CS-440 : Notes

Charlie Stuart : src322

Nowak : Spring 2021

IN PROGRESS!!!!

Note: Sections are numbered with my heart

Contents

1 Other Resources

Introduction to the Theory of Computing : From CS164 by BLS https://1513041.mediaspace.kaltura.com/media/Introduction+to+Computer+Theory/1_kw3wb7ll

Minimization of a DFA : https://www.youtube.com/watch?v=0XaGAkY09Wc

Post Correspondence Problem :

Reductions (Part 1) : What are reductions https://www.youtube.com/watch?v=U4yGQp5aCTM

Reduction (Part 2) : How to reduce https://www.youtube.com/watch?v=1KvYcoTeRbI

Turing Machines : A Turing Machine made with a white board https://youtu.be/E3keLeMwfHY

2 Math Review

2.1 Set Theory

From pages 3-7 in *Introduction to the Theory of Computation* by Michael Sipser

Set : A group of objects represented as a unit

Element : An object in a set

Member : An object in a set

Multi Set : An set containing an element that occurs multiple times

Subset : A set that consists of elements that exist in a different set

Proper Subset : A set that is a subset of another set, but not equal

Infinite Set : A set of infinitely many elements

Empty Set : A set of no elements

Singleton Set : A set of one elements

Unordered Pair : A set of two elements

Sequence : A set in a specific order

Tuple : A finite set

k-Tuple : A tuple of k elements

Ordered Pair : A 2-tuple

Power Set : All the subsets of A

$\in$: Is a member of

$\notin$: Is not a member of

$\subset$: Is a proper subset of

$\not\subset$: Is not a proper subset of

$\subseteq$: Is a subset of

$\not\subseteq$: Is not a subset of

$\cup$: Union of two sets

$\cap$: Intersection of two sets

$\times$: Cross product of two sets

$\mathbb{N}$: Set of natural numbers

$\mathbb{Z}$: Set of integers

$\mathbb{Q}$: Set of rational numbers

$\mathbb{A}$: Set of algebraic numbers

$\mathbb{R}$: Set of real numbers

A set is defined in a few ways

$S = \{7, 21, 57\}$	Finite Set
$S = \{1, 2, 3...\}$	Infinite Set of all natural numbers $\mathbb{N}$
$S = \{7, 7, 21, 57\}$	Multi Set
$S = \varnothing$	Empty Set
$S = \{5\}$	Singleton Set
$S = \{5, 3\}$	Unordered pair
$S = \{n \mid n = m^2 \text{ for some } m \in \mathbb{N}\}$	Set of perfect squares

The union of two sets is the same as an OR operator in boolean algebra. It's all the elements in both sets.

$$\begin{aligned} A &= \{1, 2, 3\} \\ B &= \{3, 4, 5\} \\ A \cup B &= \{1, 2, 3, 4, 5\} \end{aligned}$$

The intersection of two sets is the same as an AND operator in boolean algebra. It's all the elements that appear only in both sets.

$$\begin{aligned} A &= \{1, 2, 3\} \\ B &= \{3, 4, 5\} \\ A \cap B &= \{3\} \end{aligned}$$

The Cartesian product, or cross product, of two sets is the set of all ordered pairs where the first element is a member of the first set and the second element is a member of the second set for every combination.

$$\begin{aligned} A &= \{1, 2\} \\ B &= \{x, y, z\} \\ A \times B &= \{(1, x), (2, x), (1, y), (2, y), (1, z), (2, z)\} \end{aligned}$$

A relation is a subset of $X \times X$. It lists all pairs that are related by R

$$R \subset X \times X$$

Equivalence Relation

- Reflexive ()
- Symmetric ()
- Transitive (x R y and y R z then x R z)

2.2 Logical Operators

Negation

p	$\neg p$
0	1
1	0

Conjuction : Logical And

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

Disjuction : Logical Or

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

Implication : If p Then q

p	q	$p \implies q$
0	0	1
0	1	1
1	0	0
1	1	1

Equivalence : p if and only if q

p	q	$p \iff q$
0	0	1
0	1	0
1	0	0
1	1	1

For All : $\forall$

There Exists : $\exists$

2.3 Functions

From pages 7-8 in *Introduction to the Theory of Computation* by Michael Sipser

Function : An objects that sets up an input-output relationship

Domain : The set of possible inputs to a function

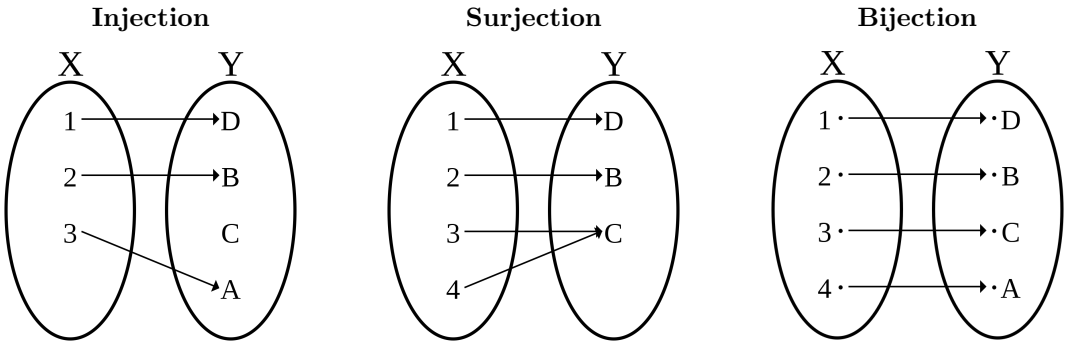
Range : The set of possible outputs to a function

$f : D \to R$ Function f has domain D and range R

One-To-One (Injection) : If $x_1 \neq x_2$ then $f(x_1) \neq f(x_2)$

Onto (Surjection) : $f(X) = Y$

Bijection : One-To-One and Onto



2.4 Logarithms

$$\log_b(XY) = \log_b(X) + \log_b(Y)$$
$$\log_b\left(\frac{X}{Y}\right) = \log_b(X) - \log_b(Y)$$
$$\log_b(X^y) = y\log_b(X)$$
$$a^{\log_b(c)} = c^{\log_b(a)}$$

2.5 Summations

From CLRS Appendix A

REMEMBER : Summations are inclusive

Constants can be "taken out":

$$\sum_{i=1}^{n} cx_i = c\sum_{i=1}^{n} x_i$$

Addition can be broken up:

$$\sum_{i=1}^{n}(x_i + y_i) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i$$

Arithmetic Series :

$$\sum_{i=1}^{n} i = 1 + 2 + ... + n$$
$$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$$
$$\sum_{i=1}^{n} i \in \Theta(n^2)$$

Sum of Squares :

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Sum of Cubes :

$$\sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Geometric Series : When $x \neq 1$ and is real

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + ... + x^n$$
$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

Geometric Series : When the summation is infinite and $|x| < 1$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

Harmonic Series :

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$
$$H(n) = \sum_{i=1}^{n} \frac{1}{i}$$
$$H(n) = \ln n + O(1)$$

Logarithms :

$$S(n) = \sum_{i=1}^{n} \log(i)$$
$$S(n) = \log(1) + \log(2) + ... + \log(n-1) + \log(n)$$
$$S(n) = \log(1 * 2 * ... * (n-1) * n)$$
$$S(n) = \log(n!)$$

3 Proofs

From pages 17-23 in *Introduction to the Theory of Computation* by Michael Sipser

Definition : Describes and object or notation

Proof : Convincing logical argument that a statement is true

Theorem : A mathematical statement proved true

Lemma : A theorem that assists in the proof of another theorem

Corollaries : The parts of a theorem where we can conclude that other related statements are true

3.1 Proof By Construction

To prove something exists, we prove we can construct the object in a general case.

3.2 Proof By Contradiction

In order to prove a theorem true, we can say it's false, then show that that leads to an even more false statement. By disproving the false statement, we show the contradiction is true.

3.3 Proof By Induction

In induction, we need an ordered set of variables with consistent variance. We create a base case, prove it is true, then for the inductive case, we prove that taking one step forward is also true and reinforces the base case

Prove:

$$S(n) = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\begin{aligned}
S(n) &= \sum_{i=0}^{n} 2^i \\
S(0) &= \sum_{i=0}^{0} 2^i && \text{Base Case} \\
S(0) &= 1 \\
2^{0+1} - 1 &= 1 \\
S(n) &\rightarrow S(n+1) && \text{Inductive Case} \\
S(n+1) &= \sum_{i=0}^{n+1} 2^i \\
S(n+1) &= \sum_{i=0}^{n} 2^i + 2^{n+1} \\
S(n+1) &= 2^{n+1} - 1 + 2^{n+1} \\
S(n+1) &= 2^{n+2} - 1 \\
S(n+1) &= 2^{(n+1)+1} - 1 && \text{Induction Holds}
\end{aligned}$$

3.4 Proof By Structural Induction

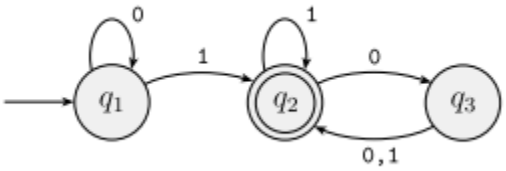
Structural induction is similar to regular induction, except instead of using a base case and a literal $n + 1$ case, it's analyzing the data structure to see what a logical "plus one" step would be. This can be adding subtrees, edges, vertices, etc.

4 Finite Automata

4.1 Deterministic Finite Automata

From pages 31-43 in *Introduction to the Theory of Computation* by Michael Sipser

A finite automaton, or finite state machine, is the simplest way to describe a computational models behavior.



In the above state machine, there are three states denoted by circles, q_1, q_2, and q_3. The transitions are denoted by arrows. The start state is denoted by the arrow coming from no where. The accept state is denoted by the bubble with a circle inside it. Upon receiving a string, it processes it then produces an "accept" or "reject" output. Only if we end in an accept state is the output "accept".

A is the set of all strings that the state machine accepts. Σ is the alphabet of symbols that a string, w, is composed of. A is the language of the machine where $L(M) = A$. M recognizes A. M accepts A.

While the simple description of a finite automaton is simple enough for me to understand, there are five things a finite automaton must include. This can be described using the 5-tuple, $(Q, \Sigma, \delta, q_0, F)$.

1. **States :** A set of states (Q)
2. **Alphabet :** The allowed input symbols (Σ)
3. **Transition Function :** The rules for moving from one state to another ($\delta : Q \times \Sigma \to Q$)
4. **Start State :** Self Explanatory ($q_0 \in Q$)
5. **Set of Accept States :** Self explanatory ($F \subseteq Q$)

Looking back at the above state machine, we can describe it as the following.

1. $Q = \{q_1, q_2, q_3\}$
2. $\Sigma = \{0, 1\}$
3. δ can be described in the following table with states on the left, input on the top, and the state transitioned to being the intersection

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. $q_0 = q_1$
5. $F = \{q_2\}$ (in this case, a singleton set)

4.2 Non-deterministic Finite Automata

From pages 47-50 and 53 in *Introduction to the Theory of Computation* by Michael Sipser

Deterministic : Given a state and an input symbol, the next state is determined by a function and we know what it will be.

Non-Deterministic : There are many options in a given state.

Non Deterministic Finite Automata (NFA)

- Can have zero, one, or many transitions for each alphabet member
- Can include ε in addition to the alphabet
- Traverses in a tree

Deterministic Finite Automata (DFA)

- Each state has one transition for each alphabet member
- Only includes alphabet members
- Traverses sequentially
- All DFAs are NFAs

An NFA is determined with a reject/accept state in a different way than a DFA. A DFA traverses sequentially, one state after the next, an NFA traverses and creates a tree of all the possible outcomes. Some rules when traversing:

- If there is not a transition for an input, the branch stops
- If there are multiple transitions for an input, it branches for as many times with than input.
- Upon reaching ε

We also have a more formal definition of an NFA:

1. **States :** A finite set of states (Q)
2. **Alphabet :** The finite allowed input symbols (Σ)
3. **Transition Function :** The rules for moving from one state to another ($\delta : Q \times \Sigma_\varepsilon \to P(Q)$) $P(Q)$ is the power set of Q
4. **Start State :** Self Explanatory ($q_0 \in Q$)
5. **Set of Accept States :** Self explanatory ($F \subseteq Q$)

4.2.1 Subset Construction

Theorem : Subset Construction

Let an NFA N be given (we assume for the sake of simplicity that N doesn't have any ε transitions), we can construct an equivalent DFA D where the language of D is the same as the language of N ($L(D) = L(N)$).

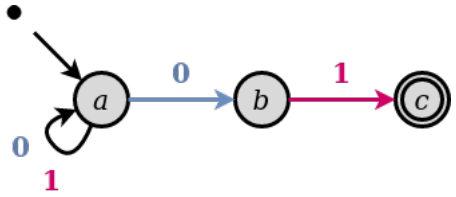
$$N = (Q, \Sigma, \delta, q_0, F)$$
$$D = (Q', \Sigma, \delta', q_0', F')$$

Where Q', q_0', δ', and F' are defined as:

$$Q' = P(Q) \qquad \text{the set consisting of all subsets of } Q$$
$$q_0' = \{q_0\}$$
$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$
$$F' = \{R \in Q' \mid R \cap F \neq \varnothing\}$$

Example:

Given the NFA for a language where all strings end in "01":



$$Q = \{a, b, c\}$$
$$\Sigma = \{0, 1\}$$

$\delta =$

	0	1
a	$\{a, b\}$	$\{a\}$
b	$\varnothing$	$\{c\}$
c	$\varnothing$	$\varnothing$

$$q_0 = a$$
$$F = \{c\}$$

To turn this into a DFA, we first find the states Q' which is a powerset of Q. *Note: I concatenated the letters for the state names instead of making a set of sets for cleanliness and readability.*

$$Q' = \{\varnothing, a, b, c, ab, ac, bc, abc\}$$

Then our starting state q_0' stays the same, only now represented as a set:

$$q_0' = \{a\}$$

Σ doesn't change, so we can move onto δ'. Let's start with a blank table:

$\delta' =$

	0	1
$\varnothing$		
a		
b		
c		
ab		
ac		
bc		
abc		

Now let's break down the big union operation:

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$

It's saying for each state R in the DFA (which is a set of states) and input character a, the state we transition to is the union of all the transitions from the substates in the NFA. So $\delta'(abc, 0)$ transitions to the union of $\delta(a, 0)$, $\delta(b, 0)$, and $\delta(c, 0)$. This will give a set we transition to, but remember that each state in the DFA, R, is a set. This gives the following δ':

$\delta' =$

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
a	ab	a
b	$\varnothing$	c
c	$\varnothing$	$\varnothing$
ab	ab	ac
ac	ab	a
bc	$\varnothing$	c
abc	ab	ac

We'll now notice that many states are unreachable. No state ever transitions to abc, so it's transitions aren't necessary and we can remove it from the diagram. We then get:

$\delta' =$

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
a	ab	a
b	$\varnothing$	c
c	$\varnothing$	$\varnothing$
ab	ab	ac
ac	ab	a
bc	$\varnothing$	c
abc	ab	ac

$\rightarrow$

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
a	ab	a
c	$\varnothing$	$\varnothing$
ab	ab	ac
ac	ab	a

$\rightarrow$

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
a	ab	a
ab	ab	ac
ac	ab	a

$\rightarrow$

	0	1
a	ab	a
ab	ab	ac
ac	ab	a

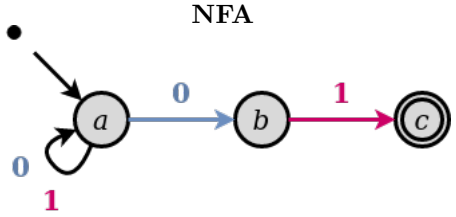
$$Q' = \{a, ab, ac\}$$

Our finishing state set F' is the set states in Q' that when unioned with the original finishing set F isn't empty. The finishing set was $F = \{c\}$, but $c \notin Q'$. However, $ac \in Q'$ which contains c and is a member of the new finishing set:

$$F' = \{ac\}$$

Putting it all together, side by side we get:

NFA



$Q = \{a, b, c\}$

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

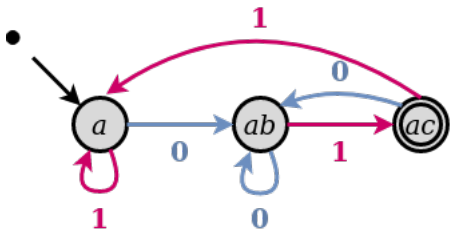
$\delta =$

	0	1
a	$\{a, b\}$	$\{a\}$
b	$\varnothing$	$\{c\}$
c	$\varnothing$	$\varnothing$

$q_0 = a$

$F = \{c\}$

DFA



$Q' = \{a, ab, ac\}$

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

$\delta' =$

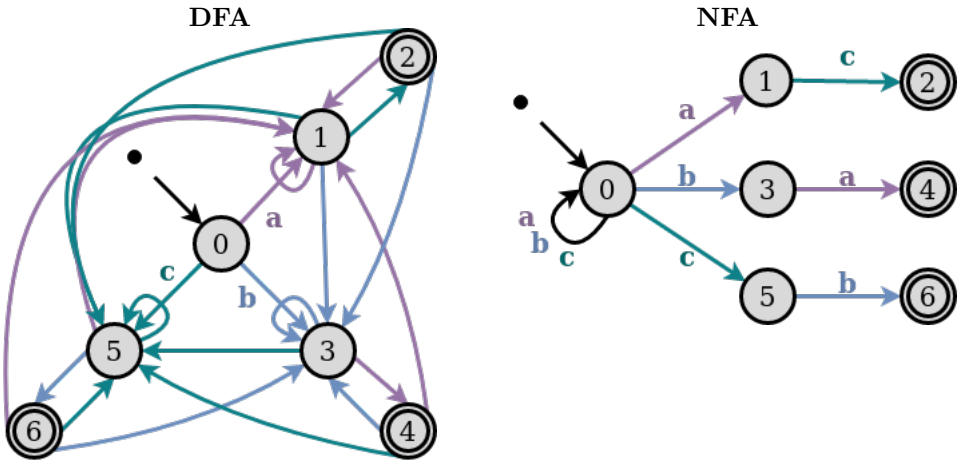
	0	1
a	ab	a
ab	ab	ac
ac	ab	a

$q_0' = a$

$F = \{ac\}$

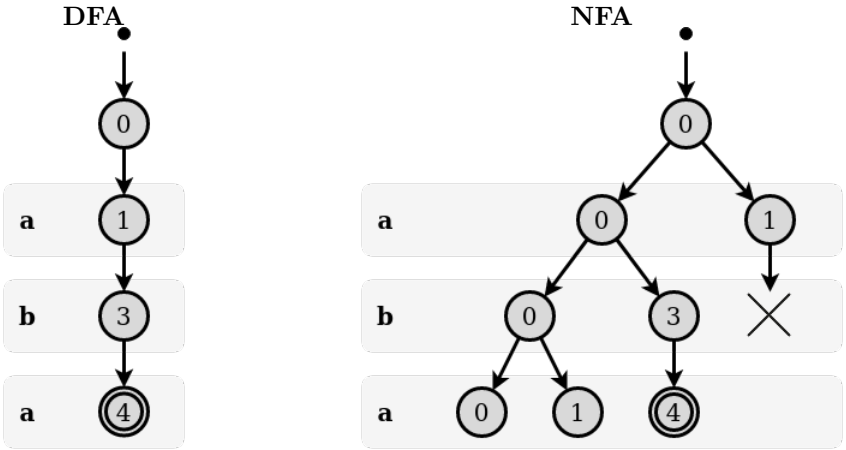
I did this before looking at the subset construction, but here's another, more complex, example showing the traversal of a DFA and NFA.

To more visually show the differences between equivalent DFAs and NFAs, consider the language that accepts all strings ending in "ac", "ba", or "cb". Since the DFA transitions got messy, I color coded them. All **"a"** transitions are **purple**, **"b"** transitions are **blue**, and **"c"** transitions are **teal**.



The NFA is a lot easier to read because for the most part, it only requires the "necessary" transitions that the DFA has. The DFA needs to have all possible transitions for each state.

