Notes on Computability, Complexity, and Algorithms – An Ode to the Unprepared Student

Brandon A. Durepo, bdurepo@gatech.edu

"Sapere aude." – Horace

Contents

Preface

The purpose of this document is to help students of all stripes succeed in Georgia Tech's course on Computability, Complexity and Algorithms more formally known as CS6505. Many students find this course to be challenging and the topics can be opaque at times to the newcomer. However with grit, determination, and luck you, the reader, can reap the many benefits that this course offers. In proceeding sections I outline some of the key concepts you will need to be successful in the course. I make no assumptions about prerequisite knowledge and attempt to provide you the green field perspective on the subject matter. In many cases, if you have already seen the information it is safe to skip forward and not concern yourself what may be obvious to you, but I encourage you to read all the sections because additional clarity may be gained with another pass through.

Part I
Mathematics and Logic

1 Discrete Mathmatics

1.1 Propositional and Sentential Logic

All logical expressions can be described in some combination of *logical operators* and *implications*. The logical operators consist of the following: the logical conjunction $\wedge$, the logical disjunction $\vee$, and the negation $\neg$. The implication is made up of two propositional variables $p$and q. $p$is often called the *premise* or *hypothesis* or *antecedent*, while $q$may be referred to as the *conclusion*, or *consequence*. Truth tables are a way of tabulating the values of arbitrary logical expressions by enumerating all possible truth values for the componenet of the expressions. The usefulness of this technique quickly dimminishes as the number of proposition variables goes beyond 4. In this case the number of rows in the truth table is $2^4 = 16$. The precedence of logical operators is negation, conjunction, disjunction.

The implication is given by the symbol $\rightarrow$. Variations on the traditional implication of $p \rightarrow q$ include its *converse* $q \rightarrow p$, the *contraposative* $\neg q \rightarrow \neg p$, and the *inverse* $\neg p \rightarrow \neg q$. A biconditional or bimplicational statement is a slightly more creative way to use the implication. A table of logical equivalances is provided in Figure 1 on the following page.

Use of these laws can allow you to estabish an equivalance of complex propositional statements with a more distilled simplified version. Propositional statements are said to be *satisfiable* if some combination of truth assignments to the propostional variables makes the statement true.

Equivalence	Name
$p \wedge T \equiv p$, $p \vee F \equiv F$	Identity Laws
$p \vee T \equiv T$, $p \wedge F \equiv F$	Domination Laws
$p \vee p \equiv p$, $p \wedge p \equiv p$	Idempotent Laws
$\neg(\neg p) \equiv p$	Double Negation Laws
$p \vee q \equiv q \vee p$, $p \wedge q \equiv q \wedge p$	Communatative Laws
$(p \vee q) \vee r \equiv p \vee (q \vee r)$, $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	Associative Laws
$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$, $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	Distributive Laws
$\neg(p \wedge q) \equiv \neg p \vee \neg q$, $\neg(p \vee q) \equiv \neg p \wedge \neg q$	De Morgan's Law
$p \vee (p \wedge q) \equiv p$, $p \wedge (p \vee q) \equiv p$	Absoption Law
$p \vee \neg p \equiv T$, $p \wedge \neg p \equiv F$	Negation Laws

Logical Equivalences Involving Conditional Statements
$p \to q \equiv \neg p \vee q \equiv \neg q \to \neg p$
$p \vee q \equiv \neg p \to q$
$p \wedge q \equiv \neg(p \to \neg q)$
$\neg(p \to q) \equiv p \wedge \neg q$
$(p \to q) \wedge (p \to r) \equiv p \to (q \wedge r)$
$(p \to r) \wedge (q \to r) \equiv (p \vee q) \to r$
$(p \to q) \vee (p \to r) \equiv p \to (q \vee r)$
$(p \to r) \vee (q \to r) \equiv (p \wedge q) \to r$

Logical Equivalences Involving Biconditional Statements
$p \leftrightarrow q \equiv (p \to q) \wedge (q \to p)$
$p \leftrightarrow q \equiv \neg p \leftrightarrow \neg q$
$p \leftrightarrow q \equiv (p \wedge q) \vee (\neg p \wedge \neg q) \vee (\neg p \wedge \neg q)$
$\neg(p \leftrightarrow q) \equiv p \leftrightarrow \neg q$

Figure 1: Logical equivalences. Idemepotent denotes an element of a set that is unchanged in value when multiplied or otherwise operated on by itself.

$\neg\forall x P(x) \equiv \exists x \neg P(x).$
$\neg\exists x P(x) \equiv \forall \neg P(\mathrm{x}).$

Figure 2: This table shows the equivalance relations on predicates with quantification symbols.

1.2 Predicate Calculus and Quantification Logic

Predicate logic is a generalization of propositional logic onto arbitrary statements. *Propositonal functions* given by a capital letter and a number of parameters are used to define propositional statements. An example, "Computer x, is functioning properly" would be written as $P(x)$.

Quantification expresses the extent to which a predicate is true over a range of elements. In English, the words all, some, many, none, and few are used in quantifications. The *univeral quantification* symbol for all is given by $\forall$. The *existential quantifiaction* symbol for exists is $\exists$. The statement $\forall x P(x)$ reads "For all computers given by x, computer $x$is functioning properly." If there is a single instance of a universal varaible which is false then the predicate becomes false. If for all instances of an existential varaibles the value is false then the statement is false. Quantification operators have higher precedence than logical operators. Predicates used with quantifiers are said to be *bound variables* while those without are said to be *free variables*.

1.3 Rules of Inference

The *validity* of a proofs arguments are derived from the truth of the premises it offers. An argument in propositional logic is a sequence of propositions. All but the final proposition in the argument are called premises and the final proposition is called the conclusion.An argument is valid if the truth of all its premises implies that the conclusion is true. An argument form in propositional logic is a sequence of compound propositions involving propositional variables. An argument form is valid no matter which particular propositions are substituted for the propositional variables in its premises, the conclusion is true if the premises are all true. The rules of inference provide the templates of logical propostions which when given in a valid combitation yield valid arguments. The rules are given in Figure 3 on the next page. Additonally the Rules of Inference applies to quantification logic as well as shown in the table.

1.4 Proofs

In order to begin to understand how to construct valid proof it is important to understand the terminology that is used.

1.4.1 Terminology of Proofs

- a *theorem* is a statement that can be shown to be true.

Tautology	Name
$(p \wedge (p \to q)) \to q$	Modus ponens
$(\neg q \wedge (p \to q)) \to q$	Modus tollens
$((p \to q) \wedge (q \to r)) \to (p \to r)$	Hypothetical syllogism
$((p \vee q) \wedge \neg p) \to q$	Disjunctive syllogism
$p \to (p \vee q)$	Addition
$(p \wedge q) \to q$	Simplification
$((p) \wedge (q)) \to (p \wedge q)$	Conjunction
$(p \vee q) \wedge (\neg p \vee r) \to (q \vee r)$	Resolution

Inference Rule	Name
$\frac{\forall x P(x)}{\therefore P(c)}$	Universal instantiation
$\frac{P(c)}{\therefore \forall x P(x)}$ for an arbitrary c	Universal generalization
$\frac{\exists x P(x)}{\therefore P(c)}$ for some element c	Existantial instantiation
$\frac{P(c)}{\therefore \exists x P(x)}$	Existential generalization

Figure 3: This table gives the Rules of Inference.

- A *proof* is a valid argument that establishes the truth of a theorem.
- *axioms* (or *postulates*) are statements we assume to be true.
- A *lemma* is a less important theorem that is helpful in the proof of other results.
- A *corollary* is a theorem that can be established directly from a theorem that has been proved.
- *without loss of generality* (WLOG) means that the author of a proof is omitting the proof of one of the cases in a proof by cases.

1.4.2 Types of Proofs

- *Direct Proofs* is a proof that assumes the antecedent of the implication is true and seeks to justify and prove the conclusion of the theorem by examining the characteristics of the relationship of the antecedent and conclusion. A direct proof may use axioms, definitions, and previously proven theorems, together with rules of inference, to show that q must also be true. Here we are establishing the truth of the proof based on row 1 of the truth table found in Figure 4 on the following page.

- In juxtaposition to the direct proof we have the indirect proof techniques. Indirect proofs do not start with the assumption of the truth of the premise to derive the truth of the conclusion. The *Proof by Contraposition* technique takes advantage of the equivalance of the contrapostion of the implication to prove the conclusion. That is, $\neg q \to \neg p \equiv p \to q$ as shown in

p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

p	q	$\neg q \to \neg p$
T	T	T
T	F	F
F	T	T
F	F	T

Figure 4: Truth table for the implication and its contraposition.

Figure 1 on page 8. The template for this type of proof is constructed by assuming the conclusion $q$is false which implies that the premise p is also false.

- The trivial and vacuous proofs strategies are the simplest strategies. A *Trivial Proof* is a proof that derives the truth of the conclusion based on the fact that the premise is false. This logic is based on row 3 of the implication truth table. A Vacuous Proof is a proof based on the fact the conclusion is true also show in row 3.

- *Proofs by Contradiction* seek to find a contradictory conclusion $q = r \wedge \neg r$, such that $\neg p \to q$ is true. We must show that $p$is true by showing $\neg p$ is false so that we can target row 4 in the implication truth table.

- *Proofs of Equivalance* To prove a theorem that is a biconditional statement, that is, a statement of the form $p \leftrightarrow q$, we show that $p \to q$ and $q \to p$. The validity of this approach is based on the tautology $(p \leftrightarrow q) \equiv (p \to q) \wedge (q \to p)$

- *Proofs by Counterexample* in this type of proof we typically try to debunk the truth of a univeral quantifier over some predicate. If we find one instance is false, then we can conclude the predicate is false.

- *Proof by Cases and the Exhaustive Proof* sometimes using a single argument to cover the premise of a proof is not sufficient. As an alternative, we can devises cases that cover the entire domain of the proof and address each case in turn to prove the conclusion. This proof technique relies on the following tautology.

$$[(p_1 \vee p_2 \vee \ldots \vee p_n) \to q] \leftrightarrow [(p_1 \to q) \wedge (p_2 \to q) \wedge \ldots \wedge (p_n \to q)]$$

- *Existence Proofs* a proof of this type is of the form $\exists x P(x)$. This type of proof can be proven by providing a *witness* a, a instance of $x$where $P(a)$ is true. This type of strategy is called a *constructive* proof. A nonconstuctive existance uses some other means to prove that $\exists x P(x)$. Often contradition is used.

- *Uniqueness Proofs* in this type of proof we show that a unique element with a particular property exists. These types of proofs are composed of

Direct Methods	Indirect Methods
Direct Proof	Proof by Contraposition
Proof by Cases	Proof by Contradiction
Exhaustive Proof	Proof by Counterexample
Existence Proof	
Uniqueness Proof	

Figure 5: Classification of the proof strategies

two steps. First you must prove the existance of the element. Then we show that if $y \neq x$, then $y$does not have the particular property of x. Formally, uniqueness proofs are of the form:

$$\exists x(P(x) \wedge \forall y(y \neq x \rightarrow \neg P(y))).$$

1.4.3 Strategies for Constructing Proofs

The following general approach for constructing proofs is often helpful.

1. Replace terms by their definitions.
2. Analyze the hypothesis and conclusion.
3. Attempt to construct the proof using the direct proof method, if the statement is a coditional statement.
   (a) If you fail to construct a direct proof, attempt to provide an indirect proof.
   (b) If both the direct and indirect proof methods don't work, then attempt a proof by contradiction.

Forward reasoning as well as *backward reasoning* is necessary to construct valid proofs. Forward reasoning uses premises as well as previously proven theorems and axioms to prove the conclusion. In a direct proof this is the method of choice. Indirect proofs often will often require backward reasoning. In backward reasoning we attempt to find a statement $p$that we can prove true with $p \rightarrow q$.

Adapatation of existing proofs is often a fruitful procedure. Even if the final conclusions required are not the same, it may be constructive to look to the previous proves for ideas which may help in your new proof.

Arithmetic Property	Equation
Modulo Positive Numbers	$a\%b = c$
Modulo Negative Numbers	$c = a - q \cdot b$, where $a < 0$, and $-1 \leq q \leq \lfloor a/b \rfloor$
Modular Congruence	$a\%b \equiv b\%c$
Modular Addition	$(a + b)\%c \equiv (a\%c + b\%c)\%c$
Modular Subtraction	$(a - b)\%c \equiv (a\%c - b\%c)\%c$
Modular Multiplication	$(a \cdot b)\%c \equiv (a\%c \cdot b\%c)\%c$

Figure 6: This table gives an overview of useful modular arithmatic equations

1.5 Sets

1.6 Relations

1.7 Functions

1.8 Induction

1.9 Modular Arithmatic

Figure 6 gives an overview of basic modular arithmatic operations.

2 Calculus and Real Analysis

2.1 Sequences and Series

2.2 Topology of Real Numbers

2.3 Functional Limits and Continuity

2.4 Derivatives

2.5 Sequences and Series of Functions

2.6 Rieman Integral

2.7 Cantor Diagonalization

3 Probability and Statistics

3.1 Discrete Probability Distributions

3.2 Continuous Probability Distributions

3.3 Combinatorics

3.4 Condiational Probability

3.5 Expected Value and Variance

3.6 Sample Space and Probability

3.7 Discrete Random Variables

3.8 General Random Variables

3.9 Limit Theorems

4 Linear Algebra and Integer Linear Programming

4.1 Linear Equations in Linear Algebra

4.2 Matrix Algebra

4.3 Determinants

4.4 Vector Spaces

4.5 Eigenvalues and Eigenvectors

4.6 Orthogonality and Least Squares

4.7 Symmetric Matrices and Quadratic Forms

4.8 Geometry of Vector Spaces

4.9 Basic Properties of Linear Progams

4.10 The Simplex Method

4.11 Duality and the Complementarity

4.12 Interior Point Methods

which the automata exists and no previous information can be remembered by the finite automata. The limits to which the finite automata can model information are expressed by the number of states.

