{TM} would have to be decidable.
  - We know A_{TM} is not decidable.
  - $\overline{A_{TM}}$ must not be Turing-recognizable.

Formal Languages, Automata and Computation

Reducibility

The Landscape of the Chomsky Hierarchy



Reducibility-The Fundamental Ideas

- A reduction is a way of converting one problem to another problem, so that the solution to the second problem can be used to solve the first problem.
  - Finding the area of a rectangle, reduces to measuring its width and height
  - Solving a set of linear equations, reduces to inverting a matrix.
- Reducibility involves two problems A and B.
  - If A reduces to B, you can use a solution to B to solve A
- When A is reducible to B, solving A can not be "harder" than solving B.
- Two very important observations:
  - If A is reducible to B and B is decidable, then A is also decidable.
    - If you want to show a problem (A) is decidable, find a decidable problem (B), and see if you can reduce A to B.
  - If A is undecidable and reducible to B, then B is undecidable.
    - If you want to show a problem (B) is undecidable, find an undecidable problem (A), and see if you can reduce A to B.

Proving Undecidability via Reductions

Theorem 5.1

$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$ is undecidable.

Proof

- Use the idea that " If A is undecidable and reducible to B, then B is undecidable."
- Suppose R decides $HALT_{TM}$. We construct S to decide A_{TM}.
- $S =$ "On input $\langle M, w\rangle$
  1. Run R on input $\langle M, w\rangle$.
  2. If R rejects *reject*.
  3. If R accepts, simulate M on w until it halts.
  4. If M has accepted, accept; If M has rejected, reject."
- So if R exists, then I can build S which can decide A_{TM} (which we already know is undecidable.)
- Since A_{TM} is reduced to $HALT_{TM}$, $HALT_{TM}$ is undecidable.

THEOREM 5.2

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Phi\}$ is undecidable.

- Suppose R decides E_{TM}. We try to construct S to decide A_{TM} using R.
  - Note that S takes $\langle M, w \rangle$ as input.
- One idea is to run R on $\langle M \rangle$ to check if M accepts some string or not – but that that does not tell us if M accepts w.
- Instead we modify M to M_1. M_1 rejects all strings other than w but on w, it does what M does.
- Now we can check if $L(M_1) = \Phi$.

Proving Undecidability via Reductions

Theorem 5.2

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Phi\}$ is undecidable.

Proof

- For any w define M_1 as
  $M_1 =$ "On input x:
  1. If $x \neq w$, *reject*.
  2. If $x = w$, run M on input w and accept if M does."
- Note that M_1 either accepts w only or nothing!

PROOF CONTINUED

- Assume R decides E_{TM}
- S defines below uses R to decide on A_{TM}
  $S =$ "On input $\langle M, w \rangle$
  1. Use $\langle M, w \rangle$ to construct M_1 above.
  2. Run R on input $\langle M_1 \rangle$
  3. If R accepts, reject, if R rejects, accept."
- So, if R decides M_1 is empty,
  - then M does NOT accept w,
  - else M accepts w.
- If R decides E_{TM} then S decides A_{TM} – Contradiction.

Testing for Regularity (or other properties)

- Can we find out if a language accepted by a Turing machine M is accepted by a simpler computational model?
  - Is the language of a TM actually a regular language? ($REGULAR_{TM}$)
  - Is the language of a TM actually a CFL? (CFL_{TM})
  - Does the language of a TM have an "interesting" property?
    - Rice's Theorem.

Testing for Regularity

$REGULAR_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language } \}$ is undecidable.

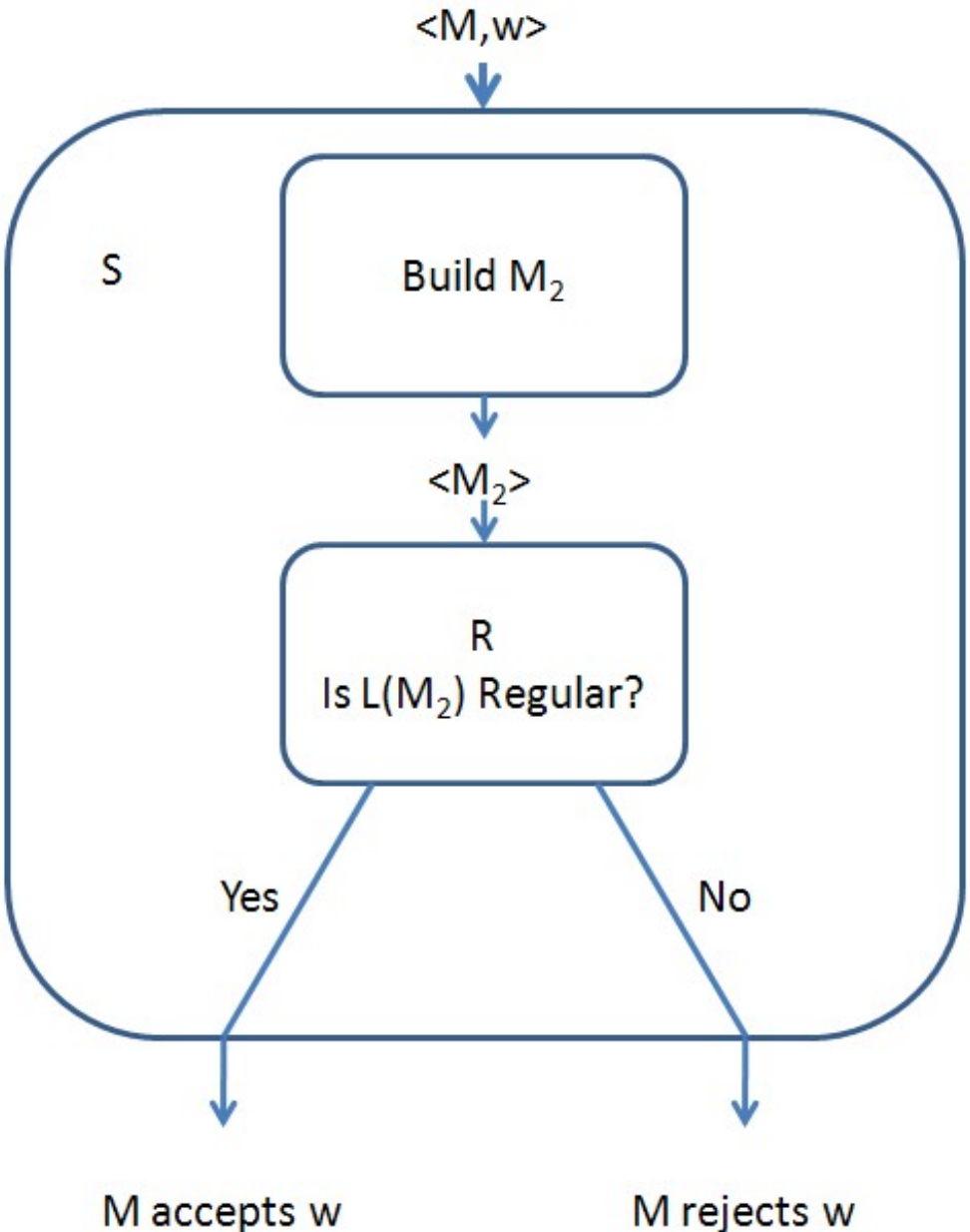
Proof Idea

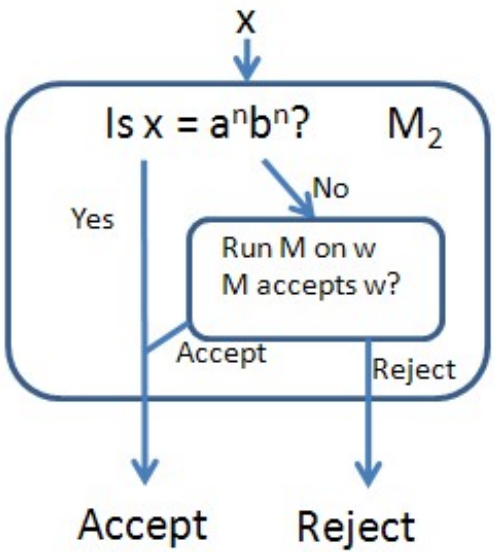
- We assume $REGULAR_{TM}$ is decidable by a TM R and use this assumption to construct a TM S that decides A_{TM}.
- The basic idea is for S to take as input $\langle M \rangle$ and modify M into M_2 so that the resulting TM recognizes a regular language if and only if M accepts w.
- M_2
  - accepts $\{0^n 1^n \mid n \geq 0\}$ if M does not accept w,
  - but recognizes Σ^* if M accepts w.

PROOF IDEA –CONTINUED

- M_2 accepts $\{0^n1^n \mid n \geq 0\}$ if M does not accept w, but recognizes Σ^* if M accepts w.
- What does M_2 look like?
- M_2 = "On input x
  1. If x has the form 0^n1^n, *accept*.
  2. If x does not have this form, run M on input w and *accept* if M accepts w." (w is set in an outer scope!)
- All strings x (that is Σ^*) are accepted if M accepts w.

Testing for Regularity





So L(M_2) is = Σ^* if M accepts w
L(M_2) is = $\{a^n b^n\}$ otherwise

Testing for Regularity

Proof

- $S =$ "On input $\langle M, w \rangle$, where M is a TM and w is a string:
  1. Construct the following TM M_2.
  2. $M_2 =$ "On input x
     1. If x has the form 0^n1^n, accept.
     2. If x does not have this form, run M on input w and accept if M accepts w."
  3. Run R on $\langle M_2 \rangle$
  4. If R accepts, accept, if R rejects, reject."
- So, R will say M_2 is a regular language, if M accepts w.
- S says "M accepts w" if R decides M_2 is regular – Contradiction!

Testing for Language Equality

Theorem 5.4

$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ is undecidable.

Proof Idea

- We reduce E_{TM} (the emptiness problem) to this problem.
- If one of the languages is empty, determining equality is the same as determining if the second language is empty!
- In fact, the E_{TM} is a special case of the EQ_{TM} problem!!

Testing for Language Equality

Theorem 5.4

$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ is undecidable.

Proof

- Assume R decides EQ_{TM}
- $S =$ "On input $\langle M \rangle$ where M is a TM:
  1. Run R on input $\langle M, M_1 \rangle$ where M_1 is a TM that rejects all inputs.
  2. If R accepts, accept; if R rejects reject"
- Thus, if R decides EQ_{TM}, then S decides E_{TM}
- But E_{TM} is undecidable, so EQ_{TM}, must be undecidable.

Reductions via Computation Histories

- An accepting computation history for a TM is a sequence of configurations

$$C_1, C_2, \ldots, C_l$$

such that

  - C_1 is the start configuration for input w
  - C_l is an accepting configuration, and
  - each C_i follows legally from the preceding configuration.

- A rejecting computation history is defined similarly.
- Computation histories are finite sequences – if M does not halt on w, there is no computation history.
- Deterministic v.s nondeterministic computation histories.

Linear Bounded Automaton

- Suppose we cripple a TM so that the head never moves outside the boundaries of the input string.
- Such a TM is called a linear bounded automaton (LBA)
- Despite their memory limitation, LBAs are quite powerful.

Lemma

Let M be a LBA with q states, g symbols in the tape alphabet. There are exactly qng^n distinct configurations for a tape of length n.

Proof.

- The machine can be in one of q states.
- The head can be on one of the n cells.
- At most g^n distinct strings can occur on the tape.

□

Decidability of LBA Problems

Theorem 5.9

$A_{LBA} = \{\langle M, w\rangle \mid M \text{ is an LBA that accepts string } w\}$ is decidable.

Proof Idea

- We simulate LBA M on w with a TM L (which is NOT an LBA!)
- If during simulation M accepts or rejects, we accept or reject accordingly.
- What happens if the LBA M loops?
  - Can we detect if it loops?
- M has a finite number of configurations.
  - If it repeats any configuration during simulation, it is in a loop.
  - If M is in a loop, we will know this after a finite number of steps.
  - So if the LBA M has not halted by then, it is looping.

Decidability of LBA Problems

Theorem 5.9

$A_{LBA} = \{\langle M, w\rangle \mid M \text{ is an LBA that accepts string } w\}$ is decidable.

Proof

- The following TM decides A_{LBA}.
- $L =$ "On input $\langle M, w\rangle$
  1. Simulate M on for qng^n steps or until it halts.
  2. If M has halted, accept if it has accepted, and reject if it has rejected. If it has NOT halted, reject."
- LBAs and TMs differ in one important way. A_{LBA} is decidable.

Computation over "Computation Histories"

- Now for a really wild and crazy idea!
- Consider an accepting computation history of a TM M, $C_1, C_2, \ldots, C_l$
- Note that each C_i is a string.
- Consider the string

$$\#\underbrace{\qquad}_{C_1}\#\underbrace{\qquad}_{C_2}\#\underbrace{\qquad}_{C_3}\#\cdots\#\underbrace{\qquad}_{C_l}\#$$

- The set of all valid accepting histories is also a language!!
- This string has length m and an LBA B can check if this is a valid computation history for a TM M accepting w.
  - Check if $C_1 = q_0 w_1 w_2 \cdots w_n$
  - Check if $C_l = \cdots q_{accept} \cdots$
  - Check if each C_{i+1} follows from C_i legally.
- Note that B is not constructed for the purpose of running it on any input!
- If $L(B) \neq \Phi$ then M accepts w

Decidability of LBA Problems

Theorem 5.10

$E_{LBA} = \{\langle M \rangle \mid M \text{ is an LBA and } L(M) = \Phi\}$ is undecidable.

Proof.

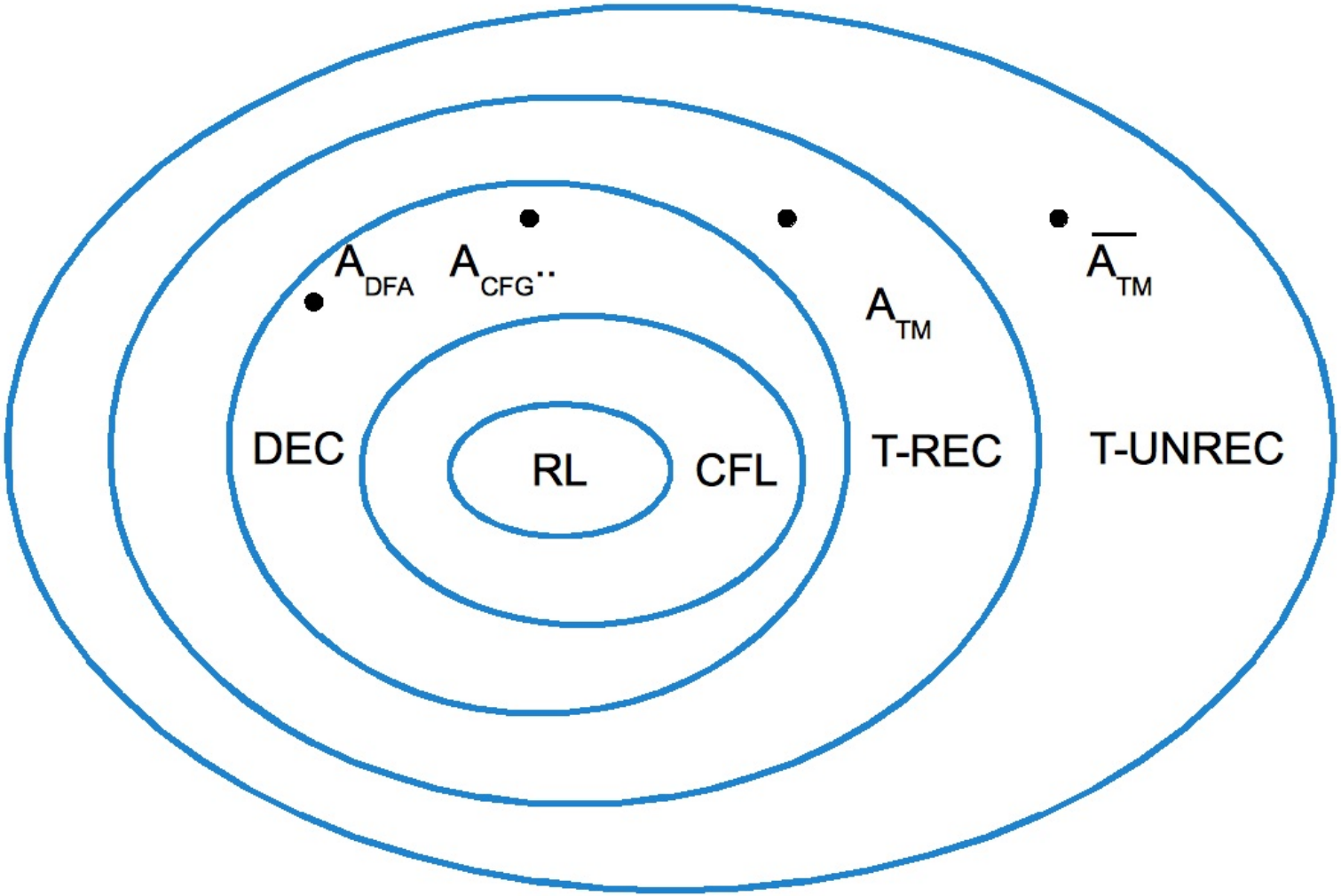
- Suppose TM R decides E_{LBA}, we can construct a TM S which decides A_{TM}
- $S =$ "On input $\langle M, w \rangle$, where M is a TM and w is a string
  1. Construct LBA B from M and w as described earlier.
  2. Run R on $\langle B \rangle$.
  3. If R rejects, accept; if R accepts, reject."
- So if R says $L(B) = \Phi$, the M does NOT accept w.
- If R says $L(B) \neq \Phi$, the M accepts w.
- But, A_{TM} is undecidable – contradiction.

□

Formal Languages, Automata and Computation

Post Correspondence Problem

Review of Decidability and Reductions



Reducibility

- A reduction is a way of converting one problem to another problem, so that the solution to the second problem can be used to solve the first problem.
  - Finding the area of a rectangle, reduces to measuring its width and height
  - Solving a set of linear equations, reduces to inverting a matrix.
- Reducibility involves two problems A and B.
  - If A reduces to B, you can use a solution to B to solve A
- When A is reducible to B, solving A can not be "harder" than solving B.
- If A is reducible to B and B is decidable, then A is also decidable.
- If A is undecidable and reducible to B, then B is undecidable.

THEOREM 5.2

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Phi\}$ is undecidable.

- Suppose R decides E_{TM}. We try to construct S to decide A_{TM} using R.
  - Note that S takes $\langle M, w \rangle$ as input.
- One idea is to run R on $\langle M \rangle$ to check if M accepts some string or not – but that that does not tell us if M accepts w.
- Instead we modify M to M_1. M_1 rejects all strings other than w but on w, it does what M does.
- Now we can check if $L(M_1) = \Phi$.

Proving Undecidability via Reductions

Theorem 5.2

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Phi\}$ is undecidable.

Proof

- For any w define M_1 as
  $M_1 =$ "On input x:
  1. If $x \neq w$, *reject*.
  2. If $x = w$, run M on input w and *accept* if M does."
- Note that M_1 either accepts w only or nothing!

PROOF CONTINUED

- Assume R decides E_{TM}
- S defines below uses R to decide on A_{TM}
  $S =$ "On input $\langle M, w \rangle$
  1. Use $\langle M, w \rangle$ to construct M_1 above.
  2. Run R on input $\langle M_1 \rangle$
  3. If R accepts, *reject*, if R rejects, *accept*.
- So, if R decides $L(M_1)$ is empty,
  - then M does NOT accept w,
  - else M accepts w.
- If R decides E_{TM} then S decides A_{TM} – Contradiction.

Reductions via Computation Histories

- An accepting computation history for a TM is a sequence of configurations

$$C_1, C_2, \ldots, C_l$$

  such that
  - C_1 is the start configuration for input w
  - C_l is an accepting configuration, and
  - each C_i follows legally from the preceding configuration.
- A rejecting computation history is defined similarly.
- Computation histories are finite sequences – if M does not halt on M, there is no computation history.
- Deterministic v.s nondeterministic computation histories.

Linear Bounded Automaton

- Suppose we cripple a TM so that the head never moves outside the boundaries of the input string.
- Such a TM is called a linear bounded automaton (LBA)
- Despite their memory limitation, LBAs are quite powerful.

Lemma

Let M be a LBA with q states, g symbols in the tape alphabet. There are exactly qng^n distinct configurations for a tape of length n.

Proof.

- The machine can be in one of q states.
- The head can be on one of the n cells.
- At most g^n distinct strings can occur on the tape.

□

Theorem 5.9

$A_{LBA} = \{\langle M, w\rangle \mid M \text{ is an LBA that accepts string } w\}$ is decidable.

Computation over "Computation Histories"

- Now for a really wild and crazy idea!
- Consider an accepting computation history of a TM M, $C_1, C_2, \ldots, C_l$
- Note that each C_i is a string.
- Consider the string

$$\#\underbrace{\qquad}_{C_1}\#\underbrace{\qquad}_{C_2}\#\underbrace{\qquad}_{C_3}\#\cdots\#\underbrace{\qquad}_{C_l}\#$$

- The set of all valid accepting histories is also a language!!
- This string has length m and an LBA B can check if this is a valid computation history for a TM M accepting w.
  - Check if $C_1 = q_0 w_1 w_2 \cdots w_n$
  - Check if $C_l = \cdots q_{accept} \cdots$
  - Check if each C_{i+1} follows from C_i legally.
- Note that B is not constructed for the purpose of running it on any input!
- If $L(B) \neq \Phi$ then M accepts w

Post Correspondence Problem

- Undecidability is not just confined to problems concerning automata and languages.
- There are other "natural" problems which can be proved undecidable.
- The Post correspondence problem (PCP) is a tiling problem over strings.
- A tile or a domino contains two strings, t and b; e.g., $\left[\frac{ca}{a}\right]$.
- Suppose we have dominos

$$\left\{\left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right]\right\}$$

- A match is a list of these dominos so that when concatenated the top and the bottom strings are identical. For example,

$$\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right] = \frac{abcaaabc}{abcaaabc}$$

- The set of dominos $\left\{\left[\frac{abc}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{acc}{ba}\right], \right\}$ does not have a solution.

AN INSTANCE OF THE PCP

A PCP instance over Σ is a finite collection P of dominos

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \cdots, \left[\frac{t_k}{b_k}\right] \right\}$$

where for all $i, 1 \leq i \leq k, t_i, b_i \in \Sigma^*$.

MATCH

Given a PCP instance P, a match is a nonempty sequence

$$i_1, i_2, \ldots, i_\ell$$

of numbers from $\{1, 2, \ldots, k\}$ (with repetition) such that $t_{i_1} t_{i_2} \cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}$

Post Correspondence Problem

Question:

Does a given PCP instance P have a match?

Language formulation:

$PCP = \{\langle P \rangle \mid P \text{ is a PCP instance and it has a match}\}$

Theorem 5.15

PCP is undecidable.

Proof: By reduction using computation histories. If PCP is decidable then so is A_{TM}. That is, if PCP has a match, then M accepts w.

PCP – The Structure of the Undecidability Proof

The reduction works in two steps:

1. We reduce A_{TM} to Modified PCP (MPCP).
2. We reduce MPCP to PCP.

MPCP as a language problem

$MPCP = \{\langle P \rangle \mid P$ is a PCP instance and it has a match which starts with index 1$\}$

- So the solution to MPCP starts with the domino $\left[\frac{t_1}{b_1}\right]$. We later remove this restriction in the second part of the proof.
- We also assume that the decider for M never moves its head to the left of the input w.

PCP – The Proof

For input $\langle M, w \rangle$ of A_{TM}, construct an MPCP instance such that M accepts w iff P' has a match starting with domino 1

- The first part of the proof proceeds in 7 stages where we add different types of dominos to P' depending on the TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$.
- Using the dominos, we try to construct an accepting computation history for M accepting w.

