Contents

1 Overview

Harder computational problems (Printer Scheduling, Sudoku, Coloring) can be solved by solving their logical encodings — convert the problem to a logical formulation with boolean variables in disjunction (proposition) or variables in graphs (CSP).

Example problems solved by logical encodings

- Syntax, checking for a well-formed program
- Type system, checking for a well-typed program
- Semantics, what happens when running a program
- Specification, check what the program is supposed to do
- Verification, prove that the program will never crash and meets spec
- Complexity, time/space complexity

Automata

- Study of idealized computing machines
- How they work, can do
- Proving properties

Formal language theory

- Study of sets of strings/words
- Related to automata theory
- Mapping grammars to automatas

Mathematic vocabulary

- Natural language is ambiguous, must use maths/logic vocab
- Used in definition and proofs

Logic symbols

- $\land$ AND, conjunction
- $\lor$ OR, disjunction
- $\implies$ Implies, implication
- $\neg$ NOT, negation

A proposition/boolean encoding is to convert the program into a proposition statement of boolean variables.

2 Formal Propositional Logic

Propositional logic is equivalent to boolean logic with boolean variables. Boolean logic models logic algebraically similar to arithmetic.

Propositional formulas are built from

- Atoms, P, Q, R, P_1 which are variables that are true or false
- Bottom, $\bot$ which is false
- Top, $\top$ which is true
- Negation, $\neg P$
- Conjunction, $P \land Q$
- Disjunction, $P \lor Q$
- Implication, $P \to Q$
- With the bnf language syntax defined that a formula φ being one of the above building blocks, with recursion in operands.

A (strict) well-formed-formula (wff) in logical proposition places explicit brackets around each infix/prefix operators, ignoring the convention of omitting parenthesis and operator precedence.

The IFF operand is defined by: $(P \leftrightarrow Q) \equiv (P \to Q) \land (Q \to P)$.

Conventions in writing propositional formula

- Omit outer parenthesis unless precedence is wrong
- Negation has highest binding power (highest precedence). Implication has weakest binding power (lowest precedence).
- Precedence from weak to strong: implication, and/or, negation
- Imply is right associative, so $P \to Q \to R$ equivalent to $P \to (Q \to R)$
- AND/OR combined have ambigious precedence

Truth functions

- A function from truth values to truth values
- Truth values are boolean true and false
- Semantics of a truth function defined in a truth table using boolean variables
- A propositional formula is a truth function. We can view its subtree truth values under a truth table by writing columns of truth values under each atoms/operands.

Truth assignments

- Propositional letters are boolean variables
- Truth assignment is a function from letters to truth values. In our case the assignments are explicit and complete
- Notation is

$$\begin{aligned} v &= \{P \to 1, Q \to 0\} \\ v(P) &= 1 \\ v(Q) &= 0 \end{aligned}$$

- A formula will have a truth value under a truth assignment v.

- The shorthand $v \models P$ means P is true under the truth assignment v. Equivalently, we say that v is a model of (or models) P.

Logical equivalence

- Formulas are equivalent iff they have identical truth values under every possible truth assignment. Equivalently if they have the same truth tables.
- Note that this means two formulas don't need the same variables to be equivalent if the assignment to that variable doesn't affect the equivalence.
- Shortand $P \equiv Q$ means that P is logically equivalent to Q

Implication is weird since it doesn't mean causation. Stick to the logical definition that

$$(P \to Q) \equiv (\neg P \vee Q)$$

Modus Ponens

- A argument to deduce a statement. The form of: $P \to Q$ (premise) is true, P (premise) is true, so Q (conclusion) is true
- A modus ponens argument is sound if: when conclusion Q is true, the premise P is also true (essentially the implication premise $P \to Q$ is true). Alternatively, when both the argument and premise are true the argument is sound.

Premise is a propositional statement that is true. Conclusion is a propositional statement that is inferred true from the premise using deductions.

3 Semantic consequence

Semantic consequence

- For propositional statements G and F, the statement G is a semantic consequence of F ($F \models G$) if and only if every model of F is also a model of G.
- If $F \models G$, F (semantically) entails G
- Can treat the ordering as implication ($F \models G, F \to G$)
- Used to relate two proposition statements where one is stronger than the other

Note that

$$A, B \models F$$

is the same as

$$A \wedge B \models F$$

Semantic consequence and equivalence

$$F \equiv G \iff F \models G \wedge G \models F$$

Semantic consequence and implication

- $$F \models G \iff \models F \to G$$

  , if F entails G, then $F \to G$ is valid
- Importantly if the LHS of $\models$ is missing, any truth assignment is a model of the LHS, so the RHS is true under all models (valid)
- Collorary of
  $$F \equiv G \iff \models F \leftrightarrow G$$

4 Satisfiability

Terms

- Tautology is a logical formula that is always true
- Contradiction is a logical formula that is always false. Negating a contradiction nets a tautology, vice versa.
- Valid statements are always true (tautology). Non-valid statements can sometimes be false. $\models F$ means F is valid by definition
- Unsatistifiable statements are always false (contradiction). Satistifiable statements can sometimes be true.
- Contingent statements can sometimes be true, sometimes be false.

Substitution

- The action of replacing all uses of a propositional variable (say P) with a given formula (say $Q \wedge R$)
- Substitution preserves the validity of the formula, and also preserves the unsatisfiability of it.
- Substitution does not preserve satisfiability, since we can replace the variable with a formula that is never true.
- Denote $F[A := B]$ as sub A with B in F.

Substitution preserves logical equivalence. Given F, G, H formulas and any propositional variable P. If $F \equiv G$
$$F[P := H] \equiv G[P := H]$$

Interchange of equivalence. If $F \equiv G$, we can replace F with G anywhere keeping all semantic properties. Formually, if $F \equiv G$
$$H[F := G] \equiv H$$

Equivalence formats

- Absorption

$$P \wedge P \equiv P$$
$$P \vee P \equiv P$$

- Communtativity

$$P \land Q \equiv Q \land P$$
$$P \lor Q \equiv Q \lor P$$

- Associativity of $\lor$ and $\land$
- Distributivity

$$P \land (Q \lor R) \equiv (P \land Q) \lor (P \land R)$$

- $\leftrightarrow$ is commutative and associative
- Double negation

$$P \equiv \neg\neg P$$

- De morgan

$$\neg(P \land Q) \equiv \neg P \lor \neg Q$$

- Implication

$$P \to Q \equiv \neg P \lor Q$$

- Contraposition

$$\neg P \to \neg Q \equiv Q \to P$$

- Biimplication

$$P \leftrightarrow Q \equiv (P \land Q) \lor (\neg P \land \neg Q)$$

- Duality

$$\neg\top \equiv \bot$$

- Negation from absurdity

$$P \to \bot \equiv \neg P$$

- Identity

$$P \lor \bot \equiv P$$

- Dominance

$$P \land \bot \equiv \bot$$

- Contradiction

$$P \land \neg P \equiv \bot$$

- Excluded middle

$$P \lor \neg P \equiv \top$$

5 Resolution

Propositional logic is decidable

- it is always possible to check if a formula is valid/contingent/unsat using a truth table
- this takes finite 2^n checks with is slow
- there are more efficient methods of checking

Either-or reasoning

- $(P \to F) \land (\neg P \to G) \models F \lor G$ for any formula F, G and proposition P
- This is a sound reasoning statement since the deduction obviously holds.

Resolution

- Resolution is to apply either-or reasoning repeatly on a CNF to net several semantic consequences
- Resolution aims to solve the RHS of $A, B, C \models X$, finding the logical formula that is a semantic consequence of the premises
- For multiple premises, select (any) two that either-or reasoning can apply to and replace them to their semantic consequences.
- Can make a resolution graph by linking semantic consequences nodes to their premises

Refutation

- If resolution shows that $A, B \models \bot$, the premises is a contradiction, and it is called a refutation
- This is because a semantic consequence of $\bot$ means no model sats premises
- To show that F is valid, refute $\neg F$ by showing $\neg F \models \bot$
- To show that $F \models G$ (alt $F \to G$ is valid), refute $\neg(\neg F \lor G) \equiv F \land \neg G \models \bot$

Failed refutation

- If resolution cannot get a refutation and every possible semantic consequence is derived, the formula must be satisfiable
- The truth assignment that sats the formula can be read off by the single literal disjunction entailed in the resolution: the consequence of $P, \neg Q$ means that $v = \{P \to \top, Q \to \bot\} \models F$ will sat the formula

Cancelling variables

- Either-or reasoning will cancel/remove one variable if it is opposite in two premises
- Do not cancel multiple variables at once, like

  $$P \lor Q, \neg P \lor \neg Q$$

  this is not sound, as in that this argument is not correct

Formal resolution proof

- A resolution proof of C_m from wff $P_1, \dots, P_n$ is a string

$$P_1, \dots, P_n \vdash C_1, \dots, C_m$$

  where either C_i follows by resolution from any two wff earlier in the string
- It is essentially proving C_m using a series of intermediate proofs involving C_i

Soundness

- Writing $\Sigma \vdash F$ means that there is a resolution proof of F from the premises Σ
- The $\vdash$ means deducing/follows
- The soundness theorem is that

$$\Sigma \vdash F \implies \Sigma \models F$$

  from definition of resolution proofs using induction on C_i

6 CNF

An observation is that resolution is easier with premises in CNF.