5.1 Notation of DFA's

Traditionally finite automata are expressed by a tuple of sets which express its behavior. The tuple, for example $L = (Q, \Sigma, \delta, q_0, F)$, contains information on the states expressed by the set Q, the input alphabet expressed by the set Σ, the transitions expressed by the function δ, the starting state given by q_0, and the set of final states given by F. An example of a finite automata can be seen in Figure13. Strings are typically represented by variables of lowercase letters at the end of the alphabet (w, x, y, and z) whereas characters are represented variables of lowercase letters at the beginning of the English alphabet (a, b, and c). Σ^* represents the set of all strings over the alphabet Σ including the empty string–the super-script star symbol is referred to as the Kleene star. The empty string is denoted by the epsilon symbol ϵ. The empty string denotes a string of length 0, a string of no characters. Length is often expressed by vertical bars surrounding the string in question, such as $|w|$. A transition function δ takes two parameters, a state and an input symbol, and returns the next state to which the automaton should transition on input of the symbol (i. e $\delta(q$,a)).

5.2 Computation of DFA's

Given a finite automata, computation can be carried out on an input string by processing each character of the input one-by-one and following the transitions based on the input character. by Upon encountering the last character in the input string computation stops and based on whether the state is a final state (accepting state) or a nonaccepting state, the string is either accepted or rejected. Accepted strings are said to be in the language of the automaton rejected strings are not in the language. Given an automaton A the language of A is denoted $L(A)$. A language is a subset of

5.3 Regular Languages

A language is a regular language if it is accepted by some deterministic finite automata (DFA). A caveat here is that the DFA must accept only those strings in the language it models and no others. Nonregular languages are those that cannot be modeled by a DFA. Examples of nonregular languages are those that require automata to count beyond a finite number of states. The classical example is the language L = $\{0^n 1^n \mid n \geq 1\}$. DFA's cannot check for the same number of symbols in a string or balanced parenthesis. This job falls to context free grammars.

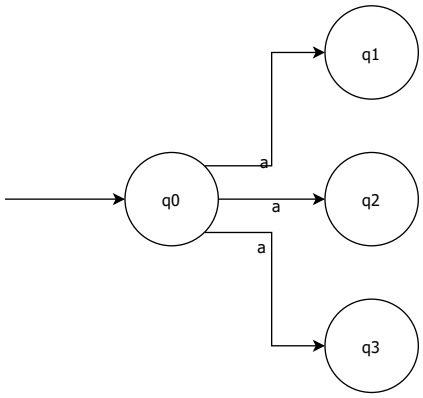


Figure 7: The NFA depicted in the diagram shows that on input symbol $a$the automaton transitions to the state q_1, q_2 and q_3 at the same time.

6 Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) differs from its deterministic counterpart in that it may be in many states at once. Transitions on an input symbol may proceed to any number states. This idea is demonstrated in Figure 7. NFA's are equivalent to DFA's and such anything that can be modeled with a NFA can be modeled with a DFA. Thus, the class of languages modeled by NFA's is the regular languages.

6.1 Proof of Equivalence of NFA's and DFA's

The proof of equivalence between DFA's and NFA's is often called the Subset Construction. In the succeeding proof given a NFA the states Q_N, inputs Σ, transition function δ_N, start state q, and final states F. The set of states of the constructed DFA $Q_D = 2^{Q_N}$, that is the power set of the set of states of the NFA. A power set of a set is defined as the set of all possible subsets of a given set. The inputs to the DFA will be the same as the inputs of the NFA, Σ. The start state of the DFA is the set containing the start state, q_0 of the NFA. The DFA states are named such that they enumerate the powerset's subsets with labels that correspond to the states of the NFA. The transition function for the DFA, δ_D is defined by $\delta_D(\{q_1,q_2,\ldots q_k\},a)$ is the union over all i, for i $= 1\ldots k$ of the NFA's transition function $\delta_N(q_i,a)$. To prove equivalence of the DFA and NFA representation we must show that $\delta_N(q_0,w) = \delta_D(\{q_0\},w)$. The proof is an induction on the string w. For the basis step we take $w = \epsilon$: $\delta_N(q_0,\epsilon) = \delta_D(\{q_0\},\epsilon) = \{q_0\}$. This is true by the basis rule for extending the delta function for NFA's and DFA's. We assume that the inductive hypothesis holds from strings shorter than w. Let $w = xa$. The inductive hypothesis holds from the string x. Let $\delta_N(q_0,x) = \delta_D(\{q_0\},x) = S$. S represents a label for a set of states of the NFA. Let T be the union over all states p in S of $\delta_N(p,a)$.

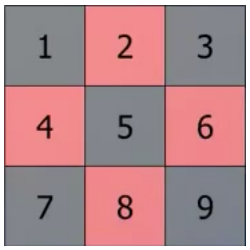


Figure 8: A traditional chess board enumerating the black and red squares with odd and even numbers, respectively. In this board a transition onto state 9 constitutes an accepting state.

Then $\delta_N(q_0, w) = \delta_D(\{q_0\}, w) = T$ by the rule for extending delta functions of NFA's and DFA's. Thus, DFA's and NFA's are equivalent.

6.2 Conversion from NFA to DFA

An example of a transition function on a chess board shown in Figure 8 is given by the NFA's transition table in Figure 9 on the next page. An equivalent DFA transition table is given by the adjoining table in the diagram. A lazy conversion technique can be carried out to only include the state sets in the DFA only when necessary. An automaton modeling the NFA representation of the chess board is depicted in Figure 10 on page 20. A corresponding DFA automaton representation is depicted in Figure

7 $\epsilon-$NFA's

NFA's that allow for epsilon transitions are called epsilon NFA. Epsilon transitions allow for an automaton to transition between states without regard to the input string. Effectively, it allows for the computation to skip forward in the automaton without processing input. An example of an epsilon NFA is depicted in Figure 12 on page 22.

7.1 Closure of States

The closure of states is defined as the set of states reachable from state in question. The function is traditionally denoted $CL(q)$. The closure of state $q2$ in Figure 12 on page 22 is $\{q2, q1, q3, q5\}$.

8 Regular Expressions

Regular expressions use three operations: union, concatenation, and Kleene star. Concatenation on languaguages L and $M$is denoted LM. LM is defined as wx where $w\varepsilon L$ and $x\varepsilon M$. Kleene star (*) is the set of strings formed by

NFA	r	b
1	2,4	5
2	4,6	1,3,5
3	2,6	5
4	2,8	1,5,7
5	2,4,6,8	1,3,7,9
6	2,8	3,5,9
7	4,8	5
8	4,6	5,7,9
*9	6,8	5

DFA	r	b
$\{1\}$	$\{2,4\}$	$\{5\}$
$\{2,4\}$	$\{2,4,6,8\}$	$\{1,3,5,7\}$
$\{5\}$	$\{2,4,6,8\}$	$\{1,3,7,9\}$
$\{2,4,6,8\}$	$\{2,4,6,8\}$	$\{1,3,5,7,9\}$
$\{1,3,5,7\}$	$\{2,4,6,8\}$	$\{1,3,5,7,9\}$
*$\{1,3,7,9\}$	$\{2,4,6,8\}$	$\{5\}$
*$\{1,3,5,7,9\}$	$\{2,4,6,8\}$	$\{1,3,5,7,9\}$

Figure 9: In the transition table for either the NFA or DFA. Odd numbered states can be viewed as the black squares of a chess board while even numbered states are the red squares. Note that the list of states in the NFA transitions expresses the fact that the NFA transitions to the enumerated states simultaneously, while the DFA bracketed states express the label of the single state to which the DFA transitions on the input symbol. In the DFA table the state set label $\{2,4\}$ is assigned the state $\{2,4,6,8\}$ on input r because it represents the union of the transitions of the states to which the NFA will transition to on input r from either state 2 or 4. This logic applies for all other states set given in the DFA table as well. Star symbols in either table indicate a final state.

concatenating 0 or more strigns of L in any order. The Kleene star of L is then $L^* = \{\epsilon\} \cup L \cup LL \ldots$

8.1 Precedence of Operators on Regular Expressions

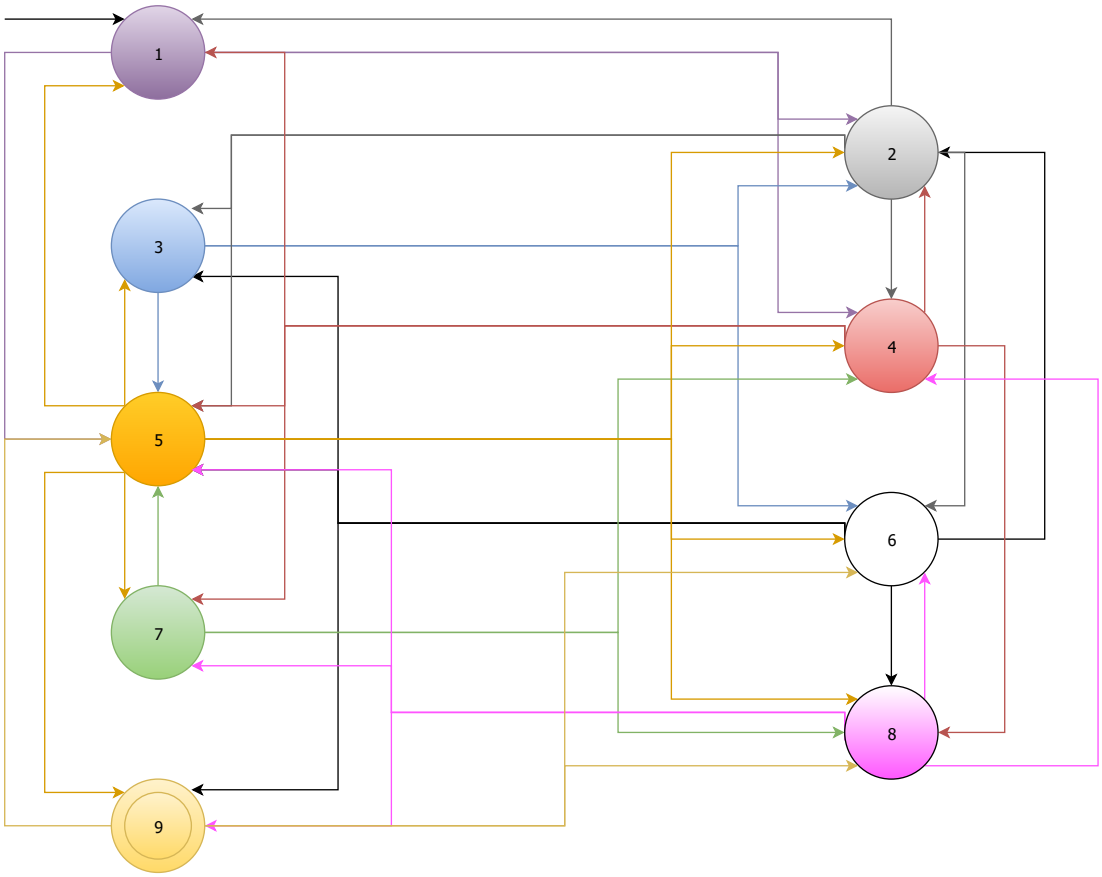
1. Parenthesis
2. Kleene star
3. Concatenation
4. Union

8.2 Algebraic Laws and Identities of Regular Expressions

Union is communative and associative. Concatenation is associative however is not communative.$\emptyset$ is the identity for union.$\epsilon$is the identity for concatenation. $\emptyset$ is the annihilator for concatenation.

9 Properties of Language Classes

A language class is a set of languages, and language classes have two important properties.



ır>

Figure 10: The chess NFA automaton. Transitions from odd numbers to even numbers occur on the input symbol r. Transitions from even numbers to odd numbers occur on input symbol b. Transition labels have been ommited for the sake of clarity.

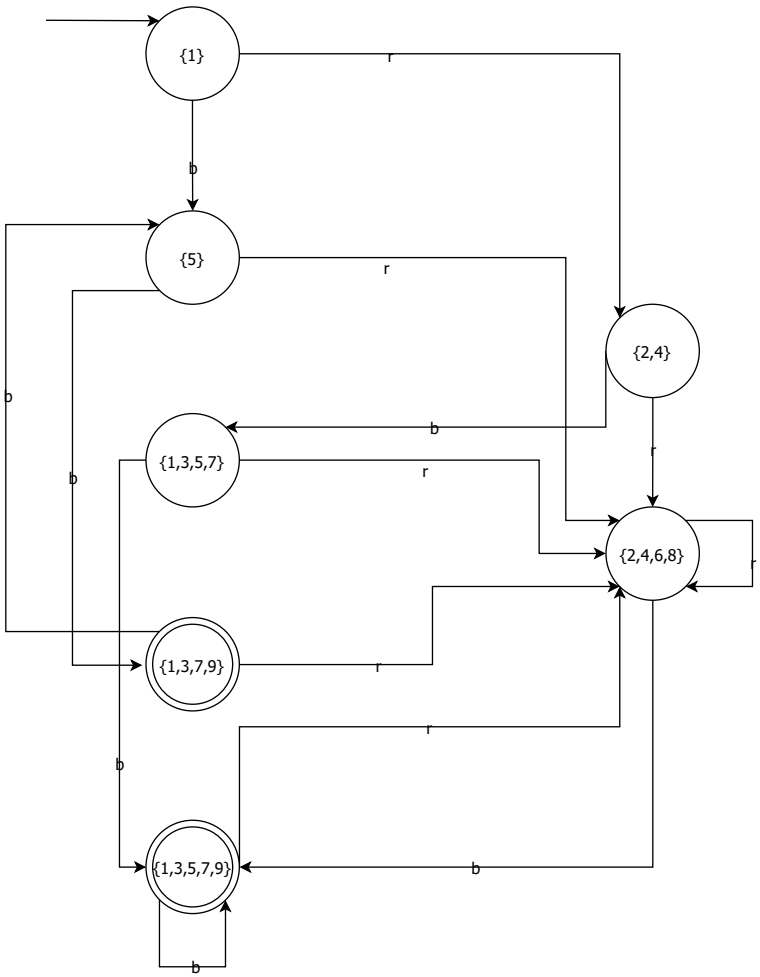


Figure 11: The chess DFA automaton.