The traversal of each given the string "aba":



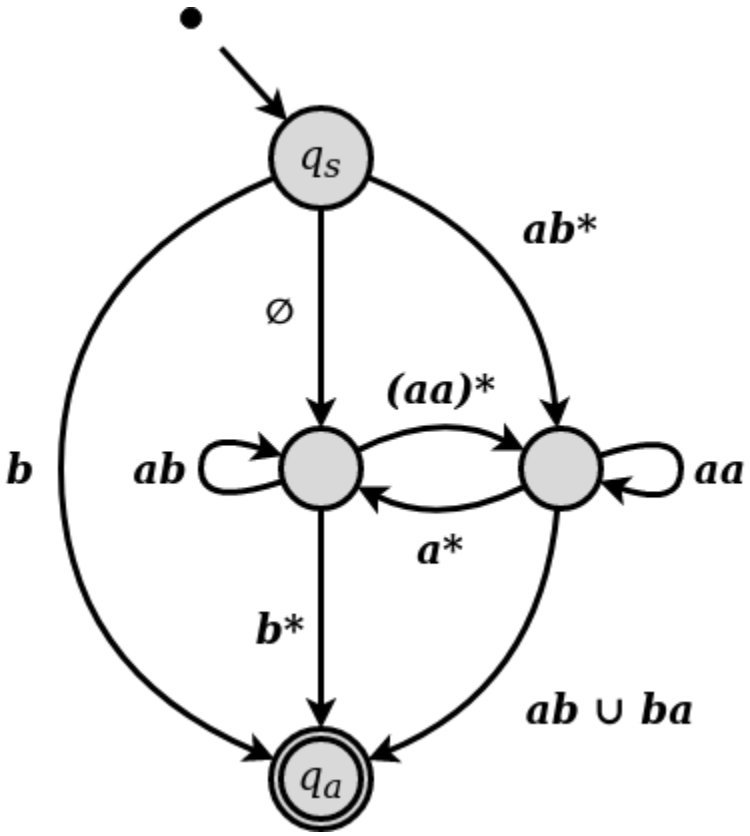
4.3 Generalized Non-Deterministic Finite Automata

A GNFA is an NFA that reads multiple input symbols at a time using regular expressions to connect states. GNFAs have a few extra properties that aren't required by NFAs:

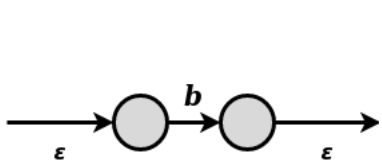
- The state state has transition arrows going to every other state and no arrows coming in from any other state
- There is only one accept state. It is pointed to by every other state and points to no states
- Every other state has an arrow to every other state and itself

The formal definition of a GNFA is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$:

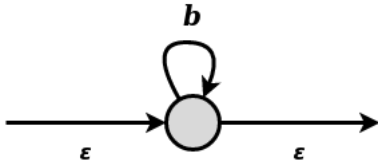
- Q : The finite set of states
- Σ : The input alphabet
- $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to \mathcal{R}$: The transition function
- q_{start} : The start state
- q_{accept} : The accept state



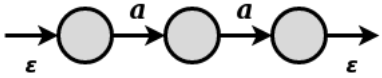
The reason we can use regular expressions as transitions is because we can expand those regular expressions into small NFAs:



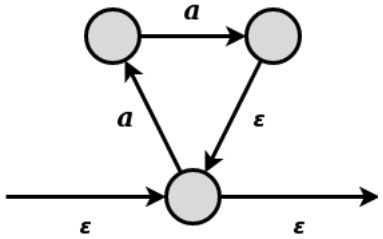
b



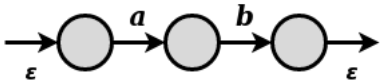
b^*



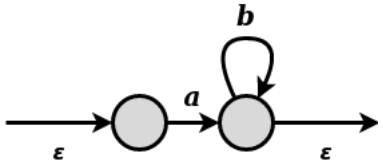
aa



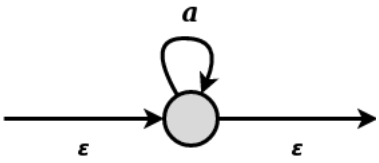
$(aa)^*$



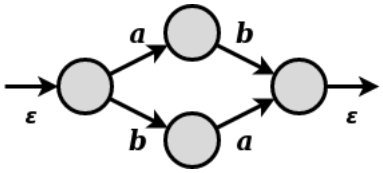
ab



ab^*

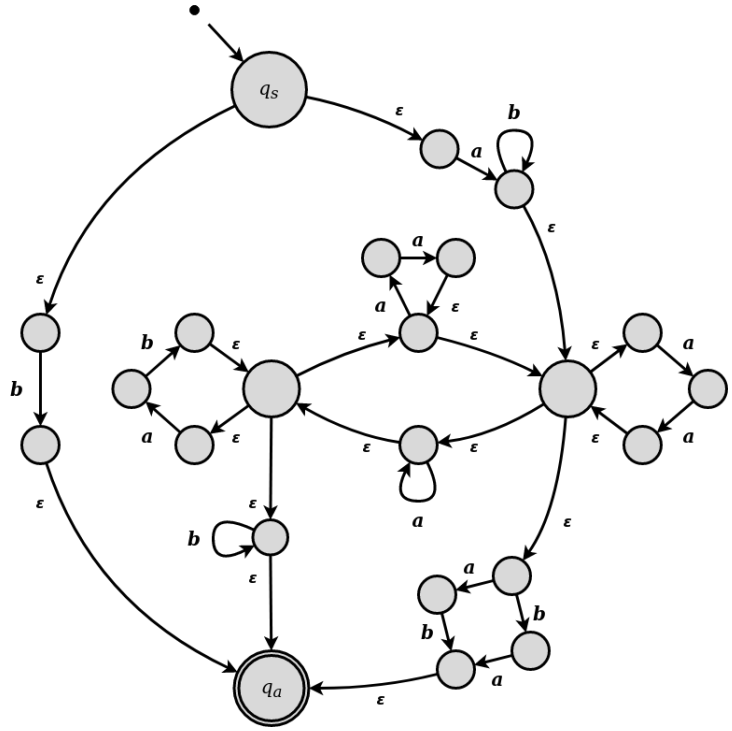


a^*

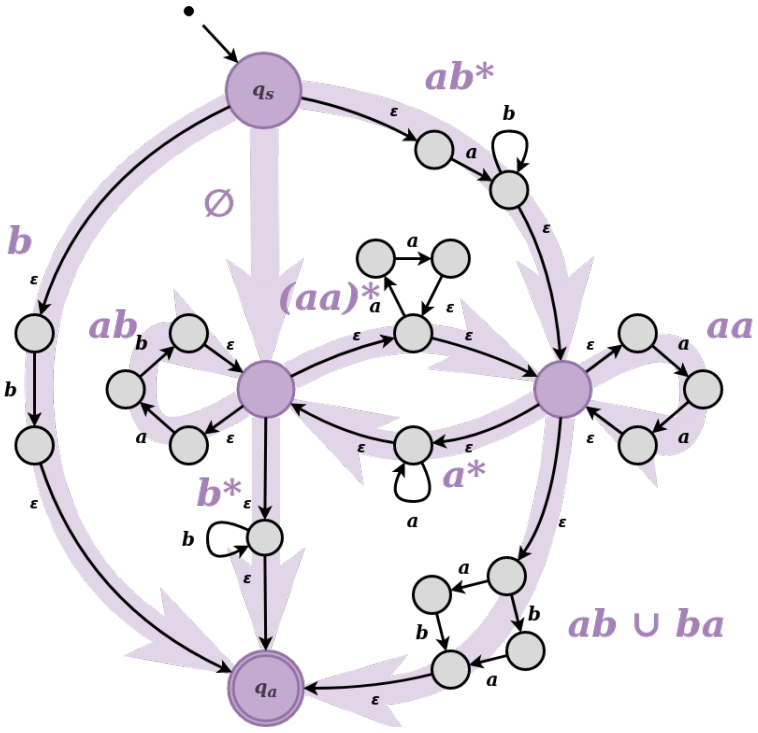


ab ∪ ba

Which we can put back together to get:



Then again, with the GNFA highlighted in purple for clarity:



4.3.1 DFAs to GNFAs to Regex

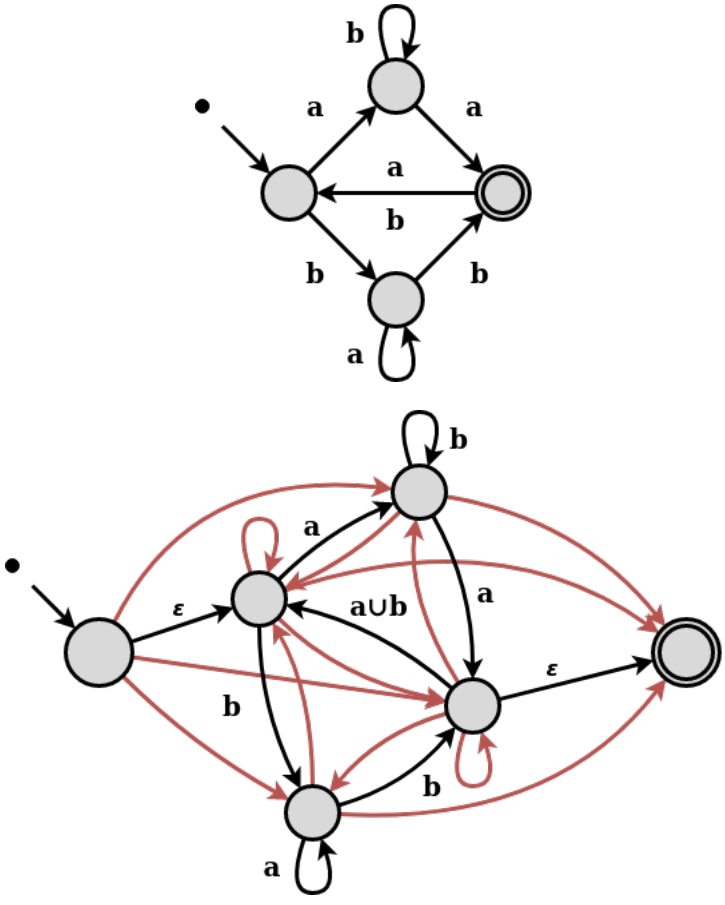
To get a regular expression from a DFA, we can:

1. Convert the DFA into a GNFA
2. Combine states in the GNFA using regex until two states and one transition remains

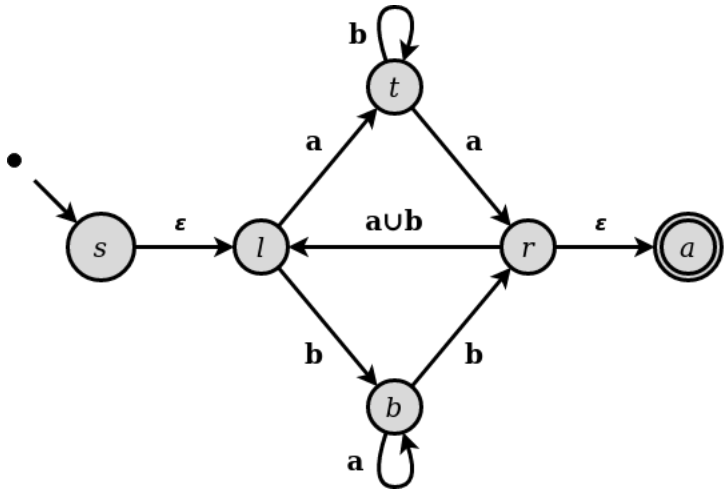
Coverting a DFA into a GNFA is easy, we need to:

1. Create a new start state with a ε transition to the old start state
2. Create a new single accept state with ε transistions from the old accept states
3. If any states have multiple labels or multiple arrows between the same states in the same direction, union them together into one arrow
4. Between states that had no arrows, add an $\varnothing$ transistion

Apply this and we'll get the following:

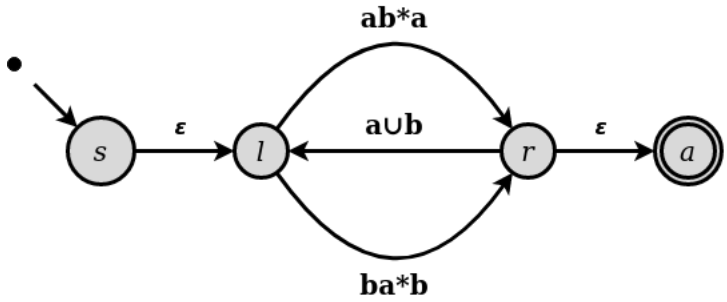


Above, the **red** arrows are the $\varnothing$ transistions. I included them for the sake of following instructions, but these transistions cannot be travelled, so I can remove them to get a much cleaner diagram that I've also given labels to assist in the next step:

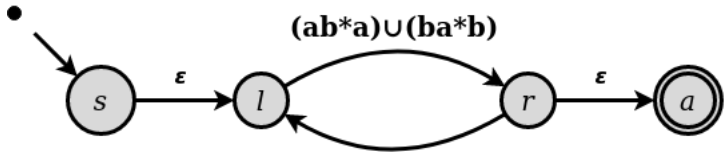


Now that we have our GNFA, we want to combine states using regular expressions to get to the point where we only have a start and accept state.

We can start by removing these top t and bottom b states. There's a single transition to each (we'll call it R_i), a single transistion out (R_o), and a repeated transistion to itself (R_r). This can be simplified to the regular expression $R_i R_r^* R_o$ and we can replace those states:



We then see that there are two transistions from the "left state" l to the "right state" r. We can union these together and get:



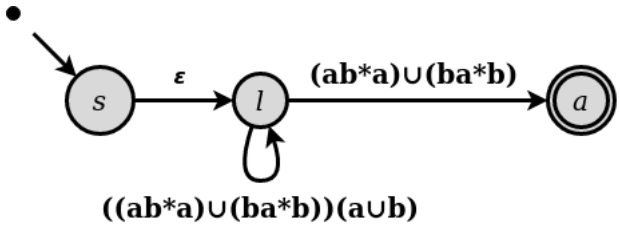
This next part is a little wonky. Now when we replace a state, we need to change the transistions from the start state or to the accept state.

If we were to just remove r and "collapse" the ε transistion, we'd break the GNFA structure because there would be an exit transistion from the accept state a to l, which isn't allowed.

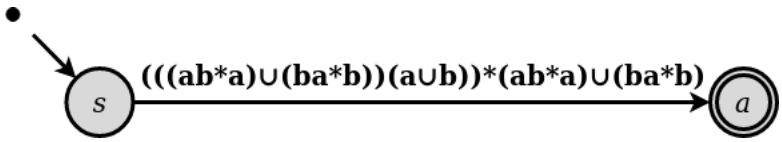
I'm going to first define $R_1 = (ab^*a) \cup (ba * b)$ (the transistion from l to r). R_1 can point directly to a without any issues since we're just concatenating it with the empty string (ε).

Then I'm going to define $R_2 = a \cup b$ (the transistion from r to l). As said before, R_2 cannot point from a to l. In this case though, we see that a "loop" is formed. Through some analysis, I see that I can create a transistion from l to l (itself) by traveling down R_1 then coming back through R_2. I can create a new transistion on l by concatenating R_1 and R_2.

This all creates:

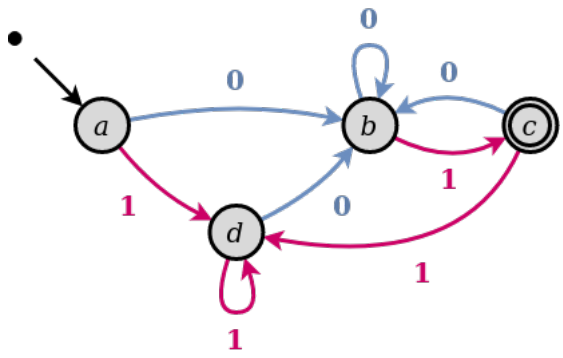


Finally, to replace l, we can apply the same property we did first with $R_i = \varepsilon$, $R_r = ((ab^*a) \cup (ba^*b))(a \cup b)$, and $R_o = (ab^*a) \cup (ba^*b)$ to get quite possibly the most hideous regular expression I've ever written:



4.4 Minimal DFAs

A DFA can have more states than necessary. Take the example of the language $L = \{\Sigma^* 01\}$. We saw before this requires only 3 states. It can also be recognized by the following DFA with four states:



We know by looking at this that I just added the redundant state d, but how can we prove that? We look at the δ function and build equivalence classes.

$\delta =$

q	0	1
a	b	d
b	b	c
c	b	d
d	b	d

Now we build equivalence classes. We start with 0 Equivalence, where we just separate the accepting and non-accepting states:

0 Equivalence: $\{a, b, d\}, \{c\}$

To move onto 1 Equivalence, we check each states δ with the other states in it's class. If each state stays "local", they continue to live in the same set. I use the $\approx$ symbol here to mean in the same set.

q_x	q_y	$\delta(q_x, 0) \approx \delta(q_y, 0)$	$\delta(q_x, 1) \approx \delta(q_y, 1)$
a	b	$b \approx b$	$d \not\approx c$
a	d	$b \approx b$	$d \approx d$
b	d	$b \approx b$	$c \not\approx d$

This means that a and d are still equivalent because they both transition to states in the same set, but b has a transition to c which is not in the same set. This means b has been voted off theisland and moves to its own set.

We don't need to check c because it has no one to be checked against.

1 Equivalence: $\{a, d\}, \{b\}, \{c\}$

We repeat the process again for 2 Equivalence, now using the newly formed sets.

q_x	q_y	$\delta(q_x, 0) \approx \delta(q_y, 0)$	$\delta(q_x, 1) \approx \delta(q_y, 1)$
a	d	$b \approx b$	$d \approx d$

Nothing changes. a and d both transition to the same sets on the same input.

2 Equivalence: $\{a, d\}, \{b\}, \{c\}$

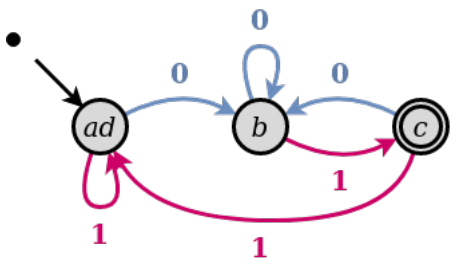
Now we notice that our 1 Equivalence and 2 Equivalence are the same. We can stop calculating equivalence sets because we'll always end up with this output.

Now we can rebuild our DFA where the states are the equivalence sets. The transitions are then adjusted to point to the equivalence set state. Since we've been building our sets based on where we transition to, we've been ensuring that each δ will transition to only one equivalence set. This set may have one or many states in it, but it doesn't matter because it's all one state now.

We finish with the following table and DFA:

$\delta =$

q	0	1
ad	b	ad
b	b	c
c	b	ad



The MINIMIZE algorithm (page 299 of Sipser) takes a DFA $M = (Q, \Sigma, \delta, q_0, A)$ as input:

1. Remove all states M that are unreachable from the start state
2. Construct the following undirected graph G whose nodes are the states of M
3. Place an edge in G connecting every accept state with every non-accept states. Additional edges are added as follows:
4. Repeat until no new edges are added to G
   (a) For every pair of distinct states q and r of M and every $a \in \Sigma$
      - Add the edge (q, r) to G if $(\delta(q, a), \delta(r, a))$ is an edge of G
5. For each state q, let $[q]$ be the collection of states $[q] = \{r \in Q \mid \text{no edge joins } q \text{ and } r \text{ in } G\}$
6. Form a new DFA $M' = (Q', \Sigma, \delta', q_0', A')$ where:
   - $Q' = \{[q] \mid q \in Q\}$ (if $[q] = [r]$, only one is in Q')
   - $\delta'([q], a) = [\delta(q, a)]$ for every $q \in Q$ and $a \in \Sigma$
   - $q_0' = [q_0]$
   - $A' = \{[q] \mid q \in A\}$
7. Return M'

5 Regular Languages

From pages 44-45 in *Introduction to the Theory of Computation* by Michael Sipser

Unary Operator : An operation on one element

Binary Operator : An operation on two elements

Regular Language : A language that can be expressed with a regular expression

5.1 Operations

Union : $A \cup B = \{x | x \in A \text{ or } x \in B\}$

Intersection : $A \cap B = \{x | x \in A \text{ and } x \in B\}$

Concatenation : $A \circ B = \{xy | x \in A \text{ and } y \in B\}$

Concatenation is like a cross product but concatenating the tuples.

Complement : $\bar{L}$ is the opposite of L in Σ^*

Star : $A^* = \{x_1 x_2 ... x_k | k \geq 0 \text{ and } x_i \in A\}$

In star, ε (the empty string) is always a member of the language. This is because the star means 0 or more times.