Conjunctive Normal Form

- Literal is a proposition atom (variable) or its negation
- Disjunctive clause is a disjunction of literals
- CNF is a conjunction of disjunctive clauses
- Examples

$$(A \vee \neg B) \wedge C$$

- Theorem, every formula has an equivalent CNF

Negation normal form

- NNF means that connectives are $\wedge, \vee$ with negations only in front of variables
- We can get NNF from any formula:
- Eliminate $\leftrightarrow$ with $\wedge$ and $\rightarrow$
- Eliminate $\rightarrow$ with $\vee$ and $\neg$
- Push $\neg$ inwards using demorgan's law
- Eliminate $\neg\neg$

To convert from NNF to CNF, we only need to distribute $\vee$ over $\wedge$ using

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

Resolution is refutation-complete

- We already know that resolution can deduce refutation implying unsatisfiability
- Theorem, every unsatisfiable formula can be refutated using resolution

- Formally, given clauses Σ, if $\Sigma \models \bot$ (it is unsat), then the $\Sigma \vdash \bot$ resolution proof must exist.

Satisfiability algorithm

- Convert formula into CNF
- Apply resolution repeatly. If derives $\bot$, unsat. Otherwise when can't derive new formulas, sat

Simplifying CNF

- Duplicated variables can be replaced with only one, $P \lor P \equiv P \land P \equiv P$
- Tautologies can be removed completely, $P \lor \neg P \lor Q \equiv \top$
- Subsumptions can be removed. For formula P, if $(P \lor Q) \land P$, can remove P to net $Q \land P$
- Note that CNF for a formula is not unique

Clausal form of CNF

- Can write CNF as set of sets of literals: outer set for conjunction, inner set for disjunction
- For disjunction: $\{A, B\}$ represents $A \lor B$, $\{A\}$ represents A, $\{\}$ represents $\bot$ (since true iff at least one literal is true)
- For conjunction: $\{A, B\}$ represents $A \land B$, $\{A\}$ represents A, $\{\}$ represents $\top$ (sicne true iff all literals are true)
- Empty conjunction is valid, empty disjunction is unsat. Therefore, empty set of clauses is valid, set of empty clauses is unsat.
- To convert CNF to clausal form, replace $\lor$ in each conjunct with $\cup$, and literals P with $\{P\}$

7 Predicate Logic

Propositional logic can encode some categories of problems where variables are either true/false, and our goal is to check whether a query is satisfiability (or valid).

- However, some statements cannot be expressed adequately (nicely) in propositional logic. In particular when the problem contains:
- Infinite collections (every X)
- Transitive/relationship verbs (Loves)
- Relative pronouns (it, whoses)

Example predicate logic translations

- No emus fly
$$\forall x(Emu(x) \to \neg Fly(x))$$
- There are black swans
$$\exists x(Black(x) \land Swan(x))$$

- If all pushes the cart, then the donkey is happy

$$(\forall x Push(x, c)) \rightarrow Happy(donkey)$$

- If anyone pushes the cart, the donkey is happy

$$(\exists x Push(x, c)) \rightarrow Happy(donkey)$$

- Tina found Rover and returned him to Anne

$$Found(tina, rover) \wedge Gave(tina, rover, anne)$$

- Tina found a dog and give it to Anne

$$\exists x(Dog(x) \wedge Found(tina, x) \wedge Gave(tina, x, anne))$$

- Mother's mothers are grandmothers

$$\forall x \forall y \forall z((Mother(x, y) \wedge Mother(y, z)) \rightarrow Grandmother(x, z))$$

- No sorting algorithm can guarantee sorting with fewer than $n \log n$ comparisons

$$\forall x(S(x) \rightarrow \neg(c(x) < n \log n))$$

Predicate logic new symbols

- Constants. Often nouns represented as a lowercase letter. Begin of alphabet
- Predicates. Often verbs that take arguments with captialized names. Predicates with no arguments are propositional variables. Predicates must be truthy or falsy. Equality is a predicate with two arguments
- Quantifiers. $\forall$ forall and $\exists$ exists
- Variables. Lowercase letters used in quantifiers. End of alphabet. Its meaning can change compared to constants
- Functions. Lowecased names, taking constants as arguments. Forms a hierarchy on constants.

Existential quantification

- Generalization of disjunction
- Example

$$P(x_1) \vee P(x_2) \vee P(x_3) \leftrightarrow \exists x P(x)$$

Universal quantification

- Generalization of conjunction
- Example

$$P(x_1) \wedge P(x_2) \wedge P(x_3) \leftrightarrow \forall x P(x)$$

Quantifiers

- The same quantifier is commutative.
- Different quantifiers are NOT commutative

Terms and atomic-formulas

- Terms are individual objects that are not formulas. They can be: constants, variables, functions, numbers. Terms are not logical formulas and don't exist in propositional logic.
- Atomic formulas represents boolean statements on their argument objects. It is a single predicate applied to some terms. They are formulas
- Convention: terms/functions written with lowercase letters, atomic-formula/predicates written with uppercase letters

Predicate logic statement formal definition

- Terms are built from variables, constants, and functions
$$t ::= x|c|f(t,\dots)$$
- Formula are built from predicates, terms, connectives, and quantifiers
$$\phi ::= P|P(t,\dots)|\bot|\top|(\neg\phi)|(\phi\wedge\phi)|(\phi\vee\phi)|(\phi\rightarrow\phi)|(\forall x\phi)|(\exists x\phi)$$
- Note that $\forall, \exists$ have the same precedence as $\neg$. Should omit parenthesis if we can

Predicates vs functions

- Connectives: proposition inputs, proposition outputs
- Predicates: object inputs, proposition outputs
- Functions: object inputs, object outputs

Quantifier scope

- The subformula attached to a quantifier is its scope
- The variable of the quantifier is bounded in the subformula (in its scope). A quantifier with variable x binds x within its scope.
- If a variable is not bounded, it is free
- Bound variables can be renamed unless there is a clash after it is renamed. Renaming free variables changing the meaning

Converting english to predicate logic

- First identify nouns, verbs, pronouns, adjectives, relative clauses (parent of)
- Then assign: constants to singular nouns, predicates to verbs and adjectèves, functions to relative clauses, variables to indefinite pronouns
- Last, replicate logical structure of the sentence
- Be careful about the word ordering, they matter!
- Quantifiers are often implicit in english, look for (plural) nouns without determiners/constants ("Humans are mortal" means "All Humans")

8 Predicate Semantic Meaning

Formula meaning

- Meaning of a formula is about whether it is true or false
- Depends on the universal of objects (and interpretation) that we assign to the formula
- Predicate formulas can also be valid, contingent, or unsatisfiable (defined later)

Universe

- A set of objects U that we care about. Must be non-empty
- Quantifiers quantify over the universe

Interpretation function

- Non-propositional-logic symbols are: predicates, function, and variables. They don't correspond to anything in the universe without an interpretation function
- Interpretation function maps every non-logical symbol to its (interpretation) object in the universe U:
- Constant mapped to an object U
- Predicate mapped to a relation on U, mapping objects (or tuple of objects) of U to true or false
- Function mapped to a function from U to U

Models

- Model is a universe and interpretation function (that makes the formula true)
- A model for a formula implies that the formula is true under the model
- N with + is not a model for $\forall x \exists y (x + y = 1)$, since it is false under the model

Free variables

- Free variables are variables that are not quantified over
- Variable assignment is a function v that maps (free) variables to U.
- A formula with free variables may have its truthiness under model M depend on the variable assignment
- Given an interpretation I, there is an automatic variable assignment function v to even terms by natural extension

  $$\begin{aligned} v(a) &= I(a) \\ v(g(a,b)) &= I(g)(v(a), v(b)) = I(g)(I(a), I(b)) \end{aligned}$$

  Note that variables are excluded in this natural extension v
- Notation of $v_{x \to d}(y)$ means $v(y)$ except mapping x to d

Formula truth

- The truth of a closed formula depends only on the model. For a formula with free variables, it also depends on the variable assignment

- We are interested in variable assignments since it allows for a compositional approach to formula truth (recursive definition)
- In a model $M = (U, I)$ under variable assignment v:
- Atomic formula $P(t_1, \dots)$ is true iff $(v(t_1), \dots) \in I(P)$, essentially: convert predicate argument terms to objects, check if predicate with object is true under the interpretation
- Existential quantifier $\exists x F$ is true iff there is one $d \in U$ where F is true in M under $v_{x \to d}$
- Universal quantifier $\forall x F$ is true iff $\neg \exists x(\neg F)$ is true
- Connectives are true in the same way as under propositional logic

Quantifiers as two-player-games

- If I make a claim that a formula is true, I get to choose the existential variables to make it true and the opposite gets to choose the universal variable to make it not true
- The selection order is done outside in
- Can use this to argument quantifier equivalences