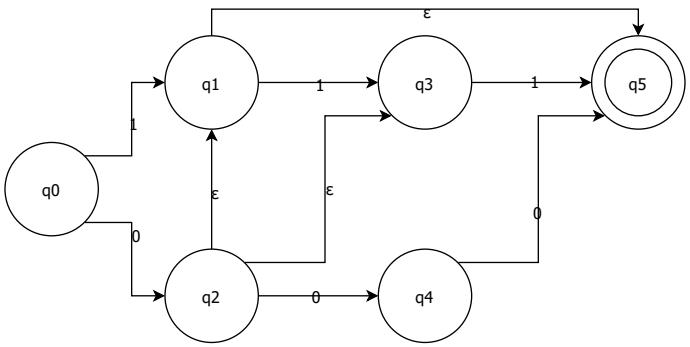


Figure 12: Epsilon transitions are marked with a ϵsymbol. Upon transitioning to state q_2 the automaton spontaneously and in parallel transitions to states q_1, q_3 and q_5.

- Decision properties - algorithms that when applied to a language determine whether a certain property holds (e.g. is a language empty).
- Closure properties - describes the operations in which when applied to a language class produce another language in the same language class (e.g. union)

9.1 Decision Properties of Regular Languages

9.1.1 The Membership Test for Regular Languages

The classic decision property for languages is the Membership Test. That is, is a given string w in the language. The algorithm to determine this is simply to simulate the string on the DFA of the language. If the simulation of w on the DFA ends in an accepting state, then the string is in the language, otherwise the string has ended in a non-accepting state of the DFA and it is thus not in the language.

9.1.2 The Emptiness Test for Regular Languages

Another example of decision properties is the Emptiness Problem. That is given a regular language, does it contain any strings at all? To determine whether the language has any strings first obtain the DFA-representation of the language and then compute the reachable states from the start state. A good way of doing this is Breadth-First Search on the graph of the DFA. If a accepting state is reachable from the start state then we know that at least one string is in the language.

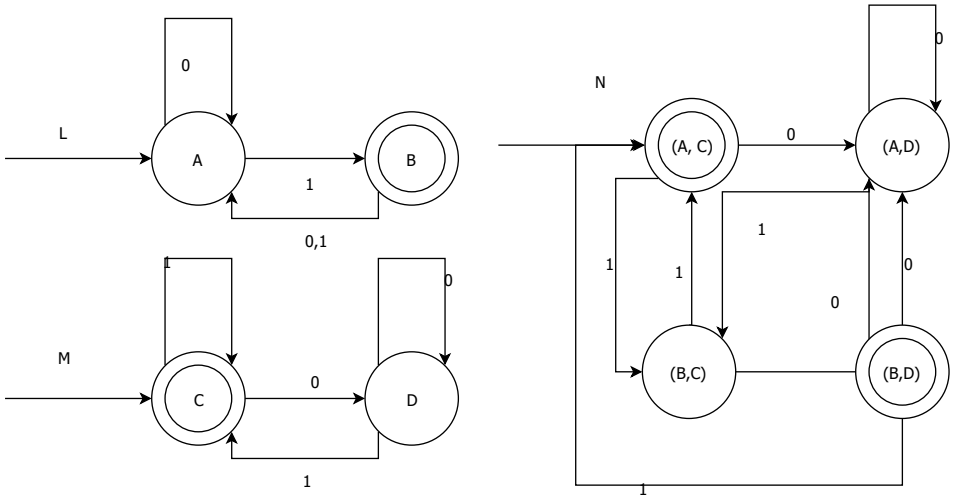


Figure 13: The transition function formed on newly formed state set of $Q \times R$ corresponds to the transitions of L and M.

9.1.3 The Equivalence Test for Regular Languages

Another example of decision properties is testing whether two languages, L and M, are equivalent. Their DFA's are specified as follows: $L = (Q, \Sigma, \delta_L, A, F_L)$ and $M = (R, \Sigma, \delta_M, C, F_M)$ where $Q = \{A, B\}$, $R = \{C, D\}$, and $\Sigma = \{0, 1\}$. The transition function on each DFA, L and M can be seen in Figure 13. To test equivalence one may form a product on the states of each DFA such that for each respective state in Q and R the product DFA's state set is $Q \times R$ has a corresponding state. The start state for the product DFA would then be the pair $[q_0, r_0]$. The transition function would then be formed as follows: $\delta([q, r], 1) = [\delta_L(q, 1), \delta_M(r, 1)]$. The product DFA N pictured in Figure 13 is assigned final states such that either the one of the state pair's respective DFA enters a final state, but not both. For example, the state B is a final state in the DFA L, and the state C is a final state in the DFA M. The state pair (B,D) is then marked final in the DFA N instead of (B,C) because the state D is not a final state in the DFA M. Similarly, the state pair (A,C) is marked a final state in the DFA N because C in final state the DFA M, but A is not a final state in the DFA L. This feature of only one state being accepting is used a differentiating characteristic to determine whether the languages of the DFA's L and M are equivalent. The languages L and M are equivalent if and only if the language of N is empty. This means that N models the feature that neither L accepts when M does not, nor does M accept when L does not. It is clear from the that the empty string ϵ is accepted by M and not by L and is so modeled by the pair (A,C) in N. N accepts the empty string and therefore, L and M are not equivalent languages.

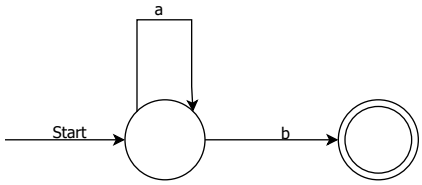


Figure 14: In the automaton pictured above a string w is formed by the the substrings x, y, and z. The substring x is the substring that takes us to the first cycle in the automaton. The cycle occurs at state q. The edge labeled y represents the edge which returns the string w to state q for the first time. Because the substring y is locked within the string w, that is its prefix is the substring x and its postfix is the substring z, it cannot be the empty string. However, the substrings x or z may be the empty string. Then xy^iz for all $i \geq 0$ is in the language, and therefore an infinite number of strings exist in the language.

9.1.4 The Infiniteness Test for Regular Languages

A final example of decision problems fro regular languages in the Infiniteness Problem. The problem asks, is the given regular language composed of an infinite number of strings? We can determine if a given language is infinite by examining the DFA-representation of the language. If the DFA has n states, and the given language has a string of n or more in length, then the language is infinite. Otherwise, the language must be finite. (If a language has a string of length n or more, then surely the DFA contains a cycle that yielded such a string). The proof of this idea follows: If there is a n-state DFA that accepts a string w of length n or more, then there must be a state that appears twice on the path traced out by the simulation of w on the DFA from the start state to the final state. This is because for all strings w of length n a DFA must traverse n+1 states. A diagram of the automaton demonstrating this principle can be seen in Figure 14.

Claim: If there is a string of length $\geq n$ in L, then there is a string of length $[n, 2n-1]$. Proof: Because y is the first cycle on the path to the accepting state, the length $|xy| \leq n$, and more specifically, $1 \leq |y| \leq n$ (x and z may be empty strings). Some state along the path xy surely must repeat. If w is the shortest possible string of length n, then it cannot be longer than 2n. However, suppose it was. The string xz is another accepted string in the language. We know that xz = w - y, and the length of $y \leq n$, so the length of $xz \geq n$. Which means that $|xz| \leq |w|$, and yet at least n in length which is accepted, but we assumed that the were no strings that were shorter than w and of length at least n. Thus, if the string w were of length $\geqq 2n$, there is a shorter string of length $[n, 2n-1]$ formed by removing the substrings represented by y

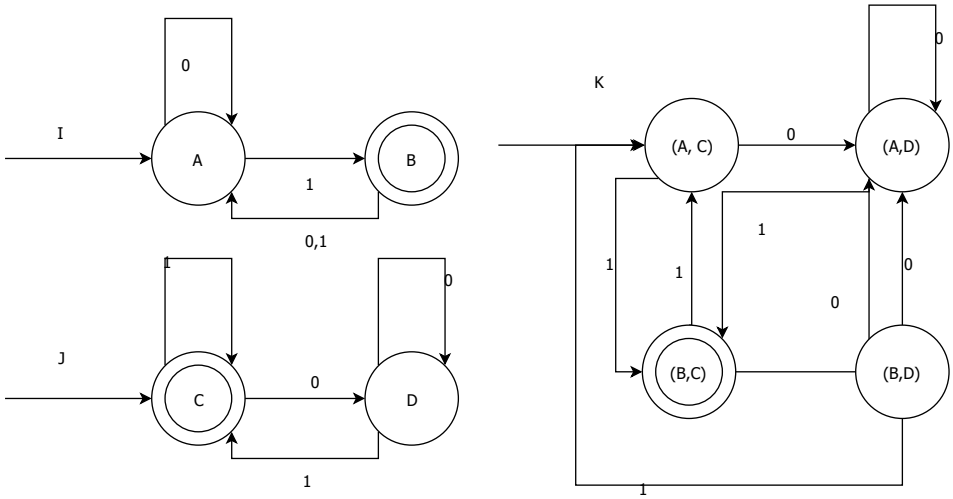


Figure 15: The product DFA K represents the intersection of L and M and its final state is the state pair for which a final state exists for both I and J.

which are $[1, n]$.

9.2 Closure Properties of Regular Languages

9.2.1 Union for Regular Languages

If L and M are regular languages, then $L \cup M$ is also a regular language. The proof of this statement is as follows: Let the languages L and M be the languages of the regular expressions R and S respectively, then $R+S$ is a regular expression whose language is the $L \cup M$.

9.2.2 Intersection for Regular Languages

If L and M are regular languages, then $L \cap M$ is also a regular language. The proof of this statement is as follows: Let I and J be the DFA's for the regular languages L and M, respectively. Form the product DFA K of I and J and mark the final states of K as the states in which both I and J are in final states. The DFA K is now a regular language representing $L \cap M$. This idea is demonstrated in Figure 15.

9.2.3 Difference for Regular Languages

If L and M are regular languages, then $L - M$ is also a regular language. $L - M$ represents the strings that are in L but not in M. The proof of this statement is as follows: Let I and J be the DFA's for the regular languages L and M, respectively. Form the product DFA K of I and J and mark the final states of

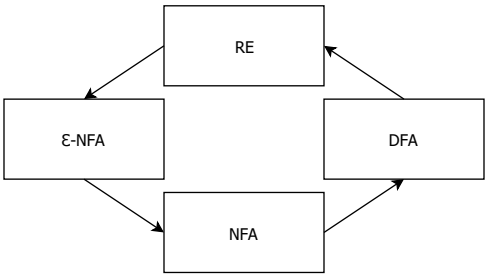


Figure 16: This diagram describes the conversion cycle of representations such that from a regular expression it is possible to obtain a DFA.

K as the states in which I is in a final state and J is not in a final state. The DFA K now represents $L - M$.

9.2.4 Concatenation for Regular Languages

If L and M are regular languages, then LM is also a regular language. The proof of this statement is as follows: Let the languages L and M be the languages of the regular expressions R and S respectively, then RS is a regular expression whose language is the LM.

9.2.5 Kleene Closure for Regular Languages

R^* is a regular expression whose language is L^*.

9.2.6 Complement for Regular Languages

The complement of a language L with respect to an alphabet Σ such that Σ^*contains L is the difference of Σ^* and the language L. Since Σ^* is surely a regular language, and regular languages are closed under difference as we have seen previously, then the complement $\Sigma^* - L$ is also regular.

10 Conversion of Language Representations

A cycle is formed on the way in which we can represent languages. There exists algorithms to convert from one representation to the next along the cycle depicted in Figure 16. As such, given a regular expression one may convert it to a DFA.

11 DFA Minimization

Minimizing a DFA can be carried out in a naive way by enumerating all smaller DFA's for any given DFA and checking for equivalence. However, this is terribly

inefficient and a smarter algorithm can be used to generate a equivalent DFA much faster.

11.1 DFA Minimization Algorithm

The key idea here is to create a table used to find distinguishable states. Distinguishable state pairs are those that have one member of the pair in an accepting state while the other is not in an accepting state. The table used to distinguish these pairs is formed by the product of the DFA's states, $Q \times Q$. If states can be distinguished then they represent a state in minimum DFA, while those that are indistinguishable can be merged into a single state. Effectively, we are recursively finding the shortest distinguishable string for a DFA.

12 The Pumping Lemma for Regular Languages

For every regular language L, there is an integer n, which happens to be the number of states in the DFA of L, such that for every w in L of length $\geq n$. We can write w = xyz such that the following properties hold:

1. $|xy| \leq n$
2. $|y| > 0$
3. For all $i \geq 0$, xy^iz is in L.

The Pumping Lemma is useful in determining whether a language is regular or nonregular.

12.1 Using the Pumping Lemma to Prove a Language is Nonregular

Claim: L = {$0^k1^k \mid k \geq 1$} is a nonregular language. Proof: Suppose for the purposes of contradiction that L is a regular language, then there exists an n for which the properties of the Pumping Lemma hold true.Let w = 0^n1^n. The string w can be written in the form w = xyx where each component x, y, z forms some substring of w where x and consist of 0's and $y \neq \epsilon$, and z is composed of 1's. However, for this to be a regular language all the properties of the Pumping Lemma must hold true. In particular particular property3. For $i = 2$, $w = xyyz$. The string formed by this construction contains more 0's than 1's, violating the conditions on the language as specified in the set former. Similarly, if yz and consist of 1's and $y \neq \epsilon$, the string formed by this construction contains more 1's than 0's again violating the conditions of the language. Thus, the language must be nonregular because it does not meet the conditions of the Pumping Lemma.

13 Context Free Grammars

A context free grammar is a notation for describing languages. It is more powerful than regular expressions and finite automata, but still cannot define all possible languages. A context free grammar is composed of variables which stand for a set of strings. These variables are defined in terms of each other. These rules for how variables should be composed are often called productions. Take for example the language L = $\{0^n 1^n \mid n \geq 1\}$. This is a familar language that we saw earlier in the examples on what is not a regular language. A context free grammar (CFG) for L can be defined as $S \rightarrow 01$, $S \rightarrow 0S1$. Here S is defined recursively such that there will always be an equal number of 0's and 1's.