Given alphabet Σ where it's the standard English alphabet, a through z. If $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{girl}, \text{boy}\}$, then

$$
\begin{aligned}
A \cup B &= \{\text{good}, \text{bad}, \text{boy}, \text{girl}\} \\
A \circ B &= \{\text{goodgirl}, \text{goodboy}, \text{badgirl}, \text{badboy}\} \\
A^* &= \{\varepsilon, \text{good}, \text{bad}, \text{goodgood}, \text{goodbad}, \text{badgood}, \text{badbad}, \text{goodgoodgood...}\}
\end{aligned}
$$

5.2 Properties

- A language $L \in \Sigma^*$ is considered regular is $L = L(A)$ for some DFA A
- A language $L \in \Sigma^*$ is considered regular is $L = L(B)$ for some NFA B
- If L_1 and L_2 are regular then so are the languages described by $L_1 \cap L_2$ and $L_1 \cup L_2$. Regular languages are closed under union and intersection
- If $L \subseteq \Sigma^*$ is regular, so is $\bar{L}$. This is achieved by inverting the "accept" and "deny" states in a DFA
- If a language can be described by a regular expression, then it is regular

Theorem

1.25 in *Introduction to the Theory of Computation* by Michael Sipser

The class of regular languages is closed under the union operation. In dumb bitch english, if A_1 is regular and A_2 is regular, then $A_1 \cup A_2$ is regular.

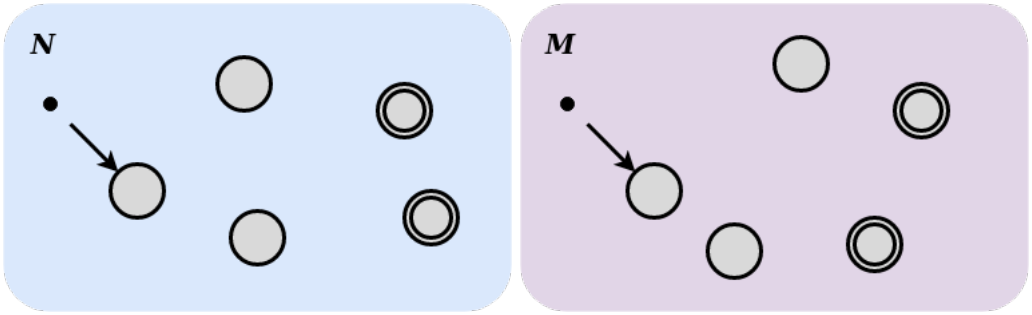
To prove this we could make two machines M_1 and M_2 that both separately accept A_1 and A_2 respectively. We then could combine them into a machine M, however the machine would not know the difference between something M_1 should accept vs M_2 should accept and could confuse inputs.

Given $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ accepts A_1, $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ accepts A_2, construct an NFA $N = (Q, \Sigma, \delta, q_0, F)$ accepts $A_1 \cup A_2$.

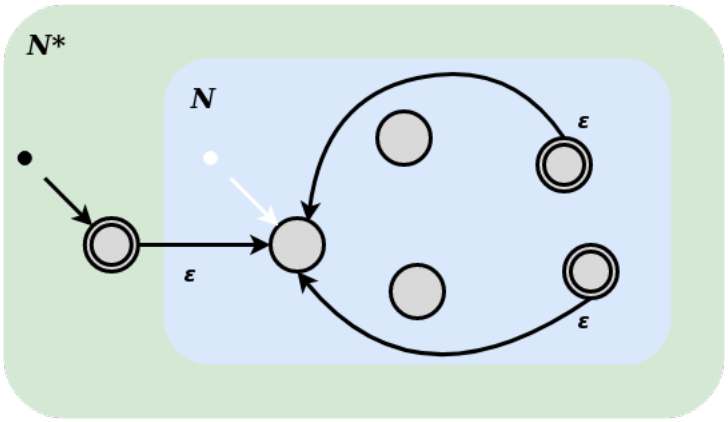
1. q_0 is the start state of N. It's new
2. $Q = \{q_0\} \cup Q_1 \cup Q_2$
3. $F = F_1 \cup F_2$
4. Define δ for any $q \in Q$ and for any $a \in \Sigma_\varepsilon$ as:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \varnothing & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

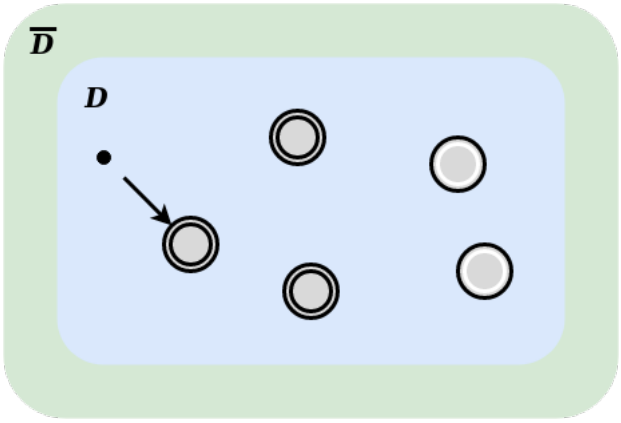
To more visually show closure under these operations, say we the NFAs N and M:



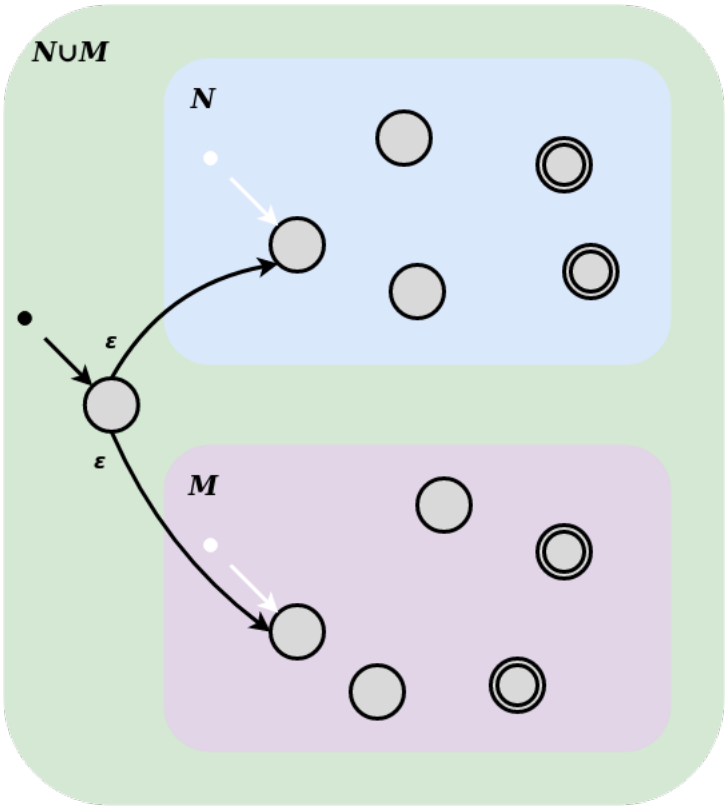
Star : Create a new start state that accepts the empty string and have a single ε transition to the former start state. Add ε transitions from each accept state to the former state state. To get the corresponding DFA, use subset construction.



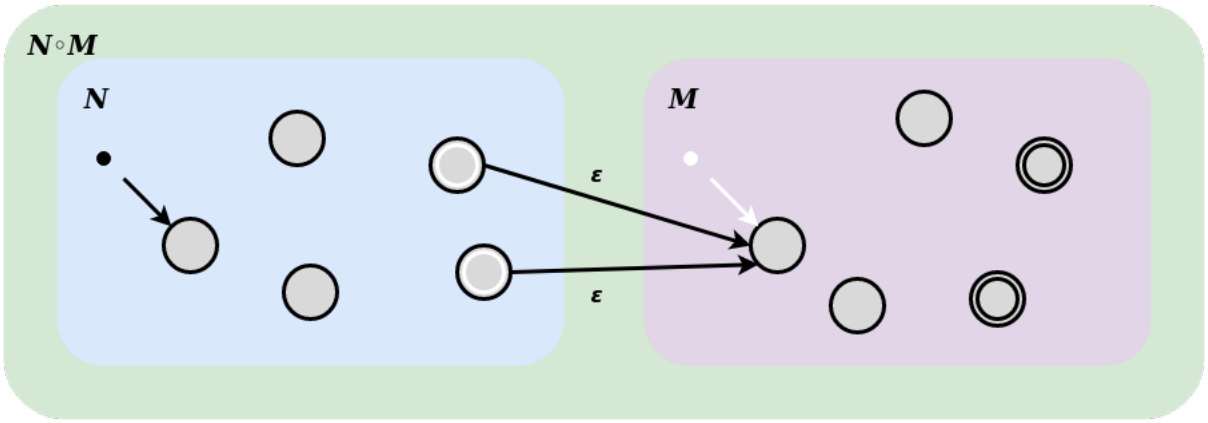
Complement : Swap the accept and not accept states. This only applies to DFAs



Union : Create a new start state with ε transitions to the former start states of N and M. Again, to get the DFA you can use subset construction



Concatenation : Change all of N's accept states to be non-accepting. Create a ε transition from each of these to M's start state. Remove M's start state. This creates one machine with N's start state and M's accepting states. Again, to get the DFA, use subset construction



Closure : Given a language L, L^* is a closure of L if $L^* = \{w = v_1 v_2 \dots v_l \mid v_i \in L, i = 1, 2, \dots, k, k \geq 1\}$

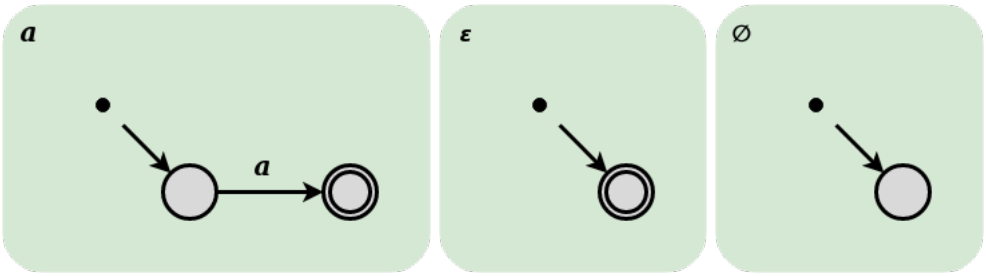
5.3 Regular Expressions

This is like regex in UNIX but a lot simpler. Below is the formal definition of a regular expression:

1. a for some a in the alphabet Σ
2. ε
3. $\varnothing$
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$ or $(R_1 R_2)$, where R_1 and R_2 are regular expressions
6. (R_1^*), where R_1 is a regular expression

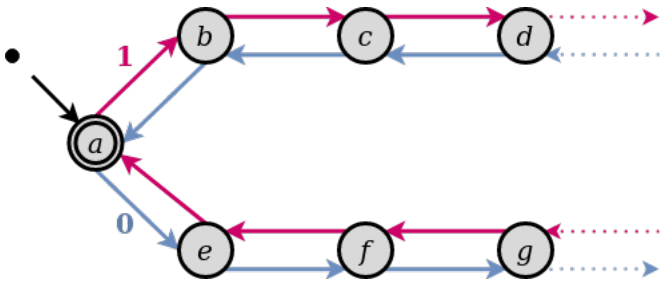
For example, $(0 \cup 1)^*$ is a sequence of as many 0s and 1s

Above, we've already seen the proofs for $(R_1 \cup R_2)$, $(R_1 R_2)$, and R_1^* in NFAs. The NFAs for a, ε, and $\varnothing$ are below:



5.4 Non Regular Languages

A language that isn't regular. It cannot be constructed with a DFA or NFA. An example of one is a language with an equal number of "0"s and "1"s. Since there can be an infinite number of "0"s and "1"s, there would have to be an infinite number of states to account for infinite possibilities where there are more "0"s than "1"s and vice versa. This is shown in the incomplete example below. *Note: **1** transitions are shown in **pink** and **0** transitions are shown in **blue***



State a is the only accept state where there are equal "0"s and "1"s. When a "1" is encountered, we transition to state b to show there's one more "1" than "0". If a "0" were encountered, we'd move back to a to show an equal number of "0"s and "1"s. If instead, a "1" were encountered, we'd move to state c to show there are two more "1"s than "0"s. If a "0" were then encountered, we'd move back to b. Since there can be an infinite number of "0"s and "1"s, there can be infinite possible of differences in counts. Since we'd need infinite states to represent this, we cannot build a *finite* automata to describe this language, proving it's not regular.

5.5 Pumping Lemma

All strings in a regular language have a special property, if the language does not have this property, it is not regular. The property is:

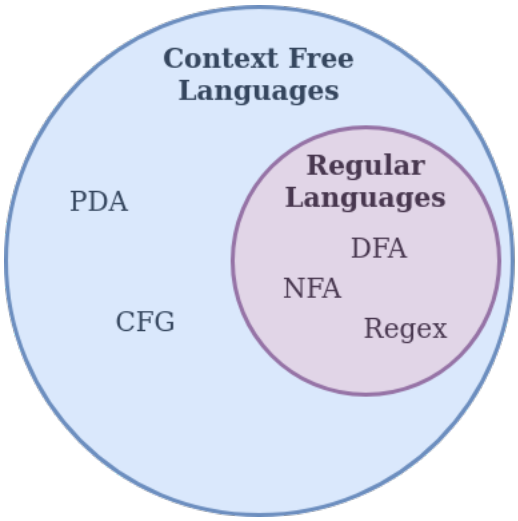
For a regular language A, there is a number p (the pumping length) where if s is any string in A of at least length p, then s can be divided into at least 3 pieces, where $s = xyz$, where the following conditions are met.

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

6 Context Free Languages

We're starting to work our way into larger "sets of computing" or hierarchies of computing.

We started with regular languages which were able to be computed through DFAs and NFAs and regular expressions. Regular languages are a subset of context free languages, which can be computed with a Pushdown Automata (PDA) or a context free grammars (CFG). All regular languages are context free. Not all context free languages are regular.

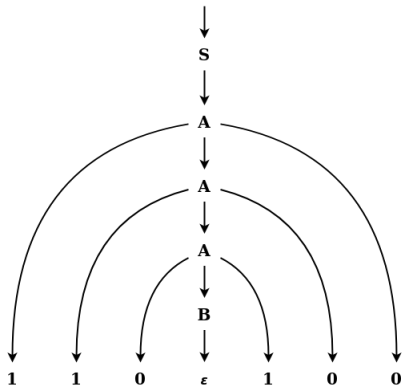


6.1 Context Free Grammars

CFGs are to context free languages as regular expressions are to regular languages. They have sexy recursive structure which allows us to represent all regular languages with a CFG plus other non regular languages.

For example, we said earlier that the language $L = \{w \mid w$ has an equal number of 0s and 1s $\}$ is not regular and showed we could not produce a DFA to accept the string. L is not regular, but it is context free and can be captured with the following CFG:

$S \to A$
$A \to 0A1 \mid 1A0 \mid B$
$B \to \varepsilon$



Productions : The set of substitution rules

Variable : The symbol on the left of a production

Terminal : The product of a production. A cobination of symbols and products

Start Variable : The start of a CFG

The formal definition of a CFG is a 4 tuple $G = (V, \Sigma, R, S)$ where:

1. V is a finite set of variables
2. Σ is a finite set, disjoint from V of terminals
3. R is a finite set of rules, each with the form $A \to w$ where $A \in V$ and $w \in (V \cup \Sigma)^*$
4. $S \in V$ is the start variable

Yields : $uAv \Rightarrow uwv$

- $u \in (V \cup \Sigma)^*$
- $v \in (V \cup \Sigma)^*$
- $A \to w$ is a production in G

Derives : $u \overset{*}{\Rightarrow} v$

- $u \in (V \cup \Sigma)^*$
- $v \in (V \cup \Sigma)^*$

- $u = v$ or $\exists u_1, u_2, \ldots, u_k$ where $u_i \in (V \cup \Sigma)^* \forall i \in [1, k]$ such that $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$

Context Free Languages are closed under union and concatenation. They are not closed under intersection.

Given a context free language L_A recognized by a CFG $G_A = (V_A, \Sigma_A, R_A, S_A)$ and a context free language L_B recognized by a CFG $G_B = (V_B, \Sigma_B, R_B, S_B)$, we can construct the following:

Concatenation

$$L = L_A L_B$$

We can construct a CFG $G = (V, \Sigma, R, S)$:

$$V = V_A \cup V_B$$
$$\Sigma = \Sigma_A \cup \Sigma_B$$
$$R = R_A \cup R_B$$
$$S \to S_A S_B$$

Union

$$L = L_A \cup L_B$$

We can construct a CFG $G = (V, \Sigma, R, S)$:

$$V = V_A \cup V_B$$
$$\Sigma = \Sigma_A \cup \Sigma_B$$
$$R = R_A \cup R_B$$
$$S \to S_A \mid S_B$$

6.1.1 Chomsky Normal Form

A CFG is in Chomsky Normal Form (CNF) if every rule has the form:

$$A \to BC$$
$$A \to a$$

Where:

- a is any terminal $(a \in \Sigma)$
- A, B, and C are any variables $(A, B, C \in V)$
- B and C are not the start state $(B \neq S)$ and $(C \neq S)$
- $S \to \varepsilon$ is allowed

To convert any CFG to Chomsky Normal Form:

1. Introduce a new start state $S_0 \to S$ where S is the former start state
2. Eliminate all rules of the form $A \to \varepsilon$
3. Eliminate all unit rules of the form $A \to B$
4. Ensure the language is still accurate
5. Convert remaining rules to the proper form

Example on pages 118 and 119 of Sipser. Let's start with the following CFG:

$$S \to ASA \mid aB$$
$$A \to S \mid B$$
$$B \to b \mid \varepsilon$$

First, we introduce the new start variable S_0:

$$S_0 \to S$$
$$S \to ASA \mid aB$$
$$A \to S \mid B$$
$$B \to b \mid \varepsilon$$

Next, we remove the ε rule $B \to \varepsilon$. To do this accurately, we need to find every production where B is referenced and add a new rule to make the same production with B replaced by ε.

For example, for $S \to aB$, we will add a rule $S \to a\varepsilon$ or just $S \to a$

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

In doing this, we created a new ε rule $A \to \varepsilon$. The process is the same, however this time the rule $S \to ASA$ contains two As. This time, we need to generate each possible combination replacing A with ε:

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

Let's remove the easy unit rule $S \to S$ first. It's redundant, I don't need to change anything else when removing this.

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

For the unit rule $S_0 \to S$, we can remove this by "duplicating" the productions of S for S_0

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

I can apply this same process for the unit rules $A \to S$ and $A \to B$

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

For the last step, we want to convert all the final rules to a proper form. Note how $S \to ASA$ does not follow the form $A \to BC$.

For each rule $X \to u_1 u_2 \dots u_k$ where $k \geq 3$ and each $u_i \in (V \cup \Sigma)$ we create the rules $X \to u_1 X_1$, $X_1 \to u_2 X_2$, $\dots$, $X_{k-2} \to u_{k-1} u_k$. If $k = 2$, we create the rule $U_i \to u_i$ and replace each occurance of u_i to follow.

To apply this to $S \to ASA$, we add the rules $A \to AA_1$ and $A_1 \to SA$. I've also created a rule $U \to a$ to replace the rule $S \to aB$

Our final CFG in Chomsky Normal Form is:

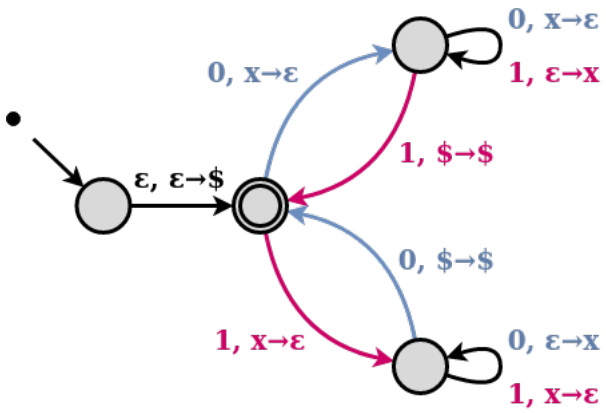
$$
\begin{aligned}
S_0 &\to AA_1 \mid AS \mid SA \mid UB \mid a \\
S &\to AA_1 \mid AS \mid SA \mid UB \mid a \\
A &\to AA_1 \mid AS \mid SA \mid UB \mid a \mid b \\
A_1 &\to SA \\
U &\to a \\
B &\to b
\end{aligned}
$$

6.2 Pushdown Automata

As DFA/NFAs are to regular languages Pushdown Automata (PDA) are to context free languages.

PDA can write to the stack, and read them back later. Just like the stack in any other CS context, you can push and pop symbols and its a last-in, first-out structure.

