

CS 357 Theory of Computation

Fall 2023

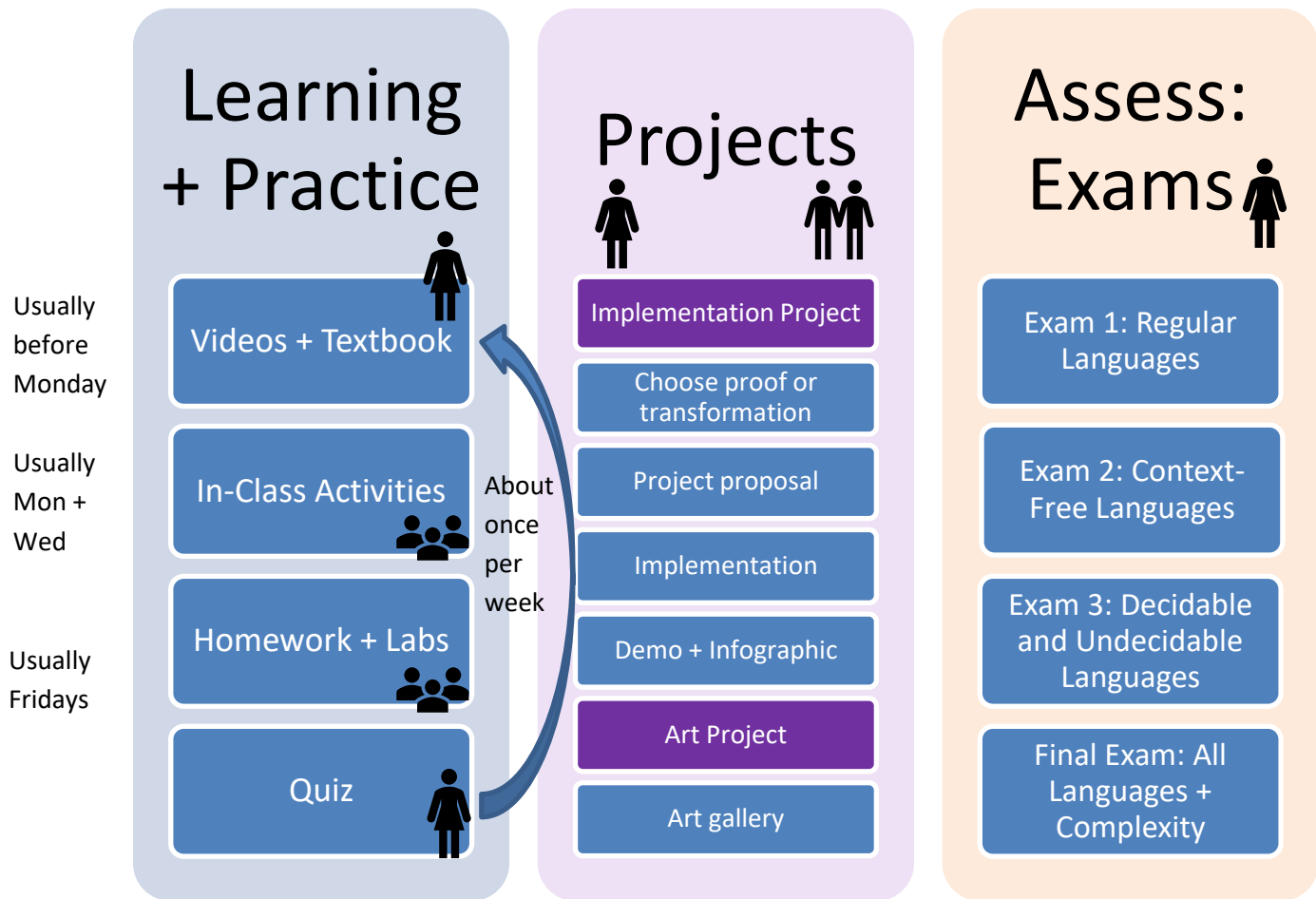
Course Handouts

Dr. Tammy VanDeGrift

Name: _____

If found, call/email: _____

Course Design for Learning



Workload Expectations (~9 hours / week):

- 3 hours in class (activity practice, homework work time, quizzes, exams)
- 1.5 hours reading textbook and/or watching videos
- 2.5 hours homework outside class
- 0.5 hours studying, extra practice, attending office hours
- 1 hour implementation project (amortized across semester)
- 0.5 hours art project (amortized across semester)

**Lecture Handouts – Please bring this booklet to class sessions and
have it available as you watch videos**

**Other course materials are posted to Moodle
Including Calendar/Syllabus**

Announcements

Homework

Resources

learning.up.edu

CS 357 Homework Formatting and Submission Guidelines

1. Write your solution neatly and scan/photo your document OR typeset your solution OR embed hand-drawn or electronic images (i.e. JFLAP) into the file.
2. Include the **complete** problem statement for each exercise before the solution.
3. Label each problem with the exercise number (#1, #2, etc).
4. List any assumptions you are making about the problem if the information is not given in the problem statement.
5. When drawing state machines, be sure the arrows are clearly labeled with the corresponding symbol and be sure the arrow head is clearly visible. Make sure the start state is annotated with an in-arrow. JFLAP is helpful for drawing neat state machines – plus you can simulate strings through the machines.
6. Include your name in the file.
7. Acknowledge other people with whom you worked for each problem.
8. Write “checked in class” after each solution that Tammy checked off in class.
9. Submit homework through Moodle links, which will have the due date and time. See the syllabus for late days policy.

Please follow these guidelines to avoid losing homework points due to formatting errors.

Advice for Working on Homework Assignments

1. Start assignments early, so you have plenty of time to ask for help, if necessary. Feel free to ask questions during class time and office hours. See the syllabus for office hours times.
2. Some class time will be used to practice problems. Some class time will be used to work on homework problems, often times in small groups. Collaborative learning is encouraged; however, you should be able to independently write and explain your solutions. **Copying someone else’s work without contributing to the solution is not okay. Working with someone else or a small study group to co-develop solutions is okay. All group members should be able to explain the solution.**
3. Homework is your chance to practice and learn the material and it may even be somewhat fun in this course. Much of this course is coding via diagrams.

Activity 0: First Day “Quiz”

This course, Theory of Computation, will introduce you to the tools and skills to characterize types of problems as “easy” or “hard” or “impossible to find a solution in my lifetime”. These problems are classified relative to the computational model (abstract machine) that is necessary to “solve” them. To get your brains warmed up after the summer, think about the following problems. For each, indicate what kind of machine would be necessary to solve that problem.

The machines include (increasing by processor speed and memory size):

- Calculator (easiest)
- Cell phone
- Desktop PC
- Parallel Supercomputer (hardest)

Problem 1

Input: Airport Name

Output: Yes if # of e’s in the name is greater than the # of a’s in the name. Otherwise, answer is no.

Example: Input: Portland, Output: No
 Input: Los Angeles, Output: Yes

Type of machine: _____

Problem 2

Input: Two airports, pricing schemes, and schedules of all airlines

Output: Cheapest flight option (direct flight need not be cheapest)

Example: Input: Portland, OR and Orlando, FL
 Pricing schemes for all airlines, routes of all airlines,
 Restrictions on pricing for all airlines

 Output: \$392.10 on United going through San Francisco,
 Chicago on flight classes U and T

Type of machine: _____

Problem 3

Input: Two airports, distances between any two airports in the world, airline routes

Output: Shortest flight plan in terms of distance

Example: Input: Portland, OR and Modesto, CA
 Output: Shortest flight plan: Portland to San Francisco, San
 Francisco to Modesto (760 miles)

Type of machine: _____

Problem 4

Input: Airport name

Output: Yes, if it has the substring "or" in it. Otherwise, no.

Example: Input: Portland Output: Yes
 Input: Los Angeles Output: No

Type of machine: _____

Problem 5

Input: Entire airline route map

Output: Yes, if there is a tour. Otherwise, no. A tour consists of visiting every airport and no airport is visited more than once.

Type of machine: _____

Problem 6

What is something you want to share about yourself with the class (hobby/skill)? _____

In this course, we will **study** and **classify** problems according to what type of computational model is necessary to solve them.

For these problems above, the correspondence between machines and computational models is:

Calculator -> Finite State Machine (Regular Languages)
Cell phone -> Pushdown Automata (Context-free Languages)
Desktop computer -> Turing Machine (Decidable Languages)

Discrete Math and Data Structures Review

This course builds from material in prerequisite courses, especially discrete math (MTH 311) and data structures (CS 305). Review your prior course materials and/or the resources posted to Moodle. Below are the essential topics that we will use from these courses. Add your own notes to the topics below.

Set: unordered collection of elements

Symbol: Denoted by curly braces, such as $\{a, b, c\}$

Tuple: ordered collection of elements

Symbol: Denoted by parentheses, such as (x, y)

Cartesian product: cross-product to create items from multiple sets

Symbol: Denoted by \times

Graph: ordered collection containing a set of vertices (also called nodes) and a set of edges

Symbol: As a picture, vertices are circles and edges are lines/arrows between vertices

Can be directed or undirected

Commonly used algorithms on graphs: breadth-first search, depth-first search, connectivity, shortest path

Tree: data structure with nodes where each node has exactly one parent and zero or more children; often represents hierarchy

Symbol: As a picture, nodes can be circles and connections to children nodes can be arrows

Stack: linear data structure where elements are organized in a last-in-first-out (LIFO) order with two main operations: push and pop

Function: mapping from domain (input set) to range (output set)

Symbol: Denoted by $F: D \rightarrow R$, where D is the domain set and R is the range set

String: finite sequence of symbols from an alphabet (order matters)

Symbol: written as the array of characters from the alphabet

Alphabet denoted by Σ

Language: set of strings (not be confused with a programming language)

Proof Techniques: direct, indirect, contradiction, induction (know these types of proof; in this course, we will mostly be doing direct proofs (constructing machines), proofs by contradiction, and induction proofs. Note that in CS, we often use strong induction rather than weak induction, since recurrences and recursive structures involve sub-problems of various sizes.

Prerequisites are your foundation – it is important to fill in the holes prior to building new skills and using new knowledge in this course. Try to make a strong “jenga” tower now. Plus, the course is like a “jenga” tower, where all material builds on to one structure.

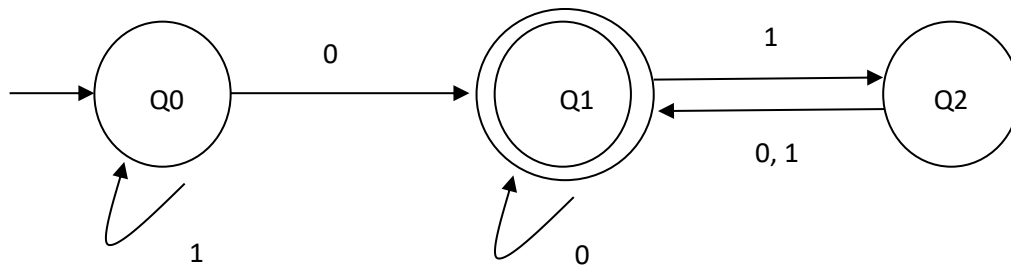
Put your own notes here:



Deterministic Finite Automata (DFA)

DFA: State machine with states (nodes) and transitions (edges) labeled with symbols.

Example DFA M1:



What strings does this machine accept?

What strings does this machine reject?

What is $L(M1)$? (set of strings that M1 accepts)

Formal description of a DFA:

1. Q is the set of states
2. Σ is the alphabet
3. $\delta: Q \times \Sigma \rightarrow Q$ is the transition function
4. q_0 is the start state
5. F (subset of Q) is the set of accepting states

What is Q for $M1$?

What is Σ for $M1$?

What is δ for $M1$?

What is q_0 for $M1$?

What is F for $M1$?

Draw a DFA that accepts the empty set (accepts no strings):

Draw a DFA that accepts just the empty string ϵ :

Draw a DFA that accepts strings with an even number of 0's over the alphabet $\{0, 1\}$.

DFA Example 1: w has odd number of a's and ends with a 'b'

Hint: Think about how to keep track of even/odd length strings.

Hint: Multiple, different DFAs can be correct, just like writing programs.

Try to create the DFA and then watch the video.

DFA Example 2: w contains substring 'baab'

Hint: think about the path through the DFA that strings with the substring 'baab' would need to follow

Try to create the DFA and then watch the video.

Activity 1: Practice with DFAs

Part 1: Determining the language for a DFA

1. Consider M1 below.

$M1 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$ where

δ :

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

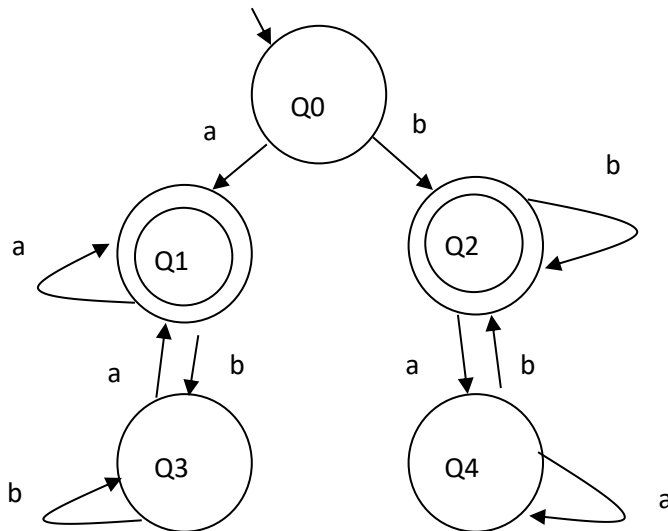
A. First, draw the machine as a state diagram here:

B. What language does M1 recognize?

$L(M1) = \{ \quad \quad \quad \}$

2. Consider M2 below.

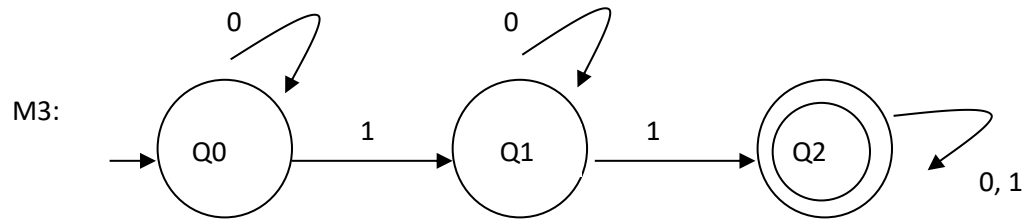
M2 =



A. What is M2's formal description as a 5-tuple? ()

B. $L(M2) = \{ \quad \quad \quad \}$

3. Consider M3 below.



A. $L(M3) = \{ \quad \quad \quad \}$

Part 2: Designing DFAs

1. $L1 = \{w \mid w \text{ has even length}\}$ over $\Sigma = \{0, 1\}$

Hint: keep track of even/odd length by going between states

2. $L2 = \{w \mid w \text{ contains the substring } 010\}$ over $\Sigma = \{0, 1\}$

Hint: must see 010 consecutively

3. $L3 = \{w \mid w \text{ starts with an 'a' and has at most one 'b'}\}$ over $\Sigma = \{a, b\}$

Extra space for notes:

Regular Operators

$A \cup B$	union	$\{x \mid x \in A \text{ or } x \in B\}$
$A \text{ (dot) } B \text{ or } AB$	concatenation	$\{xy \mid x \in A \text{ and } y \in B\}$
A^*	star	$\{x_1x_2x_3x_4\dots x_k \mid k \geq 0 \text{ and } x_i \in A\}$

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

(Translation: If A is regular and B is regular, $A \cup B$ is regular)

Proof idea: We'll construct M by using the following:

1. $Q = \{(r1, r2) \mid r1 \text{ is in } Q1 \text{ and } r2 \text{ is in } Q2\}$
2. $\Sigma \text{ is } \Sigma1 \cup \Sigma2$
3. $\delta((r1, r2), a) = (\delta1(r1, a), \delta2(r2, a))$ note: if the alphabet is not common for A and B, then the transition for any symbol not included in the machine just goes into a forever reject state
4. $q0 \text{ is } (q1, q2)$
5. $F = \{(r1, r2) \mid r1 \text{ is in } F1 \text{ or } r2 \text{ is in } F2\}$

Let's try one:

$A = \{w \mid w \text{ starts with an 'a'}\}$

$B = \{w \mid w \text{ has an even number of 'b's'}\}$

Draw DFA that accepts A over alphabet {a,b}:

Draw DFA that accepts B over alphabet {a,b}:

Using the proof idea, construct the DFA for $A \cup B$:

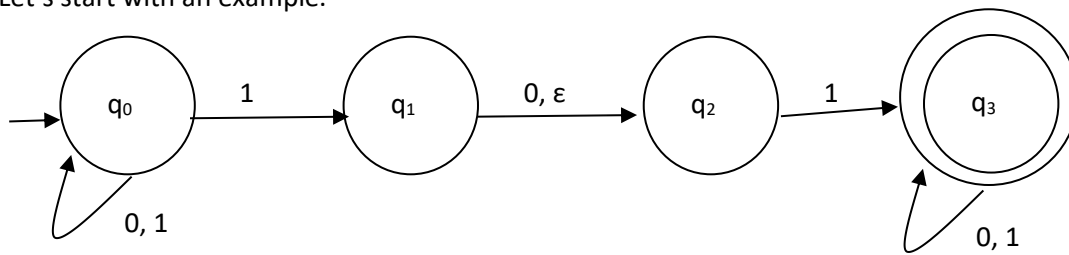
Theorem: The class of regular languages is closed under concatenation.

Ideas for proving this?

(detour: NFAs)

NFA – Nondeterministic Finite Automata

Let's start with an example.



Formal Description of NFA: 5-tuple

$(Q, \Sigma, \delta, q_0, F)$

Only difference with DFA is

$\delta: Q \times \Sigma(\text{with } \epsilon) \rightarrow P(Q)$ power set of Q
(since nondeterminism lets us to be in multiple states at once)

Model of computation for NFA:

Suppose N is an NFA $= (Q, \Sigma, \delta, q_0, F)$.

Suppose $w = w_1w_2w_3\dots w_n$.

N accepts w if there is a sequence of states $r_0r_1r_2\dots r_n$ such that:

1. $r_0 = q_0$ // starts in start state
2. $r_{i+1} \in \delta(r_i, w_{i+1})$ // next state is in set of states that follow transition(s)
3. $r_n \in F$ // last state is an accept state

(similar to DFA, but now property 2 just has to be in the set of states for the machine)

What are the tuple pieces for the NFA above?

$Q =$

$\Sigma =$

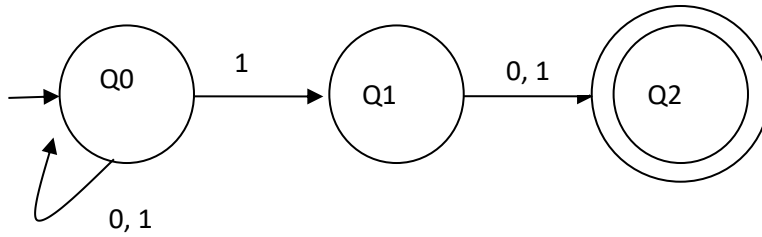
$\delta =$

$q_0 =$

$F =$

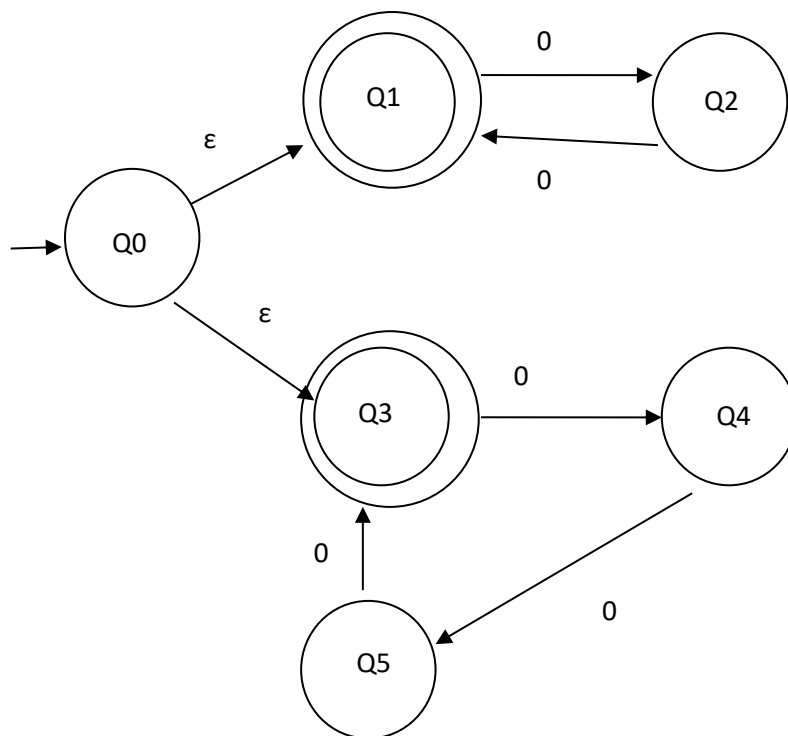
What does M2 recognize?

M2 =



What does M3 recognize?

M3 =



NFA Example 1: w ends with 'aa' over alphabet $\{a, b\}$

Hint: it can be done in 3 states

Hint: use non-determinism

Try it and then watch the video.

NFA Example 2: w contains substring 'baab'

Hint: think about the path through the NFA for the substring

Hint: is this easier than the DFA for the same language?

Try it and then watch the video.

Activity 2: DFA and NFA Creation Practice

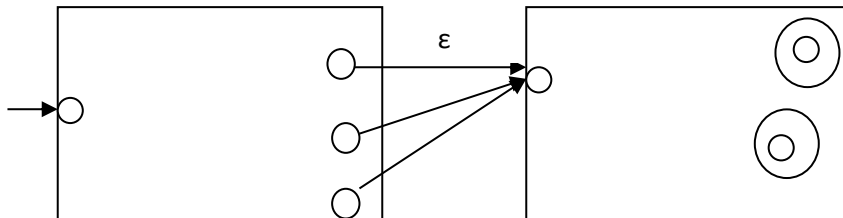
1. $A = \{w \mid w \text{ ends in } 00\}$ over alphabet $\{0,1\}$ with a **3-state NFA**
2. $B = \{w \mid w \text{ contains neither the substrings "ab" nor "ba" over } \{a,b\}\}$ with **DFA**
(Hint: construct for the complement of B and then reverse accept and reject states)
3. $C = \{w \mid \text{length of } w \text{ is at most } 4\}$ with DFA or NFA
4. $D = \{w \mid w \text{ is not the empty string}\}$ with DFA or NFA

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

(Translation: If A is regular and B is regular, AB is regular)

Proof ideas?

Idea: Assume DFAs for A and B exist. Create NFA that nondeterministically chooses when to exit machine for A and start machine for B.

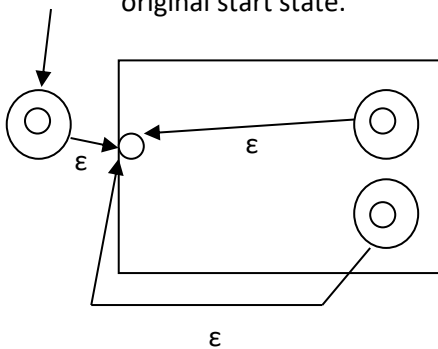


Put ϵ -transitions from accept states for machine M_A to start state of machine M_B . Remove double circles from accept states for A and keep accepts states for B.

Theorem: The class of regular languages is closed under $*$.

Proof ideas?

Ideas? Use nondeterminism to loop back from accept states to the start state. Also, must accept ϵ since A^* is 0 or more copies and 0 copies is ϵ . Add new start state that accepts, and goes on ϵ transition to original start state.



Proof

Assume A is regular and the DFA for A is $M_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$. Construct NFA $N = (Q, \Sigma, \delta, q_0, F)$ such that:

$Q = Q_1 \cup \{q_0\}$ all states in original DFA plus a new start state

$\Sigma = \Sigma_1$

$\delta(q, a) =$

$$\left\{ \begin{array}{ll} \delta_1(q, a) & q \in Q_1 \text{ and } q \text{ not in } F_1 \\ \delta_1(q, a) & q \in F_1 \text{ and } a \text{ is not } \epsilon \\ \delta_1(q, a) \cup \{q_{01}\} & q \text{ in } F_1 \text{ and } a = \epsilon \\ \{q_{01}\} & q = q_0 \text{ and } a = \epsilon \\ \emptyset & q = q_0 \text{ and } a \text{ is not } \epsilon \end{array} \right.$$

q_0 is new start state

$F = \{q_0\} \cup F_1$ all accept states in original DFA plus q_0

Closure Example: If language A is regular, then A_{reverse} is regular.

A^R is the set of all strings in A that are written in reverse.

Remember, to prove that a language is regular, we need to create a DFA or NFA that recognizes the language. For specific languages, we can just construct a specific DFA or NFA with specific states and transitions. However, in this proof, we need to build a DFA or NFA from the components of the DFA for language A . Thus, the proof will be written with the formal tuple notation.

It's easier to create the NFA for A^R from the DFA for A .

Draw picture of intuition here:

Proof using tuple notation and symbols:

Converting NFAs to DFAs

Theorem: Given NFA N , it has an equivalent DFA M .

Proof: Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA that recognizes language A . We will build $M = (Q', \Sigma', \delta', q_{0'}, F')$ that recognizes A .

For now, let us assume N has no epsilon transitions.

$$Q' = P(Q)$$

$$\Sigma' = \Sigma$$

$$\delta'(R, a) = \text{Union over } r \text{ in } R (\delta(r, a)) \quad // \text{ delta takes set of states } \times \text{ character to set of states}$$

$$q_{0'} = \{q_0\}$$

$$F' = \{R \text{ in } Q' \mid \text{set } R \text{ contains at least one accept state of } F\}$$

OK, this works for NFAs that do not contain epsilon transitions. Now, consider a general NFA that can contain epsilon transitions – how should we modify this?

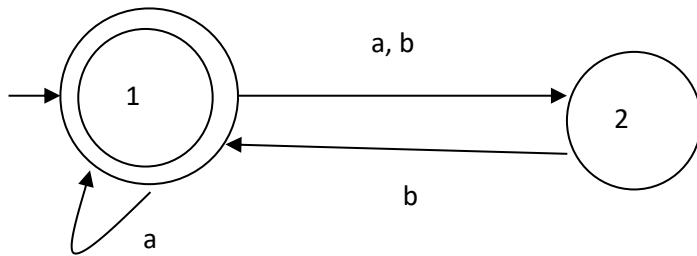
$E(R)$ is the set of states that can be reached from R following epsilon transitions

We need to modify delta and the start state to include these:

$$\delta'(R, a) = \{q \text{ in } Q \mid q \text{ is in } E(\delta(r, a)) \text{ for some } r \text{ in } R\}$$

$$q_{0'} = E(\{q_0\}) \quad // \text{ states reachable from } q_0 \text{ on epsilon } (\epsilon) \text{ transitions}$$

Let's try an example:



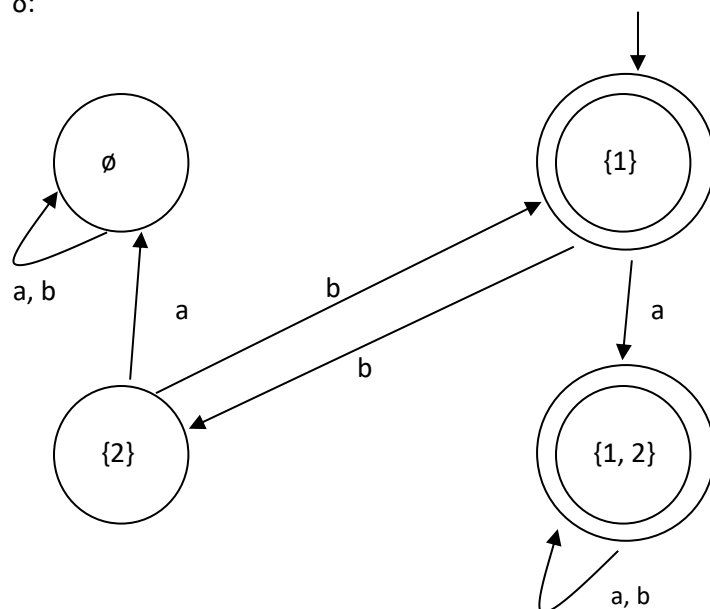
$Q = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$

$\Sigma = \{a, b\}$

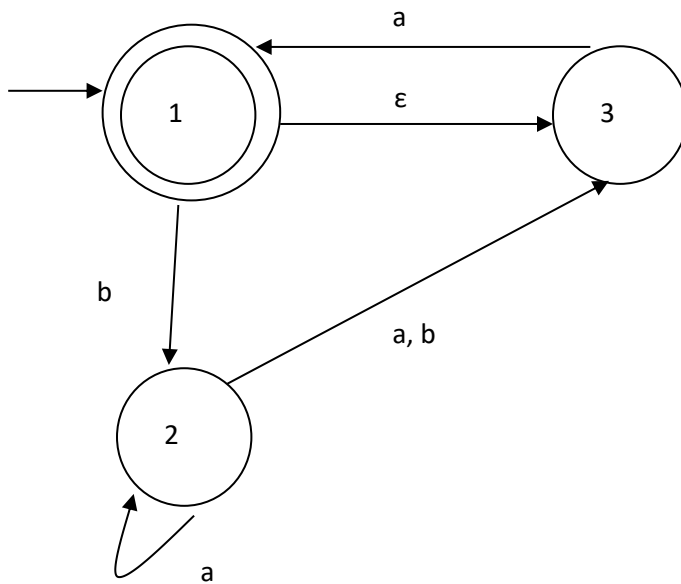
$q_0 = \{1\}$

$F = \{\{1\}, \{1, 2\}\}$

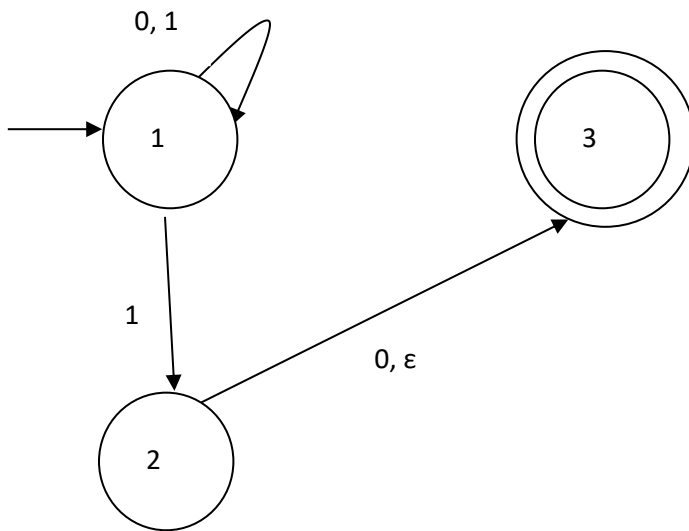
δ :



Example: Convert an NFA with epsilon transitions to an equivalent DFA:



Example: Convert the following NFA to a DFA:



Regular Expressions

Definition of regular expression:

1. $a, a \in \Sigma$
2. ϵ
3. \emptyset
4. $(R1 \cup R2)$ where $R1$ and $R2$ are regular expressions
5. $(R1 \text{ (dot) } R2)$
6. $(R1^*)$

Shorthand:

- Σ all strings of length 1 over Σ
- Σ^* all strings over Σ
- Σ^*1 all strings that end in 1
- $0\Sigma^*$ all strings that start with 0
- R^+ means 1 or more concatenations of strings in R
- R^k means k concatenations of R

Precedence of operators: $*$, dot, \cup

Examples

- $\Sigma^*1\Sigma^*$ $\{w \mid w \text{ contains at least one } 1\}$
- 0^*10^* $\{w \mid w \text{ contains exactly one } 1\}$
- $1^*(01^+)^*$ $\{w \mid \text{every } 0 \text{ in } w \text{ is followed by a } 1\}$
- $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$ $\{w \mid w \text{ starts and ends with same symbol}\}$

Examples for Creating Regular Expressions

Try creating regular expressions for the following languages over the alphabet $\{0, 1\}$ and then watch the video.