Quantifier equivalences for general F and G

$$\begin{aligned}
\forall x \forall y &\equiv \forall y \forall x \\
\exists x \exists y &\equiv \exists y \exists x \\
\neg \forall x F &\equiv \exists x(\neg F) \\
\neg \exists x F &\equiv \forall x(\neg F) \\
\exists x(F \lor G) &\equiv (\exists x F) \lor (\exists x G) \\
\forall x(F \land G) &\equiv (\forall x F) \land (\forall x G) \\
\exists x(F \to G) &\equiv (\forall x F) \to (\exists x G)
\end{aligned}$$

Quantifier equivalences if G has no free x and for any F

$$\begin{aligned}
\exists x G &\equiv G \\
\forall x G &\equiv G \\
\exists x(F \land G) &\equiv (\exists x F) \land G \\
\forall x(F \lor G) &\equiv (\forall x F) \lor G \\
\forall x(F \to G) &\equiv (\exists x F) \to G \\
\forall x(G \to F) &\equiv G \to (\forall x F)
\end{aligned}$$

Semantic consequence

- Write $M, v \models F$ to mean that F is true under the model M and variable assignment v.
- Write $M \models F$ to mean that $M, v \models F$ under all variable assignments (F is true in M, or M is a model of F)
- Write $F \models G$, if every model of F (models where F is true) is a model of G (F semantically entails G).
- Write $F \equiv G$, if $F \models G, G \models F$ (F equivalent to G)

Satisfiability

- Consider a closed formula F
- F is sat if $M \models F$ for some M
- F is valid if $M \models F$ for all M
- F is unsat if there are no M where $M \models F$
- F is non valid if there exists M where $M \not\models F$
- F is contingent if it is sat and not valid
- The same relationship between valid/unsat and non-valid/sat holds under $\neg$.

9 Resolution in Predicate Logic

Extending resolution to predicate logic

- Goal is to start with a predicate formula F, and then derive $\bot$
- CNF in predicate logic is a conjunction of disjunctions of literals, where all variables are assumed universally quantified
- Sadly, quantifiers means that not every predicate logic formula has a CNF. This means that we may give up equivalences when converting to CNF

Extended resolving to atomic formulas (variable, constants, functions)

- Universally quantified either-or on predicates

$$F(x) \vee G(x), \neg G(a) \models F(a)$$

- Fixing variables in universial quantifiers, namely $G(x) \models G(a)$

$$O(a,x), \neg O(y,b) \models O(a,b), \neg O(a,b), \bot$$

  Typically, we try to fix variables so that it matches constants in another similar disjunction
- In a resolution graph, write the variable assignment used when fixing variables for either-or reasoning
- When dropping forall, we can rename variables in different clauses.
- We can always duplicate a clause with another one with different variables

Eliminating existential quantifiers

- If it is in front, like $F : \exists x \forall y P(x,y)$, we can derive that $G : \forall y P(a,y)$ for a constant a, then G is satisfiable iff F is satisfiable. Note that this is not equivalence, and a must be selected (or assumed selected) to maximize the possibility that G sats.
- The a is new and is called a skolem constant.
- If it is nested, like $F : \forall y \exists x P(x,y)$, there can be different x for every y. We can derive that $G : \forall y P(f(y),y)$ is sat iff F is sat. This assumes that $f(y)$ maps each y to the value that may sat G.

- The function f is new and is called a skolem function.
- If it is nested under multiple universal quantifier, make a skolem function taking multiple arguments

Predicate logic to clausal form (CNF)

- Eliminate $\leftrightarrow$ and $\implies$
- Push negation in to NNF (treat universal as AND and existential as OR)
- Rename shadowed variables (no quantifier should use the same variable). Note that when forall a conjunctive clause, we can rename the forall variable within each conjunctive clauses
- Eliminate existential quantifier (skolemize)
- Drop universal quantifiers (directly drop them)
- Bring to CNF using distributive laws

10 Formal Unification and Resolution

Substitution

- A function from variables to terms
$$\theta = \{x_1 \to t_1, x_2 \to t_2\}$$
- Given formula E, the substitution using θ will replace all occurances of x_i with t_i, leading to
$$E[\theta]$$
- Only do one pass of the subtitution, and not recurse using the newly substituted formula

Unifiers

- A unifier of two expressions s and t is a substitution θ where
$$s[\theta] = t[\theta]$$
- Two formulas s and t are unifiable iff there exists a unifier θ for s and t.

Most general unifier (MGU)

- A MGU for s and t is a substitution θ where: θ is a unifier for s and t, every other unifier σ can be expressed as $\tau \circ \theta$ for some substitution τ $(\forall \sigma, \exists \tau(\sigma = \tau \circ \theta))$
- Define the substitution composition $\tau \circ \theta$ as applying θ then τ
- Essentially, MGU is the most general unifier that constrains as little variables as possible
- Theorem: If s and t are unifiable, there exists a MGU
- Note that terms must be finite: it cannot be a recursive definition or have infinite function applications

Syntactic Unification Algorithm

- Input: two expressions s and t
- Output: the MGU if it is unifiable, failure if it is not
- Algorithm:
  1. Start with the set containing $\{s = t\}$
  2. Perform one of the six actions, repeat
  3. Return result. If one assignment contains a variable assigned elsewhere, flatten/substitute the assignment for the final result.
- Actions:
  1. For functions or predicates F

     $$F(s_1, \dots) = F(t_1, \dots)$$

     replace it with $\{s_1 = t_1, \dots\}$
  2. For functions or predicates F and G

     $$F(s_1, \dots) = G(t_1, \dots)$$

     halt, failure
  3. $x = x$, delete the equation
  4. $t = x$, where t is not a variable (t is a constant), replace it with $\{x = t\}$
  5. $x = t$, where t is not x but x is in t, halt, failure
  6. $x = t$, where t contains no x, save $\{x = t\}$ for solution, also substitute x with t everywhere else.
- To apply the algorithm, each step is a set of equality equations, with actions represented as arrows between two sets of equations.
- The algorithm always halts
- If it fails, no unifier exists
- If it unifiers, the resulting unifier is in normal form: every LHS is a different variable, no LHS appears in any RHS
- The resulting unifier is always a MGU

Resolution using Unification

- Let P_1 and P_2 be unifiable atomic formulas with no common variables. Let θ be unifier
- Let C_1 and C_2 be disjunctive clauses, then

$$C_1 \wedge P_1, C_2 \wedge \neg P_2 \models (C_1 \vee C_2)[\theta]$$

- Basically, apply either-or using a MGU between the two complementory formulas
- The remaining resolution algorithm follows exactly like before

To prove that $A_1, \dots, A_n \models B$, we can refute

$$A_1 \wedge \dots A_n \wedge \neg B$$

using resolution.

Refutation proof shortcuts

- To save space, can omit curly braces of clauses and omit parenthesis around function/predicate arguments

The unification algorithm is exponential on the length of the formulas. (For an example, consider the replacement of one term with a function with two identical terms).

To apply unification to type inference in a programming language, convert each function call expression with arguments a and return value b to

$$f = a \to b = fun(a, b)$$

and try to unify all such expressions to derive the MGU on all unknown types.

11 Automatic Reasoning and Satisfiability

Factoring

- Consider clause C with unifiable subformulas A_1 and A_2. Given mgu θ, we can derive clause $C[\theta]$
- For example
$$\{P(x), P(y)\} \equiv \{P(x)\}$$
- Sometimes needed to yield a refutation in resolution

Formal resolution method

- Start with collection Σ set of clauses
- While $\perp \notin \Sigma$, do: add to Σ a factor of some $C \in \Sigma$, or a resolvent between some $C_1, C_2 \in \Sigma$.
- If MGU is used in resolution and factoring, the method is sound (correct) and refutation-complete (unsat formula iff resolution derives bottom in finite steps)

Resolution theorem

- Σ is unsat iff the resolution method can add $\perp$ after a finite number of steps
- Resolution process should be fair in unification by bfs instead of dfs
- Resolution is sound and refutation-complete
- It is not a decision procedure, as it may not terminate on finite formulas. No such procedures can ever exist for predicate logic.
- Validity and unsat are semi-decidable properties

Horn Clauses

- A horn clause is a clause with at most positive literal. This means that each clause can be written as implication

- They are rules in prolog
- For propositional horn clauses, sat can be decided in linear time. This simplifies the search in logic programs

Horn clause sat algorithm

- Maintain list containing $\bot$ and $\top$ in ϕ
- Mark $\top$ if it is in list
- while there is a horn cluase $P_1 \wedge ..P_n \to Q$ in ϕ where all P_i are marked and Q is unmarked, mark Q
- If $\bot$ is marked, then unsat. Otherwise, return sat
- The sat assignment $v(P)$ is true if it is marked and false if it is not.