Remember that problem with the equal 0s and 1s and how thats non-regular, the following pushdown automata can recognize it:

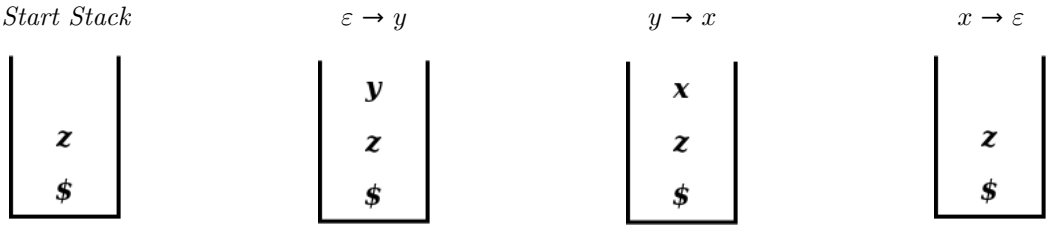


Transitions take a new format:

$$a, b \to c$$

- a is the letter read from input
- b is what we are popping off the stack
- c is what we are pushing onto the stack
- We can also say "b replaces c" on the stack
- If b doesn't exist on the stack, then we cannot transition

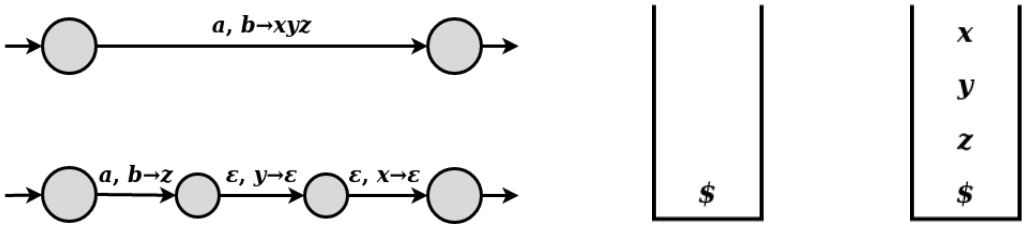
Reading a letter from input is the same as before. Here are a few example stack transistions:

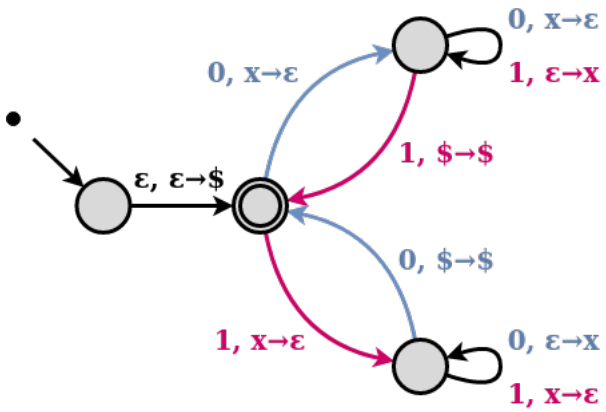


Another thing to note, we need a flag to let us know when we're at the bottom of the stack. We can only check what the top character on the stack is, we can't check if the stack is empty or not. We use $ as our

bottom flag. The first thing we need to do in a PDA is push a \$ on the stack. If we need to check if our stack is empty, we can check if the \$ is at the top.

We can also write a full string onto the stack in a single transition by using the shorthand $a, b \to xyz$. This is because we can expand this into many states:





Now to actually process the above PDA that recognizes the language of an equal number of 0s and 1s:

1. Depending on the first symbol from the string read, we assign one symbol as a "push symbol" and the other as a "pop symbol"
   - If $w \in 0\Sigma^*$ then 0 is the "push symbol" and 1 is the "pop symbol"
   - If $w \in 1\Sigma^*$ then 1 is the "push symbol" and 0 is the "pop symbol"
2. Everytime we encounter a "push symbol", we push an x onto the stack
3. Everytime we encounter a "pop symbol", we pop an x off the stack
4. We can only finish and accept the string when the stack is "empty" and we've seen an equal number of "push" and "pop" symbols (0s and 1s)

6.2.1 Formal Definition

Now a formal definition where a pushdown automata is a 6 tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of states
- Σ is a finite input alphabet
- Γ is a finite stack alphabet
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$ is the transistion function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is is the finite set of accept states

And the formal definition of computation is as follows where w is the input string, r is the sequence of states, and s is the stack.

- $r_0 = q_0$ and $s_0 = \varepsilon$. This means M starts properly on the right start state with an empty stack
- For $i = 0, ..., m-1$ we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$ This is fancy math words for making sure M moves to the next state properly according to the state, stack, and next input symbol.
- $r_m \in F$ The accept state is the last state.

6.3 Converting CFGs to PDAs

Recall the formal definition of a CFG $G = (V, \Sigma, R, S)$ and the formal definition of a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

The easiest thing to start with when constructing P is q_0 and F. There will be one start state (q_0) and one accept state (q_a). These will represent putting \$ on the stack and taking it off when we're done, only accepting when we have an empty stack.

$$q_0 = q_0$$

$$F = \{q_a\}$$

Q will contain q_0 and q_a. It will also contain q_l or q_{loop}. The process we follow when making a PDA from a CFG will be pushing and popping symbols in and out of one state until the stack is empty. As seen before with the string shorthand for stacks, we'll also need a set of "extra states" for the expanded shorthand which we'll call E. This makes:

$$Q = \{q_0, q_l, q_a\} \cup E$$

Another easy one, we know our input alphabet has the be the same alphabet as the CFG. Then based on the looping structure, the stack alphabet will be the alphabet of the CFG in addition to all the variables and the \$:

$$\Sigma = \Sigma$$

$$\Gamma = \Sigma \cup V \cup \$$$

The last thing to define is the transition function. Before I start, I'm gonna quick go over the transition function notation and the order of arguments (otherwise I will forget):

$$\delta(q_f, w_i, s_o) = \{(q_t, s_u)\}$$

q_f = The "from" state
w_i = The input letter being read
s_o = The element on the stack being popped
q_t = The "to" state
s_u = The element being pushed on the stack

Now, let's construct a transistion function. As stated before, the first thing we need to do is put \$ on the stack. We also need to put the starting variable S on the stack too. This gives us a starting point to loop on.

$$\delta(q_0, \varepsilon, \varepsilon) = \{(q_l, S\$)\}$$

Let's also define the accepting transistion where we are popping the \$ flag and have an empty stack:

$$\delta(q_l, \varepsilon, \$) = \{(q_a, \varepsilon)\}$$

We then have two other "types" of transistions:

1. The top of the stack is a variable
2. The top of the stack is a terminal

In case 1, we have a variable on the top of the stack. This variable needs to be replaced with the things it produces. For each variable, we're going to create a transistion that consumes the variable from the stack and pushes its production onto the stack. This will create the "extra" shorthand states E that will be unioned with Q. The general shorthand can be written as:

$$\delta(q_l, \varepsilon, A) = \{(q_l, w)\} \text{ for rule } A \to w$$

To formally expand this into the proper "extra" states and transistions, we'd have:

$$\begin{aligned}
w &= w_1 w_2 \dots w_k && \text{The input string of length } k \\
E &= E \cup \{e_1, e_2, \dots, e_{k-1}\} && \text{The extra states} \\
\delta(q_l, \varepsilon, A) &= \{(e_1, w_k)\} && \text{ for rule } A \to w \\
\delta(e_i, \varepsilon, \varepsilon) &= \{(e_{i+1}, w_{k-i})\} && \forall i \in [2, k-2] \\
\delta(e_{k-1}, \varepsilon, \varepsilon) &= \{(q_l, w_1)\}
\end{aligned}$$

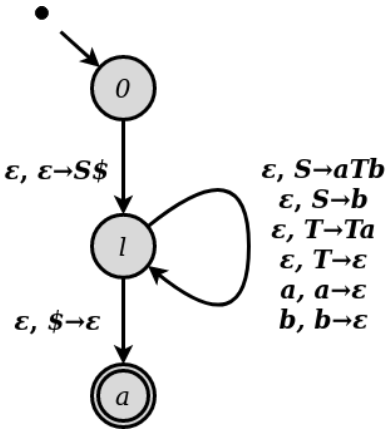
In case 2, where we have a terminal, its means that a variable had been seen earlier and told us to expect a terminal input. For the rule $S \to aTb$, we consume the S and put b, T, and a on the stack (in that order). Since the top of the stack is an a, we know we need to see an a in input and consume it before processing T or b. This gives the following transistion:

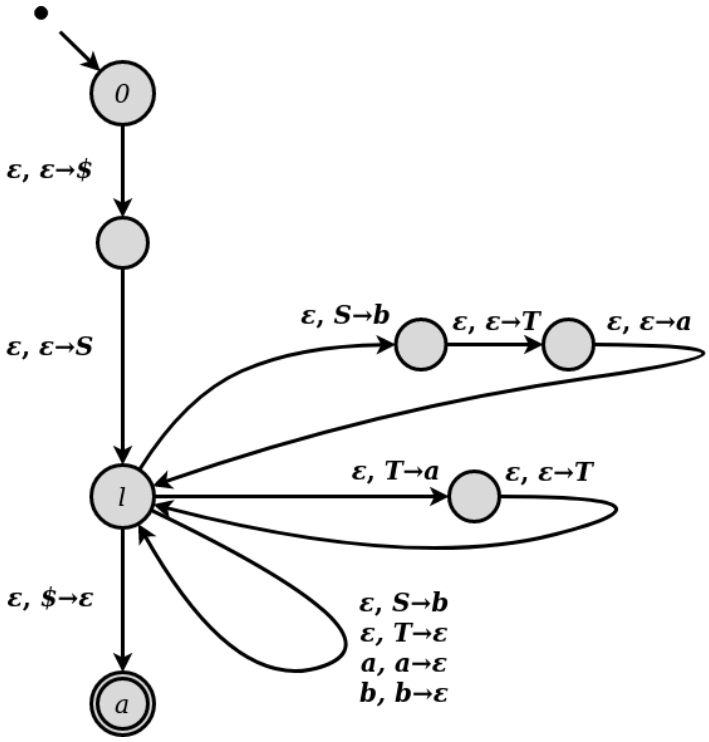
$$\delta(q_l, a, a) = \{(q_l, \varepsilon)\} \text{ for terminal } a$$

In practice, this looks like:

Note: Pages 117-119 From Sipser

$S \to aTb \mid b$
$T \to Ta \mid \varepsilon$





6.4 CYK Algorithm

The CYK algorithm takes a CFG G in Chomsky Normal Form and checks if a string $w \in L(G)$.

String w of length n is $\in L(G)$ if and only if $S \in X_{1n}$. The algorithm constructing the set X_{1n} out of w operates $\in O(n^3)$. We construct each set by doing the following:

$$
\begin{aligned}
w &\neq \varepsilon \\
w &= a_1 a_2 \dots a_n \\
X_{ij} &= \{A \in V \mid A \overset{*}{\Rightarrow} a_i a_{i+1} \dots a_j\}
\end{aligned}
$$

We start with the base case where X_{ii} the variable A that creates the single character a_i

$$X_{ii} = \{A \in V \mid A \Rightarrow a_i\}$$

Inductively, we can then build X_{ij}:

$$X_{ij} = \bigcup_{k=i}^{j-1} \{A \in V \mid A \to BC, B \in X_{ik}, C \in X_{k+1j}\}$$

6.5 Pumping Lemma

Just as before with regular languages, there is a pumping lemma for context free languages.

If A is a context free language, then there is a number p (the pumping length) where if s is any string in A of at least length p, then s can be divided into the following 5 part structure $s = uvxyz$ where

- for each $i \geq 0, uv^ixy^iz \in A$
- $|vy| > 0$
- $|vxy| \leq p$

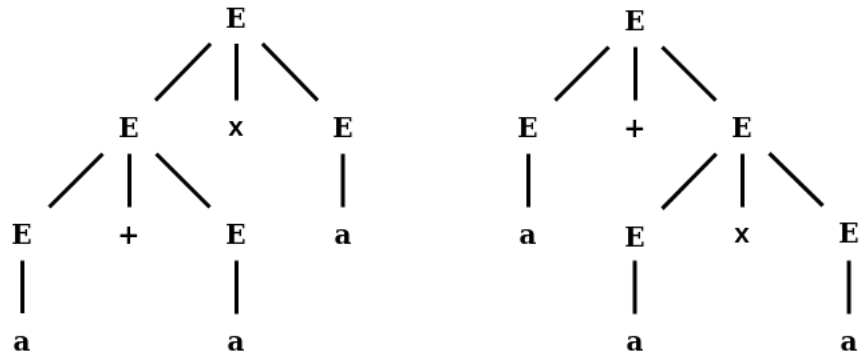
6.6 Ambigious Grammars

An ambigious Context Free Grammar is a CFG that can generate the same string in many different ways.

In the following grammar, $a + a \times a$ is generated ambiguously:

$$E \to E + E \mid E \times E \mid (E) \mid a$$

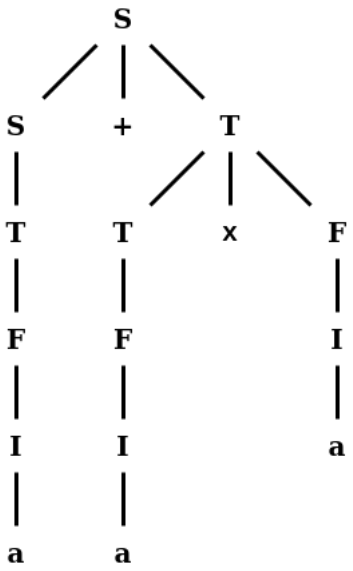
The following two parse trees are generated:



The unambigious version of this grammar is:

$$\begin{aligned} S &\to T \mid S + T \\ T &\to F \mid T \times F \\ F &\to I \mid (S) \\ I &\to a \end{aligned}$$

Here, we use T to generate *terms* from addition, F to generate *factors* from multiplication, and I to generate *identifiers*. In this CFG, addition is the lowest priority. Using this CFG, only one parse tree can be generated:



A grammar is **inherently ambigious** if it cannot be written unambigiously. The following language cannot be written unambigiously:

$$L = \{a^i b^j c^k \mid i = j \vee j = k\}$$

$S \to IC \mid AK$	$i = j$ or $j = k$
$I \to aIb \mid \varepsilon$	$i = j$
$C \to Cc \mid \varepsilon$	A lot of cs
$K \to bKc \mid \varepsilon$	$j = k$
$A \to Aa \mid \varepsilon$	A lot of as

In the case where $i = j = k$, the CFG can take the $i = j$ "path" through IC or it can take the $j = k$ "path" through AK. Both can generate the $i = j = k$ string.

There is not an algorithm that can determine if a grammar is inherently ambigious, nor is there one to convert an ambigious grammar to an unambigious grammar.

6.7 Deteministic Context Free

We know that the difference between determinism and nondeterminism is the transistion function δ. If δ goes to a single state, it's deterministic. If δ goes to a set of states, it's nondeterministic.

Deterministic regular languages and nondeterministic regular languages are equivalent.

Deterministic Turing recognizable languages and nondeterministic Turing recognizable languages are equivalent (covered later).

Deterministic context free languages and nondeterministic contex free languages are *not* equivalent.

Some properties of DCFLs:

- DCFLs are a proper subset of CFLs
- DCFLs are unambigious
- DCFLs are closed under complement
- DCFLs are not closed under union or intersection

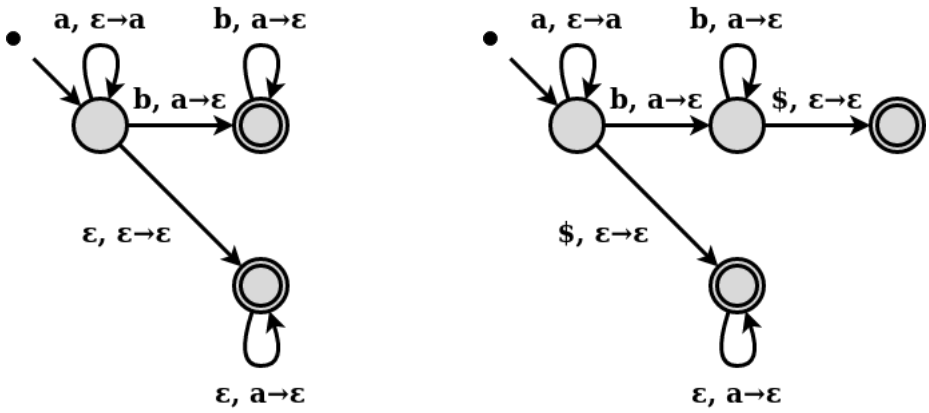
Looking at the language L, we can create a language $L\$$ where $w' \in L\$$ such that $w' = w\$$ where $w \in L$. A language L is a DCFL if and only if $L\$$ is a DCFL.

6.7.1 Deterministic Push Down Automata

A Deterministic Push Down Automata (DPDA) is a 6-tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ such that:

- Q is the set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- δ is the transistion function defined as $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$ with an extra requirement that each assigned value contains at most one pair
  - $\forall q \in Q, a \in \Sigma$ and $x \in \Gamma$, exactly one of the following values is not the empty set $\varnothing$
    - $\delta(q, a, x)$
    - $\delta(q, a, \varepsilon)$
    - $\delta(q, \varepsilon, x)$
    - $\delta(q, \varepsilon, \varepsilon)$
- $q_0 \in Q$ is the start state
- $F \subset Q$ is the set of accepting states

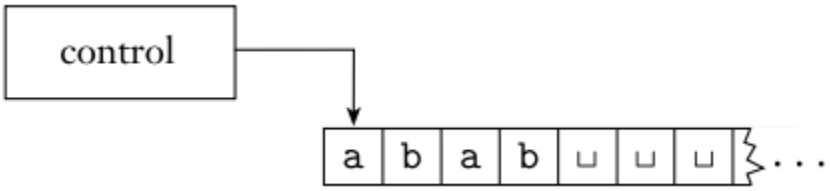
Given the language $L = a^* \cup \{a^n b^n \mid n > 0\}$, below are two PDAs that accept the language. On the left, the NPDA and on the right, the DPDA.



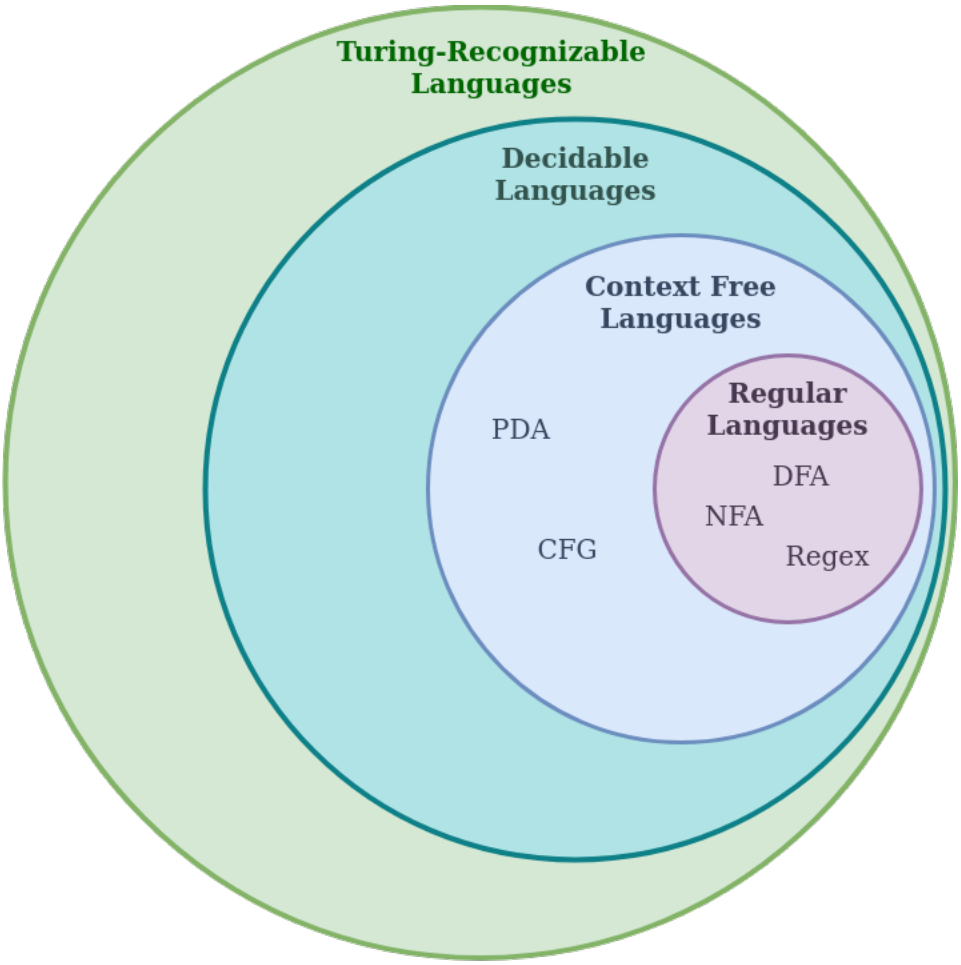
On the other hand, $L = \{a^i b^j c^k \mid i = j \wedge j = k\}$ cannot be deterministic context free because it is inherently ambigious

7 Turing Machines

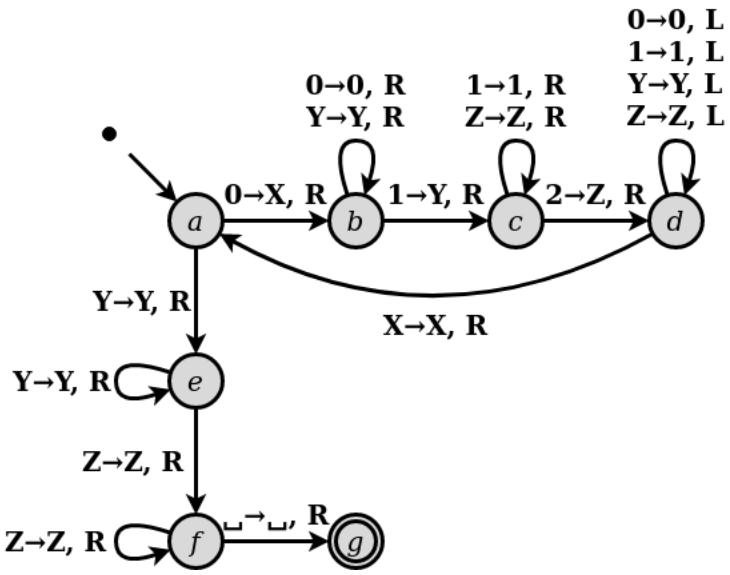
My homie Alan Turing thought of these bad boys. A Turing machine is a simple computer that uses an infinite tape as memory. This tape starts with the input string and is infinitely empty (␣) afterwards. The head can move left and right and read and write symbols. Reject and accept states happen immidiately.