$\{w \mid w \text{ starts with a '0' and ends with a '1'}\}$

$\{w \mid w \text{ contains at least three 1's}\}$

$\{w \mid w \text{ contains exactly three 1's}\}$

$\{w \mid \text{every odd position of } w \text{ is 1, where the first character is in the odd position}\}$

Activity 3: Practice with regular expressions

Part 1

Practice: What languages do these regular expressions generate?

1. $01 \cup 10$
2. $(\Sigma\Sigma)^*$
3. $(0 \cup \epsilon)(1 \cup \epsilon)$
4. $1^*\emptyset$
5. \emptyset^*

Part 2

Practice: For each language below, give 2 strings in and 2 strings not in the language

- | | | |
|--|-----|---------|
| 1. a^*b^* | in: | not in: |
| 2. $aba \cup bab$ | in: | not in: |
| 3. $\Sigma^*a\Sigma^*b\Sigma^*a\Sigma^*$ | in: | not in: |
| 4. $(aaa)^*$ | in: | not in: |
| 5. $a(ba)^*b$ | in: | not in: |
| 6. $a+b^*a+$ | in: | not in: |

Part 3

Find the regular expression for the following languages over $\{0,1\}^*$:

1. $\{w \mid w \text{ has the substring } 010\}$
2. $\{w \mid \text{every odd position of } w \text{ is } 0, \text{ where positions are numbered } 1, 2, 3, 4, \text{ etc.}\}$
3. $\{w \mid w \text{ has odd length}\}$
4. $\{w \mid w \text{ begins with } 0 \text{ and ends with } 01\}$
5. $\{w \mid w \text{ does not contain the substring } 010\}$ // note this one is hard

Space for notes:

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

Lemma A: If a language is described by a regular expression, then it is regular.

Lemma B: If a language is regular, then it is described by a regular expression.

How to show these?

A: Start with regular expression, create NFA (which we already showed can be converted to DFA)

B: Start with DFA, create regular expression

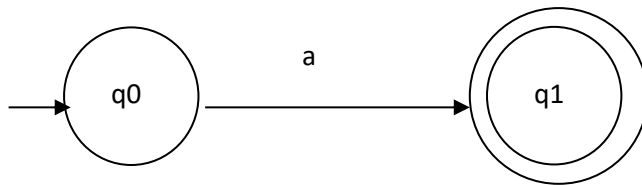
Let's convert from a regular expression to an NFA first:

If we have a regular expression, how should we build an NFA to represent it?

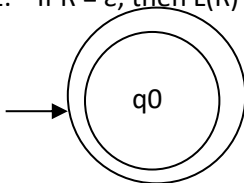
Proof: Let R be a regular expression. For any R , we will show how to generate an equivalent NFA.

Consider the 6 cases:

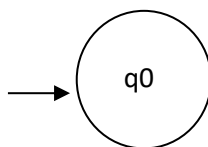
1. if $R = a$ for some a in Σ , then the following NFA recognizes A



2. If $R = \epsilon$, then $L(R) = \{\epsilon\}$



3. If $R = \emptyset$, then $L(R) = \emptyset$



4. $R = R1 \cup R2$ (use construction that we did in proof for \cup)
5. $R = R1 \cdot R2$ (use construction that we did in proof for concatenation)
6. $R = R1^*$ (use construction that we did in proof for $*$)

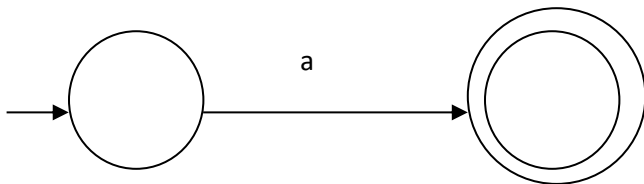
Thus, for every possible regular expression (and combinations of regular expressions), we can generate an NFA.

Example of converting regular expression to NFA:

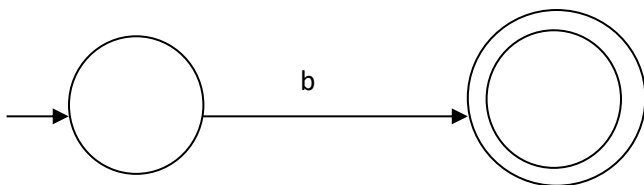
$(ab \cup b)^*$

Let's convert this to an NFA:

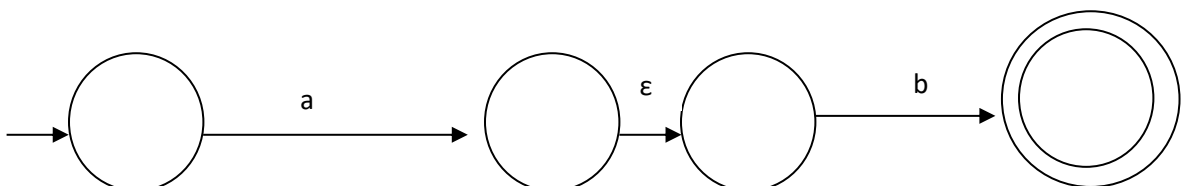
a



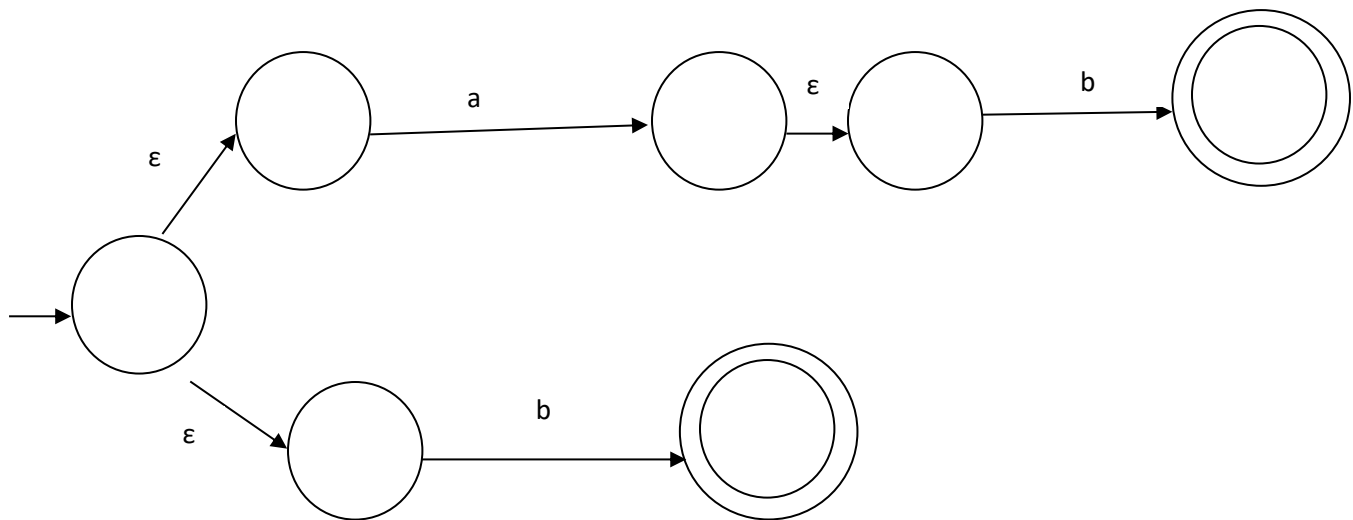
b



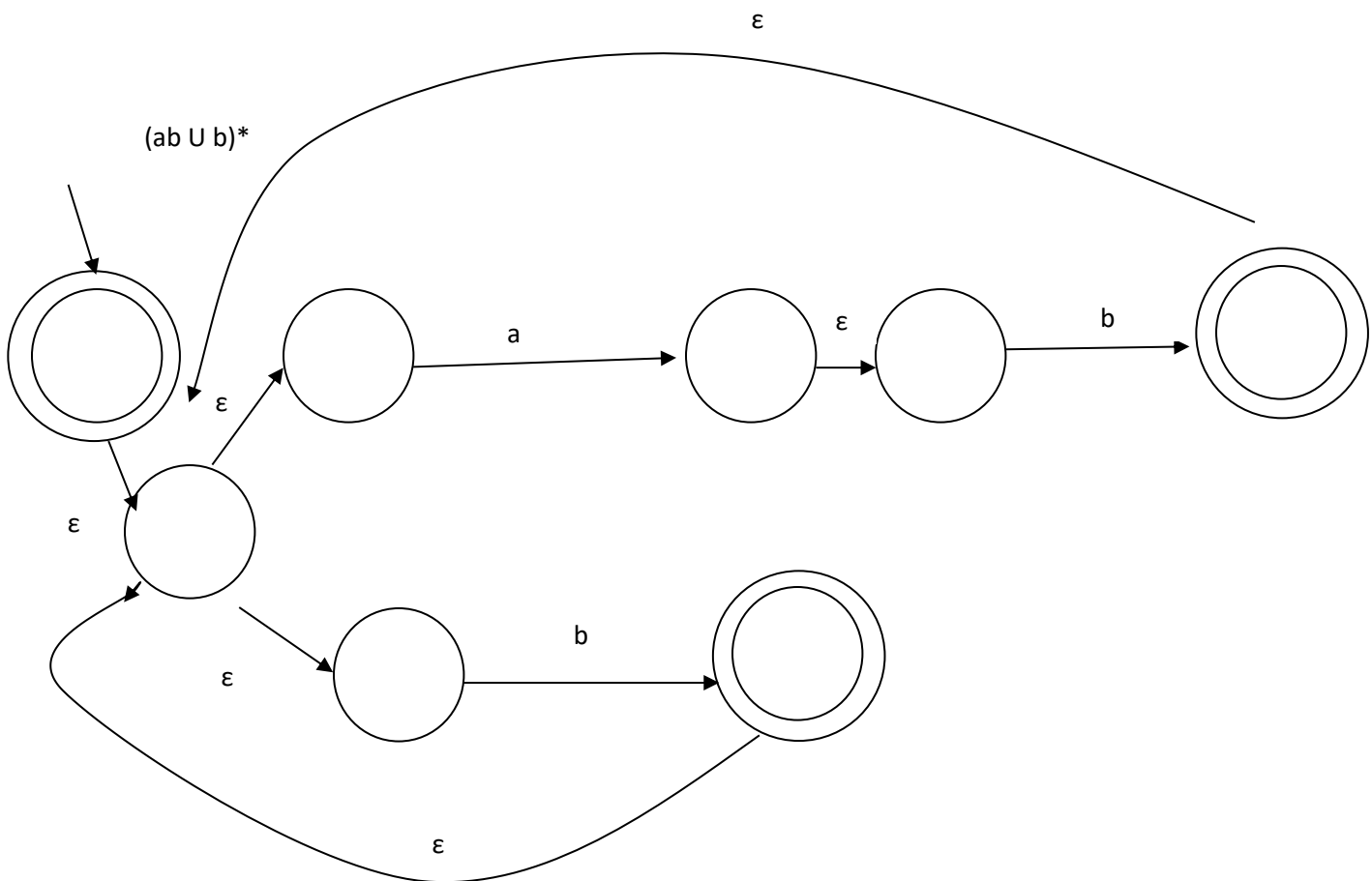
ab



$ab \cup b$



$(ab \cup b)^*$



Example to convert regex to NFA: $a(aab)^* \cup b$

Try to convert $a(aab)^* \cup b$ to an equivalent NFA using the technique.

Hint: start with the smallest parts, the NFA for 'a' and the NFA for 'b'. Create NFA for 'ab'. Then create the NFA for abb .

Try it and then watch the video.

Activity 4: Converting regex to NFA

1: Create the NFA for the following regex:

$a^* \cup (ab)^*b$

Activity 4 continued

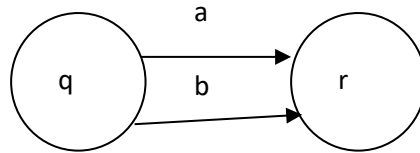
2: Create the NFA for the following regex:

$(a \cup b)a^*b^*$

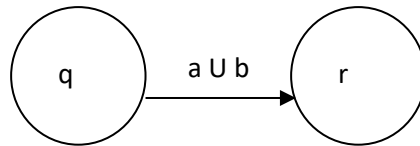
Converting DFA to a regular expression

What we'll do is collapse the DFA by following two rules:

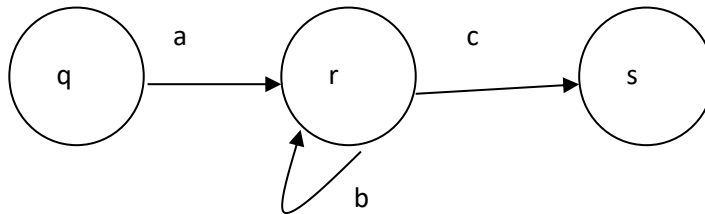
1. If there are multiple edges from state q to state r , then create a union of the edge symbols



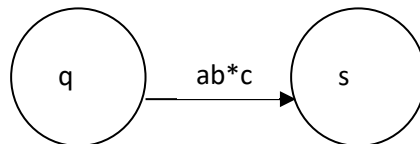
becomes



2. If there is a state $q \rightarrow r \rightarrow s$ where r has a loop, we'll replace it by $q \rightarrow s$ and replace edge with

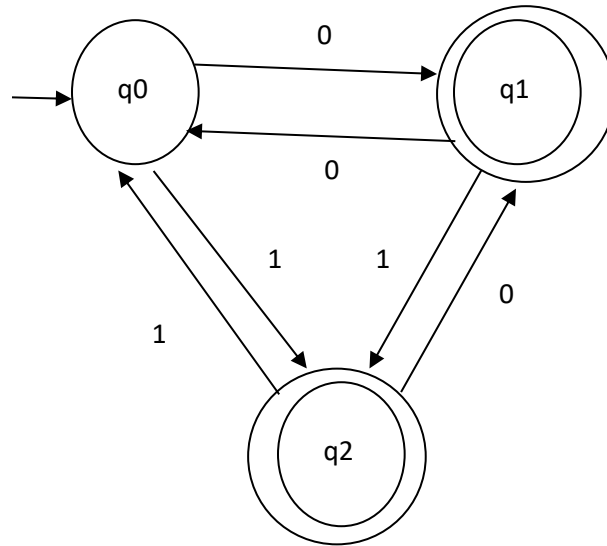


becomes

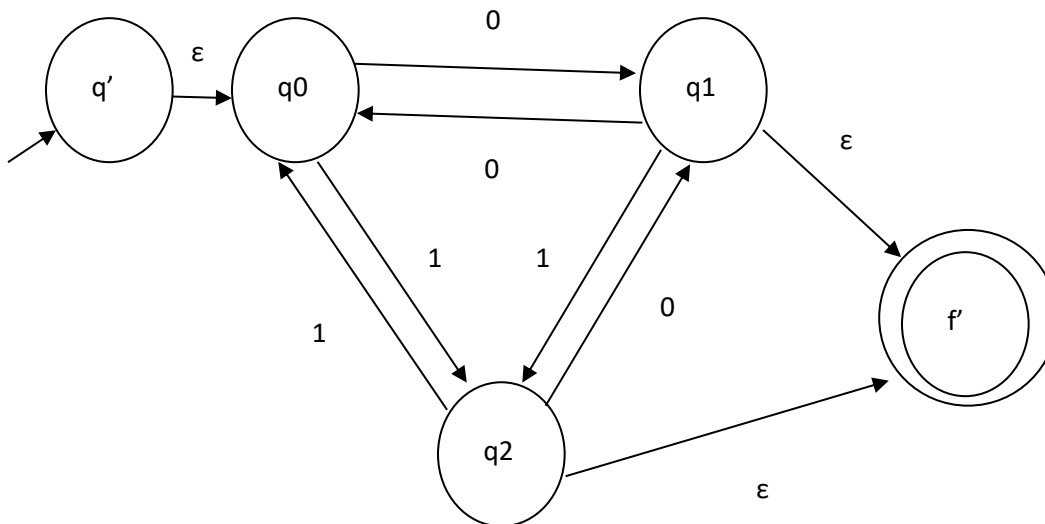


Example of converting DFA to regex:

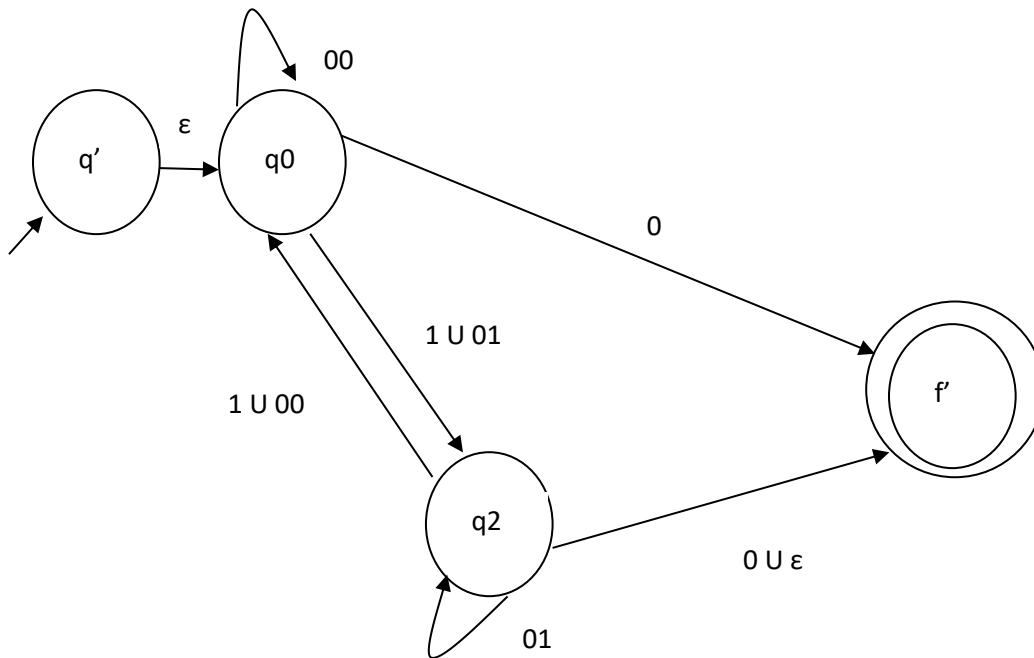
Assume DFA D is:



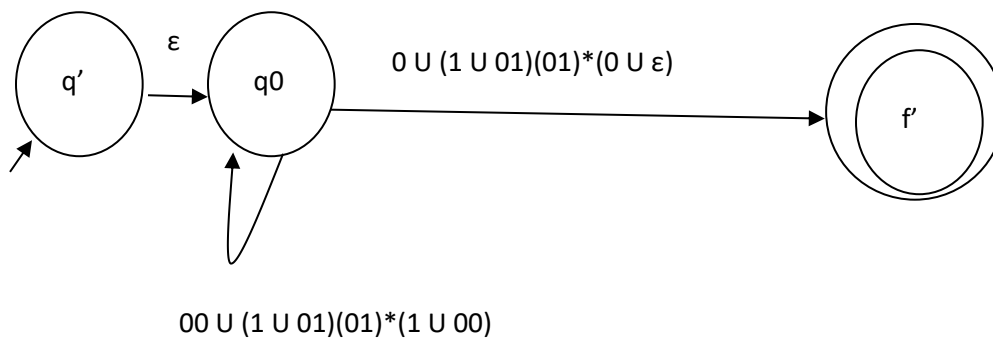
We add a new start state and new accept state to D to create a generalized NFA called G:



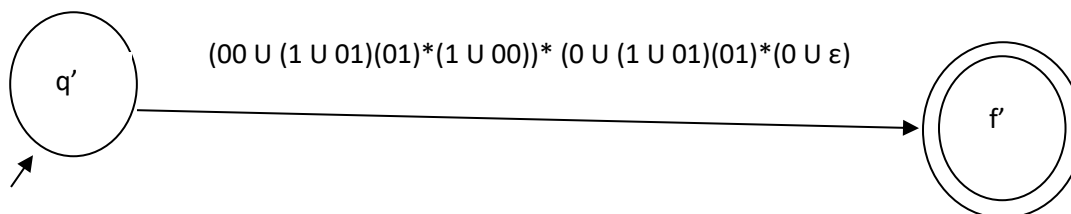
Now, we start ripping out states. Let's rip q_1 :



Now, let's rip out q2:



Now, let's rip out q0:



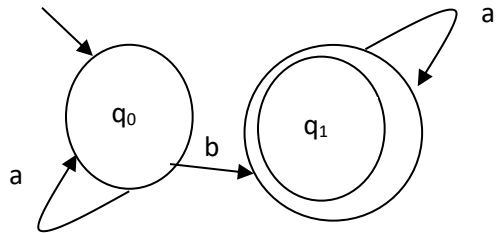
DFA to Regex Example: w does not contain the substring 010

Hint: First, create a DFA for strings that contain 010. Then, create the DFA for the complement of that language.

Try doing this and then watch the video.

Note that there should be a union epsilon at the end of the final regex. The video was cut short.

Activity 5: Practice converting a NFA->DFA and then DFA->regex



1. First, convert NFA above to an equivalent DFA:

2. Then, create the GNFA (new start state and new single accept state).

3. Then, rip a state:

4. Rip another state:

Summary (Regular Languages)

A language is regular if a DFA recognizes it (definition).

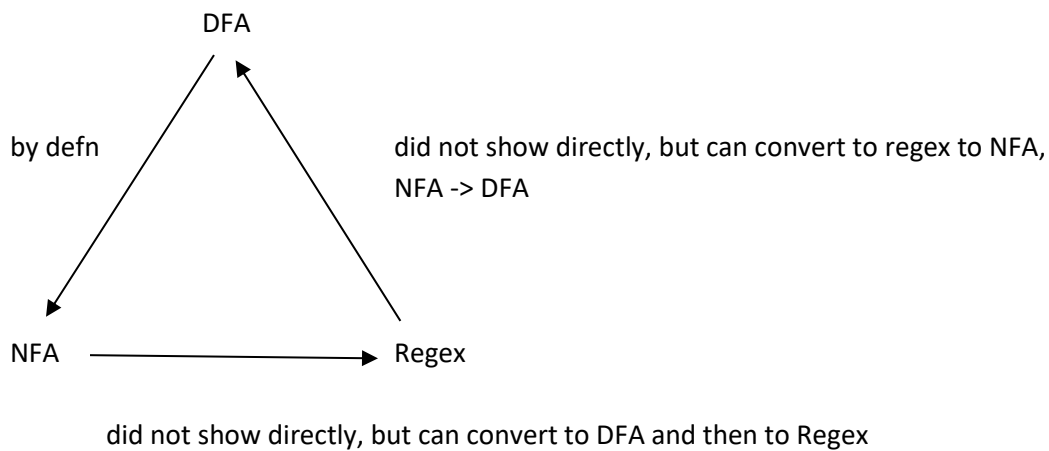
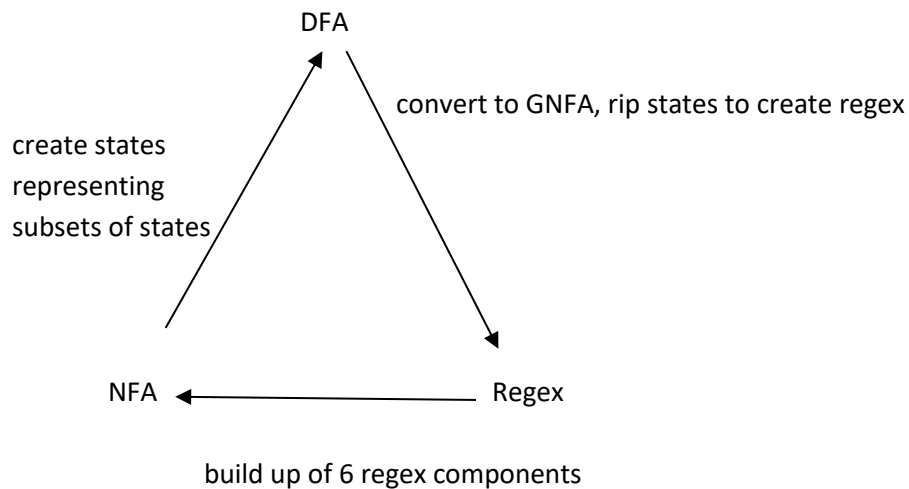
A language is regular if and only if an NFA recognizes it.

DFA is an NFA, NFA \rightarrow DFA

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

DFA \rightarrow regex, regex \rightarrow NFA

The class of regular languages is closed under U (union), dot (concatenation), and $*$.



CS357 Review Sheet – Midterm Exam #1

You may use 1 crib sheet as notes during the exam. All other resources are off limits – you are on your honor to follow these exam rules.

Content: Exam 1 will cover sections 0.1 through 1.3 of the textbook. Material will be drawn from homework assignments, lectures, and the textbook.

Procedure: Please arrive to class on time. You may use **one** sheet of 8.5" x 11" paper (**both sides**) during the exam. Other than your sheet of notes, the exam is closed-book, closed-calculator, closed-computer other than the moodle submission links, and closed-notes. All computations will be simple enough for you to do by hand.

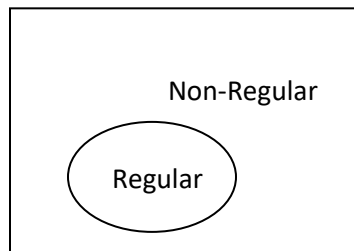
Topics: This study guide is not a contract – in other words, the exam may not cover every topic listed below and there may be topics that we covered in class that are not explicitly listed.

- Discrete Math Review
 - Sets
 - Sequences (Tuples)
 - Functions and Relations
 - Graphs
 - Strings and Languages
 - Boolean Logic
 - Proofs
 - Direct
 - Indirect
 - Contradiction
 - Induction
 - Construction
- Regular Languages (those that can be recognized by DFAs, NFAs, or written as regular expressions)
- DFAs
 - Given a language, construct the DFA
 - Given a DFA, state the language it recognizes
 - Formal definition as a tuple
- Union, Concatenation, *
 - Closure of regular languages under U, concatenation, and * (know the proofs)
- Closure of regular languages under other operations, such as reverse and perfect shuffle
- NFAs
 - Given a language, construct the NFA
 - Given an NFA, state the language it recognizes
 - Converting NFAs to DFAs
 - Formal definition as a tuple
- Regular Expressions
 - Given a regular expression, state its language
 - Given a language, create a regular expression

- Converting regular expressions to NFAs
- Converting DFAs to regular expressions

Question to ponder: Can you create a DFA or NFA for the language $L = \{0^n 1^n \mid n \geq 0\}$?

What would you need to do this?



How would we show that a language is non-regular?

Pumping Lemma for Regular Languages

Pumping Lemma for Regular Languages:

If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p , then s may be divided into 3 pieces $s = xyz$ such that:

1. For each $i \geq 0$, $xy^iz \in A$
2. $|y| > 0$
3. $|xy| \leq p$

In other words, arbitrary copies of y concatenated in the middle results in strings that are still in the language A .

Formal proof

Let A be a regular language that is recognized by DFA $M = (Q, \Sigma, \delta, q_0, F)$. Let $p = |Q|$.

Let $s = s_1s_2s_3\dots s_n$ be a string that is in A where $n \geq p$. Let $r_1, r_2, \dots, r_n, r_{n+1}$ be the set of states that are entered when executing M on s . Because $n \geq p$, then $(n+1) > p$, so there must be some state that is repeated in r_1, r_2, \dots, r_{n+1} . Let r_j be the first such repeated state in the list and let r_k be the second instance of the repeated state in the list.

Let $x = s_1 \dots s_{j-1}$

Let $y = s_j \dots s_{k-1}$

Let $z = s_k \dots s_n$

[condition 1] M must accept xy^iz for $i \geq 0$ since s is accepted by M and the y part can be repeated 0 or more times.

[condition 2] The string y has length > 0 since r_j and r_k appear as separate entries in the sequence of states entered. Thus, there is at least one transition from r_j to r_k .

[condition 3] The latest the duplicate states could appear is r_n and r_{n+1} , so $|xy| \leq n$, so $|xy| \leq p$.

Activity 6: Find the minimum pumping lengths

Examples:

0001* // 4 (000 is in but cannot be pumped down, 0001 can be pumped down)
0*1* // 1 (0 can be pumped down, 1 can be pumped down)
0*1+0+1* U 10*1 //3 (11 cannot be pumped, but strings of length 3 can be pumped by either part)
(01)*11 // 3 (no strings of length 3, 0111 can be pumped down, length 3 is “vacuously true”
11(01)* // 4 (no strings of length 3, 1101 can be pumped down, pumped string must come in
// first p characters

What are the minimum pumping lengths of the following languages?

Hint 1: think about the shortest string(s) in the language. They cannot be pumped down.

Hint 2: if there are strings of length 4 that can be pumped up and down, there are no strings of length 3 in the language, and there are strings of length 2 that cannot be pumped down, then the minimum pumping length is 3 due to the vacuously true part of the pumping lemma.

Hint 3: the pumped out or pumped in substring y must come in the first p characters where p is the minimum pumping length

1. $aa^*b \cup abab^*b$ MPL = _____

2. $ba(ab)^* \cup b$ MPL = _____

3. $bb(aaa)^*$ MPL = _____

4. $ab \cup ba$ MPL = _____

5. $bb^* \cup aaa^*$ MPL = _____

6. $babba^*$ MPL = _____

7. $(aaa)^*bb$ MPL = _____

Using the pumping lemma to prove a language is NOT regular:

Assume you are given a language A.

1. Assume A is regular. Thus, the pumping lemma is true for some pumping length p .
2. Choose a string s at least length p that is in A.
3. For all possible decompositions of s into xyz such that y has at least one character and $|xy| \leq p$, show that there is some i for which xy^iz is not in A. Because the string cannot be “pumped”, we have a contradiction. A must not be regular.

Game:

1. Adversary chooses p .
2. You choose a specific string s where $|s| \geq p$.
3. The adversary chooses the decomposition of s into xyz , subject to $|y| > 0$ and $|xy| \leq p$.
4. You choose i in such a way that xy^iz is not in the language. In many proofs, it is common to choose i to be 0 (i.e. remove substring y) or choose i to be 2 (i.e. consider string $xyyz$).

Example of proving a language is not regular

(proof by contradiction)

Show that the language $L_1 = \{0^n 1^n \mid n \geq 0\}$ is not regular.

Assume L_1 is regular. Then, the pumping lemma holds with pumping length p . Choose $s = 0^p 1^p$. Using the pumping lemma, s can be split into three parts $s = xyz$ such that $|xy| \leq p$ and $|y| > 0$. The following cases for the decomposition are as follows:

Case 1: $s = xyz$ where y contains all 0's. Then, $y = 0^k$ where $1 \leq k \leq p$. Consider xy^2z . Because y contains at least one zero, the string xy^2z must have at least $N+1$ zeros before the N ones. Thus, xy^2z is not in L_1 .

Because $|xy| \leq p$, there are no more cases to consider. Because the string $0^p 1^p$ cannot be pumped, the pumping lemma does not hold. We have a contradiction. So, L_1 is not regular.

Practice with scaffolding

Show $L_2 = \{ ww^R \mid w \text{ is in } \{0,1\}^* \}$ is not regular.

Assume L_2 is regular. Then, the pumping lemma holds with pumping length p . Choose $s = \underline{\hspace{2cm}}$.

Using the pumping lemma, s can be split into three parts $s = xyz$ such that $|xy| \leq p$ and $|y| > 0$. The following cases for the decomposition are as follows:

Case 1: $s = xyz$ where y contains $\underline{\hspace{2cm}}$. Consider xy^kz where $k = \underline{\hspace{2cm}}$. Because y contains $\underline{\hspace{2cm}}$, the string xy^kz must $\underline{\hspace{2cm}}$. Thus, xy^kz is not in L_2 .

Case 2: (if needed)

Case 3: (if needed)

Because the string s cannot be pumped for any choice of y , the pumping lemma does not hold. We have a contradiction. So, L_2 is not regular.

Activity 7: Proving Non-regular languages

Show each of the following languages are NOT regular by using the pumping lemma.