Equality in automated reasoning

- Let $eq(X, Y)$ be a new predicate symbol
- Reflexivity, symmetry, transistivity, congurence

$$\begin{aligned} x &= x \\ x = y &\iff y = x \\ x = y \wedge y = z &\implies x = z \\ x = y &\implies f(x) = f(y) \\ x = y \wedge P(x) &\implies P(y) \end{aligned}$$

- Using this axiomatic definition of equality tends to be quite inefficient

Automated reasoning

- Main question, given a set of axioms $\{A_i\}$ and a conjecture C in some formal language, do the axioms logically imply the conjecture?
- Example
$$A_1, \ldots, A_n \models C$$

Expressivity and automatizability

- Expressivity is how nautrally computable problems are encoded
- Automatizability is finding a reasonably efficient sound and complete search

12 Models of Computation

Algorithm resources

- Time
- Space
- Randomness
- Quantum entanglement

- Input access

Algorithms and computation problems

- A computational problem is a specification: for every input x, what is the correct output. Represented as a predicate $P(x, y)$ which is true only when y is the correct output for the input x
- An algorithm is an implementation (often for a computational problem): for every input x, what are the steps to obtain an (correct) output y. Represented as a function $A(x) = y$.
- Sorting is a computational problem. Quicksort is an algorithm for the sorting problem
- An algorithm $A(x)$ solves a problem $P(x, y)$ iff it sats the problem for all inputs

$$\forall x, P(x, A(x))$$

Algorithm efficiency

- Can quantify the efficiency of an algorithm based on the resources it has access to.
- A computational model is the set of resource constraints
- Theoretical computer science concerns: power and limits of computational models, tradeoffs between resource usage between models

Simplified computational problems

- Inputs are bitstrings (boolean strings)
- Outputs are YES/NO (single bit). This means a decision problem
- All problems can be reduced to solving this bitstring decision problem
- A computational problem modeled by a set of all bitstrings that are YES. Also known as a language L.
- The language of an algorithm (represented as $L(A)$) is the set of bitstrings that A accepts. An algorithm A recognizes a language L' if $L(A) = L'$

13 Finite (state) automata

Key features

- Memoryless, fixed memory for all input sizes
- Memorylessness, behavior depends only on the current state and next input. Does not remember information about past states
- Stream, does not know when the input ends

Definition

- A finite state automata M for an alphabet Σ is a tuple $M = (Q, q_0, \delta, F, \Sigma)$
- Q is the set of states
- $q_0 \in Q$ is the initial state

- $\delta : Q \times \Sigma \to Q$ is a transition function
- $F \subseteq Q$ is a set of accept states
- Σ is a set of characters in the alphabet

State transition table

- Describes δ
- Each row describes: current state, transition if 0, transition if 1.
- To use the table, find row with current state, use transition rule depending on bitstring bit

State transition diagram

- Visual description δ
- Nodes are states, directed edges are transitions.
- Each edge labeled by 0 or 1 indicating the bit to cause this transition from the state.
- Double outlined states are accepted states

Bitstring decision algorithm

- Denote a set of states F as accept states.
- To run the automata on an input bitstring, repeatly apply δ on each bit of the input. If the final state is in F, the automata accepts the input. Otherwise, it rejects
- For example, given $x = v_0 v_1 \dots v_n$, the states are

$$r_0 = q_0$$
$$r_{i+1} = \delta(r_i, v_i)$$

- Define $q(w)$ to be the automata state after processing the string w
- The sequence $r_0 \to r_1 \cdots \to r_n$ is the computational path of M on input w. It will be a path in the state transition diagram (allowing repeated vertices).
- The language L recognized by an finite automata M is just the set of bitstrings x where $M(x)$. Or $L(M)$ where we treat M as a decision algorithm.

Program analogy

- Initialization, preinitialize set of variables (including isAccept), and transition function
- Input processing, while stream has characters, update variables using transition function
- When stream ends, return isAccept

14 Regular Languages

Regular language

- A language L is regular iff there exists a finite automaton that recognizes it (exists an M such that $L(M) = L$)

Language universe

- Given alphabet Σ, define Σ^* to be the set of all strings built from the alphabet of any length, including the empty string ϵ
- Any language L over the alphabet Σ is a subset of it $L \subseteq \Sigma^*$

String operations

- The concatenation between strings x and y is denoted as xy

Language operations

- Complement. L^c is the set of strings z in Σ^* not in L

$$L^c = \Sigma^* \setminus L$$

- Intersection. $A \cap B$ is the set of strings in both languages

$$A \cap B = \{z \colon z \in A \wedge z \in B\}$$

- Union. $A \cup B$ is the set of strings in either language

$$A \cup B = \{z \colon z \in A \vee z \in B\}$$

- Concatenation. $A \circ B$ is the set of strings z such that it is a concat of strings from A and B

$$A \circ B = \{z \colon z = xy \wedge x \in A \wedge y \in B\}$$

. Define L^k to be L concat with itself k times (or $\{\epsilon\}$ when $k = 0$)

- Kleen star/closure. L^* is the set of strings containing
  1. the empty string ϵ
  2. strings in L^k for every positive k

$$\begin{aligned} L^* &= \bigcup_{i=0}^{\infty} L^k \\ &= \{\epsilon\} \circ L \circ L \ldots \quad \text{assuming } L \text{ has the emptyset} \end{aligned}$$

(Define) Regular operations are: union, concat, kleen star.

Concat rules

- $L \circ \emptyset = \emptyset$
- $L \circ \{\epsilon\} = L$
- $A \subseteq A \circ B$ when $\epsilon \in B$
- Not commutative but is associative

Regular language closure

- Regular languages are closed under all language operations
- L regular implies that L^* and L^c are regular

- A, B regular implies that $A \cup B$ and $A \cap B$ and $A \circ B$ are regular
- Note that this is implies not iff
- This means that given a finite automaton for L, we can get a finite automaton for L^* and L^c. And if there is a finite automaton for A and B, there is also one for $A \cup B$, $A \cap B$ and $A \circ B$
- Proof of closure takes the automaton for L, and derives a new automaton for L under operation

15 NFA

Non-deterministic FA

- The same as deterministic FA except ...
- The transition function can return a subset of states 2^Q, and it can also make transitions without consuming the next input character (ϵ transitions)

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$$

- On transition diagram: subset states means multiple possible transition path per state, ϵ transition means that there may be directed edges marked with ϵ that doesn't consume the input when transitioning

NFA acceptance (for NFA N)

- A state q is a resulting state after applying w to N iff there exists a sequence of state transitions where q is the final state
- Denote $Q(w)$ as the set of all possible resulting states after applying input w to N
- N accepts a string w iff $Q(w)$ contains an accept state.
- The language recognized by N (denoted $L(N)$) is the set of strings that N accepts

Computation path (for NFA N)

- A computation path is the sequence of states of N while processing w
- The path is accepting if it ends in an accept state. The path is rejecting if it does not end in an accept state
- N accepts w iff there exists an accepting computation path

Theorems

- Every DFA is a NFA, not the reverse
- Every NFA can be converted into an equivalent DFA
- Can prove the closure of regular languages by creating a NFA for the resulting language using the DFAs

16 NFA Determination

Machine equivalence

- Two machines (like DFA/NFA) are equivalent if they recognizes the same language

NFA/DFA equivalence

- Every NFA has an equivalent DFA
- Informally, this is derived from simulating the NFA. The steps are
    1. Make the start state $q_0' = \{q_0, \dots\}$ including all states reached via ϵ transitions
    2. If 0, find all the states reached from q_0 (or ϵ transition start states) using 0 (or ϵ transitions). Call this A. Add transition from q_0' to A. Similarly for 1
    3. Repeat, for each state set A and character c, add a transition to B under c where every character in B can be reached by a state in A following c and ϵ transitions
    4. The starting state is q_0'. The accept states are all state sets R such that it contains at least one accept state.
    5. State sets that are empty are dead states.

NFA to DFA algorithm

- Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA
- For all $R \subseteq Q$, define $E(R)$ be the ϵ closure of R. An element is in $E(R)$ if it is reached from a state in R following 0 or more ϵ transitions
- Consider $M = (Q', \Sigma, \delta', q_0', F')$ be a DFA where:
- $Q' = P(Q)$
- $q_0' = E(\{q_0\})$
- $F' = \{R \in Q' | \exists c \in R, c \in F\}$
- $\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$
- Then the DFA M is equivalent to NFA N

Closure proofs

- Proof idea: Suppose that A and B are regular languages. There exists N_1 and N_2 DFA that recognizes them. Construct NFA M that recognizes the combination of A and B. Determinize M into equivalent N_3 DFA. Then the combination is regular since N_3 recognizes it
- Formal proof. Define the two DFAs as tuples. Create NFA using functions of the two DFAs. Show that the NFA recognizes the new language
- Union, make ϵ transitions to start states of both DFA
- Concat, make ϵ transitions from accept states of A to start state of B
- Kleen's star, create new initial accept state, add ϵ transition to start state. For every accept state, add ϵ transition to the old start state.
- Intersect

- Universal Complement
- Set subtract
- Reversal (reverse every accepted string)

17 Minimization

There may be multiple DFAs that recognizes the same language.

Minimal DFA

- The minimal DFA for a regular language is a DFA that recognizes it with the minimum number of states
- The minimal DFA is unique. So we can check for DFA equivalence by computing the minimal DFA

Algorithms typically does not output a minimal DFA

Minimization

- An algorithm that converts an NFA and produces the equivalent DFA
- First reverse the NFA
- Determinize the result
- Reverse again
- Determinize

Reversal

- To reverse NFA A, assuming non-empty accept state
- If $F = \{q\}$, let q be the accept state
- Otherwise, add new accept state q with ϵ transitions from F to q (remove acceptance property of F)
- Reverse every transition in NFA. Let q be start state and q_0 be the accept state.