We can now update our heirarchy we saw before with Turing Machines:



To show a TM recognizing a non context free language, let's walk through the computation of the language $L = \{0^n 1^n 2^n \mid n \geq 1\}$

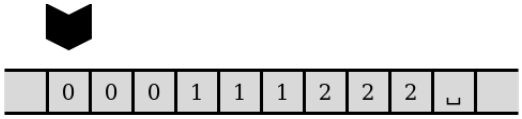


We'll notice transistions take a new format:

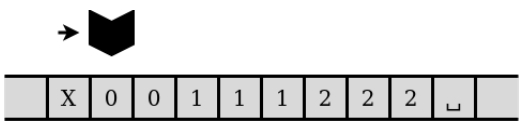
$$r \to w, d$$

- r : The symbol on the tape underneath the head being read
- w : The symbol being written on the tape which "overwrites" the existing symbol
- d : The direction the head should move. The head will only move one space
  - L : Left
  - R : Right

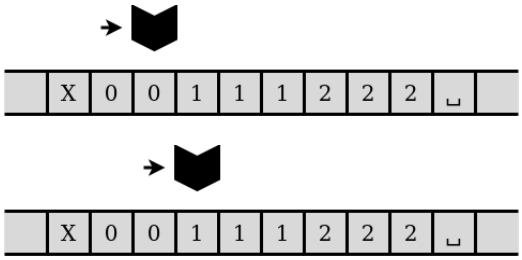
To read the string, we're going to load the full string into the tape followed by infinite blanks (␣).



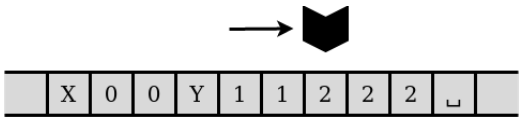
We're starting in state a. Underneath the head, there is a 0 which means we progress to state b. As we transistion, the 0 becomes an X and the head moves right:



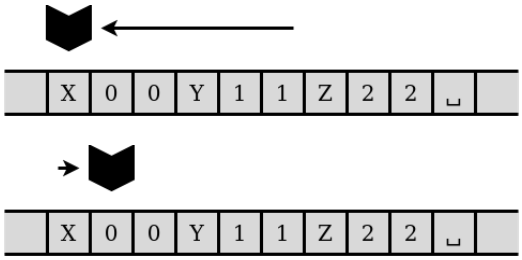
We're going to stick around in state b beause we're going to read and "overwrite" these 0s with 0s. We're going to do this as long as we see a 0 or a Y. Essentially, we're just moving the head right until we read a 1:



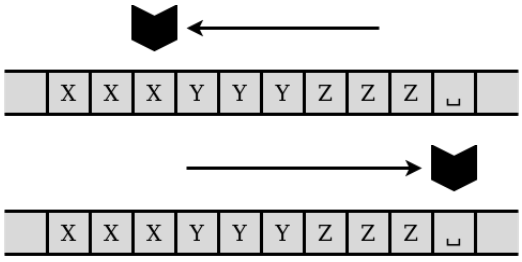
Now when we encounter a 1, we change it to a Y and push on. The same shifting process will repeat. I'm going to start generalizing some of the more repetitive movements.



Now when we read a 2, we're going to move left until we read an X, then restart the process:



We do this until we run out of 0s. Once there are no more, we read Ys and Zs until we hit a ␣, which is when we finally accept the string!



As seen here, I was able to generalize some of the transitions by describing the behavior of the TM. In this class, we're not going to have to write out formal definitions of Turing Machines with states and functions. We'll be able to use algorithmic steps to describe them instead.

7.1 Computing

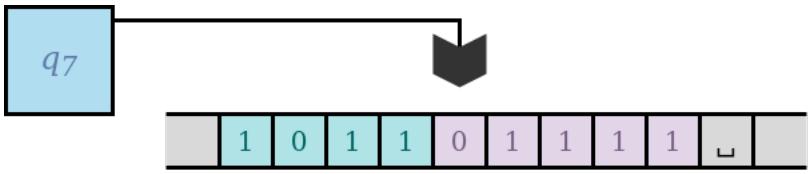
Turing-Recognizable : A language is Turing-Recognizable if some Turing machine recognizes it

Co-Turing-Recognizable : A language is Co-Turing-Recognizable if some Turing machine recognizes its complement

Configuration : A setting of the state of a Turing machine. It contains the current state, the current tape contents, and the location of the head

Configurations of a Turing machine M have the form uqv, where $u, v \in \Gamma^*$ and $q \in Q$.

The following Turing machine has the configuration **1011q$_7$01111**. (u is in **teal**, q is in **blue**, and v is in **purple**)



Looking back at the example before for the TM M that recognized $L = \{0^n 1^n 2^n \mid n \geq 1\}$, the images of the tape would be described with the following configurations:

$$
\begin{aligned}
C_1 &= q_a 000111222 \\
C_2 &= X q_b 00111222 \\
C_3 &= X0 q_b 0111222 \\
C_4 &= X00 q_b 111222 \\
C_5 &= X00Y q_c 11222 \\
&\ldots = \ldots \\
C_8 &= X00Y11 q_c 222
\end{aligned}
\qquad
\begin{aligned}
C_9 &= X00Y11 q_d Z22 \\
&\ldots = \ldots \\
C_{15} &= q_d X00Y11Z22 \\
C_{16} &= X q_d 00Y11Z22 \\
C_{17} &= XX q_a 0Y11Z22 \\
&\ldots = \ldots \\
C_k &= XXXYYYZZZ q_g
\end{aligned}
$$

There are four types of configurations:

1. Start
2. Accepting
3. Rejecting
4. Halting

A Turing machine also has three different "outputs". It can *accept* or *reject* input, or it can loop infinitely. If a TM halts on all input (doesn't infinitely loop for any input) it's called a decider. This is covered more later.

We say that M accepts a string w if there is a sequence of configurations $C_1, C_2, \ldots, C_k$ such that:

1. C_1 is the start configuration of M on w
2. Each C_i yields C_{i+1}
3. C_k is an accept configuration

We can describe a Turing machine in a few ways:

- **Formal :** Use the formal definition, choose a set of Q states, etc. This is overkill for this class
- **Implementation-Level :** "Swipe right until...", "Move the head to..."
- **High-Level :** "Simulate A on input w"

When describing inputs to Turing machines and doing high-level algorithm analysis, our inputs aren't always going to be a string w. It may be a graph G or a DFA D or some other non string object. We define the following new notations to handle these cases:

- O is an object
- $\langle O \rangle$ is the object O encoded as a string
- $O_1, O_2, \ldots, O_k$ is a collection of k objects
- $\langle O_1, O_2, \ldots, O_k \rangle$ is the collection encoded as a string

7.2 Deterministic Turing Machines

The formal definition of a Detministic Turing Machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ such that:

Q The set of states
$\Sigma \subset \Gamma$ The input alphabet
Γ The tape alphabet. It also includes the blank symbol
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
$q_0 \in Q$ the start state
$q_a \in Q$ the accept state
$q_r \in Q$ the reject state

A DTM is your standard TM with a single infinite tape that I already covered in the previous sections. The next sections are variants of TMs, but they are all equivalent and can be converted into each other and be simulated by each other easily.

7.3 Multitape Turing Machines

We can create a Turing Machine with multiple tapes. We redefine δ as the following:

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

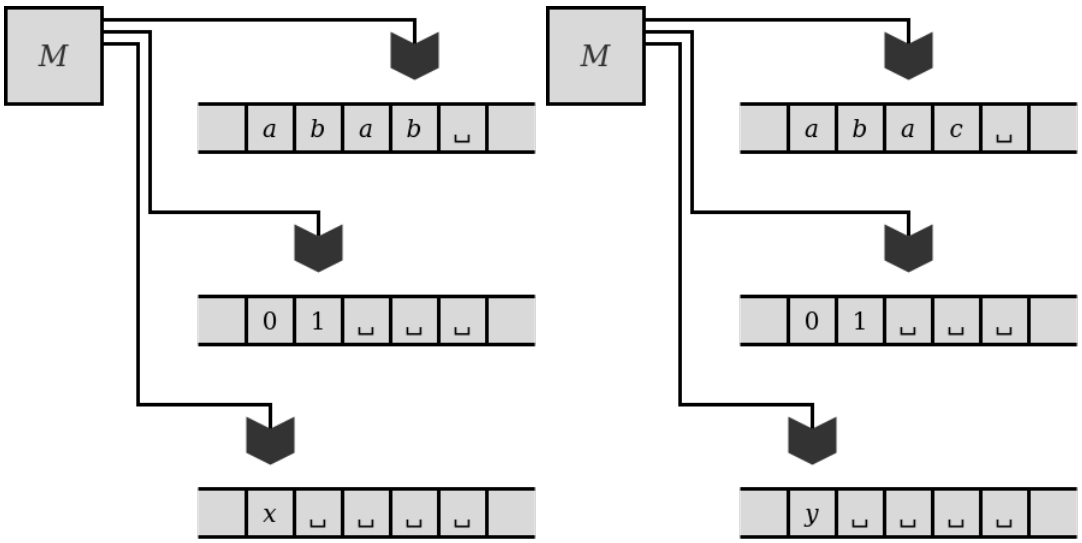
Which looks like:

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, A_1, \ldots, A_k)$$

Where:

$$\begin{aligned} q_i &\in Q \\ q_j &\in Q \\ a_m &\text{ a tape symbol} \\ b_m &\text{ a tape symbol} \\ A_m &\text{ a direction} \end{aligned}$$

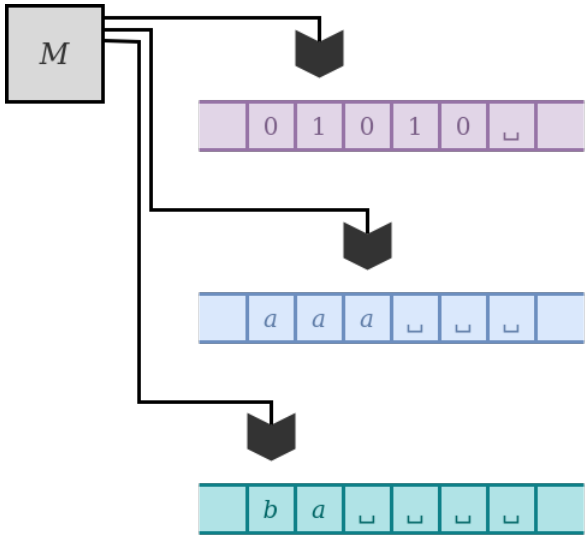
Below is a 3 tape TM before and after the transition $\delta(q, b, 1, x) = (q', c, 1, y, L, R, S)$:



7.3.1 Converting a Multitape TM to a Single Tape TM

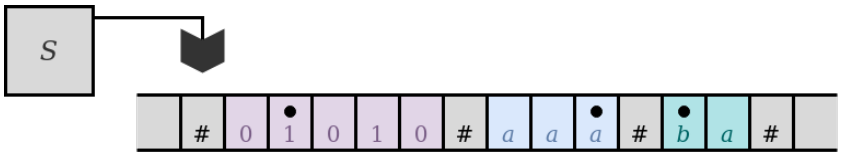
Since a Single Tape TM has infinite tape, we have infinite space to contain the multiple infinite tapes of a Multitape TM.

Let's say we have a multitape TM M with three tapes that we want to convert into a single tape TM This is figure 3.14 from Sipser.

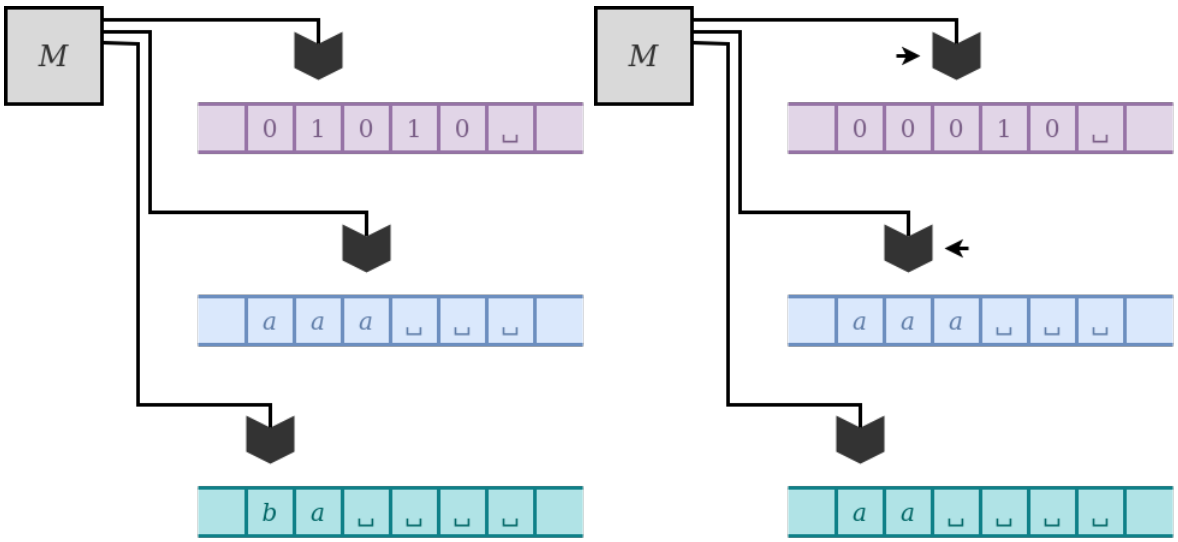


There's two main parts to the conversion. First, it's adding new "structure". Second, how do we use that structure?

To buid the Single-Tape Turing Machine, we are going to just concatenate the k tapes and put a # delimiter between them. To keep track of the heads in each tape, we add a dot above the tape location the head is at. These dotted symbols are just added to the alphabet. This creates the following Single-Tape Turing Machine:



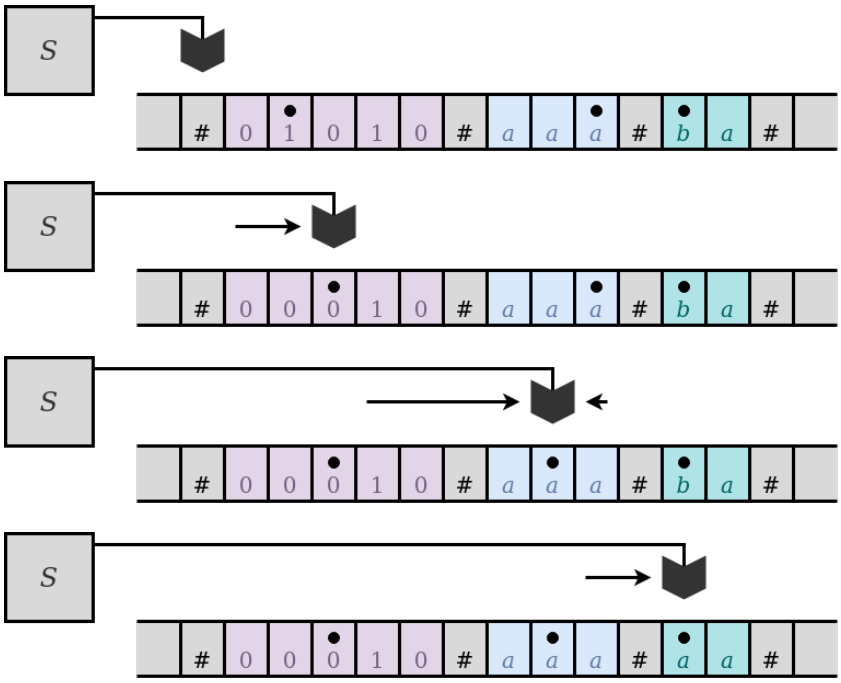
To simluate the behavior of a Multi-Tape, recall that the δ function affects all tapes at the same time. We get the following behavior after the transition $\delta(q, 1, a, b) = (q', 0, a, a, R, L, S)$:

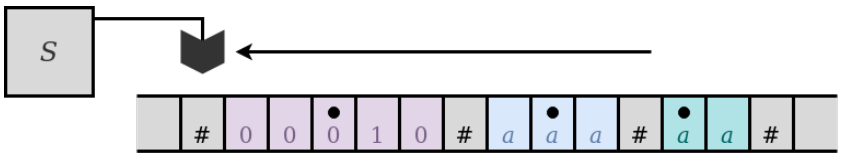


A Single-Tape TM won't be able to do this in one transition. Instead, we'll do a full swipe of the tape. One each swipe we:

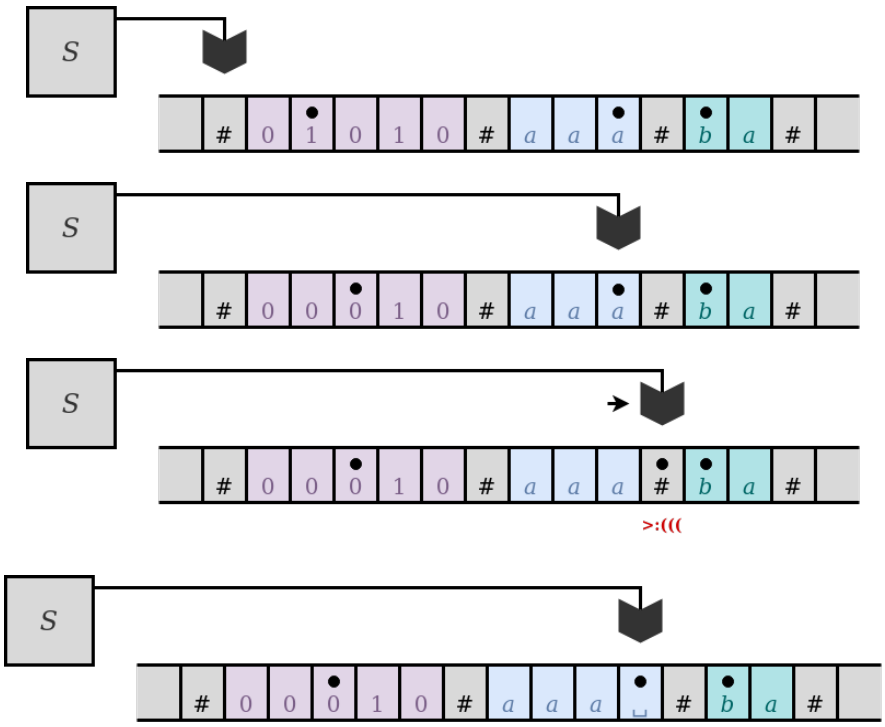
1. Swipe we find a #. This marks the start of tape i of k.
2. Swipe to find the dotted head on that tape.
3. Perform the required transition of that tape.
4. Repeat for each tape.
5. Once the k tape has been completed, swipe back to the beginning # of the first tape. This prepares us for the next Multi-Tape transition.