PCP – Adding the right kind of dominos

1. The first domino kicks of the computation history

$$\left[\frac{t_1}{b_1}\right] = \left[\frac{\#}{\#q_0 w_1 w_2 \cdots w_n \#}\right],$$

2. Handle right moving transitions. For every $a, b \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{reject}$

$$\text{if } \delta(q, a) = (r, b, R), \text{ put } \left[\frac{qa}{br}\right] \text{into } P'$$

3. Handle left moving transitions. For every $a, b, c \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{reject}$

$$\text{if } \delta(q, a) = (r, b, L), \text{ put } \left[\frac{cqa}{rcb}\right] \text{into } P'$$

4. For every $a \in \Gamma$ put $\left[\frac{\mathbf{a}}{\mathbf{a}}\right]$ into P'

5. Put $\left[\frac{\#}{\#}\right]$ and $\left[\frac{\#}{\sqcup\#}\right]$ into P'.

PCP - How the dominos work

- Let us assume $\Gamma = \{0, 1, 2, \sqcup\}$, $w = 0100$ and that $\delta(q_0, 0) = (q_7, 2, R)$
- Part 1 places the first domino and the match begins

#	q_0	0	1	0	0	#						
#	q_0	0	1	0	0	#	2	q_7	1	0	0	#

- Part 2 places the domino $\left[\frac{q_0 0}{2 q_7}\right]$
- Part 4 places the dominos $\left[\frac{0}{0}\right]$ $\left[\frac{1}{1}\right]$ $\left[\frac{2}{2}\right]$ and $\left[\frac{\sqcup}{\sqcup}\right]$ into P' so we can extend the match.
- Part 5 puts in the domino $\left[\frac{\#}{\#}\right]$
- What exactly is going on ?
- We force the bottom string to create a copy on the top which is forced to generate the next configuration on the bottom – We are simulating M on w!
- The process continues until M reaches a halting state and we then pad the upper string.

PCP – More domino types

6. For every $a \in \Gamma$,

$$\text{put } \left[\frac{\mathbf{a}\mathbf{q}_{\mathbf{accept}}}{\mathbf{q}_{\mathbf{accept}}}\right] \text{ and } \left[\frac{\mathbf{q}_{\mathbf{accept}}\mathbf{a}}{\mathbf{q}_{\mathbf{accept}}}\right] \text{into } P'$$

These dominos "clean-up" by adding any symbols to the top string while adding just the state symbol to the lower string.

Just before these apply the upper and lower strings are like

$$\ldots \#$$
$$\ldots \#\,\mathbf{2}\,\mathbf{1}\,\mathbf{q}_{\mathbf{accept}}\,\mathbf{0}\,\mathbf{2}\,\#$$

After using these dominos, we end up with

$$\ldots \#$$
$$\ldots \#\,\mathbf{q}_{\mathbf{accept}}\,\#$$

7. Finally we add the domino

$$\left[\frac{\mathbf{q}_{\mathbf{accept}}\#\#}{\#}\right]$$

to complete the match.

PCP Proof – Summary of Part 1

- This concludes the construction of P'.
- Thus if M accepts w, the set of MPCP dominos constructed have a solution to the MPCP problem.
- But not yet to the PCP problem.

PCP Proof – Part 2

- Suppose we have the MPCP instance

$$P' = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \cdots, \left[\frac{t_k}{b_k}\right] \right\}$$

- We let P be the collection

$$P = \left\{ \left[\frac{\star t_1}{\star b_1 \star}\right], \left[\frac{\star t_2}{b_2 \star}\right], \cdots, \left[\frac{\star t_k}{b_k \star}\right] \left[\frac{\star \diamond}{\diamond}\right] \right\}$$

- The only domino that could possibly start a match is the first one!
- The last domino just adds the missing $\star$ at the end of the match.

Conclusion

PCP is undecidable!

Summary of Reducibility

We know that language A is undecidable. By reducing A to B we want to show that the language B is also undecidable.

1. Assume that we have a decider M_B for B.
2. Using M_B we construct a decider M_A for the language A:

M_A = "On input $\langle I_A \rangle$

1. Algorithmically construct an input $\langle I_B \rangle$ for M_B, such that

   a) Either

   If $\langle I_A \rangle \in A$ then $\langle I_B \rangle \in B$
   If $\langle I_A \rangle \notin A$ then $\langle I_B \rangle \notin B$

   b) or

   If $\langle I_A \rangle \in A$ then $\langle I_B \rangle \notin B$
   If $\langle I_A \rangle \notin A$ then $\langle I_B \rangle \in B$

2. Run the decider M_B on $\langle I_B \rangle$ for M_B
   Case a): M_A accepts if M_B accepts, and rejects if M_B rejects
   Case b): M_A rejects if M_B accepts, and accepts if M_B reject.

3. We know M_A can not exist so M_B can not exist.
4. B is undecidable.

Computable Functions

Idea

Turing Machines can also compute function $f : \Sigma^* \longrightarrow \Sigma^*$.

Computable Function

A function $f : \Sigma^* \longrightarrow \Sigma^*$ is a computable function if and only if there exists a TM M_f, which on any given input $w \in \Sigma^*$

- always halts, and
- leaves just $f(w)$ on its tape.

Examples:

- Let $f(w) \stackrel{def}{=} ww$ be a function. Then f is computable.
- Let $f(\langle n_1, n_2 \rangle) \stackrel{def}{=} \langle n \rangle$ where n_1 and n_2 are integers and $n = n_1 * n_2$. Then f is computable.

Mapping Reducibility

Definition

Let $A, B \subseteq \Sigma^*$. We say that language A is mapping reducible to language B, written $A <_m B$, if and only if

1. There is a computable function $f : \Sigma^* \longrightarrow \Sigma^*$ such that
2. For every $w \in \Sigma^*$, $w \in A$ if and only if $f(w) \in B$.

The function f is called a reduction of A to B.

Theorem 5.22

If $A <_m B$ and B is decidable, then A is decidable.

Proof

Let M be a decider for B and f be a mapping from A to B. Then N decides A.
$N =$ "On input w

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs."

If $A <_m B$ and A is undecidable, then B is undecidable.

Mapping Reducibility

Theorem

$A_{TM} <_m HALT_{TM}$

Proof.

Construct a computable function f which maps $\langle M, w \rangle$ to $\langle M', w' \rangle$ such that

$$\langle M, w \rangle \in A_{TM} \text{ if and only if } \langle M', w' \rangle \in HALT_{TM}$$

$M_f =$ "On input $\langle M, w \rangle$

1. Construct the following machine M':
   $M' =$ "On input x
   1. Run M on x.
   2. If M accepts *accept*
   3. If M rejects *enter a loop*."
2. Output $\langle M', w \rangle$."

□

More examples of Mapping Reducibility

- Earlier we showed
  - $A_{TM} <_m MPCP$
  - $MPCP <_m PCP$
- In Theorem 5.4 we showed $E_{TM} <_m EQ_{TM}$. The reduction f maps from $\langle M \rangle$ to the output $\langle M, M_1 \rangle$ where M_1 is the machine that rejects all inputs.

Theorem 5.24

If $A <_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

Proof

Essentially the same as the previous proof.

Summary of Mapping Reducibility Results

Summary of Theorems

Assume that $A <_m B$. Then

1. If B is decidable then A is decidable.
2. If A is undecidable then B is undecidable.
3. If B is Turing-recognizable then A is Turing-recognizable.
4. If A is not Turing-recognizable then B is not Turing-recognizable.
5. $\overline{A} <_m \overline{B}$

Useful observation:

- Suppose you can show $A_{TM} <_m \overline{B}$
- This means $\overline{A_{TM}} <_m B$
- Since $\overline{A_{TM}}$ is Turing-unrecognizable then B is Turing-unrecognizable.

Example of Use

Theorem 5.30

$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1$ and M_2 are TMs and $L(M_1) = L(M_2)\}$ is neither Turing recognizable nor co-Turing-recognizable.

Proof Idea

We show

- $\overline{A_{TM}} \leq_m EQ_{TM}$
- $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$
- These then imply the theorem.

Example of Use

Proof for $\overline{A_{TM}} <_m EQ_{TM}$

We show $A_{TM} <_m \overline{EQ_{TM}}$ (and hence $\overline{A_{TM}} <_m EQ_{TM}$) with the following f:

$F =$ "On input $\langle M, w \rangle$ where M is a TM and w is a string:

1. Construct the following two machines M_1 and M_2
   $M_1 =$ "On any input:
   1. Reject"
   $M_2 =$ "On any input:
   1. Run M on w. If it accepts, *accept.*"
2. Output $\langle M_1, M_2 \rangle$."

- M_1 accepts nothing.
  - If M accepts w then M_2 accepts everything. So M_1 and M_2 are not equivalent.
  - If M does not accept w then M_2 accepts nothing. So M_1 and M_2 are equivalent.
- So $A_{TM} <_m \overline{EQ_{TM}}$ (and hence $\overline{A_{TM}} <_m EQ_{TM}$)

Example of Use

Proof for $\overline{A_{TM}} <_m \overline{EQ_{TM}}$

We show $A_{TM} <_m EQ_{TM}$ (and hence $\overline{A_{TM}} <_m \overline{EQ_{TM}}$) with the following g:

$G =$ "On input $\langle M, w \rangle$ where M is a TM and w is a string:

1. Construct the following two machines M_1 and M_2
   $M_1 =$ "On any input:
   1. Accept"

   $M_2 =$ "On any input:
   1. Run M on w. If it accepts, *accept.*"
2. Output $\langle M_1, M_2 \rangle$."

- M_1 accepts everything.
  - If M accepts w then M_2 accepts everything. So M_1 and M_2 are equivalent.
  - If M does not accept w then M_2 accepts nothing. So M_1 and M_2 are not equivalent.
- So $A_{TM} <_m EQ_{TM}$ (and hence $\overline{A_{TM}} <_m \overline{EQ_{TM}}$)

Formal Languages, Automata and Computation

Rice's Theorem – Self-reproducing TMs

Rice's Theorem – Motivation

Consider the following undecidable languages:

- $E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Phi\}$
- $TOTAL_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$
- $REGULAR_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular}\}$
- $L_{0101010} = \{\langle M \rangle \mid M \text{ is a TM and } 0101010 \in L(M)\}$

Question

What do these questions about languages have in common, so that they are all undecidable?

- They ask whether the language defined by a TM has a certain property.
- The properties are "nontrivial".
  - What is a "nontrivial" property?

Idea

We can generalize the undecidability proofs into a meta-theorem that works for all languages that talk about nontrivial properties of Turing machine languages.

What is a Nontrivial Property?

Definition (Property)

A language $\mathcal{P}$ is called a property of Turing machine languages iff

- $\mathcal{P} \subseteq \{\langle M \rangle \mid M \text{ is a TM}\}$
- For any two TMs M_1, M_2, if $L(M_1) = L(M_2)$ then $\langle M_1 \rangle \in \mathcal{P}$ iff $\langle M_2 \rangle \in \mathcal{P}$.

Definition (Nontrivial Property)

A language $\mathcal{P}$ which is a property of Turing machine languages is nontrivial iff:

- There is a TM M_1 such that $\langle M_1 \rangle \in \mathcal{P}$, and
- There is a TM M_2 such that $\langle M_2 \rangle \notin \mathcal{P}$.

- All these languages are nontrivial
  - $E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Phi\}$
  - $TOTAL_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$
  - $L_{0101010} = \{\langle M \rangle \mid M \text{ is a TM and } 0101010 \in L(M)\}$

Rice's Theorem

Theorem

Every language $\mathcal{P}$ which is a nontrivial property of Turing machine languages is undecidable!

Proof – Preliminaries

Assume a nontrivial property language $\mathcal{P} \subseteq \{\langle M \rangle \mid M \text{ is a TM}\}$. We want to show $\mathcal{P}$ is undecidable.

Consider the following two Turing machines:

- Let M_ϕ = "On input x: *reject*".
  - We can assume $\langle M_\phi \rangle \notin \mathcal{P}$.
  - If $\langle M_\phi \rangle \in \mathcal{P}$, then we show $\overline{\mathcal{P}}$ is undecidable.
- Let M_P be a TM such that $\langle M_P \rangle \in \mathcal{P}$.
  - M_P exists because $\mathcal{P}$ is nontrivial.

Proof by reduction from A_{TM} to $\mathcal{P}$

1. Assume we have a decider R_P for $\mathcal{P}$.
2. We show that using R_P we can construct a decider S for A_{TM}.

S = "On input $\langle M, w \rangle$

1. Construct a TM M_w as follows:
   M_w = "On input x:
   1. Run M on w.
      If M rejects then *reject*
   2. Else run M_P on x.
      If M_P accepts then *accept*."
2. Run R_P (the decider for $\mathcal{P}$) on $\langle M_w \rangle$
3. If R_P accepts then *accept*
   If R_P rejects then *reject*"

- If M accepts w, then $L(M_w) = L(M_P)$. So $\langle M_w \rangle \in \mathcal{P}$.
- If M does not accept w, then $L(M_w) = \Phi$.
  So $\langle M_w \rangle \notin \mathcal{P}$.
- So if R_P decides $\mathcal{P}$, then S decides A_{TM}.
- But we know the S does not exist, so R_P can not exist either.
- Conclusion: $\mathcal{P}$ is an undecidable language.

Applying Rice's Theorem

- The following languages are all undecidable:
  - $EPSILON_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } \epsilon \in L(M)\}$
  - $CFL_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a CFL}\}$
  - $DECIDABLE_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is decidable}\}$
  - $PAL_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ contains all palindromes}\}$
- Rice's Theorem is a very powerful tool
  - Very Important: we need to be checking a property of the language of the TM, not a property of the TM and the behaviour of the TM.

Common Pitfalls

Rice's Theorem can not be applied to the following languages:

- $ALL = \{\langle M \rangle \mid M \text{ is a TM } \}$
  - Note that ALL is decidable!
  - There is no language property involved here.We need to check a property of the representation!
- $TWICE = \{\langle M \rangle \mid M \text{ is a TM that visits the initial state more than twice}\}$
  - Again, this is not a question about the language defined by M but rather on the behaviour of M (Undecidable)
- $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$
  - Again, this is not a question about the property of a language. (Undecidable)

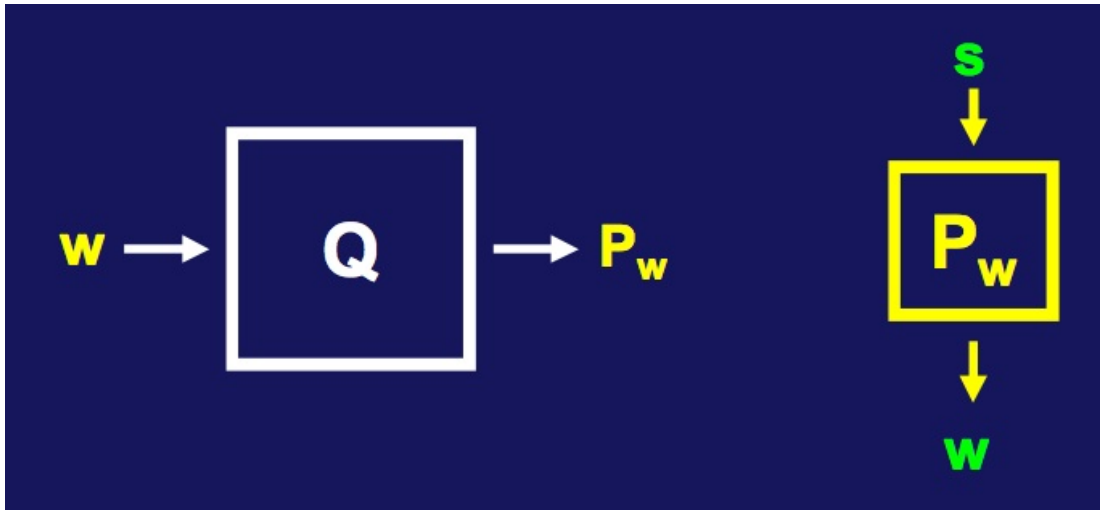
Self-Reference

- Can automata self-reproduce?
  - What do you mean?
    - Living things are "machines" and they reproduce!

Lemma

There is a computable function $q : \Sigma^* \longrightarrow \Sigma^*$ where

- if w is any string,
- $q(w)$ is the description of a Turing machine P_w that prints out w and halt.



Self-Reference

Lemma

There is a computable function $q : \Sigma^* \longrightarrow \Sigma^*$ where if w is any string, $q(w)$ is the description of a Turing machine P_w that prints out w and halt.

Proof:

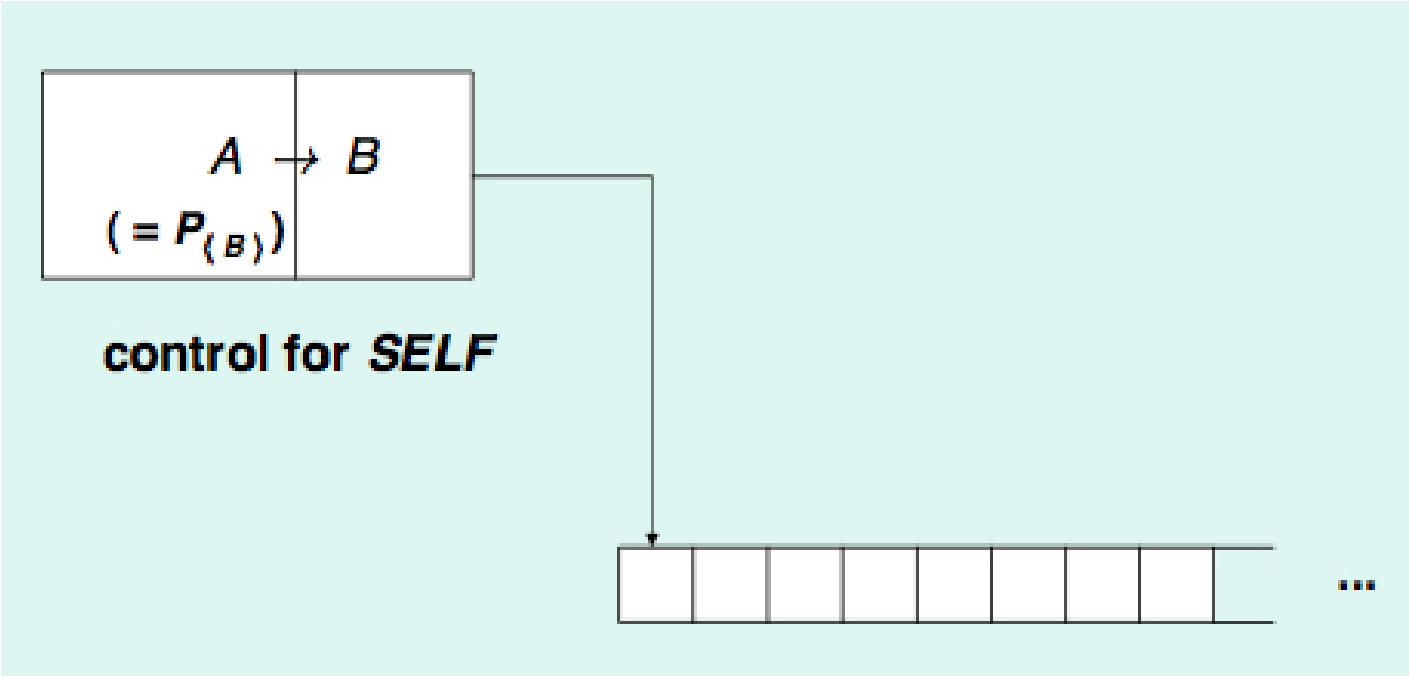
The following TM Q computes $q(w)$.
$Q =$ "On input string w:

1. Construct the following Turing machine P_w
   $P_w =$ "On any input:
   1. Erase input.
   2. Write w on tape.
   3. Halt."
2. Output $\langle P_w \rangle$."

The TM SELF

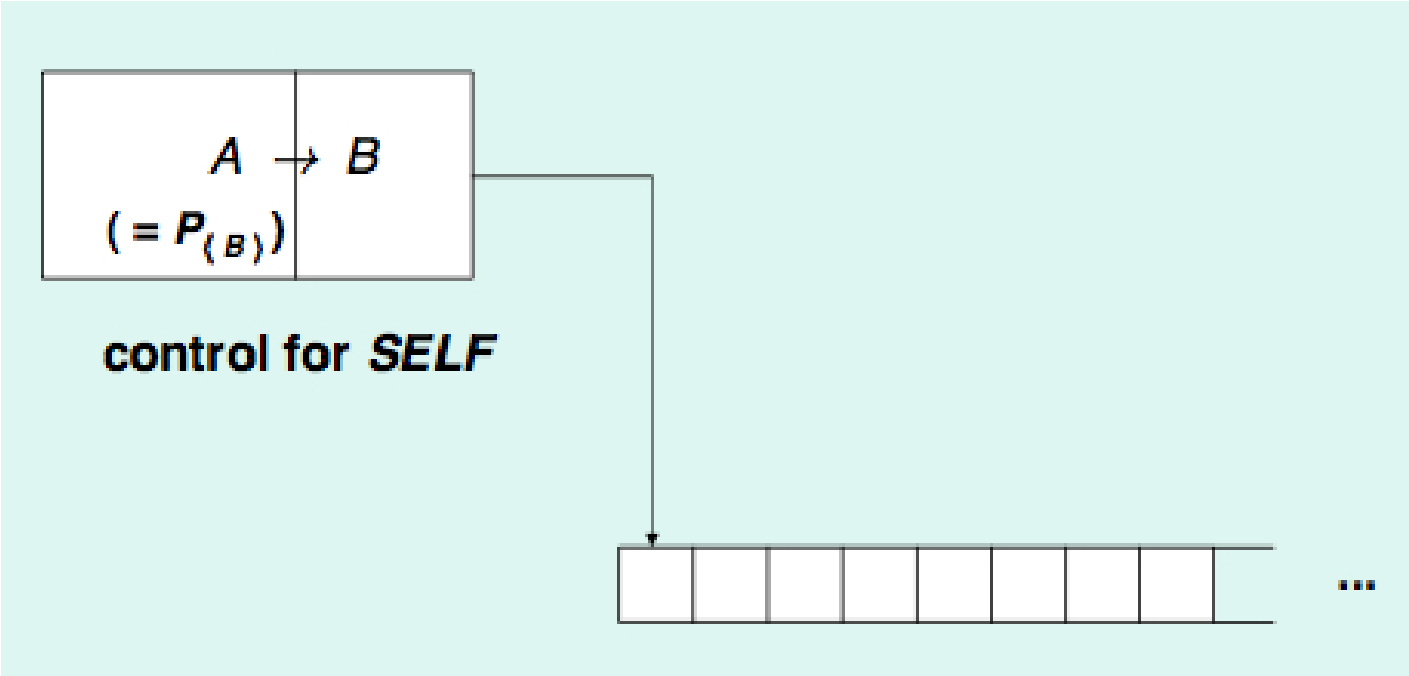
- Next we build a TM, *SELF*, that ignores its own input and prints out a copy of its description.
- Print out this sentence.
  - Not clear what "this" refers to.
- Print out two copies of the following, the second one in quotes: "Print out two copies of the following, the second one in quotes:"
- $\lambda x.x(x)(\lambda x.x(x)) \Rightarrow \lambda x.x(x)(\lambda x.x(x))$

A TM THAT PRINTS ITSELF



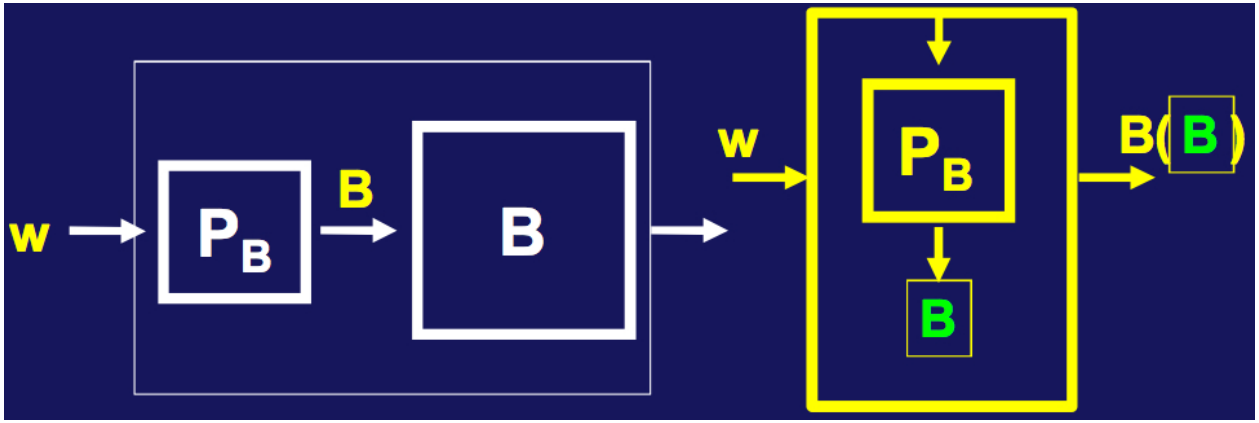
- Part A runs first and upon completion passes control to part B.
- The job of A is to print a description of B on the tape (hence $A = P_{\langle B \rangle}$).
- The job of B is (essentially) to print out a description of A.
- The tasks are similar, but are carried out differently.