18 Regular Expressions

Pattern matching

- Search engines
- Lexical analysis
- Scripting language
- DNA

Regular expressions

- Compact notation to describe regular languages
- Restricted syntax for pattern matching

- Used in most programming languages as regex
- Grammar over an alphabet Σ

  ```
  regexp ::= a1 | ... | an | e | 0 | regexp U regexp | regexp o regexp | regexp*
  ```

- Operator precedence: all concat omitt, star stronger than concat than union

Semantics (mapping regexp to formal languages)

- $L(a) = \{a\}$
- $L(\epsilon) = \{\epsilon\}$
- $L(\emptyset) = \{\}$
- $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
- $L(R_1 R_2) = L(R_1) \circ L(R_2)$
- $L(R*) = L(R)*$

Regular expression vs NFA

- A language is regular iff it can be described by a regular expression (via semantics)
- Involves showing that regular expression only describes regular languages
- And that Regular languages must have a regular expression that describes it

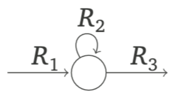
Regular expression to regular language (NFA)

- Inductive proof by building an NFA equivalent to the regular expression
- Base cases when $R = a, \epsilon, \emptyset$. Make these up
- Inductive cases when: $R = R_1 \cup R_2, R_1 \circ R_2, R_1^*$. Use the regular language closure under regular operations proof

Regular language to regular expression

- Proof by converting an NFA to an equivalent regular expression
- Lemma: given an NFA, can build an equivalent NFA with at most one accept state
- Process:
  1. Ensure that the NFA has one accept state (otherwise regexp is $\emptyset$)
  2. Convert all arcs to regexp
  3. Repeatly eliminate states that are neither start nor accept states. Create new arcs that are labeled by regexp
  4. At termination, there are two outcomes that can be directly translated to regexp

RL to Regexp state elimination

Consider a node 

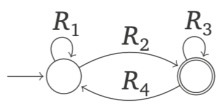
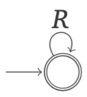
Any such pair of incoming/outgoing arcs get replaced by a single arc that bypasses the node. The new arc gets the label $R_1R_2^*R_3$.

If there are m incoming and n outgoing arcs, these arcs are replaced by $m \times n$ bypassing arcs when the node is removed.

Figure 1:

Also note that we can replace multiple arcs between A and B by a single arc that unions all arcs.

RL to Regexp termination

The process produces either  or 

Note that some Rs may be ϵ or $\emptyset$.

We get $(R_1 \cup R_2R_3^*R_4)^*R_2R_3^*$ in the first case; R^* in the second.

Not much theory to say here, mostly just practice elimination and termination.

Regexp properties

- $A \cup A = A$
- $A \cup B = B \cup A$
- $A \cup B \cup C = (A \cup B) \cup C$
- $A \circ B \circ C = (A \circ B) \circ C$
- $\emptyset \cup A = A$
- $\epsilon \circ A = A$
- $\emptyset \circ A = \emptyset$
- $(A \cup B) \circ C = (AB) \cup (AC)$
- $A \circ (B \cup C) = AB \cup AC$
- $(A^*)^* = A^*$
- $\emptyset^* = \epsilon^* = \epsilon$
- $(\epsilon \cup A)^* = A^*$

- $(A \cup B)^* = (A^* B^*)^*$

Limitations of finite automata

- No lookahead
- Fixed number of bits of memory

19 Proving Non-Regularity

Two methods of non-regularity proving

- Fooling sets (leveraging memoryless property)
- Closure properties (leveraging properties that some languages are not regular and that regular languages are closed)
- Pumping lemma (not covered coz it is too complicated)

Fooling set roadmap for language L

1. Lower bound the number of states (N) for a DFA to have to recognize L. Stating: no DFA with fewer than N states can recognize L
2. Show that this lower bound N exists for all natural numbers
3. Hence the language is non-regular. (Since if it is, there exists DFA with N states that recognize it, but the lower bound can be $N + 1$, so this DFA doesn't recognize)

Extended transition function

- Define $\delta' : Q \times \Sigma^* \to Q$

$$\delta'(q, w) = \begin{cases} q & \text{if} \quad w = \epsilon \\ \delta(r, a) & \text{if} \quad r = \delta'(q, x), w = xa, a \in \Sigma \end{cases}$$

- Essentially $\delta'(q, w)$ is the resulting automata state starting at q following each character in string w.
- Theorem

$$q(w) = \delta'(q_0, w)$$

  the resulting state of the automata when parsing w, $q(w)$, is equivalent to $\delta'(q_0, w)$.

Prefix-suffix decomposition

- For string w, a valid prefix-suffix decomposition into strings x, y means that $w = xy$. Note that both/either x, y can be ϵ
- For $w = xy$, we have

$$q(w) = \delta'(q_0, w) = \delta'(q(x), y) = \delta'(\delta'(q_0, x), y)$$

Memoryless lemma

- For strings x, y, if $q(x) = q(y)$, then for all strings z, $q(xz) = q(yz)$.

- Namely that future transitions/states depend only on: the current state $q(x) = q(y)$, and future inputs z

Characterizations of one-state dfas

- One-state dfas can only recognize the languages: Σ^* (accepts every string) or $\emptyset$ (rejects every string).
- Proof: every string has the same resulting state

Distinguishable pairs/suffix

- Given language L, strings x, y in general, x and y are distinguishable if there exists string z such that only one of xz and yz is in L
- If x, y are distinguishable, then the z that distinguishes them is the distinguishing suffix
- Example: L is set of all strings ending with 10, then 0 and 1 are distinguishable with the distinguishable suffix 0.

The memoryless lemma concerns only about finite state automata. Distinguishable pairs concern only about the underlying language. The combined lemma is

- Given language L and any DFA that recognizes L. For all distinguishable pairs x, y,

$$q(x) \neq q(y)$$

- Proof: distinguishable means that $q(xz) \neq q(yz)$, applying contrapositive memoryless lemma means that $q(x) \neq q(y)$
- Consequence: a language with at least one distinguishable pair is not recognized by any one-state DFA

Fooling set

- A set of strings F is a fooling set for language L when every distinct pair in F are distinguishable wrt L
- Fooling set lemma: if F is a fooling set for language L, then every recognizing DFA for L must have at least $|F|$ states
- Proof: suppose M is a recognizing DFA with $k < |F|$ states, then notice that $q(x)$ is unique for all $x \in F$ by definition of the fooling set, with $|F|$ unique values. By the PHP, there must also be two strings in F where $q(x) = q(y)$ ($|F|$ values out of $k < |F|$ states). Hence a contradiction.

Fooling set non-regularity theorem

- If for every $k > 0$, there exists a fooling set of size at least k, $|F| \geq k$, then the language cannot be regular
- Proof: Assume that the language is regular and there exists a DFA that recognizes it with k states. Since there is a fooling set of size greater than $|F| \geq k + 1 > k$, the recognizing DFA must have at least $|F|$ states, which is impossible since it has only $k < |F|$ state. Hence a contradiction

Fooling set examples

- L is set of strings ending with 10, $F = \{\epsilon, 1, 10\}$

- L is $\{0^n 1^n : n \geq 0\}$, $F_k = \{0^i : i \in [1..k]\}$

Fooling set construction strategies

- Consider prefixes of strings in L
- Strings requiring a counter to differentiate between them, with the counter ranging from 1 to k.
- Typically there is a unique distinguishable suffix that distinguishes one string in F from all the others. (Not required in general since the suffix can change)

Minimal witness

- A DFA is proved to be minimal for a regular language L if there is a fooling set of size $|Q|$. If so, the fooling set F is a witness to the minimal property of the DFA.

Closure non-regularity method

- If L^c is non-regular, then L is non-regular
- If $A \cap B$ is non-regular, and A is regular, then B can't be regular
- Collorary: to prove that L is non-regular, try proving that L^c is non-regular; alternatively find a regular B and prove $L \cap B$ is non-regular

Fooling set vs closure non-regularity

- Fooling set will always work to prove non-regularity (theorem behind it)
- Closure methods are shorter and more concise, but may not always work and requires finding the right operations and languages.

20 Context-free Language

Context free grammar (CFG)

- Defined as $G = (\Sigma, V, R, S)$
- Σ is a finite set of terminal symbols
- V is a finite set of variable symbols (nonterminals)
- R is a finite set of production rules of the form $A \to w$, where A is a variable and $w \in (\Sigma \cup V)^*$ is a string of variable/terminals. Also known as substitution rules. Allows recursion since w can contain A.
- S is the starting variable

CFG vs Regexp

- Regexp allows us to create regular languages using regular operations
- CFG allows us to create context-free languages. Similar to regexp except that it can do recursion.
- Every language generated by Regexp can be generated by CFG, but not vice versa
- CFG used the define schemas of programming languages

CFG OR shorthand

- Can combine production rules with the same LHS A variable into a shorthand notation where each RHS is separated by |
$$A \to x|y|z$$

CFG string generation

- Init output string $x = S$.
- While x contains at least one variable, apply a production rule to replace a single occurrence of a variable in x.
- The final string with only terminals is the produced string
- Different rule selection leads to different final strings

Application and derivation

- If applying $A \to y$ on string xAz gives xyz, we say that $xAz \implies xyz$. This is application
- x derives z if there is a finite sequence of applications from x to obtain z. We denote this as $x \implies {}^*z$.
- Formally, $x \implies {}^*z$ when: $x = z$, or there exists strings $y_1, \dots y_n$ where
$$x \implies y_1 \implies \dots y_n \implies z$$
- The sequence to derive z is called the derivation of z.

Sentential

- Strings in $(\Sigma \cup V)^*$ are in sentential forms
- Strings in Σ^* are sentences
- Sentences are sentenetial form without variables

CFG language

- The language generated by a CFG is the set of strings that the CFG can generate.
- Equivalently, this is the set of strings that can be derived from the start variable
$$L(G) = \{x \colon S \implies {}^*x\}$$

Context free language

- A language is context free iff there exists a CFG that generates it

Parse tree

- A visualization of the derivation process from S.
- Represents the syntactic structure of a string. Namely, the tree should tell us the order of operations to compute if the language is arithmetic expression.
- Formally, a parse tree for a string $w \in L(G)$ is a root tree
  - Each node is labeled by a variable or a terminal

  - Root node is S
  - Non-leaf nodes are associated with a production rule $A \to v$. Namely, A is the non-leaf node label and each symbol in v are its child node labels from left-to-right
  - Leaf nodes are all terminal. The concat of all leaf nodes from left-to-right gives w

Ambiguity

- A string is ambiguous against a grammar G iff there are more than one parse tree for the string.
- A grammar G is ambiguous iff there exists an ambiguous string with respect to it
- A context-free language L is inherently ambiguous iff every G that generates it are ambiguous

Closure

- The class of context-free languages are closed under: union, concat, kleene's star, reversal
- It is NOT closed under intersection. However, $A \cap R$ is context free if A is context free and R is regular
- To prove closure, take the two grammars and make a new context-free grammar that generates the new language

Closure proofs

- Union: Make a new start variable with production rule `S1 -> Sa | Sb` pointing to the start variable on A and B
- Kleene's star: make a new start variable with production rule `S1 -> e | Sa S1`
- Concat: new start variable with rule that is concat start variables
- Reversal: reverse all strings in rules

Regular languages

- Every regular language is context free
- Proof: A is a regular language, R is a regular expression that matches it, recursively convert the regexp to CFG rules and variables using the regexp operations (union, concat, kleenestar, atom)
- To convert regexp to CFG: convert each regexp operation to production rule inside out

Example conversion $(0 \cup 1)^*$

```
S -> A
A -> e | BA
B -> C | D
C -> 0
D -> 1
```

21 Push-down automata

Machines and Languages

- Finite Automata and Regular languages
- Pushdown automata and Context-Free languages
- Linear-bounded automata and Context-Sensitive
- Halting-turning machine and decidable
- Turing machine and turing recognizable

Pushdown automata

- An NFA with an unlimited stack memory
- Stack operations: peak stack, push stack, pop stack
- A pushdown automata is defined by the tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where
- Q is the set of states
- Σ is the alphabet set of the input string
- Γ is the alphabet set of the stack
- $\delta : (Q \times \Sigma_\epsilon \times \Gamma_\epsilon) \to \mathcal{P}(Q \times \Gamma_\epsilon)$, a function mapping (current state, input symbol, stack symbol) to a set of (next state, new stack symbol)
- $q_0 \in Q$ is the starting state
- $F \subseteq Q$ is the set of accepting states

PDA operations

- TLDR, The PDA accepts if there exists a path (non-deterministically) that ends in an accept state. Similar to NFA except with stack operations
- Starts at q_0
- Repeat while still input: non-deterministically transition via a valid transition from δ to a new state `a, b -> c`, consume input a, replace stack top b with c. Reject if no transitions are possible
- Accept if state in accept state.

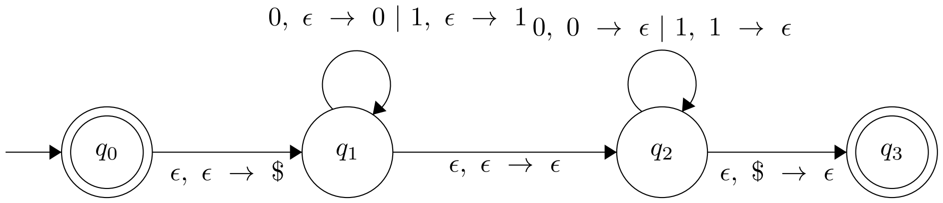
Pushdown automata transitions

- Of the form: $A, B \to C$ where
- A is the input character
- B is the stack top character that is popped (ϵ implies no pop and matches any stack symbol)
- C is the character to be pushed onto the stack (ϵ implies no push)
- Example: `_, e -> e` means noop,
  `_, a -> e` means popping `a`,
  `_, e -> a` means pushing `a`,
  `_, a -> b` means swapping `a` with `b`

Pushdown automata empty stack

- No requirement for the stack to be empty to be accepted
- To ensure an empty stack, can push a special character \$ onto the stack at the start, and pop it at the end transitioning to an accept state.
- If the stack is empty, a pop operation that is not ϵ is invalid. Any PDA that may perform this operation is invalid

The PDA to recognize $\{ww^R \colon w \in \{0,1\}^*\}$



$0,\ \epsilon \to 0 \mid 1,\ \epsilon \to 1$
$0,\ 0 \to \epsilon \mid 1,\ 1 \to \epsilon$
q_0
q_1
q_2
q_3
$\epsilon,\ \epsilon \to \$$
$\epsilon,\ \epsilon \to \epsilon$
$\epsilon,\ \$ \to \epsilon$

Figure 2: PDA to recognize panlindromes

Context-free language

- A language is context-free iff there exists a PDA that recognizes it
- PDA has equivalent power as CFG

Regular grammars

- Regular grammars (grammar that generates regular languages) are CFG with restricted rules
- The restrictions are: they must be of form $A \to w$ or $A \to wB$, where $A, B \in V$ and $w \in \Sigma^*$

22 PDA and CFG

CFG = PDA

- A language can be generated by a CFG iff it is recognized by a PDA
- Proof requires proving both ways: every CFG has a PDA, every PDA has a CFG

If language can be generated by CFG, there exists a PDA that recognizes it

- Given CFG G, consider a PDA with alphabet of the terminals and variables of G
- PDA start by pushing start variable onto stack (assuming empty stack symbol first)
- If top of stack is a variable, non-deterministically choose a production of the variable from G. Pop variable from stack and push the RHS of the rule onto stack in reverse order
- If top symbol is a terminal, pop it from stack if it matches current input symbol. Otherwise reject
- When stack is empty, accept if the input is empty, reject otherwise (link all states to a stackempty state with ϵ transition and empty stack symbol)

The reverse direction (PDA to CFG)

- Simplify the PDA to follow the conditions: single accept state, empty stack when accept, transitions only pushes char to stack or pops char from stack.
- Then we can directly create the CFG from PDA using `https://www3.nd.edu/%7Edchiang/teaching/theory/2016/notes/week06/week06b.pdf`
- The idea is to create variables A_{pq} that generates strings which corresponds to strings accepted from state p to q

Note that non-deterministic pushdown automata are strictly more powerful than deterministic versions, unlike DFA.

Can use pumping lemma to prove a language is not context-free, not examinable.

23 Turing Machine

Informal Turing Machine

- Consists of an input tape and a finite state machine
- The head can read and write on the input tape
- The head can move left or right
- The input tape is infinite to the right
- Infinitely many blank characters following the input on the tape
- The machine can accept or reject the input at any time (not just the input end)

Example TM for $\{a^n b^n c^n : n \geq 0\}$

- Scan right until blank while checking if input is in $a^* b^* c^*$. Reject if not
- Return head to left end
- Scan right, crossing off a single a, b, and c (by replacing them with the crossed character)
- If last one of each symbol, accept (make matching states indicating seeing uncrossed input)
- If last one of one symbol but not others, reject
- Otherwise, return to left end and repeat

Turing Machine

- $TM = (Q, \Sigma, \Gamma, q_0, q_a, q_r, \delta)$
- Σ is the input alphabet
- Γ is the tape alphabet ($\Sigma \subseteq \Gamma$) that has a blank character
- q_0 is initial state
- q_a, q_r are accept and reject states
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- The transitions $\delta(q, a) = (p, b, L)$ implies that from state q with tape input a, transition to state p, writing b to tape, move left

- Note that this TM is deterministic. δ also needs to be complete (requiring all transitions) for all states except for q_a, q_r
- Moving left when the tape head is on the left-most tape character does nothing

Termination

- On an input w the TM may halt (reached q_a, q_r) or loop forever
- There are three possible outcomes for each w: accept w by entering q_a (halt), reject w by entering q_r (halt), loop

TM language

- $L(M)$ is defined as the set of strings that can be accepted by w.
- Strings that are rejected or loop forever without accepting are not in $L(M)$

Decider

- A TM is a decider if it halts on all inputs. It will never loop

TM graph

- States are nodes
- Transitions: `a -> b, R` means tape input a overwriting with b and move right; `a -> L` means tape input a not overwriting, move left
- An DFA can be converted to TM by replacing a with `a -> R`

TM configurations

- A configuration for TM is a snapshot of its execution containing: current sate, current tape values, current location of head on tape
- A configuration is represented by `abqabb` where q is the state on the left of the tape head location.
- The initial configuration for input w is always q_0w
- If applying δ on config C yields C', then $C \Rightarrow C'$. For instance

$$\begin{aligned} uqbv \Rightarrow uctv \quad & \delta(q,b) = (t,c,R) \\ qbuv \Rightarrow tcuv \quad & \delta(q,b) = (t,c,L) \end{aligned}$$

- TM accepts a string w iif there exists a sequence $C_1, C_2, \dots, C_k$ where: $C_1 = q_0w$, and $C_i \Rightarrow C_{i+1}$, and state of C_k is q_a.

Recognizer and Deciders

- M recognizes a language A iff $A = L(M)$. And A is turing-recognizable iff there exists a TM that recognizes it
- M decides a language A iff $A = L(M)$ (M recognizes A) and that M is a decider (M always halts). And A is turing-decidable iff there exists a TM decider that recognizes the langauge
- All turing-decidable languages are turing-recognizable, since there exists a TM that recognizes it.

Satisifiability

- Satisifiability in propositional logic is decidable
- Satisifiability in first-order logic is only recognizable

24 Church Turing Thesis

Church Turing Thesis

- All reasonable models of unrestricted computation (lambda calculus, random access machine, programming language) are equivalent to the power of the turing machine
- Prefer turing machine since it is simple and familiar
- Essentially: all problems that are computable under any computer (solvable) must also be decidable, and vice versa. This holds even with quantum computers, time travelling, and non-determinism.

Turning machine is robust

- Adding new features to it doesn't make it more powerful

Multi-tape turing machines

- Multiple set of tapes (input tape, multiple work tapes initially blank). Number of work tapes fixed at the start
- Transition depends on all head symbols on all tapes. Can modify each tape individually on a transition

Multi-tape theorem

- Language A is t-recognizable iff there exists a multi-tape turing machine that recognizes it
- Forward direction proof obvious
- To convert a multi-tape TM into a single tape S:
  - S stores multiple tapes on a single tape in blocks separated by character `#`
  - Make a dotted variant for tape symbols to indicate tape head in eaceh block
  - To simulate a transition: scan entire tape to find dotted symbols, scan again to update S according to multi-tape δ
  - If ran out of space in a block, shift everything after updating (leave an empty space after each block)
  - The set of states (including q_a, q_r) are equivalent to multi-tape. So same accept/reject criteria

Non-deterministic turing machine

- A NTM is similar to a DTM except that its δ maps to $\mathcal{P}(Q \times \Gamma \times \{L, R\})$

NTM theorem

- Language A is t-recognizable iff there is a NTM that recognizes it

- Forward direction is obvious
- To convert a NTM into DTM:
  - See that a NTM creates a computation tree with threads of non-deterministic operations
  - The tape will store blocks of tape each corresponding to a computation fork/thread of the NTM.
  - Head of each block stored using dotted variant
  - State of each block stored as a symbol at the start of each block
  - Copy block state if fork
  - Accept if any block state is q_a, reject if any block state is q_r

25 TM Algorithms on Automatas

String encodings for TM

- For any object O, denote $\langle O \rangle$ as the string encoding of that object.
- Every discrete object has an string representation
- For a list of objects $O_1, \ldots, O_n$, write $\langle O_1, \ldots, O_n \rangle$ to be the encoding of all objects into a single string (encode each separated by a separator character)

Turing machine notation

- To describe TM algorithms, use high level english descriptions
- In principle we can convert the descriptions into states/transition-functions
- Notation to writing TM M is: $M =$ "On input w, blah english description of algorithm"

Automata problems

- Acceptance problem involves a TM checking if an automata accepts an input
- Emptiness problem involves a TM checking if an automata recognizes the $\emptyset$ language (no string is accepted). This is similar to checking unsat
- Equivalence problem involves a TM checking if two automatas are equivalent

DFA acceptance problem

- Let $A_{DFA} = \{\langle B, w \rangle :$ B is DFA, B accepts w$\}$
- Theorem is that A_{DFA} is decidable. That is, there exists a turing machine (which always halts) that can recognize the language
- The TM D_{A-DFA} that decides it is
  - On input s
  - Checks that s has the form $\langle B, w \rangle$ where B is a DFA and w is a string. Reject otherwise
  - Simulate B on w

    - If B ends in an accept state then accept. Otherwise reject
    - Specifically, follow the multi-tape model where the input tape contains B and w, and a work tape maintains the current DFA state q and the input string w location/index k

NFA acceptance problem

- Let $A_{NFA} = \{\langle B, w\rangle \colon$ B is NFA, B accepts w$\}$
- The TM D_{A-NFA} that decides it is
    - On input $\langle B, w\rangle$
    - Run algorithm to convert NFA B to DFA B'
    - Run D_{A-DFA} on $\langle B', w\rangle$, use its acceptance criteria
- A case of reduction: convert problem and use a previously constructed TM as a subroutine. We've reduced A_{NFA} to A_{DFA}s

DFA emptiness problem

- Let $E_{DFA} = \{\langle B\rangle \colon$ B is a DFA and L(B) = 0$\}$
- Theorem is that E_{DFA} is decidable.
- Idea is to check whether there is a path from start to any accept state
- The TM D_{E-DFA} that decides E_{DFA} is
    - On input $\langle B\rangle$
    - Run BFS from start state following transition arrows
    - Accept if no accept state is marked. Reject otherwise
- Alternative TM: run minimization algorithm, check if resulting DFA is the unique single-state rejection DFA

DFA equivalence

- Let $EQ_{DFA} = \{\langle A, B\rangle \colon$ A and B DFA and L(A) = L(B)$\}$
- Theorem is that EQ_{DFA} is decidable
- Idea is to reduce it to two emptiness problems
- The TM D_{EQ-DFA} that decides it is
    - On input $\langle A, B\rangle$
    - Use DFA closure to make a DFA C where

      $$L(C) = (L(A) \cap \neg L(B)) \cup (\neg L(A) \cap L(B))$$

      This $L(C)$ is known as the symmetric difference between $L(A)$ and $L(B)$
    - Use D_{E-DFA} to check if $\langle C\langle$ is empty. Accept if it is. Reject otherwise

CNF CFG

- The chomksy normal form of CFG only allows rules of

$$\begin{aligned}&S \to \epsilon\\&A \to BC\\&B \to b\end{aligned}$$

- Every CFG can be converted to its CNF
- If $w \in L(H)$ where H is in CNF, every derivation of w from H requires exactly $2|w| - 1$ steps: $|w| - 1$ max steps to produce non-terminals, and $|w|$ steps to produce terminal strings.

CFG acceptance

- Let $A_{CFG} = \{\langle G, w\rangle$: G is CFG and w is in L(G)$\}$
- A_{CFG} is decidable
- The TM D_{A-CFG} is
  - On input $\langle G, w\rangle$
  - Convert G into CNF
  - Try all derivations of length $2|w| - 1$
  - Accept if any derivation is w. Reject otherwise

Corollary: every CFL is decidable. Let A be CFL generated by G CFG. TM proof just simply calls D_{A-CFG}.

Emptiness problem for CFG

- Let $E_{CFG} = \{\langle G\rangle$: G is CFG and L(G) = 0$\}$
- E_{CFG} is decidable
- Idea is to traverse backwards from terminals
- TM proof D_{E-CFG}
  - On input $\langle G\rangle$
  - Mark all terminals in G
  - Multisource BFS along edges where mark A if all B_i are marked under

$$A \to B_1 \dots B_n$$

  - Reject if start variable is marked. Accept otherwise

Equivalence problem for CFG

- Determine if A and B CFGs are equivalent
- Not decidable

Ambigious CFG

- Determine if A CFG is ambigious

- Not decidable

Acceptance problem for TM

- Let $A_{TM} = \{\langle M, w\rangle \colon \text{M is TM and M accepts w}\}$
- Not decidable. But turing recognizable
- Undecidable proof uses diagonalization (argument)
- T-recognizable TM U is:
  - On input $\langle M, w\rangle$
  - Accept if M halts and accepts
  - Reject if M halts and reject

Universal computing machine

- Only TM can run/simulate itself
- DFA/PDA cannot simulate themselves

26 Undecidability

Decider for TM acceptance problem

- If there is a decider U for the TM acceptance problem, we can then derive very powerful theorems
- If there is a recognizer M for a language L, then we can use the acceptance TM to canonically create a decider D for L (Run U on M and input, accept if accept, reject if reject).

Cantor set cardinality

- Two sets have the same size if there exists a bijection between elements in both sets
- Bijective function is equivalent to both being injective and surjective. Injective means no two elements in A are mapped to the same in B. Surjective means all elements in B has some element in A mapped to it

Example equal cardinal sets

- Natural numbers and boolean strings, assign numbers to boolean strings by their lexiographic ordering
- Natural numbers and turing machines, since each turing machine can be encoded into a boolean string

Countable sets

- A set S is countable iff it has the same cardinality as $\mathcal{N}$
- The set of boolean strings and the set of TM are countable

Reals are not countable

- The set of real numbers has a larger cardinality than the set of naturals

- Proof, for every function $f\colon N \to [0,1]$ will miss some real number (diagonalization). Consider such a mapping f, make a real number whose ith digit is 1+ the real number digit of $f(i)$. This real number has no natural number that is mapped to it by definition, hence the mapping is not bijective.

Non-recognizable language theorem

- Theorem: there is a non-recognizable language
- Consider TM written in their lexiographic ordering under their string encodings. This has an ordering since the set of TM is countable.
- Consider this mapping from TM to the language that it recognizes (TM to a bitstring with the ith value being 1 if the ith bitstring is accepted by TM).
- Using the diagonalization argument, there exists a language L_D such that no turing machine recognizes, as it differs in bitstring acceptance on the ith index.
- Therefore there must be an unrecognizable language, namely

$$L_D = \{x\colon x \notin L(M_k))\}$$

  where k is the index of x

L_D decidability theorem

- If TM acceptance problem is decidable using U, then L_D is decidable
- Proof, a decider D for L_D would be: on input x, run U on $\langle M_k, x\rangle$, where k is the index of bitstring x, if accept then reject, if reject than accept. Hence D decides L_D.

TM acceptance problem

- If the problem is decidable, then there exists a decider for the language L_D which is not recognizable. Therefore a contradiction, so the problem is not decidable

27 Limits of computation

Identification of impossible problems

- Goal is to find unrecognizable and undecidable problems
- Two methods: diagonalization (self-reference), reduction

Self-reference

- Statements referencing themselves: this sentence is false

TM acceptance problem revisited (self-reference, contradiction version). First we establish a language L_S that is unrecognizable

- Note that the set of turing machines is countable, so consider an ordering of them $M_i, i \in \mathbb{N}$
- Consider a table where rows are $L(M_i)$ and columns are $\langle M_i\rangle$, the entry is "in" if the encoding is in the language, and "out" otherwise.
- Let L_S be the langauge whose string acceptances are the flipped variants on the diagonal. This is where

$$L_S = \{\langle M_i\rangle\colon \langle M_i\rangle \notin L(M)\}$$

or that L_S is the set of TM encodings whose TM does not accept themselves.

- Now for contradiction, assume that there exists TM R such that $L(R) = L_S$.
- If $\langle R \rangle$ is in L_S, then R does not accept themselves, due to definition. But since $\langle R \rangle \in L_S$, and $L(R) = L_S$, then R must accept $\langle R \rangle$. This assumption is impossible
- If $\langle R \rangle$ not in L_S, then R does accept themselves. But since $\langle R \rangle \notin L_S$, and $L(R) = L_S$, then R must not accept $\langle R \rangle$. This assumption is also impossible
- Therefore R must not exist, and that L_S is non-recognizable

Then to show that TM acceptance is undecidable

- For the sake of contradiction, assume that TM acceptance is decidable with decider H
- Create TM decider D using H. On input $\langle M \rangle$ (reject otherwise): run H on itself $\langle M, \langle M \rangle \rangle$, accept if H rejects, and reject if H accepts
- Hence D checks if a TM rejects themselves.
- Now note that $L(D) = L_S$, but L_S is not-recognizable, so D must not exist. Hence by contradiction, H must not exist.
- Alternatively if we don't use L_S:
  - If D accepts itself, then by definition of D, D should not accept itself. So it is not impossible
  - If D does not accept itself, then by definition of D, D should accept itself. So it is not impossible

28 Reduction

Reductions

- A problem A reduces to problem B ($A \leq B$) iff B is decidable implies that A is decidable
- To show that $A \leq B$, we use a decider for B to make a decider for A
- Finding the min element in a list is reducible to sorting a list.
- To show that a problem B is undecidable, we just have to show that: there is a A which reduces to B, and A is undecidable

Example reductions

- DFA equivalence reduces to DFA emptiness
- NFA acceptance reduces to DFA acceptance
- Deciding L_D reduces to TM Acceptance

TM acceptance problem revisited (reductions)

- Deciding L_D reduces to TM Acceptance, as in if we have a decider for TM Acceptance, we have a decider for L_D
- And we've proved that L_D is undecidable, and by contrapositive of reductions, TM Acceptance is also undecidable

Reduction direction

- $A \leq B$ is denoted as A reduces to B, meaning that we use B's decider to construct A's decider
- Algorithms: In the B to A direction, if B is decidable, then A must be decidable. This allows us to create new algorithms for deciding A using B's decider
- Hardness: In the A to B direction, if A is undecidable, then B must be undecidable. This allows us to prove hardness for problem B given that A is also hard

Static code analysis

- The idea of analyzing code without running them
- Questions: any run-time errors, security vulnerabilities, violate spec
- Requires decider for: halting, TM emptiness, TM equivalence problem
- Potential usage areas: defense, medical, LLM backdoors

Halting problem reduction

- Let the HALT language be the set of (TM, input) pairs such that the TM halts on the input. This corresponds to the problem of checking if the TM halts on an input
- Theorem. If HALT is decidable, then we can always turn recognizers into deciders
- Reduction proof of undecidability, goal is to prove that TM acceptance reduces to the halting problem: Suppose H is the decider for the halting problem. Consider decider S where, on input $\langle M, w\rangle$

  1. Run H on $\langle M, w\rangle$
  2. If H rejects, reject
  3. Otherwise, simulate M on w (which will always halt), return its result

  Hence S is a decider for TM acceptance. Therefore $A_{TM} \leq \text{HALT}_{TM}$. Because TM acceptance is undecidable, the halting problem is also undecidable

TM Emptiness reduction

- Let the EMPTY language be the set of TM whose language that it recognizes is the empty set. This corresponds to the problem of checking if TM is empty
- Reduction proof. Let H be the decider for the emptiness language. Construct decider S where on input $\langle M, w\rangle$

  1. Create new TM M_w, on input x, run M on w and returns the result
  2. Run H on $\langle M_w\rangle$, if accept, then reject; if reject, then accept

  Since M_w is only empty when M doesn't accept w, S is a decider for TM acceptance. So $A_{TM} \leq E_{TM}$, by reduction, the emptiness problem is undecidable

TM equivalence reduction

- Let the equivalence language be the set of TM pairs who are equivalent. This is the TM equivalence problem

- Reduction proof from TM emptiness. Let H be the decider for TM equivalence. Consider decider S where on input $\langle M \rangle$
  1. Create TM E that rejects on all inputs
  2. Run H on $\langle M, E \rangle$
  3. If accept, accept. If reject, reject

  Hence S is a decider for the emptiness problem, so $E_{TM} \leq EQ_{TM}$, so the equivalence problem is undecidable

29 Decidable Language Closure Properties

Closure properties on decidable languages

- Closed under complement. L decidable implies L^c decidable. Swap q_a and q_r
- Closed under union and intersection. A, B decidable means $A \cup B$ and $A \cap B$ decidable. Proof by simulating
- Closed under kleene star. L decidable implies L^* is decidable. Simulate the decider on all partitions of the input string and accept if any one partition accepts.

Closure properties on recognizable languages

- If L and L^c are recognizable, then L is decidable. A collorary is that the complement TM acc problem is not recognizable
- Proof. Let M_1, M_2 be recognizers for L and L^c. Constructor decider D that on input w
  1. Run M_1 and M_2 on w in parallel
  2. Accept when any has accepted, reject if both rejected

  D will always halt, since for any input at least one of the recognizer will halt on the input.

30 TM analysis purpose

Static code analysis, revisited

- Runtime error and security vulnerabilities are undecidable due to the halting problem
- Violating specification are undecidable due to TM equivalence

Rice's theorem

- Every non-trivial property about program behavior is undecidable
- A property implies some function that only depends on the TM langauge $L(M)$, never on the TM source code
- Non-trivial means that the property must take on at least two different values.

Computable and Efficient

- Computable problems means a finite time algorithm
- Efficient problems means a polynomial time solution

- Some computable problems are not efficient (yet): CNF sat, integer factoring, AI planning. The $P = NP$ problem

Handling intractable problems

- Relax requirements means to analyze the problem by lowering the strictness of the requirements
- Recall correctness: forall inputs, M accepts iff the input is in the language.
- A correctness relaxation could be considering only a subset of structured inputs to be correct.
- Parameterized algorithms are ones who have expoential time on general inputs, but poly time on specific types of input
- Approximately optimal solutions means to have a solution that quickly solves a less strict problem (namely for heuristic algorithms)

Computation everywhere

- Church-turing thesis: every physically realizable process can be simulated by a TM. Viewing real life processes under a computational lens
- Economics: markets always converge to an equilibrium. If we can show that under a specific model the convergence is undecidable, then it would not either in practice.

Online algorithms

- Regular language and CFL both have on-the-fly computation, meaning that they can easily handle an extra input character