This process is covered below:





If there is a situation where a head dot would be placed on a # delimiter, that tape has "run out of space". This would imply that a Multi-Tape TM has a finite tape. To preserve infinite tape, shift every character to the right one space. This is shown below if the δ applied to figure 3.14 were $\delta(q, 1, a, b) = (q', 0, a, a, R, R, S)$:



7.4 Non-Deterministic Turing Machines

Similar to how regular languages are recognized by both NFAs and DFAs, a language is Turing-recognizable if and only if some Non-deterministic Turing Maching recognizes it.

Again, determinism is based on the δ transition function. If δ produces a single state, it's deterministic. If δ produces a set of states, it's non-determistic and will have a branching pattern. As long as one branch accepts, the entire machine accepts. Even if there is an explicit reject state reached in a branch, the accept state takes priority. We define the δ function as:

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

An NTM is a **decider** is it halts on all its branches of all inputs.

7.4.1 NTMs to DTMs

We can convert every NTM into an equivalent multitape DTM by doing the following:

1. Place input string w on tape 1
2. Copy tape 1 to tape 2
3. Simulate N on tape 2 on a given branch of it's nondeterministic computation. Consult tape 3 for the next transition:
    - Abort this branch if the transition is invalid
    - Abort this branch if the branch has been exhausted
    - Accept if an accepting configuration is encountered
4. Update the string on tape 3 to the next one
5. Go to stage 2

7.5 Enumerators

An enumerator is basically a Turing Machine with a printer attached. The printer will print and output strings. If an enumerator doesn't halt, it may print out an infinite collection of strings.

Enumerators differ from Turing Machines in that they start with a blank input tape and they will

Since all the Turing machine variations are equivalent, a language is Turing-recognizable if and only if some enumerator enumerates it.

7.5.1 Enumerators to Turing Machines

Given : An enumerator E

Construct : An equivalent Turing Machine M

M = "On input w

1. Run E
2. Everytime E outputs a string, compare it with w
3. If E eventually outputs a string that matches w, *accept* "

7.5.2 Turing Machines to Enumerators

Given : A Turing Machine M

Construct : An equivalent enumerator E

Let $s_1, s_2, \ldots, s_i, \ldots$ be the enumeration of all the strings in $\Sigma*$. For example, given $\Sigma = \{a, b, c\}$:

$$s_1 = \varepsilon \qquad s_5 = aa$$
$$s_2 = a \qquad s_6 = ab$$
$$s_3 = b \qquad s_7 = ac$$
$$s_4 = c \qquad \ldots = \ldots$$

We then can construct E as:

E = "Ignore the input:

1. Repeat the following for $i = 1, 2, 3, \ldots$
   - Run M for i steps on each input, $s_j \in \{s_1, \ldots, s_i\}$
   - If any computations accept, print s_j

"

This is a little confusing to understand so:

i	**Input to M**	**Stop after...**
$i = 1$	s_1	1 step
$i = 2$	s_1	2 steps
	s_2	2 steps
$i = 3$	s_1	3 steps
	s_2	3 steps
	s_3	3 steps
...	...	...

This structure allows us to check all possible inputs. One may ask, "wHy cAn'T wE jUsT dO tHiS":

E_{bad} = "Ignore the input:

1. Repeat the following for $i = 1, 2, 3, \ldots$
   - Run M on s_i
   - If M accepts, print s_i

"

Let's say M accepts s_2, but doesn't halt on s_1. If we tried to run E_{bad} under this condition, it would get stuck in the running M stage and wouldn't check any strings past s_1.

So the reason we use this seemingly strange steps approach is so that as i approaches ∞, we will check each string for an infinite number of steps and can simulate infinite loops without actually having any.

8 Computability

Turing-Recognizable : A language is Turing-Recognizable if some Turing machine recognizes it

Co-Turing-Recognizable : A language is Co-Turing-Recognizable if some Turing machine recognizes its complement

Turing-Recognizable languages are closed under the following properties:

- Concatenation
- Star
- Intersection
- Union

8.1 Proof of Closed Properties

Concatenation : Given two Turing-recognizable languages L_a and L_b that are recognized by machines M_a and M_b, I can construct machine M that recognizes the concatenation $L = L_a L_b$

M = "On input w

1. Run M_a on w
2. If M_a halts and rejects, *reject*
3. If M_a halts and accepts, continue
4. Run M_b where the starting configuration C_1 of M_b is the ending configuration C_k from M_a
5. If M_b halts and accepts, *accept*. If M_b halts and rejects, *reject* "

Star : Given a Turing-recognizable language L that is recognized by machine M, I can construct a machine M' that recognizes the language $L' = L^*$

M' = "On input w

1. Run M on w
2. If M halts and rejects, *reject*
3. If there is no more w to be processed, *accept*
4. Run M again where the starting configuration C_1 of this run is the ending configuration C_k from the previous run
5. Go to Stage 2 "

Intersection : Given two Turing-recognizable languages L_a and L_b that are recognized by machines M_a and M_b, I can construct machine M that recognizes the intersection $L = L_a \cap L_b$

M = "On input w

1. Run M_a on w and M_b on w "simultaneously" by alternating between machines step by step (Run one step of M_a, then run a step of M_b, then the next step of M_a, etc)
2. If either machine rejects, *reject*
3. If both machines halt and accept, *accept* "

Union : Given two Turing-recognizable languages L_a and L_b that are recognized by machines M_a and M_b, I can construct machine M that recognizes the union $L = L_a \cup L_b$

M = "On input w

1. Run M_a on w and M_b on w "simultaneously" by alternating between machines step by step (Run one step of M_a, then run a step of M_b, then the next step of M_a, etc)
2. If either machine accepts, *accept*
3. If both machines halt and reject, *reject* "

8.2 Decidability

Decider : A Turing machine that halts on all inputs

Turing-Decidable : A language is Turing-Decidable (or just decidable) if there is a decider M recognizing it

A language is decidable if it is both Turing-recognizable and Co-Turing-recognizable. This means decidable languages, being a subset of Turing-recognizable languages, are closed under the following properties:

- Concatenation
- Star
- Intersection
- Union
- Complementation

8.2.1 Proof of Closed Properties

Concatenation : Given two decidable languages L_a and L_b that are decided by the machines M_a and M_b, I can construct a machine M that decides the concatenation $L = L_a L_b$.

M = "On input w

1. Run M_a on w
2. If M_a rejects, *reject*
3. Run M_b where the starting configuration C_1 of M_b is the ending configuration C_k from M_a
4. If M_b accepts, *accept*. If M_b rejects, *reject* "

Star : Given a decidable language L that is decided by machine M, I can construct a machine M' that decides the star $L' = L^*$

M' = "On input w

1. Run M on w
2. If M rejects, *reject*
3. If there is no more w to be processed, *accept*
4. Run M again where the starting configuration C_1 of this run is the ending configuration C_k from the previous run
5. Go to Stage 2 "

Intersection : Given two decidable languages L_a and L_b that are decided by the machines M_a and M_b, I can construct a machine M that decides the intersection $L = L_a \cap L_b$

M = "On input w

1. Run M_a on w
2. If M_a rejects, *reject*
3. Run M_b on w
4. If M_b accepts, *accept*. If M_b rejects, *reject* "

Union : Given two decidable languages L_a and L_b that are decided by the machines M_a and M_b, I can construct a machine M that decides the union $L = L_a \cup L_b$

M = "On input w

1. Run M_a on w
2. If M_a accepts, *accept*
3. Run M_b on w
4. If M_b accepts, *accept*. If M_b rejects, *reject* "

Complementation : Given a decidable language L that is decided by machine M, I can construct a machine M' that decides the complement $L' = \bar{L}$

M' = "On input w

1. Run M on w
2. If M accepts, *reject*. If M rejects, *accept* "

8.2.2 Decidable Problems

These were taken from the slides, and I felt they were helpful to keep around.

Accept DFA : $A_{DFA} = \{\langle A, w\rangle \mid A \text{ is a DFA that accepts input } w\}$

M = "On input $\langle A, w\rangle$

1. Simulate A on input w
2. If the simulation ends in an accept state, *accept*. If it ends in a non-accepting state, *reject*. "

Accept NFA : $A_{NFA} = \{\langle A, w\rangle \mid A \text{ is a NFA that accepts input } w\}$

N = "On input $\langle A, w\rangle$

1. Convert NFA A to an eqiuvalent DFA B
2. Simulate B on input w
3. If the simulation ends in an accept state, *accept*. If it ends in a non-accepting state, *reject*. "

Accept Regular Expression : $A_{REX} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$

P = "On input $\langle R, w\rangle$

1. Convert R into an equivalent NFA B
2. Run TM N on input $\langle B, w\rangle$
3. If N accepts, *accept*. If N rejects, *reject*. "

Empty DFA : $E_{DFA} = \{\langle A, w\rangle \mid A \text{ is a DFA and } L(A) = \varnothing\}$

Q = "On input $\langle A\rangle$

1. Mark the start state of A
2. Repeat until no new states are marked
   - Mark any state that has a transition coming into it from any state that is already marked
3. If no accept state is marked *accept*; otherwise *reject*. "

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

R = "On input $\langle A, B\rangle$

1. Construct DFA C recognizing the symmetric difference of $L(A)$ and $L(B)$
2. Run TM Q (from language E_{DFA}) on input $\langle C\rangle$
3. If Q accepts, *accept*. If Q rejects, *reject* "

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

S = "On input $\langle G, w\rangle$

1. Convert G to an equivalent grammar in Chomsky Normal Form
2. Run CYK algorithm on w
3. If CYK accepts, *accept*; if it rejects, *reject* "

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

T = "On input $\langle G\rangle$

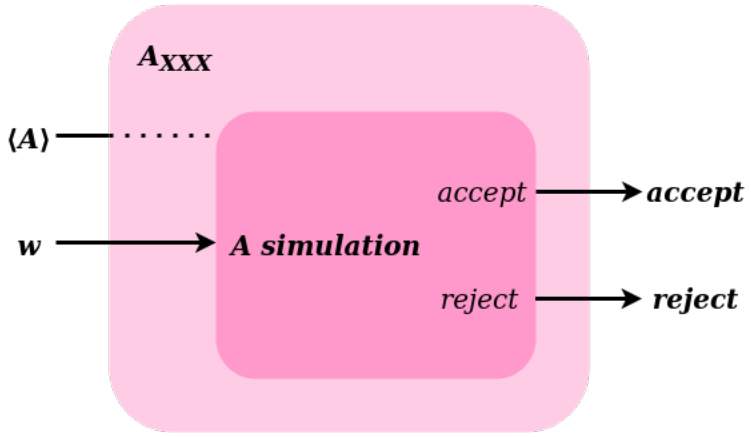
1. Mark all terminal symbols of G
2. Repeat until no new variables are marked:
   - Mark any variable A where G has a rule $A \to U_1U_2\ldots U_k$ and each symbol $U_1, U_2, \ldots, U_k$ has already been marked
3. If the start variable is marked, *accept*; otherwise *reject* "

8.3 Universal Turing Machine

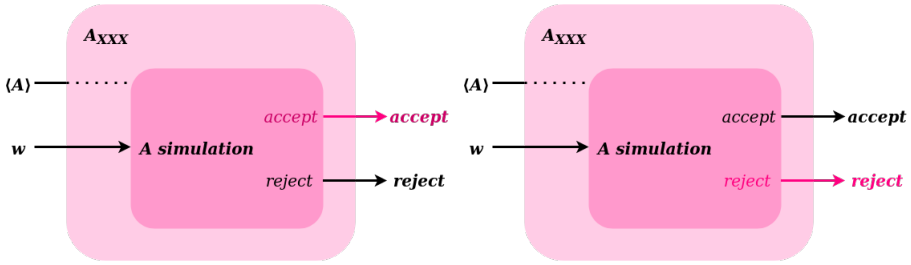
Above we showed the following problems were decidable:

$$A_{DFA} = \{\langle A, w\rangle \mid A \text{ is a DFA that accepts input } w\}$$
$$A_{NFA} = \{\langle A, w\rangle \mid A \text{ is a NFA that accepts input } w\}$$
$$A_{REX} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$$
$$A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates } w\}$$

These all follow the same type of structure where we simulate the input w on the encoded computation device.



Then based on the output of the computation device, we accepted or rejected:



These were decidable problems because regular languages and context free languages are a subset of decidable languages. This means that DFAs/NFAs/Regular Expressions/CFGs/PDAs will halt on all inputs. This does not apply to Turing-recognizable languages.

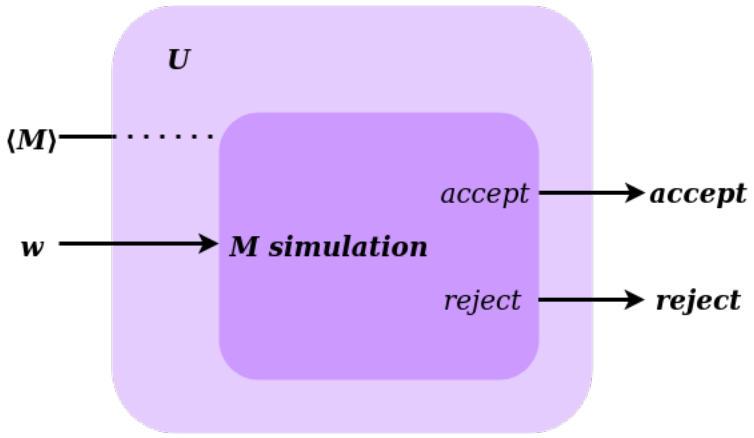
Let's construct the Universal Turing Machine U, it recognizes the following language:

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

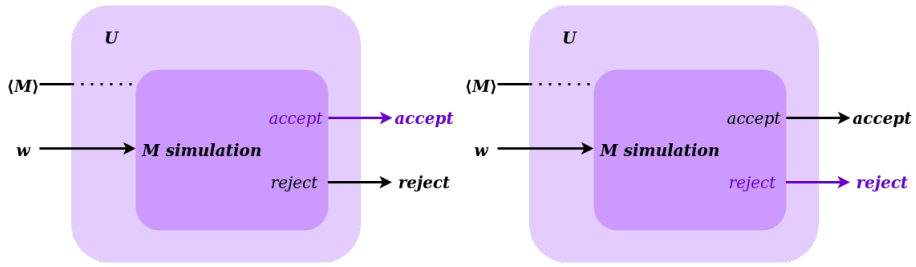
U = "On input $\langle M, w \rangle$. where M is a Turing Machine and w is a string

1. Simulate M on input w
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject* "

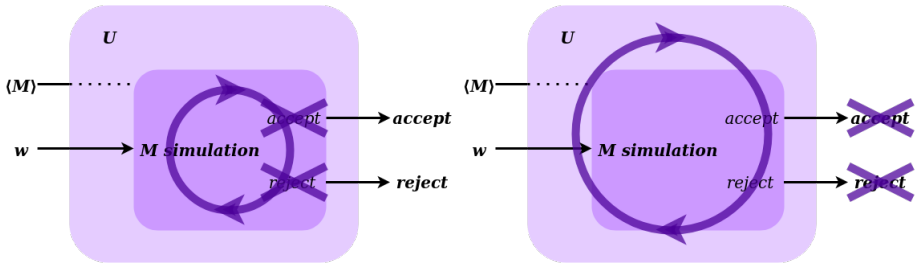
At first glance, this gives us the same set up as before:



And we start out with the same set up as before. When the simulation of M accepts, we *accept*. When the simulation of M rejects, we *reject*:



However, now we have a third option, what if the simulation of M doesn't halt? Now U won't be able to halt because it will be waiting for the simulation to end, which isn't going to happen.



A_{TM} is Turing-recognizable, but it is not decidable.

8.4 The Halting Problem

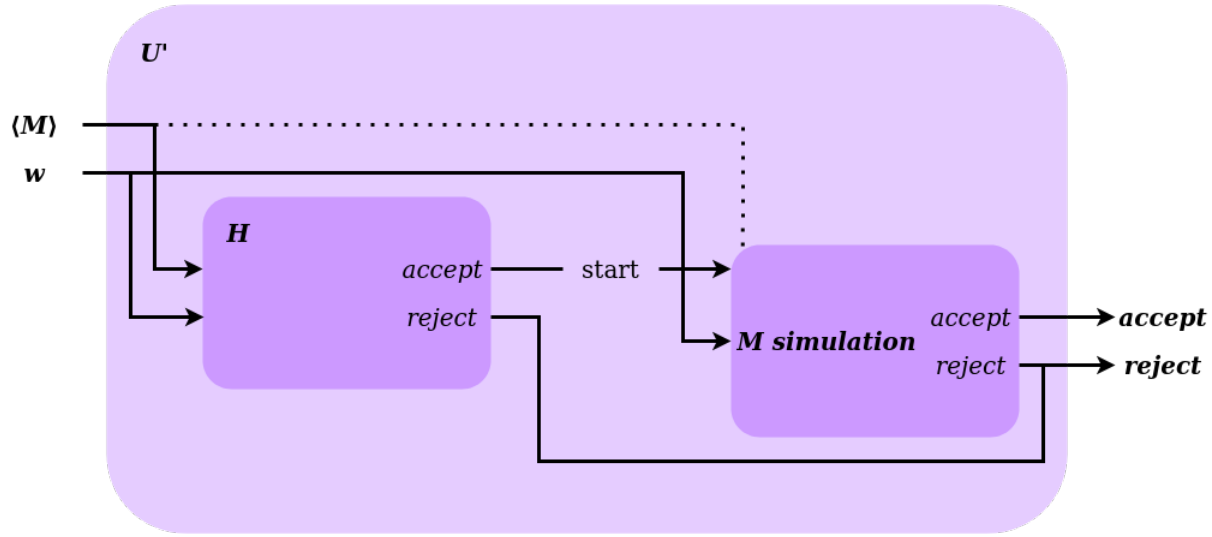
A_{TM} would've been decidable if M were decidable. If M halted, then so would A_{TM}. Let's introduce a new Turing Machine H for the following language:

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

Now let's reconstruct A_{TM} using H:

U' = "On input $\langle M, w\rangle$ where M is a Turing machine

1. Run H on input $\langle M, w\rangle$
2. If R rejects, *reject*
3. If R accepts, simulate M on input w
4. If M accepts, *accept*. If M rejects, *reject* "



However, it's not possible to check if M halts. Since there are an infinite number of configurations in a Turing Machine, there's no way to check for an infinite loop. A TM may enter a different configuration each transition and not halt.

8.5 Post Correspondence Problem

The Post Correspondence Problem is commonly called PCP. Sadly, it's not quite as exciting as the hallucinogenic drug.

We take a collection of dominoes as input:

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

We want to find a sequence $i_1, i_2, \ldots, i_l$ such that we find a matching string between the top and the bottom:

$$t_{i_1} t_{i_2} \ldots t_{i_l} = b_{i_1} b_{i_2} \ldots b_{i_l}$$

We can use a domino as many times as we'd like and we don't have to use all of them. Let's do a decidable example and an undecidable example.

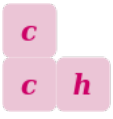
First, decidable.

Let's look at the following set of dominos:

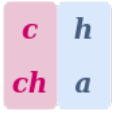
We can also represent this set of dominos as a table:

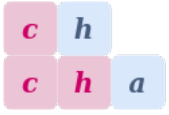
	t	b
d_1	c	ch
d_2	e	ie
d_3	h	a
d_4	rli	l
d_5	a	r

To start finding a solution, we choose domino i_1 to be a domino where the first letter of the top matches the first letter of the bottom. If these didn't match, then we'd be starting with mismatched strings that we can't recover from. In this set, we start with d_1 since t_1 and b_1 both start with c.

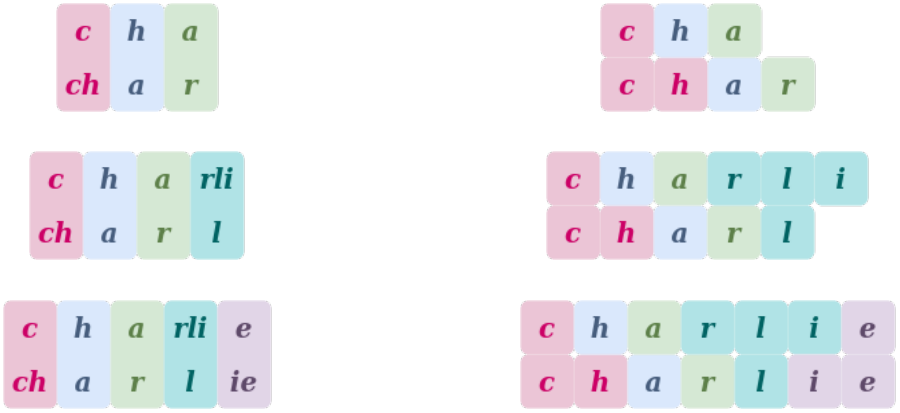


Notice how the bottom provided an extra letter. It gave an h that is currently unmatched. When we choose our next domino i_2, we need to choose one where the top starts with h, otherwise we'd lose the matching.