13.1 CFG Nomenclature

- Terminals - symbols of the alphabet of the language being defined.
- Variables (Nonterminals) - A finite set of other symbols, each of which represents a language.
- Production (Rule) - a rule defining the relationships of terminals and nonterminals. It has the form HEAD -> TAIL (BODY). The body is composed of terminals and nonterminals. Each production represents a language on its own and nontermials in the BODY represent languages on their own. Nonterminals are subsets of the parent language given by the head (e.g. $A \rightarrow XY$, where the concatenation of language X and language$Y$ forms the language A).
- Derivation - The process of repeatedly replacing symbols for terminals based on the productions of a CFG.
- Sentential form - the derived string.

13.2 Conventional Usage of Context Free Grammars

Captial letters at the beginning of the alphabet are typically used as variables (e.g A, B, C). Lowercase letters at the beginning of the alphabet are used as terminals (a, b, c). Capital letters at the end of the alphabet can be either terminals or variables (X, Y, Z). Lowercase letters at the end of the alphabet are strings containing terminals only (e.g. w, x, y, z). Greek letters are strings of terminals or variables (α,β,γ). A star (*) indicates 0 or more steps are necessary to obtain a specified derivation. A variable followed by the epsilon symbol effectively causes the variable to disappear in the derivation process. A $\Rightarrow_{lm}$ indicates the sentenial form is as specified after one setp of the left most derivation. A $\Rightarrow *_{lm}$ indicates 0 or more leftmost derivations are required to obtain the specified sentential form. Corresponding symbols for the rightmost derivations exist as well.

13.3 Context-Free Languages

A language that is defined by some CFG is called a context-free language. Intuitively, a context free language is a language that can count to infinite elements but not three. An example of a non-context freem language is $L = \{0^n 1^n 2^n | n \geq 1\}$.

13.4 Leftmost and Rightmost Derivations

Leftmost derivations process a string's variables one-at-the-time from left to right. Similarly, the rightmost derivation is processed from right to left.

13.5 Normal Forms for CFG's

Poorly designed context free grammars may have rules that are never used in the derivation of strings. This is similiar to a DFA that has unreachable states. CFG's may also have redundant productions that may be combined.

13.5.1 Eliminating Useless Variables from CFG's

If a CFG's production never derives a terminal string, then it is useless. In order to discover useless productions and eliminate them we must have an inductive algorithm which determines how the derivation proceeds by marking the variables that do derive terminals. The variables not contained in the set discovered by the algorithm can then be eliminated. The algorithm follows: For the basis step find a productions that derive a string of terminals. If there is a production that derives a string of terminals and variables whose variables have productions that yields strings of terminals alone, then this production derives terminal strings. For example if there is a production $A \to w$, where w consists only of terminals, then this qualifies as our basis. If we then have a prodcution $A \to \alpha$ who derives a string of variables and terminals than theis may qualify for our inductive step. Additionally, we may have unreachable variables in the grammar. Variables that are unreachable can be eliminated from the grammar by an induction on the variables that are reachable from the start symbol S and the productions that envolve the symbols of those that were discovered. Variables not discovered can then be eliminated.

13.5.2 Elimininating Epilson Productions

Epsilon prodcutions are of the form $A \to \epsilon$. These productions can be eliminated from the grammar, however in doing so the grammar loses the ability to represent the empty string. To eliminate epsilon productions we need to discover the nullable symbols of the grammar. A nullable symbol is a symbol that eventually derives the empty string (i.e. A$\Rightarrow^* \epsilon$). The following inductive argument can be used to discover nullable symbols. The basis is obviously if a production directly derives epsilon as in $A \to \epsilon$, then it is a nullable symbol.

The induction then becomes, if there is a production $A \to \alpha$, in which all the symbols of αeventulay derive epsilon then the production is nullable.

13.5.3 Eliminating Unit Productions

Unit productions are productions whose body consist of a single variable. The key idea is that if a variable A eventually derives a variable single variable B by a series of unit productions, and Bderives the production $B \to \alpha$, then we can rewite A as the production $A \to \alpha$ and drop all the intermediate unit productions. The algorithm to discover unit productions is as follows: Find all pairs of variables (A, B) such that $A \Rightarrow^* B$ by a sequence of unit production only. The induction has the basis (A, A) and the IH if we have found (A, B) and B derives C, then A derives C and so they form the pair (A, C).

13.5.4 Representing Grammars in Chomsky Normal Form (CNF)

Context free grammars in Chomsky Normal Formal are restricted to productions of two types:

- Productions with two variables ($A \to BC$).
- Productions with a single terminal ($A \to a$).

Theorem: If a language L is a context free language, then the language with all epsilon productions eliminated has a context free grammar in Chomsky Normal Form. Forming the CFG such that it is in CNF can be carried out by cleaning up the grammar as descibed in the subsections 13.5.2 on the preceding page, 13.5.3, and 13.5.1 on the preceding page.

14 Pushdown Automata

A pushdown automata(PDA) is equivalent in power to a context free grammar (CFG) in that any language defined in terms of a CFG can be equally defined in terms of a PDA. Traditionally, when speaking about PDA's we mean the nondeterministic type (NPDA). NPDA's are more powerful than their deterministic counterparts. Intuitively speaking a PDA is like an $\epsilon-$NFA with the additional power of manipulating a stack. The computation of the PDA is controlled by the state of its $\epsilon-$NFA, the input symbol to be processed, and finally the symbol on top of its stack. Figure 17 on page 32. In addition to transition to a new state, the PDA may push or pop symbols off of the stack. PDA's are typically described by the following components Q a finite set of states, Σ an input alphabet, Γ a stack alphabet, δ a transistion function, q_0 a start state,z_0 a stack start symbol, and a set of final states $F \subseteq Q$. The transition function δ is parameterized by three components: a state in Q, an input symbol in Σ, and a stack symbol in Γ (e.g. $\delta(q_0, a, Z)$). The δ of given parameters yields a set of tuples containing a state to transition to and a symbol to manipulate the stack (p, a). Here p represents the next state and $a$is a string of stack symbols possibly empty.

14.1 Conventional Usage of Pushdown Automata

Lowercase letters at the beginning of the alphabet (e.g. a, b, c, and ϵ) are used as input symbols. Capital letters at the end of the alphabet are used as stack symbols (e.g. X, Y, and Z). Lowercase letters at the end of the alphabet (e.g. w,x,y, and z) are used as strings of input symbols. Greek letters are used as strings of stack symbols (e.g. α, β and γ) are strings of stack symbols. An example of a PDA modeling the language $L = \{0^n 1^n | n \geq 1\}$ is given in Figure 18 on the following page.

14.2 Instantaneous Descriptions and the Goes-To Relation

An instantaneous description(ID) of a PDA is like a snapshot of a PDA as it computes. Each sucessesive step in Figure 18 on the next page has a corresponding instantaeous description triple describing the current state q, the remaining input w,and the contents of the stack a. The vertical dash symbol, $\vdash$, represents the Goes-To relation. The relation means that for a specified ID I a tranistion to ID J is possible in one step. For example if the transition function $\delta(q, a, X)$ yields (p, β) the we may specify the Goes-To relation as $(q, aw, Xa) \vdash (p, w, \beta\alpha)$. Similiarly, the $\vdash^*$ means the transition occurs in zero or more steps.

14.3 PDA Language Descriptions

The languages of PDA's are defined by $L(P)$ for the set of strings w such that for the ID $(q_0, w, Z) \vdash^* (f, \epsilon, \alpha)$ for the final state f and the stack string a. This is the set of strings w that are consumed by the PDA such that only the empy string, ϵ, is left in the final state. The notation $N(P)$ for a given PDA P describes the PDA's language when the stack becomes empty.

15 Equivalence of CFG's and PDA's

15.1 Converting CFG's $\Rightarrow$ PDA's

For some grammar G, let the language $L = L(G)$. We construct the PDA P such that upon emptying the stack we obtain a string in the language L (i.e. $N(P) = L$). P has one state q. The terminals of the grammar G become the input symbols Σ of the PDA P. The variables and terminals of G make up the stack symbols Γ of the P. The start symbol of G becomes the start symbol of P. Intutitively, we must model the left sentential form of a grammar with the PDA. If the stack of the PDA P is a, and Phas so far consumed $x$from its input, then P represents the left-sentential form xa.

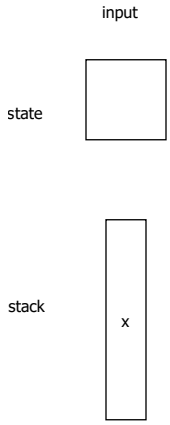


Figure 17: Three elements control how a PDA will compute: the input, the state, and the stack

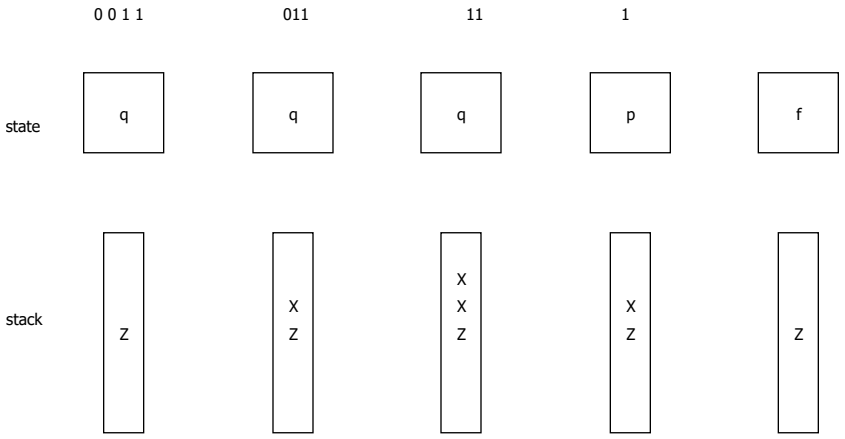


Figure 18: The computation shows the evolution of the PDA's stack, input and state as the string is processed. The initial starting state is shown on the left and the final accepting state is shown on the right. The following transitions define the behavior: $\delta(q, 0, Z) = \{(q, XZ)\}$, $\delta(q, 0, X) = \{(q, XX)\}$, $\delta(q, 1, X) = \{(p, \epsilon)\}$, $\delta(p, 1, X) = \{(p, \epsilon)\}$, $\delta(p, \epsilon, Z) = \{(f, Z)\}$

15.1.1 Creating the Transition Function

There are two kinds of rules in the transition function of P, depending on whether a terminal or variable of G is at the top of P's stack. The *type-1* rules handle the case where "a" is the terminal on top of P's stack ($\delta(q, a, a) = (q, \epsilon)$). There better be an "a" as the next input symbol, or P has guessed wrongly about the leftmost derivation of the input as it actually exists. In effect, we "cancel" the "a" on the stack against the "a" on the input. The left-sentential form represented does not change. We have now consumed one more symbol, "a", from the input so that becomes part of the left-sentiantial form. But the "a" that was on the stack is removed, so it no longer participates in the left-sentential form. The *type-2* rules handle a variable, say A, on the top of the stack ($\delta(q, \epsilon, A) = (q, a)$). We need to expand that variable by the body of one of its productions, and thus move to the next left-sentential form. Of course we're only guessing. We have to allow any of $A's$ productions to be used. If $A \to a$ is one of these productions, then a choice for P, using epsilon input, and with A on top of the stack, is to replace the A by a.

15.1.2 Proof of Equivalence between CFG and PDA

In order to prove equivalence between the two representations we need to prove that for all x,$(q, wx, S) \vdash^* (q, x, \alpha) \leftrightarrow S \Rightarrow^*_{lm} w\alpha$. That is, for all x the instantaneous decription (q, wx, S) goes to (q, x, α) in some number of steps of the PDA if and only if the CFG's leftmost derivation yields $w\alpha$. It is important to note that we allow for the suffix $x$in the string wx. The string x has no effect on the process of consuming w and so if the that statement is true for one $x$it is true for all x. The proof follow starting from the only if direction ($\to$). That is, if P transitions (q, wx, S) goes to (q, x, α), then $S \Rightarrow^*_{lm} w\alpha$. This will be an inductive proof where the base case is 0 steps have taken place and w is ϵ and $\alpha$is S. The truth of the base case becomes trivial because S is α and $w\alpha$ is just α because ϵ serves as the identity of concatenation. Surely then, S derives itself. Now we must consider the next n steps of the proof, and assume an inductive hypothesis for the sequence of $n-1$ steps. For the n^{th} step two cases can occur. A type-1 rule is used, or a type 2 rule is used. We will consider both case in turn. The descriptions of these rules are described in the previous subsection. For the use of a *type-1* rule in the n^{th} step, the ID's for the 0 through $n-1$ steps must be $(q, yax, S) \vdash^* (q, ax, a\alpha) \vdash (q, x, \alpha)$, where $ya = w$. That is, the prefix of the string $x$must end with the symbol a with the a symbol on top of the stack followed by α, and in the previous $n-1$ steps the string y is consumed by the PDA, and in the n^{th} step the PDA consumes the input symbol a and removes it from the stack. By the inductive hypothesis applied to the $n-1$ steps, we can conlude that there is a left most derivation from S to $ya\alpha$ because y was consumed and thus equivalent to $\epsilon$and $a\alpha$ is on the stack. The symbol $a$is then consumed from the input and popped from stack taking us back to the base case because $ya = w$. For the use of a *type-2* rule in the n^{th} step, the step sequence must be $(q, wx, S) \vdash^* (q, x, A\beta) \vdash (q, x, \gamma\beta)$ where $A \to \gamma$ is a production in the

grammar and $\alpha = \gamma\beta$. In this case, we have a variable A on top of the stack followed by the stack string β. In the n^{th} step no input is consumed but the variable A is replaced by the stack string γ. By the inductive hypothesis we obtain $S \Rightarrow^*_{lm} wA\beta$ after the first $n-1$ steps. Because A is the leftmost variable in the derivation it is replaced by γ. We then have $S \Rightarrow^*_{lm} w\gamma\beta = w\alpha$ returning us to our base case, thus proving the only if direction of the proof. Next, we prove the if direction ($\leftarrow$). We now must prove $S \Rightarrow^*_{lm} w\alpha \rightarrow (q, wx, S) \vdash^* (q, x, \alpha)$ for all x ...[ommited]

15.2 Converting PDA's $\Rightarrow$ CFG's

In this section we obtain the grammar G from a PDA P. We assume that the language L is accepted by the PDA upon emptying its stack, more formally $L = N(P)$. Intuitively, we should assign variables labled as pXq to the grammar G for the transitions from state $p$to state $q$when popping the symbol X from the PDA's stack. Upon popping the X from the stack it may grow, but the stack size will not shrink below the size when X was popped off the stack until the last step in processing is taken.