$L3 = \{w \mid w \text{ has an equal number of 0's and 1's in any order over } \{0,1\}^*\}$

s =

Cases:

$L4 = \{ww \mid w \text{ is over } \{0,1\}^*\}$

s =

Cases:

$L5 = \{0^i 1^j \mid i > j\}$

s =

Cases:

Space for notes:

Context Free Languages

Generated by a context free grammar (CFG)

Often used to express programming languages and natural languages

Java's grammar is here: <https://docs.oracle.com/javase/specs/jls/se18/html/jls-19.html>

Example of simple grammar:

G1:

$A \rightarrow 0A1$

$A \rightarrow B$

$B \rightarrow \#$

Grammar components:

Variables: A, B

$V = \{A, B\}$

Symbols/Terminals: 0, 1, #

$\Sigma = \{0, 1, \#\}$

A is the start variable

$S = A$

Example:

$S = 00\#11$ is in $L(G1)$

How is S derived?

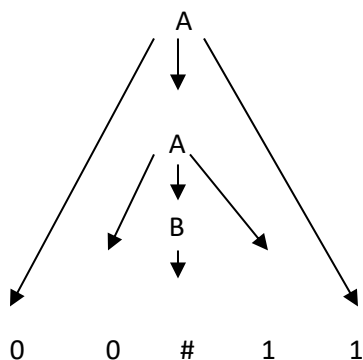
$A \rightarrow 0A1$

→ 00A11

→ 00B11

→ 00#11

Another view as a parse tree



1. What is $L(G_1)$?

Formal definition of a CFG: (V, Σ, R, S)

1. V is the set of variables
2. Σ is the set of terminals (alphabet)
3. R is the set of rules from variables to strings of variables and terminals
4. $S \in V$ is the start variable

$G_2 = (\{S\}, \{a, b\}, R, S)$ where R is $S \rightarrow aSb \mid SS \mid \epsilon$

2. What strings does G_2 generate?

3. What is $L(G_2)$?

$G3 = (V, \Sigma, R, \langle \text{expr} \rangle)$

$V = \{\langle \text{expr} \rangle, \langle \text{term} \rangle, \langle \text{factor} \rangle\}$

$\Sigma = \{a, +, x, (,)\}$

R:

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle x \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle) \mid a$

4. What strings are generated?

Definition An **ambiguous** grammar has two or more parse trees for at least one string in $L(G)$.

Example:

G4:

$E \rightarrow E + E \mid E x E \mid (E) \mid a$

5. Find a string that can be generated in two or more ways with G4.

Grammar Example 1: w has middle symbol 1

Try creating the grammar for this language and then watch the video.

Hint: think about how to elongate the string to keep the 1 in the middle.

Hint: think recursively. What is the base case (shortest string)? What are the recursive cases?

Grammar Example 2: w equals w reverse

Try to create the grammar for this language and then watch the video.

Hint: it needs to work for even-length and odd-length strings.

Hint: think about the base cases (shortest strings)

Hint: think about the recursive cases (how to make longer strings from a given string x in the language)

Theorem: If language A is regular, A is a context free language. (The regular circle is embedded in the CFL circle.)

Proof idea: Need to convert DFA to a CFG.

Each state becomes a variable.

If $d(q_i, a) = q_j$

becomes rule $R_i \rightarrow aR_j$

For accept states:

$R_i \rightarrow \epsilon$

If q_0 is start state, then:

R_0 is start rule

example:

DFA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0, q_2\})$

δ :

	0	1
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_2	q_2
q_3	q_3	q_3

Grammar is:

$R_0 \rightarrow 0R_1 \mid 1R_2 \mid \epsilon$

$R_1 \rightarrow 1R_3 \mid 0R_0$

$R_2 \rightarrow 0R_2 \mid 1R_2 \mid \epsilon$

$R_3 \rightarrow 0R_3 \mid 1R_3$

Activity 8: Construct grammars for languages below
(note: $\Sigma = \{0,1\}$ for all languages below)

$$A = \{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$$

$$B = \{w \mid w \text{ starts and ends with the same symbol}\} // \text{note this one is actually regular}$$

$$C = \{w \mid \text{length of } w \text{ is odd}\} // \text{note this one is actually regular}$$

$$D = \{w \mid \text{length of } w \text{ is odd and middle symbol is } 0\}$$

Converting to CNF (Chomsky Normal Form)

In CNF, all rules have the form:

$A \rightarrow BC$ The rule $S \rightarrow \epsilon$ is permitted, no other rule has ϵ

$A \rightarrow a$

Idea for converting a grammar into CNF:

1. Make a new start rule $S_0 \rightarrow S$
2. For any rule $A \rightarrow \epsilon$, replace occurrences of A on the righthand side of rules with ϵ
3. Remove rules $A \rightarrow B$, $B \rightarrow a$ and replace with $A \rightarrow a$ and remove rules $A \rightarrow A$
4. Remove rules that look like $A \rightarrow abcd$ with rules $A \rightarrow aA_1$, $A_1 \rightarrow bA_2$, $A_2 \rightarrow cA_3$, $A_3 \rightarrow d$
or rules that look like $A \rightarrow ABCD$ with rules $A \rightarrow AA_1$, $A_1 \rightarrow BA_2$, $A_2 \rightarrow CD$
5. Remove rules that look like $A \rightarrow aB$ with rules $A \rightarrow XB$ and $X \rightarrow a$

Example:

G: $S \rightarrow ASA \mid aB$
 $A \rightarrow B \mid S$
 $B \rightarrow b \mid \epsilon$

1. Make new start rule

$S_0 \rightarrow S$
 $S \rightarrow ASA \mid aB$
 $A \rightarrow B \mid S$
 $B \rightarrow b \mid \epsilon$

2. Remove $B \rightarrow \epsilon$ rule

$S_0 \rightarrow S$
 $S \rightarrow ASA \mid aB \mid a$
 $A \rightarrow B \mid S \mid \epsilon$
 $B \rightarrow b$

Remove $A \rightarrow \epsilon$ rule

$S_0 \rightarrow S$
 $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \mid S$
 $A \rightarrow B \mid S$
 $B \rightarrow b$

3. Remove $S \rightarrow S$

$S_0 \rightarrow S$
 $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
 $A \rightarrow B \mid S$
 $B \rightarrow b$

Remove $S_0 \rightarrow S$

$S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
 $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
 $A \rightarrow B \mid S$
 $B \rightarrow b$

Remove $A \rightarrow B$

$S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
 $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
 $A \rightarrow b \mid S$
 $B \rightarrow b$

Remove $A \rightarrow S$

$S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
 $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$
 $A \rightarrow b \mid ASA \mid aB \mid a \mid SA \mid AS$
 $B \rightarrow b$

4. Remove rules that go to three or more terms

$S_0 \rightarrow AA_1 \mid aB \mid a \mid SA \mid AS$
 $S \rightarrow AA_1 \mid aB \mid a \mid SA \mid AS$
 $A \rightarrow b \mid AA_1 \mid aB \mid a \mid SA \mid AS$
 $B \rightarrow b$
 $A_1 \rightarrow SA$

5. Remove rules $A \rightarrow aB$

$S_0 \rightarrow AA_1 \mid UB \mid a \mid SA \mid AS$
 $S \rightarrow AA_1 \mid UB \mid a \mid SA \mid AS$
 $A \rightarrow b \mid AA_1 \mid UB \mid a \mid SA \mid AS$
 $B \rightarrow b$

$A_1 \rightarrow SA$
 $U \rightarrow a$

The resulting grammar is now in CNF (all rules look like $A \rightarrow BC$ or $A \rightarrow a$)

Practice: Convert the following grammar to CNF:

G: $S \rightarrow A \mid \epsilon$
 $A \rightarrow 01 \mid 0A1$

Pushdown Automata (PDA)

It's like an NFA with a stack.

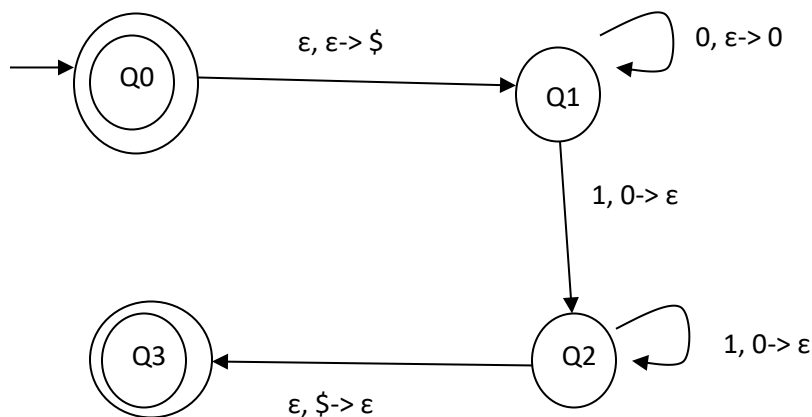
Read input, go through states (but machine also can push/pop symbols from a stack).

Push – write symbol

Pop – read symbol and remove from stack

Example: PDA for $\{0^n 1^n \mid n \geq 0\}$

Idea: use stack to store 0's it has seen. When you start reading 1's, start popping 0's off the stack. If stack is empty when entire string is read, then we accept.



Formal Definition:

$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$

Q = set of states

Σ = input alphabet

Γ = stack alphabet

$\delta: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)$

// transition takes state, input, and stack \rightarrow set of states

// (can be in several states after transition)

q_0 is start state

F is set of accept states

Definition of computation:

1. M is a PDA. M starts in q_0 with an empty stack
2. M follows transitions given input symbol, state and stack symbol
3. If M can finish reading the input string and end in an accept state, the string is accepted.

Let's see what happens when the above PDA runs on $w = 000111$.

What does delta transition table look like?

Transition table δ for the PDA for $\{0^n 1^n \mid n \geq 0\}$. Blanks represent the empty set to keep the table from getting too cluttered.

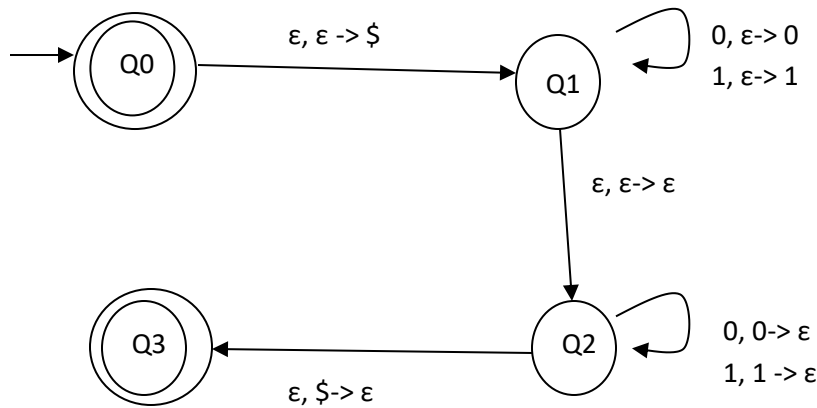
δ :

Input Stack	0			1			ϵ			
	0	\$	ϵ	0	\$	ϵ	0	\$	ϵ	
q_0									$\{(q_1, \$)\}$	
q_1			$\{(q_1, 0)\}$	$\{(q_2, \epsilon)\}$						
q_2				$\{(q_2, \epsilon)\}$				$\{(q_3, \epsilon)\}$		
q_3										

Another example of a PDA:

$\{ww^R \mid w \in \{0,1\}^*\}$

Idea: push symbols of input, nondeterministically guess middle, start popping read symbols



Try operating on 110011.

More on the stack part of the transition:

$\epsilon \rightarrow 0$ Push 0

$0 \rightarrow \epsilon$ Pop 0 (means stack must have 0 on top, if it does not, cannot follow this transition)

$\epsilon \rightarrow \epsilon$ Keep stack the same

$0 \rightarrow 1$ Replace 0 with 1 (really just shorthand for pop 0, push 1)

$0 \rightarrow 01$ Read 0 on top, push 1

$0 \rightarrow 00$ Read 0 on top, push 0 (really just shorthand for pop 0, push 0, push 0)

$\Gamma \rightarrow \epsilon$ Pop any symbol from stack

PDA Example 1: w has middle symbol 1

Try to create the PDA for this language and then watch the video.

Hint: our memory in a PDA is a stack. How can you use the stack to get to the middle of the string?

PDA Example 2: w equals w reverse

Try to create the PDA for this language and then watch the video.

Hint: our memory in a PDA is a stack. How can you use the stack to ensure the string is a palindrome?

Activity 9: Construct PDAs

Create PDAs for the following languages:

$A = \{w \mid w \text{ has more } a\text{'s than } b\text{'s over } \{a,b\}^*\}$

Hint: Use the stack to show difference of # a's versus b's or difference of # b's versus a's

$B = \{w\#x \mid w^R \text{ is a substring of } x, \text{ where } x \text{ and } w \text{ are over } \{0,1\}^*\}$

Hint: Use the stack to push w ; ND pop from stack to guess substring portion

Converting CFG to PDA (flower power)

General idea for PDA:

1. Place \$ on stack, Place S on stack
2. While(true):
 - a. If top of stack is variable A, nondeterministically select a rule for A and substitute
 - b. If top of stack is a and input is a, keep processing stack by reading a from string and popping a from stack. Otherwise, reject this branch.
 - c. If top of stack is \$, accept.

Assume A is context free given by grammar $G = (V, \Sigma, R, S)$. Construct PDA $M = (Q, \Sigma', \Gamma, \delta, q_0, F)$ as follows:

$Q = \{Q_{start}, Q_1, Q_{loop}, Q_{accept}\}$

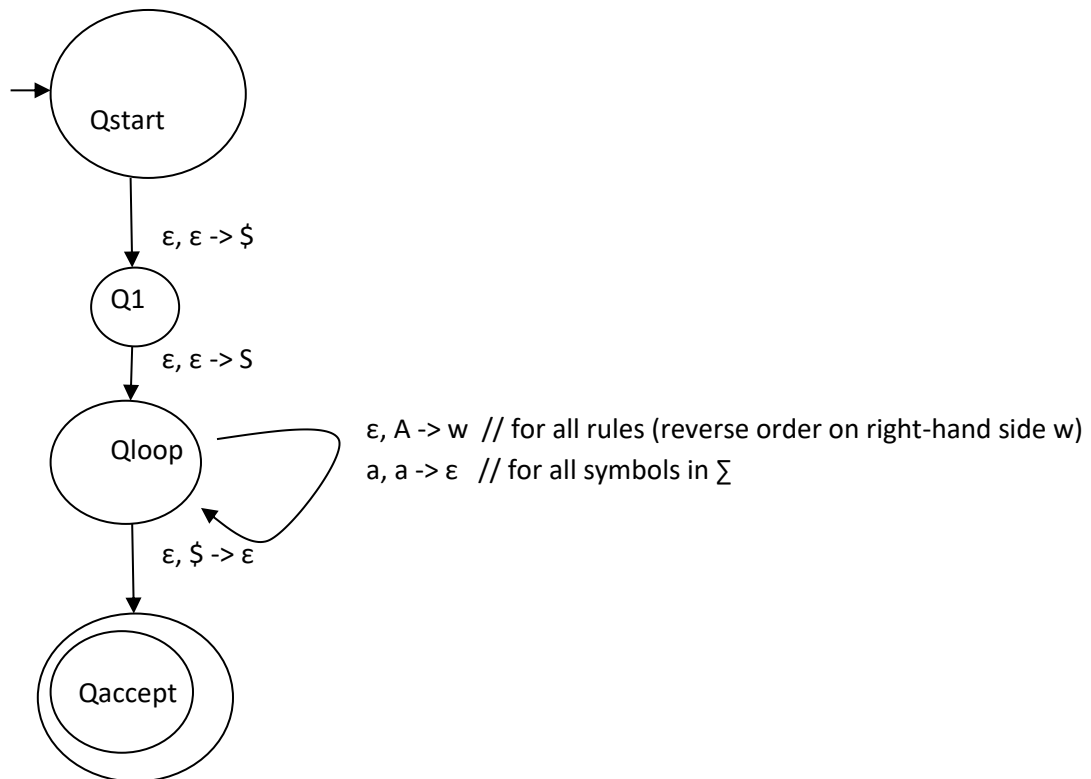
$\Sigma' = \Sigma$

$\Gamma = \Sigma \cup \{\$\}$ U V

δ (see below)

$q_0 = q_{start}$

$F = \{q_{accept}\}$

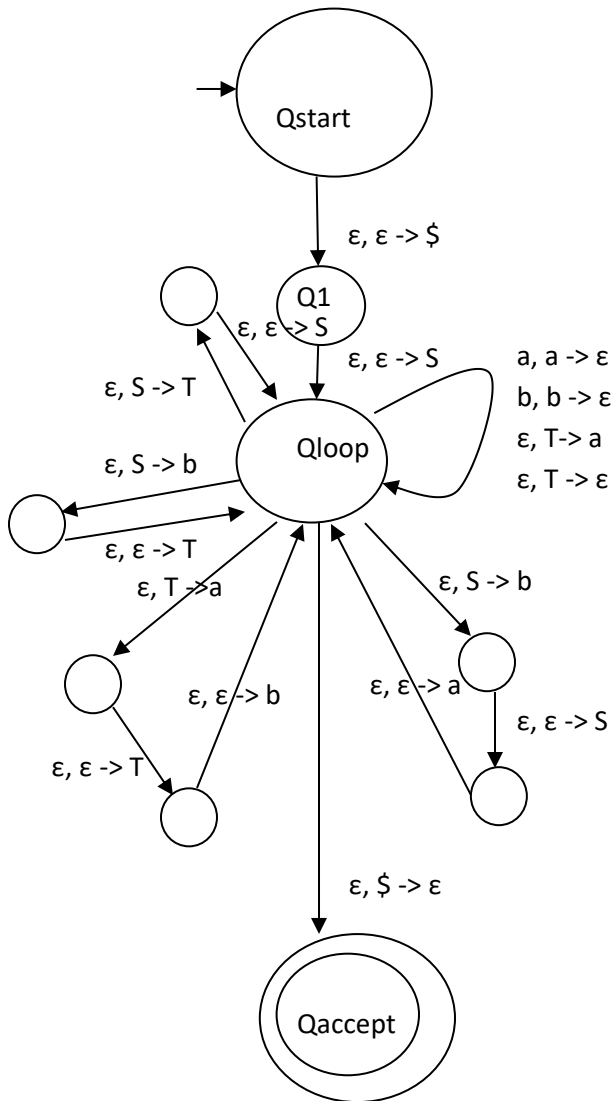


Example to convert grammar to PDA

$S \rightarrow aSb \mid ST \mid Tb$

$T \rightarrow bTa \mid a \mid \epsilon$

There are six rules, so we will have six petals (some shared) around Qloop. The alphabet (terminals) are $\{a, b\}$, so we will have a rule that reads the symbol and pops the symbol in the alphabet.



Let's see how this PDA accepts aabb:

		a		T	a					
		S	S	b	b	b				
	S	b	b	b	b	b	b			
Stack:	\$	\$	\$	\$	\$	\$	\$	\$		empty
			read a		read a	read b	read b			

CFG->PDA Example: Converting grammar to equivalent PDA

CFG:

$A \rightarrow aAb \mid Bb \mid \epsilon$

$B \rightarrow aB \mid a$

Create flow diagram for equivalent PDA:

Remember: the rules need to be pushed to the stack from right to left.

Activity 10: Practice converting grammar to PDA

Use the conversion technique (flower power) to convert the following grammar to an equivalent PDA.

$S \rightarrow aaSC \mid Cab \mid \epsilon$

$C \rightarrow aCb \mid Da$

$D \rightarrow bD \mid b$

Converting PDAs to CFGs

Theorem Assume A is $L(M)$ where M is a PDA. Then, there is a context free grammar G that generates A .

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

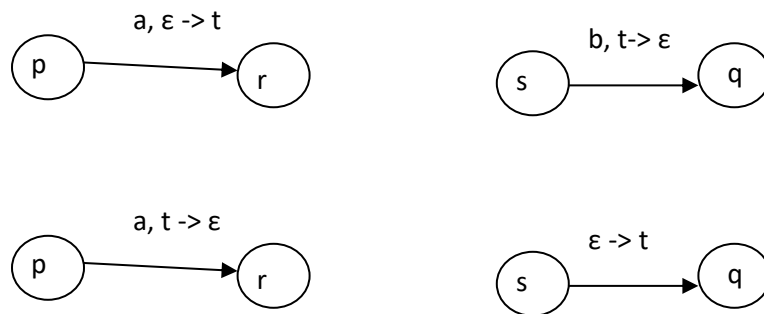
Idea:

We'll transform the PDA:

1. It has one accept state q_f
2. It empties its stack before accepting
3. Each transition either pops or pushes a symbol (no places where the stack stays the same)

To convert to grammar:

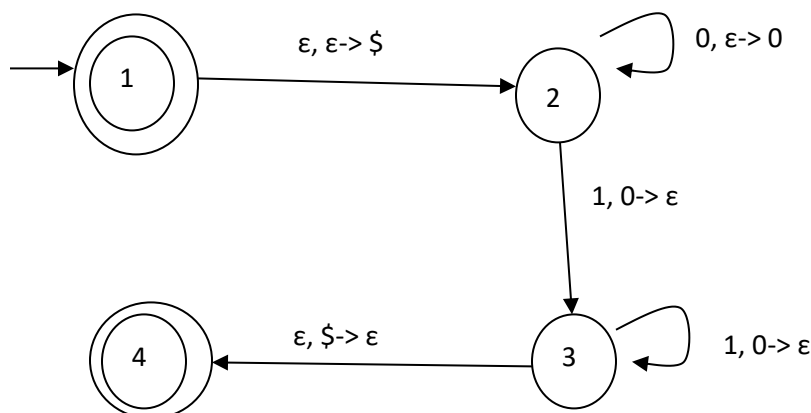
1. Variables A_{ij} will represent paths from state i to state j
2. $A_{pq} \rightarrow aA_{rs}b$ if p goes to r while reading a and pushing t AND r goes to s while reading b and popping t
OR $A_{pq} \rightarrow aA_{rs}b$ if p goes to r while reading a and popping t AND r goes to s while reading b and pushing t // legal moves through PDA that preserve stack



3. Start variable represents going from start to q_f . // must get all the way from start to accept
4. $A_{pq} \rightarrow A_{pr}A_{rq}$ for all p, q, r as states in PDA // can replace path with two adjoining paths in PDA
5. $A_{pp} \rightarrow \epsilon$ for all p in the states in PDA // can stay in state without reading anything

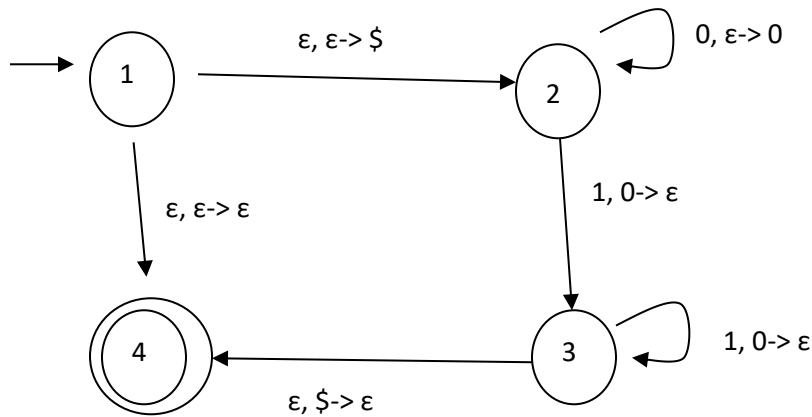
Example

PDA for 0^n1^n



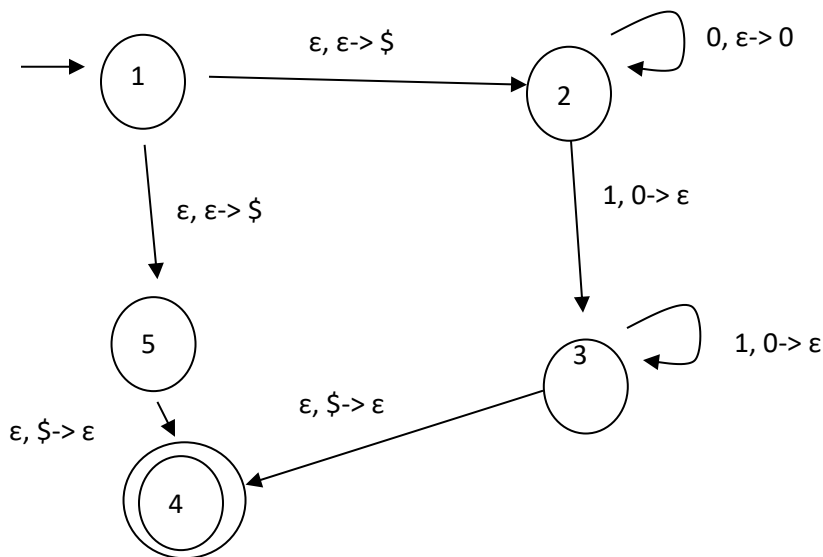
Transform PDA:

1. One accept state.



2. Empties stack before accepting (already done)

3. All transitions either pop or push a symbol:



Ok –now we’re ready to make variables for the grammar.

- 1) Identify all transitions where same symbol is (pushed and popped) or (popped and pushed):

Note: we will not worry about the pop/push pairs for this PDA, since there is no path that gets from the start to the accept state with a pop/push in that order.

	push	pop
\$	1 -> 2	5 -> 4
\$	1 -> 2	3 -> 4
\$	1 -> 5	5 -> 4
\$	1 -> 5	3 -> 4
0	2 -> 2	2 -> 3
0	2 -> 2	3 -> 3

Rules from push/pop pairs and pop/push pairs are:

$A_{14} \rightarrow \epsilon A_{25} \epsilon \mid \epsilon A_{23} \epsilon \mid \epsilon A_{55} \epsilon \mid \epsilon A_{53} \epsilon$

$A_{23} \rightarrow 0A_{22}1 \mid 0A_{23}1$

2) Add rules $A_{pp} \rightarrow \epsilon$

$A_{11} \rightarrow \epsilon$

$A_{22} \rightarrow \epsilon$

$A_{33} \rightarrow \epsilon$

$A_{44} \rightarrow \epsilon$

$A_{55} \rightarrow \epsilon$

3) Add rules $A_{ik} \rightarrow A_{ij}A_{jk}$ for all i, j, k (most will be unreachable)

$A_{15} \rightarrow A_{11}A_{15} \mid A_{12}A_{25} \mid A_{13}A_{35} \mid A_{14}A_{45} \mid A_{15}A_{55}$

$A_{25} \rightarrow \dots$

$A_{35} \rightarrow \dots$

(125 rules)

4) Start rule is A_{14}

Overall grammar: // keep the part 3 rules that appear on right-hand side

$A_{14} \rightarrow \epsilon A_{25} \epsilon \mid \epsilon A_{23} \epsilon \mid \epsilon A_{55} \epsilon \mid \epsilon A_{53} \epsilon$

$A_{23} \rightarrow 0A_{22}1 \mid 0A_{23}1$

$A_{22} \rightarrow \epsilon$

$A_{55} \rightarrow \epsilon$

$A_{25} \rightarrow A_{21}A_{15} \mid A_{22}A_{25} \mid A_{23}A_{35} \mid A_{24}A_{45} \mid A_{25}A_{55}$

$A_{23} \rightarrow A_{21}A_{13} \mid A_{22}A_{23} \mid A_{23}A_{33} \mid A_{24}A_{43} \mid A_{25}A_{53}$

$A_{55} \rightarrow A_{51}A_{15} \mid A_{52}A_{25} \mid A_{53}A_{35} \mid A_{54}A_{45} \mid A_{55}A_{55}$

$A_{53} \rightarrow A_{51}A_{13} \mid A_{52}A_{23} \mid A_{53}A_{33} \mid A_{54}A_{43} \mid A_{55}A_{53}$

$$A_{22} \rightarrow A_{21}A_{12} \mid A_{22}A_{22} \mid A_{23}A_{32} \mid A_{24}A_{42} \mid A_{25}A_{52}$$

Can delete some of these part 3 rules to tighten up grammar (the bottom five don't bottom out to terminals, so they can be removed). The first and last options of the top rule can be pruned since these do not lead to any strings.

$$A_{14} \rightarrow \epsilon A_{23} \epsilon \mid \epsilon A_{55} \epsilon$$

$$A_{23} \rightarrow 0A_{22}1 \mid 0A_{23}1$$

$$A_{22} \rightarrow \epsilon$$

$$A_{55} \rightarrow \epsilon$$

Cleaning up further through substitutions:

$$A_{14} \rightarrow A_{23} \mid \epsilon$$

$$A_{23} \rightarrow 01 \mid 0A_{23}1$$

This looks a lot like the grammar we originally created for this language.

If you want to be safe, do not remove any rules to clean up the grammar. In this course, it's ok if you keep unreachable rules. Better to be "safe" than "sorry" by accidentally removing rules. You can write the complete grammar as follows:

$$A_{14} \rightarrow \epsilon A_{23} \epsilon \mid \epsilon A_{55} \epsilon$$

$$A_{23} \rightarrow 0A_{22}1 \mid 0A_{23}1$$

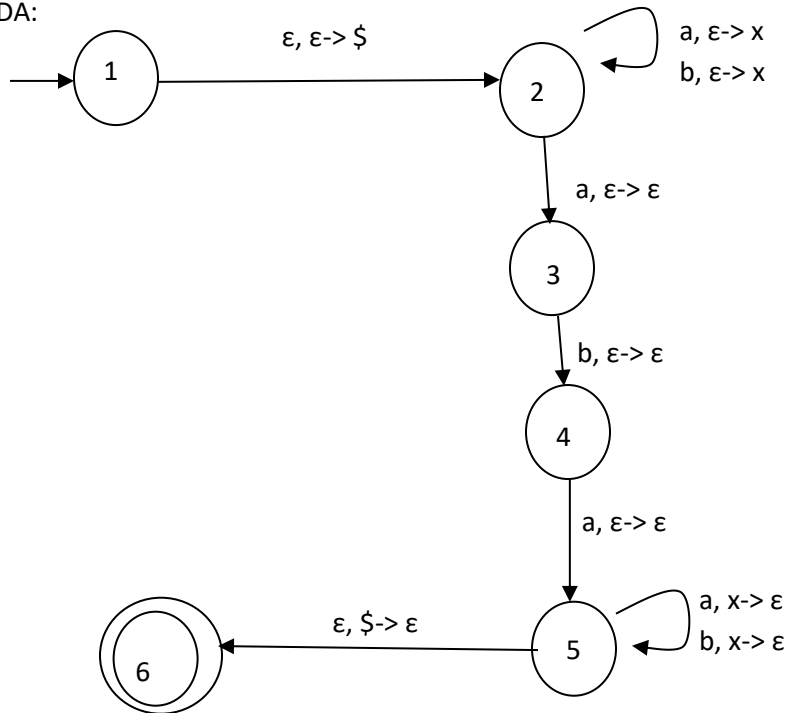
$$A_{ii} \rightarrow \epsilon \quad \text{for } 1 \leq i \leq 5$$

$$A_{ik} \rightarrow A_{ij}A_{jk} \quad \text{for } 1 \leq i \leq 5, 1 \leq j \leq 5, 1 \leq k \leq 5$$

PDA to CFG Example 2

Here is another example of the PDA to CFG conversion. Assume the language is the set of strings with middle three symbols equal to aba.

PDA:

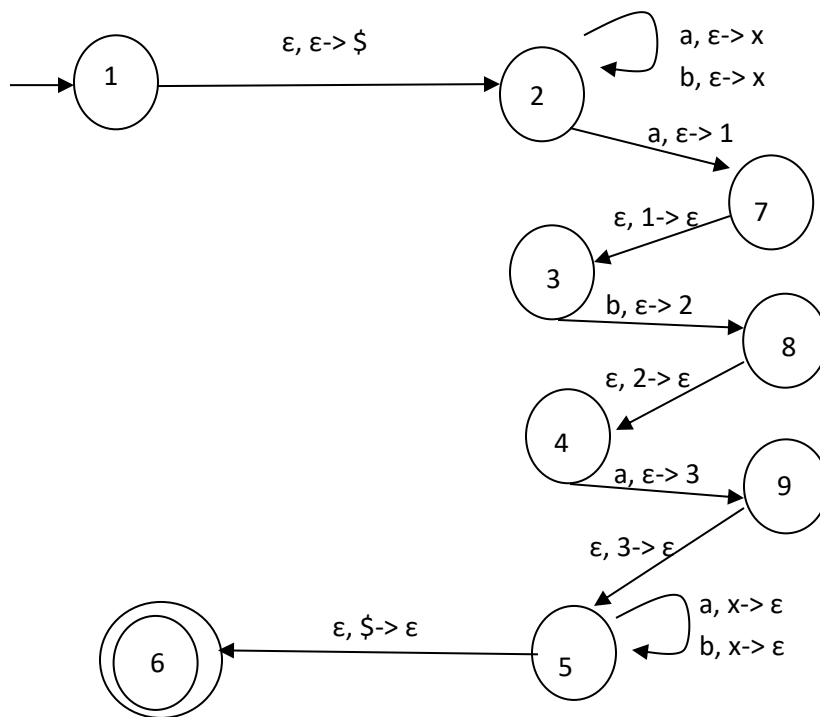


Step 1: Transform PDA, so that it:

- 1) Has one accept state
- 2) Empties stack before accepting
- 3) Each transition pushes or pops on symbol

With the PDA above, 1 and 2 are already done. The PDA below adds states and transitions, so that every transition pushes or pops one symbol.

PDA:



Ok –now we’re ready to make variables.

5) Identify all transitions where same symbol is pushed and popped or popped and pushed:

	push	pop
\$	1 -> 2	5 -> 6
x	2 -> 2	5 -> 5
1	2 -> 7	7 -> 3
2	3 -> 8	8 -> 4
3	4 -> 9	9 -> 5

Note: technically, you would also want to consider pairs that popped and then pushed the same symbol, but looking at the PDA as a graph, all symbols are pushed before they are popped, so you may ignore the pop/push pairs on this example.

1) Create rules for each of these pairs:

$A_{16} \rightarrow \epsilon A_{25} \epsilon$
 $A_{25} \rightarrow a A_{25} a \mid a A_{25} b \mid b A_{25} a \mid b A_{25} b$
 $A_{23} \rightarrow a A_{77}$
 $A_{34} \rightarrow b A_{88}$
 $A_{45} \rightarrow a A_{99}$

Note that A_{16} is the start variable, since that represents the path from the start state to the final accept state.

2) Now, we add rules for each state going to ϵ .

$A_{11} \rightarrow \epsilon$

$A_{22} \rightarrow \epsilon$

$A_{33} \rightarrow \epsilon$

$A_{44} \rightarrow \epsilon$

$A_{55} \rightarrow \epsilon$

$A_{66} \rightarrow \epsilon$

$A_{77} \rightarrow \epsilon$

$A_{88} \rightarrow \epsilon$

$A_{99} \rightarrow \epsilon$

3) Now, we add rules for paths through the PDA but most of these are not necessary:

$A_{16} \rightarrow A_{11}A_{16} \mid A_{12}A_{26} \mid A_{13}A_{36} \mid A_{14}A_{46} \mid A_{15}A_{56} \mid A_{16}A_{66} \mid A_{17}A_{76} \mid A_{18}A_{86} \mid A_{19}A_{96}$

$A_{26} \rightarrow \dots$

How do we know which rules are necessary? Look at the PDA as a graph. Include rules that are actual paths through the PDA and for which there are variables on the RHS of the rules created from the push/pop pairs.

A_{25} is the only variable on the right-hand side of a rule that is not A_{jj} . So, we need to keep the paths for A_{25} :

$A_{25} \rightarrow A_{22}A_{25} \mid A_{23}A_{35} \mid A_{24}A_{45} \mid A_{25}A_{55} \mid A_{21}A_{15} \mid A_{26}A_{65} \mid A_{27}A_{75} \mid A_{28}A_{85} \mid A_{29}A_{95}$

Of these, we only need to keep the ones for which there is a path in the PDA.

There is no path through the PDA for these rules:

$A_{21}A_{15}$

$A_{26}A_{65}$

$A_{25} \rightarrow A_{22}A_{25}$ is simply $A_{25} \rightarrow A_{25}$, so it can be removed. The same is true for $A_{25} \rightarrow A_{25}A_{55}$.

We can trim this rule to:

$A_{25} \rightarrow A_{23}A_{35} \mid A_{24}A_{45} \mid A_{27}A_{75} \mid A_{28}A_{85} \mid A_{29}A_{95}$

Then, we can see which of these rules will actually “bottom out” to actual terminals, given the other rules in the grammar. Of these, the **first two** will bottom out, so we just keep those.

Now, we have the following rules on the RHS that need to be expanded: $A_{23} A_{35} A_{24} A_{45}$

$A_{23} \rightarrow A_{22}A_{23} \mid A_{23}A_{33} \mid A_{24}A_{43} \mid A_{25}A_{53} \mid A_{21}A_{13} \mid A_{26}A_{63} \mid A_{27}A_{73} \mid A_{28}A_{83} \mid A_{29}A_{93}$
 $A_{35} \rightarrow A_{32}A_{25} \mid A_{33}A_{35} \mid A_{34}A_{45} \mid A_{35}A_{55} \mid A_{31}A_{15} \mid A_{36}A_{65} \mid A_{37}A_{75} \mid A_{38}A_{85} \mid A_{39}A_{95}$
 $A_{24} \rightarrow A_{22}A_{24} \mid A_{23}A_{34} \mid A_{24}A_{44} \mid A_{25}A_{54} \mid A_{21}A_{14} \mid A_{26}A_{64} \mid A_{27}A_{74} \mid A_{28}A_{84} \mid A_{29}A_{94}$
 $A_{45} \rightarrow A_{42}A_{25} \mid A_{43}A_{35} \mid A_{44}A_{45} \mid A_{45}A_{55} \mid A_{41}A_{15} \mid A_{46}A_{65} \mid A_{47}A_{75} \mid A_{48}A_{85} \mid A_{49}A_{95}$

We can remove rules that do not represent an actual path in the PDA:

$A_{23} \rightarrow A_{22}A_{23} \mid A_{23}A_{33} \mid A_{27}A_{73}$
 $A_{35} \rightarrow A_{33}A_{35} \mid A_{34}A_{45} \mid A_{35}A_{55} \mid A_{38}A_{85} \mid A_{39}A_{95}$
 $A_{24} \rightarrow A_{22}A_{24} \mid A_{23}A_{34} \mid A_{24}A_{44} \mid A_{27}A_{74} \mid A_{28}A_{84}$
 $A_{45} \rightarrow A_{44}A_{45} \mid A_{45}A_{55} \mid A_{49}A_{95}$

Now, we can remove RHS rules that will never bottom out using the rest of the rules in the grammar.

$A_{23} \rightarrow A_{22}A_{23} \mid A_{23}A_{33}$ // this is simply $A_{23} \rightarrow A_{23}$ for both, so can be removed
 $A_{35} \rightarrow A_{34}A_{45}$
 $A_{24} \rightarrow A_{23}A_{34}$
 $A_{45} \rightarrow A_{44}A_{45} \mid A_{45}A_{55}$ // this is simply $A_{45} \rightarrow A_{45}$ for both, so can be removed

The final grammar is:

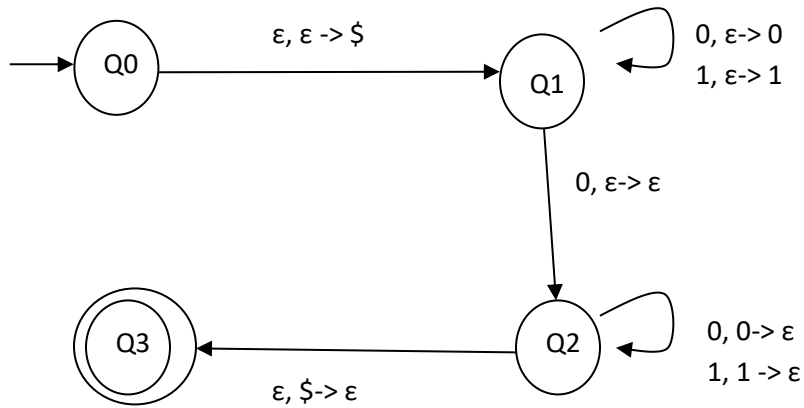
$A_{16} \rightarrow \epsilon A_{25} \epsilon$
 $A_{25} \rightarrow aA_{25}a \mid aA_{25}b \mid bA_{25}a \mid bA_{25}b \mid A_{23}A_{35} \mid A_{24}A_{45}$
 $A_{24} \rightarrow A_{23}A_{34}$
 $A_{23} \rightarrow aA_{77}$
 $A_{34} \rightarrow bA_{88}$
 $A_{35} \rightarrow A_{34}A_{45}$
 $A_{45} \rightarrow aA_{99}$
 $A_{11} \rightarrow \epsilon$ //can remove
 $A_{22} \rightarrow \epsilon$ //can remove
 $A_{33} \rightarrow \epsilon$ //can remove
 $A_{44} \rightarrow \epsilon$ //can remove
 $A_{55} \rightarrow \epsilon$ //can remove
 $A_{66} \rightarrow \epsilon$ //can remove
 $A_{77} \rightarrow \epsilon$
 $A_{88} \rightarrow \epsilon$
 $A_{99} \rightarrow \epsilon$

We could then remove any A_{jj} rule that cannot be reached.

Activity 11: Convert PDA to CFG

$$L = \{w0w^R \mid w \in \{0,1\}^*\}$$

Convert this PDA to a grammar:



Theorem: CFLs are closed under union.

Proof: Assume A and B are context free languages with CFGs $G_A = (V_A, \Sigma_A, R_A, S_A)$ and $G_B = (V_B, \Sigma_B, R_B, S_B)$. Rename all variables X in V_A as X_A and rename all variables X in V_B as X_B . Create a new grammar G for A U B as follows:

$V = V_A \cup V_B \cup \{S'\}$ where S' is a new start variable

$\Sigma = \Sigma_A \cup \Sigma_B$

$R = R_A \cup R_B \cup \{S' \rightarrow S_A \mid S_B\}$

$S = S'$

U is a grammar that generates the union of the two languages. All strings are generated from S' , a rule that goes to either the start rule for the grammar for A or the start rule for the grammar for B. Since all variables were renamed with subscripts for the language they generate, no strings will be formed from a mixture of rules in G_A and G_B .

By the way, CFLs are also closed under concatenation and star, which you may be proving for homework. However, CFLs are NOT closed under complement and intersection, which will be shown later in the course.

Can you build a PDA or CFG for the language $a^n b^n c^n$?

Hmmm... stack can get two groups to be equivalent, but how to get all three groups of symbols to be the same length? Perhaps this language is not context free.

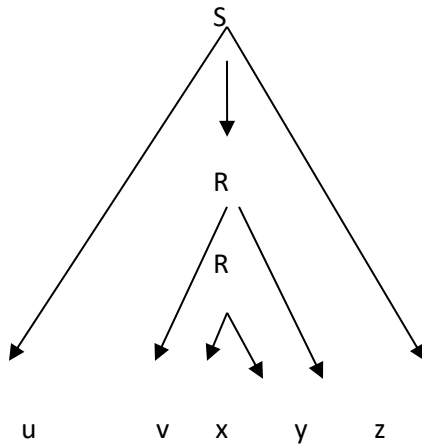
Pumping Lemma for Context Free Languages

If A is context free, there is a number p (the pumping length) such that if s is a string in A with length at least p , then s may be divided into five pieces: $s = uvxyz$, such that:

1. for each $i \geq 0$, uv^ixy^iz is in A
2. $|vy| > 0$
3. $|vxy| \leq p$

Proof idea:

Let G be a grammar that generates A . Let s be some long string in A . s has a parse tree with the root as the start variable. Since s is long, some variable V must repeat since the grammar has a finite # of variables. Assume R is the repeating variable. Then the parse tree must look like:



If we remove the bottom R (don't re-substitute), then we can create the string uxz (by having the middle R go to x).

We could also resubstitute the top R into the bottom R .

So, now $S \rightarrow uvvxyyz$.

We could do this for any number of substitutions, to get uv^ixy^iz is in A .

What must the value of p be?

Let G have $|V|$ variables.

$$p = b^{|V|+1}$$

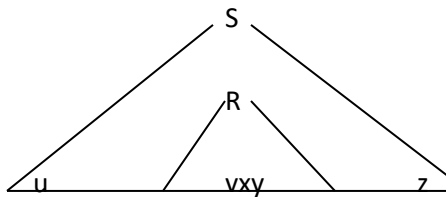
where b is the max number of symbols on the right side of any rule
 p is the number of symbols on the leaves

Then, the parse tree for S with length $\geq p$ will be at least $|V| + 1$ tall.

Why must $|vy| > 0$?

If s can have a parse tree with $|vy| = 0$ (meaning no symbols being used), then we can find a smaller parse tree for s that does not contain vy , so we'll use that minimum parse tree.

Why must $|vxy| \leq p$?



Choose R , the repeating variable lowest in the tree. The height of R to the bottom of the tree is at most $|V| + 1$ high (if each rule is used once). Each rule goes to at most b items on the right side, so a tree of height $|V| + 1$ with spread b , can generate at most $b^{|V| + 1}$ symbols. So $|vxy| \leq p$.

Let's see how this plays out with a context free language:

$A \rightarrow 0A1 \mid \#$

$p = 3^2 = 9$ //note: 9 is not necessarily the minimum pumping length – it's at least the minimum

Let $s = 0000\#1111$

Divide $s = uvxyz$, such that

$u = 000$

$v = 0$

$x = \#$

$y = 1$

$z = 111$

We can pump this up or down and the resulting string is still in the language.

Using the pumping lemma to show a language is NOT context free:

Game:

1. Adversary chooses p
2. Given p , you choose a string w in the language such that $|w| \geq p$
3. The adversary chooses the decomposition of $w = uvxyz$ such that $|vy| > 0$ and $|vxy| \leq p$
4. For each decomposition, you show that there exists a specific integer value for i such that $uv^i xy^i z$ is not in the language.

Example:

$A = \{a^n b^n c^n \mid n \geq 0\}$. Show A is not context free.

Proof: Assume A is context free. Thus, the pumping lemma for CFLs holds. Let p be the pumping length.

Let $s = a^p b^p c^p$. Consider $s = uvxyz$ subject to $|vy| > 0$ and $|vxy| \leq p$:

aaa.....aaaaabbbb....bbbbbbccc.....ccccc

<- p -><- p -><- p ->

Where could v and y be?

Step 1: Choose $s: a^p b^p c^p$

Step 2: Consider all possible decompositions

Case 1: vxy is in the set of all a 's, all b 's, or all c 's. let $i = 2$. Consider uv^2xy^2z . This pumped string has too many of one letter versus the other two letters and is not in the form $a^n b^n c^n$. The pumped string is not in A .

Case 2: v is in a 's, y is in b 's OR v is in b 's and y is in c 's. Let $i = \underline{\hspace{2cm}}$. Consider $uv^i x^i z$. Why is the pumped string not in A ?

Case 3: v or y includes a mixture of a 's/ b 's or a mixture of b 's/ c 's. Let $i = \underline{\hspace{2cm}}$. Consider $uv^i x^i z$. Why is the pumped string not in A ?

Are there any other cases? Remember that $|vxy| < p$, so v could not be in the a 's and y in the c 's. v and y would be too far apart.

Examples of proofs showing languages are not context free

$A = \{w\#t \mid w \text{ is a substring of } t \text{ over } \{a,b\}^*\}$

Prove A is not a context free language.

We will proceed with proof by contradiction. Assume A is a context free language. Thus, the pumping lemma holds and we can assume some pumping length $p > 0$ exists for language A. Then, let:

$$s = a^p b^p \# a^p b^p$$

Note that this string s has length $4p + 1$, so its length is definitely $\geq p$. Also, s is clearly in language A since the w piece of the string is equal to the t piece of the string and a substring can be the string itself.

Now, let's consider the ways s can be split into five pieces, such that:

$$s = uvxyz$$

with the constraints that $|vxy| \leq p$ and $|vy| > 0$.

For the simplicity of this proof, define a segment as one of the following strings of s: a^p or b^p

Case 1: Both the v and y substrings are in the substring before the #. Consider $i = 2$. Consider $s' = uv^2xy^2z$. Note that since $|vy| > 0$, at least one more a or one more b in one of the segments before # appears in s' versus s. Then, the substring w in s' before the # is longer than the substring t after the #, so w cannot be a substring of t. Therefore, s' is not in A.

Case 2: One of the v or y substrings contain the # character. Consider $i = 0$. Consider $s' = uv^0xy^0z$. Because the # is no longer in s' , s' is not in the right form to be in A.

Case 3: Both the v and y substrings are in the substring after the #. Consider $i = 0$. Consider $s' = uv^0xy^0z$. Since $|vy| > 0$, at least one character is removed from the substring after the # character. Because the substring w before the # in s' is longer than the substring t after the # in s' , w cannot be a substring of t. Therefore, s' is not in A.

Case 4: The non-empty v part is b^n for $1 \leq n \leq p-2$ in the segment of b's prior to the # and the non-empty y part is a^k for $1 \leq k \leq p-2$ in the segment of a's after the #. Note that if v is empty, then use case 1. If y is empty, then use case 3. Consider $i = 0$. Consider $s' = uv^0xy^0z$. Then, at least one b prior to the # is removed from s and at least one a after the # is removed from s. Because at least one a is removed after the #, the number of a's before the # is bigger than the number of a's after the # in s' . It is not in the form $w\#t$ where w is a substring of t. Therefore, s' is not in A.

We have considered all possible decompositions of s into $uvxyz$ such that $|vxy| \leq p$ and $|vy| > 0$. For each decomposition, we have found a pumping value i , that when the string s is pumped, it is not a member of A . Therefore, the pumping lemma fails to hold. We have a contradiction. Hence, the language A is not context free.

Activity 12: Proving Non-CFL using Pumping Lemma

Complete proofs to show that the following languages are NOT context-free:

$$B = \{a^n b^{2n} c^{3n} \mid n \geq 0\}$$

Activity 12 continued

$C = \{w \mid w \text{ contains an equal number of a's and b's and an equal number of c's and d's}\}$

Summary (Context Free Languages)

CFLs are closed under union, concatenation, and $*$.

Not closed under complement and intersection.

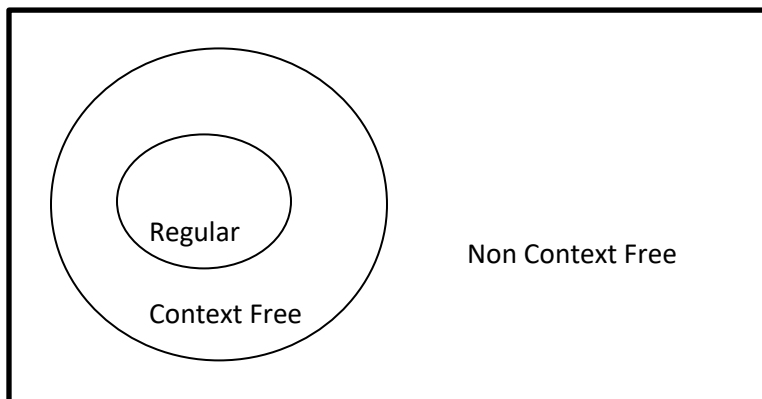
Can you name a CFL that is not closed under complement?

Can you name two CFLs that are not closed under intersection?

CFLs can be expressed using a grammar (CFG) or by a pushdown automata (PDA).

We showed earlier how to convert CFG \rightarrow PDA.

We showed earlier how to convert PDA \rightarrow CFG.



CS357 Review Sheet – Midterm Exam #2

You may use 1 crib sheet as notes during the exam. All other resources are off limits – you are on your honor to follow these exam rules.

Content: Exam 2 will primarily cover sections 1.4 through 2.3 of the textbook. Material will be drawn from homework assignments, lectures, and the textbook. Note: you should still know the content from chapters 0.1 through 1.3, as it forms the foundation for the topics most recently covered. However, the exam will focus on topics covered since the first exam.

Procedure: The exam will start promptly at the start of class. Please arrive on time. You may use one sheet of 8.5" x 11" paper (both sides) during the exam. Please prepare your own notesheet; you may type or handwrite your notesheet. Other than your sheet of notes, the exam is closed-book, closed-calculator, closed-computer other than the moodle links for the problems and solution uploads, and closed-notes.

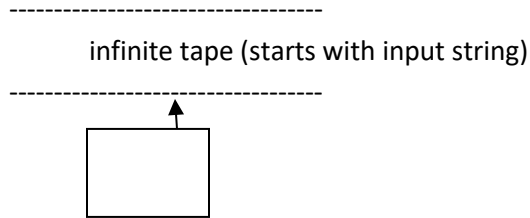
Topics: This study guide is not a contract – in other words, the exam may not cover every topic listed below and there may be topics that we covered in class that are not explicitly listed.

- Programming regular expressions
- Nonregular Languages
 - Prove using the Pumping Lemma, proof by contradiction
 - Prove via closure properties (union, concat, star, intersect, complement) and proof by contradiction
- Context-Free Languages (CFLs)
 - Grammars (CFG)
 - Given a grammar, generate the language and generate strings in the language
 - Generate parse tree for a string
 - Given a language, generate a grammar for it
 - Determine if a grammar is ambiguous
 - Converting a DFA to a grammar (shows that all regular languages are CFLs)
 - Converting a grammar to Chomsky Normal Form (CNF)
 - Pushdown Automata (PDA)
 - Formal definition
 - Given a language, generate a PDA to recognize the language
 - Given a PDA, describe the language it recognizes
 - Equivalence with CFGs (CFG→PDA, PDA→CFG conversions)
 - Proving closure properties (union, concatenation, star, etc. by modifying grammars or modifying PDAs)
- Non-Context-Free Languages (if we get through material)
 - Proving using the Pumping Lemma, proof by contradiction
 - Proving via closure properties (union, concat, *) and proof by contradiction

PREVIOUS TOPICS – EXAM 1

- Discrete Math Review
 - Sets
 - Tuples
 - Functions and Relations
 - Graphs
 - Strings and Languages
 - Boolean Logic
 - Proofs
 - Direct
 - Indirect
 - Contradiction
 - Induction
 - Construction
- Regular Languages (those that can be recognized by DFAs, NFAs, or written as regular expressions)
- DFAs
 - Given a language, construct the DFA
 - Given a DFA, state the language it recognizes
 - Formal definition as a tuple
- Union, Concatenation, *
 - Closure of regular languages under union, concatenation, and * (know the proofs)
- Closure of regular languages under other operations such as reverse and shuffle
- NFAs
 - Given a language, construct the NFA
 - Given an NFA, state the language it recognizes
 - Converting NFAs to DFAs
 - Formal definition as a tuple
- Regular Expressions
 - Given a regular expression, state its language
 - Given a language, create a regular expression
 - Converting regular expressions to NFAs
 - Converting DFAs to regular expressions

Introduction to Turing Machines



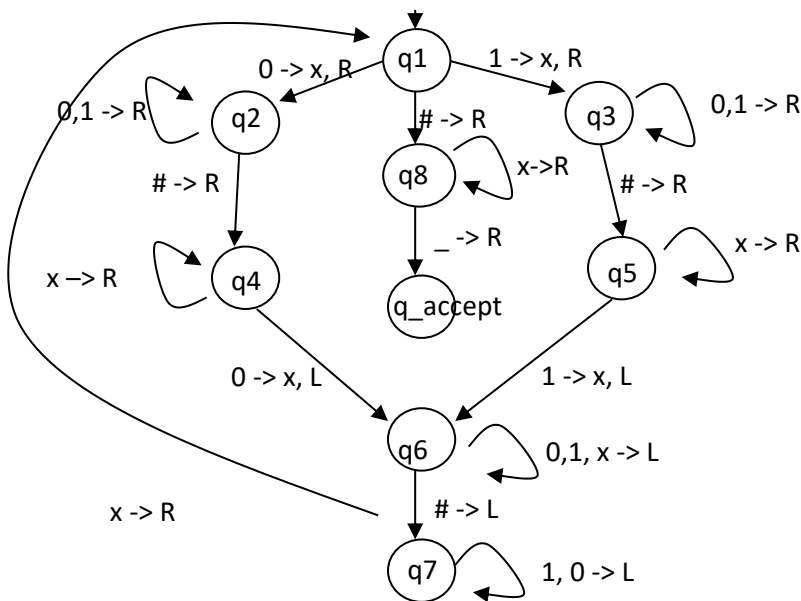
Rules about Turing machines:

1. Tape can be read and written
2. Head can move **left** and **right** (so input can be read more than once)
3. Tape is infinite in both directions
4. Accept and reject states take effect immediately (don't need to finish reading entire input string as with DFAs, NFAs, and PDAs)

Example:

$A = \{w\#w \mid w \text{ is over } \{0,1\}^*\}$

Let's see what the machine looks like as a state machine:



What do transitions mean now?

- | | |
|-----------|---|
| 1 -> x, R | Read a 1, replace it with x, and move head right |
| 1,0 -> L | Read a 1 or 0, do not change it, and move head left |
| x -> R | Read x, do not change it, and move head right |

Formal description:

$Q = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_{\text{accept}}, q_{\text{reject}}\}$

$\Sigma = \{0, 1, \#\}$

$\Gamma = \{0, 1, \#, x, _ \}$

δ (see above)

q_1 is start state

q_{accept}

q_{reject} (implicit arrows – go here if no transition is present in picture)

English description of TM M1 to accept language A:

M1:

On input w :

1. Zigzag across the $\#$, matching symbol for symbol. If the symbols across the $\#$ match, cross them out. If they do not match or there are no non-crossed-out symbols on the right to match, reject.
2. Once all symbols are crossed out before the $\#$, check for any non-crossed symbols on the right side of $\#$. If there is a non-crossed out symbol on the right, then reject. If all symbols are crossed out, accept.

The start configuration always starts with q_0 followed by the string

An *accepting configuration* is one that contains the state q_{accept}

A *rejecting configuration* is one that contains the state a_{reject}

A *halting configuration* is either an accepting configuration or a rejecting configuration.

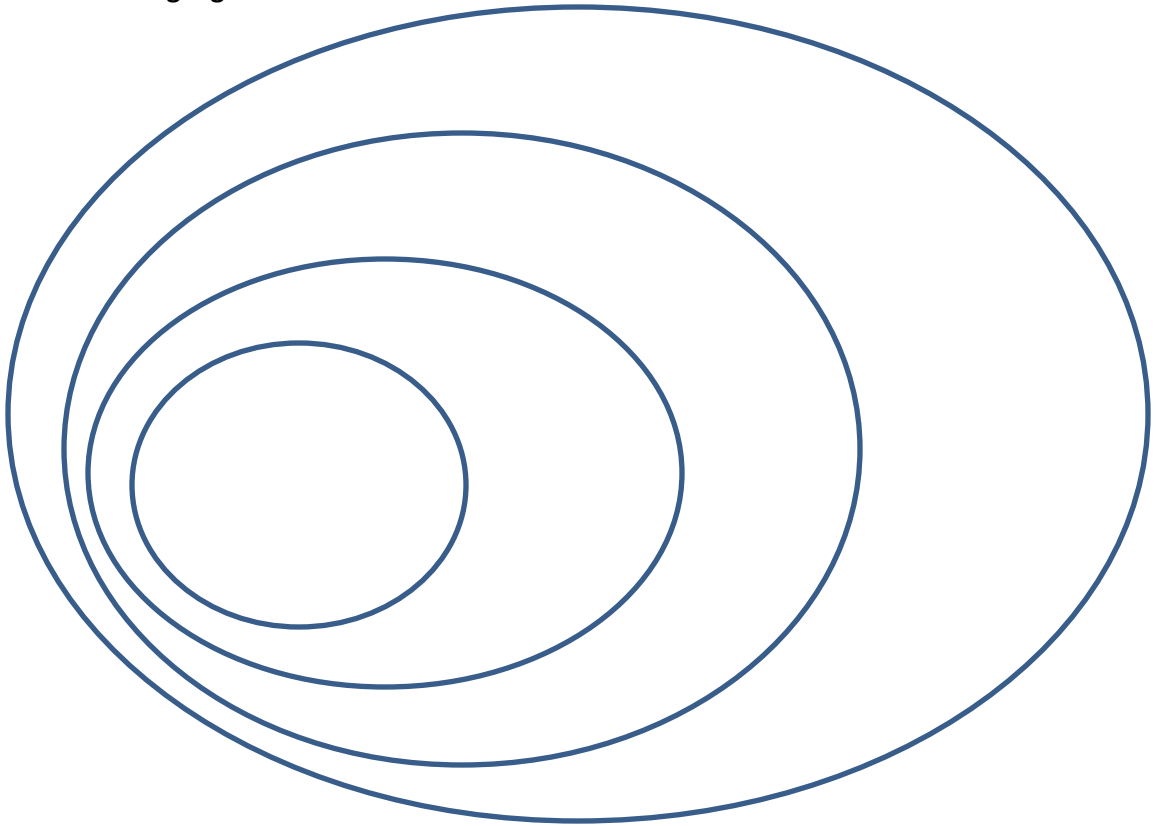
If M is a Turing machine, $L(M)$ is the language that M *recognizes/accepts*.

Defn: A language is Turing-recognizable if some Turing machine recognizes it. (If string is in the language, the TM will accept it. If the string is not in the language, the TM may reject or may not halt.)

Note that now that a TM can move left and right, we could get stuck in an infinite loop. So, the machine can now accept, reject, or loop. If the machine always halts by accepting or rejecting every string, then we say a TM decides the language.

Defn: A language is Turing-decidable if some Turing machine decides it (for all input, the machine accepts or rejects).

Put the labels for language classes here:



TM Example 1: $a^{n+1}b^n$

Try creating a TM for this language and then watch the video.

Note: this language is a CFL, so a PDA can be made for it, but you should create a TM for it.

TM Example 2: $a^{n+2}b^{n+1}c^n$

This one will be completed via a TM description (like an algorithm) for how the TM moves based on what it reads.

Hint: TM can read and overwrite characters.

Hint: TM can move left or right.

Try to create the description and then watch the video.

Variants of Turing Machines

Let's look at variants of TMs that are equivalent with the model (Turing-equivalent).

A. Stay Put

- a. Allow head to stay put in addition to moving left or right.

How is it equivalent to a regular TM? Must show a regular TM can be converted to a Stay Put TM and a Stay Put TM can be converted to a regular TM.

B. Multitape

This machine has access to a finite # of tapes and has heads pointing to each of the k tapes.

How is it equivalent?

C. Nondeterministic

A ND TM can be in more than one state. So, now the machine can be in the power set of states. Plus, if any branch accepts, the string is accepted.

Why do we have different models of TMs?

1. It's like programming languages. Some suit solving particular problems more easily.

Now, we can state:

1. A language is Turing-recognizable if and only if some ND TM recognizes it. (or multi-tape or stay-put)
2. A language is Turing-decidable if and only if some ND TM decides it.

A TM really defines what we can accomplish with an algorithm.

A computer's instruction set, a programming language, or an automaton is considered **Turing-complete** if it can be used to simulate any single-tape Turing Machine. Most programming languages are Turing-complete.

For example, if an algorithm can be coded in LISP, it can be coded in C, or Java, or Smalltalk, or R, or Prolog, etc.

We'll see later in the course that we can convert a TM to Boolean logic, which forms the basis of the architecture of many computer systems.

Another model of computation that is equivalent to Turing Machines is lambda calculus (untyped), developed by Alonzo Church. Lambda calculus defines operations as functions and functions can be applied to arguments, much like in LISP or scheme.

Activity 13: Practice Problems with Turing Machines

Complete the following TMs.

1. Draw the **state diagram** of the Turing Machine for the following language:

$$A = \{a^n b^n c^n \mid n \geq 0\}$$

2. Complete the **implementation-level** (English description) for Turing Machines for the following languages. Start with the language that Tammy assigns to your group.

$$B = \{w \mid w \text{ contains an equal number of } a\text{'s and } b\text{'s}\} \quad \Sigma = \{a, b\}$$

$$C = \{a^{2^n} \mid n \geq 0\} \quad // \text{power of 2 number of } a\text{'s } \Sigma = \{a\}$$

Activity 13 continued

$D = \{ \#x_1\#x_2\#x_3\#x_4\#\dots\#x_i \mid x_i \text{ is } \{a,b\}^* \text{ and } x_i \text{ does not equal } x_j \} \quad \Sigma = \{a, b, \#\}$

$E = \{ a^i b^j c^k \mid i \times j = k, \text{ where } i, j, k \geq 1 \} \quad \Sigma = \{a, b, c\}$

These languages show that a TM can compute in the following ways:

- A: perform equality testing of 3 unary numbers
- B: perform equality testing of 2 unary numbers
- C: check that a unary number is a power of two
- D: check all pairs of items for uniqueness
- E: perform multiplication with unary numbers

So, a TM really is an abstract model of a computer.

Examples of decidable languages (and their proofs)

Languages included below: A_DFA, A_NFA, E_DFA, EQ_DFA, A_CFG, E_CFG

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

Notation: $\langle G \rangle$ means the graph encoded as a string. For $\langle G \rangle$, this is a 2-tuple with a list of vertices followed by a list of edges. Example encoding: $(\{a, b, c, d\}, \{\{a, b\}, \{b, c\}, \{c, a\}, \{b, d\}\})$

How to build TM M?

M: On input $\langle G \rangle$, the encoded graph of G:

1. Check that G is in appropriate form. If not, reject.
 2. Mark first node of G.
 3. Repeat until no new nodes are marked:
 - a. For each node in G, mark it if it is attached by an edge to an already marked node
 4. Scan nodes of G. If all nodes in G are marked, accept. Otherwise, reject.
-

$A_{DFA} = \{ \langle D, w \rangle \mid D \text{ is a DFA that accepts } w \}$

M: On input $\langle D, w \rangle$:

1. Check that D is in proper format. If not, reject.
2. Simulate D on w (see below).
3. If D ends in an accept state, accept. If D ends in non-accepting state, reject.

How does TM do the simulation?

Well, D is $(Q, \Sigma, \delta, q_0, F)$. Keep track of B's current state along with input w on the end of the tape. Read current input symbol and current state and then find the corresponding transition in the δ list. Update the current state by overwriting the state on the tape and cross out the symbol from w that was just read. Keep doing this until all symbols in w are crossed off. Check the current state. If the current state is in the F list, accept. Otherwise, reject.

A_{DFA} is a decidable language.

$A_{NFA} = \{ \langle N, w \rangle \mid N \text{ is an NFA that accepts } w \}$

M: On input $\langle N, w \rangle$:

1. Check that N is in proper form. If not, reject.
 2. Convert N into equivalent DFA D using technique to convert NFA to DFA (create sets representing power set of states in N).
 3. Run TM M' for A_{DFA} on $\langle D, w \rangle$. //use of subroutine
 4. If M' accepts, accept. Otherwise, reject.
-

$E_{DFA} = \{ \langle D \rangle \mid D \text{ is a DFA and } L(D) \text{ is empty} \}$

Use DFA description and see if there is a series of transitions from the start state to an accept state.

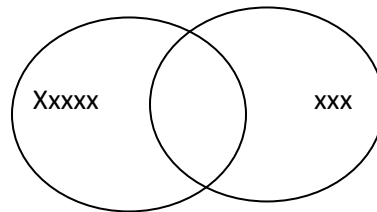
M: On input $\langle D \rangle$:

1. Check that D is in proper form. If not, reject.
 2. Check set F of D. If F is empty, accept. //this means there are no accept states in D
 3. Mark start state
 4. Repeat until no new states are marked:
 - a. Mark any state that has transition into it that is not already marked.
 5. If no accept state is marked, accept. Otherwise, reject.
-

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

Show that EQ_{DFA} is decidable.

How? Let's look at the Venn Diagram for $L(A)$ and $L(B)$.



The xxxxxx part is empty if $L(A) = L(B)$. So the idea is to build a machine to see if $L(C)$ is empty where $L(C)$ is defined as:

$$L(C) = (L(A) \cap L(B)^c) \cup (L(A)^c \cap L(B))$$

We know how to create a DFA to accept complement (reverse accept and reject states for DFA).

We know how to create a DFA to accept intersection (create pairs of states).

We know how to create a DFA to accept union (create pairs of states).

Now we can create a TM to decide EQ_{DFA} :

M: On input $\langle A, B \rangle$:

1. Check that A and B are proper DFAs. If not, reject.
2. Construct B_c to accept $L(B)$ complement.
3. Construct A_c to accept $L(A)$ complement.
4. Construct D to accept $L(A) \cap L(B)$ complement.
5. Construct E to accept $L(A)^c \cap L(B)$.
6. Construct F to accept $L(D) \cup L(E)$.
7. Run TM M' to decide E_{DFA} on $\langle F \rangle$.
8. If M' accepts, accept. If M' rejects, reject.

Now, we know we can write a program to know if two DFAs are equivalent.

$A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG and } G \text{ generates } w \}$

First, why is this language important to computer scientists?

Idea: could try all derivations of G , but could go on infinitely with recursive rule substitutions. So, this may never halt and won't decide A_{CFG} .

Need more help: convert G to CNF. In CNF, a string of length ≥ 1 will take exactly $2|w|-1$ steps to derive it.

Aside: Proof that given a grammar G in CNF, the string w with length ≥ 1 will take exactly $2|w|-1$ steps to derive.

Proof: Consider w where $|w| = n$ and w is generated by G . Each rule in G with the form $A \rightarrow BC$ increases the length of the string by 1. So, there are $n-1$ steps to generate n variables. Each variable will then go to a terminal with a rule $V \rightarrow a$, so there are n steps for these substitutions. In total, $2n-1$ steps are required.

Now, back to A_{CFG} .

M: On input $\langle G, w \rangle$:

1. Check that G and w are in proper form. If not, reject.
 2. Convert G to CNF. Let the new grammar in CNF be G' .
 3. List all derivations with $2n-1$ steps from grammar G' where $n = |w|$. If n is 0, list all derivations of one step.
 4. If any derivation generates w , accept. Otherwise, reject.
-

$E_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) \text{ is empty} \}$

Can we try every string?

Idea: can't just try every string (infinite # to try). Instead, use grammar to see if any path from terminals back to start rule exist.

TM: On input $\langle G \rangle$:

1. Check that G is in proper form. If not, reject.
 2. Mark all terminal symbols in R of G .
 3. Repeat until no new variables are marked:
 - a. Mark any variable A where G has the rule $A \rightarrow M_1 M_2 M_3 \dots M_k$ where every M_i has already been marked.
 4. If the start variable is not marked, accept. Otherwise, reject.
-

Theorem: Every CFL is decidable.

Proof: Let A be a CFL. We need to show that we can build a TM M that decides strings in A . Let w be a string that is input to the TM M and let G be the CFG for A .

Here is TM M to decide A :

On input w :

1. Run TM M' for A_{CFG} on $\langle G, w \rangle$.
 2. If M' accepts, accept. Otherwise, reject.
-

Now, what do we know about language classes? Draw the circles.

Decidable Proof Examples

Try to complete these proofs and then watch the videos. Remember, you can use already proven decidable languages' TMs as subroutines.

1. Show $A_{\epsilon_{CFG}} = \{ \langle G \rangle \mid G \text{ is a CFG that generates } \epsilon \}$ is decidable.
2. Show $A = \{ \langle D \rangle \mid D \text{ is a DFA which accepts at least one string with an even number of b's.} \}$ is decidable.
3. Show $A = \{ \langle P \rangle \mid P \text{ is a PDA that accepts no strings of even length} \}$

Activity 14: Proving Languages are Decidable

Show the following languages are decidable (i.e. you can build a TM that decides the language). In all machines below, the alphabet is $\{a,b\}^*$.

You might find the following decidable languages useful as subroutines in solving these problems:

- $A_{DFA}, A_{NFA}, E_{DFA}, EQ_{DFA}, A_{CFG}, E_{CFG}$

$A = \{ \langle D \rangle \mid D \text{ is a DFA which does not accept any string with an odd number of } a\text{'s.} \}$

$B = \{ \langle R, S \rangle \mid R \text{ and } S \text{ are regular expressions and } L(R) \text{ is a subset of } L(S). \}$

$C = \{ \langle G \rangle \mid G \text{ is a CFG that generates some string in } a^*. \}$

Activity 14 continued

$D = \{ \langle D \rangle \mid D \text{ is a DFA that accepts some string with an equal number of a's and b's.} \}$

$E = \{ \langle D \rangle \mid D \text{ is a DFA and } L(D) \text{ is an infinite language} \}$

Theorem: Decidable languages are closed under union.

Proof: Let A and B be decidable languages with always-halting TMs C and D, respectively. Create a two-tape TM E for the union of A and B as follows.

On input w:

1. Copy w to the second tape.
2. Run TM C on the first tape. If C accepts, accept. If C rejects, go to step 3.
3. Run TM D on the second tape. If D rejects, reject. If D accepts, accept.

We have created a two-tape TM E that decides the union of two decidable languages, so decidable languages are closed under union. Recall that multi-tape TMs are equivalent to single-tape TMs, so the two-tape TM E shows that the union of A and B is decidable.

How could you modify the proof to show decidable languages are closed under intersection?

Can you create a TM to decide the following language?

$F = \{ \langle M, w \rangle \mid M \text{ is a Turing Machine that accepts } w \}$

What would the decider for F need to do?

Is F Turing-recognizable?

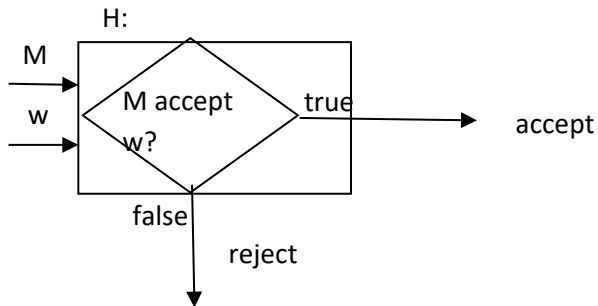
Is F Turing-decidable?

A_{TM} is undecidable

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

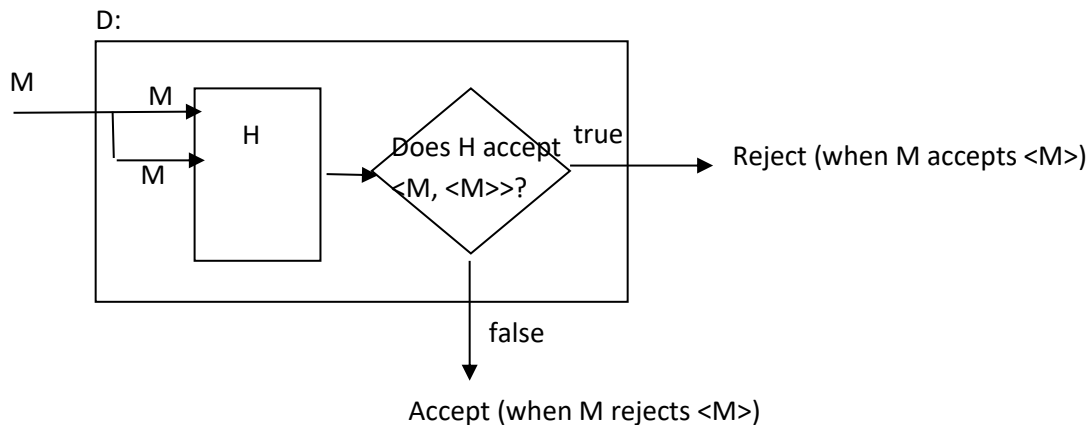
Proof that A_{TM} is undecidable: (by contradiction)

Assume A_{TM} is decidable. Then there exists a TM called H that decides A_{TM} .



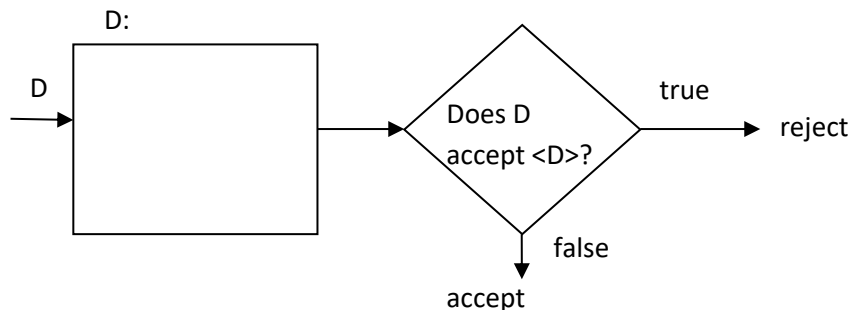
Note that if M accepts w, H accepts. If M rejects w, H rejects. If M loops on w, H rejects.

We'll create a machine D that uses H as a subroutine and reverses H's output:



Note that M is the machine H is using to simulate to see if M accepts its own string representation. This is similar to a C compiler taking in the code for the C compiler.

Well, we just created a TM D, so let's see what happens when we run D on input $\langle D \rangle$:

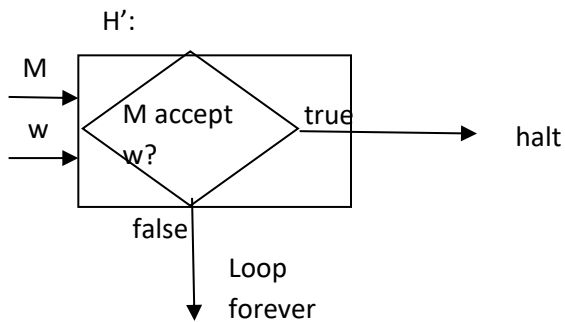


Is this possible? no (to have a machine that rejects $\langle D \rangle$ when it accepts $\langle D \rangle$ and accepts $\langle D \rangle$ when it rejects $\langle D \rangle$)

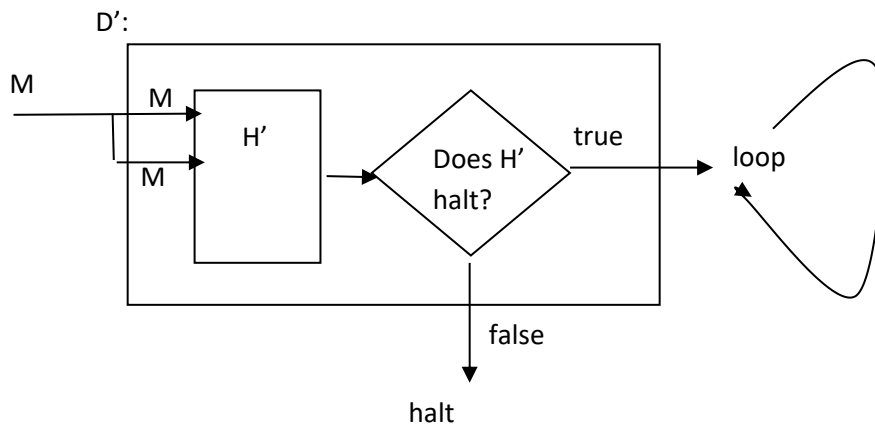
Alternate proof:

Another way to look at A_{TM} being undecidable. Instead of having H determine accept or reject, the TM will output “halt” or “loop”. So, H' will detect if the machine M will halt on input w or loop on input w .

Construct H'



Now, construct D' that uses H' :



Run D' on D' :

D' :



D' loops forever when D' halts

D' halts when D' loops forever

(impossible)

Notes about Turing-recognizable

Theorem: A language A is decidable if and only if A is Turing-recognizable and the complement of A is Turing-recognizable.

Proof:

-> Assume A is decidable. Then a TM that decides A will halt on all input. So A is TR and A complement is TR.

<- Assume A is TR and the complement of A is TR. Let M_1 recognize A and M_2 recognize A complement. Construct M as follows:

On input w:

1. Run M_1 on w on one tape.
2. Run M_2 on w on a second tape, alternating between running M_1 and running M_2 .
3. If M_1 accepts, accept. If M_2 accepts, reject.

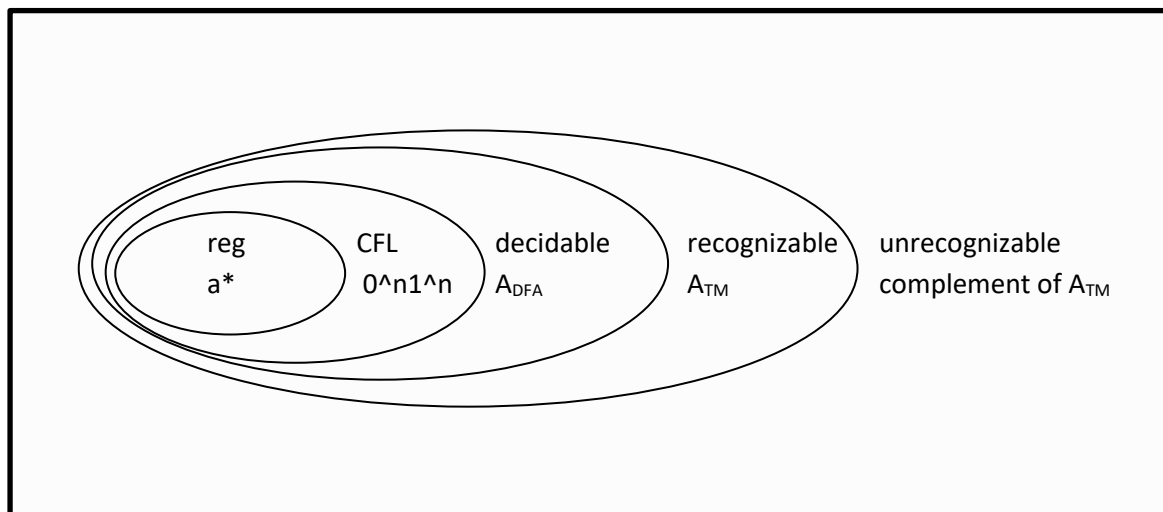
M decides A since every string is in A or A complement.

Definition: A is co-Turing-recognizable if the complement of A is Turing-recognizable.

Theorem: The complement of A_{TM} is not Turing-recognizable.

Proof: Clearly, A_{TM} is Turing-recognizable by simulating M on w. Assume A_{TM} is co-Turing-recognizable. Then by the theorem above, A_{TM} is decidable. But we showed last time that A_{TM} is undecidable, so the complement A_{TM} is not Turing-recognizable.

Now, let's look at our circles again:



Reductions (proving languages are undecidable)

What is a reduction?

Using a solution to solve a different problem (think functions or sub-routines in programming)

Example: If you already know how to calculate the area of a rectangle `rect_area(x,y)`, then you can create a function called `square_area(x)` that uses `rect_area(x,x)` as a subroutine.

Reductions will be our way to show a language is undecidable. There is no pumping lemma for showing languages are not decidable.

Theorem: If $A \leq B$ (A reduces to B) and B is decidable, then A must be decidable. (exactly what we were doing with decidable problems before)

Proof: Assume A reduces to B and B is decidable. To construct TM for A:

On input w:

1. Reduce w in A to w' in B. (mapping w to w')
2. Run TM M for B on w' . If M accepts w' , accept. If not, reject.

Corollary: (contrapositive statement) If A is undecidable and A reduces to B, then B is undecidable.

Game:

Given: a language B and want to show B is undecidable (proof by contradiction)

1. Assume B is decidable.
2. Choose another language A that is already proven to be undecidable.
3. Show A reduces to B.
4. Conclude B is undecidable (since A is undecidable and A reduces to B)

Halt_{TM} is undecidable

$\text{HALT}_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on } w \}$

Show HALT_{TM} is undecidable.

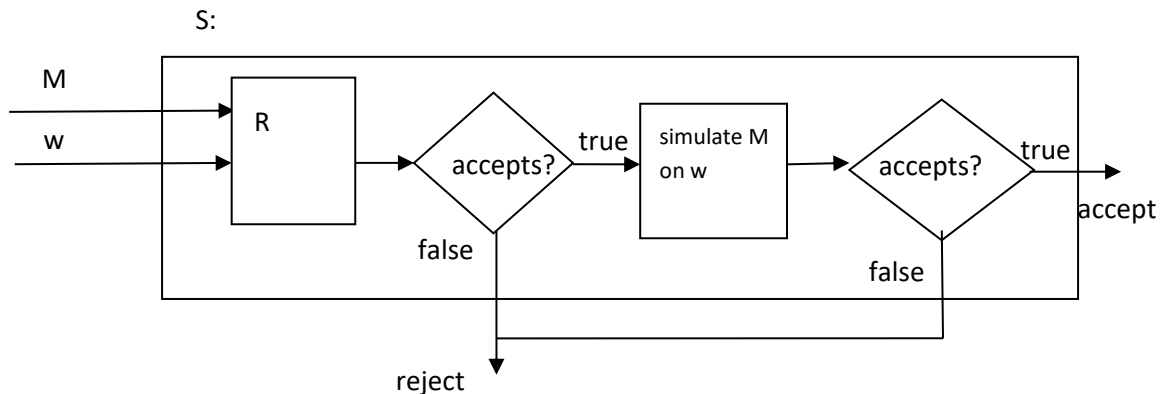
Proof: Assume HALT_{TM} is decidable by a Turing Machine R . Consider the undecidable language A_{TM} and construct TM S that decides A_{TM} by reducing A_{TM} to HALT_{TM} .

S : On input $\langle M, w \rangle$:

1. Run R on $\langle M, w \rangle$
2. If R rejects, reject.
3. If R accepts, simulate M on w until it halts. If M accepts, accept. If not, reject.

Thus, if R decides HALT_{TM} then S decides A_{TM} . We know A_{TM} is undecidable, so we have a contradiction. HALT_{TM} must be undecidable.

Picture version (remember, a picture is not a formal proof but this really helps to understand what is happening)



Code-like version:

Create function F that returns one of two values {accept, reject}, similar to a Boolean function.

Parameters: M , a Turing Machine written in formal description; w , a string of characters

```
F(<M, w>):
    result_halt = R(<M, w>) // R is a function that determines if M halts on w
    if(result_halt == reject)
        return reject
    else
        result_sim = M(w) // simulate M on w, given description of M
                           // at this point, we know M halts on w since
                           // result_halt is accept; we know the simulation
                           // will halt
        return result_sim // just return what simulating M on w returns
```

Since we created a function F that decides A_{TM} , we have a contraction with the fact that A_{TM} was already proven to be undecidable. Since the condition embedded inside function F is decidable and running M on w is executable on a TM, the only piece of the function that remains that must be undecidable is calling and running function R . Function R , therefore, cannot always return the value accept or reject. Since function R determines if M halts on w , the language $HALT_{TM}$ is undecidable.

E_{TM} is undecidable

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

Game:

1. Assume E_{TM} is decidable.
2. Show A_{TM} or $HALT_{TM}$ reduces to E_{TM} .
3. Conclude E_{TM} is undecidable.

Run machine for E_{TM} R on $\langle M \rangle$ and see if it accepts. If so, we know $L(M)$ is empty and $\langle M, w \rangle$ is not in A_{TM} . But what if E_{TM} rejects? That does not tell us anything about $\langle M, w \rangle$ being in A_{TM} . Need to build helper machine that only runs M on w if the input is equal to w.

Proof: Assume E_{TM} is decidable by TM R.

Build a helper machine M_1 :

M_1 : On input x:

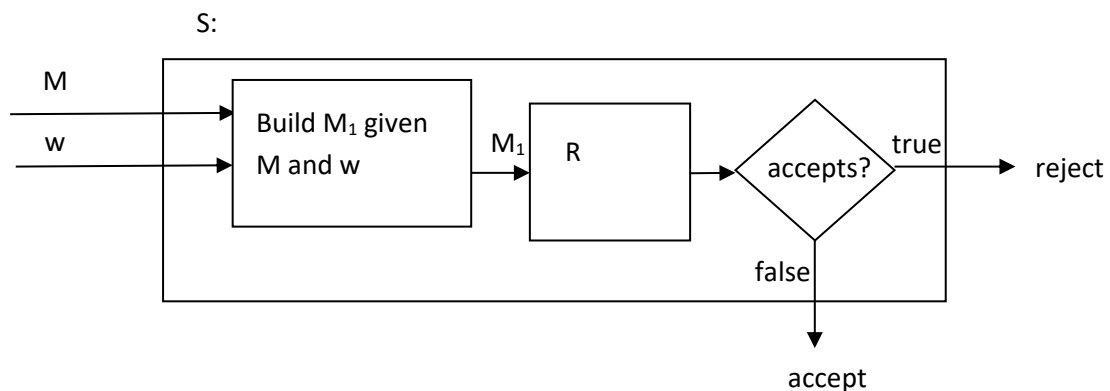
1. If x does not equal w, reject.
2. If x equals w, run M on w and accept if M accepts.

Note that M_1 is buildable since we just add TM states and transitions to check if the input on the tape is equal to w. If it is, then we simulate M on w.

Build S to decide A_{TM} as follows:

On input $\langle M, w \rangle$:

1. Use M and w to construct M_1 .
2. Run TM R for E_{TM} on $\langle M_1 \rangle$.
3. If R accepts, reject. If R rejects, accept.



Thus, if R decides E_{TM} then S decides A_{TM} . We know A_{TM} is undecidable, so we have a contradiction. E_{TM} must be undecidable.

Code-like version:

Create returns a function (higher-order function) with parameter x

Parameters: M, a Turing Machine description; w, a string

```
Create(<M, w>) :  
    Return function(x) :  
        if(x == w)  
            ret = M(w)    // run M on w  
            if(ret == accept)  
                accept //note we do not have cases for reject or loop  
        else  
            return reject
```

Create function F that returns one of two values {accept, reject}, similar to a Boolean function.

Parameters: M, a Turing Machine written in formal description; w, a string of characters

```
F(<M, w>) :  
    M1 = Create(M, w)  
    ret = R(M1)    // R is a function that determines if language of M1  
                  // is empty or not  
    if(ret == accept)  
        reject    // reverse the output of ret; if R accepts, that means  
    else          // the language is empty so M does not accept w  
        accept    // if R rejects, the language of M1 is non-empty,  
                  // so M must accept w
```


Examples of Reductions to Prove Undecidability

1. Show EQ_{TM} is undecidable.

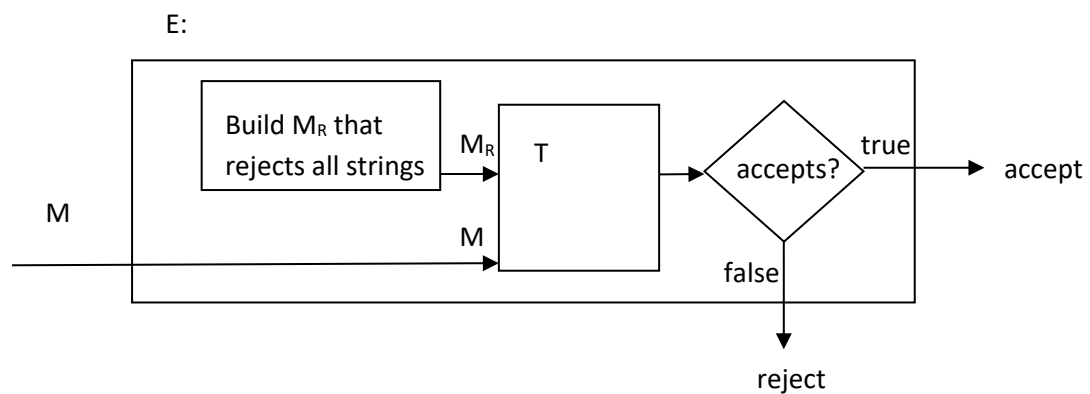
$$EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \}$$

Idea: assume EQ_{TM} is decidable. Reduce E_{TM} to EQ_{TM} .

Proof: Assume EQ_{TM} is decidable by TM T . Construct E to decide E_{TM} as follows:

On input $\langle M \rangle$:

1. Construct machine M_R that rejects all strings.
2. Run T on input $\langle M_R, M \rangle$.
3. If T accepts, accept. If not, reject.



Thus, if T decides EQ_{TM} then E decides E_{TM} . We know E_{TM} is undecidable, so we have a contradiction. EQ_{TM} must be undecidable.

At this point, we have shown that there are 4 undecidable languages:

A_{TM} , $HALT_{TM}$, E_{TM} , EQ_{TM} [Any of these can be used as the language A for reductions $A \leq B$]

If it is not immediately obvious how one of the above languages can assist with a reduction, choose A_{TM} .

2. Show $\text{REGULAR}_{\text{TM}}$ is undecidable.

$\text{REGULAR}_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language} \}$

Idea: Assume $\text{REGULAR}_{\text{TM}}$ is decidable and construct a decider for A_{TM} . We want to create a helper TM M' such that M' accepts a regular language if M accepts w and M' does not accept a regular language if M does not accept w .

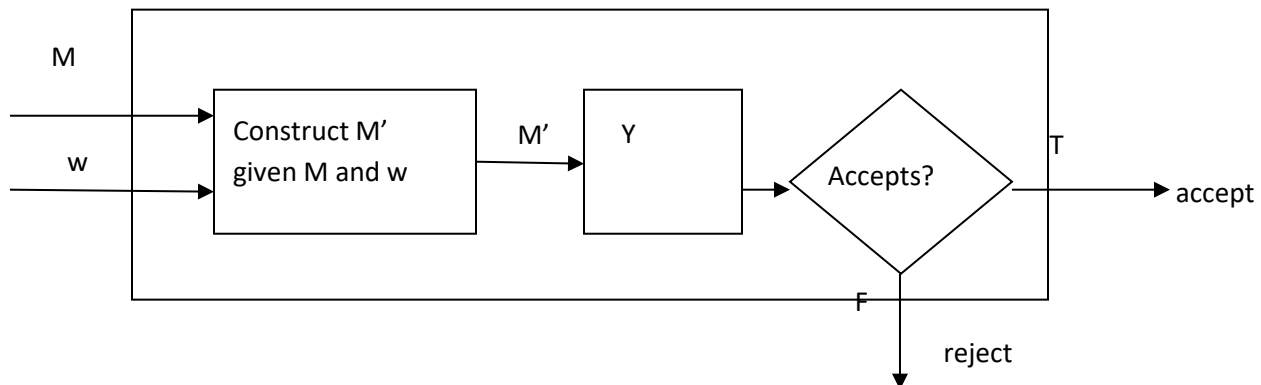
So, we'll rig the machine so: $L(M') = a^n b^n$ if M does not accept w
 $L(M') = \Sigma^*$ if M accepts w

Proof: Assume $\text{REGULAR}_{\text{TM}}$ is decidable by a TM Y . Construct S to decide A_{TM} as follows:

On input $\langle M, w \rangle$:

1. Construct TM M' where M' is the following:
 - a. On input x :
 - i. If x has the form $a^n b^n$, accept.
 - ii. If not, run M on w . If M enters its accept state, accept.
2. Run Y on $\langle M' \rangle$.
3. If Y accepts, accept. If Y rejects, reject.

S :



So, we have a decider S for A_{TM} . But A_{TM} is undecidable, so a decider Y for $\text{REGULAR}_{\text{TM}}$ must not exist. Therefore, $\text{REGULAR}_{\text{TM}}$ is undecidable.

3. Show B is undecidable.

$B = \{ \langle M, w \rangle \mid M \text{ is a two-tape TM which writes a nonblank symbol on its second tape when } M \text{ runs on } w \}$

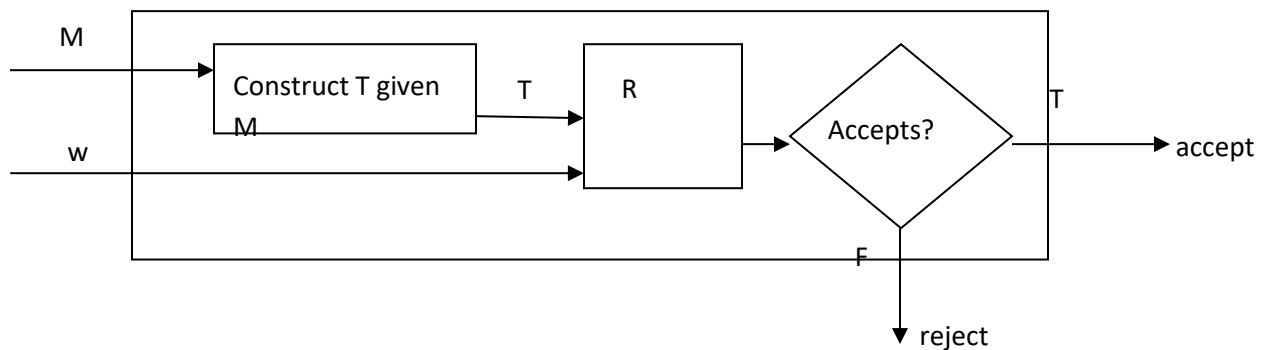
Idea: Assume B is decidable and construct a decider for A_{TM} . We want to create a helper TM T such that T writes a \$ on the second tape if M accepts w and T does not write a \$ on the second tape if M does not accept w.

Proof: Assume B is decidable by TM R. We'll construct TM S to decide A_{TM} :

S: On input $\langle M, w \rangle$

1. Use M to construct a two-tape TM T where T is:
 - a. On input x:
 - i. Simulate M on x with first tape. If M enters accept state, write a \$ on the second tape.
2. Run R on $\langle T, w \rangle$. (Note that x is assigned w as its argument to the TM)
3. If R accepts, accept. If not, reject.

S:



We have built a decider S for A_{TM} , but A_{TM} is undecidable. Thus, there is no decider R for language B, so B is undecidable.

Activity 15: Prove Languages are Undecidable
Hint: reduce from $A_{TM} \langle M, w \rangle$ for these problems

$INFINITE_{TM} = \{ \langle M' \rangle \mid M' \text{ is a TM and } L(M') \text{ is infinite} \}$

Hint: rig M' so that M' has a finite language if M (of A_{TM}) does not accept w (of A_{TM}) and accepts an infinite language if M accepts w .

$REVERSE = \{ \langle M' \rangle \mid M' \text{ is a TM that accepts the reverse of } w \text{ whenever it accepts } w \}$

Hint: rig M' so that M' accepts 'ab' and 'ba' if M (of A_{TM}) accepts w (of A_{TM}) and M' accepts only 'ab' if M does not accept w .

Activity 15 continued

$\text{WRITE_BLANK} = \{ \langle M' \rangle \mid M' \text{ is a TM that writes a blank symbol over a nonblank symbol during the course of its computation on any input string} \}$

Hint: rig M' so that it only writes a blank over a non-blank if M (of A_TM) accepts w and M' does not write a blank over a non-blank if M does not accept w .

Rice's Theorem

Rice's Theorem: Let P be a language consisting of TM descriptions where P fulfills two conditions:

1. P is non-trivial (P contains some but not all TM descriptions)
2. P is a property of the TM's language (when $L(M_1) = L(M_2)$ then $\langle M_1 \rangle$ is in P if and only if $\langle M_2 \rangle$ is in P .)

Then P is undecidable.

Proof Idea: Assume P is decidable. Show A_{TM} reduces to P , so P must then be undecidable.

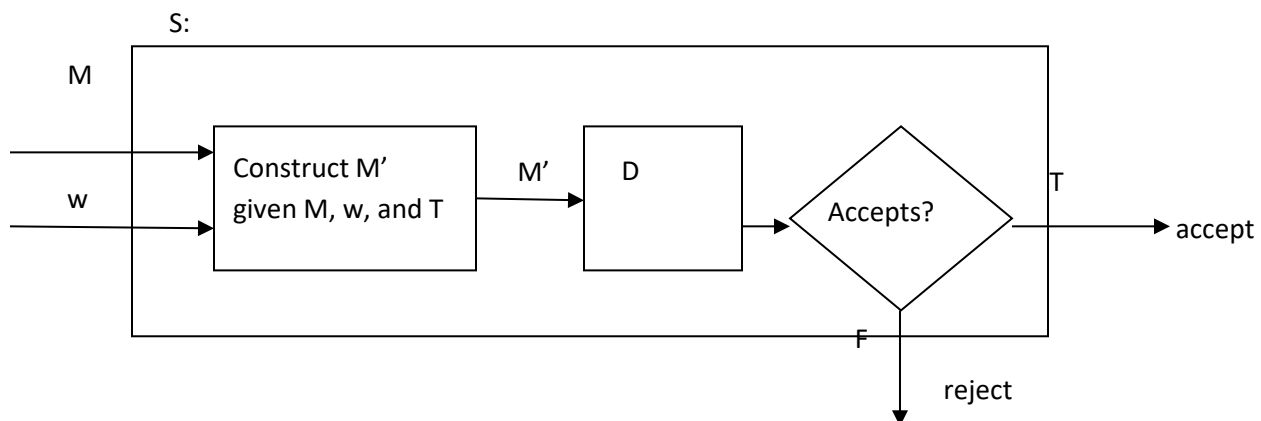
Proof: Assume P is decidable by TM D . Assume P is non-trivial, so there is some TM $\langle T \rangle$ in P . Let $\langle T_R \rangle$ be a TM that always rejects. Either $\langle T_R \rangle$ is in P or $\langle T_R \rangle$ is not in P . Without loss of generality, assume $\langle T_R \rangle$ is not in P . (If $\langle T_R \rangle$ is in P , then it is not in the complement of P and we use the decider for P complement instead.)

$\langle T_R \rangle$ is not in P . We'll build S to decide A_{TM} as follows:

S :

On input $\langle M, w \rangle$:

1. Use M and w to construct machine M' :
 - a. M' : On input x :
 - i. Simulate M on w . If it halts and rejects, reject. If it accepts, simulate T on x . If T accepts x , accept.
 - b. Run D on $\langle M' \rangle$ to determine if $\langle M' \rangle$ is in P . If D accepts, accept. If not, reject.



M' simulates exactly what T does if M accepts w . Thus, $L(M') = L(T)$ if M accepts w and $L(M') = \emptyset = L(T_R)$ if M does not accept w . $\langle T \rangle$ is in P and $\langle T_R \rangle$ is not in P . Thus, $\langle T \rangle$ is in P if and only if M accepts w . So, we have a decider S for A_{TM} which contradicts the fact that A_{TM} is undecidable. Thus, P is undecidable.

Example 1 using Rice's Theorem

$INFINITE_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is infinite} \}$. Prove this is undecidable.

Proof: Use Rice's Theorem. (Just need to show the two conditions hold)

Condition 1: $INFINITE_{TM}$ is non-trivial since some TMs have infinite languages and some do not. For example, a TM that accepts all strings has an infinite language. A TM that accepts just the string "a" accepts a finite language and would not be in $INFINITE_{TM}$.

Condition 2: If two TMs $M1$ and $M2$ recognize the same language, then either both $M1$ and $M2$ accept an infinite language and so both are in $INFINITE_{TM}$ or $M1$ and $M2$ accept a finite language, so neither are in $INFINITE_{TM}$.

By Rice's Theorem, $INFINITE_{TM}$ is undecidable.

Example 2 using Rice's Theorem

$L = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ contains exactly two distinct strings} \}.$

Show this is undecidable using Rice's Theorem. Try it and watch the video.

Hint: you need to show both conditions hold.

Activity 16: Practice with Rice's Theorem

Use Rice's Theorem to show the following languages are undecidable:

$\text{REGULAR}_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \}.$

$M_{1011} = \{ \langle M \rangle \mid M \text{ is a TM and } 1011 \text{ is in } L(M) \}$

Activity 16 continued

Questions:

$A = \{ \langle M \rangle \mid M \text{ does four consecutive right transitions on some input} \}$

// can you use Rice's Theorem?

$B = \{ \langle M \rangle \mid L(M) \text{ contains more than 100 strings} \}$

// can you use Rice's Theorem?

$C = \{ \langle M, w \rangle \mid M \text{ writes a \$ on the tape when } M \text{ is run on } w \}$

// can you use Rice's Theorem?

ALL_{CFG} is undecidable

Problem: $ALL_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^* \}$

Proof Sketch:

We will reduce A_{TM} to ALL_{CFG} . So, we'll construct a grammar G such that G generates all strings if and only if M does not accept w . The strings that will be generated by G are configurations of the TM M on w used as input to A_{TM} .

example configuration string: $\#c_1\#c_2\# \dots c_k \#$

where c_1 is the i th step of M on w .

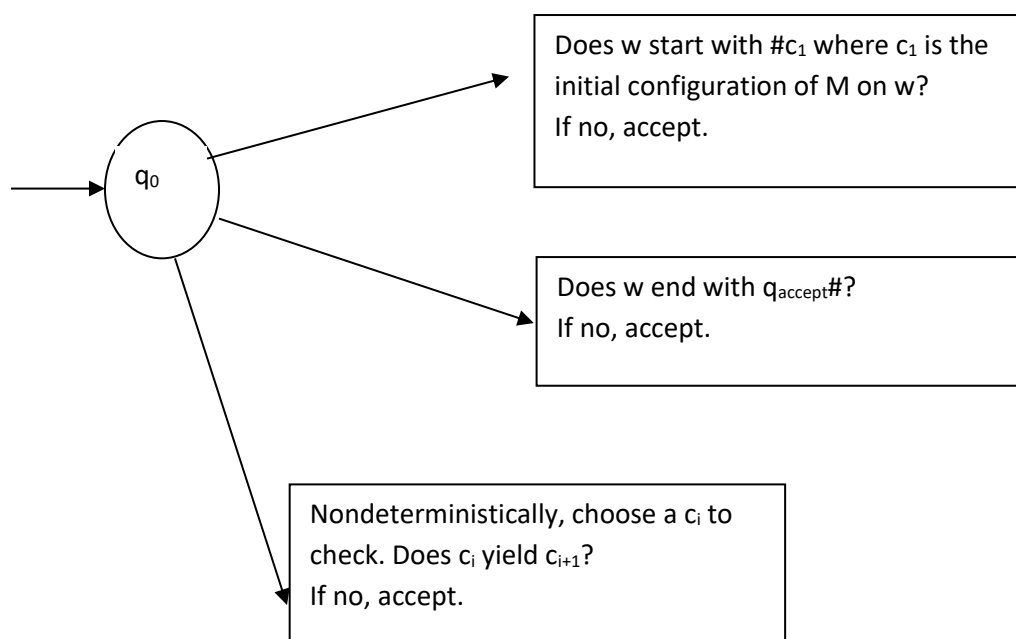
Proof: Assume ALL_{CFG} is decidable by Turing Machine A . We will reduce A_{TM} to ALL_{CFG} by creating a decider for A_{TM} that uses the decider A .

We need to create a grammar G given $\langle M, w \rangle$. We want G to generate all **invalid** series of configurations given $\langle M, w \rangle$. So what would G do?

G generates:

1. Strings that do not start with $\#c_1$ // All valid configs must start with initial config
2. Strings that do not end with accepting configuration
// We'll modify M to read to the right before accepting
3. Strings where c_i does not yield c_{i+1} given machine M and w // Invalid transition

Instead of generating G directly, it is easier to generate a nondeterministic PDA with structure as follows.



The first two branches are easy to write (simple check that a DFA could do). The third branch is a bit more tricky.

How can we see if c_i yields c_{i+1} ?

We need to push symbols of c_i onto stack and then pop symbols one at a time to see if the configuration c_{i+1} is not valid. But, when popping the symbols, c_i will be in the reverse order. Therefore, we modify the configuration strings to be as follows:

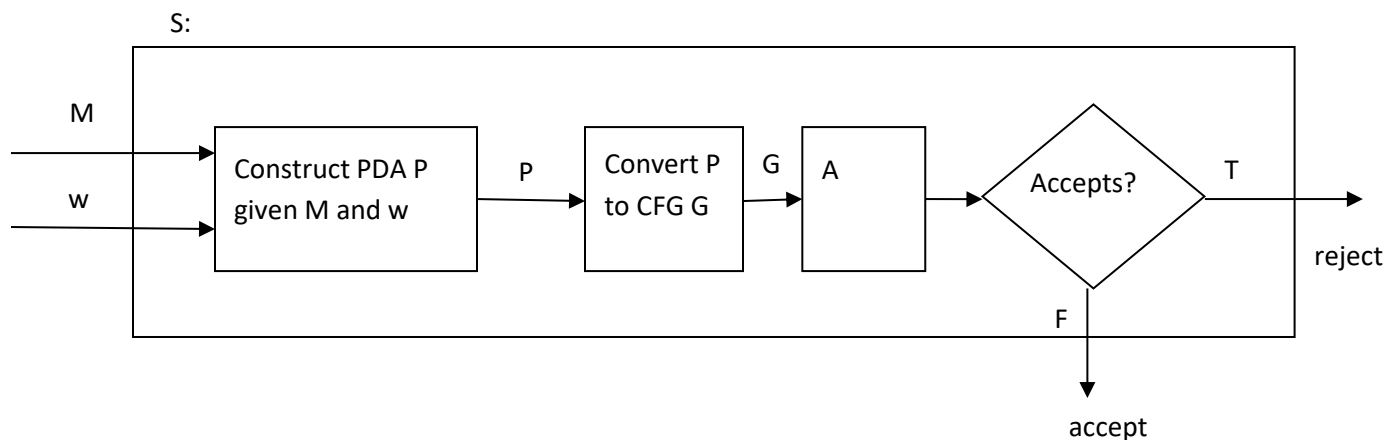
$\#c_1\#c_2^R\#c_3\#c_4^R\# \dots c_k\#$ // note c_k will be c_k^R if there is an even number of configurations

So, we now have a PDA P that can be converted to a CFG (using procedure shown in lecture).

Description of TM S :

On input $\langle M, w \rangle$:

1. Construct PDA P given above.
2. Convert P into equivalent CFG G .
3. Run A on $\langle G \rangle$. If A accepts, reject. If A rejects, accept (since the only string not generated by G would be the accepting configuration).



We have built a decider for A_{TM} , but A_{TM} is undecidable, so ALL_{CFG} must be undecidable.

Activity 17: Practice with choosing if a language is decidable or undecidable

For each language below, determine if it is decidable or undecidable. Hints: can you build a TM that accepts or rejects on all strings? Can you use Rice's Theorem? Can you reduce from A_{TM} ?

Decidable or undecidable?

$A = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is all strings that have 'bbb' as a substring} \}$

$B = \{ \langle M \rangle \mid M \text{ is a TM that has 50 states} \}$

$C = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2) \}$

$D = \{ \langle D_1, D_2 \rangle \mid D_1 \text{ and } D_2 \text{ are DFAs and } L(D_1) = L(D_2) \}$

$E = \{ \langle G, w \rangle \mid G \text{ is a grammar that generates } w \}$

$F = \{ \langle M, w \rangle \mid M \text{ is a TM that accepts } w \}$

$G = \{ \langle G \rangle \mid G \text{ is a grammar that generates no strings} \}$

$H = \{ \langle G \rangle \mid G \text{ is a grammar that generates all strings} \}$

$I = \{ \langle G \rangle \mid G \text{ is a grammar that generates at least one string that has 'bbb' as a substring} \}$

$J = \{ \langle M, w \rangle \mid M \text{ is a TM that writes } \# \text{ over a } \$ \text{ when } M \text{ runs on } w \}$

Prove the following language is undecidable:

$EQ_{CFG} = \{ \langle G_1, G_2 \rangle \mid G_1 \text{ and } G_2 \text{ are grammars and } L(G_1) = L(G_2) \}$

Hint: Reduce from ALL_{CFG} .

Summary (Decidable and Undecidable)

Decidable Languages are decided by Turing Machines – TM halts on all input.

Recognizable Languages can be recognized by a TM (always accepts strings in the language, but may not halt on strings not in the language)

Undecidable Languages cannot be decided by a TM (TM does not halt on all possible input).

To prove a language is decidable: build a TM (state machine or English description)

To prove a language is undecidable:

1. Create a reduction proof (proof by contraction)
2. Use Rice's Theorem (if P is a non-trivial property about the language of the TM)

Sets of languages:

Regular languages are CFLs.

CFLs are decidable.

Decidable languages are recognizable.

Name languages that are decidable:

Name languages that are undecidable:

CS357 Review Sheet – Midterm Exam #3

You may use 1 crib sheet as notes during the exam. All other resources are off limits – you are on your honor to follow these exam rules.

Content: Exam 3 will primarily cover sections 2.3 through 5.1 of the textbook. Material will be drawn from homework assignments, lectures, and the textbook. Note: you should still know the content from chapters 0.1 through 2, as it forms the foundation for the topics most recently covered. However, the exam will focus on topics covered since exam 2.

Procedure: The exam will start promptly at the start of class. Please arrive on time. You may use one sheet of 8.5" x 11" paper (both sides) during the exam. Please prepare your own notesheet; you may type or handwrite your notesheet. Other than your sheet of notes, the exam is closed-book, closed-calculator, closed-computer other than the moodle links to the problems and solution submissions, and closed-notes. All numerical computations (if any) will be simple enough for you to do by hand.

Topics: This study guide is not a contract – in other words, the exam may not cover every topic listed below and there may be topics that we covered in class that are not explicitly listed.

- Proving a language is not context free (pumping lemma)
- Turing Machines
 - Configurations
 - Formal definition
 - Creating a state machine
 - Equivalent models: nondeterministic, multi-tape, stay put/left/right movements
 - To show equivalence: need to convert the model to a regular TM, need to convert a regular TM to the model
 - Turing-recognizable (accepts strings in language but may not halt on all input)
 - Turing-decidable (always halts)
- Decidable Languages
 - To show A is decidable, construct a TM for A that always halts.
 - If A is a decidable language, then A is Turing-recognizable and the complement of A is Turing-recognizable (also called co-Turing-recognizable).
 - Examples of decidable languages with regular languages: A_{DFA} , A_{NFA} , A_{REG} , E_{DFA} , EQ_{DFA} , $INFINITE_{DFA}$
 - Examples with CFLs: A_{CFG} , E_{CFG}
 - Examples from homework: E_{DFA_REGEX} , ALL_{DFA} , $INFINITE_{PDA}$
 - $A = \{ \langle R \rangle \mid R \text{ is a regular expression describing a language containing at least one string } w \text{ that has } 111 \text{ as a substring.} \}$
 - $B = \{ \langle P \rangle \mid P \text{ is a PDA that has a useless state.} \}$
 - Closure: decidable languages are closed under concatenation, complement, union, intersection
- A_{TM} and the Halting Problem
 - A_{TM} is undecidable
 - A_{TM} is Turing-recognizable
 - complement of A_{TM} is not Turing-recognizable

- HALT_{TM} is undecidable
- Undecidable Languages
 - Proof by contradiction (using a reduction from language A to language B)
 - Examples of undecidable languages:
 - A_{TM} , HALT_{TM} , E_{TM} , EQ_{TM} , $\text{REGULAR}_{\text{TM}}$, ALL_{CFG}
 - Rice's Theorem
 - Example: $\text{INFINITE}_{\text{TM}}$ is undecidable
 - Example: $\{\langle M \rangle \mid M \text{ is a TM and } 1011 \text{ is in } L(M)\}$ is undecidable
 - ALL_{CFG}

PREVIOUS TOPICS – EXAM 2

- Programming regular expressions
- Nonregular Languages
 - Prove using the Pumping Lemma, proof by contradiction
 - Prove via closure properties (union, concat, star, intersect, complement) and proof by contradiction
- Context-Free Languages (CFLs)
 - Grammars (CFG)
 - Given a grammar, generate the language and generate strings in the language
 - Generate parse tree for a string
 - Given a language, generate a grammar for it
 - Determine if a grammar is ambiguous
 - Converting a DFA to a grammar (shows that all regular languages are CFLs)
 - Converting a grammar to Chomsky Normal Form (CNF)
 - Pushdown Automata (PDA)
 - Formal definition
 - Given a language, generate a PDA to recognize the language
 - Given a PDA, describe the language it recognizes
 - Equivalence with CFGs (CFG \rightarrow PDA, PDA \rightarrow CFG conversions)
 - Proving closure properties (union, concatenation, star, etc. by modifying grammars or modifying PDAs)
- Non-Context-Free Languages
 - Proving using the Pumping Lemma, proof by contradiction
 - Proving via closure properties (union, concat, *) and proof by contradiction

PREVIOUS TOPICS – EXAM 1

- Discrete Math Review
 - Sets
 - Sequences (Tuples)
 - Functions and Relations
 - Graphs
 - Strings and Languages
 - Boolean Logic
 - Proofs
 - Direct
 - Indirect
 - Contradiction
 - Induction
 - Construction
- Regular Languages (those that can be recognized by DFAs, NFAs, or written as regular expressions)
- DFAs
 - Given a language, construct the DFA
 - Given a DFA, state the language it recognizes
 - Formal definition as a tuple
- Union, Concatenation, *
- Closure of regular languages (know the proofs)
- Closure of regular languages under other operations such as reverse and perfect shuffle

- NFAs
 - Given a language, construct the NFA
 - Given an NFA, state the language it recognizes
 - Converting NFAs to DFAs
 - Formal definition as a tuple
- Regular Expressions
 - Given a regular expression, state its language
 - Given a language, create a regular expression
 - Converting regular expressions to NFAs
 - Converting DFAs to regular expressions
 - Converting DFAs to regular expressions

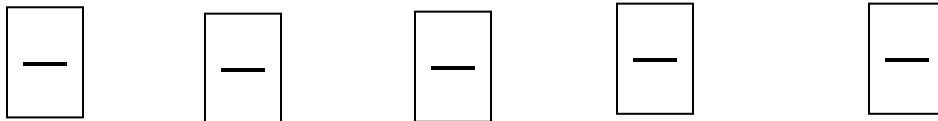
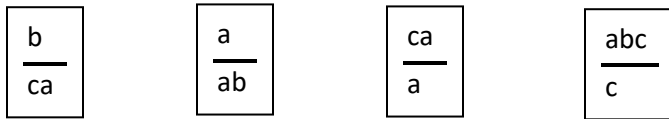
Post's Correspondence Problem

$P = \{[t_1 / b_1], [t_2 / b_2], \dots [t_n / b_n]\}$ // set of tiles with a string on top and a string on bottom

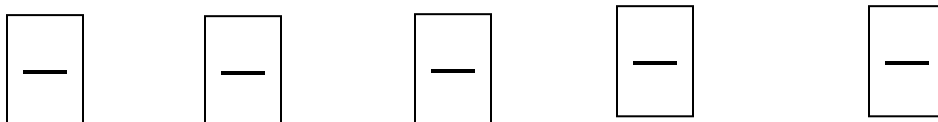
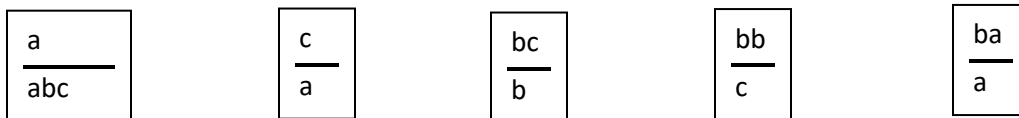
$PCP = \{ \langle P \rangle \mid P \text{ is a set of tiles and there exists a match where the sequence of } t\text{'s match the sequence of } b\text{'s} \}$

Activity 18: PCP Example

Find a match of these tiles. Note that a match does not need to use all tiles and each tile can be used more than once in a match.

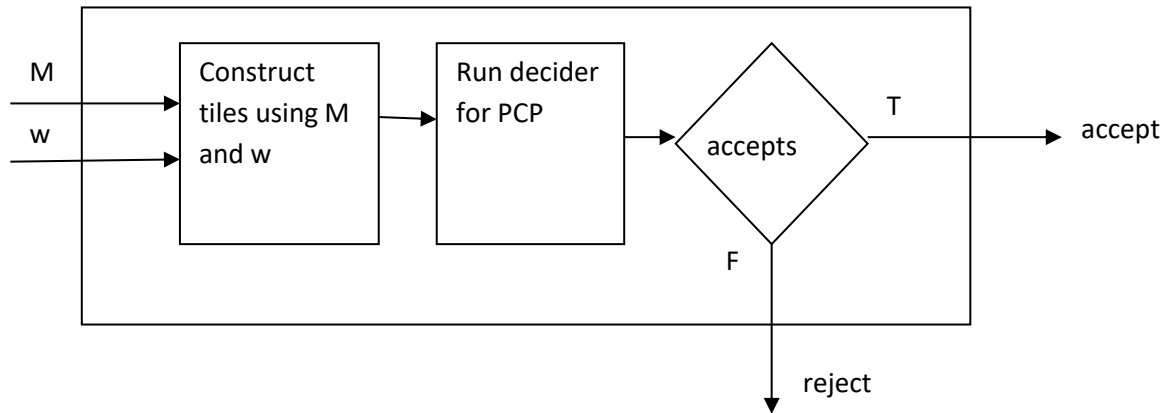


Find a match of these tiles:



Theorem: PCP is undecidable

Proof idea: Reduce from A_{TM} . We'll use computational histories of M on w to construct dominos (similar idea as ALL_{CFG}). The match in the PCP problem will be precisely the series of configurations where M accepts w .



The only difficult step is constructing the tiles given M and w . Instead of constructing tiles for PCP right now, we'll construct tiles for a simpler problem.

MPCP = modified PCP = $\{ \langle P' \rangle \mid P' \text{ is an instance of PCP and the match starts with the first domino listed in } P' \}$

To create dominos for MPCP:

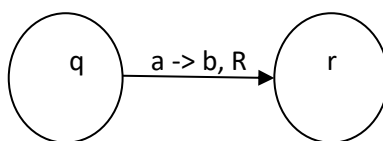
1. If $w = \epsilon$, use $_$ instead of w as the input symbol.
2. Put $[\# / \#q_0w_1w_2w_3\dots w_n\#]$ as the first tile $[t_1 / b_1]$ in P' .

Note: This tile represents the starting configuration of M on w . Note that this tile alone could not create a match since the top just has $\#$ and the bottom as at least $\#q_0w_1\#$.

3. Create tiles to represent TM head motion to the right.

For every $a, b \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{reject}$:
if $\delta(q, a) = (r, b, R)$ put $[qa / br]$ into P' .

Note: This tile represents transitions in the form:

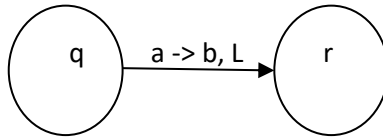


4. Create tiles to simulate head motion to the left.

For every $a, b, c \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$:

If $\delta(q, a) = (r, b, L)$ put $[cqa / rcb]$ into P' for every c in Γ

Note: This tile represents transitions in the form:



5. Create tiles to represent tape locations not adjacent to the head of the TM.

For every $a \in \Gamma$, put $[a / a]$ into P' .

Note: This tile represents characters in strings away from the head of the TM.

6. Create tiles to mark separations between configurations.

Put $[\# / \#]$ and $[\# / _ \#]$ into P' where $_$ represents blank tape cell.

Note: These tiles will delimit each configuration of M on w .

7. Create tiles to represent continuation of a match after M accepts w .

For every $a \in \Gamma$, put $[aq_{\text{accept}} / q_{\text{accept}}]$ and $[q_{\text{accept}}a / q_{\text{accept}}]$ into P' .

Note: These tiles represent continuing to move head to the right after accepting.

8. Create tiles to represent final accept state at the end of the configurations.

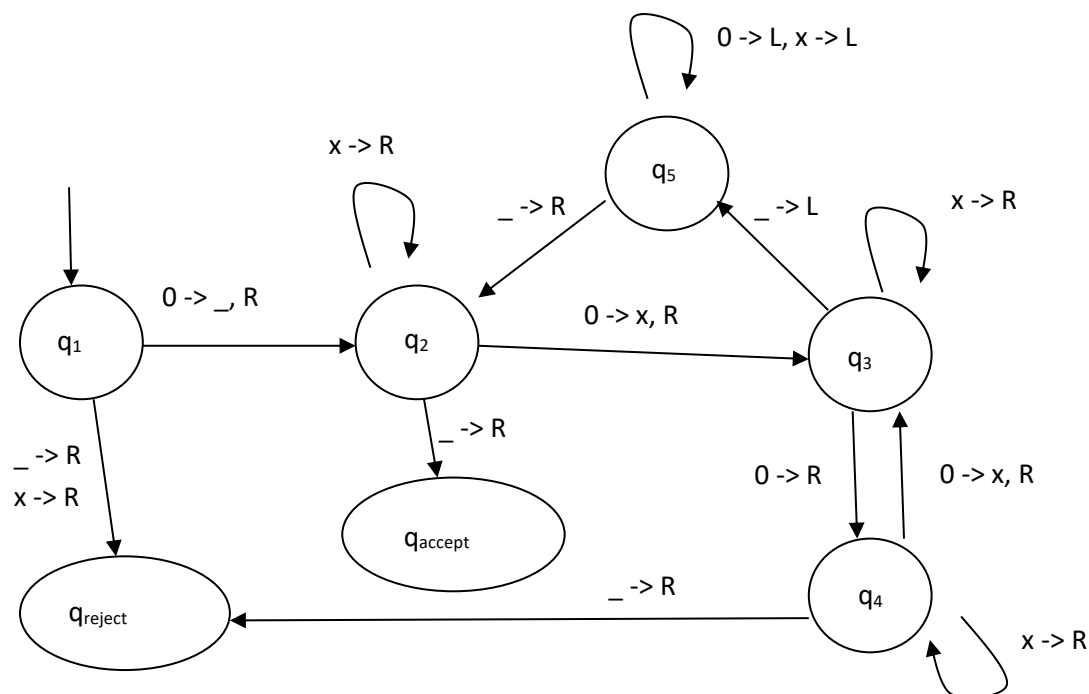
Put $[q_{\text{accept}}\#\# / \#]$ into P' .

Note: This tile is the final tile of a match.

We now have tiles for the MPCP problem.

Let's look at an example of converting a TM M and string w to MPCP tiles.

Assume M is a TM for $\{0^{2^N} \mid N \geq 0\}$:



Let's assume $w = 00$.

Following the construction in the proof:

2.

#
#q ₁ 00#

 // start configuration

3. // transitions to right

<div>q₁0</div> <div>_q₂</div>	<div>q₂x</div> <div>xq₂</div>	<div>q₂0</div> <div>xq₃</div>	<div>q₃x</div> <div>xq₃</div>	<div>q₃0</div> <div>0q₄</div>	<div>q₁_</div> <div>_q_{reject}</div>	<div>q₁x</div> <div>xq_{reject}</div>	<div>q₂_</div> <div>_q_{accept}</div>
<div>q₄x</div> <div>xq₄</div>	<div>q₄0</div> <div>xq₃</div>	<div>q₅_</div> <div>_q₂</div>	<div>q₄_</div> <div>_q_{reject}</div>				

4. // transitions to left

$_q_3_$ $q_5_ _$	$xq_3_$ $q_5x_$	$0q_3_$ $q_50_$
$_q_50$ $q_5_ 0$	xq_50 q_5x0	$0q_50$ q_500
$_q_5x$ $q_5_ x$	xq_5x q_5xx	$0q_5x$ q_50x

5. // tape symbols

0	x	—
0	x	—

6. // delimiters

#	#
#	—#

7. // accepting continuations

$0q_{\text{accept}}$ q_{accept}	xq_{accept} q_{accept}	$_q_{\text{accept}}$ q_{accept}	$q_{\text{accept}}0$ q_{accept}	$q_{\text{accept}}x$ q_{accept}	$q_{\text{accept}}_$ q_{accept}
---	---	--	---	---	--

8. // file tile in match

$q_{\text{accept}}##$
#

So, what's the match? We know M accepts w, so we should be able to find a match.

# #q ₁ 00#	q ₁ 0 _q ₂	0 0	# #	_	q ₂ 0 xq ₃	# _#	_	xq ₃ _	# #	
_q ₅ x q ₅ _ x	_	# #	q ₅ _	x x	_	# #	_	q ₂ x xq ₂	_	# #
_	x	q ₂ _	#	_	x	_q _{accept}	#	_	xq _{accept}	#
_	x	_q _{accept}	#	_	x	q _{accept}	#	_	q _{accept}	#

$_q_{\text{accept}}$	#	$q_{\text{accept}}##$
q_{accept}	#	#

Back to proof:

To convert MPCP to PCP, we will put * markers on the tiles.

In MPCP, suppose the start tile is:

t_1
b_1

Then, we'll create the following tile for PCP:

$*t_1$
$*b_1*$

In MPCP, for all other tiles besides the start tile:

t_k	->	$*t_k$
b_k		b_k*

Create end tile for match in PCP:

$*\$$
$\$$

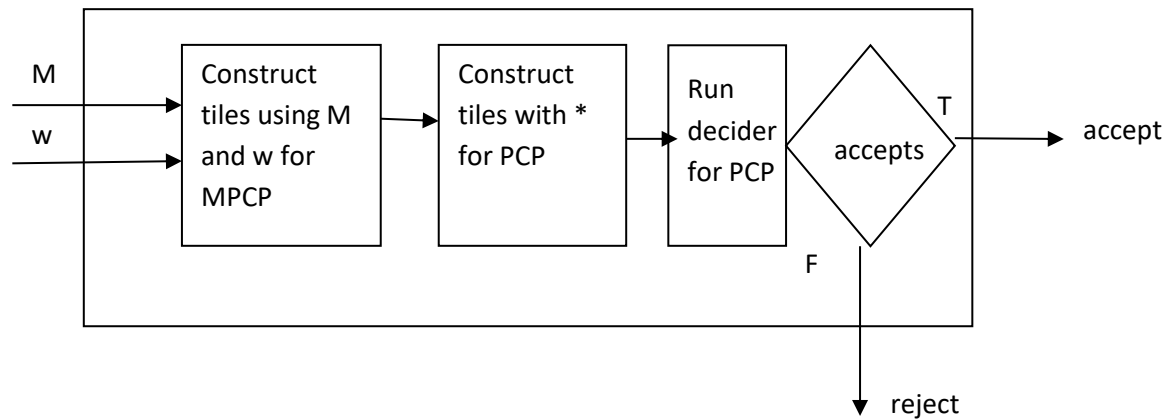
Now, any match must start with

$*t_1$
$*b_1*$

and end with

$*\$$
$\$$

which is exactly the same match we have from MPCP, so the first tile in the match is still the start configuration of M on w.



We have the full proof now. Assume there is a decider D for PCP. Reduce A_{TM} to PCP by constructing tiles for MPCP as above. Then convert the MPCP tiles to PCP tiles as above. If D accepts, the tiles form a match, so M accepts w . If D rejects, the tiles do not form a match, so M does not accept w . We have a contradiction, so PCP must be undecidable.

What other important problems are undecidable?

By Rice's Theorem, any set of machines or computers with a certain set of strings that are accepted or certain behavior when machine/program runs.

Determining the minimum airline trip cost with no restrictions on route or number of stops.

Virus detection – $\{ \langle P \rangle \mid P \text{ is a program that would cause a file on the host computer to be infected} \}$

Note: similar to determining program behavior when it runs

Vulnerability detection – $\{ \langle P \rangle \mid P \text{ is a program such that on some input } w, \text{ running } P \text{ on } w \text{ leads to a security compromise} \}$

Note: determining program behavior when it runs

Determining if a grammar is ambiguous – $\{ \langle G \rangle \mid G \text{ is a CFG and is ambiguous} \}$

Note: reduction from PCP

But, wait, there are companies that sell/produce software that solves these problems!!

- Orbitz, Expedia, Travelocity, etc.
- McAfee, Symantec, etc.
- Fortify Software, Aspect Security, PolySpace Technologies, Secure Software, etc.

What gives?

Undecidable means that there is no program / no algorithm that:

- (a) ALWAYS gives the correct answer and
- (b) ALWAYS halts/terminates

What if we have to give up just one property? Which one is acceptable? Can't really give up (b), so must give up (a). Giving up (a) means that the price may be cheap but not the cheapest for airlines. Giving up (a) means that antivirus software may give the wrong answer (say P is a virus when it is not, say P is safe when it is). We just get it right some of the time.

Next Topic: Complexity Theory
How long does it take a Turing Machine to Decide?

Now we care about how **long a TM takes to decide, given the length of the input string (Complexity)**

This should be review from Data Structures.

Questions:

1. What is the big-O running time of selection sort?
2. What is the big-O running time of merge sort?
3. What is the big-O running time of binary search?

Now, we will do our complexity analysis with respect to the number of steps (transitions) a TM makes in the worst case given input string w with length $|w|$.

$$L = \{a^m b^m \mid m \geq 0\}$$

What could the TM look like?

On input w :

1. Scan across to see if input is in form a^*b^* . If not, reject.
2. While a 's and b 's on tape:
 - a. Scan across, marking an a and b each time
3. If a 's remain after b 's are marked or b 's remain after a 's are marked, reject. If no unmarked symbols found, accept.

How much *time* does the TM take?

definition: Assume M is a TM that is a decider. Then the running time of M is the function $f: \mathbb{N} \rightarrow \mathbb{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of length n .

What is the running time of above TM?

Activity 19: Running Times and Complexity Class P

Definition: Assume M is a TM that is a decider. Then the running time of M is the function $f: \mathbb{N} \rightarrow \mathbb{N}$ where $f(n)$ is the *maximum* number of steps that M uses on any input of length n .

Definition: Big O: Let f and g be functions such that $f: \mathbb{N} \rightarrow \mathbb{R}^+$ and $g: \mathbb{N} \rightarrow \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$, $f(n) \leq c \cdot g(n)$. $g(n)$ is an upper bound for $f(n)$. Intuitively, think of $f(n) \leq g(n)$ if we disregard differences up to a constant factor.

Exercise 1: Assume the TM running time is $f(n) = 4n^3 + 6n^2 + n + 5$. Find $g(n)$ such that $f(n)$ is $O(g(n))$:

Can $g(n)$ be n^2 ?	YES	NO
Can $g(n)$ be n ?	YES	NO
Can $g(n)$ be $n \cdot \lg n$?	YES	NO
Can $g(n)$ be n^3 ?	YES	NO

Definition: Little o gives us a strict asymptotic upper bound.

$f(n) = o(g(n))$ if $\lim_{n \rightarrow \infty} [f(n)/g(n)] = 0$.

$f(n) = o(g(n))$ if for any real number $c > 0$, a number n_0 exists where $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

Exercise 2:

Is $n = o(n \cdot \lg n)$?	YES	NO
Is $5n = o(n)$?	YES	NO
Is $\sqrt{n} = o(n)$?	YES	NO
Is $n \log n = o(n^2)$?	YES	NO

A note about logs: We usually just write \lg in computer science and assume it is base 2, since the base of the log does not matter in terms of asymptotic bounds, since

$\log_b n = (\log_2 n / \log_2 b)$ so the $\log_2 b$ part just becomes a constant, which we disregard.

Activity 19 continued

Exercise 3: The following functions could represent the running times of Turing Machines. It is important to know how to rank these in terms of how quickly (or slowly) the functions grow, so you know if your TM (algorithm) is fast or slow.

Arrange the functions so that they are in order from slowest-growing (best running time) to fastest-growing (worst running time).

$$2^{\lg n}$$

$$n^3$$

$$n$$

$$n \lg n$$

$$2^n$$

$$1$$

$$\text{sqrt}(n)$$

$$n!$$

$$\lg(\lg n)$$

$$\lg n$$

$$n^n$$

$$n * 2^n$$

$$n^2$$

$$\lg(n!)$$

Activity 20: Re-Consider $L = \{a^m b^m \mid m \geq 0\}$

1. Can you find a single-tape deterministic TM that decides faster than $O(N^2)$ where $m+m = N$?
2. Assume your TM can have two tapes. How fast can you make a TM for this language?

Notice that **in computability theory**, a single tape TM and a multitape TM had the same computability power – both can be used to show a language is decidable or recognizable. But, **there's a difference in complexity theory!!**

Class P

P = class of languages that can be decided in polynomial time on a **deterministic single tape** TM.

$$P = \bigcup_k \text{TIME}(n^k).$$

- we care about P because these are the problems that are *efficiently solved* by a computer.

Show $\text{PATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t \}$ is in P.

To do this, we must show 2 things:

- 1) PATH is decidable
- 2) PATH's TM runs in poly time

Create TM M that decides PATH.

On input $\langle G, s, t \rangle$:

1. Check that $\langle G, s, t \rangle$ is in proper form. If not, reject.
2. Mark node s in vertex list of G
3. Repeat until no new nodes are marked:
 - a. 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. Otherwise, reject.

Now consider the running time. Let n be the number of vertices and m be the number of edges, so input string length is $n + m + 2$. Call the input string length N .

Running time:

1. $O(N)$ to check format
2. $O(N)$ to search for and mark vertex s in the vertex list
3. $O(N \cdot N)$ to mark new nodes in vertex list (for each node, need to scan edge set)
4. $O(N)$ to search for and find vertex t in the vertex list

Overall running time is: $O(N^2)$

Because there is a deterministic single-tape TM that decides PATH in $O(N^2)$ time, PATH is in P.

Activity 21: Show that $\text{CONNECT} = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \}$ is in P.

Create TM M that decides CONNECT. Hint: this is similar to PATH.

Running-time of the TM with respect to length N of the input string:

Theorem: P is closed under union.

Proof: Assume there are two languages, A and B, that in complexity class P. Because they are in P, there exists a deterministic TM M1 that decides A in polynomial time N^k . There exists a deterministic TM M2 that decides B in polynomial time N^p . We can create a deterministic TM M3 that decides $A \cup B$ as follows:

On input w:

1. Run M1 on w. If it accepts, accept. If not, go to step 2.
2. Run M2 on w. If it accepts, accept. If not, reject.

Running time: The runtime of step 1 is N^k . The runtime of step 2 is N^p . The total runtime is $N^k + N^p = N^{\max(k,p)}$. So, the runtime of M3 is polynomial. So, $A \cup B$ is in complexity class P.

A_{CNF}

Theorem: Every CFL is in P.

Proof idea: Convert CFL to CNF form (poly time algorithm: 4 steps, each step modifies at most all rules). Then decide $A_{CNF} = \{ \langle G, w \rangle \mid G \text{ is a grammar in CNF that generates } w \}$ in poly time with dynamic programming.

TM for A_{CNF} :

On input $\langle G, w \rangle$ where G is in CNF:

1. If $w = \epsilon$ and $S \rightarrow \epsilon$, accept.
2. For $i = 1$ to n : // fill in diagonal entries (representing substrings of length 1)
 - a. For each variable A :
 - i. test whether $A \rightarrow w_i$ is a rule. If so, put A in entry $T(i,i)$
3. For $L = 2$ to n : // L is the length of the substring
 - a. For $i = 1$ to $n-L+1$: // i is first position in substring
 - i. Let $j = i + L - 1$ // j is last position in substring
 - ii. For $k = i$ to $j-1$: // k is the split position of the two substrings
 1. For each rule $A \rightarrow BC$
 - a. If $T(i,k) = B$ and $T(k+1,j) = C$, put A in $T(i,j)$
4. If S is in $T(1,n)$, accept. Otherwise, reject.

Example:

$S \rightarrow SF \mid a$

$F \rightarrow AS$

$A \rightarrow CG \mid SS \mid CS \mid a$

$G \rightarrow CA$

$C \rightarrow b$

$w = abaa$

Step 1: w is not ϵ , so keep going.

Step 2: fill in diagonals:

	1	2	3	4
1	S, A			
2		C		
3			S, A	
4				S, A

$T(1,1) = S, A$ since $w_1 = a$ and $S \rightarrow a$ and $A \rightarrow a$

$T(2,2) = C$ since $w_2 = b$ and $C \rightarrow b$

...

Step 3:

For $L = 2$ to 4

 For $i = 1$ to $n-L+1$

$j = i+L-1$

 For $k = i$ to $j-1$

L	i	J	k	Action
2	1	2	1	$T(1,1) = S/A$, $T(2,2) = C$ so there is no rule \rightarrow SC or AC
2	2	3	2	$T(2,2) = C$, $T(3,3) = S/A$. Since $A \rightarrow CS$, $T(2,3) = A$ and $G \rightarrow CA$, $T(2,3) = G$
2	3	4	3	$T(3,3) = S/A$, $T(4,4) = S/A$. Since $A \rightarrow SS$, $T(3,4) = A$. Since $F \rightarrow AS$, $T(3,4) = F$.
3	1	3	1	$T(1,1) = S/A$, $T(2,3) = A/G$ so there is no rule
3	1	3	2	$T(1,2) = \text{empty}$, $T(3,3) = S/A$ so there is no rule
3	2	4	2	$T(2,2) = C$, $T(3,4) = A/F$. Since $G \rightarrow CA$, $T(2,4) = G$
3	2	4	3	$T(2,3) = A/G$, $T(4,4) = S/A$. Since $F \rightarrow AS$, $T(2,4) = F$
4	1	4	1	$T(1,1) = S/A$, $T(2,4) = G/F$. Since $S \rightarrow SF$, $T(1,4) = S$
4	1	4	2	$T(1,2) = \text{empty}$, $T(3,4) = A/F$ so there is no rule
4	1	4	3	$T(1,3) = \text{empty}$, $T(4,4) = S/A$ so there is no rule
DONE				

Table:

	1 a	2 b	3 a	4 a
1	S, A			S
2		C	A, G	G, F
3			S, A	A, F
4				S, A

Step 4: Since S is in $T(1,4)$, we accept.

Activity 22: Practice with filling in table for A_CNF

1. Try the algorithm for $w = aaab$. Complete the table below.

G:

$S \rightarrow SF \mid a$

$F \rightarrow AS$

$A \rightarrow CG \mid SS \mid CS \mid a$

$G \rightarrow CA$

$C \rightarrow b$

		1	a	2	a	3	a	4	b
1	a								
2	a								
3	a								
4	b								

2. Try the algorithm for $w = bbab$. Complete the table below.

		1	b	2	b	3	a	4	b
1	b								
2	b								
3	a								
4	b								

3. Try the algorithm for $w = abaa$. Create the table below. Note the dimensions of the table.

Class NP, Nondeterministic Polynomial Time

$\text{NTIME}(t(n)) = \{L \mid L \text{ is decided in time } O(t(n)) \text{ by a nondeterministic TM}\}$

$\text{NP} = \bigcup_k \text{NTIME}(O(n^k))$

So, languages that can be decided by a ND TM in poly time are in the complexity class NP. Another way to look at NP is the following:

verifier: A verifier for a language A is an algorithm V, where

$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$

What does this mean? Given a certificate c (solution), we can verify that w is in A. A polynomial time verifier runs in poly time with respect to the length of w. So, if A has a poly time verifier, it is in NP.

Example of verifier for HAMPATH:

N': On input $\langle \langle G, s, t \rangle, c \rangle$: // c is the list of ordered vertices for the path

1. Test whether c is a list of all vertices in G. If not, reject.
2. Test whether successive pairs in c are edges in G. If not, reject.
3. If both 1 and 2 pass, accept.

Running time:

1: $O(n)$ // if dictionary

2: $O(n)$ // if dictionary

3: $O(1)$

So, here's how I think of it: The certificate c is a potential solution for the input to be in the language. So, given a potential solution, can we see if it is correct in poly time?

Clique is in NP

$\text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}$

[Note: a clique of size k is a subset of k vertices such that there are edges between all pairs of the k vertices]

Show that CLIQUE is in NP by (a) building a ND TM and (b) building a poly time verifier.

(a) ND TM:

On input $\langle G, k \rangle$:

1. ND select a subset G' of k nodes
2. Test if G contains edges connecting all pairs of nodes in G'
3. If so, accept. If not, reject.

time:

1. $O(|V|)$ // select k from $|V|$ vertices
2. $O(k^2 * |E|)$ // k^2 pairs of vertices and check each against list of edges
3. $O(1)$

So, it is poly time with respect to the length of the input.

(b) Verifier V :

On input $\langle \langle G, k \rangle, c \rangle$:

1. Test if c is a subset of k nodes of G
2. Test whether G contains edges connecting pairs of nodes in c
3. If both 1 and 2 pass, accept. If not, reject.

time:

1. $O(|V|)$ // check if there are k nodes in c and k are distinct
2. $O(k^2 * |E|)$
3. $O(1)$

Activity 23: Subset Sum is in NP

$\text{SUBSET_SUM} = \{ \langle S, t \rangle \mid S = \{x_1, x_2, \dots, x_k\} \text{ and for some } \{y_1, y_2, \dots, y_j\} \text{ that is a subset of } S, \text{ we have the sum of } y_i\text{'s is equal to } t\}$

Let $S = \{2, 6, 12, 7, 9, 4, 6, 5\}$

$t = 32$

Is $\langle S, t \rangle$ in SUBSET_SUM? yes. $12 + 9 + 7 + 4$

Show that SUBSET_SUM is in NP.

a) Create solution with ND TM. Remember to analyze its runtime.

On input $\langle S, t \rangle$:

b) Create solution with a polynomial time verifier

On input $\langle \langle S, t \rangle, c \rangle$: // c is a subset of S

Questions and Reductions

Consider the complement of SUBSET_SUM. Is this in NP? We would need to verify that a subset is not present, which is harder to verify that a subset is present.

complexity class: $\text{coNP} = \{L \mid \text{the complement of } L \text{ is in NP}\}$.

We don't know if coNP is different than NP.

Is $P = NP$?

We don't yet know, but we believe they are different. No poly-time deterministic algorithms have been found for languages in NP and there are lots of languages in NP.

Back to reducibility: now we'll use the technique to show a language is in a certain complexity class.

Definition: $f: \Sigma^* \rightarrow \Sigma^*$ is a poly time computable function if some poly time TM M exists that halts with $f(w)$ on the tape given w as input.



$A \leq_p B$: A is poly time reducible to B if a poly time computable function exists where for every w , w is in A if and only if $f(w)$ is in B .

Theorem: If $A \leq_p B$ and B is in P , then A is in P .

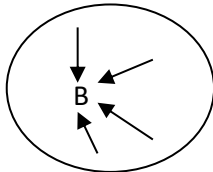
Proof: Assume B is in P with poly time TM M . Assume A is poly time reducible to B with TM N . Then create a poly time TM O for A that does the following: Runs N on w to produce $f(w)$. Runs M on $f(w)$. If M accepts, accept. If not, reject. Time: N runs in poly time and M runs in poly time, so the sequence of the machines is poly time.

NP-Complete Languages

Definition: A language B is NP-complete if:

1. B is in NP
2. Every A in NP is poly time reducible to B // this property is called NP-hard

Class NP:



Why is this important? If we could solve B in poly time, then we could solve every other language in NP in poly time!!

Theorem: If B is NP-complete, C is in NP, and $B \leq_p C$, then C is NP-complete.

All \rightarrow B -----(poly time)--- \rightarrow C

Once we know a problem is NP-complete, we can use it as language B to show a reduction from B to C to show C is NP-complete. (kind of like undecidability...there we kept adding to our collection of undecidable languages and could use any one of them to create a reduction)

But, how do we find the “first” NP-complete problem/language....

SAT is NP-complete

$SAT = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula} \}$

Cook-Levin Theorem: SAT is NP-complete.

Proof:

We want to show that SAT is NP-complete. To do this, we must show 2 things:

1. SAT is in NP
2. Every language A in NP is poly time reducible to SAT

To show 1:

Build a ND TM N: On input $\langle \phi \rangle$:

1. ND choose truth values for each variable
2. Check if ϕ is true with the assignment of truth values
3. If 2 passes, accept. Otherwise, reject.

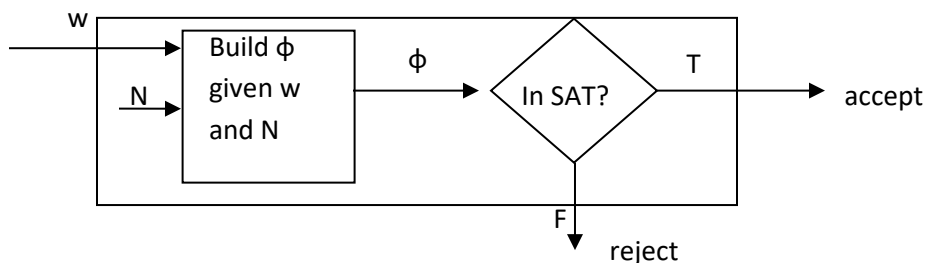
Running time:

1. $O(n)$
2. $O(n)$
3. $O(1)$

To show 2:

Let A be in NP. Then, A has a ND TM N that decides A in $O(n^k)$ time for some constant k.

We'll reduce A to SAT.



In other words, we are creating a function $f: A \rightarrow SAT$ such that w is in A if and only if $f(w)$ is in SAT.

Idea: We will use configurations of N on w on some branch of computation to create a Boolean formula.

Definition of tableau:

```

# q0w1w2w3..... _ _ _ _ _ #
#w1q1w2w3..... _ _ _ _ _ #
.....
...
#                               #
  
```

The height of this tableau is order n^k since we know the TM runs in poly time on some branch. The width of the table is n^k (we'll just pad _ after each configuration). The first row is the starting configuration of the branch of N on w . Each successive row is the next configuration of the machine on w . If any row contains q_{accept} , this is an accepting tableau. So, the problem of determining if N accepts w can be turned into "Does an accepting tableau for N on w exist?"

We'll convert the entries of the tableau into variables.

Let C = collection of tableau symbols = $Q \cup \Gamma \cup \{\#\}$

Boolean variables: $x_{i,j,s} = 1$ if $\text{cell}[i,j] = s$
 $x_{i,j,s} = 0$ if $\text{cell}[i,j] \neq s$

$$\Phi = \phi_{\text{cell}} \&\& \phi_{\text{start}} \&\& \phi_{\text{move}} \&\& \phi_{\text{accept}}$$

1. ϕ_{cell} represents that each cell contains exactly one symbol.

$$A_{ij} = x_{i,j,s1} \mid \mid x_{i,j,s2} \mid \mid x_{i,j,s3} \dots \mid \mid x_{i,j,s|C|} \quad \begin{array}{l} // |C| \text{ is the number of different tableau symbols} \\ // \text{ this expression handles at least one symbol is} \\ // \text{ in cell}[i,j] \end{array}$$

$$B_{ij} = (!x_{i,j,s1} \mid \mid !x_{i,j,s2}) \&\& (!x_{i,j,s1} \mid \mid !x_{i,j,s3}) \dots \&\& (!x_{i,j,s|C|-1} \mid \mid !x_{i,j,s|C|})$$

// includes all pairs where at most one symbol is on
// note that if the cell has two symbols, then the
// corresponding clause is false, causing B_{ij} to be false

$$\Phi_{\text{cell}} = \bigwedge_{\substack{1 \leq i \leq n^k \\ 1 \leq j \leq n^k}} (A_{ij} \&\& B_{ij})$$

2. Φ_{start} will represent the starting configuration in the top row.

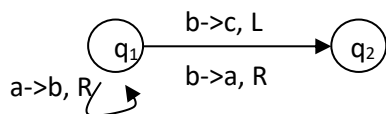
$$\Phi_{\text{start}} = x_{1,1,\#} \&\& x_{1,2,q_0} \&\& x_{1,3,w_1} \&\& \dots \&\& x_{1,n+2,w_n} \&\& x_{1,n+3,_} \&\& \dots \&\& x_{1,n^k,\#}$$

3. Φ_{accept} will represent an accepting configuration, so q_{accept} is anywhere in the row.

$$\Phi_{\text{accept}} = \bigvee_{\substack{1 \leq i \leq n^k \\ 1 \leq j \leq n^k}} x_{i,j,q_{\text{accept}}}$$

4. Φ_{move} will represent legal TM moves.

Assume we have the following transition in the ND TM N:



What are the legal tableau windows of size 3 x 2?

following $b \rightarrow c, L$

a	q ₁	b
q ₂	a	c

following $b \rightarrow a, R$

a	q ₁	b
a	a	q ₂

following $a \rightarrow b, R$

a	a	q ₁
a	a	b

on left side of tableau

#	b	a
#	b	a

following $b \rightarrow c, L$

a	b	a
a	b	q ₂

following $b \rightarrow c, L$

b	b	b
c	b	b

Many more, but this serves as examples.

$$\Phi_{\text{move}} = \bigwedge_{\substack{1 \leq i < n^k \\ 1 \leq j < n^k}} L_{ij}$$

L_{ij} = 3x2 legal window with top-middle location at row i and column j .

$$L_{ij} = x_{i,j-1,a1} \ \&\& \ x_{i,j,a2} \ \&\& \ x_{i,j+1,a3} \ \&\& \ x_{i+1,j-1,a4} \ \&\& \ x_{i+1,j,a5} \ \&\& \ x_{i+1,j+1,a6}$$

// where window is:

a1	a2	a3
a4	a5	a6

That concludes the process for creating ϕ .

Now, we need to determine that this function to map A to SAT is polynomial time. Let's look at each part separately.

1. ϕ_{cell} : The number of variables in A_{ij} is $|C|$. The number of variables in B_{ij} is $2|C|^2$. But C is constant size set, depending on the TM only and not on the length of w . Given the size of the tableau is n^{2k} , then the size of ϕ_{cell} is $O(n^{2k}) = O(n^{2k})$.

2. ϕ_{start} : The number of variables is the size of the top row, which is $O(n^k)$.

3. ϕ_{accept} : The number of variables is the size of the tableau, which is $O(n^{2k})$.

4. ϕ_{move} : The number of variables in each window is 6 and the number of windows is $O(n^{2k})$.

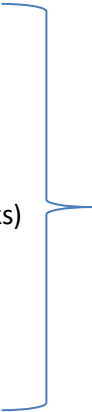
The total running time to convert language A with TM N to a Boolean formula is $O(n^{2k})$ which is polynomial time.

Since we have shown that SAT is in NP and there is a poly time reduction from any A in NP to SAT, SAT is NP-complete.

Review: To show a language is NP-complete

In general, to show A is NP-complete:

1. Show A is in NP (build a ND TM that decides in poly time)
2. Given B is NP-complete, show that $B \leq_p A$.
 - a. Provide function $F: B \rightarrow A$
 - b. Show that if w is in B, $F(w)$ is in A
 - c. Show that if $F(w)$ is in A, w is in B (b and c show reduction works)
 - d. Show that F is a poly time function



Must do all parts for a complete NP-complete proof

Coming up with the functions to map B to A can take a lot of creativity and clever thinking, especially when the languages involve different types of structures (such as Boolean functions, graphs, integers).

3SAT is NP-Complete

Earlier, we showed that SAT is NP-complete. SAT is a language of Boolean formulas that can have variables connected with ANDs and ORs and variables can also be negated. 3SAT is a language where the formulas have a specific structure. The specific structure is helpful in showing that other languages are NP-complete.

3SAT: Boolean formula that is satisfiable in 3CNF (conjunctive normal form)
3 is for the number of variables (regular or negated) per CLAUSE
Each CLAUSE has variables connected via OR
Each CLAUSE is connected to other clauses via AND

Example of format of a 3SAT instance:

$(x \vee y \vee \neg z) \wedge (\neg x \vee z \vee \neg y)$

What can we set x , y , and z to (true or false) such that the formula is satisfiable?

$x = \underline{\hspace{1cm}}, y = \underline{\hspace{1cm}}, z = \underline{\hspace{1cm}}$

To show 3SAT is NP-complete:

1. We need to show that 3SAT is in complexity class NP
Easy – we use same TM that we created for SAT
2. Build a poly function F : SAT \rightarrow 3SAT
Turn any Boolean formula into the constrained 3-cnf form

Step 1: Apply distribution laws to get into conjunctive normal form

Example: $a \vee (b \wedge c) \rightarrow (a \vee b) \wedge (a \vee c)$

Example: $(a \wedge b) \vee (c \wedge d) \rightarrow ((a \wedge b) \vee c) \wedge ((a \wedge b) \vee d) \rightarrow$
 $((a \vee c) \wedge (b \vee c)) \wedge ((a \vee d) \wedge (b \vee d))$

Step 2: Get 3 variables per clause

Case 1: if clause is (x) [just one variable], convert to $(x \vee x \vee x)$

Case 2: if clause is $(x \vee y)$, convert to $(x \vee y \vee y)$

Case 3: if clause $(x \vee y \vee z)$, keep

Case 4: if clause has more than three variables $(a_1 \vee a_2 \vee \dots \vee a_k)$ becomes
 $(a_1 \vee a_2 \vee y_1) \wedge (\neg y_1 \vee a_3 \vee y_2) \wedge (\neg y_2 \vee a_4 \vee y_3) \wedge \dots$
 $(\neg y_{k-3} \vee a_{k-1} \vee a_k)$

These transformations did not change the satisfiability of the formula, so the original formula in SAT is satisfiable if and only if the transformed formula in 3SAT is satisfiable.

F runs in poly time:

Step 1: apply distribution law takes $O(N^2)$ time

Step 2: cases 1/2/3 take constant time per clause. Case 4 takes $O(N^2)$ time

Total runtime is $O(N^2)$.

Since we have a poly time transformation from SAT to 3SAT and we know that 3SAT is in NP, 3SAT is NP-complete.

Activity 24: 3SAT practice

Part A: Are the following Boolean formulas in 3SAT **format**?

$(x \vee y \vee \neg x) \wedge (z \vee a \vee a)$	yes	no
$(x \vee y) \wedge (\neg x \vee \neg y \vee z)$	yes	no
$x \wedge \neg y \vee z$	yes	no
$x \vee y \vee z \wedge a \vee (b \wedge \neg c)$	yes	no

Part B: Are the following Boolean formulas in the 3SAT **language**?

$(x \vee y \vee \neg x) \wedge (\neg y \vee \neg y \vee \neg y)$	yes	no
$(x \vee x \vee x) \wedge (\neg x \vee \neg x \vee \neg x)$	yes	no
$(x \vee y) \wedge (\neg x \vee y \vee z)$	yes	no

Part C: Use the function to convert SAT Boolean formula to equivalent 3SAT Boolean formula shown below.

Original Boolean formula format for SAT:

$$(a \vee b \vee c \vee \neg d) \wedge (\neg c \vee d \wedge e)$$

Remember levels of operations: negation before AND before OR, so you may want to parenthesize the expression to help with order of operations.

Convert to Boolean formula format for 3SAT: (do distribution before making 3 per clause)

SUBSET_SUM is NP-complete

$\text{SUBSET_SUM} = \{ \langle S, t \rangle \mid S \text{ is a set of integers such that there is a subset } S' \text{ of } S \text{ where members of } S' \text{ add to } t \}$

Show that SUBSET_SUM (SS) is NP-complete.

Proof:

1. We showed SS is in NP during lecture earlier. But, as a review, we can ND choose a subset S' of S . Add the numbers together and accept if they add to t . $O(n)$.
2. We need to show some NP-complete problem has a poly time reduction to SS. We'll choose 3SAT to use in the reduction. Idea: We'll create two integers for each variable in Φ and we'll rig it so only one of the two integers can participate in the subset of S that sums to t .

$F(\Phi) =$

Let L = number of variables in Φ . Let K = number of clauses in Φ .

1. For each variable x_i in Φ create two numbers y_i and z_i to include in set S . Note that the numbers are written in decimal (left to right) and the 0/1 means that the value is 1 for y_i if x_i is in clause c_j and the value is 1 for z_i if $\neg x_i$ is in clause c_j .
2. For each clause, we create two numbers g_j and h_j where the clause position has a 1 for clause c_j and a 0 everywhere else.

	1	2	3	4	...	L	c_1	c_2	c_3	...	c_K
y_1	1	0	0	0		0	0/1	0/1	0/1		0/1
z_1	1	0	0	0		0	0/1	0/1	0/1		0/1
y_2	0	1	0	0		0	0/1	0/1	0/1		0/1
z_2	0	1	0	0		0	0/1	0/1	0/1		0/1
y_3	0	0	1	0		0	0/1	0/1	0/1		0/1
z_3	0	0	1	0		0	0/1	0/1	0/1		0/1
...											
y_L	0	0	0	0		1	0/1	0/1	0/1		0/1
z_L	0	0	0	0		1	0/1	0/1	0/1		0/1
<hr/>											
g_1	0	0	0	0		0	1	0	0		0
h_1	0	0	0	0		0	1	0	0		0
g_2	0	0	0	0		0	0	1	0		0
h_2	0	0	0	0		0	0	1	0		0

...										
g_k	0	0	0	0		0	0	0	0	1
h_k	0	0	0	0		0	0	0	0	1

3. Create t to be:

1 1 1 1 ... 1 3 3 3 3

Here is an example using this function on a specific Boolean formula.

Let $\Phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_1) \wedge (x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee \neg x_1)$

The numbers created are:

	1	2	3	c_1	c_2	c_3	c_4
y_1	1	0	0	1	1	1	0
z_1	1	0	0	0	0	1	1
y_2		1	0	0	1	1	1
z_2		1	0	1	0	0	0
y_3			1	1	0	0	1
z_3			1	0	1	0	0
g_1				1	0	0	0
h_1				1	0	0	0
g_2					1	0	0
h_2					1	0	0
g_3						1	0
h_3						1	0
g_4							1
h_4							1
$t =$	1	1	1	3	3	3	3

We need to show the function F is such that Φ is in 3SAT if and only if $F(\Phi)$ is in SS.

-> Assume Φ is satisfiable. For each variable x_i in Φ , choose y_i if x_i is assigned true in the satisfying assignment and choose z_i if x_i is assigned false. The left L columns of numbers should sum to 1 for each. In the upper-right hand part of the table, each of the right K columns sums between 1 and 3 since 1 to 3 variables are true in the satisfying assignment in Φ . Select enough of the g_j and h_j integers to have the K right-most columns each sum to 3. Thus, there is a subset of numbers that sums to t .

<- Assume $F(\Phi)$ is in SS. So, there is a subset S' of integers that sums to t . Since at most 5 1's appear in any column, there is no carry-over with the addition. To get a 1 in the first column, either y_1 or z_1 is in S' (but not both). To get a 1 in the second column, either y_2 or z_2 is in S' (but not both). Etc for the left L columns. If the y_i is in S' , set the variable x_i to true in Φ . If the z_i is in S' , set the variable x_i to false in Φ . Because we

use at most g_j and h_j in the subset S' , there must be at least one 1 for each of the right K columns from the y and z integers. Therefore, at least one variable is assigned true per clause, so Φ is satisfiable.

Now, we need to show the function F runs in polynomial time.

1 and 2. Creating the y_i and z_i integers and g_j and h_j integers: The size of the table is $(L + K) * (2L + 2K)$ which is $O(n^2)$.

3. Creating t is size $(L+K)$ which is $O(n)$.

Therefore, F runs in polynomial time with respect to the length of Φ .

Activity 25: Apply SUBSET SUM transformation

As defined earlier:

SUBSET_SUM = {<S, t> | S is a set of integers such that there is a subset S' of S where members of S' add to t}

Let $F: 3SAT \rightarrow SUBSET_SUM$ be the function defined in the coursepack and textbook.

Show the result of applying F to the following boolean formula:

$(!x1 \vee x2 \vee !x3) \wedge (!x2 \vee x3 \vee x4) \wedge (x1 \vee !x2 \vee !x4)$

Set S: (hint: make a table)

Target number t: (hint: really large decimal number)

Which numbers form the subset of S that sum to t?

VERTEX_COVER is NP-complete

$\text{VERTEX_COVER} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover} \}$

A vertex cover is a subset of nodes where every edge touches at least one of the nodes in the cover.

Theorem: VERTEX_COVER (VC) is NP-complete.

To show VC is in NP:

Build a nondeterministic TM N:

On input $\langle G, k \rangle$:

1. ND select k vertices of G and call set V'
2. Test if every edge of G contains a vertex in V'
3. If 1 and 2 pass, accept

Running time:

1. $O(n)$
2. $O(n^2)$ if the edges and vertices V' are both lists
3. $O(1)$

Thus, VC is in NP.

To show $3\text{SAT} \leq_p \text{VC}$:

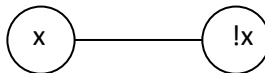
Idea: Need function $F: \Phi \rightarrow \langle G, k \rangle$

Nodes will be variables of Φ and edges between x and $\neg x$ will ensure that one or the other will be chosen for the vertex cover. The vertex cover will correspond to the true variables and the two variables per clause that could be false. Clauses will be grouped together as triangles to ensure that two of the three nodes are selected for the cover.

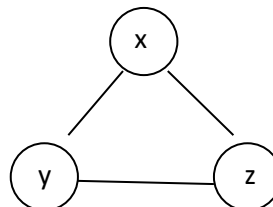
$F(\Phi) =$

1. Let $k = |V| + 2|C|$ where V is the set of variables in Φ and C is the set of clauses in Φ
2. Construct the nodes/edges in G as follows:

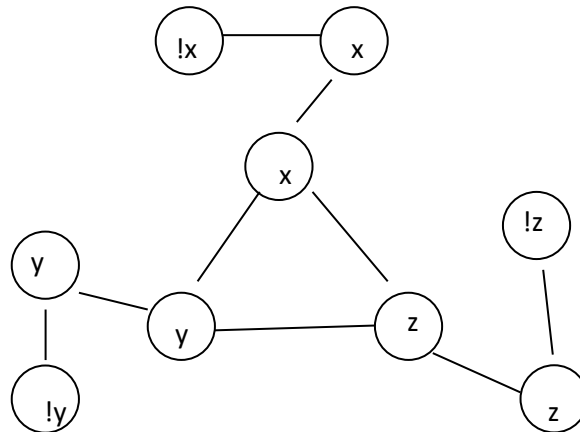
- a. For each variable x in V , create



- b. For each clause $(x \vee y \vee z)$ in C , create

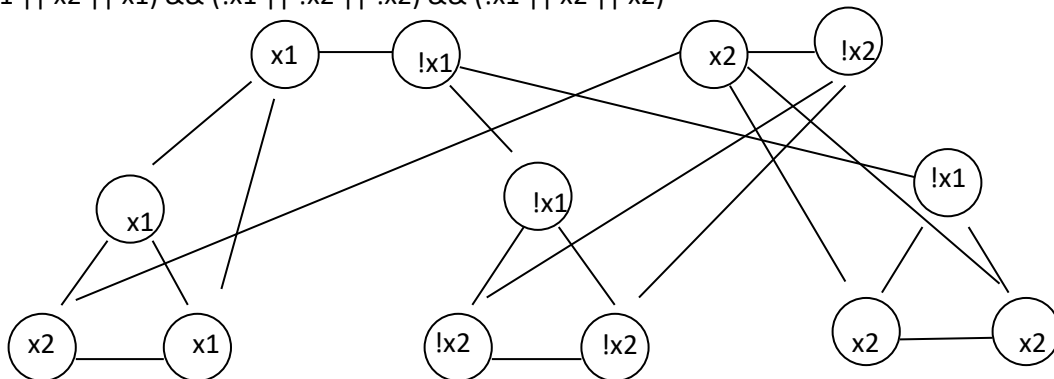


- c. Add edges connecting the clause nodes (in b) to the corresponding nodes in the variable pairs (in a).



Here is an *example construction* of G and k for a particular Boolean formula:

$$\Phi = (x_1 \vee x_2 \vee x_1) \wedge (!x_1 \vee !x_2 \vee !x_2) \wedge (!x_1 \vee x_2 \vee x_2)$$



$$k = 2 + 2 \cdot 3 = 8$$

For this example, the VC = {!x₁ (from variable gadget), x₂ (from variable gadget), x₁, x₁ (from clause gadget 1), !x₂, !x₂ (from clause gadget 2), !x₁, x₂ (from clause gadget 3)}. Note that the assignment of x₁ = 0, and x₂ = 1 satisfies the original formula.

Now, we need to show that F is a function such that Φ is satisfiable if and only if $F(\Phi)$ is in VC. Thus, we need to show both directions for the if and only if.

-> Assume Φ is satisfiable. Put nodes corresponding to true variables of the paired nodes (either x or $!x$) into the vertex cover. Put the remaining two nodes from each clause not adjacent to the chosen variable nodes into the vertex cover. Thus, there are $|V| + 2|C|$ nodes in the vertex cover. Then, $\langle G, k \rangle$ is in VC.

<- Assume $F(\Phi)$ is in VC. The cover must include at least one node in each variable gadget to cover the edge between x and $\neg x$. Two nodes from each clause triangle must be in the vertex cover to cover the three edges. Just from these nodes alone, there are $|V| + 2|C|$ nodes already in the cover, so we cannot add any more. Now take the clause gadgets and assign true to the variable not selected for the cover (since this node is attached to the variable gadget). This assignment of the variables from the variable gadget nodes in the vertex cover produces a satisfying assignment. Thus, Φ is in 3SAT.

Now, we need to consider the running time of F :

1. $k = |V| + 2|C|$ so this is $O(n)$

2a. For each variable in Φ , we create 2 nodes and 1 edge. $O(n)$

2b and 2c. For each clause in Φ , we create 3 nodes and 6 edges (3 for the triangle and 3 to the variable gadgets). $O(n)$

Therefore, we have a poly time function from SAT to VC, so VC is NP-complete.

Activity 26: Apply Vertex Cover Transformation

VERTEX_COVER = { $\langle G, k \rangle$ | G is an undirected graph that has a k -node vertex cover}

A vertex cover is a subset of nodes where every edge touches at least one of the nodes in the cover.

Assume $F: 3SAT \rightarrow VERTEX_COVER$ is the function shown above and the textbook.

Apply F to the following Boolean formula:

$(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_4)$

Draw G here:

$k =$ _____

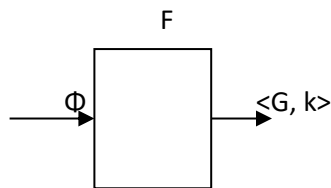
This $\langle G, k \rangle$ is the string for the language **VERTEX_COVER**. Highlight the nodes that form a k -vertex cover.

CLIQUE is NP-complete

$\text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}$

Proof:

- 1) We showed earlier in class that CLIQUE is in NP by either constructing a ND TM to decide in poly time (ND choose k nodes, see if all k are connected to the other nodes) ($O(n^3)$) OR constructing a poly time verifier: Given $\langle G, k, c \rangle$: see if c contains k nodes of G , check to see that all c are connected to the others. ($O(n^3)$)
- 2) Now, we need to show that CLIQUE is NP-hard. We do this by reducing a different NP-complete problem to CLIQUE. Well, the only ones we know about right now are SAT and 3SAT. We'll use 3SAT.



We need to create a function F that takes Φ and creates a graph G and number k such that if Φ is satisfiable, then G has a k -clique and if Φ is not satisfiable, then G does not have a k -clique.

$F(\Phi) =$

Let k be the number of clauses in Φ .

Let G be a graph with k groups of 3 nodes each where each group corresponds to one clause in Φ . Each node of G corresponds to a variable in Φ . Put edges in G between each pair of nodes such that:

- 1) No edge is present between variables in the same clause
- 2) No edge is present between any two nodes x and $\neg x$

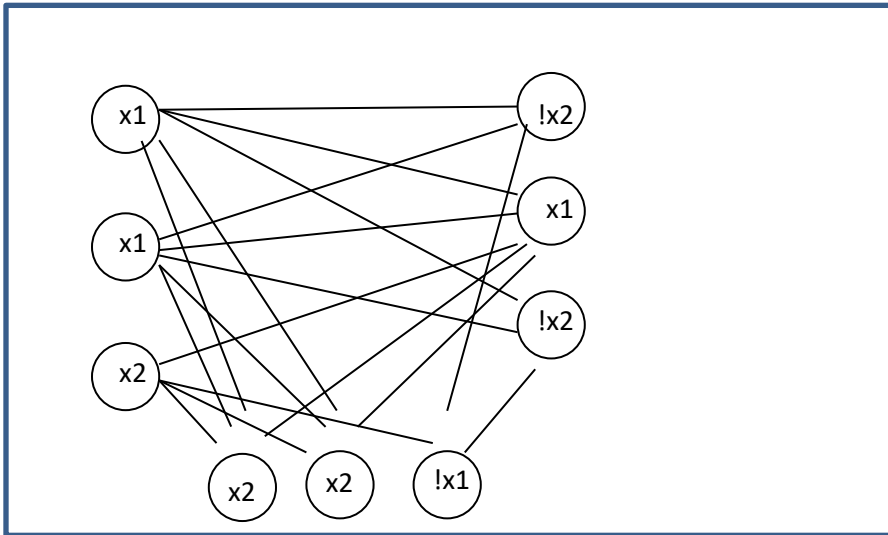
box is not part of formal proof, but shows example of reduction on a particular Boolean formula

Here is an example construction of G and k for a particular Boolean formula:

$\Phi = (x_1 \vee \neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1 \vee \neg x_2) \wedge (x_2 \vee x_2 \vee \neg x_1)$

Since Φ has 3 clauses, $k = 3$.

G will be the following graph:



Now, we need to show that F is a function such that Φ is satisfiable if and only if $F(\Phi)$ is in CLIQUE. Thus, we need to show both directions for the if and only if.

-> Assume Φ is satisfiable. Then, at least one variable in each clause is assigned true. Choose the set of variables, one from each clause, that are true and choose the corresponding nodes in G . These nodes form a k -clique in G since there are k clauses and each pair of nodes is joined by an edge since only x_i or $\neg x_i$ can be true.

<- Assume $F(\Phi)$ is in CLIQUE. Thus, the graph G generated by F has a k -clique. Since the graph was generated from Φ with k clauses, there must be one and only one node per clause that is in the k -clique. Choose these nodes and assign the corresponding variables in Φ to true. Since each clause is of the form $(x \vee y \vee z)$, at least one variable per clause is assigned true, so Φ is satisfiable.

Now, we need to show that F is a polynomial time function.

- 1) k is the number of clauses, so this is a simple count of the length of Φ , and the running time is $O(n)$ where n is the length of Φ
- 2) The number of nodes created is $3 \cdot k$, which is $O(n)$. The maximum number of edges created is $3 \cdot k \cdot 3 \cdot (k-1)$, which is $O(n^2)$.

Therefore, the construction of $\langle G, k \rangle$ runs in $O(n^2)$, so $3SAT \leq_p CLIQUE$.

That concludes the proof.

Note: can create a transformation from Vertex Cover to Clique as well through creating a complementary graph of edges. That version is shown in the Algorithms course.

Activity 27: Apply Clique Transformation

CLIQUE = $\{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}$

Assume $F: 3SAT \rightarrow \text{CLIQUE}$ is the function shown above and the textbook.

Apply F to the following Boolean formula:

$(!x_1 \mid \mid x_2 \mid \mid !x_3) \&\& (!x_2 \mid \mid x_3 \mid \mid x_4) \&\& (x_1 \mid \mid !x_2 \mid \mid !x_4)$

Draw G here:

$k =$ _____

Identify the k -clique in the above graph.

Directed Hamiltonian Path is NP-complete

$\text{HAM_PATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t \text{ that goes through every vertex of } G \}$

Show that HAM_PATH (HP) is NP-complete.

1. We showed HP is in NP during lecture earlier. A ND TM can select an ordering of vertices and then check to see that edges connect the successive pairs. This TM runs in $O(n^2)$ time with vertices and edges as lists.

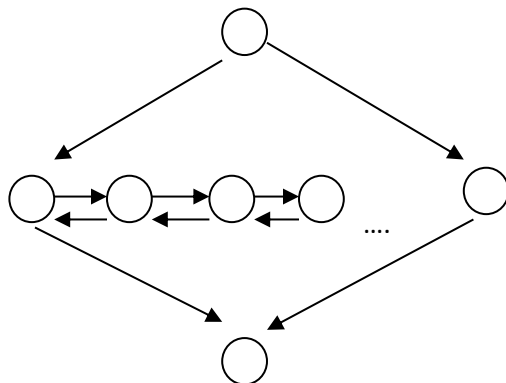
2. We will now show that $3\text{SAT} \leq_p \text{HP}$.

$F(\Phi) =$

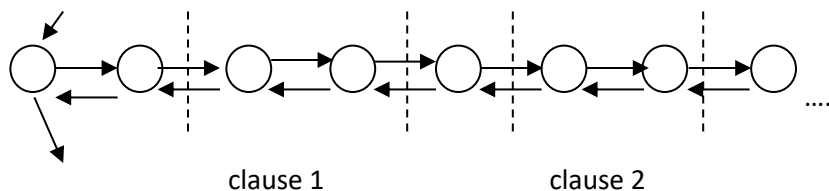
Let $K = \#$ of clauses in Φ .

Let $L = \#$ of variables in Φ .

1. Create the following graph gadget for each variable in Φ :

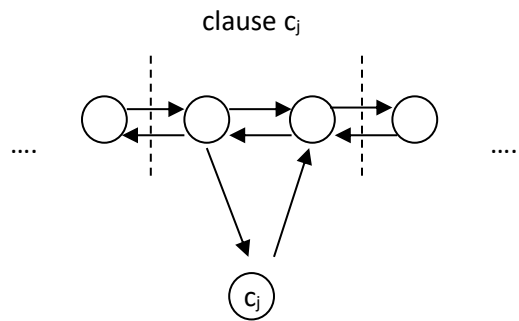


where the middle string of nodes depends on K .

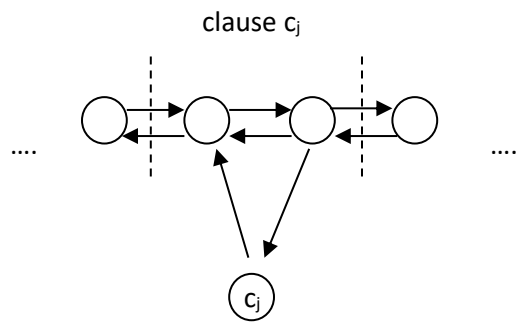


2. Connect the diamond variable gadgets to each other with the top node as s and the bottom node as t .

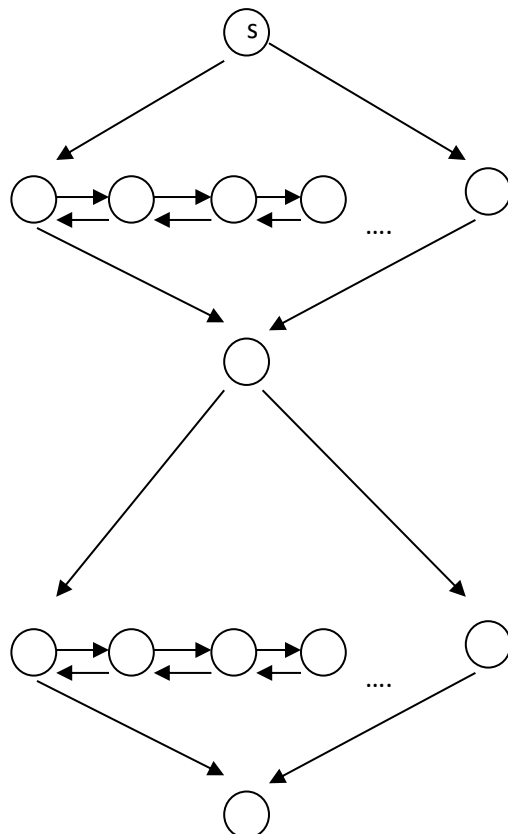
3. If x_i is in clause c_j , add edges from its interior string to a node c_j as follows:

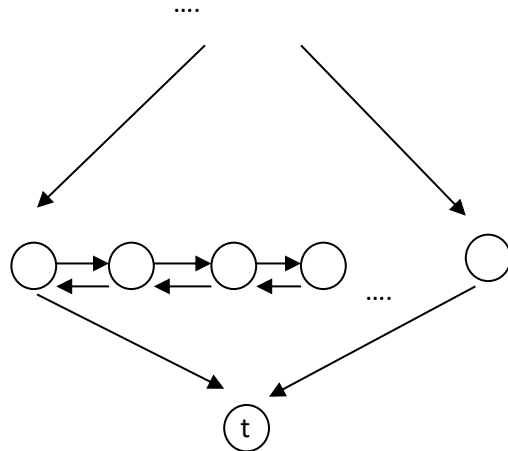


4. If $!x_i$ is in clause c_j , add edges from its interior string to a node c_j as follows:



In total, the graph G and nodes s and t will look like:





Now, we need to show that Φ is in 3SAT if and only if $F(\Phi)$ is in HP.

-> Assume Φ is satisfiable. If x_i is true, we can use the detour going right to each clause node that x_i is in. If x_i is false, we can use the detour going left to each clause node in the other direction. Because Φ is satisfiable, each clause has at least one variable that is true, so we can get to each clause node and get through the entire chain of variables for each variable gadget. Thus, there is a path from s to t that goes through each node.

<- Assume $F(\Phi)$ is in HP. Then, each variable gadget must be entered from the top and exit at the bottom. Either the path will go left or go right from the top node in each variable gadget. If the path goes left, then assign x_i to true. If the path goes to the right, then assign x_i to false. Each clause node must be part of the Hamiltonian path. If the path goes from the gadget for x_i to clause node c_j and goes back to a variable gadget x_k for $k \neq i$, then this cannot be a Hamiltonian path as there is no way to get through the string of nodes for each variable gadget. Thus, the path must include entry to c_j from some x_i and exit from c_j back to the same string in x_i . Therefore, each clause has a true variable assignment, so Φ is in 3SAT.

Now, we need to show that F runs in polynomial time.

1 and 2. For each variable, we create $4 + 3K$ nodes and about the same number of edges. $O(n^2)$

3 and 4. For each clause, we create 1 node. $O(n)$

Therefore, F runs in polynomial time so $3SAT \leq_p HP$ making HP NP-complete.

Activity 28: Apply Hamiltonian Path Transformation

$\text{HAM_PATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t \text{ that goes through every vertex of } G \}$

Assume $F: 3\text{SAT} \rightarrow \text{HAM_PATH}$ is the function described above and in the textbook.

Apply F to the following Boolean formula:

$(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_4)$

Draw $\langle G, s, t \rangle$ as the result of applying F . Note that this result is the input string for HAM_PATH :
(Hint: lots of diamond gadgets)

Highlight the Hamiltonian Path of the resulting graph.

For More Information

If you want to read about more NP-complete problems:

Computers and Intractability: A Guide to the Theory of NP-Completeness

By Michael S. Garey and David S. Johnson

What's NP-complete?

- Satisfiability of logic formulas (SAT, 3SAT)
- Constraint problems (subset sum, knapsack problem)
- Graph problems (ham path, vertex cover, clique, graph coloring, traveling salesman...)
- Touches every area of computer science

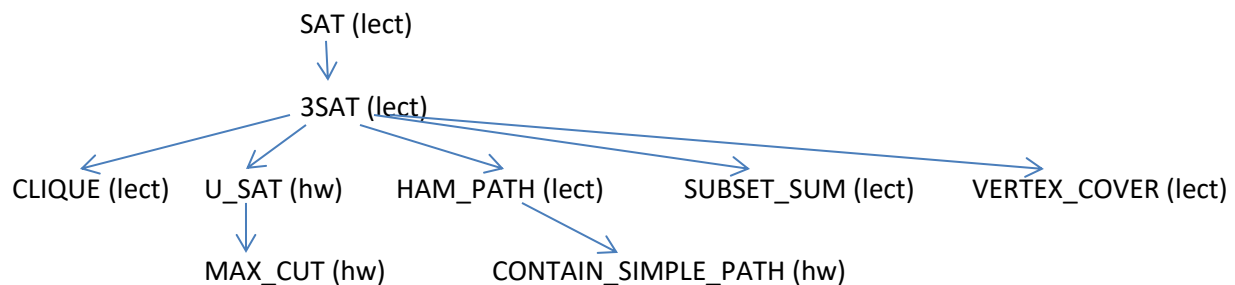
Can find exponential worst-case algorithms for problems that are NP-complete.

In practice, some tips when working with NP-complete problems:

- May find an algorithm that is fast on average, just exponential time in worst case
- Go for approximation algorithms – may not get optimal solution, but can get close to the optimal with a faster approximation algorithm

Remember, reduce the known NP-complete problem to the one you want to show is NP-complete!!

After these lectures and HW 9:



Activity 29: CS357 Last Day “Quiz”

This course, Theory of Computation, introduced you to the tools and skills to characterize types of problems with regard to the computational model necessary to solve them. Some are “easy” or “hard” or “impossible to find a solution”.

For each, indicate what kind of state machine (model of computation) would be necessary to solve that problem.

DFA or NFA (Regular Languages)

PDA (Context-free Languages)

Turing Machine (Decidable Languages)

No Machine (Undecidable Languages)

Problem 1

Input: Airport Name

Output: Yes if # of e’s in the name is greater than the # of a’s in the name. Otherwise, answer is no.

Example: Input: Portland, Output: No
 Input: Los Angeles, Output: Yes

Type of machine: _____

Problem 2

Input: Two airports, pricing schemes, and schedules of all airlines

Output: Cheapest flight option (direct flight need not be cheapest)

Example: Input: Portland, OR and Orlando, FL
 Pricing schemes for all airlines, routes of all airlines,
 Restrictions on pricing for all airlines

 Output: \$392.10 on United going through San Francisco,
 Chicago on flight classes U and T

Type of machine: _____

Problem 3

Input: Two airports, distances between any two airports in the world, airline routes

Output: Shortest flight plan in terms of distance

Example: Input: Portland, OR and Modesto, CA
 Output: Shortest flight plan: Portland to San Francisco, San
 Francisco to Modesto (760 miles)

Type of machine: _____

Problem 4

Input: Airport name

Output: Yes, if it has the substring “or” in it. Otherwise, no.

Example: Input: Portland Output: Yes
 Input: Los Angeles Output: No

Type of machine: _____

Problem 5

Input: Entire airline route map

Output: Yes, if there is a tour. Otherwise, no. A tour consists of visiting every airport and no airport is visited more than once.

Type of machine: _____

Activity 30: Course Reflection

1. What are your main takeaways from this course? (hint: what will you remember 2 years from now? What new tools do you have in your toolbox?)

2. What did you learn that was unexpected?

3. What task/skill are you most proud of from this course?

4. What was your favorite machine type, proof, or transformation from this course and why?

CS357 Review Sheet – Final Exam

You may use 2 crib sheets as notes during the exam. All other resources are off limits – you are on your honor to follow these exam rules.

Content: The final exam will be comprehensive. Material will be drawn from homework assignments, lectures, previous exams, and the textbook (Chapters 0 – 5 and Chapter 7).

Procedure: The exam will start promptly. Please arrive to the classroom on time. You may use **two** sheets of 8.5" x 11" paper (both sides) OR four sheets of 8.5" x 11" paper (one-sided) during the exam. Please prepare your own notesheets; you may type or handwrite your notesheets. Other than your notesheets, the exam is closed-book, closed-calculator, closed-computer other than the exam moodle questions and upload links, and closed-notes. All numerical computations (if any) will be simple enough for you to do by hand.

Topics: This study guide is not a contract – in other words, the exam may not cover every topic listed below and there may be topics that we covered in class that are not explicitly listed.

TOPICS SINCE EXAM 3:

- Post Correspondence Problem (PCP) [matching tiles] is undecidable (reduction from A_{TM} with tiles containing TM configs)
- Time Complexity
 - O , o (big- O and little- O)
 - Every multi-tape TM with time $t(n)$ has equivalent $O(t^2(n))$ time single tape TM
 - Every nondeterministic TM with time $t(n)$ has equivalent $2^{O(t(n))}$ time single tape TM
- Class P (Polynomial Time)
 - Showing a language is in P – create a TM that decides in polynomial time with respect to the input length
 - Examples: PATH (path from s to t in graph), TRIANGLE
 - $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that accepts } w \}$ is in P
 - Dynamic programming on to build up table of variables
 - Grammar is first converted to Chomsky Normal Form (preprocessing takes poly time)
- Class NP (Nondeterministic Polynomial Time)
 - Showing a language is in NP
 - Create a nondeterministic TM that runs in polynomial time $<OR>$
 - Create a verifier that takes input and certificate and runs in polynomial time
 - Examples: HAM_PATH, CLIQUE, SUBSET_SUM
- Does $P = NP$, Does $P \neq NP$? Most likely that $P \neq NP$ (P is a proper subset of NP)
- Reducibility
 - Computable polynomial time function from language A to language B (A is polynomial time reducible to B)
 - If $A \leq_p B$ and B is in P , then A is in P (composition of functions)
- NP-complete languages
 - A is NP-complete if:
 - A is in NP
 - Every other language B in NP is $\leq_p A$
 - Examples: SAT, 3SAT, CLIQUE, VERTEX_COVER, SUBSET_SUM, HAM_PATH, examples from homework
 - To show a language A is NP-complete:
 - Given B is NP-complete, show that $B \leq_p A$.
 - Provide function $F: B \rightarrow A$
 - Show that if w is in B , $F(w)$ is in A
 - Show that if $F(w)$ is in A , w is in B
 - Show that F is a poly time function

PREVIOUS TOPICS – EXAM 3

- Proving a language is not context free (pumping lemma)

- Turing Machines
 - Configurations
 - Formal definition
 - Creating a state machine
 - Equivalent models: nondeterministic, multi-tape, stay put/left/right movements
 - To show equivalence: need to convert the model to a regular TM, need to convert a regular TM to the model
 - Turing-recognizable (accepts strings in language but may not halt on all input)
 - Turing-decidable (always halts)
- Decidable Languages
 - To show A is decidable, construct a TM for A that always halts.
 - If A is a decidable language, then A is Turing-recognizable and the complement of A is Turing-recognizable (also called co-Turing-recognizable).
 - Examples of decidable languages with regular languages: A_{DFA} , A_{NFA} , A_{REG} , E_{DFA} , EQ_{DFA} , $INFINITE_{DFA}$
 - Examples with CFLs: A_{CFG} , E_{CFG}
 - Examples from homework: E_{DFA_REGEX} , ALL_{DFA} , $INFINITE_{PDA}$
 - $A = \{ \langle R \rangle \mid R \text{ is a regular expression describing a language containing at least one string } w \text{ that has } 111 \text{ as a substring.} \}$
 - $B = \{ \langle P \rangle \mid P \text{ is a PDA that has a useless state.} \}$
 - Closure: decidable languages are closed under concatenation, complement, union, intersection
- A_{TM} and the Halting Problem
 - A_{TM} is undecidable
 - A_{TM} is Turing-recognizable
 - complement of A_{TM} is not Turing-recognizable
 - $HALT_{TM}$ is undecidable
- Undecidable Languages
 - Proof by contradiction (using a reduction from language A to language B)
 - Examples of undecidable languages:
 - A_{TM} , $HALT_{TM}$, E_{TM} , EQ_{TM} , $REGULAR_{TM}$, ALL_{CFG}
 - Rice's Theorem
 - Examples: $INFINITE_{TM}$ is undecidable, $\{ \langle M \rangle \mid M \text{ is a TM and } 1011 \text{ is in } L(M) \}$ is undecidable
 - ALL_{CFG}

PREVIOUS TOPICS – EXAM 2

- Programming regular expressions
- Nonregular Languages
 - Prove using the Pumping Lemma, proof by contradiction
 - Prove via closure properties (union, concat, star, intersect, complement) and proof by contradiction
- Context-Free Languages (CFLs)
 - Grammars (CFG)
 - Given a grammar, generate the language and generate strings in the language
 - Generate parse tree for a string
 - Given a language, generate a grammar for it
 - Determine if a grammar is ambiguous
 - Converting a DFA to a grammar (shows that all regular languages are CFLs)
 - Converting a grammar to Chomsky Normal Form (CNF)
 - Pushdown Automata (PDA)
 - Formal definition
 - Given a language, generate a PDA to recognize the language
 - Given a PDA, describe the language it recognizes
 - Equivalence with CFGs (CFG \rightarrow PDA, PDA \rightarrow CFG conversions)
 - Proving closure properties (union, concatenation, star, etc. by modifying grammars or modifying PDAs)

PREVIOUS TOPICS – EXAM 1

- Discrete Math Review
 - Sets, Sequences (Tuples)
 - Functions, Relations

- Graphs
- Strings and Languages
- Boolean Logic
- Proofs
 - Direct, Indirect, Contradiction, Induction, Construction
- Regular Languages (those that can be recognized by DFAs, NFAs, or written as regular expressions)
- DFAs
 - Given a language, construct the DFA
 - Given a DFA, state the language it recognizes
 - Formal definition as a tuple
- Union, Concatenation, *
 - Closure of regular languages under union, concatenation, * (know the proofs)
- Closure of regular languages under other operations such as reverse and perfect shuffle
- NFAs
 - Given a language, construct the NFA
 - Given an NFA, state the language it recognizes
 - Converting NFAs to DFAs
 - Formal definition as a tuple
- Regular Expressions
 - Given a regular expression, state its language
 - Given a language, create a regular expression
 - Converting regular expressions to NFAs
 - Converting DFAs to regular expressions
 - Converting DFAs to regular expressions