The process then repeats until we find a complete matching.



We have a match which means we've found our final domino seqence!

$$i_1, i_2, i_3, i_4, i_5 = d_1, d_3, d_5, d_4, d_2$$

Now an undecidable example. We start with the following set of dominos:

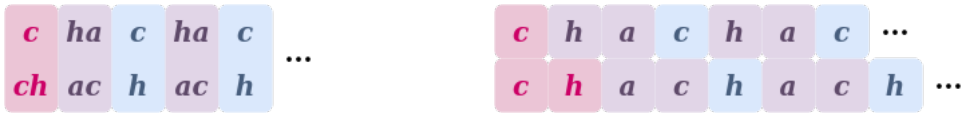
	t	b
d_1	c	ch
d_2	ha	ac
d_3	c	h

We start exactly the same way we did before. We choose the domino where the top and the bottom start with the same letter:

And we continue the same, choose dominos that fulfill the matching:

Now we need to reuse a domino. As stated before, we can do this no problem.

Again, we duplicate dominos to fulfill the matching and continue the process.



It's easy to see that if we continue choosing dominos, we'll get stuck in an infinite loop that can't be closed. Thus, this problem is undecidable.

8.6 More Undecidable Problems

Just like with the Decidable problems section, these were listed on the slides and they feel important to keep around.

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

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

Empty Turing Machine : $E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \varnothing\}$

Regular Turing Machine : $REGULAR_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular }\}$

Equal Turing Machines : $EQ_{TM} = \{\langle M, N \rangle \mid M \text{ and } N \text{ are TMs and } L(M) = L(N)\}$

Modified Post Correspondence Problem : $MPCP = \{\langle P \rangle \mid P$ is an instance of the Post Correspondenc Problem with a match that starts with the first domino $\}$

8.7 Rice's Theorem

Rice's Theorem proves that asking about the properties of a language of a Turing Machine is undeciable.

It has two main parts:

1. The property must be nontrivial, so it applies to some Turing machines, but not all
2. The property is a property of the Turing machines language

For example the following language is undecidable:

$$REGULAR_{TM} = \{\langle M \rangle \mid \ M \text{ is a Turing machine and } L(M) \text{ is regular}\}$$

Using Rice's Theorem, we first say $P = REGULAR_{TM}$. P is a language that describes a nontrivial property of a Turing machines language. Then using the two parts of the theorem:

1. P is nontrivial
   - Machine M_1 recognizes the language $L_1 = \{\Sigma^*\}$. This language is regular
   - Machine M_2 recognizes the language $L_2 = \{0^n 1^n \mid n \geq 0\}$. This language is context-free. Not regular
2. P is a property of the Turing machine's language
   - P takes a Turing machine M in as input, then accepts or rejects regarding M's language

8.8 Reducability

Computable Function : $f : \Sigma^* \to \Sigma^*$ is a computable function if some Turing machine M, for every input w, halts with just $f(w)$ on its tape

Mapping Reducibility : $A \leq_M B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every input w:

$$w \in A \iff f(w) \in B$$

Reduction : The function f is the reduction of A to B

The above is a very formal definition of reductions which makes brains hurt. Thankfully, in the Other Resources section, there are some great videos about reductions and what they are. I'm gonna primarily use one example consistently here.

Let's say that I want a pet dragon. This means my current problem is to find a dragon. However, this is a pretty hard problem to solve since dragons are really hard to find. Instead, I'll change my problem up a bit. Instead I'll say that I'm going to look for a dragon pet store. I have reduced my original problem into a new problem. If I can solve the issue of finding a dragon pet store, I can solve the issue of finding a dragon.

Let's say that

A = Finding a dragon
B = Finding a dragon pet store

We then get the following properties:

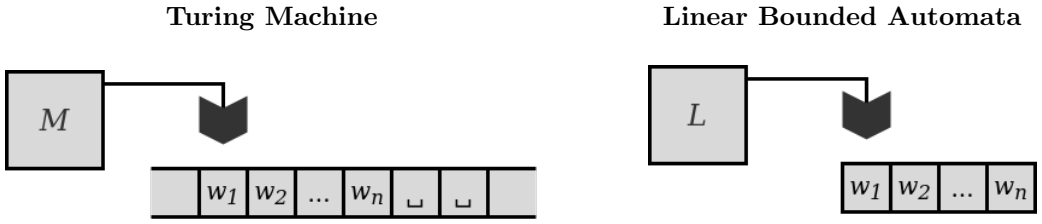
- If I can find a dragon pet store, I can find a dragon
  - If B is decidable, then A is decidable
  - If B is Turing-recognizable, then A is Turing-recognizable
  - If B is co-Turing-recognizable, then A is co-Turing-recognizable
- If dragons don't exist, I won't be able to find a dragon pet store
  - If A is undecidable, then B is undecidable
  - If A is not Turing-recognizable, then B is not Turing-recognizable
  - If A is not co-Turing-recognizable, then B is not co-Turing-recognizable

We **cannot** say that "If B is undecidable then A is undecidable" because dragons can still exist even if dragon pet stores do not.

In the same vein, we **cannot** say that "If A is decidable then B is decidable" because the existence of dragons does not imply the existence of dragon pet stores.

8.9 Linear Bounded Automatas

A linear bounded automata is a Turing machine with a restricted tape. The tape only contains the input, has a left and right end marker, and the head cannot move past either marker.



The LBA accepting language is decidable! Since there is a finite amount of tape, a finite number of states, and a finite alphabet there is a finite number of configurations. This means we can detect infinite loops.

To be exact, an LBA with q states, g symbols in the tape alphabet, and an input length of n, there are exactly qng^n possible configurations.

Accept LBA : $A_{LBA} = \{\langle M, w\rangle \mid M \text{ is an LBA and } M \text{ accepts } w\}$

L = "On input $\langle M, w\rangle$

1. Simulate M on input w for qng^n steps or until it halts
2. If M halts in an accepting state, *accept*. If it halts in a rejecting state, *reject*. If it does not halt, *reject* "

Sadly, the emptyness language is still undecidable.

Empty LBA : $E_{LBA} = \{\langle M, w\rangle \mid M \text{ is an LBA and } L(M) = \varnothing\}$

Proving this is a simple reduction $A_{TM} \leq_M E_{LBA}$. First, assume that R is a Turing machine that decides E_{LBA}. We can construct S to decide A_{TM}:

S = "On input $\langle M, w\rangle$

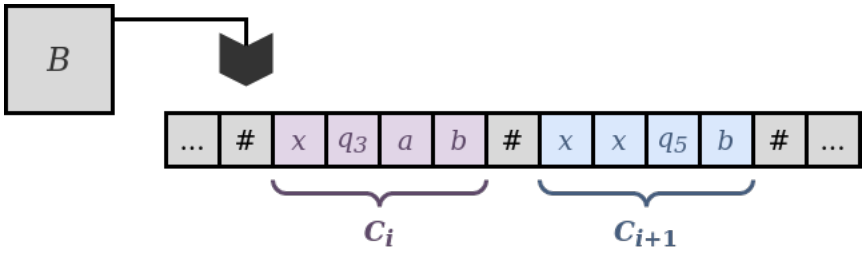
1. Construct LBA B from M and w
2. Run R on input $\langle B\rangle$
3. If R rejects, *accept*. If R accepts, *reject* "

What is B though? We construct B to be an LBA that takes an input x. It will accept x if it is an accepting configuration history for w on M. Remember that, an accepting computation history for a Turing machine M on an input string w is a sequence of configurations $C_1, C_2, \ldots, C_l$ such that:

- C_1 is the start configuration for M on w
- C_l iis an accepting configuration of M
- Each C_{i+1} is a legal move from C_i according to the δ of M

B will be an LBA that checks the validity of the above. If M accepts w, then $L(B)$ will be only a string of the accepting configuration history and $|L(B)| = 1$. If M does not accept w, then $L(B) = \varnothing$.

B will look something like this *(slightly modified Figure 5.12 from Sipser)*:



It's important to note B isn't ever run. It's just constructed for R (which we assume already exists) to see if it's language is empty or not.

9 P and NP

Ah yes. Finally, we've made it to the biggest meme in computing.

9.1 Asymptotic Notation

We can describe runtime with different types of asymptotic bounds:

9.1.1 Table of Formal Definitions

To fully prove the formal definitions below, positive constants c (or in the case of Θ, c_1 and c_2) and n_0 must be found such that the inequality holds $\forall n > n_0$.

Name	**Symbol**	**Informal Definition**	**Formal Definition** $f(n) \in X(g(n))$
Little Omega	ω	Lower Bound	$f(n) > cg(n)$
Big Omega	Ω	Tight Lower Bound	$f(n) \geq cg(n)$
Big Theta	Θ	Both Upper and Lower Bound	$c_1 g(n) \leq f(n) \leq c_2 g(n)$
Big Oh	O	Tight Upper Bound	$f(n) \leq cg(n)$
Little Oh	o	Upper Bound	$f(n) < cg(n)$

9.1.2 Table of Limit Definitions

Name	**Symbol**	**Proving with Limits**
Little Omega	ω	$\lim_{n\to\infty} f(n)/g(n) = \infty$
Big Omega	Ω	$\lim_{n\to\infty} f(n)/g(n) \neq 0$
Big Theta	Θ	$\lim_{n\to\infty} f(n)/g(n) \neq 0, \infty$
Big Oh	O	$\lim_{n\to\infty} f(n)/g(n) \neq \infty$
Little Oh	o	$\lim_{n\to\infty} f(n)/g(n) = 0$

9.1.3 Properties

Transitivity :

- $f(n) \in O(g(n))$ and $g(n) \in O(h(n)) : f(n) \in O(h(n))$
- $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n)) : f(n) \in \Omega(h(n))$

Reflexivity : $f(n) \in \Theta(f(n))$

Transpose Symmetry : $f(n) \in O(g(n) \iff g(n) \in \Omega(f(n))$

Symmetry : $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$

Trichotomy : For any two real numbers a and b: $a > b$ or $a = b$ or $a < b$

9.2 Time Complexity in Turing Machines

Given a deterministic TM M that halts on all inputs and a function $f : \mathbb{N} \to \mathbb{N}$, to say "$M$ runs in time $f(n)$" is to say $f(n)$ is the maximum number of steps that M uses on any input of length n.

Given a nondeterministic TM N that halts on all inputs and a function $f : \mathbb{N} \to \mathbb{N}$, to say "$N$ runs in time $f(n)$" is to say $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.

Theorem 7.8 : Let $t(n)$ be a function where $t(n) \geq n$. Every $t(n)$ time multi-tape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine

Theorem 7.11 : Let $t(n)$ be a function where $t(n) \geq n$. Every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine

9.3 Polynomial Class

Polynomial Bounds : $n^c, c > 0$

Exponential Bounds : $2^{n^\delta}, \delta > 0$

The Polynomial Class (P) is the class of problems that can be solved within polynomial bounds. These are the class of problems that are relaistically solvable on a computer.

The formal definition of the polynomial class is:

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

Where $TIME(t(n))$ are all languages decided by $O(t(n))$ Turing machines and $t : \mathbb{N} \to \mathbb{R}^+$

It's also important to note that P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine. We can see this below where we define three Deciders M, N, and O to decide the language $\{0^n 1^n \mid n \geq 0\}$

M = "On input w

1. Scan across the tape and *reject*is a 0 is found to the right of a 1
2. Repeat if both 0s and 1s remain on the tape
   - Scan across the tape, crossing off a single 0 and a single 1
3. If 0s still remain after all the 1s have been crossed off or 1s still remain after all 0s have been crossed off, *reject*
4. If no 0s or 1s remain on the tape, *accept* "

N = "On input w

1. Scan across the tape and *reject*is a 0 is found to the right of a 1
2. Repeat if both 0s and 1s remain on the tape
   (a) Scan across the tape, checking whether the total number of 0s and 1s is even or odd. If odd, *reject*
   (b) Scan across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1
3. If no 0s or 1s remain on the tape, *accept*. Otherwise, *reject* "

O = "On input w

1. Scan across the tape and *reject*is a 0 is found to the right of a 1
2. Scan across the 0s on tape I until the first 1. While scanning, copy the 0s to tape II
3. Scan across the 1s on tape I until the end of input. For each 1 read on tape I, cross off a 0 on tape II
4. If all 0s are crossed off before all 1s are read, or if any 0s remain after the input has been read, *reject*
5. If all 0s have been crossed off and all input is read, *accept* "

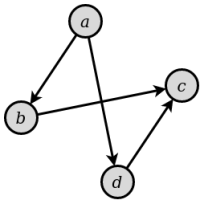
9.3.1 A Basic Path

As seen before in every data structures and algorithm course, we have algorithms like Dijkstra that can compute shortest path in P time. This is just a basic "Does a path exist?" algorithm. It's better described by the language:

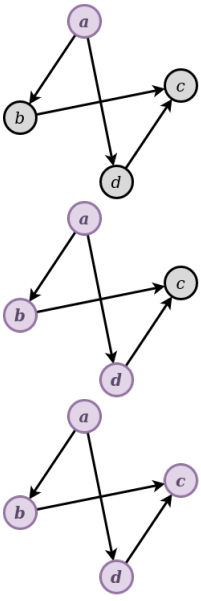
$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$$

M = "On input $\langle G, s, t\rangle$

1. Place a mark on node s
2. Repeat until no additional nodes are marked
   - Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b
3. If t is marked, *accept*. Otherwise, *reject* "

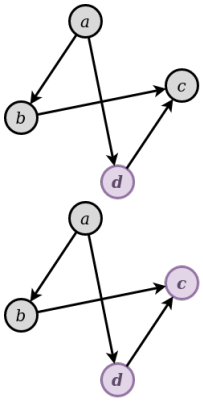


$\langle G, a, d\rangle$



accept

$\langle G, d, a\rangle$



reject

9.3.2 Relative Primeness

Relative Primes : Values x and y are relatively prime if they share no common factors. For example, niether 8 nor 9 are prime, but they are both relatively prime

M = "On input $\langle x, y \rangle$

1. Repeat until $y = 0$
   (a) Assign $x \leftarrow x \mod y$
   (b) Exchange x and y
2. If $x = 1$, *accept*. Otherwise, *reject* "

$\langle 8, 9 \rangle$

x	y	
8	9	
8	9	Mod
9	8	Exchange
1	8	Mod
8	1	Exchange
0	1	Mod
1	0	Exchange

accept

$\langle 8, 10 \rangle$

x	y	
8	10	
8	10	Mod
10	8	Exchange
2	8	Mod
8	2	Exchange
0	2	Mod
2	0	Exchange

reject

9.4 NP Class

NP is the class of languages that have polynomial time verifiers

Verifier : A verfier for a language L is an algorithm V where:

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

Polynominal Time Verifier : A verifier that runs in polynomial time in the length of w

Polynomial Verifiable : A language L has a polynomial time verifier

Theorem 7.20 : A language is NP if and only if it is decided by some nondeterministic polynomial time Turing machine

We can define the nondeterministic time complexity class similarly to how we defined $TIME(t(n))$ for deterministic time complexity:

$$TIME(t(n)) = \{L \mid L \text{ is a language decided by } O(t(n)) \text{ deterministic Turing machines}\}$$

$$NTIME(t(n)) = \{L \mid L \text{ is a language decided by } O(t(n)) \text{ nondeterministic Turing machines}\}$$

Which then again, similarly to how we defined P, we can do NP:

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

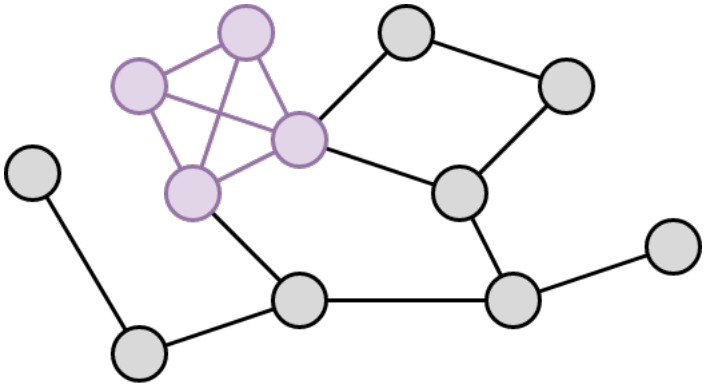
$$\text{NP} = \bigcup_k NTIME(n^k)$$

9.4.1 CLIQUE

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

***k*-Clique :** A subset C of k vertices in a graph G such that every $v_i, v_j \in C$ has an edge between them

The following graph has a 4-clique (highlighted in purple):



Theorem 7.24 : $CLIQUE$ is NP

We can build a verifier V to prove this where certificate c is the clique:

V = "On input $\langle\langle G, k\rangle, c\rangle$

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both pass, *accept*. Otherwise, *reject* "

Or we can build an NTM:

N = "On input $\langle G, k\rangle$

1. Nondeterministically select a subset c of k nodes of G
2. Test whether G contains all edges connecting nodes in c
3. If yes, *accept*. Otherwise, *reject* "

9.5 NP-Completeness and Reductions

NP-Complete : A language B is NP-Complete if the following two conditions are met:

1. B is NP
2. Every $A \in$ NP is polynomial time reducible to B

Going back to our dragon metaphor, we're going to now cover polynomial time reductions.

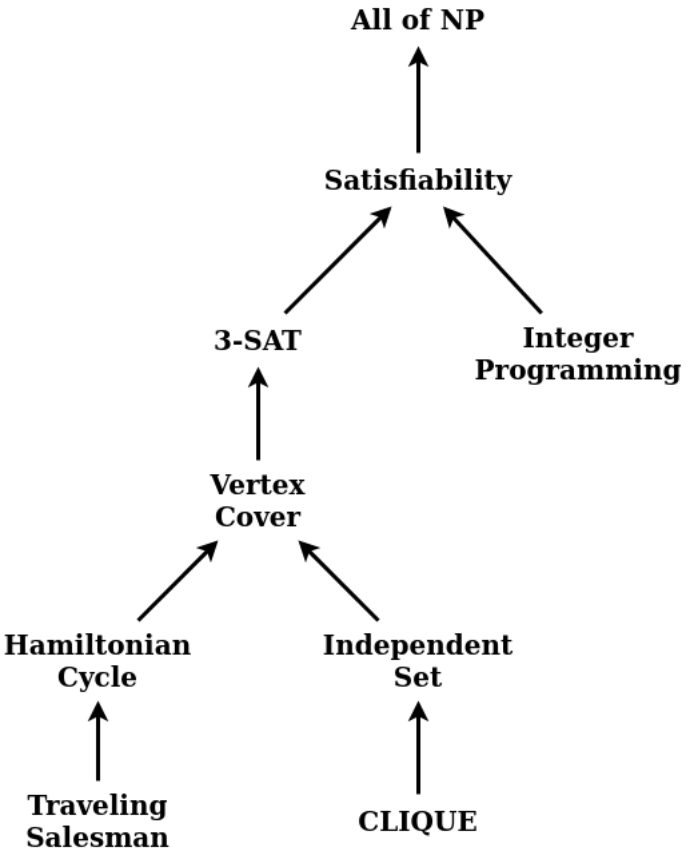
We say $A \leq_P B$ or "A is reducible to B in polynomial time" if we can reduce A into B and the time it takes to perform the reduction takes polynomial time.

Theorem 7.31 : If $A \leq_P B$ and $B \in$ P, then $A \in P$. (If we can find a dragon pet store, we can find a dragon)

Theorem 7.35 : If B is NP-Complete and $B \in$ P, then P = NP. (If we can find a "fantasy pet store", we can find not just dragons, but also *every* cool pet)

Theorem 7.36 : If B is NP-Complete and C is NP and $B \leq_P C$, then C is NP-Complete

I love this image of the reduction tree for NP-Complete problems from Skiena's *The Algorithm Design Manual*. I removed some of the irrelevant reductions (and also reductions I don't know).