15.2.1 Constructing Variables of the CFG

The variables of G are correspond to labels such as pXq which can be viewed as a single symbol modeling the transition from p to q with X on the stack in which the input is consumed and generates all and only the strings w until $(p, w, X) \vdash^* (q, \epsilon, \epsilon)$. Note that since the initial ID shows nothing below X on the stack, we know that X can't be popped until the last step, since PDA P cannot make any moves when its stack is empty.In addition to the aforementioned variable a start symbol S is needed.

15.2.2 Constructing Productions of the CFG

Intuitively, the productions or rules of the grammar represent steps of the PDA. Each rule for the example rule pXq is sourced from the PDA in state $p$with the stack symbol X. In the easiest case we have the following for the PDA's transition function. $\delta(p, a, X)$ yields (p, ϵ). Here a denotes either a input symbol or ϵ. The grammar in this case can model the PDA's behavior by popping $X$and not replacing it with any other symbol ($pXq \rightarrow a$). The next simplest case simplest case involves modeling transition between productions. Transitivy is modeled by introducing an intermediate state $r$and a variable Y. The transition function in this case would look like $\delta(p, a, X)$ yields (r, Y). The corresponding production would then be $pXq \rightarrow arYq$. This allows us to erase X and transition from p to q by way of r while reading $a$and pushing Y onto the stack. The final simplest case we will consider is $\delta(p, a, X)$ yields (r, YZ) for some state r and some symbols $Y$and Z. Here X is replaced by YZ. In order for X to be erased, there must be some input string u that has the net effect of erasing Y. And u must take the PDA from state r to some state s, which we don't know. As a

result, we're going to have to have one production for each state s. But after reaching state s, we must have some additional input v that takes the PDA from state s to state q, while popping the Z from the stack. The net effect is thatauv pops X from the stack while going from state p to q. The general case for creating productions to model the PDA follows. Suppose $\delta(p, a, X)$ yields $(r, Y_1, \ldots Y_k)$ for some state r and $k \geq 3$. Then in the grammar generate the set of productions $pXq \rightarrow arY_1s_1s_1Y_2s_2 \ldots s_{k-2}Y_{k-1}s_{k-1}s_{k-1}Y_kq$. This models the case in which X is replaced by three or more symbols transitioning from p to q.

16 Pumping Lemma for Context Free Languages

Recall the concepts for the pumping lemma on regular languages outlined in Section 12 on page 27. The pumping lemma for regular languages relied on a cycle early on in a sufficiently long string w in order "pump" an the string an arbitrary number of times, thus producing a infinite number of strings in the language. Similiarly, the pumping lemma for context free languages models this idea, but requires two cycles along the string's path in tandem.

16.1 Formal Specification of the CFL Pumping Lemma

For every context free language L, there is an integer n, such that for every string z in L of length $\geq n$ there exists $z = uvwxy$ such that the following conditions hold:

1. $|vwx| \leq n$.
2. $|vx| > 0$.
3. For all $i \geq 0$, uv^iwx^iy is in L.

If a language in question does not have these properties, then it is not a context free language.

17 Properties of Context Free Languages

17.1 Decision Properties of Context Free Languages

Like regular languages we can determine many decision properties about context free languages including whether a string in the particular CFL, whether a CFL contains any strings at all, and if the CFL contains an infinite number of strings. Unfortunately,we cannot decide whether two CFL's are equivalent, or whether they are disjoint.

0	1	3	6
	2	4	7
		5	8
			9

Figure 19: A single index array can be viewed as a triangular array by accessing it with a function mapping the rows and columns appropriately. The function $k = j(j+1)/2 + i$ maps the index value k to the appropriate i and column j where $i \leq j$.

17.1.1 Membership Test for CFL's

The membership test is used to determine whether a given string w is the language of the grammar $L(G)$. We need the grammar to be in Chomsky Normal Form, so if it isn't then convert it to this form as described in subsection 13.5.4 on page 30. Then use the CYK algorithm to determine the status of w. The CYK algorithm uses Dynamic Programming to decide whether w is in the language or not, so it may be useful to first review the section 30 on page 59 first in order to gain better understanding of CYK.

17.1.2 The CYK ALgorithm

Let $w = a_0 \dots a_n$ be a string of length $n$where a_i stores the symbol at the i^{th} position. Construct a n by n triangular array. This can done efficiently by using an indexing function where upon a single-indexed array is mapped to an upper-triangular array. The following function $k = j(j+1)/2 + i$ where k represents the index into the array properly maps the elements of the array. This idea can be viewed in Figure 19. Additionaly, each entry of the array should be viewed as a set of variables of the grammar. The set X_{ij} is stored in position (i, j) of the array where $i \leq j$ and is inteded to be the set of variables that derive the substring of the input starting from position i and ending at position j. An inductive argument is used to fill the table on the length of the input string derived . The length of the string can be computed as $j - i + 1$. We start by computing the entries $X_{i,i}$, which is the set of variables that derive the string consisting of the symbol at a_i . Next, we find the variables at $X_{i,i+1}$ each of which derive the string at $a_{i,i+1}$. Then, we move to the $X_{i,i+2}$ variables which are the sets of variables that derive the strings of length three, a_i,a_{i+1},a_{i+2} and so on . Finally, after we have computed the set $X_{1,n}$ which represents the entire input string , we can test whether the start symbol S is contained in the set . If it is then the string is in the language, otherwise it is not. Formally, the basis $X_{i,i} = \{A | A \Rightarrow^* a_i\}$ where a_i represents a single symbol in w. The induction is then $X_{i,j}$=$\{$A| there is a production $A \rightarrow BC$, and an integer k, with $i \leq k < j$, such that B is in $X_{i,k}$ and C is in $X_{k+1,j}\}$.That is, for each k between i and $j-1$ we look for some B in $X_{i,k}$ and some C in $X_{k+1,j}$ such that BC is the body of an A production. If for any k, B, and C we find such a production, we add A to $X_{i,j}$.

17.1.3 Emptiness Test for CFL's

The algorithim to test for an empty CFL uses the ideas we learned in subsection 13.5.1 on page 29. We use these ideas of checking whether variables are useful in the derivation to determine if the start symbol S is a useful symbol. If the start symbol does not derive anything, it is a useless variable, and we can state that the CFL is empty.

17.1.4 Infiniteness Test for CFL's

The test for infinteness in CFL mirrors the idea that was proposed in section 9.1.4 on page 24. Apply those priniciples with the pumping lemma for context free languages outlined in section 16 on page 35.

17.2 Closure Properties of Context Free Languages

For many of the same operations under which the class of regular languages are closed, the context-free languages are also closed. These include the regular-expression operations: union, concatenation, and closure. Also reversal, homomorphism and inverse homomorphism. But unlike the class of regular languages, the class of context-free languages is not closed under intersection or difference.

17.2.1 Union for CFL's

//TODO

17.2.2 Concatentation for CFL's

//TODO

17.2.3 Kleene Closure for CFL's

//TODO

18 Countability

It is important to understand that all computation can be encoded as integers. The ascii or if you like the UTF-8 table encodes english characters and algebraic numerals, which we can interpret as integers. This idea can be applied to all types of media. At the very root we may interpret all things as integers.

18.1 Finite Sets

A finite set is a set that contains a finite number of elements. Refer to Section 1.5 on page 13. It may be useful to refresh your memory on this topic. The formal definition of a finite set is one for which it is impossible to find a one to one

correspondence between the members of the set and a proper subset of the set. An example of a set is $\{a, b, c\}$.

18.2 Infinite Sets

An infinite set is a set for which there is a one to one correspondence between itself and a proper subset of itself. An example of a infinite set is the positive integers $\{1, 2, 3, \ldots\}$. The one-to-one correspondence which validates this argument is the mapping of the positive integers with the even integers ($1 \leftrightarrow 2, 2 \leftrightarrow 4, 3 \leftrightarrow 6, \ldots$).

18.3 Countable Sets

A countable set is a set with a one to one correspondence with the positive integers, thus all countable sets are infinite sets. As another example all integers $\mathbb{Z}$ form a countable set. 0 is mapped to 1 and the negative integers are are mapped to the even numbers ($-i \leftrightarrow 2i$, for all $i \geq 1$), while the positive integers are mapped to odd numbers($i \leftrightarrow 2i + 1$, for all $i \geq 1$). The enumerated sequence then becomes $0, -1, 1, -2, 2, -3, 3, \ldots$ Another example of an enumerable set is the binary strings, but there is a trick involved. The binary strings $101, 0101, 00101$ all cooresponds to the integer value 5, so it seems imposible to form a coorespondence such that they become countable. The trick here is to prepend a 1 to the binary strings so that the strings can become distinguishable. The aforementioned strings become $1101, 10101, 100101$ as integers they are 13, 21, and 37.

18.4 Countability of Languages Over the Binary Alphabet

The next bit is quite confusing (pun intended). The languages over the binary alphabet $\Sigma = \{0, 1\}$ are not countable. To obtain a contradiction we use a technique that confounds a set formers specification. As with the other examples suppose we could encode languages over the binary alphabet so that we could speak about some specific enumerated language. For example, the i^{th} language. Define the language $L = \{$ w $\mid$ w is the i^{th} binary string and w is not in the i^{th} language$\}$. Surely, the language L is a language over the binary alphabet. Thus, L is the j^{th} language for some particular j. Now let some binary string $x$be the j^{th} string. Now consider substituting $x$for w in the j^{th} language L_j $= \{ x \mid x$is the j^{th} binary string and $x$is not in the j^{th} language$\}$. For some arbitrary j^{th} string $x$in the j^{th} language L_j, if $x$is in L_j, then it cannot be by the definition of the language. If $x$is not in the language L_j, then $x$is in the language by the definition of the language. This makes for a very confusing situation. Remember, L contains w if and only if w is not the language that correpsonds to the smae integer i that w corresponds to. We now have a contradiction and therefore we know that we know the languages over the binary strings are not countable. Graphically, this concept can be seen in the Diagonalization Method in Figure 20 on the following page.

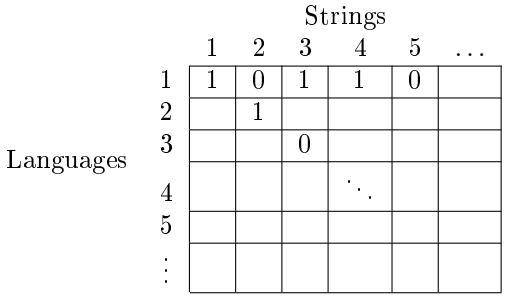


Figure 20: This diagram depicts a table used to describe whether strings are in a language or not. A binary integer 1 indicates that the string j^{th} string is in the i^{th} language, 0 indicates it is not in the language. Look at the diagonal in this table. If you were to first complement the main diagonal and then rotate it by 45 degrees upon an axis created in the i^{th} row and i^{th} column, then it too would look like a language, but would always disagree with itself at the i^{th} row and i^{th} column.

19 Turing Machines

Turing machines model the recursively enumerable languages which encompass the previously discussed languages. Sipser provides a good hierarchy to reference for how to regard which language classes are subsets of which. The diagram is provided in Figure21. The purpose of Turing Machine theory is to provide a means of proving whether or not an algorithm exists for a given language.Reductions which will be explored in detail in later sections prove common questions are undecidable.

19.1 What is a Turing Machine?

A Turing Machine is a computational model composed of a tape and a head that reads and writes symbols onto the tape based on a transition function. The tape is infinite in both directions, and there is also some state that we track throughout computation. In one step of computation the Turing machine may read one symbol from the tape and modify it, and either move the head left or right one square.

19.2 Turing Machine Notation

A Turing machine is made up of the following componenets.

- A finite set of states (Q).
- An input alphabet (Σ).
- A tape alphabet (Γ,typically $\Sigma \subseteq \Gamma$).
- A transition function (δ).

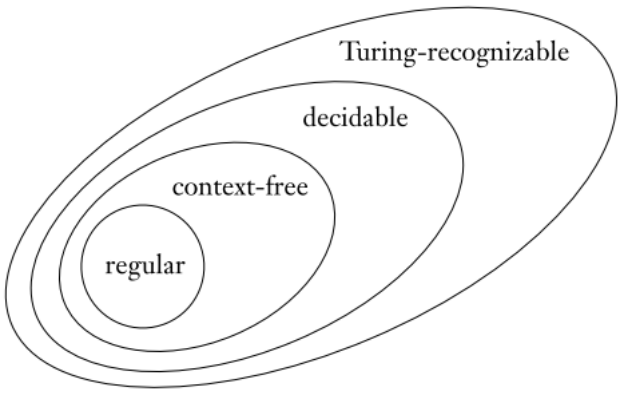


Figure 21: The hierarchy of language classes. Turing-recognizable languages are sometimes called recursively enumerable languages. Decidable languages are sometimes called recursive languages.

- A start state ($q_0 \varepsilon Q$).
- A blank symbol ($\sqcup \subseteq \Gamma - \Sigma$).
- A set of final states ($F \subseteq Q$).

Conventionally, the lowercase letters at the beginning of the alphabet are input symbols (a, b, c). Upper case letters at the end of the alphabet represetn tape symbols (X, Y, Z). Lowercase letters at the end of the alphabet represent input strings w, x, y, z. Greek symbols represent tape symbols (α, β, γ). The transition function δ takes two parameters a $q_i \varepsilon Q$ and a tape symbol in Γ. The transition function yields a triple of the form (state, tape symbol, direction). If for a given input a tranisiton is not defined then the Turing machine halts in the current state.

19.3 Instantaneous Descriptions of Turing Machines

Instantaneous descriptions of Turing Machines are encoded as $\alpha q_i \beta$ where α represents the tape before the head of the Turing Machine until the leftmost blank and β represents the tape after the head until the rightmost blank. The position of the head represented by a state symbol q_i is just left of the tape string β. Further we use the symbol $\vdash$ to indicate "becomes in one move" and $\vdash^*$ as "becomes in zero or more moves."

19.4 Languages of a Turing Machine

A Turing machine's language is defined by its final states or a halting action. An example is $L(M) = \{\ w \mid q_0 w \vdash^* I\ ,$ where I is an ID with a final state$\}$.

Halting can be described by $H(M) = \{w \mid q_0 w \vdash^* I$, and there is no move possible from $I\}$.

19.5 Recursively Enumerable Languages vs. Recursive Languages

The class of languages accepted by a Turing Machine halting or entering an accepting state is called the Recursively Enumerable language class. Some textbooks also call this the Turing Recognizable language class. Turing Machines that accept by final state, and who are *guaranteed to halt* whether it accepts or not define the Recursive Languages class. In some textbooks this is called the Decidable Language class.

19.6 Multitape Turing Machines

Multitape Turing Machines allow for a ordinary Turing Machine to have k tapes for any fixed k. The steps the Multitape Turing Machine makes are dependant on the symbols the head of each tape is pointing to and their corresponding states. Each tape has its own head and they move on their own tape without restricting each others movement. On each read of the tape a new symbol and state is assigned for each tape and head. Additionally, a head may choose to not move. This model is no more powerful than the traditional Turing Machine. One may simply expand the alphabet for a traditional Turing Machine to accomadate the number of tapes that it must simulate.

19.7 Nondeterministic Turing Machines

Nondeterministic Turing Machines are granted multiple choices based on the state, direction, move triple. Once a choice is made, then the next state, new symbol and head direction are determined. As with the NFA's, DFA's, and PDA's, the nondeterministic Turing Machine accpets if any sequence of choices leads to an ID with and accepting state.

19.8 Closure Properties of Recursive and Recursively Enumerable Languages

Recursive Languages and Recursively Enumerable Languages share the properites: union, concatenation, Kleene star, reversal, intersection, and inverse. Recursive languages have difference and complementation. Recursively Enumerable Languages have homomorphism.

19.8.1 Union

Given two Turing machines M_1 and M_2 with languages $L_1$and L_2 respectively. The union of M_1 and M_2 can be achieved by creating a third Turing Machine M. Mwill be a two-tape Turing Machine. M will proceed by copying the tape of

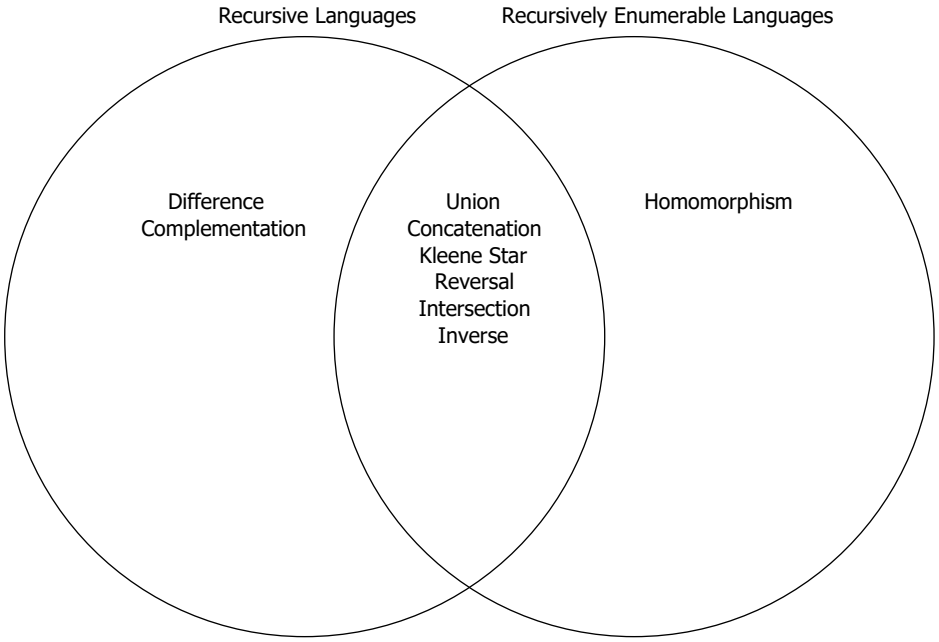


Figure 22: The closure properties of Recursive and Recursively Enumerable languages

M_1 to its first tape, and M_2 to its second tape. Next, M will simulate the input of M_1 and M_2 independently. In the case of Recursive Languages (Decidable Languages), Mwill *accept* when either of its tapes enter an accepting state, and *reject* when **both** of its tapes halt by not accepting. The case for Recursively Enumerable Languages is a bit more relaxed. In this case, M need only to enter an accepting state on one of its tapes to accept. However, it may end up running forever and never providing an answer.

19.8.2 Intersection

The idea behind intersection is quite similiar to union. Given two Turing machines M_1 and M_2 with languages $L_1$and L_2 respectively. The intersection of M_1 and M_2 can be achieved by creating a third Turing Machine M. Mwill be a two-tape Turing Machine. Mwill proceed by copying the tape of M_1 to its first tape, and M_2 to its second tape. Next, M will simulate the input of M_1 and M_2 independently. In the case of Recursive Languages (Decidable Languages), Mwill *accept* when **both** of its tapes enter an accepting state, and *reject* when **either** of its tapes halt by not accepting. In the Recursively Enumerable case, M has to enter an accepting state on **both** of its tapes to accept. However, it may end up running forever and never provide an answer.

19.8.3 Difference and Complement

Again the idea behind Difference and Complement is quite similiar to the intersection. The intersection of M_1 and M_2 can be achieved by creating a third Turing Machine M. M will be a two-tape Turing Machine. M will proceed by copying the tape of M_1 to its first tape, and M_2 to its second tape. Next, M will simulate the input of M_1 and M_2 independently. M will *accept* when M_1 accepts and M_2 rejects, otherwise reject. Corollary, for the complement run the difference algorithm over the Kleene star of the input alphabet. This approach won't work for Recursively Enumerable Languages because M_2 may never halt.

19.8.4 Concatenation

Concatenation will require the use of a two-tape nondeterministic Turing Machine M. Assume that the given machines M_1 and M_2 are semi-inifinite single-tape Turing Machines. M will proceed by nondetminitically guessing the where a given input string w should be split. Call the prefix substring of w the string x, and call the corresponding suffix y. Now copy x onto $M's$ first tape and $y$onto its second tape. Now simulate $M_1's$ transition function on the first tape and $M_2's$ transtion function on the second tape. If both accept then M should *accept*, otherwise *reject*. Because we nondeterministically guess all ways that w can be split the machine will always accept if w is in $L(L(M_1)L(M_2))$.

19.8.5 Kleene Star

For the Kleene star mirror the technique of used concatenation but instead of splitting the input string. Generate the Kleene star of the given Turing Machine and for each component of the guessed set run it against the Turing Machine and if it accepts then accept, otherwise reject.

19.8.6 Reversal

Simply reverse the input and run it against the machine.

20 Decidability

Now that we have seen some algorithms in the preceeding section it is easy to see that one may create a great number of algorithms for Turing Machines. However, we will soon find out that for some problems no algorithm exists. We will link the concepts we learned in the Section 18 on page 37 to formulate a way of encoding Turing Machines in binary. This will allow us to apply the Diagonalization technique to Turing Machines. We can then formalize the notion that a problem posed to a Turing Machine is in fact the language of the Turing Machine. We can then see that for some languages no Turing Machine can exist.

20.1 Encoding Turing Machines in Binary

It is quite simple to come up with a scheme for encoding Turing Machines into binary strings. We will assume the Turing Machines we will be encoding have a input alphate consisting of $\{0, 1\}$. However, this methodology is certainly exapandable to arbitrary alphabets. We should first assign integer values to the three classes of elements of the Turing Machine: states, symbols, and directions. Now we apply a trick for distinguishing codes of the binary string. For each integer representation of the componenet. Convert the integer to binary and insert a zero for each binary digit. Now there can never be consecutive ones in the binary representation. To distinguish between parts of the encoding add consecutive ones. Now we can concatenate the elements of the Turing machine together and prepend a 1 so that the encoding represents a unique integer value, and thus the language of all possible Turing Machines becomes enumerable. Some encodings may give invalid Turing Machines and therefore we regard theses as Turing Machines that accpet only the empty language.

20.2 Diagonalization on the Enumerated Turing Machines

In order to diagonalize the table given by Figure 23 on the following page. We form a sequence D by complementing the values along the main diagonal. Formally $D = a_{i,j}, \ldots, a_{n,n}$ where $i = j$, and the value of $a_{i,j}$ is the complement of its value in the table shown in Figure 23 on the next page. Now we come

		String j→							
		1	2	3	4	5	6	...	
	1	$a_{1,1}$							
	2		$a_{2,2}$						
	3			$a_{3,3}$		x			
TM i→	4				$a_{4,4}$				
	5					$a_{5,5}$			
	6						$a_{6,6}$		
								$\ddots$	
	⋮								

Figure 23: This table expreses which strings j are accepted by which Turing Machines i. If $x$is 0 then the string is not accpeted, otherwise if 1 then it is accepted. The main diagonal D is enumerated as $a_{1,1}, a_{2,2}, a_{3,3}, \ldots, a_{n,n}$.

to the question: Could D be a row representing the language accepted by a Turing Machine of the table? Suppose this imaginary row represented by the main diagonal D where row k, but it can't be row k because it disagrees at the k^{th} entry. Therefore this cannot be the language of any Turing Machine, and further we can descibe this language as the set that contains the k^{th} string if and only if the k^{th} Turing Machine does not accept the k^{th} string. We can name this language L_D and since we know that we can not create a Turing Machine for this langauge, then we know that it is not recursively enumeable (Turing-recogizable), and thus no algorithm can be given to decide L_D.

20.3 Decidable Problems

A problem is decidable if there is an algorithm to asnwer it. Recall, an algorithm formally speaking is a Turing Machine that halts on all inputs, accepted or not. Put another way a deciable problem is a recursive language. We may visualize the relationship among the language classes and the language L_D which we identified in subsection 20.2 on the preceding page in the Figure

20.4 The Universal Turing Machine

A Universal Turing Machine (UTM) is a Turing Machine which accepts as input a Turing Machine M and some string w. The UTM accepts M if and only if Maccepts w. A UTM will have three tapes. Tape 1 holds the input Mencoded as binary and w. Tape 2 holds the tape of M, and Tape 3 holds the state of M. The UTM proceeds by first checking the encoding of M. If M is invalid, then its language is the empty language, and the UTM halts immediately and rejects. The UTM will then examine the size of $M's$ symbols to determine the required word size. Next, the UTM will initialize Tape 2 to represent the tape of $M$with the input w, and initialize Tape 3 to hold the start state. Finally, the UTM can simulate M by looking for a step in the transition function of Mencoded

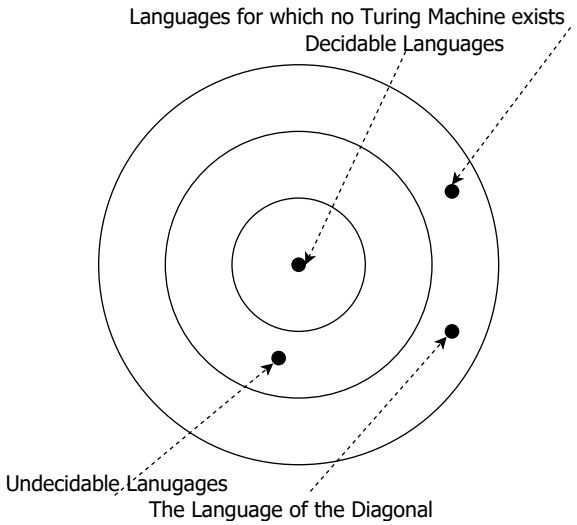


Figure 24: Here we see the hierarchy of the language classes and how they relate to L_D. The Undeciable Languages also called the Recursively Enumerable Languages have a Turing Machive and are regarded to be Turing-recognizable. That is, they have a Turing Machine that will accept strings, but there is no guarantee that the machine will halt. Within this class is the Decidable Languages also called the Recursive Languages which have Turing Machines which are guranteed to distinguish string as either accepted or rejected and they also are guranteed to halt. At the otter most ring we have the language for which no Turing Machine exists and therefore they are unrecognizable.

on Tape 1 that matches the state encoded on Tape 3 with a tape symbol under the head of Tape 2. If a match is found change the symbol and move the head marker on Tape 2 and change the State on Tape 3. If Maccepts, then the UTM accepts.

20.5 The UTM is Recursively Enumerable and Not Recursive (The Halting Problem)

Assume for the purposes of contradiction that the language of the UTM, that is $L(UTM)$, is recursive (decidable). Meaning that for every input it halts and returns an accepting or rejecting answer. We *somehow magically discover* an algorithm that can decide whether given a Turing Machine$M$ and an input string w from the acceptance table featured in Figure 23 on page 45– call this algorithm *wizzBang*. Remember *wizzBang* takes two paramters as arguments. First, we must have a valid Turing Machine M, and second we must have a string w. We're given an input M. Let's suppose M is for the i-th string w of our table. The first thing to do is check whether or not M is a valid encoding for a Turing machine. If the encoding is NOT valid, then the i-th Turing Machine defines the empty language. That means w, the i-th string, is not in the language of the i-th Turing machine. Therefore, w IS in L_D. Remeber we complemented the major diagonal so that answers here must be inverted. Now suppose M is a valid encoding for a Turing machine. Then run *wizzBang* on the input M and w. Here M represents the i-th Turing Machine processing the input that is the i-th string. Eventually, this algorithm will halt and tell us whether or not the i-th machine accepts the i-th string. If the *wizzBang* accepts the i-th machine accepts the i-th string, then we say reject because that means w is not in L_D. However, if *wizzBang* rejects, then we accept w, because w IS in L_D. HERE IS THE MAIN POINT$\Rightarrow$ We previously proved in subsection 20.2 on page 44 that no Turing Machine can exist for L_D and therefore we must conclude that *wizzBang cannot exist*. This tells us that the UTM is Recursively Enumerable (Turing-recognizable), but not Recursive (Decidable). We will regard the language of the UTM as L_U going forward.

21 Some Undeciable Problems

In this section we cover Rice's Theorem which states that almost every question we ask about Recursively Enumerable Languages is undeciable. Next we look to Post's Correspondence Problem to bridge the gap between the world of Turing Machine to solving real world problems. In addition to the classical language classes we have defined we may define properties of languages. A property is a user defined set of languages. For example, the set of languages who are infinite languages have the Infiniteness Property. We can define a language L_P as the set of TM's such that each TM has the property P. There are two trivial propertues P for which L_P is deciable. The always-false propety, which contains no Recursively Enumerable languages, and the always-true property,

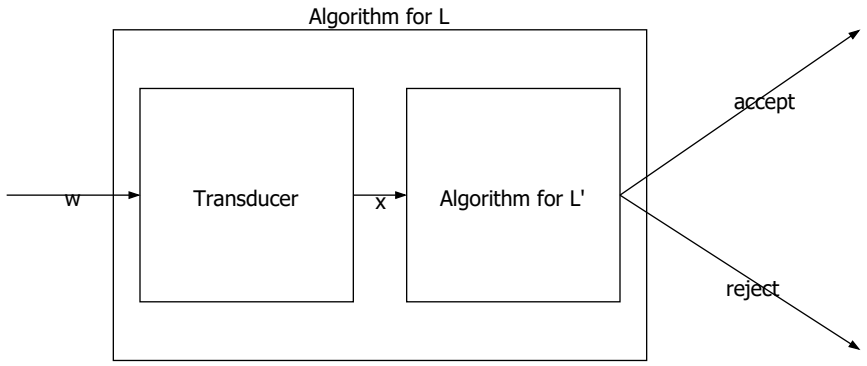


Figure 25: The transducer serves as an adaptor for the input string of w. It transforms w into a string $x$which L' can consume and produce an accept or reject decision. Algorithm L' allows us to draw conclusions about the properties of L.

which contains every Recursively Enumerable language. Rice's Theorem states that for every other property P besides the trivial ones L_P is undecidable. In order to prove Rice's Theorem we will have to introduce the concept of reductions. A reduction is an algorithm (Recall, an algorithm is a TM that always halts), which transforms an input string w in L to a string $x$in L' with the property that $x$is in L' if and only if w is in L. The takeaway here is that given the reduction for L we are able to say that L is no harder than L'. We can visualize reductions as shown in Figure 25.

21.1 Rice's Theorem

We now need to prove the claim of Rice's Theorem: Every nontrivial property P of the Recursively Enumerable languages, L_P, is undecidable. We're going to assume that the empty language does not have property P. If that is not the case, then consider the complement of P, say Q. Surely the empty language then has property Q. But if we could prove that Q were undecidable, then P also must be undecidable. That is, if L_P were a recursive language, then so would be L_Q, since the class of recursive languages is closed under complementation. We will use the UTM (i.e. L_U) obtained in Section 20.5 on the preceding page. In this reduction we reduce L_U to L_P. Because L_U is undecidable this gives us the ability to say that L_P is undecidable. Our reduction must take the parameters $M$and w and create a new Turing Machine M' such that $L(M')$ has property P if and only if M accepts w. That is, the result of M' is contingent on the result of M's result from taking in w. Our language property testing machine M' will be testing the language L such that L is any language with a property P, and we let M_L be a TM that accepts L. M' simulates M on input w, and if M accepts w then M' simulates M_L on the input $x$to M'. M' accepts its input $x$if and only if M_L accepts x. Suppose M accepts w, then M'will simulate M_L on input

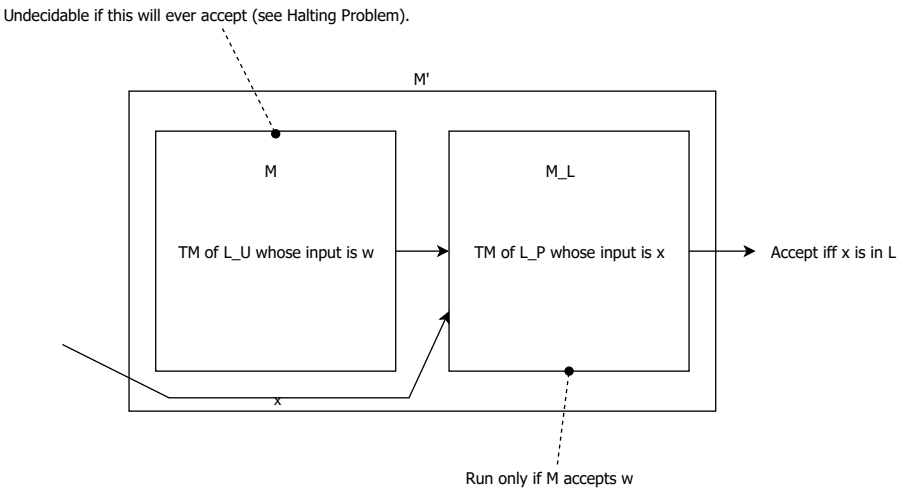


Figure 26: Here we see a schematic of the TM M' in which the language of M_L is the language L_P who can only run once if w is accepted by M whose language is L_U. As a whole this can be regarded as a reduction from L_U to L_P. Since we know L_U is undeciable then so is L_P. In essence it is bootstrapped by L_U undecidability.

$x$and therefore accepts $x$if and only if $x$is in L. Formally, $L(M') = L(M_L) = L$, the language of M' is the language of M_L, $L(M')$ has property P, and M' is in L_P. Now suppose the other case, M does not accpet (never halts) w, then M' never starts the simulation of M_L and thus never accpets the input x, so $L(M') = L(M_L) = \emptyset$. Therefore, $L(M')$ does not have the property P. M' is not in L_P. This can viewed graphically in Figure 26.

21.2 Post's Correspondence Problem

Post's Correspondence Problem (PCP) will be a tool for us to use in developing proving other problems are undecidable so its worth the effort of understanding. PCP asks: Is there some some ordering of indicies $i_1 \dots i_k$ for a list of coore-sponding strings $(w_1, x_1), (w_2, x_2), \dots, (w_n, x_n)$ such that $w_i \dots w_n = x_i \dots x_n$. A restriction we place on this question is that neither strings of w_i nor x_i can be the empty string, and they share the same alphabet. In less formal terms, PCP asks is there a way of concatenating w-type strings with x-type strings so that they beome equivalent strings.

21.3 PCP is Undecidable

The proof that PCP is undeciable relies on two reductions. The first reduction is from L_U, the language of the Univeral Turing Machine, to $MPCP$. $MPCP$ can be regarded as a modified version of the PCP problem such that the first pair

must be chosen first when forming the match on w and x. Then next reduction is from $MPCP$ to PCP.

21.3.1 Reduction from L_U to MPCP

//TODO

21.3.2 Reduction from MPCP to PCP

//TODO

21.4 Some Real Problems

//TODO

Part III
Complexity

22 Time Complexity (Bachmann-Landau Notation)

23 Space Complexity

24 Intractable Problems

In this section we learn about Time-Bounded Turing Machines, The Polynomial Class of Problems, P, and the Exponenetial Class of Problems, NP. Lastly we learn about Polynomial-Time Reductions. Refer to Michael Garey and David Johnson's *Computers and Intractability: A Guide to the Theory of NP-Completeness* as a primary source for this subject matter.

24.1 Time Bounded Turing Machines

We say a Turing machine is $T(n)$ time bounded, where$T(n)$ is some function of n, like n^2 or 2^n, if given an input of length n, the machine always halts in at most $T(n)$ steps.

24.2 The Partial Order of Equivalance Classes P, NP, and NP-complete

A Turing machine M is said to be polynomial-time bounded if it is time bounded by any polynomial. It could be linear, or quadratic, or cubic, or n^{1000}, as long as it is some polynomial. The languages who are said to be bounded by polynomial

time make up the Polynomial class P. For more complex polynomials you may wonder if they qualify for the class P. A problem qualifies to be in P if it is less than some polynomial. For example, $O(nlogn)$ is less than $O(n^2)$, and therefore it qualifies for the class P. The nondeterminitic polynomial NP class of problems is defined in term of a NTM which has a polynomial time nondeterministic algorithm. The algorithm should be composed of two seperate states. Th first is the guessing stage, and the second is the checking stage. A nondeterministic algorithm "solves" a descisions problem Π if the followung two properties hold for all instances.

1. If the instance of the problem is an accepting instance, then there exists *some* structure S that, when guessed for the input of the instance, will lead to the checking stage to *accept* for the instance and S.

2. If the instance of the problem is an rejecting instance, then there exists *no* structure S that, when guessed for the input of the instance, will lead to the checking stage to *reject* for the instance and S.

If there is a polynomial bound on the number steps an NTM must take in any branch of its computation then the problem is in NP. The term "complete problem" for class of problems means that the problem embodies the essence of all other problems in the class. Even though it may appear that the problem doesn't. PCP embodies Turing Machine computation. and the Knapsack Problem embodies all NTM computation. Polynomial Time reductions allow us to define NP-Complete problems. In order to show a problem M to be NP-complete, we have to show that every problem in NP is somehow embedded in L. We need a transformation from every problem in NP to M, and this transformation has to be sufficiently fast that if we had a deterministic polynomial time algorithm for M, then we could use it to build a deterministic polynomial time algorithm for each problem in NP. Formally, we say a problem or language M is NP-complete if, first of all, it is in NP, and for every language L in NP, there is a polytime reduction from L to M.

A final point on producing creating nondeterministic algorithms for NP problems concerns the status of the algorithms complement. There is a lack of symmetry the accepting and rejecting instances. *In a deterministic algorithm*, for a problem "Given an instance of the problem I, is X true for I", where X is some property, both the problem and its complement can be solved polynomially because a deterministic algorithm always halts. To obtain the complent you would just interchange the accepting and rejecting results. *In a nondeterministic algorithm*, for example the complement of the TSP problem–this is not the case. Consider the complement of TSP: "Given a set of cities, intercity distances, and a bound B, is it true that *no* tour of all the citeis has length Bor less? There is no know way to verify a "yes" answer to this problem other than computing all possible tours (an exponential algorithm). Thus membership P for a given problem implies membership in coP. However membership in NP does not imply membership in coNP. The prefix "co" implies the complement of this language class. A tentative view of the world of NP can be seen in Figure 27.

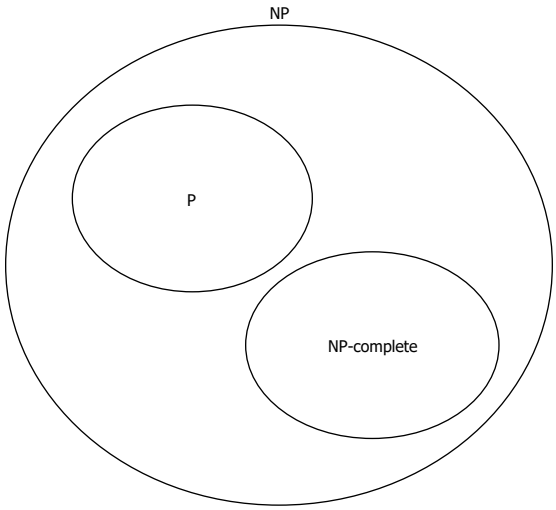


Figure 27: This diagram gives a tentative view of the membership relationship among the different language classes.

24.3 Encoding Schemes

It is important to note the effect an encoding scheme can have on an algorithm. If the encoding scheme is poorly constructed the input length might become exponential and thus violate one of the rules of prooving NP-completeness. An example of reasonable encoding schemes can be seen in in Figure28. Similary, if the output of our polynomial reduction becomes is not described in a polynomial length of the input then it too violates the rules of prooving NP-completeness. An example of this is a variant of the Traveling Salesman Problem (TSP) problem. This variant gives a bounds B and asks for all tours of length B or less yielding exponentially many tours less than B**.**

24.4 Transitivity of NP-completeness

Polynomial transformations provide membership for languages to equivalence classes with the following Lemma. Visually this can be seen in the diagram for Figure 26 on page 49.

Lemma 1. *If* $L_1 \propto L_2$ *and* $L_1 \epsilon$P, *then* $L_2 \epsilon$P. *Equivalently, If* $L_1 \propto L_2$ *and* $L_1 \notin$ P, *then* $L_2 \notin$ P.

Proof. Let $\Sigma_1$and Σ_2 be the alphabets of languages L_1 and L_2 respectively, and let the function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ be a *polytime* function from L_1 to L_2. Let M_f be a determinitic *polytime* DTM recoginizer computing the function f. Let M_2 be a *polytime* DTM recognzier for L_2. A *polytime* recognizer for L_1 can be constructed by composing M_f and M_2. For an input $x \epsilon$ L_1 we first apply the part of the program attributed to M_f which yields $f(x) \epsilon \Sigma_2$ (a string of the

Encoding Schemes	String	Length
Vertex list, Edge list	V[1]V[2]V[3]V[4](V[1]V[2])(V[2]V[3])	36
Adjacency list	(V[2])(V[1]V[3])(V[2])()	24
Adjacency matrix	0100/1010/0010/0000	19

Lower Bound	Upper Bound
$4v + 10e$	$4v + 10e + (v + 2e) \cdot \lceil \log_{10} v \rceil$
$2v + 8e$	$2v + 8e + 2e \cdot \lceil \log_{10} v \rceil$
$v^2 + v - 1$	$v^2 + v - 1$

Figure 28: The table above demonstrates how a graph's description can be described. The string length is given by length column. For sparse graphs an adjaceny list representation is beneficial. Dense graphs should be represented in an adjacency matrix. The bounds describe the input length in terms of v and e, for $G(V, E)$ where $v = |V|$ and $e = |E|$. Graphs encoded in any of these schemes will differ at most polynomially for any instance of a graph.

L_2's alphabet). We the apply the recognizer M_2 to the program to determine if the string $f(x)\ \epsilon L_2$. Since $x \epsilon L_1$ if and only if $f(x)\ \epsilon L_2$ we now have a recognizer for L_1. The fact that this yields a polynomial algorithm is derived from the fact that both M_f and M_2 are polytime algorithms themselves. □

Reductions have the feature of transitivity.

Lemma 2. *That is, if $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$.*

Proof. Let $\Sigma_1, \Sigma_2, \Sigma_3$ be the alphabets for L_1, L_2 and L_3 respectively. The function mapping the language of L_1 to the language of L_2 is given by $f_1 : \Sigma_1^* \to \Sigma_2^*$, and the function mapping the language of L_2 to the language of L_3 is $f_2 : \Sigma_2^* \to \Sigma_3^*$. Then the function transforming the language of L_1 to the language of L_3 is the composition of the given functions, $f(x) = f_2(f_1(x))$, for all strings $x$in L_1. Clearly, $f(x) \epsilon L_3$ if and only if $x \epsilon L_1$. The fact that f can be computed polynomially is derived from Lemma 1 on the preceding page. □

An import lemma that allows us to constuct the NP-complete equivalence class is the application of reduction transitivity to NP-complete problems.

Lemma 3. *If L_1 and L_2 belong to NP, L_1 is NP-complete, and $L_1 \propto L_2$, then L_2 is NP-complete.*

Proof. Since $L_2\ \epsilon NP$ all we need to do is show that, for every $L'\ \epsilon\ NP$, $L' \propto L_2$. Consider any $L'\ \epsilon\ NP$. Since L_1 is NP-complete, then surely $L' \propto L_1$ by the definition of NP-completeness. The tranistivity of $\propto$ seen in Lemma 2, and the fact that $L_1 \propto L_2$ implies that $L' \propto L_2$.

So it clear that once we have a single NP-complete problem to stand in for L_1 it serves as a means to bootstrap the process of creating the NP-complete equivalence class. The first NP-complete problem that allows the process was provided by Steve Cook and is discussed in Section 24.7 on the following page. □

24.5 Prooving NP-completeness

In order to prove that a problem is NP-complete you will need to demonstrate the following:

1. Show that the problem is in NP by giving a nondeterministic polytime decider.
2. Select an known NP-complete problem.
3. Construct a transformation function f from the NP-completer problem to the problem in question.
4. Prove that the tranformation function f is a polynomial tranformaton.

Garey and Johnson suggest that three design patterns exist for proving NP-completeness: restriction, local replacement and component design. Descriptions of the three patters are given.

24.5.1 Restriction

24.5.2 Local Replacement

24.5.3 Component Design

24.6 Polynomial Time Reductions

Polynomial time transducers are TM's that take in an input of length n, operate deterministically for some polynomial time $p(n)$. Produces an output on a separate output tape, and the output must me at most $p(n)$. Consider two languages or problems, say L and M. We say L is polytime-reducible to M if there is a polytime transducer T that takes an input w that is an instance of L, produces output x that is an instance of M, and the answer to L on w is the same as the answer to M on x. That is, w is in L if and only if x is in M.

Steve Cook and Dick Karp were the pioneers of NP-Completeness theory. Cook concentrated on 3-SAT – the question of whether an expression of propositional logic was satisfiable, that is, made true by some assignment of truth values to the propositional variables. But shortly after Cook wrote his original paper on NP-completeness, Dick Karp wrote another paper that showed many of the classical problems that had been puzzling mathematicians, sometimes for centuries, were NP-complete. Karp used only polytime reductions to the problem Cook had proved NP-complete. Since then, it is generally accepted that the preferred definition of NP-completeness is the one we gave here – the existence of polytime reductions. To make the distinction, this notion of NP-completeness is often called Karp-completeness.

24.7 Cook's Theorem

Recall from the Section 1.1 on page 7 on propositional logic that we expresses the truth of expressions with boolean operators. Expressions are built from

two components variables and constants using boolean the operators $(\wedge, \vee, \neg)$. Constants and the value of variables are either true (1) or false (0). The order of precedence for the boolean operatios is NOT, AND, OR. The Satisfiability Problem (SAT) is concened with determinig if there is a satisfying assignment of boolean values to an expression such that the expression is true. Instances of the SAT problem are represented with the parenthesis, logical operators, and variables x_i where i is the i^{th} binary integer.

In order to prove that SAT is NP-Complete we must first show that it is in NP. This can be done by giving an NTM that decides if an instance of SAT of length $n$is satisfiable. The algorithm for this Turing Machine can use nondeterminisc guessing to accomplish the goal. First the NTM guesses truth values for the variables of the expression. The NTM then checks the truth values of the expression. If the expression is true, then accept, otherwise reject. We now know that SAT is in NP.

The next step is to prove that SAT is NP-complete. Refer to Garey and Johnson pg 39-44 for the indepth proof. Prior to reading this you should understand Logic Programming `https://en.wikipedia.org/wiki/Logic_programming` and Lambda Calculus `https://en.wikipedia.org/wiki/Lambda_calculus`. At a high level the transformation function that they provide is based on the idea that the intermediate descriptions of the Turing machine can be translated into logical programming constraints which represent the clauses of the satisfiability problem.

25 Specific NP-Complete Problems

Garey and Johnson provide six additional foundational NP-complete problems which they have found to be the most useful. The relationship between these problems can be seen in Figure 29 on page 57.

1. 3SAT asks the question: Is there a truth assignment for a set of clauses in which all clauses have 3 variables?

2. Three Dimmensional Matching asks the question: For the Cartesian product of three disjount sets X, Y, and Z, is there some subset of the product that has unique coordinates for all members of the set? Formally, $X \times Y \times Z$ = P, where the $|X| = |Y| = |Z| = q$, does there exists a matching $M \subseteq P$ such that for all members of M no coordiates of the triple (x_i, y_i, z_i) are duplicated in any other element of M, with $|M| = q$? That is, for M = $\{(x_i, y_i, z_i)$,(x_j, y_j, z_j),$\ldots$, $(x_q, y_q, z_q)\}$, $x_i \neq x_j$, $y_i \neq y_j$, and $z_i \neq z_j$.

3. Vertex Cover asks the question: Is there a vertex cover for a given graph G of size K or less? A vertex cover is a set of vertices of a graph who are incident on edges which whose vertices make up all the vertices of the graph. Formally, for a graph $G(V, E)$, is there a vertex cover $V' \subseteq V$ such that $|V'| \leq K$, and for each edge $\{u, v\}$ ϵ E either u or v belong to V', but not both?

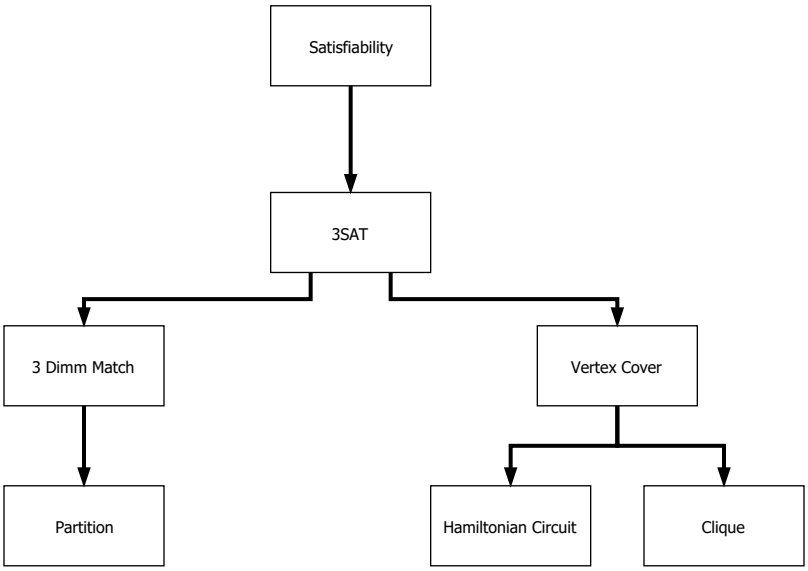


Figure 29: This diagram depicts the hiearchy of the NP-completeness proofs in this section.

4. Clique asks the question: For a given graph $G(V, E)$ is there a strongly connected component of size J or more in G? A strongly connected componenet is a set of vertices with a path between all pairs of vertices (i.e. a complete subgraph)

5. .Hamiltonian Circuit asks the question: Is there a cycle in the graph in which all vertices are visited only once? Formally, for a given graph $G(V, E)$, with $V = \{v_1, v_2, \ldots v_n\}$, where n = $|V|$. Is there an ordering of the members of V such that $\{v_1, v_n\} \epsilon E$ and $\{v_i, v_{i+1}\} \epsilon V$ for all i, $1 \leq i < n$?

6. Partition asks the question: Is there some way to evenly dividing a set of elements such that the sum of their values is equivalent? Formally, for a given set of elements $A = \{a_1, a_2, \ldots a_n\}$, where $weight(a_i)\ \epsilon \mathbb{Z}^+$, is there a set $A' \subseteq A$ such that:

$$\sum_{a \epsilon A'} weight(a) = \sum_{a \epsilon A - A'} weight(a)$$

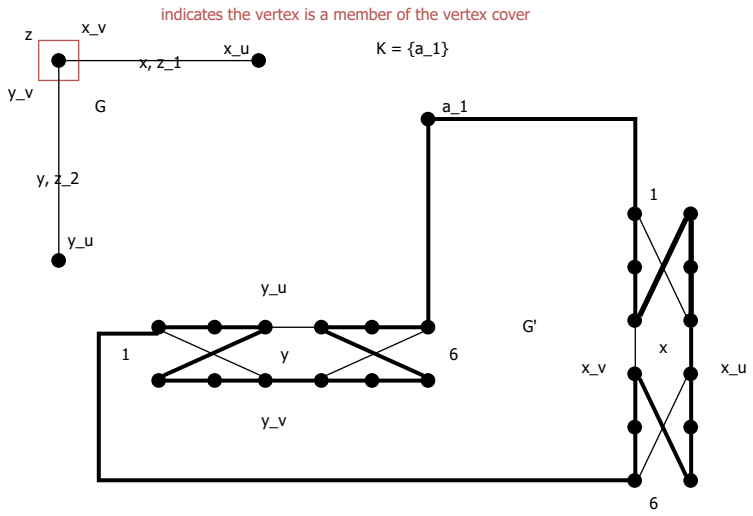


Figure 30: A polynomial time mapping reduction from a graph G with a vertex cover of size $|K|$ allows us to generate a graph G' which has a Hamiltonian circuit. Here $K = \{a_1\}$.

25.1 SAT to 3SAT

25.2 3SAT to 3DIMM Matching

25.3 3DIMM Matching to Partition

25.4 3SAT to Vertex Cover

25.5 Vertex Cover to Hamiltonian Circuit

For the given graph G, the graph G' can be generated to demonstrate the equivalence of VC and HC. You can view both of these graphs in Figure 30 on the next page.

Part IV
Algorithms

35.1 Software

- Draw.io is a online tool for creating all sorts of diagrams. It will be useful when creating automata.`https://www.draw.io`
- LyX is a LaTeX word processing application that will be useful when drafting homework.`https://www.lyx.org`
- Python is a useful programming language for experimenting with algorithms. `https://www.python.org`
- Mendeley is a good tool to organize your electronic library of books and journal articles. `https://www.mendeley.com/`
- Mathematica or MatLab to help your with linear programming `https://software.oit.gatech.edu`

35.2 Books

Even if you buy all these books it will not make you any better. I strongly suggest you acquire these for free and purchase the ones you find useful at a later date.

- *How to Prove It: A Structured Approach* by Daniel J. Velleman `https://amzn.com/0521675995`
- *Mathematical Reasoning: Writing and Proof* by Ted Sundstrom `https://amzn.com/1500143413`
- *Book of Proof* by Richard Hammack `https://amzn.com/0989472108`
- *Understanding Analysis* by Stephen Abbot `https://amzn.com/1493927116`
- *Introduction to Probability* by Dimitri P. Bertsekas and John N. Tsitsiklis `https://amzn.com/188652923X`
- *Discrete Mathematics and Its Applications* by Kenneth Rosen `https://amzn.com/0073383090`
- *Introduction to the Theory of Computation* by Michael Sipser `https://amzn.com/113318779X`
- *Introduction to Automata Theory, Languages and Computation* by John Hopcroft, Rajeev Motwani and Jeffery Ullman `https://amzn.com/0321455363`
- *Computers and Intractability: A Guide to the Theory of NP-Completeness* by Michael Garey and David Johnson `https://amzn.com/0716710455`
- *Algorithm Design* by Jon Kleinberg and Eva Tardos `https://amzn.com/0321295358`

- *Algorithms* by Sanjoy Dasgupta, Christos Papadimitriou, and Vijay Vazirani https://amzn.com/0073523402
- *Introduction to Algorithms* by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein https://amzn.com/0262033844
- *Approximation Algorithms* by Vijay Vazirani https://amzn.com/3540653678
- *Randomized Algorithms* by Rajeev Motwani and Raghavan Prabhakar https://amzn.com/0521474655
- *Linear Algebra and Its Applications* by David C. Lay https://amzn.com/032198238X
- *Linear and Nonlinear Programming* by David Luenberger and Ye Yingyu https://amzn.com/3319188410

35.3 Videos

- Professor Harry Porter's YouTube playlist on the theory of computation. https://www.youtube.com/playlist?list=PLbtzT1TYeoMjNOGEiaRmm_vMIwUAidnQz
- Professor Tim Roughgardener's playlist on Logic https://www.youtube.com/playlist?list=PLLH73N9cB21Xsgy39DP3xDqBN3lTaR8R5
- Professor Tim Roughgardener's playlist on Data Structures and Algorithms I https://www.youtube.com/playlist?list=PLLH73N9cB21W1TZ6zz1dLkyIm50HylGyg
- Professor Tim Roughgardener's playlist on Data Structures and Algorithms II https://www.youtube.com/playlist?list=PLLH73N9cB21VPj3H2xwTTye5TC8-UniA2