A TM THAT PRINTS ITSELF



- If B can obtain $\langle B \rangle$, it can apply q to that and obtain $\langle A \rangle$.
- What how can B obtain $\langle B \rangle$?
- Well, it was printed on the tape, just before A passed control to B.
- So, B computes $q(\langle B \rangle) = \langle A \rangle$ and combines these and writes a complete description $\langle AB \rangle = \langle SELF \rangle$.

A TM THAT PRINTS ITSELF



- $A = P_{\langle B \rangle}$: A is the TM that prints out the description of B (But we do not have B yet!)
- $B =$ "On input $\langle M \rangle$ where M is a portion of a TM:
  1. Compute $q(\langle M \rangle)$, (find the description of the machine which prints $\langle M \rangle$)
  2. Combine the result with $\langle M \rangle$ to make a complete TM.
  3. Print the description of this TM and halt."

How SELF Behaves

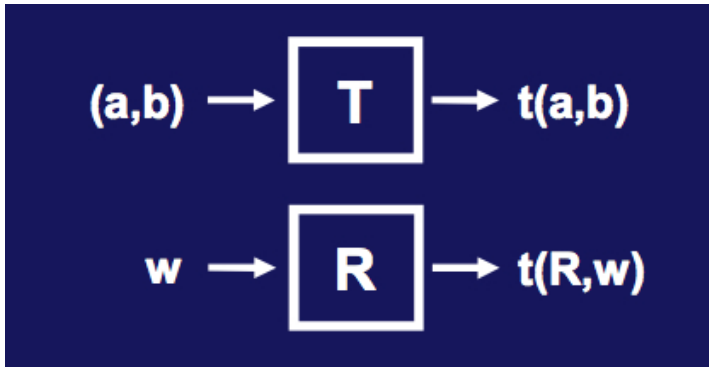
1. First A runs. It prints $\langle B \rangle$.
2. B starts. It looks at the tape and finds its input $\langle B \rangle$.
3. B computes $q(\langle B \rangle) = \langle A \rangle$ and combines that with $\langle B \rangle$ into a TM description $\langle SELF \rangle$.
4. B prints this description and halts.

THEOREM 6.3 – THE RECURSION THEOREM

Let T be a TM that computes a function $t : \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. There is a TM R that computes $r : \Sigma^* \longrightarrow \Sigma^*$, where for every w,

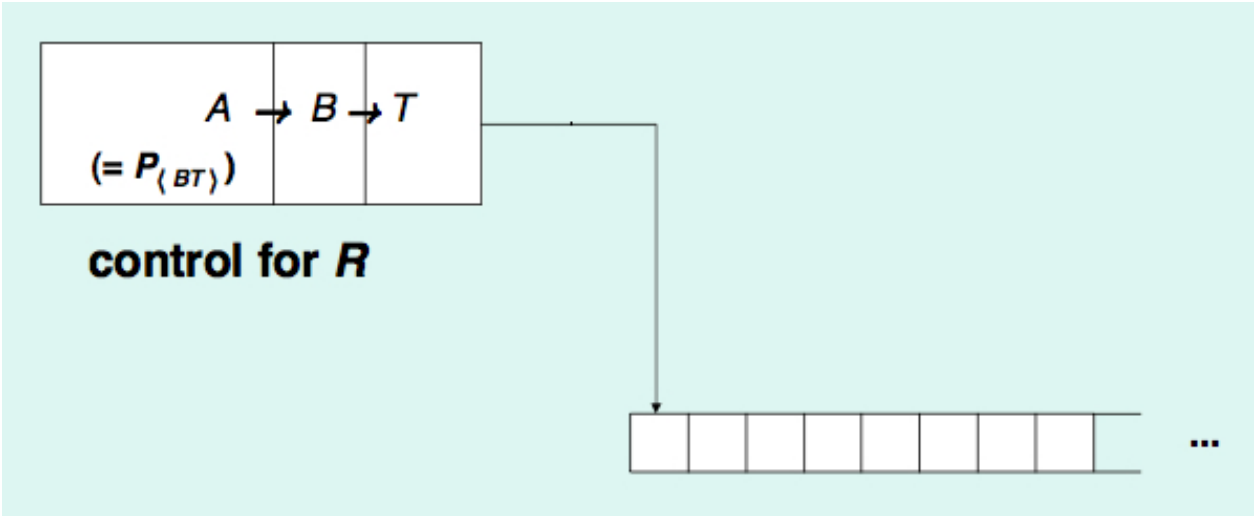
$$r(w) = t(\langle R \rangle, w)$$

- What is this Theorem saying?
- Informally, a TM can obtain its own description and compute with it.
- To make a TM, that can obtain its own description and then compute with it
  1. Make a TM T that receives the description of the machine as an extra input.
  2. Then the recursion theorem produces a new machine, R which operates as T does, with $R's$, description filled in automatically.



Proof of the Recursion Theorem

- We construct a machine with 3 parts: A, B and T.



- A is the TM $P_{\langle BT \rangle}$, described by $q(\langle BT \rangle)$
  - Technical point: We redesign q so that $P_{\langle BT \rangle}$ writes its output following any preexisting string on the tape.
- So, after A runs, the tape contains $w\langle BT \rangle$
- B examines the tape and applies q to $\langle BT \rangle$ getting $\langle A \rangle$.
- B then combines A, B and T into a single machine and obtains its description $\langle ABT \rangle = \langle R \rangle$
- It encodes these as $\langle R, w \rangle$ and places it on the tape and passes the control to T.

Significance of the Recursion Theorem

- It is yet another handy tool for solving certain problems in the theory of algorithms.
- When you are designing a TM M, you can "make a call" to "obtain own description $\langle M \rangle$" and use this description in the computation.
  - Just print out the description
  - Count the number of states in M.
  - Simulate M.
- Consider the TM

  $T =$ "On input $\langle M, w \rangle$:

  1. Print $\langle M \rangle$ and halt."

  The recursion theorem tells us how to construct R which on input w, behaves just like T on input $\langle R, w \rangle$.
- Thus R prints the description of R, exactly what is required of the machine *SELF*.
- Technology for Computer Viruses (:-)

Significance of the Recursion Theorem

Theorem 6.5

A_{TM} is undecidable.

Proof

- Suppose H decides A_{TM}, we construct B:
- $B =$ "On input w:
  1. Obtain, via the recursion theorem, own description $\langle B \rangle$.
  2. Run H on input $\langle B, w \rangle$.
  3. Do the opposite of what H says.
     - *accept* if H rejects.
     - *reject* if H accepts.
- B conflicts with itself – hence can not exist
- H can not exist.

The Fixed-point version of the Recursion Theorem

- A fixed-point of a function is a value, that is not changed by the application of a function, e.g.,
  - $f(x) = \sqrt{x}$ has a fixed-point 1.
  - $f(y(x)) = y'(x)$ has a fixed-point $y(x) = e^x$.
- We consider functions that are computable transformations of TM descriptions.
- The Fixed-point version of the Recursion Theorem shows that
  - whatever the transformation is
  - there is some TM whose behaviour is unchanged by the transformation!
- Informally, no computable function maps TMs into non-equivalent TMs

The Fixed-point version of the Recursion Theorem

Theorem 6.8

Let $t : \Sigma^* \longrightarrow \Sigma^*$. Then, there is a TM F such that $t(\langle F \rangle)$ describes a TM equivalent to F. (t is the transformation and F is the fixed point.)

Proof

- Let F be the following TM:
- $F =$ "On input w
  1. Obtain via the recursion theorem, own description $\langle F \rangle$.
  2. Compute $t(\langle F \rangle)$ to obtain the description of a TM G.
  3. Simulate G on w."
- It is clear that $\langle F \rangle$ and $\langle G \rangle$ describe equivalent TMs: they both compute what G computes with w.

Summary of Reducibility

We know that language A is undecidable. By reducing A to B we want to show that the language B is also undecidable.

1. Assume that we have a decider M_B for B.
2. Using M_B we construct a decider M_A for the language A:

$M_A =$ "On input $\langle I_A \rangle$

1. Algorithmically construct an input $\langle I_B \rangle$ for M_B, such that

   a) Either

   If $\langle I_A \rangle \in A$ then $\langle I_B \rangle \in B$
   If $\langle I_A \rangle \notin A$ then $\langle I_B \rangle \notin B$

   b) or

   If $\langle I_A \rangle \in A$ then $\langle I_B \rangle \notin B$
   If $\langle I_A \rangle \notin A$ then $\langle I_B \rangle \in B$

2. Run the decider M_B on $\langle I_B \rangle$ for M_B
   Case a): M_A accepts if M_B accepts, and rejects if M_B rejects
   Case b): M_A rejects if M_B accepts, and accepts if M_B reject.

3. We know M_A can not exist so M_B can not exist.

Computable Functions

Idea

Turing Machines can also compute function $f : \Sigma^* \longrightarrow \Sigma^*$.

Computable Function

A function $f : \Sigma^* \longrightarrow \Sigma^*$ is a computable function if and only if there exists a TM M_f, which on any given input $w \in \Sigma^*$

- always halts, and
- leaves just $f(w)$ on its tape.

Examples:

- Let $f(w) \stackrel{def}{=} ww$ be a function. Then f is computable.
- Let $f(\langle n_1, n_2 \rangle) \stackrel{def}{=} \langle n \rangle$ where n_1 and n_2 are integers and $n = n_1 * n_2$. Then f is computable.

Mapping Reducibility

Definition

Let $A, B \subseteq \Sigma^*$. We say that language A is mapping reducible to language B, written $A <_m B$, if and only if

1. There is a computable function $f : \Sigma^* \longrightarrow \Sigma^*$ such that
2. For every $w \in \Sigma^*$, $w \in A$ if and only if $f(w) \in B$.

The function f is called a reduction of A to B.

Theorem 5.22

If $A <_m B$ and B is decidable, then A is decidable.

Proof

Let M be a decider for B and f be a mapping from A to B. Then N decides A. $N =$ "On input w

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs."

Mapping Reducibility

Theorem

$A_{TM} <_m HALT_{TM}$

Proof.

Construct a computable function f which maps $\langle M, w \rangle$ to $\langle M', w' \rangle$ such that

$$\langle M, w \rangle \in A_{TM} \text{ if and only if } \langle M', w' \rangle \in HALT_{TM}$$

$M_f =$ "On input $\langle M, w \rangle$

1. Construct the following machine M':
   $M' =$ "On input x
   1. Run M on x.
   2. If M accepts *accept*
   3. If M rejects *enter a loop*."
2. Output $\langle M', w \rangle$."

□

More examples of Mapping Reducibility

- Earlier we showed
  - $A_{TM} <_m MPCP$
  - $MPCP <_m PCP$
- We showed $E_{TM} <_m EQ_{TM}$. The reduction f maps from $\langle M \rangle$ to the output $\langle M, M_1 \rangle$ where M_1 is the machine that rejects all inputs.

Theorem 5.24

If $A <_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

Proof

Essentially the same as the previous proof.

Summary of Mapping Reducibility Results

Summary of Theorems

Assume that $A <_m B$. Then

1. If B is decidable then A is decidable.
2. If A is undecidable then B is undecidable.
3. If B is Turing-recognizable then A is Turing-recognizable.
4. If A is not Turing-recognizable then B is not Turing-recognizable.
5. $\overline{A} <_m \overline{B}$

Useful observation:

- Suppose you can show $A_{TM} <_m \overline{B}$
- This means $\overline{A_{TM}} <_m B$
- Since $\overline{A_{TM}}$ is Turing-unrecognizable then B is Turing-unrecognizable.

Example of Use

Theorem 5.30

$EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ is neither Turing recognizable nor co-Turing-recognizable.

Proof Idea

We show

- $\overline{A_{TM}} <_m EQ_{TM}$
- $\overline{A_{TM}} <_m \overline{EQ_{TM}}$
- These then imply the theorem.

Example of Use

Proof for $\overline{A_{TM}} <_m EQ_{TM}$

We show $A_{TM} <_m \overline{EQ_{TM}}$ (hence $\overline{A_{TM}} <_m EQ_{TM}$) with the following f:
$F =$ "On input $\langle M, w \rangle$ where M is a TM and w is a string:

1. Construct the following two machines M_1 and M_2
   $M_1 =$ "On any input:
   1. Reject"
   $M_2 =$ "On any input:
   1. Run M on w. If it accepts, *accept.*"
2. Output $\langle M_1, M_2 \rangle$."

- M_1 accepts nothing.
  - If M accepts w then M_2 accepts everything. So M_1 and M_2 are not equivalent.
  - If M does not accept w then M_2 accepts nothing. So M_1 and M_2 are equivalent.
- So $A_{TM} <_m \overline{EQ_{TM}}$ (and hence $\overline{A_{TM}} <_m EQ_{TM}$)

Example of Use

Proof for $\overline{A_{TM}} <_m \overline{EQ_{TM}}$

We show $A_{TM} <_m EQ_{TM}$ (hence $\overline{A_{TM}} <_m \overline{EQ_{TM}}$) with the following g:
$G =$ "On input $\langle M, w \rangle$ where M is a TM and w is a string:

1. Construct the following two machines M_1 and M_2
   $M_1 =$ "On any input:
   1. Accept"
   $M_2 =$ "On any input:
   1. Run M on w. If it accepts, *accept.*"
2. Output $\langle M_1, M_2 \rangle$."

- M_1 accepts everything.
  - If M accepts w then M_2 accepts everything. So M_1 and M_2 are equivalent.
  - If M does not accept w then M_2 accepts nothing. So M_1 and M_2 are not equivalent.
- So $A_{TM} <_m EQ_{TM}$ (hence $\overline{A_{TM}} <_m \overline{EQ_{TM}}$)

Turing Reducibility

- Reducibility: If A is reducible to B then we can solve A by solving B.
- Mapping Reducibility ($A \leq_m B$) : Use a computable mapping f to transform an instance of A to an instance of B.
- It turns out that Mapping Reducibility is not general enough!
  - Consider A_{TM} and $\overline{A_{TM}}$
  - Clearly the solution to one can be used as a solution to the other, by simply reversing the answer.
  - But $\overline{A_{TM}}$ is not mapping reducible to A_{TM} because A_{TM} is Turing-recognizable while $\overline{A_{TM}}$ is not.
- We need a more general notion of reducibility.

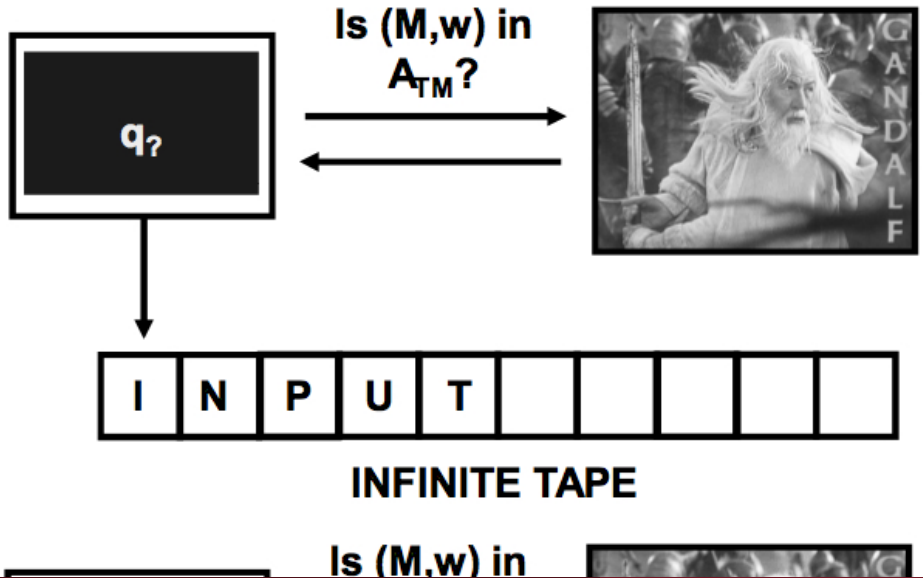
Oracles

Definition – Oracle

An oracle for a language B is an external device that is capable of answering the question "Is $w \in B$?"

Definition – Oracle Turing Machine

An oracle TM is a modified TM, M^B, that has the capability of querying an oracle for language B.



Turing Reducibility

Definition

Language A is Turing reducible to language B, written as $A \leq_T B$, if A is decidable relative to B (that is, using an oracle for B)

Theorem

If $A \leq_T B$ and B is decidable, then A is decidable.

Proof

If B is decidable, then replace the oracle with the TM for B.

- Turing reducibility is a generalization of mapping reducibility $A \leq_M B$

Formal Languages, Automata and Computation

Complexity

Complexity Theory

Question

Assume that a problem (language) is decidable. Does that mean we can realistically solve it?

Answer

NO, not always. It can require too much of time or memory resources.

Complexity Theory aims to make general conclusions of the resource requirements of decidable problems (languages).

- Henceforth, we only consider decidable languages and deciders.
- Our computational model is a Turing Machine.
  - Time: the number of computation steps a TM machine makes to decide on an input of size n.
  - Space: the maximum number of tape cells a TM machine takes to decide on a input of size n.

Time Complexity – Motivation

- How much time (or how many steps) does a single tape TM take to decide $A = \{0^k 1^k \mid k \geq 0\}$?

M = "On input w:

1. Scan the tape and *reject* if w is not of the form 0^*1^* .
2. Repeat if both 0s and 1s remain on the tape.
3. Scan across the tape crossing off one 0 and one 1.
4. If all 0's are crossed and some 1's left, or all 1's crossed and some 0's left, then *reject*; else *accept*.

Question

How many steps does M take on an input w of length n?

Answer (Worst-case)

The number of steps M takes $\propto n^2$.

Time Complexity – Some notions

- The number of steps in measured as a function of n - **the size of the string representing the input.**
- In worst-case analysis, we consider the longest running time of all inputs of length n.
- In average-case analysis, we consider the average of the running times of all inputs of length n.

Time Complexity

Let M be a deterministic TM that halts on all inputs. The time complexity of M if the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.
If $f(n)$ is the running time of M we say

- M runs in time $f(n)$
- M is an $f(n)$-time TM.

Asymptotic Analysis

- We seek to understand the running time when the input is "large".
- Hence we use an asymptotic notation or big-O notation to characterize the behaviour of $f(n)$ when n is large.
- The exact value running time function is not terribly important.
- What is important is how $f(n)$ grows as a function of n, for large n.
- Differences of a constant factor are not important.

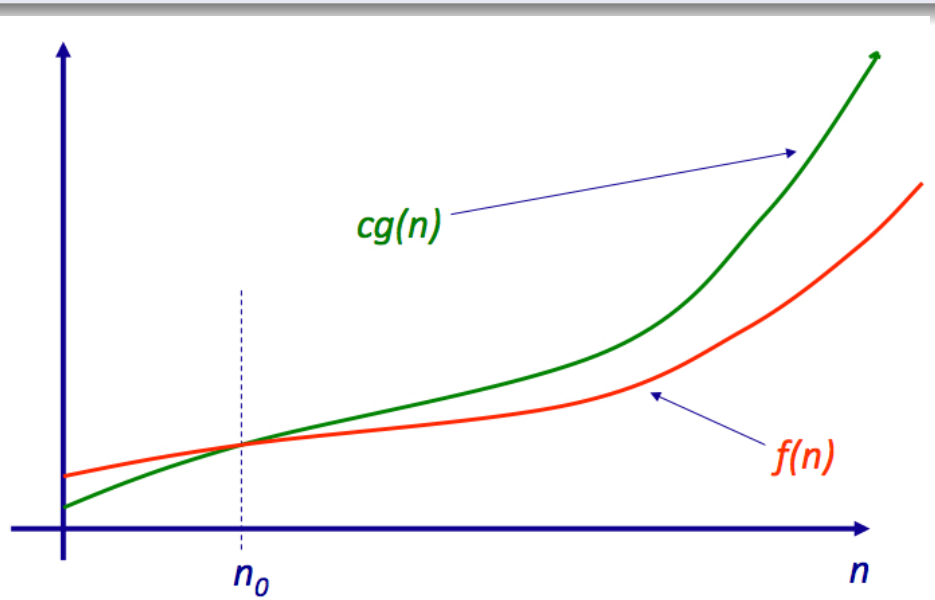
Asymptotic Upper Bound

Definition – Asymptotic Upper bound

Let $\mathcal{R}^+$ be the set of nonnegative real numbers. Let f and g be functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$. We say $f(n) = O(g(n))$, if there are positive integers c and n_0, such that for every $n \geq n_0$

$$f(n) \leq c\, g(n).$$

$g(n)$ is an asymptotic upper bound.



Asymptotic Upper Bound

- $5n^3 + 2n^2 + 5 = O(n^3)$ (what are c and n_0?)
- $5n^3 + 2n^2 + 5 = O(n^4)$ (what are c and n_0?)
- $\log_2(n^8) = O(\log n)$ (why?)
- $5n^3 + 2n^2 + 5$ is not $O(n^2)$ (why?)

- $2^{O(n)}$ means an upper bound $O(2^{cn})$ for some constant c.
- $n^{O(1)}$ is a polynomial upper bound $O(n^c)$ for some constant c.

Reality Check

Assume that your computer/TM can perform 10^9 steps per second.

$n/f(n)$	n	$n\log(n)$	n^2	n^3	2^n
10	0.01 μsec	0.03 μsec	0.1 μsec	1 μsec	1 μsec
20	0.02 μsec	0.09 μsec	0.4 μsec	8 μsec	1 *msec*
50	0.05 μsec	0.28 μsec	2.5 μsec	125 μsec	13 days
100	0.10 μsec	0.66 μsec	10 μsec	1 *msec*	$\approxeq 4 \times 10^{13}$ years
1000	1 μsec	3 μsec	1 *msec*	1 sec	$\approxeq 3.4x10^{281}$ centuries

Clearly, if the running time of your TM is an exponential function of n, it does not matter how fast the TM is!

Small-o Notation

Definition – Strict Asymptotic Upper Bound

Let f and g be functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$. We say $f(n) = o(g(n))$, if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

- $n^2 = o(n^3)$
- $\sqrt{n} = o(n)$
- $n \log n = o(n^2)$
- $n^{100} = o(2^n)$
- $f(n)$ is never $o(f(n))$.

Intuition

- $f(n) = O(g(n))$ means "asymptotically $f(n) \leq g(n)$"
- $f(n) = o(g(n))$ means "asymptotically $f(n) < g(n)$"

DEFINITION – TIME COMPLEXITY CLASS TIME($t(n)$)

Let $t : \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function.
TIME($t(n)$) $= \{L(M) \mid M$ is a decider running in time $O(t(n))\}$

- TIME($t(n)$) is the class (collection) of languages that are decidable by TMs, running in time $O(t(n))$.
- TIME(n) $\subset$ TIME(n^2) $\subset$ TIME(n^3) $\subset \ldots \subset$ TIME(2^n) $\subset \ldots$
- Examples:
  - $\{0^k1^k \mid k \geq 0\} \in$ TIME(n^2)
  - $\{0^k1^k \mid k \geq 0\} \in$ TIME($n \log n$) (next slide)
  - $\{w\#w \mid w \in \{0,1\}^*\} \in$ TIME(n^2)

$\{0^k1^k \mid k \geq 0\} \in \text{TIME}(n \log n)$

M = "On input w:

1. Scan the tape and *reject* if w is not of the form 0^*1^* .
2. Repeat as long as some 0s and some 1s remain on the tape.
   - Scan across the tape, checking whether the total number of 0s and 1s is even or odd. *Reject* if it is odd.
   - Scan across the tape, crossing off every other 0 starting with the first 0, and every other 1, starting with the first 1.
3. If no 0's and no 1's remain on the tape, *accept*. Otherwise, *reject*.

- Steps 2 take $O(n)$ time.
- Step 2 is repeated at most $1 + \log_2 n$ times. (why?)
- Total time is $O(n \log n)$.
- Hence, $\{0^k1^k \mid k \geq 0\} \in \text{TIME}(n \log n)$.
- However, $\{0^k1^k \mid k \geq 0\}$ is decidable on a 2-tape TM in time $O(n)$ (How ?)

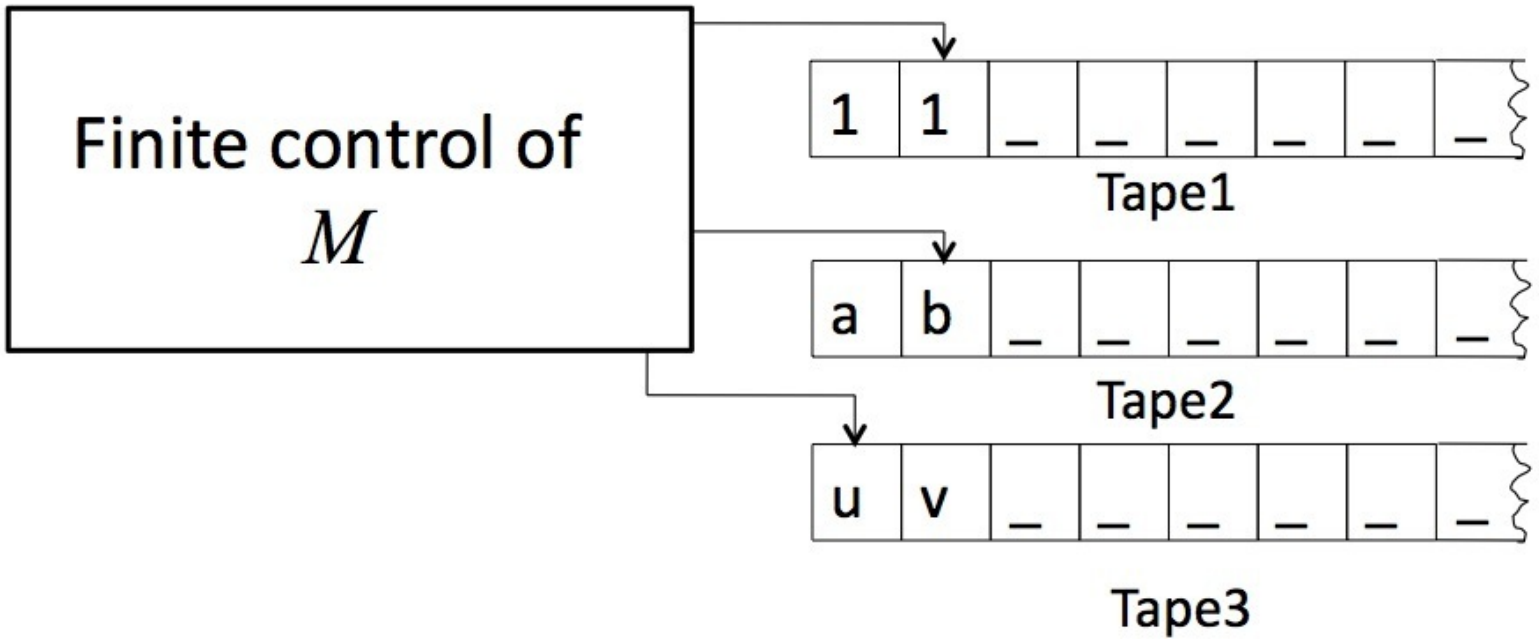
Relationship between k-tape and Single-tape TMs

Theorem 7.8

Let $t(n)$ be a function and $t(n) \geq n$. Then every multitape TM has an equivalent $O(t^2(n))$ single tape TM.

- Let's remind ourselves on how the simulation operates.

Multitape Turing Machines



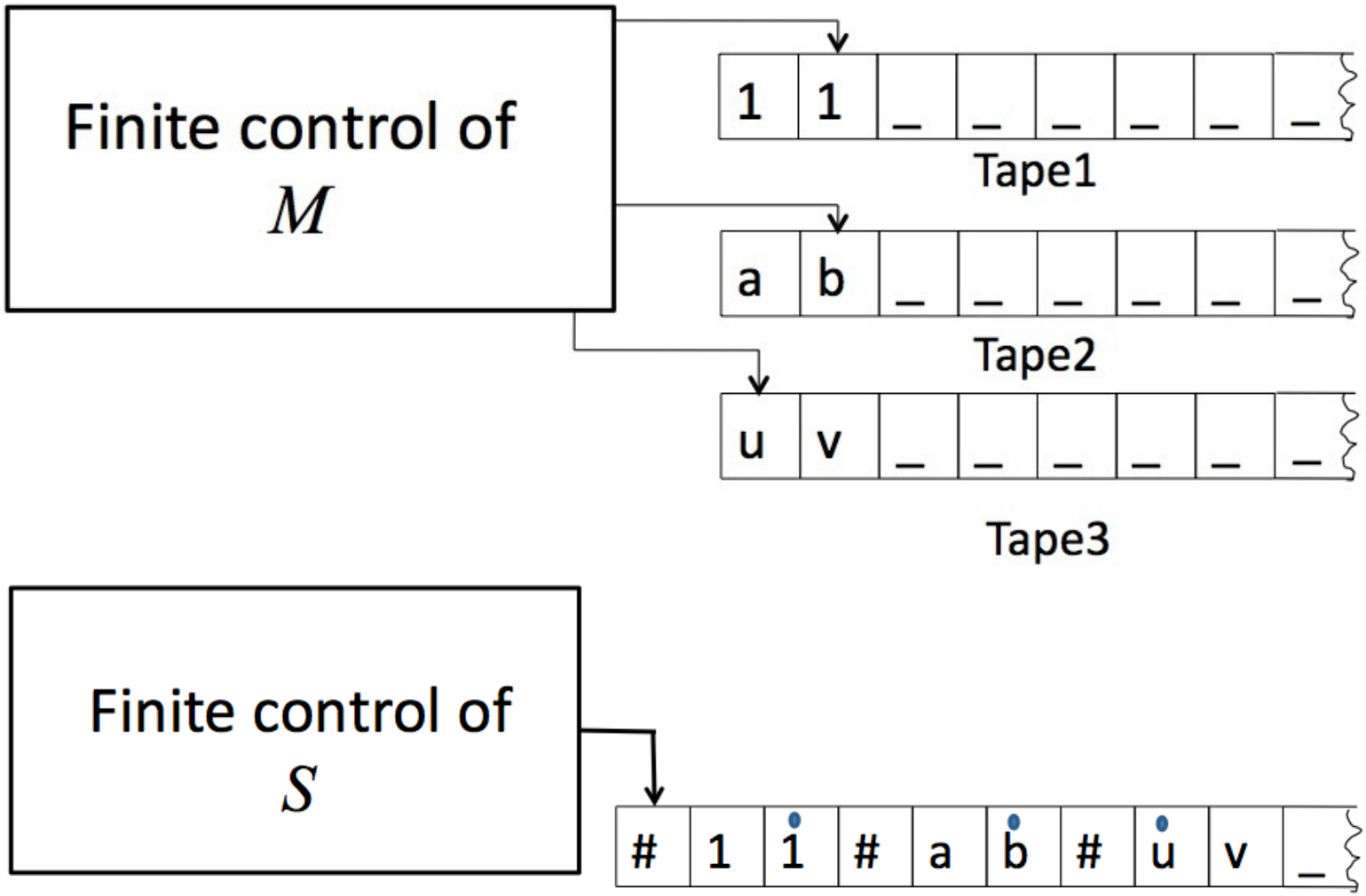
Multitape Turing Machines

- A multitape Turing Machine is like an ordinary TM
  - There are k tapes
  - Each tape has its own independent read/write head.
- The only fundamental difference from the ordinary TM is δ – the state transition function.

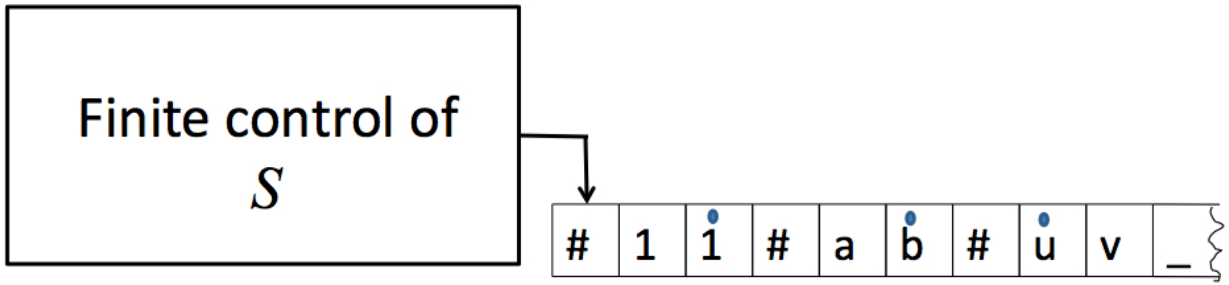
$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

- The δ entry $\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, L, \ldots L)$ reads as :
  - If the TM is in state q_i and
  - the heads are reading symbols a_1 through a_k,
  - Then the machine goes to state q_j, and
  - the heads write symbols b_1 through b_k, and
  - Move in the specified directions.

Simulating a Multitape TM with an Ordinary TM



Simulating a Multitape TM with an Ordinary TM



- We use # as a delimiter to separate out the different tape contents.
- To keep track of the location of heads, we use additional symbols
  - Each symbol in Γ (except $\sqcup$) has a "dotted" version.
  - A dotted symbol indicates that the head is on that symbol.
  - Between any two #'s there is only one symbol that is dotted.
- Thus we have 1 real tape with k "virtual' tapes, and
- 1 real read/write head with k "virtual" heads.

Simulating a Multitape TM with an Ordinary TM

- Given input $w = w_1 \cdots w_n$, S puts its tape into the format that represents all k tapes of M

$$\# \dot{w}_1 \; w_2 \cdots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$$

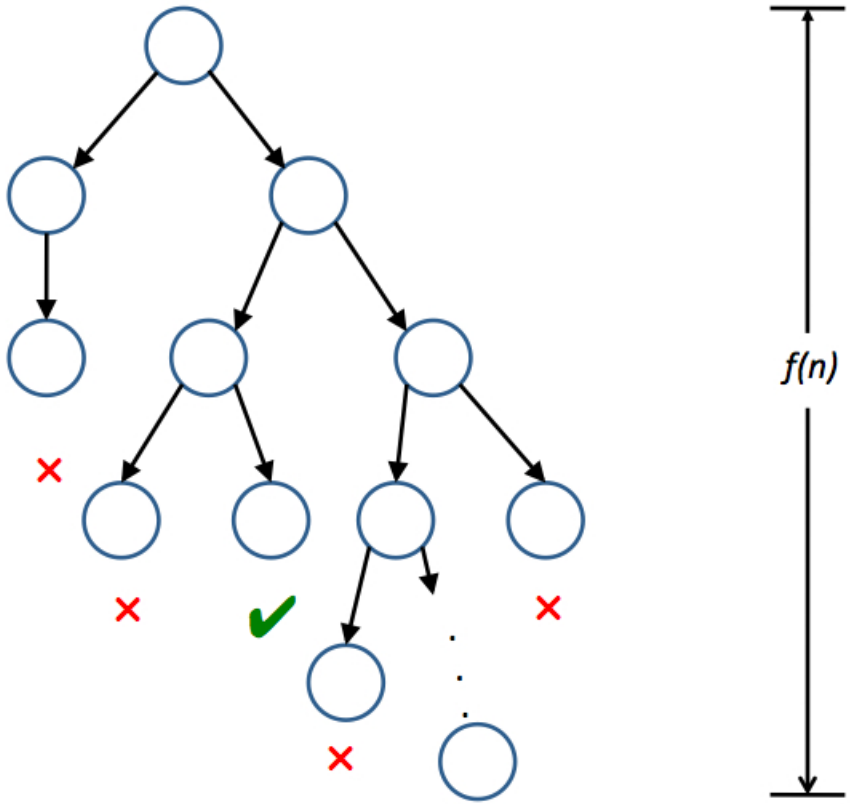
- To simulate a single move of M, S starts at the leftmost $\#$ and scans the tape to the rightmost $\#$.
  - It determines the symbols under the "virtual" heads.
  - This is remembered in the finite state control of S. (How many states are needed?)
- S makes a second pass to update the tapes according to M.
- If one of the virtual heads, moves right to a $\#$, the rest of tape to the right is shifted to "open up" space for that "virtual tape". If it moves left to a $\#$, it just moves right again.

Analysis of the Multi-tape TM Simulation

- Preparing the single simulation tape takes $O(n)$ time.
- Each step of the simulation makes two passes over the tape:
  - One pass to see where the heads are.
  - One pass to update the heads (possibly with some shifting)
- Each pass takes at most $k \times t(n) = O(t(n))$ steps (why?)
- So each simulation step takes 2 scans + at most k rightward shifts. So the total time per step is $O(t(n))$.
- Simulation takes $O(n) + t(n) \times O(t(n))$ steps $= O(t^2(n))$.
- So, a single-tape TM is only polynomially slower than the multi-tape TM.
- If the multi-tape TM runs in polynomial time, the single-tape TM will also run in polynomial time (where polynomial time is defined as $O(n^m)$ for some m.)

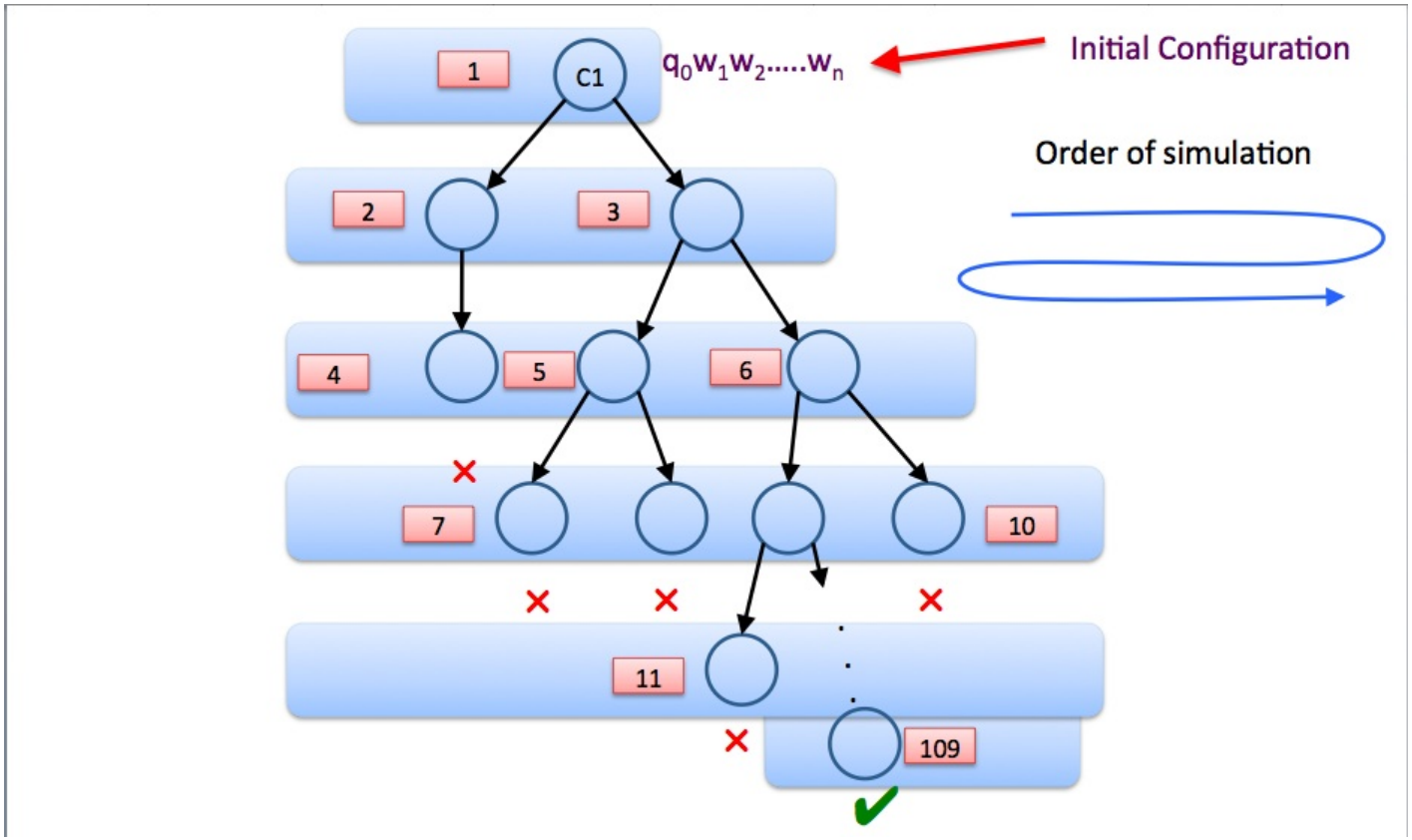
DEFINITION – NONDETERMINISTIC RUNNING TIME

Let N be a nondeterministic TM that is a decider. The running time of N is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses, on any branch of its computation on any input of length n.



THEOREM 7.11

Let $t(n)$ be a function and $t(n) \geq n$. Then every $t(n)$ time nondeterministic TM has an equivalent $2^{O(t(n))}$ time deterministic single tape TM.



Nondeterministic TMs

Theorem 7.11

Let $t(n)$ be a function and $t(n) \geq n$. Then every $t(n)$ time nondeterministic TM has an equivalent $2^{O(t(n))}$ time deterministic single tape TM.

Proof

- On an input of n, every branch of N's nondeterministic computation has length at most $t(n)$ (why?)
- Every node in the tree can have at most b children where b is the maximum number of nondeterministic choices a state can have.
- So, the computation tree has at most $1 + b^2 + \cdots + b^{t(n)} = O(b^{t(n)})$ nodes.
- The deterministic machine D takes at most $O(b^{t(n)}) = 2^{O(t(n))}$ steps.
- D has 3 tapes. Converting it to a single tape TM at most squares its running time (previous Theorem):$(2^{O(t(n))})^2 = 2^{2O(t(n))} = 2^{O(t(n))}$

The Class P

Definition

P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM.

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

- The class P is important for two main reasons:
  1. P is robust: The class remains invariant for all models of computation that are polynomially equivalent to deterministic single-tape TMs.
  2. P (roughly) corresponds to the class of problems that are realistically solvable on a computer.
- Even though the exponents can be large (though most useful algorithms have "low" exponents), the class P provides a reasonable definition of practical solvability.

Examples of Problems in P

- We will give high-level algorithms with numbered stages just as we gave for decidability arguments.
- We analyze such algorithms to show that they run in polynomial time.
  1. We give a polynomial upper bound on the number of stages the algorithm uses when it runs on an input of length n.
  2. We examine each stage, to make sure that each can be implemented in polynomial time on a reasonable deterministic time.
- We assume a "reasonable" encoding of the input.
  - For example, when we represent a graph G, we assume that $\langle G \rangle$ has a size that is poynomial the number of nodes.

Examples of Problems in P

Theorem

$PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with n nodes that has a path from s to $t\} \in$ P.

Proof

$M =$ "On input $\langle G, s, t\rangle$

1. Place a mark on s.
2. Repeat 3 until no new nodes are marked
3. Scan edges of G. If (a, b) is an edge and a is marked and b is unmarked, mark b.
4. If t is marked, *accept* else *reject*."

- Steps 1 and 4 are executed once
  - Each takes at most $O(n)$ time on a TM.
- Step 3 is executed at most n times
  - Each execution takes at most $O(n^2)$ steps ($\propto$ number of edges)
- Total execution time is thus a polynomial in n.

Examples of Problems in P

Theorem

$A_{CFG} \in \mathsf{P}$

Proof.

The CYK algorithm decides A_{CFG} in polynomial time. □

Examples of Problems in P

Definition

Natural numbers x and y are relatively prime iff $gcd(x, y) = 1$.

- $gcd(x, y)$ is the greatest natural number that evenly divides both x and y.
- $RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime numbers}\}$
- Remember that the length of $\langle x, y\rangle$ is $\log_2 x + \log_2 y = n$, that is the size of the input is logarithmic in the values of the numbers.
  - So if the number of steps is proportional to the values of x and y, it is exponential in n.

Brute force algorithm is exponential

Given an input $\langle x, y\rangle$ of length $n = \log_2 x + \log_2 y$, going through all numbers between 2 and $\min\{x, y\}$, and checking if they divide both x and y takes time exponential in n.

Examples of Problems in P

Theorem 7.15

RELPRIME $\in$ P

Proof

E implements the Euclidian algorithm.

$E =$' "On input $\langle x, y \rangle$

1. Repeat until $y = 0$
2. Assign $x \leftarrow x \mod y$.
3. Exchange x and y.
4. Output x."

Proof

R solves *RELPRIME*, using E as a subroutine.

$R =$ "On input $\langle x, y \rangle$

1. Run E on $\langle x, y \rangle$.
2. If the result is 1, *accept.* Otherwise, *reject*."

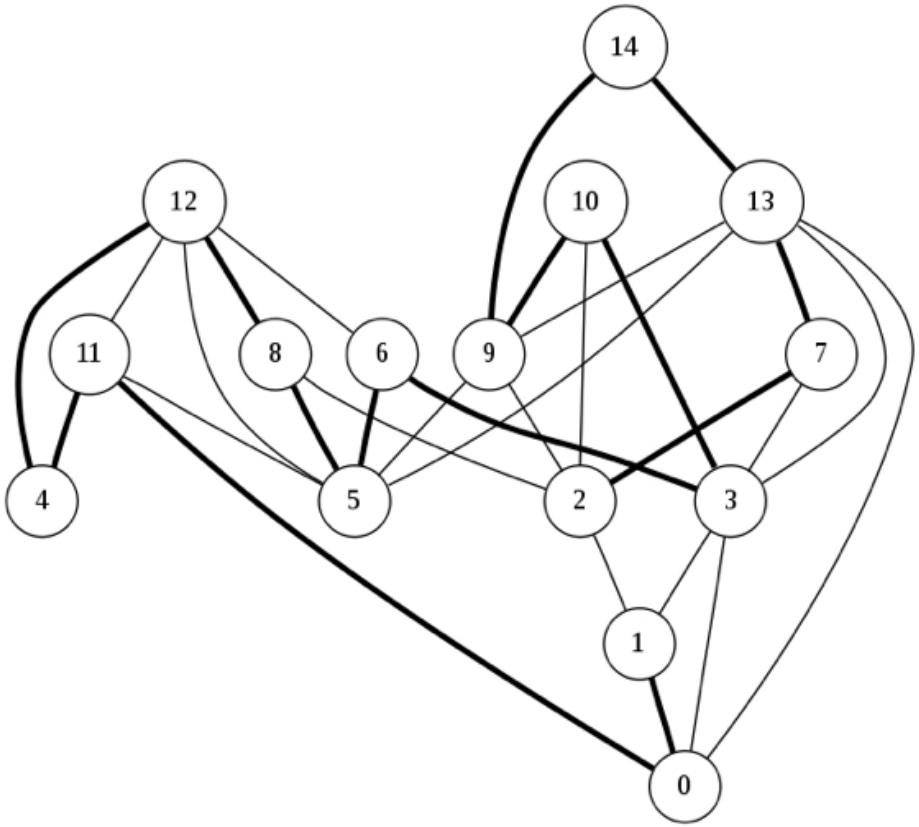
- If $E \in$ P then $R \in$ P.
- Each of x and y is reduced by a factor of 2 every other time through the loop.
- Loop is executed at most $\min\{2 \log_2 x, 2 \log_2 y\}$ times which is $O(n)$.

The Class NP

- For some problems, even though there is a exponentially large search space of solutions (e.g., for the path problem), we can avoid a brute force solution and get a polynomial-time algorithm.
- For some problems, it is not possible to avoid a brute force solution and such problems have so far resisted a polynomial time solution.
- We may not yet know the principles that would lead to a polynomial time algorithm, or they may be "intrinsically difficult."
- How can we characterize such problems?

DEFINITION – HAMILTONIAN PATH

A Hamiltonian path in a directed graph G is a directed path that goes through each node exactly once.



The Hamiltonian Path Problem

Hamiltonian Path Problem

$HAMPATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$.

- We can easily obtain an exponential time algorithm with a brute force approach.
  - Generate all possible paths between s and t and check if all nodes appear on a path!
- The *HAMPATH* problem has a property called polynomial verifiability.
  - If we can (magically) get a Hamiltonian path, we can verify that it is a Hamiltonian path, in polynomial time.
- *Verifying* the existence of a Hamiltonian path is "easier" than *determining* its existence.

Polynomial Verifiability

Composites Problem

$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\}$

- We can easily verify if a number is composite, given a divisor of that number.
- A recent (but very complicated) algorithm for testing whether a number is prime or composite has been discovered.

$\overline{HAMPATH}$ Problem

The $\overline{HAMPATH}$ problem has a solution if there is NO Hamiltonian path between s and t.

- Even if we knew, the graph did not have a Hamiltonian path, there is no easy way to verify this fact. We may need to take exponential time to verify it.

Verifiers

Verifier

A verifier for a language A is an algorithm V where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

- We measure the time of a verifier only in terms of the length of w.
- A language A is polynomially verifiable if it has a polynomial time verifier.
- c is called certificate or proof of membership in A.
  - For the *HAMPATH* problem, the certificate is simply the Hamiltonian path from s to t.
  - For the *COMPOSITES* problem, the certificate is one of the divisors.

THE CLASS NP

NP is the class of languages that have polynomial time verifiers.

- NP stands for nondeterministic polynomial time.
- Problems in NP are called NP-Problems.
- P $\subset$ ($\subseteq$?) NP.

A Nondeterministic decider for *HAMPATH*

N_1 = "On input $\langle G, s, t \rangle$

1. Nondeterministically select list of m numbers $p_1, p_2, \ldots p_m$ with $1 \leq p_i \leq m$.
2. Check for repetitions in the list; if found, *reject*.
3. Check whether $p_1 = s$ and $p_m = t$, otherwise *reject*.
4. For $1 \leq i < m$, check if (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise *accept*."

- Stage 1 runs in polynomial time.
- Stages 2 and 3 take polynomial time.
- Stage 4 takes poynomial time.
- Thus the algorithm runs in nondeterministic polynomial time.

The Class NP

Theorem 7.20

A language is in NP, iff it is decided by some nondeterministic polynomial time Turing machine.

Proof Idea

- We show polynomial time verifier $\Leftrightarrow$ polynomial time decider TM.
  - NTM simulates the verifier by guessing a certificate.
  - The verifier simulates the NTM

Proof: NTM given the verifier.

Let $A \in$ NP. Let V be a verifier that runs in time $O(n^k)$. N decides A in nondeterministic polynomial time.
$N =$ "On input w of length n

1. Nondeterministically select string c of length at most n^k.
2. Run V on input $\langle w, c \rangle$.
3. If V accepts, *accept*; otherwise *reject*."

The Class NP

Theorem 7.20

A language is in NP, iff it is decided by some nondeterministic polynomial time Turing machine.

Proof Idea

- We show polynomial time verifier $\Leftrightarrow$ polynomial time decider TM.
  - NTM simulates the verifier by guessing a certificate.
  - The verifier simulates the NTM

Proof: Verifier given the NTM.

Assume A is decided by a polynomial time NTM N. We construct the following verifier V

$V =$ "On input $\langle w, c \rangle$

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice at each step.
2. If this branch of N's computation accepts, *accept*; otherwise, *reject*."

The class NP

Definition

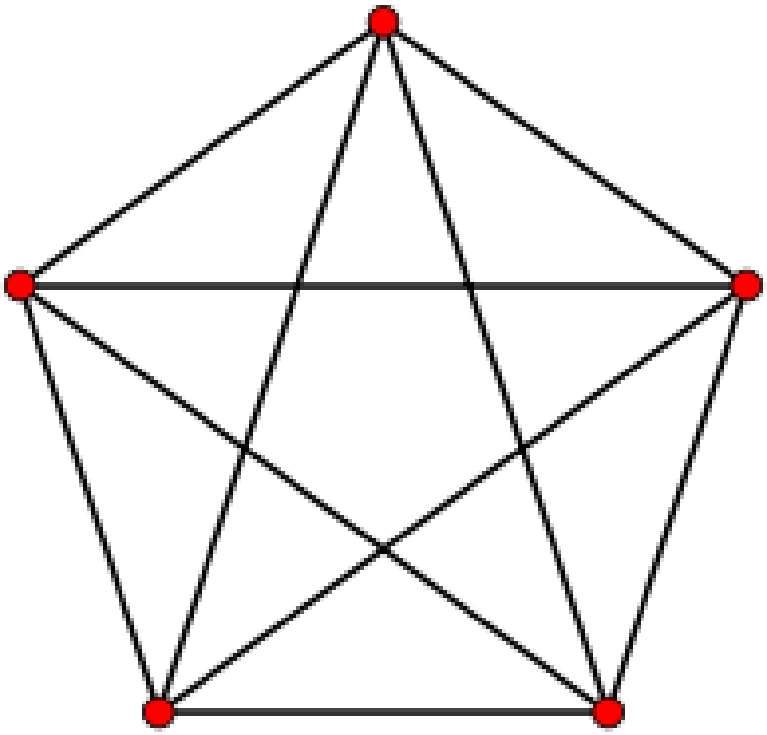
NTIME($t(n)$) $= \{L \mid L$ is a language decided by a $O(t(n))$ time nondeterministic TM.$\}$

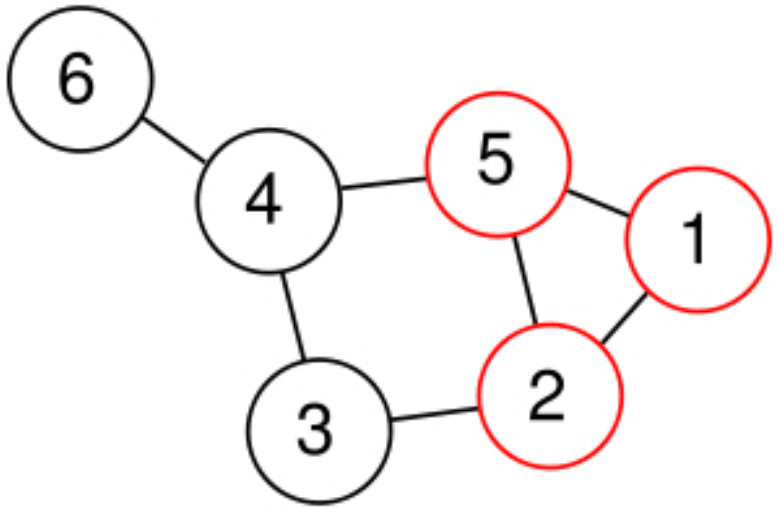
Corollary

NP $= \bigcup_k$ NTIME(n^k).

DEFINITION - CLIQUE

A clique in an undirected graph is a subgraph, wherein every two nodes are connected by an edge.
A *k*-clique is a clique that contains *k* nodes.





The Clique Problem

Theorem 7.24

$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique } \} \in$ NP.

Proof

The clique is the certificate.

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G.
2. Test whether G has all edges connecting nodes in c.
3. If both pass, *accept*; otherwise *reject*."

- All steps take polynomial time.

Alternative Proof

Use a NTM as a decider.

$N =$ "On input $\langle G, k\rangle$:

1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G has all edges connecting nodes in c.
3. If yes *accept*; otherwise *reject*."

The Subset-sum Problem

Theorem 7.25

$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$ and for some $\{y_1, \ldots, y_l\} \subseteq S, \sum y_i = t\} \in$ NP.

Proof

The clique is the certificate.

$V =$ "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a set of numbers summing to t.
2. Test whether S contains all numbers in c.
3. If both pass, *accept*; otherwise *reject*."

- All steps take polynomial time.

Alternative Proof

Use a NTM as a decider.

$N =$ "On input $\langle S, k\rangle$:

1. Nondeterministically select a subset c of numbers in S.
2. Test whether S contains all numbers in c.
3. If yes *accept*; otherwise *reject*."

The class coNP

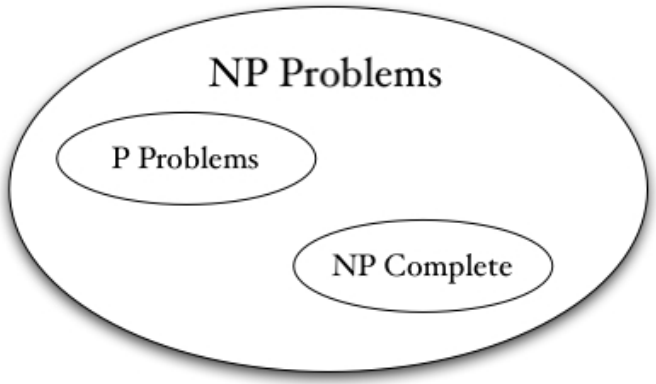
- It turns out $\overline{\textit{CLIQUE}}$ or $\overline{\textit{SUBSET-SUM}}$ are NOT in NP.
- Verifying something is NOT present seems to be more difficult than verifying it IS present.
- The class coNP contains all problems that are complements of languages in NP.
- We do not know if coNP $\neq$ NP.

Formal Languages, Automata and Computation

NP-Completeness

Summary

- Time complexity: Big-O notation, asympotic complexity
- Simulation of multi-tape TMs with a single tape deterministic TM can be done with a polynomial slow-down.
- Simulation of nondeterministic TMs with a deterministic TM is exponentially slower.
- The Class P: The class of languages for which membership can be *decided* quickly.
- The Class NP: The class of languages for which membership can be *verified* quickly.



- We do not yet know if P $=$ NP, or not.

NP Problems

- The best method known for solving languages in NP deterministically uses exponential time, that is

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k})$$

- It is not known whether NP is contained in a smaller deterministic time complexity class.

NP-Complete Problems

- Cook and Levin in early 1970's showed that certain problems in NP were such that
  - If any of these problems had a deterministic polynomial-time algorithm, then
  - All problems in NP had deterministic polynomial-time algorithms.
- Such problems are called NP-complete problems.
- This is important for a number of reasons:
  1. If one is attempting to show that P$\neq$NP, s/he may focus on an NP-complete problem and try to show that it needs more than a polynomial amount of time.
  2. If one is attempting to show that P=NP, s/he may focus on an NP-complete problem and try to come up with a polynomial time algorithm for it.
  3. One may avoid wasting searching for a nonexistent polynomial time algorithm to solve a particular problem, if one can show it reduces to an NP-complete problem (as it is generally believed that P$\neq$ NP.)

DEFINITION – BOOLEAN VARIABLES

A boolean variable is a variable that can taken on values TRUE (1) and FALSE (0).

- We have Boolean operations of AND ($x \wedge y$), OR ($x \vee y$) and NOT ($\neg x$ or $\overline{x}$) on boolean variables.

AND	OR	NOT
$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$\overline{0} = 1$
$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$\overline{1} = 0$
$1 \wedge 0 = 0$	$1 \vee 0 = 1$	
$1 \wedge 1 = 1$	$1 \vee 1 = 1$	

The Satisfiability Problem

Definition – Boolean Formula

A Boolean formula is an expression involving Boolean variables and operations.
For example: $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z}) \vee (y \wedge z)$ is a Boolean formula.

Definition – Satisfiability

A Boolean formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.
We say the assignment satisfies ϕ.

- What possible assignments satisfy the formula above?

Definition – the Satisfiability Problem

The satisfiability problem checks if a Boolean formula is satisfiable.

$$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

The Satisfiability Problem

Theorem 7.27 – The Cook-Levin Theorem

$SAT \in \mathsf{P}$ iff $\mathsf{P} = \mathsf{NP}$.

Proof

Coming slowly!

Polynomial Time Reducibility

Definition – Polynomial Time Computable Function

A function $f : \Sigma^* \longrightarrow \Sigma^*$ is a polynomial time computable function if some polynomial time TM M exists that halts with $f(w)$ on its tape, when started on any input w.

Definition – Polynomial Time Reducibility

Language A is polynomial time mapping reducible or polynomial time reducible, to language B, notated $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \Leftrightarrow f(w) \in B$$

The function f is called the polynomial time reduction of A to B.

- To test whether $w \in A$ we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.

Polynomial Time Reducibility

Theorem 7.31

If $A \leq_P B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.

Proof

- It takes polynomial time to reduce A to B.
- It takes polynomial time to decide B.

Variations on the Satisfiability Problem

- A literal is a Boolean variable or its negated version (x or $\overline{x}$).
- A clause is several literals connected with $\vee$ (OR), e.g., $(x_1 \vee \overline{x_2} \vee x_4)$.
- A Boolean formula is in conjuctive normal form (or is a cnf-formula) if it consists of several clauses connected with $\wedge$(AND), e.g.

$$(x_1 \vee \overline{x_2} \vee x_4 \vee x_5) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3 \vee \overline{x_5})$$

- A cnf-formula is a 3cnf-formula if all clauses have 3 literals, e.g.

$$(x_1 \vee \overline{x_2} \vee x_4) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_3 \vee \overline{x_5})$$

- $3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula }\}$.
  - In a satisfiable cnf-formula, each clause must contain at least one literal that is assigned 1.

An Example Reduction: Reducing 3*SAT* to *CLIQUE*

Theorem 7.32

3*SAT* is polynomial time reducible to *CLIQUE*.

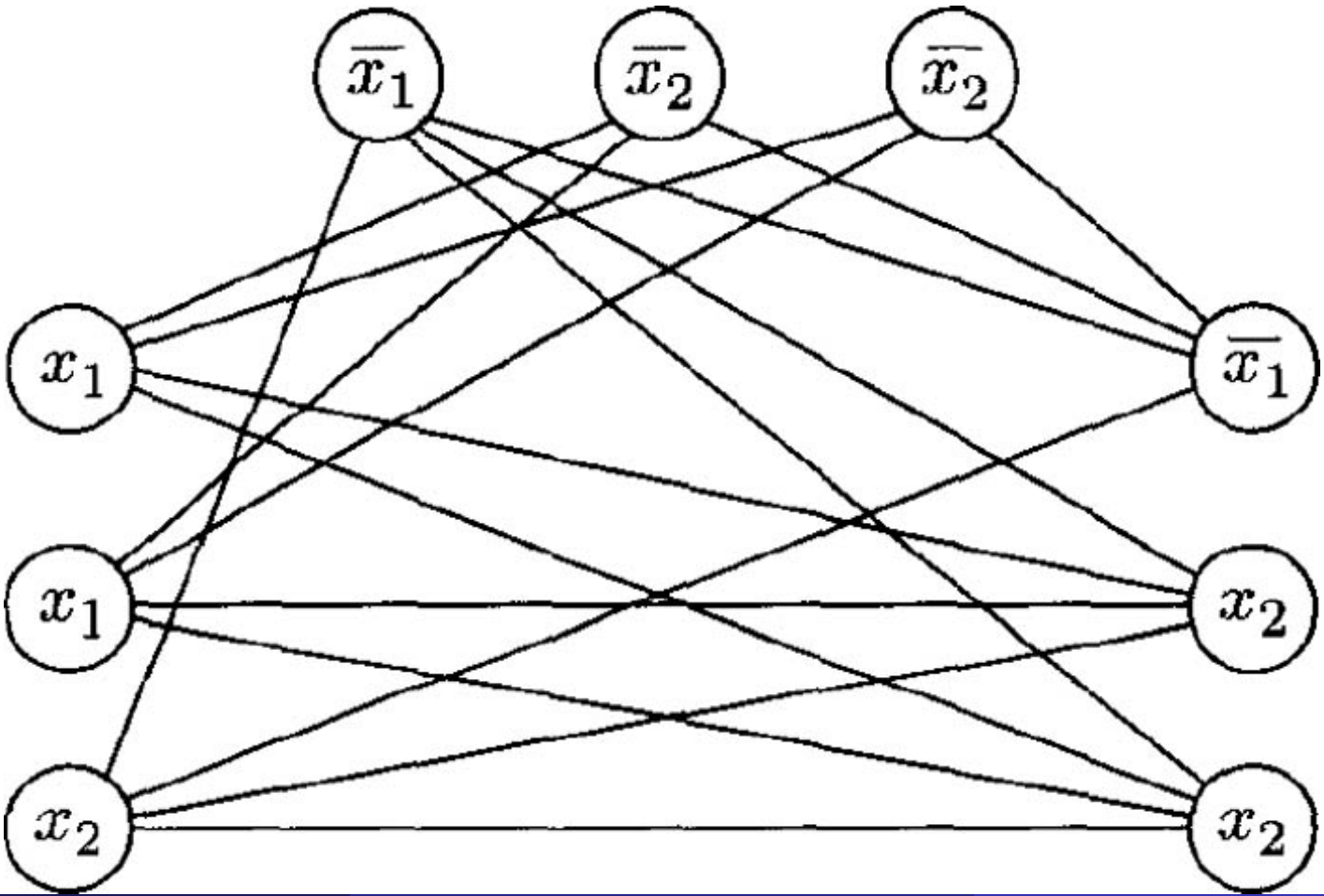
Proof Idea

Take any 3*SAT* formula and polynomial-time reduce it to a graph such that if the graph has a clique then the 3cnf-formula is satisfiable.

- Some details:
  - ϕ is a formula with k clauses each with 3 literals.
  - The k clauses in ϕ map to k groups of 3 nodes each called a triple.
  - Each node in the triple corresponds to one of the literals in the corresponding clause.
  - No edges between the nodes in a triple.
  - No edges between "conflicting" nodes (e.g., x and $\overline{x}$)

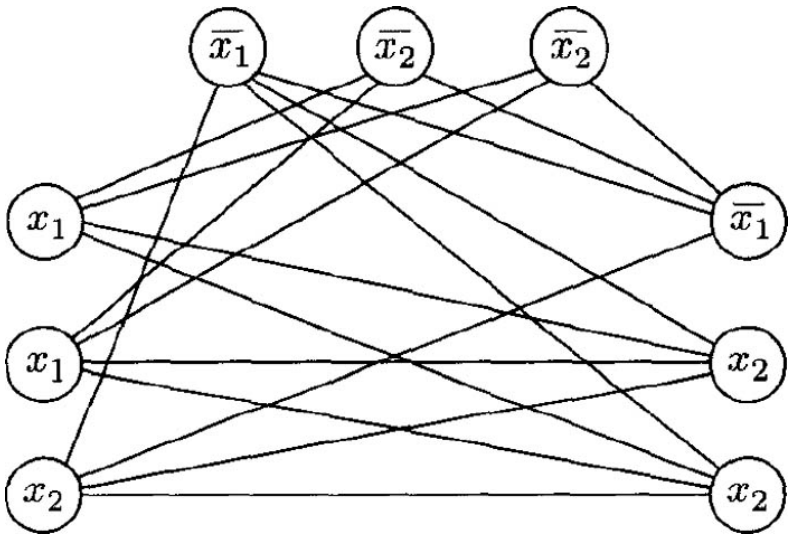
An Example Reduction: Reducing 3*SAT* to *CLIQUE*

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



An Example Reduction: Reducing 3*SAT* to *CLIQUE*

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



- If ϕ has a satisfying assignment, then at least one literal in each clause needs to be 1.
- We select the corresponding nodes in the corresponding triples.
- These nodes should form a k-clique.
- If G has a k-clique, then selected nodes give a satisfying assignment to variables.

NP-Completeness

Definition – NP-Completeness

A language B is NP-complete if it satisfies two conditions:

1. B is in NP, and
2. Every A in NP is polynomial time reducible to B.

Theorem

If B is NP-complete and $B \in$ P, then P $=$ NP. (Obvious)

Theorem

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

Proof

All $A \leq_P B$ and $B \leq_P C$ thus all $A \leq_P C$.

The Cook-Levin Theorem (again)

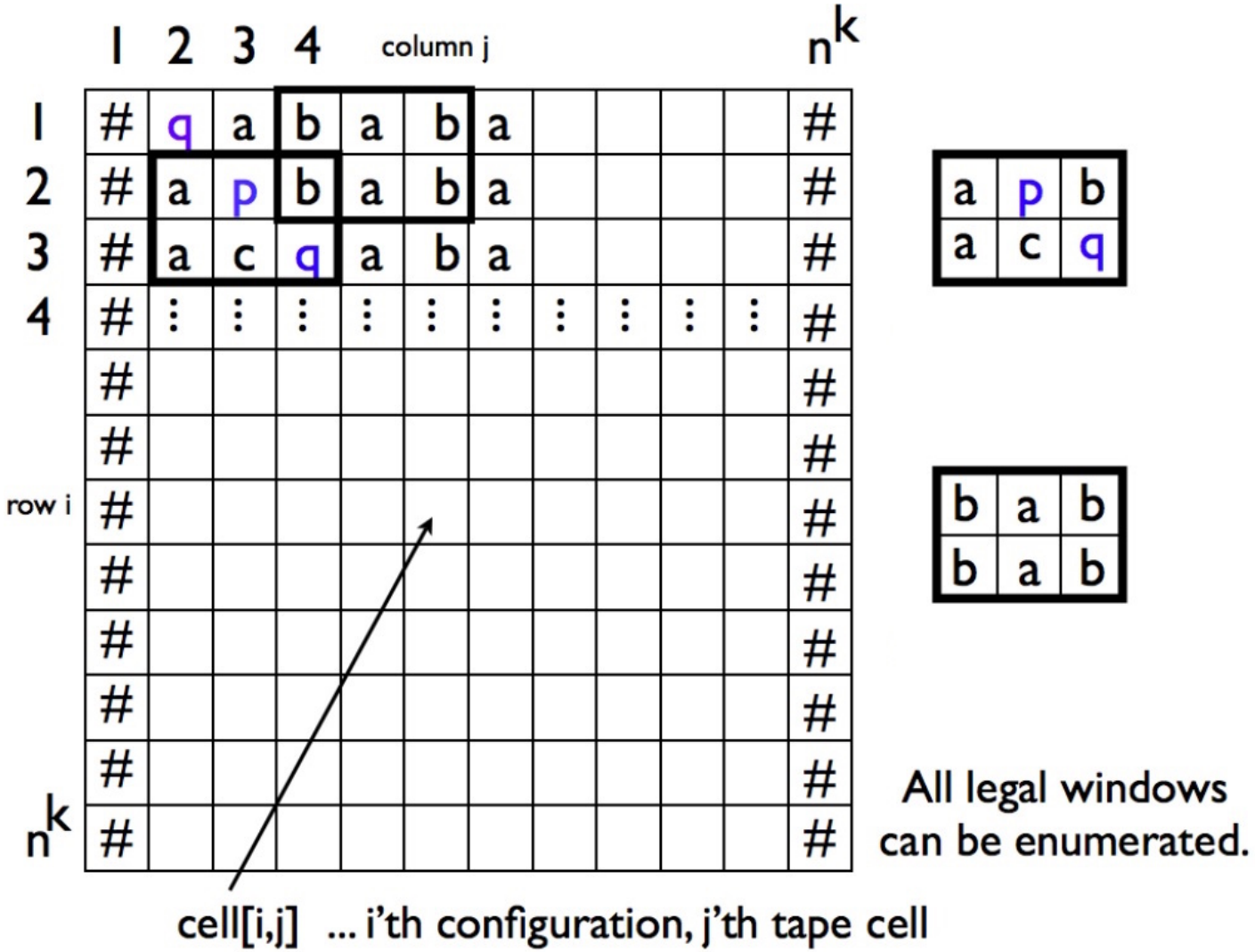
Theorem

SAT is NP-Complete.

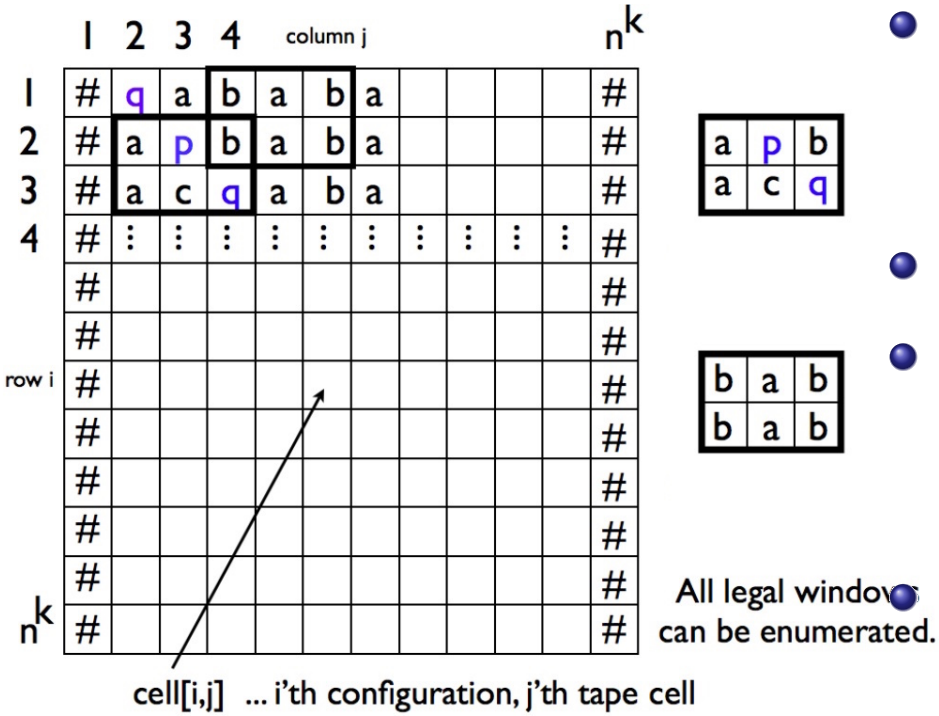
Proof Idea

- Showing *SAT* is in NP is easy.
  - Nondeterministically guess the assignments to variables and accept if the assignments satisfy ϕ
- We can encode the accepting computation history of a polynomial time NTM for every problem in NP as a *SAT* formula ϕ.
- Thus every language $A \in$ NP is polynomial-time reducible to *SAT*.
  - N is a NTM that can decide A in time $O(n^k)$
  - N accepts w if and only if ϕ is satisfiable.

BIRD'S EYE VIEW OF A POLYNOMIAL TIME COMPUTATION BRANCH



Bird's eye view of a polynomial time computation branch



- We represent the computation of a NTM N on w with a $n^k \times n^k$ table, called a tableau.
- Rows represent configurations
- First row is the start configuration (w + lots of blanks to fill the remaining of the n^k cells.)
- Each row follows from the previous one using N's transition function.

- A tableau is accepting if any row of the tableau is an accepting configuration.
- Every accepting tableau for N on w corresponds to an accepting computation branch of N on w.
- If N accepts w, then an accepting tableau exists!

Setting up formula ϕ

The variables

- Let $C = Q \cup \Gamma \cup \{\#\}$.
- For $1 \leq i, j \leq n^k$ and for each $s \in C$, we have a variable $x_{i,j,s}$.
- $x_{i,j,s} = 1$ if the $cell[i, j]$ contains the symbol s.
- Note that the number of variables is polynomial function of n.

The Formula ϕ

$$\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

- ϕ_{cell} makes sure that there is only one symbol in every cell!
- ϕ_{start} makes sure the start configuration is correct.
- ϕ_{accept} makes sure the accept state occurs somewhere.
- ϕ_{move} makes sure configurations follow each other legally.

- For all i and j, if $cell[i, j]$ contains symbol s, (that is $x_{i,j,s} = 1$), it can not contain another symbol (that is, no other variable with the same i and j, but a different symbol, is 1).

$$\phi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

- For all i and j, if $cell[i, j]$ contains symbol s, (that is $x_{i,j,s} = 1$), it can not contain another symbol (that is, no other variable with the same i and j, but a different symbol, is 1).

$$\phi_{cell} = \underbrace{\bigwedge_{1 \leq i,j \leq n^k}}_{\text{for all } i \text{and } j} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

- For all i and j, if $cell[i, j]$ contains symbol s, (that is $x_{i,j,s} = 1$), it can not contain another symbol (that is, no other variable with the same i and j, but a different symbol, is 1).

$$\phi_{cell} = \underbrace{\bigwedge_{1 \leq i,j \leq n^k}}_{\text{for all } i \text{ and } j} \left[\underbrace{\left(\bigvee_{s \in C} x_{i,j,s} \right)}_{\substack{\text{at least one symbol} \\ \text{is in a cell}}} \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

- For all i and j, if $cell[i, j]$ contains symbol s, (that is $x_{i,j,s} = 1$), it can not contain another symbol (that is, no other variable with the same i and j, but a different symbol, is 1).

$$\phi_{cell} = \underbrace{\bigwedge_{1 \le i,j \le n^k}}_{\text{for all } i \text{ and } j} \left[\underbrace{\left(\bigvee_{s \in C} x_{i,j,s} \right)}_{\substack{\text{at least one symbol} \\ \text{is in a cell}}} \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} \overbrace{\left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right)}^{\text{only one symbol in a cell}} \right) \right]$$

- Note that ϕ_{cell} is in a conjuctive normal form.

- ϕ_{start} sets up the first configuration.

$$\begin{aligned}\phi_{start} \quad = \quad & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \cdots x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \cdots x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}\end{aligned}$$

- ϕ_{start} sets up the first configuration.

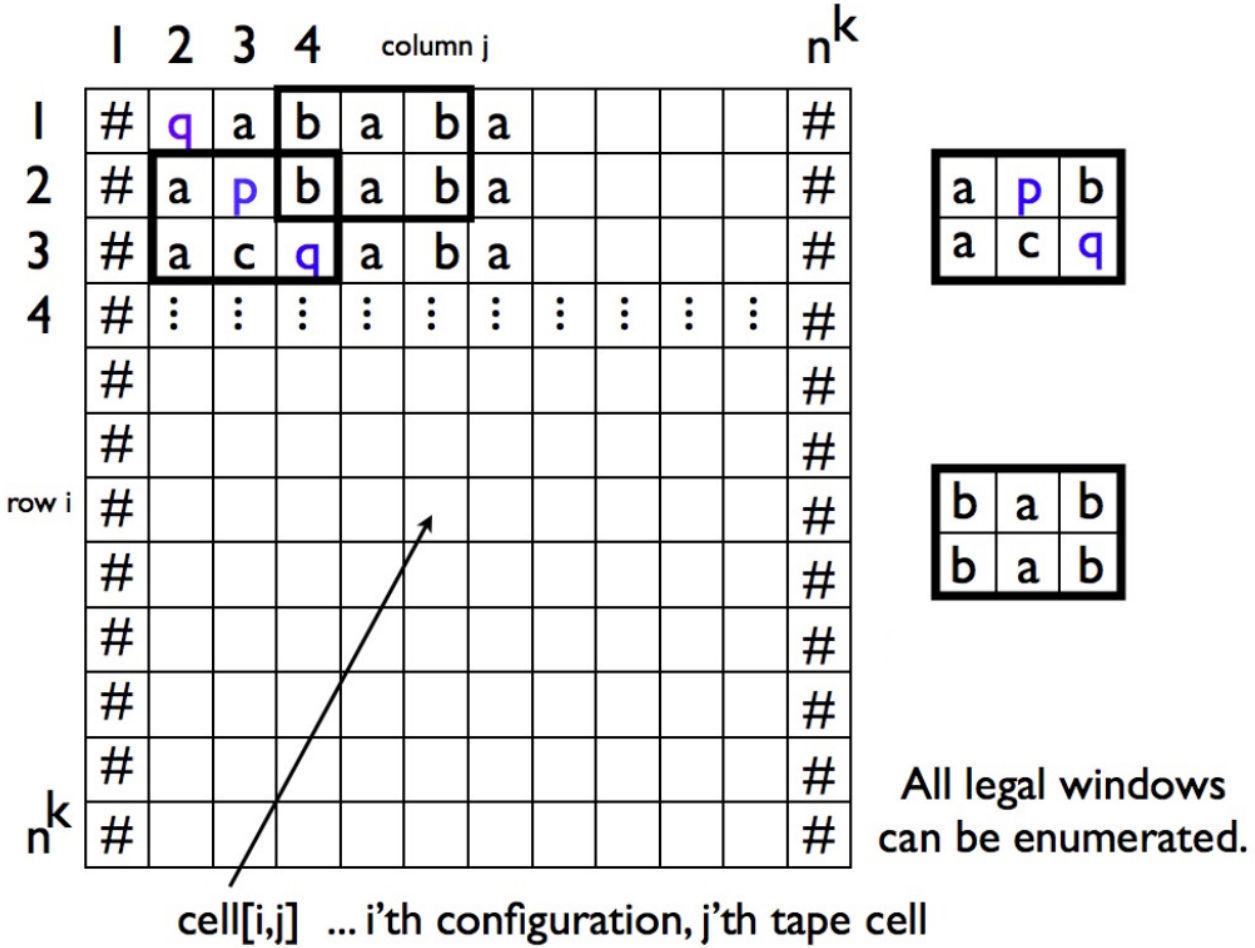
$$\phi_{start} = \overbrace{\underbrace{x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \cdots x_{1,n+2,w_n} \wedge}_{q_0 \text{ and input symbols}} \underbrace{x_{1,n+3,\sqcup} \wedge \cdots x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}}_{\text{all the blanks to the right}}}$$

ϕ_{accept}

- ϕ_{accept} says q_{accept} occurs somewhere.

$$\phi_{accept} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}}$$

- ϕ_{move} is the most interesting of the subformulas



- How many possible such windows are there?
- There are $|C|^6$ possible such windows.

Definition – Legal Window

A 2×3 window is legal if that window does not violate the actions specified by N's transition function.

- Suppose δ of N has the entries
  - $\delta(q_1, a) = \{(q_1, b, R)\}$
  - $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$
- The following windows are legal:

a	q_1	b
q_2	a	c

a	q_1	b
a	a	q_2

a	a	q_1
a	a	b

#	b	a
#	b	a

a	b	a
a	b	q_2

b	b	b
c	b	b

Definition – Legal Window

A 2×3 window is legal if that window does not violate the actions specified by N's transition function.

- Suppose δ of N has the entries
  - $\delta(q_1, a) = \{(q_1, b, R)\}$
  - $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$
- The following windows are NOT legal:

a	b	a
a	a	a

a	q_1	b
q_1	a	a

b	q_1	b
q_2	b	q_2

Claim

If the top row of the table is the start configuration and every window in the tableau is legal, then every row of the table (after the first) is a configuration that follows the preceding one!

Thus

$$\phi_{move} = \bigwedge_{1 \le i < n^k, 1 < j < n^k} \text{(the } (i,j) \text{ window is legal)}$$

Where " (the (i,j) window is legal) " is actually the following formula

$$\bigvee_{\substack{a_1,a_2,a_3,a_4,a_5,a_6 \\ \text{is a legal window}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

- i, j refers to the top middle cell of a window.
- We have $O(n^{2k})$ variables $(= |C| \times n^k \times n^k)$
- The total formula size is $O(n^{2k})$, so it is polynomial time reduction.

3*SAT* is NP-complete

Corollary

3*SAT* is NP-complete.

- Every formula in the construction of the NP-completeness proof of *SAT* can actually be written as a conjuctive normal form formula with 3 literals per clause.
  - If a clause has less that 3 literals, repeat one.
  - Disjunctive normal form clauses can be transformed into conjunctive normal form clauses, e.g.,

$$(a \wedge b) \vee (c \wedge d) = (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$$

  - Clauses longer than 3 clauses can be rewritten as clauses with 3 variable, e.g.,

$$(a \vee b \vee c \vee d) = (a \vee b \vee z) \wedge (\overline{z} \vee c \vee d)$$

Formal Languages, Automata and Computation

Proving Problems NP-Complete

Summary

- Complexity Classes: P and NP
- Polynomial time reducibility
- Satisfiability Problem (*SAT*)
  - CNF, 3CNF Forms
  - 3*SAT* Problem
- NP-Completeness
- NP-Completeness of the *SAT* problem
  - Reduction from accepting computation histories of nondeterministic TMs to a *SAT* formula such that
    - A polynomial time NTM accepts *w* iff the corresponding *SAT* formula has a satisfying assignment.
- 3*SAT* is NP-Complete.

Showing Problems NP-complete

- Remember that in order to show a language X to be NP-complete we need to show
  1. X is in NP, and
  2. Every Y in NP is polynomial time reducible to X,
- Part 1 is (usually) easy. You argue that there is polynomial time verifier for X, which, given a solution (certificate), will verify in polynomial time, that, it is a solution.
- For part 2, pick a known NP-complete problem Z
  1. We already know that all problems Y in NP reduce to Z in polynomial time.
  2. We produce a polynomial time algorithm that reduces all instances of Z to some instance of X.
  3. So $Y \leq_P Z$ and $Z \leq_P X$ then $Y \leq_P X$.

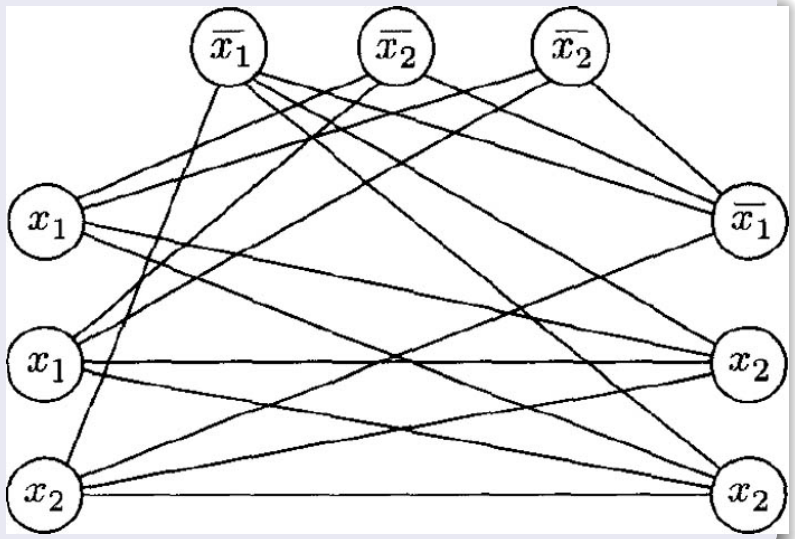
Showing Problems NP-Complete

Theorem

CLIQUE is NP-complete.

Proof

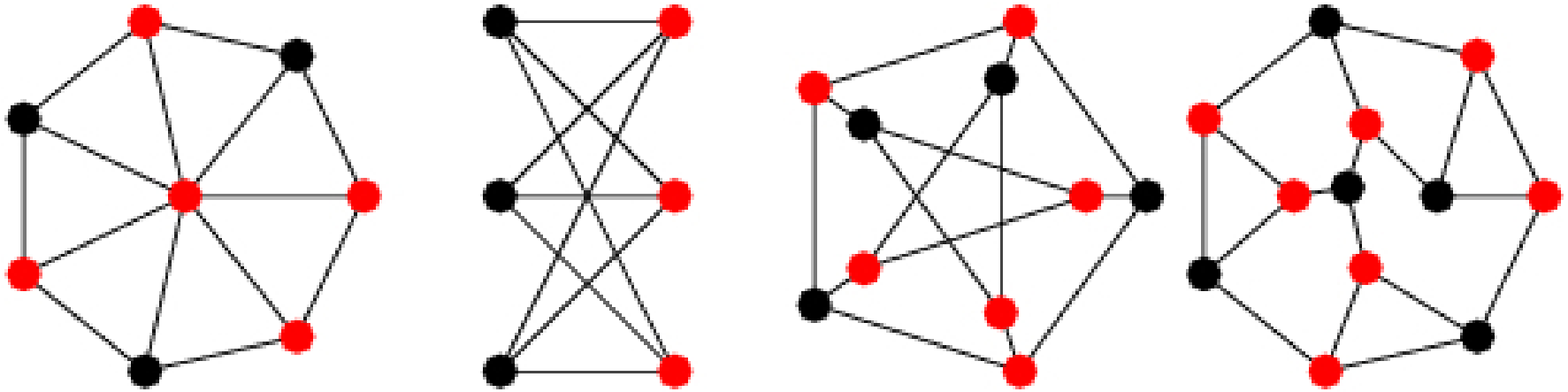
1. We know 3*SAT* is NP-complete.
2. We know that $3SAT \leq_P CLIQUE$.
3. Hence *CLIQUE* is NP-complete.



The Vertex Cover Problem

Definition – Vertex Cover

Given an undirected graph G, a vertex cover of G is a subset of the nodes where every edge of G touches one of those nodes.



- $\mathit{VERTEX\text{-}COVER} = \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$.

The Vertex Cover Problem

Theorem

VERTEX-COVER is NP-complete.

Proof Idea

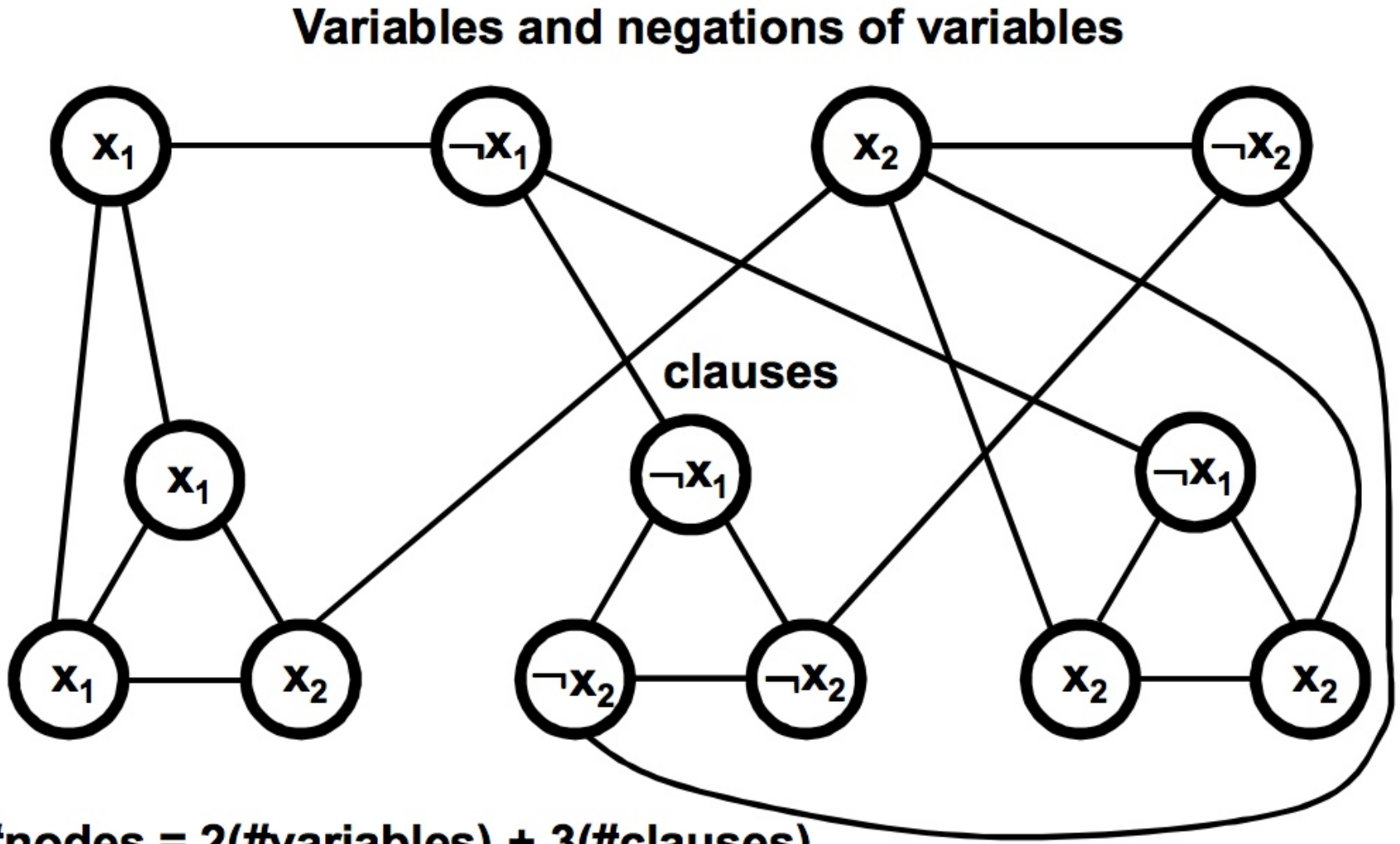
- Show *VERTEX-COVER* is in NP.
  - Easy, the certificate is the vertex cover of size k.
- We reduce an instance of 3*SAT*, ϕ, to a graph G and an integer k so that ϕ is satisfiable whenever G has a vertex cover of size k.
- We employ a concept called gadgets, groups of nodes with specific functions, in the graph.
  - Variable gadgets – representing literals
  - Clause gadgets – representing clauses

The Vertex Cover Problem

- Let ϕ be a 3-cnf formula with m variables and l clauses.
- We construct in polynomial-time, an instance of $\langle G, k \rangle$ where $k = m + 2l$.
  - For each variable x in ϕ, we add two nodes to G labeled x and $\overline{x}$, connected by an edge (variable gadget).
  - For every clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ in ϕ, we add 3 nodes labeled ℓ_1, ℓ_2 and ℓ_3, with edges between every pair so that they form a triangle (clause gadget)
  - We add an edge between any two identically labelled nodes, one from a variable gadget and one from a clause gadget.

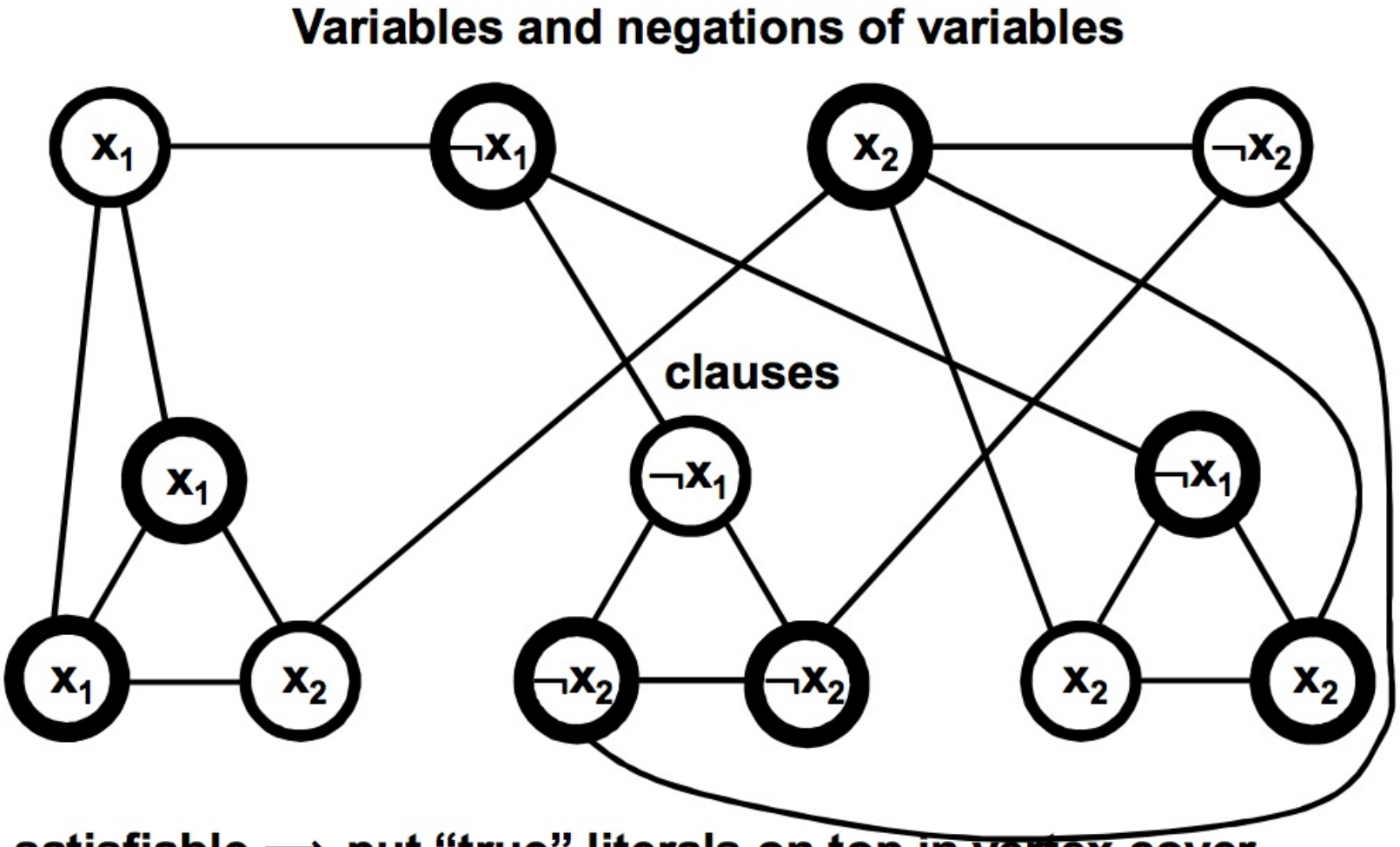
The Vertex Cover Problem

$$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$



The Vertex Cover Problem

$$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

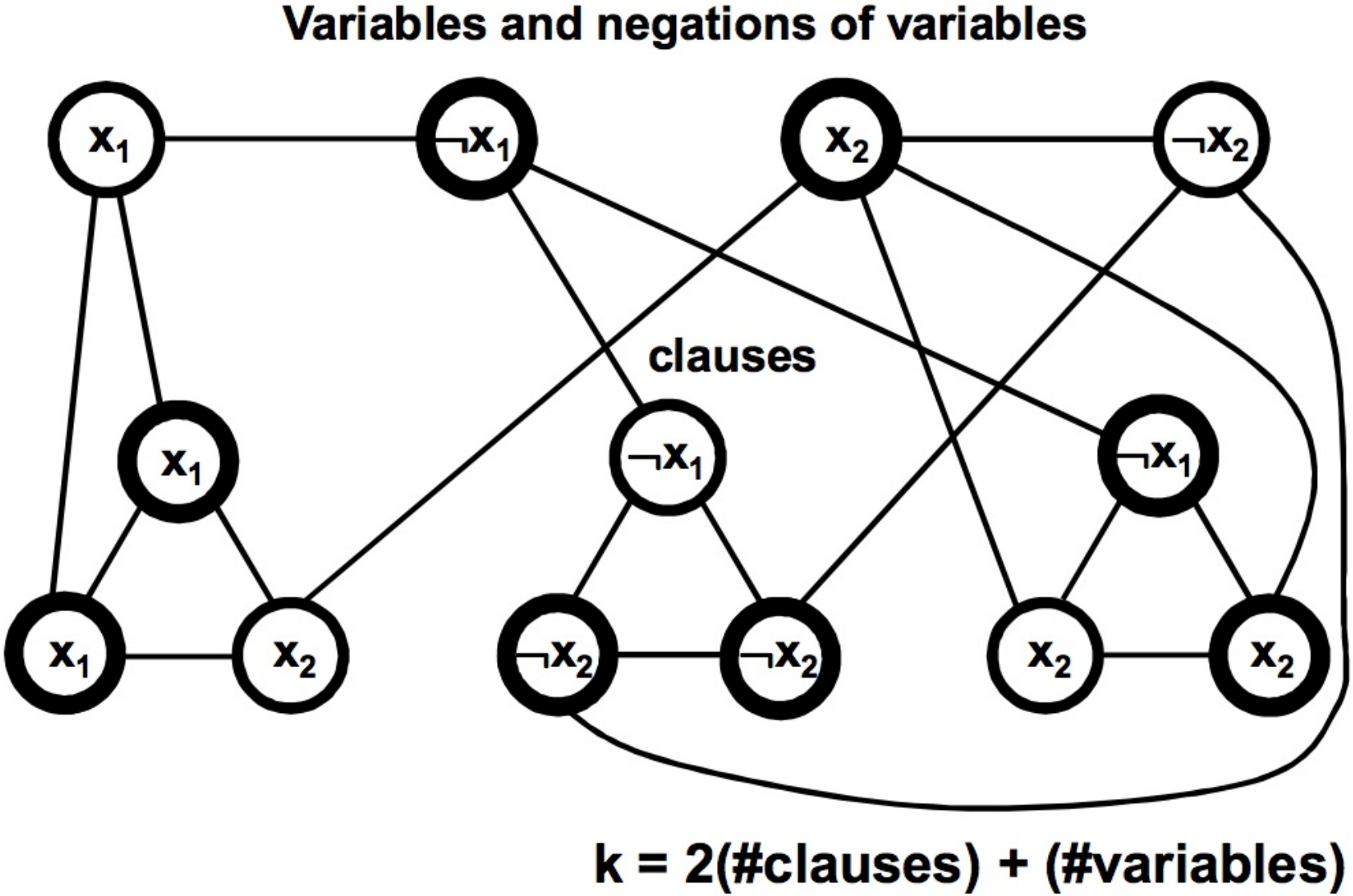


ϕ satisfiable $\Longrightarrow$ put "true" literals on top in vertex cover

For each clause, pick a true literal and put other 2 in vertex cover

The Vertex Cover Problem

$$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$



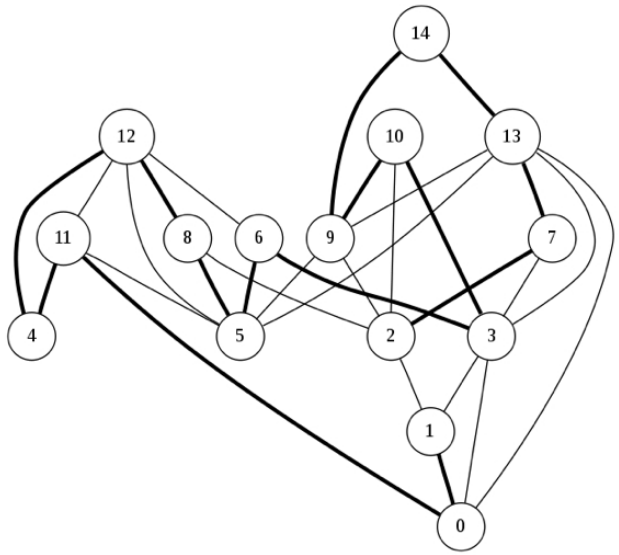
The Hamiltonian Path Problem

Definition - Hamiltonian Path

(Recall that) A Hamiltonian path in a directed graph G is a directed path that goes through each node exactly once.

Definition Hamiltonian Path Problem

$HAMPATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$.



The Hamiltonian Path Problem

Theorem

HAMPATH is NP-complete.

Proof Idea

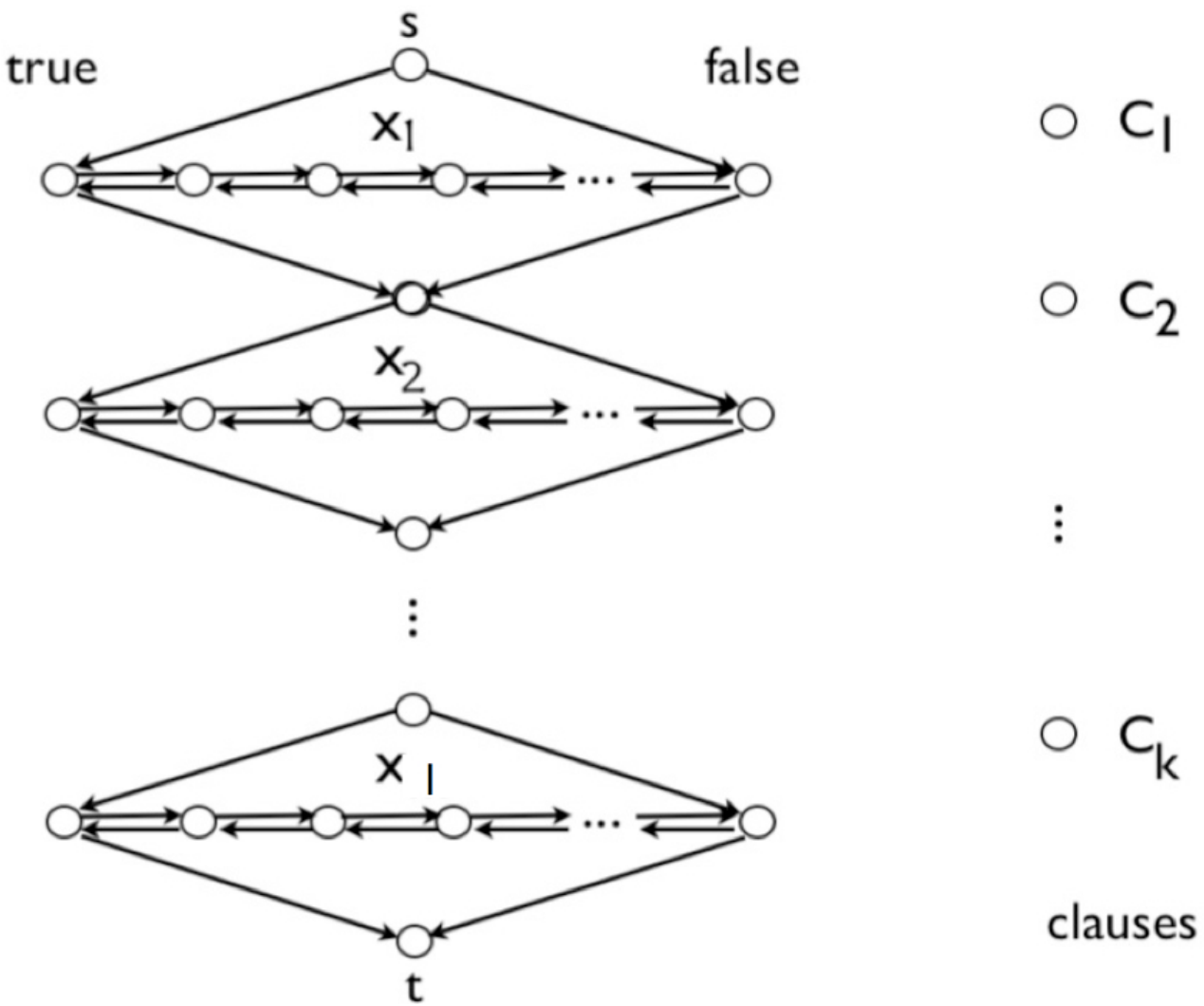
- We show $3SAT \leq_P HAMPATH$.
- We again use gadgets to represent the variables and clauses.
- For a given 3-cnf formula with k clauses

$$\phi = \underbrace{(a_1 \vee b_1 \vee c_1)}_{c_1} \wedge \underbrace{(a_2 \vee b_2 \vee c_2)}_{c_2} \wedge \cdots \wedge \underbrace{(a_k \vee b_k \vee c_k)}_{c_k}$$

where each a_i, b_i or c_i is a literal x or $\overline{x}$. We have l variables $x_1, x_2, \ldots x_l$.

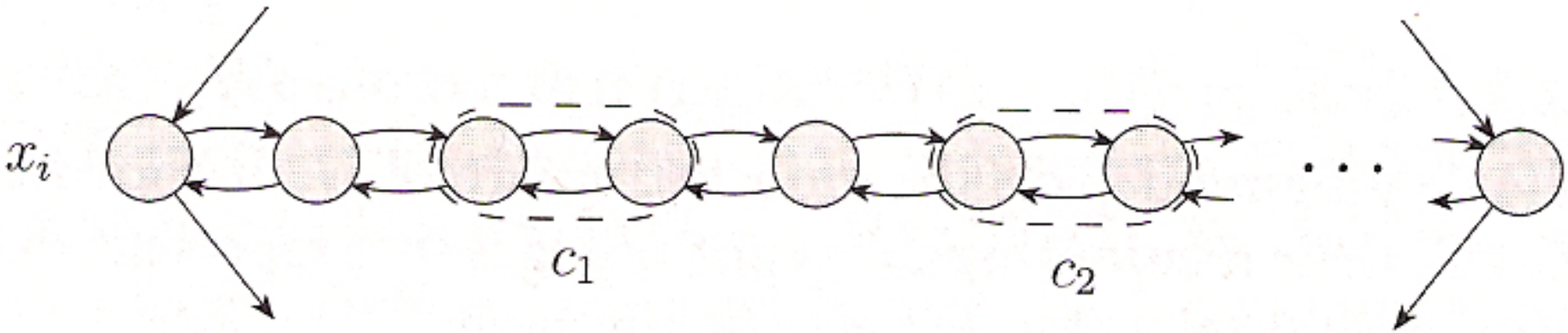
The Hamiltonian Path Problem

- 1-node gadgets for clauses
- Diamond-shaped gadgets for variables



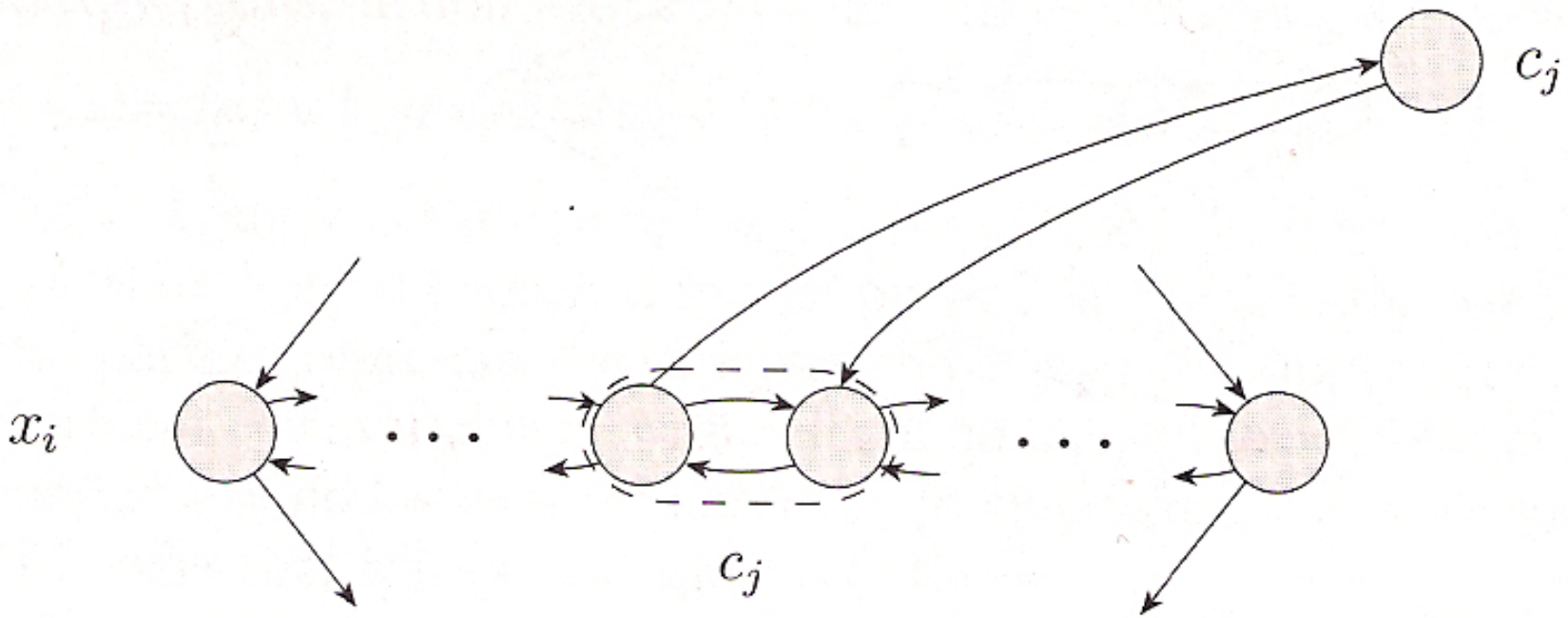
The Hamiltonian Path Problem

- The middle spine in each diamond has $3k + 3$ nodes.
  - 3 nodes per clause + 1 to isolate them from the two literal nodes and 2 nodes on each side for the literals x_i, $\overline{x_i}$.



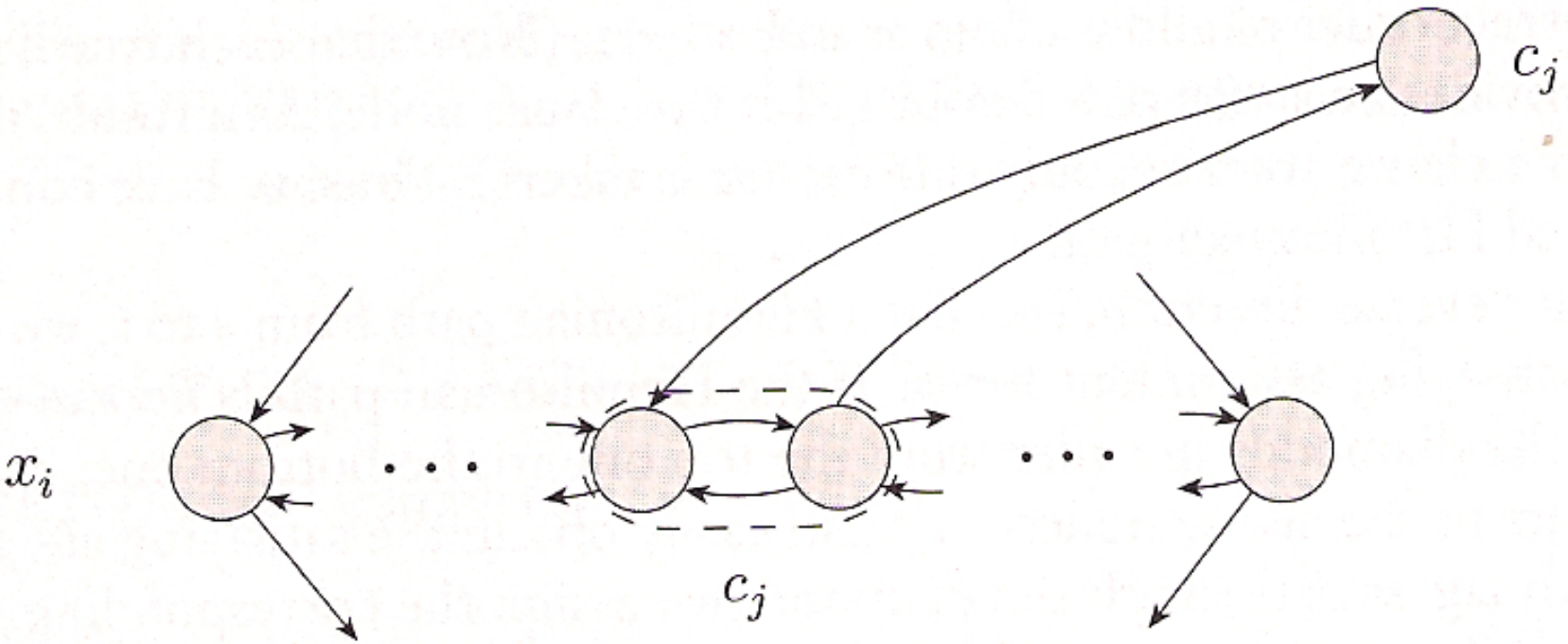
The Hamiltonian Path Problem

- If x_i appears in clause c_j, we add two edges from j^{th} group in the spine to the j^{th} clause node in the i^{th} diamond.



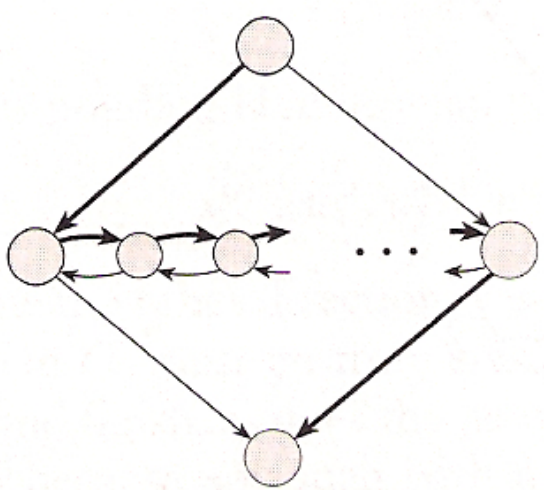
The Hamiltonian Path Problem

- If $\overline{x_i}$ appears in clause c_j, we add two edges from j^{th} group in the spine to the j^{th} clause node in the i^{th} diamond, but in the reverse direction.

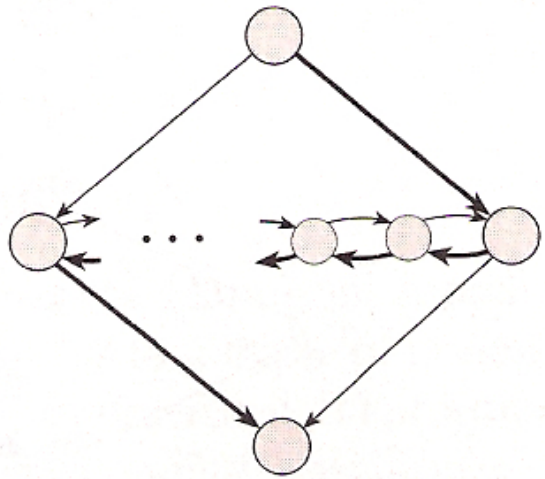


The Hamiltonian Path Problem

- Suppose ϕ is satisfiable.
- Ignoring the clause nodes, we note that the Hamiltonian path
  - starts at *s*
  - goes through each diamond
  - ends up at *t*.
- In diamond *i*, it either goes left-to-right or right-to-left depending on the truth value of variable x_i.



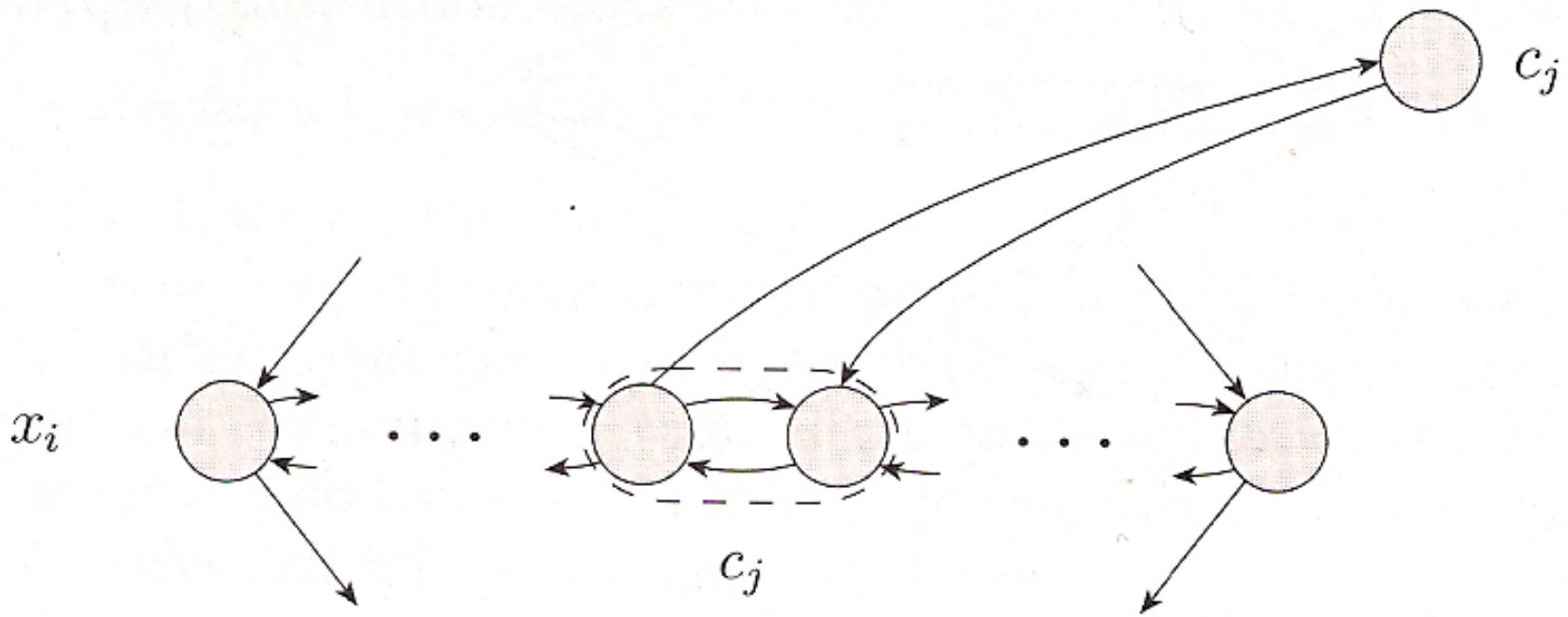
zig-zag



zag-zig

The Hamiltonian Path Problem

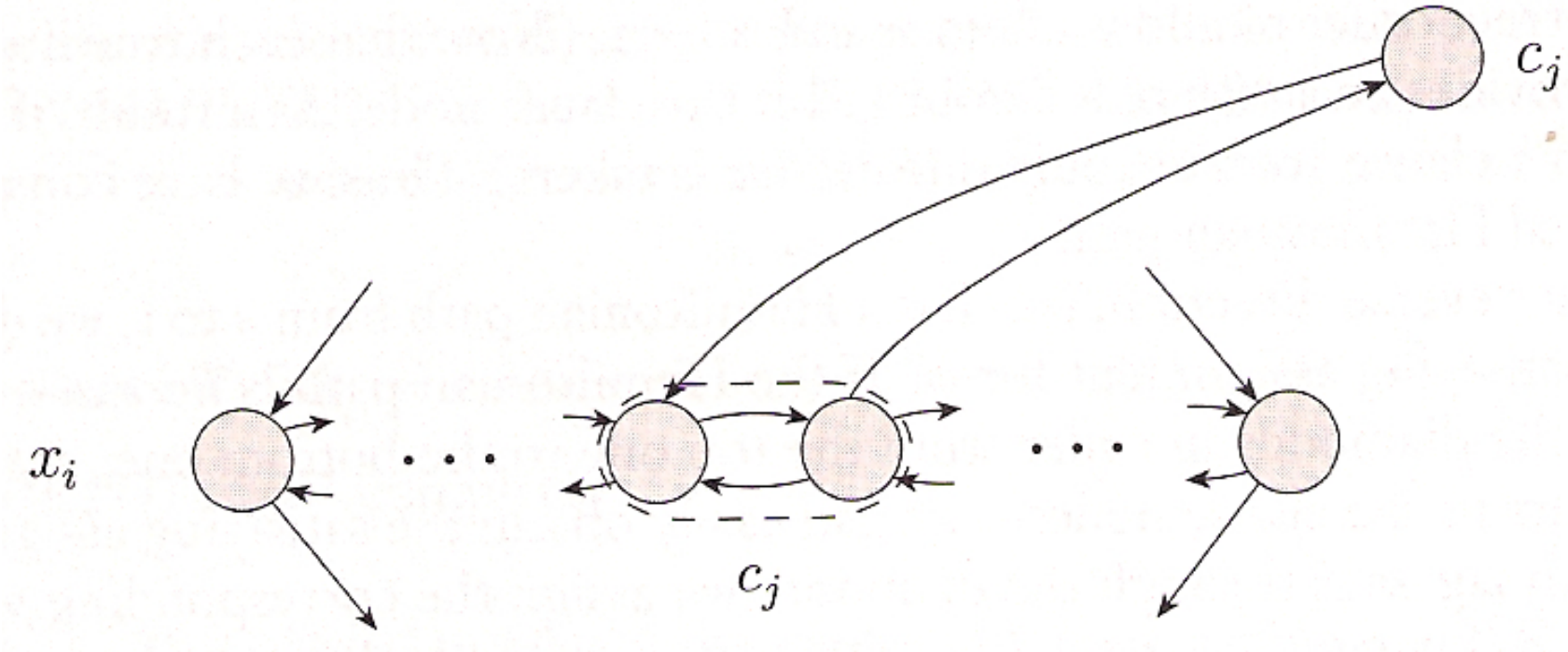
- The clause nodes can be incorporated into the path using the detours we provided.
- So if x_i is true and is in clause c_j, we can take a detour to node for c_j and back to the spine in the right direction.



- Note that each detour is optional but we have to incorporate c_j only once.

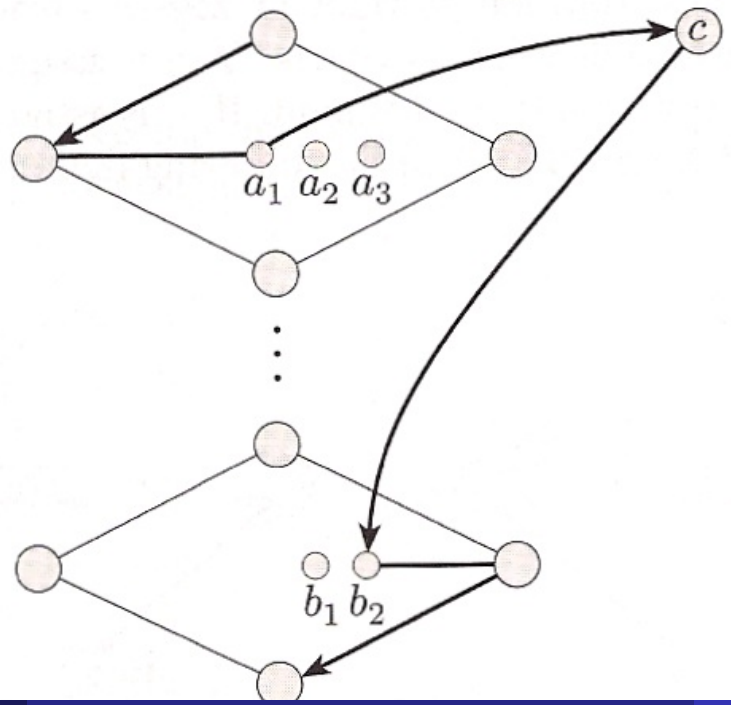
The Hamiltonian Path Problem

- The clause nodes can be incorporated into the path using the detours we provided.
- So if $\overline{x_i}$ is true and is in clause c_j, we can take a detour to node for c_j and back to the spine in the reverse direction.



The Hamiltonian Path Problem

- How about the reverse direction? If G has a Hamiltonian path then ϕ has a satisfying assignment?
- If the path is normal, that is, it goes through from s zigzagging through the diamonds, then clearly there is a satisfying assignment.
- The following case can not happen!



The Undirected Hamiltonian Path

Definition Hamiltonian Path Problem

$UHAMPATH = \{\langle G, s, t\rangle \mid G$ is an undirected graph with a Hamiltonian path from s to $t\}$.

Theorem

UHAMPATH is NP-complete.

Proof Idea

- We reduce *HAMPATH* to *UHAMPATH*.
- All nodes except s and t in the directed graph G, map to 3 nodes in the undirected graph G'.
- G has a Hamiltonian path $\Leftrightarrow$ G' has an undirected Hamiltonian path.

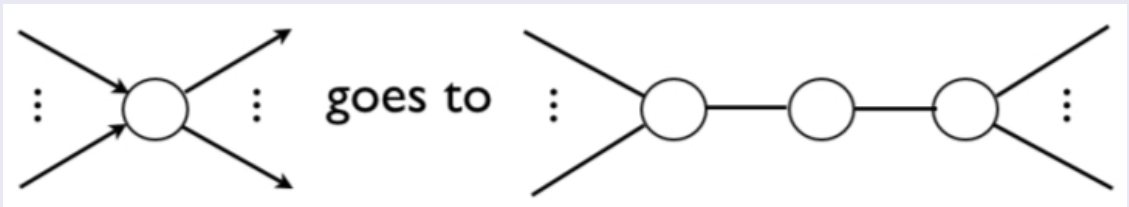
The Undirected Hamiltonian Path

Theorem

UHAMPATH is NP-complete.

Proof

- s in G maps to s^{out} in G'.
- t in G maps to t^{in} in G'.
- Any other node u_i maps to u_i^{in}, u_i^{mid}, u_i^{out} in G'.
  - All arcs coming to u_i in G become edges incident on u_i^{in} in G'.
  - All arcs going out from u_i in G become edges incident on u_i^{out} in G'.



The Undirected Hamiltonian Path

- Note that if

  $$s, u_1, u_2, \ldots, u_k, t$$

  is a Hamiltonian path in G then

  $$s^{out}, u_1^{in}, u_1^{mid}, u_1^{out}, u_2^{in}, u_2^{mid}, u_2^{out} \ldots, u_k^{out}, t^{tin}$$

  is a Hamiltonian path in G'.
- Any Hamiltonian path between s^{out} and t^{in}, must go through the triple of nodes except for the start and end nodes.

The Subset Sum Problem

$$\textit{SUBSET-SUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_m\} \text{ and for some } \{y_1, \ldots, y_n\} \subseteq S, \sum y_i = t\}$$

Theorem

SUBSET-SUM is NP-complete.

Proof Idea

- We reduce 3*SAT* to an instance of the *SUBSET-SUM* problem with a set S and a bound t,
  - so that if a formula ϕ has a satisfying assignment,
  - then S has a subset T that adds to t
- We already know that *SUBSET-SUM* is in NP.

The Subset Sum Problem

- Let ϕ be a formula with variables $x_1, x_2, \ldots, x_l$ and clauses $c_1, \ldots, c_k$.
- We compute $m = 2 \times l + 2 \times k$ (large) numbers from ϕ and a bound t
- Such that when we choose the numbers corresponding to the literals in the satisfying assignment, they add to t.

The Subset Sum Problem

S for $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \cdots) \wedge \cdots \wedge (\overline{x_3} \vee \cdots \vee \cdots)$

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

The Subset Sum Problem

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

- We choose one of the numbers y_i if $x_i = 1$, or z_i if $x_i = 0$.
- The left part of t will add up the right number.
- The right side columns will at least be 1 each
- We take enough of the g and h's to make them add up to 3.

Formal Languages, Automata and Computation

Space Complexity

Space Complexity

- (Disk) Space – the final frontier!
- How much memory do computational problems require?
- We characterize problems based on their memory requirements.
- Space is reusable, time is not!
- We again use the Turing machine as our model of computation.

Space Complexity

Definition – Space Complexity

Let M be a deterministic Turing machine that halts on all inputs. The space complexity of M is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the **maximum number of tape cells** that M scans on any input of length n.

For nondeterministic TMs where all branches halt on all inputs, we take the maximum over all the branches of computation.

Space Complexity

Definition – Space Complexity Classes

Let $f : \mathcal{N} \longrightarrow \mathcal{R}^+$. The space complexity classes are defined as follows:

$$\text{SPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space deterministic TM}\}$$

$$\text{NSPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space nondeterministic TM}\}$$

- SPACE($f(n)$) formalizes the class of problems that can be solved by computers with bounded memory. (Real world!)
- SPACE($f(n)$) problems could potentially take a long time to solve.
- Intuitively space and time seem to be interchangeable.
- Just because a problem needs only *linear space* does not mean it can be solved in *linear time*.

Deterministic Space Complexity of *SAT*

- *SAT* is NP-complete.
- But *SAT* can be solved in linear space.
- M_1 = "On input $\langle \phi \rangle$, where ϕ is a Boolean formula:
  1. For each truth assignment to the variables $x_1, x_2, \ldots, x_m$ of ϕ:
  2. Evaluate ϕ on that truth assignment.
  3. If ϕ ever evaluates to 1, *accept*; if not, *reject*."

3SAT $\in$ **SPACE(n)**

- Note that M_1 takes exponential time.

Nondeterministic Space Complexity of ALL_{NFA}

- Consider $ALL_{NFA} = \{\langle A \rangle \mid A \text{ is a NFA and } L(A) = \Sigma^*\}$
- The following nondeterministic linear space algorithm decides $\overline{ALL_{NFA}}$.
- Nondeterministically guess an input string rejected by the NFA and use linear space to guess which states the NFA could be at a given time.
- $N =$ "On input $\langle M \rangle$ where M is an NFA.
  1. Place a marker on the start state of NFA.
  2. Repeat 2^q times, where q is the number of states of M.
     2.1 Nondeterministically select an input symbol and change the position of the markers on M's states, to simulate reading that symbol.
  3. *Accept* if stage 2 reveals some string that M rejects, i.e., if at some point none of the markers lie on accept states of M. Otherwise, *reject*."

Nondeterministic Space Complexity of ALL_{NFA}

- Since there are at most 2^q subsets of the states of M, it must reject one of length at most 2^q, if M rejects any strings.
  - Remember that determinization could end up with at most 2^q states.
- N needs space for
  - storing the locations of the markers ($O(q) = O(n)$)
  - the repeat loop counter ($O(q) = O(n)$)
- Hence N runs in nondeterministic $O(n)$ space.
- Note that N runs in nondeterministic $2^{O(n)}$ time.
  - ALL_{NFA} is not known to be in NP or coNP.

Savitch's Theorem

- Remember that simulation of a nondeterministic TM with a deterministic TM requires an exponentional increase in time.
- Savitch's Theorem shows that any nondeterministic TM that uses $f(n)$ space can be converted to a deterministic TM that uses only $f^2(n)$ space, that is,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$$

- Obviously, there will be a slowdown.

Savitch's Theorem

Theorem

For any function $f : \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$$

Proof Idea

- Let N be a nondeterministic TM with space complexity $f(n)$.
- Construct a deterministic machine M that tries every possible branch of N.
- Since each branch of N uses at most $f(n)$ space, then M uses space at most $f(n)$ space $+$ space for book-keeping.
- We need to simulate the nondeterministic computation and save as much space as possible.

Savitch's Theorem

- Given two configurations c_1 and c_2 of a $f(n)$ space TM N, and a number t, determine if we can get from c_1 to c_2 within t steps.
- *CANYIELD* = " On input c_1, c_2 and t:
  1. If $t = 0$ accept iff $c_1 = c_2$
  2. If $t = 1$ accept iff $c_1 = c_2$ or c_1 yields c_2 in one step.
  3. If $t > 1$ then for every possible configuration c_m of N for w, using space $f(n)$
  4. Run *CANYIELD*$(c_1, c_m, \frac{t}{2})$.
  5. Run *CANYIELD*$(c_m, c_2, \frac{t}{2})$.
  6. If steps 4 and 5 both accept, then *accept*.
  7. If haven't yet accepted, *reject*."
- Space is reused during the recursive calls.
- The depth of the recursion is at most $\log t$.
- Each recursive step uses $O(f(n))$ space and $t = 2^{O(f(n))}$ so $\log t = O(f(n))$. Hence total space used is $O(f^2(n))$.

Savitch's Theorem

- M simulates N using *CANYIELD*.
- If n is the length of w, we choose d so that N has no more than $2^{df(n)}$ configurations each using $f(n)$ tape.
- $2^{df(n)}$ provides an upper bound on the running time on any branch of N.
- $M =$ "On input w:
  1. Output the result of *CANYIELD*$(c_{start}, c_{accept}, 2^{df(n)})$."
- At each stage, *CANYIELD* stores c_1, c_2, and t for a total of $O(f(n))$ space.
- Minor technical points with the accepting configuration and the initial value of t (e.g., how does the TM know $f(n)$?) – See the book.

The Class PSPACE

Definition – PSPACE

PSPACE is the class of languages that are decidable in polynomial space on a deterministic TM.

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

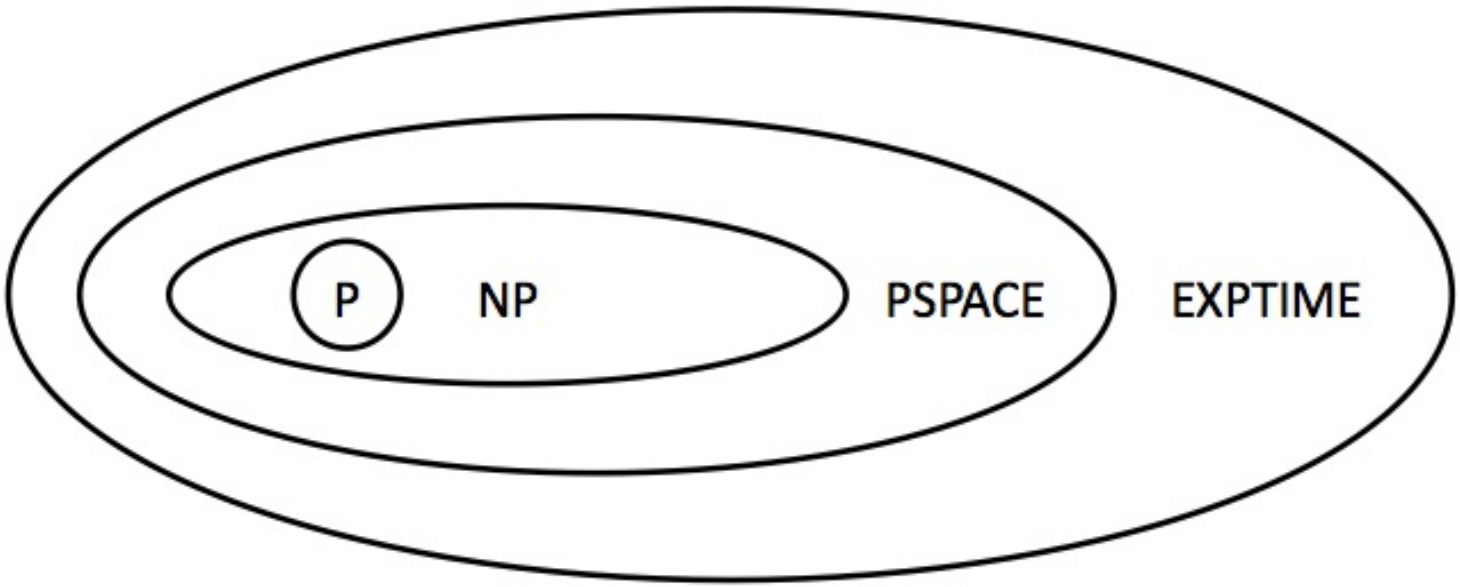
- NSPACE is defined analogously.
- But PSPACE $=$ NSPACE, due to Savitch's theorem, because the square of a polynomial is also a polynomial.

The Class PSPACE – Some observations

- We know $SAT \in \text{SPACE}(n)$.
  - $\Rightarrow SAT \in$ PSPACE.
- We know $\overline{ALL_{NFA}} \in \text{NSPACE}(n)$ and hence $\overline{ALL_{NFA}} \in \text{SPACE}(n^2)$, by Savitch's theorem.
  - $\Rightarrow \overline{ALL_{NFA}} \in$ PSPACE.
- Deterministic space complexity classes are closed under complementation, so $ALL_{NFA} \in \text{SPACE}(n^2)$.
  - $\Rightarrow ALL_{NFA} \in$ PSPACE.
- A TM that operates in $f(n) \geq n$ time, can use at most $f(n)$ space.
  - $\Rightarrow \text{P} \subseteq \text{PSPACE}$
- $\text{NP} \subseteq \text{NPSPACE} \Rightarrow \text{NP} \subseteq \text{PSPACE}$.

The Class PSPACE – Some observations

- We can also bound the time complexity in terms of the space complexity.
- For $f(n) \geq n$, a TM that uses $f(n)$ space, can have at most $f(n)2^{O(f(n))}$ configurations.
  - $f(n)$ symbols on tape, so $|\Gamma|^{f(n)}$ possible strings and $f(n)$ possible state positions and $|Q|$ possible states $= 2^{O(f(n))}$
- PSPACE $\subseteq$ EXPTIME $= \bigcup_k$ TIME(2^{n^k}).



PSPACE-Completeness

Definition – PSPACE-complete

A language B is PSPACE-complete if it satisfies two conditions:

1. B is in PSPACE, and
2. every A in PSPACE is polynomial time reducible to B.

- Note that we use polynomial-time reducibility!
- The reduction should be easy relative to the complexity of typical problems in the class.
- In general, whenever we define complete-problems for a complexity class, the reduction model must be more limited that the model use for defining the class itself.

The TQBF Problem

- Quantified Boolean Formulas are exactly like the Boolean formulas we define for the *SAT* problem, but additionally have existential ($\exists$) and universal ($\forall$) quantifiers.
  - $\forall x[x \vee y]$
  - $\exists x \exists y[x \vee \overline{y}]$
  - $\forall x[x \vee \overline{x}]$
  - $\forall x[x]$
  - $\forall x \exists y[(x \vee y) \wedge (\overline{x} \vee \overline{y})]$
- A fully quantified Boolean formula is a quantified formula where every variable is quantified.
  - All except the first above are fully quantified.
  - A fully quantified Boolean formula is also called a sentence, and is either true or false.

Definition – TQBF

$TQBF = \{\langle \phi \rangle \mid \phi$ is a true fully quantified Boolean formula$\}$

The TQBF Problem

Theorem

TQBF $= \{\langle\phi\rangle \mid \phi$ is a true fully quantified Boolean formula$\}$ is PSPACE-complete.

- Assume T decides *TQBF*.
- If ϕ has no quantifiers, it is an expression with only constants! Evaluate ϕ and accept if result is 1.
- If $\phi = \exists x\psi$, recursively call T on ψ, first with $x = 0$ and then with $x = 1$. Accept if either returns 1.
- If $\phi = \forall x\psi$, recursively call T on ψ, first with $x = 0$ and then with $x = 1$. Accept if both return 1.

The TQBF Problem

Claim

Every language A in PSPACE is polynomial-time reducible to *TQBF*.

- We build a polynomial time reduction from A to *TQBF*
- The reduction turns a string w into a TQBF ϕ that simulates a PSPACE TM M for A on w.
- Essentially the same as in the proof of the NP-completeness of *SAT* – build a formula from the accepting computation history.
- But uses the approach in Savitch's Theorem.
- Details in section 8.3 in the book.
- PSPACE is often called the class of games.
  - Formalizations of many popular games are PSPACE-complete.

The Classes L and NL

- We have so far considered time and space complexity bounds that are at least linear.
- We now examine smaller, sublinear space bounds.
  - For time complexity, sublinear bounds are insufficient to read the entire input!
- For sublinear space complexity, the TM is able to read the whole input but not store it.
- We must modify the computational model!

The Classes L and NL

- We introduce a TM with two-tapes:
  1. A read-only input tape.
  2. A read/write work tape.
- On the input tape, the head always stays in the region where the input is.
- The work tape can be read and written in the usual way.
- Only the cells scanned on the work tape contribute to the space complexity.

Definitions– Log space complexity classes

$$\mathsf{L} = \mathsf{SPACE}(\log n)$$

$$\mathsf{NL} = \mathsf{NSPACE}(\log n)$$

An Algorithm in L

- Consider the (good old) language $A = \{0^k 1^k \mid k \geq 0\}$
- Previous algorithm (zig-zag and cross out symbols) used linear space.
- We can not do this now since the input tape is read-only.
- Once the machine is certain the string is of the desired pattern, it can count the number of 0's and 1's.
- The only additional space needed are for the two counters (in binary).
- A binary counter uses only logarithmic space, $O(\log k)$.

An Algorithm in NL

- Consider the *PATH* problem
  $PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$
- *PATH* is in P, but that algorithm uses linear space.
- It is not known if *PATH* can be solved in deterministic log space.
- It can be solved in nondeterministic log space:
  1. Starting with s, the nondeterministic log space TM guesses the next node to go to on the way to t.
  2. The TM only records the id or the position of the node (so needs log space).
  3. The TM nondeterministically guesses the next node, until either it reaches t or until it has gone for m steps where m is the number of nodes.

The Classes L and NL

- Log-space reducibility
- NL-completeness
- *PATH* is NL-complete.
  - For a given log space nondeterministic TM and input w, map the accepting computation history to a graph, with nodes representing configurations.
- NL $\subseteq$ P (remember *PATH* $\in$ P)
- NL $=$ coNL.
- L $\subseteq$ NL = coNL $\subseteq$P $\subseteq$ PSPACE.

And We are done for the semester (– the final)

- Thanks for your patience and for taking the occasional mental pain.
- But then, no pain no gain!
- We do review in the remaining lecture slots, please come prepared and let me know what major concepts your still have problems with.
- Final on Sunday, December 6, 2015, at 13:00-16:00
- Good luck!