9.5.1 Satisfiability

Satisfiability : Given a set of boolean variables V and a set of clauses C over V, does there exist a satisfying truth assignment for C such that each clause C_i contains at least one true variable?

$$V = \{v_1, v_2, v_3\}$$
$$C = \{\{v_1, \overline{v_2}\}, \{v_2\}, \{v_2, v_3\}\}$$

			$\boldsymbol{C_1}$	$\boldsymbol{C_2}$	$\boldsymbol{C_3}$
$\boldsymbol{v_1}$	$\boldsymbol{v_2}$	$\boldsymbol{v_3}$	$\{v_1, \overline{v_2}\}$	$\{v_2\}$	$\{v_2, v_3\}$
0	0	0	$\{0, 1\}$	$\{0\}$	$\{0, 0\}$
0	0	1	$\{0, 1\}$	$\{0\}$	$\{0, 1\}$
0	1	0	$\{0, 0\}$	$\{1\}$	$\{1, 0\}$
0	1	1	$\{0, 0\}$	$\{1\}$	$\{1, 1\}$
1	0	0	$\{1, 1\}$	$\{0\}$	$\{0, 0\}$
1	0	1	$\{1, 1\}$	$\{0\}$	$\{0, 1\}$
1	1	0	$\{1, 0\}$	$\{1\}$	$\{1, 0\}$
1	1	1	$\{1, 0\}$	$\{1\}$	$\{1, 1\}$

Solutions:

$$\{v_1 = 1, v_2 = 1, v_3 = 0\}$$
$$\{v_1 = 1, v_2 = 1, v_3 = 1\}$$

$$V = \{v_1, v_2\}$$
$$C = \{\{v_1, v_2\}, \{v_1, \overline{v_2}\}\}$$

		$\boldsymbol{C_1}$	$\boldsymbol{C_2}$
$\boldsymbol{v_1}$	$\boldsymbol{v_2}$	$\{v_1, v_2\}$	$\{\overline{v_1}, v_2\}$
0	0	$\{0, 0\}$	$\{1, 0\}$
0	1	$\{0, 1\}$	$\{1, 1\}$
1	0	$\{1, 0\}$	$\{0, 0\}$
1	1	$\{1, 1\}$	$\{0, 1\}$

Solutions:

$$\{v_1 = 0, v_2 = 1\}$$
$$\{v_1 = 1, v_2 = 1\}$$

$$V = \{v_1, v_2\}$$
$$C = \{\{v_1, \overline{v_2}\}, \{v_1, \overline{v_2}\}, \{\overline{v_1}\}\}$$

		$\boldsymbol{C_1}$	$\boldsymbol{C_2}$	$\boldsymbol{C_3}$
$\boldsymbol{v_1}$	$\boldsymbol{v_2}$	$\{v_1, v_2\}$	$\{\overline{v_1}, v_2\}$	$\{\overline{v_1}\}$
0	0	$\{0, 0\}$	$\{1, 0\}$	$\{1\}$
0	1	$\{0, 1\}$	$\{1, 1\}$	$\{1\}$
1	0	$\{1, 0\}$	$\{0, 0\}$	$\{0\}$
1	1	$\{1, 1\}$	$\{0, 1\}$	$\{0\}$

No Solutions

9.5.2 3-SAT

3-Satisfiability (3-SAT) : A special case of Satisfiability where each clause has exactly 3 literals

How can we reduce a general Satisfiability problem to a 3-SAT problem? Lets start with the following input:

$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4, v_5\} \\ C_1 &= \{v_2, \overline{v_3}\} \\ C_2 &= \{v_1\} \\ C_3 &= \{\overline{v_1}, v_2, v_4\} \\ C_4 &= \{v_1, v_2, \overline{v_3}, v_4, v_5\} \end{aligned}$$

For each clause C_i with k literals $\{z_1, \ldots, z_k\}$, we apply the following rules:

$\boldsymbol{k = 1}$: Create two new variables, u_1 and u_2 and four new clauses:

$$\begin{aligned} &\{z_1, u_1, u_2\} \\ &\{z_1, u_1, \overline{u_2}\} \\ &\{z_1, \overline{u_1}, u_2\} \\ &\{z_1, \overline{u_1}, \overline{u_2}\} \end{aligned}$$

Note that even though we're adding variables and clauses, the only variable that affects the solution is z_1. The new clauses will only all be true when $z = 1$.

$\boldsymbol{k = 2}$:Create one new variable, u_1, and two new clauses:

$$\begin{aligned} &\{z_1, z_2, u_1\} \\ &\{z_1, z_2, \overline{u_1}\} \end{aligned}$$

Same as before, even though we're adding to the problem, u_1 does not affect the solution.

$\boldsymbol{k = 3}$

No change. Copy the clause over to the 3-SAT problem.

$\boldsymbol{k > 3}$: Create $k - 3$ new variables and $k - 2$ new clauses in a chain:

$$\begin{aligned} C_{i,1} &= \{z_1, z_2, \overline{u_{i,1}}\} \\ C_{i,j} &= \{u_{i,j-1}, z_{j+1}, \overline{u_{i,j}}\} \qquad\qquad 2 \le j \le k-3 \\ C_{i,k-2} &= \{u_{i,k-3}, z_{k-1}, z_k\} \end{aligned}$$

Now to apply this to our original problem:

Satisfiability	**3-SAT**
$V = \{v_1, v_2, v_3, v_4, v_5\}$	$V = \{v_1, v_2, v_3, v_4, v_5, u_1, u_2, u_{4,1}, u_{4,2}\}$
$C_1 = \{v_2, \overline{v_3}\}$	$C_{1,1} = \{v_2, \overline{v_3}, u_1\}$
$C_2 = \{v_1\}$	$C_{1,2} = \{v_2, \overline{v_3}, \overline{u_1}\}$
$C_3 = \{\overline{v_1}, v_4\}$	$C_{2,1} = \{v_1, u_1, u_2\}$
$C_4 = \{v_1, v_2, \overline{v_3}, v_4, v_5\}$	$C_{2,2} = \{v_1, u_1, \overline{u_2}\}$
	$C_{2,3} = \{v_1, \overline{u_1}, u_2\}$
	$C_{2,4} = \{v_1, \overline{u_1}, \overline{u_2}\}$
	$C_3 = \{\overline{v_1}, v_2, v_4\}$
	$C_{4,1} = \{v_1, v_2, \overline{u_{4,1}}\}$
	$C_{4,2} = \{u_{4,1}, \overline{v_3}, \overline{u_{4,2}}\}$
	$C_{4,3} = \{\overline{u_{4,2}}, v_4, v_5\}$

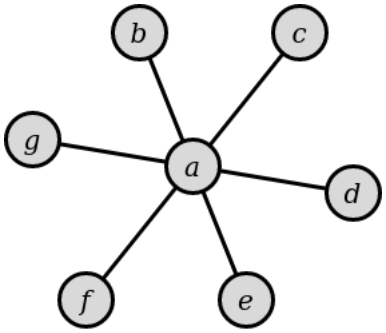
This proves:

$$\textsc{Satisfiability} \leq_P \text{3SAT}$$

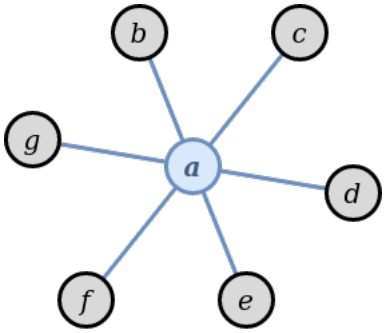
9.5.3 Vertex Cover

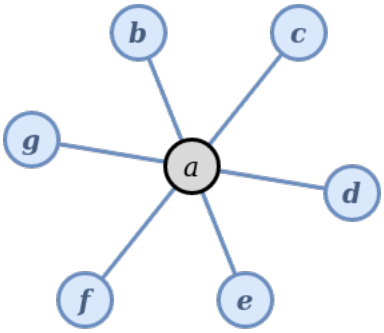
Vertex Cover : A vertex cover of a graph $G(V, E)$ is a subset $C \subseteq V$ such that every vertex $v \in V$ has an edge connecting to a $c \in C$

The following graph has two possible vertex covers:



$C = \{a\}$ $C = \{b, c, d, e, f, g\}$

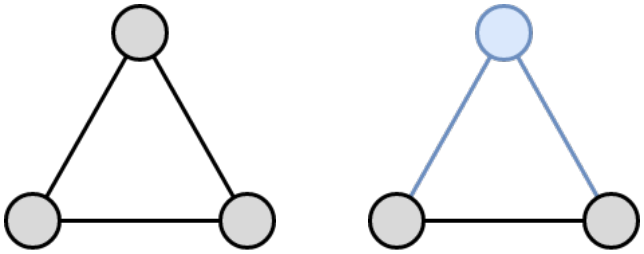




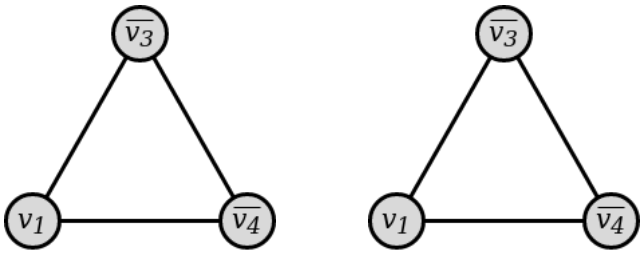
As seen in the tree before, we can reduce 3-SAT to Vertex Cover in polynomial time. Let's start with the following 3-SAT problem (Figure 9.7 in Skiena):

$$V = \{v_1, v_2, v_3, v_4\}$$
$$C = \{\{v_1, \overline{v_3}, \overline{v_4}\},$$
$$\{\overline{v_1}, v_2, \overline{v_4}\}\}$$

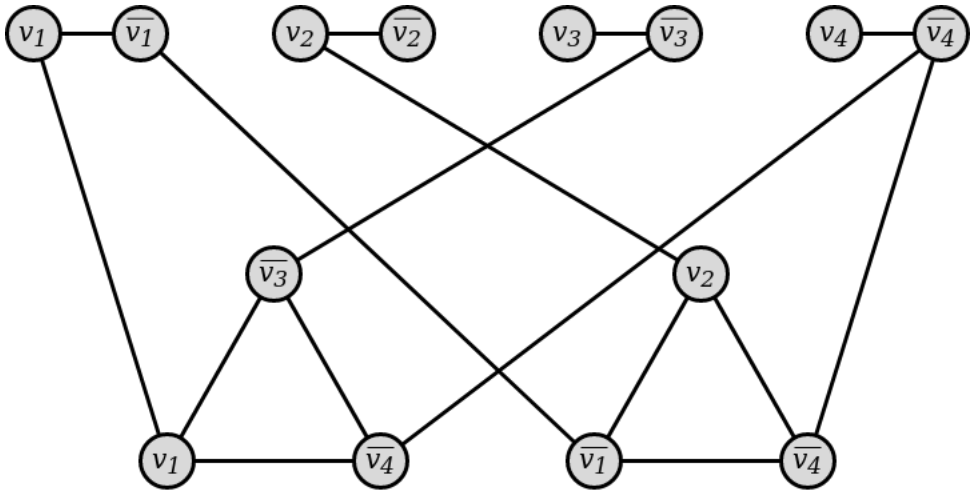
We can build triangles for each clause. If one vertex in a triangle is selected, then we have a vertex cover. If we consider the selected vertex to be "true", then we also have satified the clause.



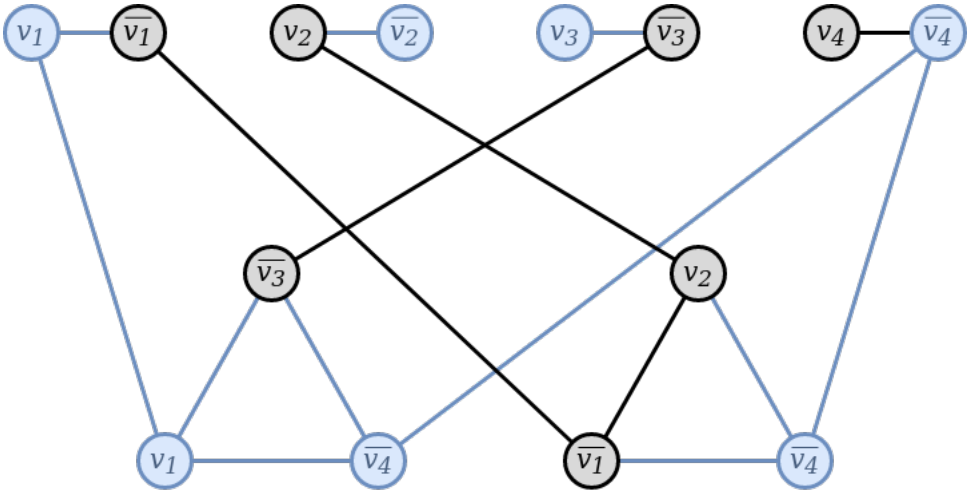
Then actually build the triangles for the clauses:



To actually check for a satifiable solution, we add a row of all possible variables to the top:



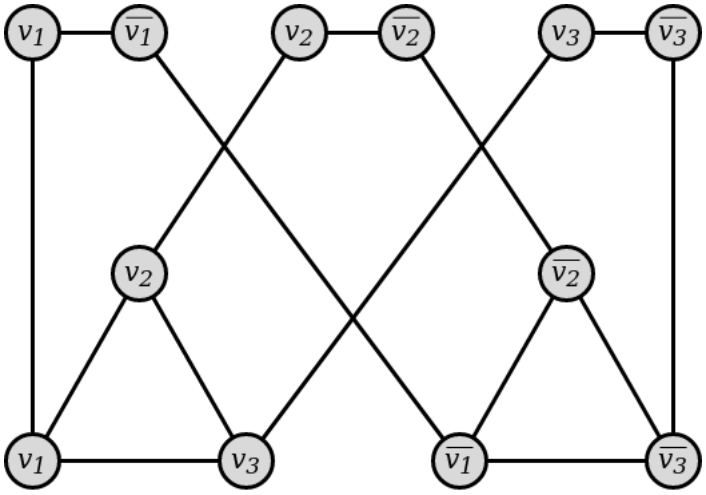
For the solution to 3-SAT $\{v_1 = 1, v_2 = 0, v_3 = 1, v_4 = 0\}$ we get the following vertex cover:



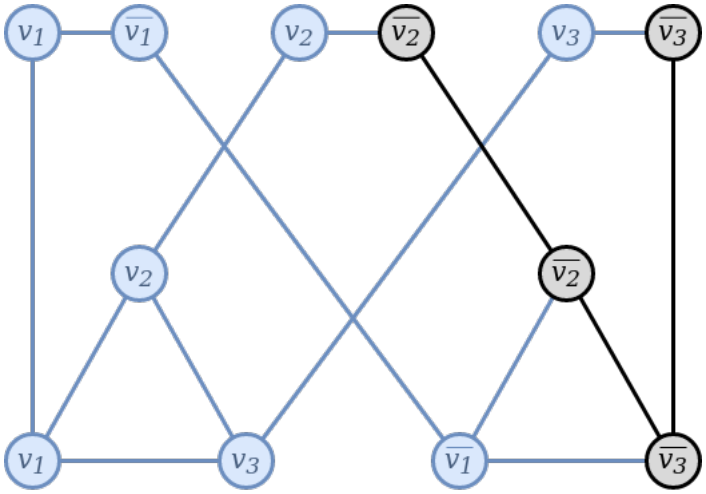
Given an unsatsifiable problem:

$$V = \{v_1, v_2, v_3\}$$
$$C = \{\{v_1, v_2, v_3\},$$
$$\{\overline{v_1}, \overline{v_2}, \overline{v_3}\}\}$$

We get the following graph:



Which has the following vertex cover:



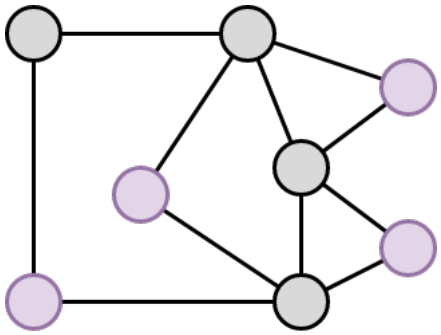
Which claims that v_1 and $\overline{v_1}$ must be both true to satisfy the clauses. This isn't possible so this 3-SAT problem cannot be satified. We have now proven:

$$\text{Satisfiability} \leq_P \text{3SAT} \leq_P \text{VertexCover}$$

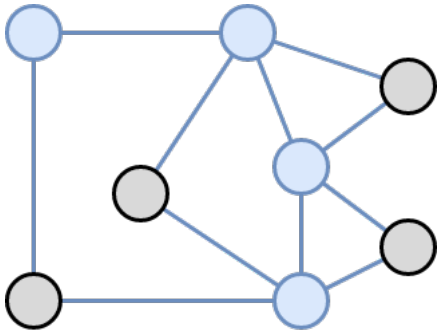
9.5.4 Independent Set

Independent Set : In a Graph $G(V, E)$ does there exist a subset C of V such that for every $v_i, v_j \in C$ that edge $(v_i, v_j) \notin E$

The below graph (modified Figure 9.4 from Skiena) has an independent set highlighted in purple:



Now we may notice that if we invert the graph, we have a vertex cover (now highlighted in blue):



And that's the whole reduction! Now we have:

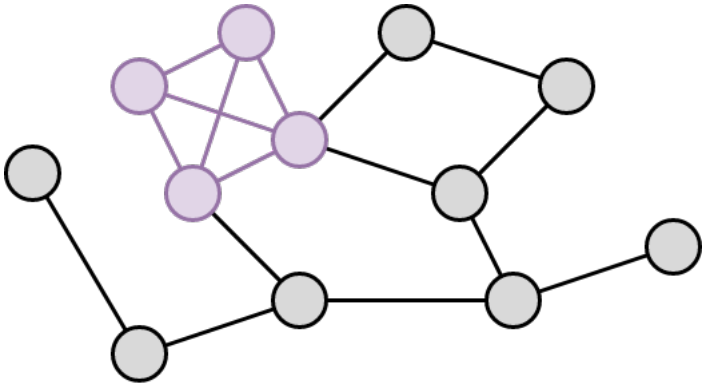
$$\text{Satisfiability} \leq_P \text{3SAT} \leq_P \text{VertexCover} \leq_P \text{IndependentSet}$$

9.5.5 CLIQUE

We saw CLIQUE earlier, but let's review.

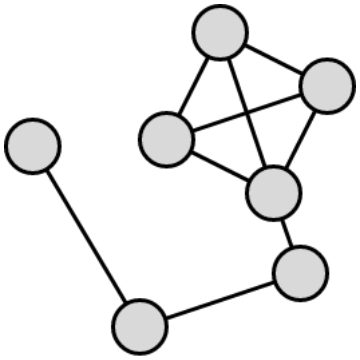
***k*-Clique :** A subset C of k vertices in a graph G such that every $v_i, v_j \in C$ has an edge between them

The following graph has a 4-clique (highlighted in purple):

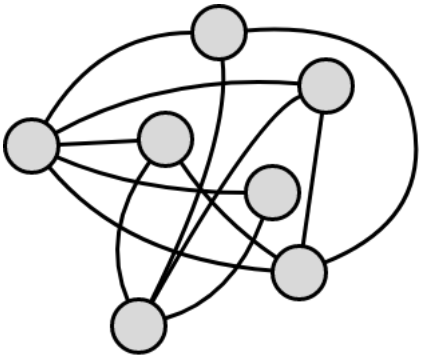


Now that we know Independent Sets, we can realize that "a set with no edges between vertices", is the opposite of "a set where every edge exists between vertices". To solve CLIQUE, we take the complement of the graph, then find the independent set.

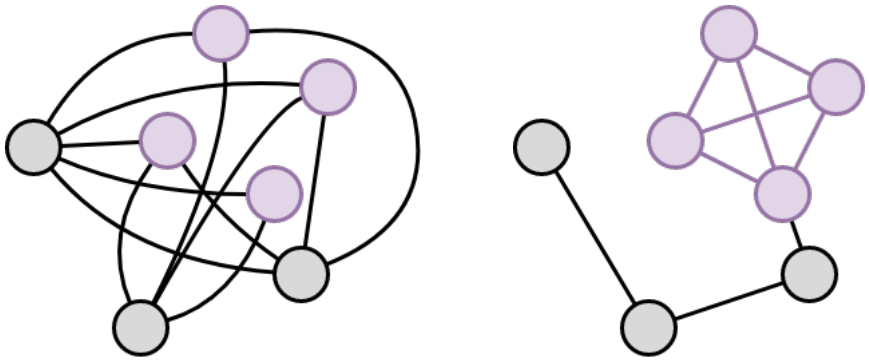
Let's start with a smaller version of the above graph (the complement got ugly real fast):



It's (ugly) complement:



Then finding the independent set of the complement also gives the CLIQUE of the original graph:



And now we get the following chain:

$$\text{Satisfiability} \leq_P \text{3SAT} \leq_P \text{VertexCover} \leq_P \text{IndependentSet} \leq_P \text{CLIQUE}$$

9.5.6 Hamiltonian Cycle

Hamiltonian Cycle : Given a graph G, does there exist a simple tour that visits every vertex in V without repetition?

The following graph has a Hamiltonian Cycle highlighted in pink: